OCCAM 2

JOHN GALLOWLY
Contents

Preface

Introduction

Occam and the transputer

1 Occam 3
2 The transputer 3
3 The origins of occam 7
Extra reading 10

Chapter 1
Occam basics 11

1.1 Primitive processes 12
  1.1.1 Assignment 12
  1.1.2 Input and output 14
    1.1.2.1 Input 15
    1.1.2.2 Output 15
1.2 The SEQ construction 16
1.3 The PAR construction 19
1.4 The ALT construction 21
1.5 Nested constructions 23
1.6 Two-way channel communication 25
1.7 Deadlock 26
1.8 SKIP and STOP 28
1.9 Termination of occam programs 28
1.10 Comments in occam programs 29
Exercises 30
Chapter 2

Data basics

2.1 Identifiers 32
2.2 Data types 35
2.3 Literals 36
2.4 Declarations 42
  2.4.1 Simple constants 42
  2.4.2 Variables 45
2.5 Simple channels 46
2.6 Scope 49
Exercises 52

Chapter 3

Operators 55

3.1 Arithmetic operators 56
  3.1.1 Modulo arithmetic operators 58
  3.1.2 MOSTPOS and MOSTNEG 59
  3.1.3 Operator precedence 59
  3.1.4 Data conversion 60
3.2 Relational operators 64
3.3 Boolean operators 66
3.4 Bitwise operators 68
3.5 Shift operators 70
Exercises 72

Chapter 4

Arrays 74

4.1 One-dimensional arrays 74
4.2 Multi-dimensional arrays 78
4.3 Array subscripts 79
4.4 Segments 82
4.5 Strings 84
Chapter 5

Constructions

5.1 Alternation construction
   5.1.1 Boolean guards
   5.1.2 SKIP guards
5.2 Conditional construction
   5.2.1 Nested conditional
5.3 Selection construction
   5.3.1 ELSE option
5.4 Repetition construction
Exercises

Chapter 6

Replicators

6.1 Replicated SEQ
6.2 Replicated PAR
6.3 Replicated ALT
6.4 Replicated IF
6.5 Segments for efficiency
6.6 Processes as data structures
Exercises

Chapter 7

Abbreviations, procedures and functions

7.1 Abbreviations
   7.1.1 Expression abbreviations
   7.1.2 Element abbreviations
Chapter 10
Configuration

10.1 PLACED PAR 181
10.2 PROCESSOR 181
10.3 Inter-transputer links 182
10.4 Examples 184

Chapter 11
Approaches to writing parallel programs in occam 191

11.1 Algorithmic parallelism 192
11.2 Geometric parallelism 197
11.3 Process farming 210
11.4 Efficiency factors 223

Appendix

The transputer development system 225

A.1 The folding editor 227
A.1.1 Example session 232
A.2 Compiling occam programs 233
A.2.1 Example session 235
A.3 TDS libraries 237
A.4 Other features of TDS 241

References 242

Further reading 244
Answers to exercises

Index
Preface

Aims and objectives

This book presents a gentle and structured introduction to occam, the programming language specifically designed for the transputer, a parallel architecture microprocessor developed by Inmos Limited. The aim of the book is to give a clear and concise description of this exciting new language. The text assumes that the reader has some knowledge of a programming language. The level is suitable for undergraduate students taking a course in parallel programming, although anyone with a knowledge of programming should have no difficulty in following the text for self-study. Each language construct is illustrated with an occam example. The book is so organised so that information given in the early chapters is used and reinforced in later chapters.

A formal syntax definition of occam 2 in terms of BNF has not been included in the book. This may be found in the definitive work on occam 2 by Inmos Limited - "The occam 2 Reference Manual".

Contents

The contents of the book grew out of a course which I gave to undergraduates at the University of Buckingham in the autumn of 1987. Chapters 1 to 9 contain a description of the basics of occam 2. These are introduced in a structured way so that the reader is gradually led along the learning path. Each of the chapters contains a number of exercises. The reader is encouraged to attempt these exercises so that the instruction contained in the text may be reinforced by practice.

Chapter 10 explains how an occam program, built using the constructs described in the earlier chapters, may be distributed over a network of transputers; whilst Chapter 11 explores how occam can be used to develop parallel solutions to real problems via the algorithmic, geometric and process farming paradigms.

The appendix contains a condensed description of the Inmos Transputer Development System (TDS). Although TDS is part of the occam ethos and many readers will probably use TDS for the development of occam programs, this book is primarily concerned with occam 2, and TDS is not part of the occam 2 definition. Hence the description of TDS was left to the last.

A list of further reading appears at the end of the book. This list, containing references to books and articles in journals, will allow the reader to follow up and extend the concepts introduced in the text.
Acknowledgements

Many colleagues and students have contributed to the development of this book. In particular, I would like to thank Peter Strain-Clark of the Open University, Roger Peel of the University of Surrey (Pitman's reviewer), and Wael Amer, Donald Macfarlane and Mark Webb-Johnson of the University of Buckingham for reading the manuscript and for their useful and helpful comments and suggestions. Permission by Inmos Limited to adapt the examples of the Newton-Raphson method, the thermal conduction of a metal plate and the Mandelbrot set in Chapter 11 from their texts and materials is gratefully acknowledged. Conor O'Brien of Inmos kindly read and suggested improvements to this chapter. Thanks also go to Jeremy Dick of Racal Research Limited for his patient explanation of the intricacies of *troff*, Bill Butcher of the University of Buckingham for commenting on my spelling and grammar, and Roger Hill of Pitman Publishing for his encouragement. Lastly, thanks to Marilyn, Neil and Gail for being so patient and understanding and putting up with all those times when I disappeared back to the office to work on the book.
To Marilyn
Introduction

Occam and the transputer

Conventional programming languages such as Pascal and C operate in a sequential fashion where one program statement at a time is executed; the sequential nature being forced by the sequential architecture of conventional computers. This type of architecture was first proposed by von Neumann in 1946, and, by and large, since then the majority of successive generations of widely-available computers have kept to the same design - a single processor and memory units linked via a single data bus (Figure Intro.1). Instructions and data are fetched by the processor from memory using the data bus, processed, then any resultant data stored back in memory again using the data bus. This fetch and store cycle operates repeatedly to sequence through a program.

However, many problems such as vision, speech and language processing, simulation and digital signal processing have an inherent parallelism. Solving these problems in a sequential fashion is at best time-consuming and constraining. These problems would be better solved in parallel mode i.e. more than one statement at a time being executed to accommodate their natural parallelism. The algorithms for computing the solutions to these problems would need to be stated in parallel terms. This in turn requires a suitable programming language having facilities for expressing parallelism to be available. The parallel computation may be effected by having a multi-computer system, with each component computer acting on part of the problem, all component computers executing in parallel. Distributing the processing load over more than one computer, as would be possible with a multi-computer system, should increase the execution performance of the system. Various approaches to the introduction of parallel architectures have been made [Hockney and Jesshope]. There has been much research and development in recent years into pipelined vector processors and array processors, together with multi-processors.

Attempts have been made to extend the simple von Neumann design to multi-processors for parallel operation without much success. It was found that adding more processors to the data bus, after an initial improvement, caused a degradation in the overall performance. This is due to the increased competition for use of the shared data bus by each extra processor - a condition known as the von Neumann bottleneck. In the von Neumann design only one processor may have access to the bus at any one time. Other processors wanting to use the bus must wait until the bus is free. The sharing of a single data bus among many processors brings more problems than it solves. The programming languages used for these systems were, by and large, extensions of existing sequential-mode languages - parallel programming features were added as an afterthought. Conventional programming languages were not
initially designed to cope with parallelism.

![Block diagram of a conventional sequential computer system](image)

Figure Intro.1  Block diagram of a conventional sequential computer system

An interesting and easily understood analogy has been drawn between parallel processing and building a wall [Fox et al.]. This analogy identifies a number of concepts and problems involved in parallel processing. For example, employing many bricklayers corresponds to using multi-processors, collaboration between bricklayers corresponds to communication between processors, and so on.
Much current research is being devoted to the design of new parallel computer architectures and corresponding formalisms for programming languages. Occam (and the transputer) represent one of the success stories.

1 Occam

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 a 14th century Oxford philosopher, William of Occam. He was particularly famous for a quotation known as Occam’s Razor - "Entia non sunt multiplicanda praeter 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 use. 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 consumption. Its real-time features endow occam with the facilities necessary for the development of real-time applications.

Various systems for developing and executing occam programs are available. Probably the most common is the Inmos IMS B004 which is a plug-in card for the IBM PC (or compatible). This card, together with the Transputer Development System software (IMS D700D), provides an integrated environment for the development and execution of occam programs (see Appendix). Occam development systems are also available for other computer systems such as VAXes, Suns and Macs, and specialised transputer equipment such as Meiko’s Computing Surface.

2 The transputer

Occam was designed specifically for the transputer, a parallel architecture microprocessor developed by Inmos. The name transputer is actually a contraction of two words: transistor and computer.

The transputer is a new generation VLSI architecture which explicitly supports concurrency and synchronisation. Synchronisation is required when different parts of a system, each with different relative times of execution, need to interact or cooperate with each other in some way, for example when passing messages between each other. Each interacting part must be ready
and prepared for the interaction to take place. Otherwise either part may get out of step and the interaction may never take place.

The transputer is really a family of related microprocessors each consistent with the transputer architecture but with different capabilities.

For example,

- IMS T212 - 16-bit microprocessor
- IMS T414 - 32-bit microprocessor
- IMS T800 - 32-bit microprocessor plus floating point unit

Though the transputer family contains processors of differing wordlength, the instruction set used by occam has been designed to be independent of the processor’s wordlength. However the difference in wordlength does mean that there are a few variations in behaviour. For example the default integer size will vary, as will the real-time clock period.

The high performance transputer, the floating-point IMS T800, is an interesting example to study. This member of the family comprises a 32-bit microprocessor, a 64-bit floating-point unit, four high-speed serial communications links, 4 Kbytes of on-chip memory and an external memory interface, all on a single chip (Figure Intro.2). Thus the transputer is really a "microcomputer on a chip".

The microprocessor, running at 20 MHz, can deliver 10 MIPS. Its minimal instruction set and its small group of 32-bit registers are designed to maximise execution speed. It contains a microcoded priority scheduler which time-shares any number of concurrent processes.† Thus a single transputer, in addition to executing processes in conventional sequential mode, may execute processes concurrently. Being constructed in hardware, context switching between processes (needed for the concurrent execution of processes on the same processor) is very fast, being of the order of a few microseconds. There is thus a small but finite overhead penalty incurred in the execution of concurrent processes.

Two levels of process priority are supported by the scheduler - priorities 0 and 1, with priority 0 being the higher. At the low priority level, processes are timesliced in execution in round-robin fashion with a timeslice of approximately one to two milliseconds. The high priority level has no timeslicing. Once a high priority process starts execution it runs to completion, unless it requires inter-process communication or system services. High priority processes are executed in preference to low priority processes.

† A distinction is drawn between concurrent execution - processes executing in "parallel" on the same processor and parallel execution - processes executing in parallel on separate processors [Burns].
Figure Intro.2  Block diagram of the IMS T800
The floating point unit implements the IEEE floating-point arithmetic standard in hardware and provides both single (32-bit) and double (64-bit) length operations. It operates concurrently with the microprocessor and can deliver 1.5 MFLOPS.

Each transputer communication link can transfer data at over 1 Mbyte per sec., with automatic hand-shaking synchronisation in each direction, and provides a bi-directional, point-to-point connection between transputers or between transputers and external devices. A single transputer link implements two occam channels, one in each direction. Occam uses such channels for inter-process communication. All links operate concurrently with the microprocessor in a DMA, block transfer fashion. Each transputer may thus be linked to up to four other transputers. In this way, networks of transputers of various sizes and topologies may be built up. Each transputer in a network operates as an independent unit communicating as and when necessary with the other transputers to which it is linked. Data is transferred using a simple one byte protocol with acknowledgement, thus ensuring synchronisation.

The IMS T800 can directly address a memory address space up to 4 Gbytes. Memory above the on-chip 4 Kbytes is accessed via the external memory interface. The on-chip memory is used as each transputer's local memory. The memory interface also supports memory-mapped devices which may use DMA.

In addition each transputer has two 16- or 32-bit timers (depending on the particular transputer wordlength), one for each of the priority levels. The high priority timer is incremented every microsecond and cycles approximately every 4,295 seconds (approximately 71 minutes) on the IMS T414 and IMS T800, and approximately every 66 milliseconds on the IMS T212. One second corresponds to 1,000,000 clock ticks. The low priority timer, on the other hand, is incremented every 64 microseconds and cycles approximately every 76 hours on the IMS T414 and IMS T800, and approximately every 4 seconds on the IMS T212. One second corresponds to 15,625 clock ticks.

A transputer follows the von Neumann design - the microprocessor, floating point unit, memory and communications links are linked via a 32-bit wide data bus. However in a multi-transputer configuration, a transputer has sole use of its own on-chip and off-chip memory and thus does not have to compete with the other transputers for memory accesses for instructions and data on a shared data bus. Consequently networks of linked transputers should scale linearly in performance according to the number of transputers in the network. There is no von Neumann bottleneck to degrade performance. Groups of parallel processes comprising an occam program may be distributed over such a network of transputers. Each transputer will execute its own processes - any communication between parallel processes on different transputers being handled by the transputer links - and so the parallel execution of the program will be effected.

Many different network topologies may be created with a system of transputers by connecting up the four transputer links in different ways. These topologies range from pipelines, through rings, to hypercubes. Much of the interest in transputer networks stems from the fact that the networks are readily re-configurable into different topologies.
The normal method of program development using transputers would be to design, implement and test the occam program on a single transputer system, and then when satisfied, to distribute the component processes over a transputer network. This necessitates configuring or mapping processes to the transputers in the network - declaring which processes will execute on which transputer. As the inter-process communication protocol does not specify where any two communicating processes reside, this distribution is transparent to the logic of the program.

The architecture of the transputers so closely implements the occam language and occam so completely provides for control of the transputer hardware that, though occam is a high-level language, it may be regarded as the "assembly language" of the transputer. Indeed this extremely close association between hardware and software is instrumental in producing such a powerful combination as occam and the transputer.

3 The origins of occam

The design of occam was heavily influenced by the work of Hoare on the theoretical model of Communicating Sequential Processes (CSP) which grew out of a study of process synchronisation problems. CSP is a mathematically-based notation for specifying the behaviour of 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 processes 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 processes 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 \oplus_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

In a similar fashion the occam language contains statements which allow the development of serial and/or concurrent processing of programs. An occam program comprises a collection of processes. Any concurrent processes which need to interact are completely synchronised i.e. communicate with each other in the fashion of CSP. An occam program (a collection of processes) may reside on a single transputer or a network of interconnected transputers (Figure Intro.3). In the former case the transputer shares its time between the concurrent processes - a sort of pseudo-parallelism. (Because the necessary process switching to perform concurrent processing is built in hardware on the transputer, it is inherently fast and does provide pseudo-parallelism.) In the latter case the processes are distributed in some way over the network of transputers and each transputer executes the processes allocated to it. This is true parallelism.

Occam is not just a programming language. It has been used in design as a specification language for both software and hardware systems. Furthermore, mathematical proof techniques which allow an occam program (or specification) to be verified for correctness have been developed.

The first version of the language (now called proto-occam or occam 1) was introduced in 1982. This was superseded by the latest version, occam 2, in 1986. Occam 2 has the following additional features:

- an increase in the number of data types supported (see Chapter 2)

- a complete change in the syntax for declaring program constants and variables - occam 1 being an untyped language, whilst occam 2 is typed (see Chapter 2)

- the introduction of protocols for describing data transfer on the occam channels (see Chapter 8)
Figure Intro.3  An occam program on a single transputer or a network of transputers
• a refinement of occam timer facilities (see Chapter 9)

• a change in the syntax for configuring transputer networks (see Chapter 10)

• the introduction of multi-dimensional arrays and an enhanced syntax for array operations (see Chapter 4)

• an increase in the number of occam reserved words

This book is concerned with the description of occam 2.

Extra reading

1. "Communicating sequential processes" by C.A.R. Hoare, Communications of the A.C.M., vol. 21, pp. 666-677, 1978. This is the original article on CSP.

2. "Communicating Processes and occam", Technical Note 20, Inmos Ltd. This is a short description of occam by its chief designer, David May.

3. "IMS T800 Architecture", Technical Note 6, Inmos Ltd. This is a description of the T800 transputer by staff of Inmos.
Chapter 1

Occam basics

As explained earlier, occam is based on the process model of CSP - an occam program comprises a collection of processes, some of which may be executing in sequence, others of which may be executing in parallel. Those processes executing in parallel may only communicate with each other via synchronised input/output operations.

Following the CSP model, the three most common basic, or primitive, processes in occam are defined to be

- the assignment process - assign a value to a variable
- the input process - input a value into a variable
- the output process - output a value from a variable

The remaining primitive processes are SKIP and STOP; the former being a process which terminates immediately, the latter being a process which never terminates. Their function will be described in more detail shortly.

More higher-level processes may be built from these "building blocks" using occam constructions. A construction encompasses a collection of component processes, the whole being yet another occam process. Thus a process in occam may be as simple as a single statement (one of the primitive processes) or more "complex" (a collection of processes built from primitive processes using constructions). The process formed by a construction may itself be used as the component of another construction. So a hierarchical or nested structure of occam processes may be formed. Constructions are introduced by occam reserved words. The most important and interesting of these are SEQ - the sequential construction, PAR - the parallel construction - and ALT - the alternation construction. Others will be described in Chapter 5.

Occam is a typed language (see Chapter 2). The name and type of program objects such as constants and variables must be declared before use. Data declaration statements are prefixed to the processes concerned. Scoping rules apply so that the data is local to the following process. Processes nested within other processes may access this data - these nested processes are within the scope of the enclosing process. Processes within a parallel construction however may only share data via an occam channel - a shared, common variable may not be used to communicate values between parallel processes. The channel is used for communication
between parallel processes. The need for this constraint will be explained shortly. The syntax of occam is reasonably straightforward. Each occam language statement (a primitive process, a construction reserved word, etc.) occupies a line by itself. Importantly, the components of a higher-level process are indented by two spaces with respect to the construction reserved word. This indentation reflects the block structure of the occam program and replaces the BEGIN and END statements of conventional block-structured languages. Processes at the same level will have the same indentation. The end of any level of indentation marks the end of the process. The level of indentation of a process marks the region of its validity. The process is defined only within the range of its indentation, as is the scope of its local variables. A long occam statement may be continued over more than one line provided that the statement is broken immediately after an operator (see Chapter 3), a comma, a semi-colon, an assignment operator (:=) or the occam reserved words IS, FROM or FOR. The continuation lines must be indented at least as far as the initial line of the occam statement.

1.1 Primitive processes

The primitive processes - assignment, input and output - are described in this section. The variables used in the examples are assumed to have previously been declared. How this is done in practice is left until Chapter 2.

1.1.1 Assignment

The assignment process assigns a value to a named occam program variable. It has the form

\[
\text{variable} := \text{expression}
\]

where

- \text{variable} is an occam program variable identifier, and

- \text{expression} is an occam expression. The value of the expression is assigned to the variable. In occam an expression may take many forms. For the moment consider an expression to be a constant, a variable or some simple arithmetic combination of these. Expressions will receive further coverage in Chapter 3.
The data type of the expression must be the same as the data type of the variable to which the value of the expression is being assigned (see Chapter 2).

For example:

\[
\text{Index} \ := \ 0
\]

is an assignment process which assigns a value of zero to the integer variable \textit{Index} (see Chapter 2 for a definition of occam types).

and

\[
\text{Left} \ := \ \text{Limit} - \ \text{Index}
\]

is an assignment process which assigns the value resulting from the evaluation of the arithmetic expression, \textit{Limit} - \textit{Index}, to the variable \textit{Left}.

Multiple assignments may be made within one statement if required.

For example,

\[
\text{Total}, \ \text{Index} \ := \ 0, \ \text{Index} + \ 1
\]

is equivalent to the two separate assignments

\[
\text{Total} \ := \ 0
\]

and

\[
\text{Index} \ := \ \text{Index} + \ 1
\]

Multiple assignments may be used in occam to interchange the values of variables.

For example,

\[
\text{Item1}, \ \text{Item2} \ := \ \text{Item2}, \ \text{Item1}
\]

successfully swaps the values of variables \textit{Item1} and \textit{Item2}, and is equivalent to the statements

\[
\begin{align*}
\text{Temp} & \ := \ \text{Item1} \\
\text{Item1} & \ := \ \text{Item2} \\
\text{Item2} & \ := \ \text{Temp}
\end{align*}
\]

where \textit{Temp} is a temporary variable.
1.1.2 Input and output

The input and output processes operate via channels and provide inter-process communication between concurrent processes. A channel is a one-way communications link between two concurrent processes. The channel is used to pass data from one concurrent process to another. A channel is shared between only two communicating processes - one process may output on the channel, the other may input. No more than one process may use the channel for either input or output. Data which has been assigned or input by a process within a parallel construction may only be shared with another process in the construction via an occam channel. Both the input and output processes must both be ready before the data transfer can take place i.e. there is complete synchronisation of channel communications. This synchronisation is automatically provided by occam.

The effect of only two concurrent processes being allowed access to any channel is illustrated when keyboard and screen input/output is performed via channels from a program comprising concurrent processes. Only one of these processes may access the keyboard or screen. Some sort of multiplexing must be provided to allow multiple input/output streams.

The channel is realised in practice by software (via memory locations), if the communicating processes are resident on the same transputer, or by hardware (via the transputer's communications links), if the communicating processes are distributed across transputers. The channel is a feature of the occam language and is implemented transparently by the occam environment.

![Figure 1.1 An input channel and process](image-url)
1.1.2.1 Input

The input process allows a value to be input from an occam channel and that value to be assigned to a named variable (Figure 1.1). The input process has the form

```
channel ? variable
```

where

- `channel` is an occam channel identifier, and
- `variable` is an occam variable which receives the value input along the channel.

The input value must be of the same data type as the named variable (see Chapter 2).

For example,

```
InChan ? Data
```

inputs a value via the channel `InChan` and assigns that value to the variable `Data`.

1.1.2.2 Output

The output process outputs the value of an expression along a named channel (Figure 1.2). It has the form

```
channel ! expression
```

where

- `channel` is an occam channel identifier, and
- `expression` is an occam expression (see Chapter 3)
For example,

```
OutChan ! Data
```

outputs the value of the variable \textit{Data} to the channel \textit{OutChan}, and

```
OutChan ! 2 * Data
```

outputs twice the value of the variable \textit{Data}.

![Diagram](image)

Figure 1.2 An output channel and process

### 1.2 The SEQ construction

The sequential construction causes component processes to be executed one after the other. This is just the normal mode of execution of conventional computers. However, because with occam the programmer has the option of executing processes in sequence or in parallel, sequential execution \textit{must} be explicitly specified. Sequential execution is not a default in occam as it is in conventional programming languages.

Each sequential construction is introduced by the reserved word \textbf{SEQ}, and this is followed by a list of processes which are to be executed in sequence.
The sequential construction has the format

$$\text{SEQ}$$

$$\quad \text{process } 1$$

$$\quad \cdot$$

$$\quad \cdot$$

$$\quad \cdot$$

$$\quad \text{process } n$$

where process 1 . . . process n represent a collection of processes to be executed in sequence and each one must be indented by two spaces from the SEQ reserved word. These processes are the components of the SEQ construction and each may be one of the primitive processes or a more "complex" process. Each component process is executed in the sequential order specified.

The SEQ construction terminates when the last of the component processes terminates. (In occam, a process, be it primitive or complex, is said to terminate if it successfully finishes execution - see section 1.9 of this chapter.) A SEQ construction without any component processes behaves as a SKIP process.

Example 1.1

SEQ

Index := 0

Limit := 100

Left := Limit - Index

is a sequential process which assigns a value of 0 to the variable Index, a value of 100 to the variable Limit and the arithmetic difference of Limit and Index to Left. (As commented on earlier, these variables must have been previously declared.) Each assignment is executed one after the other i.e. in sequence. The SEQ construction ends when the last of its component processes terminates - in this case the assignment to Left.
Example 1.2

SEQ
A := 6
B := A + 2
C := A * B
D := 4
D := 2 * D

is a sequential process containing a mixture of simple and arithmetic expression assignments.

Example 1.3

SEQ
Chan ? Data
Data := Data + 1

is a sequence of statements comprising an input, via channel Chan, and an assignment.

![Process diagram](image)

Figure 1.3 A simple buffer process

Example 1.4

SEQ
In ? Num
Out ! Num

shows a process which inputs a value from the channel In and outputs that value on the channel Out. The SEQ construction in this example behaves as a simple buffer (Figure 1.3).
This illustrates occam’s usefulness in the fabrication of "building block" processes.

Example 1.5

\[
\text{SEQ} \\
\text{InChan} ? \text{Num} \\
\text{OutChan} ! \text{Num} * \text{Num}
\]

is a process which inputs a value from the channel \text{InChan} and outputs the square of that value on the channel \text{OutChan}.

Note that in these examples the cooperating processes in the communication have been left unspecified. Proper communication between processes will be discussed shortly.

1.3 The PAR construction

The parallel construction causes component processes to be executed concurrently, each at its own rate. Each process within a parallel construction starts execution at the same time as all the other processes within the same parallel construction.†

A parallel construction comprises the occam reserved word \text{PAR} followed by a list of processes which are to be executed in parallel.

The parallel construction has the format

\[
\text{PAR} \\
\text{process 1} \\
\vdots \\
\text{process n}
\]

where \text{process 1} . . . \text{process n} represent a collection of processes to be executed in parallel, and each one may be one of the primitive processes or a more "complex" process. These processes are the components of the \text{PAR} construction. Each of the component processes must be indented by two spaces from the \text{PAR} reserved word.

† If the component processes reside on a single transputer, then the processes will be executed concurrently or in pseudo-parallel. If the component processes are distributed over a network of transputers, then the processes will be executed in true parallel.
The **PAR** construction terminates only after all of the component processes have terminated. The component processes do not necessarily terminate together since they are quite likely to be different processes. A **PAR** construction without any component processes behaves as a **SKIP** process.

![Figure 1.4 Process with two input channels](image)

**Example 1.6**

```
PAR
  Comm1 ? Item1
  Comm2 ? Item2
```

represents two inputs executing concurrently, a value being input into variable *Item1* via channel *Comm1* and another value being input into variable *Item2* via channel *Comm2* (Figure 1.4).

The fact that each process within a **PAR** construction starts execution at the same time as all the other processes within the construction means that the named order of processes within a parallel construction is immaterial. (The same is *not* true for the **SEQ** construction since, as its component processes are executed in sequence, the named ordering is essential)

For example,

```
PAR
  Comm1 ? Item1
  Comm2 ? Item2
  Comm3 ? Item3
```
is equivalent to

\[
\text{PAR} \\
\text{Comm3 ? Item3} \\
\text{Comm2 ? Item2} \\
\text{Comm1 ? Item1}
\]

or any other permutation of component process order.

![Diagram](image)

Figure 1.5 The ALT process

### 1.4 The ALT construction

The alternation construction allows a particular process from a list of component processes, or *alternatives*, to be selected for execution. In the simplest case, each component process is *guarded* by an input process. Which alternative is selected for execution depends on which input guard has input available first. The process associated with the first input guard to be ready is the one chosen for execution. Thus essentially the ALT construction selects an available input from a number of input channels (Figure 1.5). For a discussion of what happens if two or more input guards are ready at the same time see Chapter 5.

An alternation construction is introduced by the occam reserved word **ALT** and this is followed by a list of processes guarded by inputs.
The alternation construction has the format

\[
\text{ALT} \\
\quad \text{input } 1 \\
\quad \quad \text{process } 1 \\
\quad \ldots \\
\quad \ldots \\
\quad \text{input } n \\
\quad \quad \text{process } n
\]

where \text{input } 1 \ldots \text{input } n\) represent the input guards and \text{process } 1 \ldots \text{process } n\) represent the associated processes, each one of which may be one of the primitive processes or a more "complex" process. Each input guard must be indented by two spaces from the \text{ALT} reserved word and each associated process indented a further two spaces.

After the execution of the selected process, the \text{ALT} construction terminates.

![Diagram of a three-channel ALT process](image)

Figure 1.6 A three-channel ALT process
Example 1.7

ALT
  InChan1 ? Data1
  OutChan ! Data1
  InChan2 ? Data2
  OutChan ! Data2
  InChan3 ? Data3
  OutChan ! Data3

is an alternation construction which has three alternatives, each guarded by an input process. If the input on channel \textit{InChan1} becomes ready first then its associated process (\textit{OutChan} ! \textit{Data1}) is the one which is executed, and similarly for the other two input guards. This ALT process behaves as a multiplexor - accepting data from a number of input channels and feeding that data down a single output channel (Figure 1.6).

ALT enables the creation of event-driven software, a particular event being associated with a particular input channel; an interrupt handler being an example of this. The ALT construction is treated more thoroughly in Chapter 5.

1.5 Nested constructions

As discussed earlier, constructions may be nested within other constructions. In this way it is possible to build "complex" occam processes from simpler ones.

Example 1.8

PAR
  SEQ
    Index := 0
    Chan1 ! Index
  SEQ
    Limit := 100
    Chan2 ! Limit

represents a parallel process with two component sequential processes. These sequential processes, being the components of a PAR construction, execute concurrently.
Example 1.9

```
PAR
  SEQ
  Result := 42
  Comm ! Result
  SEQ
  Comm ? Answer
```

is a parallel process with two sequential components, one containing an output process, the other containing an input process. Note that both the input and output processes are using the same channel - the sequential processes executing concurrently have the ability to communicate via this channel (Figure 1.7). The components of a parallel process may only communicate via the same named channel and this channel may only link two processes. Note also that the communication is strictly one-way - in this case output from the first sequential process, input into the second sequential process.

In general, with two communicating processes, at some point one process will wish to output to the other process. This output will not continue until the execution of the second process reaches the appropriate input. Correspondingly the second process may reach the input before the first process is ready to output. In this case the input is suspended until the first process wishes to communicate. The communication is fully synchronised by occam.
1.6 Two-way channel communication

It should be reiterated that the same channel may not be used for both input and output within the same process. Two-way communication between processes requires two separate channels.

Example 1.10

```
PAR
  SEQ
    Value := 21
    Comm1 ! Value
    Comm2 ? Answer
  SEQ
    Comm1 ? Data
    Comm2 ! 2 * Data
```

is a parallel process with two sequential components. Each sequential component has two distinct channels, *Comm1* and *Comm2* (Figure 1.8). These channels allow two-way communication between the components of the **PAR** construction. The first component outputs via channel *Comm1* and inputs via channel *Comm2*, whilst the second component inputs via channel *Comm1* and outputs via channel *Comm2*.
1.7 Deadlock

When processes are participating in two-way communication, it is important to order the respective input and output processes in such a way so as to avoid deadlock. Deadlock is a situation which arises when one or more processes for some reason cannot proceed. They are held up waiting for an event which will never happen. Special care is needed to ensure that communicating processes are not held up waiting for an input or output that cannot proceed.

![Figure 1.9 Possibility of deadlock between processes](image)

**Example 1.11**

PAR
SEQ
Chan1 ! Data1
Chan2 ? Data2
SEQ
Chan2 ! Item2
Chan1 ? Item1

The first sequential process tries to output on Chan1 but is held up until the corresponding input is ready in the second sequential process. However the second sequential process can never reach this input because it will be held up as it tries to output on Chan2. There is deadlock - neither process can proceed (Figure 1.9).

As demonstrated above, deadlock can be a real problem when processes participate in two-way communication. The problem in this case usually arises from the sequential ordering of the communicating processes. Deadlock may be avoided by carefully ordering the sequence of inputs and outputs. Other possibilities of avoiding deadlock are the use of a PAR construction
instead of a SEQ for the inputs and outputs - if the sequence of inputs and outputs are independent - or the use of an ALT construction to avoid sequencing the inputs. However each case must be carefully considered for the best solution and a cavalier approach to the problem must be avoided at all costs.

Example 1.12

PAR
  SEQ
    Chan1 ! Data1
    Chan2 ? Data2
  SEQ
    Chan1 ? Item1
    Chan2 ! Item2

where the sequence of input and output in the second sequential process has been reordered to remove the deadlock.

Example 1.13

PAR
  SEQ
    Chan1 ! Data1
    Chan2 ? Data2
  PAR
    Chan2 ! Item2
    Chan1 ? Item1

where the SEQ construction of the second sequential process has been replaced by a PAR construction to remove the deadlock. Since the input and output in this new parallel process now execute concurrently and not in sequence, they will not be held up by their ordering. The same effect may have been gained by replacing the first SEQ construction by a PAR, or replacing both SEQ constructions by PARs, provided the logic of the program allows these inputs and outputs to be independent.

It must be stressed that these are artificial examples used to illustrate a point. What is possible in practice depends on the logic of the program.
1.8 SKIP and STOP

SKIP and STOP are the two remaining occam primitive processes. SKIP starts, does nothing and then terminates immediately. It behaves like a null process - program execution continues afterwards with any subsequent process. A seemingly innocuous statement, it has many uses (see Chapter 5, for example).

STOP starts, does nothing but never terminates. Program execution is held up after a STOP - as it never terminates. It behaves like a process in tight loop, which loops for ever. As such, it is to be used with caution - usually it will be used in those circumstances which are illegal in the logic of the program. A run-time error usually causes a rogue process to behave as a STOP process.

1.9 Termination of occam programs

In occam, a clear distinction is made between the notions of terminated and stopped when applied to processes. When a process finishes execution cleanly, it is said to have terminated. A process which has stopped, however, is an error condition. A process may stop as a consequence of executing a STOP process or after incurring a run-time error. The Inmos Transputer Development System (Chapter 12) admits three modes for stopped processes at run-time, a particular mode being chosen as a compiler option. The modes are as follows

- **HALT** - the system (transputer) containing the stopped process is brought to a halt. In a network of transputers, any connected processors will deadlock if they try to communicate with the halted processor. This mode is the default option.

- **STOP** - any stopped process behaves as a STOP process. The effect of a stopped process will slowly percolate through the whole system. For example, processes which try to communicate with a stopped process will hang; a PAR construction, with a constituent process which is stopped, will never terminate.

- **REDUCED** - any stopped process is ignored. This mode is only used in exceptional circumstances.

The organisation required to terminate a program comprising a collection of parallel processes needs especial care, and the onus for doing this falls on the programmer. Parallel processes need to be terminated in an orderly fashion. A parallel process must not be terminated before any processes communicating with it have been informed, as these processes may hang if they
are dependent on some communication from the terminated process. This area is still subject
to much debate and discussion, but a number of simple guiding principles have appeared
[Pountain and May].

- processes must be informed when to terminate. For example by setting a
  boolean flag as a termination notice.

- a process must pass on any termination notice to other communicating
  processes before terminating itself. Not all processes may be informed at once.
  It takes time for this termination notice to spread around the system.

- channels, and not shared variables, must be used for delivering the termina-
  tion notice

1.10 Comments in occam programs

A comment in an occam program is introduced by the double dash (--), and is terminated by
the end of the line on which it appears. Comments must be indented at least as far as the
statement on the preceding line. A comment may be added on the same line as a process.

Example 1.14

SEQ
  -- A comment here
    -- or here is legal
    .
    .
    .
    SEQ
  -- BUT not here !

Example 1.15

SEQ
  Index := 0            -- initialise Index,
  Limit := 100         -- Limit
  Left := Limit - Index    -- and Left
Exercises

1. An assignment, such as \( \text{Left} := \text{Limit} - \text{Index} \), may be written as a \textbf{PAR} construction. Write down a suitable \textbf{PAR} construction to do this.

2. Comment on the validity of the following occam code fragments.

   a)
   
   \begin{verbatim}
   PAR
   SEQ
   Item := 0
   SEQ
   Item := Item + 1
   \end{verbatim}
   
   where the variable \textit{Item} has been declared outside the \textbf{PAR} construction.

   b)
   
   \begin{verbatim}
   SEQ
   Chan ? Item
   Chan ! Data
   \end{verbatim}

   c)
   
   \begin{verbatim}
   PAR
   SEQ
   Chan1 ! A
   Chan2 ? B
   SEQ
   Chan2 ! C
   Chan1 ? D
   \end{verbatim}

   d)
   
   \begin{verbatim}
   PAR
   PAR
   Chan1 ! A
   Chan2 ! B
   PAR
   Chan2 ? C
   Chan1 ? D
   \end{verbatim}
e) PAR
   SEQ
   Chan1 ? A
   Chan2 ? B
   SEQ
   Chan2 ! C
   Chan1 ! D

3. Rewrite the following program fragment to remove the deadlock
a) by reordering the sequence of inputs and outputs, and
b) by using a PAR construction as described in the text (assuming the sequence of inputs and outputs is independent).

PAR
   SEQ
   Chan1 ? A
   Chan2 ? B
   SEQ
   Chan2 ! C
   Chan1 ! D
Chapter 2

Data basics

In occam, program objects such as constants and variables must be declared before use - the name or identifier of the program object and its type must be specified. There are a number of typed objects used in occam

- data types - these are the familiar integer, floating-point, etc. types

- channels - these are inter-process communication links which allow values to be communicated between concurrent processes

- timers - these are special input-only channels which provide access to the transputer’s real-time clock facility

Channels and timers will be dealt with in Chapters 8 and 9 respectively. Data types (and simple channels) will be discussed in this chapter.

2.1 Identifiers

Occam is a strongly typed, hierarchical block-structured language. That is to say, program objects such as constants and variables together with their data type must be declared before they are used. Moreover, operations defined for a particular data type, for example addition or comparison, may only be performed with constants and variables of that data type. Program objects may be declared throughout the program at the head of a suitably indented process, making the objects local to that process.
The identifier of a constant or variable in an occam program

- may be of any length
- must start with a letter, but may include digits and full-stops
- is case-sensitive

Occam has a number of reserved words. These words must always be written in upper-case. Within this book, reserved words will appear in **bold-face** type. A constant or variable identifier must not clash with a reserved word.

As occam is case-sensitive, the identifiers `count` and `COUNT`, for example, represent different objects.

There are two common conventions in use for writing an identifier in an occam program

a) the identifier is written in lower-case, but with the initial letter in upper-case

For example,

```
Count

Temperature

Average
```

If the identifier comprises more than one word, the initial letter of each word will be written in upper-case.

For example,

```
SensorReading

CharacterCount

EndOfFile
```
b) the identifier is written in lower-case. For example,

```
count

temperature

average
```
If the identifier comprises more than one word, each word is separated by a full-stop. For example,

```
sensor.reading

count.character

end.of.file
```
This book adopts the former convention.

The following are invalid occam identifiers

```
1Data - starts with a digit

count$ - includes a $ character

Time_Out - includes an underscore character

TIMER - clashes with an occam reserved word
```
2.2 Data types

The basic or *primitive* data types available for constants and variables in occam 2 are given by the following reserved words:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>representing a small integer (0 to 255) or a character</td>
</tr>
<tr>
<td>INT16</td>
<td>representing a signed integer occupying 16-bits</td>
</tr>
<tr>
<td>INT32</td>
<td>representing a signed integer occupying 32-bits</td>
</tr>
<tr>
<td>INT64</td>
<td>representing a signed integer occupying 64-bits</td>
</tr>
<tr>
<td>REAL32</td>
<td>representing a signed real occupying 32-bits</td>
</tr>
<tr>
<td>REAL64</td>
<td>representing a signed real occupying 64-bits</td>
</tr>
<tr>
<td>BOOL</td>
<td>representing a truth value</td>
</tr>
</tbody>
</table>

A further data type, INT, is also defined. This type represents a signed integer, occupying an implementation-defined number of bits. For a T212 transputer (16-bit wordlength), INT values lie in the range -32768 to 32767, whilst for a T414 and a T800 transputer (32-bit wordlength), INT values lie in the range -2147483648 to 2147483647. Integers (INT, INT32 and INT64) are represented internally in two’s complement form.

The real types, REAL32 and REAL64, internally use the ANSI/IEEE floating point standard representation, with 1 sign bit, 8 or 11 exponent bits, and 23 or 52 fraction bits respectively. The range for REAL32 values is -3.40282348 E38 to 3.40282347 E38, and the range for REAL64 values is -1.7976931348623158 E308 to 1.7976931348623157 E308.

The BYTE type, representing an unsigned integer in the range 0 to 255 or a character, is stored in 8-bits. The BOOL type is also stored in 8-bits - TRUE having the numeric value 1, FALSE having the numeric value 0 - although this is not part of the occam language definition.
2.3 Literals

A literal is the textual representation of a known value, and may be used, for example, to assign a constant value to a variable or define a constant within an occam process. The format of the literal will depend on the data type required. Importantly, for literals of type BYTE, INT16, INT32, INT64, REAL32 or REAL64, the data type must be explicitly specified within brackets immediately following the literal value wherever the literal is used in any occam statement. The exceptions to this rule are

- integer literals of type INT
- single character and character string literals
- boolean literals

No explicit typing is required for these literals.

a) integer literals

Integer literals are signed decimal (whole) numbers. Valid data types for integer literals are INT, INT16, INT32 and INT64. Additionally, the BYTE data type may be used for an integer literal if the literal numeric value lies in the range 0 to 255 (but their use in expressions is restricted to comparisons - see Chapter 3).

Examples are valid integers are

11

-273

32767

-90

Examples of invalid integer forms are

6,536 - comma not allowed

13.0 - decimal point not allowed for an integer
Example 2.1

AbsoluteZero := -273
assigns the integer value -273 to the INT variable AbsoluteZero. (No explicit typing is required for INT literals. If no data type is specified for an integer literal, then by default type INT is assumed.)

Count := Count + 1
adds the integer value 1 to the INT variable Count and then assigns the result to Count.

Maximum := 32767 (INT16)
assigns the integer value 32767 to the INT16 variable Maximum.

SmallNumber := 32 (BYTE)
assigns the byte value 32 to the BYTE variable SmallNumber.

An integer literal may also be expressed as a hexadecimal (base 16) number. Such a number must be preceded by a hash (#).
Examples of hexadecimal integers are

#FC

#1FFF

Example 2.2

BitMask := #1F
assigns the integer value #1F to the INT variable BitMask.
b) real literals

A real literal is a signed, decimal number (with the decimal point and fractional part), optionally followed by an exponent value. Allowable data types for real literals are REAL32 and REAL64.

Examples of valid real forms are

- 2.0
- -0.00005
- 1.0E-6
- -1.3E3

Examples of invalid real forms are

- 235 - no decimal point
- -5,000.0 - comma not allowed
- 2. - no fractional part

Example 2.3

Epsilon := 1.0E-6 (REAL64)

assigns the real value 1.0E-6 to the REAL64 variable Epsilon.

Double := 2.0 (REAL32) * Single

multiplies the variable REAL32 Single by the real value 2.0 and then assigns the result to the REAL32 variable Double.
c) character literals

A character literal is a single character enclosed by single quotes ('). Single characters are represented internally by their numeric ASCII code values as a single byte.

Examples of character literals are

'J'

'E'

'G'

Example 2.4

Char := 'M'

assigns the single character 'M' to the BYTE variable Char. (No explicit typing is required for a character literal.)

A, E, I, O, U := 'a', 'e', 'i', 'o', 'u'

assigns the characters 'a', 'e', 'i', 'o', 'u' to the BYTE variables A, E, I, O, U respectively.

It is possible to specify that a character be of type INT instead of the default BYTE type. To do this, the INT type must be specified in brackets after the character.

For example,

Return := '*c' (INT)

specifies that the character '*c' (carriage return) is to be of non-default type and assigns the character to the INT variable Return.

Certain characters, for example single and double quote marks, and non-printable control characters, for example carriage return and tab, are specified with a different format as follows

\[
\text{\texttt{*code}}
\]

where code is a specific letter for each character.

The table on the following page lists those characters which must be specified with this format.
\*c - carriage return
\*n - line feed
\*t - tab
\*s - space
\' - single quote
\" - double quote
** - asterisk

Alternative denotations for \*c, \*n, \*t and \*s are \*C, \*N, \*T and \*S respectively.

Example 2.5

Asterisk := '**'

assigns the character '***' (asterisk) to the BYTE variable Asterisk.

Space := '*s'

assigns the character '*s' (space) to the BYTE variable Space.

Note that the space character may be represented as

Space := ' *s'
or

Space := ' '

Any character may also be written in the form

\*hex_code

where hex_code is the ASCII value (expressed in hexadecimal) of the character.
Example 2.6

Bell := ' *#07'

assigns the character constant having value #07 (control G) to the BYTE variable Bell.

d) string literals

A string literal is a character string enclosed by double quotes ("). Strings are represented internally as an array of bytes of the corresponding ASCII code values (see Chapter 4.5). Examples of character strings are

"Greetings!"

"Time flies"

Example 2.7

Message := "Hello"

assigns the character string "Hello" to the variable Message. Since a string is treated as an byte array, Message would be declared as a BYTE array variable of size 5.

Greetings := "Have a nice day*c*n"

assigns the character string "Have a nice day" plus the carriage return and line feed characters to the BYTE array variable Greetings.

Prompt := "****>

assigns the character string comprising two asterisks and angle bracket to the BYTE array variable Prompt.

e) a boolean literal is a truth value, denoted by the occam reserved words TRUE and FALSE. Booleans are represented internally as a single byte.

Example 2.8

EndOfFile := FALSE

assigns the boolean value FALSE to the BOOL variable EndOfFile.
2.4 Declarations

The identifiers and types of all constants and variables in an occam program must be declared before use, each declaration statement being terminated by a colon (:). The colon "fixes" or binds the declaration to the process which immediately follows it. The declarations are local to that following process and any component processes (see Section 2.7 of this Chapter). Any declaration statement must be indented by the same amount as the process to which it belongs. Occam does not force a particular ordering on any declarations. A common ordering within an occam process is as follows [Pountain and May]

- channels
- timers
- abbreviations (constants)
- variables

This chapter introduces simple i.e. scalar constants and variables. Later chapters (Chapters 4 and 7) show how structured constants and variables may be declared using arrays.

2.4.1 Simple constants

A simple constant is an example of an occam abbreviation. Abbreviations are an important feature of occam and will be described fully in Chapter 7. A constant declaration specifies the data type of the constant and assigns that constant a value.

It has the format

\[ \text{VAL} \ type \ constant \ IS \ constant\_expression:\]
where

- `type` is the data type of the constant. The inclusion of `type` is optional and may be omitted if the type of the constant may be determined from the type of `constant_expression`,

- `constant` is the occam identifier of the constant, and

- `constant_expression` is the value which is assigned to the constant, and may be a literal or an expression (see Chapter 3) which evaluates to a constant value. The data type of `constant_expression` must be the same as `type`, if `type` is specified.

Example 2.9

```
VAL BYTE Limit IS 255 (BYTE) :

VAL INT Emergency IS 999 :

VAL INT OneThousand IS Emergency + 1 :

VAL INT16 Maximum IS 32767 (INT16) :

VAL REAL32 Pi IS 3.14159 (REAL32) :

VAL REAL32 TwoPi IS 2.0 (REAL32) * Pi :

VAL REAL64 Epsilon IS 1.0E-6 (REAL64) :
```

are simple constant declarations which use the long-hand form of specification. For example, `Limit` is specified to be of type `BYTE` and have a byte value of 255, `Emergency` is specified to be of type `INT` and have an integer value of 999, and `Pi` is specified to be of type `REAL32` and have a real value of 3.14159. The declarations for `OneThousand` and `TwoPi` are examples of expressions which evaluate to constants.

As noted above, `type` may be omitted if the type of the constant may be determined from the data type of `constant_expression`. Thus the above constant declarations may be written less tediously as shown overleaf.
Example 2.10

VAL Limit IS 255 (BYTE) :

VAL Emergency IS 999 :

VAL OneThousand IS Emergency + 1 :

VAL Maximum IS 32767 (INT16) :

VAL Pi IS 3.14159 (REAL32) :

VAL TwoPi IS 2.0 (REAL32) * Pi :

VAL Epsilon IS 1.0E-6 (REAL64) :

In accordance with the previously stated rule, explicit typing of INT, single character, character string and boolean literals may be omitted. This is illustrated in the following example.

Example 2.11

VAL Space IS ' ' :

VAL Bell IS '*#07' :

VAL Initial IS 'J' :

VAL Asterisk IS '**' :

VAL Message IS "Hello" :

VAL Prompt IS "*c*n>" :

VAL Updating IS TRUE :

In the above, the character string constants, Message and Prompt, are effectively byte arrays of size five and three respectively (see Chapter 4).
Example 2.12

```occam
VAL  Pi  IS  3.14159 (REAL32) :
VAL  TwoPi  IS  2.0 (REAL32) * Pi :
SEQ
  Radius := 12.0 (REAL32)
  Circumference := TwoPi * Radius
  RadiusSquared := Radius * Radius
  Area := Pi * RadiusSquared
```

shows the use of constant declarations in an occam fragment. (The occam variables, \( \text{Radius} \), \( \text{Circumference} \), \( \text{RadiusSquared} \) and \( \text{Area} \), must also be declared - how this is done will be explained in the next section.)

### 2.4.2 Variables

A variable declaration specifies the data type of a variable. It has the format

```
    type  variable :
```

where

- `type` specifies the data type of the variable, and
- `variable` is the occam identifier of the variable

Example 2.13

```occam
BYTE  Char :

INT  Index :

REAL32  Average :

BOOL  EndOfFile :
```

declare variables `Char` of type `BYTE`, `Index` of type `INT`, `Average` of type `REAL32`, and `EndOfFile` of type `BOOL`. 
More than one variable of the same type may be declared within the same specification - the identifiers being separated by commas.

For example,

```
INT Count, Sum :

REAL32 Average, Total, Error :

BOOL EndOfLine, EndOfFile :
```

Example 2.14

```
VAL Pi IS 3.14159 (REAL32) :
VAL TwoPi IS 2.0 (REAL32) * Pi :
REAL32 Radius, RadiusSquared, Circumference, Area :
SEQ
  Radius := 12.0 (REAL32)
  Circumference := TwoPi * Radius
  RadiusSquared := Radius * Radius
  Area := Pi * RadiusSquared
```

shows a previous example, but this time with the variables declared.

### 2.5 Simple channels

The channels used by a process must be declared in the same way as the constants and variables used by the process must be declared. As noted in Chapter 1, two processes wishing to communicate must each use the *same* channel. One process will be the output process, the other will be the input process.

A channel has an associated *protocol*. The protocol defines the type of data (i.e. the data type and format of the variables) to be transferred along the channel in any communication - the associated protocol effectively types the channel. The channel syntax allows for this protocol to be explicitly specified. Thus it must be declared whether the channel will support the transfer of three integer values, an integer followed by two real values or whatever. Protocols will be described fully later on in this book (Chapter 8). For now, only *simple* protocol channels - channels which allow the communication of a single value of a single data type at a time - will be considered. Channels may be used over and over again for the communication of data as specified by the protocol i.e. they support bursts of communication. This communication is achieved using the *input* and *output* primitives described in Chapter 1.
A channel specification has the format

```
CHAN OF type channel :
```

where

- `type` is the data type supported by the channel, and
- `channel` is the occam identifier of the channel

Example 2.15

```
CHAN OF BYTE Input :
```

declares a channel named `Input` which will support the successive transfer of single bytes.

Example 2.16

```
CHAN OF INT Error :
```

declares a channel named `Error` capable of communicating single integers.

More than one channel of the same type may be specified within the same declaration, the individual identifiers being separated by commas.

For example,

```
CHAN OF BOOL Flag, Semaphore :
```

declares two channels, `Flag` and `Semaphore`, of type `BOOL`.

Example 2.17

```
CHAN OF BYTE Chan :
PAR
  BYTE Data :
  SEQ
    Data := 100 (BYTE)
    Chan ! Data
  BYTE Value :
  SEQ
    Chan ? Value
```

depicts two sequential processes executing concurrently, but communicating with each other.
via the named channel *Chan* (Figure 2.1). Note that the channel declaration is placed before the **PAR** construction. This is necessary because the communicating processes within the **PAR** must both have access to the channel specified in the declaration. This is all tied up with the concept of scope, which will be discussed in the next section. Note also how the variables for the input and output processes - *Value* for the input process, *Data* for the output process - have been declared. These variables are *local* to their respective processes (see next section for a full explanation).

![Diagram](image)

*Figure 2.1 Two processes communicating via a channel*

**Example 2.18**

```plaintext
CHAN OF REAL32 Chan1, Chan2 :
PAR
    REAL32 Item, Result :
    SEQ
        Chan1 ! Item
        Chan2 ? Result
    REAL32 Data, Value :
    SEQ
        Chan1 ? Data
        Chan2 ! Value
```

depicts two sequential processes executing concurrently. Two-way communication between the two processes is achieved by specifying two channels. Local variables have again been specified for the communicating processes. Note the ordering of the input and output processes in both components of the **PAR** construction. This ordering avoids deadlock, as discussed in Chapter 1.
2.6 Scope

As discussed above, 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 their 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.

Example 2.19

\[
\begin{align*}
\text{INT } & \text{ Item : } \quad \text{-- declaration of Item} \\
\text{SEQ} & \quad \text{-- scope of Item} \\
\quad & \text{Item} := 1024 \quad \text{--} \\
\quad & \quad \text{--} \\
\quad & \quad \text{--} \\
\quad & \text{SEQ} \quad \text{--} \\
\quad & \quad \text{Item} := \text{Item} + 1024 \quad \text{--}
\end{align*}
\]

In this example, the variable \textit{Item} is available to the inner \texttt{SEQ} construction since this variable is still in scope.

Example 2.20

\[
\begin{align*}
\text{CHAN OF INT } & \text{ Comm : } \quad \text{-- declaration of Comm} \\
\text{PAR} & \quad \text{-- scope of Comm} \\
\quad & \text{INT } \text{ Value : } \quad \text{--} \\
\quad & \text{SEQ} \quad \text{--} \\
\quad & \quad \text{Value} := 512 \quad \text{--} \\
\quad & \quad \text{Comm} ! \text{ Value} \quad \text{--} \\
\quad & \text{INT } \text{ Result : } \quad \text{--} \\
\quad & \text{SEQ} \quad \text{--} \\
\quad & \quad \text{Comm} ? \text{ Result} \quad \text{--}
\end{align*}
\]

shows how the channel \textit{Comm}, being declared at the outermost level, has a scope which covers the whole program fragment. Thus the channel is common to both component sequential processes - as it must be for inter-process communication.
Example 2.21

CHAN OF INT Comm :
PAR
  INT Value : -- declaration of Value
  SEQ  Value := 512 --
  Comm ! Value --
  INT Result : -- declaration of Result
  SEQ  Comm ? Result --

illustrates the scope of the variables in the inner processes of the previous example.

A 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.

Example 2.22

CHAN OF BYTE Chan :
PAR
  BYTE Char : -- first declaration of Char
  SEQ  Char := 'J' --
  Chan ! Char --
  BYTE Char : -- second declaration of 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 the 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, 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.
Example 2.23

```oml
BYTE Item : -- declaration as BYTE
SEQ
  Item ::= 50 (BYTE) -- scope as BYTE
INT Item :
SEQ
  Item ::= 500 -- declaration as INT
  Chan2 ! Item -- scope as INT (BYTE declaration masked)
  Chan1 ! Item -- gives 500
-- 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 is 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 scope.

Any variable should be assigned a value - either by direct assignment or input - before being used. Occam does not itself initialise any variable before use. Chapter 7 explains how an occam procedure may be executed many times. It is important to realise that occam does not guarantee that the value of a local variable be kept from one execution of a 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 a free variable.

A free variable is available to PAR component processes so long as that variable is not assigned to or input by the component processes.
Exercises

1. Which of the following are valid occam identifiers?
   a) chan
   b) CHAN
   c) Raw.Data
   d) Raw Data
   e) Raw_Data

2. Comment on the validity of the following occam code fragments.
   a)  
      
      INT  Count, Limit :  
      REAL32  Factor :  
      BYTE  Limit :

   b)  
      
      BYTE  Small :  
      SEQ  
      Small := 55

   c)  
      
      BYTE  Small :  
      SEQ  
      Small := -1 (BYTE)

   d)  
      
      REAL32  Real :  
      SEQ  
      Real := 3.0 (REAL32)

   e)  
      
      INT  INDEX  
      SEQ  
      Index := 0
f)
CHAN OF BYTE Chan:
PAR
  BYTE Char:
  SEQ
    Char := 'J'
    Chan ! Char
  SEQ
    Chan ? Char

g)
CHAN OF BYTE Chan:
PAR
  BYTE Char:
  SEQ
    Char := 'G'
    Chan ! Char
  INT Char:
  SEQ
    Chan ? Char

h)
CHAN OF INT Chan1:
CHAN OF BYTE Chan2:
PAR
  INT Item:
  SEQ
    Chan1 ? Item
  BYTE Item:
  SEQ
    Item := 250 (BYTE)
    Chan2 ! Item
i)

CHAN OF INT Chan1:
CHAN OF BYTE Chan2:
INT Item:
PAR
  SEQ
    Chan1 ? Item
    BYTE Item:
    SEQ
      Chan2 ? Item

j)

INT Count, Index:
SEQ
  Index := Count

k)

INT Count:
SEQ
  Count := 0
  INT Index:
  SEQ
    Index := Count

l)

INT Index:
SEQ
  ...
  ...
  ...
  INT Count:
  SEQ
    ...
    ...
    ...
  Index := Count
Chapter 3

Operators

Constants and variables may be used in occam as operands which may be combined with operators to form simple expressions. The expressions so formed may be combined again with further constants, variables or expressions to form more complex expressions. There are a range of operators available in occam to allow the formation, for example, of arithmetic and boolean expressions. Most operators are dyadic, requiring two operands, but some are monadic, requiring one operand.

Other objects, such as array elements and tables (Chapter 4), are also classed as operands and these may also participate in the formation of expressions.

Expressions may be assigned to a variable, output to a channel or take part in a condition evaluation.

Occam enforces strong typing

- operands of any operator must be of the same data type

- the type of any variable being assigned to in an assignment statement must be the same as the type of the expression being assigned

The repertoire of operators available is

- arithmetic

- relational

- boolean

- bitwise

- shift
3.1 Arithmetic operators

Constants, variables and literals of integer or real data types may be combined with arithmetic operators to form arithmetic expressions, with the proviso that the operands must be of the same data type, either explicitly or via type conversion. (Type conversion is discussed later in this chapter.) The type of the resultant expression is the same as that of its operands.

The basic arithmetic operators are

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>REM</td>
<td>remainder (from division)</td>
</tr>
</tbody>
</table>

The subtraction operator may be used both as a monadic and dyadic operator.

For example,

MinInt := -32768

RoomLeft := BufferSize - NumberInBuffer

The resultant value from an integer division is truncated i.e. rounded toward zero.

For example,

11 / 6            - result is 1

(-11) / 6         - result is -1

This example introduces the concept of precedence brackets - the order of evaluation of operators in an occam expression must be specified with precedence brackets. Operator precedence is an important feature of occam and is described shortly.

An alternative denotation of REM is \. Both denotations may be used interchangeably.
For example,

\[
\begin{align*}
11 & \text{ REM } 6 & \text{ - result is 5} \\
(-11) & \text{ \textbackslash 6} & \text{ - result is -5}
\end{align*}
\]

Note that the result of a real arithmetic expression is rounded to the closest value which can be represented by the real type (see Section 3.1.4 of this chapter for a discussion of rounding).

**Example 3.1**

\[
\begin{align*}
\text{INT} & \quad \text{Index, Seconds, Minutes} : \\
\text{REAL32} & \quad \text{Average, Sum, Count, Double, Value} : \\
\text{SEQ} & \\
\text{Index} & \quad := \text{Index} + 1 \\
\text{Seconds} & \quad := 60 \times \text{Minutes} \\
\text{Count} & \quad := \text{Count} + 1.0 \,(\text{REAL32}) \\
\text{Average} & \quad := \text{Sum} / \text{Count} \\
\text{Double} & \quad := 2.0 \,(\text{REAL32}) \times \text{Value}
\end{align*}
\]

is an occam fragment containing examples of arithmetic operations, where the variables *Index, Minutes, Count, Sum* and *Value* are assumed to have been already assigned to by an input or an assignment. Note the use of typed literals in the expressions. As explained in Chapter 2, except for type **INT** integer, single character, character string and boolean literals, the type of any literal must be explicitly specified within brackets immediately following the literal value.

**Example 3.2**

\[
\begin{align*}
\text{REAL32} & \quad \text{Data} : \\
\text{SEQ} & \\
\text{InChan} & \quad \? \text{ Data} \\
\text{OutChan} & \quad \! \text{ Data} / 2.0 \,(\text{REAL32})
\end{align*}
\]

is an example of an arithmetic expression being used in an output process, where the protocol of the channels, *InChan* and *OutChan*, is specified to have type **REAL32**.

An arithmetic operation is considered to be invalid if the result produced is out of range of the given type; for example, arithmetic overflow or underflow, and division by zero.

A few specialised arithmetic operators are available.
3.1.1 Modulo arithmetic operators

Modulo arithmetic produces values which neither overflow nor underflow. Instead, values wrap round cyclically, either from the most positive value to the most negative or from the most negative value to the most positive, depending on the particular values of the operands and operator. For example, modulo arithmetic values, using INT16 types, cycle round as follows

\[ \ldots -1, 0, 1, \ldots 32766, 32767, -32768, -32767, \ldots -1, 0, 1, \ldots \]

There are three modulo arithmetic operators available. These are

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>modulo addition</td>
</tr>
<tr>
<td>MINUS</td>
<td>modulo subtraction</td>
</tr>
<tr>
<td>TIMES</td>
<td>modulo multiplication</td>
</tr>
</tbody>
</table>

The respective operands must be of the same integer type.

**Example 3.3**

```plaintext
VAL INT16 MaxInt IS 32767 :
INT16 Result :
SEQ
   Result := MaxInt PLUS 1 (INT16) -- value is -32768
   Result := MaxInt PLUS 2 (INT16) -- value is -32767
```

**Example 3.4**

```plaintext
VAL INT16 MinInt IS -32768 :
INT16 Result :
SEQ
   Result := MinInt MINUS 1 (INT16) -- value is 32767
   Result := MinInt MINUS 2 (INT16) -- value is 32766
```

Such arithmetic is most useful in connection with timers (Chapter 9).
3.1.2 MOSTPOS and MOSTNEG

The operators MOSTPOS and MOSTNEG, when applied to an integer type, produce the most positive and the most negative values respectively of that type.

\[
\begin{align*}
\text{MOSTPOS} & \quad \text{type} \\
\text{MOSTNEG} & \quad \text{type}
\end{align*}
\]

For example,

\[
\text{MOSTPOS INT32}
\]

produces a value of 2147483647.

3.1.3 Operator precedence

In contrast with other languages, occam does not have a hierarchy of operator precedence. In Pascal, for example, the arithmetic expression \(2 + 3 * 4\) unambiguously produces the value 14 and not the value 20 since, in Pascal, the * operator takes precedence over (is evaluated before) the + operator. Such an expression is illegal in occam.

The reason for this is that operators in occam have equal precedence - there is no preferred order for evaluating terms containing operators. The required order of evaluation of multi-operator expressions must be explicitly specified with the use of brackets.

For example,

\[
(A + B) + C \quad \text{is legal}
\]

\[
A + (B + C) \quad \text{is legal}
\]

but \(A + B + C\) is illegal.
and

\[(3 \times 2) + 6\] is legal, result = 12

\[3 \times (2 + 6)\] is legal, result = 24

but \[3 \times 2 + 6\] is illegal

Complex expressions may contain nested bracketed terms to indicate the order of evaluation.

Example 3.5

Expression := \(((\text{R} + \text{S}) / 2) + \text{T}\)

Discriminant := \((\text{B} \times \text{B}) - (4.0 \times \text{REAL32}) \times (\text{A} \times \text{C}))\)

DifferenceOfSquares := \((\text{X} \times \text{X}) - (\text{Y} \times \text{Y})\)

shows the use of precedence brackets in valid assignment statements.

These remarks concerning operator precedence also apply to the other operators, and not just the arithmetic operators.

3.1.4 Data conversion

Occam allows an operand declared as one primitive type to be expressed as certain other primitive types; for example a real type to be expressed as an integer and vice versa, an integer type to be expressed as a byte and vice versa. This mechanism is known as data conversion. Data conversion allows, for example, the writing of a mixed data type expression i.e. an expression with operands of different declared types. Any data conversion must be specified explicitly by writing the target type ahead of the operand due to be converted.

\[
\text{type operand}
\]

where type is the target type and is one of the primitive data types. Data conversion is classed as an operation. Consequently, any conversion appearing in an expression must be enclosed within precedence brackets.
Example 3.6

INT Number:
SEQ
 InChan ? Number
 OutChan ! BYTE (Number + (INT '0'))

where the channels, InChan and OutChan, are specified as type INT and BYTE, respectively. This ocass fragment shows an integer number (assumed to lie in the range 0 - 9) being converted to the corresponding ASCII character for output. The conversion proceeds by adding the integer value of character '0' (the base value) to the number.† This produces the corresponding ASCII value of the number. Note the use of the type conversions. The integer value of character '0' is obtained by the INT conversion, and the final value converted to type BYTE.

Conversion between the real types is allowed (but see the discussion on ROUND and TRUNC in this section), as is the conversion between the integer types, with the proviso that any conversion is only valid if the result is in the range of values supported by the target type. For example, an INT32 variable with a value 32768 cannot be converted to type INT16 as the value is too large to be represented in INT16.

Example 3.7

REAL64 Double1, Double2:
REAL32 Single:
SEQ
 Single := 3.144 (REAL32)
 Double1 := 6.6666666 (REAL64)
 Double2 := (REAL64 Single) * Double1

shows the conversion of a REAL32 variable to REAL64.

Data conversion between integer (or byte) and boolean types is possible as long as the conversion is in range. (Boolean values are represented internally by the byte values, 0 and 1.)

† In the ASCII representation of characters, the character '0' has a numeric value of 48, '1' has a value of 49, . . . and '9' has a value of 57.
For example,

BOOL 1 converts to TRUE.

INT FALSE converts to 0.

BOOL 2 is, however, invalid.

The conversion between real and byte types, and real and boolean types is not specifically allowed, but may be staged via an integer conversion. For example,

Real := REAL32 ROUND (INT Byte)

demonstrates the two-stage process of converting a byte type to a real.

Data conversion between integer and real types causes complications because of the difference in internal representation of these two types. (Integer types are represented in two’s complement form - a single sign bit plus 15, 31, or 63 bits for the integer value - whilst real types are represented by the ANSI/IEEE floating point standard - see Chapter 2.) It is necessary to state in these cases whether the conversion is to incorporate a rounding or truncation of the result. This is achieved by inserting the occam reserved word ROUND or TRUNC between the target type and operand.

```
type ROUND operand

type TRUNC operand
```

The effect of ROUND is to round the operand value to the nearest number. When the operand value is equidistant from the two nearest numbers, rounding is away from zero. The effect of TRUNC is to round the operand value towards zero.
For example, the effect of ROUND is shown by

```
INT ROUND 3.4 (REAL32) -- result is 3
INT ROUND -3.4 (REAL32) -- result is -3
INT ROUND 3.8 (REAL32) -- result is 4
INT ROUND -3.8 (REAL32) -- result is -4
INT ROUND 3.5 (REAL32) -- result is 4
INT ROUND -3.5 (REAL32) -- result is -4
```

and the effect of TRUNC is shown by

```
INT TRUNC 3.4 (REAL32) -- result is 3
INT TRUNC -3.4 (REAL32) -- result is -3
INT TRUNC 3.8 (REAL32) -- result is 3
INT TRUNC -3.8 (REAL32) -- result is -3
INT TRUNC 3.5 (REAL32) -- result is 3
INT TRUNC -3.5 (REAL32) -- result is -3
```

The same observations apply to the conversion of integers to reals.

For example,

```
REAL32 TRUNC 4  -- result is 4.0
REAL32 ROUND 4  -- result is 4.0
```

The effect of ROUND and TRUNC on the same integer does not necessarily produce the same real value if the integer value cannot be exactly represented in the real representation.

The conversion of a value of type REAL64 to type REAL32 is allowed provided that the value is in the range of the REAL32 type. Moreover, the conversion must specify a truncation or rounding.
For example,

SmallReal := REAL32 ROUND BigReal

shows such a conversion.

### 3.2 Relational operators

Relational operators allow operands to be compared; for example, to test whether an operand is less than or equal to another operand. The result of the comparison is a boolean value, i.e. **TRUE** or **FALSE**. The expression formed by a relational operator and its operands is a boolean expression.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

Operands, which may be constants, variables or expressions, used with the the equality (=) and inequality (<> operators may be of any primitive data type, but the operands used with the remaining relational operators must be of **byte**, **integer** or **real** types only. Both operands must be the same data type.

Characters and strings may be compared with relational operators, according to their ASCII ordering.
For example, if $Comp$ is declared to be of type $\text{BOOL}$, then

$\text{Comp} := (7 = 14)$ evaluates to $\text{FALSE}$

$\text{Comp} := (7 < 9)$ evaluates to $\text{TRUE}$

$\text{Comp} := (6.9 \text{ (REAL32)} \geq 1.3 \text{ (REAL32)})$ evaluates to $\text{TRUE}$

$\text{Comp} := ('Z' > 'A')$ evaluates to $\text{FALSE}$

$\text{Comp} := ('\text{Hello}' > 'Greetings')$ evaluates to $\text{FALSE}$

(Note that the brackets around each of the boolean expressions in the above examples are not really required but their inclusion perhaps aids readability.)

**Example 3.8**

$\text{Count} = \text{Limit}$

is a boolean expression which evaluates to the value $\text{TRUE}$ if the value of $\text{Count}$ is equal to the value of $\text{Limit}$, and to $\text{FALSE}$ otherwise, and

$\text{Error} <= \text{Epsilon}$

evaluates to $\text{TRUE}$ if the value of $\text{Error}$ is less than or equal to the value of $\text{Epsilon}$, and to $\text{FALSE}$ otherwise.

**Example 3.9**

```
\text{BOOL Flag :}
\text{BYTE Char :}
\text{SEQ}
\quad \text{InChan ? Char}
\quad \text{Flag := (Char > 'A')}
```

where the protocol of the channel $\text{InChan}$ is declared to be of type $\text{BYTE}$. In this occam fragment, $\text{Flag}$ is assigned the value $\text{TRUE}$ if $\text{Char}$ has a value greater than the ASCII value of 'A'.

3.3 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**.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>boolean and</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>boolean or</td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>boolean not</td>
</tr>
</tbody>
</table>

For example,

```
(Count < Limit) OR (Error > Epsilon)
```

is the logical or combination of the boolean expressions, **Count < Limit** and **Error > Epsilon**. The value of the resultant expression depends on the current value of the numeric variables being compared.

**Example 3.10**

```
(Value >= Minimum) AND (Value <= Maximum)
```

is a combination of boolean expressions representing a test which evaluates to the value **TRUE** if a value falls within the range defined by **Minimum** and **Maximum**.

Boolean expressions are typically used in control expressions (see Chapter 5), examples of which are

**WHILE** Active

where **Active** is a boolean variable, and

**WHILE NOT** (Running OR Waiting)

where **Running** and **Waiting** are boolean variables.
Example 3.11

BOOL Flag:
BYTE Char:
SEQ
   InChan ? Char
   Flag := (Char >= '0') AND (Char <= '9')

sets the value of Flag to TRUE if Char is an ASCII digit.

Example 3.12

BOOL Level1, Level2:
SEQ
   PAR
      InPin1 ? Level1
      InPin2 ? Level2
      OutPin ! NOT (Level1 AND Level2)

represents a simplistic software simulation of a NAND logic gate, the inputs proceeding in parallel, followed by the production of the output. (A NAND operation is defined as being the negation of an AND operation.) It is assumed that the channels, InPin1, InPin2, and OutPin, have been specified as CHAN OF BOOL.

The requirement of precedence brackets is relaxed for boolean operators - brackets may be omitted between expressions containing multiple boolean operators.

For example,

    Bool1 OR Bool2 OR Bool3

is a valid combination of boolean operands.
3.4 Bitwise operators

These operators allow various operations, for example, setting and masking, 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 value being operated on; the second operand represents the bit pattern which operates on this value, and itself may be a constant, variable or expression of integer type.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITAND</td>
<td>bitwise and</td>
</tr>
<tr>
<td>BITOR</td>
<td>bitwise or</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>BITNOT</td>
<td>bitwise not</td>
</tr>
</tbody>
</table>

The operands must be of the same integer type.
Alternative denotations for bitwise and, or and not are \( \land \), \( \lor \), and \( \sim \) respectively.

Example 3.13

\[ \text{Int} \land \#7F \]

shows the bitwise anding of a variable \( \text{Int} \) with the constant \#7F. The bit pattern of \#7F is 0111 1111, and so this operation will set to zero all but the 7 low-order bits of \( \text{Int} \).
Example 3.14

\[ \text{Int} \lor \#1F \]

shows the bitwise oring of a variable \textit{Int} with the constant \#1F. The bit pattern of \#1F is 0001 1111, and so this operation will set to one the 5 low-order bits of \textit{Int}.

Example 3.15

\[ \text{UpperCaseChar} := \text{LowerCaseChar} \land \#\text{DF} \]

converts a lower-case ASCII character to its upper-case equivalent (both assumed to be represented by integer variables) by anding with the bit pattern 1101 1111 (#DF). (Lower- and upper-case ASCII characters differ by the setting of the fifth bit.)

Example 3.16

\[ \text{BITNOT (Int1 BITOR Int2)} \]

computes the NOR value for variables \textit{Int1} and \textit{Int2} - NOR being defined as the negation of OR operation.

Example 3.17

\[ \text{StatusRegister} := \text{StatusRegister} >\!< \#4000 \]

uses a mask of \#4000 to invert the fourteenth bit of \textit{StatusRegister}.

Example 3.18

\[
\begin{align*}
\text{INT} & \quad \text{Pulse1, Pulse2 :} \\
\text{SEQ} & \\
\text{PAR} & \\
\text{InPin1} & ? \text{ Pulse1} \\
\text{InPin2} & ? \text{ Pulse2} \\
\text{OutPin} & ! \text{ BITNOT (Pulse1 BITAND Pulse2)}
\end{align*}
\]

represents a variation on the simulation of a logic gate - this time using integer types, and channels specified as \textit{CHAN OF INT}, together with bitwise operators. Other, more complex, digital circuits may be simulated quite easily in occam using the simple logic gates as building blocks. However, deadlock problems can easily arise in such circuits. One solution is to introduce a propagation delay in to the gate model [Welch].
3.5 Shift operators

The bits comprising the value of an integer constant, variable or expression may be shifted left or right a specified number of bit positions (Figure 3.1). 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 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 << 2

shifts the value of the variable Int left by two bit positions; this is equivalent to multiplying by 4, and

Int >> 8

shifts the value of Int right by eight bit positions; this is equivalent to dividing by 256.

**Example 3.19**

```
INT16   Packet, TerminalNumber :
BYTE    Char :
SEQ
   InChar  ? Char
   InInt16 ? TerminalNumber
   Packet := (TerminalNumber << 8) \lor (INT16 Char)
```

forms a composite Packet, comprising TerminalNumber in the top 8 bits, and Char in the bottom 8 bits.
Exercises

1. Comment on the validity of the following occam expressions and fragment.
   a) 100 - 5 * 5
   b) (100 - (5 * 5) + 20)
   c) 100 - (5 * 5) > Limit
   d) 100 - (Count > Limit)
   e)  

   \[
   \text{INT16 Result :}
   \]
   \[
   \text{SEQ}
   \]
   \[
   \text{Result := 32767 (INT16) + 1 (INT16)}
   \]
   \[
   \text{Result := 33 (INT16) * 10}
   \]
   \[
   \text{Result := 330 (INT16) * 100 (INT16)}
   \]

2. Write down the resultant value for the following occam expressions.
   a) \text{NOT TRUE}
   b) \text{TRUE OR BooleanValue}
   c) \text{TRUE AND BooleanValue}
   where BooleanValue has a value \text{TRUE or FALSE}.
   What happens if \text{TRUE} is replaced by \text{FALSE} in the above expressions?

3. Write down the resultant values for the following data conversions.
   a) \text{BYTE TRUE}
   b) \text{BYTE FALSE}
   c) \text{BOOL 0}
   d) \text{INT ROUND 3.142 (REAL32)}
   e) \text{INT TRUNC 3.142 (REAL32)}
   f) \text{REAL32 TRUNC 3}
   g) \text{REAL32 ROUND 3}

4. Write an occam program which will set bit 4 of an integer variable.

5. Write a set of occam programs which will simulate various logic gates, for example, AND, OR and NOR.
6. Write an occam program which will give the number of hundreds, tens and units in the three-digit integer, 123.

7. Write an occam program which will simulate a XOR gate, composed of AND and OR logic gates.
(Hint: \( A \ XOR \ B = ((\ NOT \ A) \ AND \ B) \ OR \ (A \ AND \ (\ NOT \ B)) \ ))
Chapter 4

Arrays

Occam 2 has only one *structured* type defined - this is the multi-dimensional array. Arrays provide the ability to address an ordered sequence of objects of the same type via a common name. These objects comprise the *components* of the array. Arrays may be formed from any of the primitive data types, or the channel or timer types. (Channel and timer types are described in Chapters 8 and 9 respectively.) They must be named in a declaration statement before use - they are bound by the usual scoping rules to the process immediately following their specification.

4.1 One-dimensional arrays

The format of a 1-dimensional array declaration is

```
[size] type array :
```

where

- *size* is the number of components in the array
- *type* is the type of the array components, and
- *array* is the array identifier

The *size* must be a non-zero value of type INT, enclosed within square brackets. It may be represented by a literal, a previously declared constant or an expression which evaluates to a constant (but *not* by a variable). The type may be one of the primitive data types, or the channel or timer types. If the type is one of the primitive data types, then operations may be performed on single components, sections of the array or on the whole array itself. Only single components of channel or timer arrays may be referenced. The mechanism for accessing the various parts of an array - single components, sections or the whole array - will be described in this chapter.
Example 4.1

[50] REAL32 Readings :

declares an array Readings having 50 components, each component being of type REAL32.

In the example above, Readings is regarded as an array of type [50] REAL32. This is quite general in occam; the array type is determined by both its size and the type of its components. For occam arrays to have the same type, they must have the same size and components of the same type.

Example 4.2

[10] BOOL Flags :

is the declaration for an array Flags of type [10] BOOL.

Example 4.3

VAL Panel IS 25 :

[Panel] CHAN OF BYTE Switches :

declares an array of 25 channels named Switches, each channel being capable of communicating single byte values. Such a channel specification is just the array extension of a simple channel protocol.

Example 4.4

[26] BYTE Alphabet :

[5] BYTE Vowels :


Arrays of the same type may be specified in the same statement. For example,

[16] BYTE WhitePieces, BlackPieces :

declares two arrays, each of type [16] BYTE.
Example 4.5

VAL Size IS 16:
[Size] TIMER Clock, Watch:

specifies two timer arrays, Clock and Watch, each with 16 components. (Timers are special
channels connected to the transputer internal clock - see Chapter 9.)

A whole array may be input, output or assigned to. In such operations, the whole array is
accessed by referencing the array identifier. In any assignment or input, the array type must be
strictly observed.

The following example illustrates how an array may be transferred from one parallel process to
another. The example also shows how a simple channel protocol may be extended to cope
with the transfer of arrays of values (more of this in Chapter 8).

Example 4.6

CHAN OF [10] INT Chan: -- extension of simple channel
-- for the transmission of an array

PAR
[10] INT Array1:
SEQ
.
  -- assign values to Array1
  Chan ! Array1
[10] INT Array2:
SEQ
  Chan ? Array2
Note the specification of the arrays and the channel. Both array variables, Array1 and Array2, and the channel, Chan, must be of type [10] INT, i.e. the array type is strictly observed. An alternative to extending the simple channel protocol is the transfer of single array components within a loop. The loop is executed the required number of times in order to transfer the whole array. This approach is further explored in Chapter 6.

Note the difference between

\[ [10] \text{CHAN OF INT Chan} : \]

and

\[ \text{CHAN OF [10] INT Chan} : \]

The former is an array of ten channels each capable of transmitting a single integer; whilst the latter is a single channel capable of transmitting an array of ten integers (Figure 4.1).

![Diagram](image)

**Figure 4.1** Difference between an array of channels and a channel of arrays
4.2 Multi-dimensional arrays

Multi-dimensional arrays are specified in an analogous manner - the size of each dimension being enclosed within square brackets. For example, a 2-dimensional array is specified as

\[
[size_1] [size_2] type array :
\]

where \textit{size 1} and \textit{size 2} are the number of components in each dimension. For example,

\[ [17] [15] \text{REAL32} \ Grid : \]

represents a two-dimensional array \textit{Grid} with 17 components, each of which is a 15-element one-dimensional array.

The array \textit{Grid} may be regarded as an array of an array of type [15] \text{REAL32}. In general, multi-dimensional arrays may be considered as arrays of array types.

The format for the declaration of a general multi-dimensional array is

\[
[size_1] [size_2] \ldots [size_n] type \ array :
\]

where, again, \textit{size 1}, \textit{size 2}, \ldots \textit{size n} are the number of components in each dimension.

Example 4.7

\[ [20] [80] \text{BYTE} \ DisplayScreen : \]

\[ [8] [8] [8] \text{REAL32} \ Cube : \]

\[ [5] [4] \text{REAL64} \ Matrix : \]

are examples of multi-dimensional array declarations.

Similar to one-dimensional arrays, whole multi-dimensional arrays may be input, output or assigned to. Again, in any assignment or input, the \textit{array type} must be strictly observed.
For example,

```ocahom
CHAN OF [5] [4] REAL64 Comm :
PAR
[5] [4] REAL64 AMatrix :
SEQ
  -- assign values to AMatrix
  Comm ! AMatrix
[5] [4] REAL64 BMatrix :
SEQ
  Comm ? BMatrix
```

demonstrates the transfer of a two-dimensional array between two parallel processes.

### 4.3 Array subscripts

The position of an object within an array i.e. a component of an array is given by the array index or *subscript*. The subscript may be a constant (literal or occam constant) or an expression involving constants and/or variables, but *must* always be of data type INT. The initial subscript of an array is 0. A component of an array is referenced via

```
array [subscript]
```

The *subscript* must always lie within the bounds of the array i.e. equal to or greater than 0 and less than *size*.

For example, the declaration

```
[50] REAL32 Readings :
```

would have 50 components as follows

```
Readings [0]
Readings [1]
Readings [2]
  .
  .
  .
Readings [49]
```
The individual components of an array may be used in exactly the same way as an unstructured occam variable - for example, array components may be used in assignment, input and output statements, and may be used as operands in expressions.

Example 4.8


Readings [I] := 12.0 (REAL32)

Sum := Readings [0] + Readings [1]

Readings [I + J] := Value

Total := Total + Readings [I + J]

Display ! Readings [15]

where the subscripts \( I \) and \( J \) are of type \( \text{INT} \). These are all legal examples involving the use of array components (provided the expressions involving \( I \) and \( J \) evaluate to valid subscripts for the array \( \text{Readings} \)).

Example 4.9

\[
\begin{align*}
[3] \text{REAL32 } & \quad X, Y, Z : \\
\text{SEQ} \\
\end{align*}
\]

is an occam program fragment which computes the vector cross product of arrays \( X \) and \( Y \).

Components of channel and timer arrays may only be used in the same way as their unstructured equivalents - as part of an input or output process. Only single components of these types of arrays may be referenced. References to sections of an array or to the whole array are invalid for channel and timer arrays.
Example 4.10

[10] CHAN OF INT Sensors:
INT Voltage, Temperature:
PAR
   Sensor [0] ? Voltage
   .
   .

shows an occam fragment which has declared an array of ten channels, Sensor. These channels may only be accessed individually, as shown.

Components of multi-dimensional arrays are accessed in an analogous manner to their one-dimensional counterparts. In occam, multi-dimensional arrays are stored in row-major order. This means that, for example, with a two-dimensional array, the second subscript will vary more rapidly than the first, and so on, for each extra dimension.

For example, the components of an array declared as

[17] [15] REAL32 Grid:

are stored in memory in the following order

   Grid [0] [0]
   Grid [0] [1]
   Grid [0] [2]
   .
   .
   .
   Grid [0] [14]
   Grid [0] [15]
   Grid [1] [0]
   Grid [1] [1]
   Grid [1] [2]
   .
   .
   .
   Grid [17] [14]
   Grid [17] [15]
Example 4.11

\[ \text{Grid}[7][7] := 0.0 \ (\text{REAL32}) \]

\[ \text{Grid}[I][J] := \text{REAL32 \ ROUND} \ (I + J) \]

\[ \text{Grid}[I + J][I + J] := 0.0 \ (\text{REAL32}) \]

where the subscripts \( I \) and \( J \) are of type \text{INT}. These are all legal examples involving the use of array components (provided the expressions involving \( I \) and \( J \) evaluate to valid subscripts for the array \text{Grid}).

Care must be taken with the use of arrays in \text{PAR} constructions. The rules for their use say that an array may be used in more than one component process provided that the values of the array subscripts are constants and provided that the parts of the array accessed in each component process do not overlap.

When arrays are referenced within a loop, the accesses may be made more efficient with the use of segments and abbreviations (Chapter 7).

### 4.4 Segments

Consecutive components of an array comprising a \textit{segment} or a slice of the array may be referenced as a single unit.

The format for an array segment is

\[
[ \text{array FROM first FOR count} ]
\]

where

- \text{array} is the array name,

- \text{first} is the subscript of the first component of the segment, and

- \text{count} is the number of components

Both \text{first} and \text{count} must be of data type \text{INT}. The value of \text{count} must be non-negative and must be so constrained that the components referenced lie within the bounds of the array. A segment is just another array comprising the referenced components of the original array.
For example,

[Readings FROM 10 FOR 5]

picks out the eleventh, twelfth, ... and fifteenth components of the array Readings, i.e. Readings [10], Readings [11], ... Readings [14].

Segments may take part in input, output and assignment processes. In an assignment, the array type of the segment must be observed i.e. the data type and number of components of the left- and right-hand sides of the assignment must be the same. Like any other array, a segment may be subscripted.

For example,

[Readings FROM 10 FOR 5] [1]


Example 4.12

[100] INT Array1, Array2 :
SEQ
  . -- assign values to the first fifty components of Array1
  .
  [Array2 FROM 0 FOR 50] := [Array1 FROM 0 FOR 50]

copies the first fifty components of Array1 to Array2.

Example 4.13

[100] REAL32 SensorReadings :
SEQ
  PAR
    Sensor [0] ? [SensorReadings FROM 0 FOR 50]
  .
  . -- process sensor readings
  .

inputs the two halves of the array SensorReadings as separate inputs.
Example 4.14

[30] INT WholeRange:
[10] INT MiddleRange:
SEQ
  MiddleRange := [WholeRange FROM 10 FOR 10]
  -- modify contents of MiddleRange
  [WholeRange FROM 10 FOR 10] := MiddleRange

copies part of the array WholeRange into the array MiddleRange and back again after some processing.

Example 4.15

CHAN OF [1000] REAL32 Chan1:
CHAN OF [1000] REAL32 Chan2:
[2000] REAL32 Digitising:
SEQ
  -- assign values to Digitising
  PAR
    Chan1 ! [Digitising FROM 0 FOR 1000]
    Chan2 ! [Digitising FROM 1000 FOR 1000]

outputs the two halves of the array Digitising on two separate channels.

4.5 Strings

In occam, a string of characters is internally represented by an array of bytes, each byte being the ASCII representation of a character of the string. As such, string constants and variables may be manipulated in the fashion of arrays.

For example,

[10] BYTE Message:
SEQ
  Message := "Greetings!"
Here

Message [0] equals 'G'
Message [1] equals 'r'
.
.
Message [9] equals '!'  
(Note that the number of characters in the string must match the declared size of the array.)

Example 4.16

VAL Message IS "Hello":

is equivalent to

VAL [5] BYTE Message IS "Hello":

and may be accessed as such.
This point will be covered in more detail in Chapter 7 on abbreviations.

4.6 Size of arrays

The number of components of an array may be determined at run-time with the SIZE operator. This is particularly useful with procedures (Chapter 7) for writing general purpose array handling code; the size of the actual array is not a constraining factor - it can be determined at run-time with the SIZE operator.

The format is

SIZE array

where array is an array identifier. The result is a value of type INT which is the number of components of the array.

Example 4.17

The statement

NoOfComponents := SIZE Signal

would assign a value of 32 to NoOfComponents, if Signal is an array which has been specified as having 32 components. NoOfComponents must be specified as being of type INT.
The SIZE operator may also be used with multi-dimensional arrays.

Example 4.18

```
INT  Num1, Num2 :
[5] [4] REAL64 Matrix :
SEQ
    Num1 := SIZE Matrix [0]
    Num2 := SIZE Matrix
```

assigns a value of 4 to Num1 and a value of 5 to Num2. The same result for Num1 would be realised with statements of the form

```
Num1 := SIZE Matrix [1]
```
or

```
Num1 := SIZE Matrix [2]
```
and so on, up to

```
Num1 := SIZE Matrix [4]
```

4.7 Tables

A table allows an array to be generated from would-be components. It provides a convenient notation for initialising arrays, either by assignment or by abbreviation (Chapter 7). The individual components, each of which may comprise any occam expression, must all be of the same data type. Within a table statement, the components are separated by commas and enclosed with square brackets.

```
[component 1, ... component n]
```

where component 1, ... component n are the individual components of the table.
For example,

\textbf{VAL} \hspace{5pt} [10] \textbf{INT} \hspace{5pt} \textbf{Primes} \hspace{5pt} \textbf{IS} \hspace{5pt} [2, 3, 5, 7, 11, 13, 17, 19, 23, 29] :

shows the use of a table in an abbreviation to initialise an integer array constant, \textit{Primes}. Alternatively, initialisation may be performed by assignment as follows

\begin{verbatim}
[10] INT Primes :
SEQ
Primes := [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
\end{verbatim}

where \textit{Primes} is now an integer array variable.

Other data types besides integers may be used in a table. For example,

\textbf{VAL} \hspace{5pt} [10] \textbf{BYTE} \hspace{5pt} \textbf{CharDigits} \hspace{5pt} \textbf{IS} \hspace{5pt} ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'] :

depicts the initialisation with the ten ASCII digits of a byte array constant, \textit{CharDigits}.

\textbf{Example 4.19}

\begin{verbatim}
[50] INT Array :
SEQ
[Array FROM 0 FOR 10] := [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
\end{verbatim}

initialises the first ten components of \textit{Array} by assignment.

Multi-dimensional tables are allowed. The number of sub-components in each component of the table must be the same.

\textbf{Example 4.20}

\begin{verbatim}
[5] [2] BYTE Vowels :
SEQ
Vowels := ['a', 'A'], ['e', 'E'], ['i', 'I'], ['o', 'O'], ['u', 'U']
\end{verbatim}

It has been observed that multi-dimensional arrays may be considered as arrays of array types. For example, the array in the example above may be treated as a one-dimensional array of type \[2] \textbf{BYTE} or a two-dimensional array of type \textbf{BYTE}. In the same way, the components of multi-dimensional arrays may be accessed as arrays of array types.
For example, consider again the array, *Vowels*, in the example above. This array is stored in the order

- Vowels [0] [0] -- value 'a'
- Vowels [0] [1] -- value 'A'
- Vowels [1] [0] -- value 'e'
- Vowels [1] [1] -- value 'E'
- Vowels [2] [0] -- value 'i'
- Vowels [2] [1] -- value 'I'
- Vowels [3] [0] -- value 'o'
- Vowels [3] [1] -- value 'O'
- Vowels [4] [0] -- value 'u'
- Vowels [4] [1] -- value 'U'

Components may be referenced treating the array as one-dimensional of type [2] **BYTE** - for example, *Vowels [1]* would reference ['e', 'E'] - or treating the array as two-dimensional of type **BYTE** - for example, *Vowels [3,1]* would reference 'O'. Thus, *Vowels [1]* picks out all those components with initial index of 1.
Exercises

1. Devise suitable array specifications for
   a) the colours of traffic lights (the initial letter of each colour)
   b) a chessboard (8 rows by 8 columns), with each square capable of holding an integer
   c) the print positions of printer paper (66 rows by 132 columns), with each position capable of holding a character
   d) a Scrabble board (15 rows by 15 columns), with each square capable of holding a character
   e) the names of the months (the first three characters of each name)

2. Distinguish between

   [8] CHAN OF REAL64 Chan :

   and

   CHAN OF [8] REAL64 Chan :

3. Write an occam fragment for transferring a [25] REAL32 array between two component processes of a PAR process.

4. a) Write down the component of the middle square of a Scrabble board (15 rows by 15 columns).
   b) The first word placed on a Scrabble board must lie horizontally across the middle square. If the first word is "ACE", what are the components of the Scrabble board which will hold this word?

5. Consider the following occam fragment.

   ```occam
   INT Num1, Num2, Num3 :
   [15] [10] [5] INT64 ThreeD :
   SEQ
   Num1 := SIZE ThreeD [0] [0]
   Num2 := SIZE ThreeD [0]
   Num3 := SIZE ThreeD
   ```

   What are the values assigned to Num1, Num2 and Num3?
6. Comment on the validity of the following tables
   a) 
      \[256 \text{(.INT16)}, 300\]
   b) 
      ['a', ['b', 'c']]
   c) 
      ['Hello', "Greetings"]
Chapter 5

Constructions

Higher-level processes in occam are built from other (simpler) processes using constructions. A hierarchy of nested processes, each built with a construction, may be formed. Earlier the SEQ, PAR and ALT constructions were introduced. Apart from SEQ, PAR and ALT, the other occam constructions provide for

- condition
- selection
- repetition

This chapter deals with the ALT construction in more detail, and also describes the constructions for dealing with condition, selection and repetition.

5.1 Alternation construction

The alternation construction selects a process for execution from a number of alternatives. In the simplest case, each alternative process is guarded by an input process. The selection is made according to which of the guard inputs is ready first. The alternation construction terminates successfully after execution of the selected process. Other guards comprise the combination of a boolean expression with an input, and the combination of a boolean expression with a SKIP process. These other guards will be dealt with shortly.

The operation of the ALT construction is non-deterministic - if more than one input is available at the same time, an arbitrary one is chosen and its associated process executed. The occam definition does not specify which input will be chosen in such circumstances and so the actual input chosen may well vary from implementation to implementation. Some implementations may give rise to the indefinite postponement of alternative processes if more than one input is available very quickly and at the same time. The same alternative process may be repeatedly chosen as the one for execution.
The format of the basic ALT construction is

```
ALT
  input guard 1
  process 1
  .
  .
  .
  input guard n
  process n
```

Each input guard must be indented two spaces from the ALT reserved word. Each associated process is further indented two spaces.

![Diagram](image)

**Figure 5.1** A simple multiplexor process

**Example 5.1**

```
ALT
  Input1 ? Data
  Output ! Data
  Input2 ? Data
  Output ! Data
  Input3 ? Data
  Output ! Data
  Input4 ? Data
  Output ! Data
```
represents an occam simulation of a 4-to-1 multiplexor - data is accepted from any one of four input channels, whichever is available and output on a single channel (Figure 5.1).

Example 5.2

```
VAL Terminator IS '?:
BYTE Char, Any :
ALT
   InChan ? Char
   OutChan ! Char
   Stop ? Any
   OutChan ! Terminator
```

represents a simple buffer which is stopped by a signal (Any) on the Stop channel (Figure 5.2). On receipt of a stop signal, the process outputs a terminator character. In the event of inputs being simultaneously available on the InChan and Stop channels, which of the alternatives that is chosen is undefined. A solution to this difficulty is to be found with the priority ALT (Chapter 9).

![Diagram](image)

**Figure 5.2 A simple buffer process with a stop channel**
5.1.1 Boolean guards

An extension of the simple input guard is the combination of a boolean expression with the input. In this case a process is selected for execution if the boolean expression is true and the input is ready. The format of this type of guard is

\[ \text{boolean expression} \ & \ 	ext{input} \]

Example 5.3

ALT

(State = Receiving) & Chan ? Signal

-- perform processing in receiving state

(State = Transmitting) & Chan ? Signal

-- perform processing in transmitting state

(State = Idle) & Chan ? Signal

-- perform processing in idle state

depicts an occam fragment of an ALT construction with boolean guards. Which alternative is chosen depends now not only on the availability of an input, but also on the truth value of the associated boolean expression. In the example above, input may be available on the channel Chan but the alternate chosen will now also be governed by the value of the variable State.
5.1.2 SKIP guards

This variation of guard comprises a combination of boolean expression with the SKIP process. The format is

\[
\text{boolean & SKIP}
\]

This type of guard may be considered as a catch-all for the input part of a boolean guard. If the boolean expression is true, the SKIP acts as a ready input.

For example,

\[
\text{ALT}
\]

\[
\text{Flag & Chan1 ? Message}
\]

\[
\quad . \quad \text{-- perform processing}
\]

\[
\text{Flag & Chan2 ? Message}
\]

\[
\quad . \quad \text{-- perform processing}
\]

\[
\text{(NOT Flag) & SKIP}
\]

\[
\quad . \quad \text{-- perform processing}
\]

shows an occam fragment of an ALT construction with a SKIP guard. If Flag is FALSE, the SKIP guard alternative is chosen. If Flag is TRUE, which alternative is chosen depends on the availability of inputs on the channels, Chan1 and Chan2.

A further extension of this is a catch-all for both the boolean and input parts of the guard. This is the TRUE & SKIP combination. Such a guard, since it is always ready, is really only useful when used with the priority ALT construction (Chapter 9). This construction allows alternates to be placed in a priority order and the ready guard may be used in some circumstances as the lowest priority alternate guard.

Note that the different types of guard may be used in the same ALT construction. The example above contains both boolean guards and a SKIP guard.
5.2 Conditional construction

The conditional construction allows a process to be executed provided an associated condition is satisfied i.e. found to be true. The construction consists of the reserved keyword IF followed by a list of choices. A choice comprises a boolean expression i.e. a condition, plus an associated process. The boolean expression is known as a guard.

The format of the conditional construction is

```
IF
  boolean expression 1
  process 1
  .
  .
  .
  boolean expression n
  process n
```

Note that each boolean expression is indented by two spaces from the IF keyword, and that each associated process is indented by a further two spaces from its boolean expression.

Each boolean expression is evaluated in the sequence stated in the construction. The associated process of the first conditional found to be true is executed, and then the conditional construction terminates successfully.

Note that IF guards comprise purely static boolean tests whereas ALT guards comprise dynamic inputs, the choice in the latter case being made from the channels ready to provide an input.

Example 5.4

```
IF
  Data > 0
  Output ! "Positive"
  Data = 0
  Output ! "Zero"
  Data < 0
  Output ! "Negative"
```

If the value of Data is greater than zero, the message "Positive" is output; if the value of Data is equal to zero, the message "Zero" is output; and if the value of Data is less than zero, the message "Negative" is output.
If none of the boolean expressions within the IF construction is found to be true, the construction behaves as a STOP process i.e. it never terminates. It is therefore important that all likely conditions in any IF construction are specified. In the example above, it is possible for the variable, Data, to have a positive, zero or negative value. Hence, three conditions are specified within the IF construction to test for these values.

The boolean literal TRUE may be used as a guard in conjunction with a process to provide a catch-all and eliminate the STOP behaviour. Such a combination must be placed at the end of the conditional. TRUE always evaluates to true, so, if none of the preceding guards is true, the TRUE guard will be taken and its associated process will be executed.

For example,

```plaintext
IF
  Data > 0
  Output ! "Positive"
  Data = 0
  Output ! "Zero"
  TRUE
    Output ! "Negative"
```

where the boolean expression testing for a negative value has been replaced by the SKIP process.

A further possibility is to make the associated process the SKIP process. This produces a catch-all which does no processing at all.

For example,

```plaintext
IF
  Data > 0
  Output ! "Positive"
  TRUE
    SKIP
```

where a message is now output when the data has positive values. Zero and negative values are catered for by the catch-all.
Example 5.5

IF
   (Char >= 'a' (INT)) AND (Char <= 'z' (INT))
   Char := Char BITAND #DF
   TRUE
   SKIP
is an occam fragment which converts lower-case alphabetic characters to upper-case ones. (The variable Char is assumed to have been declared as type INT.) Characters which are not lower-case alphabetic are caught by the TRUE guard.

Example 5.6

IF
   Signal <= Maximum
   SKIP
   TRUE
   STOP
shows the use of STOP to halt a process if something untoward happens.

5.2.1 Nested conditional

As well as being followed by a choice, a conditional construction may alternatively be followed by another conditional, producing a hierarchy of nested conditionals. So constructs such as the following are possible.

IF
   boolean expression
      IF
         boolean expression
            process
and even

IF
   IF
      boolean expression
         process
- the latter construct being most useful with replication (see Chapter 6).
For example,

```
IF
  Colour = Red
    IF
      Suit = Hearts
        Rank := 3
      Suit = Diamonds
        Rank := 2
    Colour = Black
      IF
        Suit = Spades
          Rank := 4
        Suit = Clubs
          Rank := 1
```

shows a nested IF construction. An outer IF comprising two choices testing the value of the variable Colour; and, within each of these choices, inner IF's testing the value of the variable Suit.

Example 5.7

```
IF
  Num1 > Num2
    IF
      Num1 > Num3
        Max := Num1
      TRUE
        Max := Num3
    Num2 > Num3
      Max := Num2
    TRUE
      Max := Num3
```

is an occam fragment for comparing the values of three numbers, and assigning the largest value to the variable Max.
5.3 Selection construction

The selection construction allows a process to be selected from a list of processes according to the value of an expression. Processes are guarded by case expressions, the combination being known as an option. Each case expression must evaluate to a different constant value. The value of a variable expression, called a selector, is matched against the value of each of the case expressions in turn. If a match is found i.e. the selector has the value of a case expression, the associated process of the option is executed and then the selection terminates successfully. If no match is found, the construction behaves as a STOP process. The selection construction comprises the CASE reserved word and selector, followed by zero or more options. There must be at least one case expression in any option. Both the selector and case expressions must be of the same type - either integer or byte.

```
    CASE selector
        case expression 1
            process 1
        ...
        ...
        case expression n
            process n
```

Each case expression must be indented by two spaces with respect to the CASE reserved word; each associated process is indented by a further two spaces from its respective case expression.
Example 5.8

VAL North IS 0 :
VAL South IS 1 :
VAL East IS 2 :
VAL West IS 3 :
INT Direction, XCoord, YCoord :
SEQ
XCoord := 0
YCoord := 0
InChan ? Direction
CASE Direction
    North
        YCoord := YCoord + 1
    South
        YCoord := YCoord - 1
    East
        XCoord := XCoord + 1
    West
        XCoord := XCoord - 1

The value of the selector, Direction is compared with each case expression, North, South, East and West, in turn. If Direction has one of these values, the associated process is executed, otherwise the selection behaves as a STOP process.

A limb of the selection may be guarded by more than one case expression, if required. Multiple case expressions must be separated by commas.

For example,

CASE Suit
    Heart, Diamond
    Colour := Red
    Club, Spade
    Colour := Black

If the selector, Suit, has a value equal to Heart or Diamond, then the variable Colour is assigned the value Red. If Suit has a value equal to Club or Spade, then Colour is assigned the value Black.
Example 5.9

CASE Month
   February
      Days := 28
   April, June, September, November
      Days := 30
   January, March, May, July, August, October, December
      Days := 31

sets the variable Days to the appropriate value for non-leap year months.

Example 5.10

CASE Operator
   '+'
      Result := Operand1 + Operand2
   '-'
      Result := Operand1 - Operand2
   '*'
      Result := Operand1 * Operand2
   '/'
      Result := Operand1 / Operand2

shows the use of characters as selectors and constant expressions in a CASE construction.

5.3.1 ELSE option

This option behaves as a catch-all for the CASE construction. If none of the case expressions match the selector, the ELSE option is taken and the component process executed. The ELSE option may be conveniently placed at the end of the CASE construction.

ELSE
   process
Example 5.11

CASE Day
    Saturday, Sunday
    State := WeekEnd
ELSE
    State := WorkDay

If the selector, Day, is Saturday or Sunday, then the value WeekEnd is assigned to the variable State; otherwise the value WorkDay is assigned to State.

Example 5.12

CASE Month
    February
        Days := 28
    April, June, September, November
        Days := 30
ELSE
    Days := 31

If selector Month has the value February, then the variable Days is assigned the value 28; otherwise, if Month has the value April, June, September or November, then Days is assigned the value 30, else Days is assigned the value 31.

5.4 Repetition construction

A loop may be created within an occam program with a WHILE construction. It has the format

```
WHILE boolean expression
    process
```

The component process must be indented by two spaces with respect to the WHILE keyword. If the boolean expression evaluates to the value TRUE, the component process is executed. On termination of the component process, the procedure is repeated. The loop is repeated an indefinite number of times and only terminates when the boolean expression becomes false. Some action which alters the value of the boolean expression must take place within the loop to enable loop termination.
Example 5.13

\[
\begin{align*}
\text{VAL} & \quad \text{Eof IS ‘.’ :} \\
\text{BYTE} & \quad \text{Char :} \\
\text{SEQ} & \quad \text{Char := ‘ ’} \\
\text{WHILE} & \quad \text{Char <> Eof} \\
\text{SEQ} & \quad \text{InChan ? Char} \\
& \quad \text{OutChan ! Char}
\end{align*}
\]

depicts a WHILE loop which repeatedly inputs and outputs a character until a ‘.’ character is input. This character acts as a terminator for the WHILE loop.

Example 5.14

\[
\begin{align*}
\text{VAL} & \quad \text{Eof IS ‘.’ :} \\
\text{VAL} & \quad \text{Space IS ‘ ’ :} \\
\text{BYTE} & \quad \text{Char :} \\
\text{INT} & \quad \text{CountOfSpaces :} \\
\text{SEQ} & \quad \text{CountOfSpaces := 0} \\
& \quad \text{Char := ‘ ’} \\
\text{WHILE} & \quad \text{Char <> Eof} \\
\text{SEQ} & \quad \text{InChan ? Char} \\
& \quad \text{IF} \\
& \quad \quad \text{Char = Space} \\
& \quad \quad \text{CountOfSpaces := CountOfSpaces + 1} \\
& \quad \quad \text{TRUE} \\
& \quad \quad \text{SKIP}
\end{align*}
\]

is an occam fragment which counts the number of spaces in a piece of text; a character repeatedly being input until a ‘.’ character is found.
A non-terminating loop - one which repeats for ever, may be effected with the boolean expression set to TRUE.

For example,

```
WHILE TRUE
  BYTE Char :
  SEQ
    InChan ? Char
    OutChan ! Char
```

represents a simple buffer process - it repeatedly accepts a character then outputs it (Figure 5.3).

**Example 5.15**

```
BYTE Char :
INT Number :
SEQ
  Number := 0
  InChan ? Char
  WHILE (Char >= '0') AND (Char <= '9')
    SEQ
      Number := (Number * 10) + ((INT Char) - (INT '0'))
      InChan ? Char
```

shows an occam program fragment which forms an integer from ASCII digits i.e. characters. The integer is formed by subtracting the numeric value of the character '0' from the ASCII digit. As more digits are input, the previous number is scaled up by a factor of ten.
Example 5.16

```
VAL  Eof  IS  '?' :
BOOL  Going :
SEQ
  Going := TRUE
  WHILE  Going
  SEQ
    InChan ? Char
    OutChan ! Char
    IF
      Char = Eof
      Going := FALSE
      TRUE
      SKIP
```

represents a WHILE loop, which repeatedly inputs and outputs a character, governed by a boolean variable, *Going*. The loop repeats until a '?' is input, which has the effect of setting *Going* to FALSE and terminating the WHILE loop.

Example 5.17

```
INT  Number, Maximum, Minimum :
SEQ
  Minimum := MOSTPOS INT
  Maximum := MOSTNEG INT
  Input ? Number
  WHILE  Number <> 0
  SEQ
    IF
      Number < Minimum
      Minimum := Number
      Number > Maximum
      Maximum := Number
    TRUE
    SKIP
  Input ? Number
```

depicts an occam fragment which sets the variables *Maximum* and *Minimum* to the largest and smallest numbers read, respectively.
Exercises

1. Devise alternative occam code which will replace the TRUE guard in the following occam program fragment, but still keep the sense of the program.

```occam
IF
    Data < 0
    .  -- some process
    .
    Data = 0
    .  -- some process
    .
TRUE
    SKIP
```

2. Devise alternative occam code which will replace the TRUE guard in the following occam program fragment, but still keep the sense of the program.

```occam
BOOL Running : 
    .  -- some process
    .
IF
    Running
    .  -- some process
    .
TRUE
    SKIP
```

3. Write an occam fragment which will determine the larger of two integers, and will assign the larger to the variable Max and the smaller to the variable Min. Extend this to determine the largest of three integers.

4. Write a WHILE loop to convert a positive integer number to ASCII digits for output as characters. (Your program fragment should cater for the output of leading spaces. Assume a maximum of ten digits.) Extend the fragment to cope with negative and positive integers.
5. A double buffer is a process in which one buffer is used to input whilst another buffer is used to output the previous input. Two simple ways of implementing such a buffer in occam are essentially via
a) a PAR construction comprising two SEQs, and
b) a SEQ construction comprising two PARs.
Write occam programs which will implement a double buffer following these two schemes.

6. Extend the days in the month example given in the text to cater for leap years.

7. Write an occam program fragment which will output a text message for each different vowel input.

8. Write an occam fragment to implement a round robin ALT with two alternatives i.e. an ALT in which each input is guaranteed to be eventually serviced. This may be done, for example, by using a boolean flag which selectively enables successive alternatives of the ALT.
Chapter 6

Replicators

Many programming languages possess a feature to enable the formation of loops which are repeated a definite, specified number of times; for example, a FOR loop in Pascal. Occam too has an analogous, but much more powerful and interesting feature applying specifically to processes. It is known as replication - a process may be replicated a stated number of times. Replication not only applies to the SEQ construction but also to the PAR, IF and ALT constructions. Any of these constructions may be appended with a replicator which specifies a replication count. A following process is then replicated the stated number of times. The format of the replicator is

\[
\text{index = start FOR count}
\]

where

- \textit{index} is the occam identifier of the replicator index. Within the replication loop, the index behaves as an ordinary occam variable, assuming values between \textit{start} and \((\textit{start} + \textit{count}) - 1\). The value of the index must not be changed by the replicated process, but may be used to reference the replicated process within the loop. The replicator statement also serves to declare the index identifier - the index should not be re-specified elsewhere.

- \textit{start} is an expression, the value of which is the initial value of \textit{index}, and

- \textit{count} is an expression, the value of which is the number of times the process is replicated. A count value of zero causes the construction to behave like the SKIP process for a replicated SEQ or PAR, and to behave like a STOP process for a replicated IF or ALT.

The \textit{index} is incremented in steps of one from the value \textit{start} for \textit{count} values. The data type of \textit{index}, \textit{start} and \textit{count} must be INT.
6.1 Replicated SEQ

A replicated SEQ executes the replicated processes in sequence. The format is

\[
\text{SEQ index = start FOR count process}
\]

where \textit{process} is the process to be replicated, and which must be indented by two spaces from the preceding SEQ. The replicated SEQ is analogous to a FOR loop in conventional programming languages, since each execution of the replicated process is performed in sequence.

Example 6.1

\begin{verbatim}
[1024] INT Buffer :
   SEQ Index = 0 FOR 1024
       Buffer [Index] := 0
\end{verbatim}

initialises the contents of a 1024 component buffer, with the replication index, \textit{Index}, taking values 0 to 1023.

A "flattened out" version of this replication would behave as if it had been defined as

\begin{verbatim}
[1024] INT Buffer :
   SEQ
       Buffer [0] := 0
       Buffer [1] := 0
       Buffer [2] := 0
       ...
       ...
       Buffer [1023] := 0
\end{verbatim}
Example 6.2

\[
\begin{align*}
\text{VAL} & \text{ NumberOfRows IS 50 :} \\
\text{VAL} & \text{ NumberOfColumns IS 100:} \\
[\text{NumberOfRows}] & [\text{NumberOfColumns}] \text{ REAL32 Matrix :} \\
\text{SEQ} & \\
\begin{aligned}
& \quad \text{-- initialise matrix} \\
& \quad \text{SEQ} \text{ Row = 0 FOR NumberOfRows} \\
& \quad \quad \text{SEQ} \text{ Column = 0 FOR NumberOfColumns} \\
& \quad \quad \text{OutChan ! Matrix [Row] [Column]} \\
\end{aligned}
\end{align*}
\]

shows an occam fragment which outputs the contents of a two-dimensional matrix. It is assumed that the channel, OutChan, has been specified as type REAL32.

Example 6.3

\[
\begin{align*}
\text{SEQ} & \text{ Index = 0 FOR SIZE String} \\
& \text{OutChan ! String [Index]} \\
\end{align*}
\]

depicts an occam fragment to output an array of bytes (representing a character string), a character at a time.

Example 6.4

\[
\begin{align*}
\text{VAL} & \text{ Count IS 3 :} \\
[\text{Count}] & \text{ REAL64 Vector1, Vector2 :} \\
\text{SEQ} & \\
\begin{aligned}
& \text{DotProduct := 0.0 (REAL64)} \\
& \text{SEQ} \text{ Index = 0 FOR Count} \\
& \quad \text{DotProduct := DotProduct + (Vector1 [Index] * Vector2 [Index])} \\
\end{aligned}
\end{align*}
\]

depicts an occam fragment for calculating the dot product \((\text{Vector1 [1]} \times \text{Vector2 [1]} + \text{Vector1 [2]} \times \text{Vector2 [2]} + \text{Vector1 [3]} \times \text{Vector2 [3]})\) of two vectors.
6.2 Replicated PAR

A replicated PAR executes the replicated processes in parallel. It has the format

\[
\text{PAR index} = \text{start FOR count process}
\]

where process, the replicated process, is indented by two spaces with respect to the PAR. For the replicated PAR only, both start and count must be constant values, as parallel processes cannot be created dynamically.

The replicated PAR is a vital feature of the occam language. It allows the creation of various structures of concurrent processes.

A common use of the replicated PAR is the generation of a pipeline - a set of communicating processes executing in parallel. Each process in the pipeline inputs data from the preceding process, and outputs data to the succeeding process, performing some processing of the data (Figure 6.1). The pipeline may thus produce an overlapped operation - each component process of the replicated PAR executing concurrently with every other component process, input and output being automatically synchronised between processes, with a stream of data passing through the pipeline.

![Diagram of a pipeline with buffers and processes](image-url)

Figure 6.1 Pipeline of processes, with data being passed along the pipeline
Example 6.5

[1025] CHAN OF INT Buffer :
PAR Index = 0 FOR 1024
INT Item :
SEQ
  Buffer [Index] ? Item
  Buffer [Index + 1] ! Item

represents the heart of a 1024-item FIFO buffer process - comprising a concatenation of 1024 simple buffer processes. This concatenation is constructed by the parallel replication of a simple buffer process. In this example, an item is input by the first simple buffer process via the channel Buffer [0], then passed on to the next simple buffer process via the channel Buffer [1], and so on down the line.

If the above replicated PAR process was "flattened out", it would look like

[1025] CHAN OF INT Buffer :
PAR
  INT Item :
  SEQ
    Buffer [0] ? Item
    Buffer [1] ! Item
  INT Item :
  SEQ
    Buffer [1] ? Item
    Buffer [2] ! Item
    ...
    ...
  INT Item :
  SEQ
    Buffer [1023] ? Item
    Buffer [1024] ! Item
This process requires "topping and tailing" - processes need to be provided to cater for the initialisation and termination phases of the pipeline. These processes, typically, would be provided as procedures (Chapter 7). The body of such processes for the initialisation and termination phases would look like

```
-- initialisation
INT Data :
SEQ
  .  -- input or produce an item of data
  .  -- inject the data in to the buffer
    Buffer [0] ! Data
```

```
-- termination
INT Data :
SEQ
  -- extract the data from the buffer
    Buffer [1024] ? Data
  .  -- consume or output an item of data
  .
```

The initialisation and termination processes provide interfaces between the pipeline buffer and the outside world.

The full pipeline program will be an amalgam of these three processes - initialisation, pipeline and termination. Each process should execute concurrently with the other two. This can be arranged in occam by allowing each process to be the component of an outer PAR process. The synchronisation of the communication between each component process will be automatically guaranteed by occam. As a final step, each process must be enclosed within a loop to ensure a stream of data passes along the pipeline. In this fashion, the composite process is capable of buffering up to 1024 items.

For example,

```
PAR
  initialisation process
  pipe process
  termination process
```
CHANG OF INT InChan, OutChan:
[1025] CHAN OF INT Buffer:
PAR
WHILE TRUE
  -- loop to produce stream of data
  -- initialisation
  INT Data:
SEQ
  .
  . -- input or produce an item of data
  .
  Buffer [0] ! Data
PAR Index = 0 FOR 1024
  -- pipeline
WHILE TRUE
  INT Item:
SEQ
  Buffer [Index] ? Item
  Buffer [Index + 1] ! Item
WHILE TRUE
  -- loop to consume stream of data
  -- termination
  INT Data:
SEQ
  Buffer [1024] ? Data
  .
  . -- consume or output an item of data
  .

The program as written above is rather idealised in that it runs for ever. In order to introduce some reality into it, termination must be considered. Chapter 1 discussed how a system of parallel processes may terminate; for example, by setting a boolean flag as a termination notice and by allowing each process to receive, to act on and to pass on this termination notice. In terms of the pipeline, each process must receive the termination notice from the preceding process and pass on the termination to the succeeding process. This may be accomplished by a small modification to each of the above processes - the WHILE is now terminated on some boolean condition, and the body of each process now contains a test for the termination condition.
For example,

```
-- initialisation
BOOL Terminate :
SEQ
  Terminate ::= FALSE
  WHILE NOT Terminate
    INT Data :
    SEQ
      InChan ? Data
      IF
        Data = Eof
        Terminate ::= TRUE
      TRUE
      SKIP
    Buffer [0] ! Data
```

This example assumes that the value used as the terminator is not present as ordinary buffer
data; for example, the terminator may be a negative value if the buffer data comprised only
non-negative values.

The pipeline and termination processes may be similarly extended.

The pipeline is a very potent concept in occam. This type of structure may be very usefully
employed to exploit concurrency in an algorithm (Chapter 11) and so produce more efficient
processing. For example, the technique may be used for writing parallel sorters [Pountain and
May] and parallel prime number generators [Burns]. In order to do this, the conditions need to
be arranged so that each pipeline process can, in fact, execute concurrently with each other
pipeline process i.e. there is overlapped execution. In the case of the FIFO buffer, this means
that there must be a stream of data continually entering and leaving the buffer, so that it is kept
at maximum capacity. A pipeline in which each process executes one after the other produces
no benefit.
6.3 Replicated ALT

In common with the other constructions, an ALT may be replicated. The format is

```
ALT index = start FOR count
    input guard
    process
```

where `input guard` and `process` are replicated. The input guard must be indented by two spaces from the ALT, and the process indented a further two spaces.

The other types of alternative guard - boolean and `SKIP` - may be used instead of an input guard.

The replicated ALT is useful for building a *multiplexor* - a process which accepts inputs from a set of channels and sends the outputs to a single channel.

**Example 6.6**

```
INT Value :
ALT Index = 0 FOR 10
    InChan [Index] ? Value
    OutChan ! Value
```

where the channels have been declared as

```
[10] CHAN OF INT InChan :
CHAN OF INT OutChan :
```

This occam fragment produces an ALT with 10 guards and guarded processes, each inputting from a different channel (`InChan [0] . . . InChan [9]`), but outputting to the same channel, `OutChan` (Figure 6.2).
A "flattened out version of this alternation would look like

```
INT Value:
ALT
    InChan [0] ? Value
    OutChan ! Value
    InChan [1] ? Value
    OutChan ! Value
    ...
    ...
    InChan [9] ? Value
    OutChan ! Value
```
Example 6.7

```plaintext
BOOL Going :
SEQ
   Going := TRUE
   WHILE Going
      BYTE Char :
      INT Any :
      ALT
         ALT Index = 0 FOR 10
            InChan[Index] ? Char
            OutChan ! Char
      Stop ? Any
      Going := FALSE
```

is a variation of the multiplexor theme - the replicated alternatives perform the multiplexor process, whilst a further channel, Stop, provides a means of interrupting the multiplexor (Figure 6.3). Note the interesting use of nested alternations. The Stop channel cannot be accommodated in the replicated ALT, so the replicated ALT must be nested within an outer ALT which contains the Stop channel.

![Figure 6.3 A multiplexor process with a stop channel](image)

The semantic difficulty of the ALT construction carries over to its replicated version - which of the alternatives is chosen when more than one input is ready at the same time? Occam
does not guarantee ever to choose the Stop channel in the example above if inputs are available on other channels. This difficulty is only resolved with the priority ALT (Chapter 9).

6.4 Replicated IF

A number of similar choices may be generated with a replicated IF statement. This has the format

\[
\text{IF } \text{index} = \text{start} \text{ FOR } \text{count} \\
\quad \text{boolean expression} \\
\quad \text{process}
\]

where \text{boolean expression} and \text{process} are replicated. The guard must be indented by two spaces with respect to the IF keyword, and the process indented a further two spaces.

A common use of a replicated IF is in performing a sequential search; for example, searching for a particular number in a list or a particular character in a string.

Example 6.8

\[
\text{IF} \\
\quad \text{IF} \quad \text{Index} = 0 \text{ FOR SIZE List} \\
\quad \quad \text{List [Index]} = \text{RequiredNumber} \\
\quad \text{SEQ} \\
\quad \quad \text{Found} := \text{TRUE} \\
\quad \quad \text{Position} := \text{Index} \\
\quad \text{TRUE} \\
\quad \text{SEQ} \\
\quad \quad \text{Found} := \text{FALSE} \\
\quad \quad \text{Position} := -1
\]

shows a replicated IF construction being used to search for a particular number in a list. The position of the first occurrence of the number is recorded. Note the use of nested conditionals. A replicated IF cannot have a catch-all, so it must be nested within an outer IF to accommodate a catch-all in this outer IF. In this way, if the list does not contain the required number, the process does not behave as a STOP process.
6.5 Segments for efficiency

Use of segments in replicated constructions accessing large arrays can increase the efficiency of occam code.

For example, instead of writing

\[
\begin{align*}
[10000] & \text{ INT } \text{ Original :} \\
[5000] & \text{ INT } \text{ Copy :} \\
\text{ SEQ } & \text{ Index = 0 FOR 3000} \\
& \text{ Copy [Index] := Original [Index]} \\
\end{align*}
\]

it is better to write

\[
\begin{align*}
[10000] & \text{ INT } \text{ Original :} \\
[5000] & \text{ INT } \text{ Copy :} \\
\text{ SEQ} & \text{ [Copy FROM 0 FOR 3000] := [Original FROM 0 FOR 3000]} \\
\end{align*}
\]

The segment version is compiled using a block move instruction, and speed improvements of nearly 50% have been recorded [Winder].

6.6 Processes as data structures

An occam process need not only represent the conventional active processing element of a program. The exciting, innovative nature of occam allows the novel creation of data structures from processes, using replication to generate the required structure. A data structure node is defined as a collection of operations within a process, and the process is replicated to form the structure. Thus a binary tree, for example, may consist of a number of processes (representing the nodes of the tree) linked via channels (representing the branches of the tree) [Burns, Dowsing, Redfern].

Consider the representation of a binary tree. Apart from the leaf nodes, the root and intermediate nodes can each be represented as a process with six channels - one input and one output channel to the parent node, and one input and one output channel to each of the left and right child nodes (Figure 6.4). (It is assumed that the parent of the root node is a user interface process.) Each leaf node can be represented by a process with two channels - one input and one output channel to the parent node.
For simplicity, consider a balanced binary tree of height three (Figure 6.5). The node processes (root, intermediate and leaf) may be most succinctly generated by specifying these processes to be occam procedures (see Chapter 7). These procedures are then replicated the required number of times to form the tree. The root and intermediate nodes may be represented by a procedure with six channel parameters (assumed to be of type INT for this example), and the leaf nodes may be represented by a procedure with two channel parameters.
Thus, the procedure heading for the *Node* process is

```plaintext
PROC Node (CHAN OF INT FromParent, ToLeft, ToRight,
           ToParent, FromLeft, FromRight)
```

and the procedure heading for the *Leaf* process is

```plaintext
PROC Leaf (CHAN OF INT FromParent, ToParent)
```

(The variables within the brackets are the formal i.e. dummy parameters of the procedures.)

The operations of each node will be performed by processes defined within the procedure. An
array of channels is used to provide specific channels i.e. actual parameters of each node generated by the replication. The data structure may now be generated with two replicated \texttt{PAR} constructions - one for the root and intermediate nodes, and one for the leaf nodes. By inspection, the tree can be seen to have seven intermediate nodes and eight leaf nodes. If the channels are numbered as shown in Figure 6.5, then for a node with a parent channel numbered \( N \), the left and right child channels will be numbered \( 2 \times N + 1 \) and \( 2 \times N + 2 \), respectively. Thus, the data structure may be generated as

\begin{verbatim}
[15] CHAN OF INT Down, Up :

PAR
  User (Down [0], Up [0])
  PAR Index = 0 FOR 7
    -- generate root and intermediate nodes
    VAL Parent IS Index :
    VAL Left IS (2 * Index) + 1 :
    VAL Right IS (2 * Index) + 2 :
    Node (Down [Parent], Down [Left], Down [Right],
      Up [Parent], Up [Left], Up [Right])
  PAR Index = 7 FOR 8
    -- generate leaf nodes
    Leaf (Down [Index], Up [Index])
\end{verbatim}

The index values of each replicated \texttt{PAR} construction are chosen to match the channel numbers (Figure 6.5).
Exercises

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

2. Write an fragment using a replicated SEQ to display the alphabet on the screen.

3. Using replication, write an occam program fragment to search for a particular character in a string.

4. Write an occam fragment which will compare two equal length strings in terms of their ASCII character ordering. If the strings are the same, then some variable is set to 0. If the strings are not the same then, if the first string is greater than the second string, the variable is set to 1; otherwise the variable is set to -1.

5. Write an program fragment to perform the matrix multiplication of two two-dimensional square matrices.

6. Modify the pipe and terminate processes given in the text so that they cope with the termination notice.

7. Write a pipeline process to raise each number in a list to the nth power.

8. Write an occam fragment which will act as a multiplexor for a series of inputs. As well as performing the multiplexing function, the multiplexor should output the number of the relevant input port, preceding the data.

9. Write an occam fragment to implement a circular buffer as an array, using the ALT construction to discriminate between the enqueue and dequeue operations.
Chapter 7

Abbreviations, procedures and functions

7.1 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. As remarked in Chapter 2, the constant declaration is a simple form of abbreviation. Abbreviations are categorised as being either expression abbreviations or element abbreviations.

7.1.1 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 expression :
```

where

- `type`, the type of the abbreviation, is one of the primitive data types. The inclusion of `type` is optional and may be omitted if the type may be determined from the data type of the `expression`.

- `name` is the 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 7.1**

**VAL** Return IS ’*c’ :

**VAL** AbsoluteZero 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

\[
\text{Circumference} := (2.0 \text{ (REAL32)} * 3.14159 \text{ (REAL32)}) * \text{Radius}
\]

and

\[
\text{ThisVolume} := (\text{Length} * \text{Breadth}) * \text{Height}
\]

may be written more succinctly as

\[
\text{Circumference} := \text{TwoPi} * \text{Radius}
\]

and

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

Example 7.2

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 or an array of values. Such a mechanism is convenient for data initialisation. The format is

\[
\text{VAL [size] type name IS expression :}
\]

or

\[
\text{VAL [ ] type name IS expression :}
\]

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 7.3

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

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

\[
\text{VAL} \ \text{Primes} \ IS \ [2, 3, 5, 7, 11, 13, 17, 19, 23, 29] :
\]
The first version is useful when it is required to emphasise the size of the array.

Example 7.4

\[
\text{VAL} \ [5] [2] \ \text{BYTE} \ \text{Vowels \ IS} \ [\{'a','A'\}, \{'e','E'\}, \{'i','I'\}, \{'o','O'\}, \{'u','U'\} ] :
\]
and

\[
\text{VAL} \ \text{Vowels} \ IS \ [\{'a','A'\}, \{'e','E'\}, \{'i','I'\}, \{'o','O'\}, \{'u','U'\} ] :
\]
are equivalent initialisations of the two-dimensional array, \text{Vowels}.

Example 7.5

\[
\text{VAL} \ \text{WhiteQueen} \ IS \ \text{ChessBoard} [3] [0] :
\]

\[
\text{VAL} \ \text{Punctuation} \ IS \ ['\.', ',', ';', ';', '?', '!'] :
\]

\[
\text{VAL} \ \text{SummerMonths} \ IS \ ["Jun", "Jul", "Aug"] :
\]

\[
\text{VAL} \ [\ ] \ \text{INT} \ \text{ValidRange} \ IS \ [	ext{Spectrum FROM Start FOR Length}] :
\]
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

\[
\text{NextPieceToMove} := \text{WhiteQueen}
\]

instead of

\[
\text{NextPieceToMove} := \text{ChessBoard [3] [0]}
\]
and

Acceptable := ValidRange

instead of

Acceptable := [Spectrum FROM Start FOR Length]

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

Example 7.6

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 := 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.

7.1.2 Element abbreviations

This form of abbreviation is 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

\[
\text{type name IS element :}
\]

where type is the type of the abbreviation, and which may be omitted if the type may be determined from the type of the element. If type is present, the type of the element must match 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 :
SEQ
  . -- 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 $\text{List}[\text{Middle}]$ must not be referenced as such within the scope of $\text{Pivot}$ - any reference to $\text{List}[\text{Middle}]$ is only valid via a reference to $\text{Pivot}$. Moreover, references to other components of $\text{List}$, for example, $\text{List}[0]$, are illegal within the scope of $\text{Pivot}$. Other components of $\text{List}$ may only be referenced via other abbreviations for these components; for example, declaring the abbreviation $\text{Initial}$ for $\text{List}[0]$ and referencing $\text{List}[0]$ 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 7.7**

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

declares that $\text{MiddleRange}$ is an alias for a segment of the array, $\text{WholeRange}$, and may be referenced as an array. So, for example, $\text{MiddleRange}[4]$ refers to $\text{WholeRange}[14]$. 
7.2 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 [Atkin].

For example, rather than write

```
VAL Start IS 1000 :
[5000] INT Vector :
SEQ
  Vector [Start + Offset1] := Value
  Vector [Start + Offset2] := Value
  .
  .
```

(where Offset1, Offset2, etc. have been declared as type INT constants), it is better to write

```
VAL Length IS 3000 :
VAL Start IS 1000 :
[5000] INT Vector :
Array IS [Vector FROM Start FOR Length] :
SEQ
  Array [Offset1] := Value
  Array [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.
7.3 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 pattern of bits used to represent the program object. Data conversion, on the other hand, may 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 data conversion is.

The format of retyping declarations is

\[
\text{VAL type name RETYPES expression :}
\]

and

\[
type name RETYPES element :
\]

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 7.8

\[
\text{INT32 PackedNumber : [4] BYTE SmallNumber RETYPES PackedNumber :}
\]

retype 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.
7.4 Procedures

The *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 a 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 a 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 defined only within the scope of the following process. Procedures must therefore be defined 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 7.9

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 procedure. This is quite a general idea for the development of a working program - as yet unwritten procedures are represented by procedure stubs, to be filled in later as the development progresses.

Example 7.10

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 an *occam timer* - Chapter 9.)

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 7.11

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

shows an instance of the procedure *Delay* in 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 :              -- |
    SEQ                      -- | instance of Delay procedure
        Count := 0            -- | expanded in-line
        WHILE Count < Limit  -- |
            Count := Count + 1 -- |
```
7.4.1 Parameters

Procedures may have parameters which allow the effect of the procedure to be applied to different values or variables, if required, each time an instance of a procedure occurs. The formal parameters are specified in the procedure heading within the brackets.

```
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 specifer of that type.

Parameters may be one of 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 as parameters), and

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

indicates the values of the first two parameters, `Item1` and `Item2`, remain constant, whilst the values of the second two, `Max` and `Min`, are variable and may be modified within the procedure.
Example 7.12

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

is a procedure to skip a given number of spaces, specified as the parameter value, Number. The channel, Chan, is assumed to be declared with type BYTE. It would be preferable to pass this channel name to the procedure as a channel parameter (see Example 7.18).

Example 7.13

PROC Exchange (INT Item1, Item2)
   SEQ
      Item1, Item2 := Item2, Item1
   :

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

Example 7.14

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 the radius.
Example 7.15

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 a 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 the simple format

```
name (actual 1, ..., actual n)
```

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 :
SEQ
.
.
.
```

in-line expansion

```
-- | 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,

\[
\text{PROC Circle (VAL REAL32 Radius, REAL32 Area, Circumference)} \\
\text{VAL Pi IS 3.14159 (REAL32) :} \\
\text{SEQ} \\
\text{Area := Pi * (Radius * Radius)} \\
\text{Circumference := 2.0 (REAL32) * (Pi * Radius)} \\
\text{SEQ} \\
\text{Circle (R, A, C) -- instance of procedure}
\]

is equivalent to

\[
\text{PROC Circle (VAL REAL32 Radius, REAL32 Area, Circumference)} \\
\text{VAL Pi IS 3.14159 (REAL32) :} \\
\text{SEQ} \\
\text{Radius IS R : -- |} \\
\text{Area IS A : -- |} \\
\text{Circumference IS C : -- |} \\
\text{VAL Pi IS 3.14159 (REAL32) : -- | in-line expansion} \\
\text{SEQ} \\
\text{Area := Pi * (Radius * Radius) -- | of procedure instance} \\
\text{Circumference := 2.0 (REAL32) * (Pi * Radius) -- |} \\
\text{Circle (R, A, C) -- instance of procedure -- |}
\]

The actual parameters \( R, A \) and \( C \) are abbreviated to formal parameters \( \text{Radius}, \text{Area} \) and \( \text{Circumference} \), respectively, for the duration of the procedure invocation. Any changes to the values of \( \text{Area} \) and \( \text{Circumference} \) within the procedure will cause corresponding changes to the values of \( A \) and \( C \).

As remarked at the beginning of the section, parameters may be of any occam type. In particular, this includes channels and arrays. In the 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 \text{SIZE} operator, this allows quite general array handling procedures to be written, without having to be specific about the size of arrays catered for.
Example 7.16

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 7.17

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 7.18

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

is a simple buffer process written as a 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.
7.5 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 or values 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 optional parameters, separated by commas. The kind of any parameter used in a 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 type 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 7.19

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

is a function which delivers the maximum of two integers.

Example 7.20

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

depicts a function for calculating the factorial \((n! = 1 \times 2 \times 3 \times \ldots \times n)\) of a given integer.
Example 7.21

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.

Instances 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,

\[
\text{variable} := \text{name (actual parameters)}
\]

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 must 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

```
type 1, . . . type n  FUNCTION  name (parameters)  
declarations  
VALOF  
  function body  
  RESULT  expression 1, . . . expression n  
:  
```

where the list of types, separated by commas, must match in type and number the list of expressions, 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.
7.6 Function definitions

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

\[
\text{type FUNCTION } \text{name (parameters) IS expression :}
\]

Example 7.22

\[
\text{REAL32 FUNCTION InchesToCms (VAL REAL32 Inches) IS}
\text{ 2.54 (REAL32) * Inches :}
\]

\[
\text{REAL32 FUNCTION Area (VAL REAL32 Length, Breadth) IS}
\text{  Length * Breadth :}
\]

\[
\text{BOOL FUNCTION BufferFull (VAL INT Count) IS Count > 1024 :}
\]

\[
\text{REAL32 FUNCTION Disc (VAL REAL32 A, B, C) IS}
\text{ (B * B) - (4.0 (REAL32) * (A * C)) :}
\]

\[
\text{BOOL FUNCTION IsaDigit (VAL BYTE Char) IS}
\text{ (Char >= '0') AND (Char <= '9') :}
\]

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 kmph)

2. Comment on the validity of the following abbreviations
   a) \texttt{VAL [5] BYTE Greetings IS "Hello" :}
   b) \texttt{VAL [ ] BYTE Greetings IS "Hello" :}
   c) \texttt{VAL Greetings IS "Hello" :}

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

4. Write a set of procedures 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 a palindrome or not. Assume the first space read signifies the end of the word. (A palindrome is a word which reads the same forwards as well as backwards.)

8. Write a function definition to convert from miles per hour to kilometres per hour (50 mph = 80 kph).
9. Write a function definition to convert a Fahrenheit temperature to a Celsius temperature.
   \[ \text{Celsius} = (\text{Fahrenheit} - 32) \times \frac{5}{9} \]

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

11. Write a multi-value function which returns the mean and standard deviation of a list of values.
Chapter 8

Channel protocols

An occam channel protocol specifies the type and format of data which may be transferred on that channel. Such a specification allows the occam system to ensure the correct usage of the channel by a process, in terms of data type and format transferred. Any misuse of the channel is treated as an error. Correct usage also implies that the protocol specifications within the corresponding output and input processes must match. This provides security for the transfer of data between concurrent processes.

To sum up, a protocol in occam terms is just the specification of the data types and number of items of data that a channel is allowed to transfer.

A channel protocol may be one of the following types

- simple
- sequential
- variant
- special

The order specified above denotes the level of protocol sophistication, successive protocols being essentially extensions of the preceding one.

8.1 Simple protocol

The simple protocol is used for transferring either successive single primitive data types or successive arrays of the same primitive data type. The array version is useful, for example, for communicating character strings between processes.
The format of the simple protocol for communicating single values is as follows

\[
\text{CHAN OF type channel :}
\]

where

- \textit{type} is the data type of the channel, and
- \textit{channel} is the channel identifier

For example,

\[
\text{CHAN OF BOOL Lock :}
\]

specifies a channel named \textit{Lock} which supports the successive transfers of a boolean value.

More than one channel of the same type may be specified within the same declaration. Each channel identifier must be separated by a comma.

For example,

\[
\text{CHAN OF BYTE CurrentSensor, VoltageSensor :}
\]

The format of the simple protocol for communicating arrays of values is

\[
\text{CHAN OF [size] type channel :}
\]

where \textit{size} specifies the size of the array.

For example,

\[
\text{CHAN OF [16] INT Switch :}
\]

specifies a channel \textit{Switch} which may communicate 16 integer values at a time.
Example 8.1

CHAN OF [50] INT Stream :
PAR
 [50] INT Results :
 SEQ
 .  -- initialise array
 .
 STREAM ! Results
 [50] INT Data :
 SEQ
STREAM ? Data

shows a process which outputs the array Results along the channel named Stream. The corresponding input process must declare an array of the appropriate size to receive the data.

Multi-dimensional arrays may also be used in channel specifications.

8.1.1 Counted array protocol

An extension to the array version allows any number of array components, up to some maximum number, to be communicated. At run-time, the actual number of array components being transferred is communicated first across the channel before any array components. The associated input process must read this count value, then the array components themselves. Thus, a channel specified in this manner can support successive variable length transfers.

CHAN OF count :: [ ] type channel :

where count represents the occam reserved word for a byte or integer data type. It represents the maximum value of array components which can be transferred. Note the empty array dimensions and the double colon.

For example,

CHAN OF BYTE :: [ ] REAL32 RawData :

depicts a channel which can communicate variable length arrays of real numbers, up to a maximum of 255 (as allowed by the BYTE specifier).
The corresponding input and output processes which use such a protocol have a modified format, the input process having the form

\[
\text{input } ? \text{ length :: array}
\]

and the output process having the form

\[
\text{output } ! \text{ length :: array}
\]

where

- \textit{length} is the number of actual array components being transferred. The data type of \textit{length} must be the same as that declared in the protocol specification.

- \textit{array} is the occam identifier of the array, and

- \textit{input} and \textit{output} are the channel identifiers for the declared counted array protocol.

The data type of the arrays in the input and output processes must be the same, and must be the same as the component type declared in the protocol specification.

Example 8.2

CHAN OF BYTE :: [ ] REAL32  DataStream :
PAR
  [255] REAL32  Buffer :
    SEQ
      DataStream ! 100 (BYTE) :: Buffer
  [255] REAL32  Data :
    BYTE  Length :
    SEQ
      DataStream ? Length :: Data

shows a counted array protocol, \textit{DataStream}, being used to communicate an array between two concurrent processes.
Example 8.3

CHAN OF INT :: [ ] BYTE Stream :
PAR
[1024] BYTE Buffer :
SEQ
Stream ! SIZE Buffer :: Buffer
[4096] BYTE Data :
INT Length :
SEQ
Stream ? Length :: Data

shows the use of a counted array protocol capable of supporting variable length byte transfers. The 1024 components of Buffer are transferred into the first 1024 elements of Data. The maximum theoretical length transfer supported by this protocol will be the maximum size integer allowed by INT - on 32-bit processors, this will be 2147483647 bytes. However, any attempt to transfer an amount greater than the size defined in the array declarations will produce an error. The example also illustrates how the SIZE operator may be used to good effect.

Such a protocol may also be used for transferring parts of an array or string between processes. For example,

CHAN OF BYTE :: [ ] BYTE Comm :
PAR
[10] BYTE Message :
SEQ
Message := "Greetings!"
Comm ! 5 (BYTE) :: Message
[5] BYTE Data :
BYTE Length :
SEQ
Comm ? Length :: Data

transfers the first five characters of Message, "Greet", and so the output corresponds to

Comm ! 5 (BYTE) :: [Message FROM 0 FOR 5]
8.2 Protocol names

Occam has a facility which allows a protocol to be given an explicit name. Such a facility is useful if, for example, the same protocol is being used by a number of channels. The protocol is named and specified once only. Thereafter, any channel declaration using that protocol need only specify the protocol name and not the protocol.

The format is

```
PROTOCOL name IS protocol :
```

where

- *name* is the occam identifier of the protocol name, and
- *protocol* is the protocol specification

A common convention is to write protocol names in upper-case.

Example 8.4

```
PROTOCOL DOUBLE IS REAL64 :
CHAN OF DOUBLE Precision :
```

defines a protocol *Double* capable of supporting the transfer of single *REAL64* types, and a channel *Precision* which uses this protocol definition.

Example 8.5

```
PROTOCOL RECORD IS INT :: [ ] BYTE :
CHAN OF RECORD Stream :
```

represents a protocol capable of dealing with variable length records of bytes.

The naming facility is also used for defining the more sophisticated protocols described in the following sections.
8.3 Sequential protocol

This protocol allows a sequence of one or more simple data types or arrays to be communicated over the channel. In essence, the sequence is a concatenation of simple protocols. The sequential protocol has the format

\[
\begin{array}{l}
\text{PROTOCOL name IS simple 1 ; simple 2 ; \ldots simple n :} \\
\text{CHAN OF name IS channel :}
\end{array}
\]

where \( simple 1, simple 2, \ldots simple n \) are the names of simple protocols, separated by semicolons. Note that the simple protocols used in a sequential protocol must be defined previously with a \text{PROTOCOL} statement, there must be at least one simple protocol in the sequence and that semi-colon separators are only required for more than one simple protocol.

Example 8.6

\[
\begin{array}{l}
\text{PROTOCOL DIMENSIONS IS REAL32 ; REAL32 ; REAL32 :} \\
\text{CHAN OF DIMENSIONS IS Measurements :}
\end{array}
\]

defines a protocol, \text{DIMENSIONS}, for communicating three real values.

In use, the values or variables named in the associated input and output processes must match the specified protocol in type and number. As in the protocol specification, any values or variables must be separated by semi-colons, if there is more than one.

Example 8.7

Given the protocol

\[
\begin{array}{l}
\text{PROTOCOL COORDINATES IS BYTE ; BYTE :} \\
\text{CHAN OF COORDINATES Plotter :}
\end{array}
\]
then the sequence

```plaintext
BYTE XCoord, YCoord :
SEQ
  Plotter ! 110 (BYTE) ; 50 (BYTE)
  XCoord := 15 (BYTE)
  YCoord := 25 (BYTE)
  Plotter ! XCoord ; YCoord
```

represents an occam fragment which uses this protocol to transfer two single byte values.

Protocols comprising mixed data types may also be communicated with the sequential protocol.

**Example 8.8**

Given the protocol

```plaintext
PROTOCOL GRIDVALUE IS BYTE ; BYTE ; REAL32 :
CHAN OF GRIDVALUE Chan :
```

then the sequence

```plaintext
BYTE XCoord, YCoord :
REAL32 Value :
SEQ
  XCoord := 200 (BYTE)
  YCoord := 125 (BYTE)
  Value := 12.5 (REAL32)
  Chan ! XCoord ; YCoord ; Value
```

uses this extended protocol to transfer a real value, in addition to two single byte values.
Example 8.9

```
PROTOCOL DATASTREAM IS INT ; [256] REAL32 :
CHAN OF DATASTREAM Chan :
PAR
[512] REAL32 Data :
SEQ
-- initialise array
.
Chan ! 256 ; [Data FROM 256 FOR 256]
[256] REAL32 Buff :
INT Length :
SEQ
Chan ? Length ; Buff
```

shows how a sequential protocol may be used to transfer the segment of an array, in contrast to the use of a counted array protocol. However, with the sequential protocol only a fixed length transfer is possible - the length of the array specified in the protocol definition.

### 8.4 Variant protocol

The variant protocol allows a channel to transfer a selection of variant, or different, protocols. Only one of these protocol variants may be used at any given time and the selection of which variant occurs at run-time. Each variant of the protocol is known as a tagged protocol because each variant is identified by a tag. The tag is just an occam identifier which distinguishes that variant from the others, and is used in the selection procedure. Each tag must be unique.
The format of this protocol is

```
PROTOCOL name
CASE
  tag 1 ; sequential 1
  .
  .
  .
  tag n ; sequential n
:
CHAN OF name channel :
```

where `tag 1 ; sequential 1, . . . tag n ; sequential n` represent tagged protocols. A tagged protocol may comprise the tag and sequential protocol, separated by a semi-colon, or solely the tag - the sequential protocol may be omitted - as dictated by the requirements of the particular protocol. The reserved word `CASE` must be indented two spaces with respect to the reserved word `PROTOCOL`. Each tagged protocol must be indented a further two spaces. The protocol is terminated by a colon on a line by itself directly under the `P` of `PROTOCOL`. Channels using this protocol may then be declared by specifying the protocol name.

In order for an associated input process to determine which tagged process is being used at any time, the output process must first communicate the tag of that protocol to the input process. Any data associated with that tag is then transmitted.

**Example 8.10**

```
PROTOCOL ARITHMETIC
CASE
  add ; REAL32 ; REAL32
  subtract ; REAL32 ; REAL32
  multiply ; REAL32 ; REAL32
  divide ; REAL32 ; REAL32
:
```

represents a protocol for a process performing arithmetic operations on data received. The tags define the required arithmetic operation - `add`, `subtract`, `multiply` or `divide`. These are followed, in each case, by two real operand values.
Example 8.11

\begin{verbatim}
PROTOCOL IO
CASE
    -- tag, length of file name, file name and open option
    Open ; BYTE :: [ ] BYTE ; BYTE
    -- tag
    Close
    -- tag, length of data to be sent and data
    Put ; INT :: [ ] REAL32
    -- tag and length of data required
    Get ; INT

: CHAN OF IO File:
\end{verbatim}

defines a protocol for communicating with a process to read or write the contents of a file (on disk, say). The required file must first be opened for reading or writing. (The example assumes that only one file is open at a time.) After the get or put operations, the file is then closed. The open, put and get operations comprise tagged sequential protocols whilst the close operation comprises a tag-only protocol. The channel \textit{File} has been defined using this protocol.

Another protocol may be declared to cater for acknowledgements from the process performing the actual I/O as shown in the next example.

Example 8.12

\begin{verbatim}
PROTOCOL IOREPLY
CASE
    -- tag and status
    OpenAck ; BYTE
    -- tag
    CloseAck
    -- tag and status
    PutAck ; BYTE
    -- tag and status
    GetAck ; BYTE
    -- tag, length of data sent and data
    GetReply ; INT :: [ ] REAL32

: CHAN OF IOREPLY Results:
\end{verbatim}
Example 8.13

VAL FileName IS "Sensor.Data" :
[512] REAL32 Readings :
BYTE Status :
SEQ
   -- read sensor data
   Sensor ? Readings
   -- now file it away
   File! Open ; BYTE (SIZE FileName) :: FileName ; 'W'
   Results ? OpenAck ; Status
   File! Put ; SIZE Readings :: Readings
   Results ? PutAck ; Status
   File! Close
   Results ? CloseAck

depicts the use of the previous variant protocols, with the channels, File and Results, being declared via the protocols, IO and IOREPLY, respectively. A file is opened by sending the Open tag, and then the size of the file name, followed by the file name itself and the open option ('W' for write). Next, a block of data is written to the file using the Put tag. Finally, the file is closed - no information needs to be sent apart from the Close tag.

In order to accommodate the input of a number of variant protocols, a special form of the input process, the case input, has been defined as

```
channel ? CASE
   tag 1 ; sequential 1
       process 1
   ...
   ...
   ...
   tag n ; sequential n
       process n
```

where each tagged protocol must be indented two spaces and the associated process indented a further two spaces. Input proceeds according to which tag is received from the output process along the named channel, the relevant component process then being executed. If a mismatch in the tagged protocols occurs, for example the wrong sequential protocol is used for a particular tag or an unknown tag is received, then the case input behaves as STOP process.
Example 8.14

[32] BYTE FileName :
[1024] REAL32 Data :
BYTE FileSize, OpenOpt :
INT Length :
BYTE Status :
SEQ
  -- respond to the relevant tag
File ? CASE
  Open ; FileSize :: FileName ; OpenOpt
  SEQ
    .
    .  -- open process
    .
    Results ! OpenAck ; Status
Close
  SEQ
    .
    .  -- close process
    .
    Results ! CloseAck
Put ; Length :: Data
  SEQ
    .
    .  -- put process
    .
    Results ! PutAck ; Status
Get ; Length
  SEQ
    .
    .  -- get process
    .
    Results ! GetAck ; Status
    .
    .  -- send data if status ok
    .
    Results ! GetReply ; Length :: Data

depicts the use of the CASE input process with the tagged protocol IO. Depending on the tag received, the appropriate protocol and process will be invoked.
Another facet of the case input is its use as an alternative in ALT constructions. This allows such constructions to accept inputs from channels which use variant protocols. As well as the standard format, the case input may be used in conjunction with a boolean expression in the alternative. Thus, quite sophisticated forms of alternation may be built up from combinations of case input alternatives and standard alternatives. ALT constructions with a case input have the format

\[
\text{ALT} \\
\text{channel} ? \text{CASE} \\
\tag 1 ; \text{sequential} 1 \\
\text{process} 1 \\
\ldots \\
\tag n ; \text{sequential} n \\
\text{process} n
\]

and

\[
\text{ALT} \\
\text{boolean} \& \text{channel} ? \text{CASE} \\
\tag 1 ; \text{sequential} 1 \\
\text{process} 1 \\
\ldots \\
\tag n ; \text{sequential} n \\
\text{process} n
\]

where \text{process} 1 \ldots \text{process} n represent the processes to be selected for execution by the appropriate protocol variant. Any number of case inputs may be used as alternatives in the ALT construction.
Example 8.15

ALT

Real ? CASE

--- real arithmetic

add ; REAL32 ; REAL32

. -- process to perform addition

subtract ; REAL32 ; REAL32

. -- process to perform subtraction

multiply ; REAL32 ; REAL32

. -- process to perform multiplication

divide ; REAL32 ; REAL32

. -- process to perform division

Integer ? CASE

--- integer arithmetic

add ; INT ; INT

. -- process to perform addition

subtract ; INT ; INT

. -- process to perform subtraction

multiply ; INT ; INT

. -- process to perform multiplication

divide ; INT ; INT

. -- process to perform division

represents an ALT containing two case inputs - one alternative performing real arithmetic
operations, the other performing integer arithmetic operations. Which set of variants is chosen for execution depends on which case input is available first.

8.5 Special protocol

This protocol is the most general and least secure of all the types of protocol. It allows the transfer of any type and format of data, without any checks being performed. As such, it should be used with caution. With such a protocol, data is treated as a stream of bytes, higher level types being decomposed for output or reconstituted on input. Typically, it is only used to communicate to external devices, such as the screen, which may be considered to be alien processes. The format is

CHAN OF ANY \textit{channel}:

where \textit{channel} is the channel identifier.

For example,

CHAN OF ANY Device:

declares the channel \textit{Device} to be a special protocol channel.

Inmos themselves refer to this type of protocol as anarchic. This is perhaps an indication that the special protocol is no longer favoured owing to the problems which may easily be encountered when the protocol is misused.

In the Transputer Development System (Chapter 12), the screen channel is treated as a special channel (CHAN OF ANY screen), whereas the keyboard channel is treated as a simple channel (CHAN OF INT keyboard). Note that the TDS automatically supplies the specification of the screen and keyboard, and so these channels must not be specified before being used in an occam program executed under the TDS system.
Exercises

1. Write down a channel specification suitable for transferring
   a) a real array of variable size.
   b) the contents of a chess board (the components are assumed to be integers).
   c) a variable length string of characters.

2. Write down a protocol suitable for transferring
   a) a complex number.
   b) single components of a real array, which may be selected at random.
   c) personal details, e.g. name, age, sex.

3. Comment on the validity of the following occam program fragments.
   a)
   
   SEQ
   Message ! "Type response now"
   Message ! "Type response"
   
   where the channel is specified as

   CHAN OF [13] BYTE Message :

   b)

   CHAN OF INT :: [ ] INT Comm :
   PAR
   [512] INT Buffer :
   SEQ
   Comm ! SIZE Buffer :: Buffer
   INT Count :
   [256] INT Block :
   SEQ
   Comm ? Count :: Block
4. Write occam processes to simulate a multiplexor and a demultiplexor. (The multiplexor will need to record the channel number on which data was received.) Define suitable protocols for use with these processes.

5. Devise a variant protocol for the operations \textit{peek} (read a value from a memory address) and \textit{poke} (write a value to a memory address).
Chapter 9

Timers, priority, placement and ports

9.1 Timers

The timer feature in occam makes use of the real-time clock facility of the transputer and is particularly useful in real-time programming applications. As described in the Introduction, the transputer has two timers, one for each of its priority levels. The high priority timer is incremented every microsecond, whilst the low priority timer is incremented every 64 microseconds. The scaling factors for converting clock ticks to seconds are 1,000,000 and 15,625 for the high and low priority levels respectively. By default, occam processes execute at low priority and so access the low priority timer. The priority at which an occam process executes, and hence which priority timer it accesses, may be changed by using the PRI reserved word. Changing the priority of an occam process will be described in the next section.

The timer behaves as an input-only channel - its value can only be read. However, unlike channels, more than one component of a PAR construction may input from the same timer. Timers, in common with variables and channels, must be declared via a timer specification statement. This has the format

```
TIMER timer :
```

where timer is the identifier of the timer within the program.

For example,

```
TIMER Clock :
```

declares a timer named Clock.

More than one timer may be declared in the same specification statement, and arrays of timers may be declared.

For example,

```
TIMER Clock1, Clock2 :
```
and

[5] TIMER Clock:

The value of the real-time clock is regularly updated by the transputer system, and the value at any instant (the current number of clock ticks) may be read using an timer input statement. The format of the timer input statement is

\[ \text{timer ? time} \]

The variable, \textit{time}, which receives the clock value must be of type \textbf{INT}.

Example 9.1

\begin{verbatim}
    TIMER Clock:
    INT Time:
    SEQ
        Clock ? Time
\end{verbatim}

reads the current value of \textit{Clock} into the variable \textit{Time}.

The clock updating mechanism is cyclic such that the clock value wraps round to negative values after passing through its most positive value - the clock values behave in modulo arithmetic value fashion. Consequently, modulo arithmetic must be used in conjunction with timers for calculating time delays and differences. These time delays and differences are only valid if the times compared are within a half a timer cycle of each other. The cycle time depends on the type of the transputer and also the priority level. Approximate cycle times are as follows

\begin{center}
\begin{tabular}{lcc}
\hline
 & low priority & high priority \\
\hline
\text{T212} & 4.2 secs & 65.5 msecs \\
\text{T414, T800} & 76 hours & 1.2 hours \\
\hline
\end{tabular}
\end{center}
9.1.1 Generating delays with timers

One use of the timer is to generate a delay in an occam program. To effect a timed delay, the timer input statement is used in conjunction with the **AFTER** keyword.

\[
timer \ ? \ AFTER \ expression
\]

causes the timer input to be held up until the current clock value is later than the value of the expression (allowing for modulo arithmetic).

Since the current value of the clock is not known until read, a more useful ploy is to read the current clock value and use that value as the starting point of the delay.

**Example 9.2**

\[
VAL \ TicksPerSecond \ IS \ 15625 : \\
TIMER \ Clock : \\
INT \ Now : \\
SEQ \\
\hspace{1em} \text{Clock} \ ? \ Now \\
\hspace{1em} \text{Clock} \ ? \ AFTER \ (Now \ PLUS \ (TicksPerSecond \ * \ 10))
\]

The completion of the timer input is now held off until the ten second delay specified by the expression, \( \text{Now PLUS (TicksPerSecond \ * \ 10)} \) - modulo arithmetic - has elapsed. The process is descheduled for the extent of the delay. Scheduling delays, introduced by the transputer's scheduler, may slightly increase the delay [Inmos Communicating Process Architecture].

9.1.2 Generating timeouts with timers

A timer may be used in conjunction with an **ALT** construction to build a process which will timeout after a certain delay. If no regular input occurs within a specified time, a timeout will be triggered by the timer input and the timeout component process will be executed.
Example 9.3

VAL TicksPerSecond IS 15625 :
VAL OneMinute IS 60 * TicksPerSecond :
TIMER Clock :
SEQ
  Clock ? Now
  ALT
    InChan ? Item
    . -- default process
    .
    Clock ? AFTER (Now PLUS OneMinute)
    . -- timeout process
    .

represents a process with a one minute timeout. If an input occurs on the channel, InChan, before one minute has elapsed, the default process is executed; otherwise the timeout process will be executed.

9.2 Priority

Frequently in real-time applications some processes require more favourable treatment than other processes. This is arranged in occam by allowing processes in PAR and ALT constructions to be assigned a priority for execution - higher priority processes are selected in preference to lower priority processes.

To specify priority, the words PAR and ALT are preceded by the reserved word PRI. The component processes of such priority constructions are given a priority ordering dependent on their textual ordering - the first process receiving the highest priority, and so on. Processes within PRI PAR or PRI ALT constructions must therefore be classed as high or low priority processes.

Priority PARs and ALTs may also be replicated.
9.2.1 PRI PAR

The priority PAR construction provides a mechanism for specifying the priority at which the component processes of a PAR execute. (In a simple PAR, processes execute at the same default low priority level.) This facility uses the hardware priority levels provided by the microcoded scheduler of the transputer as described in the Introduction. Such a construction is useful for ensuring that a high priority device driver process, for example a disk, gets properly scheduled with respect to lower priority device driver processes. The format of a priority PAR construction is

\[
\text{PRI PAR} \\
\text{\hspace{1em} process 1} \\
\hspace{2em} . \\
\hspace{2em} . \\
\hspace{2em} . \\
\hspace{2em} \text{process n}
\]

where process 1 is the high priority process and the other component processes have decreasing priority. Although the occam language definition permits any number of priority levels, the transputer currently supports only two levels of priority - low (timeslicing of processes) and high (no timeslicing). A number of processes may share the priority level by being grouped within an inner PAR construction.

For example,

\[
\text{PRI PAR} \\
\text{\hspace{1em} PAR} \\
\hspace{2em} \text{process 1} \\
\hspace{3em} \text{process 2} \\
\text{\hspace{1em} PAR} \\
\hspace{2em} \text{process 3} \\
\hspace{3em} \text{process 4} \\
\hspace{2em} \text{process 5}
\]

shows how processes 1 and 2 may share the high priority level, whilst processes 3, 4 and 5 share the low priority level.

Using a timer in conjunction with a PRI PAR construction, it is possible to obtain a measure for the elapsed time for process execution. In the following example the process under consideration is run at high priority. This eliminates any timeslicing effects introduced by other processes if low priority was used. The SKIP represents a null low priority process.
Example 9.4

\begin{verbatim}
INT Before, Now, TimeDiff :
PRI PAR
SEQ
  Clock ? Before
  .
  .  process being timed
  .
  Clock ? Now
  TimeDiff := Now MINUS Before
SKIP
\end{verbatim}

9.2.2 PRI ALT

With the priority ALT, if input is available for more than one channel at once, the highest priority input is taken. This device solves the semantic difficulty encountered with the non-priority ALT (Chapter 5). When more than one input is ready at the same time, the process associated with the highest priority input guard is the one selected for execution.

It has the format

\begin{verbatim}
PRI ALT
  input 1
  process 1
  .
  .
  .
  input n
  process n
\end{verbatim}

where the first alternative process is the high priority process and the other alternatives are lower priority processes. The other forms of the alternation guard may be used besides the input guard.

One use of PRI ALT is to provide a priority interrupt channel.
Example 9.5

```latex
\textbf{WHILE} Going
\textbf{PRI} ALT
  Stop ? Any
  Going := FALSE
\textbf{InChan} ? Char
\textbf{OutChan} ! Char
```

If input is available on the \textit{Stop} channel then that alternative will be taken in preference to the \textit{InChan} alternative, even if both inputs are ready at the same time.

Example 9.6

```latex
\textbf{WHILE TRUE}
\textbf{PRI} ALT Index = 0 \textbf{FOR} 10

\textbf{InChan}[\text{Index}] ? Char
\textbf{OutChan} ! Char
```

represents a prioritised multiplexor, input being taken from the lowest indexed channel if more than one input is available simultaneously.

Example 9.7

```latex
\textbf{WHILE TRUE}
\textbf{SEQ}
\textbf{PRI} ALT
  Poll ? Message

  \textbf{TRUE \& SKIP}
  \textbf{SKIP}

  \textbf{-- do other work}
```

depicts a polling process - on every cycle of the \textbf{WHILE} loop, a channel is polled for the availability of a message. If the message is available, it will be processed. Then, other lower priority processes are executed.
9.3 Placement

Usually, program objects, such as constants and variables, are automatically allocated memory locations by the compiler. However, in occam, the programmer is allowed to place variables, channels, timers or arrays at specified physical memory addresses, if necessary. For program objects such as simple variables or arrays memory placement must be regarded as being uncommon, and usually occurs when the use of memory space needs to be especially optimised or external devices need to be "attached" to occam processes. Generally, an occam compiler will try to optimise data space versus code space by attempting to allocate data objects in on-chip memory. Memory allocation is achieved with the PLACE statement. The Inmos TDS compiler (Chapter 12) has especially extended versions of the PLACE statement which must be used for the optimal placement of arrays in memory. Overriding the default allocation of program variables is a non-trivial task, fraught with difficulties. For further information, the necessary compiler and system documentation should be consulted.

The main use of placement is in the allocation of transputer links to memory addresses. Such allocation is necessary for mapping logical channels to physical links (see Chapter 10).

The address space of the transputer is signed and byte addressed. Words are aligned on two-byte or four-byte boundaries depending on processor type. Table 9.1, given overleaf, displays the vital statistics of the transputers' memory address space. However, all this is hidden from the occam programmer. Occam views the memory of the transputer as an array of type INT to provide wordlength independent code. Memory addresses in occam are mapped in to the correct address space by the compiler.

The format of the PLACE statement is

\[
\text{PLACE name AT address :}
\]

where

- \text{name} is the occam identifier of the variable, etc., and

- \text{address} is the absolute memory address, of type INT

The object being allocated memory space must be specified before the placement.
<table>
<thead>
<tr>
<th></th>
<th>T212</th>
<th>T414</th>
<th>T800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word alignment boundary</td>
<td>two-byte</td>
<td>four-byte</td>
<td>four-byte</td>
</tr>
<tr>
<td>Start of internal address space</td>
<td>#8000</td>
<td>#80000000</td>
<td>#80000000</td>
</tr>
<tr>
<td>Number of words used by system at</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start of address space</td>
<td>36</td>
<td>72</td>
<td>112</td>
</tr>
<tr>
<td>Start of user memory</td>
<td>#8024</td>
<td>#80000048</td>
<td>#80000070</td>
</tr>
<tr>
<td>Range of internal address space</td>
<td>#8000-</td>
<td>#80000000-</td>
<td>#80000000-</td>
</tr>
<tr>
<td></td>
<td>#87FF</td>
<td>#80007FF</td>
<td>#8000FFF</td>
</tr>
<tr>
<td>Start of external address space</td>
<td>#8800</td>
<td>#80000800</td>
<td>#80001000</td>
</tr>
<tr>
<td>End of external address space</td>
<td>#7FFF</td>
<td>#7FFFFFFF</td>
<td>#7FFFFFFF</td>
</tr>
</tbody>
</table>

Table 9.1 Transputer memory address space characteristics (byte addresses)
For example,

\[\text{INT Event :}
\text{PLACE Event AT 8 :}\]

places \textit{Event} at memory address 8. This address (of type INT) corresponds to an offset of #10 on a 16-bit T212 (two bytes per INT), and an offset of #20 on a 32-bit T414 or T800 (four bytes per INT). So the end result is \textit{Event} being placed at byte address #8010 on a T212, or #80000020 on a T414 or T800.

Similarly,

\[\text{[8] BYTE Sensors :}
\text{PLACE Sensors AT #400 :}\]

corresponds to array \textit{Sensors} being placed in memory starting at byte address #8800 (T212) or #80001000 (T414, T800).

### 9.4 Ports

Occam supports direct access to external devices via memory-mapping - the hardware registers of devices, such as the input, output and status registers, may be allocated memory addresses, or \textit{ports}, in the transputer’s memory. The registers may then be treated as ordinary program variables and be accessed from an occam process - for example, performing an input or output, or checking the status of the device. However, only input and output operations are allowed with a port. Essentially, a port behaves as a very special channel and must be declared. This declaration has the format

\[
\text{PORT OF type port :}
\]

where

- \textit{type} is the data type of the port
- \textit{port} is the occam identifier of the port

After it has been declared, the port must be \textit{placed} at a memory address.

For example,

\[
\text{VAL Address IS 8 :}
\text{PORT OF BYTE StatusReg :}
\text{PLACE StatusReg AT Address :}
\]

specifies a port, \textit{StatusReg}, placed at memory address 8.
Example 9.8

VAL InputAvailableMask IS #40 :
VAL Reset IS #BF (BYTE) :
PORT OF BYTE InReg : -- input register
PLACE InReg AT #200 :
PORT OF BYTE StatusReg : -- status register
PLACE StatusReg AT #201 :
PORT OF BYTE ControlReg : -- control register
PLACE ControlReg AT #202 :
BYTE Status, Data :
BOOL DataNotAvailable :
SEQ
  ControlReg ! Reset -- initialise control register
  DataNotAvailable := TRUE
  WHILE DataNotAvailable
    SEQ
      StatusReg ? Status -- examine status register
      IF
        (INT Status BITAND InputAvailableMask) = 0
        SKIP -- no data available
        TRUE
        SEQ
          InReg ? Data -- read data
          ControlReg ! Reset -- reset control register
          DataNotAvailable := FALSE
        .
      . -- process data read
      .
represents an occam fragment which implements a busy polling loop, testing a device status register until input in an associated input register is available - as indicated by a bit being set in the status register. The input register is then read and the data processed. (A more efficient scheme would be to employ an interrupt-driven input handler using an ALT construction - this would do away with the unrestrained looping of busy polling.)
Exercises

1. Write an occam program to produce a delay of 2 seconds
   a) at low priority, and
   b) at high priority.

2. In a network of transputers, messages from a sending process may need to be transmitted through intervening transputers to reach the destination process. Using a PRI PAR construction, write a router program to do this. The router should decide whether an incoming message is for this transputer or another. Messages for other transputers should be passed on. For simplicity, assume a pipeline topology. Using a replicated ALT construction, extend this example to cater for a general topology - messages may be input from either of five input channels, representing four transputer links and an internal message channel. Use a look-up table to determine which of five output channels the message is to be sent along.

3. Write a program to time the execution of a SKIP process. To minimise timing errors, time the execution of a large number of SKIPs.

4. Write a program to time the execution of the following process

   \[
   \begin{align*}
   &\text{INT } X, Y, I : \\
   &\text{SEQ} \\
   &\quad X := 0 \\
   &\quad Y := 9 \\
   &\quad I := 0 \\
   &\quad \text{WHILE } I <> 30001 \\
   &\quad \quad \text{SEQ} \\
   &\quad \quad \quad X := ((X + ((Y * Y) - Y)) / Y) \\
   &\quad \quad \quad I := I + 1
   \end{align*}
   \]
Chapter 10

Configuration

So far, all discussion has assumed that the occam program is running on a single transputer system. This need not be the case. Components of an occam program may be distributed over a network of transputers to achieve greater concurrency. In order to do this, component parts of the occam program must be independent (apart from communication). These independent components may then be placed on individual transputers and executed in parallel, any required communication taking place via the transputers’ physical links. (Being independent means that these components are capable of being executed in parallel.) Thus each parallel component may execute concurrently on separate transputers, or, as is more likely to be the case, groups of parallel processes may be placed on separate transputers for concurrent execution to achieve better load sharing. The logic of an occam program is independent of the hardware configuration on which it executes. Once an occam program is proven on a single transputer, it may be distributed over a network of transputers quite safely without any effect on its logical correctness. This chapter explains how an occam program containing identified parallel processes may be distributed over a given network of transputers. Some common approaches for exploiting inherent parallelism in applications are discussed in Chapter 11.

When processes are being distributed over a multi-transputer network, it is necessary to configure the processes for the network of processors (transputers). Configuration, then, is the mapping of processes to a topology of transputers. This configuration procedure takes place on a higher plane than the logic of the program. For best results, the configuration must be tailored to suit the application. For example, a linear configuration of processors, rather than a tree-structure, will be best suited to a pipeline of processes. But in many applications the optimal configuration may not be so obvious. This area is subject to ongoing research and development.

There are three actions which must be performed to achieve a configuration of processes for a network of processors

- specify the type of processors in the system
- declare which processes will execute on which processor. One or more processes may execute on a single transputer.
- map occam channels to transputer links for inter-transputer communications.
Occam is used to achieve this configuration. Occam is not just a programming language, but also serves as a configuration language. The statements described in this chapter only apply to occam programs distributed over a network of transputers.

10.1 PLACED PAR

This construction is used to specify that following named processes will be allocated to different processors. As pointed out above, each collection of processes allocated to a transputer must be capable of executing in parallel with every other collection of processes allocated to other transputers. The PLACED PAR is, in effect, the outermost PAR statement which groups collections of concurrent processes - each collection capable of executing in isolation, apart from inter-process communication.

PLACED PAR

To allow the efficient and easy configuration of large transputer networks, for example a pipeline, the PLACED PAR construction may be replicated.

10.2 PROCESSOR

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

PROCESSOR number 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 processor - T2, T4 or T8 - and is used to check that the program has been compiled for the correct processor.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>16-bit transputer, T212</td>
</tr>
<tr>
<td>T4</td>
<td>32-bit transputer, T414</td>
</tr>
<tr>
<td>T8</td>
<td>32-bit floating-point transputer, T800</td>
</tr>
</tbody>
</table>

### 10.3 Inter-transputer links

The four communications links of a transputer have fixed addresses in the transputer's memory space. These addresses are the first eight words of the internal memory. (Chapter 9). Conventionally, the addresses are mapped into occam identifiers using a **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 inter-*processor* communication or communication with external devices, must be mapped onto the transputer links. *Soft* channels, on the other hand, are implemented via memory locations since the processes using such channels for inter-*process* communication reside on the same transputer. Any link may be chosen to act as a channel to a communicating processor provided it is of the 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* or *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 *outer* procedure with the addition of a procedure heading and terminator. The parameters for this outer procedure are the necessary hard channels for communicating with the outer procedures on other transputers. A characteristic of TDS, the Inmos occam development system, is that each outer procedure must be separately compiled (see Chapter 12).
Each hard channel between two processes on different transputers must be specified in two separate \texttt{PLACE} statements; once as an :input channel for one transputer and once as an output channel for the other transputer.

The order of declarations must follow the general format

\begin{verbatim}
global declarations
PLACED PAR
PROCESSOR ...
  local declarations
  procedure instance
\end{verbatim}

where

- \textit{global declarations} are hard channel declarations, constant declarations, etc.

- \textit{local declarations} are \texttt{PLACE} statements, etc. particular to each transputer.

- \textit{procedure instance} is the invocation of the outer procedure, particular to each transputer. The parameters of this procedure instance are the hard channels between processes on different processors.

These statements may be repeated as necessary to achieve the desired network of transputers.

A number of examples follow which show how these statements may be used to specify the configuration of various topologies of networks. The type and names of the channels, the types of the processors and the procedure names chosen are arbitrary. It should be stressed that these examples are used only to illustrate the principle of placement. Some transputer switches, for example the Inmos C004 switch, place restrictions on which links may be connected together. The appropriate technical manual should be consulted. The configuration statements in the examples are just a direct mapping of the processors, links and connections shown in the accompanying diagrams.
10.4 Examples

1. Two processors connected via two channels, Raw and Processed (Figure 10.1).

CHAN OF REAL32  Input, Output, Raw, Processed :
PLACED PAR
PROCESSOR 0 T4
PLACE Input AT link0in : -- input from outside world
PLACE Output AT link0out : -- output to outside world
PLACE Raw AT link2out : -- output to second processor
PLACE Processed AT link2in : -- input from second processor
Interface (Input, Output, Raw, Processed)
PROCESSOR 1 T8
PLACE Raw AT link0in : -- input from first processor
PLACE Processed AT link0out : -- output to first processor
Transform (Raw, Processed)

Processes Interface and Transform each execute on separate processors. Interface accepts data from the Input channel and passes it to Transform for processing via the Raw channel. Interface accepts processed data from Transform via the Processed channel and outputs it via the Output channel. Hard channels between processes are matched in the procedure instances. (Channels Input and Output are assumed to be host system channels.)

![Figure 10.1 Two processors connected by two channels](image-url)
2. A pipeline of processors (Figure 10.2).

VAL PipeSize IS 50 : -- for example
CHAN OF INT Input, Output :
[PipeSize + 1] CHAN OF INT Pipe :
PLACED PAR
PROCESSOR 0 T4 -- initialisation
   PLACE Input AT link0in : -- input from outside world
   PLACE Pipe [0] AT link2out :
      Initialise (Input, Pipe [0])
PLACED PAR Index = 1 FOR PipeSize -- the pipeline
PROCESSOR Index T4
   PLACE Pipe [Index - 1] AT link0in : -- in to pipe
   PLACE Pipe [Index] AT link2out : -- out of pipe
      PipeProcess (Pipe [Index -1], Pipe [Index])
PROCESSOR PipeSize + 1 T4 -- termination
   PLACE Pipe [PipeSize] AT link0in :
   PLACE Output AT link2out : -- output to the outside world
      Terminate (Pipe [PipeSize], Output)

Processor 0

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3
\end{array}
\]

Processor 1

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3
\end{array}
\]

Processor n

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3
\end{array}
\]

Figure 10.2 Pipeline of processors

Processor 0 executes the initialisation process, whilst processor PipeSize + 1 executes the termination process. Processors in between these two execute the pipeline processes. Again, each procedure instance has matched hard channels. This example illustrates the use of a replicated PLACED PAR to generate the configuration details for the pipeline.
3. A ring of 4 processors (Figure 10.3),

**CHAN OF REAL64 Input, Output:**

[4] **CHAN OF REAL64 Link:**

**PLACED PAR**

**PROCESSOR 0 T8**

- PLACE Input AT link0in: -- input from outside world
- PLACE Output AT link0out: -- output to outside world
- PLACE Link [3] AT link3in: -- receive
- PLACE Link [0] AT link2out: -- transmit
  MasterNode (Input, Output, Link [3], Link [0])

**PROCESSOR 1 T8**

- PLACE Link [0] AT link0in: -- receive
- PLACE Link [1] AT link3out: -- transmit
  Node (Link [0], Link [1])

**PROCESSOR 2 T8**

- PLACE Link [1] AT link1in: -- receive
- PLACE Link [2] AT link0out: -- transmit
  Node (Link [1], Link [2])

**PROCESSOR 3 T8**

- PLACE Link [2] AT link2in: -- receive
- PLACE Link [3] AT link1out: -- transmit
  Node (Link [2], Link [3])
Figure 10.3 A ring of processors
By using links link0in and link3out for input and output on each of the last three processors in the ring (Figure 10.4), the configuration may be written more succinctly as

CHAN OF REAL64 Input, Output :
[4] CHAN OF REAL64 Link :
PLACED PAR
PROCESSOR 0 T8
PLACE Input AT link0in : -- input from outside world
PLACE Output AT link0out : -- output to outside world
PLACE Link [3] AT link3in : -- receive
PLACE Link [0] AT link2out : -- transmit
MasterNode (Input, Output, Link [3], Link [0])
PLACED PAR Index = 1 FOR 3
PROCESSOR Index T8
PLACE Link [Index - 1] AT link0in : -- receive
PLACE Link [Index] AT link3out : -- transmit
Node (Link [Index - 1], Link [Index])

Figure 10.4 A re-arranged ring of processors
4. A figure-of-eight network of processors (Figure 10.5).

CHAN OF REAL32 Input, Output:

[4] CHAN OF REAL32 Link:

PLACED PAR

PROCESSOR T4 0

PLACE Input AT link0in: -- input from outside world
PLACE Output AT link0out: -- output to outside world
PLACE Link [3] AT link2in: -- receive
PLACE Link [0] AT link1out: -- transmit
MasterNode (Input, Output, Link [3], Link [0])

PROCESSOR T4 1

PLACE Link [0] AT link1in: -- receive
PLACE Link [1] AT link0out: -- transmit
Node (Link [0], Link [1])

PROCESSOR T4 2

PLACE Link [1] AT link2in: -- receive
PLACE Link [2] AT link3out: -- transmit
Node (Link [1], Link [2])

PROCESSOR T4 3

PLACE Link [2] AT link3in: -- receive
PLACE Link [3] AT link0out: -- transmit
Node (Link [2], Link [3])

Again, this configuration may be made more succinct by judicious placement of the links.
Figure 10.5 A figure-of-eight network of processors
Chapter 11

Approaches to writing parallel programs in occam

The previous chapters have concentrated on the various program constructs available in occam and have explained their use. In order to reap the most benefit from the use of occam and transputers, programs need to exhibit some degree of parallelism. Every application needs to be considered to see how the inherent parallelism may be best expressed and exploited. But this is not an easy exercise. The conventional mould of thinking and expressing the solution to a programming problem as a sequence of steps must be broken. A number of approaches for applying parallelism to problems have yielded promising results for occam and the transputer. However much more experience is needed in the art and science of parallel programming.

Parallelism is expressed within an occam program by parallel constructions. Given the fact that each parallel process may be mapped on to a separate processor for execution, the potential benefits in terms of speed and efficiency may be enormous. Nevertheless, efficiency considerations need to be taken into account so that the benefits gained from parallelism are not lost.

The granularity of the parallelism in the application and how it is distributed over a transputer network need to be carefully assessed. Granularity is a measure of parallelism - the number of parallel processes in an application. A large amount of parallelism is not necessarily an ideal situation. It is necessary to achieve a balance between keeping each processor busy with computation and maintaining inter-process communication time at a minimum. The current versions of the transputer do not provide message routing in hardware; valuable computation time can be taken up in providing extra occam processes to implement software message routing between processes. With a too fine-grained granularity, there will be many parallel processes and then the communications overhead can dominate the computation. However in coarse-grained granularity, with not so many parallel processes, the computation should dominate the communications.

The organisation of programs around the parallel paradigm has been approached in a number of different ways. These various approaches may be broadly classified into the following categories

- algorithmic - quasi-independent tasks which execute sections of the problem solution algorithm (and are therefore non-identical), data and computed results being passed among the tasks as the algorithm dictates.
• geometric - quasi-independent but identical tasks which process a portion of
the data, and which interact with (or are affected by) neighbouring tasks,
according to the geometry of the problem.

• process farming - fully independent but identical tasks which process the data
in any order.

Analysis of the problem using data flow diagrams presents a graphical method for visualising
the component processes of a program - the processes are represented by circles, the channels
are represented by arcs connecting the circles. Such a method has been found useful in the
design of occam programs [Kerridge]. Another graphical method which may also prove useful
for designing occam programs is the Petri-net notation [Peterson]. This technique allows the
 graphical expression of parallel strands of computation, so that its suitability to the design of
 occam programs should be apparent. CSP [Hoare], the theoretical basis for occam, is an
excellent tool for expressing the design of occam programs.

The following sections of the chapter discuss in more detail the three approaches to writing
parallel programs in occam, giving a concrete example in each case. For simplicity of
presentation, the handling of process termination has been omitted in the first two examples.
Realistically, this should always be provided - see Chapter 6 for an example how this might be
achieved. The third example contains code for process termination.

11.1 Algorithmic parallelism

The algorithmic approach, also known as data flow decomposition, is concerned with injecting
parallelism into the algorithm being used to solve the problem at hand. The algorithm may be
an already existing sequential one or totally new. Parallelism can be introduced by considering
how the algorithm may be broken up into separate, quasi-independent sections. Each section
can then be executed in parallel, with data flowing between the sections as necessary. Each
section will perform some computation with the data, and then pass the data on to the next
section.

The inherent parallelism is frequently found in a loop or iteration. Consider a linear search for
example. In the sequential case, each item in the list is compared one at a time as the search
sequences through the list. This comparison may however be performed more efficiently in
parallel - each comparison may be performed at the same time.

The algorithmic approach is modelled by occam parallel processes, each parallel process
responsible for the execution of a section of the algorithm, using the synchronisation and
communication provided by occam channels to transfer data between the processes. The
communication overhead between the parallel processes in such circumstances can become
quite significant.

A common example of the algorithmic approach is the pipeline - each unit of the pipeline contributes by executing a section of the algorithm. The independent units may operate on separate portions of the data as the stream of data is fed down the pipeline. The overall effect of this overlapped operation is the realisation of parallel execution.

Such organisation is not limited to one-dimensional cases. For example, a systolic array is effectively a two-dimensional pipeline that may be used to great effect for the parallel execution of matrix operations [Jones and Goldsmith].

As an example of the algorithmic approach, consider the Newton-Raphson estimate technique for evaluating square-roots. This method starts off with the value of the number whose square-root is required and an initial estimate of the 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 of times, the final estimate will be a close approximation to the real value.

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

\[
y_{i+1} = \frac{1}{2}(y_i + \frac{x}{y_i})
\]

where \( y_i, y_{i+1} \) are successive estimations of the root \( y \).

This formula may be expressed in occam as

\[
\text{Estimate} := (\text{Estimate} + (\text{Number} / \text{Estimate})) / 2.0 \text{ (REAL32)}
\]

where \( \text{Number} \) is the number whose square root is required and the initial value for \( \text{Estimate} \) is given. Usually, the initial value for \( \text{Estimate} \) is taken to be

\[
\text{Number} / 2.0 \text{ (REAL32)}
\]

If the iteration is performed \( \text{Iterations} \) times, then a sequential solution may be written as

**SEQ**

`Input ? Number`  
`Estimate := Number / 2.0 (REAL32)`  
`SEQ Index = 0 FOR Iterations`  
`Estimate := (Estimate + (Number / Estimate)) / 2.0 (REAL32)`  
`Output ! Estimate`
Figure 11.1 A pipeline of processes to calculate the square root of a number using the Newton-Raphson method

The Newton-Raphson procedure may be written in a concurrent form by considering each iteration as an occam parallel process. Each of these processes accepts the number and previous estimate as input, calculates the new estimate and produces the number and new estimate as output (Figure 11.1). Thus the approximation technique may be written as a pipeline of identical processes with the necessary initialisation and termination processes. The pipeline may be generated with a replicated PAR statement. The top level of the program will have the following form

```
global declarations
procedures comprising
   Initialisation process
   Pipeline processes
   Termination process
main process
```

The global declarations comprise the channels for the pipeline processes and the channels for the initialisation and termination processes. Assuming 50 iterations, this section may be written as

```
-- global declarations
VAL Iterations IS 50 :
[Iterations + 1] CHAN OF REAL32 Pipe :
CHAN OF REAL32 InChan, OutChan :
```
The * Initialise * process inputs the number whose square root is required, and outputs the value of this number and initial square root estimate to the first pipeline process. This may be expressed in pseudo-code as

```plaintext
WHILE data is available
  SEQ
    INPUT a number
    SEND this number and initial square root estimate to first pipeline process
```

In occam, this may be written as

```plaintext
PROC Initialise (CHAN OF REAL32 Input, Inject)
  WHILE TRUE
    REAL32 Number :
    SEQ
      Input ? Number
      -- feed number and initial estimate into pipeline
      Inject ! Number
      Inject ! Number / 2.0 (REAL32)
  :
```

As indicated previously, the pipeline comprises *Iterations* identical processes which are generated with a *PAR* replicator. Each of these processes inputs the number and the previous estimate from the preceding pipeline process and outputs the number and new estimate to the succeeding process. Expressed in pseudo-code, this becomes

```plaintext
WHILE data is available
  SEQ
    RECEIVE a number and previous square root estimate from preceding pipeline process
    SEND this number and new square root estimate to next pipeline process
```
Written in occam, this becomes

```
PROC Pipeline (CHAN OF REAL32 InPipe, OutPipe)
  WHILE TRUE
    REAL32 Number, Estimate :
    SEQ
      -- accept number and previous estimate
      InPipe ? Number
      InPipe ? Estimate
      -- pass on number and new estimate
      OutPipe ! Number
      OutPipe ! (Estimate + (Number / Estimate)) / 2.0 (REAL32)
  :
```

Finally, the Terminate process inputs the number and final estimate of the square root from the last pipeline process and outputs this value. Writing this in pseudo-code

```
WHILE data is available
  SEQ
    RECEIVE a number and final square root estimate from last pipeline process
    OUTPUT this number and final estimate as results
```

In occam, this becomes

```
PROC Terminate (CHAN OF REAL32 Extract, Output)
  WHILE TRUE
    REAL32 Number, Root :
    SEQ
      -- extract number and root (final estimate) from pipeline
      Extract ? Number
      Extract ? Root
      -- output results
      Output ! Number
      Output ! Root
  :
```

The main process will comprise a PAR construction containing instances of Initialise, Pipeline and Terminate. The Pipeline process is replicated the desired number of times.
PAR
  Initialise (InChan, Pipe [0])
  PAR Index = 0 FOR Iterations
  In IS Pipe [Index] :
  Out IS Pipe [Index + 1] :
  Pipeline (In, Out)
  Terminate (Pipe [Iterations], OutChan)

The amounts of computation required for the sequential and parallel solutions are the same. However, the benefit derived by expressing the sequential algorithm as a parallel one accrues only when there are many numbers requiring the calculation of their square roots. The partial estimates for these numbers may all be within the pipeline at the same time (depending on the length of the pipeline) - each pipeline process can be calculating a different partial estimate. The automatic synchronisation of occam ensures the correct order of communication and, hence, the correct order of computation.

Other examples of the use of this approach are sorting [Fountain and May], prime number generating [Burns], systolic array processing [Jones and Goldsmith], compiling and solid modelling [May and Shepherd].

11.2 Geometric parallelism

With this approach, parallelism is introduced by making use of any regular spatial geometry or structure present in the problem. Rather like a large cube may be divided up into a number of smaller constituent cubes, so the spatial geometry of the problem is divided up in some symmetrical fashion, assuming a uniform distribution of data over the geometry, to allow a more tractable solution to be expressed. Each of these small units acts as a quasi-independent entity, responsible for the data in its own spatial region. The computation performed by each small unit is summed to give an overall effect. Interactions between neighbouring units may be incorporated to give a more realistic solution. This approach is also known as data structure decomposition.

Each small unit is modelled by an identical occam parallel process, each parallel process operating on the data relevant to its own domain. Interactions between nearest neighbours may be introduced with occam channels connecting the neighbouring units. The communications overhead between these communicating processes may become quite appreciable.

An example of the geometric approach is its use in the simulation of thermal conduction in a two-dimensional rectangular metal plate which is being heated by a heat source at a certain point. Simulation of thermal conduction over the whole plate is difficult. So, to simplify the problem, the geometry of the situation is utilised and the plate is subdivided into a number of
rectangular areas - these areas being the quasi-independent units which will be represented by occam processes. The heat conduction i.e. temperature of each of these areas may be estimated and summed to give an approximate effect for the heat conduction over the whole plate. The temperature of each area will depend on that of its surroundings i.e. the neighbouring areas. It is assumed that one of the areas contains the heat source.

For the example, consider a metal plate, n by m units (Figure 11.2). The program is required to monitor the temperature at the centre of each of these areas. Also, for the example, consider that two boundaries (top and left-hand side) of the plate are adjacent to an infinite heat sink and that the other two boundaries (bottom and right-hand side) of the plate are adjacent to a perfect heat insulator.

The simulation program will comprise a set of identical parallel processes, each responsible for determining the temperature of one of the areas of the metal plate. This temperature is taken to be an average of the temperatures of the four neighbouring areas. In addition areas on the boundaries of the plate will be affected by the type of adjacent boundary - the heat sink will maintain a constant (base) temperature and the heat insulator will reflect the temperature of the boundary areas. These boundary effects will be simulated by extra parallel processes.

Each plate area will have nine channels - an input and output channel for each of the neighbouring areas, up, down, left and right plus a result channel (Figure 11.3). The result channel communicates with a monitor process to display the current temperature of each area on the screen.

There will be n * m processes for calculating the areas' temperatures plus n + m processes for each of the two different boundary effects. This number of processes may be generated by suitable replication of the following processes

- Sink - simulate the effect of a heat sink
- Insulator - simulate the effect of a heat insulator
- Source - simulate the effect of a heat source
- CalcTemp - calculate the temperature of an area. Account must be taken of the fact that one of the areas will contain the heat source.

The top level of the program will have the following form

global declarations
procedures comprising
    Sink
    Insulator
    Source
    CalcTemp
main process
Figure 11.2 Metal plate subdivided into smaller areas
Assuming a metal plate of dimension 3 by 3 units, the global declarations are as follows:

```
VAL Height IS 3 :
VAL Width IS 3 :
VAL TwiceHeight IS 2 * Height :
VAL TwiceWidth IS 2 * Width :
VAL Rectangle IS Height * Width :
VAL TwiceRectangle IS 2 * Rectangle :
VAL SourceX IS 1 :
VAL SourceY IS 2 :
VAL BaseTemp IS 50.0 (REAL32) :
[TwiceRectangle + TwiceHeight] CHAN OF REAL32 Horizontal :
[TwiceRectangle + TwiceWidth] CHAN OF REAL32 Vertical :
[Rectangle] CHAN OF REAL32 Result :
```

The initial temperature, `BaseTemp`, is assumed to be 50 degrees. The position of the area containing the heat source at its centre is given by `SourceX` and `SourceY`. The areas communicate via the channels `Horizontal`, `Vertical` and `Result`. The horizontal channels are numbered in right/left pairs down the columns of the array of areas. The vertical channels are numbered in down/up pairs along the rows of the array. The result channels are numbered along the rows (see Figure 11.4 for an example of the 3 by 3 array).

The `Sink` process is simulated by maintaining (outputting) a constant temperature, regardless of the adjacent temperature - the temperature input is disregarded. This constant temperature is taken to be the initial temperature of the plate. Expressing this in pseudo-code gives:

```
WHILE simulation is required
  PAR
  RECEIVE a temperature of an adjacent area and IGNORE
  SEND a constant (base) temperature back to the adjacent area
```

Written as occam code, this gives:

```
PROC Sink (CHAN OF REAL32 In, Out)
  -- left or bottom boundary
  WHILE TRUE
    REAL32 Any :
    PAR
      -- ignore adjacent temperature
      In ? Any
      -- output a steady temperature
      Out ! BaseTemp
  :
```
Figure 11.3 Channels of a plate area
The *Insulator* process does not allow any heat to escape - the temperature read from the adjacent area is returned. Written in pseudo-code, this is

```plaintext
WHILE simulation is required
  SEQ
    RECEIVE a temperature from an adjacent area
    SEND the temperature received back to the adjacent area

Rewriting in occam gives

```plaintext
PROC Insulator (CHAN OF REAL32 In, Out)
  -- top or right boundary
  WHILE TRUE
    REAL32 Temp :
    SEQ
      -- input temperature of adjacent area
      In ? Temp
      -- return last temperature read
      Out ! Temp

Process *Source* simulates a heat source by generating a temperature which increases at a steady rate - one degree higher than the previous value. This temperature is transmitted to the surrounding areas, ignoring the present temperature of these areas. The heat source will be surrounded by four neighbouring areas. Expressing these requirements in pseudo-code gives

```plaintext
WHILE simulation is required
  SEQ
    PAR
      RECEIVE a temperature from four surrounding areas and IGNORE
      SEND new temperature back to four surrounding areas
      INCREMENT temperature by one degree
Figure 11.4 The areas and channels for a 3 by 3 array
In occam, this may be written

```
PROC Source (CHAN OF REAL32 UpIn, DownIn, LeftIn, RightIn,  
              UpOut, DownOut, LeftOut, RightOut,  
              Result)

VAL TempIncrease IS 1.0 (REAL32) :

REAL32 Temp :

SEQ
  -- initial condition
  Temp := BaseTemp

WHILE TRUE
  -- output new temperature to four surrounding areas, ignoring any inputs

SEQ

PAR
  REAL32 Any :
  PAR
    -- area below
    DownIn ? Any
    DownOut ! Temp

REAL32 Any :
  PAR
    -- area to left
    LeftIn ? Any
    LeftOut ! Temp

REAL32 Any :
  PAR
    -- area above
    UpIn ? Any
    UpOut ! Temp

REAL32 Any :
  PAR
    -- area to right
    RightIn ? Any
    RightOut ! Temp

Result ! Temp    -- output new temperature to Monitor

Temp := Temp + TempIncrease    -- increase temperature of source
```

(The interactions with each adjacent area have been grouped in separate PAR constructions for clarity.)
Process \textit{CalcTemp} will calculate the rise in temperature of each area due to the temperature of the neighbouring areas. Account must be taken of the fact that one area will contain the heat source. Written in pseudo-code, this is

\begin{verbatim}
IF area = heat source THEN
    GENERATE temperature rise
ELSE
    WHILE simulation is required
        SEQ
            PAR
                RECEIVE the temperature of four surrounding areas
                SEND the temperature of this area to four surrounding areas
                CALCULATE new temperature of this area, based on temperature
                rises of surrounding areas
\end{verbatim}

Expressing this pseudo-code in occam,

\begin{verbatim}
PROC CalcTemp (BOOL HotSpot,
                CHAN OF REAL32 UpIn, DownIn, LeftIn, RightIn,
                UpOut, DownOut, LeftOut, RightOut,
                Result)

    IF
        -- if area contains heat source
        HotSpot
            -- generate temperature rise
            Source (UpIn, DownIn, LeftIn, RightIn,
                    UpOut, DownOut, LeftOut, RightOut, Result)
    TRUE
        -- area does not contain heat source
    REAL32 Temp :
        SEQ
            -- initial conditions
            Temp := BaseTemp
\end{verbatim}
WHILE TRUE

REAL32 SumOfTemps, MeanTemp, 
UpTemp, DownTemp, LeftTemp, RightTemp, 
DeltaUp, DeltaDown, DeltaLeft, DeltaRight :

SEQ
-- interact with neighbouring areas

PAR
-- area below

PAR
DownIn ? DownTemp
DownOut ! Temp
-- area to the left

PAR
LeftIn ? LeftTemp
LeftOut ! Temp
-- area above

PAR
UpIn ? UpTemp
UpOut ! Temp
-- area to the right

PAR
RightIn ? RightTemp
RightOut ! Temp

DeltaDown := DownTemp - Temp
DeltaLeft := LeftTemp - Temp
DeltaUp := UpTemp - Temp
DeltaRight := RightTemp - Temp
-- now average these temperatures to find mean rise
SumOfTemps := ((DeltaUp + DeltaDown) 
+ DeltaLeft) + DeltaRight
MeanTemp := SumOfTemps / 4.0 (REAL32)
-- increase temperature by half average temperature rise
Temp := Temp + (MeanTemp / 2.0 (REAL32))
-- output the result to Monitor
Result ! Temp


Process Monitor will be responsible for keeping a record of the temperature of each area, and
displaying this temperature on the screen. Expressing this in pseudo-code gives

WHILE simulation is required
SEQ
PAR
    INPUT temperature from each area
SEQ
    IF temperature <> last temperature from each area THEN
        DISPLAY temperature

Writing this in occam gives

PROC Monitor ([ ] CHAN OF REAL32 Result)
    [Rectangle] REAL32 LastTemp :
SEQ
    -- initialise array holding temperatures
SEQ Index = 0 FOR Rectangle
    LastTemp [Index] := 0.0 (REAL32)
    [Rectangle] REAL32 Temp :
WHILE TRUE
SEQ
    -- input temperature of areas
PAR Index = 0 FOR Rectangle
    Result [Index] ? Temp [Index]
SEQ Index = 0 FOR Rectangle
    -- check for a temperature change
    IF
        Temp [Index] <> LastTemp [Index]
        INT Row, Col :
        SEQ
            -- display new temperature
            Row := Index / Width
            Col := Index REM Width
            Display ! Row ; Col ; Temp [Index]
            LastTemp [Index] := Temp [Index]
        TRUE
        SKIP
        :
The overall structure of the main process will be an outer **PAR** enclosing the requisite number of instances of *CalcTemp, Sink* and *Insulator* processes. In addition there will be an instance of the *Monitor* process. Thus the main process may be expressed in pseudo-occam as

```
PAR
  Monitor process
  n * m  CalcTemp processes
  n + m  Sink processes
  n + m  Insulator processes
```

This process may be rewritten in terms of nested replicated **PAR**s (assuming n rows by m columns of rectangular areas - the index *Row* moving from top to bottom, the index *Column* moving from left to right) as follows

```
PAR
  Monitor process

  PAR  Row = 0 FOR n  -- left-hand side areas
       Sink process

  PAR  Column = 0 FOR m

  PAR  -- top side areas
       Insulator process

       -- middle areas
       PAR  Row = 0 FOR n
            CalcTemp process

       -- bottom side areas
       Sink process

  PAR  Row = 0 FOR n  -- right-hand side areas
       Insulator process
```
Using the (more meaningful) constants Height (for n) and Width (for m) as specified in the global declarations, this may be rewritten as

\textbf{PAR}

\texttt{Monitor (Result)}

\textbf{PAR Row = 0 FOR Height} \quad -- \text{left-hand side areas}

\textbf{VAL Out IS Row + Row :}

\textbf{VAL In IS Out + 1 :}

\texttt{Sink (Horizontal [In], Horizontal [Out])}

\textbf{PAR Col = 0 FOR Width}

\textbf{PAR}

\quad \quad -- \text{top side areas}

\textbf{VAL Out IS Col + Col :}

\textbf{VAL In IS Out + 1 :}

\texttt{Insulator (Vertical [In], Vertical [Out])}

\quad \quad -- \text{middle areas}

\textbf{PAR Row = 0 FOR Height}

\textbf{VAL Up IS ((TwiceWidth * Row) + Col) + Col :}

\textbf{VAL Down IS ((TwiceWidth * (Row + 1)) + Col) + Col :}

\textbf{VAL Left IS ((TwiceHeight * Col) + Row) + Row :}

\textbf{VAL Right IS ((TwiceHeight * (Col + 1)) + Row) + Row :}

\textbf{VAL Hot IS (Row = SourceY) AND (Col = SourceX) :}

\textbf{SEQ}

\texttt{CalcTemp (Hot,}

\quad \texttt{Vertical [Up],}

\quad \texttt{Vertical [Down +1],}

\quad \texttt{Horizontal [Left],}

\quad \texttt{Horizontal [Right + 1],}

\quad \texttt{Vertical [Up + 1],}

\quad \texttt{Vertical [Down],}

\quad \texttt{Horizontal [Left + 1],}

\quad \texttt{Horizontal [Right],}

\quad \texttt{Result ((Width * Row) + Col))}
-- bottom side areas  
VAL In IS (TwiceRectangle + Col) + Col :  
VAL Out IS In + 1 :  
Sink (Vertical [In], Vertical [Out])

PAR Row = 0 FOR Height -- right-hand side areas  
VAL In IS (TwiceRectangle + Row) + Row :  
VAL Out IS In + 1 :  
Insulator (Horizontal [In], Horizontal [Out])

Other examples of the use of this approach are the modelling of a statistical "spin" system as may be found in liquid crystal films [Askew et al.].

11.3 Process farming

The farm approach is applicable to problems whose solution will succumb to a decomposition into many smaller parts and where these parts are independent of each other. As the parts are independent, each may be executed concurrently, in isolation, and the effect summed to give a solution to the whole problem. The solution is analogous to a farmer supervising the toil of many farm workers, each worker performing any given task in isolation from the other workers - hence the name.

A farm is modelled by a set of occam processes. One process is nominated the farmer. The farmer process controls the organisation and allocation of work. The controlling process farms or hands out work to its subordinate worker processes. The worker processes are modelled as identical parallel occam processes. As and when each worker process completes the given task, the farmer process will issue further work for completion. Thus a farm of worker processes toil away on parts of the problem, finishing one task and starting another, until the whole problem is complete. Typically little inter-process communication is needed in such applications. However, depending on the number of worker processes and their configuration, for example whether they are organised in a linear or tree fashion, the routing of messages between the farmer and workers may well cause communications overhead problems.

An example of the process farm approach is its use in producing a graphical representation of the Mandelbrot set, or more exactly, a graphical representation of those points which lie within and without the Mandelbrot set [Barnsley et al., Peitgen and Richter]. This set comprises all complex numbers, \( c = a + ib \), for which the recurrence relation

\[
z_{n+1} = z_n^2 + c \quad \text{for } n = 0, 1, 2, \ldots
\]

converges to a finite complex number (where \( z_n \) and \( z_{n+1} \) are complex numbers computed in
successive iterations of the recurrence relation, and \( z_0 = 0 \) is the initial condition). It can be demonstrated that, if for some \( n \),

\[
|z_n| > 2
\]

then the iteration diverges and hence \( c \) does not belong to the Mandelbrot set.

In practice, the iteration is performed a given number of times, \( m \), and \( c \) is considered to belong to the Mandelbrot set, \( M \), if

\[
|z_n| < 2 \quad \text{for all } n \leq m \quad (11.2)
\]

The graphical display of the members of \( M \) produces quite vivid and intriguing self-similar shapes known as *fractals*.

For display purposes, the complex number, \( c = a + ib \), is taken to be a graphics screen pixel with coordinates \((a,b)\) - the graphics screen representing the complex plane. For every screen pixel the recurrence relation is applied. If the pixel belongs to the Mandelbrot set, it is coloured black, otherwise it is allocated a colour from the graphics palette which is graded according to the speed at which the iteration diverges i.e. the smallest natural number \( n < m \) for which \( |z_n| \geq 2 \).

Such computation is quite intensive for a suitable number of iterations and, depending on the size of the graphics screen and hence the number of pixels, needs to be performed a large number of times. The actual computational task to be performed for each pixel is the same but the amount of computation will vary depending on whether or not the sequence of recurrence values for that pixel converges or diverges.

The general form of a farm in terms of pseudo-occam is as follows

```
global declarations
PAR
  Farmer process
    PAR Index = 0 FOR NumberOfWorkers
    Worker process
```

Each worker process accepts data from the farmer process, works with this data and then sends the result back to the farmer process, becoming available to accept more data. In the current context this work will be the calculation of the recurrence relation for the given data i.e. pixel coordinates \((a,b)\).
Assuming a graphics area of 512 by 512 pixels, with 50 workers, the global declarations section may be written as

```plaintext
VAL NumberOfWorkers IS 50:
VAL NumberOfPixels IS 512 * 512:
PROTOCOL RAW
  CASE
    Data ; [2] INT
    Terminate
  :
PROTOCOL PROCESSED
  CASE
    Results ; [2] INT ; INT
    Quit
  :
[NumberOfWorkers + 1] CHAN OF RAW ToFarm :
[NumberOfWorkers + 1] CHAN OF PROCESSED FromFarm :
```

The channels, `ToFarm` and `FromFarm`, allow the `Farmer` process to send data to the `Worker` processes and receive results from the `Worker` processes. The protocols, `RAW` and `PROCESSED`, will be explained shortly.

In practice, to improve the efficiency trade-off between computation and communications, each worker would be given a line of pixels as data. The processor overhead setting up a transmission over a transputer link is the same for many bytes as for a few bytes. (Once a data transfer has been initiated, the transfer of data over the link is autonomous of the processor.) For simplicity, this example considers the data to be a single pixel. Also in practice, it may be advantageous to have a division of labour in the farmer process, having a farmer process proper and a separate graphics process. The function of the farmer process would be to hand out pixel coordinates to the worker processes, whilst that of the graphics process would be to accept the results (pixel coordinates and colour) and display them on the graphics screen.

Rather than allowing a worker process to idle whilst the farmer issues new work, the worker process may buffer an extra unit of work so that it may proceed immediately with this new work once it has completed the previous work. This scheme keeps the workers constantly busy [Packer].

Logically, each worker process may be connected via a channel to the farmer. Practically, since the transputer has only four links and if the workers are distributed over several transputers, there may be many tiers of worker processes. Because of this, each worker process will not just be concerned with the iteration of the recurrence relation. It will also act as a message switch, passing on data to processes further down the farm. The whole farm
process becomes self-regulating, message passing being synchronised by the occam input/output primitives. In addition to forwarding work to outlying workers, the worker process will gather results from these workers for onward delivery to the farmer (or graphics) process.

Each worker process will comprise three processes: \textit{Switch}, \textit{Feedback} and \textit{Mandelbrot} (Figure 11.5). This arrangement may be expressed in pseudo-occam as

\begin{verbatim}
PROC Worker
    PRI PAR
    PAR
        Switch process
        Feedback process
        Mandelbrot process

: This arrangement of processes in the PRI PAR construction ensures a high throughput for communications. This is important for processes which may use the transputer links, so that messages are transmitted without delay. If a high priority process was not used, the message would not be examined until the message switch was scheduled by the low priority round-robin scheduler of the transputer [May and Shepherd].

Tagged protocols, \textit{RAW} and \textit{PROCESSED}, are defined for the data which is sent to and received from the worker processes. The tag \textit{Data} of protocol \textit{RAW} corresponds to the transfer of two integers (a pixel), whilst the tag \textit{Results} of protocol \textit{PROCESSED} corresponds to the transfer of three integers (a pixel and its colour). In addition, each of these protocols has a tag which is used to pass a termination notice to the participating processes at the end of the calculation.

The \textit{Worker} process may now be rewritten with the addition of channels.

\begin{verbatim}
PROC Worker (CHAN OF RAW FromPrevious, ToNext,
                CHAN OF PROCESSED ToPrevious, FromNext)

CHAN OF BYTE MoreWork :
CHAN OF RAW Work :
CHAN OF PROCESSED WorkDone :
PRI PAR
    PAR
        Switch (MoreWork, FromPrevious, ToNext, Work)
        Feedback (ToPrevious, FromNext, WorkDone)
        Mandelbrot (MoreWork, Work, WorkDone)

:
\end{verbatim}

The \textit{Switch} process is responsible for accepting pixels (work) from the \textit{Farmer} process, buffering a pixel for its \textit{Mandelbrot} process, and forwarding excess work to workers further...
down the farm. Expressing this in pseudo-code gives

```
WHILE pixels are available
  ALT
    RECEIVE request from mandelbrot for another pixel
    IF buffer = full THEN
      SEND buffered pixel to mandelbrot
    ELSE
      SET mandelbrot = idle
    RECEIVE pixel from farmer
    IF mandelbrot = idle THEN
      SEND pixel to mandelbrot
    ELSEIF buffer = empty
      BUFFER pixel
    ELSE
      SEND pixel to next worker
```

Such a structure with more than one input to react to may be conveniently programmed using an ALT construction. The actual code will be slightly more complicated than the above pseudo-code owing to the need to watch out for and pass on the termination notice to the Mandelbrot process and the next worker. This is just a matter of reacting to the relevant tag of the channel protocol. The occam for the process is

```
PROC Switch (CHAN OF BYTE MoreWork,
             CHAN OF RAW FromPrevious, ToNext, Work)
  BOOL Busy, Buffered, Running :
  SEQ
    Busy := FALSE
    Buffered := FALSE
    Running := TRUE
  WHILE Running OR Busy
    [2] INT Coords, BufferedCoords :
    BYTE Any :
    ALT
      -- mandelbrot requesting more work
      Busy & MoreWork ? Any
      IF
        Buffered -- check for buffered work
        -- pass mandelbrot the buffered work
        SEQ
          Work ! Data ; BufferedCoords
        Buffered := FALSE
```

Figure 11.5 The component processes of a worker process
TRUE -- no buffered work
Busy := FALSE
-- a message from the farmer
Running & FromPrevious ? CASE
    Data ; Coords -- another pixel
    IF
        NOT Busy -- check if mandelbrot busy
        SEQ
            Work ! Data ; Coords
            Busy := TRUE
        NOT Buffered -- check if pixel buffered
        SEQ
            BufferedCoords := Coords
            Buffered := TRUE
        TRUE -- pass on pixel
            ToNext ! Data ; Coords
        Terminate -- termination notice
            Running := FALSE
        ToNext ! Terminate -- pass on termination notice
        Work ! Terminate
    :

In the above process, the boolean variable, Running, records whether or not a termination notice has been received from the farmer, whilst the boolean variable, Busy, marks whether or not the Mandelbrot process is processing a pixel. The Mandelbrot process performs the iteration procedure for a given pixel and assigns that pixel a colour dependent on the degree of convergence. With a little rearrangement the recurrence relation may be simplified for computation. Substituting $z = x + iy$, then equation (11.1) may be written as

$$x_{n+1} = x_n^2 - y_n^2 + a$$

and

$$y_{n+1} = 2x_ny_n + b$$

Since

$$| z_{n+1} | \equiv \sqrt{x_{n+1}^2 + y_{n+1}^2}$$

the condition for Mandelbrot set occupancy - equation (11.2) - may be written as

$$x_{n+1}^2 + y_{n+1}^2 \leq 4$$

Pixels are passed from the Switch process each time the Mandelbrot process completes the previous calculation and requires more work. Completed work, in terms of the pixel colour, is
passed on to the \textit{Feedback} process. In pseudo-code, this gives

\begin{verbatim}
    WHILE pixels are available
        SEQ
        RECEIVE pixel
        loop:
            CALCULATE next iteration of recurrence relation
            IF iteration count = maximum THEN
                ASSIGN black to colour
                EXIT
            IF modulus squared > constant THEN
                ASSIGN count to colour
                EXIT
        SEND pixel and colour to feedback
\end{verbatim}

Again the actual code will be complicated by the termination condition. This time the termination notice is passed on to the \textit{Feedback} process. Expressed in occam (assuming a graphics palette of 256 colours, with the colour black having a value of 0, and a maximum number of iterations of 255), this gives

\begin{verbatim}
PROC Mandelbrot (CHAN OF BYTE MoreWork, 
                CHAN OF RAW Work, 
                CHAN OF PROCESSED WorkDone)

    BOOL Running :
    SEQ
        Running := TRUE
        WHILE Running
            BYTE Any :
            [2] INT Coords :
            SEQ
            -- ask for some work
            MoreWork ! Any
            Work ? CASE
                Data ; Coords -- next set of coordinates
                VAL Constant IS 4.0 (REAL32) :
                VAL Two IS 2.0 (REAL32) :
                VAL MaxIterations IS 255 :
                VAL Black IS 0 :
                REAL32 A, B, X, Y, ZSquared :
                INT Colour, Count :
\end{verbatim}
SEQ
  A := Coords [0]
  B := Coords [1]
  X := 0.0 (REAL32)
  Y := 0.0 (REAL32)
  Count := 0
  ZSquared := 0.0 (REAL32)
  -- calculate next iteration of recurrence relation
  -- and test for divergence
  WHILE (Count < MaxIterations) AND (ZSquared <= Constant)
    SEQ
      X := ((X * X) - (Y * Y)) + A
      Y := (Two * (X * Y)) + B
      ZSquared := (X * X) + (Y * Y)
      Count := Count + 1
    IF
      Count = MaxIterations -- pixel in Mandelbrot set
      Colour := Black
      ZSquared > Constant -- pixel outside Mandelbrot set
      Colour := Count
      WorkDone ! Results ; Coords ; Colour -- send results back
      Terminate -- termination notice
      Running := FALSE
      WorkDone ! Quit -- pass on the termination notice

The Feedback process multiplexes the results from its Mandelbrot process and those received from other workers on the farm, and feeds them back to the Farmer process. Putting this in pseudo-code

    WHILE pixels are available
      ALT
      RECEIVE pixel and colour from our worker
      SEND pixel and colour back to graphics process
      RECEIVE pixel and colour from other workers
      SEND pixel and colour back to graphics process

The code may be succinctly expressed in occam using an ALT construction. This time the treatment of the terminating condition needs more effort. The termination notice is only passed on when one has been received from both the local Mandelbrot process and the next worker [Jones and Goldsmith].
PROC Feedback (CHAN OF PROCESSED ToPrevious, FromNext, WorkDone)
[2] INT Coords :
INT Colour :
BOOL Local, Other :
SEQ
Local := TRUE
Other := TRUE
WHILE Local OR Other
ALT
Local & WorkDone ? CASE
Results ; Coords ; Colour -- results from our worker
ToPrevious ! Results ; Coords ; Colour -- pass back to farmer
Quit -- termination notice from our worker
Local := FALSE
Other & FromNext ? CASE
Results ; Coords ; Colour -- results from another worker
ToPrevious ! Results ; Coords ; Colour -- pass back to farmer
Quit -- termination notice from next worker
Other := FALSE
ToPrevious ! Quit -- pass on termination notice
:

In the above process boolean variables, Local and Other, record whether or not a termination notice has been received from the local worker or another worker respectively. Only when a termination notice has been received from both these processes does the Feedback process terminate.

A simplistic farmer process, which assumes that the farm is arranged as a chain of worker processes, is presented below. The farmer sends each pixel to the first worker in the chain for redistribution. The farm is primed by issuing $2 \times \text{NumberOfWorkers}$ pixels to the work force. This amount of data just fills up each worker and each buffer. As each pixel is reported processed, another pixel is issued to the farm, until all the pixels have been processed [Atkin]. At this point a termination notice is issued to the farm of worker processes and the farmer waits to receive this back before finally terminating itself. In pseudo-code this is

PRIME farm with pixels
loop:
RECEIVE completed work from farm
SEND another pixel to fram
UNTIL pixels exhausted
The termination notice will pass down the chain of worker processes via the switch processes causing each switch process to terminate on receipt. Before terminating, each switch process will inform its mandelbrot process to terminate, which in turn will inform the feedback process of a local termination. When the termination notice reaches the end of the chain of worker processes, it must be returned to the farmer process via the feedback processes, causing each feedback process to terminate (provided the local termination has also been received). Writing this in occam gives

```
PROC Farmer (CHAN OF RAW ToWorker,
             CHAN OF PROCESSED FromWorker)

    INT WorkDone, WorkWanted :
    BOOL Running, Terminating :

SEQ
    -- prime the farm
    SEQ Index = 0 FOR 2 * NumberOfWorkers
        -- send pixel (row and column)
        ToWorker ! Data ; [Index / 512, Index REM 512]
    WorkDone := 2 * NumberOfWorkers
    WorkWanted := 0
    Running := TRUE
    Terminating := FALSE

    WHILE Running
        INT Colour :
            [2] INT ResultCoords :

        PRI ALT
            FromWorker ? CASE
                Results ; ResultCoords ; Colour -- receive completed work
                    SEQ
                    .
                    . -- plot result
                    .
                WorkWanted := WorkWanted + 1
            Quit -- termination notice returned
            Running := FALSE
```
(WorkWanted > 0) & SKIP
IF
  WorkDone < NumberOfPixels
  SEQ
    -- send another pixel
    ToWorker ! Data ; [Index / 512, Index REM 512]
    WorkDone := WorkDone + 1
    WorkWanted := WorkWanted - 1
  NOT Terminating
  SEQ
    Terminating := TRUE
    ToWorker ! Terminate -- issue termination notice to farm
  TRUE
  SKIP
:

In the above process, the variable WorkDone keeps a count of the number of pixels processed, while the variable WorkWanted keeps a count of the number of pixels required to top up the farm.

The main process will comprise a PAR construction containing instances of the Farmer process and a number of replicated Worker processes. The last worker in the chain is a special case as it has no one else further down the chain to communicate with. A dummy process, EndStop, is provided to match the channels of this last worker so they are not left dangling. This dummy process accepts the termination notice from the switch process of the last worker (protocol RAW) and and generates a new one to pass back down the chain of feedback processes to the farmer (protocol PROCESSED) [Jones and Goldsmith].

PAR
  Farmer (ToFarm [0], FromFarm [0])
  PAR Index = 0 FOR NumberOfWorkers
    Worker (ToFarm [Index], ToFarm [Index + 1],
      FromFarm [Index], FromFarm [Index + 1])
  EndStop (ToFarm [NumberOfWorkers], FromFarm [NumberOfWorkers])
An alternative to the dummy process approach is to treat the last worker separately [Packer]. This approach requires that the protocols *RAW* and *PROCESSED* be combined into one protocol, *COMBINED*, say.

**PROTOCOL COMBINED**

**CASE**

*Data ; [2] INT*

*Results ; [2] INT ; INT*

*Terminate :*

This means, of course, that the process and channel declarations, and the tag input and outputs must be altered accordingly. For example, the *Worker* process becomes

**PROC Worker (CHAN OF COMBINED FromPrevious, ToNext,**

*ToPrevious, FromNext)*

**CHAN OF BYTE** MoreWork :

**CHAN OF COMBINED** Work, WorkDone :

**PRI PAR**

**PAR**

*Switch (MoreWork, FromPrevious, ToNext, Work)*

*Feedback (ToPrevious, FromNext, WorkDone)*

*Mandelbrot (MoreWork, Work, WorkDone)*

: and the *Feedback* process becomes

**PROC Feedback (CHAN OF COMBINED ToPrevious, FromNext, WorkDone)**

*[2] INT Coords :*

**INT** Colour :

**BOOL** Local, Other :

**SEQ**

*Local := TRUE*

*Other := TRUE*

**WHILE** Local OR Other

**ALT**

*Local & WorkDone ? CASE*

*Results ; Coords ; Colour -- results from our worker*

*ToPrevious ! Results ; Coords ; Colour -- pass back to farmer*

*Terminate -- termination notice from our worker*

*Local := FALSE*
Other & FromNext ? CASE
    Results ; Coords ; Colour -- results from another worker
    ToPrevious ! Results ; Coords ; Colour -- pass back to farmer
    Terminate -- termination notice from next worker
    Other := FALSE
    ToPrevious ! Terminate -- pass on termination notice
:

A special channel, LoopBack is declared as follows

CHAN OF COMBINED LoopBack :

This channel is looped back in the last worker process to provide a return path for the termination notice.

PAR
    Farmer (ToFarm [0], FromFarm [0])
    PAR Index = 0 FOR NumberOfWorkers - 1
        Worker (ToFarm [Index], ToFarm [Index + 1],
                 FromFarm [Index], FromFarm [Index + 1])
    Worker (ToFarm [NumberOfWorkers], LoopBack
             FromFarm [NumberOfWorkers], LoopBack)

Another example of process farming is its application to ray tracing to generate realistic images of scenes [Packer]. Such an application requires considerable amounts of processing power. It has been shown that the processing speed is directly proportional to the number of transputers used for this generation of images.

11.4 Efficiency factors

Even after designing a parallel algorithm, there are a number of competing factors to be taken into consideration for an efficient implementation.

- processor connectivity - the transputer has only four physical links. Depending on the distribution of processes on processors, communications between processes may need to pass through several intervening transputers. This routing of messages imposes an extra overhead on each transputer, and the balance between computation and communication needs to be carefully assessed.

- processor loading - the processing load of each transputer in a network must
be taken into consideration. The system is likely to run at the speed of the transputer with the highest processing load, as the other transputers in the system will probably be held up, waiting to communicate with this transputer. So the processing load should be shared as evenly as possible among the available transputers, and not left to chance or haphazard placement. The farm approach semi-dynamically balances the load for each processor, since a processor only receives more work when it becomes idle. The algorithmic approach needs especial care, as an overloaded processor may create a bottleneck in the pipeline or whatever configuration is chosen.

- processor type and memory - there are different types of transputer available with different word sizes and floating point capabilities. The choice of transputer for a particular function within a network needs to be carefully considered to match applications with suitable processors. For example, computation-bound tasks involving floating-point operations will obviously benefit from the use of a T800 processor. As regards memory, the program memory requirement must be balanced against the use of internal memory (fast but finite) and external memory.
Appendix

The transputer development system

Probably the most common way of developing occam programs is via the Inmos transputer development system (TDS). This chapter provides an introduction to TDS; the system is fully described in the relevant Inmos document [Inmos Transputer Development System].

The transputer development system is an integrated, interactive, program development environment for occam programs. It provides a folding editor, an occam compiler, a run-time environment, a debugger, a library system containing many procedures and functions for I/O and numerical computation, and a number of other utilities. Folding is a novel structuring of files and parts of files to allow abstraction of program detail, and will be described shortly.

Within the TDS environment, occam programs may be developed to execute

- on the TDS board transputer

- on a network of transputers connected to the TDS board

- on a standalone transputer system, entirely separate from the TDS

A typical TDS system comprises an IMS B004 (or compatible) plug-in board for an IBM PC (or compatible). This board contains an IMS T414 transputer with 2Mb of memory. Also required is the TDS software (IMS D700D). The PC provides the input/output facilities (keyboard, display screen and filestore) for the development software which executes on the transputer board.

The main user interaction with TDS is via the PC’s keyboard and screen. Instead of interacting with the development system via a set of typed commands, the user communicates commands via key presses. One key press may be a combination of SHIFT, CTRL or ALT keys with other keys. When a particular TDS application is loaded, the keys are automatically assigned specific functions.

225
For example, keys are programmed to provide

- movement of the cursor around the screen
- scrolling of the screen
- insertion and deletion of characters and lines
- compilation, loading and execution of programs

In this chapter, TDS key presses will be denoted by the name of the function, enclosed within square brackets. For example, [ENTER FOLD] denotes the key press to enter a TDS fold. The Inmos documentation fully describes the assignments of particular key presses to functions.

Before any program development can take place, the TDS utilities must be copied from a TDS directory to the user’s directory. These utilities reside in the file \tds2\system\toplevel.tkt, which is known as the *toolkit fold*.

The organisation of occam source program files, compiled files, etc., is based on a *fold* structure, which comprises a set of nested files. The top level file of this structure is a special DOS file which has a .top extension. An empty top level file must be created from the DOS command level with the command

```
   tds2 -t name.top
```

before any program files can be created. (Here *name* is a DOS file name.) Thereafter, the TDS system is entered by typing the command

```
   tds2
```

from the DOS command level. This causes the development system to be loaded into the transputer, and the current top levels files to be displayed, with the following message (TDS reserves the top line of the screen for messages).

```
   Press [ENTER FOLD] to start session
```

This invites the user to enter any of the current top level files displayed, by positioning the cursor on the selected file and pressing [ENTER FOLD].

During use, the TDS utilities may create a number of files with differing DOS file extensions to delineate the use of each file.
For example,

- **.top** - top level file
- **.tsr** - occam program source file
- **.dcd** - occam program object code file
- **.ddb** - debugging information file
- **.tc1** - configuration information file

### A.1 The folding editor

The editor environment is automatically entered whenever the tds2 command is given. The TDS editor embodies a rather novel concept of *folding* - nominated sections of a file may be folded away, out of sight. If and when required, these folded sections may be opened out for viewing. This is directly analogous to a piece of paper being folded in such a way that the contents on part of the paper are not visible, but may be opened out for viewing if required (Figure 12.1). Such sections of a file are known as *folds*. Folds may be nested within other folds to form a hierarchical fold structure. Nested folds may be opened out to reveal inner structure. The current view on the screen at any time is the current fold, or that part of it which can fit on the screen.

The concept of folds allows the top-down development, stepwise refinement of programs. Folds at the top level contain the top level program structure. Nested folds contain more and more program detail. Viewing this detail may not be necessary in all circumstances, and the fold structure allows the user to zoom in to the required level of detail as and when necessary.

Folds may be either open or closed. When a fold is open, its contents are visible and are sandwiched between two marker lines known as *creases*. Creases are represented on the screen by opening and closing pairs of three curly brackets.

```plaintext
{{ - top crease
}}
```

When a fold is closed, its contents are hidden from view and its presence is indicated on the screen by a marker line known as a *fold line*. The fold line is represented on the screen by three fullstops.

```
. . . - fold line
```
A fold line may be annotated with some text which describes the function of the particular fold.
For example,

```
... declarations
... procedures
... process
```

shows three folds and associated annotating text. This annotation is automatically appended to the top crease when the fold is opened.

For example,

```
{{
... declarations
... channels
... constants
... variables
}}

... procedures
... process
```

shows the fold *declarations* open and having three inner (nested) folds.

A more concrete example is the following occam program fragment.

```
... declarations
SEQ
... initialisation
... process body
```
This fragment has three folds as shown. Opening the *process body* fold produces

```plaintext
... declarations
SEQ
  ... initialisation
  {{
    process body
    WHILE TRUE
    SEQ
      InChan ? Char
      ... IF construction
  }}
```
This shows the *process body* fold contains some occam statements and another fold. Note that folds are indented to follow the program structure indentation - when the fold is created, it is automatically given the indentation of the first line in the fold. Opening the *IF construction* fold gives

```
... declarations
SEQ
... initialisation
{{{  process body
WHILE TRUE
  SEQ
    InChan ? Char
    {{{  IF construction
      IF
        IF Index = 0 FOR 5
          Char = Vowels[Index]
        SEQ
          VowelCount := VowelCount + 1
        InChan ? Char
      TRUE
      SKIP
    }}}
}}}
```

The concept of folds applies not only to occam source programs. Other text, and indeed binary information, may be folded. Some of the TDS utilities create data folds for holding binary data.

A fold may be nominated as a *filed* fold. With this type of fold, the contents are stored in a file separate to any other folds in the current file. A filed fold is denoted on the screen by letter F appended to either the top crease (if the file is open) or to the fold line (if the file is closed).
For example,

```occam
{{
  F  program
  . . . F  declarations
  . . . F  procedures
  . . . F  process
}}
```

shows a filed fold `program` open and having three nested file folds.

When a filed fold is opened, the contents are automatically accessed from the file, and when a filed fold is closed, the file is automatically updated if the fold contents have been modified.

The editor provides facilities for the input of an occam program from the keyboard. A user is allowed to move through the program source by scrolling through pages on the screen and to move the cursor to any point on the screen, all via the use of cursor keys. There are a number of operations available via the function keys for manipulating folds. Some of the available operations are shown below.

- enter a fold - a nominated fold is entered and displayed on the screen. The preceding fold is stacked away. A fold is nominated by positioning the cursor on its fold line and pressing the [ENTER FOLD] function key.

- exit a fold - the current fold is closed, and the preceding fold is unstacked and displayed on the screen.

- open a fold - a variant of enter fold. The nominated fold is shown displayed in the context of its preceding (next higher level) fold.

- close a fold - a variant of exit fold, used in conjunction with open fold.

- create a fold - a nominated section of the program becomes a fold. A section of program is nominated by positioning the cursor at the beginning and on the line following the end of the section, and pressing the [CREATE FOLD] function key at each position. The three curly brackets are created on each key press and, on the second key press, the cursor is positioned to allow the addition of the fold name.

- remove a fold - a fold structure is removed, the contents of the fold becoming part of the preceding fold.
• file a fold - a nominated fold is filed as a separate file.

• unfile a fold - a filed fold is converted to an ordinary fold, no longer being filed separately.

Other functions provided by the editor environment include

• get code - load a compiled program into memory.

• run - execute a loaded program.

• finish the session - the TDS session may be terminated, and a return made to DOS.

To compile an occam program, the compiler utility must be loaded. This utility, along with others, resides in the toolkit fold. These utilities must be specifically loaded into memory via function key presses.

• autoload - load a set of utilities from the toolkit fold.

• next util - cycle through the set of utilities loaded, and make one the current utility.

• enter toolkit fold - the toolkit fold may be entered and a specific utility loaded via the [GET CODE] key press.

### A.1.1 Example session

Typing

tds2 -t example.top

creates an empty top level file called example and produces the message

Press [ENTER FOLD] to start session

. . . F EXAMPLE.TOP

Entering this fold, by positioning the cursor on the fold line . . . F EXAMPLE.TOP, gives

```om
{{ F EXAMPLE.TOP
}}
```

This is an empty fold - there is nothing between the crease marks. An occam program fold may be created within it, either by typing the program and creating a fold around the program,
or by creating an empty fold and putting the occam program in it, as described previously.

For example,

```occam
{ {{ F EXAMPLE.TOP
 { {{
  double
  INT A, B :
  SEQ
    B := 3
    A := 2 * B
  }}
}}
```n
shows the result of creating an occam program fold called `double`. The program fold may be closed by pressing [EXIT].

For example,

```occam
{ {{ F EXAMPLE.TOP
   . . . double
  }}
}
```

With the cursor on `double`'s fold line and pressing [FILE] produces

Filed OK as double.tsr

```occam
{ {{ F EXAMPLE.TOP
   . . . F double
  }}
}
```

The program fold `double` has now been filed.

### A.2 Compiling occam programs

The compiler and other utilities are loaded by pressing [AUTOLOAD]. As each utility is loaded, a message is displayed on the screen. The compiler is the last utility loaded, and responds with the message

```
CODE UTIL occam 2 compiler utilities
```

Besides the compiler itself, the compiler utility contains other facilities; for example for making foldsets and for checking the validity (syntax) of the occam program.

Before an occam program may be compiled, the program fold must be filed, and then enclosed within a special fold called a `compilation` fold. Such an enclosure is termed a `foldset`. This compilation fold is created by using the [MAKE FOLDSET] function key. The main types of
compilation fold are

- **EXE** - for a program which executes within TDS i.e. on a single processor.

- **PROGRAM** - for a program which executes on a network of transputers. The fold contains the processes which execute on each transputer and configuration details. The configuration details for the particular network of transputers, discussed in Chapter 10, must be placed in the PROGRAM fold. TDS will check the validity of the PROGRAM fold, producing an inter-processor physical link connection list and loading the required SCs (see below) on to the relevant processors.

- **SC** - for a program which is part of another compilation unit. A set of processes to be allocated a processor must be encapsulated within an SC fold. Each collection of processes must be separately compiled (SC), then grouped together in a PROGRAM fold. One SC may be loaded on to more than one transputer. Such an SC may be a pipeline process, for example.

TDS treats an EXE compilation fold as if it is an occam procedure with a number of predefined channel parameters connecting it to the PC keyboard, screen and the DOS file system. These channels are automatically provided by TDS and do not have to specified within the occam program.

After the creation of the foldset, the occam program syntax may optionally be checked via the [CHECK] key press. Following a successful check, the program may be compiled (and linked) by pressing [COMPILE].

To sum up, the sequence of operations required to compile, link, load and execute an occam program are

- file occam program source fold

- load compiler utility

- create an EXE (or other suitable) compilation fold around the filed fold

- check the program syntax

- compile (and link) the program

- load the compiled program
• execute the compiled program

A.2.1 Example session

The last session ended with the creation of a filed program fold.

```occam
{[{ F EXAMPLE.TOP
    . . . F double
}]
}
```

Positioning the cursor on `double`'s fold line and pressing [MAKE FOLDSET] produces the message

```occam
{[{ Make foldset parameter
    VAL make.foldset.type IS SC : -- SC | EXE | UTIL | PROGRAM | LIB
}
```

This message is a prompt for the user to specify the type of foldset required. The default is SC, but may optionally be EXE, UTIL, PROGRAM or LIB. The default is changed by pressing [SELECT PARAMETER] until the required type, for example EXE, appears next to the IS reserved word. After selection of the foldset type, [EXIT FOLD] is typed to confirm creation of the required foldset.

For example,

```occam
{[{ F EXAMPLE.TOP
    . . . EXE double
}]
}
```

Note that the foldset is just an enclosing fold, and may be entered by positioning the cursor on the fold line and pressing [ENTER FOLD].

For example,

```occam
{[{ EXE double
    . . . F double
}]
}
```

shows that EXE foldset `double` contains the filed program fold, which in turn contains the occam program source text.
The program syntax is next checked by positioning the cursor on *double*'s EXE fold line and pressing [CHECK]. The screen displays a series of check options.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>error.checking</td>
<td>IS HALT: -- REDUCED</td>
</tr>
<tr>
<td>alias.checking</td>
<td>IS TRUE:</td>
</tr>
<tr>
<td>usage.checking</td>
<td>IS TRUE:</td>
</tr>
<tr>
<td>separate.vector.space</td>
<td>IS TRUE:</td>
</tr>
<tr>
<td>create.debugging.info</td>
<td>IS TRUE:</td>
</tr>
<tr>
<td>range.checking</td>
<td>IS TRUE:</td>
</tr>
<tr>
<td>compile.all</td>
<td>IS FALSE:</td>
</tr>
<tr>
<td>force.pop.up</td>
<td>IS FALSE:</td>
</tr>
<tr>
<td>use.standard.libs</td>
<td>IS TRUE:</td>
</tr>
<tr>
<td>target.processor</td>
<td>IS T4: -- T2</td>
</tr>
<tr>
<td>code.inserts</td>
<td>IS NONE: -- NONE</td>
</tr>
</tbody>
</table>

Usually the only default which may need changing is target.processor. This change is performed with the [SELECT PARAMETER] key press, after positioning the cursor on the relevant line. The checking may then be carried out by pressing [EXIT FOLD].

The occam program may now be compiled by placing the cursor on the EXE fold line and pressing [COMPILE]. If the compilation is successful, extra, new data folds are created within the foldset and the compiled code automatically linked. If the compilation is unsuccessful, a suitable error message is displayed. The occam source program must be edited to remove the error, and the check-compile cycle repeated.

For example, a successful compilation of the program *double* would produce the following sequence of messages

```
Compiling EXE double
Linking EXE double
Linked EXE double OK
```

and create the following folds

```
{
  EXE double
  ... F double
  ... F code
  ... F descriptor
  ... F link
  ... F debug
  ... F CODE EXE double
}
```

The program may be loaded and run by first moving the cursor to the EXE fold line and pressing [GET CODE], invoking the message
Got code ok
then by pressing [RUN EXE].

A.3 TDS libraries

TDS contains a number of procedure libraries which may be invoked from an occam program. Libraries exist for

- integer and floating point arithmetic
- bit manipulation
- trigonometric functions
- number to string (and vice versa) conversion
- string handling
- terminal and file store I/O

A few libraries are automatically referenced when a relevant procedure reference is made. However, most libraries have to be specifically loaded by inserting the #USE directive in the occam program. This has the format

#USE  library name

Probably the most immediately useful TDS library is the general purpose I/O procedure library, USERIO. This library contains procedures for the input of values from the keyboard and output of values to the screen, in a range of formats. To use any of the procedures within USERIO in an occam program, the statement

#USE USERIO

must be included at the top of the program.

TDS has defined a number of channels which are automatically available to an occam program. In particular, the keyboard and screen channels are defined as

CHAN OF INT keyboard :

CHAN OF ANY screen :

These channels may be used directly or via the I/O libraries provided with TDS. The channels
have a set of predefined protocols, which must be followed if the channels are used directly. The screen, for example, has a protocol which comprises a tag followed by data to move the cursor to a particular (x,y) screen position.

A selection of the I/O procedures is given below. The examples assume the use of standard TDS channels, *keyboard* and *screen*. (Note the lower-case channel names.)

Procedures for output include

- newline (screen) - writes a newline (carriage return and line feed) to the screen

- write.full.string (screen, text) - writes a string of characters, contained in *text*, to the screen

- write.len.string (screen, len, text) - writes *len* characters of a string, *text*, to the screen

- write.char (screen, character) - writes a character, *character*, to the screen

- write.int (screen, intnumber, field) - writes an integer, *intnumber*, to the screen in a specified field width, *field*.

- write.real32 (screen, realnumber, numberbefore, numberafter) - writes a real, *realnumber*, to the screen with *numberbefore* digits before the decimal point and *numberafter* digits after the decimal point.

where *character* is a BYTE literal or variable, *text* is a BYTE string literal or variable, *intnumber*, *field*, *numberbefore*, *numberafter* are INT literals or variables, and *realnumber* is a REAL32 literal or variable.

Procedures for input include

- read.char (keyboard, character) - reads a character from the keyboard into the variable *character*

- read.text.line (keyboard, len, line, character) - reads a string of characters in to the array *line*, up to and including a carriage return. The number of string characters read is given by *len*. The variable *character* will be set to INT '*c*' i.e. carriage return, or an error code if there was a read error.

- read.int (keyboard, intnumber, character) - reads an integer from the keyboard into the variable *intnumber*. The variable *character* must be initialised to the first character of the number. Afterwards, *character* will
contain the first non-numeric character read.

- read.real32 (keyboard, realnumber, character) - reads a real from the keyboard into the variable realnumber, with the first non-numeric character returned in the variable character.

where line is a BYTE array, character, intnumber, len are INT variables, and realnumber is a REAL32 variable.

A comparable set of input procedures exist which echo the input to the screen. The procedure names incorporate the word "echo" and an extra channel parameter for the screen is included. For example,

read.echo.char (keyboard, screen, character)

reads a character from the keyboard into the variable character, and echos it to the screen.

It is important to note that the keyboard channel is specified to deliver integer values of characters, and not byte values. Thus the occam program must be written either to convert input characters to byte values or to convert program characters to integer values. An example of this approach is given below. The example also illustrates the use of the USERIO procedures.

```
USE USERIO
[256] BYTE String :
VAL MaxLength IS 256 :
INT Length, Any :
BOOL EoS :
SEQ
  EoS := FALSE
  Length := 0
  write.full.string (screen, "Type in a number of characters terminated by a full-stop")
  newline (screen)
  WHILE (NOT EoS) AND (Length < MaxLength)
    -- integer declaration for character input
    INT Char :
    SEQ
      -- read integer character from the keyboard
      read.char (keyboard, Char)
  ```
IF
  -- test against integer
  Char = INT ' '
  EoS := TRUE
  TRUE
  SEQ
    -- convert to byte
    String [Length] := BYTE Char
    Length := Length + 1
    write.full.string (screen, "Characters typed were")
    newline (screen)
    SEQ  I = 0 FOR Length
      write.char (screen, String [I])
    newline (screen)
    -- hold the screen
    keyboard ? Any

The keyboard and screen channels may be used directly by the user, without recourse to the TDS library procedures. The example given above shows the keyboard being accessed directly in the last line. (This trick is required to hold the output on the screen, otherwise the screen would be cleared on return to TDS after the program completes execution.) The screen channel uses a tagged protocol which the user must implement if the channel is to be used directly. For example, the tagged protocol for outputting a string of characters is

    tt.out.string ; INT :: [ ] BYTE string

where tt.out.string is the tag, and has a numeric value of 8,
and the tagged protocol for moving the cursor to row x, column y of the screen is

    tt.goto ; INT x ; INT y

where tt.goto is the tag, and has a numeric value of 5.
A.4 Other features of TDS

Other features available in TDS include

- a DOS file store interface. Besides the *screen* and *keyboard* channels available to an EXE process, the channels *to.user.file* and *from.user.file* are also available for interfacing to the MS-DOS file store and handling folds.

- a debugger which allows post-mortem debugging of occam programs. It includes facilities for inspecting the values of program variables and the symbolic debugging of occam source.

- an alien file server (afs) which provides input and output streams and a file interface to a host (or alien) system independent of TDS. Such a service is needed when the occam program is run in standalone mode. A subset of the terminal protocols available in TDS is provided. The user must provide software to interface to these protocols.
References

Askew, Carpenter, Chalker, Hey and Nicole, "Simulation of Statistical Mechanical Systems on Transputer Arrays", Computer Physics Communications, 42, pp 21-26, 1986

Atkin, "Performance Maximisation", Technical Note 17, Inmos Ltd


Burns, "Programming in occam 2", Addison-Wesley, 1988

Dowsing, "Introduction to Concurrency using occam", Van Nostrand Reinhold, 1988


Hoare, "Communicating Sequential Processes", Prentice-Hall, 1985

Hockney and Jesshope, "Parallel Computers 2", Adam Hilger, 1988


Inmos, "Transputer Development System", Prentice-Hall, 1988

Jones and Goldsmith, "Programming in occam 2", Prentice-Hall, 1988


May and Shepherd, "Communicating Process Computers", Technical Note 22, Inmos Ltd

Packer, "Exploiting Concurrency : a Ray Tracing Example, Technical Note 7, Inmos Ltd


Peitgen and Richter, "The Beauty of Fractals", Springer-Verlag, 1987
Redfern, "Implementing data structures and recursion in occam", Technical Note 38, Inmos Ltd


Winder, "Parallel Processing with the Disputer", Proceedings of the 8th Technical Meeting of the Occam Users Group, pp 31-43, IOS, 1988
Further reading

Books

The definitive guide to occam 2. This text is the final arbiter for questions concerning occam 2 usage.

2) "Transputer Development System", Inmos Ltd., Prentice-Hall, 1988
The definitive guide to TDS. It contains full details of TDS.

A collection of some Inmos Technical Notes. Of particular interest are the sections dealing with real-time programming and exploiting concurrency.

A guide to the transputer architecture, with sections dealing with T800, T414 and T212 hardware.

A book dealing with occam 1; it has now been superseded by the Jones and Goldsmith book.

This book uses the data flow approach to developing occam programs. Various lengthy examples are given, including process control and data processing applications. The book is written for occam 1.

A very readable and easy-going introduction to occam 2.

8) "Programming in occam 2", A. Burns, Addison-Wesley, 1988
A good book on occam 2. It includes sections on the transputer, the transformation of occam programs and a comparison between occam and Ada.
9) "An Introduction to occam 2 Programming", K. Bowler, R. Kenway, G. Pawley and D. Roweth, Chartwell-Bratt, 1987
A good introductory book to occam 2. It contains interesting chapters dealing with configuration, parallel programs and a survey of parallel architectures.

A detailed book on programming in occam 2. It contains many lengthy examples of occam 2 programs including the Game of Life, process farming and Huffman encoding.

Another book written for occam 1. It contains many interesting examples, including simulating digital logic, data structures and the simulation of an operating system.

A very thorough theoretical treatment of CSP, the basis of occam. It is a must for any serious reader.

Papers

1) "Configuring occam programs", L. Pegrum, Technical Note 31, Inmos Ltd.
A guide to configuring occam programs on a network of transputers.

2) "Occam input and output procedures for the TDS", M. Poole, Technical Note 28, Inmos Ltd.
A guide to I/O library procedures provided for TDS.

3) "Program design for concurrent systems", P. Mattos, Technical Note 5, Inmos Ltd.
An introduction to programming software for parallel implementation on a network of transputers.

4) "Communicating process computers", D. May and R. Shepherd, Technical Note 22, Inmos Ltd.
A description of approaches to parallel programming.

5) "Exploiting concurrency: a ray tracing example", J. Packer, Technical Note 7, Inmos Ltd.
An example of the use of a processor farm.

6) "Getting started with the TDS", D. Cormie, Technical Note 3, Inmos Ltd.
A simple introduction to TDS.

7) "Occam program development using the IMS D700C transputer development system", M. Poole, Technical Note 16, Inmos Ltd.
A general description of TDS.

8) "Performance maximisation", P. Atkin, Technical Note 17, Inmos Ltd.
Hints on how to improve occam performance.

9) "Communicating processes and occam", D. May, Technical Note 20, Inmos Ltd.
A general discussion of the relationship of communicating processes with occam.

A presentation of occam 1, with examples of its use in concurrent systems and program transformations.

An introduction to occam 1, concentrating on the use of occam in parallel systems.

12) "occam 2 : an overview from a software engineering perspective", R. Wayman, Microprocessors and Microsystems, vol. 11. no. 8, pp. 413 - 422, 1987
A review of occam 2 and how it relates to various stages of the software development process.

13) "The transputer", P. Walker, Byte, pp. 219 - 235, May 1985
An introduction to the transputer, some of its uses and its relation to occam.

An in-depth exposition of the T800 transputer.

15) "T800 and Counting", R. Stein, Byte, pp. 287 - 296, November 1988
A readable account of the occam and transputer philosophy.

Group technical monograph, PRG-53, Oxford

An exposition of the semantics of occam, which relates the various program transformations which are possible in occam.

A discussion of input and output guards in occam.


An exposition of problems in the implementation of the PRI ALT construction and suggestions on how to make best use of the construction.

An interesting description of the emulation of digital circuits using occam, and the problems which may arise.

An exploration of methods for extracting parallelism, together with an investigation of communications overheads and load balancing.

Geometric and algorithmic concurrency approaches are applied to a simple statistical physics model.

The algorithmic, geometric and process farming approaches to concurrency are analysed in terms of performance.

In addition, there are a number of papers on research and applications with occam and the transputer in the Occam User Group (OUG) Conference Proceedings 7 (September 1987), 8 (March 1988) and 9 (September 1988), published by IOS, which are worthwhile reading.
Answers to exercises

Chapter 1

1.

PAR
  Chan ! Limit - Index
  Chan ? Left

2.

a) Invalid. Variable Item is being used in two components of a parallel process without being communicated via a channel.

b) Invalid. Same channel is being used for both input and output in same process.

c) Invalid. Deadlock will occur - first SEQ process will wait for second SEQ process to input, whilst second SEQ process will wait for first SEQ process to input.

d) Valid. No deadlock occurs as component processes are executing in parallel not in sequence.

e) Invalid. Deadlock will occur - first process will wait for second process to output on channel Chan1, whilst second process will wait for first process to input on channel Chan2.

3.

a)

PAR
  SEQ
    Chan1 ? A
    Chan2 ? B
  SEQ
    Chan1 ! D
    Chan2 ! C
b)  

```
PAR
  PAR
    Chan1 ? A
    Chan2 ? B
SEQ
    Chan2 ! C
    Chan1 ! D
```

(Either or both SEQs may be replaced by a PAR if the inputs and outputs are independent.)

Chapter 2

1.
   a) Valid.
   
   b) Invalid - occam reserved word.
   
   c) Valid.
   
   d) Invalid - space not permitted in identifier.
   
   e) Invalid - underscore not permitted in identifier.
   
2.
   a) Valid. Variable Limit re-declared as type BYTE.
   
   b) Invalid. Assignment should be Small := 55 (BYTE)
   
   c) Invalid. BYTE values are valid only in the range 0 - 255.
   
   d) Valid.
   
   e) Invalid. Variables INDEX and Index are not the same.
   
   f) Invalid. Variable Char of second process not declared. Char is only in scope of first sequential process.
   
   g) Invalid. Channel protocol is specified as single byte, but input process variable is specified as an integer.
   
   h) Valid. Identifiers Item as type INT and Item as type BYTE have separate scope and use
different channels for input/output.

i) Valid. Second declaration of identifier Item as BYTE supersedes the first declaration in scope.

j) Invalid. Value of variable Count not defined.

k) Valid. Variable Count is in scope of variable Index.

l) Invalid. Variable Count is out of scope.

Chapter 3

1.
   a) Invalid. Needs precedence brackets, for example 100 - (5 * 5)

b) Invalid. Precedence brackets wrongly placed, for example 100 - ((5 * 5) + 20)

c) Invalid. Needs precedence brackets, for example (100 - (5 * 5)) > Limit

d) Invalid. Mixed arithmetic and boolean types.

e) First assignment is invalid. Arithmetic overflow - result too large.
   Second assignment is invalid. Mixed arithmetic types. A valid statement is
   Result := 33 (INT16) * 10 (INT16)
   Third assignment is invalid. Arithmetic overflow - result too large.

2.
   a) FALSE

   b) TRUE

   c) BooleanValue

If TRUE is replaced by FALSE, then
   a) TRUE

   b) BooleanValue

   c) FALSE
3.
   a) 1
   b) 0
   c) FALSE
   d) 3
   e) 3
   f) 3.0
   g) 3.0

4.

    VAL BitNumber IS 4 :
    INT Number :
    SEQ
       Number := 3    -- for example
       Number := Number BITOR (1 << BitNumber)

5. An AND gate is

    CHAN OF BOOL InPin1, InPin2, OutPin :
    BOOL Level1, Level2 :
    SEQ
       PAR
          InPin1 ? Level1
          InPin2 ? Level2
          OutPin ! Level1 AND Level2

Similarly for the OR and NOR gates. (The example may also be expressed using integer types, together with the appropriate channel specifications and bitwise operators.)

6.

    VAL Number IS 123 :
    INT Hundreds, Tens, Units :
    SEQ
       Hundreds := Number / 100
       Tens := (Number REM 100) / 10
       Units := Number REM 10
7.

CHAN OF INT InPin1, InPin2, OutPin :
INT A, B, Part1, Part2 :
SEQ
   PAR
      InPin1 ? A
      InPin2 ? B
      Part1 := (BITNOT A) BITAND B
      Part2 := A BITAND (BITNOT B)
   OutPin ! Part1 BITOR Part2

(The example may also be expressed using boolean types, together with the appropriate channel specifications and boolean operators.)

Chapter 4

1.

a)
[3] BYTE TrafficLights :
b)
[8] [8] INT ChessBoard :
c)
[66] [132] BYTE PrinterPaper :
d)
[15] [15] BYTE ScrabbleBoard :
e)
[12] [3] BYTE Months :

2. The first specification is an array of 8 channels, each capable of transmitting a single REAL64 value; whilst the second is a single channel capable of transmitting an array of 8 REAL64 values.
3.

CHAN OF [25] REAL32 Chan:
PAR
  [25] REAL32 X:
  SEQ
    Chan ! X
  [25] REAL32 Y:
  SEQ
    Chan ? Y

4.

a)
ScrabbleBoard [7] [7]
(Rows and columns take values between 0 and 14.)

b)

5. Num1 is assigned a value 5, Num2 is assigned a value 10, and Num3 is assigned a value 15.

6.
a) Invalid. Mixed data types.

b) Invalid. First component of table is primitive data type, whilst second component is another table.

c) Invalid. Strings are unequal lengths.
Chapter 5

1.

IF
   Data < 0
   .
   -- some process
   .
   Data = 0
   .
   -- some process
   .
   Data > 0
   SKIP

2.

BOOL Running
IF
   Running
   .
   .
   .
   NOT Running
   SKIP

3.

INT Int1, Int2, Min, Max :
SEQ
   .
   -- assign values to Int1 and Int2
   .
   IF
      Int1 >= Int2
      Max, Min := Int1, Int2
   Int1 < Int2
      Min, Max := Int1, Int2
4.

[10] BYTE Digits : -- assume maximum of ten digits
INT Index, Number :
SEQ
Number := 12345 -- for example
Index := 9 -- assume maximum of ten digits
-- store digits in array in reverse order, ready for output
WHILE Number <> 0
   SEQ
      Digits [Index] := BYTE ((Number REM 10) + INT '0')
      Number := Number / 10
      Index := Index - 1
   -- fill in any leading spaces
WHILE Index >= 0
   SEQ
      Digits [Index] := ' '
      Index := Index - 1
   -- output digits
-- (use replicated SEQ construction - see Chapter 6)
SEQ Count = 0 FOR Index
   Output ! Digits [Count]

5.

a)

CHAN OF BYTE Chan, In, Out :
PAR
WHILE TRUE
   BYTE Char :
   SEQ
      In ? Char
      Chan ! Char
WHILE TRUE
   BYTE Char
   SEQ
      Chan ? Char
      Out ! Char
b)

CHAN OF BYTE In, Out:
BYTE Char1, Char2:
SEQ
  In ? Char1
WHILE TRUE
  SEQ
    PAR
      In ? Char2
      Out ! Char1
    PAR
      In ? Char1
      Out ! Char2

6.

CASE Month
  February
  IF
    (Year REM 4) = 0
    Days := 29
  TRUE
    Days := 28
  April, June, September, November
  Days := 30
ELSE
  Days := 31

7.

VAL Message1 IS "a or A typed" :
VAL Message2 IS "e or E typed" :
VAL Message3 IS "i or I typed" :
VAL Message4 IS "o or O typed" :
VAL Message5 IS "u or U typed" :
BYTE Vowel :
SEQ
  Input ? Vowel
  CASE Vowel
    'a', 'A'
      Output ! Message1
    'e', 'E'
      Output ! Message2
'i', 'I'
  Output ! Message3
'o', 'O'
  Output ! Message4
'u', 'U'
  Output ! Message5

8.

WHILE TRUE
  BOOL Flag :
  BYTE Char :
  SEQ
    Flag := TRUE
  ALT
    Flag & InChan1 ? Char
    SEQ
      .
        -- perform process
      .
        -- switch flag value
        Flag := FALSE
    NOT Flag & InChan2 ? Char
    SEQ
      .
        -- perform process
      .
        -- switch flag value
        Flag := TRUE

Chapter 6

1.

SEQ Index = 0 FOR 20
  Screen ! '***'
2.

VAL Alphabet IS "ABCDEFGHIJKLMNOPQRSTUVWXYZ" :
SEQ Index = 0 FOR 26
    Display ! Alphabet [Index]

3.

VAL Char IS 'm' :      -- for example
VAL String IS "example" :  -- for example
INT Position :
IF
    IF Index = 0 FOR SIZE String
        String [Index] = Char
        Position := Index
    TRUE
        Position := -1

4.

VAL StringSize IS 80 :      -- for example
INT Order :           -- variable for setting
[StringSize] BYTE String1, String2 :
SEQ
    IF
        IF Index = 0 FOR StringSize
            String1 [Index] <> String2 [Index]
            IF
                String1 [Index] > String2 [Index]
                Order := 1
            TRUE
                Order := -1
        TRUE
            Order := 0
VAL N IS 8 : -- for example
  [N] INT A, B, C :
  SEQ Row = 0 FOR N
  SEQ Col = 0 FOR N
  SEQ
  C [Row, Col] := 0
  SEQ Index = 0 FOR N
  C [Row, Col] := C [Row, Col] + A [Row, Index] * B [Index, Col]

(A more efficient solution would be to have an abbreviation for $C [Row, Col]$ as follows)

VAL N IS 8 : -- for example
  [N] INT A, B, C :
  SEQ Row = 0 FOR N
  SEQ Col = 0 FOR N
  Sum IS C [Row, Col] :
  SEQ
  Sum := 0
  SEQ Index = 0 FOR N
    Sum := Sum + A [Row, Index] * B [Index, Col]

)

6.
Pipe

PAR Index = 0 FOR 1024
  BOOL Terminate :
  SEQ
    Terminate := FALSE
    WHILE NOT Terminate
      INT Item :
      SEQ
        Buffer [Index] ? Item
        IF
          Item = Eof
            Terminate := TRUE
        TRUE
        SKIP
        Buffer [Index + 1] ! Item
Termination

**BOOL** Terminate :

**SEQ**

Terminate := FALSE

**WHILE NOT** Terminate

**INT** Data

**SEQ**

Buffer [1024] ? Data

**IF**

Data = Eof

Terminate := TRUE

**TRUE**

**SKIP**

-- consume or output an item of data

.

7.

**VAL** N **IS** 10 : -- for example

[N + 1] **CHAN** OF **INT** Pipe :

**PAR** Index = 0 **FOR** N

**INT** Number, LastPower, NextPower

**SEQ**

Pipe [Index] ? Number

Pipe [Index] ? LastPower

NextPower := LastPower * Number

Pipe [Index + 1] ! Number

Pipe [Index + 1] ! NextPower

8.

**VAL** Chan1 **IS** 1 :

**VAL** Chan2 **IS** 2 :

**VAL** Chan2 **IS** 3 :

**VAL** Chan2 **IS** 4 :

**INT** Data1, Data2, Data3, Data4 :

**ALT**

InChan1 ? Data1

**SEQ**

OutChan ! Chan1

OutChan ! Data1
InChan2 ? Data2
    SEQ
    OutChan ! Chan2
    OutChan ! Data2
InChan3 ? Data3
    SEQ
    OutChan ! Chan3
    OutChan ! Data3
InChan4 ? Data4
    SEQ
    OutChan ! Chan4
    OutChan ! Data4

where the channels InChan1, InChan2, InChan3, InChan4 and OutChan are specified as type INT.

9.

VAL QSize IS 20: -- for example
ALT
    (Count < QSize) & Enqueue ? Item
    -- enqueue item
    SEQ
    Queue [Head] := Item
    Head := (Head + 1) REM Size
    Count := Count + 1
    (Count > 0) & Dequeue ? Any
    -- dequeue item
    SEQ
    Item := Queue [Tail]
    Tail := (Tail + 1) REM Size
    Count := Count - 1
    Result ! Item

where Queue is declared as

[QSize] INT Queue :

and the global variables, Head, Tail and Count, are all initialised to 0. Channel Enqueue is used to enqueue an item provided the queue is not full. Channel Dequeue is used to request an item be dequeued provided the queue is not empty; the item being returned via channel Result.
Chapter 7

1. 
   a) 
   
   \[ \text{VAL} \text{ DegPerRad IS 3.142 (REAL32) / 180.0 (REAL32)} : \]

   b) 

   \[ \text{VAL} \text{ MinutesInDay IS 60 * 24 :} \]

   c) 

   \[ \text{VAL} \text{ MphToKmph IS 80.0 (REAL32) / 50.0 (REAL32)} : \]

2. 
   a), b), c) Valid. All three alternatives are equally acceptable.

3. 

   \[ \text{PROC} \text{ Swap (VAL INT Number, INT Reverse)} \]
   \[ \text{SEQ} \]
   
   Hundreds := Number / 100
   Tens := (Number REM 100) / 10
   Units := Number REM 10
   Reverse := (100 * Units) + ((10 * Tens) + Hundreds)

4. 

   \[ \text{PROC} \text{ CreateStack ()} \]
   \[ \text{SEQ} \]
   
   Top := -1

   \[ \text{PROC} \text{ Push (VAL INT Item)} \]
   \[ \text{SEQ} \]
   
   Top := Top + 1
   Stack [Top] := Item
PROC Pop (INT Item)  
   SEQ  
      Item := Stack [Top]  
      Top := Top - 1

:  

BOOL FUNCTION StackEmpty () IS Top = -1 :

BOOL FUNCTION StackFull () IS Top = (StackSize - 1) :

where Stack is declared as

[StackSize] INT Stack :

and variable Top is global.

5.

PROC Assemble ([ ] BYTE String, INT Length)  
   VAL EndOfString IS ' ' :  
   VAL MaxLength IS 80 :  
   BOOL EoS :  
   SEQ  
      EoS := FALSE  
      Length := 0  
      WHILE (NOT EoS) AND (Length < MaxLength)  
         BYTE Char :  
         SEQ  
            InChan ? Char  
            IF  
               Char = EndOfString  
               EoS := TRUE  
               TRUE  
               SEQ  
                  String [Length] := Char  
                  Length := Length + 1  
               :  
            :  
         :  
      :
6.

```plaintext
INT FUNCTION MinOfThree (VAL INT Int1, Int2, Int3)
  INT Minimum :
  VAL OF
  IF
    Int1 <= Int2
    Minimum := Int1
  TRUE
    Minimum := Int2
  IF
    Int3 < Minimum
    Minimum := Int3
  TRUE
  SKIP
  RESULT Minimum
:
```

7.

```plaintext
BOOL FUNCTION IsaPalindrome (VAL [ ] BYTE Word)
  INT I, J :
  BOOL Palindrome :
  VAL OF
  SEQ
    I := SIZE Word
    J := 0
    WHILE (J < I) AND (Word [J] = Word [I])
    SEQ
      I := I - 1
      J := J + 1
  IF
    J >= I
    Palindrome := TRUE
  TRUE
    Palindrome := FALSE
  RESULT Palindrome
:
```

8.

```plaintext
REAL32 FUNCTION ToKmph (VAL REAL32 Mph) IS
  Mph * (80.0 (REAL32) / 50.0 (REAL32)) :
```
9.

REAL32 FUNCTION ToCelsius (VAL REAL32 Fahrenheit) IS
(Fahrenheit - 32.0 (REAL32)) * (5.0 (REAL32) / 9.0 (REAL32)) :

10.

BOOL FUNCTION IsAVowel (VAL BYTE Char)

BOOL Vowel :
VALOF
IF
(Char = 'a') OR (Char = 'A') OR
(Char = 'e') OR (Char = 'E') OR
(Char = 'i') OR (Char = 'I') OR
(Char = 'o') OR (Char = 'O') OR
(Char = 'u') OR (Char = 'U') OR
Vowel := TRUE
TRUE
Vowel := FALSE
RESULT Vowel :

11.

REAL32, REAL32 FUNCTION Stats (VAL [ ] REAL32 List)

REAL32 Mean, StdDev, Sum, SumSq, Size :
VALOF
SEQ
Sum := 0.0 (REAL32)
SumSq := 0.0 (REAL32)
SEQ Index = 0 FOR SIZE List
SEQ
Sum := Sum + List [Index]
SumSq := SumSq + (List [Index] * List [Index])
Size := REAL32 ROUND (SIZE List)
Mean := Sum / Size
StdDev := (SumSq / Size) - (Mean * Mean)
RESULT Mean, StdDev :
Chapter 8

1.

a) \[ \text{CHAN OF INT :: [ ] REAL32 Chan :} \]

b) \[ \text{CHAN OF [16] [16] INT Chan :} \]

c) \[ \text{CHAN OF INT :: [ ] BYTE Chan :} \]

2.

a) \[ \text{PROTOCOL COMPLEX IS REAL32 ; REAL32 :} \]

b) \[ \text{PROTOCOL ARRAYCOMPONENT IS INT ; REAL32 :} \]

(component index followed by component value)

c) \[ \text{PROTOCOL PERSONAL IS BYTE :: [ ] BYTE ; INT ; BYTE} \]

(number of characters in name followed by name, age and sex)

3.

a) First message is invalid - too long. Second message is valid.

b) Invalid - size of receiving array, *Block*, is too small.

4.

\[ \text{PROTOCOL MUXDEMUX IS INT ; REAL32 :} \]

Multiplexor

\[ \text{ALT Channel = 0 FOR Number} \]

\[ \text{In [Channel] ? Data} \]

\[ \text{Out ! Channel ; Data} \]
Demultiplexor

SEQ
 In ? Destination ; Data
 Out [Destination] ! Data

5.

PROTOCOL PeekAndPoke
 CASE
 -- peek a memory address (supplied as an integer)
   peek ; INT
 -- data resulting from peek request (assumed to be an integer)
   result ; INT
 -- poke a memory address with data (both assumed to be integers)
   poke ; INT ; INT

Chapter 9

1.
 a)

 VAL TicksPerSecond IS 15625 :
 VAL TwoSeconds IS 2 * TicksPerSecond :
 TIMER Clock :
 INT Now :
 SEQ
   Clock ? Now
   Clock ? AFTER (Now PLUS TwoSeconds)

 b)

 VAL TicksPerSecond IS 1000000 :
 VAL TwoSeconds IS 2 * TicksPerSecond :
 TIMER Clock :
 INT Now :
 PRI PAR
   SEQ
     Clock ? Now
     Clock ? AFTER (Now PLUS TwoSeconds)
   SKIP
2.

PRI PAR
   -- router process
SEQ
   InChan ? Destination ; Data
   IF
      Destination = Me
      Internal ! Data
   TRUE
      OutChan ! Destination ; Data
SEQ
   .
   .  -- internal process
   .

3.

VAL INT NumberOfSkips IS 1000 :
TIMER Clock :
INT Before, Now, TimeDifference, TimeOfSkip :
SEQ
   Clock ? Before
   SEQ Index = 0 FOR NumberOfSkips
      SKIP
      Clock ? Now
      TimeDifference := Now MINUS Before
      TimeOfSkip := TimeDifference / NumberOfSkips

(Assuming no other processes are running on the transputer and ignoring the overhead of the replication.)

4.

TIMER Clock :
INT Start, End, Time :
SEQ
   Clock ? Start
   .
   .  -- program
   .
   Clock ? End
   Time := End MINUS Start
Index

abbreviation 42,126
element 126,130
expression 126
addition 56
AFTER 170
algorithmic parallelism 191,192,224
alien file server 241
ALT 11,21,27,91,92,117,163,170
alternation construction 21,91
alternative 21,91
anarchic 165
AND 66
ANY 165
arithmetic operator 55
array 74,84,151,168
corponent 74,79,80,82,86,131,152
multi-dimensional 78,81
one-dimensional 74
segment 82,121
size 74,75,85
subscript 79
type 75,76
whole 74,76
assignment
multiple 13
process 11,12,145
AT 175

BITAND 68
BITNOT 68
BITOR 68
BOOL 35
boolean
guard 94
expression 94,96,103,163
type 35
buffer 113
BYTE 35
CASE 100,102,159,161
case 100,102
case input 161,163
CHAN OF 47,151,152,156,159,165
channel 5,11,14,15,182
hard 182
protocol 150
simple 46,150
soft 182
type 32,74
choice 96,98
comment 29
communication link 4,14,182
compile fold 233
compile 232,233
component 74,79,80,82,86,131,132
concurrent
applications 3
execution 4,180
process 7
concurrency 3
conditional construction 91,96
configuration 6,180
constant 42
construction 11,91
ALT 21,91,92,163
alternation 21,91
CASE 100
conditional 91,96
IF 96
nested 23
PAR 11,19
parallel 19,191
repetition 91,103
selection 91,100
SEQ 11,16
sequential 16
WHILE 103
continuation line 12
conversion 60,134
crease 227
CSP 7,11,192
data
conversion 60,134
initialisation 128
structures 121
type 13,15,32,33,43,45,47,126,143,151,177
data flow decomposition 192
data structure decomposition 197
deadlock 26
declaration 32,42
   channel 150
   constant 42
   variable 45
delay 170
dyadic 55
editor 225,227,231
ELSE 102
expression 12,15,127
   arithmetic 55
   boolean 64,94
   case 100
   constant 43
FALSE 66
farmer process 210
filed fold 230
floating point 5
fold 227
foldset 233
fold line 227
folding editor 225,227
FOR 82,109,110,112,117,120
fractals 211
free variable 51
FROM 82
FUNCTION 143,146,147
function 143,145
   definition 147
geometric parallelism 192,197
granularity 191
   coarse-grained 191
   fine-grained 191
guard 21,91,96
   boolean 91,94,117
   input 21,91,117
   SKIP 91,117
HALT 28
hard channel 182
Hoare 7
identifier 32,33,43,45,47,74,109,126,143,151,168,175,177
IF 96,120
indefinite postponement 91
indentation 12
input
   process 11,14,15,153
   guard 21,91,117
instance 136,140,145
INT 35,44,74,82,85,109,175
INT16 35,61
INT32 35,61
INT64 35
integer type 35
IS 42,126,128,130,132,147,155,156
library 237
link 4,14,182,223
literal 36
   boolean 41
   character 39
   integer 36
   real 38
   string 41
local 32,42
   scope 49
   logic gate 67,69
Mandelbrot set 210
mask 69
memory-mapping 174
message routing 191,223
MINUS 58
modulo arithmetic 58,170
monadic 55
MOSTNEG 59
MOSTPOS 59
multiple assignment 13
multiplication 56
multiplexer 23,117,174
nested
   alternation 119
   conditional 98
   construction 102
network 6,8,180,181
Newton-Raphson estimation 193
non-printable character 39
NOT 66

occam 3,7,8,11,191
one-way communication 14
operand 55
operator 55
  addition 56
  arithmetic 55
  bitwise 68
  bitwise and 68
  bitwise exclusive or 68
  bitwise not 68
  bitwise or 68
  boolean 66
  boolean and 66
  boolean not 66
  boolean or 66
  division 56
  equal to 64
  greater than 64
  less than 64
  multiplication 56
  not equal to 64
  precedence 59
  relational 64
  remainder 56
  shift 70
  shift left 71
  shift right 72
  subtraction 56
option 100
OR 66
output process 11,14,153

PAR 11,19,26,48,82,112,114
  PLACED 181,183
  PRI 171,172
parallel
  architecture 1
  computation 1,191
  construction 16,191
  execution 4
  process 8,11
parameter 138,143
  actual 140
  formal 38
pipeline 112,116,193,194
PLACE 175

placement 175
PLACED PAR 181,183
PLUS 58
PORT OF 177
port 177
precedence brackets 56,59
PRI
  ALT 93,95,120,171,173
  PAR 171,172
primitive
  data type 33,126,143
  process 11,12,28
priority
  high 168
  level 168,171
  low 168
PROC 135,138
procedure 135,136,138
process 135,180
  concurrent
  model 11
  parallel 8,11,19
  priority 4
  primitive 11,28
  sequential 11,16
process farming 192,210
PROCESSOR 181,183
processor 1,180
  connectivity 223
  loading 223
  type 4,224
PROTOCOL 155,156,159
protocol
  counted array 152
  name 155
  sequential 156
  simple 150,156
  special 165
  tagged 158
  variant 158
real type 35
real-time
  clock 168
  programming 168
REAL32 35,63
REAL64 35,63
REDUCED 28
relational operator 64
REM 56
remainder 56
replicate 109,171,181
ALT 109,117
IF 109,120
PAR 109,112,124,195,208
SEQ 109,110
replication 109,198
count 109
index 109
RESULT 143,146
reserved word 32
RETYPES 134
retyping 134
ROUND 62
rounding 62

scheduler 4
scope 49,126
segment 82,121
selector 100
SEQ 11,16,27,110
sequential
    architecture 1
    process 7,16
shift 70
SIZE 85,141
SKIP 11,16,20,28,95,97,109
skip guard 95,117
soft channel 182
special channel 165,168
STOP 11,28,97,100,109
STOP 28
stopped 28
string 41,84
strongly typed 32
structured type 74
subscript 74
subtraction 56
synchronisation 3,7,14
systolic array 193,197

T212 4,176,182
T414 4,176,182
T800 4,176,182
table 86
tag 158,161
tagged protocol 158,161
terminated 28
termination notice 29,115
thermal conduction 197
timeout 170
TIMER 168
timer 6,168
type 32,74,168
TIMES 58
Transputer Development System 3,165,175
182,225
transputer 3,168,171,174,180,182
TRUE 66,97,103,105
TRUNC 62
two-way communication 25
type 32,35
type conversion 55

USE 237,239
USERIO 237,239

VAL 42,126,128,134,138,140,143,182
VALOF 143,146
variable 12,15,42,45
    free 51
von Neumann 1

WHILE 103
William of Ockham 3
worker process 210