**XDS Family of Products**

# ISO Modula-2 Library Reference


http://www.excelsior-usa.com

# Contents

# Chapter 1

# Input/Output

The input/output library defined in this chapter provides facilities for reading and writing of data streams over one or more *channels*. Channels are connected to sources of input data, or to destinations of output data, known as *devices* or *device instances*. There is a separation between modules that are concerned with device-independent operations, such as reading and writing, and modules concerned with device-dependent operations, such as making connections to named files. This separation allows the library to be extended to work with new devices. The module structure of the library is depicted in the following figure.

**The figure will be available in the final release**

Channels already open to standard sources and destinations can be identified using procedures provided by the module `StdChans`. This module also provides for the identification and selection of channels used by default for input and output operations.

The modules `TextIO`, `WholeIO`, `RealIO`, and `LongIO`, provide facilities that allow the reading and writing of high-level units of data, using *text operations* on channels specified explicitly by a parameter. These high-level units include characters, strings, and whole numbers and real numbers in decimal notation. The module `RawIO` provides facilities for reading and writing of arbitrary data types, using *raw (binary) operations* on explicitly specified channels.

Text operations produce or consume data streams as sequences of characters and line marks. Raw operations produce or consume data streams as sequences of storage locations (i.e. as arrays whose component type is `SYSTEM.LOC`).

The library allows devices to support both text and raw operations on a single channel, although this behaviour is not required.

The module `IOResult` provides the facility for a program to determine whether the last operation to read data from a specified input channel found data in the required format.

Corresponding to the `TextIO` group of modules is a group of modules `STextIO`, `SWholeIO`, `SRealIO`, `SLongIO`, `SRawIO` and `SIOResult`. The prefix `"S"` serves as an abbreviation for `"Simple"`. The procedures exported from this group do not take parameters identifying a channel. They operate on the default input and output channels, as identified by the module `StdChans`.

The module `IOConsts` defines types and constants used by `IOResult` and `SIOResult`.

The device modules `StreamFile`, `SeqFile`, `RndFile`, and `TermFile` provide facilities that allow a channel to be opened to a named stream, to a rewindable sequential file, to a random access file, or to a terminal device respectively. The device module `ProgramArgs` provides an open channel from which program arguments may be read. Device specific operations, such as positioning within a random access file, are also defined by the appropriate device module.

The module `ChanConsts` defines the constants and types used in those device module procedures that open channels.

The primitive device-independent operations on channels are provided by the module `IOChan`.

The module `IOChan` defines general input/output library exception values that may be raised when using any device through a channel. Device errors, such as a hardware read/write error, are reported by raising one of the general exception values, and providing an implementation-defined error number. Exception values associated with device- specific operations are defined by the appropriate device module.

The module `IOLink` provides facilities that allow a user to provide further specialized device modules for use with channels, following the pattern of the rest of the library.

**NOTE:**

Partial implementations of the input/output library may provide modules selected exclusively from the group `STextIO`, `SWholeIO`, `SRealIO`, and `SLongIO`, normally with `SIOResult` and `IOConsts`. If any other module is provided, the module `IOChan` must also be provided, in accordance with the import dependencies between the definition modules of the library.

## 1.1 Standard and Default Channels

*Standard channels* do not have to be opened by a client program since they are already open and ready for use. Under some operating systems they may be connected to sources and destinations specified before the program is run, while on a stand-alone system they may be connected to a console terminal.

No method is provided for closing a standard channel, and the values used to identify standard channels are constant throughout the execution of the program.

*Default channels* are channels whose identities have been stored as those to be used by default for input and output operations. Initially these correspond to the standard channels, but their values may be varied to obtain the effect of redirection.

### 1.1.1 Module StdChans

The module `StdChans` defines functions that identify channels already open to implementation-defined sources and destinations of standard input, standard output, and standard error output. Access to a 'null device' is provided to allow unwanted output to be suppressed. The null device throws away all data written to it, and gives an immediate end of input indication on reading.

The module `StdChans` provides procedures for identification and selection of the channels used by default for input and output operations.

**ChanID** *Channel identity*

```
TYPE
  ChanId = IOChan.ChanId;
```

The type `IOChan.ChanId` which is used to identify channels is reexported.

**StdInChan** *Get standard input channel id*

```
PROCEDURE StdInChan (): ChanId;
```

The function procedure `StdInChan` returns a value identifying a channel open to the implementation-defined standard source for program input.

**StdOutChan** *Get standard output channel id*

```
PROCEDURE StdOutChan (): ChanId;
```

The function procedure `StdOutChan` returns a value identifying a channel open to the implementation-defined standard destination for program output.

**StdErrChan** *Get standard error channel id*

```
PROCEDURE StdErrChan (): ChanId;
```

The function procedure `StdErrChan` returns a value identifying a channel open to the implementation-defined standard destination for program error messages.

**NullChan** *Get null device channel id*

```
PROCEDURE NullChan (): ChanId;
```

The function procedure `NullChan` returns a value identifying a channel open to the null device.

**NOTE:**

The null device supports all operations by discarding all data written to it, or by giving an immediate end of input indication on reading.

**InChan** *Get current default input channel id*

```
PROCEDURE InChan (): ChanId;
```

The function procedure `InChan` returns the identity of the current default input channel. This is the channel used by input procedures that do not take a channel parameter. Initially this is the value returned by `StdInChan`.

**OutChan** *Get current default output channel id*

```
PROCEDURE OutChan (): ChanId;
```

The function procedure `OutChan` returns the identity of the current default output channel. This is the channel used by output procedures that do not take a channel parameter. Initially this is the value returned by `StdOutChan`.

**ErrChan** *Get current default error channel id*

```
PROCEDURE ErrChan (): ChanId;
```

The function procedure `ErrChan` returns the identity of the current default output channel for program error messages. Initially this is the value returned by `StdErrChan`.

**SetInChan** *Set current default input channel*

```
PROCEDURE SetInChan (cid: ChanId);
```

The procedure `SetInChan` sets the current default input channel to that identified by `cid`.

**SetOutChan** *Set current default output channel*

```
PROCEDURE SetOutChan (cid: ChanId);
```

The procedure `SetOutChan` sets the current default output channel to that identified by `cid`.

**SetErrChan** *Set current default output channel*

```
PROCEDURE SetErrChan (cid: ChanId);
```

The procedure `SetErrChan` sets the current default output channel for error messages to that identified by `cid`.

## 1.2 Reading and Writing of Data

The module `TextIO` provides facilities for input and output of characters, character strings, and line marks, using text operations.

The module `WholeIO` provides facilities for input and output of whole numbers in decimal text form.

The modules `RealIO` and `LongIO` provide facilities for input and output of real numbers in decimal text form.

The module `RawIO` provides facilities for direct input and output of data, using raw operations (i.e. without any interpretation).

The input procedures of the modules `TextIO`, `WholeIO`, `RealIO`, `LongIO`, and `RawIO` are sufficient for use where the format of the input data is known. Since, in practice, their use may be inconsistent with the format of the input data, they have the effect of setting a *'read result'* for the used channel. The module `IOResult` provides the facility for obtaining the read result applicable to the most recent input operation on a given channel.

In all cases, channels are selected explicitly by passing an actual parameter of the type `ChanId` to the procedures of these modules.

The modules `STextIO`, `SWholeIO`, `SRealIO`, `SLongIO`, `SRawIO`, and `SIOResult` provide the set of similar procedures set that operate over default input and output channels, and so do not take a parameter identifying a channel.

### 1.2.1 Modules TextIO and STextIO

The module `TextIO` provides facilities for input and output of characters, character strings, and line marks, using text operations.

The procedures of the module `STextIO` behave as the corresponding procedures of the module `TextIO`, except that input is taken from the default input channel, and output is sent to the default output channel.

**ReadChar** *Read a character*

```
PROCEDURE ReadChar (cid: IOChan.ChanId; VAR ch: CHAR);
PROCEDURE ReadChar (VAR ch: CHAR);
```

If there is a character next in the input stream identified by `cid`, the procedure

`ReadChar` removes it from the stream and assigns its value to `ch`; otherwise the value of `ch` is not defined. The read result for the channel is set to the value

`allRight` if a character is read;

`endOfLine` if no character is read, the next item being a line mark;

`endOfInput` if no character is read, the input stream having ended.

**ReadRestLine** *Read rest of line*

```
PROCEDURE ReadRestLine (cid: IOChan.ChanId;
                        VAR s: ARRAY OF CHAR);
PROCEDURE ReadRestLine (VAR s: ARRAY OF CHAR);
```

If there is a character next in the input stream identified by `cid`, the procedure `ReadRestLine` reads a string of characters; reading continues as long as there are still characters before the next line mark or the end of the stream. As much of the string as can be accommodated is copied to `s` as a string value. The read result for the channel is set to the value

`allRight` if `s` is not empty and accomodates all of the string that has been read;

`outOfRange` if `s` is not empty but does not accommodate all of the string;

`endOfLine` if `s` is empty, the next item being a line mark;

`endOfInput` if `s` is empty, the input stream having ended.

**ReadString** *Read a string*

```
PROCEDURE ReadString (cid: IOChan.ChanId;
                      VAR s: ARRAY OF CHAR);
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
```

If there is a character next in the input stream identified by `cid`, the procedure `ReadString` reads a string of characters; reading continues as long as there are still characters before the next line mark or the end of the stream and the capacity of `s` has not been exhausted. The string is copied to `s` as a string value. The read result for the channel is set to the value

`allRight` if `s` is not empty;

`endOfLine` if `s` is empty, the next item being a line mark;

`endOfInput` if `s` is empty, the input stream having ended.

**ReadToken** *Read a space-delimited token*

```
PROCEDURE ReadToken (cid: IOChan.ChanId;
                     VAR s: ARRAY OF CHAR);
PROCEDURE ReadToken (VAR s: ARRAY OF CHAR);
```

The procedure `ReadToken` first skips any leading spaces in the input stream identified by `cid`. If the next item is a character, a string of characters is read; reading continues as long as there are still non-space characters before the next line mark or the end of the stream. As much of the string as can be accommodated is copied to `s` as a string value. The read result for the channel is set to the value

`allRight` if `s` is not empty and accomodates all of the string that has been read;

`outOfRange` if `s` is not empty but does not accommodate all of the string;

`endOfLine` if `s` is empty, the next item being a line mark;

`endOfInput` if `s` is empty, the input stream having ended.

**SkipLine** *Skip rest of input line*

```
PROCEDURE SkipLine (cid: IOChan.ChanId);
PROCEDURE SkipLine ();
```

The procedure `SkipLine` reads successive items from the input stream identified by `cid` up to and including the next line mark, or until the end of the stream is reached.

The read result for the channel is set to the value

`allRight` if a line mark is read;

`endOfInput` if no line mark is read, the input stream having ended.

**WriteChar** *Write a character*

```
PROCEDURE WriteChar (cid: IOChan.ChanId; ch: CHAR);
PROCEDURE WriteChar (ch: CHAR);
```

The procedure `WriteChar` writes the character `ch` to the output stream identified by `cid`.

**WriteLn** *Write a line mark*

```
PROCEDURE WriteLn (cid: IOChan.ChanId);
PROCEDURE WriteLn ();
```

The procedure `WriteLn` writes a line mark to the output stream identified by `cid`.

**WriteString** *Write a string*

```
PROCEDURE WriteString (cid: IOChan.ChanId;
                       s: ARRAY OF CHAR);
PROCEDURE WriteString (s: ARRAY OF CHAR);
```

The procedure `WriteString` writes the string value in `s` to the output stream identified by `cid`.

### 1.2.2 Modules WholeIO and SWholeIO

The module `WholeIO` provides facilities for input and output of whole numbers in decimal text form.

The text form of a signed whole number is

```
["+" | "-"], decimal digit, {decimal digit}
```

The text form of an unsigned whole number is

```
decimal digit, {decimal digit}
```

The procedures of the module `SWholeIO` behave as the corresponding procedures of the module `WholeIO`, except that input is taken from the default input channel, and output is sent to the default output channel.

**ReadInt** *Read an INTEGER value*

```
PROCEDURE ReadInt (cid: IOChan.ChanId;
                   VAR int: INTEGER);
PROCEDURE ReadInt (VAR int: INTEGER);
```

The procedure `ReadInt` skips any leading spaces from the input stream identified by `cid`, and then reads characters that form a signed whole number. The read result for the channel is set to the value

`allRight` if a signed whole number is read, and its value is in the range of the type `INTEGER`; the value of this number is assigned to `int`;

`outOfRange` if a signed whole number is read, but its value is out of range of the type `INTEGER`; the value `MAX(INTEGER)` or `MIN(INTEGER)` is assigned to `int` according to the sign of the number;

`wrongFormat` if there are characters read or to be read, but these are not in the format of a signed whole number; the value of `int` is not defined;

`endOfLine` if no characters are read, the next item being a line mark; the value of `int` is not defined;

`endOfInput` if no characters are read, the input having ended; the value of `int` is not defined.

**WriteInt** *Write an INTEGER value*

```
PROCEDURE WriteInt (cid: IOChan.ChanId;
                    int: INTEGER;
                    width: CARDINAL);
PROCEDURE WriteInt (int: INTEGER;
                    width: CARDINAL);
```

The procedure `WriteInt` writes the value of `int` to the output stream identified by `cid` in text form, with leading spaces as required to make the number of characters written at least that given by `width`. A sign is written only for negative values. In the special case of a value of zero for `width`, exactly one leading space is written.

**ReadCard** *Read a CARDINAL value*

```
PROCEDURE ReadCard (cid: IOChan.ChanId;
                    VAR card: CARDINAL);
PROCEDURE ReadCard (VAR card: CARDINAL);
```

The procedure `ReadCard` skips any leading spaces from the input stream identified by `cid`, and then reads characters that form an unsigned whole number. The read result for the channel is set to the value

`allRight` if an unsigned whole number is read, and its value is in the range of the type `CARDINAL`; the value of the number is assigned to `card`;

`outOfRange` if a signed whole number is read, but its value is out of range of the values of the type `CARDINAL`; the value `MAX(CARDINAL)` is assigned to `card`;

`wrongFormat` if there are characters read or to be read, but these are not in the format of an unsigned whole number; the value of `card` is not defined;

`endOfLine` if no characters are read, the next item being a line mark; the value of `card` is not defined;

`endOfInput` if no characters are read, the input having ended; the value of `card` is not defined.

**WriteCard** *Write a CARDINAL value*

```
PROCEDURE WriteCard (cid: IOChan.ChanId;
                     card: CARDINAL;
                     width: CARDINAL);
PROCEDURE WriteCard (card: CARDINAL;
                     width: CARDINAL);
```

The procedure `WriteCard` writes the value of `card` to the output stream identified by `cid` in text form, with leading spaces as required to make the number of characters written at least that given by `width`. In the special case of a value of zero for `width`, exactly one leading space is written.

### 1.2.3 Modules RealIO, SRealIO, LongIO, and SLongIO

The modules `RealIO` and `LongIO` provide facilities for input and output of real numbers in decimal text form.

In the case of `RealIO`, real number parameters are of the type `REAL`. In the case of `LongIO`, real number parameters are of the type `LONGREAL`.

The semantics of the two modules are the same, except that when module `RealIO` refers to real number values, these values are of the type `REAL`, and when module `LongIO` refers to real number values, these values are of the type `LONGREAL`.

**NOTE:**

The above statement is merely to avoid needless repetition of the semantics for the two modules.

The text form of a signed fixed-point real number is

```
["+" | "-"], decimal digit, {decimal digit},
[".", {decimal digit}]
```

The text form of a signed floating-point real number is

```
signed fixed-point real number,
"E"|"e", ["+" | "-"], decimal digit, {decimal digit}
```

The procedures of the module `SRealIO` behave as the corresponding procedures of the module `RealIO`, except that input is taken from the default input channel, and output is sent to the default output channel.

The procedures of the module `SLongIO` behave as the corresponding procedures of the module `LongIO`, except that input is taken from the default input channel, and output is sent to the default output channel.

| **ReadReal** | *Read a real value* |
|---|---|

```
PROCEDURE ReadReal (cid: IOChan.ChanId;
                    VAR real: REAL);
PROCEDURE ReadReal (cid: IOChan.ChanId;
                    VAR real: LONGREAL);
PROCEDURE ReadReal (VAR real: REAL);
PROCEDURE ReadReal (VAR real: LONGREAL);
```

The procedure `ReadReal` skips any leading spaces from the input stream identified by `cid`, and then reads characters that form a signed fixed or floating point number. The read result for the channel is set to the value

`allRight` if a signed real number is read, and its value is in the range of the type of `real`; the value of this number is assigned to `real`;

`outOfRange` if a signed real number is read, but its value is out of range of the type of `real`; the maximum or minimum value of the type of `real` is assigned to `real` according to the sign of the number;

`wrongFormat` if there are characters read or to be read, but these characters are not in the format of a signed real number; the value of `real` is not defined;

`endOfLine` if no characters are read, the next item being a line mark; the value of `real` is not defined;

`endOfInput` if no characters are read, the input having ended; the value of `real` is not defined.

**WriteFloat** *Write a real value in floating-point format*

```
PROCEDURE WriteFloat (cid: IOChan.ChanId;
                      real: REAL;
                      sigFigs: CARDINAL;
                      width: CARDINAL);
PROCEDURE WriteFloat (cid: IOChan.ChanId;
                      real: LONGREAL;
                      sigFigs: CARDINAL;
                      width: CARDINAL);
PROCEDURE WriteFloat (real: REAL;
                      sigFigs: CARDINAL;
                      width: CARDINAL);
PROCEDURE WriteFloat (real: LONGREAL;
                      sigFigs: CARDINAL;
                      width: CARDINAL);
```

The procedure `WriteFloat` writes the value of `real` to the output stream identified by `cid` in floating-point text form, with leading spaces as required to make the number of characters written at least that given by `width`. A sign is written only for negative values. In the special case of a value of zero for `width`, exactly one leading space is written.

One significant digit is included in the whole number part. The signed exponent part is included only if the exponent value is not zero. If the value of `sigFigs` is greater than zero, that number of significant digits is included, otherwise an implementation-defined number of significant digits is included. The decimal point is not included if there are no significant digits in the fractional part.

The following table gives examples of output by `WriteFloat`:

| sigFigs | 3923009 | 39.23009 | 0.0003923009 |
|---|---|---|---|
| 1 | 4E+6 | 4E+1 | 4E-4 |
| 2 | 3.9E+6 | 3.9E+1 | 3.9E-4 |
| 5 | 3.9230E+6 | 3.9230E+1 | 3.9230E-4 |

**WriteEng** *Write a real value in engineering format*

```
PROCEDURE WriteEng (cid: IOChan.ChanId;
                    real: REAL;
                    sigFigs: CARDINAL;
                    width: CARDINAL);
PROCEDURE WriteEng (cid: IOChan.ChanId;
                    real: LONGREAL;
                    sigFigs: CARDINAL;
                    width: CARDINAL);
PROCEDURE WriteEng (real: REAL;
                    sigFigs: CARDINAL;
                    width: CARDINAL);
PROCEDURE WriteEng (real: LONGREAL;
                    sigFigs: CARDINAL;
                    width: CARDINAL);
```

The procedure `WriteEng` behaves as the procedure `WriteFloat` except that the number is scaled with one to three digits in the whole number part, and with an exponent that is a multiple of three.

The following table gives examples of output by `WriteEng`:

| sigFigs | 3923009 | 39.23009 | 0.0003923009 |
|---|---|---|---|
| 1 | 4E+6 | 40 | 400E-6 |
| 2 | 3.9E+6 | 39 | 390E-6 |
| 5 | 3.9230E+6 | 39.230 | 392.30E-6 |

**WriteFixed** *Write a real value in fixed-point format*

```
PROCEDURE WriteFixed (cid: IOChan.ChanId;
                      real: REAL;
                      place: INTEGER;
                      width: CARDINAL);
PROCEDURE WriteFixed (cid: IOChan.ChanId;
                      real: LONGREAL;
                      place: INTEGER;
                      width: CARDINAL);
PROCEDURE WriteFixed (real: REAL;
                      place: INTEGER;
                      width: CARDINAL);
PROCEDURE WriteFixed (real: LONGREAL;
                      place: INTEGER;
                      width: CARDINAL);
```

The procedure `WriteFixed` writes the value of `real` to the output stream identified by `cid` in fixed-point text form with leading spaces as required to make the number of characters written at least that given by `width`. A sign is written only for negative values. In the special case of a value of zero for `width`, exactly one leading space is written.

At least one digit is included in the whole number part. The value is rounded to the given value of `place` relative to the decimal point. The decimal point is suppressed if `place` is less than zero.

The following table gives examples of output by `WriteFixed`:

| places | 3923009 | 39.23009 | 0.0003923009 |
|---|---|---|---|
| -5 | 3920000 | 0 | 0 |
| -2 | 3923010 | 40 | 0 |
| -1 | 3923009 | 39 | 0 |
| 0 | 3923009. | 39. | 0. |
| 1 | 3923009.0 | 39.2 | 0.0 |
| 4 | 3923009.0000 | 39.2301 | 0.0004 |

**WriteReal** *Write a real value*

```
PROCEDURE WriteReal (cid: IOChan.ChanId;
                     real: REAL;
                     width: CARDINAL);
PROCEDURE WriteReal (cid: IOChan.ChanId;
                     real: LONGREAL;
                     width: CARDINAL);
PROCEDURE WriteReal (real: REAL;
                     width: CARDINAL);
PROCEDURE WriteReal (real: LONGREAL;
                     width: CARDINAL);
```

If the sign and magnitude of real can be expressed in a field given by `width`, the procedure `WriteReal` behaves as the procedure `WriteFixed`, with a value of `place` chosen to fill exactly the remaining field. Otherwise it behaves as the procedure `WriteFloat`, with a value of `sigFigs` of at least one, limited to those that can be included together with the sign and exponent part in the given `width`.

In the special case of a `width` of zero, the effect is as for the procedure `WriteFloat` with a value of `sigFigs` equal to zero.

### 1.2.4 Modules RawIO and SRawIO

The module `RawIO` provides facilities for direct input and output of data using raw operations (i.e. without any interpretation).

The procedures of the module `SRawIO` behave as the corresponding procedures of the module `RawIO`, except that input is taken from the default input channel, and output is sent to the default output channel.

**Read** *Read storage units*

```
PROCEDURE Read (cid: IOChan.ChanId;
                VAR to: ARRAY OF SYSTEM.LOC);
PROCEDURE Read (VAR to: ARRAY OF SYSTEM.LOC);
```

While the stream identified by `cid` is not exhausted, the procedure `Read` reads

successive storage units from that channel, and assign them without interpretation to successive components of `to`. The read result for the channel is set to the value

`allRight` if items are read for all components;

`wrongFormat` if some items are read, but not for all components;

`endOfInput` if no items are read, the input having ended.

**Write** *Write storage units*

```
PROCEDURE Write (cid: IOChan.ChanId;
                 from: ARRAY OF SYSTEM.LOC);
PROCEDURE Write (from: ARRAY OF SYSTEM.LOC);
```

The procedure `Write` writes successive components of `from` to the channel identified by `cid`, as storage units without interpretation.

### 1.2.5 Module IOConsts

The module `IOConsts` defines the enumeration type `ReadResults` used to express read results. Programs do not normally need to import from `IOConsts` directly, since client modules define identifiers that correspond to those defined by this module.

**ReadResults** *Read result identities*

```
TYPE
  ReadResults =  (* This type is used to classify the result
                    of an input operation *)
  (
    notKnown,    (* no read result is set *)
    allRight,    (* data is as expected or as required *)
    outOfRange,  (* data cannot be represented *)
    wrongFormat, (* data not in expected format *)
    endOfLine,   (* end of line seen before expected data *)
    endOfInput   (* end of input seen before expected data *)
  );
```

### 1.2.6 Modules IOResult and SIOResult

The module `IOResult` provides the facility for a program to determine whether the last operation to read data from a specified input channel found data in the required format.

The procedure of the module `SIOResult` behaves as the corresponding procedure of the module `IOResult`, except that the read result for the default input channel is returned.

**NOTE:**

The existence of the module `IOConsts` allows the definition module `SIOResult` to be independent of the modules `IOResult` and `IOChan`.

**ReadResults** *Read result identities*

```
TYPE
  ReadResults = IOConsts.ReadResults;
```

The type `IOConsts.ReadResults` is re-exported.

**ReadResult** *Get read result for channel*

```
PROCEDURE ReadResult (cid: IOChan.ChanId): ReadResults;
PROCEDURE ReadResult (): ReadResults;
```

The function procedure `ReadResult` returns the stored read result for the channel identified by `cid`.

## 1.3 Device-Independent Channel Operations

The module `IOChan` provides access to channel operations that are provided in a device-independent manner for all channels.

Device-dependent operations (which include operations for opening new channels and subsequently closing them) are defined in the definition module for each device.

### 1.3.1 Module IOChan

The module `IOChan` defines the hidden type `ChanId` that is used to identify channels throughout the input/output library, and provides facilities for device-independent access to operations supported by the device to which a channel is connected.

**ChanId** *Channel identity*

```
TYPE
  ChanId;
```

Values of this type are used to identify channels throughout the input/output library.

**InvalidChan** *Get an invalid channel id*

```
PROCEDURE InvalidChan (): ChanId;
```

The function procedure `InvalidChan` returns the identity of the invalid channel.

**NOTE:**

The invalid channel is a channel on which no data transfer operations are available; enquiries on the invalid channel indicate that this is the case. The identity of the invalid channel can be used to initialize variables of the type `ChanId`.

### 1.3.2 Module IOChan - Text Operations

Each of the following procedures invokes a corresponding operation for the device associated with the given channel. If the associated device supports the operation on the channel, the behaviour of the procedure conforms with the given description. The full behaviour is defined separately for each device.

These device operations produce a *text stream*. A text stream is a sequence of items, each of which corresponds either to a character or a line mark. The sequence may be empty.

The text operations provided by a device module perform any necessary translation between the internal representation (as a sequence of characters and line marks) and the external representation used by the source or destination. This may involve, for example, translation to and from escape sequences used in a coded character set, mapping between the external and internal representation of lines, or the interpretation of format effectors.

The interpretation of control characters is implementation-defined. The exception `textParseError` occurs (but need not be raised) if input data does not correspond to a character or line mark.

If the device does not support the operation on the channel, it raises the exception `notAvailable`.

**Look** *Invoke Look operation*

```
PROCEDURE Look (cid: ChanId;
                VAR ch: CHAR;
                VAR res: IOConsts.ReadResults);
```

The procedure `Look` invokes the `Look` operation for the device that is associated with the channel identified by `cid`.

**NOTE:**

If supported on the channel, the device `Look` operation attempts to examine the next item in the input stream for the channel identified by `cid`, without removing it. If the next item is a character, its value is assigned to `ch`; otherwise, the value of `ch` is not defined. `res` is set to the same value as the stored read result for the channel `cid`, this being:

`allRight` if a character is seen next;

`endOfLine` if no character is seen, the next item being a line mark;

`endOfInput` if no character is seen, the input having ended.

**Skip** *Invoke Skip operation*

```
PROCEDURE Skip (cid: ChanId);
```

The procedure `Skip` invokes the `Skip` operation for the device that is associated with the channel identified by `cid`.

**NOTE:**

If supported on the channel, the device `Skip` operation attempts to remove the next item in the input stream for the channel identified by `cid`. If there is no next item, the end of the input stream having been reached, the exception `skipAtEnd` is raised; otherwise the next character or line mark in the stream is removed, and the stored read result for the channel `cid` is set to the value `allRight`.

**SkipLook** *Invoke SkipLook operation*

```
PROCEDURE SkipLook (cid: ChanId;
                    VAR ch: CHAR;
                    VAR res: IOConsts.ReadResults);
```

The procedure `SkipLook` invokes the `SkipLook` operation for the device that is associated with the channel identified by `cid`.

**NOTE:**

If supported on the channel, the device `SkipLook` operation attempts to remove the next item in the input stream for the channel identified by `cid` and then to examine the following item without removing it. If there is no next item, the end of the input stream having been reached, the exception `skipAtEnd` is raised; otherwise the next character or line mark in the stream is removed. If this is followed by a character as the next item in the stream, its value is assigned to `ch`, without removing the character from the stream; otherwise, the value of `ch` is not defined. `res` is set to the same value as the stored read result for the channel `cid`, this being:

`allRight` if a character is seen next;

`endOfLine` if no character is seen, the next item being a line mark;

`endOfInput` if no character is seen, the input having ended.

**WriteLn** *Invoke WriteLn operation*

```
PROCEDURE WriteLn (cid: ChanId);
```

The procedure `WriteLn` invokes the `WriteLn` operation for the device that is associated with the channel identified by `cid`.

**NOTE:** If supported on the channel, the device `WriteLn` operation writes a line mark to the output stream identified by `cid`.

| **TextRead** | *Invoke TextRead operation* |
|---|---|

```
PROCEDURE TextRead (cid: ChanId;
                    to: SYSTEM.ADDRESS;
                    maxChars: CARDINAL;
                    VAR charsRead: CARDINAL);
```

The procedure `TextRead` invokes the `TextRead` operation for the device that is associated with the channel identified by `cid`.

**NOTES:**

- If supported on the channel, the device `TextRead` operation reads at most `maxChars` characters from the current line on the input stream for the channel identified by `cid`, and assigns their values to successive components of an array variable of the character type for which the address of the first component is `to`. The number of characters read is assigned to `charsRead`. The read result for the channel `cid` is set to the value

  `allRight` if `'maxChars = charsRead = 0' or ('maxChars > 0' and 'charsRead > 0')`;

  `endOfLine` if `'maxChars > 0' and 'charsRead = 0'` , the next item being a line mark;

  `endOfInput` if `'maxChars > 0' and 'charsRead = 0'`, the input having ended.

- The intention is to allow 'sub-arrays' to be selected, by passing the address of a starting component within a larger array. An exception occurs, but need not be raised, if the call leads to an attempt to access a non-existent component of the larger array.

| **TextWrite** | *Invoke TextWrite operation* |
|---|---|

```
PROCEDURE TextWrite (cid: ChanId;
                     from: SYSTEM.ADDRESS;
                     charsToWrite: CARDINAL);
```

The procedure `TextWrite` invokes the `TextWrite` operation for the device that is associated with the channel identified by `cid`.

**NOTES:**

- If supported on the channel, the device `TextWrite` operation copies `charsToWrite` characters, from successive components of an array variable of the character type, for which the address of the first component is `from`, to the output stream for the channel identified by `cid`. Copying starts from the index given by `offset`.
- The intention is to allow 'sub-arrays' to be selected, by passing the address of a starting component within a larger array. An exception occurs, but need not be raised, if the call leads to an attempt to access a non-existent component of the larger array.

### 1.3.3 Module IOChan - Raw Operations

Each of the following procedures invokes a corresponding operation for the device associated with the given channel. If the associated device supports the operation on the channel, the behaviour of the procedure conforms with the given description. The full behaviour is defined for each device module.

The raw operations provided by a device module transfer data location by location with no translation or interpretation.

If the device does not support the operation on the channel, it raises the exception `notAvailable`.

**RawRead** *Invoke RawRead operation*

```
PROCEDURE RawRead (cid: ChanId;
                   to: SYSTEM.ADDRESS;
                   maxLocs: CARDINAL;
                   VAR locsRead: CARDINAL);
```

The procedure `RawRead` invokes the `RawRead` operation for the device that is associated with the channel identified by `cid`.

**NOTES**

- If supported on the channel, the device `RawRead` operation reads at most `maxLocs` items from the input stream for the channel identified by `cid`, and assigns their values to successive components of an array variable of the location type for which the address of the first component is `to`. The number of items read is assigned to `locsRead`. The read result for the channel cid is set to the value

  `allRight` if `('maxLocs = locsRead = 0') or ('maxLocs > 0 and 'locsRead > 0')`

  `endOfInput` if `'maxLocs > 0' and 'locsRead = 0'`

- The intention is to allow 'sub-arrays' to be selected, by passing the address of a starting component within a larger array. An exception occurs, but need not be raised, if the call leads to an attempt to access a non-existent component of the larger array.

**RawWrite** *Invoke RawWrite operation*

```
PROCEDURE RawWrite (cid: ChanId;
                    from: SYSTEM.ADDRESS;
                    locsToWrite: CARDINAL);
```

The procedure `RawWrite` invokes the `RawWrite` operation for the device that is associated with the channel identified by `cid`.

**NOTES:**

- If supported on the channel, the device `RawWrite` operation copies `locsToWrite` items, from successive components of an array variable of the character type, for which the address of the first component is `from`, to the output stream for the channel identified by `cid`. Copying starts from the index given by `offset`.
- The intention is to allow 'sub-arrays' to be selected, by passing the address of a starting component within a larger array. An exception occurs, but need not be raised, if the call leads to an attempt to access a non-existent component of the larger array.

### 1.3.4 Module IOChan - Common Operations

Each of the following procedures invokes a corresponding operation for the device associated with the given channel. The behaviour of the procedure conforms with the given description. The full behaviour is defined for each device module.

| **GetName** | *Invoke GetName operation* |
|---|---|

```
PROCEDURE GetName (cid: ChanId;
                   VAR s: ARRAY OF CHAR);
```

The procedure `GetName` invokes the `GetName` operation for the device that is associated with the channel identified by `cid`.

**NOTES:**

- The device `GetName` operation copies to `s` (as a string value) a name associated with the channel identified by `cid`.
- The name is truncated if the capacity of `s` is inadequate.

| **Reset** | *Invoke Reset operation* |
|---|---|

```
PROCEDURE Reset (cid: ChanId);
```

The procedure `Reset` invokes the Reset operation for the device that is associated with the channel identified by `cid`.

**NOTE**:

The device `Reset` operation resets the device associated with the channel identified by `cid` to a state defined by the device module.

| **Flush** | *Invoke Flush operation* |
|---|---|

```
PROCEDURE Flush (cid: ChanId);
```

The procedure `Flush` invokes the `Flush` operation for the device that is associated with the channel identified by `cid`.

**NOTE:**

The device `Flush` operation flushes any data buffered by the device module out to the destination associated with `cid`.

### 1.3.5 Module IOChan - Access to Read Results

Higher-level data input procedures, for units such as strings and numerals, may alter the read result for a channel to indicate success or a particular kind of failure of interpretation. The result can be recovered, if necessary, by the caller of the data input procedure.

| **SetReadResult** | *Set read result for channel* |
|---|---|

```
PROCEDURE SetReadResult (cid: ChanId;
                         res: IOConsts.ReadResults);
```

The procedure `SetReadResult` sets the read result for the channel identified by `cid` to the value given by `res`.

| **ReadResult** | *Get read result for channel* |
|---|---|

```
PROCEDURE ReadResult (cid: ChanId): IOConsts.ReadResults;
```

The function procedure `ReadResult` returns the stored read result for the channel identified by `cid`.

### 1.3.6 Module IOChan - Channel Enquiries

| **CurrentFlags** | *Get current flags for channel* |
|---|---|

```
PROCEDURE CurrentFlags (cid: ChanId): ChanConsts.FlagSet;
```

The function procedure `CurrentFlags` returns the set of flags that currently apply to the channel identified by `cid`, as defined for the associated device.

### 1.3.7 Module IOChan - Exceptions and Device Errors

The device-independent exceptions raised by the input/output library are identified by the values of the enumeration type `ChanExceptions`:

| **ChanExceptions** | *Channel exceptions identities* |
|---|---|

```
TYPE
  ChanExceptions =
    (wrongDevice,
       (* device specific operation on wrong device *)
     notAvailable,
       (* operation attempted is not available on the channel *)
     skipAtEnd,
       (* attempt to skip data from a stream that has ended *)
     softDeviceError,
       (* device specific recoverable error *)
     hardDeviceError,
       (* device specific non-recoverable error *)
     textParseError,
       (* input data does not correspond to a character
          or line mark - optional detection *)
     notAChannel
       (* given value does not identify a channel -
          optional detection *)
   );
```

**NOTE:**

The detection of the exceptions `textParseError` and `notAChannel` is implementation-defined.

### IsChanException — *Query exceptional state*

```
PROCEDURE IsChanException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of an exception from `ChanExceptions`, the function procedure `IsChanException` returns `TRUE`; otherwise it returns `FALSE`.

### ChanException — *Query exception id*

```
PROCEDURE ChanException (): ChanExceptions;
```

If the calling coroutine is in the state of exceptional execution because of the raising of an exception from `ChanExceptions`, the function procedure `ChanException` returns the value that identifies the raised exception; otherwise the language exception `exException` is raised.

### DeviceErrNum — *Device error number*

```
TYPE
  DeviceErrNum = INTEGER;
```

Values of the type `DeviceErrNum` are used to identufy the implementation-defined error number for a chennel in the device exception handler.

See `DeviceError` procedure.

### DeviceError — *Get device error number*

```
PROCEDURE DeviceError (cid: ChanId): DeviceErrNum;
```

The function procedure `DeviceError` returns the error number stored by the device module for the channel identified by `cid`, provided that a device error

exception has been raised during an operation on that channel; otherwise the value of the call is not defined.

**NOTE:**

When a device procedure detects a device error, it raises the exception `softDeviceError` or `hardDeviceError`. If these exceptions are handled, the procedure `DeviceError` may be used to discover the implementation-defined error number stored by the device module for the channel that was in use when the device error occurred.

## 1.4 Obtaining Channels from Device Modules

Separate device modules are defined that provide a program with the facility to obtain a new channel, connected either to a sequential stream, a rewindable sequential file, a random access file, or a terminal device.

A request to obtain a channel is made by calling an appropriate 'open procedure', in general supplying a name that identifies the source or destination to which the connection is to be made.

The required input/output operations are specified using combinations of flags that are defined in terms of constants imported from the module `ChanConsts`.

An open procedure returns a parameter of an enumeration type (exported from the module `ChanConsts`) that indicates the success, or otherwise, of the request.

Each of these device modules defines a predicate allowing a check to be made that a given channel was opened by that module, as well as a 'close procedure' that allows a program to break the connection and release the channel.

Procedures are also provided for device-dependent operations, such as setting the read/write position on a random access file.

A further device module is defined to allow access to the program arguments over a pre-opened channel.

### 1.4.1 Module ChanConsts

The module `ChanConsts` defines common types and values for use with open procedures. Programs do not normally need to import from `ChanConsts` directly, since device modules define identifiers that correspond to those defined by this module.

**ChanFlags** *Channel open flags*

```
TYPE
  ChanFlags =
  ( readFlag,
      (* input operations are requested/available *)
    writeFlag,
      (* output operations are requested/available *)
    oldFlag,
      (* a file may/must/did exist
         before the channel is opened *)
    textFlag,
      (* text operations are requested/available *)
    rawFlag,
      (* raw operations are requested/available *)
    interactiveFlag,
      (* interactive use is requested/applies *)
    echoFlag
      (* echoing by interactive device on removal of characters
         from input stream requested/applies *)
  );
```

The elements of the enumeration type `ChanFlags` identify channel *flags* that are specified when a channel is opened and can be obtained for an open channel.

**NOTE:**

The type `FlagSet` is used in actual calls.

**FlagSet** *Channel open flags set*

```
  FlagSet = SET OF ChanFlags;

CONST
  read = FlagSet{readFlag};
    (* input operations are requested/available *)
  write = FlagSet{writeFlag};
    (* output operations are requested/available *)
  old = FlagSet{oldFlag};
    (* a file may/must/did exist
       before the channel is opened *)
  text = FlagSet{textFlag};
    (* text operations are requested/available *)
  raw = FlagSet{rawFlag};
    (* raw operations are requested/available *)
  interactive = FlagSet{interactiveFlag};
    (* interactive use is requested/applies *)
  echo = FlagSet{echoFlag};
    (* echoing by interactive device on removal of characters
       from input stream requested/applies *)
```

Values of the type `FlagSet` are used in the calls to channel open procedures. Singleton values of `FlagSet` are provided for convinience. For example, `read + write` can be used instead of `FlagSet{read,write}`.

**OpenResults** *Results of an open request*

```
TYPE
  OpenResults =
  ( opened,
      (* the open succeeded as requested *)
    wrongNameFormat,
      (* given name is in the wrong format
         for the implementation *)
    wrongFlags,
      (* given flags include a value
         that does not apply to the device *)
    tooManyOpen,
      (* this device cannot support any more open channels *)
    outOfChans,
      (* no more channels can be allocated *)
    wrongPermissions,
      (* file or directory permissions do not allow request *)
    noRoomOnDevice,
      (* storage limits on the device prevent the open *)
    noSuchFile,
      (* a needed file does not exist *)
    fileExists,
      (* a file of the given name already exists
         when a new one is required *)
    wrongFileType,
      (* the file is of the wrong type to support
         the required operations *)
    noTextOperations,
      (* text operations have been requested,
         but are not supported *)
    noRawOperations,
      (* raw operations have been requested,
         but are not supported *)
    noMixedOperations,
      (* text and raw operations have been requested, but they
         are not supported in combination *)
    alreadyOpen,
      (* the source/destination is already open
         for operations not supported in combination
         with the requested operations *)
    otherProblem
      (* open failed for some other reason *)
  );
```

The elements of the enumeration type `OpenResults` identify possible results of

an open request.

**The Use of ChanConsts**

To save repetition in the natural language definition of the device modules, the meaning given to some values of `FlagSet` and `OpenResults` is defined here. The meaning of the other flags is given for the open operations to which they apply.

In a call of a device module open procedure that has a request parameter of the type `FlagSet` and a result parameter of the type `OpenResults`:

If the result is `opened`, the following operations are provided for the opened channel for the combinations of request flags shown:

| | `read` | `write` | |
|---|---|---|---|
| `text` | text input | text output | as defined for the device |
| `raw` | raw input | raw output | as defined for the device |

**NOTE:**

The supplied flags specify the minimal functionality that must be available for the open operation to succeed. Implementations are free to allow operations in addition to those specified in the request flags provided that these are reflected in the enquiry flags returned for the channel.

If the result is other than `opened`, the channel parameter is assigned the value identifying the invalid channel, on which no input/output operations are provided. The result is chosen according to the following table:

| | |
|---|---|
| `wrongNameFormat` | if the given name is not in the format defined for the implementation |
| `wrongFlags` | if the given flags include a value that does not apply to the device |
| `tooManyOpen` | if the device cannot support any more open channels |
| `outOfChans` | if no more channels can be allocated |
| `wrongPermissions` | if file or directory permissions do not allow the request to be met |
| `noRoomOnDevice` | if storage limits on the device do not allow the request to be met |
| `noSuchFile` | if a needed file does not exist |
| `fileExists` | if a file of the given name already exists when a new one is required |
| `wrongFileType` | if the named file is of the wrong type to support the required operations |
| `noTextOperations` | if text operations have been requested, but are not supported by the device |
| `noRawOperations` | if raw operations have been requested, but are not supported by the device |
| `noMixedOperations` | if text and raw operations have been requested, but they are not supported in combination by the device |
| `alreadyOpen` | if the source/destination is already open for operations that are not supported in combination with the operations now requested |
| `otherProblem` | if the open failed for a reason other than the above |

### 1.4.2 Module StreamFile

The module `StreamFile` provides facilities for obtaining and releasing channels that are connected to named sources and/or destinations for independent sequential data streams.

The types `IOChan.ChanId`, `ChanConsts.FlagSet`, and `ChanConsts.OpenResults` are re-expored. The singleton values of the type `FlagSet` are declared for convinience:

```
TYPE
  ChanId = IOChan.ChanId;
```

```
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;

CONST
  read = FlagSet{ChanConsts.readFlag};
    (* input operations are requested/available *)
  write = FlagSet{ChanConsts.writeFlag};
    (* output operations are requested/available *)
  old = FlagSet{ChanConsts.oldFlag};
    (* a file may/must/did exist
       before the channel is opened *)
  text = FlagSet{ChanConsts.textFlag};
    (* text operations are requested/available *)
  raw = FlagSet{ChanConsts.rawFlag};
    (* raw operations are requested/available *)
```

In a request to open a sequential stream, the flags `read`, `write`, `old`, `text`, and `raw` apply. If `raw` is not included in the request parameter `flags`, inclusion of `text` is implied.

| **Open** | *Open sequential stream* |
|---|---|

```
PROCEDURE Open (VAR cid: ChanId;
                name: ARRAY OF CHAR;
                flags: FlagSet;
                VAR res: OpenResults);
```

If successful, the procedure `Open` assigns to `cid` the identity of a channel that is connected to a sequential stream specified by `name`, and the value `opened` is assigned to `res`.

If `write` is not included in `flags`, inclusion of `read` is implied; if `read` is given or implied, inclusion of `old` is implied; a source of the given name has to already exist if the call is to succeed.

If `write` is included, a destination of the given name has to not already exist, unless the flag `old` is given or implied.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**NOTE:**

Distinct modes in combination with `text` and/or `raw` are given by the following equivalent sets of flags:

- read from an existing source: `read` `old` `read+old`
- write to a new destination: `write`
- write to a new or old destination: `write+old`
- read/write an existing source/destination: `read+write` `read+write+old`

| **IsStreamFile** | *Query whether stream is sequential* |
|---|---|

```
PROCEDURE IsStreamFile (cid: ChanId): BOOLEAN;
```

The function procedure `IsStreamFile` returns `TRUE` if the channel identified by `cid` is open to a sequential stream, and `FALSE` otherwise.

| **Close** | *Close sequential stream* |
|---|---|

```
PROCEDURE Close (VAR cid: ChanId);
```

If the channel identified by `cid` is open to a sequential stream, the procedure `Close` closes the channel and assigns the value identifying the invalid channel to `cid`; otherwise, the exception `wrongDevice` is raised.

### 1.4.3 Module SeqFile

The module `SeqFile` provides facilities for obtaining and releasing channels that are connected to named rewindable sequential stored files.

If opened for both writing and reading, data written to the file may be read back from the start of the file. Rewriting from the start of the file causes the previous contents to be lost.

The types `IOChan.ChanId`, `ChanConsts.FlagSet`, and `ChanConsts.OpenResults` are re-expored. The singleton values of the type `FlagSet` are declared for convinience:

```
TYPE
  ChanId = IOChan.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;

CONST
  read = FlagSet{ChanConsts.readFlag};
    (* input operations are requested/available *)
  write = FlagSet{ChanConsts.writeFlag};
    (* output operations are requested/available *)
  old = FlagSet{ChanConsts.oldFlag};
    (* a file may/must/did exist before the channel is opened *)
  text = FlagSet{ChanConsts.textFlag};
    (* text operations are requested/available *)
  raw = FlagSet{ChanConsts.rawFlag};
    (* raw operations are requested/available *)
```

In a request to open a rewindable sequential file, the flags `read`, `write`, `old`, `text`, and `raw` apply. If `raw` is not included in the request parameter flags, inclusion of `text` is implied.

Channels open to rewindable sequential files may be in *input mode* or in *output mode*. In input mode, only input operations are available, `'(IOChan.Flags()*(read+write) = read)'` is true, and an attempt to write over the channel raises the exception `notAvailable`. In output mode, only output operations are available, `'(IOChan.Flags()*(read+write) = write)'` is true, and an attempt to read from the channel raises the exception `notAvailable`. All data written to a rewindable sequential file is appended to previous data written to that file.

## OpenWrite — *Open sequential file for writing*

```
PROCEDURE OpenWrite (VAR cid: ChanId;
                     name: ARRAY OF CHAR;
                     flags: FlagSet;
                     VAR res: OpenResults);
```

If successful, the procedure `OpenWrite` assigns to `cid` the identity of a channel that is connected to a stored file specified by `name`; the value `opened` is assigned to `res`. Output mode is selected and the file is truncated to zero length.

Inclusion of the `write` flag in the parameter `flags` is implied.

If the call is to succeed, a destination of the given name has to not already exist unless the flag `old` is given; if the `read` flag is included in the request, the `Reread` operation is available.

The effect of a `Reset` operation on the channel is to truncate the file to zero length and to select output mode.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**NOTE:**

Distinct modes in combination with `text` and/or `raw` are given by the following equivalent sets of flags:

- write to a new file: `write`
- write to a new file or a truncated existing file: `old` `write+old`
- write to a new file, need read operations: `write+read` `read`
- write to a new or existing file, need read operations: `old+read` `write+old+read`

| **OpenAppend** | *Open sequential file for appending* |
|---|---|

```
PROCEDURE OpenAppend (VAR cid: ChanId;
                      name: ARRAY OF CHAR;
                      flags: FlagSet;
                      VAR res: OpenResults);
```

If successful, the procedure `OpenAppend` assigns to `cid` the identity of a channel that is connected to a stored file specified by `name`; the value `opened` is assigned to `res`. Output mode is selected.

**Have to write something here.**

Inclusion of the `write` and `old` flags in the parameter `flags` is implied; a destination of the given name may already exist.

If the `read` flag is included in the request, the `Reread` operation is available if the call is to succeed.

The effect of a `Reset` operation on the channel is to select output mode.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**NOTE:**

Distinct modes in combination with `text` and/or `raw` are given by the following equivalent sets of flags:

- write to a new or append to an existing file: `write` `old` `write+old`
- write to a new or append to an existing file, need read operations: `read` `write+read` `old+read` `write+old+read`

## OpenRead *Open sequential file for reading*

```
PROCEDURE OpenRead (VAR cid: ChanId;
                    name: ARRAY OF CHAR;
                    flags: FlagSet;
                    VAR res: OpenResults);
```

If successful, the procedure `OpenRead` assigns to `cid` the identity of a channel that is connected to a stored file specified by `name`; the value opened is assigned to `res`. Input mode is selected and the read position correspond to the start of the file.

Inclusion of the `read` and `old` flags in the parameter `flags` is implied; a destination of the given name has to already exist if the call is to succeed.

If the `write` flag is included in the request, the `Rewrite` operation is available if the call is to succeed.

The effect of a `Reset` operation on the channel is to select input mode and to set the read position to the start of the file.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**NOTE:**

Distinct modes in combination with `text` and/or `raw` are given by the following equivalent sets of flags:

- read from an existing file: `read` `old` `read+old`

- read from an existing file, need write operations: `write` `read+write` `old+write` `read+old+write`

**IsSeqFile** *Query whether channel is open to a sequential file*

```
PROCEDURE IsSeqFile (cid: ChanId): BOOLEAN;
```

The function procedure `IsSeqFile` returns `TRUE` if the channel identified by `cid` is open to a rewindable sequential file, and `FALSE` otherwise.

**Reread** *Rewind and select input mode*

```
PROCEDURE Reread (cid: ChanId);
```

If the channel identified by `cid` is open to a rewindable sequential file, the procedure `Reread` attempts to set the read position of the channel to the start of the file, and to select input mode; otherwise, the exception `wrongDevice` is raised.

If the operation cannot be performed, perhaps because of insufficient permissions, neither input mode nor output mode are selected.

**Rewrite** *Rewind and select output mode*

```
PROCEDURE Rewrite (cid: ChanId);
```

If the channel identified by `cid` is open to a rewindable sequential file, the procedure `Rewrite` attempts to set the write position of the channel to the start of the file, to truncate the file to zero length, and to select output mode; otherwise, the exception `wrongDevice` is raised.

If the operation cannot be performed, perhaps because of insufficient permissions, neither input mode nor output mode are selected.

**Close** *Close sequential file*

```
PROCEDURE Close (VAR cid: ChanId);
```

If the channel identified by `cid` is open to a rewindable sequential file, the procedure `Close` closes the channel and assigns the value identifying the invalid channel to `cid`; otherwise, the exception `wrongDevice` is raised.

### 1.4.4 Module RndFile

The module `RndFile` provides facilities for obtaining and releasing channels that are connected to named random access files.

The types `IOChan.ChanId`, `ChanConsts.FlagSet`, and `ChanConsts.OpenResults` are re-expored. The singleton values of the type `FlagSet` are declared for convinience:

```
TYPE
  ChanId = IOChan.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;

CONST
  read = FlagSet{ChanConsts.readFlag};
    (* input operations are requested/available *)
  write = FlagSet{ChanConsts.writeFlag};
    (* output operations are requested/available *)
  old = FlagSet{ChanConsts.oldFlag};
    (* a file may/must/did exist before the channel is opened *)
  text = FlagSet{ChanConsts.textFlag};
    (* text operations are requested/available *)
  raw = FlagSet{ChanConsts.rawFlag};
    (* raw operations are requested/available *)
```

Channels opened by the module `RndFile` have an associated read/write position in the corresponding random-access file. The read/write position is at the start of the file after opening, or after a `Reset` operation on the channel. It is moved forward by the number of positions occupied by data that are taken from the file by an input operation, or written to the file by an output operation.

```
CONST
  FilePosSize = <implementation-defined whole
                 number greater than zero>;

TYPE
  FilePos = ARRAY [1 .. FilePosSize] OF SYSTEM.LOC;
```

**NOTE**:

The implementation-defined type `FilePos` has been specified in a way that enables values of this type to be read from or written to a file, while maintaining a degree of opacity for the type.

A random-access file have a length corresponding to the position after the highest read/write position at which data have been written. This length is zero if no data have been written to the file. If the read/write position is set at the current length, either implicitly on an input or output operation, or explicitly by a positioning operation, the effect of an input operation is as if the input stream had ended. A write at that position, if necessary, attempts to allocate more physical storage for the file.

In a request to open a random-access file, the flags `read`, `write`, `old`, `text`, and `raw` apply. If `text` is not included in the request parameter flags, inclusion of `raw` is implied.

| **OpenOld** | *Open existing random-aceess file* |
|---|---|

```
PROCEDURE OpenOld (VAR cid: ChanId;
                   name: ARRAY OF CHAR;
                   flags: FlagSet;
                   VAR res: OpenResults);
```

If successful, the procedure `OpenOld` assigns to `cid` the identity of a channel that is connected to a random access file specified by `name`; the value opened is assigned to `res`. The read/write position correspond to the start of the file.

Inclusion of the `old` flag in the parameter flags is implied; a file of the given name have to already exist if the call is to succeed.

If the `write` flag is not included in the request, inclusion of the `read` flag is implied.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**NOTE:**

Distinct modes in combination with `text` and/or `raw` are given by the following equivalent sets of flags:

- read from an existing file: `read` `old` `read+old`
- write to an existing file: `write` `write+old`

- read/write an existing file: `read+write` `read+write+old`

## OpenClean *Open and clear random-aceess file*

```
PROCEDURE OpenClean (VAR cid: ChanId;
                     name: ARRAY OF CHAR;
                     flags: FlagSet;
                     VAR res: OpenResults);
```

If successful, the procedure `OpenClean` assigns to `cid` the identity of a channel that is connected to a random access file specified by `name`; the value `opened` is assigned to `res`. The file is truncated to zero length.

Inclusion of the `write` flag in the parameter flags is implied; a destination of the given name has to not already exist unless the flag `old` is given.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**NOTE:**

Distinct modes in combination with `text` and/or `raw` are given by the following equivalent sets of flags:

- write to a new file: `write`
- write to a new file or a truncated existing file: `old` `write+old`
- write to a new file, read operations are needed: `read` `write+read`
- write to a new file or a truncated existing file, read operations are needed: `old+read` `write+old+read`

## IsRndFile *Query whether channel is open to a random access file*

```
PROCEDURE IsRndFile (cid: ChanId): BOOLEAN;
```

The function procedure `IsRndFile` returns `TRUE` if the channel identified by `cid` is open to a random access file, and `FALSE` otherwise.

**IsRndFileException** *Query exceptional state*

```
PROCEDURE IsRndFileException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of the `RndFile` exception, the function procedure `IsRndFileException` returns `TRUE`; otherwise it returns `FALSE`.

**StartPos** *Query start position*

```
PROCEDURE StartPos (cid: ChanId): FilePos;
```

If the channel identified by `cid` is open to a random access file, the function procedure `StartPos` returns the position of the start of the file; otherwise the exception `wrongDevice` is raised.

**CurrentPos** *Query current position*

```
PROCEDURE CurrentPos (cid: ChanId): FilePos;
```

If the channel identified by `cid` is open to a random access file, the function procedure `CurrentPos` returns the current read/write position of the file; otherwise the exception `wrongDevice` is raised.

**EndPos** *Query end position*

```
PROCEDURE EndPos (cid: ChanId): FilePos;
```

If the channel identified by `cid` is open to a random access file, the function procedure `EndPos` returns the first position in the file at or after which no data have been written; otherwise the exception `wrongDevice` is raised.

**NewPos** *Calculate new position*

```
PROCEDURE NewPos (cid: ChanId;
                  chunks: INTEGER;
                  chunkSize: CARDINAL;
                  from: FilePos): FilePos;
```

If the channel identified by `cid` is open to a random access file, the function procedure `NewPos` returns the read/write position `chunks * chunkSize` places relative to the position in the file given by the value of `from`; otherwise, the exception `wrongDevice` is raised. The `RndFile` exception is raised if the required position cannot be represented as a value of the type `FilePos`.

**NOTE:**

Calculation of the position in a random access file at which to issue text operations is dependent upon knowledge of the external representation of text items in a particular file; the amount by which the read/write position is moved as a result of a text operation may vary depending upon the item that is read or written. For raw operations, the read/write position is always moved by a value equal to the storage size of variables of the type of the item read or written.

**SetPos** *Set new position*

```
PROCEDURE SetPos (cid: ChanId; pos: FilePos);
```

If the channel identified by `cid` is open to a random access file, the procedure `SetPos` sets the read/write position for the file to the position given by the value of `pos`; otherwise the exception `wrongDevice` is raised.

If the position given by the value of `pos` is beyond the value returned by a call of `EndPos`, `'read <= IOChan.Flags()'` is false, and a call of an input operation raises the exception `notAvailable`; the value of `'write <= IOChan.Flags()'` is implementation-defined and correspond to the availability of output operations in this case. If data are subsequently written at such a position, those positions that have not been written to are filled with implementation-defined padding values.

**NOTE:**

Setting the read/write position beyond the value returned by `EndPos` does not of itself affect the size of the file.

| **Close** | *Close random access file* |
|---|---|

```
PROCEDURE Close (VAR cid: ChanId);
```

If the channel identified by cid is open to a random access file, the procedure `Close` closes the channel and assign the value identifying the invalid channel to `cid`; otherwise, the exception `wrongDevice` is raised.

### 1.4.5 Module TermFile

The module `TermFile` provides facilities that allow elementary access to an interactive terminal.

The types `IOChan.ChanId`, `ChanConsts.FlagSet`, and `ChanConsts.OpenResults` are re-expored. The singleton values of the type `FlagSet` are declared for convinience:

```
TYPE
  ChanId = IOChan.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;

CONST
  read = FlagSet{ChanConsts.readFlag};
    (* input operations are requested/available *)
  write = FlagSet{ChanConsts.writeFlag};
    (* output operations are requested/available *)
  text = FlagSet{ChanConsts.textFlag};
    (* text operations are requested/available *)
  raw = FlagSet{ChanConsts.rawFlag};
    (* raw operations are requested/available *)
  echo = FlagSet{ChanConsts.echoFlag};
    (* echoing by interactive device on reading of
       characters from input stream requested/applies *)
```

Channels connected to the terminal device are opened in *line mode* or in *single-character mode*. In line mode, items are echoed before being added to the input stream and are added a line at a time. In single character mode, items are added to the input stream as they are typed, and are echoed as they are removed from the

input stream by a text read device operation, provided they have not already been echoed.

Typed characters are distributed between multiple channels according to the sequence of read requests.

**NOTE:**

If all the channels open to the terminal are open in line mode, the terminal device operates exclusively in line mode; in that case, echoing might be performed by an underlying operating system. Similarly, if all the channels open to the terminal are open in single-character mode, the terminal device operates exclusively in single-character mode; in that case, echoing only occurs on reading from a channel and not on looking or skipping: this allows interactive input routines to suppress the echoing of unwanted or unexpected characters.

If an implementation allows it, there might be one or more channels open in line mode, and one or more channels open in single-character mode. In that case, echoing is postponed until the treatment of characters can be determined according to the sequence of calls of input operations. This behaviour allows programs that use the terminal in different modes to be written in a modular fashion, there being no need explicitly to save and restore the state of the terminal device.

In a request to open a channel to the terminal device, the flags `read`, `write`, `text`, `raw`, and `echo` apply. If `raw` is not included in the request parameter flags, inclusion of `text` is implied. If the `read` flag is not included in the request, inclusion of the `write` flag is implied.

| **Open** | *Open terminal* |
|---|---|

```
PROCEDURE Open (VAR cid: ChanId;
                flags: FlagSet;
                VAR res: OpenResults);
```

If successful, the procedure `Open` assigns to `cid` the identity of a channel that is connected to the terminal device.

If the `echo` flag is included in the request, single-character mode is available if the call is to succeed and the channel operates in single-character mode. Without the `echo` flag, line mode is available if the call is to succeed and the channel operates in line mode.

If a channel cannot be opened as required, the value of `res` indicates the reason, and `cid` identifies the invalid channel.

**IsTermFile** *Query whether channel is opened to terminal*

```
PROCEDURE IsTermFile (cid: ChanId): BOOLEAN;
```

The function procedure `IsTermFile` returns `TRUE` if the channel identified by `cid` is open to the terminal device, and `FALSE` otherwise.

**Close** *Close terminal*

```
PROCEDURE Close (VAR cid: ChanId);
```

If the channel identified by `cid` is open to the terminal device, the procedure `Close` closes the channel and assigns the value identifying the invalid channel to `cid`; otherwise, the exception `wrongDevice` is raised.

### 1.4.6 Module ProgramArgs

The module `ProgramArgs` provides a channel from which input can be taken from any arguments given to the program.

```
TYPE
  ChanId = IOChan.ChanId;
```

The initialization of the module `ProgramArgs` opens the channel from which the implementation-defined program arguments may be read.

**ArgChan** *Get program arguments channel id*

```
PROCEDURE ArgChan (): ChanId;
```

The function procedure `ArgChan` returns a value identifying a channel from which the implementation-defined program arguments may be read.

**IsArgPresent** *Query whether an argument is present*

```
PROCEDURE IsArgPresent (): BOOLEAN;
```

The function procedure `IsArgPresent` returns `TRUE` if there is a current argument from which to read, and `FALSE` otherwise.

If there is no current argument, `'read <= IOChan.Flags()'` is false, and attempting to read from the argument channel raises the exception `notAvailable`.

**NextArg** *Skip to next argument*

```
PROCEDURE NextArg ();
```

After the call to the procedure `NextArg`, if there is another argument, subsequent input from the argument channel is taken from the start of that argument; otherwise a call of `IsArgPresent` returns `FALSE`.

**NOTE:**

Provision of `NextArg` allows the treatment of arguments that contain spaces or line marks.

## 1.5 Interface to Channels for New Device Modules

Additional device modules may be provided to allow the library to be used with other input sources and output destinations. These might include, for example, files opened with host-specific options or parameters or with host-specific behaviour, a windowing system, or a speech output device.

### 1.5.1 Module IOLink

The module `IOLink` provides facilities that allow a user to provide specialized device modules for use with channels, following the pattern of the rest of the library.

A device needs to identify itself in order to allow a check to be made that device-dependent operations are applied only for channels opened to that device. To

this end, values of the hidden type `DeviceId` are used to identify new device modules, and are normally obtained by them during their initialization by a call to the procedure `AllocateDeviceId`.

```
TYPE
  DeviceId;
```

A device module procedure provided for opening a channel can obtain a new channel by calling the procedure `MakeChan`. If a channel is allocated, but the call of the device module open procedure is not successful for some reason, the device module should release the channel by calling the procedure `UnMakeChan`, and return the value identifying the invalid channel to its client.

A call to `UnMakeChan` is also made on a successful call of a device module procedure provided for closing a channel.

If a call of a device module 'open' procedure is successful, then by calling the function procedure `DeviceTablePtrValue`, a device module can obtain a pointer (of the type `DeviceTablePtr`) to a 'device table' (of a record type `DeviceTable`) for the channel. The fields of this record are initialized by `MakeChan`, but the procedure can then change any fields of the device table needed to install its own values for the device data, supported operations, and flags.

Device tables have:

1. a field in which the device module can store private data,
2. a field in which the value identifying the device module is stored,
3. a field in which the value identifying the channel is stored,
4. a field in which the read result is stored,
5. a field in which device error numbers are stored prior to the raising of a device error exception,
6. a field in which flags are stored indicating those which apply,
7. a field for each device procedure.

(The fields are initialized by `MakeChan` to the values shown in the definition module below.)

By calling the function procedure `IsDevice`, a device module can enquire whether it was responsible for opening a given channel. This allows it to implement a corresponding enquiry function that is exported from the device module itself.

Client modules may raise appropriate exceptions; to support this facility, the type `DevExceptionRange` and the procedure `RAISEdevException` can be used.

```
TYPE
  DeviceTablePtr = POINTER TO DeviceTable;
    (* Values of this type are used to refer to device tables *)

TYPE
  LookProc      = PROCEDURE (DeviceTablePtr,
                             VAR CHAR,
                             VAR IOConsts.ReadResults);
  SkipProc      = PROCEDURE (DeviceTablePtr);
  SkipLookProc  = PROCEDURE (DeviceTablePtr,
                             VAR CHAR,
                             VAR IOConsts.ReadResults);
  WriteLnProc   = PROCEDURE (DeviceTablePtr);
  TextReadProc  = PROCEDURE (DeviceTablePtr,
                             SYSTEM.ADDRESS,
                             CARDINAL,
                             VAR CARDINAL);
  TextWriteProc = PROCEDURE (DeviceTablePtr,
                             SYSTEM.ADDRESS,
                             CARDINAL);
  RawReadProc   = PROCEDURE (DeviceTablePtr,
                             SYSTEM.ADDRESS,
                             CARDINAL,
                             VAR CARDINAL);
  RawWriteProc  = PROCEDURE (DeviceTablePtr,
                             SYSTEM.ADDRESS,
                             CARDINAL);
  GetNameProc   = PROCEDURE (DeviceTablePtr,
                             VAR ARRAY OF CHAR);
  ResetProc     = PROCEDURE (DeviceTablePtr);
  FlushProc     = PROCEDURE (DeviceTablePtr);
  FreeProc      = PROCEDURE (DeviceTablePtr);
```

```
    (* Carry out the operations involved in closing
       the corresponding channel, including flushing buffers,
       but do not unmake the channel.
    *)

TYPE
  DeviceData = SYSTEM.ADDRESS;

DeviceTable =
  RECORD           (* Initialized by MakeChan to: *)
    cd:          DeviceData;
                   (* the value NIL *)
    did:         DeviceId;
                   (* the value given in the call of MakeChan *)
    cid:         IOChan.ChanId;
                   (* the identity of the channel *)
    result:      IOConsts.ReadResults;
                   (* the value notKnown *)
    errNum:      IOChan.DeviceErrNum;
                   (* undefined *)
    flags:       ChanConsts.FlagSet;
                   (* ChanConsts.FlagSet{} *)
    doLook:      LookProc;
                   (* raise exception notAvailable *)
    doSkip:      SkipProc;
                   (* raise exception notAvailable *)
    doSkipLook:  SkipLookProc;
                   (* raise exception notAvailable *)
    doLnWrite:   WriteLnProc;
                   (* raise exception notAvailable *)
    doTextRead:  TextReadProc;
                   (* raise exception notAvailable *)
    doTextWrite: TextWriteProc;
                   (* raise exception notAvailable *)
    doRawRead:   RawReadProc;
                   (* raise exception notAvailable *)
    doRawWrite:  RawWriteProc;
                   (* raise exception notAvailable *)
    doGetName:   GetNameProc;
                   (* return the empty string *)
    doReset:     ResetProc;
```

```
                  (* do nothing *)
    doFlush:     FlushProc;
                  (* do nothing *)
    doFree:      FreeProc;
                  (* do nothing *)
  END;

TYPE
  DevExceptionRange =
    z[IOChan.notAvailable .. IOChan.textParseError];
```

**AllocateDeviceId** *Allocate device id*

```
PROCEDURE AllocateDeviceId (VAR did: DeviceId);
```

The procedure `AllocateDeviceId` allocates an unique value of the type `DeviceId`, and assign this value to `did`.

**MakeChan** *Allocate a new channel for device*

```
PROCEDURE MakeChan (did: DeviceId;
                    VAR cid: IOChan.ChanId);
```

The procedure `MakeChan` attempts to allocate a new channel for the device module identified by `did`. If no more channels can be allocated, the value identifying the invalid channel is assigned to `cid`. Otherwise, a value identifying a new initialized channel is assigned to `cid`.

**UnMakeChan** *Deallocate channel from device*

```
PROCEDURE UnMakeChan (did: DeviceId;
                      VAR cid: IOChan.ChanId);
```

Provided the device module identified by `did` is the module that made the channel identified by `cid`, the procedure `UnMakeChan` deallocates the channel identified

by `cid`, and assigns the value identifying the invalid channel to `cid`; otherwise the exception `wrongDevice` is raised.

**DeviceTablePtrValue** *Get device table for channel*

```
PROCEDURE DeviceTablePtrValue (cid: IOChan.ChanId;
                               did: DeviceId
                              ): DeviceTablePtr;
```

Provided that the device module identified by `did` is the module that made the channel identified by `cid`, the function procedure `DeviceTablePtrValue` returns a pointer to the device table for the channel identified by `cid`; otherwise the exception `wrongDevice` is raised.

**IsDevice** *Query channel's device*

```
PROCEDURE IsDevice (cid: IOChan.ChanId;
                    did: DeviceId
                   ): BOOLEAN;
```

The function procedure `IsDevice` returns `TRUE` if the device module identified by `did` is the module that made the channel identified by `cid`, and otherwise returns `FALSE`.

**RAISEdevException** *Raise device exception*

```
PROCEDURE RAISEdevException (cid: IOChan.ChanId;
                             did: DeviceId;
                             x: DevExceptionRange;
                             s: ARRAY OF CHAR);
```

Provided that the device module identified by `did` is the module that made the channel identified by `cid`, the procedure `RAISEdevException` raises the exception given by `x`, and includes the string value in `s` in the exception message; otherwise the exception `wrongDevice` is raised.

## IsIOException — *Query exceptional state*

```
PROCEDURE IsIOException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of an exception from `IOChan.ChanExceptions`, the function procedure `IsIOException` returns `TRUE`; otherwise it returns `FALSE`.

## IOException — *Query exception id*

```
PROCEDURE IOException (): IOChan.ChanExceptions;
```

If the calling coroutine is in the state of exceptional execution because of the raising of an exception from `IOChan.ChanExceptions`, the function procedure `IOException` returns the value that identifies the raised exception; otherwise the language exception `exException` is raised.

**NOTE:**

A single value of `EXCEPTIONS.ExceptionSource` is used to identify the source of input/output library exceptions corresponding to `IOChan.ChanExceptions`. The procedures `IsIOException` and `IOException` are included so that this value need not be exported for corresponding procedures to be provided through the `IOChan` interface.

# Chapter 2

# Mathematical

The mathematical libraries provide the common mathematical functions and some constants.

The module `RealMath` provides the constants and functions for the type `REAL`, while the module `LongMath` provides similar constants and functions for the type `LONGREAL`.

The module `ComplexMath` provides the constants and functions for the type `COMPLEX`, while the module `LongComplexMath` provides similar functions for the type `LONGCOMPLEX`.

## 2.1 Modules RealMath and LongMath

The semantics of the two modules is the same, except that where the module `RealMath` refers to the pervasive type `REAL`, the corresponding function in `LongMath` refers to the pervasive type `LONGREAL`.

**NOTE:**

The above statement is merely to avoid needless repetition of the semantics for `LongMath`.

The units used for angular quantities are radians.

## Constants — *Useful constants*

```
CONST
  pi   = 3.14159265358979323846264338327950288419 72;
  exp1 = 2.71828182845904523536028747135266249775 72;
```

The constant `pi` provides an implementation-defined approximation to the mathematical constant $\pi$. The constant exp1 provides an implementation-defined approximation to the mathematical constant $e$.

**NOTE:**

Due to the approximations involved, `sin(pi)` might not equal zero exactly; similarly, `exp1` might not equal `exp(1)` exactly.

## sqrt — *Calculate square root*

```
PROCEDURE sqrt (x: REAL): REAL;
PROCEDURE sqrt (x: LONGREAL): LONGREAL;
```

The function procedure `sqrt` returns an implementation-defined approximation to the positive signed square root of `x`. An exception is raised if `x` is negative.

## exp — *Calculate exponent*

```
PROCEDURE exp (x: REAL): REAL;
PROCEDURE exp (x: LONGREAL): LONGREAL;
```

The function procedure `exp` returns an implementation-defined approximation to the mathematical constant $e$ raised to the power of `x`.

## ln — *Calculate natural logarithm*

```
PROCEDURE ln (x: REAL): REAL;
PROCEDURE ln (x: LONGREAL): LONGREAL;
```

The function procedure `ln` returns an implementation-defined approximation to the natural logarithm of `x`. An exception is raised if `x` is zero or negative.

**sin** *Calculate sine*

```
PROCEDURE sin (x: REAL): REAL;
PROCEDURE sin (x: LONGREAL): LONGREAL;
```

The function procedure `sin` returns an implementation-defined approximation to the sine of `x`.

**cos** *Calculate cosine*

```
PROCEDURE cos (x: REAL): REAL;
PROCEDURE cos (x: LONGREAL): LONGREAL;
```

The function procedure `cos` returns an implementation-defined approximation to the cosine of `x`.

**tan** *Calculate tangent*

```
PROCEDURE tan (x: REAL): REAL;
PROCEDURE tan (x: LONGREAL): LONGREAL;
```

The function procedure `tan` returns an implementation-defined approximation to the tangent of `x`. An exception is raised if `x` is an odd multiple of $\pi/2$.

**arcsin** *Calculate arcsine*

```
PROCEDURE arcsin (x: REAL): REAL;
PROCEDURE arcsin (x: LONGREAL): LONGREAL;
```

The function procedure `arcsin` returns an implementation-defined approximation to the arcsine of `x`. An exception is raised if the absolute value of `x` is greater than one.

**arccos** *Calculate arccosine*

```
PROCEDURE arccos (x: REAL): REAL;
PROCEDURE arccos (x: LONGREAL): LONGREAL;
```

The function procedure `arccos` returns an implementation-defined approximation to the arccosine of `x`. An exception is raised if the absolute value of `x` is greater than one.

**arctan** *Calculate arctangent*

```
PROCEDURE arctan (x: REAL): REAL;
PROCEDURE arctan (x: LONGREAL): LONGREAL;
```

The function procedure `arctan` returns an implementation-defined approximation to the arctangent of `x`.

**power** *Calculate power*

```
PROCEDURE power (base, exponent: REAL): REAL;
PROCEDURE power (base, exponent: LONGREAL): LONGREAL;
```

The function procedure `power` returns an implementation-defined approximation to the result obtained by raising `base` to the power of `exponent`. An exception is raised if the value of base is zero or negative.

This function is mathematically equivalent to $\exp(y \ln(x))$ but may be computed differently.

**round** *Round*

```
PROCEDURE round (x: REAL): INTEGER;
PROCEDURE round (x: LONGREAL): INTEGER;
```

The function procedure `round` returns the nearest integer to the value of `x`. The result is an implementation-defined selection of the two possible values if the

value of x is midway between two integer values. An exception occurs and may be raised if the mathematical result is not within the range of the type INTEGER.

**IsRMathException** *Query exceptional state*

```
PROCEDURE IsRMathException (): BOOLEAN;
PROCEDURE IsRMathException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of the RealMath exception, the function procedure RealMath.IsRMathException returns TRUE; otherwise the value is FALSE.

If the calling coroutine is in the state of exceptional execution because of the raising of the LongMath exception, the function procedure LongMath.IsRMathException returns TRUE; otherwise the value is FALSE.

## 2.2 Modules ComplexMath and LongComplexMath

The semantics of the two modules are the same, except that where the module ComplexMath refers to the pervasive type COMPLEX, the corresponding function in LongComplexMath refers to the pervasive type LONGCOMPLEX.

**NOTE:**

The above statement is merely to avoid needless repetition of the semantics for LongComplexMath.

**Constants** *Useful constants*

```
CONST
  i = CMPLX (0.0, 1.0);
  one = CMPLX (1.0, 0.0);
  zero = CMPLX (0.0, 0.0);
```

The constants i, one, and zero are the implementation-defined approximations to the specified values.

**NOTE:**

These constants are provided for convenience.

### abs — *Calculate modulus*

```
PROCEDURE abs (z: COMPLEX): REAL;
PROCEDURE abs (z: LONGCOMPLEX): LONGREAL;
```

The function procedure `abs` returns an implementation-defined approximation to the modulus (otherwise known as the length, or absolute value) of `z`.

**NOTE:**

An overflow exception may be raised in this computation, even when the complex number is itself well defined.

### arg — *Calculate argument*

```
PROCEDURE arg (z: COMPLEX): REAL;
PROCEDURE arg (z: LONGCOMPLEX): LONGREAL;
```

The function procedure `arg` returns an implementation-defined approximation to the angle that `z` subtends to the positive real axis in the complex plane. An exception is raised if the modulus of `z` is zero.

### conj — *Calculate conjugate*

```
PROCEDURE conj (z: COMPLEX): COMPLEX;
PROCEDURE conj (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `conj` returns an implementation-defined approximation to the complex conjugate of `z`.

**power** *Calculate power*

```
PROCEDURE power (base: COMPLEX;
                 exponent: REAL
                ): COMPLEX;
PROCEDURE power (base: LONGCOMPLEX;
                 exponent: LONGREAL
                ): LONGCOMPLEX;
```

The function procedure `power` returns an implementation-defined approximation to the result obtained by raising `base` to the power of `exponent`.

**sqrt** *Calculate square root*

```
PROCEDURE sqrt (z: COMPLEX): COMPLEX;
PROCEDURE sqrt (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `sqrt` returns an implementation-defined approximation to the principal square root of `z`.

**NOTE:**

That is, the result is the complex number with an argument of half the value of the argument of `z`, and whose modulus has the value of the positive square root of the modulus of `z`.

**exp** *Calculate exponent*

```
PROCEDURE exp (z: COMPLEX): COMPLEX;
PROCEDURE exp (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `exp` returns an implementation-defined approximation to the mathematical constant $e$ raised to the power of `z`.

**ln** *Calculate natural logarithm*

```
PROCEDURE ln (z: COMPLEX): COMPLEX;
PROCEDURE ln (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `ln` returns an implementation-defined approximation to

the principal value of the natural logarithm of `z`.

**sin** *Calculate sine*

```
PROCEDURE sin (z: COMPLEX): COMPLEX;
PROCEDURE sin (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `sin` returns an implementation-defined approximation to the complex sine of `z`.

**cos** *Calculate cosine*

```
PROCEDURE cos (z: COMPLEX): COMPLEX;
PROCEDURE cos (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `cos` returns an implementation-defined approximation to the complex cosine of `z`.

**tan** *Calculate tangent*

```
PROCEDURE tan (z: COMPLEX): COMPLEX;
PROCEDURE tan (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `tan` returns an implementation-defined approximation to the complex tangent of `z`. An exception is raised if `z` is an odd multiple of $\pi/2$.

**arcsin** *Calculate arcsine*

```
PROCEDURE arcsin (z: LONGCOMPLEX): LONGCOMPLEX;
PROCEDURE arcsin (z: COMPLEX): COMPLEX;
```

The function procedure `arcsin` returns an implementation-defined approximation to the principal value of the complex arcsine of `z`.

**arccos** *Calculate arccosine*

```
PROCEDURE arccos (z: COMPLEX): COMPLEX;
PROCEDURE arccos (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `arccos` returns an implementation-defined approximation to the complex arccosine of `z`.

**arctan** *Calculate arctangent*

```
PROCEDURE arctan (z: COMPLEX): COMPLEX;
PROCEDURE arctan (z: LONGCOMPLEX): LONGCOMPLEX;
```

The function procedure `arctan` returns an implementation-defined approximation to the complex arctangent of `z`.

**polarToComplex** *Convert from polar to complex*

```
PROCEDURE polarToComplex (abs, arg: REAL): COMPLEX;
PROCEDURE polarToComplex (abs, arg: LONGREAL): LONGCOMPLEX;
```

The function procedure `polarToComplex` returns an implementation-defined approximation to the complex number that has a modulus given by `abs` and an argument given by `arg`.

**scalarMult** *Scalar Multiplication*

```
PROCEDURE scalarMult (scalar: REAL;
                      z: COMPLEX
                     ): COMPLEX;
PROCEDURE scalarMult (scalar: LONGREAL;
                      z: LONGCOMPLEX
                     ): LONGCOMPLEX;
```

The function procedure `scalarMult` returns an implementation-defined approximation to the scalar product of the real value `scalar` with the complex value `z`.

**IsCMathException** *Query exceptional state*

```
PROCEDURE IsCMathException (): BOOLEAN;
PROCEDURE IsCMathException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of a `ComplexMath` exception, the function procedure `ComplexMath.IsCMathException` returns `TRUE`; otherwise it returns `FALSE`.

If the calling coroutine is in the state of exceptional execution because of the raising of a `LongComplexMath` exception, the function procedure `LongComplexMath.IsCMathException` returns `TRUE`; otherwise it returns `FALSE`.

# Chapter 3

# Concurrent Programming

## 3.1 Module Processes

The module `Processes` provides a basic set of facilities for use in concurrent programs. These can be used on their own, or in conjunction with those from the module `Semaphores` which provide for potentially parallel parts of the program to exclude one another from regions of interaction.

A concurrent program consists of a number of *processes*, each of which may potentially run in parallel with the others but is distinguishable from them. At any one time, a process may be in one of four states: It may be *ready*, that is, eligible to use the processor but not actually doing so. It may be *current*, that is, actually using the processor. It may be *passive*, that is, ineligible to use the processor until another process makes it eligible. Lastly, it may be *waiting*, that is, ineligible to use the processor until the occurrence of one of a set of events for which it is waiting.

At all times there must be at least one process using the processor, or, if no process is eligible, there must be at least one process waiting for some external event.

Two general styles of use are envisaged, and both may be present in a single program.

In the first style (using `Switch`), the model is of a set of closely coupled processes, which explicitly choose which of them is to run next, and which pass information between themselves as part of the choice. The intention is to provide a higher level coroutine-like facility between processes of the same urgency.

In the second style (using `Activate` and `SuspendMe`), the processes are written to be less dependent on one another, and the choice of which of them runs

is left to an internal scheduler, which is invoked whenever one process is suspended or another one is reactivated. This internal scheduler makes use of the fact that each process has an associated static 'urgency', specified by an `INTEGER` parameter when it is first created. The scheduler ensures that it cannot be the case that a process eligible to use the processor has an urgency greater than one of the processes currently doing so. The 'main process' (the parent program) is given a default urgency of zero; for other processes, the more positive the value of urgency, the more urgent is the process.

Those processes that suspend themselves to wait for external events must first associate themselves with one or more sources of such events. The International Standard does not prescribe how events occur, or what the sources of events must be, other than to require that they be mapped in an implementation-defined way to values of the pervasive type `CARDINAL`, and to require that a source of events cannot be connected to more than one process simultaneously.

**NOTES:**

- There is no requirement that pre-emptive scheduling be employed, although an implementation is free to incorporate such scheduling if this is desired.
- Although the International Standard applies to single-processor machines, if a pre-emptive (time-sliced) scheduler is present there may conceptually be more than one 'current' process; the description above is phrased so as to emphasize this.
- A program that uses the module `Processes` should not make explicit use of coroutines (except, perhaps, in the implementation of `Processes` itself).
- This is a partial specification: Various of the procedures in the modules `Processes` and `Semaphores` have semantics expressed in terms of preconditions. Their behaviour in situations where these preconditions are not met is deliberately not specified; in particular, the raising of exceptions is not required. This is a deliberate decision that allows to achieve maximum efficiency in the implementation of these modules.

The exceptions raised by `Processes` are identified by the values of the enumeration type `ProcessesExceptions`:

```
TYPE
  ProcessesExceptions = (passiveProgram, processError);
```

The detection of the exception `processError` is implementation-defined.

After the module is initialized, there is exactly one process, known as the 'main process'. This process have an urgency of `0` and a parameter of `NIL`; initially, it is not associated with any source of events.

**CHANGE:**

This module is not based on the module `Processes` described in *Programming in Modula-2*.

### 3.1.1 Types of Processes

```
TYPE
  ProcessId;
    (* Used to identify processes *)
  Parameter = SYSTEM.ADDRESS;
    (* Used to pass data between processes *)
  Body = PROC;
    (* Used as the type of a process body *)
  Urgency = INTEGER;
    (* Used by the internal scheduler *)
  Sources = CARDINAL;
    (* Used to identify event sources *)
```

### 3.1.2 The Procedures of Processes

The semantics of certain of the procedures of `Processes` require that a process be selected from the set of processes that are eligible to run, and be scheduled for execution. The selection algorithm shall guarantee that no process eligible to use the processor has an urgency greater than one of the processes currently doing so.

**NOTES:**

- The procedures in this category are `Start`, `StopMe`, `SuspendMe`, `Activate`, `SuspendMeAndActivate` and `Wait`.
- The urgency of a process is specified, when it is created, by a value of the type `INTEGER`, and cannot be changed dynamically. The more positive the value of urgency, the more urgent the process.

Certain of the procedures of `Processes` require that a process be associated with a source of external events. There shall be an implementation-defined mapping of values of the type `CARDINAL` to such sources of events.

A source of events shall not be associated with more than one process at any instant.

**NOTES:**

- The International Standard does not prescribe how events occur, or what the sources of events must be, other than in terms of this implementation-defined mapping.
- The International Standard does not specify the consequences if a value (of the type `Sources`) that is not mapped to a source of events is passed as an actual parameter to any of the procedures `Attach`, `Detach`, `IsAttached` or `Handler`.

## Create — *Create new process*

```
PROCEDURE Create (procBody: Body;
                  extraSpace: CARDINAL;
                  procUrg: Urgency;
                  procParams: Parameter;
                  VAR procId: ProcessId);
```

The procedure `Create` creates a new process. An unique value of the type `ProcessId` is assigned to `procId` as an identity for the process. The urgency and parameters for the created process are those given by `procUrg` and `procParams` respectively. `extraSpace` specifies the amount of workspace (in units of `SYSTEM.LOC`) that is required by the process, above any fixed overhead needed by the implementation of `Processes`.

The process will be ineligible to run (i.e. it is created in the *passive* state). When the process is first activated, it will start execution by invoking the procedure that is denoted by `procBody`.

The usage made of the workspace is implementation-dependent.

If the end of this procedure body is reached, or if a return statement is executed in the procedure body, then the effect will be the same as calling the protection domain exit procedure (if any), followed by an explicit call of `StopMe`.

**NOTES:**

- The process will be activated when another process makes it eligible by calling `Switch` or `Activate` or `SuspendMeAndActivate`.
- The standard library makes no provision to handle exceptions generated by created processes, which must handle all exceptions themselves if it is wished to avoid exceptional termination of a program.

**Start** *Start new process*

```
PROCEDURE Start (procBody: Body;
                 extraSpace: CARDINAL;
                 procUrg: Urgency;
                 procParams: Parameter;
                 VAR procId: ProcessId);
```

The procedure `Start` have an identical effect to the procedure `Create`, except that the created process will be eligible to run immediately (i.e. it is created in the *ready* state).

**StopMe** *Terminate calling process*

```
PROCEDURE StopMe ();
```

If the calling process is not associated with any source of events, the procedure `StopMe` causes the calling process to be terminated and removed from the system. The procedure does not return; the calling process will not again become eligible to run. If there are no other processes, then normal termination of the program is initiated. If there are other processes eligible to run, then one of them is selected for execution. The exception `passiveProgram` is raised if there are other processes, but none of them is eligible to run and none of them is waiting for an event to occur.

**NOTES:**

- If the main process stops, the other processes will continue to run.
- The behaviour of `StopMe` in situations where the calling process is associated with a source of events is implementation-dependent.

**SuspendMe** *Suspend calling process*

```
PROCEDURE SuspendMe ();
```

The procedure `SuspendMe` causes the calling process to become ineligible to run (i.e. to enter the *passive* state). If there are other processes eligible to run, then one of them is selected for execution. The exception `passiveProgram` is raised if no other process is eligible to run and no other process is waiting for an event to occur.

**NOTE:**

The suspended process can be reactivated when another process again makes it eligible by calling `Switch` or `Activate` or `SuspendMeAndActivate`.

**Activate** *Activate process*

```
PROCEDURE Activate (procId: ProcessId);
```

If the process identified by `procId` is passive or waiting, the procedure `Activate` causes that process to become eligible to run (i.e. to enter the *ready* state); otherwise it have no effect.

**NOTE:**

If the designated process was suspended by a call of `Wait`, it will become ready in the same way as if the event had occurred. Thus, if the procedure `Activate` (or `SuspendMeAndActivate`) is used to reactivate a waiting process, further checking will usually be required in that process to determine whether or not the event for which it was waiting had actually taken place.

**SuspendMeAndActivate** *Suspend current process and activate another*

```
PROCEDURE SuspendMeAndActivate (procId: ProcessId);
```

If the process identified by `procId` is passive, waiting, or is the calling process, the procedure `SuspendMeAndActivate` causes the calling process to become ineligible to run (i.e. to enter the *passive* state), and causes the process identified

by `procId` to become eligible to run (i.e. to enter the *ready* state); otherwise the call have no effect.

**NOTE**

`SuspendMeAndActivate(procId)` effectively performs an *atomic* (i.e. 'indivisible') sequence of the calls `SuspendMe()` and `Activate(procId)`. If applied to the identity of the calling process, the effect is to force a scheduling operation.

| **Switch** | *Switch to another process* |
|---|---|

```
PROCEDURE Switch (procId: ProcessId; VAR info: Parameter);
```

If the calling process has an urgency no greater than that of the process identified by `procId`, the procedure `Switch` causes the calling process to become ineligible to run (i.e. to enter the *passive* state), and resumes execution of the process identified by `procId`. The exception `processError` occurs (but need not be raised) if this process is already eligible to run. `info` is passed as a parameter to the process identified by `procId`. If the calling process is reactivated as a result of another process calling `Switch`, then `info` is assigned the value passed by that other call. If the calling process is reactivated as a result of another process calling `Activate` or `SuspendMeAndActivate`, then `info` is assigned the value `NIL`.

**NOTE:**

`Switch` is intended to allow a high-level coroutine facility for use within concurrent programs. Several consequences follow:

1. The process that is resumed will only be able to retrieve the parameter passed to it as `info` if it was ineligible to run by virtue of itself having called `Switch`.

2. The behaviour of `Switch` in situations where the urgency of the calling process is greater than the urgency of the process identified by `procId` is implementation-dependent.

3. While a call of `Switch` may be used instead of `Activate` to activate a process, `p`, of a higher urgency than the caller, `p` will not be able to use `Switch` if it wishes to reactivate that caller; it will be obliged to use `Activate` or `SuspendMeAndActivate`.

4. If the designated process was suspended by a call of `Wait`, it will become ready in the same way as if the event had occurred. Thus, if the procedure `Switch` (or `Activate` or `SuspendMeAndActivate`) is used to reactivate a waiting process, further checking will usually be required in that process to determine whether or not the event for which it was waiting had actually taken place.

| **Wait** | *Wait for event* |
|---|---|

```
PROCEDURE Wait ();
```

The procedure `Wait` causes the calling process to become ineligible to run (i.e. to enter the *waiting* state) if it is associated with a source of events. If there are other processes eligible to run, then one of them is selected for execution.

**NOTES:**

- One of the ready processes is selected for execution to replace the caller of `Wait`. This is on the assumption that, if there are ready processes that are not executing, the scheduler imposes a fixed upper limit on the number of executing processes.
- The process will remain ineligible to run until an event occurs from one of the sources to which it is attached, or until it is made eligible by another process calling `Switch` or `Activate` or `SuspendMeAndActivate`.
- If a process waiting for an event is reactivated by virtue of a call being made by another process to `Switch` or `Activate` or `SuspendMeAndActivate`, it will become ready in the same way as if the event had occurred. Thus, in such situations, further checking will usually be required to determine whether or not the event for which it was waiting had actually taken place.
- The behaviour of `Wait` if the calling process has not been attached to a source of events, or if another process is attached to the event source after the calling process has called `Wait` is implementation-dependent. In such circumstances the calling process may never again become eligible to run, and the system could deadlock.

**Attach** *Associate event source*

```
PROCEDURE Attach (eventSource: Sources);
```

The procedure `Attach` associates the source of events given by `eventSource` with the calling process. If the source of events is already associated with a process, then that association is first broken.

**Detach** *Dissociate event source*

```
PROCEDURE Detach (eventSource: Sources);
```

The procedure `Detach` dissociates the source of events given by `eventSource` from the program.

**NOTE:**

`Detach` has no effect if the program is not associated with `eventSource`.

**IsAttached** *Query event source*

```
PROCEDURE IsAttached (eventSource: Sources): BOOLEAN;
```

The function procedure `IsAttached` returns `TRUE` if and only if the source of events given by `eventSource` is associated with one of the processes in the program.

**Handler** *Query event handler*

```
PROCEDURE Handler (eventSource: Sources): ProcessId;
```

If the source of events identified by `eventSource` is associated with a process, the function procedure `Handler` is the identity of that process.

**NOTE:**

The value of the call `Handler(eventSource)` is implementation-dependent in the situation where the value of `eventSource` is not mapped to any real source of events.

**Me** *Query current process id*

```
PROCEDURE Me (): ProcessId;
```

The function procedure `Me` returns the identity of the calling process.

**MyParam** *Query current process parameter*

```
PROCEDURE MyParam (): Parameter;
```

The function procedure `MyParam` returns the value of the parameter denoted by `procParams` at the time the process was created.

**UrgencyOf** *Query process urgency*

```
PROCEDURE UrgencyOf (procId: ProcessId): Urgency;
```

The function procedure `UrgencyOf` returns the urgency of the process identified by `procId`.

**NOTE:**

This urgency of a process is statically assigned when the process is created; it is not possible for a process to alter its urgency dynamically.

**IsProcessesException** *Query exceptional state*

```
PROCEDURE IsProcessesException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of a `Processes` exception, the function procedure `IsProcessesException` returns `TRUE`; otherwise it retruns `FALSE`.

**ProcessesException** *Query exception id*

```
PROCEDURE ProcessesException (): ProcessesExceptions;
```

If the calling coroutine is in the state of exceptional execution because of the raising of a `Processes` exception, the function procedure

`ProcessesException` returns the value that identifies the raised exception; otherwise the language exception `exException` is be raised.

## 3.2 Module Semaphores

The module `Semaphores` provides facilities for potentially parallel parts of a program (i.e. *processes*) to exclude one another from regions of interaction by using the *semaphore mechanism* first proposed by Dijkstra. The semaphores provided by the module are general or counting semaphores (as opposed to binary semaphores).

The hidden type `SEMAPHORE` is used to identify semaphores.

```
TYPE
  SEMAPHORE;
```

Each semaphore have a unique identity. Associated with each semaphore there is a non-negative count, and a set of zero or more processes waiting for it to become free. A semaphore is said to be *'free'* if its associated count is non-zero.

After the module has been initialized, no semaphores are in existence.

**NOTES:**

- For convenience, the semantics of this module are illustrated in terms of the semantics of the `Processes` module. However, there is no requirement that an implementation provide `Processes` in addition to providing `Semaphores`.
- The behaviour of the procedures `Destroy`, `Claim`, `Release` and `CondClaim` in situations where the actual parameter passed is not a valid semaphore (i.e. is not the identity of a semaphore allocated by a call to `Create`) is implementation-dependent.

| **Create** | *Create new semaphore* |
|---|---|

```
PROCEDURE Create (VAR s: SEMAPHORE; initialCount: CARDINAL);
```

The procedure `Create` creates a new semaphore, if there are sufficient resources to do so, and assigns its identity to `s`; otherwise the `Semaphores`

exception is raised. The count associated with the semaphore is initialized to `initialCount`, and there will then be no process waiting for the semaphore to be free.

**Destroy** *Destroy semaphore*

```
PROCEDURE Destroy (VAR s: SEMAPHORE);
```

Provided that no process is waiting for the semaphore identified by the value of `s` to become free, the procedure `Destroy` removes that semaphore and recovers the resources used to implement it.

The variable `s` is set to a value that is invalid for semaphore operations.

**NOTE:**

The behaviour of `Destroy(s)` in situations where there are processes waiting for `s` to become free is implementation-dependent.

**Claim** *Claim semaphore*

```
PROCEDURE Claim (s: SEMAPHORE);
```

The procedure `Claim` claims the semaphore identified by `s`. If the count associated with `s` is non-zero, then it is decremented, and the calling process continues execution; otherwise the calling process becomes ineligible to run, and is added to the set waiting for `s` to become free.

**Release** *Unclaim semaphore*

```
PROCEDURE Release (s: SEMAPHORE);
```

The procedure `Release` unclaims the semaphore identified by `s`. If no process is waiting on `s`, the count associated with `s` is incremented; otherwise one process is selected from those waiting for `s` to become free; this process is removed from the waiting set, and becomes eligible to run (i.e. enter the ready state).

**NOTES:**

- No requirement is imposed about which of the waiting processes is chosen for activation.
- Preemption occurs if the newly eligible process has an urgency greater than that of the calling process.

### CondClaim — *Claim semaphore safely*

```
PROCEDURE CondClaim (s: SEMAPHORE): BOOLEAN;
```

If the call `Claim(s)` would have caused the calling process to become ineligible to run, the procedure `CondClaim` returns `FALSE`, and the count associated with `s` is unchanged. Otherwise the count associated with `s` is decremented, and the procedure returns `TRUE`.

**NOTE:**

The calling process is never suspended as a result of calling `CondClaim`.

### IsSemaphoresException — *Query exceptional state*

```
PROCEDURE IsSemaphoresException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of the `Semaphores` exception, the procedure `IsSemaphoresException` returns `TRUE`; otherwise it returns `FALSE`.

# Chapter 4

# String Manipulation

## 4.1 Module Strings

The module `Strings` provides facilities for manipulating character arrays as representations of strings. The procedures provided accept any character array type, but manipulate all as if their index types were whole numbers and zero-based.

The module also provides predicates that check whether an operation to assign, delete, insert, replace or append strings or characters will work without loss of information. These predicates check that parameters indexing the concrete representation of a string (i.e. the character array containing the string value) fall within its length, thereby allowing the programmer to maintain the string abstraction.

A general-purpose string type `String1` is provided for convenience when handling single characters by using a value constructor. An enumeration type `CompareResults` is provided for use when comparing string values:

```
TYPE
  String1 = ARRAY [0..0] OF CHAR;

  CompareResults = (less, equal, greater);
```

**NOTES:**

- The procedures provided accept any character array type (i.e. with any index type) but because of the use of open array parameters the procedures manipulate all as if their index types were whole numbers and zero-based.

- The array type `String1` may be used as the array type identifier of an array constructor when constructing an array value from a value of the character type. The constructed array value may be used as actual parameter in calls of procedures having a formal open array value parameter, for example to assign, insert, replace, append, concatenate, or find a single character.

- Since function procedures cannot return open arrays, many of the operations that might more logically have been provided as function procedures have to be provided as proper procedures.

- Predicates with the prefix '`Can`' and the suffix '`All`' are provided to check the operation-completion condition of string operations (e.g. `CanInsertAll` checks the operation-completion condition for `Insert`). Failure to satisfy the operation-completion condition of a string handling procedure does not lead to an exception; i.e. the semantics of string operations are defined for all well-formed parameters.

- Value parameters are used where a string value is not changed by a procedure. This is not just a matter of programming clarity, but allows the parameterization of programs using constants.

- Because string constants cannot be assigned to, nor appended to, nor have characters replaced or inserted, the predicates that test the operation completion conditions of procedures use VAR-parameters for the string parameters.

- The International Standard has not adopted a change to Modula-2 (described in the fourth edition of *Programming in Modula-2*) of always requiring a string terminator for a string value.

## **Length** *Query string length*

```
PROCEDURE Length (stringVal: ARRAY OF CHAR): CARDINAL;
```

The function procedure `Length` returns the length of the string value in `stringVal`. This is the same as the value of `LENGTH(stringVal)`.

## CanAssignAll — *Check whether Assign will succeed*

```
PROCEDURE CanAssignAll (sourceLength: CARDINAL;
                        VAR destination: ARRAY OF CHAR
                       ): BOOLEAN;
```

The function procedure `CanAssignAll` returns the value of the Boolean expression `sourceLength <= HIGH(destination) + 1`

**NOTES:**

- `CanAssignAll` may be used to check whether complete assignment of a string value to a string variable will be possible using, for example, the procedure `Assign`.
- Since a string variable must have at least one element, single-character string assignment is always valid.

## Assign — *Assign string value*

```
PROCEDURE Assign (source: ARRAY OF CHAR;
                  VAR destination: ARRAY OF CHAR);
```

The procedure `Assign` assigns the string value in `source` to `destination`. If the length of `source` exceeds the capacity of `destination`, the assigned value is truncated to the capacity of `destination`. If the length of `source` is less than the capacity of destination, a string terminator is appended to `source` when assignment takes place.

EXAMPLE - String assignment.

In these, and later, examples the following declarations are assumed:

```
VAR
  small: ARRAY [0 .. 4] OF CHAR;
  large: ARRAY [0 .. 255] OF CHAR;
  alpha: ARRAY ['A' .. 'E'] OF CHAR;
  ch: CHAR;
  found, areDiff: BOOLEAN;
  pos: CARDINAL;
```

1. `ch := "X";`
   `Assign(String1{ch}, small)`
   results in `small` having value `"X"`

2. `Assign("pq", small)`
   results in `small` having value `"pq"`

3. `Assign("", small)`
   results in `small` having value ""

4. `Assign("Hello!", small)`
   results in `small` having value `"Hello"` *without* string terminator

5. the call
   `CanAssignAll(6, small)`
   returns the value `FALSE`

6. `Assign("Go", alpha)`
   results in `alpha` having value `"Go"`

7. `small := "Hello"; large := "";`
   `IF CanAssignAll(LENGTH(small), large)`
   `THEN`
   `   Assign(small, large)`
   `END`
   results in `large` having value "Hello"

| **CanExtractAll** | *Check whether Extract will succeed* |
|---|---|

```
PROCEDURE CanExtractAll (sourceLength,
                         startPos,
                         numberToExtract: CARDINAL;
                         VAR destination: ARRAY OF CHAR
                        ): BOOLEAN;
```

The function procedure `CanExtractAll` returns the value of the Boolean expression

```
(startPos + numberToExtract <= sourceLength) AND
(numberToExtract <= HIGH(destination) + 1)
```

**NOTE:**

`CanExtractAll` may be used to check whether complete extraction of a substring from a string variable will be possible using, for example, the procedure `Extract`.

| **Extract** | *Extract substring* |
|---|---|

```
PROCEDURE Extract (source: ARRAY OF CHAR;
                   startPos,
                   numberToExtract: CARDINAL;
                   VAR destination: ARRAY OF CHAR);
```

The procedure `Extract` creates a new string value of at most `numberToExtract` characters extracted from `source`. Extraction starts at position `startPos` in `source`, and continues as long as there are characters left to extract from `source` and no more than `numberToExtract` characters have been extracted. If the length of the created string value exceeds the capacity of `destination`, the string value is truncated to the capacity of `destination`, and the truncated value is assigned to `destination`. If the length of the created string value is less than the capacity of `destination`, a string terminator is appended to the string value, and the resulting value is assigned to `destination`. An empty string value is extracted if `startPos` is greater than or equal to `LENGTH(source)`.

EXAMPLE - String extraction.

1. ```
   large := "ABCDE"; small := "";
   IF CanExtractAll(LENGTH (large), 2, 3, small) THEN
      Extract(large, 2, 3, small)
   END
   ```
   results in `small` having value `"CDE"`

2. ```
   large := "ABCDE"; small := "";
   Extract(large, 2, 3, small)
   ```
   results in `small` having value `"CDE"`

## CanDeleteAll — *Check whether Delete will succeed*

```
PROCEDURE CanDeleteAll (stringLength,
                        startPos,
                        numberToDelete: CARDINAL
                       ): BOOLEAN;
```

The function procedure `CanDeleteAll` returns the value of the Boolean expression `startPos + numberToDelete <= stringLength`

**NOTE:**

`CanDeleteAll` may be used to check whether complete deletion of a substring value from a string variable will be possible using, for example, the procedure `Delete`.

## Delete — *Delete substring*

```
PROCEDURE Delete (VAR stringVar: ARRAY OF CHAR;
                  startPos,
                  numberToDelete: CARDINAL);
```

The procedure `Delete` creates a new string value by deleting at most `numberToDelete` characters from `stringVar`. Deletion starts at position `startPos` in `stringVar` and continues as long as there are characters left to delete in `stringVar` and no more than `numberToDelete` characters have been deleted. If any characters are deleted, a string terminator is appended to the created string value, and the resulting value is assigned to `stringVar`. The string value in `stringVar` is not altered if `startPos` is greater than or equal to `LENGTH(stringVar)`.

EXAMPLE - String deletion.

1. ```
   small := "ABCDE";
   IF CanDeleteAll(LENGTH(small), 2, 2) THEN
      Delete(small, 2, 2)
   END
   ```
   results in `small` having value `"ABE"`

2. after the assignment
   ```
   small := "ABC";
   ```
   ```

the call
```
CanDeleteAll(3, 2, 2)
```
returns the value `FALSE`

3. `small := "ABC";`
   `Delete(small, 2, 2)`
   results in `small` having value `"AB"`

## CanInsertAll — *Check whether Insert will succeed*

```
PROCEDURE CanInsertAll (sourceLength, startPos: CARDINAL;
                        VAR destination: ARRAY OF CHAR
                       ): BOOLEAN;
```

The function procedure `CanInsertAll` returns the value of the Boolean expression
```
(startPos <= LENGTH(destination)) AND
(sourceLength + LENGTH(destination) <=
HIGH(destination) + 1)
```

**NOTE:**

`CanInsertAll` may be used to check whether complete insertion of a string value into a string variable will be possible using, for example, the procedure `Insert`.

## Insert — *Insert substring*

```
PROCEDURE Insert (source: ARRAY OF CHAR;
                  startPos: CARDINAL;
                  VAR destination: ARRAY OF CHAR);
```

The procedure `Insert` creates a new string value by inserting the substring `source` into `destination`. The string in `destination` is first splitted at the position given by `startPos`; the created string value is the concatenation of the first part of `destination`, the substring `source`, and the second part of `destination`. If the length of the created string value exceeds the capacity of `destination`, the string value is truncated to the capacity of `destination`, and the truncated value is assigned to `destination`. If the length of the created

string value is less than the capacity of `destination`, a string terminator is appended to the string value, and the resulting value is assigned to `destination`. The string value in `destination` is not altered if `startPos` is greater than or equal to `LENGTH(destination)`.

EXAMPLE - String insertion.

1. after the assignment
   ```
   small := "ABCD";
   ```
   the call
   ```
   CanInsertAll(LENGTH("XYZ"), 2, small)
   ```
   returns the value `FALSE`

2. ```
   small := "ABCD";
   Insert("XYZ", 2, small)
   ```
   results in `small` having value `"ABXYZ"` *without* terminator

3. ```
   large := "ABC";
   IF CanInsertAll(3, 2, large) THEN
      Insert("XYZ", 2, large)
   END
   ```
   results in `large` having value `"ABXYZC"`

4. ```
   large := "ABCD"; ch := "X";
   Insert(String1{ch}, 2, large)
   ```
   results in `large` having value `"ABXCD"`

## CanReplaceAll — *Check whether Replace will succeed*

```
PROCEDURE CanReplaceAll (sourceLength, startPos: CARDINAL;
                         VAR destination: ARRAY OF CHAR
                        ): BOOLEAN;
```

The function procedure `CanReplaceAll` returns is the value of the Boolean expression `sourceLength + startPos <= LENGTH(destination)`

**NOTES:**

- `CanReplaceAll` may be used to check whether complete replacement of a substring value within a string variable will be possible using, for example, the procedure `Replace`.
```

- The preservation of the string abstraction is taken as the goal of the string module. This means that the operation completion condition of `CanReplaceAll` only tests whether the proposed replacement is valid within the given string length; the procedure `Replace` always preserves the length of its `destination` string.

## Replace — *Replace substring*

```
PROCEDURE Replace (source: ARRAY OF CHAR;
                   startPos: CARDINAL;
                   VAR destination: ARRAY OF CHAR);
```

The procedure `Replace` modifies the string value from `destination` by overwriting characters in `destination` with characters extracted from the string value in `source`. Overwriting begins at the position given by `startPos` and continues as long as there are characters left to overwrite in `destination` and characters left to extract from `source`. The string value in `destination` is not altered if `startPos` is greater than or equal to `LENGTH(destination)`.

**NOTE:**

The length of the string value in `destination` is always preserved by `Replace`.

EXAMPLE - String replacement.

1. after the assignment
   ```
   small := "ABC"
   ```
   the call
   ```
   CanReplaceAll(LENGTH("XY"), 2, small)
   ```
   returns the value `FALSE`

2. ```
   small := "ABC";
   Replace("XY", 2, small)
   ```
   results in `small[0]` having value `"ABX"`

3. ```
   large := "ABCDEF";
   IF CanReplaceAll(3, 2, large)
   THEN
      Replace("XYZ", 2, large)
   END
   ```
   results in `large` having value `"ABXYZF"`

## CanAppendAll — *Check whether Append will succeed*

```
PROCEDURE CanAppendAll (sourceLength: CARDINAL;
                        VAR destination: ARRAY OF CHAR
                       ): BOOLEAN;
```

The function procedure `CanAppendAll` returns the value of the Boolean expression

```
LENGTH(destination) + sourceLength <=
HIGH(destination) + 1
```

**NOTE:**

`CanAppendAll` may be used to check whether it will be possible to append a string value to another string value held within a string variable using, for example, the procedure `Append`.

## Append — *Append string*

```
PROCEDURE Append (source: ARRAY OF CHAR;
                  VAR destination: ARRAY OF CHAR);
```

The procedure `Append` creates a new string value by appending the string value in `source` onto `destination`. If the length of the created string value exceeds the capacity of `destination`, the string value is truncated to the capacity of `destination`, and the truncated value is assigned to `destination`. If the length of the created string value is less than the capacity of `destination`, a string terminator is appended to the string value, and the resulting value is assigned to `destination`.

EXAMPLE - Appending to strings.

1. after the assignment
   ```
   small := "pqr"
   ```
   the call
   ```
   CanAppendAll(LENGTH("XYZ"), small)
   ```
   returns the value `FALSE`
2. ```
   small := "pqr";
   Append("XYZ", small)
   ```
   results in `small` having value `"pqrXY"` *without* terminator
   ```

3. `small := "pqr";`
   `ch := "s";`
   `Append(String1{ch}, small)`
   results in `small` having value `"pqrs"`

## CanConcatAll — *Check whether Concat will succeed*

```
PROCEDURE CanConcatAll (source1Length, source2Length: CARDINAL;
                        VAR destination: ARRAY OF CHAR
                       ): BOOLEAN;
```

The function procedure `CanConcatAll` returns the value of the Boolean expression
`source1Length + source2Length <= HIGH(destination) + 1`

**NOTE:**

`CanConcatAll` may be used to check whether complete concatenation of two string values will be possible within the capacity of a specified string variable using, for example, the procedure `Concat`.

## Concat — *Concatenate strings*

```
PROCEDURE Concat (source1, source2: ARRAY OF CHAR;
                  VAR destination: ARRAY OF CHAR);
```

The procedure `Concat` creates a new string value by appending the substring value `source2` onto `source1`. If the length of the created string value exceeds the capacity of `destination`, the string value is truncated to the capacity of `destination`, and the truncated value is assigned to `destination`. If the length of the created string value is less than the capacity of `destination`, a string terminator is appended to the string value, and the resulting value is assigned to `destination`.

EXAMPLE - String concatenation.

1. after the assignment
   `small := "pqr"`

the call
```
CanConcatAll(4, LENGTH(small), small)
```
returns the value `FALSE`

2. `small := "pqr";`
   `Concat("WXYZ", small, small)`
   results in `small` having value `"WXYZp"` *without* terminator

3. `small := "jk"; large := "";`
   `ch := "s";`
   `Concat(String1{ch}, small, large)`
   results in `large` having value `"skj"`

## Capitalize — *Capitalize string*

```
PROCEDURE Capitalize (VAR stringVar: ARRAY OF CHAR);
```

The procedure `Capitalize` applies the standard function `CAP` to each character of the string value in `stringVar`.

**NOTE:**

`Capitalize` may be used to achieve case-insensitive use of the procedures `Compare`, `FindNext`, `FindPrev` and `FindDiff`.

EXAMPLE - String capitalization.

The following example assumes a capitalization mapping which maps p to P, q to Q and r to R.

1. `small := "pqr";`
   `Capitalize(small)`
   results in `small` having value `"PQR"`

## Compare — *Compare strings*

```
PROCEDURE Compare (stringVal1, stringVal2: ARRAY OF CHAR
                  ): CompareResults;
```

The function procedure `Compare` returns a value of the enumeration type `CompareResults` depending on the lexical ordering of the type `CHAR`. The

value returned is `less`, `equal` or `greater` according as the string value in `stringVal1` is lexically less than, equal to, or greater than the string value in `stringVal2`.

**NOTES:**

- The result of this function is dependent upon the full collating sequence of character values. This sequence is implementation-defined, although it must have certain properties.
- In general, the result of this function is case-sensitive.

EXAMPLE - String comparison.

1. `Compare("", "")` returns `equal`
2. `Compare("", "abc")` returns `less`
3. `Compare("abc", "")` returns `greater`
4. `Compare("pqr", "pqr")` returns `equal`
5. `Compare("pqr", "pqrstuv")` returns `less`
6. `Compare("pqrstuv", "pqr")` returns `greater`
7. `Compare("abc", "pqr")` returns `less`
8. `Compare("pqr", "abc")` returns `greater`
9. `Compare("abcdef ", "p")` returns `less`
10. `Compare("p", "abcdef ")` returns `greater`

| **Equal** | *Compare strings* |
|---|---|

```
PROCEDURE Equal (stringVal1, stringVal2: ARRAY OF CHAR): BOOLEAN;
```

The function procedure `Equal` returns the value of the Boolean expression
`Strings.Compare(stringVal1, stringVal2) = Strings.equal`

**FindNext** *Search string forward*

```
PROCEDURE FindNext (pattern, stringToSearch: ARRAY OF CHAR;
                    startPos: CARDINAL;
                    VAR patternFound: BOOLEAN;
                    VAR posOfPattern: CARDINAL);
```

The procedure `FindNext` searches forwards for the next occurrence of `pattern` in `stringToSearch`, starting the search in `stringToSearch` at position `startPos`. If `pattern` is found, the value `TRUE` is assigned to `patternFound`, and `posOfPattern` contains the starting position of `pattern` in `stringToSearch`; `posOfPattern` is in the range `[startPos..LENGTH(stringToSearch)-1]`. Otherwise the value `FALSE` is assigned to `patternFound` and `posOfPattern` is unchanged.

**NOTES:**

- The `pattern` might be found at the given value of `startPos`.
- If `startPos > LENGTH(stringToSearch) - LENGTH(pattern)` then `patternFound` is returned as `FALSE`.

EXAMPLE - Forwards string search.

1. ```
   large := "Hello hello hello";
   FindNext("ll", large, 0, found, pos)
   ```
   results in:
   `found` having value `TRUE`
   `pos` having value `2`

2. ```
   large := "Hello hello hello";
   FindNext("ll", large, 0, found, pos);
   FindNext("ll", large, pos+1, found, pos)
   ```
   results in:
   `found` having value `TRUE`
   `pos` having value `8`

3. ```
   large := "abcdefghijklmnopqrstuvwxyz";
   ch := "x";
   FindNext(String1fchg, large, 0, found, pos)
   ```
   results in:
   ```

found having value TRUE
pos having value 23

4. large := "abcdefghijklmnopqrstuvwxyz";
   ch := "x";
   FindNext(String1fchg, large, 26, found, pos)
   results in:
   found having value FALSE
   pos remains unchanged

## FindPrev — *Search string backward*

```
PROCEDURE FindPrev (pattern, stringToSearch: ARRAY OF CHAR;
                    startPos: CARDINAL;
                    VAR patternFound: BOOLEAN;
                    VAR posOfPattern: CARDINAL);
```

The procedure FindPrev looks backwards for an occurrence of pattern in the string value in stringToSearch, starting the search in stringToSearch at position startPos. If pattern is found, the value TRUE is assigned to patternFound, and posOfPattern contains the starting position of pattern in stringToSearch; posOfPattern is in the range [0..startPos]. Otherwise the value FALSE is assigned to patternFound and posOfPattern is unchanged.

**NOTES:**

- The pattern might be found at the given value of startPos.
- If startPos > LENGTH(stringToSearch)-LENGTH(pattern) the whole string value is searched.

EXAMPLE - Backwards string search.

1. large := "aabbbcccc";
   FindPrev("cc", large, 200, found, pos)
   results in:
   found having value TRUE
   pos having value 7

2. `large := "aabbbcccc";`
   `FindPrev("cc", large, 200, found, pos);`
   `FindPrev("cc", large, pos-1, found, pos)`
   results in:
   `found` having value `TRUE`
   `pos` having value `6`

3. `large := "Maybe this makes sense";`
   `FindPrev("se", large, 200, found, pos)`
   results in:
   `found` having value `TRUE`
   `pos` having value `20`

4. `large := "Maybe this makes sense";`
   `FindPrev("se", large, 20, found, pos);`
   `FindPrev("se", large, pos-1, found, pos)`
   results in:
   `found` having value `TRUE`
   `pos` having value `17`

## FindDiff — *Find position of string difference*

```
PROCEDURE FindDiff (stringVal1, stringVal2: ARRAY OF CHAR;
                    VAR differenceFound: BOOLEAN;
                    VAR posOfDifference: CARDINAL);
```

The procedure `FindDiff` compares the string values in `stringVal1` and `stringVal2`. The value `FALSE` is assigned to `differenceFound` if the string values are equal and `TRUE` otherwise. If `differenceFound` is `TRUE`, the position of the first difference between the string values is assigned to `posOfDifference`; otherwise `posOfDifference` is unchanged.

**Examples** - Finding string differences.

1. `FindDiff("", "", areDiff, pos)` results in:
   `areDiff` having value `FALSE`
   `pos` being unchanged

2. `FindDiff("abc", "", areDiff, pos)` results in:
   `areDiff` having value `TRUE`
   `pos` having value `0`

3. `FindDiff("", "abc", areDiff, pos)` results in:
   `areDiff` having value `TRUE` `pos` having value `0`

4. `FindDiff("pqr", "pqt", areDiff, pos)` results in:
   `areDiff` having value `TRUE`
   `pos` having value `2`

5. `FindDiff("pqr", "pqrstuv", areDiff, pos)` results in:
   `areDiff` having value `TRUE`
   `pos` having value `3`

6. `FindDiff("pqrstuv", "pqr", areDiff, pos)` results in:
   `areDiff` having value `TRUE`
   `pos` having value `3`

# Chapter 5

# String Conversions

The string conversions library allows the conversion of the values of numeric data types to and from character string representations. The modules `WholeStr`, `RealStr`, and `LongStr` provide simple high-level facilities for converting to and from strings and whole number and real number data types. Low-level facilities are provided by the corresponding modules `WholeConv`, `RealConv`, and `LongConv`. Common data types and values that are used in the definition modules are defined by the module `ConvTypes`.

The formats for string conversions correspond to those for numeric input and output, except that the numeric output routines provide for (right) alignment in a specified field width | see 9.2.2.2 and 9.2.2.3.

## 5.1 Common Data Types

The module `ConvTypes` defines types and values that are used in the high-level and low-level string conversion definition modules. Where appropriate, the conversion modules define types in terms of those defined in `ConvTypes`. Direct import from this module is not normally necessary in modules that are clients of the high-level conversion modules.

### 5.1.1 Module ConvTypes

The module `ConvTypes` defines the enumeration type `ConvResults` with values for expressing the format of strings that are interpreted as representing values of numeric data types. The module also defines the types `ScanClass` and

`ScanState`, which the low-level conversion modules use in the definition of procedures that control lexical scanning.

```
TYPE
  ConvResults =     (* Values of this type are used
                       to express the format of a string: *)
  (
    strAllRight,    (* the string format is correct
                       for the corresponding conversion *)
    strOutOfRange,  (* the string is well-formed but the value
                       cannot be represented *)
    strWrongFormat, (* the string is in the wrong format
                       for the conversion *)
    strEmpty        (* the given string is empty *)
  );

  ScanClass =  (* Values of this type are used to classify input
                  to finite state scanners: *)
  (
    padding,   (* a leading or padding character at this point
                  in the scan - ignore it *)
    valid,     (* a valid character at this point
                  in the scan - accept it *)
    invalid,   (* an invalid character at this point
                  in the scan - reject it *)
    terminator (* a terminating character at this point
                  in the scan (not part of token) *)
  );

  ScanState = (* The type of lexical scanning
                 control procedures *)
   PROCEDURE (CHAR, VAR ScanClass, VAR ScanState);
```

## 5.2 High-Level String Conversion Modules

Separate high-level string conversion modules are defined for the whole number types (`INTEGER` and `CARDINAL`), and for the real number types (`REAL` and `LONGREAL`). These all use decimal notation.

In calls of procedures converting from strings, the source parameter `str` is as-

sumed to contain a string value (which is terminated by the string terminator character if the length of the string is less than the capacity of the array). While leading spaces are ignored, the entire remainder of the string has to be in the correct format for the conversion to take place.

In calls of procedures converting to strings, the destination parameter `str` is assigned a string value (which is terminated by the string terminator character if the length of the string is less than the capacity of the array). If the destination parameter is of insufficient capacity, the string is truncated. Users may determine whether truncation will occur by using procedures such as `LengthCard` from the low-level string conversion modules | see 9.5.3.1.2.

**NOTE:**

The string conversion procedures may be used with strings read by `STextIO.ReadToken` or `TextIO.ReadToken`, if it is required that entire space-character delimited tokens are in the correct format | see 9.2.2.1.2.

### 5.2.1 EXAMPLE - Conversion of strings read by ReadToken

This example shows how space-character delimited tokens that cannot be converted to values of type `CARDINAL` can be replaced by descriptive text:

```
firstOnLine := TRUE;
STextIO.ReadToken(inStr);
WHILE SIOResult.ReadResult() <> SIOResult.endOfInput DO
  WholeStr.StrToCard(inStr, inCard, inRes);
  CASE inRes OF
  | ConvTypes.strAllRight .. ConvTypes.strWrongFormat:
     IF firstOnLine THEN
        firstOnLine := FALSE
     ELSE
        STextIO.WriteString(" ")
     END
  | ConvTypes.strEmpty:
  firstOnLine := TRUE;
  STextIO.SkipLine
  END;
  CASE inRes OF
  | ConvTypes.strAllRight:
     STextIO.WriteString(inStr)
  | ConvTypes.strOutOfRange:
```

```
      STextIO.WriteString("out-of-range")
  | ConvTypes.strWrongFormat:
      STextIO.WriteString("wrong-format")
  | ConvTypes.strEmpty:
      STextIO.WriteLn
  END;
  STextIO.ReadToken(inStr)
END;
```

It is assumed here that the capacity of the character array variable `inStr` is sufficient to accommodate the longest token in the input stream.

### 5.2.2 Module WholeStr

The module `WholeStr` provides procedures for the conversion of values of the type `INTEGER` and the type `CARDINAL` to and from strings.

The string form of a signed whole number is

```
["+" | "-"], decimal digit, {decimal digit}
```

The string form of an unsigned whole number is

```
decimal digit, {decimal digit}
```

**StrToInt** *Convert string to INTEGER*

```
PROCEDURE StrToInt (str: ARRAY OF CHAR;
                    VAR int: INTEGER;
                    VAR res: ConvResults);
```

The procedure `StrToInt` ignores leading spaces in `str` and assigns values to `int` and `res` as follows:

`strAllRight` if the remainder of `str` represents a complete signed whole number in the range of the type `INTEGER`; the value of this number is assigned to `int`;

`strOutOfRange` if the remainder of `str` represents a complete signed whole number but its value is out of the range of the type `INTEGER`; the value `MAX(INTEGER)` or `MIN(INTEGER)` is assigned to `int` according to the sign of the number;

`strWrongFormat` if there are remaining characters in `str` but these are not in the form of a complete signed whole number; the value of `int` is not defined;

`strEmpty` if there are no remaining characters in `str` | the value of `int` is not defined.

### IntToStr — *Convert INTEGER to string*

```
PROCEDURE IntToStr (int: INTEGER; VAR str: ARRAY OF CHAR);
```

The procedure `IntToStr` assigns to `str` the possibly truncated string corresponding to the value of `int`. A sign is included only for negative values.

### StrToCard — *Convert string to CARDINAL*

```
PROCEDURE StrToCard (str: ARRAY OF CHAR;
                     VAR card: CARDINAL;
                     VAR res: ConvResults);
```

The procedure `StrToCard` ignores leading spaces in `str` and assigns values to `card` and `res` as follows:

`strAllRight` if the remainder of `str` represents a complete unsigned whole number in the range of the type `CARDINAL`; the value of this number is assigned to `card`;

`strOutOfRange` if the remainder of `str` represents a complete unsigned whole number but its value is out of the range of the type `CARDINAL`; the value `MAX(CARDINAL)` is assigned to `card`;

`strWrongFormat` if there are remaining characters in `str` but these are not in the form of a complete unsigned whole number; the value of `card` is not defined;

`strEmpty` if there are no remaining characters in `str`; the value of `card` is not defined.

| **CardToStr** | *Convert CARDINAL to string* |
|---|---|

```
PROCEDURE CardToStr (card: CARDINAL; VAR str: ARRAY OF CHAR);
```

The procedure `CardToStr` assigns to `str` the possibly truncated string corresponding to the value of `card`.

### 5.2.3 Modules RealStr and LongStr

The modules `RealStr` and `LongStr` provide procedures for the conversion of real numbers to and from strings. In the case of `RealStr`, real number parameters are of the type `REAL`. In the case of `LongStr`, real number parameters are of the type `LONGREAL`.

The semantics of the two modules is the same, except that when module `RealStr` refers to the pervasive type `REAL`, the corresponding procedure in `LongStr` refers to the pervasive type `LONGREAL`.

**NOTE:**

The above statement is merely to avoid needless repetition of the semantics for `LongStr`.

The string form of a signed fixed-point real number is

```
["+" | "-"], decimal digit, {decimal digit},
[".", {decimal digit}]
```

The string form of a signed floating-point real number is

```
signed fixed-point real number,
"E"|"e", ["+" | "-"], decimal digit, {decimal digit}
```

## StrToReal — *Convert string to real*

```
PROCEDURE StrToReal (str: ARRAY OF CHAR;
                     VAR real: REAL;
                     VAR res: ConvResults);
PROCEDURE StrToReal (str: ARRAY OF CHAR;
                     VAR real: LONGREAL;
                     VAR res: ConvResults);
```

The procedure `StrToReal` ignores leading spaces in `str` and assigns values to `res` and `real` as follows:

`strAllRight` if the remainder of `str` represents a complete signed real number in the range of the type of `real`; the value of this number is assigned to real;

`strOutOfRange` if the remainder of `str` represents a complete signed real number but its value is out of the range of the type of `real`; the maximum or minimum value of the type of `real` is assigned to `real` according to the sign of the number;

`strWrongFormat` if there are remaining characters in `str` but these are not in the form of a complete signed real number; the value of `real` is not defined;

`strEmpty` if there are no remaining characters in `str`; the value of `real` is not defined.

## RealToFloat — *Convert real to string (float notation)*

```
PROCEDURE RealToFloat (real: REAL;
                       sigFigs: CARDINAL;
                       VAR str: ARRAY OF CHAR);
PROCEDURE RealToFloat (real: LONGREAL;
                       sigFigs: CARDINAL;
                       VAR str: ARRAY OF CHAR);
```

The procedure `RealToFloat` assigns to `str` the possibly truncated string corresponding to the value of `real` in floating-point form. A sign is included only

for negative values. One significant digit is included in the whole number part. The signed exponent part is included only if the exponent value is not 0. If the value of `sigFigs` is greater than 0, that number of significant digits is included, otherwise an implementation-defined number of significant digits is included. The decimal point is not included if there are no significant digits in the fractional part.

**RealToEng** *Convert real to string (eng. notation)*

```
PROCEDURE RealToEng (real: REAL;
                     sigFigs: CARDINAL;
                     VAR str: ARRAY OF CHAR);
PROCEDURE RealToEng (real: LONGREAL;
                     sigFigs: CARDINAL;
                     VAR str: ARRAY OF CHAR);
```

The procedure `RealToEng` behaves like the procedure `RealToFloat` except that the number is scaled with one to three digits in the whole number part and with an exponent that is a multiple of three.

**RealToFixed** *Convert real to string (fixed notation)*

```
PROCEDURE RealToFixed (real: REAL;
                       place: INTEGER;
                       VAR str: ARRAY OF CHAR);
PROCEDURE RealToFixed (real: LONGREAL;
                       place: INTEGER;
                       VAR str: ARRAY OF CHAR);
```

The procedure `RealToFixed` assigns to `str` the possibly truncated string corresponding to the value of `real` in fixed-point form. A sign is included only for negative values. At least one digit is included in the whole number part. The value is rounded to the given value of `place` relative to the decimal point. The decimal point is suppressed if `place` is less than 0.

| **RealToStr** | *Convert real to string (auto notation)* |
|---|---|

```
PROCEDURE RealToStr (real: REAL; VAR str: ARRAY OF CHAR);
PROCEDURE RealToStr (real: LONGREAL; VAR str: ARRAY OF CHAR);
```

If the sign and magnitude of `real` can be shown within the capacity of `str`, the call `RealToStr(real,str)` behaves like the call `RealToFixed(real,place,str)`, with a value of `place` chosen to fill exactly the remainder of `str`. Otherwise, the call behaves as the call `RealToFloat(real,sigFigs,str)`, with a value of `sigFigs` of at least one, but otherwise limited to the number of significant digits that can be included together with the sign and exponent part in `str`.

## 5.3 Low-Level String Conversion Modules

Separate low-level string conversion modules are defined for the whole number types (`INTEGER` and `CARDINAL`), and for the real number types (`REAL` and `LONGREAL`). These all use decimal notation.

Procedures are defined to return the length of the string that is required to represent a given value, to return the format of a given string, to return the value of a string known to be in the correct format for conversion, and to allow control of lexical scanning of character sequences.

**NOTE:**

The types designated by `ConvTypes.ScanClass` and `ConvTypes.ScanState` are not given aliases in the low-level conversion modules. This is because clients of a separate finite state interpreter module need to refer only to procedures such as `WholeConv.ScanInt`, which represent the start state, and not to the types themselves.

### 5.3.1 EXAMPLE - Use of ScanInt

The following procedure uses `WholeConv.ScanInt` to locate the position of the first character in a string that follows any leading spaces and to locate the position of the first character that is not part of an integer. These positions will coincide if the string does not contain an integer after any leading spaces, and will be equal to the string length if no such character is contained in the string.

```
PROCEDURE FindInt(str: ARRAY OF CHAR; VAR first, next: CARDINAL);
VAR
  ch: CHAR;
  len, index: CARDINAL;
  state: ConvTypes.ConvState;
  class: ConvTypes.ConvClass;
BEGIN
  len := LENGTH(str);
  index := 0;
  first := len;
  state := WholeConv.ScanInt;
  LOOP
    IF index = len THEN EXIT END;
    state(str[index], class, state);
    CASE class OF
    | ConvTypes.padding:
    | ConvTypes.valid:
      IF index < first THEN first := index END;
    | ConvTypes.invalid, ConvTypes.terminator:
      EXIT
    END;
    INC(index)
  END;
  next := index
END FindInt;
```

### 5.3.2 Module WholeConv

The module `WholeConv` provides low-level string conversion procedures for values of the type `INTEGER` and values of the type `CARDINAL`.

| **ScanInt** | *Scan one character of INTEGER* |
|---|---|

```
PROCEDURE ScanInt (inputCh: CHAR;
                   VAR chClass: ConvTypes.ScanClass;
                   VAR nextState: ConvTypes.ScanState);
```

The procedure `ScanInt` assigns values to `chClass` and `nextState` depending upon the value of `inputCh` as shown in the following table:

| Procedure | inputCh | chClass | nextState<br>a procedure with<br>behaviour of |
|---|---|---|---|
| ScanInt | space | padding | ScanInt |
| | sign | valid | *S* |
| | decimal digit | valid | *W* |
| | other | invalid | ScanInt |
| *S* | decimal digit | valid | *W* |
| | other | invalid | *S* |
| *W* | decimal digit | valid | *W* |
| | other | terminator | \| |

**NOTE:**

The procedure `ScanInt` corresponds to the start state of a finite state machine to scan for a character sequence that forms a signed whole number. Like `ScanCard` and the corresponding procedures in the other low-level string conversion modules, it may be used to control the actions of a finite state interpreter. As long as the value of `chClass` is other than `terminator` or `invalid`, the interpreter should call the procedure whose value is assigned to `nextState` by the previous call, supplying the next character from the sequence to be scanned. It may be appropriate for the interpreter to ignore characters classified as `invalid`, and proceed with the scan. This would be the case, for example, with interactive input, if only valid characters are being echoed in order to give interactive users an immediate indication of badly-formed data. If the character sequence ends before one is classified as `terminator`, the string-terminator character should be supplied as input to the finite state scanner. If the preceding character sequence formed a complete number, the string-terminator is classified as `terminator`, otherwise it is classified as `invalid`.

## FormatInt — *Query INTEGER format*

```
PROCEDURE FormatInt (str: ARRAY OF CHAR): ConvResults;
```

The function procedure `FormatInt` queries the format of `str` and returns:

`strAllRight` if `str` has a value representing a complete signed whole number that is in the range of the type `INTEGER`;

`strOutOfRange` if `str` has a value representing a complete signed whole number that is not in the range of `INTEGER`;

`strWrongFormat` if `str` has a value with remaining characters that do not form a complete signed whole number;

`strEmpty` if `str` has a value with no remaining characters;

`FormatInt` ignores any leading space characters in `str`.

### ValueInt — *Query INTEGER value*

```
PROCEDURE ValueInt (str: ARRAY OF CHAR): INTEGER;
```

If `str` has a value representing a signed whole number, the function procedure `ValueInt` returns the `INTEGER` value that corresponds to that number. Otherwise, the `WholeConv` exception is raised.

### LengthInt — *Query INTEGER length*

```
PROCEDURE LengthInt (int: INTEGER): CARDINAL;
```

The function procedure `LengthInt` returns the number of characters in the string representation of the value of `int`.

**NOTE:**

This value corresponds to the capacity of an array `str` which is of the minimum capacity needed to avoid truncation of the result in the call `WholeStr.IntToStr(int,str)`.

### ScanCard — *Scan one character of CARDINAL*

```
PROCEDURE ScanCard (inputCh: CHAR;
                    VAR chClass: ConvTypes.ScanClass;
                    VAR nextState: ConvTypes.ScanState);
```

The procedure `ScanCard` assigns values to `chClass` and `nextState` depending upon the value of `inputCh` as shown in the following table:

| Procedure | inputCh | chClass | nextState<br>a procedure with<br>behaviour of |
|---|---|---|---|
| ScanCard | space | padding | ScanCard |
| | decimal digit | valid | *W* |
| | other | invalid | ScanCard |
| *W* | decimal digit | valid | *W* |
| | other | terminator | \| |

**FormatCard** *Query CARDINAL format*

```
PROCEDURE FormatCard (str: ARRAY OF CHAR): ConvResults;
```

The function procedure `FormatCard` queries the format of `str` and returns:

`strAllRight` if `str` has a value representing a complete unsigned whole number that is in the range of the type `CARDINAL`;

`strOutOfRange` if `str` has a value representing a complete unsigned whole number that is not in the range of `CARDINAL`;

`strWrongFormat` if `str` has a value with remaining characters that do not form a complete unsigned whole number;

`strEmpty` if `str` has a value with no remaining characters;

`FormatCard` ignores any leading space characters in `str`.

**ValueCard** *Query CARDINAL value*

```
PROCEDURE ValueCard (str: ARRAY OF CHAR): CARDINAL;
```

If `str` has a value representing an unsigned whole number, the function procedure `ValueCard` returns the `CARDINAL` value that corresponds to that number. Otherwise, the `WholeConv` exception is raised.

**LengthCard** *Query CARDINAL length*

```
PROCEDURE LengthCard (card: CARDINAL): CARDINAL;
```

The function procedure `LengthCard` returns the number of characters in the string representation of the value of `card`.

**NOTE:**

This value corresponds to the capacity of an array `str` which is of the minimum capacity needed to avoid truncation of the result in the call `WholeStr.CardToStr(card,str)`.

**IsWholeConvException** *Query exceptional state*

```
PROCEDURE IsWholeConvException (): BOOLEAN;
```

The function procedure `IsWholeConvException` returns `TRUE` if the calling coroutine is in the state of exceptional execution because of the raising of the `WholeConv` exception, and `FALSE` otherwise.

### 5.3.3 Modules RealConv and LongConv

The modules `RealConv` and `LongConv` provide low-level string conversion procedures for values of the type `REAL` and values of the type `LONGREAL`. In the case of `RealConv`, real number parameters are of the type `REAL`. In the case of `LongConv`, real number parameters are of the type `LONGREAL`.

The semantics of the two modules are the same, except that when module `RealConv` refers to the pervasive type `REAL`, the corresponding procedure in `LongConv` refers to the pervasive type `LONGREAL`.

**NOTE:**

The above statement is merely to avoid needless repetition of the semantics for `LongConv`.

**ScanReal** *Scan one character of real*

```
PROCEDURE ScanReal (inputCh: CHAR;
                    VAR chClass: ConvTypes.ScanClass;
                    VAR nextState: ConvTypes.ScanState);
PROCEDURE ScanReal (inputCh: CHAR;
                    VAR chClass: ConvTypes.ScanClass;
                    VAR nextState: ConvTypes.ScanState);
```

The procedure `ScanReal` assigns values to `chClass` and `nextState` depending upon the value of `inputCh` as shown in the following table:

| Procedure | inputCh | chClass | nextState a procedure with behaviour of |
|---|---|---|---|
| ScanReal | space | padding | ScanReal |
| | sign | valid | *RS* |
| | decimal digit | valid | *P* |
| | other | invalid | ScanReal |
| *RS* | decimal digit | valid | *P* |
| | other | invalid | *RS* |
| *P* | decimal digit | valid | *P* |
| | "." | valid | *F* |
| | "E" | valid | *E* |
| | other | terminator | \| |
| *F* | decimal digit | valid | *F* |
| | "E" | valid | *E* |
| | other | terminator | \| |
| *E* | sign | valid | *SE* |
| | decimal digit | valid | *WE* |
| | other | invalid | *E* |
| *SE* | decimal digit | valid | *WE* |
| | other | invalid | *SE* |
| *WE* | decimal digit | valid | *WE* |
| | other | terminator | \| |

## FormatReal — *Query real format*

```
PROCEDURE FormatReal (str: ARRAY OF CHAR): ConvResults;
PROCEDURE FormatReal (str: ARRAY OF CHAR): ConvResults;
```

The function procedure `FormatReal` queries the format of `str` and returns:

`strAllRight` if `str` has a value representing a complete signed real number that is in the range of the real number type corresponding to the module;

`strOutOfRange` if `str` has a value representing a complete signed real number that is not in the range of the real number type corresponding to the module;

`strWrongFormat` if `str` has a value with remaining characters that do not form a complete signed real number;

`strEmpty` if `str` has a value with no remaining characters;

`FormatReal` ignores any leading space characters in `str`.

## ValueReal — *Query real value*

```
PROCEDURE ValueReal (str: ARRAY OF CHAR): REAL;
PROCEDURE ValueReal (str: ARRAY OF CHAR): LONGREAL;
```

If `str` has a value representing a real number, the function procedure `ValueReal(str)` returns the `REAL` (or `LONGREAL`) value that corresponds most closely to that number. Otherwise, an exception is raised.

## LengthFloatReal — *Query float length*

```
PROCEDURE LengthFloatReal (real: REAL;
                           sigFigs: CARDINAL): CARDINAL;
PROCEDURE LengthFloatReal (real: LONGREAL;
                           sigFigs: CARDINAL): CARDINAL;
```

The function procedure `LengthFloatReal` returns the number of characters in the floating-point string representation of the value of `real` when using `sigFigs` significant figures.

**NOTE:**

This value corresponds to the capacity of an array `str` which is of the minimum capacity needed to avoid truncation of the result in the call `RealStr.RealToFloat(real,sigFigs,str)` (in the case of `RealConv.LengthFloatReal`) or the call `LongStr.RealToFloat(real,sigFigs,str)` (in the case of `LongConv.LengthFloatReal`).

## LengthEngReal — *Query engineering length*

```
PROCEDURE LengthEngReal (real: REAL;
                         sigFigs: CARDINAL): CARDINAL;
PROCEDURE LengthEngReal (real: LONGREAL;
                         sigFigs: CARDINAL): CARDINAL;
```

The function procedure `LengthEngReal` returns the number of characters in the floating-point engineering string representation of the value of `real` when using `sigFigs` significant figures.

**NOTE:**

This value corresponds to the capacity of an array `str` which is of the minimum capacity needed to avoid truncation of the result in the call `RealStr.RealToEng(real,sigFigs,str)` (in the case of `RealConv.LengthEngReal`) or the call `LongStr.RealToEng(real,sigFigs,str)` (in the case of `LongConv.LengthEngReal`).

## LengthFixedReal — *Query fixed length*

```
PROCEDURE LengthFixedReal (real: REAL;
                           place: INTEGER): CARDINAL;
PROCEDURE LengthFixedReal (real: LONGREAL;
                           place: INTEGER): CARDINAL;
```

The function procedure `LengthFixedReal` returns the number of characters in the fixed-point string representation of the value of `real` when rounded to the place relative to the decimal point given by the value of `place`.

**NOTE:**

This value corresponds to the capacity of an array `str` which is of the minimum capacity needed to avoid truncation of the result in the call `RealStr.RealToFixed(real,place,str)` (in the case of `RealConv.LengthFixedReal`) or the call `LongStr.RealToFixed(real,place,str)` (in the case of `LongConv.LengthFixedReal`).

### IsRConvException — *Query exceptional state*

```
PROCEDURE IsRConvException (): BOOLEAN;
PROCEDURE IsRConvException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of the `RealConv` exception, the function procedure `RealConv.IsRConvException` returns `TRUE`; otherwise it returns `FALSE`.

If the calling coroutine is in the state of exceptional execution because of the raising of the `LongConv` exception, the function procedure `LongConv.IsRConvException` returns `TRUE`; otherwise it returns `FALSE`.

# Chapter 6

# Miscellaneous

## 6.1 Module CharClass

The full set of values of the character type (the elementary type denoted by the pervasive identifier `CHAR`) is implementation-defined. The module `CharClass` allows a program to determine the classification of a given value of the character type of an implementation in a way that does not rely on there being a known literal representation of all members of the character set.

The module `CharClass` provides predicates to test if a given value of the character type in an implementation is classified as a numeric, a letter, an upper-case letter, a lower-case letter, a value representing a control function, or as white space.

**IsNumeric** *Test for numeric character*

```
PROCEDURE IsNumeric (ch: CHAR): BOOLEAN;
```

The function procedure `IsNumeric` returns `TRUE` if `ch` is a member of an implementation-defined set of numeric characters that should include the decimal digits, and `FALSE` otherwise.

**IsLetter** *Test for letter*

```
PROCEDURE IsLetter (ch: CHAR): BOOLEAN;
```

The function procedure `IsLetter` returns `TRUE` if `ch` is a member of an

implementation-defined set of letters that should include the required letters, and FALSE otherwise.

**IsUpper** *Test for upper case letter*

```
PROCEDURE IsUpper (ch: CHAR): BOOLEAN;
```

The function procedure `IsUpper` returns `TRUE` if `ch` is a member of an implementation-defined set of upper case letters that should include the required upper case letters, and FALSE otherwise.

**IsLower** *Test for lower case letter*

```
PROCEDURE IsLower (ch: CHAR): BOOLEAN;
```

The function procedure `IsLower` returns `TRUE` if `ch` is a member of an implementation-defined set of lower case letters that should include the required lower case letters, and FALSE otherwise.

**IsControl** *Test for control character*

```
PROCEDURE IsControl (ch: CHAR): BOOLEAN;
```

The function procedure `IsControl` returns `TRUE` if `ch` is a member of an implementation-defined set of characters that represent control functions, and FALSE otherwise.

**IsWhiteSpace** *Test for white space character*

```
PROCEDURE IsWhiteSpace (ch: CHAR): BOOLEAN;
```

The function procedure `IsWhiteSpace` returns `TRUE` if `ch` is either a space character or a member of an implementation-defined set of characters that represent format effectors, and FALSE otherwise.

## 6.2 Modules LowReal and LowLong

Two modules are provided to give access to the underlying properties of the types REAL and LONGREAL. The two modules share common concepts, functions and values, and hence both modules are considered together.

The module LowReal gives access to the underlying properties of the type REAL, while LowLong gives access to the same properties for the type LONGREAL.

For implementations that conform to *ISO/IEC 10967-1:199x Information technology - Language independent arithmetic - Part1: Integer and floating point arithmetic*, a more precise specification is given in that International Standard.

If the implementation of the corresponding real number type conforms to *ISO/IEC 10967-1:199x (LIA-1)*, procedure functions of a similar name correspond to the operations required by that International Standard.

### Constants and Types

```
CONST
  radix = <implementation-defined whole number value>;
  places = <implementation-defined whole number value>;
  expoMin = <implementation-defined whole number value>;
  expoMax = <implementation-defined whole number value>;
  large = <implementation-defined real number value>;
  small = <implementation-defined real number value>;
  IEC559 = <implementation-defined BOOLEAN value>;
  LIA1 = <implementation-defined BOOLEAN value>;
  rounds = <implementation-defined BOOLEAN value>;
  gUnderflow = <implementation-defined BOOLEAN value>;
  exception = <implementation-defined BOOLEAN value>;
  extend = <implementation-defined BOOLEAN value>;
  nModes = <implementation-defined whole number value>;
TYPE
  Modes = PACKEDSET OF [0 .. nModes-1];
```

The values denoted by the constant identifiers exported from LowReal and LowLong are the implementation-defined values specified below.

If an implementation provides facilities for dynamically changing the properties of the real number types, the constant values refer to the default properties.

The value of `places`, and the other facilities in these modules, refer only to the representation used to store values.

**NOTE:** Some implementations may choose to compute expressions to greater precision than that used to store values.

If the implementation of the corresponding real number type conforms to *ISO/IEC 10967-1:199x (LIA-1)*, the following correspondences hold:

**NOTE:** The value of the parameter `fmax`, required by *ISO/IEC 10967-1:199x*, is given by the predefined function `MAX` when applied to the corresponding real number type.

`radix` The whole number value of the radix used to represent the corresponding real number values.

`places` The whole number value of the number of `radix` places used to store values of the corresponding real number type.

`expoMin` The whole number value of the exponent minimum.

`expoMax` The whole number value of the exponent maximum. **NOTE:** An implementation may choose values such that `expoMin = expoMax` (which will presumably be the case for a fixed point representation).

`large` The largest value of the corresponding real number type. **NOTE:** On some implementations this may be a machine representation of infinity.

`small` The smallest positive value of the corresponding real number type, represented to maximal precision. **NOTE:** If an implementation has stored values strictly between 0.0 and `small`, then presumably the implementation supports gradual underflow.

`IEC559` A Boolean value that is true if and only if the implementation of the corresponding real number type conforms to *IEC 559:1989 (IEEE 754:1987)* in all regards.

**NOTES:**

- If `IEC559` is true, the value of radix is `2`.
- If `LowReal.IEC559` is true, the 32-bit format of *IEC 559:1989* is used for the type REAL.
- If `LowLong.IEC559` is true, the 64-bit format of *IEC 559:1989* is used for the type LONGREAL.

`LIA1` A Boolean value that is true if and only if the implementation of the corresponding real number type conforms to *ISO/IEC 10967-1:199x (LIA-1)* in all regards: parameters, arithmetic, exceptions, and notification.

**NOTE:** For implementations not conforming to *ISO/IEC 10967-1:199x*, the corresponding properties are implementation-defined | see 6.8.2.2.

`rounds` A Boolean value that is true if and only if each operation produces a result that is one of the values of the corresponding real number type nearest to the mathematical result.

**NOTE:** If `rounds` is true, and the mathematical result lies mid-way between two values of the corresponding real number type, then the selection from the two possible values is implementation-dependent.

`gUnderflow` A Boolean value that is true if and only if there are values of the corresponding real number type between 0.0 and `small`.

`exception` A Boolean value that is true if and only if every operation that attempts to produce a real value out of range raises an exception.

`extend` A Boolean value that is true if and only if expressions of the corresponding real number type are computed to higher precision than the stored values.

**NOTE:** If `extend` is true, then values greater than `large` can be computed in expressions, but cannot be stored in variables.

`nModes` The whole number value giving the number of bit positions needed for the status flags for mode control.

**exponent** *Exponent value*

```
PROCEDURE exponent (x: REAL): INTEGER;
PROCEDURE exponent (x: LONGREAL): INTEGER;
```

The function procedure `exponent` returns the exponent value of `x` that lies between `expoMin` and `expoMax`. An exception occurs and may be raised if `x` is equal to 0.0.

**fraction** *Significand part*

```
PROCEDURE fraction (x: REAL): REAL;
PROCEDURE fraction (x: LONGREAL): LONGREAL;
```

The function procedure `fraction` returns the significand (or significant) part of `x`. Hence the following relationship shall hold:

```
x = scale(fraction(x), exponent(x))
```

**sign** *Signum*

```
PROCEDURE sign (x: REAL): REAL;
PROCEDURE sign (x: LONGREAL): LONGREAL;
```

The function procedure `sign` returns 1.0 if `x` is greater than 0.0, -1.0 if `x` is less than 0.0, or either 1.0 or -1.0 if `x` is equal to 0.0.

**NOTE:** The uncertainty about the handling of 0.0 is to allow for systems that distinguish between +0.0 and -0.0 (such as IEEE 754 systems).

**succ** *Next greater value*

```
PROCEDURE succ (x: REAL): REAL;
PROCEDURE succ (x: LONGREAL): LONGREAL;
```

The function procedure `succ` returns the next value of the corresponding real number type greater than `x`, if such a value exists; otherwise an exception occurs and may be raised.

**ulp** *Unit in the last place*

```
PROCEDURE ulp (x: REAL): REAL;
PROCEDURE ulp (x: LONGREAL): LONGREAL;
```

The function procedure `ulp` returns the value of the corresponding real number type equal to a unit in the last place of `x`, if such a value exists; otherwise an exception occurs and may be raised.

**NOTE:** Thus, when the value exists, either `ulp(x) = succ(x)-x` or `ulp(x) = x-pred(x)` or both is true.

| **pred** | *Previous less value* |
|---|---|

```
PROCEDURE pred (x: REAL): REAL;
PROCEDURE pred (x: LONGREAL): LONGREAL;
```

The function procedure `pred` returns the next value of the corresponding real number type less than `x`, if such a value exists; otherwise an exception occurs and may be raised.

| **intpart** | *Integral part* |
|---|---|

```
PROCEDURE intpart (x: REAL): REAL;
PROCEDURE intpart (x: LONGREAL): LONGREAL;
```

The function procedure `intpart` returns the integral part of `x`. For negative values, this shall be `-intpart(abs(x))`.

| **fractpart** | *Fractional part* |
|---|---|

```
PROCEDURE fractpart (x: REAL): REAL;
PROCEDURE fractpart (x: LONGREAL): LONGREAL;
```

The function procedure `fractpart` returns the fractional part of `x`. This satisfies the relationship `fractpart(x)+intpart(x)=x`.

| **scale** | *Scale* |
|---|---|

```
PROCEDURE scale (x: REAL; n: INTEGER): REAL;
PROCEDURE scale (x: LONGREAL; n: INTEGER): LONGREAL;
```

The function procedure `scale` returns the value `x * radix ** n` if such a value exists; otherwise an exception occurs and may be raised.

**trunc** *Truncate*

```
PROCEDURE trunc (x: REAL; n: INTEGER): REAL;
PROCEDURE trunc (x: LONGREAL; n: INTEGER): LONGREAL;
```

The function procedure `trunc` returns the value of the most significant `n` places of `x`. An exception occurs and may be raised if `n` is less than or equal to zero.

**round** *Round*

```
PROCEDURE round (x: REAL; n: INTEGER): REAL;
PROCEDURE round (x: LONGREAL; n: INTEGER): LONGREAL;
```

The function procedure `round` returns the value of `x` rounded to the most significant `n` places. An exception occurs and may be raised if such a value does not exist, or if `n` is less than or equal to zero.

**synthesize** *Construct value*

```
PROCEDURE synthesize (expart : INTEGER;
                      frapart: REAL): REAL;
PROCEDURE synthesize (expart : INTEGER;
                      frapart: LONGREAL): LONGREAL;
```

The function procedure `synthesize` returns a value of the corresponding real number type constructed from the values of `expart` and `frapart`. This value shall satisfy the relationship:
`synthesize(exponent(x),fraction(x))=x`

**setMode** *Set status flags*

```
PROCEDURE setMode (m: Modes);
PROCEDURE setMode (m: Modes);
```

The procedure setMode sets status flags from the value of `m`, appropriate to the underlying implementation of the corresponding real number type. **NOTES:**

- Many implementations of floating point provide options for setting status flags within the system which control details of the handling of the type. Although two procedures are provided, one for each real number type, the effect may be the same. Typical effects that can be obtained by this means are:
  - Ensuring that overflow will raise an exception;
  - Allowing underflow to raise an exception;
  - Controlling the rounding;
  - Allowing special values to be produced (e.g. NaNs in implementations conforming to *IEC 559:1989 (IEEE 754:1987)*);
  - Ensuring that special value access will raise an exception;

  Since these effects are so varied, the values of type `Modes` that may be used are not specified by this International Standard.
- The effect of `setMode` on operations on values of the corresponding real number type in coroutines other than the calling coroutine is not defined. Implementations are not required to preserve the status flags (if any) with the coroutine state.

**currentMode** *Current status flags*

```
PROCEDURE currentMode (): Modes;
PROCEDURE currentMode (): Modes;
```

The function procedure `currentMode` returns the current status flags (in the form set by `setMode`), or the default status flags (if `setMode` is not used).

**NOTE:** The returned value is not necessarily the value set by `setMode`, since a call of `setMode` might attempt to set flags that cannot be set by the program.

**IsLowException** *Query exceptional state*

```
PROCEDURE IsLowException (): BOOLEAN;
PROCEDURE IsLowException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of the `LowReal` exception, the function procedure `LowReal.IsLowException` returns `TRUE`; otherwise is returns `FALSE`.

If the calling coroutine is in the state of exceptional execution because of the raising of the `LowLong` exception, the function procedure `LowLong.IsLowException` returns `TRUE`; otherwise is returns `FALSE`.

## 6.3 Module Storage

The module `Storage` provides facilities for dynamically allocating and deallocating storage for variables that are not declared in variable declarations. Variables with storage allocated in this way are designated by dereferenced variable designators. The facilities are often invoked by using the predefined procedures `NEW` and `DISPOSE`. The facilities can also be used to allocate storage to be used as coroutine workspace.

The semantics are described in terms of allocating storage for a variable since the allocator must take account of any address alignment requirements for the storage of variables of the given size. **CLARIFICATIONS**

- *Programming in Modula-2* adopts two approaches to handling situations where it is not possible to allocate sufficient storage; the procedure `Allocate` of Chapter 25 assigns `NIL` to the first parameter, whereas the procedure `ALLOCATE` of Appendix 2 causes the program to terminate. The International Standard requires the procedure `ALLOCATE` to assign the value `NIL` to its first parameter in this situation.

- Although the first parameter of the procedure `DEALLOCATE` given in Appendix 2 of *Programming in Modula-2* is a variable parameter, it is not stated whether `DEALLOCATE` assigns a value to its parameter. The International Standard requires the procedure `DEALLOCATE` to assign the value `NIL` to its parameter.

## StorageExceptions — *Storage exceptions identities*

```
TYPE
  StorageExceptions = (
    nilDeallocation,
      (* first argument to DEALLOCATE is NIL *)
    pointerToUnallocatedStorage,
      (* storage to deallocate not allocated by ALLOCATE *)
    wrongStorageToUnallocate
      (* amount to deallocate is not amount allocated *)
);
```

The exceptions raised by `Storage` are identified by the values of the enumeration type `StorageExceptions`.

The detection of the exception `wrongStorageToUnallocate` is implementation-defined.

## ALLOCATE — *Allocate storage*

```
PROCEDURE ALLOCATE (VAR addr: SYSTEM.ADDRESS;
                    amount: CARDINAL);
```

The procedure `ALLOCATE` allocates storage for a variable of size `amount`, and assigns the address of this variable to `addr`. The allocated locations will not be in use for the storage of any other variable. If it is not possible to allocate this storage, the value `NIL` is assigned to `addr`.

**NOTES:**

1. If an address passed back by a call of `ALLOCATE` is assigned to a pointer variable `vp` that is to be dereferenced to designate a variable of type `T`, the value for the second parameter to `ALLOCATE` may be obtained from evaluation of the expression `SIZE(T)`. An equivalent effect may be obtained as `NEW(vp)`.

2. If an address passed back by a call of `ALLOCATE` is to be given directly as the workspace address in a call of `COROUTINES.NEWCOROUTINE`, the value for the second parameter to `ALLOCATE` is the size of workspace required.

## DEALLOCATE *Deallocate storage*

```
PROCEDURE DEALLOCATE (VAR addr: SYSTEM.ADDRESS;
                      amount: CARDINAL);
```

The procedure `DEALLOCATE` deallocates `amount` locations for the storage of the variable addressed by `addr` and assigns the value `NIL` to `addr`.

The exception `nilDeallocation` is raised if the given value of `addr` is the nil value. The exception `pointerToUnallocatedStorage` is raised if the given value of `addr` is not the address of a variable whose storage has been allocated by `ALLOCATE`. The exception `wrongStorageToUnallocate` occurs and may be raised if `amount` is not equal to the number of locations allocated for the storage of the variable addressed by `addr`.

**NOTES:**

1. If an address passed to a call of `DEALLOCATE` is the value of a pointer variable `vp` that is dereferenced to designate a variable of type `T`, the value for the second parameter to `DEALLOCATE` may be obtained from evaluation of the expression `SIZE(T)`. An equivalent effect may be obtained as `DISPOSE(vp)`.

2. The variable whose storage is deallocated no longer exists and hence an exception occurs, which may be raised, if there is a subsequent attempt to access the variable through a dereferenced designator.

3. This International Standard gives no meaning for a program that deallocates dynamic storage given as workspace in a call of `COROUTINES.NEWCOROUTINE` since the use made of coroutine workspace is implementation-dependent.

4. It need not be the case that storage locations deallocated by a call of `DEALLOCATE` are available for re-use by a subsequent call of `ALLOCATE`.

## IsStorageException *Query exceptional state*

```
PROCEDURE IsStorageException (): BOOLEAN;
```

If the calling coroutine is in the state of exceptional execution because of the raising of a `Storage` exception, the function procedure `IsStorageException` returns `TRUE`; otherwise it returns `FALSE`.

| **StorageException** | *Query exception id* |
|---|---|

```
PROCEDURE StorageException (): StorageExceptions;
```

If the calling coroutine is in the state of exceptional execution because of the raising of a `Storage` exception, the function procedure `StorageException` returns the value that identifies the raised exception; otherwise the language exception `exException` is raised.

## 6.4 Module SysClock

The module `SysClock` provides facilities for accessing a system clock that records the date and time of day. **NOTES:**

- No provision is made for leap seconds.
- 'UTC' is 'Universal Coordinated Time'. This is the correct international designation for what was once called 'GMT' (Greenwich Mean Time).
- The field `summerTimeFlag` is present for information only. UTC can always be obtained by subtracting the `UTCDiff` value from the time data, regardless of the value of the `summerTimeFlag`. However, its presence does allow a program to know whether or not the date and time data represents standard time for that location, or 'summer time'. A program could therefore be written to change the system clock to summer time automatically on a certain date, provided it had not already been changed.

### 6.4.1 The Constants and Types of SysClock

```
CONST
  maxSecondParts = <implementation-defined integral value>;

TYPE
  Month = [1 .. 12];
  Day = [1 .. 31];
  Hour = [0 .. 23];
  Min = [0 .. 59];
  Sec = [0 .. 59];
```

```
  Fraction = [0 .. maxSecondParts];
  UTCDiff = [-780 .. 720];
  DateTime =
    RECORD
      year:   CARDINAL;
      month:  Month;
      day:    Day;
      hour:   Hour;
      minute: Min;
      second: Sec;
      fractions: Fraction; (* parts of a second *)
      zone:   UTCDiff;
        (* Time zone differential factor which is the number
           of minutes to add to local time to obtain UTC. *)
      summerTimeFlag: BOOLEAN;
        (* Interpretation of flag depends on local usage. *)
    END;
```

## CanGetClock — *Query system clock read permission*

```
PROCEDURE CanGetClock (): BOOLEAN;
```

The function procedure `CanGetClock` returns an implementation-defined `BOOLEAN` value. If the value `TRUE` is returned, there is a system clock which the program is permitted to read.

## CanSetClock — *Query system clock write permission*

```
PROCEDURE CanSetClock (): BOOLEAN;
```

The function procedure `CanSetClock()` returns an implementation-defined `BOOLEAN` value. If the value `TRUE` is returned, there is a system clock which the program is permitted to set.

## IsValidDateTime — *Verify date and time*

```
PROCEDURE IsValidDateTime (userData: DateTime): BOOLEAN;
```

The function procedure `IsValidDateTime` returns `TRUE` if the value of `userData` represents a valid date and time, and is `FALSE` otherwise. **NOTE:** Only the date components of `userData` need to be validated since all combinations of values of the time components are known to be valid.

## GetClock — *Determine current date and time*

```
PROCEDURE GetClock (VAR userData: DateTime);
```

The function procedure `GetClock` assigns values for each field of the variable `userData` for which information is available. Each of the remaining fields of `userData` are set to zero, where this is a valid value, and otherwise are set to the lower bound of the range of allowed values.

## SetClock — *Set current date and time*

```
PROCEDURE SetClock (userData: DateTime);
```

The function procedure `SetClock` sets the system clock to the date and time specified by `userData`, provided that the program may set the system clock, and that the value of `userData` represents a valid date and time. If the program cannot set the system clock, a call of `SetClock` have no effect.

**NOTE:** The effect of a call of `SetClock` is implementation-dependent if it is permitted to set the system clock, but an invalid date and time is given.

# Index

This page had been intentionally left blank.


**Excelsior, LLC**
6 Lavrenteva Ave.
Novosibirsk 630090 Russia
Tel: +7 (383) 330-5508
Fax: +1 (509) 271-5205
Email: info@excelsior-usa.com
Web: http://www.excelsior-usa.com