ISO/IEC 10514-1, the standard for Modula-2:

Changes, Clarifications and Additions

M. Schönhacker
Vienna University of Technology
Austria
schoenhacker@eiunix.tuwien.ac.at

C. Pronk
Delft University of Technology
The Netherlands
c.pronk@twi.tudelft.nl

June 27, 1996

1 Introduction

In this article, which is an accompanying article to [2], we will concentrate on the clarification, changes and additions made to the language Modula-2 during the standardization process.

Obviously, changes and new features interact with what is already in the language. We have sacrificed on rationale to present more data on the clarifications and additions.

We will describe the most important changes and clarifications in section 2. Section 3 will describe the extensions made to the language. In appendix A we will give the text of module `SYSTEM` of the new standard.[1] Unfortunately, it is impossible to present all the definition modules in this article. The reader is referred to the standard itself, to the FAQ or to compiler documentation.

2 Clarifications to the language

As has been stated in the accompanying article, the original definitions of the language Modula-2 contained several areas of uncertainty or ambiguity that needed clarification. Please note that this does not necessarily mean that things were *changed*, it just means they were given a well-defined meaning.

There is insufficient room here to mention all the clarifications to the language and system modules, but we would like to draw the reader's attention to an earlier article by Mark Woodman [3] which comprehensively lists these clarifications as well as the numerous changes. Unfortunately, that article does not give many examples and is based upon an earlier version of the Standard. Therefore, some of the more important issues in the standard have been dealt with more extensively below.

2.1 Single pass/Multi pass

The language defined in PIM allows mutually importing local modules, thereby requiring compilers to make at least two scans of the source text to resolve use-before-declaration situations. Some compilers depart from the definition in PIM by enforcing declare-before-use (with the exception

[1] The authors would like to thank ISO for its permission to use parts of the standard in this paper.

of pointers, of course). This simplifies compilers by only requiring a single pass over the source text, but the simplification comes at a cost: the addition of the keyword `FORWARD` to resolve mutual recursion in procedures.

Both approaches have been reconciled in the Standard by allowing both kinds of compilers. Single pass compilers may reject programs that are 'too complex' with respect to multi-pass dependencies, but are required to produce an appropriate error message. Both kinds of compilers are required to process the keyword `FORWARD` correctly.

2.2 Type transfers

Traditionally, two ways of converting values from one type to another (similar) type existed: an unsafe way in which a bit pattern is copied unchanged, and a safe way in which the value is converted to the new type, retaining the semantics. In PIM and in several implementations these two kinds have not always been clearly separated.

In the Standard, a machine dependent and therefore unsafe conversion can be done by using the new function procedure `CAST` which has to be imported from `SYSTEM`.

The new standard function `VAL` allows all safe conversions. Additionally, the functions `FLOAT`, `INT`, `LFLOAT`, `TRUNC`, `ORD` and `CHR` are equivalent with VAL when it is applied to the correct kinds of parameters.

The table below gives an overview of legal combinations. Please note that space restrictions do not allow us to discuss several subtleties regarding subranges, truncation and exceptions.

In the following table, $\surd$ denotes a valid combination of types, and $\times$ denotes an invalid combination; `T` is some enumeration type.

the type of the expression	the type denoted by the type parameter						
	`CARDINAL`*	`INTEGER`*	`REAL`	`LONGREAL`	`CHAR`*	`BOOLEAN`*	the type `T`*
`CARDINAL`	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
`INTEGER`	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$\mathbb{Z}$-type	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
`REAL`	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$
`LONGREAL`	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$
$\mathbb{R}$-type	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$
`CHAR`†	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\times$	$\times$
`BOOLEAN`	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$
the type `T`	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\surd$

* or a subrange of this type

† or string literal type of length 0 or 1

Some examples:

```
FROM SYSTEM IMPORT CAST;
VAR
  i : INTEGER;
  c : CARDINAL;
BEGIN
  i := 5;
  c := CAST(CARDINAL, i);          (* unsafe *)
```

```
  c := VAL (CARDINAL, i);          (* safe *)
  i := VAL (INTEGER,  3.141592);   (* SAFE, truncates *)
  i := VAL (INTEGER,  "C");        (* SAFE, as ORD *)
(* ... *)
```

2.3 Type compatibility

Modula-2 is known to be a strongly typed language. PIM is, however, not always clear on type compatibility. The Standard distinguishes between, and gives precise rules for three forms of compatibility: expression compatibility, assignment compatibility and parameter compatibility. The latter form has variants for value parameters and variable parameters. Where a formal parameter is of a type exported from `SYSTEM` the rules for parameter compatibility are weakened.

2.4 Opaque types

In some implementations where the types `CARDINAL` and pointer types had the same storage requirements, opaque types could be declared as scalar types. The second edition of PIM allowed this, the third edition of PIM only allowed opaque types being declared as pointer types.

The Standard agrees with the latter position, but also allows an opaque type to be defined as another opaque type.

2.5 Characters, Strings and LENGTH

In most implementations the type `CHAR` is implemented using the ASCII character set. The Standard has been purposely formulated to allow other character sets to be used (e.g. EBCDIC, ISO/IEC 10646 [1]).

In the fourth edition of PIM the use of a string terminating character is compulsory. This change from earlier versions of PIM has not been adopted in the Standard. In the Standard there is also no requirement that the string terminating character is equal to `0C` as in PIM.

The Standard has extended the operations available on strings in two ways:

- a predefined function procedure `LENGTH` has been provided,
- A facility for string concatenations of string literals is provided:

  ```
  CONST M2 = "Modula" + 55C +"2 is a Standard now";
  ```

  Please note that `55C` is of type string; concatenation is not defined for the type `CHAR` and not available for string variables.

2.6 CASE statement and ELSE clause

The `CASE` statement has now been defined more precisely. It is now required that each of the values contained in the case label lists be distinct. The same holds for variant field lists.

In PIM, an optional `ELSE` clause has been introduced. However, this was done without specifying precisely what will happen when the value of the case selector is not contained in any of the case label lists, and the `ELSE` clause is missing.

The Standard requires that in such a case an exception will be raised (see section 3.7 of this paper).

3 Extensions to the language

This section will describe some of the additions to PIM. Again, we realize that we cannot be complete and regret not to have included much rationale.

3.1 Modules and libraries

The varied language extensions necessitated a change in the structure of the library modules. Instead of one module SYSTEM and several library modules, the Standard now contains so-called system modules, required library modules and standard library modules. These modules have been specified by presenting a (pseudo-)definition module and a VDM-SL definition.

The section on *system modules* now contains modules called SYSTEM, COROUTINES, TERMINATION, EXCEPTIONS and M2EXCEPTIONS. The module SYSTEM is much like the one in PIM, but has several extensions to facilitate machine-independent low-level programming. Coroutine facilities have been moved from SYSTEM to the system module COROUTINES because it was felt that the concept of coroutines was sufficiently important to be treated separately, as it is done with exceptions and termination. It was also decided to switch back from PROCESS to COROUTINE as the latter was found more correct. The other new system modules will be discussed in other sections of this paper.

The following *required library modules* shall be provided by a conforming implementation: Storage (containing NEW and DISPOSE), LowReal and LowLong (allowing for dependable floating point programming) and CharClass (providing extendable character sets).

The following groups of *standard library modules* may be provided by an implementation: Mathematical Libraries (modules RealMath and LongMath), Input-Output Library (22 modules, see section 3.8), Concurrent Programming Library (modules Processes and Semaphores), Strings, StringConversions and SysClock. In particular the Mathematical Library, the Input-Output Library and the String libraries were developed to counteract the large diversity in facilities offered by current libraries. These standard library modules *may* be provided, meaning that there is no obligation to provide them, but when they are provided they *must* have the same definition modules and the same semantics as defined in the standard.

3.2 Types

The set of types available in the language has been extended by the numeric types LONGREAL, COMPLEX and LONGCOMPLEX (with matching operators). It was decided not to provide the types LONGINTEGER and LONGCARDINAL but to depend on the subrange mechanism instead. Additionally, the types PACKEDSET, PROTECTION, COROUTINE and INTERRUPTSOURCE have been provided, but will not be discussed here.

3.3 Return types of function procedures

It has always been an unsatisfactory restriction that return types of function procedures could not be structured types. Many implementations do allow structured types to be returned, and so does the Standard.

3.4 HIGH

Modula-2 as defined in PIM has so-called open array parameters. This feature allows e.g. writing sort routines without knowing the bounds of the array to be sorted. The lower bound is mapped to 0, the upper bound may be requested at run-time by a call of `HIGH`. According to PIM only one-dimensional arrays can be used as parameter to `HIGH`.

The Standard has extended this facility to multi-dimensional arrays:

```
TYPE colour = (red, yellow, green);

VAR a : ARRAY [-7..-2] OF ARRAY BOOLEAN OF ARRAY colour OF REAL;

PROCEDURE p (f: ARRAY OF ARRAY OF ARRAY OF REAL);
```

For a call of `HIGH` from within the body of `p` that was activated by '`p(a)`':

expression	result
`HIGH(f)`	5
`HIGH(f[0])`	1
`HIGH(f[0,0])`	2
`HIGH(f[0,0,0])`	error

As a matter of clarification: the use of `HIGH` on ordinary arrays is not allowed in the Standard.

3.5 Value constructors

One of the clarifications to the language states that the result type of a function procedure may be an elementary type as well as a structured type. This allows for functions whose result value can directly be assigned to record or array variables. Therefore it seemed only logical to remove the restriction that had so far disallowed the declaration of structured constants and the dynamic construction of values of structured data types.

This facilitates a more consistent style of programming where all the constants can really be declared as such and values of structured types can be constructed in a relatively easy way, thereby preventing the generation of initialization errors which are difficult to trace.

The new features supporting the construction of array values include a 'replicator' called BY, an old keyword in another context. Replication of elements allows for particularly easy initialization of array values:

```
TYPE
  Matrix3D    = ARRAY [1..3,1..3] OF REAL;
  LargeVector = ARRAY [1..1234] OF REAL;

CONST
  EmptyMatrix = Matrix3D {{0.0 BY 3} BY 3};
  UnityMatrix = Matrix3D {{1.0, 0.0 BY 2},
                          {0.0, 1.0, 0.0},
                          {0.0 BY 2, 1.0}};
  ConstVector = LargeVector {1.0, 2.0, 0.0 BY 1231, 3.0};
```

```
VAR
  myMatrix : Matrix3D;
  c1,c2,c3 : REAL;

BEGIN
  (* ... *)
  MyMatrix := Matrix3D {{0.0 BY 3} BY 2, {c1, 0.0, c3}};
  (* ... *)
```

Please note that structured value constructors disallow the creation of anonymous types by always requiring a type identifier, which precludes any ambiguities with regard to the component types involved. It should also be noted that value constructors can be used to compose element values contained in variables, not just constants.

Record values may be constructed by providing an enumeration of the record's elements in the order of declaration:

```
IMPORT SysClock;

TYPE
  DocumentStatus = (NWI, CD, DIS, IS);
  Date =
    RECORD
      year  : CARDINAL;
      month : SysClock.Month;
      day   : SysClock.Day;
    END;
  ISODocument =
    RECORD
      committee : ARRAY [1..48] OF CHAR;
      number    : CARDINAL;
      part      : CARDINAL;
      title     : ARRAY [1..64] OF CHAR;
      status    : DocumentStatus;
      date      : Date;
    END;

VAR
  d   : Date;
  bfd : ISODocument;

BEGIN
  (* ... *)
  d := Date {1996, 6, 1};
  bfd := ISODocument {"ISO/IEC JTC1/SC22/WG13", 10514, 1,
                      "Modula-2", IS, d};
  (* or let´s do it the nested way: *)
  bfd := ISODocument {"ISO/IEC JTC1/SC22/WG13", 10514, 1,
```

```
                        "Modula-2", IS, Date {1996, 6, 1}};
  (* ... *)
```

3.6 Finalization

The original definition of Modula-2 did not provide for a way to execute code *after* a program or library module has been used. However, this was soon found necessary and there were several different implementations around. During standardization, it was decided that there should be a consistent way of handling module termination, or finalization, as it was ultimately called.

It turned out to be difficult to find a way to dynamically register finalization code without running into problems, e.g. with exception handling (see below). Therefore it was decided to use a language extension and thereby provide static registration of finalization code. A module block body may now contain an initialization part (introduced by BEGIN as usual) as well as a finalization part (introduced by the new keyword FINALLY) which will be executed on module finalization:

```
MODULE Client;
  (* ... *)
BEGIN
  ConnectToServer;
  (* ... *)
FINALLY
  DisconnectFromServer;
END Client.
```

In this example, the finalization part is being used to make sure that the connection to a server does not remain open when the program terminates, even if the termination is due to an error.

The standard extends this concept to procedure body finalization. Procedure bodies can therefore also contain a static finalization part which will be executed on finalization of the procedure. It is worth noting that in contrast to the finalization of static modules, procedures and dynamic modules may be finalized more than once (as they may also be called, or initialized, more than once).

3.7 Exceptions

Given that Modula-2 had always been considered a Systems Programming Language, there was considerable pressure from users to provide a standardized way of handling exceptions. It was decided to provide that functionality using a combination of language extension and system modules, in order to keep the change to the language itself as small as possible.

The new keyword EXCEPT introduces an optional part of a module or procedure body that may contain code to handle exceptions. The programmer *may* provide code that tries to eliminate the reason of an exception and then executes the initialization part of the block body again (using the new statement RETRY).

Using the procedure EXCEPTIONS.RAISE, which takes an exception source, an exception number and a message string as parameters, any module may raise its own exceptions (see the longer example below). Because of this way of *optionally* handling exceptions, the standard provides both 'termination' and 'retry' semantics.

As exceptions may occur in the initialization as well as the finalization part of a module or procedure body, both sections may contain individual EXCEPT parts to handle exceptions differently. There is a mechanism that allows the handling of exceptions outside the block body they occur in,

if it does not provide an exceptional execution part itself. For instance, one could write a 'wrapper' procedure that calls another procedure which will cause an exception for certain parameter values, but does not provide an exception handler. In the case of an exception, the exceptional execution part of the 'wrapper' procedure will be called and may identify and handle the exception.

A procedure body using all the facilities provided may look like this:

```
PROCEDURE Exceptional;
BEGIN
  (* initialization code *)
 EXCEPT
  (* exception handler for initialization *)
FINALLY
  (* finalization code *)
 EXCEPT
  (* exception handler for finalization *)
END Exceptional;
```

The system module M2EXCEPTION provides facilities for identifying language exceptions that have been raised (e.g. a range overflow). Another system module EXCEPTIONS allows for the identification of user-defined exceptions, the reporting of their occurrence, and for making inquiries concerning the current state of execution. In this context, coroutines have to be considered very carefully. However, that is beyond the scope of this paper.

For a slightly more substantial example, let us declare a module that is a possible source of user-defined exceptions:

```
DEFINITION MODULE SC22;
TYPE SC22Exceptions = (rejected, cancelled);
  (* Enumeration of the exceptions this module may raise. *)

PROCEDURE IsSC22Exception (): BOOLEAN;
PROCEDURE SC22Exception (): SC22Exceptions;
END SC22.
```

The implementation contains a private variable `source` which is assigned a unique value in the module's initialization body. This value can then be used to globally identify the module as a source of exceptions. Note that this value is assigned dynamically, which allows for independent sources of exceptions.

```
IMPLEMENTATION MODULE SC22;
IMPORT EXCEPTIONS;

VAR source : EXCEPTIONS.ExceptionSource;

PROCEDURE IsSc22Exception (): BOOLEAN;
BEGIN
  RETURN EXCEPTIONS.IsCurrentSource (source);
END IsSC22Exception;
```

```
PROCEDURE SC22Exception (): SC22Exceptions;
BEGIN
  RETURN VAL (SC22Exceptions, EXCEPTIONS.CurrentNumber (source));
END SC22Exception;

(* ... *)
  EXCEPTIONS.RAISE (source, ORD (rejected), "Document rejected");
(* ... *)

BEGIN
  EXCEPTIONS.AllocateSource (source);
END SC22.
```

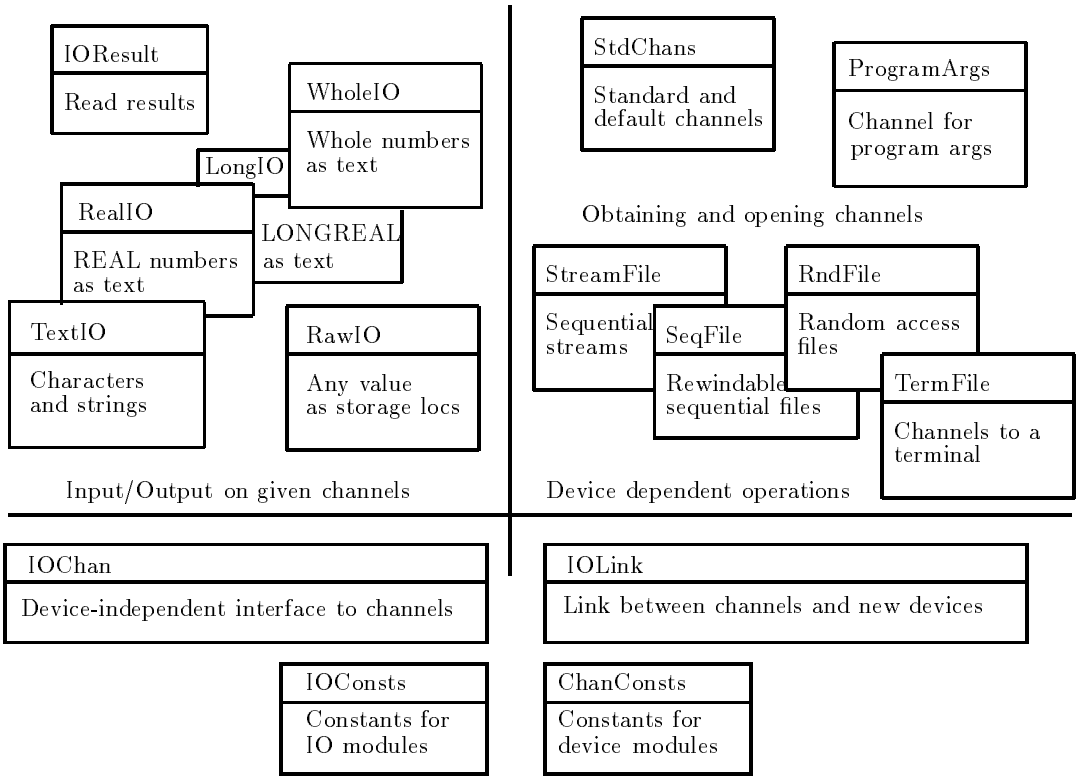
This module could then be used as part of a program that needs to detect various sources of exceptions, e.g.:

```
PROCEDURE Standardize;
BEGIN
  StartProject;
  Work;
  SubmitDocument
 EXCEPT
  IF SC22.IsSC22Exception () THEN
    CASE SC22.SC22Exception () OF
      SC22.rejected  :
        ChangeDocument;
        RETRY;  (* start initialization part again *)
    | SC22.cancelled :
        GiveUp;
        RETURN;  (* leave the procedure *)
    ELSE
      (* ... *)
    END;
  (* ... *)
  END
FINALLY
  PublishDocument
 EXCEPT
  (* ... *)
END Standardize;
```

3.8 I/O Library

In the original reports on Modula-2, some input/output modules had always been assumed to be present, although their 'definition' never got elaborated. Not only did implementors have to guess about various semantic details; the library severely lacked functionality. It was soon decided that a standardized set of library modules for input/output purposes should be devised.

The structure and level of functionality of this library more than once gave rise to heated debates, in particular because the 'perceived subjective complexity' was considered much too high at times. However, it was eventually agreed that the desired level of functionality, flexibility and extensibility required the substantial number of 22 modules to be included in the standard. It should however be noted that the entire I/O Library is optional (see 3.1). The following figure shows the module structure of the I/O library (except for 'simple' operations, see below).



The library allows for reading and writing of data streams over one or more *channels*. Channels may be connected to sources of input data or to destinations of output data, known as *devices* or *device instances*. As can be seen from the figure, there is a separation between modules that form the common base of the library (IOConsts, ChanConsts, IOChan, IOLink), modules concerned with device-dependent operations (StreamFile, SeqFile, RndFile, TermFile) or providing access to standard channels (StdChans, ProgramArgs), modules concerned with device-independent operations (RawIO, TextIO, WholeIO, LongIO, RealIO, IOResult), and modules providing 'simple' I/O, i.e. all the operations provided by the device-independent modules, applied to standard channels which do not need to be named explicitly (SRawIO, STextIO, SWholeIO, SLongIO, SRealIO, SIOResult).

The modules TextIO, WholeIO, LongIO and RealIO provide facilities for reading and writing high-level units of data, using text operations on channels that have to be specified explicitly. This includes characters, strings, as well as whole and real numbers in decimal notation. The module RawIO allows for the reading and writing of arbitrary data types, using raw (binary) operations. The module IOResult provides a facility to determine whether the last operation on a particular channel was successful, or which error it generated.

The modules SRawIO, STextIO, SWholeIO, SLongIO, SRealIO and SIOResult provide the same functionality, except that they do not take parameters identifying a channel. Instead, they operate on

the default input and output channels, as identified by the module StdChans. A practical example is shown in the section on specifying minimal requirements clauses in the accompanying paper.

StdChans provides access to standard and default channels. Standard channels do not have to be opened or closed, and the values used to identify them remain constant throughout the execution of the program. The identification values of default channels initially correspond to those of the standard channels, but may be changed to reflect the effect of input/output redirection.

The standard device modules provided allow channels to be opened to named streams (StreamFile), to rewindable sequential files (SeqFile), to random access files (RndFile) and to terminal devices (TermFile). The module IOChan provides primitive device-independent operations on channels. Furthermore, it defines values for general exceptions that may be raised when using any device through a channel. Additional exceptions related to device-specific operations may be provided by the appropriate device modules.

The module IOLink allows the user to provide additional specialized device modules for use with channels, corresponding to the pattern of the rest of the library.

A Module SYSTEM

```
DEFINITION MODULE SYSTEM;

  (* Gives access to system programming facilities that are probably non portable. *)

  (* The constants and types define underlying properties of storage *)

CONST
  BITSPERLOC    = <implementation-defined constant> ;
  LOCSPERWORD   = <implementation-defined constant> ;

TYPE
  LOC; (* A system basic type. Values are the uninterpreted contents of the smallest
          addressable unit of storage *)
  ADDRESS = POINTER TO LOC;
  WORD = ARRAY [0 .. LOCSPERWORD-1] OF LOC;

  (* BYTE and LOCSPERBYTE are provided if appropriate for machine *)
CONST
  LOCSPERBYTE = <implementation-defined constant> ;

TYPE
  BYTE = ARRAY [0 .. LOCSPERBYTE-1] OF LOC;

PROCEDURE ADDADR (addr: ADDRESS; offset: CARDINAL): ADDRESS;
  (* Returns address given by (addr + offset), or may raise an exception if this
     address is not valid.
  *)

PROCEDURE SUBADR (addr: ADDRESS; offset: CARDINAL): ADDRESS;
  (* Returns address given by (addr - offset), or may raise an exception if this address
     is not valid.
  *)
```

```
PROCEDURE DIFADR (addr1, addr2: ADDRESS): INTEGER;
  (* Returns the difference between addresses (addr1 - addr2), or may raise an exception
     if the arguments are invalid or if the address space is non-contiguous.
  *)

PROCEDURE MAKEADR (val: <some type>; ... ): ADDRESS;
  (* Returns an address constructed from a list of values whose types are
     implementation-defined, or may raise an exception if this address is not valid.
  *)

PROCEDURE ADR (VAR v: <anytype>): ADDRESS;
  (* Returns the address of variable v. *)

PROCEDURE ROTATE (val: <a packedset type>; num: INTEGER): <type of first parameter>;
  (* Returns a bit sequence obtained from val by rotating up or down (left or right) by
     the absolute value of num.  The direction is down if the sign of num is negative,
     otherwise the direction is up.
  *)

PROCEDURE SHIFT (val: <a packedset type>; num: INTEGER): <type of first parameter>;
  (* Returns a bit sequence obtained from val by shifting up or down (left or right) by
     the absolute value of num, introducing zeros as necessary.  The direction is down
     if the sign of num is negative, otherwise the direction is up.
  *)

PROCEDURE CAST (<targettype>; val: <anytype>): <targettype>;
  (* CAST is a type transfer function.  Given the expression denoted by val, it returns
     a value of the type <targettype>.  An invalid value for the target value or a
     physical address alignment problem may raise an exception.
  *)

PROCEDURE TSIZE (<type>; ... ): CARDINAL;
  (* Returns the number of LOCS used to store a value of the specified <type>.   The
     extra parameters, if present, are used to distinguish variants in a variant record.
  *)

END SYSTEM.
```

References

[1] ISO/IEC 10646 Universal Multiple-Octet Coded Character Set(UCS). ISO/IEC JTC1/SC2, 1993.

[2] C. Pronk and M. Schönhacker. ISO/IEC 10514-1, the standard for Modula-2: Process Aspects. *Sigplan Notices, this issue*, 1996.

[3] M. Woodman. A Taste of the Modula-2 Standard. *ACM Sigplan Notices*, 28(9), sept 1993.