ISO/IEC JTC1/SC22/WG13 XXXXX

DATE: 1998-11-30

Replaces Draft from June 1998

ISO/IEC JTC1 SC22/WG13 (Modula-2)

DOC TYPE: Draft (Technical Report)

TITLE: Interfacing Modula-2 to C

SOURCE:

PROJECT: JTC1.22.15436

STATUS:

ACTION ID:

DUE DATE:

DISTRIBUTION:

MEDIUM: server

NO. OF PAGES: 34

Convenor of ISO/IEC JTC1/SC22/WG13
Dr. Martin Schoenhacker
Telephone: +43 1 58801 4079; Facsimile: +43 1 5042583; e-mail: schoenhacker@eiunix.tuwien.ac.at

Project Editor
Eberhard Enger
e-mail: enger@zi.biologie.uni-muenchen.de

Contents

Foreword

[This page to be provided by ISO CS]

Introduction

This Technical Report provides rules to define interfaces for Modula-2 to call C procedures and to access C data entities in a portable way.

This Technical Report has been prepared by ISO/IEC JTC1/SC22/WG13, the technical Working Group for the Modula-2 language.

The motivation for this paper is the need for interfacing important C libraries to Modula-2 programs. A starting point is the binding of the work on Modula-2 bindings to POSIX [3] within JTC1/SC22/WG13 (see document D205 [7]).

1 Scope

This Technical Report gives guidelines for defining Modula-2 interfaces to call C procedures and to access C data objects in a portable way.

This Technical Report is applicable to software engineering in a mixed programming language environment.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. At the time of publication, the edition indicated was valid. Members of IEC and ISO maintain registers of currently valid International Standards and Technical Reports.

ISO/IEC 646 : 1991, *Information Technology -- ISO 7-bit coded character set for information interchange.*

ISO/IEC 9899 : 1990, *Information Technology -- Programming Languages -- C.*

ISO/IEC 10514-1 : 1996, *Information Technology -- Programming Languages -- Modula-2, Base Language.*

3 Definitions

For the purposes of this Technical Report, the terms and definitions given in ISO/IEC 10514-1:1996 (some of which are repeated below for convenience) apply with the following.

TO DO: environment, linkage system, object code

alignment
: A requirement in which objects of a particular data type can be located on storage boundaries with addresses that are particular multiples of a byte address.

implementation
: A particular set of software, running in a particular translation environment under particular control options, that performs translation of programs, and supports execution of functions in a particular execution environment.

implementation defined
: A term applied to a value that is not defined by this Technical Report, but that requires definition before the meaning of a program that relies upon that value can be determined.

4 General

4.1 Purpose and justification

Almost all Modula-2 development systems offer a way to interface other programming languages, especially C. However the solutions offered are very different, and this makes it difficult to write portable Modula-2 programs with an interface to C libraries, in a way that conforms to ISO/IEC 10514-1:1996.

Modula-2 programs using interfaces derived from a set of C header files according to the rules of this Technical Report will be portable between different implementations of the interface to the same C library.

Because of the large amount of C libraries and the continuing commercial importance of C and C++ there is considerable interest in interfacing to the C language.

Benefits are a much more widespread usage of existing C libraries, an easier mixing of language systems (e.g. within educational institutions) and the support of the use of compilation systems based on programming language standards ISO/IEC 9899:1990 and ISO/IEC 10514-1:1996.

4.2 Specific aims

The rules in this Technical Report define the transformation from C header files to Modula-2 pseudo definition modules. These pseudo Modula-2 definition modules are used to describe the interface. It is not intended to define a new binding, but to apply the standards for ISO/IEC 11404, *Language Independent Datatypes* [5], for *Language Independent Procedure Calling* , and for ISO/IEC 10182, *Guidelines for Language Bindings* [4], to define (concrete) rules for interfacing Modula-2 to C.

In addition, restrictions and constraints of these rules are considered.

4.3 Exclusions

This Technical Report does not define the way Modula-2 implementations implement the access to C procedures and C data objects, nor does it describe a tool to generate interfaces to them, but a mapping from C to Modula-2 entities.

4.4 Cooperation and liaison

JTC1/SC22/WG5 (Fortran) is developing an ISO/IEC Technical Report on *Interoperability between Fortran and C* [6]. A cooperation between WG5 and WG13 on the subject of interfacing to C will to be advantageous and first contacts to WG5 have been made to establish a liaison.

4.5 Preparatory work

Since 1992 work has been done within SC22/WG13 on project item JTC 1.22.18.03: *Modula-2 binding to POSIX* . After the discussions of the very first draft of interfacing POSIX, SC22/WG13 proceeded with drafting a C interface to Modula-2, because all libraries on POSIX are written in C. At the Oxford meeting, June 1996, an internal paper *Guidelines for Modula-2 to C Bindings"* [9] was discussed and it was decided that this document should become a replacement for JTC 1.22.18.03.

5 Rationale

Details of a given C library package (such as POSIX [3] , X11 [14]) is found in

- Header files
- Man pages (UNIX)
- Object code library

In most cases such a library depends on the ubiquitous C standard library and the runtime system (kernel). These libraries are mostly written in a mixture of C and assembler language. For interfacing them to Modula-2 programs at least two things are needed:

Definition module(s):
The definition module offers datatypes, constants, variables, and procedures to the application program in a manner that conforms to the Modula-2 standard. This module acts as a connection link to the interfaced library: the library entities have to be interfaced with corresponding Modula-2 entities. A set of rules are given to provide such modules.

Implementation module(s):
The implementation of the correspondend definition modules depends from the compiler and

the system. In some cases no special implementation module is needed. Since standard Modula-2 defines only a small set of facilities within the system modules, compiler and linker extensions for interfacing other libraries are unavoidable. However this should be strictly separated from the definition module (the language) and are hidden to the application program.

NOTE The Standard C programming language used in this Technical Report corresponds to the American National Standards Institute (ANSI) standard for the C language [1], and the identical standard by the International Organization for Standardization (ISO) ISO/IEC 9899:1990. This common standard was developed through the joint efforts of the ANSI-authorized C Programming Language Committee X3J11 and the ISO/IEC committee JTC1/SC22/WG14.

6 Requirements

6.1 Compilation system

The compilation system must be able to support the following features:

- The Modula-2 compiler must be able to build C types.
- Setting of alignment for structures and arrays.
- Setting of enumerations (8-Bit, 16-Bit).
- 8-Bit characters, signed and unsigned characters.
- C calling convention and parameter passing.
- Strings:
  - String literals as pragma
  - char as SS-Type
  - Some mechanism for control characters
- Switching off upperbound for open arrays.
- Handling of functions with variable parameter lists.
- Accessing of external (global) C variables.

Some of them may be offered as pragmas; see proposal in annex B.3

6.2 Required modules

In annex A a template of the definition module *C_Types* is given. An interface has to provide the required module with concrete values for the implementation dependent values.

TO DO: Should further modules for constants and types declared in C standard library *limits.h* and *sys/types.h* be required or recommended? Rick: It seems to be neccessary.

7 Guidelines for interfacing C

7.1 Introduction to the guidelines

The rules below are an attempt for making a simple interface of C definitions within header files to Modula-2. To discuss and test these rules, they will be applied to the POSIX C-bindings. They need to be completed and to be refined.

Notational conventions

The conventions used in this Technical Report are to be interpreted in the same way as in ISO/IEC 10514-1:1996 with the exception that this document does not include VDM-SL (Vienna Development Method Specification Language) at this time.

TO DO: layout convention for guidelines, and for "mappings"

7.2 Lexis

7.2.1 Notation for identifiers

Guideline for notation for identifiers

The *identifiers* for datatypes, constants, and procedures are directly taken and derived from the header files, thus they may have underscores.

NOTE This is not typical Modula-2 style, but it helps the programmer to find the original names within the widespread language C literature and documentation.

7.2.2 Notation of literals

Guideline for notation of literals

A C constant *literal* ending with *L* or *F* denotes *long* or *float* numerical values and is simply mapped to a constant expression.

An escape sequence starting with a backslash character \ is mapped to a character constant. The concatenation operator can be used for adding substrings and single character constants within the C string.

EXAMPLE 1

```
C:                                ==>     Modula-2:

199009L                                   199009
1.40129846e-45F                           1.40129846E-45
```

EXAMPLE 2

```
C:                                ==>     Modula-2:

"\007"                                    07C          (* single char. *)
"\xF"                                     17C
"abcdef ghij\007"                         "abcdef ghi" + 07C
```

7.2.3 Collisions of names

Sometimes the same identifier is used for different enitities. E.g.: Within the C header file *sys/stat.h* we find a definition for a structure named *stat* , and a definition for a function named *stat* :

EXAMPLE 1

```
#define _SYS_STAT_H

struct stat {
          dev_t   st_dev;
          ino_t   st_ino;
          /* ... */
};

extern int stat (const char *name,
                 struct stat *buffer);
```

Guideline for collisions of names

In the case of multiple defined identifiers for different entities within the same scope a short ending is appended to the type identifier in order to get unique Modula-2 names. Table 1 contains a proposal for endings.

Table 1 Endings for identifiers

Type	*Ending*
array type	_arr
pointer type	_ptr
procedure type	_proc
record type	_struct
set type	_set
string type	_str

EXAMPLE 2

The header file *sys/stat.h* from example 1 above then becomes:

```
C:                            ==>     Modula-2:

struct stat {                         TYPE stat_struct = RECORD
  dev_t st_dev; ...                          st_dev : dev_t; ...
};                                         END;

int stat (                            PROCEDURE stat (
       const char *name,                             name : ARRAY OF char;
       struct stat *buffer );                    VAR buffer : stat_struct);
```

TO DO: What about mapping of linker names?

TO DO: Should further endings defined for "type", "variable", "module", "enumeration", and/or others?

7.3 Mapping of datatypes

7.3.1 Introduction

A *datatype* is a collection of distinct values, a collection of properties of those values and a collection of characterizing operations on those values. The representation of those values, relationships and operations may be different between datatypes, as defined in language C and Modula-2. The ideal case is a complete mapping of a cncrete datatype in language C to a corresponding Modula-2 datatype.

Modula-2 is a strong datatype oriented language; C is not. Thus certain definitions for types have to be added or completed, in order to conform to Modula-2; see 7.3.14.

In C language *types* are partitioned into *object types* (types that describe`objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes). An overview of the classification is given by the following list (see also: *Plauger, Brodie: Standard C* [13]).

Classification of C types

- function types
- object types
    - nonscalar types
        - struct types
        - union types
        - array types
    - scalar types
        - arithmetic types
            - integer types
                - char, short, int, long
                - bitfields
                - enumeration types
            - floating-point types
                - float
                - double
                - long double
        - pointer types
            - pointer to function types
            - pointer to object types
            - pointer to incomplete types
- incomplete types
    - struct types
    - union types
    - array types
    - *void*

C types correspond with Modula-2 datatype according to table 2.

Table 2 Correspondance of C and Modula-2 types

C Type	*Modula-2 type*
function	procedure
array	array
struct	record
union	variant record
basic integer	*BOOLEAN, CHAR, INTEGER*
enumeration	enumeration or constants
floating-point	real number
pointer	pointer
pointer to function	procedure type
*void**	*SYSTEM.ADDRESS*

NOTE In language C an *object* is defined as: A region of data storage in the execution environment, the contents of which can represent values. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. When referenced, an object may be interpreted as having a particular type (from ISO/IEC 9899, clause 3).

7.3.2 Representation

TO DO ... The representation of a computational datatype consists of its size in bytes, the ordering, alignment, ...

Guideline for mapping of datatypes

> The mapping has to take into consideration the size of the C datatype. Thus: the representation (size, alignment, access, ...) of the C datatype and its corresponding Modula-2 datatype must be equal.

7.3.3 Basic integer types

An important issue of interfacing C libraries is the type size of the used variable, parameter, component. There are at least two categories, which determine the size of an element. One is the fixation by an external definition, e.g. X11 protocol with names like *CARD8* , *CARD16* , *CARD32* , *INT8* , *INT16* , *INT32* for 1, 2, and 4 Byte long elements etc.. (see: X Interface library *Xlib* , header file *Xmd.h*). The other depends on the implementation and the compilation modes. Thus the most used integer type *int* may be 2 or 4 or 8 Bytes long.

Looking at Modula-2 sources, one can find other naming conventions: *INTEGER16* , *INTEGER32* (Stony Brook Modula-2), or *Int8* , *Int16* (Gardens Point Modula-2), *SHORTINT* and *SHORTCARD* etc. for integer types.

Guideline for basic integer types

> The naming of the basic integer type should reflect these different categories. In order to show their origin -- from a C header file -- types for Modula-2 are chosen with the same name as in C. This makes it easy for the programmer to read both C and Modula-2 source code. These definitions are put into a required module *C_Types* , see annex A.
>
> In those cases where a new type has to be introduced, the X11 style with its clear convention is proposed (*INT8* etc.).

In ANSI-C there are nine basic *integer types* with some restrictions on their representation.

Table 3 shows the counterpart between C and Modula-2 names for *char* -type.

The mapping of *char* and small integer is not easy, because in C header sources a char is sometimes used as an element of a string, and sometimes it is simply a byte. This is why there are different mappings for the same C type.

Table 3 Mapping of *char* -type

C designations	*intended use*	*Modula-2 identifiers*
char	character	char
char	cardinal	unsigned_char
char	integer	signed_char
char	BYTE	LOC | BYTE
signed char	character	char
signed char	integer	signed_char
signed char	BYTE	LOC | BYTE
unsigned char	character	char
unsigned char	cardinal	unsigned_char
unsigned char	BYTE	LOC | BYTE

Table 4 Mapping of *int* -type

C designations	*intended use*	*Modula-2 identifiers*
short	integer	short
signed short	"	short
short int	"	short
signed short int	"	short
unsigned short	cardinal	unsigned_short
unsigned short int	"	unsigned_short
int	integer	int
signed	"	int
signed int	"	int
none	"	int
unsigned int	cardinal	unsigned_int
unsigned	"	unsigned_int
unsigned	set	PACKEDSET
long	integer	long
signed long	"	long
long int	"	long
signed long int	"	long
unsigned long	cardinal	unsigned_long

7.3.4 Enumerations

An enumaration is not actual a distinct type in C (but in C++ enumeration is a distinct type). In opposite to Modula-2 is possible in C to specify explicitly enumeration constant values.

Guideline for enumerations

If an *enumeration* in C is declared without enumeration constants, it can be mapped to a

Modula-2 enumeration type. Otherwise the C enumeration is translated to Modula-2 constants.

EXAMPLE 1

```
   C:                                ==>     Modula-2:

   typedef enum T { c1, c2, ...};           TYPE T = ( c1, c2, ... );
```

EXAMPLE 2

```
   C:                                ==>     Modula-2:

   typedef enum T { c1=1, c2, ...};         TYPE T = C_Types.int;
                                            CONST c1 = VAL(T, 1);
                                                  c2 = VAL(T, 2);
```

7.3.5 Floating-point types

C defines three floating-point types: *float* , *double* , and *long double* . All represent values that are approximations to real values, to some minimum specified precision, over some minimum specifed range. Most current inplementations of floating-point can be expected to conform to the specifications of IEEE 754. Beside of *long double* , which precision and range can be larger than that of *LONGREAL* , a mapping is straightforword.

Guideline for floating-point types

For the three C floating-point types three corresponding Modula-2 types *float* , *double* , and *long_double* are introduced. They are defined in the definition module *C_Types* , see annex A.

If *long double* cannot be represented by *C_Types.long_double* a warning should be issued.

7.3.6 Pointer types

A *pointer type* in C describes a data object whose values are the storage addresses that the program uses to designate functions or data objects of a given type. Some examples from POSIX.1:

EXAMPLE 1

```
   char *getlogin(void);    /* returns a pointer to string */
   extern char **environ;   /* pointer variable to array of strings */
```

Guideline for pointer types

(1) A simple C pointer type is directly mapped to Modula-2 pointer type.

(2) A typical C pointer type defines a pointer to an array, instead of to a single entity. There is an implicit assumption of a continuous storage of the elements (e.g.: array of char).

(3) Pointers to the special incomplete type *void* are mapped to addresses.

EXAMPLE 2

```
   C:                            ==>   Modula-2:

   typedef T1 *T;                TYPE T = POINTER TO T1;
```

(*T* is the pointer type for a single entity of a base type *T1*)
Thus the same C definition as above in example 2 is differently mapped, if *T* points to an array:

EXAMPLE 3

```
C:                             ==>   Modula-2:

typedef T1 *T;                 TYPE T1_arr = ARRAY [0 .. N-1] OF T1;
                                    T = POINTER TO T1_arr;
```

Although the name *T* is the same as in (1), the datatype is very different! *N* is an arbitrary integer value. If it is not known at compile time, *MAX(CARDINAL)* should be used. Of course at runtime the upper limit must be fixed.

EXAMPLE 4

The define directive for *ptr_t* translates to a Modula-2 address type:

```
C:                             ==>   Modula-2:

#define ptr_t void *                 TYPE ptr_t = SYSTEM.ADDRESS;
```

See also 7.6.4 and 7.3.13.

7.3.7 Structure types

A *structure type* describes a data object whose values are composed of *sequences* of *members* .

Guideline for structure types

A *structure type* is mapped to a Modula-2 record type with fixed fields.

The memory image of the complete record type in Modula-2 has to be the same as for the same type in C. It must have the same alignments and sizes for the component types. There may be (additional) fill bytes.

Within a fixed field of a record declaration there must be a type denoter.

EXAMPLE

```
C:                             ==>   Modula-2:

typedef struct S{                   TYPE T = RECORD
          T1 m1;                               m1 : T1;
           ... ;                                  ... ;
} T;                                         END;
```

Annotation: The structure *tag S* can be usually omitted in the Modula-2 record definition

7.3.8 Union types

An *union type* describes a data object whose values are composed of *alternations* of *members* .

Guideline for union types

An *union type* is mapped to variant fields, but tag fields have to be created.

If an union definition occurs within a structure definition the *RECORD END* pair should *not* be omitted.

The memory image of the complete record type in Modula-2 has to be the same as for the same type in C. Each indexable element must have the same size and therefore the same offset.

EXAMPLE

```
C:                           ==>    Modula-2:

typedef union T{                    TYPE T = RECORD
                                                CASE : CARDINAL OF
          T1 f1;                                  0: f1 : T1;
          T2 f2;                                | 1: f2 : T2;
          ... ;                                 | ... ;
                                                END;
 };                                           END;
```

7.3.9 Array types

An *array type* describes a data object whose values are composed of repetitions of *elements* . Identifying individual components is done by indexing.

Guideline for array types

A C *array type* is mapped to a Modula-2 aray type with an appropiate component type.

EXAMPLE 1

Common case:

```
C:                             ==>  Modula-2:

typedef componenttype T[N];         TYPE T = ARRAY [0 .. N-1]
                                               OF componenttype;
```

EXAMPLE 2

Special case: string of characters:

```
C:                             ==>  Modula-2:

typedef unsigned char T[N];         TYPE T = ARRAY [0 .. N-1]
                                               OF C_Types.char;
```

7.3.10 Bitfields

A *bitfield* in C is an integer that occupies a specific number of contiguous bits within a data object that has an integer type. They can be declared only as members of a structure or of an union. The equivalent type in ASN.1 [2] is *BIT STRING* .

Here is an example from *Linux* : in */usr/include/ieee.h* the bits describing the single-precision floating-point format:

EXAMPLE 1

```
union i387_float
  {
    float f;
    struct i387
      {
       unsigned int mantissa:23;
```

```
        unsigned int exponent:8;
        unsigned int negative:1;
       };
   };
```

The bitfields *mantissa* , *exponent* and *negative* are forming one single data object, packed into four bytes. Thus both variants of *f* and *i387* occupies the same storage.

Guideline for bitfields

The *bitfield* construction can not translated to Modula-2 exactly. One possible mapping could be the usage of *packed sets* .

TO DO: The following note needs further discussion (Rick)

NOTE It is recommended to use Modula-2 *set types* , where set operations are used for integer types in C.

EXAMPLE 2

```
   C:                           ==>    Modula-2:

   unsigned int v:N                    v : PACKEDSET OF [0 ... N-1]  ;
```

For example 1 of the floating-point format above, N is 32. Using masking and shifting parts of a *PACKEDSET* variable can be extracted and thus all desired operations can be done.

7.3.11 Opaque types

Guideline for opaque types

TO DO: The rules in this clause have to be corrected! (Ed.)

In Modula-2 opaque types are used to hide the internal datatype representation from the application module(s). In the C language *opaque types* are fully visible, because the concrete data representation is given within the header file. The structure varies from implementation to implementations, of course. Opaque type are normally written in upper case letters. E.g.: *DIR* represents the type of a directory stream, used in header file *dirent.h* . A concrete definition for *DIR* looks like, taken from *Linux* :

EXAMPLE 1

```
   /* The file information header for the Directory stream type.  */
   typedef struct {
     int dd_fd;                   /* file descriptor */
     int dd_loc;                  /* offset in buffer */
     int dd_size;                 /* # of valid entries in buffer */
     struct dirent *dd_buf;       /* -> directory buffer */
   } DIR;                         /* stream data from opendir() */
```

(1) Explicit type definitions, as in the given example *DIR* are directly mapped to Modula-2:

EXAMPLE 2

```
   C:                           ==>    Modula-2:

   typedef struct {                    TYPE T = RECORD
             T1 m1                                m1 : T1;
            ...  ;                                ... ;
   } T;                                         END;
```

(2) Simple opaque structures are mapped to ... Modula-2 types:

EXAMPLE 3

```
C:                              ==>    Modula-2:

typedef struct S *T;                   TYPE T = POINTER TO S;
                                            S = RECORD ... END;
```

(3) For some further resolved issues with a concrete example about the database library *db* from the Berkeley distribution see SC22/WG13 document D228, annex D.4 [9]. (4) The following structured type is mapped to:

EXAMPLE 4

```
C:                              ==>    Modula-2:

typedef struct {                       TYPE DBM = RECORD
          int dummy[10];                            dummy : ARRAY [ 0 .. 9 ]
                                                             OF C_Types.int;
} DBM;                                            END;
```

Although *DBM* might be defined shorter as an array type, both guidelines for structures and arrays are applied for mapping. This mapping is the only way to have the same internal representation of the datatype.

7.3.12 Procedure types

In C a *function type* describes a function whose value is either a data object or an incomplete type rather than an array type. The incomplete type *void* indicates that the function returns no result.

Guideline for procedure types

Mapping:

```
C:                              ==>    Modula-2:

typedef T (*F) ();                     TYPE F = PROCEDURE() : T;

typedef T (*F) (T1, T2, ...);          TYPE F = PROCEDURE( T1, T2, ...) : T;

typedef void (*P) ();                  TYPE P = PROCEDURE;

typedef void (*P) (T1, T2, ...);       TYPE P = PROCEDURE( T1, T2, ...);
```

7.3.13 Incomplete types

An *incomplete type* in C can be a structure type whose whose members have not been specified, an union type whose members have not been specified, an array type whose repetition count is not yet specified, or the type *void* . It is completed by specifying the missing information elsewhere in the source text.

Guideline for incomplete types

When ever possible the incomplete types should be completed before applying the other rules. If not, forward references should be used.

EXAMPLE 1

As an example the definition for the incomplete structure *complex* is shown:

```
struct complex *pc;
```

this type is completed by declaring:

```
struct complex {
        float re, im;
};
```

Now it is defined as a *struct* type with two fields. If there is no declaration given, then suggestive presumptions should be made for the the components.

EXAMPLE 2

The following example for the type definition of an array variable is incomplete:

```
char a[];
```

It is completed by redeclaring the same name later in the same scope with the repetition count specified, as in:

```
char a[25];
```

Before mapping is applied, the incomplete types must be completed. For example 2 above, a constant for the upper bound has to be introduced:

```
C:                            ==>    Modula-2:

typedef componenttype T[N];          TYPE T = ARRAY [0 .. N - 1] OF
                                                componenttype;
```

N is an integer constant expression.

EXAMPLE 3

An empty structure type becomes a pointer type or a pointer to an array depending on the intended usage. If *T1* is a simple type, then:

```
C:                            ==>    Modula-2:

typedef struct T1 * T;                 TYPE T = POINTER TO T1;
```

or, if *T* points to an array or list of *T1* , then:

```
C:                            ==>    Modula-2:

typedef struct T1 * T;                TYPE T1_arr = ARRAY [0 .. N] OF T1;
                                        T = POINTER TO T1_arr;
```

EXAMPLE 4

TO DO: An example for void!

7.3.14 Additional datatypes

Guideline for additional datatypes

Instead of using new types for components of record types additional type-identifiers are introduced.

EXAMPLE

```
C:                           ==>    Modula-2:

typedef struct {                    TYPE datum = RECORD
        char *dptr;                          dptr : C_Types.cstring_ptr;
        int   dsize;                         dsize : C_Types.int;
} datum;                                   END;
```

The pointer type *cstring_ptr* is imported from *C_Types* , see annex A. There is also a string type *cstring* with a maximal possible length defined, thus it can be used within all application packages.

7.4 Mapping of constants

7.4.1 Constant declarations

In C there are different ways of declaring constants: constants using macros, using *const* , and using enums. Here some examples for compile time macro definitions:

EXAMPLE 1

```
#define _ERRNO_H                            /* implicit:  _ERRNO_H = 1 */
#define EPERM      1                        /* operation not permitted */
#define MAXDOUBLE  1.79769313486231570e+308    /* real number */
#define NULL ((void*)0)                     /* null pointer */
#define _IO_MAGIC_MASK 0xFFFF0000           /* hexadecimal number */
```

Guidline for definition of constants

(1) The constant declarations are mapped to Modula-2 constant definitions:

```
C:                           ==>    Modula-2:

#define id value                    CONST id = value;
#define id ( (T)value )             CONST id = VAL( T, value);
#define id ( (T)value )             CONST id = SYSTEM.CAST( T, value);
const T id = value                  CONST id = VAL( T, value);
```

value can be of type numeric, character, and string. The usage of *VAL* or *CAST* for type transfer depends on the C semantics. An example for a mapping of a macro definition with a constant expression:

EXAMPLE 2

```
C:                           ==>    Modula-2:

#define BitsPerCBitset              CONST BitsPerCBitset =
    ( sizeof(unsigned long) * 8)           TSIZE(C_Types.unsigned_long)

typedef unsigned long CBitset       TYPE CBitset = PACKEDSET OF
                                           [0 .. BitsPerCBitset - 1];
```

(2) Within a declaration of C enumeration type one can define enumeration constants with integer values. Thus the mapping of the enumeration may not me contiguous. If the missing values are easy to survey, the Modula-2 definition should introduce dummies. A mapping for an example from *Plauger, Brodie: Standard C (p. 42)* [13] could be look like this:

EXAMPLE nnn

```
C:                          ==>    Modula-2:

enum Hue {black, red, green,       TYPE Hue = (black, red, green,
          blue = 4, magenta};                  hue_3, blue, magenta);
```

(3) Mapping of string constants:

EXAMPLE nnn

```
C:                          ==>    Modula-2:

char m[] = "ABC...";               CONST m = "ABC...";
```

7.4.2 Initialization of structured constants

Guideline for initialization of structured constants

A constant value of an array type, a record type, or a set type can be defined in C in a single declaration. This is mapped to an explicit type declaration and a constant declaration of a value constructor.

EXAMPLE 1

```
C:                          ==>   Modula-2:

struct complex {                  TYPE complex = RECORD
          float re, im;                            re, im : C_Types.float;
} neg_one = {-1, 0};                             END;
                                   CONST neg_one = complex
                                                    { -1.0, 0.0 };
```

EXAMPLE 2

Example for a constant of an *array type* with definitions of datatype identifiers for the array constructor:

```
C:                          ==>   Modula-2:

                                  TYPE names_arr = ARRAY [0 .. 4] OF
                                         ARRAY [0 .. 7] OF C_Types.char;

static char *names[] = {          CONST names = names_arr{
  "skip",                                          "skip",
  "outline",                                       "outline",
  "ra_grid",                                       "ra_grid",
  "dec_grid",                                      "dec_grid",
  "ecliptic"                                       "ecliptic"
  };                                               };
```

7.4.3 Additional constants

Guideline for additional constants

Some important and often used constants should be added. These are given in the definition module *C_Types* , see annex A.

One of the most used constant in C sources is *NULL* . The same identifier *NULL* denotes an empty string, a pointer to an undefined entity, an integer constant with a zero value, and others.

Because in Modula-2 there are only typed constants, new identifiers for the most important forms are introduced: *C_Types.NULL* is a pointer constant, and *C_Types.NULL_str* is a string constant.

7.5 Mapping of variables

Guideline for global variables

The generated Modula-2 definition modules may export global variables. If the Modula-2 linkage system is not able to access global variables, procedure functions for getting and setting values should be introduced.

EXAMPLE

```
C:                              ==>    Modula-2:

extern T v;                            PROCEDURE Get_v() : T;
                                       PROCEDURE Set_v( v : T );
```

Guideline for initialization of variables

The processing of initialization variables defined in a C header file is done during the linkage of the object files. Therefore no general solution can be offered, because this dependents on the used software development environment.

7.6 Mapping of functions

7.6.1 General considerations

TO DO: some introduction

7.6.2 Function headers

Guideline for function headers

Functions in C are mapped to procedure functions or to proper procedures depending solely on the definitions within the C libraries. "Lazy" declarations - like *int xyz()* - in header files, which are not ANSI C conformant should be completed with the real parameter list.

7.6.3 Parameter passing

Guideline for parameter passing

The *parameter passing* conventions should/ have to conform to the same rules as for the interface library.

7.6.4 Formal parameters and types

Guideline for formal parameters and types

Looking at existing header files, e.g. *X/Open SPEC1170* (see [16]), one can find a manyfold of definitions for function parameters. A knowledge about the usage is necessary to give a correct mapping from C to Modula-2. Although within the *X/Open SPEC1170* there are sometimes the same kind of definition for value and variable parameter (e.g. *int addstr (char *str)* and *int getstr(char *str)* .

The mapping depends first on the given types in function header and second on the intended usage.

The C keyword *const* can be used to decide, whether it is a value or variable parameter.

(1) Simple types

Value Parameter:

EXAMPLE nnn

```
   C:                             ==>    Modula-2:

   const T p                             p : T;
   T p                                   p : T;
```

Variable Parameter:

EXAMPLE nnn

```
   C:                             ==>    Modula-2:

   T *p                                   VAR p : T;
```

A formal parameter of type *void* is treated in a special way:

EXAMPLE nnn

```
   C:                             ==>    Modula-2:

   void *p                               p : SYSTEM.ADDRESS
   void **p                              VAR p : SYSTEM.ADDRESS
   const void *p                         p : SYSTEM.ADDRESS
```

Only the referenced values may change, not the reference itself, both for value and for variable parameters, so no *VAR* has to be used. See: procedures *RawRead* and *RawWrite* in the Modula-2 Standard Library module *IOChan* ISO/IEC 10514-1:1996.

(2) Array and *set types*

Value Parameter:

EXAMPLE nnn

```
   C:                             ==>    Modula-2:

   const T1 *p                           p : ARRAY OF T1;
   const T1 *                            p : ARRAY OF T1;

   const char *p                         p : ARRAY OF C_Types.char;
   const char *                          p : ARRAY OF C_Types.char;
```

Variable Parameter:

EXAMPLE nnn

```
C:                             ==>    Modula-2:

T1 *p                                 VAR p : ARRAY OF T1;
```

(3)Structure types

Value Parameter:

EXAMPLE nnn

```
C:                             ==>    Modula-2:

struct T p                            p : T;
T *                                   p : T;
```

Variable Parameter:

EXAMPLE nnn

```
C:                             ==>    Modula-2:

struct T *p                           VAR p : T;
T *p                                  VAR p : T;
```

(4)Types implemented as a pointer

Value Parameter:

EXAMPLE nnn

```
C:                             ==>    Modula-2:

typedef ...  T ... ;                 TYPE T = ... ;
                                           T_ptr = POINTER TO T;

const T *p  /* array */               p : T_ptr;
const struct T *p                     p : T_ptr;
```

In this case a pragma for calling by reference may be needed.

Variable Parameter:

EXAMPLE nnn

```
C:                             ==>    Modula-2:

typedef ... T ... ;                   TYPE T = ... ;
                                           T_ptr = POINTER TO T;

T *p   /* array */                    VAR p : T_ptr;
struct T *p                           VAR p : T_ptr;
```

Looking at the POSIX function *get current working directory* , defined in header file *unistd.h* : The first parameter is used as an in/out parameter: on input a *NULL* value can be given (then an array with the length of *size* bytes will be allocated. Thus this parameter has to be translated to a pointer type. The result type is a pointer type too. On output they contain a directory name or they are *NULL* .

EXAMPLE nnn

```
C:                             ==>    Modula-2:
```

```
char *getcwd                         PROCEDURE getcwd
   ( char *buf,                         ( VAR buf : C_Types.cstring_ptr;
     size_t size );                           size : unistd.size_t )
                                        : C_Types.cstring_ptr;
```

The semantics for *getcwd* allow a *NULL* pointer as a valid input value for the parameter *buf* , thus this parameter has to be a pointer type and not an open array parameter.

NOTE The HIGH parameter of an open array parameter should not be given as a further argument in the parameter list.

7.6.5 Variable argument lists

Guideline for variable argument lists

For mapping of C functions with a variable list of arguments a pragma for C calling convention is necessary. If such a pragma is not offered within a Modula-2 implementation, a lot of C library functions can not be used in Modula-2.

See SC22/WG13 document D228, annex D.1 [9] for resolved issues.

7.6.6 Return values

Guideline for return values

Mapping of *return values* causes much discussions, because most C routines are declared as a function with sometimes tricky result datatype.

There was a long discussion about the nature of the mapping of return values. From the two main alternatives, a direct mapping and a mapping by using proper procedures with an additional variable paramater at the end of the parameter, the direct mapping was chosen. The strongest argument is that this way is closest to the C source. (The DIN group is also voting for function procedures).

See SC22/WG13 document D228, annex D.2 [9] for resolved issues.

EXAMPLE

```
C:                          ==>    Modula-2:

T1 fct ( T2 p2,  ... );            PROCEDURE fct (  p2 : T2; ... )
                                      : T1;
void fct ( T2 p2, ... );           PROCEDURE fct ( p2 : T2; ... );
```

If the result is omitted in the C source, it depends on the intended usage of the result value, whether the result type is, for example *int* or none (proper procedure).

7.6.7 Environment

Guideline for environment

The usage of C libraries in Modula-2 involves a lot of dependencies on the internal behavier of the routines, e.g. implicit allocation of storage. E.g.: In the operating system *Linux* the storage for the function *getcwd* is automatically resized by calling the *malloc* function. This is an implementation dependent behavior for this POSIX version, and may not be true for other UNIX systems. Another example from the X11 library:

EXAMPLE

```
char **XListFonts(display, pattern,
   maxnames, actual_count_return)
```

The client should call function *XFreeFontNames* when finished with the result to free the memory (from: *Scheifler et al.: X Window System* [14]) .

7.7 Processing of directives and macros

7.7.1 General considerations

The processing of macros gives a great power to the C language: including other source files, conditional compilation, redefinitions of already declared constants and functions etc.

The *#define* directive performs a text substitution in the C source file. It has two main uses: simple text replacement and creation for function-like macros.

Some examples for macros from C:

EXAMPLE nnn

```
Macro         ANSI C definition        Traditional C definition
-----         ---- - ----------        ----------- - ----------
PTR           `void *'                 `char *'
LONG_DOUBLE   `long double'            `double'
CONST         `const'                  `'
VOLATILE      `volatile'               `'
SIGNED        `signed'                 `'
PTRCONST      `void *const'            `char *'
```

7.7.2 Simple text replacements

Guideline for macro constants

A macro definition, using the *#define* directive becomes a constant declaration in Modula-2.

Examples from *errno.h* :

EXAMPLE

```
C:                             ==>    Modula-2:

#define EPERM   1                     CONST EPERM = 1;
#define ENOENT  2                     CONST ENOENT = 2;
```

See also guidelines *Definition of Constants* in clause 7.4.

7.7.3 Function-like macros

In C, predefined functions with default values provided for parameters are often used. Some Examples for such macros:

EXAMPLE 1

```
#define SHORTBITS   BITS(short)
#define INTBITS     BITS(int)
```

```
#define S_ISDIR(mode) ((mode) & S_IFMT) == S_IFDIR) /* sys/stat.h */

#define XtUnspecifiedWindow ((Window)2)            /* X11/Intrinsic.h */
#define XtNumber(arr) ((Cardinal) (sizeof(arr) / sizeof(arr[0])))
                                                   /* X11/Intrinsic.h */
#define XtNiconName ((char*)&XtShellStrings[0]     /* X11/Shell.h */
#define ConnectionNumber(dpy) ((dpy)->fd)          /* X11/Xlib.h */
```

TO DO: some further explanations

Guideline for macro functions

There are no general rules for mapping macro functions to Modula-2 constructs, because for each given macro, semantic actions have to be defined within an implementation module.

Example: Given the simple macro function *S_ISDIR* from *sys/stat.h*

EXAMPLE 2

```
C:                            ==>   Modula-2:
                                    definition module:
                                    FROM ... IMPORT st_mode;

#define S_ISDIR(mode)               PROCEDURE S_ISDIR( mode : st_mode )
  ((mode) & S_IFMT) == S_IFDIR)               : int;

                                    implementation module:
                                    FROM ... IMPORT
                                      S_IFMT, S_IFDIR, st_mode;

                                    PROCEDURE S_ISDIR( mode : st_mode )
                                              : int;
                                    BEGIN
                                      IF (mode * S_IFMT) = S_IFDIR THEN
                                        RETURN -1;
                                      ELSE
                                        RETURN 0;
                                      END (* IF *);
                                    END S_ISDIR;
```

7.8 Modularization

Names, circumreference of C header files depends much on history and the state of programming style. Newer libraries, such as X11, are much more structured than the older ones (*stdlib.h*). Some header files have only few definitions.

Recommendation:

The Modula-2 binding should be clearly arranged.

Guideline for module structuring

Recommendation: The set of definition module for a given C library should be layered and should reflect the origin. E.g. for the ANSI C standard library one single definition module could be introduced (*CStdLibrary*) or could be split in corresponding definition modules, the first chararaters of the module names should start with *C_* (e.g.: *C_stdlib* , *C_dirent* , ...) to indicate that it is based on a C library definition.

Annex A

(normative)

Definition Module C_Types

In order to make the identifiers more readable, the full C names are used and concatenated by an underscore (e.g.: C *unsigned long* becomes *unsigned_long*).

In order to have consistent name style within this definition module the type names *cstring* and *cstring_ptr* start with small letters. However *NULL* and *NULL_str* are written in capital letters, as they are used in C sources.

The interface to *C_Types* behaves as if the following were its definition module:

```
DEFINITION MODULE C_Types;

(*
 *  Required module for basic types and constants.
 *)

TYPE
  char = <implementation-defined CHARACTER-TYPE>;
         (*  sizeof(char) = SIZE(char)  *)

  signed_char = <implementation-defined INTEGER-TYPE>;
         (*  sizeof(signed char) = SIZE(signed_char)  *)

  unsigned_char = <implementation-defined CARDINAL-TYPE>;
         (*  sizeof(unsigned char) = SIZE(unsigned_char)  *)

  short = <implementation-defined INTEGER-TYPE>;
         (*  sizeof(short) = SIZE(short)  *)

  unsigned_short = <implementation-defined CARDINAL-TYPE>;
         (*  sizeof(unsigned short) = SIZE(unsigned_short)  *)

  int = <implementation-defined INTEGER-TYPE>;
         (*  sizeof(int) = SIZE(int)  *)

  unsigned_int = <implementation-defined CARDINAL-TYPE>;
         (*  sizeof(unsigned int) = SIZE(unsigned_int)  *)

  long = <implementation-defined INTEGER-TYPE>;
         (*  sizeof(long) = SIZE(long)  *)

  unsigned_long = <implementation-defined CARDINAL-TYPE>;
         (*  sizeof(unsigned long) = SIZE(unsigned_long)  *)

  float = <implementation-defined REAL-TYPE>;
         (*  sizeof(float) = SIZE(float)  *)

  double = <implementation-defined LONGREAL-TYPE>;
         (*  sizeof(double) = SIZE(double)  *)

  long_double = <implementation-defined extented REAL-TYPE>;
         (*  sizeof(long double) = SIZE(long_double)  *)

CONST
  cstring_max = <implementation-defined CARDINAL-constant>;

TYPE
```

```
  cstring = ARRAY [ 0 .. cstring_max ] OF char;  (*  0C terminated  *)
  cstring_ptr = POINTER TO cstring;

CONST
  NULL = SYSTEM.MAKEADR( <implementation-defined ADDRESS constant> );
         (*  must have the C representation of NULL  *)

VAR
  NULL_str [ NULL ] : ARRAY [ 0 .. 0 ] OF char;
                      (*  should be treated as a constant!  *)

END C_Types.
```

Annex B

(informative)

Pragmas

B.1 Introduction

This annex contains a suggested syntax for pragmas. It was first presented by DIN in a report [8]. It includes a proposal for assignment, conditional compilation, predefined identifiers and related stuff; using the directive delimiters <* and *> , see clause 5.5.3, Source Code Directives of the Modula-2 base language standard ISO/IEC 10514-1:1996.

Our main goal for the definition of syntax and semantics for the pragmas is to allow the programmer to use the same pragmas on various compilers and platforms; the goal is to get a warning if the specific compiler can not handle the pragma at all or as it is expected. This requests a standardized syntax and semantics that allows the compiler to detect whether or not it has to know a name used in a pragma. Additionally, standard names have to be provided for pragmas common to (nearly) all compilers.

B.2 Instructions

A set of standard names should control the detection of the Modula-2 standard exceptions defined in the system module *M2EXCEPTION* (ISO/IEC 10514-1:1996, clause 7.5.1):

IndexException , *RangeException* , *CaseSelectException* , *InvalidLocation* , *FunctionException* , *WholeValueException* , *WholeDivException* , *RealValueException* , *RealDivException* , *ComplexValueException* , *ComplexDivException* , *ProtException* , *SysException* , *CoException* , *ExException* .

These names, if known by a compiler, shall represent values defining whether or not code shall be generated for the detection of the specified exception.

The following standard names should also be provided:

StackCheck
  generate stack tests
PragmaCheck
  generate warnings for unrecognized pragmas
ConformantMode

assures that all requirements of ISO/IEC 10514-1:1996 are fulfilled

DebugMode

generate debug information .

The assignment of values to these names is done in the form of assignments with the arguments *TRUE* or *FALSE* .

EXAMPLE 1: generate code for index check:

```
        <*ASSIGN(IndexCheck,TRUE)*>
```

EXAMPLE 2: switch off conformant mode (no warranty!)

```
        <*ASSIGN(ConformantMode,FALSE)*>
```

If the compiler does not know the identifier or cannot provide the desired function (e.g. cannot generate code for a specific check, check is always done by hardware and cannot be turned off) it shall produce a warning.

The more general form of this assignment - especially for non-boolean values - shall be

```
        "ASSIGN" "(" var "," value ")"
```

where *var* is an identifier known to the compiler and *value* something that the compiler can assign to the identifier. This would allow something like

EXAMPLE 3

```
        <*ASSIGN(Optimization,"4")*>
```

given that the compiler knows "Optimization" and can assign the value "4" to this variable.

Alternatively, maybe the compiler knows something like

EXAMPLE 4

```
        <*ASSIGN(Optimization,"Speed")*>
```

If one of these conditions does not hold, the compiler shall produce an error message.

The statement

EXAMPLE 5

```
        var "(" value ")"
```

shall be equivalent to

EXAMPLE 6

```
        "ASSIGN" "(" var "," value ")"
```

Setting of variables from the command line or other parts of the Environment is done by

```
        "ENVIRON" "(" var "," default ")"
```

If an environment, e.g. a commandline, exists and a value is assigned to the variable *var* , then *var* shall be set to this value. Otherwise the contents of *default* shall be used. An implicit *ASSIGN* is done by the compiler for each predefined variable.

Two pragmas (*PUSH* / *POP*) shall allow to save the state of all pragmas in effect on a stack and restore this state later.

If the compiler allows conditional compilation, new variables are defined by

```
        "DEFINE" "(" var "," value ")"
```

The identifier *var* must not be already known (or an error message is produced), that is, one cannot override existing variables (known to the compiler or defined earlier). Once defined, the value can be changed with *ASSIGN* .

The condition pragmas shall have the form:

```
        "IF" boolean_expression "THEN"
        "ELSIF" boolean_expression "THEN"
        "ELSE"
        "END"
```

boolean_expression is either the comparison of an already known variable with a value or the test of a variable with a boolean value. Only equality and inequality are provided for comparison. *NOT* may be used in testing a boolean variable.

Example of a conditional compilation:

EXAMPLE 7

```
        <*DEFINE(CpuType,"Intel")*>

        <*IF CpuType = "Intel" THEN*>
        FROM IntelPrimitives IMPORT Intr, Registers;
        <*ELSIF CpuType = "Motorola" THEN*>
        FROM MotorolaPrimitives IMPORT Trap, Registers;
        <*END*>
```

Some properties of pragmas:

- Pragma sequences within "<*" and "*>" are allowed.
- the pragmas are separated by ";".
- WhiteSpace in pragmas is allowed and skipped.
- Pragmas within comments are not recognized.
- The empty pragma exists.

Collected syntax in EBNF:

```
 pragma              = "<*" single_pragma {";" single_pragma } "*>" .
 single_pragma       = [ assignment | environment | definition | save_restore |
                         condition ] .
 assignment          = known_variable "(" value ")" |
                       "ASSIGN" "(" known_variable "," value ")" .
 environment         = "ENVIRON" "(" unknown_variable "," default_value ")" .
 definition          = "DEFINE" "(" unknown_variable "," value ")" .
 save_restore        = "PUSH" | "POP" .
 condition           = ifpart | elsifpart | elsepart | endifpart .
 ifpart              = "IF" boolean_expression "THEN" .
 elsifpart           = "ELSIF" boolean_expression "THEN" .
 elsepart            = "ELSE" .
 endifpart           = "END" .
 boolean_expression  = [ "NOT" ] known_boolean_variable |
                         value ( "=" | "<>" | "#" ) value .
 known_variable      = identifier .
```

```
unknown_variable    = identifier .
known_boolean_variable = identifier .
identifier          = any legal Modula-2 identifier .
value               = any legal Modula-2 string | "TRUE" | "FALSE" |
                      known_variable .
```

List of predefined identifiers for exceptions:

- *CaseSelectException*
- *CoException*
- *ComplexDivException*
- *ComplexValueException*
- *ExException*
- *FunctionException*
- *IndexException*
- *InvalidLocation*
- *ProtException*
- *RangeException*
- *RealDivException*
- *RealValueException*
- *SysException*
- *WholeDivException*
- *WholeValueExcption*

The following identifiers assumed to be useful (*these names especially are open to discussion*):

- *AssignmentCheck* : check assignments
- *ComplexCheck* : check complex arithmetics
- *ConformantMode* : disable compiler-specific extensions
- *NilCheck* : check dereferencing of *NIL* pointer
- *Optimization* : control optimization
- *RealCheck* : check real arithmetics
- *StackCheck* : check for stack overflow
- *SubscriptCheck* : check array indices
- *VariantCheck* : check variant record tags
- *WholeCheck* : check whole arithmetics

Annotation

- A conditional compilation clause consists of an opening *ifpart* , any number of *elsifpart* s, an optional *elsepart* , and a terminating *endifpart* . Nested conditional compilation clauses are allowed. If the *boolean_expression* of the *ifpart* is true, the text until the next corresponding *elsifpart* , *elsepart* , or *endifpart* is fully interpreted. The remaining text until the corresponding *endifpart* is skipped. Otherwise, the text following the *ifpart* is skipped. If the next condition part is an *elsifpart* , its *boolean_expression* is tested and so on. If no condition holds, the text following the *elsepart* is interpreted. "Skipping" of text means that no interpretation is done except recognition of comments and nested condition clauses. That is, the pragma syntax in the skipped text is checked, though the meaning of the pragmas is not recognized.

EXAMPLE 8

```
        <*IF condition THEN; StackCheck(TRUE)*>
```

is valid.

- Rational:

no number literals are allowed for the sake of simplicity, because no arithmetics are allowed.

B.3 Foreign language bindings

Considering the binding to foreign languages - especially C - we came to the following conclusions which could be seen as a separate report extending this paper and defining a "set of rules" to make the binding portable across different targets and compilers.

The predefined identifier *Foreign* exists, it is found before the *DEFINITION* of the definition module; it's sort of a container for various aspects of the binding. Assigning a value to *Foreign* should imply the following wherever applicable:

- whether an implementation module has to be provided (*Implementation*)
- whether an module initialization part has to be generated (*ModuleInit*)
- whether the upper bound of an array parameter is to be pushed to the stack (*PushHigh*)
- whether RECORDs are pushed as value or as reference (*RecordAsRef*)
- a common rule for the modification of linker names (e.g. prefix every global name with a "_")
- alignments (alignment classes) for types, parameters, records and arrays (*AlignParameter*)
- calling convention (parameter order on the stack, who removes the parameters) (*Reversed*)

For every aspect there should exist a single pragma that allows fine grain control over every aspect of the interface. The use of *Foreign* should be sufficient in all standard cases.

In addition to that, the following pragma shall exist:

```
        "EXTERNAL" "(" identifier "," value ")"
```

This shall have the meaning of replacing the textual representation of the *value* to the Modula-2 identifier *identifier* for the linker, so that the linker is able resolve this reference and the programmer is able to use Modula-2 names for externally defined functions and variables.

If these foreign language extensions are supported the collected syntax changes as follows:

```
single_pragma       = [ assignment | environment | definition | save_restore |
                        condition | external ] .
external            = "EXTERNAL" "(" identifier "," value ")" .
```

To the list of predefined identifiers the following is appended:

- *Foreign*
- *Implementation*
- *ModuleInit*
- *PushHigh*
- *RecordsAsRef*
- *AlignParameter*
- *Reversed*

Annex C

(informative)

Further Information

[To be removed in the final version]

Institutions and Companies, Modula-2 Systems

JTC1/SC22/WG13 FTP server
: Updated versions of several WG13 documents are available by anonymous ftp from ftp://dutiba.twi.tudelft.nl/pub/wg13

Gardens Point Modula-2
: *gardens point modula (gpm)* is an implementation of Modula-2 for 32-Bit platforms since 1989, devopeled by John Gough and co-workers at Queens University of Technology, see URL: http://www.fit.qut.edu.au/CompSci/PLAS/GPM/ . The implementation of *gpm* is based on the ISO standard for Modula-2. The long used macro preprocessor *mpp* [10] provides simple facilities for macros expansion and conditional file extraction. The source is public domain and can be found at the ftp-site: ftp://ftp.fit.qut.edu .

MOCKA
: *MOCKA* (MOdula Compiler KArlsruhe [12]) is the Modula-2 system developed by the GMD research laboratory Karlsruhe, Germany (Gesellschaft für Mathematik und Datenverarbeitung, German national computer science institute). - For more information see URL: http://i44s11.info.uni-karlsruhe.de/~modula/ .

mtc
: *mtc* translates Modula-2 programs into readable C code. It was developed by Matthias Martin and J. Grosch, University of Karlsruhe, Germany; [11]. The source code is public domain.

Stony Brook Modula-2
: Stony Brook compiler is based on ISO Modula-2. An unsupported tool *H2P* is offered, which converts C headers to Turbo Pascal units or Modula-2 definition modules.

xTech
: The tool *H2D* [15] is created by xTech to translate C header files to Modula-2 definition modules. It is available for most popular platforms, including Linux, MS-DOS, OS/2, Windows 95, Windows NT. See URL: http://www.xds.su/xds/xds.html .

Annex D

(informative)

Glossary

The following explanations of terms are given as an aid to the reading of this Technical Report.

POSIX
: Portable Operating System Interface (POSIX)

VDM-SL
: Vienna Development Method Specification Language

Annex E

(informative)

Participating Individuals

This Technical Report was prepared by Eberhard Enger, as project editor on behalf of ISO/IEC JTC1/SC22/WG13. The following individuals participated in the process as the nominated experts ISO/IEC JTC1/SC22/WG13:

Frank Bender	Germany
Eberhard Enger	Germany
John Gough	Australia
Elmar Henne	Germany
Keith Hopper	New Zealand
Herbert Klaeren	Germany
John Lancaster	United Kingdom
David Lightfood	United Kingdom
Cornelius Pronk	Netherlands
Wolfgang Redtenbacher	Germany
Martin Schleußer	Germany
Martin Schönhacker	Austria
Richard J. Sutcliffe	Canada
Albert Wiedemann	Germany
Mark Woodmann	United Kingdom

The assistance of the following individuals is also gratefully acknowledged:

Ulrich Kaiser	Germany
Holger Kleinschmidt	Germany

Annex F

(informative)

Bibliography

[1]
ANSI X3.159-1989, *Information systems -- Programming language -- C* .

[2]
ISO/IEC 8224, *Specification of Abstract Syntax Notation One (ASN.1)* .

[3]
ISO/IEC 9945-1:1990, *Information Technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API)* .

[4]
ISO/IEC TR 10182:1994, *Information Technology -- Guidelines for Language Bindings* .

[5]
ISO/IEC 11404:1996, *Information Technology -- Programming languages, their environments and system software interfaces -- Language-independent datatypes* .

[6]
ISO/IEC JTC1/SC22/WG5 N1277, *Interoperability of Fortran and C. Technical Report defining extensions to ISO/IEC 1539-1:1997* , March 1997 . First Draft PDTR .

[7]
ISO/IEC JTC1/SC22/WG13 D205, *Information Technology -- Programming Language -- Modula-2 Binding: POSIX* . June 1994 (Author: Keith Hopper) .

[8]
ISO/IEC JTC1/SC22/WG13 D224, *A Report on Pragmas* . November 1995 (Author: Bernhard Cygan et al.) .

[9]
ISO/IEC JTC1/SC22/WG13 D228, *Guidelines for Modula-2 to C Bindings (Draft)* . March 1996 (Ed.: Eberhard Enger) .

[10]
N.N., *Gardens Point Modula Release Notes* . *mpp supplement* . Brisbane: Queensland University of Technology, January 1991 .

[11]
Matthias Martin, *Entwurf und Implementierung eines Übersetzers von Modula-2 nach C* . Universität Karlsruhe, February 1990 .

[12]
Helmut Emmelmann, Jürgen Vollmer, *GMD Modula System MOCKA. User Manual* . Universität Karlsruhe, April 1994 .

[13]
P.J. Plauger, Jim Brodie, *Standard C. A Reference* . New Jersey: Prentice Hall PTR, 1996 . ISBN: 0-13-436411-2 (Note: HTML Hypertext on diskette included) .

[14]
Robert W. Scheifler, James Gettys, and Ron Newman, *X WINDOW SYSTEM. C Library and Protocol Reference* . Bredford, Massachusetts: Digital Press, 1988 .

[15]
N.N., *XDS Family of Products. H2D User's Guide* . xTech Ltd., 1996 .

[16]
Spec 1170 Volume 1, *X/Open Snapshot. System Interfaces and Headers. Issue 4, Version 2* . X/Open Company Ltd., April 1994 (Note: issue S418 published solely in electronic form) .

This draft uses the ITSIG exchange DTD (version 0.93, 1998) with small modifications for the table of contents, the title page, and this last page.

[THIS PAGE TO BE REPLACED BY ISO CS]