sun microsystems

Using Modula-2 on the Sun Workstation®

Part Number: 800-1514-10
Revision: A of 20 October 1986
For: Sun Unbundled Software Release

Contents

3

The Modula-2 Language on the Sun Workstation

This chapter describes the implementation of Modula-2 on the Sun workstation, comparing it to the language as defined in *Programming in Modula-2*.[2] In addition, it provides information that may be helpful for interfacing Modula-2 routines with routines written in other languages.

The information in this chapter is implementation-dependent. To write programs that are as portable as possible, encapsulate this information in as few of your modules as possible, and carefully document these modules to assist in later program maintenance.

3.1. Type Coercions

In Modula-2, you can use a type identifier as if it is a function to force the compiler to interpret an expression of one type according to the rules of a different given type. For example, if the expression `(I + J)` normally has the type `INTEGER`, you could use the construct `CARDINAL (I + J)` instead.

Such constructs are *coercions*; they defeat the standard type-checking rules of the language. No conversion of value takes place. Any use of a coercion requires that the original type and the new type have the same size in storage. Most importantly, coercions are implementation-dependent, and may not be portable to other Modula-2 environments.

Where at all possible, use *conversion functions*, such as `ORD`, `CHR`, `FLOAT`, `VAL`, and `TRUNC` instead. These conversion functions are portable, since the compiler translates between the two types, changing the object size if necessary.

3.2. Language Extensions

Here are some features added to this implementation of Modula-2:

Subrange Types
: You may precede subrange types by an optional base type identifier, as in `SHORTINT [10..20]`, in which case the subrange type inherits the size and base type of the identifier.

Built-in ''pervasive'' types
: This Modula-2 implementation includes `SHORTINT` (-32768..32767), `SHORTCARD` (0..65535), and `LONGREAL` (64-bit IEEE double). This implementation also includes `LONGCARD` and `LONGINT`, which are

[2] Niklaus Wirth, Springer-Verlag, 3rd edition, 1985.

synonymous with CARDINAL and INTEGER respectively.

Underscores in identifiers

To make it more convenient to use Modula-2 in conjunction with other languages, the underscore character (_) may appear in identifiers anywhere a letter would be permitted. Note that CCALL and DEFINITION FOR C MODULE automatically add the leading underscore used when referring to an identifier defined in C.

Augmented definition module syntax

The syntax of a definition module has been augmented to allow a special definition module to serve as a ''cover'' for procedures and variables implemented in other languages.

The SIZE *Function*

In accordance with the third edition of Wirth's book, SIZE is now a standard identifier rather than an identifier in SYSTEM. The SIZE function accepts either a variable name or a type identifier as its argument. It does not allow tag-field parameters.

For more information on using the special definition modules, refer to the chapter on using Modula-2 with other languages (Chapter 13).

3.3. The Module SYSTEM

Note: Don't confuse module SYSTEM with System, described in the chapter on accessing a Modula-2 program's environment (Chapter 8).

The module SYSTEM is a special module that the compiler recognizes automatically. It exports a set of implementation- and system-dependent constants, types, and function procedures, which are needed for systems programming. Modules importing from SYSTEM are very system-dependent; to make them portable to other systems, you may have to alter them.

Table 3-1 *Identifiers Exported by* SYSTEM

Constants	*Types*	*Procedures*
BITSFROMLEFT	ADDRESS	ADR
BITSPERWORD	BYTE	CCALL
BYTESFROMLEFT	WORD	TSIZE
BYTESPERWORD		NEWPROCESS
		TRANSFER

Constants Exported by SYSTEM

The constants exported by SYSTEM are:

BYTESPERWORD

Number of bytes in a Modula-2 WORD, with value 4 on a Sun.

BITSPERWORD

Number of bits in a Modula-2 WORD, with value 32 on a Sun.

BYTESFROMLEFT

Order of bytes within a word, with value TRUE on a Sun.

`BITSFROMLEFT`
Order of bits within a byte or word, with value `FALSE` on a Sun.

Types Exported by `SYSTEM`

`ADDRESS`
Byte address of an addressable location. The type `ADDRESS` is compatible with all pointer types and (subranges of) the type `CARDINAL`. It is defined as `POINTER TO WORD`. All integer arithmetic operators apply to this type.

`BYTE`
Representation of a one-byte storage unit. The only operations applicable to variables of type `BYTE` are assignment and equality testing. Any type with a one-byte representation may substitute for the formal parameters of type `BYTE`. You may supply dynamic array parameters of type `ARRAY OF BYTE` with elements of any type.

`WORD`
Representation of a four-byte storage unit (one word). The only operations applicable to variables of type `WORD` are assignment and equality testing. Any type with a four-byte representation may substitute for the formal parameters of type `WORD`. For value parameters, you may also substitute types with representations of less than four bytes. You may supply dynamic array parameters of type `ARRAY OF WORD` with elements of any type represented in at least four bytes. If the size of the supplied parameter is not a multiple of four bytes, Modula-2 ignores the last few bytes.

Function Procedures Exported by `SYSTEM`

`SYSTEM` exports these function procedures:

`ADR(var) : ADDRESS`
Storage address of the given variable.

`CCALL('c-routine', par1, par2, ... ) : INTEGER`
This procedure provides an interface to an arbitrary C procedure.

For more information on `CCALL`, see the chapter on using Modula-2 with other languages (Chapter 13).

Note: The identifier `SIZE`, which was in module `SYSTEM` in the second edition of Wirth's book, is now a standard identifier rather than part of module `SYSTEM`.

`TSIZE(type, tag1Const, tag2Const, ....) : CARDINAL`
Number of words occupied by a variable of the given type. If the type is a record ending in a variant, you may provide the tag constants of the `FieldList`[3] in their proper order. If you do not specify all of the tag constants, Modula-2 assumes the remaining variant of maximum size.

`NEWPROCESS(procedure , workspaceaddr, workspacesize, coroutinevariable)` This procedure initializes a coroutine variable, preparing that coroutine for a `TRANSFER` operation. Coroutine variables are of type `ADDRESS`. They are implemented as pointers to a hidden

[3] See the Modula-2 syntax in Wirth's *Programming in Modula-2*, Springer-Verlag, 3rd edition. 1985.

structure.

`NEWPROCESS` takes as its first parameter the name of a parameterless procedure at the outermost level which executes when the coroutine begins. Under ordinary circumstances, this procedure should be cyclic, and should never return. If it does, Modula-2 emits the message `-- return from coroutine procedure`, followed by a core dump, and the entire program stops.

`NEWPROCESS` takes as its second parameter the address of an area the new coroutine will use as its workspace. The third parameter is the size of this workspace, in bytes (the same units returned by `SIZE` and `TSIZE`). This workspace must be at least 100 bytes in size, and the coroutine may need considerably more space, especially if it calls to many levels or allocates many local variables on the stack (which is inside the workspace). You may have to experiment to determine how much workspace the coroutine actually needs. In the interest of efficient operation, Modula-2 checks for stack overflow only when creating the new coroutine, and not after it begins to run.

The fourth parameter is a variable of type `ADDRESS`, which returns the identity of the newly-created coroutine for later calls to `TRANSFER`. Change any programs that use the type `PROCESS` to use `ADDRESS` instead, for compatibility with the current definition of the language.

`TRANSFER(fromCoroutine, toCoroutine)`

This procedure alters the current flow-of-control by suspending the current coroutine and restarting the execution of another.

Both parameters are `VAR` parameters, so you must pass explicit variables, not expressions. The first parameter receives the saved state of the currently executing coroutine, and the coroutine specified by the second parameter becomes the new current coroutine. No ill effects occur if the same variable is passed to both parameters. In other words, `TRANSFER(XXX, XXX)` means, ''save the current context in variable `XXX`, and restart execution of the suspended coroutine previously held in the variable `XXX`.''

`dbx` cannot ''single-step'' through calls to `TRANSFER`. Breakpoints may be set after each `TRANSFER` instruction in order to trace execution flow.

While `TRANSFER` saves all ordinary state information of the currently executing coroutine, it does not save the state of the floating-point hardware. Because of this, you should avoid using coroutine transfers inside of function procedures (ordinary procedures do no harm), and you must exercise caution when mixing Modula-2 programs with C procedures which declare register floating point variables.

Here is an elementary example using coroutines :

```
MODULE TestProcesses;
 FROM SYSTEM IMPORT TRANSFER, NEWPROCESS, ADDRESS, ADR;
 FROM SimpleIO IMPORT WriteString, WriteLn;

  VAR
    A,B : ADDRESS; (* COROUTINE VARIABLES *)
    I : CARDINAL;
    WorkSpace : ARRAY [0..500] OF CARDINAL;

 PROCEDURE Proc1;
  BEGIN
    LOOP (* forever *)
      WriteString ("Proc1");
      WriteLn;
      TRANSFER (B,A);
     END (* LOOP *)
  END Proc1;

BEGIN
 NEWPROCESS (Proc1, ADR (WorkSpace), SIZE (WorkSpace), B);
 FOR I := 1 TO 3 DO
  WriteString ("Main ");
  TRANSFER (A,B);
  END; (* FOR *)
END TestProcesses.
```

This program will print:

```
Main Proc1
Main Proc1
Main Proc1
```

3.4. Runtime Checks

By default, the compiler generates code to check that array indices, case indices, and values in assignments are within the correct ranges. You can control the generation of range checks by using special comments of the form:

(* $*xc*$ *)

where *x* is either `R` or `T`, and *c* is `+`, `-`, or `=`. The `$` should be the first non-blank character of a (non-nested) comment. The `R` switch controls subrange testing on assignments, while the `T` switch controls array bounds and case value testing. The + character turns a switch on, the – character turns it off, and the = character restores its previous value. The default for both switches is ''on'' but you can override it with the `-norange` and `-nobounds` switches to `m2c`.

The compiler does not check for overflow on 32-bit operations.

Compiler options do not control pointer dereferencing, since system hardware performs this task.

3.5. Implementation Restrictions

This implementation imposes the following restrictions on the Modula-2 language as defined in the *Modula-2 Report*:[4]

Function procedures

The result type of a function procedure must not be an array or a record.

Sets

Sets must consist of elements whose ordinal values are in the range 0 to 31, inclusive. Set constructors are restricted to be constant elements. For example, when `C` is a `CARDINAL` variable, Modula-2 accepts

```
... IF C IN { 1,5 .. 10 } ...
```

but **not**

```
... IF 1 IN { C,5 .. 10 } ...
```

Constant expressions containing real numbers or built-in functions

With the exception of sign inversion, the compiler does not evaluate constant expressions containing real numbers or built-in functions. The compiler generates an error message when it expects a compile-time constant, for example with constant declaration.

`IOTRANSFER`

The procedure `IOTRANSFER` doesn't exist within the Sun Modula-2 implementation because it implies direct interaction with hardware protected from access within the UNIX operating system.

`CASE` *statement code size*

In this implementation, `CASE` statements use a table of 16-bit jump displacements, requiring that the total code size of the statements in the `CASE` not exceed 32,768 bytes. In the rare event that you exceed this limit, convert one or more of the constituent cases into a procedure.

`TRUNC` *returns* `INTEGER` — *conversion* to `CARDINAL`

The standard procedure `TRUNC` accepts either a `REAL` or a `LONGREAL` argument and returns type `INTEGER`, **not** type `CARDINAL`. This is because `REAL` and `LONGREAL` are symmetrical about zero.

[4] Niklaus Wirth, 3rd edition.

When you want to convert into type `CARDINAL`, adjust the value of the floating-point number into `INTEGER` range, then assign the result to a `CARDINAL`. For example:

```
IF RealVar < 0.0 THEN
  HALT; (* error -- cannot be represented as a CARDINAL *)
ELSIF RealVar > FLOAT ( MAX (INTEGER) ) THEN
  RealVar := RealVar - FLOAT (MAX (INTEGER) );
  CardinalVar := TRUNC (RealVar) + MAX (INTEGER) ;
ELSE
  CardinalVar := TRUNC (RealVar);
END;
```

3.6. Data Representation

The Sun Modula-2 implementation closely models the implementation of C. This allows Modula-2 and C routines to run together in the same program. The Modula-2 compiler front end generates intermediate code, acceptable to the code generator shared with the Sun C and Pascal compilers, so simple types have the same representation in all of these languages.

The minimum addressable unit is one byte (eight bits). This is also the allocation unit and the unit used for the sizes of variables (`SIZE`) and types (`TSIZE`). Modula-2 always allocates elements that require more than one byte at even byte addresses. Since Modula-2 allocates variables consecutively according to the declaration sequence, byte-sized gaps may occur between variables or within record types.

Here is a description of the allocated sizes and value ranges of the data types of Modula-2:

`BITSET`
: Defined as `SET OF [0..31]` (see ‘‘*Set types*’’ below).

`CHAR`
: Modula-2 stores variables of type `CHAR` in one byte. The value range extends from `0C` (ordinal value 0) to `377C` (ordinal value 255).

`BOOLEAN`
: Modula-2 stores variables of type `BOOLEAN` in one byte. Their values correspond to an enumeration type with the values `FALSE` (ordinal value 0) and `TRUE` (ordinal value 1).

Enumeration types
: Modula-2 stores variables of enumeration types in one byte, if possible. If the number of constants in an enumeration type exceeds 256, the type requires two bytes. Modula-2 assigns the ordinal values of the enumeration constants according to the declaration sequence, starting with value 0 for the first constant in the list.

`CARDINAL` and `LONGCARD`
: Modula-2 stores variables of types `CARDINAL` and `LONGCARD` as unsigned values in four bytes. The value range extends from 0 to

4,294,967,295.

`INTEGER` and `LONGINT`

Modula-2 stores variables of types `INTEGER` and `LONGINT` as signed values in four bytes. The value range extends from - 2,147,483,648 to 2,147,483,647. Bit 31 is the sign bit.

Subrange types

Modula-2 stores variables of subrange types in the number of bytes (one, two, or four) needed for the (signed or unsigned) representation of the values in the range. For example, the subranges `[0 .. 255]` and `[-128 .. 127]` fit into one byte, while the subrange `[200 .. 300]` requires two bytes. The representation of subrange types declared with an explicit base type (for example `INTEGER[10 .. 20]`) has the same number of bytes as the base type. The standard type `SHORTCARD` is defined as the subrange `[0 .. 65535]` of type `CARDINAL` and the standard type `SHORTINT` is defined as the subrange `[-32768 .. 32767]` of type `INTEGER`. Each of these type requires two bytes.

`REAL`

Modula-2 stores variables of type `REAL` in four bytes in IEEE single-precision floating-point format. Bit 31 is the sign bit. Bits 30..23 are an eight-bit exponent biased by 127. Bits 22..0 are the fraction part of the significand, with an implicit integer part of 1 for normalized numbers and 0 for subnormal numbers, whose exponent is minimal (exponent bits all are 0). Values range in magnitude from the smallest subnormal number, about 1.5E-45, to the largest normalized number, about 3.4E38. Values with maximal exponent (exponent bits all are 1) represent infinity or `NaN`, Not-a-Number. Infinity is the usual result of floating-point overflow; `NaN` the usual result of an invalid operation such as 0.0/0.0.

A consequence of this representation is that Modula-2 can not represent exactly `FLOAT`(x) for (most) cardinal values $x > 16,777,216$ (= 2**24).

`LONGREAL`

Modula-2 stores variables of type `LONGREAL` in eight bytes in IEEE double-precision floating-point format. Bit 63 is the sign bit. Bits 62..52 are an eleven-bit exponent biased by 1023. Bits 51..0 are the fraction part of the significand, with an implicit integer part of 1 for normalized numbers and 0 for subnormal numbers, whose exponent is minimal (exponent bits all are 0). Values range in magnitude from the smallest subnormal number, about 5.0E-324, to the largest normalized number, about 1.7E308. Values with maximal exponent (exponent bits all are 1) represent infinity or `NaN`, Not-a-Number. Infinity is the usual result of floating-point overflow; `NaN` the usual result of an invalid operation such as 0.0/0.0.

Pointer types

Modula-2 stores variables of pointer types in four bytes. Pointers to objects whose sizes are not one byte must always be even. The pointer constant `NIL` has the ordinal value 0.

Set types

A set consists of elements with ordinal values in the range 0 to 31, inclusive. According to the number of set elements, Modula-2 stores variables of set types in either one, two or four bytes. Modula-2 represents the first value of the set range by the rightmost bit of the value. The type `BITSET` is defined as:

```
SET OF [0 .. 31]
```

The value of `BITSET {0}` is the binary value:

```
0000 0000 0000 0000 0000 0000 0000 0001
```

Array types

Arrays are sequences of elements of the same type. If the size of each array element is one byte, Modula-2 allocates the elements in successive bytes. If the element size is larger than one byte, Modula-2 allocates he elements at even addresses. If the resulting size of the array is odd and not one byte, Modula-2 enlarges the array by one byte to make the total size even.

Record types

Modula-2 stores records as contiguous blocks of bytes, with the fields allocated in the sequence in which they are declared. If the size of a field element is larger than one byte, Modula-2 allocates it at an even offset, so unused bytes may occur within a record. If the resulting size of the record is odd and not one byte, Modula-2 enlarges the array by one byte to make the total size even.

Procedure types

Variables of procedure types are stored in four bytes. They contain the entry addresses of the assigned procedures.

Opaque types

Variables of opaque types are stored in four bytes. They may be implemented as pointer types, `INTEGER`s or `CARDINAL`s.

3.7. Parameter Passing

Modula-2 passes parameters in inverse order on the stack. This is compatible with the order used in other languages on Sun Workstations.

Variable parameters

Modula-2 passes the address of the supplied parameter, always passing variable parameters in four bytes. For open arrays, Modula-2 passes the value `HIGH`, in addition to the address, in another four bytes.

Value parameters

Modula-2 passes the value of the supplied parameter. For each parameter, Modula-2 reserves at least four bytes on the stack with one- and two-byte objects right-justified in a field of four bytes. Modula-2 passes objects requiring more than four bytes in an even number of bytes. For open arrays, Modula-2 passes the address of the array, and the called procedure copies the value. In addition to the address, Modula-2 passes the value `HIGH` in another four bytes. When you pass parameters to C routines, you must consider that parameters of type `REAL` and array parameters are treated differently by Modula-2 and C (see `CCALL`).

3.8. Module Initialization

The specifications of Modula-2 dictate that the initialization part, or *body* of every imported module must execute before the body of the importing module. When mutually-referent modules, or chains of modules exist, the order of initialization, according to the *Modula-2 Report*[5] is undefined. Avoid writing programs that depend on this initialization order when circular references among a set of modules is possible.

[5] Niklaus Wirth, *Programming in Modula-2*, 3rd edition, page 169.