Copyright © 1999-2011 Excelsior LLC. All rights reserved.

Information in this document is subject to change without notice and does not represent a commitment on the part of Excelsior, LLC.

Excelsior’s software and documentation have been tested and reviewed. Nevertheless, Excelsior makes no warranty or representation, either express or implied, with respect to the software and documentation included with Excelsior product. In no event will Excelsior be liable for direct, indirect, special, incidental or consequential damages resulting from any defect in the software or documentation included with this product. In particular, Excelsior shall have no liability for any programs or data used with this product, including the cost of recovering programs or data.

XDS is a trademark of Excelsior LLC.

All trademarks and copyrights mentioned in this documentation are the property of their respective holders.
5 Oberon-2 Oakwood libraries 41
Chapter 1

Overview

The following sets of libraries are provided:

- ISO standard Modula-2 libraries (Chapter 2)
- A subset of libraries defined in [PIM] (Chapter 4)
- Utility libraries (Chapter 3)
- A subset of Oberon-2 Oakwood standard libraries (Chapter 5)

Note: All these libraries can be used with both Modula-2 and Oberon-2 compilers. Some libraries are written in Oberon-2, others in Modula-2. If a program consists of both Oberon-2 and Modula-2 modules, two memory managers are in use. In rare cases it can lead to problems with memory allocation or deallocation. A special care was taken to eliminate these problems.
Chapter 2

ISO Modula-2 standard libraries

It is our aim to provide the full set of ISO Modula-2 libraries. However, some modules are unimplemented on a particular hardware/software platform. System libraries are described in the XDS Modula-2 Chapter of the XDS User's Guide. This chapter does not contain a complete reference. A brief description and some samples are provided for each group of modules. For more information, refer to the ISO Modula-2 Library Reference.

2.1 Input/output library

The IO library allows one to read and write the data streams over one or more channels. Channels are connected to the source of input data, or to destination of output data, known as devices. A set of devices can be extended.

A group of modules is provided to operate on the default input and output channel (2.1.1). Another group of modules provide facilities to operate on channels specified explicitly by a parameter (2.1.2). The device modules provide facilities to get a channel connected to a source (2.1.3). The primitive device-independent operations are provided by the module IOChan; the module IOLink allows specialized device module to be implemented (See 2.1.4).

2.1.1 Reading and writing via default channels

The following modules provide procedures that operate via default input and output channels and do not take a parameter which identifies a channel:


CHAPTER 2. ISO MODULA-2 STANDARD LIBRARIES

IOConsts  Types and constants for IO modules
SLongIO   LONGREAL numbers IO operations
SRawIO    Raw IO operations (no conversion or interpretation)
SRealIO   REAL numbers IO operations
SResultIO Read results for the default input channel
STextIO   Character and string types IO operations
SWholeIO  Whole numbers IO operations

The module STextIO resembles the well-known InOut library. The Hello, World program is implemented in the following example:

MODULE Hello;

IMPORT STextIO;

BEGIN

STextIO.WriteString('Hello, World!');
STextIO.WriteLn;

END Hello.

The modules SWholeIO, SRealIO, SLongIO provides facilities for the input and output of whole and real numbers in a decimal form using text operations on a channel.

PROCEDURE Print(stage: CARDINAL; val: REAL);
BEGIN

STextIO.WriteString('On stage ');
SWholeIO.WriteCard(stage,0);
STextIO.WriteString(' the value is equal to ');
SRealIO.WriteReal(val,15);
STextIO.WriteLn;

END Print;

The module SIOResult allows one to determine whether the last input operation from a default input channel succeed. Text operations produce or consume data streams as a sequence of characters and line marks. The text input procedures (such as ReadString never remove a line mark from the input stream. The procedure SkipLine should be used to pass a line mark.

The Copy procedure reads strings from the input channel and copies them to the output channel.
2.1. INPUT/OUTPUT LIBRARY

PROCEDURE Copy;
   VAR s: ARRAY [0..63] OF CHAR;
BEGIN
   LOOP
      STextIO.ReadString(s);
      CASE SIOResult.ReadResult() OF
         |SIOResult.allRight:
            STextIO.WriteString(s);
         |SIOResult.endOfLine:
            STextIO.SkipLine;
            STextIO.WriteLn;
         |SIOResult.endOfInput:
            EXIT
      END;
   END;
END Copy;

No procedure is provided to get the result of a ‘write’ operation. Device errors are reported by raising an exception (See module IOChan).

2.1.2 Reading and writing data

For all modules in this group a channel is specified by an actual parameter of the type IOChan.ChanId.

- IOResult: Read results for specified channels
- LongIO: LONGREAL numbers IO operations
- RawIO: Raw IO operations (no conversion or interpretation)
- RealIO: REAL numbers IO operations
- TextIO: Character and string types IO operations
- WholeIO: Whole numbers IO operations

The following procedure copies an input channel to an output channel byte by byte:

PROCEDURE CopyChars(in, out: IOChan.ChanId);
   VAR ch: CHAR;
BEGIN
   LOOP
      TextIO.ReadChar(in, ch);
   END;
END CopyChars;
2.1.3 Device modules

The device modules allows to get a channel connected to a stream, a file, program arguments and to default channels.

- **ChanConsts**: Common types and values for channel open requests and results
- **ProgramArgs**: Access to program arguments
- **RndFile**: Random access files
- **SeqFile**: Rewindable sequential files
- **StdChans**: Access to standard and default channels
- **StreamFile**: Independent sequential data streams

In the following example a channel connected to a rewindable file is opened:

```
MODULE Example;

IMPORT SeqFile, STextIO, TextIO;

CONST flags = SeqFile.text + SeqFile.old;

VAR
  cid: SeqFile.ChanId;
  res: SeqFile.OpenResults;
  i : CARDINAL;

BEGIN
  SeqFile.OpenWrite(cid,"example.dat",flags,res);
```
IF res = SeqFile.opened THEN
  FOR i:=0 TO 9 DO
    TextIO.WriteString(cid,"Hello");
    TextIO.WriteLine(cid);
  END;
  SeqFile.Close(cid);
ELSE
  STextIO.WriteString("Open error");
  STextIO.WriteLine;
END;
END Example.

The module StdChans allows one to get channels already open to sources and destinations of standard input, standard output and standard error output. Default channels initially corresponds to the standard channels, but their values may be changed to redirect input or output.

PROCEDURE RedirectOutput(cid: StdChans.ChanId);
BEGIN
  (* writing to the current default channel: *)
  STextIO.WriteString("Redirecting output...");
  STextIO.WriteLine;
  (* redirecting output: *)
  StdChans.SetOutChan(cid);
END RedirectOutput;

After the call of RedirectOutput(cid) all subsequent output via modules STextIO, SWholeIO, etc will be written to the channel cid. To restore output call

  StdChans.SetOutChan(StdChans.StdOutChan());

The module ProgramArgs provides a channel to access program’s arguments. The following program prints all its arguments.

MODULE Args;

IMPORT ProgramArgs, TextIO, STextIO;

VAR
str: ARRAY [0..255] OF CHAR;
cid: ProgramArgs.ChanId;

BEGIN
  cid:=ProgramArgs.ArgChan();
  WHILE ProgramArgs.IsArgPresent() DO
    TextIO.ReadToken(cid,str);
    (* Note: read result test is omitted *)
    STextIO.WriteString(str); STextIO.WriteLn;
  END;
END Args.

2.1.4 Low-level IO modules

Two low-level modules are described in this section. The module IOChan defines the type ChanId that is used to identify channels and provides a set of procedures forming the channel’s interface in a device-independent manner.

The module IOLink provides facilities that allow one to define new device modules. Let us implement an encryption channel, i.e. a channel that encrypts all information that is written to it. To make the encryption device-independent we need a channel for input/output operations.

In the following example a sketch of the encryption device module is shown.

DEFINITION MODULE EncryptChan;

IMPORT IOChan, ChanConsts;

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

PROCEDURE Connect(VAR cid: ChanId;
                   io: ChanId;
                   VAR res: OpenResults);
  (* Attempts to open an encryption channel. All I/O operations will be made through "io" channel. *)
PROCEDURE Close(VAR cid: ChanId);
2.1. INPUT/OUTPUT LIBRARY

(* Closes the channel. *)

END EncryptChan.

Values of the type DeviceId are used to identify device modules. By calling the procedure DeviceTablePtrValue, a device module can obtain a pointer to a device table for the channel. Each channel has its own copy of a device table. A device table contains a field in which the device module can store private data. In our example, the io channel will be stored in this field. The device table also serves as a method table (or virtual function table) in object-oriented languages. It contains the procedure variables for each device procedure. All fields are initialized by the call of MakeChan procedure. A device module has to assign its own device procedures to the fields of a device table. See the Connect procedure below.

IMPLEMENTATION MODULE EncryptChan;

IMPORT IOChan, IOLink, ChanConsts, SYSTEM;

(* "did" is used to identify the channel’s kind: *)
VAR did: IOLink.DeviceId;

PROCEDURE EncryptChar(from: SYSTEM.ADDRESS;
i: CARDINAL;
VAR ch: CHAR);
BEGIN
   ch:='a'; (* very simple encryption :-) *)
END EncryptChar;

PROCEDURE TextWrite(x: IOLink.DeviceTablePtr;
         from: SYSTEM.ADDRESS;
         len: CARDINAL);
VAR i: CARDINAL;
   ch: CHAR;
   cid: IOChan.ChanId;
BEGIN
   (* get the channel id *)
   cid:=SYSTEM.CAST(IOChan.ChanId,x^.cd);
   FOR i:=0 TO len-1 DO
      (* encrypt i-th character *)
      EncryptChar(from,i,ch);
   END TextWrite;
(* write an encrypted character *)
IOChan.TextWrite(cid,SYSTEM.ADR(ch),1);
END;
END TextWrite;

PROCEDURE Connect(VAR cid: ChanId;
                  io: ChanId;
      VAR res: OpenResults);
VAR x: IOLink.DeviceTablePtr;
BEGIN
IOLink.MakeChan(did,cid);
IF cid = IOChan.InvalidChan() THEN
  res:=ChanConsts.outOfChans
ELSE
  (* get a pointer to the device table *)
  x:=IOLink.DeviceTablePtrValue(cid,did,
                                IOChan.notAvailable,"");
  (* store the channel id *)
  x^.cd:=SYSTEM.CAST(SYSTEM.ADDRESS,io);
  x^.doTextWrite:=TextWrite;
  (* ... *)
END;
END Connect;

PROCEDURE Close(VAR cid: ChanId);
BEGIN
  IOLink.UnMakeChan(did,cid);
END Close;

BEGIN
  IOLink.AllocateDeviceId(did);
END EncryptChan.

The module EncryptChan can be used as any standard device module.

2.2 String conversions

The string conversion library admits the conversion of the values of numeric data
types to and from the character string representation. It contains the following
modules:

- **ConvTypes**: Common types used in the string conversion modules
- **LongConv**: Low-level LONGREAL/string conversions
- **LongStr**: LONGREAL/string conversions
- **RealConv**: Low-level REAL/string conversions
- **RealStr**: REAL/string conversions
- **WholeConv**: Low-level whole number/string conversions
- **WholeStr**: Whole number/string conversions

The module **ConvTypes** defines the enumeration type **ConvResults**. It also defines the types **ScanClass** and **ScanState** to use in the low-level conversion modules.

The low-level conversion modules allow to control lexical scanning of character sequences. For example, the **WholeConv** module implements procedures **ScanInt** and **ScanCard** representing the start state for a finite state scanner for signed and unsigned whole numbers. In the following example the procedure **ScanInt** is used to locate a position of the first character in a string which is not a part of an integer.

```pascal
PROCEDURE SkipInt(str: ARRAY OF CHAR;
    VAR pos: CARDINAL);
    VAR len: CARDINAL;
    state: ConvTypes.ConvState;
    class: ConvTypes.ConvClass;
BEGIN
    pos:=0; len:=LENGTH(str);
    state:=WholeConv.ScanInt;
    WHILE pos < len DO
        state(str[pos],class,state);
        IF (class = WholeConv.invalid)
            OR (class = WholeConv.terminator)
        THEN
            RETURN
        END;
        INC(pos);
    END;
END SkipInt;
```
2.3 Mathematical libraries

The following modules constitute a mathematical library:

- **ComplexMath** — COMPLEX mathematical functions
- **LongComplexMath** — LONGCOMPLEX mathematical functions
- **LongMath** — LONGREAL mathematical functions
- **RealMath** — REAL mathematical functions

2.4 Processes and Semaphores

The following modules allows concurrent algorithms to be expressed using processes:

- **Processes** — Provides process creation and manipulation facilities.
- **Semaphores** — Provides mutual exclusion facilities for use by processes.

Example

```modula-2
MODULE Test;

IMPORT Processes, Semaphores, STextIO;

VAR
  sig : Semaphores.SEMAPHORE;
  prs : Processes.ProcessId;
  main: Processes.ProcessId;

PROCEDURE Proc;
BEGIN
  STextIO.WriteString('Proc 1'); STextIO.WriteLn;
  Semaphores.Claim(sig); (* suspend until released *)
  STextIO.WriteString('Proc 2'); STextIO.WriteLn;
  Processes.StopMe;
END Proc;

BEGIN
  STextIO.WriteString('Main 1'); STextIO.WriteLn;
```
2.5. OTHER LIBRARIES

Semaphores.Create(sig,0);
main:=Processes.Me();
Processes.Start(Proc,40000,Processes.UrgencyOf(main)+1,NIL,prs);
STextIO.WriteString('Main 2'); STextIO.WriteLn;
Semaphores.Release(sig);
STextIO.WriteString('Main 3'); STextIO.WriteLn;
Processes.StopMe;
END Test.

2.5 Other libraries

In this section we list the ISO modules that do not belong to any of the above groups:

- **CharClass**: provides predicates to test a value of a character type
- **Storage**: Facilities for allocating and deallocating storage
- **Strings**: Facilities for string manipulation
- **SysClock**: Access to a system clock
Chapter 3

Utility libraries

Starting from XDS v2.0 the general purpose modules used in XDS compilers and utilities, which are portable between all versions of XDS on all platforms, are included in the utility library. If you use the ISO library only, your program may be portable to any ISO compliant Modula-2 compiler. However, there are some essential features which are omitted in the ISO library. The utility library covers some of those omissions.

Note: Some library modules are written in Oberon-2, others in Modula-2. In general, any library can be used from both languages. However, do not forget that Oberon modules use implicit memory deallocation scheme and require garbage collection. Refer to the Multilangauge programming Chapter of the XDS User’s Guide.

The following modules are provided in the utility library (implementation language is pointed out in parentheses):

- FileName (M2) Creating and parsing file names
- FileSys (M2) Common file operations
- FormOut (M2) Generic module for formatting output
- FormStr (M2) Formatting output to strings
- Printf (M2) C style formatting output procedures
- ProgEnv (M2) Access to program environment
- TimeConv (M2) Operations on time and date values
- DStrings (O2) Dynamic strings
- FilePath (O2) File search operations
- RegComp (O2) Regular expressions
3.1 FileName module

The module provides operations for parsing and constructing file names. A file name consists of three parts: the directory, name and extension.

All the procedures that construct a string value (Get, GetDir, GetName, GetExt, Convert, Create), have the common behaviour: if the length of a constructed string value exceeds capacity of a variable parameter, a truncated value is assigned. If the length of a constructed string value is less than the capacity of a variable parameter, a string terminator is appended.

<table>
<thead>
<tr>
<th>Format</th>
<th>File name format record</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td></td>
</tr>
<tr>
<td>Format = RECORD</td>
<td></td>
</tr>
<tr>
<td>ok: BOOLEAN;</td>
<td></td>
</tr>
<tr>
<td>(* directory position and length: *)</td>
<td></td>
</tr>
<tr>
<td>dirPos, dirLen : CARDINAL;</td>
<td></td>
</tr>
<tr>
<td>(* name position and length: *)</td>
<td></td>
</tr>
<tr>
<td>namePos, nameLen: CARDINAL;</td>
<td></td>
</tr>
<tr>
<td>(* extension position and length: *)</td>
<td></td>
</tr>
<tr>
<td>extPos, extLen : CARDINAL;</td>
<td></td>
</tr>
</tbody>
</table>

GetFormat

PROCEDURE GetFormat(str: ARRAY OF CHAR; VAR f: Format);  

The GetFormat procedure writes the position and length of file name parts to a Format type record passed in f. If f.ok = FALSE, values of all other fields are undefined.
3.1. FILENAME MODULE

**Get**

Get file name parts

PROCEDURE GetDir (fname: ARRAY OF CHAR;
    VAR dir: ARRAY OF CHAR);
PROCEDURE GetName (fname: ARRAY OF CHAR;
    VAR name: ARRAY OF CHAR);
PROCEDURE GetExt (fname: ARRAY OF CHAR;
    VAR ext: ARRAY OF CHAR);
PROCEDURE Get (fname: ARRAY OF CHAR;
    VAR dir, name, ext: ARRAY OF CHAR);

These procedures return corresponding file name parts.

**Convert**

Convert String to File Name

PROCEDURE Convert (str: ARRAY OF CHAR;
    VAR fname: ARRAY OF CHAR);

Converts a string to a file name according to the conventions of the underlying file system.

**ConvertExt**

Convert File Name Extension

PROCEDURE ConvertExt (VAR ext: ARRAY OF CHAR);

Converts an extension according to the conventions of the underlying file system.

**Length**

Calculate File Name Length

PROCEDURE Length (dir, name, ext: CARDINAL): CARDINAL;

Using the lengths of the directory, name and extension returns an estimated file name length which is greater than or equal to the length of the name generated by the Create procedure call.
CHAPTER 3. UTILITY LIBRARIES

Create

PROCEDURE Create(dir, name, ext: ARRAY OF CHAR;
    VAR fname: ARRAY OF CHAR);

Creates a file name from its parts.

3.1.1 Example

The following procedure can be used to change file name extension:

PROCEDURE ChangeExt(VAR fname: ARRAY OF CHAR;
    newext: ARRAY OF CHAR);

CONST Len = 64;
VAR
dir, name: ARRAY [0..Len-1] OF CHAR;
f: FileName.Format;
len: CARDINAL;
BEGIN
    FileName.GetFormat(fname,f);
    IF NOT f.ok THEN Error("wrong format")
    ELSIF (f.dirLen > Len) OR (f.nameLen > Len) THEN
        Error("too long part");
    ELSE
        len:=FileName.Length(f.dirLen, f.nameLen, LENGTH(newext));
        IF len-1 > HIGH(fname) THEN
            Error("cannot create file name")
        ELSE
            FileName.Create(dir, name, newext, fname);
        END;
    END;
END ChangeExt;

When programming in Oberon-2 dynamic strings can be used to create strings of a required length:

PROCEDURE ChangeExt(VAR fname: ARRAY OF CHAR;
    newext: ARRAY OF CHAR);

VAR
3.2. FILESYS MODULE

```pascal
dir, name: DStrings.String;
f: FileName.Format;
BEGIN
FileName.GetFormat(fname, f);
IF NOT f.ok THEN Error("wrong format")
ELSE
  NEW(dir, f.dirLen+1);
  NEW(name, f.nameLen+1);
  ...
END;
END ChangeExt;
```

3.2 FileSys module

The module provides file common operations.

<table>
<thead>
<tr>
<th>Exists</th>
<th>Is File Exist</th>
</tr>
</thead>
</table>

PROCEDURE Exists(fname: ARRAY OF CHAR): BOOLEAN;

Returns TRUE, if file `fname` exists.

<table>
<thead>
<tr>
<th>ModifyTime</th>
<th>Return Modify Time</th>
</tr>
</thead>
</table>

PROCEDURE ModifyTime(fname: ARRAY OF CHAR;
  VAR time: LONGCARD;
  VAR exists: BOOLEAN);

Returns a file modification time; `time` is valid only if `exists`=TRUE.

<table>
<thead>
<tr>
<th>Rename</th>
<th>Rename File</th>
</tr>
</thead>
</table>

PROCEDURE Rename(fname, newname: ARRAY OF CHAR;
  VAR done: BOOLEAN);

Renames the file `fname` to `newname`. 
PROCEDURE Remove(fname: ARRAY OF CHAR;
               VAR done: BOOLEAN);

Removes a file.

### 3.3 FormOut module

The FormOut module implements a generic formatted output procedure which outputs its arguments according to the format parameter. The syntax of a format string is similar to one used by the printf family of C functions; some useful extensions are provided as well.

Format = { character | Specifier }.
Specifier = "%" Modifier Width
Modifier = "+" | "-" | "\" | "0" | "$" | "#".
Width = [ unsigned number | "*" ].
Precision = [ unsigned number | "*" ].
Start = [ unsigned number | "*" ].
Base = "c" | "d" | "i" | "u" | "o" | "x" | "X" | "e" | "f" | "g" |
       "s" | "{".

The following pairs of symbols starting from the backslash are recognized in Format string (but *not* in strings printed using "%s" specifier):

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>the line separator</td>
</tr>
<tr>
<td>\r</td>
<td>CR (15C)</td>
</tr>
<tr>
<td>\f</td>
<td>FF (14C)</td>
</tr>
<tr>
<td>\t</td>
<td>TAB (11C)</td>
</tr>
<tr>
<td>\</td>
<td>backslash</td>
</tr>
</tbody>
</table>

Base characters and their meanings are listed in the following table:
### 3.3. FORMOUT MODULE

<table>
<thead>
<tr>
<th>Base</th>
<th>Argument</th>
<th>Output format</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>CHAR</td>
<td>single character</td>
</tr>
<tr>
<td>di</td>
<td>integer</td>
<td>signed decimal integer</td>
</tr>
<tr>
<td>u</td>
<td>integer</td>
<td>unsigned decimal integer</td>
</tr>
<tr>
<td>o</td>
<td>integer</td>
<td>unsigned octal integer</td>
</tr>
<tr>
<td>x</td>
<td>integer</td>
<td>unsigned hexadecimal integer</td>
</tr>
<tr>
<td>E,e</td>
<td>real</td>
<td>floating-point real</td>
</tr>
<tr>
<td>f</td>
<td>real</td>
<td>fixed-point real</td>
</tr>
<tr>
<td>G,g</td>
<td>real</td>
<td>either as &quot;f&quot; or &quot;e&quot;</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>{}</td>
<td>BITSET</td>
<td>bitset, e.g. {1,3..5}</td>
</tr>
</tbody>
</table>

Modifier meanings are as follows:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;+&quot;</td>
<td>always print a sign for numbers</td>
<td>only for negative</td>
</tr>
<tr>
<td>&quot;-&quot;</td>
<td>justify the result to the left</td>
<td>right-justify</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
<td>center the result</td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>print leading zeroes for numbers</td>
<td></td>
</tr>
<tr>
<td>&quot;$&quot;</td>
<td>the same as &quot;0&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;#&quot;</td>
<td>print a base character (&quot;H&quot; or &quot;B&quot;)</td>
<td>no base character</td>
</tr>
</tbody>
</table>

**Width** is used to specify the minimum number of characters in the output value. Shorter values are padded with blanks or zeroes to the left and/or to the right, depending on whether a "-", "|", "0", or "$" modifier is present. Specifying **Width** never causes longer values to be truncated; if it is not given or is less than the number of characters in the output value, all characters will be output. In some cases, **Precision** may be used for truncation.

If **Width** is set to an asterisk ("*"), its actual value will be retrieved from the argument list. It has to be specified before the value being formatted.

**Precision** specifies the number of characters to output or the number of decimal places. Unlike **Width**, it may cause the output value to be truncated or rounded.

If **Precision** is set to an asterisk ("*"), its actual value will be taken from the argument list. It has to be specified before the value being formatted, but after the actual value for **Width**, if the latter is also set to "*".

**Precision** value is interpreted depending upon the **Base**, and specifies:
iduox

The minimum number of digits to output. Shorter values are padded on the left with blanks; longer values are not truncated. Default is 1.

fFeE

The number of positions after the decimal point. The last digit output is rounded. If Precision is 0, decimal point is not printed. Default is 6.

gG

The maximum number of significant digits to output. By default, all significant digits are printed.

c

The number of times the character is to output, default is 1.

s

The maximum number of characters to output. By default, characters are output until 0C is reached.

{}

Has no effect.

Start may be used with the "s" (string) base only and specifies an initial offset in the string being formatted. If Start is set to an asterisk ("*"), the actual value will be taken from the argument list. It has to be specified before the value being formatted, but after the actual values for Width and/or Precision, if either of them is also set to "*".

Notes:

- neither the compiler nor the library checks the correctness of actual arguments against format specifications. However, all specifiers, for which no arguments are passed, are ignored, while C printf-like functions may produce unpredictable results in such case.

- the ISO conversion library (See 2.2) is used to convert numbers to strings, therefore the results may differ from produced by C functions.
### 3.3. FORMOUT MODULE

#### writeProc

**Write Procedure Type**

```pascal
TYPE
    writeProc = PROCEDURE(
        (*handle:*) SYSTEM.ADDRESS,
        (*string:*) ARRAY OF CHAR,
        (*length:*) INTEGER
    );
```

A procedure of this type is passed to the `format` procedure which uses that procedure to perform output.

#### format

**Generic Formatting Procedure**

```pascal
PROCEDURE format(handle : SYSTEM.ADDRESS;
    write : writeProc;
    fmt : ARRAY OF CHAR;
    linesep: CHAR;
    args : SYSTEM.ADDRESS;
    size : CARDINAL);
```

The `format` procedure forms a string and outputs it via the `write` procedure parameter. The `handle` parameter is passed to the procedure `write` and provides a useful method to pass any information between the caller and the `write` procedure (e.g. output channel or something like it). The `(args, size)` pair denotes the address and size of the parameter block. The `linesep` parameter determines the line separator character sequence corresponding to "\n". Several standard values of the parameter are defined in the definition module:

- `default`  default line separator for binary files
- `text`      default line separator for text files
- `crlf`      CR LF character sequence

If the `linesep` is not equal to any of the values above, its value will be used as a line separator.
PROEDURE LineSeparator(nl: ARRAY OF CHAR);

Sets the default line separator for binary files. The correct value for the given platform is set in the module initialization.

PROEDURE TextSeparator(nl: ARRAY OF CHAR);

Sets the default line separator for text files. The correct value for the given platform is set in the module initialization.

### 3.3.1 Examples

The following example shows the implementation of a procedure which produces a format output to an ISO channel.

PROCEDURE ChanWrite(handle: SYSTEM.ADDRESS;
    str: ARRAY OF CHAR;
    len: INTEGER);

VAR chan: IOChan.ChanId; pos: INTEGER;
BEGIN
    chan:=SYSTEM.CAST(IOChan.ChanId,handle);
    pos:=0;
    WHILE len > 0 DO
        IF str[pos] = ASCII.LF THEN IOChan.WriteLn
        ELSE IOChan.TextWrite(chan,SYSTEM.ADR(str[pos]),1)
        END;
        INC(pos); DEC(len);
    END;
END ChanWrite;

PROCEDURE Print(chan: IOChan.ChanId;
    format: ARRAY OF CHAR;
    SEQ args: SYSTEM.BYTE);
3.3. FORMOUT MODULE

BEGIN
    FormOut.format(chan, ChanWrite, format, FormOut.text,
        SYSTEM.ADR(args), SIZE(args));
END Print;

The procedure printf prints to the standard output channel:

PROCEDURE printf(f: ARRAY OF CHAR; SEQ x: SYSTEM.BYTE);
BEGIN
    Print(StdChans.StdOutChan(), f, x);
END printf;

The procedure printf can be used in the conventional for C programmers way,
e.g. the call

    printf("%d! = %d\n", 5, Factorial(5));

will produce the line ¹

    5! = 120

The Printf module implements C-like procedures printf, sprintf, and fprintf.

Examples:

<table>
<thead>
<tr>
<th>Call</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;%5.3s&quot;, &quot;abcdef&quot;)</td>
<td>abc</td>
</tr>
<tr>
<td>printf(&quot;%-5.3s&quot;, &quot;abcdef&quot;)</td>
<td>abc</td>
</tr>
<tr>
<td>printf(&quot;%</td>
<td>5.3s&quot;, &quot;abcdef&quot;)</td>
</tr>
<tr>
<td>printf(&quot;%..3s&quot;, &quot;abcdef&quot;)</td>
<td>def</td>
</tr>
<tr>
<td>printf(&quot;pos=%3d&quot;, 13)</td>
<td>pos=13</td>
</tr>
<tr>
<td>printf(&quot;%$3o&quot;, 13)</td>
<td>015</td>
</tr>
<tr>
<td>printf(&quot;%04X&quot;, 33C)</td>
<td>001B</td>
</tr>
<tr>
<td>printf(&quot;%{}&quot;, 13)</td>
<td>{0,2..3}</td>
</tr>
</tbody>
</table>

¹Provided that the implementation of the procedure Factorial corresponds to its name.
3.4 FormStr module

WARNING:
Language extensions are used in the interface of this module.
All your modules importing this one may be non-portable to other compilers.

A string is an array of characters of an arbitrary length. The procedures print, append and image guarantee the presence of the string terminator (0C) in the resulting string. See the FormOut module overview for the format string syntax.

<table>
<thead>
<tr>
<th>print</th>
<th>Print to string</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE print(VAR str: ARRAY OF CHAR;</td>
<td></td>
</tr>
<tr>
<td>format: ARRAY OF CHAR;</td>
<td></td>
</tr>
<tr>
<td>SEQ args: SYSTEM.BYTE);</td>
<td></td>
</tr>
</tbody>
</table>

Constructs a string specified by the pair (format, args) and places it into str.

<table>
<thead>
<tr>
<th>append</th>
<th>Append to the end of string</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE append( VAR str: ARRAY OF CHAR;</td>
<td></td>
</tr>
<tr>
<td>format: ARRAY OF CHAR;</td>
<td></td>
</tr>
<tr>
<td>SEQ args: SYSTEM.BYTE);</td>
<td></td>
</tr>
</tbody>
</table>

Appends a string specified by the pair (format, args) to the end of the string str.

<table>
<thead>
<tr>
<th>image</th>
<th>Print from the given position</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE image( VAR str: ARRAY OF CHAR;</td>
<td></td>
</tr>
<tr>
<td>VAR pos: LONGINT;</td>
<td></td>
</tr>
<tr>
<td>format: ARRAY OF CHAR;</td>
<td></td>
</tr>
<tr>
<td>SEQ args: SYSTEM.BYTE);</td>
<td></td>
</tr>
</tbody>
</table>

Places a string specified by the pair (format, args) in the string str starting from the position pos. After the procedure call, pos points to the 0C or to the position next to the end of the string.
### 3.5. PRINTF MODULE

**iscan**  
Read integer in Modula-2 format

PROCEDURE iscan(  
    VAR num: INTEGER;  
    str: ARRAY OF CHAR;  
    VAR pos: CARDINAL;  
    VAR done: BOOLEAN);  

Reads an integer value from the string `str` starting from the position `pos`. After the procedure call:

- `done` becomes TRUE, if the attempt was successful;
- `pos` is the index of the first not scanned character;
- `num` is the read value when `done=TRUE`.

The number may be represented in any form permitted in Modula-2. In case of an integer overflow `done=FALSE`.

---

### 3.5 Printf module

The `Printf` module provides C-like formatted output procedures `fprintf`, `printf`, and `sprintf`.

This module is based on the `FormOut` module and is provided for convenience.

#### ChanId  
I/O channel identity

```modula2
type ChanId = IOChan.ChanId;
```

The type `IOChan.ChanId`, which is used in ISO Modula-2 library to identify I/O channels, is reexported.

#### fprintf  
Write formatted data to channel

```modula2
PROCEDURE fprintf(file : ChanId;  
    format : ARRAY OF CHAR;  
    SEQ args : SYSTEM.BYTE);  
```

The procedure `fprintf` formats and outputs a series of characters and values to
the channel identified by file. Each argument in args is converted and written to file according to the corresponding format specification in format. See 3.3 for a description of the format specification and the argument list.

```
printf
```

Print formatted data

PROCEDURE printf(format : ARRAY OF CHAR;
    SEQ args : SYSTEM.BYTE);

The procedure printf formats and outputs a series of characters and values to the current standard output channel. Each argument in args is converted and written to the standard output according to the corresponding format specification in format. See 3.3 for a description of the format specification and the argument list.

```
sprintf
```

Put formatted data to buffer

PROCEDURE sprintf(VAR buf : ARRAY OF CHAR;
    format : ARRAY OF CHAR;
    SEQ args : SYSTEM.BYTE);

The procedure sprintf formats and stores a series of characters and values in the array buf. Each argument in args is converted and put out according to the corresponding format specification in format. See 3.3 for a description of the format specification and the argument list.

### 3.6 ProgEnv module

The ProgEnv module provides access to the program name, arguments, and environment strings.

```
ArgNumber
```

Return the number of arguments

PROCEDURE ArgNumber(): CARDINAL;

Returns the number of arguments (0 if there is no arguments).
3.6. PROGENV MODULE

GetArg

PROCEDURE GetArg(n: CARDINAL; VAR arg: ARRAY OF CHAR);
Copies n-th argument (n >= 0) to arg, or empties it if n >= ArgNumber().

ArgLength

PROCEDURE ArgLength(n: CARDINAL): CARDINAL;
Returns the length of the n-th argument, or 0 if n >= ArgNumber().

ProgramName

PROCEDURE ProgramName(VAR name: ARRAY OF CHAR);
Copies a program name to name.

ProgramNameLength

PROCEDURE ProgramNameLength(): CARDINAL;
Returns the length of the program name.

String

PROCEDURE String(name: ARRAY OF CHAR;
VAR str: ARRAY OF CHAR);
Copies a value of the environment variable name to str (empty string if the variable is undefined).

StringLength

PROCEDURE StringLength(name: ARRAY OF CHAR): CARDINAL;
Returns the length of the environment variable name (0 if the variable is undefined).
3.6.1 Example

The following procedure (in Oberon-2) prints all program arguments:

PROCEDURE ShowArgs;
VAR
  str: POINTER TO ARRAY OF CHAR;
  i, args: LONGINT;
BEGIN
  i:=0;
  args:=ProgEnv.ArgNumber();
  FOR i:=0 TO args-1 DO
    NEW(str, ProgEnv.ArgLength(i)+1);
    ProgEnv.GetArg(i, str^);
    STextIO.WriteString(str^); STextIO.WriteLn;
  END;
END ShowArgs;

3.7 TimeConv module

The Modula-2 module TimeConv provides operations on values of type SysClock.DateTime.

<table>
<thead>
<tr>
<th>DateTime</th>
<th>Date and Time Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE DateTime = SysClock.DateTime;</td>
<td></td>
</tr>
</tbody>
</table>

The SysClock.DateTime type is re-exported for convenience.

<table>
<thead>
<tr>
<th>Compare</th>
<th>Compare two clock values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE Compare(dl, dr: DateTime): INTEGER;</td>
<td></td>
</tr>
</tbody>
</table>

The function procedure Compare returns:

- zero, if dl represents the same moment as dr
• value less than zero, if \( d_l \) is before \( d_r \)

• value greater than zero, if \( d_l \) is after \( d_r \)

**Note:** If either \( d_l \) or \( d_r \) is invalid, zero is returned.

### SubDateDays

**Date difference in days**

PROCEDURE SubDateDays\((d_l, d_r: \text{DateTime})\): CARDINAL;

The function procedure `SubDateDays` returns the number of days passed from \( d_r \) to \( d_l \).

**Note:** If one of the parameters is invalid or if \( d_l \) is before \( d_r \), zero is returned.

### SubDateSecs

**Time difference in seconds**

PROCEDURE SubDateSecs\((d_l, d_r: \text{DateTime})\): CARDINAL;

The function procedure `SubDateSecs` returns the number of seconds passed from \( d_r \) to \( d_l \).

**Note:** If one of the parameters is invalid or if \( d_l \) is before \( d_r \), zero is returned.

### AddDateDays

**Add whole number of days to date**

PROCEDURE AddDateDays\((d: \text{DateTime};
\hspace{1cm} days: \text{CARDINAL};
\hspace{1cm} VAR res: \text{DateTime})\);

The procedure `AddDateDays` adds \( \text{days} \) days to \( d \) and assigns the resulting date to \( res \).

**Note:** If "\( d \)" is invalid, "\( res \)" is assigned the first valid date.
CHAPTER 3. UTILITY LIBRARIES

**AddDateSecs** Add seconds to date

PROCEDURE AddDateSecs(d: DateTime;
                     secs: CARDINAL;
                     VAR res: DateTime);

The procedure AddDateSecs adds secs seconds to date d and assigns the resulting date to res.

**Note:** If "d" is invalid, "res" is assigned the first valid date.

**TheDayNumber** Ordinal day number

PROCEDURE TheDayNumber(d: DateTime): CARDINAL;

The function procedure TheDayNumber returns the ordinal number of the day for the date d.

**Note:** If d is invalid, zero is returned.

**TheFractionNumber** Number of fractions passed from midnight

PROCEDURE TheFractionNumber(d: DateTime): CARDINAL;

The function procedure TheFractionNumber returns the number of fractions passed from time 0:00:00.00 of the day for the date d.

**Note:** If d is invalid, zero is returned.

**WeekDay** Determine day of the week

PROCEDURE WeekDay(d: DateTime): CARDINAL;

The function procedure WeekDay returns day of the week for the date d. 0 represents Sunday, 1 - Monday, etc.

**Note:** If d is invalid, zero is returned.
3.7. TIMECONV MODULE

**millisecs**  
*Milliseconds passed from midnight*

PROCEDURE millisecs(): CARDINAL;

The function procedure `millisecs` returns the number of milliseconds passed from the time 0:00:00.00 of current date as known to the system.

**Note:** This procedure is system-dependent.

**time**  
*System time in seconds*

PROCEDURE time(): CARDINAL;

The function procedure `time` returns the number of seconds passed from the time 0:00:00.00 at first valid date for the system.

**Note:** This procedure is system-dependent.

**unpack**  
*Unpack system time*

PROCEDURE unpack(VAR d: DateTime; secs: CARDINAL);

The procedure `unpack` assigns to `d` the value corresponding to date/time which is `secs` seconds later than the first valid time/date for the system.

This procedure can be used to examine the first system time/date as follows:

```
unpack(firstDateTime,0);
```

**Note:** This procedure is system-dependent.

**pack**  
*Pack system time*

PROCEDURE pack(d: DateTime; VAR secs: CARDINAL);

The procedure `pack` assigns to `secs` the number of seconds passed from a first valid system date/time to `d`. Its effect is opposite to the `pack` procedure.

**Note:** This procedure is system-dependent.
PROCEDURE weekday(t: CARDINAL): CARDINAL;

The function procedure weekday behaves exactly the same as if it contains the following code:

    unpack(tmpDateTime, t); RETURN WeekDay(tmpDateTime);

Note: This procedure is system-dependent.

### 3.8 DStrings module

The module DStrings (written in Oberon-2) defines a dynamic string type and provides some conventional operations.

#### String Dynamic String Type

TYPE String* = POINTER TO ARRAY OF CHAR;

#### Assign Create and Initialize String

PROCEDURE Assign*(s: ARRAY OF CHAR; VAR d: String);

Allocates a new dynamic string and copies string s to it. The resulting string always contains a terminator character (0C).

#### Append Append to Dynamic String

PROCEDURE Append*(s: ARRAY OF CHAR; VAR d: String);

Appends the string s to d. d is extended if necessary. The resulting string always contains the string terminator (0C).
3.9 FilePath module

The FilePath module (written in Oberon-2) provides directory search facilities. In the following procedures path is a list of directories separated by semicolons, e.g.

`.\SYM;C:\LIB\SYM;C:\XDS\LIB\SYM;.` (Windows)
`.~/sym;~/lib/sym;/usr/bin/xds/sym;.` (Unix)

<table>
<thead>
<tr>
<th>IsSimpleName</th>
<th>Is just a File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE IsSimpleName*(name: ARRAY OF CHAR): BOOLEAN;</td>
<td></td>
</tr>
<tr>
<td>Returns TRUE, if name contains a file name only (without directories).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lookup</th>
<th>Look up File</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE Lookup*(path,name: ARRAY OF CHAR; VAR fname: DStrings.String; VAR n: INTEGER);</td>
<td></td>
</tr>
<tr>
<td>Builds a filename using the search path. After a call:</td>
<td></td>
</tr>
<tr>
<td>n = -1 if name is not simple (fname = name)</td>
<td></td>
</tr>
<tr>
<td>n = 0 file is not found (the first directory is used)</td>
<td></td>
</tr>
<tr>
<td>n &gt; 0 file is found in the n-th directory</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UseFirst</th>
<th>Use First Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE UseFirst*(path,name: ARRAY OF CHAR; VAR fname: DStrings.String);</td>
<td></td>
</tr>
<tr>
<td>Builds a filename using the first directory from the search path.</td>
<td></td>
</tr>
</tbody>
</table>
3.10  RegComp module

This module (written in Oberon-2) implements a comparison of a string with regular expression.

3.10.1  Regular expressions

A regular expression is a string which may contain certain special symbols:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>an arbitrary sequence of any characters, possibly empty</td>
</tr>
<tr>
<td></td>
<td>(equivalent to \000−\377 expression)</td>
</tr>
<tr>
<td>?</td>
<td>any single character</td>
</tr>
<tr>
<td></td>
<td>(equivalent to \000−\377 expression)</td>
</tr>
<tr>
<td>[...]</td>
<td>one of the listed characters</td>
</tr>
<tr>
<td>{...}</td>
<td>an arbitrary sequence of the listed characters, possibly empty</td>
</tr>
<tr>
<td>\nnn</td>
<td>the ASCII character with octal code nnn, where n is [0–7]</td>
</tr>
<tr>
<td>&amp;</td>
<td>the logical operation AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>the logical operation NOT</td>
</tr>
<tr>
<td>(...)</td>
<td>the priority of operations</td>
</tr>
<tr>
<td>$digit</td>
<td>subexpression number (see below)</td>
</tr>
</tbody>
</table>

A sequence of the form a–b used within either [ ] or { } brackets denotes all characters from a to b.

$digit may follow *, ?, [ ], [ ], or ( ) subexpression. For a string matching a regular expression, it represents the corresponding substring.

If you need to use any special symbol as an ordinary symbol, you should precede it with a backslash (\), which suppresses interpretation of the following symbol.

3.10.2  Examples of regular expressions

\{0–9A–F\} defines set of hexadecimal numbers
[ a–zA–z_ ] defines a single small or capital letter or an underscore character.

\((\{0–9A–F a–f\})$1 | (\{a–zA–Z_\})$2) $3 matches both hexadecimal numbers and Modula-2 identifiers. After a successful match, a program may access the hexadecimal number by the $1 reference, the identifier by the $2 reference and either of them by the $3 reference.
\\\$\\{\\}\\[\\]\\*\\? represents the string \$\{\}[]*?.

<table>
<thead>
<tr>
<th>Expr</th>
<th>Regular expression</th>
</tr>
</thead>
</table>

**TYPE**

- Expr* = POINTER TO ExprDesc;
- ExprDesc = RECORD END;

**Compile**

*Compile regular expression*

PROCEDURE Compile*(expr: ARRAY OF CHAR;
    VAR reg: Expr;
    VAR res: LONGINT);

Compiles a regular expression to an internal form.

<table>
<thead>
<tr>
<th>Value of res</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>res ≤ 0</td>
<td>error in position ABS(res)</td>
</tr>
<tr>
<td>res &gt; 0</td>
<td>done</td>
</tr>
</tbody>
</table>

**Const**

*Is constant expression*

PROCEDURE Const*(re: Expr): BOOLEAN;

Returns TRUE, if the expression does not contain wildcards.

**Match**

*Compare string with expression*

PROCEDURE Match*(re: Expr;
    s: ARRAY OF CHAR;
    pos: LONGINT): BOOLEAN;

Returns TRUE, if the expression matches the string s starting from the position pos.
CHAPTER 3. UTILITY LIBRARIES

| Len | Length of substring |

PROCEDURE Len*(re: Expr; n: INTEGER): LONGINT;

Returns the length of the substring which corresponds to $n$ in the last call of the Match procedure with the parameter re.

| Pos | Position of substring |

PROCEDURE Pos*(re: Expr; n: INTEGER): LONGINT;

Returns the position of the substring which corresponds to $n$ in the last call of the Match procedure with the parameter re.

| Substitute | Substitute substrings |

PROCEDURE Substitute*(re: Expr;
    s,m: ARRAY OF CHAR;
    VAR d: ARRAY OF CHAR);

The substrings of $s$ which matched $re$ are substituted instead of $digit$ into $m$ and the result string is copied into $d$.

**Note:** The Match($re,s,0$) call should be issued and tested for success prior to a call to Substitute.

**Example**

After the following sequence of calls

```
Compile("{a-z}$1{0-9}$2",re,res);
IF Match(re,"abcdef153",0) THEN
    Substitute(re,"abcdef153","tail: $2 head: $1",dest);
END;
```

the dest string will contain

"tail: 153 head: abcdef"
Chapter 4

PIM standard libraries

The following libraries defined in [PIM] are provided:

- **InOut**: general-purpose IO operations
- **LongInOut**: LONGREAL numbers IO operations
- **MathLib0**: mathematical functions
- **RealInOut**: REAL numbers IO operations
- **Terminal**: computer’s terminal IO operations

The library **LongInOut** (similar to **RealInOut**) is not described in [PIM]. All PIM libraries are implemented on a basis of the ISO library. Since PIM **Storage** library is not compatible with the corresponding ISO library, it is omitted.
Chapter 5

Oberon-2 Oakwood libraries

The Oakwood Guidelines for the Oberon-2 Compiler Developers specifies a set of libraries that should be provided with all Oberon implementations. The current XDS release does not contain all these libraries yet. The following libraries are currently available:

- **In**: input from a standard stream
- **MathR**: mathematical functions for REAL
- **MathL**: mathematical functions for LONGREAL
- **MathC**: mathematical functions for COMPLEX
- **MathLC**: mathematical functions for LONGCOMPLEX
- **Out**: output to a standard stream
- **O2Strings**: simple manipulations for strings

The `Math` library is renamed to `MathR`, because it coincides with the POSIX `math` library interface. The complex types and, hence, `MathC` and `MathLC` modules may be not available for other Oberon implementations.

The `Strings` library is renamed to `O2Strings`, since it is not compatible with the correspondent ISO library.
Bibliography


Index

AddDateDays, 31
AddDateSecs, 32
Append, 34
append, 26
ArgLength, 29
ArgNumber, 28
Assign, 34

ChanConsts, 6
ChanId, 27
CharClass, 13
Compare, 30
Compile, 37

ComplexMath, 12
Const, 37
Convert, 17
ConvertExt, 17

ConvTypes, 11
Create, 18

DateTime, 30
DStrings, 34
  Append, 34
  Assign, 34
  String, 34
Exists, 19
Expr, 37

FileName, 16
  Convert, 17
  ConvertExt, 17
  Create, 18
  Format, 16
  Get*, 17

GetFormat, 16
Length, 17

FilePath, 35
  IsSimpleName, 35
  Lookup, 35
  UseFirst, 35

FileSys, 19
  Exists, 19
  ModifyTime, 19
  Remove, 20
  Rename, 19
Format, 16
format, 23
FormOut, 20
  format, 23
  LineSeparator, 24
  TextSeparator, 24
  writeProc, 23
FormStr, 26
  append, 26
  image, 26
  iscan, 27
  print, 26
fprintf, 27

Get*, 17
GetArg, 29
GetFormat, 16

image, 26
In, 41
InOut, 39
IOChan, 8
IOConsts, 4
INDEX

IOLink, 8
IOResult, 5
iscan, 27
ISO libraries, 3
IsSimpleName, 35
Len, 38
Length, 17
libraries, 1
  ISO, 3
  PIM, 39
  utility, 15
LineSeparator, 24
LongComplexMath, 12
LongConv, 11
LongInOut, 39
LongIO, 5
LongMath, 12
LongStr, 11
Lookup, 35
Match, 37
MathC, 41
MathL, 41
MathLC, 41
MathLib0, 39
MathR, 41
millisecs, 33
ModifyTime, 19
O2Strings, 41
Out, 41
pack, 33
PIM libraries, 39
Pos, 38
print, 26
Printf, 27
  ChanId, 27
  fprintf, 27
  printf, 28
  sprintf, 28
printf, 28
Processes, 12
ProgEnv, 28
  ArgLength, 29
  ArgNumber, 28
  GetArg, 29
  ProgramName, 29
  ProgramNameLength, 29
  String, 29
  StringLength, 29
ProgramArgs, 6
ProgramName, 29
ProgramNameLength, 29
RawIO, 5
RealConv, 11
RealInOut, 39
RealIO, 5
RealMath, 12
RealStr, 11
RegComp, 36
  Compile, 37
  Const, 37
  Expr, 37
  Len, 38
  Match, 37
  Pos, 38
  Substitute, 38
regular expressions, 36
Remove, 20
Rename, 19
RndFile, 6
Semaphores, 12
SeqFile, 6
SLongIO, 4
sprintf, 28
SRawIO, 4
SRealIO, 4
SResultIO, 4
StdChans, 6
STextIO, 4
Storage, 13
StreamFile, 6
String, 29, 34
StringLength, 29
Strings, 13
SubDateDays, 31
SubDateSecs, 31
Substitute, 38
STWholeIO, 4
SysClock, 13

Terminal, 39
TextIO, 5
TextSeparator, 24
TheDayNumber, 32
TheFractionNumber, 32
time, 33
TimeConv, 30
   AddDateDays, 31
   AddDateSecs, 32
   Compare, 30
   DateTime, 30
   millisecs, 33
   pack, 33
   SubDateDays, 31
   SubDateSecs, 31
   TheDayNumber, 32
   TheFractionNumber, 32
time, 33
   unpack, 33
   WeekDay, 32
   weekday, 34
   unpack, 33
   UseFirst, 35
   utility libraries, 15

WeekDay, 32
weekday, 34
WholeConv, 11
WholeIO, 5
writeProc, 23
WholeStr, 11
This page had been intentionally left blank.