4.5 Abbreviation of variables

A variable abbreviation specifies a new name for a variable. Consider

```
INT n IS m :
```

This abbreviation specifies the name **n** as the new name for **m**. Also, consider the following example:

```
INT user IS lines[8] :
```

This abbreviation specifies the name **user** for a component of the array **lines**. All subscript expressions used in an abbreviation must be valid. The type of the abbreviated variable must be the same as the data type specified, so in this example, **lines** has to be an array of **INT**. Other components of the array **lines** may be used only in abbreviations within the scope (page 39) of **user**, but they must not include the component **lines[8]**. Here are some more examples of abbreviations:

x IS y :	specifies a new name **x** for **y**
INT c IS a[i] :	specifies a name for a component of the array **a**
[]REAL32 s IS [a FROM 8 FOR n] :	specifies a name for a segment of **a**
[]BYTE fname IS [fname FOR len] :	specifies the used part of an array

An abbreviation simply provides a name to identify an existing variable. The name **c** in the above example identifies the existing variable **a[i]**. In the scope of the abbreviation, **c := e** is an assignment to the original variable **a[i]**. A variable used in a subscript to select a component or components of an array may not be assigned to within the *scope* of the abbreviation. For example, no assignment or input can be made to **i** within the scope of **c**. As a result the abbreviation always refers to the same variable throughout its scope. This allows various optimisations to be performed, such as evaluating any expression within the abbreviated variable only once. The original variable **a[i]** may not be used within the scope of the abbreviation **c**.

The abbreviation of **fname** as a leading sub-array of a previously declared array of the same name is a common device for allocating an array whose size is the upper bound of possible sizes and then only using part of it in a particular scope.

The specifier can usually be omitted from an abbreviation, as the type can be inferred from the type of the variable. A specifier **[]** *type* with one or more empty dimensions defines the abbreviation as being any array with components of the specified type. The type of an array on the right hand side of an abbreviation whose specifier has an empty dimension is said to be *compatible* with the specifier.

Where an abbreviation is of a component of an array no other reference may be made to any other part of that array, except in a further abbreviation. Consider the following example:

```
[60][72]INT page :
...
  first.line IS page[0] :
  last.line  IS page[59] :
  SEQ
    first.line := last.line
    last.line  := page[58]          -- This assignment is INVALID!
    ...
    next.to.last.line IS page[58] : -- This abbreviation is valid
    last.line  := next.to.last.line -- and so too, this assignment
    ...
```

Also consider the following example:

```
WHILE i < limit
  this.line IS page[i] :
  next.line IS page[i+1] :
  SEQ
    this.line := next.line
    ...
    i := i + 1                -- this assignment is INVALID!
```

The assignment in the above example is invalid as **i** is used to select components of the array **page** in an abbreviation within the scope of the assignment. This is how the above should be written:

```
WHILE i < limit
  SEQ
    this.line IS page[i] :
    next.line IS page[i+1] :
    SEQ
      this.line := next.line
      ...
    i := i + 1
```

It is important to ensure that all the components of an array remain identified by a single name within any given *scope*. Identification of any component of an array by more than one name constitutes an invalid usage of the component, and it is especially important to be aware of this of when abbreviating components of an array. Once any component of an array is abbreviated then reference to other components of the array must be made by further abbreviation. Checks are made to ensure that two abbreviations which identify segments from the same array do not overlap. When necessary a compiler may generate code to perform this check at run time. Further discussion on abbreviation is given in the chapter on procedures (page 69).

The syntax for abbreviations of variables is:

abbreviation	=	*specifier name* **IS** *variable* **:**
	|	*name* **IS** *variable* **:**
specifier	=	*data.type*
	|	**[]** *specifier*
	|	**[** *expression* **]** *specifier*

The abbreviation of a variable begins with an optional specifier, specifying a data type. The name specified appears to the right of the optional specifier followed by the keyword **IS**, the abbreviated variable appears to the right of the keyword **IS**. The line on which the abbreviation occurs may be broken after the keyword **IS** or at some legal point in the variable. The type of the variable must be the same as the data type specified.

4.6 Abbreviation of values

The last section described variable abbreviations. This section describes abbreviations of values. Consider the example:

```
VAL INT days.in.week IS 7 :
```

This abbreviation specifies the name **days.in.week** for the value 7. This construct is always used in occam for named constants. Here are some more abbreviations for values:

VAL REAL32 y IS (m * x) + c :	specifies a name for the current value of an expression
VAL INT n IS m :	specifies a name for the current value of the variable **m**
VAL []BYTE vowels IS ['a', 'e', 'i', 'o', 'u'] :	specifies a name for a table of values

The abbreviated value must be a valid expression, so it must not overflow, and all subscripts must be in range. Variables used in an abbreviated expression may not receive new values by input or assignment within the *scope* (page 39) of the abbreviation. This ensures that the value of the expression remains constant for the scope of the abbreviation. For example, in the following abbreviation

```
VAL REAL32 y IS (m * x) + c :
```

no assignment or input may be made to **m**, **x**, or **c** within the scope of this abbreviation defining **y**. The effect of the abbreviation may be obtained by replacing each occurrence of **y** in its scope by the abbreviated value **((m * x) + c)**. Similarly for the following abbreviation of the value **[screen FROM line FOR length]**

```
VAL []INT scan IS [screen FROM line FOR length] :
```

no assignment or input may be made to **screen**, **line** or **length** within the scope of **scan**. The effect of the abbreviation is the same as each instance of **scan** being replaced by the abbreviated value, thus

```
VAL []INT scan IS [screen FROM line FOR length] :
SEQ
  row := scan
  ...
```

is equivalent to

```
SEQ
  row := [screen FROM line FOR length]
  VAL []INT scan IS [screen FROM line FOR length] :
  ...
```

The syntax for abbreviations of values is:

abbreviation	=	**VAL** *specifier name* **IS** *expression* **:**
	|	**VAL** *name* **IS** *expression* **:**

The abbreviation of a value begins with the keyword **VAL**. An optional specifier (which specifies the data type of the abbreviation) appears to the right of **VAL**, followed by the name, and the keyword **IS**. The abbreviated value appears to the right of the keyword **IS**. Line breaks are permitted after the keyword **IS**. The type of the value must be compatible with the specifier. The specifier can usually be omitted from the abbreviation, as the type can be inferred from the type of the value. An important exception to this is if the value is an undecorated literal constant. The following will be treated as equivalent by the compiler:

```
VAL REAL32 pi IS 3.14159(REAL32) :
VAL pi IS 3.14159(REAL32) :
VAL REAL32 pi IS 3.14159 :
```

The choice between these is a matter for individual preference.

4.7 Disjoint arrays in parallels

Abbreviations may be used to decompose an array into a number of disjoint parts, so that each part may have a unique name in all or several processes in parallel. Components of each disjoint part may then be selected by a variable subscript (a subscript whose value is dependent on a procedure parameter, a variable, or a replicator index whose base or count is not a constant value), for example:

```
frame1 IS [page FROM 0 FOR 512] :
frame2 IS [page FROM 512 FOR 512] :
PAR
  INT i :
  SEQ
    ...
    c1 ? frame1[i]
    ...
  INT j :
  SEQ
    ...
    c2 ? frame2[j]
    ...
```

This example divides the array **page** into two parts, and provides a name for those parts in each of the two parallel processes. These parts may then be selected by using variable subscripts.

5 Channels and their protocols

occam programs act upon *variables*, and communicate using *channels* and *timers*. A variable has a value, and may receive a new value in an *assignment* or *input*. Channels communicate values. Timers produce a value which represents time.

This chapter describes communication channels, their declaration, the specification of the format and data type of communications, and the construction of arrays of channels.

Variables (page 25) and timers (page 81) are discussed elsewhere in the manual.

A Communication channel provides unbuffered, unidirectional point-to-point communication of values between two concurrent processes, which are components of a parallel or of constructions which are themselves components of a parallel. The format and type of values passed on a channel is specified by the channel *protocol*. The name and protocol of a channel are specified in a channel declaration.

5.1 Channel type

The type of a channel is:

channel.type = **CHAN OF** *protocol*

The keyword **CHAN** is always followed by the keyword **OF** which is followed by a protocol according to syntax elaborated below (page 47). Channel types cannot be named, but protocols can.

5.2 Declaring a channel

A channel is declared in the same way as variables are declared. Consider the following example:

```
CHAN OF BYTE screen :
```

This declaration introduces a channel named **screen** with a protocol of type **BYTE**. The protocol in this example specifies that each communication on this channel must be a value of type **BYTE**. An output on this channel could be:

```
screen ! 'H'
```

Several channels with the same protocol can be declared together, for example:

```
CHAN OF BYTE screen, keyboard :
CHAN OF INT from.a, from.b, from.c :
```

The type of the declarations is determined, and then the declarations are made.

The syntax of channel declarations is:

declaration = *channel.type* $\{_1$ **,** *name* $\}$ **:**

channel = *name*

A channel declaration consists of the channel type, and a comma separated list of names to identify the channels. The declaration appears on a single line, which may be split after any comma, and is terminated by a colon.

5.3 Arrays of channels

Arrays of channels can be declared in the same way as arrays of variables (see page 36). The following, for example, declares an array of channels:

```
[4]CHAN OF BYTE screens :
```

This declaration introduces an array **screens** of four channels.

Multidimensional arrays of channels are built in the same way as multidimensional arrays of variables, for example:

```
[5][5]CHAN OF PACKETS node :
```

There is a subtle semantic distinction to be made between an array of data type and an array of channels. An array of variables is itself a variable, as it may receive a new value by assignment or input. However, an array of channels is not itself a channel, as only single components of the array may be used in input/output, but a means of referencing a number of distinct channels identified by consecutive subscripts. This distinction is not made in the description of the syntax of channels.

Several arrays of the same type can be declared together. Consider the following example:

```
[users]CHAN OF BYTE screen, keyboard :
```

The type of the declarations is determined, and then the declarations are made.

The syntax of channel types is extended with:

channel.type = **[** *expression* **]** *channel.type*

A channel type may be preceded by an expression in square brackets. The value of the expression must be a constant and is the number of components in arrays of the channel type.

5.3.1 Channel array components and segments

Components and segments of channel arrays are denoted in the same way as components and segments of variable arrays.

Subscripted names select a component of an array. Suppose **user.in** is declared as follows:

```
[12]CHAN OF MESSAGES user.in :
```

Consider the example:

user.in[9]	the tenth component of the array **user.in**, of type **CHAN OF MESSAGES**.

A segment of an array is itself an array. The segment has zero or more components, as shown in the following examples:

[user.in FROM 9 FOR 1]	the tenth component of the array **user.in**, of type **[1]CHAN OF MESSAGES**.
[user.in FROM 9]	the tenth, eleventh and twelfth components of the array **user.in**, of type **[3]CHAN OF MESSAGES**.

A segment of an array has the same number of dimensions as the array.

The syntax is:

channel = *channel* **[** *expression* **]**
| **[** *channel* **FROM** *base* **FOR** *count* **]**
| **[** *channel* **FROM** *base* **]**
| **[** *channel* **FOR** *count* **]**

This syntax matches exactly the corresponding syntax for variables (page 37).

5.4 Channel protocol

A channel communicates values between two concurrent processes. The format and data type of these values is specified by the channel protocol. The channel protocol is specified when the channel is declared. Each input and output must be compatible with the protocol of the channel used. Channel protocols enable the compiler to check the usage of channels, and to ensure the same effect whether the sending or the receiving process is ready to communicate first.

5.4.1 Simple protocols

The simplest protocols consist of a data type. Examples of channels with byte and integer protocols have already been given. A protocol with an array or record type can be declared in the same way, for example, using types declared in chapter 3:

```
CHAN OF [36]BYTE message :          -- explicit array type
CHAN OF DOSFNAME fdir :             -- named array type
CHAN OF COMPLEX32 imp :             -- named record type
```

The first declaration introduces a channel with a byte array protocol which is identified by the name **message**. The protocol of this channel specifies that each communication on the channel consists of a byte array with 36 components. The second declaration declares **fdir** to be a channel on which are communicated file names of type **DOSFNAME**, defined on page 31 to be arrays of 12 bytes. The third declaration declares a channel for communicating complex numbers, each constructed as a record of type **COMPLEX32** (page 31).

The following outputs use these channels:

```
message ! "The vorpal blade went snicker-snack."
fdir ! "AUTOEXEC.BAT"(DOSFNAME)
fdir ! sourcename
imp ! [cosx, sinx](COMPLEX32)
```

The variable **sourcename** in the third example must be an array of 12 bytes of type **DOSFNAME**. Note that in the fourth example it would be possible to omit the type decoration **(COMPLEX32)** because the type of the record can be inferred from the type of the protocol.

Counted array protocols

It is often desirable to have a channel that will pass arrays of values, where the number of components in the array is not known until the output occurs. A special protocol, called a *counted array* protocol, enables this kind of array communication by passing a length and that number of components from the array. A declaration for such a channel looks like this:

```
CHAN OF INT::[]BYTE message :
```

This declaration introduces a channel which passes an integer value and that number of components from the array. An output on this channel will look like this:

```
message ! 16::"The vorpal blade went snicker-snack."
```

This has the effect of outputting the integer **16** and then the string **"The vorpal blade"**, the first **16** bytes of the array. The associated input could look like this:

```
message ? len::[buffer FROM start]
```

This input receives an integer value (**16** in this example), which is assigned to the variable **len**, and that number of components, which are assigned to components of the array starting at **buffer[start]**. The assignments to **len** and **buffer** happen in parallel and therefore the same rules apply as for parallel assignment. That is, no name appearing to the left of **::** may be used in the array variable on its right and *vice versa*[1]. The input is invalid if the number of components in the destination array is less than the count value input to **len**.

[1] In occam 2, the count was input first and the parallel assignment rules did not apply. Some occam 2 programs are invalidated by the new rule and implementors may provide a compiler option to accept programs written assuming the old definition.

All the above protocols are called *simple protocols*, the syntax of these and of inputs and outputs using them is:

simple.protocol	=	*data.type*
	|	*data.type* **::[]** *data.type*
input	=	*channel* **?** *input.item*
input.item	=	*variable*
	|	*variable* **::** *variable*
output	=	*channel* **!** *output.item*
output.item	=	*expression*
	|	*expression* **::** *expression*
protocol	=	*simple.protocol*

This syntax has extended the earlier syntax for *input* and *output* (page 6). A simple protocol is either a data type or a counted array as described above. A counted array is specified by the data type of the count (which may be either an integer or byte type), followed by a double colon, square brackets (**::[]**), and the specifier indicating the type of the components.

An input is a channel followed by the symbol **?** and an input item. An input item may be either a variable, of any data type, or a variable of integer or byte type followed by a double colon (**::**) and a variable of an array data type.

An output is a channel followed by the symbol **!** and an output item. An output item may be either an expression, of any data type, or an expression of integer or byte type followed by a double colon (**::**) and an expression of an array data type.

5.4.2 Naming a protocol

A protocol can be given a name in a *protocol definition*, as shown in the following example:

```
PROTOCOL CHAR IS BYTE :
```

A channel can now be declared with the protocol **CHAR**, for example:

```
CHAN OF CHAR screen :
```

While the naming of simple protocols is optional, a protocol definition must be used if more complex protocols, like the *sequential protocol* described in the following section are required.

The syntax for protocol definition is:

definition	=	**PROTOCOL** *name* **IS** *simple.protocol* **:**
	|	**PROTOCOL** *name* **IS** *sequential.protocol* **:**
protocol	=	*name*

A protocol definition defines a name for the simple protocol or sequential protocol (described in the following section) which appears to the right of the keyword **IS**. A simple or sequential protocol definition appears on a single line, and is terminated by a colon. The line may be broken after the keyword **IS** or after a semi-colon in a sequential protocol.

5.4.3 Sequential protocol

Simple protocols have been discussed earlier. Sequential protocols specify a protocol for communication which consists of a sequence of simple protocols. Consider the following example:

```
PROTOCOL COMPLEX IS REAL32; REAL32 :
```

Channels declared with this protocol (**CHAN OF COMPLEX**) pass pairs of values. An input or output on a channel with sequential protocol is a sequence of distinct inputs or outputs. An input on a channel with the above protocol **COMPLEX** is shown below:

```
items ? real.part; imaginary.part
```

Each value is input in sequence and assigned to each variable in turn.

It is important to note the difference between this protocol and the simple protocol **COMPLEX32** derived from the record data type declared on page 31. Communications using a simple protocol derived from a single record type pass a complete record value of several fields in a single communication, but communications using a sequential protocol use a sequence of separate communications and so do not require either the outputting or the inputting processes to have declared variables of record type from which and to which the data will be communicated. There will be situations in which each of these alternatives is the better choice of coding style.

Consider also the example:

```
COMPLEX32 z1, z2 : -- COMPLEX32 already declared as a record data type
PROTOCOL COMPLEX32 IS COMPLEX32 :   -- unnecessary protocol definition
CHAN OF COMPLEX32 cz :
PAR
  cz ! z1
  cz ? z2
```

This is a legal occam program, but the effect of the named protocol definition is to introduce a new meaning of **COMPLEX32** as a protocol and so in the scope of that declaration it would not be possible to declare new variables of the record type **COMPLEX32**. However variables of the record type which were declared before the protocol definition are still in scope after it and may be used in processes within its scope. In the absence of the protocol definition the channel declaration would still be legal as it would reference the simple protocol derived from the type definition of **COMPLEX32**. In this case new variables of this type may be declared in the scope of the channel declaration.

Here are some more examples of sequential protocol definitions:

```
PROTOCOL DIR.ENTRY IS INT16; [14]BYTE :
PROTOCOL INODE IS INT16;INT16;INT32;INT32;INT16;[7]INT16;INT16;INT16 :
PROTOCOL LINE IS INT16::[]BYTE :
```

Declarations of channels with these protocols would look like this:

```
CHAN OF DIR.ENTRY directory :
CHAN OF INODE  sys :
CHAN OF LINE blocks :
```

The syntax of sequential protocols, which can only appear in protocol definitions, is:

sequential.protocol	=	$\{_1$ **;** *simple.protocol* $\}$
input	=	*channel* **?** $\{_1$ **;** *input.item* $\}$
output	=	*channel* **!** $\{_1$ **;** *output.item* $\}$

A sequential protocol is one or more simple protocols separated by semicolons. A communication on a channel with a sequential protocol is legal provided the type of each item input or output is compatible with the corresponding component of the protocol.

5.4.4 CASE protocol

It is often convenient to use a single channel to communicate messages with different formats. A **CASE** protocol specifies a number of possible formats for communication on a single channel. Consider the following example:

```
PROTOCOL FILES
  CASE
    request;  BYTE
    filename; DOSFNAME
    word;     INT16
    record;   INT32; INT16::[]BYTE
    error;    INT16; BYTE::[]BYTE
    halt
:
```

This example defines a case protocol named **FILES**. **CASE** combines a number of *tags*, each of which may identify a tagged sequential protocol. The case protocol defined here has six variants.

A channel declared with this protocol would look like this:

```
CHAN OF FILES to.dfs :
```

A communication on this channel first sends a tag to inform the receiving process of the format for the rest of the communication. So, for example

```
to.dfs ! request; get.record
```

first sends the tag **request** followed by a **BYTE** value (**get.record**). Consider the output:

```
to.dfs ! halt
```

This output sends only the tag **halt** and according to the above **CASE** protocol definition requires no further output.

The syntax for a case protocol and the associated output is:

definition	=	**PROTOCOL** *name*
		  CASE
		    { *tagged.protocol* }
		:

tagged.protocol	=	*tag*
	|	*tag* **;** *sequential.protocol*
tag	=	*name*
output	=	*channel* **!** *tag*
	|	*channel* **!** *tag* **;** $\{_1$ **;** *output.item* $\}$

In a definition of a case protocol the name which identifies the protocol appears to the right of the keyword **PROTOCOL**, this is followed at an indentation of two spaces by the keyword **CASE**, which in turn is followed on succeeding lines at a further indentation of two spaces by a number of tagged protocols. The definition of a case protocol is terminated by a colon, which appears on a line by itself, at the same level of indentation as the **P** of the keyword **PROTOCOL**. A tagged protocol is either a tag by itself or a tag followed by a semi-colon, and a sequential protocol.

A name may be used as a tag within multiple protocols in the same scope. A tag name may also be the same as a name defined in any other specification in the scope.

An output on a channel of case protocol is a tag by itself or a tag followed by a number of output items separated by semi-colons. The output is valid only if the tag and the associated output items are compatible with one of the tagged protocols specified in the definition of the protocol.

An implementation may restrict the number of tags in a protocol definition so that a tag may be represented by a value in a limited range, but this range must be at least 0 to 255.

Input on a channel with case protocol

So far only output on a channel with case protocol has been shown. A special form of input is required (called *case input*) to provide for input on channels with a case protocol. The previous example is suggestive of a *conversation* with a *disc filing system*, and is a reminder that channels are unidirectional. So, for a user process to "listen to" the other side of this conversation, another channel must be declared, as shown below:

```
CHAN OF FILES from.dfs :
```

This example declares another channel with the protocol **FILES**. The process which outputs **request; get.record**, might reasonably expect to receive a reply on a channel with this protocol. Consider a more complete example of this conversation:

```
SEQ
  to.dfs ! request; get.record
  from.dfs ? CASE
    record; rnumber; rlen::buffer
      ...  do whatever
    error;  enumber; elen::buffer
      ...  handle error
```

Illustrated in the above example is a case input on the channel **from.dfs**. This accepts a variant input with either the tag **record** or the tag **error**; any other tag would be invalid and the input would behave like the primitive process **STOP**.

A special form of case input simply receives a tag from the channel named on the left of the case input symbol (**? CASE**), and then compares the tag for equality with the tag of the tagged list which appears to the right of the symbol. A tag is input, then if the tags match the process next inputs the remainder of the tagged list. If the tags do not match, the process next behaves like the primitive process **STOP**. For example:

```
from.dfs ? CASE filename; name.buffer
```

This process inputs a tag. If the tag is **filename** the input is completed, and a value assigned to the variable **name.buffer**. Otherwise, no further input is performed, and the input behaves like the primitive process **STOP** (page 7). A case input is legal only if the tagged lists are compatible with tagged protocols specified in the definition of the case protocol.

Consider the following:

```
PROTOCOL COMMS
  CASE
    packet;INT::[]BYTE
    sync
:
CHAN OF COMMS route :
PAR
  SEQ
    route ! packet; 11::"Hello world"
    R ()
  SEQ
    route ? CASE sync
    S ()
```

In this example the input **route ? CASE sync** will behave like the primitive process **STOP** as the tags do not match. The associated output will also behave like **STOP**, for although the output of the tag **packet** succeeds, the output **11::"Hello world"** does not. In this example the procedures **R()** and **S()** will not be performed. Also consider the following:

```
PAR
  SEQ
    route ! sync
    P ()
  SEQ
    route ? CASE packet; length::message
    Q ()
```

Each communication of a sequential protocol, or of a tagged sequential protocol is in fact a sequence of separate communications. So, in the above example, the input **route ? CASE packet; length::message** will behave like the primitive process **STOP** because the tags do not match. However, the associated output **route ! sync** will succeed as the output of the tag has completed, and the variant requires no further output. Thus, the *procedure* (page 69) **P()** will be performed, and the procedure **Q()** will not be performed.

The syntax for case input is:

case.input	=	*channel* **? CASE**
		{ *variant* }
variant	=	*tagged.list*
		process
	|	*specification*
		variant
tagged.list	=	*tag*
	|	*tag* **;** $\{_1$ **;** *input.item* }
process	=	*case.input*
input	=	*channel* **? CASE** *tagged.list*

A process which is a case input receives a tag from the channel named on the left of the case input symbol (**? CASE**), and then the tag is used to select one of the variants. These appear on the following lines, indented by two spaces and optionally preceded by local specifications. A tag is input, then if a variant with that tag is present the process next inputs the remainder of the tagged list, and an associated process, indented a further two spaces, is performed. If no variant with that tag is found the process next behaves like the primitive process **STOP**.

An input may contain the case input symbol followed by a single tagged list only, as shown in the earlier examples.

Variants in alternatives

A case input may also be used as an input in an alternation (chapter 2, page 19). Consider the following example:

```
ALT
  from.dfs ? CASE
    request; query
      ...  do query
    error; enumber; elen::buffer
      ...  handle dfs error
    record; rnumber; rlen::buffer
      ...  accept record

  from.network ? CASE
    request; query
      ...  do query
    error; enumber; elen::buffer
      ...  handle network error
    record; rnumber; rlen::buffer
      ...  accept record
```

This alternation accepts input from either of the two channels (**from.dfs** and **from.network**). These inputs are explained in the previous section. This alternation could have included a mix of case inputs, and the alternatives described on page 19. The syntax for case inputs in an alternative is:

alternative	=	*channel* **? CASE**
		{ *variant* }
	|	*boolean* **&** *channel* **? CASE**
		{ *variant* }

A case input as an alternative is either a case input with variants as described in the earlier syntax, or such a case input preceded by a boolean guard and an ampersand (**&**) to the left of the channel name. The case input is not considered by the alternation if the boolean guard is false.

Scope of tag names

A name may be used as a tag within more than one case protocol in the same scope. For example:

```
PROTOCOL LINES
  CASE
    aline; INT::[]BYTE
    eof
:
PROTOCOL HEATING
  CASE
    kettle; KETTLE
    heater; HEATER
    eof
:
CHAN OF LINES document:
CHAN OF HEATING installation:
PAR
  ...
  SEQ
    ...
    document ! eof                  -- eof of LINES
    ...
    installation ! eof              -- eof of HEATING
    ...
```

It is possible for a tag name of a tagged protocol to be used again in a specification or as a field name in a record during the scope of the protocol. For example:

```
PROTOCOL COMMS
  CASE
    packet;INT::[]BYTE
    sync
:
[4]BYTE packet :
```

However, it is still possible to use the name **packet** to denote a tag of the protocol **COMMS** in outputs and **CASE** inputs on channels with protocol **COMMS**. Consider the following simple program:

```
PROTOCOL COMMS
  CASE
    packet;INT::[]BYTE
    sync
:
[4]BYTE packet :
CHAN OF COMMS c :
INT len :
PAR
  c ! packet; 3::"xyz"
  c ? CASE packet; len::packet
```

The name **packet** is used to denote the variant tag in both the input and the output. The name is also used to specify the destination of the input. Confusing usages such as this are usually better avoided.

5.4.5 Anarchic protocol

In some situations it may be necessary to specify a channel protocol where the format of the protocol for some reason cannot be defined. Such situations are rare, and are likely to occur only when communicating with an external device such as a printer, terminal or other device controller. Such a device can be considered an *alien process* where the protocol for communication with that process is dictated by the nature of the device. A special protocol exists which allows the input and output of any format without checking. The protocol is specified by the keyword **ANY**, as illustrated in the following examples:

```
CHAN OF ANY mouse:
PROTOCOL PRN IS ANY :
CHAN OF PRN printer :
```

The keyword **ANY** may be used directly in a channel declaration such as that of **mouse** above, or it may be used as a simple protocol as in the definition of **PRN**.

A channel with a protocol defined to be **ANY** in either of these ways can only input or output data values. It cannot handle case protocols. The effect of an output on a channel with an **ANY** protocol is that the value is mapped down into its constituent bytes, and output as an array of bytes. An input on a channel with any **ANY** protocol inputs the array of bytes and converts (by retyping conversion, see page 85) the value to the type of the receiving variable.

A protocol defined to be **ANY** may be redefined to have any other protocol by a channel retyping definition (page 86). Such retypings are necessary when passing a channel with anarchic protocol as a actual parameter to a procedure whose corresponding formal has any other protocol.

The syntax extends that of simple protocol:

simple.protocol	=	**ANY**

5.5 Abbreviation of channels

Channel abbreviations are similar to variable abbreviations (see page 42). A channel abbreviation specifies a new name for a channel or a channel array. Consider the following examples:

```
[]CHAN OF INT clients IS [from.a, from.b, from.c, from.d, from.e] :
CHAN OF INT client2 IS clients[2] :
clients01 IS [clients FOR 2] :
otherclients IS [clients FROM 3] :
```

This introduces the name **clients** for the constructed array shown. It then introduces **client2** as the new name for **clients[2]** and **clients01** and **otherclients** for non-overlapping segments of the array **clients**.

The syntax of channel abbreviation is:

abbreviation	=	*specifier name* **IS** *channel* **:**
	|	*name* **IS** *channel* **:**
	|	*specifier name* **IS** **[**$\{_1$ **,** *channel* $\}$**]** **:**
	|	*name* **IS** **[**$\{_1$ **,** *channel* $\}$**]** **:**
specifier	=	*channel.type*
	|	**[]** *specifier*
	|	**[** *expression* **]** *specifier*

The specifier must specify a channel type but may be omitted whenever the type of the abbreviation can be inferred from the type of the channel. Channel abbreviations are subject to the same usage restrictions as variable abbreviations. These are summarised in appendix E.

5.5.1 Channel array constructors

In the syntax above it can be seen that a channel array in an abbreviation may be constructed as a table of individual channels. Consider the example:

```
sources IS [from.a, from.b, from.c] :
sources[i] ? x
```

The three channels originally declared separately have been constructed into an array which can then be named in an abbreviation so that it may be subscripted or segmented. This can be particularly useful when **i** is the replication index of a replicated alternation, and the construction is being used to receive input on one of a group of channels of the same protocol.

This example shows how the constructor might be used to combine an arbitrary collection of channels of the same protocol for use by a process which listens to all of them:

```
CHAN OF BYTE from.a, from.b, from.c:
PAR
  SEQ
    ...  process a
    from.a ! 'A'
    ...
  SEQ
    ...  process b
    from.b ! 'B'
    ...
  SEQ
    ...  process c
    from.c ! 'C'
    ...
  sources IS [from.a, from.b, from.c]:
  [3]BYTE bb:
  SEQ j = 0 FOR 3
    ALT i = 0 FOR 3
      sources[i] ? bb[i]
        SEQ
          ...
```

This example has been so simplified that the need for a constructor is not immediately obvious. The need becomes greater if the individual channels are allocated to non-contiguous hardware addresses (page 94).

The channels combined in a channel array constructor must all be used in the same direction. Channel array constructors can only be used within channel abbreviations (and as actual parameters to procedure instances, see page 70).

6 Expressions

This chapter is about *expressions*, and describes the range of *operators* provided by occam. The chapter also describes *tables* and *data type conversions*.

An expression is evaluated and produces a result. The result of an expression has a value and a data type. The simplest expressions are literals and variables. More complex expressions are constructed from *operands*, *operators* and *parentheses*. An operand is a *variable* (page 35), a literal, a table, or an expression in parentheses. An operator performs an operation, for example an addition, upon its operand(s). The following are all valid expressions:

`5(INT64)`	a literal value
`x`	a variable
`6 * 4`	multiplication of two literal operands
`x * y`	multiplication of two variable operands
`y << 2`	shift of a variable operand by a literal
`x = y`	comparison of two variable operands
`NOT TRUE`	a boolean expression

An expression in parentheses may itself be an operand in an expression. In this way larger expressions are built, as shown in the following examples:

`(1 + 2) - 1`	subtract `1` from the result of `(1 + 2)`
`(x * y) * (w * z)`	multiply the results of the expressions `(x * y)` and `(w * z)`

There is no operator precedence and so the hierarchical structure of a large expression must be defined by parentheses. With the exception of shift operations, where the number of bits to be shifted must be indicated by a value of type `INT`, the data type of the two operands in a dyadic expression must be of the same type. In an assignment the value of the expression must be of the same data type as the variable to which it is to be assigned. Consider in detail the following example:

```
y := (m * x) + c
```

Each of the variables in this example (`y`, `m`, `x` and `c`) must be of the same data type. The value of an expression is of the same type as its operand(s). The expression in this example - `(m * x) + c` - has two operators. The parentheses indicate that the expression `(m * x)` is an operand of the operator `+`, and thus must be evaluated before the `+` operation can be performed.

The right hand second operand of a shift operator must always be, or be converted to be, of the explicit type `INT`. Apart from this, where an operator is defined in this chapter to operate on operands of a particular primitive data type, it may be assumed also to operate on operands of a named type derived from that underlying type. The two operands of a dyadic operator (excluding shifts) must both be of the same type, and so must be explicitly converted (see page 66) to the same type if they are declared as, or computed to be, of different types.

The syntax for expressions includes:

expression	=	*monadic.operator operand*
	|	*operand dyadic.operator operand*
	|	*operand*
	|	*conversion*
operand	=	*variable*
	|	*literal*
	|	*table*
	|	`(`*expression*`)`
variable	=	*name*
monadic.operator	=	`-` | `MINUS` | `~` | `BITNOT` | `NOT` | `SIZE`
dyadic.operator	=	`+` | `-` | `*` | `/` | `\` | `REM` | `PLUS` | `MINUS` | `TIMES`
	|	`/\` | `\/` | `><` | `BITAND` | `BITOR` | `AND` | `OR`
	|	`=` | `<>` | `<` | `>` | `>=` | `<=` | `AFTER`

A monadic operator precedes its one operand. A dyadic operator appears between its two operands. Tables, operators and conversions are detailed in the following sections. Variables (page 35) and literals (page 27) have been explained earlier. An operand which is a table can only be used where an array may be used. Conversions are decribed on page 66.

6.1 Tables and strings

A table constructs an array of values from a number of expressions which must yield values of the same data type. A string is a compact representation of a table of bytes. The value of each component of the array is the value of the corresponding expression. Consider the following example:

```
[1, 2, 3]
```

This example constructs an array with three components, each of type **INT**. Here are some more examples:

`['a', 'b', 'c']`	a table of three bytes (equivalent to `"abc"`)
`[x, y, z]`	a table of three values
`[x * y, x + 4]`	a table with two component values
`[(m * x) + c]`	a table with a single component
`[6(INT64), 8888]`	a table of two **INT64** integers
`"occam"`	a string literal
`"RUNHELLO.BAT"(DOSFNAME)`	a string literal of a named type

A table that is not a string is one or more expressions of the same data type, separated by commas, and enclosed in square brackets, optionally followed by the type name of an array or record type in parentheses. Line breaks are permitted after a comma.

A table is a value of an array type. If the variables **m**, **x** and **c** are of type **INT**, then the table **[(m * x) + c]** is an expression whose type is **[1]INT**. **['a', 'b', 'c']** is an expression whose type is **[3]BYTE**, and so on.

A table may be subscripted or segmented like an array variable (page 35).

If a table appears in a context where its type is known, the type of its components are then known and may be untyped literals (page 29) whose type is implicitly determined. However if a table appears in a context where its type is not known the type of its components will be inferred from the first component found to have a known type. Consider:

```
VAL []REAL32 tab1 IS [0.0, 1.1] :
VAL tab2 IS [2.0(REAL32), 2.1, 2.2] :
VAL tab2a IS [1.9, 2.0(REAL32), 2.2] :
VAL tab3 IS [3.0, 3,1] :     -- ILLEGAL
DATA TYPE VEC2 IS [2]REAL32 :
VAL tab4 IS [4.0, 4.1](VEC2) :
```

The type of **tab1** is determined from the specifier **[]REAL32**. The type of **tab2** is determined from its first component which is an explicitly decorated literal. The type of **tab2a** is determined from its second component which is an explicitly decorated literal. The type of **tab3** cannot be determined and so the abbreviation is illegal. The type of **tab4** is determined by the explicit decoration of the table, which must be done by a type name rather than a type specifier including subscript brackets.

A string is a sequence of characters enclosed in double quotes. A string of a named byte array type may be decorated with a type name in parentheses. Note that though a string is lexically a literal constant, it is treated syntactically as a table of bytes.

The type of a string is an array of type **[]BYTE**. The string **"zen"** is an array of type **[3]BYTE**. Each component of the array is the ASCII value of the corresponding character in the string.

Special character sequences allow certain control values such as Tabulation and Carriage Return to be included in strings. Full details of the occam character set and special characters are given in the appendix (page 103).

A string may be broken over several lines by terminating broken lines with an asterisk, and starting the continuation on the following line with another asterisk. The indentation of the continuation should be no less than the current indentation, as illustrated in the following example:

```
occam := "Beware the jabberwock my son, the jaws that bite, the*
         * claws that catch, beware the jubjub bird, and shun the*
         * frumious bandersnatch."
```

The syntax for tables and strings is:

table	=	*string*
	|	*string* **(***name***)**
	|	**[** $\{_1$ **,** *expression* $\}$ **]**
	|	**[** $\{_1$ **,** *expression* $\}$ **]** **(***name***)**
	|	*table* **[** *expression* **]**
	|	**[** *table* **FROM** *base* **FOR** *count* **]**
	|	**[** *table* **FROM** *base* **]**
	|	**[** *table* **FOR** *count* **]**

A table is either a string or is a sequence of expressions separated by commas inside a pair of square brackets. It may optionally be decorated by a an array data type name in parentheses. A table may be subscripted or segmented in the same way as an array variable (see page 37). A segmented or subscripted table may not be decorated.

Tables are syntactically very similar to record literals, (page 32), the difference being that all components of a table must be of the same type, and so the application of the rules of literal type inference are different within a table.

6.2 Operations on values

An operation evaluates its operand(s) and produces a result. The result of an operation has a value and a data type. These are the operators which operate on values, which are defined in related groups in subsections below:

+	addition	**>>**	shift right
-	subtraction	**<<**	shift left
*****	multiplication	**AND**	boolean and
/	division	**OR**	boolean or
\ REM	remainder	**NOT**	boolean not
PLUS	modulo addition	**=**	equal
MINUS	modulo subtraction	**<>**	not equal
TIMES	modulo multiplication	**<**	less than
/\ BITAND	bitwise and	**>**	greater than
\/ BITOR	bitwise or	**<=**	less than or equal
><	bitwise exclusive or	**>=**	greater than or equal
~ BITNOT	bitwise not	**AFTER**	later than
SIZE	array size	**BYTESIN**	element size

Some operators are symbols, others keywords. Those symbols which use characters which may be absent in some national variants of ASCII (i.e. **** or **~**) have alternative representations as keywords.

6.2.1 Arithmetic operators

The arithmetic operators are:

+	addition
–	subtraction
*****	multiplication
/	division
\ REM	remainder

Arithmetic operators perform an arithmetic operation upon operands of the same real, integer or byte type (not on booleans), for example:

39 + 3	produces a value of 42
'T' + 32	produces a value equal to **'t'**
45 – 3	produces a value of 42
6 * 7	produces a value of 42
2.0(REAL32) * 3.14159	produces an approximation to 2π
126 / 3	produces a value of 42
128 REM 3	produces a value of 2

The final example in this list may also be written: **128 \ 3**. The symbols **REM** and **** both signify the remainder operation. A remainder operation produces a value which is the remainder of the division of the two operands. The sign of an integer remainder operation is the sign of the left hand operand (except where the result is zero) regardless of the sign of the right hand operand. The result of an integer division is rounded toward zero (i.e. truncated), for example:

3 / 2	produces a value of 1
(–3) / 2	produces a value of -1
(–9) / 4	produces a value of -2
(–9) REM 4	produces a value of -1

The operator **–** is also a monadic negation operator, which has the effect of negating the value of its operand which must be of an integer or real type, for example:

– x	has the value (0 – **x**)
– 5	minus 5
– 1.0E+06(REAL32)	minus one million

An arithmetic operation produces a result of the same data type as the operands. An arithmetic operation is not valid if the resulting value cannot be represented by the same data type as the operands, for example where the result of a multiplication of two large integers produces a value which exceeds the range of the type (arithmetic overflow). Division by zero is also treated as invalid.

Remainder operations, on both integers and reals, obey the following law:

$$((x/y) * y) + (x \,\mathbf{REM}\, y) = x$$

Here are some examples of real expressions, in which **x** is a value of **39.0(REAL32)**, and **y** is a value of **3.0(REAL32)**:

x + y	produces a value of 42.0 of type **REAL32**
x – y	produces a value of 36.0 of type **REAL32**
x * y	produces a value of 117.0 of type **REAL32**
x / y	produces a value of 13.0 of type **REAL32**
x REM y	produces a value of 0.0 of type **REAL32**

Rounding the results of real operations

The result of a real arithmetic expression (which is considered to be infinitely precise) is rounded to the nearest value which can be represented by the type. That is, the value will be adjusted, if necessary, to fit into the representation of its type. The precision of an operation is that of the type of the operands.

It is possible for the result of a real remainder operation to be negative. Consider the following example:

```
1.5(REAL32) REM 2.0(REAL32)
```

The result of this expression is (-0.5). If x and y are real values, the result of x **REM** y is $(x-(y*n))$, where n is the result of dividing x and y rounded toward zero. Applying this to the above example, n is 0.75 rounded to the nearest integer (1), leaving : $(1.5-(2.0*1)) = (-0.5)$.

Full details of IEEE rounding modes are given in the appendix (page 98).

6.2.2 Modulo arithmetic operators

The modulo arithmetic operators are:

PLUS	modulo addition
MINUS	modulo subtraction
TIMES	modulo multiplication

These modulo arithmetic operators perform an operation upon operands of the same integer or byte data type (not on reals or booleans). Whilst the effect of these operations is similar to the corresponding arithmetic operations, no overflow checking takes place, and thus the values are cyclic. For example, adding one to the most positive integer will produce a value equal to the most negative integer (i.e. ($MOSTPOS$ **PLUS** 1) = $MOSTNEG$), and subtracting one from the most negative integer will produce a value equal to the most positive integer (i.e. ($MOSTNEG$ **MINUS** 1) = $MOSTPOS$). Consider these examples:

32767(INT16) + 1(INT16)	causes an arithmetic overflow. **INVALID!**
32767(INT16) PLUS 1(INT16)	produces the value -32768.
(-32768(INT16)) - 1(INT16)	causes an arithmetic overflow. **INVALID!**
(-32768(INT16)) MINUS 1(INT16)	produces the value 32767.
20000(INT16) * 10(INT16)	causes an arithmetic overflow. **INVALID!**
20000(INT16) TIMES 10(INT16)	produces the value 3392
255(BYTE) + 1(BYTE)	causes an arithmetic overflow. **INVALID!**
255(BYTE) PLUS 1	produces the byte value 0
128(BYTE) TIMES 3	produces the byte value 128

MINUS is also a valid monadic operator. **MINUS z** is always equivalent to **0 MINUS z** and so if **z** is of a byte type then it is equal to **256 - z**.

6.2.3 Bit operations

Bitwise operators perform operations on the bit pattern of a value of an integer or byte type. The bitwise operators are:

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

Here are some example expressions using the bitwise operators. The results shown are correct if the value of **`pixel`** is **`#1010`**, and the value of **`pattern`** is **`#FFFF`**, and their type is **`INT16`**:

`pixel /\ pattern`	produces a result **`#1010(INT16)`**
`~ pixel`	produces a result **`#EFEF(INT16)`**
`pixel \/ pattern`	produces a result **`#FFFF(INT16)`**
`pixel >< pattern`	produces a result **`#EFEF(INT16)`**

The operands of **`/\`**, **`\/`** and **`><`** must both be of the same integer or byte type. The following table illustrates how each bit of the result is produced from the corresponding bits in the operand.

1 >< 0 = 1	1 /\ 0 = 0	1 \/ 0 = 1
0 >< 0 = 0	0 /\ 0 = 0	0 \/ 0 = 0
1 >< 1 = 0	1 /\ 1 = 1	1 \/ 1 = 1
0 >< 1 = 1	0 /\ 1 = 0	0 \/ 1 = 1

The bitwise not operator (**`~`**) has a single operand which must be of an integer or byte type. Each bit of the result is the inverse of the corresponding bit in the operand, as shown in the following table:

~1 = 0
~0 = 1

The result of a bitwise operation is of the same integer or byte type as the operand(s). The keywords **`BITAND`**, **`BITOR`** and **`BITNOT`** are equivalent to **`/\`**, **`\/`**, **`~`** respectively, and are included especially for implementations which have a restricted character set.

6.2.4 Shift operations

The shift operators perform a logical shift on a value of an integer or byte type. The shift operators are:

`>>`	shift right
`<<`	shift left

The shift operators shift the bit pattern of a value of any integer or byte type by a number of places determined by a count value which must be of type **`INT`**. For example, if the value of **`n`** is **`#FFFF`**, and of type **`INT16`**:

`n << 4`	produces a result **`#FFF0(INT16)`**
`n >> 4`	produces a result **`#0FFF(INT16)`**

The result is of the same integer or byte type as **`n`**. The bits vacated by the shift become zero, the bits shifted out of the pattern are lost. The left shift operator shifts toward the most significant end of the pattern, the right shift operator shifts toward the least significant end of the pattern.

Consider these further examples, where **`n`** is a value of type **`INT32`**:

`n << 0`	produces the value **`n`**
`n >> 0`	produces the value **`n`**
`n >> 32`	produces the value 0
`n << 32`	produces the value 0

A shift by a negative value, or by a value which exceeds the number of bits in the representation, is invalid.

6.2.5 Boolean operations

The boolean operators operate on operands of boolean type, and produce a boolean result. The boolean operators are:

AND	boolean and
OR	boolean or
NOT	boolean not

The following table shows the results for each operation:

$false$	**AND**	$true$	$= false$	$false$	**OR**	$true$	$= true$	**NOT**	$false$	$= true$
$false$	**AND**	$false$	$= false$	$false$	**OR**	$false$	$= false$	**NOT**	$true$	$= false$
$true$	**AND**	$false$	$= false$	$true$	**OR**	$false$	$= true$			
$true$	**AND**	$true$	$= true$	$true$	**OR**	$true$	$= true$			

The operand to the left of a boolean operator is evaluated, and if the result of the operation can be determined evaluation ceases. This differs from the behaviour of other expressions. Consider the following example:

```
IF
  ((ch >= 'a') AND (ch <= 'z')) OR ((ch >= 'A') AND (ch <= 'Z'))
    ...
  (ch = cr) OR (ch = down) OR (ch = up)
    ...
  ((ch = escape) AND shift)) OR ((ch = escape) AND control))
    ...
```

Note that parentheses may be omitted between expressions containing adjacent **AND** or **OR** operators. The evaluation of the boolean expression **((ch >= 'a') AND (ch <= 'z'))** ceases if the expression **(ch >= 'a')** is false, in which case the evaluation of the expression **(ch <= 'z')** does not take place. If the result is true, the expression **((ch >= 'A') AND (ch <= 'Z'))** to the right of **OR** is not evaluated. The rule is that evaluation of a boolean expression will cease if the operand to the left of **AND** is false, or if the operand to the left of **OR** is true.

6.2.6 Relational operations

The relational operators perform a comparison of their operands, and produce a boolean result. The relational operators are:

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

Here are examples of relational expressions using **=** and **<>**. In these examples the operands, **x** and **y**, can be any primitive data type:

x = y	is true if the value of **x** is equal to the value of **y** the result is false otherwise
x <> y	is true if the value of **x** is not equal to the value of **y** the result is false otherwise

The following are examples using the other relational operators. In these examples the operands, **x** and **y**, can be an integer, byte or real type, but may not be a boolean:

x < y	is true if the value of **x** is less than the value of **y** the result is false otherwise
x > y	is true if the value of **x** is greater than the value of **y** the result is false otherwise
x <= y	is true if the value of **x** is less than or equal to the value of **y** the result is false otherwise
x >= y	is true if the value of **x** is greater than or equal the value of **y** the result is false otherwise

AFTER (later than)

The special modulo operator **AFTER** performs a comparison operation, and returns a boolean result, for example:

```
(a AFTER b)
```

This expression is true if **a** is later in a cyclic sequence than **b**, just as one o'clock *pm* can be considered later than eleven o'clock *am*. The first operand is considered the starting point on a "clock face" of integer values. If the shortest route to the value of the second operator is clockwise, then the value is later than the first operand and the result of the expression is true. If the shortest route to the value of the second operand is anticlockwise, or the routes are equal in length, then the value of the second operand is earlier, and the result of the expression is false.

If the type of the operands is a (signed) integer type (a **AFTER** b) produces the same value as (a **MINUS** b) > 0.

If the type of the operands is a byte type (which is unsigned) (a **AFTER** b) produces the same value as ((a **MINUS** b)**MINUS**1) < 127.

6.2.7 SIZE (number of components in an array)

The special operator **SIZE** may have a single operand of any array type, and produces an integer value of type **INT**, equal to the number of components in the array. For example, if **a** is an array of type **[8]INT**, then:

SIZE a	produces the value 8

If **b** is of type **[8][4]DOSFNAME** (see page 31), then:

SIZE b	produces the value 8
SIZE b[1]	produces the value 4
SIZE b[0][0]	produces the value 12

This operator is most often used to determine the size of an array passed as a formal parameter in a procedure or function (pages 69 and 76). The operator **SIZE** may also be applied to named array data types (page 65).

6.2.8 BYTESIN (bytes occupied by implementation of array element)

The operation of **BYTESIN** on operands is discussed below together with its operation on types (page 65).

6.3 Operations on types

An operation on a type produces a result based on a property of the implementation of all values of the type. The result of an operation on a type has a value and a data type.

6.3.1 MOSTPOS and MOSTNEG (integer range)

MOSTNEG	most negative
MOSTPOS	most positive

The operator **MOSTPOS** produces the most positive value of an integer or byte type. The operator **MOSTNEG** produces the most negative value of a (signed) integer type, and the value 0 for an (unsigned) byte type. The type of the result is the same as the operand. Consider the following examples:

MOSTPOS BYTE	has the value 255
MOSTPOS INT16	has the value 32767
MOSTPOS INT32	has the value 2147483647
MOSTPOS INT64	has the value 9223372036854775807
MOSTNEG BYTE	has the value 0
MOSTNEG INT16	has the value −32768

The syntax for these operators extends that of *expression*, and is:

expression	=	**MOSTPOS** *data.type*
	|	**MOSTNEG** *data.type*

The keyword (**MOSTPOS** or **MOSTNEG**) appears to the left of the name of an integer or byte data type.

6.3.2 SIZE (number of components in an array type)

SIZE	array size

The operator **SIZE** may have a single operand which is an array data type, and produces an integer value of type **INT**, equal to the number of components in arrays of that type. For example, if **DOSFNAME** is a name defined as an array of type **[12]BYTE**, then:

SIZE DOSFNAME	produces the value 12

The syntax for **SIZE** applied to types extends that of *expression*, and is:

expression	=	**SIZE** *data.type*

where the data type is any array data type. The operator **SIZE** may also be applied to arrays, see page 64.

6.3.3 BYTESIN and OFFSETOF (field positions in records)

BYTESIN	bytes in a record
OFFSETOF	offset of record field

It is sometimes necessary to discover how the compiler has mapped the fields of a record data type onto the computer's memory. The mapping may be determined by applying the **BYTESIN** operator to discover the total number of bytes occupied by a variable of the record type, and the **OFFSETOF** operator to a pair of names to discover the offset of a field within variables of a particular record type. The values returned by these operators are of type **INT**.

The **BYTESIN** operator may be applied to any data type or to any expression yielding a value of that type and returns the number of bytes that would be occupied by a component of that type in an array. This takes account of any need to add padding at the end of a record to meet implementation dependent alignment restrictions. This implies that:

BYTESIN([n**]** *type***)** $= n *$ **(BYTESIN(***type***))**

It is illegal to apply **BYTESIN** to an object whose size cannot be represented as an integral number of bytes.

The **OFFSETOF** operator is applied to two operands in parentheses. The first operand must be the name of a record data type and the second operand the name of a field declared in the definition of that type. The value returned is the offset in bytes from the start of the record in memory and the start of the field indicated.

OFFSETOF is invalid if its first operand is not the name of a record or packed record type, or if its second operand is not a field of that record type. It is invalid to apply **OFFSETOF** to a field whose offset cannot be represented as an integral number of bytes.

Let us assume that the following record type meets any restrictions on alignment imposed by the implementation and that the compiler has not needed to force the insertion of padding bytes. If so the operations yield the results indicated:

```
DATA TYPE MIXED
  PACKED RECORD
    BYTE b1, b2:
    INT16 i1:
    REAL32 r1:
    REAL64 r2:
    DOSFNAME n1:
:
MIXED m1 :
INT a, b, c, d, e, ob1, ob2, or2, on1 :
SEQ
  a := BYTESIN (MIXED)          -- 28
  b := BYTESIN (m1)             -- 28
  c := BYTESIN (m1[r1])         -- 4
  d := BYTESIN (m1[n1])         -- 12
  e := BYTESIN (m1[n1][0])      -- 1
  ob1 := OFFSETOF (MIXED, b1)   -- 0
  ob2 := OFFSETOF (MIXED, b2)   -- 1
  or2 := OFFSETOF (MIXED, r2)   -- 8
  on1 := OFFSETOF (MIXED, n1)   -- 16
```

The syntax for operands using these operators is:

operand	=	**BYTESIN (** *operand* **)**
	|	**BYTESIN (** *data.type* **)**
	|	**OFFSETOF (** *name* **,** *field.name* **)**

The operand of **BYTESIN** may be syntactically either an operand or any data type. The operands of **OFFSETOF** must be names, the first of a structured data type and the second of a field of that type. Note that **BYTESIN** and its operand in parentheses and **OFFSETOF** and its operands in parentheses are treated syntactically as operands, as if they were function instances, see page 78.

6.4 Data type conversion

With the exception of logical shifts (where the number of bits to shift must be of type **INT**), the types of the operands in an expression must be of the same type. Operands may explicitly have their data type converted.

A data type conversion permits a value of any non-array data type to be converted to a numerically similar value of another non-array data type. A data type conversion produces the value of its operand as a value of the specified data type, for example:

```
j := (k * 4.5(REAL64)) * (REAL64 n)
```

The value of **n** in this example is converted to a value of type **REAL64**. Note that **4.5(REAL64)** is a literal value of type **REAL64**, whereas **(REAL64 n)** is a data type conversion of the value of **n**.

The syntax for data type conversions is:

conversion	=	*data.type operand*
	|	*data.type* **ROUND** *operand*
	|	*data.type* **TRUNC** *operand*

The data type must be a primitive data type, or a named type derived from a primitive data type and appears to the left of the operand. A data type conversion which includes the keyword **ROUND** as described by the syntax, produces a value rounded to the nearest value of the specified type. Where two values are equally near, the value is rounded toward the nearest even number. A data type conversion which includes the keyword **TRUNC** as described by the syntax, produces a value truncated (rounded toward zero) to a value of the specified type. A real value of appropriate magnitude can be rounded or truncated to a value of type **BYTE**.

A conversion between any of the integer types, and conversions between those types and type **BYTE**, is valid only if the value produced is within the range of the receiving type. Byte and integer values may be converted to boolean values if their value is one or zero. The boolean value is true if the value is one, and false if the value is zero. That is:

BOOL 1	evaluates to **TRUE**
BOOL 0	evaluates to **FALSE**
INT TRUE	evaluates to 1
INT FALSE	evaluates to 0

Conversions from integer or byte values to real values, and vice versa, must specify whether the result is to be rounded or truncated. A value of type **REAL32** can be extended to an exact value of type **REAL64**. Values of type **REAL64** can be converted to values of type **REAL32**, providing the value is in the range of the **REAL32** type. The conversion must specify if the value is to be rounded or truncated. Consider these examples, where **n**, and **m** are integers of type **INT64**, and **n** has a value 255 and **m** has a value 3:

BYTE n	produces a byte value 255
REAL32 ROUND n	produces a **REAL32** value 255.0
REAL64 TRUNC n	produces a **REAL64** value 255.0
REAL64 ROUND(n * m)	produces a **REAL64** value 765.0
(REAL64 ROUND n) * (REAL64 ROUND m)	produces a **REAL64** value 765.0

Conversions may be applied to operands of the same type, but will have no effect. The truncation and rounding of integer types to real types occurs where the integer cannot be exactly represented as a value of the real type. Consider the following example:

```
SEQ
  i := 33554435 (INT32)  -- hex #2000003
  a := REAL32 ROUND i
  b := REAL32 TRUNC i
```

The value in this example has been chosen specifically to illustrate the behaviour of explicitly rounding an integer value which cannot be directly represented in the floating point representation of **REAL32**. The value of **a** after this sequence is 33554436.0, and the value of **b** is 33554432.0. For **b**, the two least significant bits of the integer representation have been lost (they had held the value 3). For **a** the value of those bits has been rounded to the next nearest representable value. Further detail of rounding is given in the appendix on page 98.

Conversion of real values to integers has the effect illustrated by the following examples:

INT32 ROUND 0.75(REAL32)	produces a value of 1
INT32 ROUND 0.25(REAL32)	produces a value of 0
INT32 TRUNC 0.75(REAL32)	produces a value of 0
INT32 TRUNC 0.25(REAL32)	produces a value of 0

Consider these examples, where **x**, and **y** are type **REAL32**, **x** has a value 3.5, **y** has a value 2.5.:

INT16 TRUNC y	produces the value 2, **y** truncated
INT16 ROUND y	produces the value 2, **y** rounded to nearest even
INT32 ROUND x	produces the value 4, **x** rounded to nearest even
INT16 TRUNC (x / y)	produces the value 1
(INT ROUND x) * 10	produces the value 40
REAL64 x	produces the value 3.5

A full explanation of the IEEE rounding modes is given in the appendix (page 98).

Type conversions may be needed if named types have been introduced to improve type abstraction in a program. For example if the types **AREA** and **LENGTH** (page 26) have been defined with real underlying types then it may be necessary to convert expressions of this type if they are to be used as actual parameters of calls to mathematical functions:

```
AREA a:
LENGTH b:
b := LENGTH ROUND (DSQRT (REAL64 a))
```

This example shows a named type being used as a conversion operator. As the conversion from **REAL64** to **LENGTH** may not be exact then a rounding operator must also be used.

Similarly if a named integer type has been defined then conversions will be needed in contexts which explicitly require a value of type **INT**. Consider:

```
DATA TYPE INDEX IS INT :
INDEX i :
VAL INDEX max IS 8192 :
[INT max]INT heap :   -- explicit conversion for array size
SEQ
  i := max - 3        -- 3 is an implicitly typed INDEX
  heap [INT i] := 4   -- explicitly converted INDEX
  ...
```

7 Procedures

This chapter describes *procedures* in occam. A procedure definition in occam defines a name for a process. Consider the following example:

```
PROC increment (INT x)
  x := x + 1
:
```

This example defines **increment** as the name for the process, **x := x + 1**. *Formal parameters* of a procedure are specified in parentheses after the procedure name. In this example, **x** is a formal parameter, and its specifier is the type **INT**. The procedure **increment** may be used as shown in the following example:

```
INT y :
SEQ
  ...
  increment (y)
  ...
```

A formal parameter is an *abbreviation* of the *actual parameter* used in an *instance* of a procedure. The type of the actual parameter matches the type of the corresponding formal parameter.

An *instance* of a procedure has the same effect as the substitution of the process named in the procedure's definition. This instance of **increment** can be expanded to show its effect:

```
INT y :
SEQ
  ...
  x IS y :                -- variable abbreviation
  x := x + 1
  ...
```

which is equivalent to

```
INT y :
SEQ
  ...
  y := y + 1
  ...
```

Here is a further example:

```
PROC writestr (CHAN OF BYTE stream, VAL []BYTE string)
  SEQ i = 0 FOR SIZE string
    stream ! string[i]
:
```

This procedure takes a channel (**stream**) and an array value (**string**) as parameters, and outputs the components of the array to the channel. Note the use of the operator **SIZE** (page 64) on the formal to obtain the number of components in the actual parameter when the specifier of a formal has an empty dimension.

An instance of the procedure **writestr** looks like this:

```
SEQ
  ...
  writestr (screen, "Hello world!")
  ...
```

The type of the second actual parameter is compatible (page 42) with the specifier of the formal parameter.

Again, this instance can be expanded to show the effect:

```
SEQ
  ...
  CHAN OF BYTE stream IS screen :          -- channel abbreviation
  VAL []BYTE string IS "Hello World!" :    -- value abbreviation
  SEQ i = 0 FOR SIZE string
    stream ! string[i]
  ...
```

To show the use of a user defined named type this example may be rewritten to use the type **DOSFNAME** defined on page 31, together with the use of the operator **SIZE** on such a type (page 65):

```
PROC writefname (CHAN OF BYTE stream, VAL DOSFNAME string)
  SEQ i = 0 FOR SIZE DOSFNAME
    stream ! string[i]
:
```

This procedure has the same effect as **writestr** but is restricted in its application to strings of type **DOSFNAME**.

An instance of the procedure **writefname** looks like this:

```
SEQ
  ...
  writefname (screen, "autoexec.bat"(DOSFNAME))
  -- the decoration (DOSFNAME) could have been omitted here!
  ...
```

Again, this instance can be expanded to show the effect:

```
SEQ
  ...
  CHAN OF BYTE stream IS screen :          -- channel abbreviation
  VAL DOSFNAME string IS "autoexec.bat" : -- value abbreviation
  SEQ i = 0 FOR SIZE DOSFNAME
    stream ! string[i]
  ...
```

The correspondence between formal parameters and actual parameters is defined in terms of abbreviations. There are three kinds of abbreviation: value abbreviations (page 43), variable abbreviations (page 42) and channel abbreviations (page 54) and procedure parameters may also be of these three kinds (they may also be timers, page 83).

If the formal is qualified by **VAL**, then it is a value parameter and the corresponding actual must be an expression as the abbreviation is a value abbreviation (page 43). If the formal is a data type not qualified by **VAL**, then the actual must be a variable which can receive a value by input or assignment as the abbreviation is a variable abbreviation (page 42).

A name which is *free* in the body of the procedure (page 40) is statically bound to the specification of the name in scope at the position of the procedure definition, for example:

```
INT step :
SEQ
  step := 39
  PROC next.item (INT next, VAL INT this)
    next := this + step
  :
  INT g, step :
  SEQ
    step := 7
    next.item (g, 3)
    ...       -- at this point the value of g is 42
```

The free variable **step**, in scope when the procedure **next.item** was defined, is *bound* to the occurrence of the name in the procedure **next.item**. The rules of occam ensure that distinct names identify distinct objects. The second declaration of a variable with the name **step** introduces a distinct new name. This means that in the example, the scope and binding of the variables can be seen more clearly by making systematic changes of name. Once this is done, the example is equivalent to:

```
INT step :
SEQ
  step := 39
  INT g, curb :  -- name step changed to curb
  SEQ
    curb := 7
    INT next IS g :  -- expand instance of next.item
    VAL INT this IS 3 :
    next := this + step
    ...          -- at this point the value of g is 42
```

In this transformation of the earlier example, it can be seen that the variable used in the instance of **`next.item`** is the variable named **`step`** declared before the procedure definition of **`next.item`**, and not the second variable declared with the same name.

Further rules for procedure parameters follow from those for abbreviations (see appendix E). The principal purpose of these rules is to ensure that at any one point in a program there is only one name that refers to any one variable. *Alias checking* is the name given to the checks performed by a compiler to ensure the absence of illegal *aliases*.

The rules for abbreviations (page 99) lead to restrictions on the actual parameters which may be used in procedure instances. Consider the procedure:

```
INT x, y, step :
PROC next.item (INT next, VAL INT this)
  next := this + step
:
```

And now consider the following equivalences of instances that may appear in the scope of the procedure:

`next.item (x, y)`	is equivalent to:	**`INT next IS x :`** **`VAL INT this IS y :`** **`next := this + step`**	
`next.item (x, step)`	is equivalent to:	**`INT next IS x :`** **`VAL INT this IS step :`** **`next := this + step`**	
`next.item (step, x)`	is equivalent to:	**`INT next IS step :`** **`VAL INT this IS x :`** **`next := this + step`**	which is **ILLEGAL!**
`next.item (x, x)`	is equivalent to:	**`INT next IS x :`** **`VAL INT this IS x :`** **`next := this + step`**	which is **ILLEGAL!**

Here it can be seen how the meaning of each procedure parameter is defined in terms of an abbreviation, the ordering of parameters corresponds to a sequence of abbreviations. **`next.item (step, x)`** is illegal because the variable **`step`** is used in the expression **`next := this + step`**, after it has been abbreviated, and the example **`next.item (x, x)`** is illegal as **`x`** has already been used in the previous abbreviation of the variable **`x`** (and the rules state that a variable used in such an abbreviation may not be used within the associated scope). Notice also the effect with the order of parameters used in **`next.item`** changed:

```
INT x, y, step :
PROC next.item (VAL INT this, INT next)
  next := this + step
:
```

With this re-ordering, **`next.item (x, x)`** is still illegal, although now for a different reason, as follows:

```
next.item (x, x)   is equivalent to:   VAL INT this IS x :   which is ILLEGAL!
                                       INT next IS x :
                                       next := this + step
```

`next.item (x, x)` is illegal here as there is an assignment to **`x`** (via **`next`**) within the scope of the first abbreviation. Now consider the following example:

```
PROC nonsense (INT x, VAL INT y)
  SEQ
    x := x + y
    x := x - y
:
```

This procedure should leave the value of the variable used as the actual parameter for **`x`**, unchanged, as the following expansion shows:

```
nonsense (n, 3)   is equivalent to:     INT x IS n :
                                        VAL INT y IS 3 :
                                        SEQ
                                          x := x + y
                                          x := x - y

                  and by substitution:  SEQ
                                          n := n + 3
                                          n := n - 3
```

The value of **`n`** after this instance is **`n`**, as might be expected. However, the following instance is illegal, which is just as well, as the effect is non-intuitive:

```
nonsense (n, n)   is equivalent to:     INT x IS n :         which is ILLEGAL!
                                        VAL INT y IS n :
                                        SEQ
                                          x := x + y
                                          x := x - y

                  and by substitution:  SEQ                  a non-intuitive effect!
                                          n := n + n
                                          n := n - n
```

The value of **`n`** after this instance, if it were legal, would be 0, which is counterintuitive. The following example highlights the problem further.

```
nonsense (i, v[i])   is equivalent to:     INT x IS i :           which is ILLEGAL!
                                           VAL INT y IS v[i] :
                                           SEQ
                                             x := x + y
                                             x := x - y

                     and by substitution:  SEQ                    a non-intuitive effect!
                                             i := i + v[i]
                                             i := i - v[i]
```

If this instance were legal, the value of **`i`** after the instance of **`nonsense`** would be difficult to predict, as in each of the assignments **`v[i]`** will probably reference a different component of **`v`**, as the value of the subscript **`i`** may be changed by the first assignment.

The syntax for a procedure definition is:

definition	=	**PROC** *name* **(** $\{_0$ **,** *formal* $\}$ **)**
		process
		:
formal	=	*specifier* $\{_1$ **,** *name* $\}$
	|	**VAL** *specifier* $\{_1$ **,** *name* $\}$

The keyword **PROC**, the name of the procedure, and a formal parameter list enclosed in parentheses is followed by a process, indented two spaces, which is the body of the procedure. Any data type may be used as a specifier following **VAL**, any type as a specifier otherwise. The procedure definition is terminated by a colon which appears on a new line at the same indentation level as the start of the definition.

The syntax for a procedure instance is:

proc.instance	=	*name* **(** $\{_0$ **,** *actual* $\}$ **)**
actual	=	*variable*
	|	*channel*
	|	*timer*
	|	*expression*
process	=	*proc.instance*

An instance of a procedure is the procedure name followed by a list of zero or more actual parameters in parentheses. An actual parameter is a variable, channel, timer or expression. The list of actual parameters must correspond directly to the list of formal parameters used in the definition of the procedure. The actual parameter list must have the same number of entries, each of which must be compatible with the kind (**VAL** or non-**VAL**) and type of the corresponding formal parameter. In a program in which all names are distinct, an instance of a procedure behaves like the substitution of the procedure body. Notice that all programs can be expressed in a form in which all names are made distinct by systematic changes of name. A channel parameter or free channel may only be used for input or output (not both) in the procedure.

Procedures in occam cannot be recursive as the procedure name is not defined until the end of its body and so cannot itself be used inside it. It is, however, possible to produce the effect of recursion, to a fixed maximum depth, by means of multiple definitions of the same procedure, in each of which a reference to the procedure name identifies the immediately preceding definition.

An instance of a procedure defined with zero parameters must be followed by empty parentheses. Where a number of parameters of the same type appear in the formal parameter list, a single specifier may specify several names. For example:

```
PROC snark (VAL INT butcher, beaver, LENGTH boojum, jubjub)
  ...
:
```

This example, is equivalent to:

```
PROC snark (VAL INT butcher, VAL INT beaver,
            LENGTH boojum,   LENGTH jubjub)
  ...
:
```

The optimisation of procedure instances by generation of inline code is discussed in appendix A.

8 Functions

The previous chapter discusses named processes (called *procedures*). This chapter describes *functions* in occam. A function defines a name for a special kind of process, called a *value process*. A value process produces a result of any data type, and may appear in expressions. Value processes may also produce more than one result, which may be assigned in a multiple assignment. occam functions are free from all side effects, as they are forbidden to communicate or to assign to free variables. This helps to ensure that programs are clear and easy to maintain.

8.1 Value processes

A value process performs an enclosed process and produces a result. Consider the following example:

```
total := subtotal + (INT sum :
                     VALOF
                       SEQ
                         sum := 0
                         SEQ i = 0 FOR SIZE v
                           sum := sum + v[i]
                       RESULT sum
                    )
```

In the example shown here, the value process produces the sum of the array **v**, and is equivalent to

$$\sum_{i=0}^{(SIZE v)-1} v[i]$$

The syntax of value processes is:

value.process	=	**VALOF**
		  process
		  RESULT *expression.list*

	|	*specification* *value.process*
operand	=	**(** *value.process* **)**
expression.list	=	**(** *value.process* **)**

A value process consists of zero or more specifications which precede the keyword **VALOF**, followed by a process at an indentation of two spaces, and the keyword **RESULT** at the same indentation. The keyword **RESULT** is followed by an expression list on the same line. The expressions in this list may be of any fixed size data type. The line may be broken after a comma, or at a legal point in an expression. An operand of an expression may consist of a left parenthesis, a value process, and a right parenthesis. The structured parentheses appear at the same indentation as each other, and are equivalent to the left hand and right hand parentheses of a bracketed expression respectively. So, where the value process produces a single result, the upper bracket may be preceded by an operator, or the lower bracket may be followed by an operator.

8.2 Functions

More commonly the value process is the body of a function definition, as illustrated in the following example:

```
INT FUNCTION sum (VAL []INT values)
  INT accumulator :
  VALOF
    SEQ
      accumulator := 0
      SEQ i = 0 FOR SIZE values
        accumulator := accumulator + values[i]
    RESULT accumulator
:
```

This function definition defines the name **sum** for the associated value process. The type of the result of this function is **INT**. The result type or types, which may be any explicit or named fixed size data types appear in a comma separated list before the keyword **FUNCTION**.

Just as the behaviour of procedure instances is defined by the substitution of the procedure body, instances of functions behave like a substitution of the function body. In fact the value process is introduced into the language for the purpose of providing a clean definition of function instances in terms of substitution, rather than as a construct that will normally be used in its own right. It follows that the example which starts this chapter is an expansion of the following:

```
total := subtotal + sum (v)
```

A function definition may also define a name for an expression list which does not need a value process for its evaluation, so that simple, single line functions can be defined in the following fashion:

```
BYTE FUNCTION noparity (VAL BYTE ch) IS (ch /\ #7F) :
BOOL FUNCTION lowercase (VAL BYTE ch) IS (ch >= 'a') AND (ch <= 'z') :
BOOL FUNCTION uppercase (VAL BYTE ch) IS (ch >= 'A') AND (ch <= 'Z') :
BOOL FUNCTION isletter (VAL BYTE ch) IS uppercase (ch) OR lowercase (ch) :
```

Each of these functions returns a single byte or boolean result. The definition of the function **isletter** is equivalent to the following:

```
BOOL FUNCTION isletter (VAL BYTE ch)
  VALOF
    SKIP
    RESULT uppercase (ch) OR lowercase (ch)
:
```

A number of rules apply to functions to ensure they are free from side effects. As for procedures, the correspondence between the formal and actual parameters of a function is defined in terms of *abbreviations*, and follows the associated scope rules. However, an argument of a function may only be a value parameter. Only variables declared within the body of a value process or function may be assigned to. There may be no parallels within a value process. There can be no inputs, outputs or alternations within a value process. The evaluation of a function can therefore never be explicitly delayed by the action of another process running concurrently.

Any procedure used within a function must also be free from side effects on variables outside the function. A variable which is free within the result list or value process (Scope, page 39) can be used only in expressions within the value process or function body; it may not receive new values by assignment. Consider the following:

```
[]THING stack:
INT stack.pointer:
SEQ
  ...
  THING FUNCTION read.top.of.stack () IS stack[stack.pointer] :
  BOOL FUNCTION empty () IS (stack.pointer = 0) :
```

stack and **stack.pointer** are free in the definition of **read.top.of.stack** and **stack.pointer** in the definition of **empty**, but can be used in expressions in those definitions.

A value process may produce more than one result, which may then be assigned using a multiple assignment. Consider the following example:

```
point, found := (VAL BYTE char IS 'g' :
                  VAL []BYTE string IS message :
                  BOOL ok :
                  INT ptr :
                  VALOF
                    IF
                      IF i = 0 FOR SIZE string
                        string[i] = char
                          SEQ
                            ok := TRUE
                            ptr := i
                      TRUE
                        SEQ
                          ok := FALSE
                          ptr := -1
                    RESULT ptr, ok
                 )
```

This value process searches the byte array **string** for the character **'g'**. The result is produced from the expression list which follows **RESULT**, and is then assigned to **point**, and **found**. This value process can be given a name in a function definition, as follows:

```
INT, BOOL FUNCTION instr (VAL BYTE char, VAL []BYTE string)
  BOOL ok :
  INT ptr :
  VALOF
    IF
      IF i = 0 FOR SIZE string
        string[i] = char
          SEQ
            ok := TRUE
            ptr := i
      TRUE
        SEQ
          ok := FALSE
          ptr := -1
    RESULT ptr, ok
:
VAL message IS "Twas brillig and the slithy toves" :
INT point :
BOOL found :
SEQ
  point, found := instr ('g', message)
  ...
```

This example finds the position of **'g'** in the string **message**. After the multiple assignment in this example, the value of **point** will be **11**, and the value of **found** will be **TRUE**.

Single line functions with multiple results may also be defined:

```
INT, INT FUNCTION div.rem (VAL INT x, y) IS x / y, x REM y :
```

This function produces the quotient and remainder when **x** is divided by **y**. If an error occurs within a function or value process, it will behave like the primitive process **STOP**. This behaviour is equivalent to the behaviour of a numerical overflow in an arithmetic expression (see page 101 for details of the behaviour of invalid processes). Consider the behaviour of an instance of the following function:

```
INT FUNCTION factorial (VAL INT n)
  INT product :
  VALOF
    SEQ
      product := 1
      SEQ i = 1 FOR n
        product := product * i
    RESULT product
:
```

This function will behave like the primitive process **STOP** if **n** is less than zero, or if an overflow occurs in the evaluation of the factorial. In either case the behaviour is equivalent to the behaviour of any other in valid expression (page 101).

The syntax for functions is:

function.header	=	**FUNCTION** *name* **(** $\{_0$ **,** *formal* $\}$**)**
definition	=	$\{_1$ **,** *data.type* $\}$ *function.header* *value.process* **:**
	|	$\{_1$ **,** *data.type* $\}$ *function.header* **IS** *expression.list* **:**
operand	=	*name* **(** $\{_0$ **,** *expression* $\}$ **)**
	|	*operand* **[** *subscript* **]**
expression.list	=	*name* **(** $\{_0$ **,** *expression* $\}$ **)**

A function header consists of the keyword **FUNCTION**, followed by the name of the function and a formal parameter list enclosed in parentheses. A function definition consists of a comma separated list of the types of the result(s) produced by the function, followed by a function header. This is followed by a value process, indented two spaces, which forms the body of the function. The function definition is terminated by a colon which appears on a new line at the same indentation level as the start of the definition. Alternatively, a function definition may consist of the result type list and function header followed by the keyword **IS**, an expression list, and a colon, on the same line. The line may be broken after the keyword **IS**, a comma, or at a legal point in an expression.

An instance of a function is an operand and consists of a function name followed by a list of actual parameter expressions in parentheses. An operand that is a function instance or value process returning a single result of an array or record type may be subscripted by an expression or a field name respectively. An instance of a function defined to have zero parameters must be followed by empty parentheses. Where a number of parameters of the same type appear in the formal parameter list, a single specifier may specify several names. For example, using the type **COMPLEX32** declared on page 31:

```
INT FUNCTION alice (VAL COMPLEX32 tweedle.dum, tweedle.dee,
                    VAL INT cheshire.cat)
  ...
:
```

This example is equivalent to:

```
INT FUNCTION alice (VAL COMPLEX32 tweedle.dum,
                    VAL COMPLEX32 tweedle.dee,
                    VAL INT cheshire.cat)
  ...
:
```

A function may return results of a record type such as **COMPLEX32**. For example the multiplication of complex numbers may be expressed by this function:

```
COMPLEX32 FUNCTION cmul (VAL COMPLEX32 z1, z2)
  COMPLEX32 z :
  VALOF
    SEQ
      z[real] := (z1[real] * z2[real]) - (z1[imag] * z2[imag])
      z[imag] := (z1[real] * z2[imag]) + (z1[imag] * z2[real])
    RESULT z
:
COMPLEX32 w1, w2, w3 :
SEQ
  ...
  w1 := cmul (w2, w3)
  ...
```

The final example of a function shows both the use of a named array type as a result type and the use of two results to return two values. An explicit array type could have been used here instead of a named one. Given an array of program names separated by commas this function will select one of these names, append **".exe"** to it and return it with its length:

```
INT, DOSFNAME FUNCTION exename (VAL[]BYTE names, VAL INT j)
  -- extract (j-1)'th name from comma separated names and append ".exe"
  DOSFNAME fname :
  INT n :
  VALOF
    INT m, s :
    SEQ
      m, s := 0, 0
      SEQ i = 0 FOR j
        SEQ
          WHILE names [m] <> ','
            m := m + 1
          s := m + 1                -- names [s] starts next name
          m := s
      WHILE isletter (names [m])    -- instance of FUNCTION isletter
        m := m + 1
      n := m - s                    -- n is length of j'th name
      [fname FOR n] := [names FROM s FOR n]
      [fname FROM n FOR 4] := ".exe"
    RESULT n + 4, fname
:
```

Use of this function to select a name from a list and then use of that name as a parameter to a procedure is shown in this example:

```
VAL prognames IS "oc,occonf,ilink,icollect,imakef,":
DOSFNAME tool :
INT len :
SEQ
  len, tool := exename (prognames, 2)
  callprog ([tool FOR len])
```

Note that the expression list returned as a function result cannot be used directly as an actual parameter list, but must be assigned to a variable list (or used after **IS** or **RESULT** in another function definition).

The optimisation of function instances by generation of inline code is discussed in appendix A.

9 Timers

Timers produce a value which represents the time, and allow processes to be delayed until the time has reached or passed a particular value. The use of timers is essential in most real time control systems.

This chapter describes timers, the declaration of timers, and access to them.

Syntactically timers have many similarities to channels (discussed on page 45).

A timer provides a clock which can be accessed by any number of concurrent processes. The relationship between the time returned by an occam timer and real time is not defined by the language, i.e. implementations may differ in the granularity of timers and consequently in their cycle period.

9.1 Timer type

The type of a timer is:

timer.type	=	**TIMER**

Timer arrays have type similar to channel arrays, for example:

```
[10]TIMER
```

The syntax of timer array types is:

timer.type	=	**[** *expression* **]** *timer.type*

9.2 Declaring a timer

A timer is declared in a manner similar to channels and variables. Consider the following example:

```
TIMER clock :
```

This declaration introduces a timer which is identified by the name **clock**. Several timers may be declared together, for example:

```
TIMER clockA, clockB :
```

The type of the declarations is determined, and then the declarations are made. Timer arrays are declared in the same way as other arrays, for example:

```
[10]TIMER clocks :
```

Components and segments of timer arrays are denoted in the same way as components and segments of variable arrays (page 36) and channel arrays (page 46).

The syntax of timer declarations is:

declaration	=	*timer.type* $\{_1$ **,** *name*$\}$ **:**
timer	=	*name*
	|	*timer* **[** *expression* **]**
	|	**[** *timer* **FROM** *base* **FOR** *count* **]**
	|	**[** *timer* **FROM** *base* **]**
	|	**[** *timer* **FOR** *count* **]**

A value input from a timer provides an integer value of type **INT** representing the time. The value is derived from a clock, which changes by an increment at regular intervals. The value of the clock is cyclic (*ie* when the value reaches the most positive integer value, an increment results in the most negative integer value). The special operator **AFTER** can be used to compare times even though the value may have crossed from most positive to most negative, just as one o'clock *pm* may be considered later than eleven o'clock *am*. If **t1** and **t2** are successive inputs from the same timer, then the expression **t1 AFTER t2** is true if **t1** is later than **t2**. This behaviour is only sensible if the second value (**t2**) is input within half a cycle of the timer. **AFTER** is also explained in the chapter on expressions (page 57).

The rate at which a timer is incremented is implementation dependent, and may depend on the priority at which a process is run (see page 94).

9.3 Timer input

Timers are accessed by special forms of *input* called *timer inputs*, which are syntactically similar to channel inputs, for example:

```
clock ? t
```

This example inputs a value from the timer **clock** and assigns the value to the variable **t**. Unlike channels, inputs from the same timer may appear in any number of components of a parallel.

Another special timer input (called a *delayed input*) specifies a time, after which the input terminates, for example:

```
clock ? AFTER t
```

This input waits until the value of the timer **clock** is later than the value of **t**. In other words, if **c** is the value of the timer **clock**, then the input will wait until **(c AFTER t)** is true. The value of **t** is unchanged.

A delay can be caused by this sequence:

```
SEQ
  clock ? now
  clock ? AFTER now PLUS delay
```

The first input inputs a value representing the current time and assigns it to the variable **now**. The second (delayed) input waits until the value input from **clock** is later than the value of **now PLUS delay**. **PLUS** (page 61) is a *modulo operator*. Note that because of the cyclic nature of timers it is important to use **PLUS** and **MINUS** when adding and subtracting integers derived from a timer.

The syntax for timer inputs is:

input	=	*timer.input*
	|	*delayed.input*
timer.input	=	*timer* **?** *variable*
delayed.input	=	*timer* **? AFTER** *expression*

A timer input receives a value from the timer named on the left of the input symbol (**?**), and assigns that value to the variable named on the right of the symbol. A delayed input waits until the value of the timer named on the left of the input symbol (**?**) is later than the value of the expression on the right of the keyword **AFTER**.

9.4 Timers in alternations

Timer inputs and delayed inputs may be used as guards in alternations. This gives a simple way in which to program timeouts wherein a process waits for one or more inputs on communication channels for up to some maximum time and performs some other action if no input was received. Consider the process:

```
SEQ
  to.server ! request
  time ? request.time
  ...
  ALT
    from.server ? reply
      ...  the server has replied in time
    time ? AFTER request.time PLUS time.out
      ...  the server has missed the deadline
```

In this example, the process sends a request to a server and notes the time at which the request was sent. When the process is ready to receive the reply, it waits alternatively for the server to become ready with the reply or for the timeout period to pass. If the server has not become ready to reply before the end of the timeout period, then the process will execute the branch of the alternation associated with the delayed input. Notice that the timeout period starts from the time of the timer input following the request, not from the beginning of the alternation.

9.5 Timer abbreviation

Timers may be abbreviated in the same way as variables (page 42) and channels (page 54). The same rules, summarised in appendix E, apply to abbreviated timer names as apply to abbreviated variable or channel names. Timers may be used as actual parameters in procedure instances.

The syntax of timer abbreviation is

abbreviation	=	*specifier name* **IS** *timer* **:**
	|	*name* **IS** *timer* **:**
specifier	=	*timer.type*
	|	**[]** *specifier*
	|	**[** *expression* **]** *specifier*
actual	=	*timer*

This syntax is closely related to the corresponding syntax for channels.

10 Retyping and reshaping

This chapter describes retyping conversions. A retyping conversion is syntactically a specification (a definition) in whose scope the data type of a bit pattern is changed from one data type to another. There are three kinds of retyping conversions: conversions which convert a variable, conversions which convert the value of an expression, and conversions which change the protocol of a channel. There is also the related reshaping conversion.

The length (i.e. the number of bits) of the new type specified must be the same as the length of the bit pattern being retyped. A retyping conversion has no effect upon the bit pattern, and differs from *type conversion* (page 66) where the value of one type is converted into an equivalent value of another type.

10.1 Retyping variables and values

As retyping conversions are dependent on the exact mapping of types in the computer, the word size, the significance of the order of bytes in a word, etc., their use can lead to programs which are not portable between different target computer types. See also the discussion of word size on page 92 in appendix A.

The retyping conversion of a value may be used to specify a name for a particular bit pattern described by a hexadecimal constant. Consider the following example:

```
VAL REAL32 root.NaN RETYPES #7F840000(INT32) :
```

The advantage of the above conversion is that it has been possible to specify the exact representation of a value otherwise difficult to represent. Use of this conversion may be dependent on the target machine's use of IEEE floating point arithmetic. Consider also the following example:

```
VAL INT64 pattern RETYPES 42.0(REAL64) :
```

The bit pattern for the real representation of the value 42.0 is mapped to a name **pattern** of type **INT64**. As for the *abbreviation* (page 43) of expressions, no variable used in the expression may receive a new value by input or assignment within the scope of the conversion.

The retyping conversion may also specify a name of a new type for an existing variable of the same length. For example:

```
INT64 condition :
...
SEQ
  [8]BYTE state RETYPES condition :
  ...
```

In this example, **condition**, a variable of type **INT64**, is made accessible as an array of 8 bytes. Each byte is accessible via subscript, any change to the bit pattern as a result of an assignment or input will directly affect the value of the original variable.

The same rules apply to names specified by retyping conversions as apply to abbreviations (page 99). That is, no variable used in a subscript, base or count expression which selects a component or segment of an array may receive a new value by input or assignment within the *scope* (page 39, the region of a program where use of the name is legal) of the conversion. The variable converted may not be used within the scope of the conversion.

The syntax for retyping conversion is:

definition	=	*specifier name* **RETYPES** *variable* **:**
	|	**VAL** *specifier name* **RETYPES** *expression* **:**

A retyping conversion is syntactically a definition. The retyping conversion of a variable begins with a specifier, followed by the name specified, and the keyword **RETYPES**, the variable appears to the right of the keyword **RETYPES**. The retyping conversion of a value begins with the keyword **VAL**, a data type specifier appears to the right of **VAL**, followed by the name specified, and the keyword **RETYPES**; the expression appears to the right of the keyword **RETYPES**. The number of bytes in the representation of values of the type of the specifier must be the same as in the variable or value retyped. The line on which a retyping conversion occurs may be broken after the keyword **RETYPES**, or at any legal point in the expression.

A retyping conversion where the destination type is an array type may use a specifier with one empty dimension. The size of this dimension is determined from the size of the variable or value being retyped. For example:

```
[10]REAL64 w :
SEQ
  []BYTE bw RETYPES w :
  -- here we can send the binary array to a file
  ...
  [10][]INT iw RETYPES w :
  -- here we can look at each REAL as a bit pattern
  ...
```

If a variable being retyped is implemented with different alignment requirements to the destination type then the retyping conversion is invalid unless it is determined that alignment mismatches do not occur. This can be important when using retyping to pack binary values into a buffer for sending to a file. Consider the following example:

```
[4096]BYTE diskbuff :
INT di :
DATA TYPE MYINT IS INT :
MYINT x :
SEQ
  ...
  MYINT db0 RETYPES [diskbuff FOR BYTESIN (MYINT)] :
  --  ok because start of array always aligned
  di := x
  []BYTE bx RETYPES x :
  [diskbuff FROM di FOR SIZE bx] := bx
  ...
  MYINT dbi RETYPES [diskbuff FROM di FOR BYTESIN (MYINT)] : -- INVALID!
  -- fails because value of di might cause misaligment
  ...
```

An implementation may reject an invalid definition as above at compile time or may cause the generation of code to reject it at run time.

10.2 Retyping channels

The protocol of a channel is used by a compiler to check the format and content of messages that are communicated by inputs and outputs. Sometimes it is necessary to allow different protocols to be used for communications using the same channel at different places within a program.

This may be achieved by using a retyping conversion applied to channels. Consider this example:

```
PROC real.buffer (CHAN OF REAL32 rin, rout)
  REAL32 r :
  WHILE TRUE
    SEQ
      rin ? r
      rout ! r
:
PROTOCOL COMPLEX IS REAL32; REAL32 :
PROC generate.complex (CHAN OF COMPLEX c)
  ...  process outputting complex values on c
:
PROC consume.complex (CHAN OF COMPLEX c)
  ...  process inputting complex values on c
:
CHAN OF COMPLEX cin, cout :
PAR
  generate.complex (cin)
  CHAN OF REAL32 crin RETYPES cin :
  CHAN OF REAL32 crout RETYPES cout :
  real.buffer (crin, crout)
  consume.complex (cout)
```

This is a trivial example of a situation that sometimes arises in occam programming. The procedure **real.buffer** represents an existing general purpose procedure that it is not convenient to rewrite. This happens to buffer single values. Similarly the procedure **generate.complex** has been written to output complex numbers as pairs of values using the protocol **COMPLEX**. As this is a sequential protocol each value will be sent as a single communication and so will match one input in the buffer procedure. The use of channel retyping allows these separately written procedures to be called without changing either.

If channel retyping is used in only one of the processes that use a channel as here, then it is important to ensure that individual inputs and outputs remain compatible in terms of the numbers of bytes in each communication. Channel retyping may also be used to change the protocol both in the process that outputs on a channel and the process which inputs on the channel. In this case there is no need to take special care about the details of implementation of communication.

A common use of channel retyping is to apply protocol checking to the use of one or both ends of a channel that was declared with an anarchic protocol (see page 53).

The syntax of channel retyping is similar to variable retyping:

definition = *specifier name* **RETYPES** *channel* **:**

The specifier must be of the form **CHAN OF** protocol. Channel arrays may be retyped or reshaped (see below).

10.3 Reshaping

A special kind of retyping conversion may be used to change the interpretation of an array in terms of its subdivision into arrays of fewer dimensions. A reshaping conversion may increase or decrease the number of dimensions or may merely change the sizes of the dimensions. A reshaping conversion is independent of details of representation and therefore programs using them are portable to different target computer architectures. Consider this example:

```
VAL [2][6]REAL32 nums IS [[0.0, 1.0, 2.1, 3.1, 4.2, 5.2]
                          [6.3, 7.3, 8.4, 9.4, 10.5, 11.5]] :
SEQ
  --  here we want to use two arrays of six reals each
  ...
  VAL [6][2]REAL32 nums.6.2 RESHAPES nums :
  SEQ
    --  here we want to use six arrays of two reals each
    ...
  VAL [12]REAL32 nums.12 RESHAPES nums :
  SEQ
    --  here we want to use one array of twelve reals
    ...
  VAL [2][2][]REAL32 nums.2.2. RESHAPES nums :
  SEQ
    --  here we want a 3-dimensional subdivision
    ...
```

In all the reshapings shown the number of components of the array base type on the two sides of the keyword **RESHAPES** must be the same. One pair of brackets in the specifier on the left may be empty, and such a reshaping is valid if and only if there is an integer such that the product of the sizes of all dimensions on the left is equal to the number of base type components in the array on the right. The empty brackets may be in any position. In the example shown the type of **nums.2.2.** will be found by the compiler to be **[2][2][3]REAL32**.

The syntax for reshaping conversion is:

definition	=	*specifier name* **RESHAPES** *variable* **:**
	|	**VAL** *specifier name* **RESHAPES** *expression* **:**
	|	*specifier name* **RESHAPES** *channel* **:**

A reshaping conversion of a variable begins with a specifier for an array type, followed by the name specified, and the keyword **RESHAPES**. The variable which must be an array built from components of the same base type as the specifier appears to the right of the keyword **RESHAPES**.

A reshaping conversion of a value begins with the keyword **VAL**, a data type specifier appears to the right of **VAL**, followed by the name specified, and the keyword **RESHAPES**. The expression which must be of an array type built from components of the same base type as the specifier appears to the right of the keyword **RESHAPES**.

A reshaping conversion of a channel begins with a specifier for a channel array type, followed by the name specified, and the keyword **RESHAPES**. The channel which must be an array built from components of the same base type as the specifier appears to the right of the keyword **RESHAPES**.

The line on which a conversion occurs may be broken after the keyword **RESHAPES**, or at any valid point in the variable or expression. Specifiers appearing in reshaping conversions may contain one empty pair of brackets (open dimension).

Appendices

A Implementation dependent features

As far as possible an implementation of occam should place no unreasonable quantitative restrictions on the components of an occam program. If any restrictions that impact the class of acceptable occam programs are imposed by a compiler then they must be described in accompanying documentation. Examples of such limits and restrictions would be those concerning the length of names, length of formal parameter lists, number of tags in a protocol or requirements that certain constants be known at compile time.

An implementation of occam may need to extend the language in various ways to facilitate the solution of common software engineering problems such as source code management, optimisations and interfacing to other software and special purpose hardware. Such extensions should always obey the spirit of occam in keeping things as simple as possible and should as far as possible be limited to structures introduced by explicit keywords easily recognised by a compiler or human reader.

Directives to a compiler which can occupy a line on their own should always do so and be introduced by a keyword whose first character is the special character **#**. This should be the first lexical item on a line and should be allowed at any indentation consistent with the surrounding occam source code. Allowable indentations match those for comments (page 4). Any such compiler directive line must be allowed to end with an occam comment, and apart from its interpretation at compile time is treated as a comment.

These keywords and others which need to be embedded within occam source should all obey the convention that no keyword contains a lower case letter.

It is not the purpose of this appendix to describe any such language extensions in detail, but to mention some of those that have been found useful in existing implementations, and so reduce the likelihood of new extensions being devised for similar purposes when existing ones could sensibly be reused. Readers should consult detailed documentation accompanying a particular compiler for details of the syntax and semantics of these extensions.

A.1 Compiler directives

From the strict syntactic point of view compiler directives behave as comments and may appear wherever a comment can appear. They are interpreted by the compiler. Particular directives may be allowed by a compiler only in a more restricted set of positions, and their indentation may be significant if they introduce additional specifications, whose scope they will determine.

#INCLUDE introduces (by name in some filing system or operating system name space) a source text file to be inserted at the current source line position and indentation.

#USE introduces an object file or library file from which interface information from a pre-compiled body of occam code is to be inserted at the current source line position and indentation.

#OPTION introduces a list of compilation options.

#PRAGMA introduces further information that may influence what a compiler does to the source text during compilation. Versions of this directive have also been used to define interfacing to code compiled from languages other than occam, selective control of usage checking, etc.

#COMMENT introduces a text string that is to be explicitly copied to an object file. This may be needed to ensure that copyright etc is identified.

A.2 Special keywords introducing language extensions

INLINE

This keyword may be inserted before **PROC** or **FUNCTION** to suggest to the compiler that calls to the routine should be compiled to inline code rather than to a closed subroutine.