GARDENS POINT MODULA
Library Definitions
Reference Manual

Contents

1 Library Definition Parts **3**
1.1 Pre-declared (pervasive) objects 4
1.2 Ascii 8
1.3 AsciiTime 9
1.4 BuildArgs 10
1.5 CardSequences 12
1.6 CardStr 14
1.7 CharInfo 16
1.8 ConvTypes 17
1.9 Coroutines 18
1.10 Exceptions 19
1.11 FREXP 22
1.12 GenSequenceSupport 23
1.13 GpFiles 25
1.14 InOut 28
1.15 IntStr 31
1.16 PathLookup 33
1.17 PipeUtilities 35
1.18 ProgArgs 37
1.19 Random 39
1.20 RealInOut 40
1.21 RealMath 41
1.22 RealStr 43
1.23 ShellPipes 46
1.24 StdError 47
1.25 StdStrings 48
1.26 Storage 51
1.27 SysClock 52
1.28 SYSTEM 54
1.29 Terminal 56
1.30 TextInOut 57
1.31 Types 60
1.32 UxFiles 61
1.33 UxHandles 64
1.34 UxProcesses 67

1.35 The ISO Standard Input/Output Library 69
1.36 ChanConsts 71
1.37 IOChan 73
1.38 IOConsts 77
1.39 IOLink 78
1.40 IOResult 83
1.41 RawIO 84
1.42 RealIO 85
1.43 RndFile 87
1.44 SeqFile 90
1.45 SIOResult 93
1.46 SRawIO 94
1.47 SRealIO 95
1.48 StdChans 97
1.49 STextIO 99
1.50 StreamFile 101
1.51 SWholeIO 103
1.52 TermChan 104
1.53 TermFile 105
1.54 TextIO 107
1.55 WholeIO 109

Chapter 1

Library Definition Parts

There are a moderately large number of libraries supplied with the compiler system. Some of these are traditional style modules as described in *Programming in Modula-2*, while others are implementations of standard library modules as currently proposed by ISO WG-13. Other modules again are intended to allow easy integration of programs with the standard facilities of the *UNIX* environment and its rich supply of tools and libraries.

As a statement of policy, we will continue to support traditional style modules whenever the use of these modules does not clash with the new standard. However, we are committed to the ISO standard and will implement all of the required standard modules, as the definitions of these become available.

In this revision of the manual, the ISO Standard Input/Output library is introduced. To ease the distinction between this library and the alternative *PIM-2* library, the ISO library is presented as a separate alphabetically-ordered group, after the unchanged alphabetically-ordered group from previous revisions.

All of the supplied modules make use of the special features of **gpm** as described in the *implementation specifics* chapter. All of the modules use special pragmas to indicate to the compiler that calls to the procedures of these modules will not give rise to cross-module recursion. This knowledge allows the compiler to apply certain optimizations which would otherwise be unsafe. Most of the modules use the !*LIBRARY* pragma, and are written in Modula and compiled by **gpm** itself. Several others are marked !*SYSTEM*. This pragma indicates to the compiler that there is no separate reference or object file for the module. These modules have implementations which are known to the compiler or its runtime system. The module SYSTEM is listed in the following sections, but is included only for purposes of documentation. The facilities of SYSTEM cannot be described in Modula, and the definition part file cannot actually be compiled.

A number of these modules, particularly those which interface with the *UNIX* -language libraries are implemented in *C*, or even (in the case of **Coroutines**) in assembly language. All such modules are distinguished by the context sensitive marks *FOREIGN* or *INTERFACE* in the definition part file, immediately before the keyword *IMPLEMENTATION*. The possibilty of producing such foreign modules is available to users, and is documented in the *interfacing to other languages* chapter.

This chapter contains "definition modules" for the pervasive identifiers and also for the module *SYSTEM*. In neither case can the actual definitions be expressed in standard Modula, so the definitions are partly Modula, partly English text.

1.1 Pre-declared (pervasive) objects

Pervasive identifiers are those which are visible everywhere in a Modula program. They are known to the compiler, and do not require any import or export. They would be rendered invisible if a user foolishly defined a new identifier with the same name.

The identifiers are shown here in the form of a *DEFINITION* part, just for the use of the human reader. Many of the facilities cannot actually be defined in this way.

```
DEFINITION MODULE $Pervasives$;
  FROM SYSTEM IMPORT BIN;
(* This module is for the human reader only. It cannot actually *)
(* be defined in this way, and cannot be compiled. All of the   *)
(* facilities defined here are built into the compiler itself.  *)

CONST NIL      = " SYSTEM.CAST(ADDRESS, 0) ";
      FALSE    = " VAL(BOOLEAN, 0) ";
      TRUE     = " VAL(BOOLEAN, 1) ";

TYPE BITSET   = " SET OF BIN[0 .. 31] ";
     BOOLEAN  = " (FALSE, TRUE) ";
     CARDINAL = " [0 .. 2 ^ 32 - 1] ";
     CHAR     = " [0C .. 377C] ";
     INTEGER  = " [-2 ^ 31 .. 2 ^ 31 - 1] ";
     PROC     = " PROCEDURE() ";
     REAL     = " IEEE floating point 'double' ";
     SHORTREAL = " single precision IEEE 'float' ";
     LONGREAL = REAL;

PROCEDURE ABORT(); (* causes a core dump *)
(* this procedure cannot be be caught by an          *)
(* exception handler, it always causes termination   *)
(********* WARNING *********** this procedure will   *)
(* move to new module next release, according to ISO *)
```

... *Continued*

```
PROCEDURE ABS(val : AnyNumericType) : AnyNumericType;
(* returns absolute value of val                      *)

PROCEDURE CAP(val : CHAR) : CHAR;
(* if val is a lower case character, return is        *)
(* corresponding upper case char. Otherwise return    *)
(* is the unchanged character.                        *)

PROCEDURE CHR(val : CARDINAL) : CHAR;
(* precondition  : val is in the range [0 .. 255]     *)
(* postcondition : returns char with ordinal val      *)

PROCEDURE DEC(VAR val : AnyOrdinalType);
PROCEDURE DEC(VAR val : AnyOrdinalType;
                  dec : CARDINAL);
(* postcondition : val = pre(val) - 1;   first form  *)
(*                 val = pre(val) - dec; second form *)
(*                 if destination type has a limited *)
(*                 range then a range check is done  *)

PROCEDURE DISPOSE(VAR ptr : AnyPointerType);
(* precondition  : ptr^ is an dynamic memory object  *)
(* postcondition : ptr^ is de-allocated, ptr = NIL   *)
(* note : variant tags are permitted, but ignored    *)

PROCEDURE EXCL(VAR set : SetType; elem : ElemType);
(* precondition  : elem must be a valid value of the *)
(*                 base type of the set type         *)
(* postcondition : set = pre(set) - {elem}           *)

PROCEDURE FLOAT(val : NumericType) : REAL;
(* precondition  : TRUE                              *)
(* postcondition : returns real with ordinal val     *)

PROCEDURE HALT();  (* quiet termination  *)
(* this procedure cannot be be caught by an          *)
(* exception handler, it always causes termination   *)

PROCEDURE HIGH(val : OpenArrayType) : CARDINAL;
(* precondition  : val must be a visible open array  *)
(* postcondition : returns max index of val          *)
```

... Continued

```
PROCEDURE INC(VAR val : AnyOrdinalType);
PROCEDURE INC(VAR val : AnyOrdinalType; inc : CARDINAL);
(* postcondition : val = pre(val) + 1;   first form  *)
(*                 val = pre(val) + inc; second form *)
(*                 if destination type has a limited *)
(*                 range then a range check is done  *)

PROCEDURE INCL(VAR set : SetType; elem : ElemType);
(* precondition  : elem must be a valid value of the *)
(*                 base type of the set type         *)
(* postcondition : set = pre(set) + {elem}           *)

PROCEDURE LENGTH(String or ARRAY OF CHAR) : CARDINAL;
(* postcondition : function returns length of string *)
(*            if possible calculated at compile time *)

PROCEDURE LFLOAT(val : NumericType) : LONGREAL; (* == REAL *)
(* precondition  : TRUE                              *)
(* postcondition : returns real with value=val       *)

PROCEDURE MAX(AnyOrdinalTypeName) : AnyOrdinalType;
PROCEDURE MIN(AnyOrdinalTypeName) : AnyOrdinalType;
(* returns the max and min value of the param type   *)

PROCEDURE NEW(VAR ptr : AnyPointerType);
(* postcondition : ptr points to a newly allocted    *)
(*                 object with correct size for ptr^ *)
(* note : variant tags are permitted as additional   *)
(*                 parameters, but gpm ignores them  *)

PROCEDURE ODD(val : AnyOrdinalType) : BOOLEAN;
(* postcondition : returns (val MOD 2 <> 0)          *)

PROCEDURE ORD(val : AnyOrdinalType) : CARDINAL;
(* precondition  : val is in the [0 .. MAX(CARDINAL) *)
(* postcondition : returns ordinal value of val      *)

PROCEDURE SIZE(AnyTypeName or Variable) : CARDINAL;
(* returns the storage size of the type or varible   *)
(* in the units of smallest addressable location SAL *)
```

... Continued

```
PROCEDURE SFLOAT(val : NumericType) : SHORTREAL;
(* precondition  : TRUE                            *)
(* postcondition : returns short real with value=val *)

PROCEDURE TRUNC(val : AnyRealType) : CARDINAL;
(* precondition  : val is >= 0.0                   *)
(* postcondition : returns integer part of val     *)

PROCEDURE VAL(AnyOrdType; val : CARDINAL) : AnyOrdType;
(* precondition  : ORD(val) is in [MIN(T) .. MAX(T)] *)
(* postcondition : returns value of T with ord = val *)

END $Pervasives$.
```

1.2 Ascii

Name definitions for the *USASCII* control characters.

```
  (* ========================================================== *)
  (* Preliminary library module for Gardens Point Modula       *)
  (* ========================================================== *)

(* !SYSTEM! *) DEFINITION MODULE Ascii;

  (* this is a system module; it is known to the compiler and  *)
  (* activated by import. There is no explicit implementation. *)

      (* standard short character names for control chars *)

  CONST nul = 00C;      soh = 01C;      stx = 02C;      etx = 03C;
        eot = 04C;      enq = 05C;      ack = 06C;      bel = 07C;
        bs  = 10C;      ht  = 11C;      lf  = 12C;      vt  = 13C;
        ff  = 14C;      cr  = 15C;      so  = 16C;      si  = 17C;
        dle = 20C;      dc1 = 21C;      dc2 = 22C;      dc3 = 23C;
        dc4 = 24C;      nak = 25C;      syn = 26C;      etb = 27C;
        can = 30C;      em  = 31C;      sub = 32C;      esc = 33C;
        fs  = 34C;      gs  = 35C;      rs  = 36C;      us  = 37C;

        del = 177C;

      (* standard synonyms for certain control characters *)

  CONST xon = dc1;      xoff = dc3;

END Ascii.
```

1.3 AsciiTime

The *SysClock* and *AsciiTime* Libraries are concerned with the fetching and displaying of time and date information. *AsciiTime* converts time in either of two forms into a printable text string.

```
FOREIGN DEFINITION MODULE AsciiTime;
    IMPORT IMPLEMENTATION FROM "asciitime.o";

   (* ========================================================= *)
   (*  This module converts Time in the structure specified in  *)
   (*  the MODULE SysClock into a printable ascii string.       *)
   (*  There is also a procedure for converting time in the     *)
   (*  "seconds since 1970 " cardinal form, into a DateTime     *)
   (*  structure.                                               *)
   (*                                                           *)
   (*  Note : This module is useful in conjunction with the     *)
   (*  the procedures for obtaining the time that are present   *)
   (*  in the UxFiles, and ProgArgs MODULES.                    *)
   (* ========================================================= *)

FROM SysClock IMPORT DateTime;

PROCEDURE StructTimeToAscii(VAR str:ARRAY OF CHAR; time:DateTime);

(*
 *  Places ascii representation of 'time' in 'str'. If there is
 *  not enough room in 'str' then the representation is truncated
 *  as required by the strings length. If there is adequate length
 *  then the string is terminated with a null character.
 *)



PROCEDURE TimeToStructTime(time : CARDINAL; VAR structTime:DateTime);
(* Converts time cardinals into the structured form. *)

END AsciiTime.
```

1.4 BuildArgs

General facilities for building argument blocks in the style required by *UNIX* facilities such as *exec*.

```
(* ========================================================== *)
(* Preliminary library module for Gardens Point Modula        *)
(* ========================================================== *)

(* !LIBRARY! *) DEFINITION MODULE BuildArgs;

  TYPE ArgPtr; (* ==> "pointer to array of strings" in UNIX  *)

  (* The first set of procedures are simple & efficient and  *)
  (* have several restrictions to their use. However, they   *)
  (* suffice for most purposes, when used as shown below.    *)

  PROCEDURE Arg1(a1 : ARRAY OF CHAR) : ArgPtr;
  PROCEDURE Arg2(a1, a2 : ARRAY OF CHAR) : ArgPtr;
  PROCEDURE Arg3(a1, a2, a3 : ARRAY OF CHAR) : ArgPtr;
  PROCEDURE Arg4(a1, a2, a3, a4 : ARRAY OF CHAR) : ArgPtr;
  (* preconditions : a's may be literals or variable arrays. *)
  (* These procedures safely copy array parameters into a    *)
  (* dynamically allocated block, adding NUL termination if  *)
  (* necessary. Actual param variables may thus be reused.   *)

  (* usage example:
        ...
        FROM BuildArgs IMPORT Arg3;
        FROM UxProcesses IMPORT Execp;
          VAR cmd, fNm : ARRAY [0 .. 39] OF CHAR;
          ...
          Execp("foo", Arg3("foo", cmd, fNm));
          Error("Can't exec 'foo'");
          ...
  *)

  (* The next set of procedures allow argument blocks of any *)
  (* size to be built, and allow for explicit reclaiming of  *)
  (* memory space from used blocks, where that is necessary. *)
  (* These procedures safely copy array parameters into a    *)
  (* dynamically allocated block, adding NUL termination if  *)
  (* necessary. Actual param variables may thus be reused.   *)
```

... Continued

```
  TYPE ArgBlock; (* args + builder state information *)

  PROCEDURE NewArgBlock(VAR b : ArgBlock; max : CARDINAL);
  (* postcondition : buffer space for max args is allocated
                     and the block state is initialized      *)

  PROCEDURE DisposeArgBlock(VAR b : ArgBlock);
  (* postcondition : buffer space is reclaimed, b is NIL     *)

  PROCEDURE AppendArg(b : ArgBlock;
                      a : ARRAY OF CHAR);
  (* precondition  : b has been initialized by NewArgBlock;
                     is not fully occupied;
     postcondition : the block designated by b is updated so
                     that a is its final argument. On block
                     overflow an index error is raised       *)

  PROCEDURE ArgsOf(b : ArgBlock) : ArgPtr;
  (* extracts the args from the valid ArgBlock buffers       *)

  (* usage example:
        ...
        WHILE condition DO
          NewArgBlock(blk,64);
          WHILE xxx DO (* build block *)
            ....
            AppendArg(blk,str);
          END;
          Foo(ArgsOf(blk)); (* use block *)
          DisposeArgBlock(blk); (* reclaim space *)
        END;
        ...
  *)
END BuildArgs.
```

1.5 CardSequences

Support for traversible sequences of cardinals. Elements may be added to either end of the sequences. One or more *cursors* may be attached to a sequence, and the values traversed from left to right.

The functionality of this module is similar to *GenSequenceSupport*, but is type-safe unlike the generic sequences of the other module. Both are quite efficient, due to internal handling of free list.

```
(*****************************************************************)
(******** support for cardinal sequences...no random access ********)
(*****************************************************************)

(* !LIBRARY *) DEFINITION MODULE CardSequences; (* kjg nov '84 *)

TYPE ElemPtr;
TYPE Sequence = RECORD
                  first : ElemPtr; (* ptr to first element *)
                  last  : ElemPtr  (* ptr to last element  *)
                END;

PROCEDURE InitSequence(VAR seq : Sequence);
          (* sets all fields NIL *)

PROCEDURE LinkLeft (VAR seq : Sequence; Element : CARDINAL);
PROCEDURE LinkRight(VAR seq : Sequence; Element : CARDINAL);

PROCEDURE InitCursor(seq : Sequence; VAR cursor : ElemPtr);
(* post: cursor is attached. GetNext will get first.      *)

PROCEDURE GetFirst(       seq : Sequence;
                   VAR cursor : ElemPtr;
                   VAR result : CARDINAL );
(* post  : cursor is attached, next GetNext will fetch 2nd *)
(* errors: GetFirst on empty sequence raises memory error  *)

PROCEDURE GetNext( VAR cursor : ElemPtr;
                   VAR result : CARDINAL);
(* pre   : cursor is already attached.                     *)
(* post  : returns next element and "increments" cursor    *)
(* errors: raises memory error is sequence already ended   *)
```

... *Continued*

```
PROCEDURE Ended(cursor : ElemPtr) : BOOLEAN;
(* precondition: cursor is attached.                        *)

PROCEDURE NextIsLast(cursor : ElemPtr) : BOOLEAN;
(* precondition: cursor is attached.                        *)

PROCEDURE IsEmpty(seq : Sequence) : BOOLEAN;
(* postcondition : returns "seq is the empty sequence"      *)

PROCEDURE LengthOf(seq : Sequence) : CARDINAL;

PROCEDURE DisposeList(VAR seq : Sequence);
  (* reinitializes the sequence header *)

END CardSequences.
```

1.6 CardStr

This module provides conversions between unsigned numbers and strings. It is one of the family of numeric conversions which comprises *CardStr, IntStr, RealStr*. All of these have consistent and similarly-named facilities.

```
(* !LIBRARY! *) DEFINITION MODULE CardStr;

(* Proposed BSI/ISO Standard Modula-2 I/O Library
 * Copyright Roger Henry, University of Nottingham
 * Version WG/4.01, February 1989
 * Permission is given to copy this Definition Module, with the
 * copyright notice intact, for the purposes of evaluation and test.
 * At the stage of a formal draft standard, Copyright will be transferred
 * to BSI (and through BSI to other recognised standards bodies).
 * Status: for review by BSI/IST/5/13;
           for review by ISO/IEC JTC1/SC22/WG13
 --* this version editted to comform to D103 kjg September 1989 *--
*)

IMPORT ConvTypes;
FROM   ConvTypes IMPORT ScanProgress, ScanProc;

(* the text form of an unsigned whole number is                  *)
(*          [whitespace] digit {digit}                           *)

PROCEDURE Scan(
      this: CHAR; VAR progress: ScanProgress; VAR nextScanner: ScanProc
);

TYPE
  ConvResults = ConvTypes.ConvResults;

PROCEDURE Format(str: ARRAY OF CHAR): ConvResults;
  (* pre:  "str" has a string value                              *)
  (* post: returned value corresponds to format of string        *)
  (*       value with respect to the type CARDINAL               *)

PROCEDURE Value(str: ARRAY OF CHAR): CARDINAL;
  (* pre:  "str" has a string value                              *)
  (*       and format is "allRight" with respect to CARDINAL     *)
  (* post: returned value is the corresponding CARDINAL          *)
```

... *Continued*

```
PROCEDURE Take(
  str: ARRAY OF CHAR; VAR card: CARDINAL; VAR format: ConvResults
);
  (* pre:  "str" has a string value                               *)
  (* post: either value of "format" is "allRight",                *)
  (*         value of "card" is the corresponding CARDINAL        *)
  (*       or value of "format" is "outOfRange",                  *)
  (*         value of "card" is MAX(CARDINAL)                     *)
  (*       or value of "format" is "wrongFormat"                  *)
  (*         value of "card" is undefined                         *)
  (*       or value of format is "noData"                         *)
  (*         value of "card" is unchanged                         *)

TYPE
  Alignment = ConvTypes.Alignment;

PROCEDURE Length(card: CARDINAL): CARDINAL;

PROCEDURE Give(
  VAR str: ARRAY OF CHAR; card: CARDINAL; width: CARDINAL; where: Alignment);
  (* post: as far as capacity of "str" allows,                              *)
  (*       the character representation of "card" is contained in "str"    *)
  (*       in a field of at least "width" characters                        *)
  (*       left, centre (numeric), or right justified;                      *)
  (*       for the special case of "width" = 0, a leading space is written *)

END CardStr.
```

1.7 CharInfo

Character information as proposed by ISO WG-13.

The predicates are very low cost, as the compiler will use macro expansion when it is able to *prove* that it is safe to do so, otherwise an actual function procedure will be called.

```
FOREIGN DEFINITION MODULE CharInfo;
  IMPORT IMPLEMENTATION FROM "charinfo.o";
(* Proposed BSI/ISO Standard Modula-2 I/O Library
 * Copyright Roger Henry, University of Nottingham
 * Version WG/4.01, February 1989
 * Permission is given to copy this Definition Module, with the
 * copyright notice intact, for the purposes of evaluation and test.
 * At the stage of a formal draft standard, Copyright will be transferred
 * to BSI (and through BSI to other recognised standards bodies).
 * This version editted by kjg to conform to D103, September 1989.
*)
PROCEDURE IsEOL(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is the character     *)
  (*       used to represent end of line internally            *)

PROCEDURE IsDigit(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is a decimal digit   *)

PROCEDURE IsSpace(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is a whitespace char *)

PROCEDURE IsSign(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is + or - sign       *)

PROCEDURE IsLetter(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is a letter          *)

PROCEDURE IsUpper(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is upper case char   *)

PROCEDURE IsLower(ch: CHAR): BOOLEAN;
  (* post: returned value is true iff "ch" is lower case char   *)

PROCEDURE IsControl(ch : CHAR): BOOLEAN;
  (* post: returns true iff "ch" is a control char, incl. EOL   *)

PROCEDURE EOL(): CHAR;
  (* post: returned value is the implementation-defined char    *)
  (*       used to represent end of line internally            *)

END CharInfo.
```

1.8 ConvTypes

This module is the basis of the numeric-to-string conversions in the ISO style input-output libraries. It defines a number of types used by the conversion modules.

```
DEFINITION MODULE ConvTypes;

(* Proposed BSI/ISO Standard Modula-2 I/O Library
 * Copyright Roger Henry, University of Nottingham
 * Version WG/4.01, February 1989
 * Permission is given to copy this Definition Module, with the
 * copyright notice intact, for the purposes of evaluation and test.
 * At the stage of a formal draft standard, Copyright will be transferred
 * to BSI (and through BSI to other recognised standards bodies).
 * Status: for review by BSI/IST/5/13;
           for review by ISO/IEC JTC1/SC22/WG13
 -- * This copy editted to conform to D103 by kjg September 1989 *--
*)

TYPE  (* result of a conversion operation *)
  ConvResults = (
    allRight,     (* data is as expected or as required *)
    outOfRange,   (* data cannot be represented         *)
    wrongFormat,  (* data not in expected format        *)
    noData,       (* no data or insufficient data       *)
    noRoom);      (* no room to store input data        *)

  (* types for lexical scanners *)

TYPE
  ScanProgress = (
    ignore,       (* leading or padding character       *)
    accept,       (* character that was expected        *)
    reject,       (* character that was not expected    *)
    stop);        (* character that terminates the item *)

  ScanProc = PROCEDURE(CHAR, VAR ScanProgress, VAR ScanProc);

  (* type and constants for alignment of text *)

  Alignment = (left, centre, right);

END ConvTypes.
```

1.9 Coroutines

The coroutines library is an ordinary library in the sense that no knowledge of the library is required by the compiler. It is implemented in assembly language and uses the usual *FOREIGN* mechanism. The library must be explicitly imported by user programs, in keeping with the proposals of *ISO WG-13*. However, this version implements the *old* coroutines model exactly as specified by Wirth. **gpm** will provide the new coroutines model when acceptance of the change is confirmed. In the meanwhile, the old style library will not break existing programs.

The library as supplied does not support the *IOTRANSFER* mechanism. However, in future releases *IOTRANSFER* will provide an interface to the *UNIX* signal handling mechanisms. In effect, a correspondence will be made between signal numbering and the "interrupt vectors" in the *PIM* definition of coroutines.

```
(* ======================================================== *)
(* Preliminary library module for Gardens Point Modula      *)
(* ======================================================== *)

FOREIGN DEFINITION MODULE Coroutines;(* coroutines as in PIM *)

  IMPORT IMPLEMENTATION FROM "coroutines.o";
  FROM SYSTEM IMPORT ADDRESS;

  TYPE Coroutine = ADDRESS;  (* use of this type avoids SYSTEM import *)

  PROCEDURE NEWPROCESS (code  : PROC;          (* body of coroutine  *)
                        space : ADDRESS;       (* ptr to workspace   *)
                        size  : CARDINAL;      (* size of workspace  *)
                    VAR this  : Coroutine);    (* returned coroutine *)

  PROCEDURE TRANSFER (VAR thisCo : Coroutine;   (* current saved here *)
                      VAR destCo : Coroutine);  (* target to activate *)

END Coroutines.
```

1.10 Exceptions

Exception handling facilities as proposed by ISO WG-13. Note that in this release an experimental version of *Call* named *CALL*, is available in module *SYSTEM*. The experimental *SYSTEM.CALL* provides compatible functionality, but with the ability to pass parameters to the call body, and to use nested procedures for both the call body and the recovery procedure. These features may or may not survive into later releases.

```
(* ======================================================== *)
(* Preliminary library module for Gardens Point Modula      *)
(* ======================================================== *)

FOREIGN DEFINITION MODULE Exceptions;
  IMPORT IMPLEMENTATION FROM "exceptions.o";
(* this is a library module; it is guaranteed not to lead to *)
(* cross module recursion -- the implementation is tightly   *)
(* linked to the runtime system module -- edit with care!    *)

  TYPE Exception;
  (* Unique exception values. Each allocated exception value is unique
     and has a text string associated with it. The text string is
     specified by the user in the case of user defined exceptions and
     is implementation defined for the standard exception values. *)

  PROCEDURE AllocateValue(text : ARRAY OF CHAR;
                          VAR newExcept : Exception);
  (* pre:    true
     post:   a unique exception value is allocated, and is assigned to
             the second actual parameter. The string corresponding to the
             first actual parameter is permanently associated with this
             exception value *)

  PROCEDURE ExtractMessage(exValue : Exception;
                           VAR text : ARRAY OF CHAR);
  (* pre:    true
     post:   the message associated with exValue is retrieved *)

  PROCEDURE Raise(reason : Exception);
  (* pre:    true
     post:   control passes to the active exception handler, with the
             specified exception value signalled
     errors: rangeError: an attempt was made to signal normalReturn *)
```

... *Continued*

```
TYPE RecoveryProc = PROCEDURE(Exception,
                              VAR BOOLEAN);
(* Procedures of this type are called when exceptions are raised.
   The Boolean parameter signals to the processor whether or not the
   parameterless procedure guarded by the exception handler should
   be re-invoked. A return value of true signals "re-invoke" *)

PROCEDURE Call(codeBody : PROC;
               onError  : RecoveryProc);
(* pre:    true
   post:   all postconditions of codeBody apply
   errors: exceptions raised in codeBody or onError may propagate
           to the exception handler which was active prior to the
           call of Exceptions.Call *)

TYPE StdExceptions =
       (normalReturn, indexError, rangeError, caseSelectError,
        invalidLocation, functionError, wholeValueError,
        wholeDivError, realValueError, realDivError, priorityError);

(* Standard exceptions correspond to exceptions defined in the
   dynamic semantics of the language. A function procedure is
   provided which returns the value of Exceptions type
   associated with each value of this enumeration *)

PROCEDURE ExceptionValue(stdValue : StdExceptions) : Exception;
(* pre:    true
   post:   the Exception value associated with stdValue is returned *)
```

... *Continued*

```
(*
      example of normal use of these facilities:
      In the module which defines and raises the exception ---

        FROM Exceptions IMPORT
                Exception, AllocateValue, Raise;

        VAR broken : Exception; (* NOT exported *)

          PROCEDURE Foo;
          BEGIN
            ...
            IF bad THEN Raise(broken) END;
            (* If Foo is called in the context of a handler the
               recovery proc will be notified reason = broken,
               else the program aborts with the specific message *)
            ...
          END Foo;

          PROCEDURE BrokenError() : Exception;
          BEGIN (* recovery proc will test "res = BrokenError()" *)
            RETURN broken;
          END BrokenError;

        BEGIN (* main body *)
          AllocateValue("broken error",broken);
          ...
        END Main.
  *)
END Exceptions.
```

1.11 FREXP

This module provides low level floating point number operations. It is used (for example) in the implementation of *RealStr*.

```
INTERFACE DEFINITION MODULE FREXP;
  (* this module provides a direct access to the libc *)
  (* functions with the same names. No special import *)
  (* statement is needed for interfaces to libc       *)

  PROCEDURE frexp(val : REAL; VAR exp : INTEGER) : REAL;
  (* returns the mantissa, and the exponent in exp *)

  PROCEDURE ldexp(val : REAL; exp : INTEGER) : REAL;
  (* returns val * 2^exp *)

  PROCEDURE modf(val : REAL; VAR iPart : REAL) : REAL;
  (* returns signed fraction part, and integer part in iPart *)

END FREXP.
```

1.12 GenSequenceSupport

Generic sequence handler for opaque or pointer types. For usage examples see chapter 7 of *Gough and Mohay*, **Modula-2, a second course in programming**, Prentice-Hall, 1988, or K.John Gough, "Writing generic utilities in Modula-2" *Journal of Pascal Ada and Modula-2*, May 1986.

```
(****************************************************************)
(******** General support for sequences...no random access *********)
(****************************************************************)

(* !LIBRARY *) DEFINITION MODULE GenSequenceSupport; (* kjg nov '84 *)

FROM SYSTEM IMPORT ADDRESS;

TYPE ElemPtr;
TYPE Sequence = RECORD
                  first : ElemPtr; (* ptr to first element *)
                  last  : ElemPtr  (* ptr to last element  *)
                END;

PROCEDURE InitSequence(VAR seq : Sequence);
          (* sets all fields NIL *)

PROCEDURE LinkLeft (VAR seq : Sequence; Element : ADDRESS);
PROCEDURE LinkRight(VAR seq : Sequence; Element : ADDRESS);

PROCEDURE InitCursor(seq : Sequence; VAR cursor : ElemPtr);
(* postcondition: cursor is attached. GetNext will get first. *)

PROCEDURE GetFirst(       seq : Sequence;
                   VAR cursor : ElemPtr;
                   VAR result : ADDRESS );
(* returns the first element. GetFirst on empty sequence rtns NIL *)
(* postcondition: cursor is attached, next GetNext will fetch 2nd *)

PROCEDURE GetNext( VAR cursor : ElemPtr;
                   VAR result : ADDRESS );
(* precondition: cursor is already attached. Returns element and- *)
(* "increments" cursor. Returns NIL if sequence is already ended. *)
```

... Continued

```
PROCEDURE Ended(cursor : ElemPtr) : BOOLEAN;
(* precondition: cursor is attached. Returns "cursor = NIL" *)

PROCEDURE NextIsLast(cursor : ElemPtr) : BOOLEAN;
(* precondition: cursor is attached. Returns "cursor = seq.last" *)

PROCEDURE IsEmpty(seq : Sequence) : BOOLEAN;
(* postcondition : returns "seq is the empty sequence" *)

PROCEDURE LengthOf(seq : Sequence) : CARDINAL;

PROCEDURE DisposeList(VAR seq : Sequence);
          (* returns list but not the linked nodes,
             also reinitializes the sequence header *)

END GenSequenceSupport.
```

1.13 GpFiles

This module provides file lookup facilities as used by the compiler itself, and all its associated tools. It allows for the previous gpm policy of lower-case-only file names, as well as mixed-case file names as in module names, in a backward-compatible manner. The mode is selected by the environment variable *GPNAMES*, with a default to lower-case-only if *GPNAMES* is undefined. Because of the backward-compatible search strategy described below, both lower-case and mixed-case file names will be found in either mode. However, if you wish to use mixed-case file names and have gpm generate matching mixed-case names, set *GPNAMES* to *MIXED*. Note too that if the final lookup of a file name truncated to DOS length is needed, lower-case is implied, since DOS file names are not case sensitive.

```
(**************************************************************)
(*                                                            *)
(*           Gardens Point Compiler Source Module             *)
(*                                                            *)
(*               File name lookup and create                  *)
(*                                                            *)
(*    (c) copyright 1992 Faculty of Information Technology.   *)
(*      This module may be used freely for the construction   *)
(*      of tools which interwork with gardens point compilers *)
(*                                                            *)
(*      original module : kjg July 1992                       *)
(*      modifications   : kjg  Sep 1992 lowercase default, use *)
(*                               GPNAMES=MIXED for mixed case... *)
(*                                                            *)
(**************************************************************)

DEFINITION MODULE GpFiles;

    FROM UxFiles IMPORT File;
(*
 *      this module implements the gpm file lookup strategy.
 *      It was introduced to make the change from lower case
 *      to mixed case file names in a consistent manner for
 *      gpm, build, gpmake, gpscript, decode, mkenumio and
 *      other tools which lookup files based on module names.
 *      The lookup strategy is backward compatible with the
 *      filenames created by previous versions of gpm.
 *
 *        GPNAMES=MIXED                   GPNAMES=LOWER
 *      lookup(1) name as given;        lookup(1) lower case;
 *      lookup(2) lower case;           lookup(2) name as given;
 *      lookup(3) DOS length;           lookup(3) DOS length;
 *)
```

... Continued

```
CONST fileNameLength    = 79;       (* max base name length *)
      oldFileNameLength = 8;        (* DOS compatibility    *)

PROCEDURE GpCreateFile(     base : ARRAY OF CHAR;
                            ext  : ARRAY OF CHAR;
                        VAR name : ARRAY OF CHAR;
                        VAR file : File);   (* NIL if can't create *)
(*          Truncate base name at fileNameLength.    *)
(*          If ext <> "" then a dot and the ext are  *)
(*          appended, otherwise the base unchanged.  *)
(*          Name is then made lower case unless the  *)
(*          env variable GPNAMES has first character *)
(*          "m" or "M" at module initialization time *)
(*  post : name is the name of the created file      *)

PROCEDURE GpFilename(       base : ARRAY OF CHAR;
                            ext  : ARRAY OF CHAR;
                        VAR name : ARRAY OF CHAR);
(*          Truncate base name at fileNameLength.    *)
(*          If ext <> "" then a dot and the ext are  *)
(*          appended, otherwise the base unchanged.  *)
(*          Name is then made lower case unless the  *)
(*          env variable GPNAMES has first character *)
(*          "m" or "M" at module initialization time *)
(*  post : name is the name of the created file      *)

PROCEDURE GpFindLocal(      base : ARRAY OF CHAR;
                            ext  : ARRAY OF CHAR;
                        VAR name : ARRAY OF CHAR;
                        VAR file : File);   (* or NIL if not found *)
(*          Look for file in current directory ...   *)
(*          If ext <> "" then look for "base.ext",   *)
(*          else look for the file "base"            *)
(*  post : name is the name of the file, if found in *)
(*          the current directory, else file = NIL   *)
```

... Continued

```
    PROCEDURE GpFindOnPath(     path : ARRAY OF CHAR;
                                base : ARRAY OF CHAR;
                                ext  : ARRAY OF CHAR;
                            VAR name : ARRAY OF CHAR;
                            VAR file : File);   (* or NIL if not found *)
    (*          Look for file first in current directory, *)
    (*          then on the given path.                   *)
    (*          If ext <> "" then look for "base.ext",    *)
    (*          else look for the file "base"             *)
    (*          Actual "name" must be long enough for the *)
    (*          longest possible path name!               *)
    (*  post : name is the pathname of the file, if       *)
    (*          file found, else file = NIL               *)

END GpFiles.
```

1.14 InOut

```
(*******************************************************************)
(*                                                                 *)
(*               Modula-2 Compiler InOut Library Module            *)
(*                                                                 *)
(*         High level input and output procedures for              *)
(*         characters, strings, integers and cardinals             *)
(*         and allows redirection of I/O  to/from files            *)
(*                                                                 *)
(*      original module : N. Wirth, PIM-2,  1982                   *)
(*      modifications   : 20-MAR-89 redirection implemented        *)
(*                                                                 *)
(*******************************************************************)

FOREIGN DEFINITION MODULE InOut;

    CONST EOL  = 12C;        (* End-of-line character *)

    VAR   Done : BOOLEAN ;  (* Status of some InOut procedure calls.
                            TRUE if the operation was successful,
                            FALSE otherwise. *)
          termCh : CHAR ;   (* Terminating character of some input
                             procs. ReadString, ReadInt, ReadCard *)

    PROCEDURE OpenInput(Extension : ARRAY OF CHAR);
    (* Precondition  : TRUE
       Postcondition : The primary input stream is redefined to be
                       the external file name whose names has been
                       supplied by the user. If the name ends with
                       a "." the string Extension is appended
                       to the end of it. Done = TRUE if and
                       only if the file is opened successfully *)

    PROCEDURE OpenOutput(Extension : ARRAY OF CHAR);
    (* Precondition  : TRUE
       Postcondition : The primary output stream is redefined to
                       be the external file name whose names has
                       been supplied by the user.
                       If the name ends with a "." the string
                       Extension is appended to the end of it.
                       Done = TRUE if and only if the file is
                       opened successfully *)
```

... Continued

```
PROCEDURE CloseInput;
(* Precondition  : TRUE
   Postcondition : The primary input stream is redefined to
                   be the terminal keyboard and the previously
                   used input stream is closed *)

PROCEDURE CloseOutput;
(* Precondition  : TRUE
   Postcondition : The primary output stream is redefined to
                   be the terminal keyboard and the previously
                   used output stream is closed *)

PROCEDURE Read(VAR c:CHAR);
(* Precondition  : TRUE
   Postcondition : Done = FALSE if and only if the end of the
                   primary input stream is reached, otherwise
                   c is the next character in the stream.  *)

PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
(* Precondition  : TRUE
   Postcondition : Inputs a character string from the primary
                   input stream until any character less than or
                   equal to a blank is read. The variable termCh
                   is set to the value of this final character.
                   The nul character (0C) or the end of the array
                   is used to mark the end of the string. Leading
                   blanks and/or tabs are ignored. Excess characters
                   beyond the length of s are discarded. *)

PROCEDURE ReadCard(VAR n: CARDINAL);
(* Precondition  : TRUE
   Postcondition : Done = TRUE if and only if the next sequence of
                   characters on the input stream represents a
                   CARDINAL value. The variable termCh is set to the
                   value of the character that ends this sequence. *)

PROCEDURE ReadInt(VAR i : INTEGER);
(* Precondition  : TRUE
   Postcondition : Done = TRUE if and only if the next sequence of
                   characters on the input stream represents an
                   INTEGER value. The variable termCh is set to the
                   value of the character that ends this sequence. *)
```

... Continued

```
    PROCEDURE Write(c:CHAR);
    (* Precondition  : c is defined.
       Postcondition : The character representation corresponding to
                       the value of c is written to the output stream *)

    PROCEDURE WriteLn;
    (* Precondition  : TRUE
       Postcondition : Equivalent to Write(EOL).  *)

    PROCEDURE WriteString(s : ARRAY OF CHAR);
    (* Precondition  : s is defined.
       Postcondition : Outputs a string of characters until an Ascii.nul
                       or the end of the array is encountered.  *)

    PROCEDURE WriteCard(n: CARDINAL; w: CARDINAL);
    (* Precondition  : n and w are defined.
       Postcondition : The value of n is written to the output
                       stream occupying at least w character
                       positions. Leading blanks fill out
                       the space if it is not all required.
                       The decimal number system is used.  *)

    PROCEDURE WriteInt(i: INTEGER; w: CARDINAL);
    (* Precondition  : i and w are defined.
       Postcondition : The value of i is written to the output stream
                       occupying at least w character positions.
                       Leading blanks fill out the space if it is not
                       all required. The decimal number system is used
                       and a sign is displayed only
                       for negative numbers.  *)

    PROCEDURE WriteOct(n: CARDINAL; w: CARDINAL);
    (* Precondition  : n and w are defined.
       Postcondition : The value of n is written to the output stream
                       occupying at least w character positions.
                       Leading blanks fill out the space if it is not
                       all required. The octal number system is used. *)

    PROCEDURE WriteHex(n: CARDINAL; w: CARDINAL);
    (* Precondition  : n and w are defined.
       Postcondition : The value of n is written to the output stream
                       occupying at least w character positions.
                       Leading blanks fill out the space if it is not
                       all required. The hexadecimal number system
                       is used.  *)
END InOut.
```

1.15 IntStr

This module provides conversions between unsigned numbers and strings. It is one of the family of numeric conversions which comprises *CardStr, IntStr, RealStr.* All of these have consistent and similarly-named facilities.

```
(* !LIBRARY! *) DEFINITION MODULE IntStr;

(* Proposed BSI/ISO Standard Modula-2 I/O Library
 * Copyright Roger Henry, University of Nottingham
 * Version WG/4.01, February 1989
 * Permission is given to copy this Definition Module, with the
 * copyright notice intact, for the purposes of evaluation and test.
 * At the stage of a formal draft standard, Copyright will be transferred
 * to BSI (and through BSI to other recognised standards bodies).
 * Status: for review by BSI/IST/5/13;
           for review by ISO/IEC JTC1/SC22/WG13
 --* this version editted to conform to D105 kjg September 1989 *--
*)

IMPORT ConvTypes;
FROM   ConvTypes IMPORT ScanProgress, ScanProc;

(* the text form of a signed whole number is                      *)
(*          [whitespace] ["+" | "-"] digit {digit}                *)

PROCEDURE Scan(
      this: CHAR; VAR progress: ScanProgress; VAR nextScanner: ScanProc);
(* corresponds to the start state of a FSA scanner for integers *)

TYPE ConvResults = ConvTypes.ConvResults;

PROCEDURE Format(str: ARRAY OF CHAR): ConvResults;
  (* pre:  "str" has a string value                              *)
  (* post: returned value corresponds to format of string        *)
  (*       value with respect to the type INTEGER                *)

PROCEDURE Value(str: ARRAY OF CHAR): INTEGER;
  (* pre:  "str" has a string value                              *)
  (*       and format is "allRight" with respect to INTEGER      *)
  (* post: returned value is the corresponding INTEGER           *)
```

... *Continued*

```
PROCEDURE Take(
  str: ARRAY OF CHAR; VAR int: INTEGER; VAR format: ConvResults
);
  (* pre:  "str" has a string value                             *)
  (* post: either value of "format" is "allRight",              *)
  (*         value of "int" is the corresponding INTEGER        *)
  (*       or value of "format" is "outOfRange",                *)
  (*         value of "int" is MAX(INTEGER) or MIN(INTEGER)     *)
  (*       or value of "format" is "wrongFormat"                *)
  (*         value of "int" is undefined                        *)
  (*       or value of format is "noData"                       *)
  (*         value of "int" is unchanged                        *)

TYPE Alignment = ConvTypes.Alignment; (* left, centre, right *)

PROCEDURE Length(int: INTEGER): CARDINAL;

PROCEDURE Give(
  VAR str: ARRAY OF CHAR; int: INTEGER; width: CARDINAL; where: Alignment);
  (* post: as far as capacity of "str" allows,                          *)
  (*       the character representation of "int" is contained in "str"  *)
  (*       in a field of at least "width" characters                    *)
  (*       left, centre (numeric), or right justified;                  *)
  (*       for the special case of "width" = 0, a leading space is written *)

END IntStr.
```

1.16 PathLookup

File lookups on directory paths

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

(* !LIBRARY! *) DEFINITION MODULE PathLookup;

  FROM UxFiles IMPORT File;

  PROCEDURE FindAndOpen(pathString   : ARRAY OF CHAR;
                        baseName     : ARRAY OF CHAR;
                        VAR openName : ARRAY OF CHAR;
                        VAR openFile : File);

  (* precondition  : pathString is nul terminated and has    *)
  (*                 components separated by colon chars,    *)
  (*                 openName is long enough for pathname    *)
  (* postcondition : openFile <> NIL ==> file was found.     *)
  (*                 openName has nul terminated absolute    *)
  (*                 path name, or nul string if not found   *)


  PROCEDURE FindAbsName(pathString   : ARRAY OF CHAR;
                        baseName     : ARRAY OF CHAR;
                        VAR openName : ARRAY OF CHAR;
                        VAR found    : BOOLEAN);

  (* precondition  : pathString is nul terminated and has    *)
  (*                 components separated by colon chars,    *)
  (*                 openName is long enough for pathname    *)
  (* postcondition : found = TRUE ==> file was found.        *)
  (*                 openName has nul terminated absolute    *)
  (*                 path name, or nul string if not found   *)
```

... *Continued*

```
(* example of usage : to find file "foo.mod" on path $FOOPATH --

  FROM PathLookup IMPORT FindAbsName;
  FROM ProgArgs IMPORT EnvironString;

  VAR file  : File;
      path  : ARRAY [0 .. 99] OF CHAR;
      name  : ARRAY [0 .. 79] OF CHAR;
      found : BOOLEAN;

  BEGIN
    EnvironString("FOOPATH",path);
    FindAbsName(path,"foo.mod",name,found);
    IF found THEN
      WriteString("found ");
      WriteString(name); WriteLn;
       -- and do whatever
    ELSE WriteString("not found");
    END;
*)
END PathLookup.
```

1.17 PipeUtilities

This library provides access to the underlying pipe mechanisms of *UNIX*. The usage example gives most of the needed information.

Note carefully the need to close unused file descriptors. If this is not done, the termination of a process on one end of the pipe will not send an end-of-file to any processes waiting at the other end of the pipe. This is a common cause of error in the use of pipes.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula      *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE PipeUtilities;
  IMPORT IMPLEMENTATION FROM "pipeutilities.o";
  FROM UxFiles IMPORT File;

  CONST input  = 0; output = 1; stderr = 2;

  TYPE  FDPair = ARRAY [input .. output] OF INTEGER;

  PROCEDURE GetPipe(VAR pipeFDs : FDPair);
  (* Opens a pair of file descriptors with the pipe ends in them *)

  PROCEDURE ClosePipe(pipeFD : FDPair);
  (* Closes both descriptors of the given pipe  *)

(*
 *  The next procedures connect an input or output stream (UxFiles.File)
 *  to one end of the pipe, the unused end is closed. Use UxFiles
 *  operations on the stream, and call UxFiles.Close(stream) to finish.
 *)

  PROCEDURE FileInFromPipe(    pipeFD : FDPair;
                           VAR stream : File;
                           VAR done   : BOOLEAN);

  PROCEDURE FileOutToPipe (    pipeFD : FDPair;
                           VAR stream : File;
                           VAR done   : BOOLEAN);
```

... Continued

```
(*
 *  The next procedures attach the standard input or output streams to
 *  the opened pipe. Terminal, InOut, and StdError may then be used.
 *)

  PROCEDURE InputFrom(pipe : FDPair);
  (* Dups the nominated pipe to standard input  *)

  PROCEDURE OutputTo(pipe : FDPair);
  (* Dups the nominated pipe to standard output *)

  PROCEDURE ErrorOutTo(pipe : FDPair);
  (* Dups the nominated pipe to standard error  *)

  (* typical usage :

        FROM PipeUtilities IMPORT
                FDPair, GetPipe, ClosePipe, InputFrom, OutputTo;
        FROM UxProcesses IMPORT Fork, Execp, Wait;
        FROM BuildArgs IMPORT Arg2;
          VAR  pip1 : FDPair;
          ...
          GetPipe(pip1);
          IF Fork() = 0 THEN (* in child 1 here *)
            OutputTo(pip1);  (* copy pipe desc  *)
            ClosePipe(pip1); (* close original  *)
            Execp("cat",Arg2("cat","file.mod"));
          ELSIF Fork() = 0 THEN (* child 2 here *)
            InputFrom(pip1); (* copy pipe desc  *)
            ClosePipe(pip1); (* close original  *)
            Execp("grep",Arg2("grep","foo"));
          ELSE (* in parent process here *)
            ClosePipe(pip1); (* so end of child 1 will wake child 2 *)
            id := Wait(res); id := Wait(res); (* wait for both *)
          END;
  *)
END PipeUtilities.
```

1.18 ProgArgs

Various *UNIX* facilities. Access to program arguments, passed to the program at call, and the return of a result value on exit.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

(* !SYSTEM! *) DEFINITION MODULE ProgArgs;

(* this is a system module; it is known to the compiler and  *)
(* activated by import. There is no explicit implementation. *)

  PROCEDURE ArgNumber() : CARDINAL;
  (* postcondition : returns number of arguments with UNIX   *)
  (*                 conventions. Returns 1 if no args given *)

  PROCEDURE GetArg(num : CARDINAL; VAR arg : ARRAY OF CHAR);
  (* precondition  : 0 <= num <= ArgNumber() - 1             *)
  (* postcondition : arg is a nul terminated string. ( ==>   *)
  (*                 it is fast even if HIGH(arg) >> length) *)

  (* usage example: to simply print all the arguments --

       FOR ix := 0 TO ArgNumber() - 1 DO
         GetArg(ix,str); WriteString(str); WriteLn;
       END;
  *)

  PROCEDURE EnvironString(inStr  : ARRAY OF CHAR;
                      VAR outStr : ARRAY OF CHAR);
  (* precondition  : HIGH(outStr) must be > than length of   *)
  (*                 outString ... no checking is done.      *)
  (* postcondition : outStr holds value of environ variable, *)
  (*                 or an empty string if inStr not defined *)

  PROCEDURE VersionTime(VAR outStr : ARRAY OF CHAR);
  (* precondition  : HIGH(outStr) > 26                       *)
  (* postcondition : outStr is a nul terminated string with  *)
  (*                 the date and time at which the program  *)
  (*                 was built. The string includes newline  *)
```

... *Continued*

```
  (* usage example: to print version time --

       WriteString("Program version of ");
       VersionTime(str);
       WriteString(str); (* no WriteLn needed *)
  *)

  PROCEDURE UNIXtime() : CARDINAL;
  (* returns time in seconds since 00:00:00, GMT Jan 1 1970 *)

  PROCEDURE UNIXexit(res : CARDINAL);
  (* exits program returning result code res to UNIX caller *)

  PROCEDURE Assert(expr : BOOLEAN);
  (* if expr is FALSE, the program aborts with a message    *)
  (* including automatically generated module name and line *)
  (* NOTE: "gpm -a file" suppresses all assertion tests     *)

END ProgArgs.
```

1.19 Random

```
DEFINITION MODULE Random;
(*
  Random number generator.
  Uses the 'Minimal standard random number generator' described by
  Park & Miller, CACM 31,10,Oct 88 p1192. The code has been checked
  for the 10001st random as specified in Park & Miller p1195.

  This version returns a REAL randomly distributed in the closed range
  [0.0,1.0], and will be correct if the real mantissa is 46 bits
  or larger (including sign bit). The sequence is guaranteed to produce
  all values of the form n/m for n = 0 to m, where m is 2147483646
  (=2**31-2), before cycling.
*)

PROCEDURE InitRandom (seed : REAL);
(* Initialise the random number generator with the given seed.          *)
(* The seed should be in the range [1.0,2147483646.0] - values outside  *)
(* this range are clamped to the extreme values.                        *)
(* If InitRandom is not called, the system clock is used to initialise  *)
(* the sequence; thus InitRandom must be called to create a             *)
(* reproducible sequence.                                               *)
(* InitRandom may be called as often as needed.                         *)

PROCEDURE Random() : REAL;
(* Return the next pseudo-random number in the closed range [0.0,1.0]   *)

END Random.
```

1.20 RealInOut

```
(*****************************************************************)
(*                                                               *)
(*           Modula-2 Compiler RealInOut Library Module          *)
(*                                                               *)
(*       High level input and output procedures for              *)
(*       REAL numbers. RealInOut will be redirected              *)
(*       when InOut is redirected.                               *)
(*                                                               *)
(*                                                               *)
(*     original module : N. Wirth, PIM-2, 1982                   *)
(*     modifications   :                                         *)
(*                                                               *)
(*****************************************************************)

(* !LIBRARY! *)DEFINITION MODULE RealInOut;

VAR
   Done : BOOLEAN ;

   PROCEDURE ReadReal(VAR x : REAL);
   (* Precondition  : TRUE
      Postcondition : Done = TRUE if and only if the next sequence of
                      characters represents a REAL value.  *)

   PROCEDURE WriteReal(x : REAL; width :CARDINAL);
   (* Precondition  : x and width are defined.
      Postcondition : Outputs a REAL value occupying at least width
                      character positions. Leading blanks fill out the
                      space if is not all required.  *)

END RealInOut.
```

1.21 RealMath

```
(*********************************************************************)
(*                                                                   *)
(*              Modula-2 Compiler RealMath Library Module            *)
(*                                                                   *)
(*          Provides a number of mathematical                        *)
(*          functions and REAL/INTEGER conversions                   *)
(*                                                                   *)
(*     original module : ISO WG-13, 1989                             *)
(*     modifications   :                                             *)
(*                                                                   *)
(*********************************************************************)

FOREIGN DEFINITION MODULE RealMath;
   IMPORT IMPLEMENTATION FROM "-ln";

   CONST
     PI = 3.1415926535897932385;
     Exp1 = 2.7182818285450452354;

   PROCEDURE sqrt(x : REAL):REAL;
   (* Precondition  : x is defined and >=0.0
      Postcondition : returns the square root of x.
   *)

   PROCEDURE exp(x : REAL):REAL;
   (* Precondition  : x is defined.
      Postcondition : returns the exponential of x.   *)

   PROCEDURE ln(x : REAL):REAL;
   (* Precondition  : x is defined and >= 0.0.
      Postcondition : returns the natural logarithm of x.  *)

   PROCEDURE sin(theta: REAL):REAL;
   (* Precondition  : theta is defined and is an angle in radians.
      Postcondition : returns the sine of theta.  *)

   PROCEDURE cos(theta: REAL):REAL;
   (* Precondition  : theta is defined and is an angle in radians.
      Postcondition : returns the cosine of theta.  *)
```

... *Continued*

```
   PROCEDURE tan(x : REAL):REAL;
   (* Precondition  : x is defined
      Postcondition : returns the tangent of x *)

   PROCEDURE arctan(x : REAL):REAL;
   (* Precondition  : x is defined
      Postcondition : returns the arctangent of x expressed in radians. *)

   PROCEDURE power(x, y : REAL):REAL;
   (* Precondition  : true
      Postcondition : returns the REAL approximation to
                      "x raised to the power of y" *)

   PROCEDURE round(x : REAL):INTEGER;
   (* Precondition  : x is defined and the result is representable as an
                      INTEGER.
      Postcondition : returns the INTEGER approximation to x  *)

END RealMath.
```

1.22 RealStr

This module is one of a family of numeric to string conversion modules which comprises *CardStr, IntStr, RealStr*. These all have consistent procedure names and behaviour.

RealStr provides three different formats for result strings: *Float, Eng* and *Fixed*. The default significance is 16 decimal digits.

```
(* !LIBRARY! *) DEFINITION MODULE RealStr;

(* Proposed BSI/ISO Standard Modula-2 I/O Library
 * Copyright Roger Henry, University of Nottingham
 * Version WG/4.01, February 1989
 * Permission is given to copy this Definition Module, with the
 * copyright notice intact, for the purposes of evaluation and test.
 * At the stage of a formal draft standard, Copyright will be transferred
 * to BSI (and through BSI to other recognised standards bodies).
 * Status: for review by BSI/IST/5/13;
           for review by ISO/IEC JTC1/SC22/WG13
 --* this version editted to comform to D105 kjg September 1989 *--
*)

IMPORT ConvTypes;
FROM   ConvTypes IMPORT ScanProgress, ScanProc;

(* the text form of an real number is                              *)
(* where w ~ whitespace, and d ~ decimal digit                     *)
(*      [w] ["+"|"-"] d{d} ["." {d}] ["E" ["+"|"-"] d{d}]          *)

PROCEDURE Scan(
      this: CHAR; VAR progress: ScanProgress; VAR nextScanner: ScanProc
);

TYPE
  ConvResults = ConvTypes.ConvResults;

PROCEDURE Format(str: ARRAY OF CHAR): ConvResults;
  (* pre:  "str" has a string value                                *)
  (* post: returned value corresponds to format of string          *)
  (*       value with respect to the type REAL                     *)

PROCEDURE Value(str: ARRAY OF CHAR): REAL;
  (* pre:  "str" has a string value                                *)
  (*       and format is "allRight" with respect to REAL           *)
  (* post: returned value is the corresponding REAL                *)
```

... *Continued*

```
PROCEDURE Take(
  str: ARRAY OF CHAR; VAR real: REAL; VAR format: ConvResults
);
  (* pre:  "str" has a string value                          *)
  (* post: either value of "format" is "allRight",           *)
  (*         value of "real" is the corresponding REAL       *)
  (*       or value of "format" is "outOfRange",             *)
  (*         value of "real" is the NaNS "HUGE"              *)
  (*       or value of "format" is "wrongFormat"             *)
  (*         value of "real" is undefined                    *)
  (*       or value of format is "noData"                    *)
  (*         value of "real" is unchanged                    *)

TYPE
  Alignment = ConvTypes.Alignment;

PROCEDURE LengthFloat(real: REAL; sigFigs: CARDINAL): CARDINAL;
  (* Returns the number of characters in the floating point     *)
  (* string of real using the given number of significant places *)

PROCEDURE GiveFloat(
        VAR str    : ARRAY OF CHAR;
            real   : REAL;
            sigFigs: CARDINAL;
            width  : CARDINAL;
            where  : Alignment
);
  (* post: as far as capacity of "str" allows,               *)
  (*       the string representation of "real" is contained in   *)
  (*       "str" in a field of at least "width" characters      *)
  (*       using sigFigs significant places                  *)

PROCEDURE LengthEng(real: REAL; sigFigs: CARDINAL): CARDINAL;
  (* Returns the number of characters in the engineering format  *)
  (* string of real using the given number of significant places *)
```

... *Continued*

```
PROCEDURE GiveEng(
        VAR str    : ARRAY OF CHAR;
            real   : REAL;
            sigFigs: CARDINAL;
            width  : CARDINAL;
            where  : Alignment
);
  (* post: as far as capacity of "str" allows,                   *)
  (*       the engineering representation of "real" is contained *)
  (*       in "str" in a field of at least "width" characters    *)
  (*       using sigFigs significant places                      *)

PROCEDURE LengthFixed(real: REAL; place: INTEGER): CARDINAL;
  (* Returns the number of characters in the fixed point         *)
  (* string of real rounded at position "place". Positive values *)
  (* of "place" correspond to positions in the fraction part.    *)

PROCEDURE GiveFixed(
        VAR str    : ARRAY OF CHAR;
            real   : REAL;
            place  : INTEGER;
            width  : CARDINAL;
            where  : Alignment
);
  (* post: as far as capacity of "str" allows,                   *)
  (*       the fixed point representation of "real" is contained *)
  (*       in "str", rounded at position place. Positive values  *)
  (*       of "place" correspond to the fraction part.           *)

END RealStr.
```

1.23 ShellPipes

This module provides a facility to perform input and output to input/output streams which are pipes from specified shell commands. The module is used in two different ways: a shell command may be executed so that the output of the command provides input to the stream, with the stream being read by the standard facilities of module *UxFiles*. Alternatively, output to the file is directed via a pipe to the specified shell command. In this latter case the shell command receives its standard input from the pipe.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula      *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE ShellPipes;
  IMPORT IMPLEMENTATION FROM "shellpipes.o";
  FROM UxFiles IMPORT File;

  PROCEDURE PipeInput(command : ARRAY OF CHAR;
                  VAR stream  : File;
                  VAR opened  : BOOLEAN);
  (* An input stream is returned, or opened is returned false  *)
  (* The standard output of the shell command is piped to the  *)
  (* stream, where it may be read by standard input operations *)

  PROCEDURE PipeOutput(command : ARRAY OF CHAR;
                   VAR stream  : File;
                   VAR opened  : BOOLEAN);
  (* An output stream is returned, or opened is returned false *)
  (* Output to the stream is piped as standard input to the    *)
  (* specified shell command                                   *)

  PROCEDURE ClosePipe(stream : File);
  (* closes the pipe associated with the specified stream      *)

  (* typical usage :

      FROM ShellPipes IMPORT PipeInput, ClosePipe;
        ...
        PipeInput("ls -1 *.def", inFile, ok);
        IF ok THEN
          WHILE NOT EndFile(inFile) DO
            ...
          END;
          ClosePipe(inFile);
        ELSE ...
  *)
END ShellPipes.
```

1.24 StdError

This module provides very simple facilities for writing characters, strings and unsigned numbers to the standard output stream.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE StdError;
   IMPORT IMPLEMENTATION FROM "stderror.o";

(* these procedures are identical to the     *)
(* procedures from module Terminal with the  *)
(* same names, except that output goes to    *)
(* the standard UNIX error stream "stderr"   *)

  PROCEDURE WriteString(str : ARRAY OF CHAR);

  PROCEDURE WriteLn;

  PROCEDURE WriteCard(card  : CARDINAL;
                      width : CARDINAL);

  PROCEDURE Write(ch : CHAR);

END StdError.
```

1.25 StdStrings

String handling procedures, and predicates. This is the stardard module as proposed by ISO WG-13.

```
(* ====================================================== *)
(* Preliminary library module for Gardens Point Modula       *)
(* in this implementation the string terminator is Ascii.nul *)
(* ====================================================== *)

(* !LIBRARY! *) DEFINITION MODULE StdStrings;

TYPE String1 = ARRAY [0 .. 0] OF CHAR;
(* allows single char variables to be used as strings *)

CONST StringCapacity = 256;
TYPE String  = ARRAY [0 .. StringCapacity - 1] OF CHAR;
(* a conveniently sized array for general purposes *)

PROCEDURE Length(stringValue : ARRAY OF CHAR) : CARDINAL;
(* number of character in string, not including nul *)

PROCEDURE CanAssignAll(sourceLength : CARDINAL;
                   VAR destination : ARRAY OF CHAR) : BOOLEAN;
(* Returns sourceLength <= HIGH(destination) + 1 *)

PROCEDURE Assign    ( source : ARRAY OF CHAR;
             VAR destination : ARRAY OF CHAR);
(* Source is copied to destination. If shorter, string
   is nul terminated, if too long it is truncated      *)

PROCEDURE CanExtractAll(sourceLength : CARDINAL;
                          startIndex : CARDINAL;
                     numberToExtract : CARDINAL;
                     VAR destination : ARRAY OF CHAR) : BOOLEAN;
(* Returns (startIndex + numberToExtract <= sourceLength) AND
          (HIGH(destination) + 1 >= numberToExtract)  *)

PROCEDURE Extract   ( source : ARRAY OF CHAR;
                  startIndex : CARDINAL;
             numberToExtract : CARDINAL;
             VAR destination : ARRAY OF CHAR);
(* Extracts numberToExtract characters starting at startIndex.
   If source string is too short less will be extracted, even
   zero characters if source does not extend past startIndex *)
```

... *Continued*

```
PROCEDURE CanDeleteAll(stringLength : CARDINAL;
                          startIndex : CARDINAL;
                      numberToDelete : CARDINAL) : BOOLEAN;
(* Returns (startIndex < stringLength) AND
           (startIndex + numberToDelete <= stringLength) *)

PROCEDURE Delete (VAR string : ARRAY OF CHAR;
                  startIndex : CARDINAL;
              numberToDelete : CARDINAL);
(* Deletes numberToDelete characters starting at startIndex.
   If string is short then fewer may be deleted, even none *)

PROCEDURE CanInsertAll(sourceLength : CARDINAL;
                         startIndex : CARDINAL;
                    VAR destination : ARRAY OF CHAR) : BOOLEAN;
(* Returns (startIndex < Length(destination)) AND
         (sourceLength + Length(destination) <= HIGH(destination) + 1) *)

PROCEDURE Insert    ( source : ARRAY OF CHAR;
                  startIndex : CARDINAL;
             VAR destination : ARRAY OF CHAR);
(* Inserts source string starting at startIndex. Resulting
   string is truncated if necessary to fit in destination array *)

PROCEDURE CanReplaceAll(sourceLength : CARDINAL;
                          startIndex : CARDINAL;
                     VAR destination : ARRAY OF CHAR) : BOOLEAN;
(* Returns (sourceLength + startIndex <= Length(destination)) *)

PROCEDURE Replace   ( source : ARRAY OF CHAR;
                  startIndex : CARDINAL;
             VAR destination : ARRAY OF CHAR);
(* Replaces destination characters with those of source, starting
   at startIndex. Result string is always the same length as before *)

PROCEDURE CanAppendAll(sourceLength : CARDINAL;
                    VAR destination : ARRAY OF CHAR) : BOOLEAN;
(* Returns Length(destination) + sourceLength <= HIGH(destination) + 1 *)

PROCEDURE Append    ( source : ARRAY OF CHAR;
             VAR destination : ARRAY OF CHAR);
(* Appends source string to destination. Resulting string is
   truncated if necessary, to fit into destination array *)
```

... Continued

```
PROCEDURE Capitalize(VAR stringVar : ARRAY OF CHAR);
(* Applies the CAP function to each character in string *)

TYPE CompareResult = (less, equal, greater);

PROCEDURE Compare(stringVal1 : ARRAY OF CHAR;
                  stringVal2 : ARRAY OF CHAR) : CompareResult;
(* Compares according to the underlying collating sequence
   of the character set (Ascii in this case). Returns less
   if stringVal1 < stringVal2 and so on                 *)

PROCEDURE FindNext ( pattern : ARRAY OF CHAR;
                 stringValue : ARRAY OF CHAR;
                  startIndex : CARDINAL;
            VAR patternFound : BOOLEAN;
            VAR posOfPattern : CARDINAL);
(* Finds next occurrence of pattern in stringValue,
   starting at index greater than or equal to startIndex.
   If pattern is not found, posOfPattern is not changed *)

PROCEDURE FindPrev ( pattern : ARRAY OF CHAR;
                 stringValue : ARRAY OF CHAR;
                  startIndex : CARDINAL;
            VAR patternFound : BOOLEAN;
            VAR PosOfPattern : CARDINAL);
(* Finds previous occurrence of pattern in stringValue,
   starting at index greater than or equal to startIndex
   If pattern is not found, posOfPattern is not changed *)

PROCEDURE FindDiff(stringVal1 : ARRAY OF CHAR;
                   stringVal2 : ARRAY OF CHAR;
          VAR differenceFound : BOOLEAN;
          VAR posOfDifference : CARDINAL);
(* Finds first character index in which the strings differ *)

END StdStrings.
```

1.26 Storage

Simple dynamic storage procedures. Usually accessed via *NEW* and *DISPOSE*.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE Storage;

  IMPORT IMPLEMENTATION FROM "storage.o & -lmalloc";

  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE(VAR ptr : ADDRESS; size : CARDINAL);
  (* postcondition : returns a ptr to object of size bytes   *)
  (*                 rounded up to a whole number of words   *)

  PROCEDURE DEALLOCATE(VAR ptr : ADDRESS; size : CARDINAL);
  (* precondition  : ptr^ must have been gained from heap    *)
  (* postcondition : (pre-ptr)^ is disposed, ptr = NIL       *)

END Storage.
```

1.27 SysClock

The *SysClock* and *AsciiTime* Libraries are concerned with the fetching and displaying of time of day information. *SysClock* is a preliminary implementation of a proposed standard module.

```
FOREIGN  DEFINITION MODULE SysClock;
  IMPORT IMPLEMENTATION FROM "sysclock.o";

  (* ========================================================= *)
  (* Preliminary module for Gardens Point Modula               *)
  (* Proposed BSI/ISO Standard Modula-2 Library                *)
  (* copyright assigned by R. Sutcliffe to BSI                 *)
  (* version 1990 12 14                                        *)
  (* Permission is given to copy this Definition Module, with  *)
  (* the copy  right  notice intact,  for the  purpose of      *)
  (* evaluation and test.                                      *)
  (* Status : for review by ISO/IEC STC1/SC22/WG13             *)
  (* Note : This module may change form in the future          *)
  (* ========================================================= *)

  CONST maxSecondParts = 0;  (* implementation defined constant *)
   (* The value here implies the clock delivers only whole seconds *)

  TYPE  Month = [1 .. 12];
        Day   = [1 ..31];
        Hour  = [0 .. 23];
        Min   = [0 .. 59];
        Sec   = [0 .. 59];
        Fraction = [0 .. maxSecondParts];      (* parts of a second *)
        UTCDiff  = [-780 .. 720]; (* Time zone Differential Factor  *)
                               (* the number of minutes that must be *)
                               (* added to local time to obtain UTC. *)

  TYPE  DateTime = RECORD
                     year      : CARDINAL;
                     month     : Month;
                     day       : Day;
                     hour      : Hour;
                     minute    : Min;
                     second    : Sec;
                     fractions : Fraction;
                     zone      : UTCDiff;
                     SummerTimeFlag : BOOLEAN;  (* local usage *)
                   END;
```

... *Continued*

```
PROCEDURE CanGetClock () : BOOLEAN;
(* Tests if a clock can be read *)

PROCEDURE CanSetClock () : BOOLEAN;
(* Tests if a clock can be set *)

PROCEDURE IsValidDateTime (userData : DateTime) : BOOLEAN;
(* Tests if the value of userData is a valid *)

PROCEDURE GetClock (VAR userData : DateTime);
(* Assigns local date and time of day to userData *)

PROCEDURE SetClock (userData : DateTime);
(* Sets the system clock to the given local date and time *)

END SysClock.
```

1.28 SYSTEM

This is a dummy definition. It describes the SYSTEM facilities for the human reader. The module conforms to the ISO WG-13 proposals. Compared to earlier versions, as described in Wirth's *Programming in Modula-2*, the following functions and procedures are new

INCADR, DECADR, DIRADR, CAST, SHIFT and *ROTATE*

The first three perform address arithmetic, *CAST* replaces the unsafe *type transfer functions*, while the last two manipulate bitsets.

Missing from the this module are the traditional coroutine facilities, which have been moved to the separate module **Coroutines**

```
  (* ========================================================= *)
  (* Preliminary library module for Gardens Point Modula       *)
  (* ========================================================= *)

(* !SYSTEM! *) DEFINITION MODULE $SYSTEM$;

  (* This is a system module; it is known to the compiler and  *)
  (* activated by import. It is pre-loaded, and thus has no    *)
  (* symbol file. Some of the following declarations are sym-  *)
  (* bolic only and use generic forms which cannot be compiled *)
  (* The system types have special semantics in most cases.    *)

  TYPE  LOC  = "The smallest addressable unit of storage";
        BIN  = [0 .. 31]; (* BITSET = SET OF BIN *)
        WORD = "natural word size";
        BYTE = LOC;
        ADDRESS = POINTER TO LOC;

  PROCEDURE ADR   (VAR v : AnyType) : ADDRESS;
  (* returns storage address of the parameter *)

(******** WARNING ***********************************
  the following three procedures may change their
  names in the next release, according to ISO D 103
*)
  PROCEDURE INCADR(a : ADDRESS; i : CARDINAL) : ADDRESS;
  (* returns the address a "incremented" by i *)
  PROCEDURE DECADR(a : ADDRESS; d : CARDINAL) : ADDRESS;
  (* returns the address a "decremented" by i *)
  PROCEDURE DIFADR(a, b : ADDRESS) : INTEGER;
  (* returns the difference (a - b) in bytes  *)
(******** WARNING **********************************)
```

... *Continued*

```
  PROCEDURE TSIZE (AnyType; optional tags) : CARDINAL;
  (* returns storage size of t, ignoring tags *)

  PROCEDURE CAST  (AnyType1; x : AnyType2) : AnyType1;
  (* returns x cast to type AnyType1, if poss *)
  (* ! there are some hardware restrictions ! *)

(******** WARNING ***********************************
  the following two procedures will change to
  value returning functions in the next release
*)
  PROCEDURE SHIFT (VAR b : BITSET, n : INTEGER);
  PROCEDURE ROTATE(VAR b : BITSET; n : INTEGER);
  (* the direction of shifts and rotates is   *)
  (* such that
        b := BITSET{0};
        SHIFT(b,1);
     leaves b = BITSET{1}, rotate is same *)
(******** WARNING ***********************************)

  PROCEDURE CALL(callBody : AnyProcType;
                ... args for body if needed ...
                onError  : Exceptions.RecoveryProc);
  (* semantics are as for Exceptions.Call, except  *)
  (* that parameters may be passed to the callBody *)
  (* and the procedure actuals may be nested procs *)
END SYSTEM.
```

1.29 Terminal

Simple reading and writing character by character, and using strings and cardinals. For many programs this module suffices for all input and output, possibly when used in combination with the redirection facilities of *UNIX*.

The special procedure *GetKeyStroke* reads a single character from the keyboard without waiting for a newline. It performs no echoing. Because this procedure call performs three *ioctl* system calls per character, it uses many more resources than *Read*.

The separate module *StdError* provides similar functionality, but writes to the standard error stream.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula      *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE Terminal;
    IMPORT IMPLEMENTATION FROM "terminal.o";
(*
 *  output always goes to the UNIX std output
 *  input always comes from the UNIX std input
 *
 *  Beware -- attempts to read past end of file will return
 *  character 377C. With 8-bit byte input, EofIn() must be used
 *  to determine if such a result is end of file or a real 377C
 *)

  PROCEDURE WriteString(str : ARRAY OF CHAR);

  PROCEDURE WriteLn;

  PROCEDURE WriteCard(card  : CARDINAL;
                      width : CARDINAL);

  PROCEDURE Write(ch : CHAR);
  PROCEDURE Read(VAR ch : CHAR);

  PROCEDURE EofIn() : BOOLEAN; (* returns "input ended" *)
  PROCEDURE EofOut();          (* closes stdout stream  *)

  PROCEDURE GetKeyStroke(VAR ch : CHAR);
  (* gets char without waiting for <EOL>. Does not echo *)
  (* the char, there is no erase and kill processing.   *)
  (* Restores tty mode after reading each character in  *)

END Terminal.
```

1.30 TextInOut

This library provides exactly the same input and output procedures as the traditional library *InOut*, but in a file oriented fashion.

InOut provides a convenient interface for simple programs which perform input and output to a single input stream and a single output stream. This module allows any number of files to be opened, using em UxFiles, and formatted text input and output to be performed. In every case the procedures here differ from those in *InOut* by having an additional parameter of type *UxFiles.File*. In every case this additional parameter is the first parameter.

```
(*******************************************************************)
(*                                                                 *)
(*            Modula_2 Compiler TextInOut Library Module           *)
(*                                                                 *)
(*          High level input and output procedures for             *)
(*          characters, strings, integers and cardinals            *)
(*                                                                 *)
(*       THIS MODULE PROVIDES THE SAME PROCEDURES AS InOut         *)
(*       BUT USES NAMED FILES AS SOURCE AND DESTINATION            *)
(*                                                                 *)
(*       original module : N. Wirth, PIM-2,  1982 (InOut)          *)
(*       modifications   :                                         *)
(*                                                                 *)
(*******************************************************************)

FOREIGN DEFINITION MODULE TextInOut;

  IMPORT IMPLEMENTATION FROM "textinout.o & uxfiles.o";

  FROM UxFiles IMPORT File;

  CONST

    EOL = 12C;        (* End-of-line character *)

  VAR

     Done : BOOLEAN ;  (* Status of some TextInOut procedure calls.
                          TRUE if the operation was successful,
                          FALSE otherwise. *)

     termCh : CHAR ;   (* Terminating character of some input
                           procedures. ReadString, ReadInt, ReadCard *)
```

... Continued

```
PROCEDURE Read(inFile : File; VAR c:CHAR);
(* Precondition  : TRUE
   Postcondition : Done = FALSE if and only if the end of the primary
                   input stream is reached, otherwise c is the next
                   character in the stream.  *)


PROCEDURE ReadString(inFile : File; VAR s: ARRAY OF CHAR);
(* Precondition  : TRUE
   Postcondition : Inputs a character string from the primary input
                   stream until any character less than or equal to
                   a blank is read. The variable termCh is set to the
                   value of this terminating character.
                   The NUL character (0C) or the end of the array is
                   used to mark the end of the string.
                   Leading blanks and/or tabs are ignored. Excess
                   characters beyond the length of s are discarded. *)


PROCEDURE ReadCard(inFile : File; VAR n: CARDINAL);
(* Precondition  : TRUE
   Postcondition : Done = TRUE if and only if the next sequence of
                   characters on the input stream represents a
                   CARDINAL value. The variable termCh is set to the
                   value of the character that terminates this
                   sequence.  *)


PROCEDURE ReadInt(inFile : File; VAR i : INTEGER);
(* Precondition  : TRUE
   Postcondition : Done = TRUE if and only if the next sequence of
                   characters on the input stream represents a
                   INTEGER value. The variable termCh is set to the
                   value of the character that terminates this
                   sequence.  *)


PROCEDURE Write(outFile : File; c:CHAR);
(* Precondition  : c is defined.
   Postcondition : The character representation corresponding to the
                   value of c is written to the output stream.  *)
```

... *Continued*

```
  PROCEDURE WriteLn(outFile : File);
  (* Precondition  : TRUE
     Postcondition : Equivalent to Write(EOL).  *)


  PROCEDURE WriteString(outFile : File; s : ARRAY OF CHAR);
  (* Precondition  : s is defined.
     Postcondition : Outputs a string of characters until a NUL character
                     or the end of the array is encountered.  *)


  PROCEDURE WriteCard(outFile : File; n: CARDINAL; w: CARDINAL);
  (* Precondition  : n and w are defined.
     Postcondition : The value of n is written to the output stream
                     occupying at least w character positions. Leading
                     blanks fill out the space if it is not all required.
                     The decimal number system is used.  *)


  PROCEDURE WriteInt(outFile : File; i: INTEGER; w: CARDINAL);
  (* Precondition  : i and w are defined.
     Postcondition : The value of i is written to the output stream
                     occupying at least w character positions. Leading
                     blanks fill out the space if it is not all required.
                     The decimal number system is used and a sign is
                     displayed only for negative numbers.  *)


  PROCEDURE WriteOct(outFile : File; n: CARDINAL; w: CARDINAL);
  (* Precondition  : n and w are defined.
     Postcondition : The value of n is written to the output stream
                     occupying at least w character positions. Leading
                     blanks fill out the space if it is not all required.
                     The octal number system is used.  *)


  PROCEDURE WriteHex(outFile : File; n: CARDINAL; w: CARDINAL);
  (* Precondition  : n and w are defined.
     Postcondition : The value of n is written to the output stream
                     occupying at least w character positions. Leading
                     blanks fill out the space if it is not all required.
                     The hexadecimal number system is used.  *)


END TextInOut.
```

1.31 Types

This module provides common names for various numeric types, and provides handy declarations for subranges corresponding to the short types of language C. Remember however, that **gpm** will in any case allocate lesser storage to any subrange which does not require the whole 32 bits used by the basic host types.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

(* !SYSTEM! *) DEFINITION MODULE Types;
(* no implementation exists or is needed for this module *)

   (* signed types *)
   TYPE
     BYTEINT   = INTEGER [-128 .. 127];      (* == char      *)
     SHORTINT  = INTEGER [-32768 .. 32767];  (* == short int *)
     LONGINT   = INTEGER;                    (* == int       *)

   (* cardinal types *)
   TYPE
     BYTECARD  = CARDINAL [0 .. 255];   (* == unsigned char  *)
     SHORTCARD = CARDINAL [0 .. 65535]; (* == unsigned short *)
     LONGCARD  = CARDINAL;              (* == unsigned       *)

   (* alternative names *)
   TYPE
     Int8   = BYTEINT;
     Card8  = BYTECARD;
     Int16  = SHORTINT;
     Card16 = SHORTCARD;

END Types.
```

1.32 UxFiles

Interface to the *UNIX* files facilities. Creating, opening and closing files, plus character and block input and output. This is the basis on which most other input-output modules are built.

A slightly different version of this library exists for gpm-pc. Read the release notes.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE UxFiles;
  IMPORT IMPLEMENTATION FROM "uxfiles.o";

(* This module provides the low level interface to the  *
 * UNIX file system, it links to the library <stdio.h>  *
 * The user programs are protected against the UNIX     *
 * identifiers which are introduced in the header file  *)

FROM SYSTEM   IMPORT ADDRESS, BYTE;

TYPE
  File;
  OpenMode = (ReadOnly, WriteOnly, ReadWrite);

  FilePermissionBits =
        (ox, ow, or, (* others permissions *)
         gx, gw, gr, (* group permissions *)
         ux, uw, ur, (* user permissions *)
         sticky, sgid, suid);
  FileMode =
        SET OF FilePermissionBits;

PROCEDURE GetMode(  name : ARRAY OF CHAR;
                VAR mode : FileMode;
                VAR done : BOOLEAN);
(* precondition  : name must be a nul-terminated variable
                   array,or a literal string.
   postcondition : if done then mode has permission bits *)
```

... Continued

```
PROCEDURE SetMode(  name : ARRAY OF CHAR;
                    mode : FileMode;
                VAR done : BOOLEAN);
(* precondition  : name must be a nul-terminated variable
                   array,or a literal string.
   postcondition : if done then file has permission bits *)

PROCEDURE Open(VAR f:    File;       (* Open an existing file *)
                   name: ARRAY OF CHAR;
                   mode: OpenMode;
               VAR done: BOOLEAN);

PROCEDURE Create(VAR f:    File;     (* Open a new file *)
                     name: ARRAY OF CHAR;
                 VAR done: BOOLEAN);

PROCEDURE Close(VAR f:    File;      (* Close a file *)
                VAR done: BOOLEAN);

PROCEDURE Delete(str : ARRAY OF CHAR;
             VAR ok  : BOOLEAN);

PROCEDURE Reset(f: File);
(* Position the file at the beginning and set to "ReadMode"   *)

PROCEDURE ReadNBytes(    f:              File;
                         buffPtr:        ADDRESS;
                         requestedBytes: CARDINAL;
                     VAR read:           CARDINAL);
  (* Read requested bytes into buffer at address *)
  (* 'buffPtr', number of effectiv read bytes is *)
  (* returned in 'read'                          *)

PROCEDURE WriteNBytes(    f:              File;
                          buffPtr:        ADDRESS;
                          requestedBytes: CARDINAL;
                      VAR written:        CARDINAL);
  (* Write requested bytes into buffer at address    *)
  (* 'buffPtr', number of effectiv written bytes is  *)
  (* returned in 'written'                           *)
```

... *Continued*

```
PROCEDURE ReadByte(    f: File;      (* Read a byte from file *)
                   VAR b: BYTE);

PROCEDURE WriteByte( f: File;     (* Write a byte to file *)
                     b: BYTE);

PROCEDURE EndFile( f : File) : BOOLEAN;
  (* returns true if an attempt has been made
     to read past the physical end of file   *)

PROCEDURE GetPos(   f : File;
                VAR p : CARDINAL);

PROCEDURE SetPos( f : File;
                  p : CARDINAL);
  (* GetPos and SetPos get and set the file position *)

PROCEDURE AccessTime(path     : ARRAY OF CHAR;
                     VAR time : CARDINAL;
                     VAR ok   : BOOLEAN);
  (* finds time of last access to named file *)

PROCEDURE ModifyTime(path     : ARRAY OF CHAR;
                     VAR time : CARDINAL;
                     VAR ok   : BOOLEAN);
  (* finds time of last modification to file *)

PROCEDURE StatusTime(path     : ARRAY OF CHAR;
                     VAR time : CARDINAL;
                     VAR ok   : BOOLEAN);
  (* finds time of last status change of file *)
END UxFiles.
```

1.33 UxHandles

This library provides two different facilities.

Firstly, the library allows file descriptors to be opened and closed, and allows these to be accessed using the stream oriented procedures of *UxFiles* (or even *TextInOut*). The companion module *PipeUtilities* allows stream oriented input/output to pipes.

The *UxHandles* module also allows general redirection of handles. The procedure *RedirectHandle* redirects a handle, but saves the overwritten handle for later restoration. This would be the normal case, when standard handles are redirected. The procedure *RestoreHandle* redirects a handle without saving the overwritten one. It is most often used to restore a previously overwritten handle.

Because redirection of handles is preserved across a *Fork* and *Exec*, it is common to redirect handles in conjunction with the use of the facilities of the module *UxProcesses*.

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula       *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE UxHandles;

  IMPORT IMPLEMENTATION FROM "uxhandles.o";
  FROM UxFiles IMPORT File, OpenMode;

(*
 *  This module complements UxFiles by giving stream access to
 *  handles including standard handles --- stdIn, stdOut, stdErr
 *
 *  BEWARE that streams attached to standard handles stdOut and
 *  stdErr may require a FlushStream or CloseStream call to
 *  force output to appear on interactive terminals
 *
 *  TYPE UxFiles.OpenMode = (ReadOnly, WriteOnly, ReadWrite);
 *)

  TYPE  FileDesc = INTEGER [-1 .. 32767];
  CONST error    = -1;
        stdIn    = 0;
        stdOut   = 1;
        stdErr   = 2;
```

... *Continued*

```
 PROCEDURE OpenFileHandle(path : ARRAY OF CHAR;
                          mode : OpenMode) : FileDesc;
(*
 *  precondition  : path must be a constant string, or be nul
 *                    terminated if a variable array of chars.
 *  postcondition : returns error if cannot be opened, else
 *                    the opened file handle.
 *)

 PROCEDURE CloseHandle(handle : FileDesc;
                       VAR ok : BOOLEAN);
(*
 *  the file descriptor "handle is closed
 *)

 PROCEDURE StreamOfHandle(fd : FileDesc) : File;
 PROCEDURE CloseStream(stream : File; VAR ok : BOOLEAN);
 PROCEDURE FlushStream(stream : File; VAR ok : BOOLEAN);
(*
 *  These procedures open, close and flush a stream on the
 *  nominated file handle, allowing UxFiles procs to be used.
 *  CloseStream is identical to UxFiles.Close
 *  FlushStream is often needed to force output
 *
 *  Example -- to place a stream on a standard handle
 *
 *        stdInFile := StreamOfHandle(stdIn);
 *)

 PROCEDURE RedirectHandle(old   : FileDesc;  (* overwritten *)
                          new   : FileDesc;  (* replacement *)
                       VAR save : FileDesc;  (* copy of old *)
                       VAR done : BOOLEAN);
(*
 *  precondition  : old and new are open file descriptors.
 *  postcondition : input/output sent to "old" will go to the
 *                  stream previously associated with "new"
 *                  The descriptor "new" is closed. The handle
 *                  previously "old" is duplicated in "save"
 *)
```

... Continued

```
  PROCEDURE RestoreHandle (old   : FileDesc;  (* overwritten *)
                          saved : FileDesc;  (* replacement *)
                      VAR done : BOOLEAN);
(*
 *  precondition  : old and new are open file descriptors.
 *  postcondition : input/output sent to "old" will go to the
 *                  stream previously associated with "saved"
 *                  The descriptor "saved" is closed, and the
 *                  previous handle "old" is closed also
 *
 *  Example of typical use -- redirecting standard error
 *
 *        newFD := Open("foo.err", WriteOnly);
 *          -- and check that newFD is not error ...
 *        RedirectHandle(stdErr,newFD,errSav,ok);
 *        (* closes newFD, and does 2> foo.err *)
 *          -- now fork, exec or whatever ...
 *        RestoreHandle(stdErr,errSav,ok);
 *        (* stderr restored, foo.err is closed *)
 *)

END UxHandles.
```

1.34 UxProcesses

Provides packaged access to the *fork, exec, sleep* and *wait* system calls of the *UNIX* environment. Access is also provided to process and user identifiers.

gpm-pc has its own library *PcProcesses*, see the pc release notes.

```
(* ======================================================== *)
(* Preliminary library module for Gardens Point Modula      *)
(* ======================================================== *)

FOREIGN DEFINITION MODULE UxProcesses;
  IMPORT IMPLEMENTATION FROM "uxprocesses.o";
  FROM BuildArgs IMPORT ArgPtr;

  PROCEDURE Fork() : INTEGER;
  (* Spawns a duplicate process. Same as UNIX fork(2).      *)
  (* Returns 0 to child process, and the child's process ID  *)
  (* to the parent process. Returns -1 if fork unsuccessful  *)

  PROCEDURE Exec (path : ARRAY OF CHAR; (* absolute file name *)
                  argv : ArgPtr);       (* the argument array *)
  PROCEDURE Execp(path : ARRAY OF CHAR; (* base of file name  *)
                  argv : ArgPtr);       (* the argument array *)
  (* postcondition : the process designated by path is exec- *)
  (*                 uted in place of the current process.   *)
  (*                 Execp searches for the file on $PATH.   *)
  (*                 Only returns if file cannot be execed.  *)
  (*    Exec  ==>  Equivalent to UNIX execv(2) system call.  *)
  (*    Execp ==>  Equivalent to UNIX execvp(2) system call. *)

  PROCEDURE Wait(VAR result : CARDINAL) : INTEGER;
  (* postcondition : Waits for the termination of a child,   *)
  (*                 or the receipt of a signal. Returns -1  *)
  (*                 for a signal, child ID for termination. *)
  (*                 (result DIV 256) holds 0 for signal, or *)
  (*                    exit code for termination.           *)
  (*                 (result MOD 256) holds signal number,   *)
  (*                    or zero for termination of child.    *)

  PROCEDURE Sleep(time : CARDINAL) : CARDINAL;
  (* postcondition : (time - function-return-value) seconds  *)
  (*                  have elapsed. Returns early (after     *)
  (*                  signal handling) if a signal is caught *)
```

... *Continued*

```
  (* Various procedures for fetching process and parent ids  *)
  PROCEDURE ProcessID() : INTEGER;
  PROCEDURE ProcessGroupID() : INTEGER;
  PROCEDURE ParentProcessID() : INTEGER;

  (* Various procedures for fetching user and group idents   *)
  PROCEDURE UserID() : CARDINAL;
  PROCEDURE GroupID () : CARDINAL;
  PROCEDURE EffectiveUID() : CARDINAL;
  PROCEDURE EffectiveGID() : CARDINAL;

  (* Typical Usage :
      FROM UxProcesses IMPORT Fork, Exec, Wait;
      FROM BuildArgs IMPORT ArgPtr, Arg3;

      VAR result : CARDINAL; childId : INTEGER;
      ...
      IF Fork() = 0 THEN (* do child stuff here *)
      ELSE (* in the parent here *)
        childId := Wait(result);
        Exec("/usr/bin/vi",Arg3("vi","+39","uxproces.def"));
        Error("Couldn't exec vi");
      END;
  *)
END UxProcesses.
```

1.35 The ISO Standard Input/Output Library

The following is the introduction to the Input/Output Library clause of the ISO Standard:

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

```
Input/Output on given channels:
        TextIO          Characters and strings
        WholeIO         Whole numbers as text
        RealIO          REAL numbers as text
        RawIO           Any value as storage locs
        IOResult        Read results

        IOChan          Device-independent interface to channels
        IOConsts        Constants for I/O modules

Device dependent operations (opening and obtaining channels):
        StreamFile      Sequential streams
        SeqFile         Rewindable sequential streams
        RndFile         Random access files
        TermFile        Channels to a terminal
        StdChans        Standard and default channels

        IOLink          Link between channels and new devices
        ChanConsts      Constants for device modules
```

> Channels already open to standard sources and destinations can be identified using procedures provided by the module **StdChans**. This module also provides for the identification and selection of channels used by default for input and output operations.
>
> The modules **TextIO**, **WholeIO**, and **RealIO**, provide facilities that allow the reading and writing of high-level units of data, using text operations on channels specified explicitly by a parameter. These high-level units include characters, strings, and whole numbers and real numbers in decimal notation. The module **RawIO** provides facilities that allow for the reading and writing of arbitrary data types, using raw (binary) operations on explicitly specified channels.
>
> Text operations produce or consume data streams as sequences of characters and line marks. Raw operations produce or consume data streams as sequences of storage locations (ie as arrays whose component type is **SYSTEM.LOC**).

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

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

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

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

The device modules **StreamFile**, **SeqFile**, **RndFile**, and **TermFile** provide facilities that allow a channel to be opened to a named stream, to a rewindable sequential file, to a random access file, or to a terminal device respectively. Device specific operations, such as positioning within a random access file, are also defined in the appropriate device module.

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

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

...

Most users will need only **STextIO**, **SWholeIO** and **SRealIO** to read and write character and numeric data from the standard input (stdin - initially the keyboard, but can be switched to a file by the command line redirection operator "<") and to the standard output (stdout - display, or redirect to a file by ">"), plus **SIOResult** to determine the result of Read operations.

More complex programs will typically use **StreamFile** to open files by name, **TextIO**, **WholeIO** and **RealIO** to read and write those files, and **IOResult** to check read results.

Random access to files will require **RndFile** instead of **StreamFile**; the ability to rewind and sequentially reread or rewrite will require **SeqFile**; interaction specifically with a terminal (including buffering and echo control) will require **TermFile**.

1.36 ChanConsts

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

```
DEFINITION MODULE ChanConsts;
(* Common types and values for device open requests and results *)

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

(* Singleton values of FlagSet, to allow for example, read+write. *)
CONST
  read        = FlagSet{readFlag};
        (* input operations are requested/available *)
  write       = FlagSet{writeFlag};
        (* output operations are requested/available *)
  old         = FlagSet{oldFlag};
        (* a file may/must/did exist before the channel
           was opened *)
  text        = FlagSet{textFlag};
        (* text operations are requested/available *)
```

... *Continued*

```
  raw         = FlagSet{rawFlag};
        (* raw operations are requested/available *)
  interactive = FlagSet{interactiveFlag};
        (* interactive use is requested/available *)
  echo        = FlagSet{echoFlag};
        (* echoing by interactive device on removal of
           characters from input stream requested/applies *)

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

1.37 IOChan

The module **IOChan** defines the type **ChanId** that is used to identify channels, and provides facilities for device-independent access to operations supported by the device to which a channel is connected.

```
DEFINITION MODULE IOChan;
(* Types and procedures forming the interface to channels for
   device-independent data transfer modules *)

FROM   IOConsts  IMPORT ReadResults;
IMPORT IOLink;
FROM   ChanConsts IMPORT FlagSet;
FROM   SYSTEM    IMPORT ADDRESS;


TYPE
  ChanId         = IOLink.ChanId;
  ChanExceptions = IOLink.ChanExceptions;
  DeviceErrNum   = IOLink.DeviceErrNum;


(* There is one pre-defined value identifying an invalid channel on which no
   data transfer operations are available. It is used to initialize
   variables of type ChanId: *)

PROCEDURE InvalidChan () : ChanId;
(* Returns the value identifying the invalid channel *)

(* For each of the following operations, if the device supports the
   operation on the channel, the behaviour of the procedure conforms with the
   description below. The full behaviour is defined for each device module.
   If the device does not support the operation on the channel, the behaviour
   of the procedure is to raise the exception notAvailable. *)

(* Text operations - these perform any required translation between
   the internal and external representation of text. *)
```

... *Continued*

```
PROCEDURE Look (    cid : ChanId;
                VAR ch  : CHAR;
                VAR res : ReadResults);
(* If there is a character as the next item in the given input stream,
   assigns its value to the parameter ch without removing it from the stream.
   Otherwise, the value of the parameter ch is not defined.
   The parameter res, and the stored read result, is set to the value
   allRight, endOfLine, or endOfInput. *)

PROCEDURE Skip (cid : ChanId);
(* If the input has ended, the exception skipAtEnd is raised,
   otherwise, the next character or line mark in the input is removed
   and the stored read result is set to the value allRight. *)

PROCEDURE SkipLook (    cid : ChanId;
                    VAR ch  : CHAR;
                    VAR res : ReadResults);
(* If the stream has ended, the exception skipAtEnd is raised,
   otherwise, the next character or line mark is removed.
   If there is a character as the next item in the given input stream,
   assigned its value to the parameter ch without removing it from the stream.
   Otherwise, the value of the parameter ch is not defined.
   The parameter res, and the stored read result, is set to the value
   allRight, endOfLine or endOfInput. *)

PROCEDURE TextRead (    cid       : ChanId;
                        to        : ADDRESS;
                        maxChars  : CARDINAL;
                    VAR charsRead : CARDINAL);
(* Reads at most maxChars characters from the current line and assigns
   corresponding values to successive locations, starting at the address
   given by the parameter to, and continuing at increments corresponding
   to the address difference between successive components of an ARRAY OF
   CHAR. The number of characters read is assigned to the parameter charsRead.
   The read result is set to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE TextWrite (cid   : ChanId;
                     from  : ADDRESS;
                     charsToWrite : CARDINAL);
(* Writes a number of characters given by the value of the parameter
   charsToWrite,
   starting as the address given by the parameter from and continuing at
   increments corresponding to the address difference between successive
   components of an ARRAY OF CHAR. *)
```

... Continued

```
PROCEDURE WriteLn (cid : ChanId);
(* Writes a line mark over the channel *)



(* Raw operations *)

PROCEDURE RawRead (    cid      : ChanId;
                       to       : ADDRESS;
                       maxLocs  : CARDINAL;
                   VAR locsRead : CARDINAL);
(* Reads at most maxLocs items and assigns corresponding values to
   successive locations, starting at the address given by the parameter
   to.
   The number of items read is assigned to the parameter locsRead.
   The read result is set to the value allRight or endOfInput. *)

PROCEDURE RawWrite (cid  : ChanId;
                    from : ADDRESS;
                    locsToWrite : CARDINAL);
(* Writes a number of items given by the value of the parameter locsToWrite from
   successive locations starting as the address given by the parameter
   from. *)


(* Common operations *)

PROCEDURE GetName (    cid : ChanId;
                   VAR s   : ARRAY OF CHAR);
(* Copies to the parameter s a name associated with the channel,
   possibly truncated depending on the capacity of s. *)

PROCEDURE Reset (cid : ChanId);
(* Reset to a state defined by the device module *)

PROCEDURE Flush (cid : ChanId);
(* Flush any data buffered by the device module out to the destination *)


(* Access to read results *)

PROCEDURE SetReadResult (cid : ChanId;
                         res : ReadResults);
(* Sets the read result value for the channel to the value res *)
```

... Continued

```
PROCEDURE ReadResult (cid : ChanId) : ReadResults;
(* Returns the stored read result value for the channel *)
(*  (This is initially the value notKnown)  *)


(* Users can discover which flags actually apply to a channel *)

PROCEDURE CurrentFlags (cid : ChanId) : FlagSet;
(* Returns the set of flags that apply to the given channel *)

PROCEDURE ChanException () : ChanExceptions;
(* returns the ChanException value for the current context *)

END IOChan.
```

1.38 IOConsts

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

```
DEFINITION MODULE IOConsts;
(* Types and constants for input/output modules *)

(* The following type is used to classify the result of an input operation: *)

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

END IOConsts.
```

1.39 IOLink

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

```
DEFINITION MODULE IOLink;
(* Types and procedures giving the standard implementation of channels: *)

FROM IOConsts  IMPORT ReadResults;
FROM ChanConsts IMPORT FlagSet;
FROM SYSTEM    IMPORT ADDRESS;


TYPE
  ChanId;      (* values of this type are used to identify channels *)


TYPE
  DeviceId;     (* values of this type are used to identify new device
                   modules and are normally obtained by them during their
                   intialization *)

PROCEDURE AllocateDeviceId (VAR did : DeviceId);
(* Allocates a unique value of type DeviceId and assigns this value to
   the parameter did *)

(* a new device module open procedure obtains a channel by calling MakeChan *)

PROCEDURE MakeChan (    did : DeviceId;
                    VAR cid : ChanId);
(* Attempts to make a new channel for the device module identified by did.
   If no more channels can be made, the identity of the invalid channel is
   assigned to cid. Otherwise, the identity of a new channel is assigned to
   cid. *)

PROCEDURE UnMakeChan (    did : DeviceId;
                      VAR cid : ChanId);
(* If the device module identified by the parameter did is not the module that
   made the channel identified by the parameter cid, the exception wrongDevice
   is raised.
   Otherwise, the channel is deallocated and the value identifying the invalid
   channel is assigned to cid. *)
```

... *Continued*

```
PROCEDURE GetStdChanId ( stdCode : INTEGER) : ChanId;
(* If stdCode corresponds to one of the standard channels, then
   returns the ChanId of that channel.
   Otherwise, the ChanId of the invalid channel is returned. *)



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

(* Device modules supply procedures of the following types: *)

TYPE
  LookProc = PROCEDURE (DeviceTablePtr, VAR CHAR, VAR ReadResults);
  SkipProc = PROCEDURE (DeviceTablePtr);
  SkipLookProc = PROCEDURE (DeviceTablePtr, VAR CHAR, VAR ReadResults);
  TextReadProc = PROCEDURE (DeviceTablePtr, ADDRESS, CARDINAL, VAR CARDINAL);
  TextWriteProc = PROCEDURE (DeviceTablePtr, ADDRESS, CARDINAL);
  WriteLnProc = PROCEDURE (DeviceTablePtr);
  RawReadProc = PROCEDURE (DeviceTablePtr, ADDRESS, CARDINAL, VAR CARDINAL);
  RawWriteProc = PROCEDURE (DeviceTablePtr, ADDRESS, CARDINAL);
  GetNameProc = PROCEDURE (DeviceTablePtr, VAR ARRAY OF CHAR);
  ResetProc = PROCEDURE (DeviceTablePtr);
  FlushProc = PROCEDURE (DeviceTablePtr);
  FreeProc = PROCEDURE (DeviceTablePtr);
    (* Carry out the operations involved in closing the corresponding
       channel, including flushing buffers, but do not unmake the channel.
       This procedure is called for each open channel at program
       termination. *)


(* When a device procedure detects a device error, it raises the exception
   softDeviceError or hardDeviceError. If these exceptions are handled, the
   following procedure may be used to discover an implementation-defined error
   number for the channel. *)

TYPE
  DeviceErrNum = INTEGER;


PROCEDURE DeviceError (cid : ChanId) : DeviceErrNum;
(* If a device error exception has been raised, returns the error
   number stored by the device module. *)
```

... Continued

```
TYPE
  DeviceData = ADDRESS;

  DeviceTable = RECORD
                  cd : DeviceData;
                    (* the value NIL *)
                  did : DeviceId;
                    (* the value given in the call MakeChan *)
                  cid : ChanId;
                    (* the identity of the channel *)
                  result : ReadResults;
                    (* the value notKnown *)
                  errNum : DeviceErrNum;
                    (* undefined *)
                  flags : FlagSet;
                    (* FlagSet{} *)
                  doLook : LookProc;
                    (* raise exception notAvailable *)
                  doSkip : SkipProc;
                    (* raise exception notAvailable *)
                  doSkipLook : SkipLookProc;
                    (* raise exception notAvailable *)
                  doTextRead : TextReadProc;
                    (* raise exception notAvailable *)
                  doTextWrite : TextWriteProc;
                    (* raise exception notAvailable *)
                  doLnWrite : WriteLnProc;
                    (* raise exception notAvailable *)
                  doRawRead : RawReadProc;
                    (* raise exception notAvailable *)
                  doRawWrite : RawWriteProc;
                    (* raise exception notAvailable *)
                  doGetName : GetNameProc;
                    (* return the empty string *)
                  doReset : ResetProc;
                    (* do nothing *)
                  doFlush : FlushProc;
                    (* do nothing *)
                  doFree : FreeProc;
                    (* do nothing *)
                END;
```

... Continued

```
PROCEDURE DeviceTablePtrValue (cid : ChanId) : DeviceTablePtr;
(* If the device module identified by the parameter did is not the module
   that made the channel identified by the parameter cid, the exception
   wrongDevice is raised.
   Otherwise, a pointer to the device table for the channel is returned. *)

PROCEDURE AddDeviceTable (dtp : DeviceTablePtr);
(* Includes dtp in record of open channels *)


PROCEDURE IsDevice (cid : ChanId;
                    did : DeviceId) : BOOLEAN;
(* Tests if the device module identified by the parameter did is the
   module that made the channel identified by the parameter cid. *)


(* The following exceptions are defined for this module and its clients *)

TYPE
  ChanExceptions = (
    ChanNoException,
        (* there is no exception in this context *)
    notChanException,
        (* there is an exception in this context, but from another source *)
    wrongDevice,
        (* device specific operations on wrong device *)
    notAvailable,
        (* operation attempted that is not available on that channel *)
    skipAtEnd,
        (* attempt to skip data from a stream that has ended *)
    softDeviceError,
        (* device specific recoverable error *)
    hardDeviceError,
        (* device specific non-recoverable error *)
    textParseError,
        (* input data does not correspond to a character or line mark
           - optional detection *)
    notAChannel
        (* given value does not identify a channel - optional detection *)
  );
```

... *Continued*

```
(* Client modules may raise exceptions: *)

TYPE
  DevExceptionRange = [wrongDevice .. notAChannel];

PROCEDURE RAISEdevException (cid : ChanId;
                             did : DeviceId;
                             x   : DevExceptionRange;
                             s   : ARRAY OF CHAR);
(* If the device module identified by the parameter did is not the module
   that made the channel identified by the parameter cid, the exception
   wrongDevice is raised. Otherwise the given exception is raised and the
   string value of the parameter s is included in the exception message. *)

PROCEDURE IOException () : ChanExceptions;
(* If the current coroutine is in the exceptional execution state because of
   the raising of an exception from ChanExceptions, returns the corresponding
   enumeration value, and otherwise raises an exception.
*)

END IOLink.
```

1.40 IOResult

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

```
DEFINITION MODULE IOResult;
(* Obtain read results on specified channels *)

IMPORT IOConsts;
FROM   IOLink    IMPORT ChanId;

TYPE
  ReadResults = IOConsts.ReadResults;

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

PROCEDURE ReadResult (cid : ChanId) : ReadResults;
(* Returns the result for the last read operation on the channel *)


END IOResult.
```

1.41 RawIO

The module **RawIO** provides facilities that allow for the direct input and output of data using raw operations (ie without any interpretation).

```
DEFINITION MODULE RawIO;
(* Reading and writing data over specified channels using raw operations,
   that is, with no conversion or interpretation. The read result is of the
   type IOConsts.ReadResults.
*)

FROM IOLink    IMPORT ChanId;
FROM SYSTEM    IMPORT LOC;

PROCEDURE Read (    cid      : ChanId;
                VAR to       : ARRAY OF LOC);
(* Reads storage units and assigns to successive components of the
   parameter to.
   The read result is set to the value allRight, wrongFormat, or
   endOfInput *)

PROCEDURE Write (cid     : ChanId;
                 from    : ARRAY OF LOC);
(* Writes storage units from successive components of the
   parameter from *)

END RawIO.
```

1.42 RealIO

The modules **RealIO** and **LongIO** provide facilities that allow for the input and output of real numbers in decimal text form.

```
DEFINITION MODULE RealIO;
(* Input and output of real numbers in decimal text form over specified
   channels.
   The read result is of type IOConsts.ReadResults.
*)

FROM IOLink   IMPORT ChanId;

(* The text form of a signed fixed-point real number is
     ["+"|"-"], decimal digit, {decimal digit}, [".", {decimal digit}] *)

(* The text form of a signed floating-point real number is
     signed fixed-point real number,
     "E", ["+"|"-"], decimal digit, {decimal digit} *)

PROCEDURE ReadReal (    cid  : ChanId;
                    VAR real : REAL);
(* Skips leading spaces and removes any remaining characters
   that form part of a signed fixed or floating point number.
   A corresponding value is assigned to the parameter real.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteFloat (cid     : ChanId;
                      real    : REAL;
                      sigFigs : CARDINAL;
                      width   : CARDINAL);
(* Writes the value of the parameter real in floating-point text form
   with sigFigs significant figures in a field of the given minimum width. *)

PROCEDURE WriteEng (cid     : ChanId;
                    real    : REAL;
                    sigFigs : CARDINAL;
                    width   : CARDINAL);
(* As for WriteFloat except that the number is scaled with one to
   three digits in the whole number part and with an exponent that is
   a multiple of three. *)
```

... *Continued*

```
PROCEDURE WriteFixed (cid   : ChanId;
                      real  : REAL;
                      place : INTEGER;
                      width : CARDINAL);
(* Writes the value of the parameter real in fixed-point text form,
   rounded to the given place relative to the decimal point,
   in a field of the given minimum width. *)

PROCEDURE WriteReal (cid   : ChanId;
                     real  : REAL;
                     width : CARDINAL);
(* Writes the value of real as WriteFixed if the sign and magnitude
   can be shown in the given width, otherwise as WriteFloat.
   The number of places or significant digits depend on the given width. *)


END RealIO.
```

1.43 RndFile

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

```
DEFINITION MODULE RndFile;
(* Random access files *)

IMPORT IOLink;
IMPORT ChanConsts;
FROM   ChanConsts IMPORT ChanFlags;
FROM   SYSTEM IMPORT TSIZE;

TYPE
  ChanId  = IOLink.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;


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

... *Continued*

```
PROCEDURE OpenOld (VAR cid   : ChanId;
                       name  : ARRAY OF CHAR;
                       flags : FlagSet;
                   VAR res   : OpenResults);
(* The old flag is implied;
   without the write flag, read is implied;
   without the text flag, binary is implied.
   If successful, assigns to the parameter cid the identity of a channel
   open to a random access file of the given name and assigns the value
   opened to the parameter res.
   The read and/or write position is at the start of the file.
   If a channel cannot be opened as required, the value of the parameter
   res indicates the reason and cid identifies the invalid channel. *)

PROCEDURE OpenClean (VAR cid   : ChanId;
                         name  : ARRAY OF CHAR;
                         flags : FlagSet;
                     VAR res   : OpenResults);
(* The write flag is implied;
   without the text flag, binary is implied.
   If successful, assigns to the parameter cid the identity of a channel
   open to a random access file of the given name and assigns the value
   opened to the parameter res.
   The file is of zero length.
   If a channel cannot be opened as required, the value of the parameter
   res indicates the reason and cid identifies the invalid channel. *)

PROCEDURE IsRndFile (cid : ChanId) : BOOLEAN;
(* Tests if the channel is open to a random access file *)

TYPE
  RndFileExceptionEnum = (
    rndFileNoException,
        (* there is no exception in this context *)
    notRndFileException,
        (* there is an exception in this context, but from another source *)
    posRange
        (* required new file position cannot be represented as a value
           of type FilePos *)
  );

PROCEDURE IsRndFileException () : RndFileExceptionEnum;
(* returns the RndFileExceptionEnum value for the current context *)
```

... Continued

```
CONST
  FilePosSize = TSIZE(CARDINAL);

TYPE
  FilePos = CARDINAL;  (* ARRAY [1..FilePosSize] OF LOC *)

PROCEDURE StartPos (cid : ChanId) : FilePos;
(* If the channel is not open to a random access file, the exception
   wrongDevice is raised. Otherwise, returns the position of the
   start of the file. *)

PROCEDURE CurrentPos (cid : ChanId) : FilePos;
(* If the channel is not open to a random access file, the exception
   wrongDevice is raised. Otherwise, returns the current
   read/write position. *)

PROCEDURE EndPos (cid : ChanId) : FilePos;
(* If the channel is not open to a random access file, the exception
   wrongDevice is raised. Otherwise, returns the first position at
   or after which there have been no writes. *)

PROCEDURE NewPos (cid       : ChanId;
                  chunks    : INTEGER;
                  chunkSize : CARDINAL;
                  from      : FilePos) : FilePos;
(* If the channel is not open to a random access file, the exception
   wrongDevice is raised. Otherwise, returns the position
   chunks*chunkSize relative to the parameter from or raises the
   exception posRange if the required position cannot be represented
   as a value of type FilePos. *)

PROCEDURE SetPos (cid : ChanId;
                  pos : FilePos);
(* If the channel is not open to a random access file, the exception
   wrongDevice is raised. Otherwise, sets the read/write position to
   the value given by the parameter pos. *)

PROCEDURE Close (VAR cid : ChanId);
(* If the channel is not open to a random access file, the exception
   wrongDevice is raised. Otherwise, the channel is closed and the value
   identifying the invalid channel is assigned to the parameter cid. *)

END RndFile.
```

1.44 SeqFile

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

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

```
DEFINITION MODULE SeqFile;
(* Rewindable sequential files *)

IMPORT IOLink;
IMPORT ChanConsts;
FROM   ChanConsts IMPORT ChanFlags;

TYPE
  ChanId  = IOLink.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;


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

... *Continued*

```
PROCEDURE OpenWrite (VAR cid   : ChanId;
                         name  : ARRAY OF CHAR;
                         flags : FlagSet;
                     VAR res   : OpenResults);
(* The write flag is implied;
   without the binary flag, text is implied.
   If successful, assigns to the parameter cid the identity of a channel
   open to a stored file of the given name and assigns the value
   opened to the parameter res.
   The file is of zero length.
   Output mode is selected.
   If a channel cannot be opened as required, the value of the parameter
   res indicates the reason and cid identifies the invalid channel. *)

PROCEDURE OpenAppend (VAR cid   : ChanId;
                          name  : ARRAY OF CHAR;
                          flags : FlagSet;
                      VAR res   : OpenResults);
(* The write and old flags are implied;
   without the binary flag, text is implied.
   If successful, assigns to the parameter cid the identity of a channel
   open to a stored file of the given name and assigns the value
   opened to the parameter res.
   Output mode is selected and the write position corresponds to the
   length of the file.
   If a channel cannot be opened as required, the value of the parameter
   res indicates the reason and cid identifies the invalid channel. *)

PROCEDURE OpenRead (VAR cid   : ChanId;
                        name  : ARRAY OF CHAR;
                        flags : FlagSet;
                    VAR res   : OpenResults);
(* The read and old flags are implied;
   without the binary flag, text is implied.
   If successful, assigns to the parameter cid the identity of a channel
   open to a stored file of the given name and assigns the value
   opened to the parameter res.
   Input mode is selected.
   If a channel cannot be opened as required, the value of the parameter
   res indicates the reason and cid identifies the invalid channel. *)

PROCEDURE IsSeqFile (cid : ChanId) : BOOLEAN;
(* Tests if the channel is open to a rewindable sequential file *)
```

... *Continued*

```
PROCEDURE Reread (cid : ChanId);
(* If the channel is not open to a rewindable file, the exception
   wrongDevice is raised. Otherwise, if successful, sets the read
   position to the start of the file and selects input mode.
   If the operation cannot be performed, perhaps because of
   insufficient permissions, neither input nor output mode is selected. *)

PROCEDURE Rewrite (cid : ChanId);
(* If the channel is not open to a rewindable file, the exception
   wrongDevice is raised. Otherwise, if successful, truncates the
   file to zero length and selects output mode.
   If the operation cannot be performed, perhaps because of
   insufficient permissions, neither input nor output mode is selected. *)

PROCEDURE Close (VAR cid : ChanId);
(* If the channel is not open to a rewindable file, the exception
   wrongDevice is raised. Otherwise, the channel is closed and the value
   identifying the invalid channel is assigned to the parameter cid. *)

END SeqFile.
```

1.45 SIOResult

The module **SIOResult** corresponds to **IOResult**.

```
DEFINITION MODULE SIOResult;
(* Obtain read results on the default channel *)

IMPORT IOConsts;

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

PROCEDURE ReadResult () : ReadResults;
(* Returns the result for the last read operation on the default
   input channel *)


END SIOResult.
```

1.46 SRawIO

The module **SRawIO** corresponds to **RawIO**.

```
DEFINITION MODULE RawIO;
(* Reading and writing data over specified channels using raw operations,
   that is, with no conversion or interpretation. The read result is of the
   type IOConsts.ReadResults.
*)

FROM IOLink    IMPORT ChanId;
FROM SYSTEM    IMPORT LOC;

PROCEDURE Read (    cid      : ChanId;
                VAR to       : ARRAY OF LOC);
(* Reads storage units and assigns to successive components of the
   parameter to.
   The read result is set to the value allRight, wrongFormat, or
   endOfInput *)

PROCEDURE Write (cid     : ChanId;
                 from    : ARRAY OF LOC);
(* Writes storage units from successive components of the
   parameter from *)

END RawIO.
```

1.47 SRealIO

The module **SRealIO** corresponds to **RealIO**.

```
DEFINITION MODULE SRealIO;
(* Input and output of real numbers in decimal text form over default
   channels.
   The read result is of the type IOConsts.ReadResults.
*)

(* The text form of a signed fixed-point real number is
     ["+"|"-"], decimal digit, {decimal digit}, [".", {decimal digit}] *)

(* The text form of a signed floating-point real number is
     signed fixed-point real number,
     "E", ["+"|"-"], decimal digit, {decimal digit} *)

PROCEDURE ReadReal (VAR real : REAL);
(* Skips leading spaces and removes any remaining characters
   that form part of a signed fixed or floating point number.
   A corresponding value is assigned to the parameter real.
   The real result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteFloat (real    : REAL;
                      sigFigs : CARDINAL;
                      width   : CARDINAL);
(* Writes the value of the parameter real in floating-point text form
   with sigFigs significant figures in a field of the given minimum width. *)

PROCEDURE WriteEng (real    : REAL;
                    sigFigs : CARDINAL;
                    width   : CARDINAL);
(* As for WriteFloat except that the number is scaled with one to
   three digits in the whole number part and with an exponent that is
   a multiple of three. *)

PROCEDURE WriteFixed (real  : REAL;
                      place : INTEGER;
                      width : CARDINAL);
(* Writes the value of the parameter real in fixed-point text form,
   rounded to the given place relative to the decimal point,
   in a field of the given minimum width. *)
```

... *Continued*

```
PROCEDURE WriteReal (real  : REAL;
                     width : CARDINAL);
(* Writes the value of real as WriteFixed if the sign and magnitude
   can be shown in the given width, otherwise as WriteFloat.
   The number of places or significant digits depend on the given width. *)


END SRealIO.
```

1.48 StdChans

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

The module **StdChans** further allows for identification and selection of the channels used by default for input and output operations.

```
DEFINITION MODULE StdChans;
(* Standard and default channels *)

IMPORT IOLink;

TYPE
  ChanId = IOLink.ChanId;
  (* Values of this type are used to identify channels *)

(* The following functions return the standard channel values.
   These channels cannot be closed *)

PROCEDURE StdInChan() : ChanId;
(* Returns a value identifying the implementation-defined standard source
   for program input *)

PROCEDURE StdOutChan() : ChanId;
(* Returns a value identifying the implementation-defined standard
   destination for program output *)

PROCEDURE StdErrChan() : ChanId;
(* Returns a value identifying the implementation-defined standard
   destination for program error messages *)

(* The null device throws away all data written to it and gives an
   immediate end of input indication on reading *)

PROCEDURE NullChan() : ChanId;
(* Returns a value identifying a channel open to the null device *)
```

... *Continued*

```
(* The default channel values *)

PROCEDURE InChan() : ChanId;
(* Returns the identity of the current default input channel,
   as used by input procedures that do not take a channel parameter.
   Initially this is the value returned by the procedure StdInChan. *)

PROCEDURE OutChan() : ChanId;
(* Returns the identity of the current default output channel,
   as used by output procedures that do not take a channel parameter.
   Initially this is the value returned by the procedure StdOutChan. *)

PROCEDURE ErrChan() : ChanId;
(* Returns the identity of the current default error message channel.
   Initially this is the value returned by the procedure StdErrChan. *)

(* The following procedures allow for redirection of the default channels *)

PROCEDURE SetInChan (cid: ChanId);
(* Sets the current default input channel identity to that given by the
   value of the parameter cid. *)

PROCEDURE SetOutChan (cid: ChanId);
(* Sets the current default Output channel identity to that given by the
   value of the parameter cid. *)

PROCEDURE SetErrChan (cid: ChanId);
(* Sets the current default error channel identity to that given by the
   value of the parameter cid. *)


END StdChans.
```

1.49 STextIO

The module **STextIO** corresponds to **TextIO**.

```
DEFINITION MODULE STextIO;
(* Input and output of character and string types over default channels.
   The read result is of the type IOConsts.ReadResults.
*)

(* The following procedures do not read past line marks: *)

PROCEDURE ReadChar (VAR ch : CHAR);
(* If possible, removes a character from the input stream and assigns
   the corresponding value to the parameter ch.
   The read result is set to allRight, endOfLine or endOfInput. *)

PROCEDURE ReadRestLine (VAR s : ARRAY OF CHAR);
(* Removes any remaining characters before the next line mark copying
   as many as can be accommodated to the parameter s as a string value.
   The read result is set to the value allRight, outOfRange, endOfline,
   or endOfInput. *)

PROCEDURE ReadString (VAR s : ARRAY OF CHAR);
(* Removes and copies only those characters before the next line mark
   that can be accommodated to the parameter s as a string value.
   The read result is set to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE ReadToken (VAR s : ARRAY OF CHAR);
(* Skips leading spaces and then removes characters before the next
   space or line mark copying as many as can be accommodated to the
   parameter s as a string value.
   The read result is set to the value allRight, outOfRange, endOfLine,
   or endOfInput. *)

(* The following procedure reads past the next line mark *)

PROCEDURE SkipLine;
(* Removes successive items from the input stream up to and including
   the next line mark or until the end of input is reached.
   The read result is set to the value allRight or endOfInput. *)
```

... *Continued*

```
(* Output procedures *)

PROCEDURE WriteChar (ch : CHAR);
(* Writes the parameter ch to the output stream *)

PROCEDURE WriteLn;
(* Writes a line mark to the output stream *)

PROCEDURE WriteString (s : ARRAY OF CHAR);
(* Writes the string value of the parameter s to the output stream *)

END STextIO.
```

1.50 StreamFile

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

```
DEFINITION MODULE StreamFile;
(* Independent sequential data streams *)

IMPORT IOLink;
IMPORT ChanConsts;
FROM   ChanConsts IMPORT ChanFlags;

TYPE
  ChanId  = IOLink.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;


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

PROCEDURE Open (VAR cid   : ChanId;
                    name  : ARRAY OF CHAR;
                    flags : FlagSet;
                VAR res   : OpenResults);
(* The read flag implies old;
   without the binary flag, text is implied.
   If successful, assigns to the parameter cid the identity of a channel
   open to a source/destination of the given name and assigns the value
   opened to the parameter res.
   If a channel cannot be opened as required, the value of the parameter
   res indicates the reason and cid identifies the invalid channel. *)
```

... Continued

```
PROCEDURE IsStreamFile (cid : ChanId) : BOOLEAN;
(* Tests if the channel is open to a sequential stream *)

PROCEDURE Close (VAR cid : ChanId);
(* If the channel is not open to a sequential stream, the exception
   wrongDevice is raised. Otherwise, the channel is closed and the value
   identifying the invalid channel is assigned to the parameter cid. *)

END StreamFile.
```

1.51 SWholeIO

The module **SWholeIO** corresponds to **WholeIO**.

```
DEFINITION MODULE SWholeIO;
(* Input and output of whole numbers in text form over default channels.
   The read result is of the type IOConsts.ReadResults.
*)

(* The text form of a signed whole number is
     ["+"|"-"], decimal digit, {decimal digit}

   The text form of an unsigned whole number is
        decimal digit, {decimal digit}
*)

PROCEDURE ReadInt (VAR int : INTEGER);
(* Skips leading space and removes any remaining characters
   that form part of a signed whole number.
   A corresponding value is assigned to the parameter int.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteInt (int   : INTEGER;
                    width : CARDINAL);
(* Writes the value of the parameter int in text form in a field of the
   given minimum width. *)

(* The text form of an unsigned whole number is
     decimal digit, {decimal digit} *)

PROCEDURE ReadCard (VAR card : CARDINAL);
(* Skips leading space and removes any remaining characters
   that form part of an unsigned whole number.
   A corresponding value is assigned to the parameter card.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteCard (card  : CARDINAL;
                     width : CARDINAL);
(* Writes the value of the parameter card in text form in a field of the
   given minimum width. *)

END SWholeIO.
```

1.52 TermChan

The module **TermChan** in NOT part of the Standard library. It has been added to provide the 'no-wait' unbuffered input provided by Terminal.GetKeyStroke in the (extensions to the) **PIM-2** library.

```
DEFINITION MODULE TermChan;
(* Extra interface(s) to the terminal channel,
   not included in the Standard *)

FROM   IOConsts  IMPORT ReadResults;
IMPORT IOLink;


TYPE
  ChanId            = IOLink.ChanId;


PROCEDURE LookKey (    cid : ChanId;
                   VAR ok  : BOOLEAN;
   VAR ch  : CHAR;
   VAR res : ReadResults);
(* Performs 'no-wait' input:
   If there is a character available on TermFile channel cid, return ok TRUE
   and the results of a Look(cid/dtp, ch, res);
   otherwise return ok FALSE. *)


PROCEDURE Echo ();
(* Enable echo for all terminal channels.
   Echoing is on by default, and can be controlled by use of
   TextIO.Read (echo) cf IOChan.Look/Skip (no echo).
   However NoEcho turns off echoing globally, and this procedure
   allows it to be turned back on. *)

PROCEDURE NoEcho ();
(* Disable echoing for all terminal channels.
   LookKey disables echo when it checks for (and reads) a character;
   however a typical LookKey loop will often be outside LookKey when a key is
   pressed, and default operating system echo will occur.
   This procedure allows echoing to be globally suppressed, so that LookKey
   echo is entirely under user control. *)

END TermChan.
```

1.53 TermFile

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

```
DEFINITION MODULE TermFile;
(* Channels opened by this module are connected to a single terminal
   device; typed characters are distributed between channels according
   to the sequence of read requests. *)

IMPORT IOLink;
IMPORT ChanConsts;
FROM   ChanConsts IMPORT ChanFlags;

TYPE
  ChanId  = IOLink.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;


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

(* In line mode, items are echoed before being added to the input stream
   and are added a line at a time.
   In single character mode, items are added to the input stream one at a
   time and are echoed as they are removed from the input stream by a
   read operation *)
```

... *Continued*

```
PROCEDURE Open (VAR cid   : ChanId;
                    flags : FlagSet;
                VAR res   : OpenResults);
(* Without the binary flag, text is implied.
   Without the echo flag, line mode is requested, otherwise single character
   mode is requested.
   If successful, assigns to the parameter cid the identity of a channel
   open to the terminal and assigns the value, opened to the parameter res.
   Otherwise, the value of the parameter res indicates the reason for failure
   and cid identifies the invalid channel. *)

PROCEDURE IsTermFile (cid : ChanId) : BOOLEAN;
(* Tests if the channel is open to the terminal *)

PROCEDURE Close (VAR cid : ChanId);
(* If the channel is not open to the terminal, the exception
   wrongDevice is raised. Otherwise, the channel is closed and the value
   identifying the invalid channel is assigned to the parameter cid. *)

END TermFile.
```

1.54 TextIO

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

```
DEFINITION MODULE TextIO;
(* Character and string text operations *)

FROM IOLink   IMPORT ChanId;

(* The following procedures do not read past line marks: *)

PROCEDURE ReadChar (    cid : ChanId;
                    VAR ch  :  CHAR);
(* If possible, removes a character from the input stream and assigns
   the corresponding value to the parameter ch.
   The read result is set to allRight, endOfLine or endOfInput. *)

PROCEDURE ReadRestLine (    cid : ChanId;
                    VAR s   : ARRAY OF CHAR);
(* Removes any remaining characters before the next line mark copying
   as many as can be accommodated to the parameter s as a string value.
   The read result is set to the value allRight, outOfRange, endOfline,
   or endOfInput. *)

PROCEDURE ReadString (    cid : ChanId;
                      VAR s   : ARRAY OF CHAR);
(* Removes and copies only those characters before the next line mark
   that can be accommodated to the parameter s as a string value.
   The read result is set to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE ReadToken (    cid : ChanId;
                     VAR s   : ARRAY OF CHAR);
(* Skips leading spaces and then removes characters before the next
   space or line mark copying as many as can be accommodated to the
   parameter s as a string value.
   The read result is set to the value allRight, outOfRange, endOfLine,
   or endOfInput. *)

(* The following procedure reads past the next line mark *)

PROCEDURE SkipLine (cid : ChanId);
(* Removes successive items from the input stream up to and including
   the next line mark or until the end of input is reached.
   The read result is set to the value allRight or endOfInput. *)
```

... Continued

```
(* Output procedures *)

PROCEDURE WriteChar (cid : ChanId;
                     ch  : CHAR);
(* Writes the parameter ch to the output stream *)

PROCEDURE WriteLn (cid : ChanId);
(* Writes a line mark to the output stream *)

PROCEDURE WriteString (cid : ChanId;
                       s   : ARRAY OF CHAR);
(* Writes the string value of the parameter s to the output stream *)

END TextIO.
```

1.55 WholeIO

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

```
DEFINITION MODULE WholeIO;
(* Input and output of whole numbers in decimal text form over specified
   channels.
   The read result is of the type IOConsts.ReadResults.
*)
FROM IOLink IMPORT ChanId;

(* The text form of a signed whole number is
     ["+"|"-"], decimal digit, {decimal digit}

   The text form of an unsigned whole number is
        decimal digit, {decimal digit}
*)

PROCEDURE ReadInt (    cid : ChanId;
                   VAR int : INTEGER);
(* Skips leading space and removes any remaining characters
   that form part of a signed whole number.
   A corresponding value is assigned to the parameter int.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteInt (cid   : ChanId;
                    int   : INTEGER;
                    width : CARDINAL);
(* Writes the value of the parameter int in text form in a field of the
   given minimum width. *)

PROCEDURE ReadCard (    cid  : ChanId;
                    VAR card : CARDINAL);
(* Skips leading space and removes any remaining characters
   that form part of an unsigned whole number.
   A corresponding value is assigned to the parameter card.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteCard (cid   : ChanId;
                     card  : CARDINAL;
                     width : CARDINAL);
(* Writes the value of the parameter card in text form in a field of the
   given minimum width. *)
END WholeIO.
```