XDS Family of Products

Library Guide and Reference

EXCELSIOR

http://www.excelsior-usa.com

Contents

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/sortware 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:

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.

```
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);
    CASE IOResult.ReadResult(in) OF
     |IOResult.allRight:
        TextIO.WriteChar(out,ch);
     |IOResult.endOfLine:
        TextIO.SkipLine(in);
        TextIO.WriteLn(out);
     |IOResult.endOfInput:
        EXIT
    END;
  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.WriteLn(cid);
    END;
    SeqFile.Close(cid);
  ELSE
    STextIO.WriteString("Open error");
    STextIO.WriteLn;
  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.WriteLn;
(* 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);
```

```
(* 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 it 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);
    (* 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.

```
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

```
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;
  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 precedures
`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.

Format *File name format record*

```
TYPE
  Format = RECORD
    ok: BOOLEAN;
    (* directory position and length: *)
    dirPos, dirLen : CARDINAL;
    (* name position and length: *)
    namePos,nameLen: CARDINAL;
    (* extension position and length: *)
    extPos, extLen : CARDINAL;
  END;
```

GetFormat *Get file name format*

```
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`, vaues of all other fields are undefined.

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.

Create *Create File Name*

```
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
```

```
    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;
END ChangeExt;
```

3.2 FileSys module

The module provides file common operations.

Exists *Is File Exist*

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

Returns `TRUE`, if file `fname` exists.

ModifyTime *Return Modify Time*

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

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

Rename *Rename File*

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

Renames the file `fname` to `newname`.

Remove *Remove File*

```
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
            [ "." Precision [ "." Start ] ] Base.
  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):

Input	Output
`\n`	the line separator
`\r`	CR (15C)
`\f`	FF (14C)
`\t`	TAB (11C)
`\\`	backslash

`Base` characters and their meanings are listed in the following table:

Base	**Argument**	**Output format**
`c`	CHAR	single character
`d`, `i`	integer	signed decimal integer
`u`	integer	unsigned decimal integer
`o`	integer	unsigned octal integer
`X`, `x`	integer	unsigned hexadecimal integer
`E`, `e`	real	floating-point real
`f`,	real	fixed-point real
`G`, `g`	real	either as ”f” or ”e”
`s`	string	string
`{}`	BITSET	bitset, e.g. `{1,3..5}`

`Modifier` meanings are as follows:

Modifier	**Meaning**	**Default**
`"+"`	always print a sign for numbers	only for negative
`"-"`	justify the result to the left	right-justify
`"|"`	center the result	
`"0"`	print leading zeroes for numbers	spaces
`"$"`	the same as ”0”	
`"#"`	print a base character (”H” or ”B”) for ”o”, ”x” and ”X” bases only	no base character

`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:

i d u o x X
: The minimum number of digits to output. Shorter values are padded on the left with blanks; longer values are not truncated. Default is 1.

f F e E
: 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.

g G
: 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.

writeProc *Write Procedure Type*

```
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*

```
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.

LineSeparator *Set Line Separator*

```
PROCEDURE 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 initializaion.

TextSeparator *Set Line Separator*

```
PROCEDURE 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 initializaion.

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);
```

```
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:

Call	**Output**
`printf("%5.3s","abcdef")`	`␣␣abc`
`printf("%-5.3s","abcdef")`	`abc␣␣`
`printf("%|5.3s","abcdef")`	`␣abc␣`
`printf("%..3s","abcdef")`	`def`
`printf("pos=%3d",13)`	`pos=␣13`
`printf("%$3o",13)`	`015`
`printf("%04X",33C)`	`001B`
`printf("%{}",13)`	`{0,2..3}`

[1]Provided 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.

print *Print to string*

```
PROCEDURE print(VAR str: ARRAY OF CHAR;
                 format: ARRAY OF CHAR;
               SEQ args: SYSTEM.BYTE);
```

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

append *Append to the end of string*

```
PROCEDURE append( VAR str: ARRAY OF CHAR;
                   format: ARRAY OF CHAR;
                 SEQ args: SYSTEM.BYTE);
```

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

image *Print from the given position*

```
PROCEDURE image( VAR str: ARRAY OF CHAR;
                 VAR pos: LONGINT;
                  format: ARRAY OF CHAR;
                SEQ args: SYSTEM.BYTE);
```

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.

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*

```
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*

```
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).

GetArg *Get argument*

```
PROCEDURE GetArg(n: CARDINAL; VAR arg: ARRAY OF CHAR);
```

Copies `n`-th argument (`n >= 0`) to `arg`, or empties it if `n >= ArgNumber()`.

ArgLength *Return length of argument*

```
PROCEDURE ArgLength(n: CARDINAL): CARDINAL;
```

Returns the length of the `n`-th argument, or 0 if `n>=ArgNumber()`.

ProgramName *Get program name*

```
PROCEDURE ProgramName(VAR name: ARRAY OF CHAR);
```

Copies a program name to `name`.

ProgramNameLength *Length of program name*

```
PROCEDURE ProgramNameLength(): CARDINAL;
```

Returns the length of the program name.

String *Get environment 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 *Return environment string length*

```
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`.

DateTime	*Date and Time Type*

```
TYPE DateTime = SysClock.DateTime;
```

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

Compare	*Compare two clock values*

```
PROCEDURE Compare(dl,dr: DateTime): INTEGER;
```

The function procedure `Compare` returns:

- zero, if `dl` represents the same moment as `dr`

- value less than zero, if `dl` is before `dr`
- value greater than zero, if `dl` is after `dr`

Note: If either `dl` or `dr` is invalid, zero is returned.

SubDateDays *Date difference in days*

```
PROCEDURE SubDateDays(dl,dr: DateTime): CARDINAL;
```

The function procedure `SubDateDays` returns the number of days passed from `dr` to `dl`.

Note: If one of the parameters is invalid or if `dl` is before `dr`, zero is returned.

SubDateSecs *Time difference in seconds*

```
PROCEDURE SubDateSecs(dl,dr: DateTime): CARDINAL;
```

The function procedure `SubDateSecs` returns the number of seconds passed from `dr` to `dl`.

Note: If one of the parameters is invalid or if `dl` is before `dr`, zero is returned.

AddDateDays *Add whole number of days to date*

```
PROCEDURE AddDateDays(d: DateTime;
                      days: CARDINAL;
                      VAR res: DateTime);
```

The procedure `AddDateDays` adds `days` days to date `d` and assigns the resulting date to `res`.

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

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.

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.

weekday *Determine day of the week for system time*

```
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)
```

IsSimpleName *Is just a File Name*

```
PROCEDURE IsSimpleName*(name: ARRAY OF CHAR): BOOLEAN;
```

Returns `TRUE`, if `name` contains a file name only (without directories).

Lookup *Look up File*

```
PROCEDURE Lookup*(path,name: ARRAY OF CHAR;
                  VAR fname: DStrings.String;
                  VAR n: INTEGER);
```

Builds a filename using the search path. After a call:

`n = -1`	if `name` is not simple (`fname = name`)
`n = 0`	file is not found (the first directory is used)
`n > 0`	file is found in the `n`-th directory

UseFirst *Use First Directory*

```
PROCEDURE UseFirst*(path,name: ARRAY OF CHAR;
                    VAR fname: DStrings.String);
```

Builds a filename using the first directory from the search path.

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:

Sequence	Denotes
`*`	an arbitrary sequence of any characters, possibly empty (equivalent to `{\000-\377}` expression)
`?`	any single character (equivalent to `[\000-\377]` expression)
`[...]`	one of the listed characters
`{...}`	an arbitrary sequence of the listed characters, possibly empty
`\nnn`	the ASCII character with octal code `nnn`, where n is `[0-7]`
`&`	the logical operation AND
`|`	the logical operation OR
`^`	the logical operation NOT
`(...)`	the priority of operations
`$digit`	subexpression number (see below)

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-Fa-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 `\${}[]*?`.

Expr — *Regular expression*

```
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.

Value of `res`	Meaning
`res`≤ 0	error in position ABS(res)
`res`> 0	done

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`.

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

[MöWi91] H.Mössenböck, N.Wirth. The Programming Language Oberon-2. Structured Programming,1991, 12, 179-195.

[PIM] N.Wirth. Programming in Modula-2. 4th edition. Springer-Verlag, 1988. ISBN 0-387-50150-9.

[Wirth88] N.Wirth. From Modula-2 to Oberon. Software, Practice and Experience 18:7(1988), 661-670.

[ReWi92] M.Reiser, N.Wirth. Programming in Oberon - Steps Beyond Pascal and Modula. ACM Press, Addison Wessley, 1992. ISBN 0-201-56543-9

[Mö93] H.Mössenböck. Object Oriented Programming in Oberon-2. Springer-Verlag, 1993. ISBN 3-540-56411-X

Index

EXCELSIOR

Excelsior, LLC
6 Lavrenteva Ave. Suite 441
Novosibirsk 630090 Russia
Tel: +7 (3832) 39 78 24
Fax: +1 (509) 271 5205
Email: info@excelsior-usa.com
Web: http://www.excelsior-usa.com