Rihtan

Version 1.0

20 Jan 2016

1 Document license

2 Overview

2.1 Program structure and compilation model

The basic unit of program composition is the package. All program entities are contained within packages, which regulate access to the entities that are defined within them. A program consists of a hierarchy of nested packages, which come in three varieties – ordinary packages, subsystems and systems. A program contains exactly one system package, which logically contains all other packages (including zero or more subsystems). Subsystems represent independent sequences of program flow. The system itself represents the flow of the main program.

```
system Radio_transponder is

  // The nature of the target is defined by pragmas. Typically, these are
  // collected into a package. Here the package is stored in a separate
```

```
  // file but appears logically at this point in the system so that
  // its definitions are available for the rest of the compilation.
  package Target is separate("micro16");

    Definitions that are visible to the entire system

  // A package for communicating with hardware devices through
  // an SPI serial bus, marked 'shared' in order to serialise
  // access to the package contents
  shared package SPI_interface is
    Declarations, including
    - Private global variables that are used by the SPI routines
    - Internal procedures
    - Interface (public) procedures for communicating with devices
  begin
    // Initialise the SPI port
    Package initialisation code
  end SPI_interface;

  // The subsystem for implementing the receive functions (this could also
  // be declared separate if desired)
  subsystem Receiver is
    Definitions

    // The main procedure of this subsystem
    main procedure Operate_receiver is
    begin
      Statements
    end Operate_receiver;

  // This subsystem starts by running the procedure Operate_receiver (after
  // package initialisation, if any)

  begin
    Initialise the receiver subsystem
  end Receiver;

  // The subsystem for implementing the transmit functions
  subsystem Transmitter is
    Definitions

    main procedure Operate_transmitter is
    begin
      Statements
    end Operate_transmitter;

  begin
    Initialise the transmitter subsystem
  end Transmitter;

  // A package that implements a local IO interface, stored in a separate source file,
  // but logically appearing at this point in the system
  package IO_Interface is separate;

  More definitions that are only visible from this point down

  // The procedure that embodies the main program flow (if any)
  main procedure Main_start is
  begin
    Statements
  end Main_start;

begin
  Global initialisations (if required)
final
  Global finalisations (if required)
end Radio_transponder;
```

The translation model is 'single compilation', i.e. the entire system is always analysed as a whole. The position of all constituent packages (including subsystems) in the package hierarchy is explicit

and can be read off from the source code (even though there may be several source files). The order in which packages are initialised and finalised is determined unambiguously by the program structure.

2.2 The target definition

The Rihtan compiler does not generate code for a particular target. It generates C source. The target is defined by a series of pragmas that specify the details about underlying C types that can be used, along with other target-specific information. Thus a single Rihtan compiler can be used for many different targets. Typically, the target specification pragmas are collected in a 'target package' and referenced at the start of the system.

2.3 Data types and units

2.3.1 Integer types

Rihtan only allows explicitly-defined integer types (there is no 'int', for example). It is always possible to determine the exact range of an integer-valued object from examination of the source code.

```
type Sensor_index is range 1..10;
```

It is an error to assign an out-of-range value to a variable or to pass an out-of-range value as an actual parameter, for example to assign 0 to a variable that is of a type with range 1..10. Rihtan detects errors of this kind at time of compilation, and will refuse to compile a program unless it can establish that no out-of-range assignment can occur. For most statements it can establish this itself. In some cases, the programmer might need to embed additional information in the source code in the form of `advise` statements, but this is explicit and can be identified for review.

Normally Rihtan selects an appropriate underlying C integer type from the target definition. The target definition does export types that correspond to the C types, but they should only be used where that concept is appropriate. It is possible to tell Rihtan explicitly which target C type to use for a Rihtan integer type using a representation clause, provided that the C type is capable of representing the stated range.

A different problem to a value being out of range is overflow, where an arithmetic operation exceeds the range of a machine type and generates a notionally undefined, and almost certainly erroneous, value. Rihtan analyses expressions and inserts conversions between the underlying target types where necessary in order to avoid overflow. Precise definition of ranges makes this more likely to succeed. If freedom from overflow cannot be guaranteed, then the program will not compile.

In summary, in Rihtan an integer expression is not allowed to overflow, and the result of the expression must provably be in range of the object to which that result is being assigned (intermediate values need not be).

2.3.2 Units

As well as a range, an integer type or constant can have a unit. The unit helps to prevent incorrect assignments to variables and parameters. Units can be simple, like 'metres', or compound, like 'metres per second squared'. Compound units can be given names, like 'area' for 'metres squared'. In assignments, the unit of the left and right hand sides must match, after simplification and rearrangement. In expressions, units combine in a conventional way: under multiplication, units are multiplied together; under division, units are divided; under addition and subtraction and other

operators, units must match.

Units do not need to correspond to physical quantities.

2.3.3 Enumerated types

Enumeration types in Rihtan are integers, and enumeration literals are integer constants, but with a unique unit that is created when the type is declared. The unit prevents the arbitrary mixing of values between enumeration types and between enumeration types and other numbers. The unit can be obtained by applying the `'unit` attribute to the type or a variable of the type. Explicit conversions are possible in the same way as for other numerical types by multiplying and dividing by the unit, subject to the usual range checking.

2.3.4 Character type

The Rihtan `character` type is an integer type, and character literals are integers, but with a unique unit (`character'unit`). As with enumerated types, the unit regulates the interactions between character values and other integers.

2.3.5 Boolean type

Rihtan has a predefined type `boolean` that has two values, `false` and `true`. It is neither a number nor an enumeration, but a separate basic type. In Rihtan the only way to generate a boolean value is either to use the constant literals `false` and `true` or to use an operator that yields a boolean, like `>` or `and`, and the only valid comparisons between boolean values are equality and inequality.

2.3.6 Floating point types

Rihtan does not provide predefined floating point types like ‘float’ and ‘double’. All floating point types are declared with a minimum precision (in decimal digits) and minimum magnitude (in powers of 10). Rihtan will select a suitable C type from the target definition. Rihtan does not provide extensive detailed support for floating point types, but it does perform some basic checking, for example that the magnitude of a variable will not be exceeded. Rihtan does not provide a test for exact equality or inequality for floating point values. Floating point quantities are inherently approximations and tests for exact equality are generally mistakes (the correct test is normally *abs(a – b) < epsilon*, for some appropriate relatively small value *epsilon*).

Floating point types and constants can take units.

2.3.7 Arrays

Rihtan has first-class array types. Array types have an index type, which must be an integer type of some sort (including enumerations), and an element type, which can be any type.

Arrays can have either fixed or 'open' ends. An array that is open at one or both ends can only serve as a formal parameter that can match an actual parameter with the same element type and a unit-compatible index type that has a range that does not exceed that of the formal parameter.

Special forms of assignment statement are used to initialise entire arrays, after which individual elements can be modified.

2.3.8 Records

Records are aggregates of fields of various types. Records in Rihtan come in three varieties: regular records in which every field is present, unions in which only one field is present at a time, for which static checks are made to ensure that the correct field is accessed at any point in the program, and unchecked unions, which are C-like unions for which no check is made.

A special form of assignment statement is used to initialise a record, after which individual fields can be modified.

2.3.9 Access types

Access types are pointers. They come in various flavours, depending on whether the object that is referenced can be modified, whether the pointer must always refer to an object or can be `null`, whether the access value references objects that are persistent throughout the lifetime of the program, and whether or not referenced objects are automatically freed from memory when no more references to them exist. The `new` operator creates an object of the referenced type. The `free` statement frees objects that are not automatically freed.

2.4 Named constants

Constants can be named. They can be simple literal values or the result of evaluating constant expressions at compile time. *Named numbers* and *named booleans* do not occupy storage on the target.

2.5 Variable declarations

The declaration of a variable can either employ a type that has already been declared, or it can give an anonymous type declaration, which can be a useful abbreviation when declaring unique objects such as memory-mapped registers.

A variable declaration can include an initialisation expression. If it does, then the expression will be evaluated at run-time but the variable cannot be reassigned for its lifetime. In the case of a variable that is local to a procedure or function, it will maintain its value for the duration of the call to the procedure or function (it could take different values on subsequent calls); in the case of a global variable, it will retain the value for the duration of the program run.

Numerous representation clauses can be applied to variable declarations, to set such things as address in memory and means of access.

2.6 Statements

2.6.1 Assignments

Assignment in Rihtan is a statement, not an operator. The left hand side of an assignment is an object; the right hand side is an expression. In Rihtan, expressions are not allowed to have side-effects. One consequence of this is that functions cannot have side-effects[1], either directly or indirectly through something that they call (Rihtan distinguishes between functions, which return values, and procedures, which do not).

2.6.2 Loops

Rihtan provides indefinite loops, while loops and for loops:

```
loop
…
end loop;
while condition loop
…
end loop;
```

[1]Except by explicitly applying an exemption to this rule. Naturally this should only be done in exceptional cases.

```
for var : sometype in firstval..lastval loop
…
end loop;
```

Indefinite loops loop forever and can only be exited with an exit statement:

```
exit when condition;
```

They can be restarted from before the end with a repeat statement:

```
repeat when condition;
```

In a `while` loop the condition is tested at the start of every loop, and the loop terminates when the condition is false.

A `for` loop iterates in steps of 1 over a range of the nominated type.

In Rihtan, `for` and `while` loops always run to completion as stated. Loops that can be exited at multiple points, or other than at the top, are coded with an indefinite `loop` and `exit` statements.

A `for` statement in Rihtan declares a loop control variable that is in scope within the loop body. The type of the loop control variable must be stated and hence can always be read directly from the source. In the common case that the loop is to be over the entire range of the type (for example, when iterating over an array index), the range part can be omitted as an abbreviation. If the type doesn't already exist (for example, we just want to do execute the body a fixed number of times), a `range` declaration can appear instead of the type name, creating an anonymous type for the loop control variable.

2.6.3 If and do statements

The Rihtan `if` statement has the general form:

```
if condition1 then
…
elsif condition2 then
…
else
…
end if;
```

The `elsif` and `else` parts can be omitted if not required.

Rihtan also provides a `do` statement, the purpose of which is to flatten nested `if` structures. This performs a similar role to `elsif`, but where nesting occurs in the true rather than the false branches:

```
do
  …
  exit when condition1;
  …
  exit when condition2;
  …
end do;
```

2.6.4 Case statements

A Rihtan case statement has the form:

```
case expression is
  when opt1 => statements1
```

```
  when opt2 | opt3 => statements2
  …
  when optn => statementsn
end case;
```

In Rihtan there are no ranges within the case options and there is no optional catch-all 'when others' or 'default' clause. Every value that the expression can take at that point in the program must be listed within the `case` statement. Often these are the literals of an enumerated type, but other integer types can be used. If it is desired to to restrict the `case` statement to a sub-range of the type, it can be surrounded by an `if` statement.

2.6.5 Blocks

A block allows for local declaration of types, units, constants and variables for a group of statements, and for modifiers such as exemptions to rules to be applied to those statements. It has the general form

```
declare
  declarations
begin using (modifiers)
  statements
end;
```

The `declare` and `using` sections can be omitted if not required.

2.6.6 The null statement

Rihtan provides a `null` statement that does nothing. It is used in situations where a statement is required syntactically but there is nothing to do (statement lists cannot just be left empty like they can in C).

2.6.7 Procedure call statements

Rihtan distinguishes between functions, which return a value, and procedures, which do not. Function calls can only appear in expressions. Procedure calls can only appear as statements[2].

Rihtan supports both positional and named notations for parameters in calls; furthermore, Rihtan insists that named notation be used if there is a possibility of parameters being supplied in the wrong order (based on their types). The parameter mode (input, output, input-output etc.) can, but need not, be stated (but if it is stated, it must match the mode of the formal parameter).

```
p(x, y, 1);
p(source => x, dest => y, numvals => 1);
p(source => in x, dest => out y, numvals => 1);
```

2.7 Unit testing

Unit test code is embedded directly within the source, although it can be separated out into different files using `separate` declarations in order to improve readability. Such code is only compiled into the target program when the compiler is instructed to perform a unit test build. In this case, the compiler will also generate code to run the unit tests and perform code coverage recording in order to check that every path through the code is exercised. In addition, variables that are declared as representing memory-mapped devices are converted to ordinary variables so that unit testing code can assign test values to them, and so that unit test code can be run on a different target (typically

[2] With one exception for record and array initialisation and finalisation clauses

the same as the compiler host) without generating memory access faults.

2.8 Pragmas, exemptions and 'advise' statements

Pragmas can supply additional information to the compiler, request particular cross-checks, or influence the generated code. Exemptions can be applied to blocks of code or to procedure or function call parameters - they are used to relax some of the checking that the compiler normally applies. `Advise` statements give extra information to the compiler that might be needed when it cannot establish a particular fact for itself.

Pragmas, exemptions and `advise` statements do not affect the meaning of the program, but they can affect its legality. The compiler is strict by default, and if it cannot establish that a computation can be performed safely (for example, that the value for an assignment is definitely in range), then it will refuse to compile the program. These tests can be relaxed on a case-by-case basis, but only by explicitly stating the intention to do so.

For example, an `advise` statement can tell the compiler that the range of values that a variable can take at a particular point in the program is more restricted than what the compiler can determine by itself. This can make a program legal when it would otherwise be illegal due to a possible out-of-range assignment or an overflow. An exemption can be used to allow a side-effect in a function, or to suppress the check for aliasing of a global variable in a procedure call.

Pragmas can also affect implementation details, for example the selection of underlying memory allocation routines.

2.9 Subsystems

Subsystems represent separate threads of control. They are a first-class language feature. There is generally no need to explicitly manage threads with calls to a threads library.

Shared packages serialise access by subsystems (and the main program) to resources that are managed through procedural interfaces.

Shared variables are locked when they are passed as parameters or by calls to procedures that list them as needing to be locked, and can only be accessed by one thread of control at a time.

Guard expressions provide regulation of access without requiring explicit polling, for example between a producer that generates values and stores them in a buffer, and consumers.

2.10 Interoperability with C

The compiler generates readable C source code in a predictable fashion. This can be incorporated into existing C projects, or it can be used as the primary program structure, with which C code can be combined.

The compiler usually generates a C name for types, constants, variables and routines, but this can be overridden and an explicit name given.

An object can be declared to be a reference for a C object that is declared elsewhere.

C code can be embedded directly within procedure and function bodies and initialisation and finalisation sections. Syntax is provided to translate between source names for objects and the generated C names.

C fields can be included in record definitions.

2.11 Features for embedded programming

The machine word types that are used to represent values of declared types can be specified explicitly.

The layout in memory of records can be defined.

The addresses of variables in memory can be specified.

Mode of access can be declared. These include volatile access, memory-mapped registers, and program image. The latter allows global variables with initialisation expressions that can be evaluated completely at compile time to be included in the program image rather than in the data space; an example might be a constant look-up table.

The routines that are used for memory allocation and deallocation (if any) can be specified for packages and for individual types.

The compiler attempts to derive an expression for stack size requirements for the system and subsystems. Target pragmas allow this to be fine-tuned or overridden.

2.12 Generics

Generic packages (including subsystems) can be declared. These can be instantiated one or more times with different actual types, constants, procedures, functions and other packages replacing formal parameters in order to generate new packages.

2.13 Other program integrity features

Other features for program integrity include assert statements, and pre-conditions and post-conditions for routines. These are all checked statically. In addition, runtime checks can be inserted using pragmas.

3 Lexical elements

3.1 Identifiers

Identifiers start with an alphabetic character (`A-Z`, `a-z`) followed by zero or more alphabetic characters, digits (`0-9`) and underscores (_), except that two underscores cannot occur together without any intervening character.

Examples

```
A, a, input_B, test01
```

But not

```
2xy, X__Y
```

Identifiers are not case-sensitive, in so far as they cannot be distinguished by case alone (so, for example, `Abc` and `abc` are not different identifiers). However, it is illegal for two identifiers in the same scope to differ only in case, so that, for example, an entity named `Abc` cannot be referred to as `abc`.

The declaration of a named entity (unit, type, variable, function, procedure or package) cannot mask another entity in the same scope that has the same name. For example, if an entity with name `n` is visible at the point of the declaration of a new entity with name `n`, then the declaration is not allowed. This restriction does not apply to entities that are accessible but not in direct scope, such as a public object named `n` within another package `p` that is accessed as `p.n`.

3.2 Integer literals

Integer literals are represented either as sequences of decimal digits, or using the C notation for hexadecimal literals, `0x` followed by one or more hexadecimal digits (the case of digits `a-f` is not significant). Decimal literals can be prefixed with a '`-`' character to indicate negative values.

Examples

```
1, 234, 0x0a, 0xB3, -2
```

3.3 Floating point literals

Floating point literals consist of an optional '`-`' sign, followed by one or more decimal digits, then a '`.`', then one or more decimal digits, then optionally an exponent. The exponent consists of the character '`E`' or '`e`', followed by an optional '`-`' sign, followed by one or more decimal digits.

Examples

```
0.0, -1.5, 2.8E15, 1.0e-9, -1.0E30
```

3.4 Character literals

Character literals are one of the following:

- A single character enclosed in single quotes
- One of the C escaped literals `\n \r \b \t \f \a \v \' \\` enclosed in single quotes
- The C notation for a hexadecimal character code `\x` followed by one or two hex digits, all enclosed in single quotes
- The C notation for an octal character code, `\` followed by one, two or three octal digits, all enclosed in single quotes.

Examples

```
'a', '\n', '\\', '\x0a', '\12'
```

3.5 String literals

String literals consist of sequences of zero or more characters enclosed in double quotes. Strings can contain escaped C character literals (e.g. `\n`). Strings cannot span the end of source lines.

Examples

```
“String 1”, “”, “Line\n”
```

3.6 Boolean literals

The boolean literals are `true` and `false`.

3.7 Comments

Comments start with `//` and extend to the end of the line, except that `//` within a string literal does

not indicate a comment (it is part of the string).

Comments that appear alone on a line (i.e. not the right of something else) can optionally be copied to the generated C program as comments. These comments can generally only appear in places where a complete declaration or statement could appear. They cannot appear inside a declaration except within the declaration of a record type, where they can be used to comment fields. They cannot appear in a file but outside any package, except that one or more comments can appear immediately before the outermost package (or system or subsystem) declaration within a file (where they are typically used to state the purpose of that package).

However, comments that start with `//-` will be discarded and can appear at any point (on a line by themselves or otherwise).[3]

Examples

```
// This comment can be copied to the C file
x := 1;  // This comment will not be copied


//- This comment will not be copied to the output
x := 1;  //- This comment will not be copied either


// This comment is allowed and can be copied to the header file
type R is record
  // This is allowed and can be copied
  x : range 1..10;  // This is allowed but will not be copied
  //- This is allowed but will not be copied
  y : boolean;
end record;


type Arr is array [Index] of
  // This is not allowed
  //- But this is
  R;


// This is allowed here and can be copied
package p is
  // This is allowed
  …
end p;
// This is not allowed
//- But this is
```

3.8 Reserved words

The following identifiers are reserved as part of the language syntax and cannot be used for other purposes (such as variable names). They must appear in lower case.

and	**exit**	**not**	**then**
access	**false**	**null**	**true**
all	**advise**	**assert**	
array	**final**	**or**	**type**

[3]Comments that do not occupy their own line or that start with //- are simply discarded; comments that start with // on their own line are included in the structure that is built up by the parser so that they can be output to the C program with correct formatting and indentation.

begin	**for**	**out**	**unchecked**
case	**function**	**package**	**union**
closed	**generic**	**pragma**	**unit**
constant	**if**	**procedure**	**using**
controlled	**in**	**public**	**when**
declare	**is**	**range**	**while**
digits	**loop**	**record**	**with**
do	**magnitude**	**separate**	**restart**
else	**mod**	**shared**	**repeat**
elsif	**name**	**subsystem**	**use**
end	**new**	**system**	

4 Basic type definitions

This section describes the definition of units, integers, enumerated types, characters, floating-point types and booleans. The description of composite and access (pointer) types appears later.

4.1 Unit declarations

All numeric types can specify a unit. Units are used to check the integrity of arithmetic, logical, parameter passing and assignment operations.

Units must be declared. A *basic unit declaration* simply states that a unit exists:

```
unit basic_unit_name;
```

A *derived unit declaration* combines basic and derived unit declarations to create a more complex unit:

```
unit derived_unit_name is unit_ref [power_term] {* unit_ref [power_term] }*
```

Unit_ref is either the name of a unit, or has the form *name*'unit where *name* is the name of a type or a variable, meaning the unit of that type or variable. The power terms are optional. A power term is of the form ^*integer_constant*, meaning raised to the power *integer_constant*. The * to the left of *unit_ref* is present literally and can be read as 'times'.

The name of a unit can be used in an expression. It has the value 1 and unit itself. Multiplying and dividing by a unit can be used to perform unit conversions.

The unit of a type or a variable can be retrieved by applying the attribute 'unit to its name. The term *name*'unit has the value 1. It can also be used in expressions.

In expressions, assignments and parameter passing, units match when, after they have been normalised so that every basic unit appears once in each, the basic units in each appear raised to the same power, and neither contains a basic unit that does not appear in the other (i.e. they can be rearranged to make them identical).

Examples

```
unit cycles;
unit seconds;
unit cycle_rate is cycles*seconds^-1;
unit cycle_acceleration is cycle_rate*seconds^-1;
```

4.2 Integer types

4.2.1 General integer types

General integer types are created with range declarations. There are no predefined integer types[4]. A *basic range declaration* has the form:

```
type type_name is range integer_constant_low..integer_constant_high;
```

This declares that objects of type `type_name` can take integer values in the range `integer_constant_low` through `integer_constant_high` inclusive. The integer constants must be statically-computable unitless integer expressions.

A basic range declaration can include a unit:

```
type type_name is range integer_constant_low..integer_constant_high unit
unit_specification;
```

where `unit_specification` is the name of a previously declared basic or derived unit, or takes the form of an anonymous derived unit declaration.

The following variation creates a new anonymous unique unit:

```
type type_name is range integer_constant_low..integer_constant_high new unit;
```

This unit can be retrieved with `type_name'unit`.

A *computed range declaration* has the form

```
type type_name is range for expression;
```

where `expression` is an integer-valued arithmetic expression. In this case the compiler determines a range that is exactly sufficient to represent the result of computing the expression, given what it knows about the values that the constituent terms of the expression can take. The type's unit is also derived from the expression. Within `expression` the name of an integer type (or an equivalent `'type` attribute) can appear, in which case the full range of that type is used for the purpose of evaluating the range of the expression.

The minimum and maximum values that an integer type `t` can take can be retrieved with the attributes `t'first` and `t'last` respectively, and the unit with `t'unit`. These attributes yield constants with the relevant unit. The `'unit` attribute has the value 1. These attributes can also be applied to a variable `v` of type `t`, in which case they yield the same values as if applied to the type `t`, i.e. `v'first = t'first` etc.

When applied to an integer-valued variable `v`, `v'minimum` yields the lowest value that the variable can take at that point in the program (which in general could be greater than `v'first`), and

[4]Apart from the type ***character***, which is an integer type with a unique unit.

similarly *v*`'maximum` yields the highest value.

Examples

```
type workstation_ID is range 1..Number_of_workstations;
type operator_ID is range 1..Number_of_workstations * Seats_per_workstation;
type cycle_number is range 0..100 unit cycles;
type rate_of_change is range -300..300 unit cycles*time_intervals^-1;
type intermediate_value_range is range for object_count * object_size + start_offset;
mid : intermediate_value_range :=
        (intermediate_value_range'first + intermediate_value_range'last) / 2;
```

4.2.2 Enumerated types

Enumerated types specify a set of discrete, named values:

```
type type_name is (name1, name2, …, namen);
```

This is equivalent to the following by definition[5]:

```
unit unique_unit;
type type_name is range 0..n – 1 unit unique_unit;
name1 : constant := 0 unit unique_unit;
name2 : constant := 1 unit unique_unit;
...
namen : constant := n - 1 unit unique_unit;
```

Thus enumerated types can be used in expressions, but this use is regulated by the unit. As with other integer types, the unit can be retrieved with the `'unit` attribute, and the minimum and maximum values with `'first` and `'last`.

Examples

```
type Day_of_week is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

4.2.3 The type character

The predefined type `character` represents a single character. It is an integer type with a unique unit. The specific type is given in the target definition. The unit can be retrieved with `character'unit`, and the range with `character'first` and `character'last`. Character literals and variables of type character have the unit `character'unit`.

Examples

The expression `char / character'unit`, where `char` is of type `character`, yields a unitless integer with the encoding value that represents `char`*:*

```
ASCII_A_code : constant := 'A' / character'unit;
```

4.3 Floating point types

Floating point types are declared with a magnitude, a precision, and optionally a unit.

```
type type_name is magnitude integer_constant digits integer_constant;
```

[5]Except that the base 0 can be changed - see 22.1

```
type type_name is magnitude integer_constant digits integer_constant unit
unit_specification;
type type_name is magnitude integer_constant digits integer_constant new unit;
```

The magnitude term indicates that the compiler should choose a type that can represent numbers of magnitude at least 10 raised to the indicated power. The digits term indicates that the compiler should choose a type with at least the indicated number of decimal digits of precision. The unit is specified in the same manner as for integer types.

4.4 The type boolean

The predefined type `boolean` has only two values: `true` and `false`. It is not a numeric type and does not have a unit.

4.5 The type string

The type `string` represents a null-terminated C string, i.e. a pointer to an array of characters that is terminated with a zero byte. This is not the same as an array of elements of type `character`.

A string can be assigned to an array of characters, however, in which case it will be copied to the array. If the string is shorter that the array, then the array will be padded on the right with zeroes. If the string (including the trailing zero byte) is longer than the array then it only the leftmost part of the string will be copied. At least one zero will be included in the array.

A global array of characters can be assigned to a string, which means simply taking the address of the array. The address of an array slice can also be taken. It is up to the programmer to ensure that the relevant part of the array is terminated with a zero character. A local character array variable cannot be assigned to a string because of the danger of leaving references to deallocated objects. For the same reason, a local character array variable cannot be passed as a string parameter, except in the case of passing it as an `in` mode parameter to a function or a closed procedure (i.e. to a routine that cannot make a copy of the string that lasts longer than the call).

String constants are like C string contants (i.e. enclosed in double quotes), including the standard escape sequences. The symbol *null* can also be used, and corresponds to C `NULL`.

5 Named constant declarations

Named number declarations give names to numerical constants, optionally with a unit:

```
constant_name : constant := integer_expression;
constant_name : constant := integer_expression unit unit_specification;
constant_name : constant := floating_point_expression;
constant_name : constant := floating_point_expression unit unit_specification;
```

The expressions that appear to the right of the `:=` must be static, i.e. it must be possible to evaluate them fully at compile-time. If a unit is specified, then the expression itself must either be unitless, or must yield a value with the same unit.

Named boolean declarations declare boolean constants:

```
constant_name : constant := boolean_expression;
```

Boolean_expression must be static.

Named access declarations declare access constants (see 25).

```
constant_name : constant := var'access;
constant_name : constant := null;
```

The variable *var* must be global (not a local variable or a parameter).

Named string declarations declare strings:

```
constant_name : constant := "string";
```

Named constant declarations do not denote runtime objects. All occurrences of them could, in principle, be replaced with the corresponding literal (and unit)[6]. However, in the case of a string constant only a single array of characters is actually stored in the program image, regardless of how many times the constant is referenced.

Examples

```
Number_of_inputs : constant := 5;                          // unitless integer
Mast_height : constant := 4.2 unit metres;                 // floating-point metres
Trace : constant := true;                                  // boolean
Even_number_of_inputs : constant := Number_of_inputs mod 2 = 0; // boolean
Maximum_velocity : constant := 30.0 unit metres*seconds^-1;
Maximum_time : constant := 5.0 unit seconds;
Maximum_distance : constant := Maximum_velocity * Maximum_time; // unit metres
Address_of_x : constant := x'access;
Error_message : constant := "An error";
```

6 Structured types

6.1 Record types

Record types group items together into composite objects. There are three kinds of record types: basic record types, unions and unchecked unions. Record type declarations can be annotated with representation clauses that control their layout (see 65).

6.1.1 Basic record types

Basic record type declarations are as follows:

```
type type_name is
  record
    field_name1 : field_type1;
    field_name2 : field_type2;
    ...
    field_namen : field_typen;
  end record;
```

Every value of type *type_name* includes one of each of the named fields.

Each of *field_typei* is either the name of a type, or a type attribute of a type or variable, or is a type definition, in which case an anonymous type is created. If a representation clause (`using (…)`) is appended to the field type definition then it applies to the field; in order to make it apply to the

[6]The keyword `constant` always indicates a value that can be determined statically by direct examination of the source text.

field type, enclose the type definition (including its representation clause) in parentheses.

In the absence of overriding representation clauses, the compiler chooses the layout of the fields in memory.

6.1.2 Union record types

Union record types are declared in a similar manner to basic record types:

```
type type_name is
  union
    field_name1 : field_type1;
    field_name2 : field_type2;
    ...
    field_namen : field_typen;
  end union;
```

However, a variable of a union record type takes the value of only one of the named fields at any time. Compile-time restrictions ensure that values are used in a consistent way. This are described here, but see section 38 for a general description of the `case` statement that is used.

Every union record includes a tag that indicates the field that is set at any time. All accesses to union record fields must be take place within a case statement that tests the tag:

```
case union_variable is
  when field_name1 => statement_list1;
  when field_name2 => statement_list2;
   …
  when field_namen => statement_listn;
end case;
```

All possibilities must be covered by the case statement. Within any *statement_list*$_j$, the corresponding field can be accessed. The field can be modified within the appropriate branch of the `case` statement (unless the record is an `in` parameter); however, it is not possible to assign to the union record as a whole within the `case` statement, because that could change the tag.

Outside of `case` statements like these it is only possible to modify a union record by performing a whole record or copy assignment.

Null fields

The following special form of field definition is provided:

```
field_namej => null;
```

A field of this sort can be used when only the tag is of interest. For example, a union might represent either a valid value, or the fact that no value is available. Rather than associating an arbitrary type with the latter, a null field can be used. Such a field requires no storage, and no runtime code is required to initialise it. A specific syntax is used when assigning to a `null` field (see 33).

```
type SensorReading is
  union
    InputData    : SensorValue;
    NotAvailable : null;
  end union;
```

6.1.3 Unchecked union record types

Unchecked union record types are declared in a similar manner to other record types:

```
type type_name is
  unchecked union
    field_name1 : field_type1;
    field_name2 : field_type2;
    ...
    field_namen : field_typen;
  end union;
```

Like a union record type, a variable of an unchecked union record type takes the value of only one of the named fields at any time. But, unlike with a union record type, there are no restrictions to ensure that values are used in a consistent way (and no tag). Thus an unchecked union is a simple uncontrolled overlay, mapped directly to a C union.

6.1.4 C field declarations

All record type declarations can include C field declarations at any point where a field declaration can appear. C field declarations are marked with a `!` character as the first non-whitespace character of a line. The rest of the line is a C code field declaration, as it would appear in the declaration of a C struct. At least one character of whitespace must separate the `!` from the code. The C field is included in the generated structure definition, but is otherwise invisible to the compiler. It can be accessed by C code statements.

6.1.5 Attributes of record types

The attribute `'field_type(`*`fieldname`*`)`, when applied to a record type or a variable of a record type, yields the type of the field called *`fieldname`*.

Examples

```
type Sensor_record is
  record
    Previous_timestamp : Timestamp;
    Previous_value     : Sensor_value;
    Current_timestamp  : Timestamp;
    Current_value      : Sensor_value;
    Statuses           : array [range 0..MAX_COUNT] of boolean;
  end record;

type Temperature_record is
  union
    Temperature_value : Temperature;
    Error_code        : (NO_ERROR, READ_ERROR);
  end record;

type Register_view is
  unchecked union
    As_fields  : Register_fields;
    As_integer : Register_integer;
  end union;

type File_access is
  record
    File_open : boolean;
  ! FILE *Descriptor;
```

```
end record;
```

6.2 Array types

6.2.1 Declarations

Arrays are collections of objects of a given type, indexed by an integer type. An array declaration can declare a type of fixed size, called a fixed array declaration, or a type with an index that is not fixed at one or both ends (i.e. where either the minimum or maximum or both are allowed to take different values), called an open array type.

Fixed array types are used to declare objects. Open array types are used as formal parameters that can match type-compatible objects of different sizes (see below).

```
type type_name is array [index_type] of element_type;
type type_name is array [index_type]> of element_type;
type type_name is array <[index_type] of element_type;
type type_name is array <[index_type]> of element_type;
```

The first of these declares a fixed array type. The other three declare array types that are open at the right, at the left, and at both the right and the left, in that order.

The index type must be an integer type, i.e. a general integer type, an enumeration type, or the type `character`. It can name a type that has already been declared, or it can take the form of an integer type specification (`range …`), in which case an anonymous type is created for the purpose.

The element type can either name a type that already exists, or it can be the specification of a new type, in which case the type is anonymous. The element type can be any type, including another array type, but not an open array type. If a representation clause (`using (…)`) is appended to the array type definition then it applies to the array as a whole; in order to make it apply to the element type, enclose the element type definition (including its representation clause) in parentheses.

A formal parameter of an array type that is fixed can match a type-compatible actual parameter with the same minimum and maximum indexes.

A formal parameter of an array type that is open at the left can match a type-compatible actual parameter, the minimum index of which is greater than or equal to the minimum value of the formal's index type, and the maximum index of which is equal to the maximum value of the formal's index type.

A formal parameter of an array that is open at the right can match a type-compatible actual parameter, the maximum index of which is less than or equal to the maximum value of the formal's index type, and the minimum index of which is equal to the minimum value of the formal's index type.

A formal parameter of an array that is open at both the left and the right can match a type-compatible actual parameter, the minimum index of which is greater than or equal to the minimum value of the formal's index type, and the maximum index of which is less than or equal to the maximum value of the formal's index type.

An actual array parameter is type-compatible with a formal array parameter when their element types are the same and their index types have the same unit.

All array types are single-dimensional. Multi-dimensional arrays are declared as arrays of arrays.

Examples

```
type Sensor_index is range 1..20 unit sensor_ID;
type Sensors is array [Sensor_index] of Sensor_data;
type Sensors2 is array [range 0..50] of Sensor_data;

type Array_for_sorting is array <[Some_index]> of Some_element;

type Earliest_readings is array [Reading_index]> of Reading;

type FractionPercent is array [range 1..10] of range 0..100 unit Percent;

type MappedArray is
  array [range 0..127] of (range 0..15 using (word_type => unsigned_8));
```

6.2.2 Array attributes

For an array type *t*, the term `t'first` yields the lowest value of the index, and the term `t'last` yields the highest value, with the appropriate unit.

For an array variable *a*, the term *a*`'first` yields the lowest value of the variable's index, and *a*`'last` yields the highest value.

For an array type *t*, the term `t'length` yields `(t'last-t'first+1)`.

For an array variable *a*, the term `a'length` yields `(a'last-a'first+1)`.

For a formal parameter *p* of an array type (open or fixed), the term *p*`'first` yields the lowest index value of the corresponding actual parameter, *p*`'last` yields the highest value, and `p'length` yields `(p'last-p'first+1)`. If the formal parameter's type is open at the left, then the values of *p*`'first` and `p'length` can vary from one call to another, and similarly if the formal is open at the right, then *p*`'last` and `p'length` can vary.

For an array type *t* or an array variable *a*, the terms *t*`'index_unit` and *a*`'index_unit` yield the unit of the index type of *t* or *a* respectively, with value 1.

For an array type *t*, the term *t*`'index_type` is the type of the index of *t*, and *t*`'element_type` is the type of the elements of *t*. These expressions can be used in contexts where a type name is expected.

For a variable or formal parameter *v* of an array type *t*, *v*`'index_type` is the index type of *t*, and *v*`'element_type` is the element type of *t*. As above, these expressions can be used in contexts where a type name is expected.

6.2.3 Array indexing

The element of array *a* at index *index_expr* is referenced with the notation *a*`[`*index_expr*`]`. *index_expr* must be a value within the range of the index of *a*. The unit of *index_expr* must either be the same as the unit of the index type of *a*, or else it must be unitless.

6.2.4 Array slices

For an array variable *a* of objects of type *t*, the notation *a*[*first*..*last*], where *first* and *last* are expressions that are unit-compatible with the index type of *a*, or else unitless, is an array slice that means the part of the array between indexes *first* and *last* inclusive; the type of the slice is an array of *t* with an index type of the same unit as that of *a*.

The notation *a*[*first* for *length*], where *first* is an expression that is unit-compatible with the index type of *a*, or else unitless, and *length* is a unitless positive integer expression, is an array slice that means the part of the array between indexes *first* and *first* + *length* − 1 inclusive.

The notation *a*[*first*..], where *first* is an expression that is unit-compatible with the index type of *a*, or else unitless, is an indefinite slice starting from index *first* that can be used in the following situations:

- On the right-hand side of an assignment to another array, where sufficient elements from *a* will be taken to satisfy the assignment (and sufficient elements must be available)
- On the left-hand side of a whole-array assignment [range …]
- As an actual parameter, where the corresponding formal parameter has a fixed maximum index

In each case checks will be made at compile-time to ensure that the assignment can be performed, or parameter passed, consistently.

A slice can only be taken from an array variable or a parameter of a fixed array type, not from a parameter of an open array type.

An array slice can be passed as a parameter. In this case the 'first attribute of the formal parameter will have the value of *first*, and the 'last attribute will have the value of *last* in the case of the first kind of slice (a[*first*..*last*]), or of *first* + *length* − 1 with the second kind (a[*first* for *length*]).

An array slice can be assigned to an array or slice of compatible type, or from an array or slice of compatible type. In these cases, the lengths of the source and target of the assignment must be equal and static (i.e. they can be evaluated at compile-time); the actual indexes however do not need to be the same. With the first kind of slice the *first* and *last* expressions must be static; with the second kind, the *length* expression must be static. An overlapping assignment within a single array is allowed and is handled correctly (i.e. as if the source were evaluated and then written to the target).

An array slice can be the target of a whole-array assignment (see 33), in which case *first* and *last* or *length* (as the case may be) do not need to be static, but it must be possible to prove, at compile-time, that *last* >= *first* or *length* > 0 respectively, and that the bounds of the slice are within the bounds of the array.

An assignment to a slice of an array variable does not count as initialising the variable.

7 Access types

Access types provide for dynamic allocation of objects.

```
type access_type_name is access accessed_type;
type access_type_name is access or null accessed_type;
```

```
type access_type_name is access constant accessed_type;
type access_type_name is access or null constant accessed_type;
type access_type_name is access new accessed_type;
type access_type_name is access or null new accessed_type;
type access_type_name is access constant new accessed_type;
type access_type_name is access or null constant new accessed_type;
type access_type_name is managed access accessed_type;
type access_type_name is managed access or null accessed_type;
type access_type_name is managed access constant accessed_type;
type access_type_name is managed access or null constant accessed_type;
type access_type_name is persistent access accessed_type;
type access_type_name is persistent access constant accessed_type;
```

The option `or null` indicates that variables of the access type can take the special value `null`, which indicates that they do not access anything. If `or null` is omitted, then values of the access type must always refer to objects of type *`accessed_type`*.

`Accessed_type` can either name a type that already exists, or it can be the specification of a new type, in which case the type is anonymous. If a representation clause (`using (…)`) is appended to the access type definition then it applies to the access type; in order to make it apply to the accessed type, enclose the type definition (including its representation clause) in parentheses.

The forms that include the keyword `constant` define types that provide read-only access; they cannot be used (directly or indirectly) on the left-hand-side of an assignment statement to address the target of the assignment.

Objects are allocated by the operator `new`, which returns either a value of type `managed access` *`type_name`*, if the target of the assignment is a variable of a managed access type, or of type `access new` *`type_name`* if the target is of a non-managed access type. Objects that are created with a managed access type are deallocated automatically as soon as no references to them exist[7]; if the access value with which an object is allocated is not managed then the object must be freed explicitly using the `free` statement (if it needs to be freed at all). The `free` statement can only be applied to a value of type `access new` that is known to be not `null` at the point of the statement.

The object that is addressed by an access value *`p`* is referred to as *`p`*`.all` . If the object is a record, then a field *`f`* of that record can be referred to as *`p`*`.all.`*`f`*, or using the abbreviation *`p`*`.`*`f`* . If the object is an array, then the value at index *`n`* of the array can be referred to as *`p`*`.all[`*`n`*`]`, or using the abbreviation *`p`*`[`*`n`*`]`.

The attribute `'access`, when applied to a variable (or a field of a variable) of type *`t`*, yields a value of type `access` *`t`* that addresses that variable (or field), unless the variable is constant, in which case the type is `access constant` *`t`*.

A value of type `access` *`t`* cannot be assigned to a variable or field of type `access` *`u`* unless *`t`* is the same type as *`u`*; this rule also applies to variants of access types listed above with `constant`, `managed` and `or null`.

A value of type `access` *`t`* can be assigned to a variable or parameter of type `access or null` *`t`*. However, a value of type `access or null` *`t`* can only be assigned to a value of type `access` *`t`* if it can be established by analysis that it is not `null` at the point of the assignment. This works in the same manner as for analysis of integer assignments, except that the only values are `null` and not-`null`. See section 44. A test could be something like `if a /= null then … end if`.

[7]This is done by reference-counting, not by garbage collection.

Similarly, a value of type `access or null t` can only be dereferenced if it is known that it is not `null` at the point of dereferencing. If a type can never be `null` by design, then it can be declared appropriately and tests will not be required.

The same conditions apply to assignments and dereferences of managed access values, and variants with `constant`.

A value of type `managed access t` cannot be assigned to a variable of type `access t`, and vice-versa. The same rule applies to variants with `constant` and `or null`. However, the attribute `'access` can be used to obtain an unmanaged access to the object to which a managed access value refers.

A value of type `access constant t` cannot be assigned to a variable of type `access t`. The same rule applies to variants with `or null` and `managed`.

A value of type `access t` cannot be assigned to a global variable or a field of one, and a global variable or field of such a type cannot be passed as an `out`, `in out` or `final in out` parameter. These rules are intended to prevent dangling pointers to local variables that no longer exist.

A value of type `persistent access t` or `persistent access constant t` can only refer to a global variable. These values are created by applying the attribute `'persistent_access` to the variable. The `new` operator cannot be used to generate persistent access values, and the `free` statement cannot be applied to variables of persistent access types.

A value of type `persistent access t` cannot be assigned to a variable (or field) of a non-persistent access type, and vice-versa.

A value of type `access new t` can be assigned to a variable of type `access t`. However, the reverse is not allowed - this is to prevent attempts to free objects that were not allocated with `new` (but rather had `'access` applied to them).

The checks on assignment compatibility that are described above can be suppressed by enclosing the relevant statements in a block with an `unchecked_access_conversion` exemption (40). This exemption does not affect the test that an access value must be known not to be `null` in order to dereference it, however (the `advise` statement can be used for that purpose).

7.1 Attributes of access types

For an access type `t`, the term `t'accessed_type` means the type accessed by `t`. For a variable or formal parameter `v` of an access type `t`, `v'accessed_type` means the type accessed by `t`.

7.2 Compatibility with C

Unmanaged access values are compatible with C pointers to the accessed type. Managed access values are not, but a C pointer to the addressed object can be obtained, as described above.

Examples

```
type Sensor_array_access is managed access Sensor_array;
type Report_access is access or null Report;

a : Sensor_array_access;
```

```
…
a := new [range x => 1];

Report1 : Report;
r : Report_access;
...
r := Report1'access;

type t is record
  n1 : number;
  c1 : character;
end record;
a : access new t;
...
a := new (n1 => 1, c1 => 'a');
...
free a;

type a is access or null constant array [range 1..10] of character;
```

8 The type address

The predefined type `address` is assignment- and parameter- compatible with all unmanaged access types[8], except that an address value cannot be assigned to a variable or parameter that is not marked `or null`. Within the scope of an `unchecked_access_conversion` exemption, however, type `address` is compatible with all access types.

9 Variables

9.1 Declarations

Variables can be declared in the declaration sections of procedures and functions, in which case they are referred to as *local variables*, or in the declaration sections of packages, in which case they are referred to as *global variables*. Local variables exist for the lifetime of a procedure or function invocation. Global variables exist for the lifetime of the program.

An uninitialised variable declaration has the form

```
variable_name : type_identifier;
variable_name1, variable_name2, …, variable_namen : type_identifier;
```

or

```
variable_name : type_specification;
variable_name1, variable_name2, …, variable_namen : type_specification;
```

This declares variables that do not initially have values. The first form takes the name of a type that has already been declared. The second form declares a type anonymously; in the case of a multi-variable declaration, only one type is created and used for all of them.

An initialised variable declaration has the form

```
variable_name : type_identifier := expression;
```

[8]It is similar to C 'void *'.

or

```
variable_name : type_specification := expression;
```

This declares a variable and assigns to it the value returned by evaluating the expression. In this case the variable cannot subsequently be modified. In the case of a local variable, the variable will maintain the value for the lifetime of the procedure or function call (it might take different values in subsequent calls); in the case of a global variable, it will maintain the value for the lifetime of the program. Only one variable name is allowed in this case in order to avoid ambiguity (and only one is likely to be useful).

Type_identifier can be the name of a type, or an attribute of a variable or a type that identifies a type. For a variable *v* of type *t*, the term *v'type* means the type *t*. Additional attributes apply to array, record and access types - these are described in the relevant sections.

Variable declarations can also take a number of prefixes that specify their visibility and usage. They will be covered in the appropriate sections below.

Examples

```
DailyReport : Report;
OverallLength : range 1..100 unit Metres;
Average_value : Some_numeric_type := Calculate_average(Some_array);
LocalMean : range for (A + B) / 2 := (A + B) / 2;
RegisterSet, AlternateRegisterSet : record
  AReg  : register_8;
  BReg  : register_8;
  IXReg : register_16;
end record;
WorkingCopy : SomeParameter'type;
```

9.2 Obtaining the address of a variable

The address of a variable can be retrieved with the attribute `'access`. (see 25).

Examples

```
P(X'access);
```

9.3 Obtaining the size of a type

The size of a type in basic addressable storage units (usually 8-bit bytes), equivalent to C `sizeof`, can be retrieved with `'size`. Unlike other attributes such as `'first`, `'size` is not static, and its use in an expression will cause that expression to be non-static. This is because the actual size in storage units is not known to the compiler, which instead writes a target expression for it.

```
type T is …
a : some_array_type;
S1 := T'size;
S2 := A'element_type'size;
```

10 Procedure declarations

Basic procedure declarations have the form

```
procedure proc_name(formal_parameter_list) is
  local_declaration1;
   …
  local_declarationm;
begin
  statement1;
  …
  statementn;
end proc_name;
```

The local declarations can include the declarations of types, units, named constants, and initialised and uninitialised variables.

The formal parameter list consists of parameter declarations separated by semicolons:

```
parameter1; parameter2; … parametern
```

The formal parameter list can be empty, in which case the surrounding parentheses can be omitted.

Each parameter specification has the form[9]

```
formal_name : mode type_name
formal_name1, formal_name2, …, formal_namen : mode type_name
```

or

```
formal_name : mode type_declaration
formal_name1, formal_name2, …, formal_namen : mode type_declaration
```

`Formal_name` is the name of the formal parameter. `Type_name` is its type. `Mode` can be any of the following:

- `in` - the parameter is an input to the procedure and is read-only within it
- `out` - the parameter is an output from the procedure and must be completely assigned (i.e. initialised) by the statements of the procedure; it cannot be read by statements in the procedure until it has been assigned
- `in out` - meaning that the parameter is both an input and an output; it has a value at the time of the call and can be read before being assigned, and the procedure may (but is not required to) make assignments to either all or part of it
- `final in out` – the parameter is an input, but the procedure must finalise it before it returns. Finalisation is described below.

If `type_declaration` is given instead of `type_name`, then an anonymous type can be used. Typically this is used for open array formal parameters (e.g. `p : array [range 1..100]> of t`), but it can be the declaration of other types as well. For numerical types, a unit can be given as usual. Representation clauses (apart from `access_mode`) and controlled type declarations cannot be used however.

The type of a formal parameter `f` of a procedure or function `p` can be obtained with `p'parameter_type(f)`.

[9]The access mode can be given – see 22.6

Parameter passing method

Parameters of mode `in` that are of numerical or access types are passed by value, i.e. a copy is passed.

Parameters of all structured types and all parameters of modes other than `in` are passed by reference, i.e. a pointer is passed.

11 Function declarations

Basic function declarations have the form

```
function fn_name(formal_parameter_list) : return_type is
  local_declaration1;
  …
  local_declarationm;
begin
  statement1;
  …
  statementn;
end fn_name;
```

`Return_type` gives the type that the function returns, which must be a numerical, boolean, string or access type. It can either name a type that has already been declared, or it can declare an anonymous type, similar to the case of formal parameters. Otherwise function declarations are similar to procedure declarations, except that the only parameter mode that is allowed is `in`.

The return value of a function is set by assigning to a virtual variable that has the same name as the function. This variable behaves like an `out` parameter in that it can be read and modified after it has first been assigned, and the function is required to initialise it before it exits.

The return type of a function *`f`* can be retrieved from outside the function with *`f`*`'return_type`. This is mainly useful with anonymous return types. Within the body of *`f`*, however, *`f`* refers to the variable that receives the return value, not to the function itself, so `'return_type` is not applicable to it (*`f`*`'type`, *`f`*`'unit` etc. can be used).

Examples

```
function Average(Value1 : in Data_type; Value2 : in Data_type) : Data_type is
begin
  Average := (Value1 + Value2) / 2;
end Average;
```

12 Renaming declarations

A renaming declaration does not declare a new object, but a name for an existing one. For a variable:

```
new_name : type_name renames object_reference;
```

`Object_reference` can refer to a whole variable, a record component, an array element or the object referenced by an access variable[10]. Since such a declaration establishes an alias, writing to

[10]In the case of naming whole variable, a simple reference translation with no runtime implications is established. In the case of a field, component or access reference, a pointer is created and references to the new name become pointer

the referenced object, other than through *new_name*, is prohibited within the scope of *new_name*. Any procedure call within the scope of *new_name* is also checked for indirect aliases, i.e. references to the variable that contains the referenced entity[11]. This check can be suppressed with a 'using' clause:

```
new_name : type_name renames object_reference using (exemption => unchecked_alias);
```

A renaming declaration can also declare a new name for a package, subsystem, procedure, function, type, unit or constant:

```
new_name : package renames package_reference;
new_name : subsystem renames subsystem_reference;
new_name : procedure renames procedure_reference;
new_name : function renames function_reference;
new_name : type renames type_reference;
new_name : unit renames unit_reference;
new_name : constant renames constant_reference;
```

Examples

```
Max_temperature : Temperature_type renames Sensors[max].Sensor_data.Temperature;
P : package renames x.y.z;
```

13 Statements

Statements are terminated by semicolons. Statements are executed in the sequence that they appear in the source (reading from top to bottom and left to right), except where otherwise noted.

13.1 Assignment statements

Assignment statements assign values to variables.

13.1.1 Assignments to basic variables

Assignments to variables of integer, floating-point and boolean types take the form

```
variable_reference := expression;
```

Variable_reference can identify a simple variable, or a component of a record or array, possibly indirectly through an access value.

Examples

```
X := 10;
Y := X * 2 + 1;
rec.n := 1;
A[Y] := fn(Z);
```

dereferences.

[11] Alias checking is intended to prevent analysis with the new name being invalidated by writes to the base object, and related subtle bugs.

```
Daily_report := new Report;
Daily_report.Timestamp := Current_time;
```

13.1.2 Assignments to records

A record can be assigned from a variable of the same type, which simply copies the entire record (a *copy assignment*):

```
Rec1 := Rec2;
```

A record can also be assigned using a *whole record assignment*, which names and assigns a value to every field:

```
Rec := (field1 => expression1, field2 => expression2, …, fieldn => expressionn);
```

In this case *expression*$_1$ is evaluated and assigned to *field*$_1$, *expression*$_2$ to *field*$_2$ and so on. Every field must be given. The expressions can be the following:

- An arithmetic or logical expression that evaluates to a value of a basic type
- The right hand side of a whole record assignment
- The right hand side of a whole array assignment
- A variable of a record type (indicating a copy assignment)
- A variable of an array type (indicating a copy assignment)
- An expression that allocates an object with `new` and returns an access type
- A call to a *closed procedure*[12] that sets a value of the field type as an out parameter

In the last case, the actual parameter that corresponds to the field is named `this`. This case allows for a structured type that is generated by a procedure, since structured objects cannot be returned as function results.

Whole-record assignments to union and unchecked union variables use the same syntax, but assign to one, rather than all, of the type's fields.

In the case of null fields, the syntax is *(*`field =>`*)*.

Examples

```
Previous_sensor_record := Current_sensor_record;
Current_state := (Initialised => false, Cycle => 0);
Daily_report.all := Completed_report;
Allocated_state.all := (Initialised => true, Cycle => 1);
NewValue := (NotAvailable =>);
Var1 := (Int1 => 1, GenerateRec(Rec1));
```

13.1.3 Assignments to arrays

An array can be assigned from another variable of the same type (a *copy assignment*):

```
Arr1 := Arr2;
```

An array can also be assigned with a *whole array assignment*:

[12] See 14.3

```
Arr := [range index_variable => expression];
```

Index_variable is a variable of the index type of *Arr* that takes values over *Arr*'s index range. *Index_variable* is declared by the assignment statement itself; its scope is *expression*. For each value of *index_variable*, *expression* is evaluated and the result is assigned to the corresponding element of the array. The expression is of the types described above for whole-record assignments.

Alternatively, a whole-array assignment can name every element:

```
Arr := [index₁ => expr₁, index₂ => expr₂, …];
```

In the second form, the abbreviation

```
index₁ | index₂ | … | indexₙ => expr
```

is equivalent to

```
index₁ => expr, index₂ => expr, …, indexₙ => expr
```

However, in order to minimise the consequences of mistakes of interpretation (in particular, whether `expr` is evaluated once or *n* times), `expr` in this latter case (`index₁ | … | indexₙ => expr`) is restricted to simple static expressions, such as could be used when defining constants.

A whole array assignment can also list the values in increasing order of array index without stating the indexes explicitly:

```
Arr := [expr₁, expr₂, … exprₙ];
```

The number of entries listed must be equal to the range of the index.

See also section 25 for details about assigning to and from array slices.

Examples

```
Accumulated_values := [range j => 0.0];
Squares := [range n => n * n];
Report_records := [range n => (ID => Compute_ID(n), Timestamp => Current_time)];
Report_record_accesses :=
  [range n => new (ID => Compute_ID(n), Timestamp => Current_time)];
Matrix => [range row => [range column => Initial_value(row, column)]];
Report_records := [range n => Generate_report(n, this)];
Initial_values := [deviceA => 1, deviceB => 2, deviceC | deviceD => 3];
```

13.1.4 Assignments to access variables

An access variable can be assigned directly from another variable of the same type:

```
Access1 := Access2;
```

It can also be assigned with an allocator expression:

```
Access1 := new expression;
```

where *expression* yields a value of the accessed type. *Expression* can be

- An arithmetic or logical expression that yields a value of a basic type
- The right hand side of a whole record assignment
- The right hand side of a whole array assignment
- A variable of a record type (indicating a copy assignment)
- A variable of an array type (indicating a copy assignment)

Examples

```
Int_access := new 2 + 3;
Report_access := new (Timestamp => Current_time, Status => Start_report);
Matrix_access := new [range y => [range x => 0.0]];
Matrix_access := new Identity_matrix;
```

13.1.5 Rules for assignments of numerical values

1. In an assignment to an object of integer or floating-point type, the unit of the value being assigned must match the unit of the target of the assignment.
2. In an assignment to an object of integer type, the range of possible values of the quantity being assigned must provably fall within the range of the target of the assignment.
3. An integer can be assigned to a floating-point variable, provided that the floating-point variable is of sufficient precision and magnitude to accommodate the range of the integer
4. When a floating-point value is assigned to a floating-point variable, the target of the assignment must be of at least equal magnitude and at least equal precision as the value
5. In the case of a named floating-point constant, the magnitude is calculated by the compiler and the precision is taken to be 1 digit for the purpose of assignment, so it can be assigned to any floating-point variable of sufficient magnitude
6. A floating-point value cannot be assigned to an integer (but see below).

13.1.6 Floating-point to integer conversions

The following generic package (see section 60) exists for conversions from floating-point values to integers.

```
generic
  type Ftype is digits;
  type IType is range;
package Floating_point_conversions is

  // Truncate floating-point value F to integer value I; if F is out of the range
  // of I then set I to its maximum or minimum value (depending on whether F is
  // greater than the maximum or less than the minimum respectively) and
  // out_of_range to true; otherwise out_of_range will be set to false.
  public procedure Floor(F : in FType; I : out IType; out_of_range : out boolean) is
    ...

  // Round floating-point value F to integer value I; if F is out of the range
  // of I then set I to its maximum or minimum value (depending on whether F is
  // greater than the maximum or less than the minimum respectively) and
  // out_of_range to true; otherwise out_of_range will be set to false.
  public procedure Round(F : in FType; I : out IType; out_of_range : out boolean) is
    ...

  // Truncate floating-point value F to integer type I without checking for overflow
  public function Unchecked_floor(F : in FType) : IType is
    ...

  // Round floating-point value F to integer type I without checking for overflow
```

```
  public function Unchecked_round(F : in FType) : IType is
    ...

end Floating_point_conversions;
```

13.2 If statements

If statements have the general form

```
if boolean_expression1 then
  statements1
elsif boolean_expression2 then
  statements2
…
elsif boolean_expressionn then
  statementsn
else
  statementselse
end if;
```

Boolean_expression$_1$ is evaluated first; if it is true then the sequence of statements *statements*$_1$ is executed, and control then passes to the statement following `end if`. Otherwise, *boolean_expression*$_2$ is evaluated; if it is true then *statements*$_2$ is executed and control then passes to the statement after `end if`. Otherwise tests proceed from the next boolean expression, and so on. If none of the boolean expressions evaluates to true, then the sequence *statements*$_{else}$ is executed. The `elsif` and/or `else` parts can be omitted. If there is no `else` part, then it is possible that the `if` statement will execute no statements.

Examples

```
if x < 0 then
  RecordVal(Negative_value);
  y := -x;
elsif X = 0 then
  RecordVal(Zero_value);
  y := 0;
else
  RecordVal(Positive_value);
  y := x;
end if;
```

13.3 Do statements

A *do* statement has the form

```
do
  statement1;
  …
  statementn;
end do;
```

This statement executes the sequence *statement*$_1$ through *statement*$_n$ once. The purpose of a `do` statement is to allow transfer to the end of the `do` statement from before the end of the statement list, using an `exit` statement:

```
exit when boolean_expression;
```

If *boolean_expression* evaluates to `true`, then control proceeds immediately to the statement that

follows `end do`. An `exit` statement can only appear as one of the statements *statement*$_1$ through *statement*$_{n-1}$ of the body of the `do` statement (the last statement, *statement*$_n$, cannot be an `exit`). The `exit` statement cannot be nested inside another statement (such as an `if` statement). Consequently, there is no use for an unconditional `exit` statement. The statement list can contain more than one `exit` statement.

A `do` statement performs a similar function to `elsif` – it allows structures that might otherwise be written as nested `if` statements to be expressed in a flatter and more readable way.

13.4 Loop statements

Loop statements come in three varieties: indefinite loops, while loops and for loops.

13.4.1 Indefinite loops

An indefinite loop has the form

```
loop
  statement1;
  …
  statementn;
end loop;
```

This loop executes the sequence *statement*$_1$ through *statement*$_n$ repeatedly. To exit an indefinite loop, use an `exit` statement:

```
exit when boolean_expression;
```

If *boolean_expression* evaluates to `true`, then control proceeds immediately to the statement that follows `end loop`. The same restrictions on the placement of `exit` statements apply to `loop` statements as to `do` statements, except that the last statement, *statement*$_n$, can be an `exit` in the case of a loop.

An indefinite loop can be restarted before the end of the loop body with a `repeat` statement:

```
repeat when boolean_expression;
```

If *boolean_expression* evaluates to `true`, then control transfers immediately to *statement*$_1$ at the top of the loop body. A `repeat` statement can only appear as one of the statements *statement*$_1$ through *statement*$_{n-1}$ of the body of the `loop` statement (the last statement, *statement*$_n$, cannot be a `repeat`). The loop body can contain more than one `repeat` statement, and it can contain both `repeat` and `exit` statements.

13.4.2 While loops

A while loop is of the form

```
while boolean_expression loop
  statement1;
   …
  statementn;
end loop;
```

It loops as long as *boolean_expression* evaluates to `true` at the top of the loop·

`Exit` and `repeat` statements are not permitted in the statement list *statement*$_1$ through

statement$_n$ in the body of a `while` loop. The only way to exit a `while` loop is for *boolean_expression* to evaluate to `false`; the entire loop body is run in each iteration.

13.4.3 For loops

A for loop has the form

```
for control_variable : control_type in low_value..high_value loop
  statement1;
   …
  statementn;
end loop;
```

For the common special case

```
for control_var : control_type in control_type'first..control_type'last loop
```

the following abbreviation is provided:

```
for control_var : control_type loop
```

For the case where the type of the control variable is not relevant outside the loop, and the loop range is defined at compile-time, an anonymous type declaration in the `for` statement can be used:

```
for control_var : range low_value..high_value loop
```

In all forms, *control_var* is a variable of integer type *control_type* that is created for the purpose of executing the `for` loop. Its scope is the body of the loop; its lifetime is the duration of the loop. *Control_var* successively takes values from *low_value* through *high_value* with an increment of 1, each time executing the statements in the loop body. *Low_value* and *high_value* are expressions that yield values that are compatible with *control_type*, i.e. they are in its range and have the same unit; these expressions are both evaluated once before the loop body is executed. If *low_value* is greater than *high_value* then the loop body is not executed. `Exit` and `repeat` statements are not permitted in the sequence *statement*$_1$ through *statement*$_n$ in the body of a `for` loop.

13.5 Case statements

A *case* statement selects between a number of alternatives:

```
case expression is
  when case_set1 => statements1
  when case_set2 => statements2
   …
  when case_setn => statementsn
end case;
```

The expression must be of an integer type (typically an enumerated type). Each *case_set* lists one or more values of that type, separated by `|` (vertical bar) characters. If any of the values in the *case_set* equals the value of the expression, then the corresponding sequence of statements is executed; control then passes to the statement following `end case`.

The *case_sets* in the statement must include every possible value of the type of the expression at the point in the program where the `case` statement appears, without any duplicates. There are no ranges or default options, because these can mask the accidental omission of a value.

If the expression is a simple identifier, then assignments to it are prohibited within the body of the `case` statement.

Example

```
type Sensor_state is (Uninitialised, Waiting, Processing, Value_available);

…

case Sensor1State is
  when Uninitialised =>
    RecordState(Current_time, Uninitialised);
    Initialise_sensor;
  when Waiting | Processing =>
    RecordState(Current_time, Sensor1State);
  when Value_available =>
    Report_value(Input_value);
    Reset_sensor;
end case;

if Sensor1State /= Uninitialised then
  case Sensor1State is
    when Waiting | Processing =>
      RecordState(Current_time, Sensor1State);
    when Value_available =>
      Report_value(Input_value);
      Reset_sensor;
  end case;
end if;
```

13.6 Procedure call statements

Procedure call statements name the procedure to be called and supply actual parameters. There are two forms of procedure call syntax: *positional notation* and *named notation*.

Positional notation lists the actual parameters in the same order as the formal parameters of the procedure definition:

```
proc_name(actual1, actual2, …, actualn);
```

Named notation explicitly names the formal parameters that the actual parameters correspond to. In this case the order does not have to be the same as that of the procedure definition (but all parameters must be named).

```
proc_name(formal_name1 => actual1, formal_name2 => acual2, …, formal_namen => actualn);
```

If the procedure has no parameters, then the parentheses can be omitted:

```
proc_name;
```

or

```
proc_name();
```

Named notation can always be used. Named notation *must* be used if there a possibility that parameters could be supplied in the wrong order, based on the types of the formal parameters (for example, if there are two integer parameters with the same unit). In the rare case that this restriction

might cause a problem, it can be overridden by enclosing the procedure call in a block (40) that specifies an `unchecked_ambiguous_order` exemption.

The parameter mode (`in`, `out`, `in out`, `final in out`) can optionally be written before any of the *actual*$_j$. If it is present, then it must match the mode of the formal parameter.

Corresponding to a formal parameter of mode `in` that is of a numeric, boolean, string or access type, the actual parameter can be an expression that yields a value that is compatible by type and unit (where relevant). For formal parameters of all other modes and all other types, the actual parameter must name a variable that must be compatible by type and unit (where relevant). The variable can, in general, be a simple local or global variable, a record component, an array element, or the object referenced by an access value. This can however be subject to usage restrictions that are described later.

Parameter aliasing is not allowed – the same variable (or parameter) cannot be used for two or more actual parameters to the same call. Indirect aliasing is also prohibited – a variable cannot be passed to a procedure or function that accesses it as a global, either itself or in a procedure or function that it calls. The aliasing check can be suppressed for individual parameters by appending the modifier `using (exemption => unchecked_alias)` to the actual parameter.

Range checking can be suppressed for the expression used for an actual `in` parameter by appending `using (exemption => unchecked_range)` to it. This is preferable to enclosing the entire call in a block that applies that exemption if it is only required for the parameter.

Examples

```
Generate_summary(Current_report, Output_log);
Generate_summary(Report => Current_report, Log => Output_log);
Generate_summary(in Current_report, in out Output_log);
Generate_summary(Report => in Current_report, Log => in out Output_log);

Normalise_reading(Input      => Sensor_reading,
                  Normalised => Adjusted_reading,
                  Scale      => Base_scale + 1);

begin using (exemption => unchecked_ambiguous_order)
  Normalise_reading(Sensor_reading, Adjusted_reading, Base_scale + 1);
end;

Proc1(P1 => Var1  using (exemption => unchecked_alias),
      P2 => Var1  using (exemption => unchecked_alias),
      P3 => 2,
      P4 => X + 1 using (exemption => unchecked_range)
     );
```

13.7 Blocks

Blocks allow local declarations at the statement level, and modifiers (usually exemptions) to be applied to statements.

```
declare
  local_declaration1;
  …
  local_declarationm;
begin using (modifier => modifier_value, ...)
  statement1;
```

```
  …
  statement_n;
end;
```

The local declarations can include the declarations of types, units and named constants, initialised and uninitialised variables, and `renames` declarations. The scope of the declarations is the statement list between `begin` and `end`. If there are no local declarations, then `declare` can be omitted. If no modifiers are required, then the `using` clause can be omitted. Specific modifiers are described where relevant elsewhere in this document.

Examples

```
if x > 5 and x < 20 then
  declare
    type comp_type is range for x * 2;
    tmp1 : comp_type;
  begin
    tmp1 := x * 2;
    if tmp1 < y'last then
      y := tmp1;
    else
      y := 0;
    end if;
  end;
  proc1(y);
  //
  begin using (exemption => unchecked_side_effect)
    g[2] := 0;
  end;
end if;
```

13.8 The null statement

The `null` statement is used in contexts where a statement is required, but there is nothing to do.

Examples

```
procedure p is
begin
  null;
end p;
```

```
case v is
  when On      => Turn_off(v);
  when Waiting => Turn_on(v);
  when Idle    => null;
end case;
```

13.9 The free statement

An object that has been allocated through an unmanaged access variable is not automatically freed when no more references to it exist. If it is appropriate to free it, the `free` statement is used:

```
free access_variable;
```

Examples

```
a1 : access character;
```

```
a1 := new 'x';
```

```
…
```

```
free a1;
```

13.10C statements

An embedded C statement line is marked by a `!` character as the first non-whitespace character of the line and extends to the end of the line. A C statement line is a C code literal that is included in the generated program. At least one whitespace character must separate the `!` from the code. C statements can span multiple lines, provided that every line is introduced with `!`. A block of such lines, with no other lines intervening apart from comments and whitespace, will be surrounded by braces in the generated program in order to reduce the potential for syntactic chaos in the event of a mistake. A C statement line can also be introduced by `!c`.

A C header line is introduced by `!h`. It is written to the generated package header file rather than the .c file.

Embedded C lines can refer to entities that are declared in the non-C parts of the source. Within a C line, the notation `` `entityname` ``, where *entityname* the name of an entity in the non-C source, will be replaced by the corresponding C name. Backquotes can be escaped with `\` to suppress this.

`pragma system_header("line1", … "linen")` writes lines to the header file of the system, which is included by all generated C modules.

Pragma `include` is a convenient way to include header files:

```
pragma include(file1, file2,...);
```

Each of the $file_i$ is a string; if the first character of the string is <, then the C include statement will not have surrounding quotes.

Examples

```
! f = fopen("report.txt", "w");
for line_num in Report_text'first..Report_text'last loop
  Prepare_line(Report_text[line_num]);
! fputs(f, `Report_text`[`line_num`]);
end loop;
! fclose(f);

! printf("Count=%d\n", `Updates`.Count);

!h #define MAXIMUM_RECORDS_SYM `MaximumRecords`

pragma system_header("#include <inttypes.h>", "#include \"interfaces.h\"");

pragma include("<stdio.h>", "something.h");
```

14 Expressions and operators

14.1 Operators

The following operators exist:

Operator symbol	Operation	Applies to	Result type
`+`	Addition	Integer and floating-point types	Integer or floating-point
`-`	Subtraction or unary negation	Integer and floating-point types	Integer or floating-point
`*`	Multiplication	Integer and floating-point types	Integer or floating-point
`/`	Division	Integer and floating-point types	Integer or floating-point
`mod`	Remainder after division (C %)	Integer types	Integer
`=`	Equality	Integer, boolean and access types	Boolean
`/=`	Inequality	Integer, boolean and access types	Boolean
`<`	Less than	Integer and floating-point types	Boolean
`<=`	Less than or equal	Integer and floating-point types	Boolean
`>`	Greater than	Integer and floating-point types	Boolean
`>=`	Greater than or equal	Integer and floating-point types	Boolean
`and`	Logical and	Boolean	Boolean
`or`	Logical or	Boolean	Boolean
`not`	Logical not	Boolean	Boolean
`iand`	Integer bitwise and	Non-negative integer types	Integer
`ior`	Integer bitwise or	Non-negative integer types	Integer
`ixor`	Integer bitwise xor	Non-negative integer types	Integer

Exact equality and inequality are not defined for floating point types because floating point values are generally approximations[13].

Operator precedence

The precedence of operators, in order of highest to lowest, is:

1. Unary `-`
2. `*`, `/`, `mod`

[13]The appropriate formulation for floating-point equality is usually abs(x – y) < epsilon, where epsilon is a relatively small number, the value of which depends on the details of the application.

3. +, binary -
4. iand, ior, ixor
5. =, /=, <, <=, >, >=
6. and, or, not

The logical operators `and`, `or`, `not` are at the same precedence level as each other but require explicit parentheses when mixed in (sub)expressions, e.g. `(a and b) or (c and d) or e` rather than `a and b or c and d or e`.

Similarly, the bitwise operators `iand`, `ior`, `ixor` require explicit parentheses when mixed, e.g. `(a iand b) ior c` rather than `a iand b ior c`.

There is no bitwise 'not' operator because it is not possible in general to compute a definitive range for one. Bitwise xor can be used to achieve the same effect but with a definite range of values.

The arithmetic operators +, binary `-`, `*`, `/`, `mod` are left-associative, so, for example, `7 - 2 - 1` is equivalent to `(7 - 2) - 1`.

The bitwise operators `iand`, `ior`, `ixor` are also left-associative.

14.2 Rules for evaluation of expressions

14.3 Functions, closed functions and closed procedures

15 Analysis of integer-valued objects

15.1 Purpose and summary

For integer-valued variables, including formal parameters, the compiler tracks the following quantities:

- The ranges of values that individual variables can have at any point in a procedure or function
- The ranges of values that the differences between pairs of variables can have at any point in a procedure or function

These ranges are affected by:

- The types of the variables
- Assignments and parameter passing in procedure calls
- Enclosing `if`, `do`, `loop`, `while`, `for` and `case` statements
- `advise` statements
- preconditions and postconditions of procedures and functions

Analysis of ranges serves two purposes:

- To establish that the expression in an assignment statement or a procedure or function call is within the range of the target variable or formal parameter. This check can be suppressed by enclosing a statement within a block that specifies an `unchecked_range` exemption.
- To determine whether an expression can be computed without overflow, and what type casts are required in order to compute it.

When analysing a procedure or function, the ranges of variables (including globals) are set to the ranges of their types before the first statement of the procedure body, except where ranges are restricted by preconditions (164).

The ranges of record fields, array components and dereferenced objects is always taken to be the full range of their type, except that renaming declarations (31) can be used to identify such fields, components and objects in order to enable more detailed analysis with them.

15.2 The range of expressions

The range of an integer-valued expression is determined by the ranges of its sub-expressions combined according to the operators that connect them, in such a manner that the computed range of the combined sub-expressions encompasses the maximum range that could be obtained at that point in the program, and no more.

The range of a function call is the range of the return type of the function.

15.3 Effects of statements

15.3.1 Assignment statements

An assignment to a formal parameter or a variable updates the object's range according to the range that is determined by analysis of the expression that is assigned.

15.3.2 Procedure call statements

If a variable is passed as an `out`, `in out` or `final in out` parameter to a procedure, then its range after the call is set to the full range of the formal parameter type, except where this is modified by a postcondition.

A call to a procedure causes the ranges of all global variables that it modifies, directly or indirectly, to be reset to the full ranges of their types at that point in the calling routine, except where this is modified by a postcondition.

15.3.3 If statements

The condition expressions in `if` statements work as follows.

A relational term that is of the form '*simple-variable relop expression*' is said to be analysable. A simple variable is one that does not contain array, field or pointer dereferences (only integer types are relevant here). Renaming declarations that refer to simple integer-valued objects count as simple variables.

A *relop* is one of =, /=, <, <=, >, >=.

Examples

```
x > 0
y > x + 1
```

A conditional expression that consists of only a single analysable relational term, or that consists of a number of analysable relational terms or conditional expressions connected by `and`, or that is the logical negation of a group of analysable terms or conditional expressions that are connected by `or`, is analysable.

Examples

```
x > 0 and y > 2
x > fn(2) and y <= z * t - 2 and p /= q
```

```
not (x < 1)
not (x > 0 or y > 1)
x > 0 and y > x + 1
```

In an `if` statement of the form

```
if conditional_expression then true_part else false_part end if
```

if *`conditional_expression`* is analysable, then the terms in *`conditional_expression`* are analysed from left to right, with the ranges being updated at every stage, and the resulting ranges are applied during the analysis of *`true_part`*. If the logical negation of *`conditional_expression`* is analysable, then it is analysed similarly and the resulting ranges are applied in the analysis of *`false_part`*.

An if statement of the general form

```
if cond1 then part1 elsif cond2 then part2 elsif ... else partn end if
```

is analysed as if it were an equivalent set of nested if statements:

```
if cond1 then part1 else if cond2 then part2 else if ... else partn end if end if end if
```

Analysable terms can have the effect of restricting the range that the variable on the left-hand-side can take, or else have no effect on it. For example, if we know that z > 0 and z <= 10, then

- `x > z` implies that *`x > 1`* (since z must be at least 1)
- `x <= z * 2` implies that *`x <= 20`*

If function 'fn' returns a value of type Counter that has range 1..10, then

- `x > fn(y)` implies that *`x`* is at least 2.

In addition, analysable terms that have the more restricted form

```
variable1 relop variable2 [+/- expression]
```

can have the effect of updating the range of values that the difference between *`variable1`* and *`variable2`* can take. For example

- `x > y` implies that *`x - y > 0`*. If we also know that `y >= 5`, then this term also has the effect that *`x > 5`*.
- `x <= y - 10` implies that *`x - y <= -10`*. If we also know that `y <= 50`, then this term also implies that *`x <= 40`*.

Negated expressions work at expected. So

`not (x > 1 or y >= z)` is equivalent to `not(x > 1) and not(y >= z)`

which is equivalent to `x <= 1 and y < z`

At the conclusion of an `if` statement that has an `else` part, any variable that has its range restricted in both the `if` and the `else` parts will have its range updated to accommodate both of these new ranges. Any variable that is modified in one branch of an `if` statement will have its range set to a range that will encompass the range immediately before the `if` statement and the modified range.

15.3.4 Do statements

A `do` statement of the form

```
do
  statements1
  exit when condition1;
```

```
  statements2
  exit when condition2;
  statements3
end do;
```

is analysed as if it were

```
statements1
if not condition1 then
  statements2
  if not condition2 then
    statements3
  end if;
end if;
```

15.3.5 Loop and while statements

A `loop` of the form

```
loop
  statements1
  exit when condition1;
  statements2
  exit when condition2;
  statements3
  repeat when condition3;
  statements4
end loop;
```

is analysed as if it were

```
loop
  statements1
  if not condition1 then
    statements2
    if not condition2 then
      statements3
      if not condition3 then
        statements4
      end if;
    else
      exit;
    end if;
  else
    exit;
  end if;
end loop;
```

However, all variables that are assigned to or passed as `out`, `in out`, or `final in out` parameters within the loop have their ranges initialised to the full range of their types at the start of the loop. If

the loop contains a call to a procedure, then the ranges of all global variables that the procedure modifies (directly or indirectly) are reset to the full ranges of their types at the start of the loop. The same resetting of ranges is also applied immediately after the loop.

A `while` loop of the form

```
while condition loop
  statements
end loop;
```

is analysed as if it were

```
loop
  if condition then
    statements
  else
    exit;
  end if;
end loop;
assume (not condition)
```

The treatment of variables that are modified within the loop body is the same as for a `loop` statement. However, immediately after the loop, the negation of *`condition`* is assumed.

15.3.6 For loops

A `for` loop

```
for indexvar : some_type in L..H loop
  stamements
end loop;
```

is analysed as if it were

```
indexvar := L;
while indexvar <= H loop
  assume that indexvar >= L and indexvar <= H
  statements
  if indexvar /= H then
    indexvar := indexvar + 1;
  end if
end loop;
```

The treatment of variables that are modified within the loop body is the same as for a `loop` statement.

15.3.7 Case statements

In a `case` statement of the form

```
case simple_variable of
  when case1 => statements1
  …
  when casen => statementsn
end case
```

within every set $statements_j$, the range of *`simple_variable`* is restricted to the range implied by

case$_j$, which will be a single value if *case*$_j$ does not contain alternatives, and the smallest range that will encompass all the alternatives if it does. After the `case` statement, the ranges of any simple variables that are modified in all branches of the statement will be updated to the range that will just encompass the ranges that were determined in the branches. The ranges of any simple variables that are modified in some (but not all) of the branches will be updated to the range that will just encompass the range that was in force immediately before the statement and that resulting from any of those branches.

15.4 Analysis statements

An occurrence of the statement

`assert` *boolean_expression*$_1$`,` *boolean_expression*$_2$`, …;`

states that each *boolean_expression*$_j$, which must be either a static boolean expression or an analysable expression (15.3.3), should be true at that point; a compilation error results if the compiler cannot establish that it is true.

An occurrence of the statement

`advise` *boolean_expression*$_1$`,` *boolean_expression*$_2$`, …;`

states that each *boolean_expression*$_j$, which must be analysable, is in fact true at that point; the compiler is instructed to assume that it is true. This statement can be used where the compiler's own analysis is not able to establish the fact. However, the advice must be plausible; it is an error to assert something that is provably false. For example, if `V`, which is of a type with range 0..9, is known to be in the range 1..5 at a particular point, then `advise V = 2` is allowed at that point, but `advise V = 0` is not.

Pragma `unchecked_reset(`var$_1$`, `var$_2$`, …)` tells the compiler to forget what it knows about the ranges of var$_1$, var$_2$ etc. and assume that they can take the full ranges of their types. This could be used, for example, prior to `advise` in order to allow the assertion `V = 0` in the previous example.

Pragma `runtime_check(`*boolean_expression*$_1$`, `*boolean_expression*$_2$`, ...)` has the same effect as `advise`, but also writes a runtime check of the expressions into the generated program; if this check fails at runtime then a (sub)system restart is performed (60). Pragma `runtime_check` cannot be used in package initialisation and finalisation sections, or in any routine called by them.

Pragma `test_assert(`*boolean_expression*$_1$`, `*boolean_expression*$_2$`, …)` can appear at any point at which a statement can appear. Under test builds it will be compiled into a series of `if` statements that test each boolean expression (these can be general boolean expressions); if an expression evaluates to false then the program will fail and display an error message that identifies the test that failed. Under normal (non-test) builds, pragma `test_assert` does nothing (it must, however, have correct syntax). Test builds also automatically include numerous other runtime checks. These are mainly for the purposes of testing the compiler's own analysis and testing for errors in `advise` statements.

16 Packages

16.1 Declaration and visibility

Packages are the basic units of program structure. A package declaration has the form:

```
package package_name is

  declaration_list

begin
  initialisation_statement_list
```

```
final
  finalisation_statement_list
end package_name;
```

The initialisation (after begin) and finalisation sections are optional syntactically, but might be required for particular declaration lists.

Declaration_list can include the declarations of units, types, named constants, variables, procedures, functions and packages[17], and can include certain pragmas.

Packages regulate the visibility of entities that are declared within them:

1. Any entity that is declared in *declaration_list* will be visible from that point on within the package in which it is declared, including in any packages that are nested within it.
2. No entity in *declaration_list* will be visible outside the package in which it is declared unless its declaration is prefixed with the keyword `public`.
3. If a package *s* is nested within a package *p*, then an entity *e* that is declared public within *s* will be exported from package *p* (as *p.s.e)* if and only if package *s* is itself declared public (i.e. `public package s is ...`).
4. For a type declaration, prefixing the declaration with `public` (before `type`) causes the existence of the type to be visible outside the package, but not its structure. Prefixing the type definition (after `is`) with `public` as well causes the type's structure to be visible[18].
5. A global variable can be declared with the prefix `public out`, which is like `public` except that the variable cannot be assigned to from outside the package in which it is declared.
6. Within the package *p* in which an entity *e* is declared (including within sub-packages of *p*), it is referred to by the simple name *e*. Within a package *q* inside of which package *p* is declared, it is referred to as *p.e*. Within a package *r* in which package *q* is declared, it is referred to as *q.p.e*, and so on.

The following declares a public type with private structure:

```
public type type_name is type_definition;
```

Outside the package in which type *type_name* is declared, it is possible to declare variables, procedure and function parameters, record fields and array components of type *type_name*, and types that access *type_name*, to allocate objects of that type with `new`, and to pass parameters of that type, but it is not possible to perform any other operation that would require knowledge of *type_definition*, such as performing arithmetic or logical operations (for numeric and boolean types), accessing fields (for record types) or components (for array types), or dereferencing (for access types). It is not possible to perform assignment of objects of type *type_name* (i.e. make a copy). Assignment can only be done within the package in which *type_name* is defined; the package can of course export a copy procedure. Outside the package, assignments to variables and components of type *type_name* can only be done by passing them as parameters to procedures within the package. If a function within the package returns a value of type *type_name*, a call to this function cannot be used outside the package in an expression, including as the right-hand-side of an assignment statement.

The following declares a fully visible type:

[17]Including subsystems

[18]'public type *t* is *definition*' is analogous to an Ada limited private type; 'public type *t* is public *definition*' corresponds to an Ada public type. C does not have equivalent concepts.

```
public type type_name is public type_definition;
```

In this case both the existence and the structure of the type are visible outside the package and the full range of operations on objects of that type can be performed.

Example

```
package p is

  type t1 is range 1..2;
  // type t1 is visible here

  package q is
   // type t1 is visible here

   type t2 is range 2..3;
   // types t1 and t2 are visible here
   public type t3 is range 3..4;
   // types t1, t2 and t3 are visible here
   public type t4 is public range 4..5;
   // types t1, t2, t3 and t4 are visible here

   package r is
     // types t1, t2, t3 and t4 are visible here
     type t5 is range 5..6;
     // types t1, t2, t3, t4 and t5 are visible here
     public type t6 is range 6..7;
     // types t1, t2, t3, t4, t5 and t6 are visible here
   end r;

   public package s is
     // types t1, t2, t3, t4 and r.t6 are visible here
     public type t7 is range 7..8;
     // types t1, t2, t3, t4, r.t6 and t7 are visible here
   end s;

   // types t1, t2, t3, t4, r.t6 and s.t7 are visible here

  end q;

  // types t1, q.t3, q.t4 and q.s.t7 are visible here
  // the structure of types q.t3 and q.s.t7 are not visible here
  // the structure of types q1 and q.t4 are visible here

end p;
```

16.2 Initialisation and finalisation of packages

If the initialisation section of a package is present, then the list of statements within it is executed before the main procedure of the program starts to run. The initialisation statements of any packages that are nested within a package *p* are executed in order before the initialisation of package *p*. This amounts to executing the initialisation sections in the order in which they are encountered when reading the source code from top to bottom (regarding separate packages as being encountered at the point of their `separate` declarations – see below).

If the package contains initialised global variables (other than ones with access mode `image` – see 22.6), then their initialisation expressions are evaluated in order as part of the package initialisation, as if assignments to those variables appeared at the start of the package initialisation section. This means that their values are not available to the initialisation code of sub-packages of the package in

which they are defined, even though they may appear earlier in the source (this does not apply to named constants (5), which are not initialised at runtime - their values are available from the point at which they are declared).

If the finalisation section is present, then the list of statements within it is executed after the main program and all subsystems have exited. The finalisation sections of any packages that are nested with a package *p* are executed in reverse order immediately after the finalisation section of package *p*. Consequently, package finalisation occurs in the reverse order of initialisation, which means that packages are finalised in order in which they are encountered when reading the program from bottom to top (allowing for separate packages as before).

16.3 Separate packages

A package can be logically nested within another without its source code being present in the same file. This is accomplished with a `separate` declaration:

```
package p is

  package q1 is separate;
  package q2 is separate("filename");
  package q3 is separate("@configname");

end p;
```

Corresponding to a `separate` declaration for a package *q*, the source code that defines package *q* is contained in a separate file (named according to local conventions unless *filename* or *@configname* is specified). That file must contain the declaration of a package named *q* (i.e. `package q is … end q;`) and nothing else (apart from comments). The effect is as if that package declaration was inserted in place of the `separate` declaration.

The form *@configname* means that the file name is obtained from a build configuration variable, typically specified on the command line, e.g. `-c:variant=rev01.rihtan` with `separate("@variant")`.

A C source file and a C header file is produced for the system, and for every `separate` package.

If a file is used in two `separate` declarations, then two packages will be created (this will lead to a compilation error if they are in the same scope). A warning is issued if the same filename is used twice. This can be suppressed by prefixing the file name with a - character in a form where the filename is given explicitly (*filename* or *@configname*).

Variants of the separate declaration are possible for shared and unit test packages (52, 83):

```
  shared package q1 is separate;
  unit_test package q2 is separate;
  not unit_test package q3 is separate;
```

16.4 Shared packages

A package can be declared as `shared`. Such a package is intended to serialise access to resources in a system that contains more than one thread of control (subsystems).

```
shared package package_name is
```

```
  declaration_list

begin
  initialisation_statement_list
final
  finalisation_statement_list
end package_name;
```

A shared package cannot contain public global non-constant variables, with the only exception being variables that are declared with access mode `shared_atomic`. All other access must be through public procedures. Calls to public (i.e. interface) routines are serialised, so that only one thread of control can be executing within the package at a time. The restrictions that are normally applied to shared routines (see 55) are applied to all routines that are declared within a shared package, except that a routine within a shared package may access and update global variables that are declared within the same package - this is safe because only one thread of control can be running within the package at a time, and none of those globals are public, so they cannot be accessed directly from outside the package. For this purpose, sub-packages of the shared package are regarded as part of the same package.

Shared packages can contain sub-packages, including public ones. However, they cannot contain subsystems or other shared packages.

Interface procedures of shared packages can have guard expressions (see 18.1). However, a routine within a shared package cannot call another routine within the same package (or any sub-package of it, as above) that has a guard expression. This is because there would be no way in general to change the value of the guard expression if the caller was to block on it.

Shared packages cannot export public functions.

17 The system

The outermost element of the program is the `system`, which is a package that logically encloses all other declarations in the program, including all other packages.

```
system system_name is

  declaration_list

begin
  statement_list
final
  statement_list
end system_name;
```

As with ordinary packages, the initialisation and finalisation sections are optional syntactically, although they might be required in particular cases.

Apart from marking the boundary of the program, the system represents a thread of control. If a procedure within the system is prefixed with `main` then it receives control after system initialisation. The main procedure must be parameterless and must appear directly within the system's declaration list (not in a sub-package). If the system does not have a main program (apart from those in subsystems) then it must contain `pragma no_main` to indicate that this is not an oversight; in that case it functions simply as a container. The system, and any subsystem, can only contain one main procedure.

If the declaration list of the system contains `pragma no_c_main`, then the programmer must supply a main program that calls a C function `__rihtan_initialisation()` to initialise the system, then `__rihtan_run()` to run it, then `__rihtan_finalisation()`. If `pragma no_c_main` is not present, then the compiler will write the C *main* function. There is an exception to this in the case of builds for Contiki, in which case the specific Contiki platform always provides the C main function, and the compiler builds a program that works within that context.

Every procedure and function within the entire system (including all subsystems and other packages), apart from `main` procedures, must be called at least once. This check can be overridden for an individual routine with `pragma unchecked_use(`*`routine_name`*`)`, which can appear either among declarations or statements, or by declaring it to be a library routine (72).

Examples

```
system sys1 is

  … // declarations

  main procedure mainproc is
  begin
    …
  end mainproc;

// an initialisation section could go here
// a finalisation section could go here
end sys1;
```

This example calls the program from a programmer-defined C function.

```
system sys2 is

  … // declarations

  main procedure mainproc is
  begin
    …
  end mainproc;

  pragma no_c_main;

// an initialisation section could go here
// a finalisation section could go here
end sys2;

int main(void)
{
  //
  // do something
  //
  // Initialise, run and finalise the system
  __rihtan__initialisation();
  __rihtan__run();
  __rihtan__finalisation();
  //
  // do something
  //
  return 0;
}
```

18 Subsystems

18.1 Definition and use

A subsystem is similar to a package, except that it also represents a thread of control.

```
subsystem subsystem_name is

  declaration_list

  main procedure subsystem_procedure is
  begin
    …
  end subsystem_procedure;

begin
  statement_list
final
  statement_list
end subsystem_name;
```

The system can contain zero or more subsystems. Every subsystem must contain one parameterless procedure marked `main` that receives its thread of control.

The system, all subsystems and all other packages are initialised sequentially in the order described in (49) before any of the system or subsystem main procedures starts to run.

The main system, all subsystems and all other packages are finalised sequentially in the order described in (49) after all of the system and subsystem main procedures have exited.

Interactions between subsystems and between subsystems and the main system are restricted in various ways that are designed to avoid problems that are associated with access conflicts and race conditions.

1. A non-constant global variable (i.e. one that was not declared with an initialiser) cannot be accessed directly across a subsystem boundary (see below) for the purpose of using it in an expression or assigning to it unless it has access mode `shared_atomic`, or it is atomic and the access is performed within a block that explicitly permits such access (see below).
2. A non-constant global variable cannot be accessed across a subsystem boundary for the purpose of passing it as a parameter unless it is marked `shared`.
3. A procedure that passes one or more `shared` variables as parameters to another procedure (including within expressions for `in` mode parameters) thereby simultaneously acquires exclusive locks on those variables for the duration of the procedure call; it may be suspended until it can acquire those locks. Shared variables cannot be passed to functions[19].
4. A procedure cannot be called across a subsystem boundary unless it is marked `shared`.
5. A function cannot be called across a subsystem boundary unless it is marked `shared` or `closed`.
6. Any access of a global variable by a routine that is marked `shared` is subject to the same restrictions as if that access was crossing a subsystem boundary.
7. A routine that is marked `shared` can only call shared routines and closed functions.
8. A procedure can be declared to have a boolean guard expression; when a procedure attempts to acquire a lock by passing a shared variable to a procedure that has a guard, it cannot acquire the lock unless the guard evaluates to `true`; it will be suspended until that is the

[19]This is to avoid complications when evaluating expressions.

case. Functions cannot have guard expressions.

9. A call to a procedure that has a guard expression must include at least one shared variable as an actual parameter, unless that procedure is within a shared package (16.4).

'Across a subsystem boundary' means either outside the subsystem *s* from where the access is being made, or inside a different subsystem *t*, even if subsystem *t* is nested within subsystem *s*. In this context the main system is regarded as a subsystem itself.

A shared variable is declared by prefixing its declaration with the keyword `shared`. This can in turn be prefixed with `public`.

```
shared var_name : type_name;
public shared var_name : type_name;
```

A shared procedure or function is declared by prefixing its declaration with the keyword `shared`. This can also be prefixed with `public`.

```
shared procedure proc_name(formal_parameter_list) is
begin … end proc_name;
public shared procedure proc_name(formal_parameter_list) is
begin … end proc_name;
shared function fn_name(formal_parameter_list) : return_type is
begin … end fn_name;
public shared function fn_name(formal_parameter_list) : return_type is
begin … end fn_name;
```

A guard is declared as follows:

```
procedure proc_name(formal_parameter_list)
  when guard_expression is
begin
   …
end proc_name;
```

Guard_expression is a boolean-valued expression. It can refer to the formal parameters (and usually will do so).

Explanation of the rules

- Variables must not be accessed in an uncontrolled manner by different threads of control (different subsystems) because that could lead to the problems of corruption, reading partially updated values, and so on.
- If a subsystem *s* exports a routine *r*, and that routine is ever called by another subsystem *t*, then routine *r* must be subject to similar sharing requirements that a procedure in *t* would be subject to (it will of course have different visibility into *s* than a procedure in *t* would have). This is indicated by marking routine *r* as `shared`.
- If a procedure has a guard, then calls to it must be able to block on something that another subsystem can release, i.e. a shared variable.

A procedure can name a shared variable in a `lock` clause. This has the effect that the variable is locked along with other shared variables that are passed to it as actual parameters (if any) at the point of the call.

```
procedure p(...) using (lock => s1, lock => s2, …) is ...
```

One use of this is to avoid double-handling whereby a subsystem might otherwise need an extra procedure simply in order to lock one of its own variables as part of the processing of an interface routine. A function cannot have a `lock` clause for the same reason that it cannot receive shared variables as parameters.

A procedure can name a shared variable in an `unchecked_lock` clause. This is like a `lock` clause, except that it does not require the calling routine to actually obtain a lock on the variable. This can be useful when analysis for locking is desired, but it is known (for some reason) that a locking operation is not actually needed in order to obtain coherent access to the variable, and furthermore that the calling routine cannot be allowed to perform such an operation (if, for example, it is an interrupt handler that must be allowed to run to completion without interruption).

```
procedure p(...) using (unchecked_lock => s1, …) is ...
```

The representation clause `restriction => no_locks` informs the compiler that a routine cannot be allowed to attempt to obtain a lock (and hence potentially switch context). Again, this might be the case if it is to be called from an interrupt hander. If a locking operation would actually be required, then a compilation error is raised. The conditions under which a lock would be required vary slightly between different task implementations. For example, under the non-preemptive models (longjmp and cyclic – see 57), a call to a procedure within a shared package does not require a lock if no routine within the package itself obtains a lock or invokes `pragma dispatch`.

```
procedure p(...) using (restriction => no_locks, …) is ...
```

The compiler detects potential deadlock situations. These are of two types: those arising from an aliasing of a shared variable (as with aliases of global variables), which could lead to self-deadlock of subsystems, and those arising from the situation where a routine acquires a lock on a shared variable *s1* by passing it as a parameter to another routine which, directly or indirectly, acquires a lock on another shared variable *s2*, while another routine acquires a lock on *s2* by passing it as a parameter to a routine which itself acquires a lock on *s1*. The latter type could cause two subsystems to deadlock each other. The check for this kind of potential deadlock between a pair of variables *s1* and *s2* can be suppressed with `pragma unchecked_deadlock(s1, s2).`

The restrictions on accessing entities across a subsystem boundary can be suppressed for particular statements by enclosing them in a block (40) that specifies an `unchecked_share` exemption.

A global variable that has access mode `shared_atomic` (68) can be accessed directly across a subsystem boundary.

A variable or a field or component of a variable can be accessed directly across a subsystem boundary if its representation is atomic (78) and the statements that perform the access are enclosed in a block that specifies a `shared_atomic_access` exemption. This is generally safer than using `unchecked_share`.

18.2 Program termination

The program terminates when the main procedures of the main system and all subsystems have returned, and all finalisation sections have subsequently run to completion.

18.3 Implementation options

There are currently four implementation options for subsystems. One uses the pthreads library and hence is suitable for multicore implementations. The second is a single-threaded multi-stack

coroutine system using setjmp/longjmp that is generated by the compiler - this option does not require a thread library. The third (cyclic) and fourth (contiki) implement a single-stack cyclic call model. Even on single cores, subsystems are useful for program structuring.

The implementation is selected with the following representation clause, which must be applied to the system.

```
system S using (task_implementation => pthreads) is ...
system S using (task_implementation => longjmp) is ...
system S using (task_implementation => cyclic) is ...
system S using (task_implementation => contiki) is ...
```

Tasks (i.e. the threads of control of subsystems) in the longjmp version continue to run until they cannot proceed, at which point a dispatch to another task occurs. This will happen when a task attempts to call a guarded procedure but the guard evaluates to `false`. In that case the task cannot continue until another task changes something so that the guard expression becomes `true`, so a dispatch occurs at such a point. A dispatch can also be written into the code explicitly by means of a pragma:

```
pragma dispatch;
```

An occurrence of `pragma dispatch` at a position where a statement could appear causes a call to the task dispatcher to be inserted at that point.

Under the longjmp implementation on certain targets, it might be necessary to embed some code to temporarily disable interrupts during the dispatch operation. This can be specified by pragmas:

```
pragma pre_dispatch(string_disable);
pragma post_dispatch(string_enable);
```

where *`string_disable`* and *`string_enable`* are inserted into the generated program followed by a newline immediately before the calls to the low-level dispatching operation and at the target of the transfer respectively. Pragmas `pre_dispatch` and `post_dispatch` might also be needed on those targets if the `restart` statement is used (60), even under the other tasking implementations. These pragmas, if required, would normally be written into the target specification package.

Under the single-stack cyclic and contiki models, dispatches from the main procedures of the system and the subsystems may occur on calls to guarded routines or at occurrences of `pragma dispatch`. There is an implementation restriction in this case that requires that only the main procedure of a system or subsystem may dispatch. Nested calls must always return without dispatching.

Under the contiki model, the following pragma is available to autostart C-based Contiki threads:

```
pragma contiki_autostart(c_process_name1, c_process_name2, ...);
```

where the C process names are the names that would be listed in the `AUTOSTART_PROCESSES` macro under Contiki. More than one `contiki_autostart` pragma can be used. This pragma is not required if all threads are defined by the main system and subsystems. Also, when compiling for Contiki, it might be easiest to use the `-oi` compiler option to cause #includes to be written for C files (to avoid the need to link separate object files).

18.4 Subsystem stack size

The size of the stack of a subsystem (relevant for the multi-stack models) can be calculated, or

simply stated. The compiler keeps track of the maximum amount of stack that is required by every procedure or function, taking account of the requirements of any procedures and functions that it calls; the size of a subsystem's stack then becomes the stack requirement of the subsystem's main procedure.

A number of pragmas influence the determination of the stack size.

```
pragma stack_overhead(static_integer_expression);
```

If this pragma appears in the body of a procedure or function, then *`static_integer_expression`* stack units are added to the stack size estimate for that procedure/function (the definition of a stack unit is provided by the target definition as described below (78) – typically it will be `unsigned`). If this pragma appears more than once in a given procedure or function, then the maximum value of the expressions is applied (not their sums). This allows a pragma to be placed before calls of external library functions with known requirements (for example), with the stack being compensated for the largest requirement.

If pragma `stack_overhead` appears outside any procedure or function, however, then the adjustment is applied to the stack of every subsystem. This must be done at least once in order to set a value for the subsystem stack overhead (even if that value is zero). Similarly to the above, if the pragma appears more than once outside any procedure or function, then the maximum value is used (not the sum).

```
pragma stack_size(static_integer_expression);
```

If this pragma appears in the definition of a subsystem, then the subsystems's stack is simply set to *`static_integer_expression`* stack units, and the calculated value (whether or not it includes stack overhead pragmas) is ignored. This pragma can only appear once within any individual subsystem.

Some quantities from the target definition apply to stack size calculations:

The *stack unit* gives the C type that is the unit of the stack pragmas (typically `unsigned`):

```
pragma target_stack_unit(internal_name);
```

The *call overhead* is the number of stack words that are always at a minimum required by any call of a procedure or function.

```
pragma call_overhead(overhead);
```

Pragma `interrupt_overhead` specifies the fixed part of the overhead, in stack units, of an interrupt (in addition to the call overhead):

```
pragma interrupt_overhead(overhead);
```

These pragmas and others are described in section 78.

The size of the stack of the system is calculated in the same manner as for subsystems. However, under the longjmp implementation, the system simply receives the amount of stack that remains after allocating the subsystem stacks from the process stack. If the system specifies pragma no_start then its stack requirements will be minimal.

Under the single-stack models, the stack that is allocated by the linker is used, as with a single-threaded C program.

18.5 Restarting a subsystem

A subsystem or the system can restart itself using the statement `restart subsystem`. This causes the current thread of control to be abandoned and the (sub)system to restart by running its main procedure again. The package initialisation section (if any) is not run again. Any locks on shared variables that the (sub)system holds will be released. This should only normally be used to allow a subsystem to deal with an abnormal situation.

The file name and line number of the point of the restart are recorded and can be retrieved by calling routines in the package `SystemInformation` (87).

```
package SystemInformation is separate;
…
If IrrecoverableProblem then
  RestartReason := SOME_ERROR_CODE;
  restart subsystem;
end if;
…

main procedure MainProc is
begin
  if SystemInformation.GetRestartLine() /= 0 then
    // This is a restart
    ! printf("Restart at line %d of %s\n",
             `SystemInformation.GetRestartLine`(),
             `SystemInformation.GetRestartFile`());
    case RestartReason is
      when SOME_ERROR_CODE => …
      ...
      when NOT_RESTARTED   => null;
    end case;
    RestartReason := NOT_RESTARTED;
  end if;
  …
end MainProc;

begin
  // Package initialisation is not repeated after a restart
  RestartReason := NOT_RESTARTED;
end;
```

19 Generic packages

19.1 Overview

A generic package is a package that is parameterised by types, constants, procedures, functions and packages. In order to use the package, it must be instantiated, which generates a version of the package with the formal parameters replaced by actual types, constants, procedures, functions and packages.

The scope environment of a generic package is the point at which the package is declared, not the point at which it is instantiated.

19.2 Declaration

```
generic
```

```
  type t1;
  type t2;
  type t3 is range;
  type t4 is digits;
  type t4 is array;
  type t4 is access;
  constant c1;
  constant c2;
  constant c3 is range;
  constant c4 is digits;
  constant c4 is boolean;
  constant c4 is string;
  constant c4 is access;
  procedure p1;
  procedure p2(formal1 : mode1 type1; … ; formaln : moden typen);
  function f1 : return-type;
  function f2(parameter-list) : return-type;
  with package pack1 is generic gen1;
  with package pack2 is generic gen2(type1 => actualtype1, ...);
package generic_name is

  declarations

end generic_name;
```

t1, *t2*, and so on are the names of the formal generic parameters. Any simple identifier (i.e. one without a '.') can be used, apart from reserved words or names that clash with other declarations in the scope in which the generic package is declared.

The formal parameters can be the following:

`type` *name* `is range`	Any integer type (including enumerated types and characters)
`type` *name* `is digits`	Any floating-point type
`type` *name* `is array`	Any array type[20]
`type` *name* `is access`	Any access type[21]
`type` *name*	Any type
`constant` *name* `is range`	An integer constant
`constant` *name* `is digits`	A floating point constant
`constant` *name* `is boolean`	A boolean constant
`constant` *name* `is string`	A string constant
`constant` *name* `is access`	An access constant
`constant` *name*	Any constant
`procedure` *name(parameter-list)*	A procedure. The parameter list can refer to formal type parameters of the generic package, as well as other types that are visible at the point of the definition of the generic package.[22]
`function` *name(parameter-list) : return-type*	A function. The parameter list and the return type can refer to formal type parameters of the generic package, as well as other types.[23]

[20]The types of the index and element can be recovered within the generic package using 'index_type and 'element_type
[21]The accessed type can be recovered with 'accessed_type
[22]The parentheses that surround the parameter list can be omitted if the procedure has no formal parameters
[23]The parentheses that surround the parameter list can be omitted if the function has no formal parameters. The return type must be included.

`package name is new generic_package_name`	Any package that is an instantiation of the named generic package
`package name is new generic_package_name( formaltype1 => actualtype1, ...)`	Any package that is an instantiation of the named generic package, with the restriction that any parameters *formaltype$_j$* of that instantiation must have been instantiated with types *actualtype$_j$* (i.e. *formaltype1 => actualtype1* and so on). Not all parameters need to be restricted in this way.

A generic declaration can be separate:

```
generic package generic_name is separate;
```

Generic subsystems are also possible:

```
generic
  generic parameters
subsystem generic_name is

  declarations

end generic_name;

generic subsystem generic_name is separate;
```

Shared packages are also possible:

```
shared generic
  generic_parameters
package generic_name is
  declarations
end generic_name;

shared generic package generic_name is separate;
```

19.3 Instantiation

```
package actual_name is new generic_name (
  t1 => actual_type_name1,
  t2 => actual_type_name2,
  t3 => actual_type_name3,
  t4 => actual_type_name4,
  c1 => actual_constant_expression1,
  c2 => actual_constant_expression2,
  p1 => actual_procedure_name1,
  p2 => actual_procedure_name2(formal1 => actual_formal1, …, formaln => actual_formaln),
  f1 => actual_function_name1 : return_type,
  f2 => actual_function_name2(formal1 => actual_formal1, …, formaln => actual_formaln)
          : return_type,
  pack1 => actual_package_name1,
  pack2 => actual_package_name2
);
```

This creates a package `actual_name` at the point of the instantiation.

An instantiated generic package can export the full range of entities that any package can export, including entities that are constructed from its parameters.

In the case of procedures and functions, the formal parameters `actual_formal`$_j$ of the actual

procedure named in the generic instantiation must match the parameters *formal$_j$* of the formal procedure by type and mode (`in out` etc.) They do not have to match by name (i.e. *actual_formal$_j$* does not have to be identical to *formal$_j$*), and they do not have to be listed in the order of either the formal or the actual procedure, since all parameters are named. Within the body of the generic package, the parameter names *formal$_j$* from the generic parameter list are used. Within the model above, the actuals for `p1` and `f1` do not have parameter matching lists because `p1` and `f1` are parameterless.

The actual instantiation parameters are exported from the newly created package under their formal names (i.e. the formal parameters of the generic package are public in the instantiated package).

The scope environment of a generic package is that of the package definition, not of the instantiation.

Instantiations of generic subsystems and generic shared packages are similar:

```
subsystem actual_name is new generic_name (
  actual_parameters
);

shared package actual_name is new generic_name (
  actual_parameters
);
```

20 Separate blocks

As well as separate packages, sequences of statements and package-level[24] declarations can be separated out physically from a source file.

A separate block declaration names either a sequence of statements or a set of declarations:

```
declare separate name begin
  statement-list
end name;

declare separate name declare
  declaration-list
end name;
```

The name is always public (there would be little point in it being private). The syntax:

```
use separate name;
```

causes the named statements or declarations to be 'read in' to the source code at that point. This must of course make sense in context.

An example of the use is to separate out the code of the unit test sections of procedures into another package in cases where the length of the code would make the source more difficult to read otherwise. For example:

```
package TestSupport is

  declare separate utHelp1 begin
```

[24] i.e. not local to a procedure

```
    declarations of unit_test procedures to support unit testing

  end utHelp1;

  declare separate utProc1 begin

    statements for unit-testing Proc1

  end utProc1;

end TestSupport;

…

package p is

  use separate TestSupport.utHelp1;

  procedure Proc1 is
  begin
    …
  unit_test
    use separate TestSupport.utProc1;
  end Proc1;

end p;
```

Another possible use of separate blocks is to make source code more readable by rearranging it so as to separate out sequences of statements (for example) that have been thoroughly tested and only rarely need to be examined.

21 Preconditions and postconditions

Procedures and functions can be declared to have a precondition, a postcondition, or both:

```
procedure p (parameter_list)
  using (precondition  => boolean_expression1,
         postcondition => boolean_expression2)
   is …
```

The precondition *boolean_expression*$_1$ can refer to the formal parameters of the procedure or function and to globals. It is analysed at the point of every call to the routine, and must be analysable and provably true. When analysing the body of the routine itself, the precondition is asserted to be true at the start.

The postcondition *boolean_expression*$_2$ is analysed at the end of the routine body, where it must be provably true. Immediately after every point of calling the routine, the postcondition is asserted to be true.

Within the postcondition, and within assert statements, the attribute `'initial` refers to the value that the variable had at the start of the routine's body. For example:

```
type TNum is range 1..5;
G: Tnum;

procedure SetG(N : in TNum)
  using (postcondition => G = N) is
begin
```

```
  G := N;
end SetG;

procedure IncG
  using (precondition  => G < TNum'last,
         postcondition => G = G'initial + 1) is
begin
  G := G + 1;
end IncG;

…

SetG(3);  // G is now asserted to be 3
IncG;     // OK, G is now asserted to be 4
IncG;     // OK, G is now asserted to be 5
IncG;     // Compile error – precondition failure
```

The pre- and post- conditions form part of the formal definition of a routine, analogous to formal parameters.

For integer types, postconditions do not weaken the usual requirements that an actual `out` or `in out` parameter must encompass the range of the corresponding formal parameter. For example, if a formal `out` parameter *n* is of a type with range 0..3 and the corresponding actual parameter in a call has range 1..3, then the call is invalid and will be rejected by the compiler even if the procedure has a postcondition *`n > 0`*.

22 Representation clauses

Representation clauses give the programmer control over how an object is represented and accessed. Normally they are appended to the declaration of a type, variable or record field.

22.1 Representation of numeric types

Part of the compiler configuration is a target definition (78) that lists the basic types that are available in the target environment. When presented with the definition of a numeric type, the compiler will normally choose a basic type from the target definition to represent it. However, this can be overridden with a representation clause:

```
type integer_type_name is range low..high using (word_type => target_integer_type);
type enumerated_type_name is (e1, e2, … en) using (word_type => target_integer_type);
type floating_type_name is digits d magnitude m using (word_type => target_float_type);
```

The chosen target type must be capable of representing the type that is being defined. The target type names are the *inames* from the relevant target definition pragmas (see 78).

For enumeration types, the value of the first symbol can be specified (the default is zero):

```
type enumerated_type_name is (e1, e2, … en) using (first_value =>
constant_integer_expression);
```

The other values count up by one from the first value.

Both can be specified:

```
type enumerated_type_name is (e1, e2, … en)
  using (word_type => unsigned16, first_value => 1);
```

Examples

```
type Sensor_register is range -10..63 using (word_type => signed_16);
type IO_status is (Reset, Busy, Available)
  using (word_type => unsigned_16, first_value => 0x10);
type Angle is digits 9 magnitude 4 using (word_type => ieee_double);
```

22.2 Specifying the address and C name of a global variable

The address in memory of a global variable can be specified:

```
var_name : type_name using (address => constant_integer_expression);
var_name : type_name using (address => "constant_string");
```

Constant_integer_expression is interpreted as a machine address. If *constant_string* is given instead then the string is used as an address expression (without the quotes), which allows a macro from a C header file to be used (typically a header file that gives register addresses for a microcontroller).

This will often be used with an `access_mode` clause.

It is also possible to bind the variable to an external symbol:

```
var_name : type_name using (external => "external_name");
```

References to *var_name* will be translated to references to *external_name*, which must be defined in the external environment (for example by the linker, or in a separate object file). The compiler will add a C 'extern' definition.

A variant of this is

```
var_name : type_name using (reference => "external_reference");
```

References to *var_name* will be translated to references to *external_reference*. In this case the compiler will not add a C 'extern' definition, and it is possible for *external_reference* to name a field or an array element (it is simply inserted literally in the code that is generated). The programmer must arrange to import any relevant definitions (e.g. by using pragma `include`).

Normally the compiler generates a C name for an object, but this can be overridden to specify the name explicitly:

```
var_name : type_name using (cname => "name");
```

In this case the compiler still creates the object as usual; the difference is the name that is used for it in the C code. This can help in interfacing with C language modules. The programmer must ensure that there are no name clashes (a clash will normally lead to a C compiler or linker error).

Examples

```
Temperature_input : Temperature_sensor using (address => 0xf0001024);
IO_resister :  IO_type using (external => "IO_A_reg");
Update_result : Result_code using (cname => "_T_update_res");
```

22.3 Specifying the names of constants and enumeration constants

Constant and enumeration constant definitions are written to C header files as `#defines` in order to make them available to C modules. Normally the compiler will generate a name that includes a prefix that specifies the package in which the constant was defined. This can be overridden with a cname representation clause.

```
const_name : constant := constant_expression using (cname => "name");
type enum_type_name is (Sym1 using (cname => "name1"), …, Symn);
```

Examples

```
MaximumElements : constant := Index1'last - Index1'first + 1
  using (cname => "MAX_ELTS");

type DeviceStatus is (
  Reset   using (cname => "DEV_RESET"),
  Reading using (cname => "DEV_READING"),
  Writing using (cname => "DEV_WRITING"));
```

22.4 Specifying the C name of a procedure or function

The C name of a procedure or function can be specified:

```
procedure p(parameter_list) using (cname => "name") is
begin
  …
end p;
```

22.5 Declaring that the implementation of a procedure or function is a C function

A procedure or function can be declared in the usual way, but its implementation can be supplied by a C function:

```
procedure p(parameter_list) using (reference => "c_function_name") is separate;

procedure p(parameter_list) using (reference => "c_function_name") is
begin
  null;
end p;
```

In the second form, the body of a procedure (between `begin` and `end`) that is declared in this way can only contain `null` statements and pragmas.

Similarly:

```
function f(parameter_list) : return_type using (reference => "c_function_name") is
separate;

function f(parameter_list) : return_type using (reference => "c_function_name") is
begin
  null;
end f;
```

22.6 Specifying the means of access to a variable or formal parameter

A variable or a formal parameter of a procedure or function can be declared to be *volatile*, which means that all accesses to it occur directly to memory rather than possibly making use of intermediate storage:

```
var_name : type_name using (access_mode => volatile);
formal_parameter_name : mode type_name using (access_mode => volatile);
```

A variable can be declared to be a *mapped device*, which means that it is volatile, but also that it does not have to be initialised before being read, or read after being assigned (which generally avoids the need for `unchecked_initialisation` and `unchecked_use` to be applied to it), or indeed ever used at all:

```
var_name : type_name using (access_mode => mapped_device);
```

A variable can be declared with access mode `shared_atomic`, which means that it is volatile and also that it can be directly accessed across subsystem boundaries. This can only be declared if the target representation of the variable is atomic (78).

```
var_name : type_name using (access_mode => shared_atomic);
```

If, in a procedure or function call, an actual parameter has an access mode of a volatile type (i.e. `volatile`, `mapped_device` or `shared_atomic`), then the corresponding formal parameter must have access mode `volatile`. However, a formal parameter of a numerical type with mode `in` cannot be declared to have access mode `volatile`. This is to avoid potential confusion, because these parameters will be passed by value not by reference. The solution is to copy the volatile variable to an ordinary variable before passing it.

A variable with any volatile access mode is always taken to have the full range of its type when performing analysis of integer expressions (regardless, for example, of the effect that an enclosing `if` statement or a preceding `advise` statement might otherwise have).

A variable that has an initialisation clause can be declared to be part of the program image (so it can be placed in ROM, for example).

```
var_name : type_name := initialisation_clause using (access_mode => image);
```

It must be possible for the compiler to evaluate the initialisation clause statically. Such a declaration might be used for a fixed lookup table, for example. In the case of an array, the `[range j => ...]` form cannot be used; only the forms that list all the values can be used.

An integer variable (including those of enumeration and character types) can be declared to be *virtual*, which means that no storage is associated with it. Such a variable cannot be passed as a parameter, or used in an expression except in the case of assignment to another virtual variable. However, attributes of it that have static values (such as `'first`) can be used in expressions. A virtual variable is used for the purpose of analysis - for example, for regulating the sequence in which calls can be made to routines by use of preconditions and postconditions - without incurring the overhead of a real variable.

```
var_name : type_name using (access_mode => virtual);
```

Another possible use is to track parts of the program that must be modified together, or at least checked when another related piece of code is changed. This can be done by creating a virtual variable that only has a range of 1 or a single enumeration value, assigning to it in each part, and changing the value that is assigned whenever a part of the code is changed (so that the program will not compile until the declaration of the variable and all other updates of it have been modified).

Examples

```
Temperature_input : Temperature_sensor
  using (addess => 0x8000, access_mode => mapped_device);


States : State_table := [
  FirstState  => (Output => 1, NextState => ThirdState),
  SecondState => (Output => 2 * 3, NextState => FirstState),
  ThirdState  => (Output => 0, NextState => SecondState)
] using (access_mode => image);


public out IgnitionState : public (Off, Accessory, Start, Run)
  using (access_mode => virtual);
public rev : public range 2..2 using (access_mode => virtual);
```

22.7 Specifying the minimum storage used by a record

```
type record_type_name is
  record
    field_definitions
  end record
using (storage => n * target_type_word);
```

For every variable of this record type, at least *n* words of *target_type_word* are used. If this is insufficient, then more will be used as necessary. *Target_type_word* is an iname[25] from the target definition.

Example

```
type Rec is
  record
    F1 : Field_1_type;
    F2 : Field_2_type;
  end record
using (storage => 5 * unsigned_32);
```

22.8 Specifying the offset of integer fields within records

The declaration of an integer- or boolean- valued field can be suffixed with a clause that gives its offset from the base of the record and size in bits:

```
field_name : field_type
               using (word_offset => n * target_type_word, bit_offset => b, bits => s);
```

where *n*, *b* and *s* are constant integer expressions, and *target_type_word* is the iname of an integer type from the target definition.

[25] See 24

The field is declared to occupy bits *b* through `b + s - 1` of the *n*th *target_type_word* from the start of the record. The programmer must ensure that these values are correct and do not cause unwanted clashes. The `bit_offset` and `bits` can be omitted, but if either is specified then both must be, as well as the `word_offset`.

Examples

```
type IO_register is
  record
    Mode : Mode_value
      using (word_offset => 0 * unsigned_16, bit_offset => 2, bits => 4);
    Address_register => Address_type
      using (word_offset => 1 * unsigned_16);
  end record;
```

Field offsets and record sizes can both be specified:

```
type IO_register is
  record
    Mode : Mode_value
             using (word_offset => 0 * unsigned_16, bit_offset => 2, bits => 4);
    Address_register : Address_type using (word_offset => 1 * unsigned_16);
  end record
using (storage => 2 * unsigned_16);
```

22.9 Specifying storage allocators

The definition of an access type can specify the names of the external C functions that are used for storage allocation and deallocation:

```
type access_type_name is access object_type_name
       using (allocate => "allocator_function_name", free => "free_function_name");
```

Allocator_function_name and *free_function_name* are the names of C functions that perform allocation and freeing of memory.

The allocator should have the prototype: `void *allocator_function_name(size_t size);`
The free function should have the prototype: `void free_function_name(void *ptr);`

If this is not done, then the currently applicable default routines will be used. The defaults are set with:

```
pragma set_default_allocate("allocator_function_name");
pragma set_default_free("free_function_name");
```

These pragmas apply from the point where they appear to the end of the immediately enclosing package (or system or subsystem) (unless changed by another pragma in the meantime).

At the start of the compilation, the defaults are the standard C `malloc` and `free` functions.

22.10 Specifying C attributes

The clause `cprefix => "attributes"` can be applied to type, variable, procedure and function definitions. When corresponding objects are declared, *attributes* will be prefixed to their declarations (in the case of variables, the attributes from both the type and the variable will be prefixed if both have been specified).

The clause `cprefix_line => "attributes"` is similar, except that the attributes string is prefixed as a separate line. This is more useful for pragmas, for example.

The `cprefix` and `cprefix_line` attributes are not applied under unit test builds (83).

The clause `ccast => "C_type_name"` can be applied to a type. It indicates that values of that type should be cast to the indicated C type in the generated program (this is not normally required).

The clause `interrupt_handler => with_interrupt` or `interrupt_handler => without_interrupt` can be specified for a procedure and indicates that the procedure is an interrupt handler that respectively allows or does not allow other interrupts to occur while it is running. This is only used for calculation of stack requirements.

Examples

```
procedure Port2Interrupt using (cprefix => "__interrupt",
                                interrupt_handler => without_interrupt) is
begin
  // etc.
end Port2Interrupt;

procedure Port3Interrupt using (cprefix_line => "#pragma interrupt") is
begin
  // etc.
end Port3Interrupt;

type F is digits 5 magnitude 12 using (ccast => "special_float_type");
```

22.11Indicating that a routine should be inlined

The clause `code => inline` can be applied to a procedure or function definition. It causes the procedure or function to take the attribute that is given by the target configuration pragma `inline_prefix` or `inline_prefix_line` and to be written into the appropriate C header file rather than the source file.

Example

```
procedure Adjust(x : in out some_int_type)
  using (code => inline) is
begin
  x := (x * 2 + 1) mod (some_int_type'last + 1);
end Adjust;
```

22.12Indicating that a routine should be implemented as a C macro

The clause `code => macro` can be applied to a procedure or function definition. The routine will be written to the appropriate C header file as a macro. The body of such a routine can only contain C code lines and pragmas. The visibility and calling discipline are applied as for a regular routine, however (including parameter types etc.). If the routine reads or writes global variables, then `unchecked_use` and `unchecked_initialisation` pragmas should be embedded in the routine body to indicate this. Preconditions and postconditions can be used, but the correctness of the postcondition will not be checked (it will still be applied at the site of any call to the routine as usual, however).

Example

```
generic
  type SourceType;
  type TargetType;
package unchecked_conversions using (usage => library) is

  public function convert_from(Source : in SourceType) : TargetType
    using (code => macro) is
  begin
  ! (`TargetType`)(`Source`)
  end convert_from;

  public procedure convert(SourceVar : in SourceType; TargetVar : out TargetType)
    using (require_named_parameters, code => macro) is
  begin
  ! *(`TargetVar`) = (`TargetType`)(`SourceVar`);
  end convert;

end unchecked_conversions;
```

22.13Indicating whether named parameters are required

The clause `require_named_parameters => boolean_literal` is applied to a procedure or function definition and specifies whether or not named parameters are required in calls. If this clause is not given, then the compiler works this out for itself based on the types of the formal parameters. *Boolean_literal* can be omitted, in which case it is taken to be true.

Examples

```
procedure TraceDisplay(x : in tNum1; y : in tNum2)
  using  (require_named_parameters => false, reference => "Trace") is
begin null; end;
```

22.14Identifying library routines and variables

Usually procedures and functions must be called from at least one point[26] within the system, and global variables much be accessed from at least one point (unless their access mode is `mapped_device`). However, by declaring that a routine or variable is intended to be used as a library routine or variable, it need not be called or accessed respectively, and furthermore if it is not, then it will be excluded from the program that is generated.

```
procedure Print(str : in string)
  using (usage => library) is
begin ... end;

Buffer : array [1..MaxChars] of character
  using (usage => library);
```

If a routine or variable is declared to have `target_library` usage, then it is always included in the program, whether or not it is called. This can be useful if a routine is only to be called from C code or is an interrupt handler.

```
procedure CalculateArea(Boundary : in Segments; InnerSize : out Area)
  using (usage => target_library) is
```

[26]i.e. from at least one point that is actually used; calling from a library routine that is never used or from a block of code that is excluded by conditional compilation does not count

```
begin ... end;
```

Library usage (including target_library usage) can be set at the package level, in which case it applies throughout the package (including in sub-packages) unless overridden in the case of specific routines, variables and sub-packages by the clause `usage => required`.

```
package PrintLibrary using (usage => library) is

  procedure Print(str : in string) is
  begin … end;

  procedure NewLine is
  begin … end;

end PrintLibrary;
```

22.15 Representation for anonymous types

In variable declarations of the form

var_name : *type_specification* using (*representation*);

the representation applies to the variable, not to the anonymous type, so, for example, it might specify the address of the variable, but not the word type. In order to make it apply to the type, enclose the type specification in parentheses:

var_name : (*type_specification* using (*type_representation*));

It is possible to do both:

var_name : (*type_specification* using (*type_representation*)) using (*var_representation*);

In fields with anonymous types in records, the representation applies to the field as a whole (so it might specify the position of the field within the record, for example). In order to make it apply to the type specification, enclose the specification in parentheses:

field_name : (*type_specification* using (*type_representation*));

As above, it is possible to do both:

field_name : (*type_specification* using (*type_representation*)) using (field_*representation*);

Examples

```
Test : (range 0..100 using (word_type => unsigned_32));
Sample : (range 0..10 using (word_type => unsigned_16)) using (address => 0x1234);
type rec is record
  field1 : some_type;
  field2 : (enum1, enum2) using (word_offset => 3 * unsigned_16);
  field3 : ((enum3, enum4) using (first_value => 5));
  field4 : ((enum5, enum6) using (first_value => 5))
             using (word_offset => 6 * unsigned_16);
end record;
```

23 Initialisation and finalisation of variables

23.1 Initialisation of variables

Every variable must provably be initialised before it can be read, in one of the following ways:

- By passing it to a procedure as an `out` parameter

- For numerical, boolean and access variables, by assignment
- For record variables, by a whole-record assignment or by a copy assignment from an object that has already been initialised
- For array variables, by a whole-array assignment or by a copy assignment from an object that has already been initialised
- For arrays of character, as for other array variables or by assignment from a string
- By naming the variable in an `unchecked_initialisation` pragma[27]

A piecemeal assignment to record fields or array components does not count as initialisation.

An `unchecked_initialisation` pragma can appear in a sequence of statements.

```
pragma unchecked_initialisation(variable1, variable2, ...);
```

It avoids the need (for example) to perform an expensive and unnecessary initialisation of a large array that is used as a buffer.

Formal parameters of mode `out` must provably be initialised before the procedure in which they are declared returns.

23.1.1 Global variables

If the definition of a global variable includes an initialisation expression, then it will be initialised automatically during package initialisation. A global variable that does not have an initialisation expression can be initialised either within the initialisation section of the package in which it is declared or at any other point, but must provably have been initialised before it is used according to analysis of program flow. Initialisation can also be declared with pragma `unchecked_initialisation(var)`.

It is an error if a global variable is not assigned to anywhere in the program, either in its declaration or by a statement; `pragma unchecked_initialisation(var)` has the effect of declaring it to have been assigned.

It is an error if a global variable is not read anywhere in the program. This can be overridden with pragma `unchecked_use(var)`, which has the effect of declaring it to have been read without actually reading it.

These pragmas can appear in place of statements.

The checks described in this section do not apply to variables with access mode `mapped_device`.

23.1.2 Local variables

If the definition of a local variable includes an initialisation expression, then it will be initialised automatically at every call of the procedure or function in which it is defined (the value might differ from call to call). A local variable that does not have an initialisation expression must be initialised by a statement within the body of the procedure or function.

It is an error if a value that has been written to a local variable is not definitely used, either before

[27]'Initialisation' can also be spelt 'initialization'

another value is written to the variable or before the routine returns. This check can be overridden with `pragma unchecked_use(var)`, which has the effect of a statement that reads the variable without actually reading it.

23.1.3 Effects of flow control statements

If statements

If a variable is initialised before an `if` statement then, provided that it is not finalised (see below) by any branch of the `if` statement, it remains initialised after the statement.
If a variable is not initialised before an `if` statement, then it is initialised after the statement if and only if it is initialised by every branch.

Case statements

If a variable is initialised before a `case` statement then, provided that it is not finalised by any branch of the `case` statement, it remains initialised after the statement.
If a variable is not initialised before a `case` statement, then it is initialised after the statement if and only if it is initialised by every branch.

Loop statements

If a variable is initialised before a `loop` statement, then provided that no path through the body of the loop finalises it, then it remains initialised after the loop.
Otherwise if every path through the body of a `loop` statement definitely initialises the variable, and the loop body definitely runs at least once, then the variable is initialised after the loop.

23.2 Controlled types

A type can be declared to be `controlled`, which means that variables of that type must provably be finalised before they cease to exist.

```
type type_name is controlled type_declaration;
```

A record type that has at least one field of a controlled type must itself be declared `controlled`. An array type whose components are of a controlled type must also be declared `controlled`. For access types see below.

A variable of a controlled type, or a `final in out` formal parameter, is finalised in one of the following ways:

- In the case of a record, by performing a whole-object assignment that finalises every controlled field of the record, and names no other fields
- In the case of an array, by performing a whole-object assignment that finalises its components
- By passing it to a procedure as a `final in out` parameter

- By declaring it to be finalised with a `finalised` pragma[28]
- By declaring it to be finalised with an `unchecked_finalisation` pragma

The difference between `pragma finalised` and `pragma unchecked_finalisation` is that `pragma finalised` cannot be applied to records that have fields of controlled types or to arrays of elements of controlled types in order to finalise the entire record or array. If a record or array type is itself controlled but does not have controlled components, then `pragma finalised` can be applied to it. `Pragma finalised` is the normal way to declare that objects are finalised (apart from objects that themselves have controlled components, as described above).

Unlike `unchecked_initialisation` pragmas, `finalised` and, possibly, `unchecked_finalisation` pragmas will generally be involved at some level because the compiler has no direct way to know that objects have been finalised beyond a certain level of decomposition.

Formal parameters of mode `final in out` must provably be finalised before the procedure in which they are declared returns.

Only local variables and formal parameters of mode `final in out` can be finalised, except in package finalisation sections where all controlled globals can, and must, be finalised (see below).

A variable cannot be finalised twice without being reinitialised in between.

Example

```
type Num is range 0..10;
type Index is range 0..100;

type Arr is array [Index] of Num;

type NumRec is controlled
  record
    N1 : Num;
    N2 : Num;
  end record;

type Rec is controlled
  record
    Na : NumRec;
    A  : Arr;
    Nb : NumRec;
  end record;

type Index2 is range 1..5;
type ArrRec is controlled array [Index2] of Rec;

// A controlled type can of course be public as well:

public type Rec2 is public controlled
  record
    N1 : Num;
    N2 : Num;
  end record;

// A variable is finalised by passing it as a 'final in out' parameter:
closed procedure FinaliseNumRec(NR : final in out NumRec) is
begin
  null; // Do something with NR...
  // Since in this case there is nothing more to do, we can declare
  // the formal parameter to have been finalised using a pragma:
```

[28]This can also be spelt 'finalized', and similarly for unchecked_finalization

```
  pragma finalised(NR);
end FinaliseNumRec;

// In general a record is finalised by a form of whole-record assignment
// that finalises all of its controlled components (and no others):
closed procedure FinaliseRec(R : final in out Rec) is
begin
  R := (Na => FinaliseNumRec(this), Nb => FinaliseNumRec(this));
end FinaliseRec;

closed procedure InitialiseRec(R : out Rec) is
begin
  R := (Na => (N1 => 0, N2 => 1),
        A  => [range k => 0],
        Nb => (N1 => 2, N2 => 3));
end InitialiseRec;

// An array is finalised by finalising all of its components with a form
// of whole-array assignment:
procedure Test1 is
  AR1 : ArrRec;
begin
  // Initialise AR1
  AR1 := [range j => InitialiseRec(this)];
  //
  // Finalise it
  AR1 := [range j => FinaliseRec(this)];
end Test1;
```

A use for controlled types might be to represent a resource like a file that must be closed:

```
type FileIO is controlled record
! FILE *f;
end record;
```

23.2.1 Global variables

Controlled global variables must be finalised during package finalisation. This might make a package finalisation section necessary. Controlled global variables must be initialised during package initialisation (unlike global variables in general, which can be initialised during program flow when needed).

23.2.2 Local variables

Controlled local variables must be finalised before the procedure or function in which they are declared returns.

23.2.3 Access types

If a type `access t` (or variants like `access or null t`) is defined for a controlled type `t`, then the access type must itself be declared `controlled`. A variable of the access type is finalised by finalising the object it refers to.

However, a type `managed access t`, where `t` is controlled, does not need to be declared `controlled` (and normally wouldn't be). In this case the type `t` must have an accompanying finalisation procedure, which must be a closed procedure that accepts a single `final in out` parameter of type `t`. The association is made with a `using` clause:

```
type t is controlled type_declaration;

closed procedure finalisation_routine_name(R : final in out t) is
begin
   statement_list
end finalisation_routine_name;

type t is controlled using (access_finalisation => finalisation_routine_name);

type access_t is managed access t;
```

The finalisation procedure will be called whenever an object of type `t` is about to be freed.

Example

```
// A controlled type
type Rec2 is controlled record
    N1 : Num1;
  end record;

// A finalisation routine for that type
closed procedure Finalise_Rec2(R : final in out Rec2) is
begin
  // For this example there is nothing to do
  pragma finalised(R);
end FinaliseRec2;

// Declare the finalisation routine that will be called when a Rec2 is
// about to be freed.
type Rec2 is controlled using (access_finalisation => Finalise_Rec2);

// Declare an access type
type Rec2_access is managed access Rec2;

procedure Test4 is
  P : Rec2_access;
begin
  P := new (N1 => 1);
end Test4;
```

24 The target definition

The target definition specifies which types are available on the target along with some other details about the target environment. It consists of a set of pragmas. Typically, these will be wrapped in a package and referenced at the start of the system with a `separate` clause. The target definition must be referenced before any other types can be defined.

To define an integer type from the target, use `pragma target_integer_type`:

```
pragma target_integer_type(
         cname         => "c_type_name",
         min_value     => minimum_value,
         max_value     => maximum_value,
         arithmetic    => use_for_arithmetic,
         preferred     => preferred_for_arithmetic,
         iname         => internal_name,
         bits          => size_in_bits,
         atomic        => atomic_type,
         format_string => "c_format_string");
```

where

c_type_name is the name of the C type (e.g. `unsigned char` or `uint8_t`),
minimum_value and *maximum_value* are static integer expressions that give the range of the C type,
use_for_arithmetic is a static boolean expression that indicates whether or not the type can be used for arithmetic,
preferred_for_arithmetic is a static boolean expression that indicates whether or not the type is preferred for arithmetic (in comparison with a type that is allowed but not preferred),
internal_name is a symbol by which the type will be referenced within the program (except in C code),
bits is the size of the type in bits (actual bits to represent the values, not `sizeof*8`),
atomic_type is a static boolean expression that indicates whether or not the type is manipulated by single indivisible operations at the machine level, and
c_format_string is a `printf` format string that is used to convert values for output (normally only used in test builds).

To define the boolean type, define a suitable integer type first and use `pragma boolean_type`:

```
pragma boolean_type(integer_type_iname);
```

The parameter is the iname (internal name as above) of an integer type that has already been defined. This type's range must include 0..1, which will be used to represent `false` and `true` respectively.

Similarly, to define the character type, define a suitable integer type first and the use pragma character_type:

```
pragma character_type(integer_type_iname);
```

The named type's range must include at least 0..127.

Typically, `unsigned char` will be defined first as a target type and used for `boolean` and `character`.

To define a floating point type from the target, use `pragma target_float_type`:

```
pragma target_float_type(
         cname           => "c_type_name",
         float_digits    => digits,
         float_magnitude => magnitude,
         arithmetic      => use_for_arithmetic,
         iname           => internal_name,
         bits            => size_in_bits,
         format_string   => "c_format_string");
```

where

c_type_name is the name of the C type (e.g. `double`),
digits is a static integer expression that gives the number of decimal digits of precision of the C type,
magnitude is a static integer expression that gives the largest power of 10 that the C type can represent,
arithmetic is a static boolean expression that indicates whether or not the type should be used for arithmetic,

internal_name is a symbol by which the type will be referenced within the program (except in C code),
bits is the size of the type in bits (actual bits to represent the values, not sizeof*8), and
c_format_string is a suitable `printf` format string.

With both `target_integer_type` and `target_float_type`, the type is added as an ordinary publicly-visible type within the package in which the pragma appears, using its *internal_name*.

Pragma `target_stack_unit` is used to set the unit for stack usage pragmas:

```
pragma target_stack_unit(internal_name);
```

where *internal_name* is the internal name of an integer type from the target definitions. If this pragma is omitted, then the C type `unsigned` is used by default.

If pragma `clear_target_stack` appears, then subsystem stacks will be zeroed prior to use under the longjmp tasking implementation:

```
pragma clear_target_stack;
```

Pragma `call_overhead` specifies the fixed part of the overhead, in stack units, of a C function call:

```
pragma call_overhead(overhead);
```

where *overhead* is a static integer expression. This pragma is required if functions are generated (i.e. almost always); an error will be emitted if the call overhead is needed but has not been specified. For a single-subsystem (i.e. single-task) program running under a general-purpose operating system, the value might not actually be used.

Pragma `interrupt_overhead` specifies the fixed part of the overhead, in stack units, of an interrupt (in addition to the call overhead):

```
pragma interrupt_overhead(overhead);
```

where *overhead* is a static integer expression. This pragma is required if a routine is marked as an interrupt handler with the representation clause `interrupt_handler => .`

Pragma `memory_move` names a C function that performs memory block moves:

```
pragma memory_move(function_name);
```

where *function_name* is the name, as a string, of a C function that has the same profile and effect as the C library function `memmove` (the default is `memmove`).

Pragma `inline_prefix` gives the prefix that is applied for the procedure and function attribute `code => inline`. Use pragma `inline_prefix_line` instead if the prefix is on a separate line.

```
pragma inline_prefix("static inline");

pragma inline_prefix_line("#pragma inline");
```

If the inline prefix is required but it has not been specified, then a compilation error will be raised.

In most cases `inline_prefix("static inline")` will be appropriate.

Pragma `universal_index_type` declares a unitless type that has sufficient range for any array index:

```
pragma universal_index_type(
         word_type => integer_type_iname,
         min_value => minimum_index,
         max_value => maximum_index);
```

where *`integer_type_iname`* is the iname of an integer type that has already been defined, and *`minimum_index`* and *`maximum_index`* are static integer expressions that give the minimum and maximum values that an array index can have. The word type becomes the default type for target array indexes from that point.

The type that is created is called `universal_index`. This type has range *`minimum_index..maximum_index`*.

Variables and parameters of type `universal_index` can be declared. Type `universal_index` can be used as the index type in the declaration of an array type; however, variables and fields of such array types cannot be declared. The main uses of an array type with a `universal_index` index are for formal parameters and as the accessed type in access type declarations.

```
type AccessBytes is unchecked access array [universal_index] of Target.unsigned_8;
```

Pragma `maximum_object_size` sets the maximum value that can be returned when a `'size` attribute is applied to a variable or a type. A value must be used for analysis purposes, so any use of the `size` attribute will raise an error if `pragma maximum_object_size` has not been given.

```
pragma maximum_object_size(0x7fff);
```

A general target specification should also define the following

```
unit address_unit;
type address_integer is range 0..target_defined unit address_unit;
procedure integer_to_address(Int : in address_integer; Addr : out address) is ...
procedure address_to_integer(Addr : in address; Int : out address_integer) is ...
```

For each zero-based unsigned integer type *`t`* from the target types (*`t`* being the iname from the defining pragma):

```
procedure t_modulo_add(Var : in out t; Val : in t) is ...
```

which should implement addition for type *`t`* modulo `(t'last + 1)`.

These definitions are available to the programmer but are not used by the compiler itself.

25 Conditional and partial compilation

25.1 Configuration variables and separate packages

As described in section 52, the notation `package p is separate("@configname")` takes the code for package *`p`* from the file that is named by the configuration variable *`configname`*. Typically the mapping from *`configname`* to a file name is specified on the command line of the compiler. Usually

this file will contain the definitions of build-specific constants and routines.

25.2 Conditional compilation of statements

If the boolean condition in an `if` statement is static (i.e. determined at compile-time), then code is not generated for the evaluation of the boolean condition itself or for any part of the `if` statement that cannot possibly be executed, i.e. the `if` part if the condition is false, or the `else` part (if present) if the condition is true, and equivalently for `elsif`. In addition, code that is excluded in this way is not analysed semantically, and so can contain statements that would otherwise be rejected on semantic grounds (by generating out-of-range values, for example). Its syntax must however be correct.

Example

Given a package `Build_options` that exports a named integer `Hardware_rev`:

```
if Build_options.Hardware_rev < 3 then
  // Code for hardware revisions less than 3
else
  // Code for hardware revisions 3 and up
end if;
```

25.3 Conditional compilation of declarations

The declarations of variables, types, constants, procedures, functions and packages can include a representation clause `build_condition => boolean_expression` that determines whether the declaration is evaluated. *`Boolean_expression`* must be static.

Example

```
temp_value : temp_type       using (build_condition => Build_options.hardware_rev < 3);
temp_value : rev_3_temp_type using (build_condition => Build_options.hardware_rev >= 3);
```

No name clash can occur in this case because only one of the declarations will be evaluated.

The declaration must be syntactically valid even if *`boolean_expression`* is false.

25.4 Incomplete routines

During the development of a program, the writing of the bodies of procedures and functions can be deferred while still allowing analysis of the system to be carried out. Usually just using a `null` statement as the body will lead to errors that parameters have not been accessed or assigned, the return value of a function has not been set, and the postcondition (if any) is not valid. This can be prevented by including `pragma incomplete` in the body of the routine.

```
function CalcNextValue(PreviousValue : in ValueType; Offset : in OffsetType) :
  ValueType is
begin
  pragma incomplete;
end CalcNextValue;
```

`Pragma incomplete` just suppresses the relevant checks. Any code that is present in the body must be correct.

`Pragma incomplete` can only be used when running the compiler in one of the analyse-only

modes. An error will be output if the pragma is encountered when generating code.

26 Descriptive pragmas

Some pragmas insert comments into the generated program in order to help to debug and document it. They have no effect on the meaning of the program.

Pragma `describe` can appear wherever a statement can be placed or in a declaration list.

```
pragma describe(expression1, expression2, …, expressionn);
```

It emits (as a comment) information about the expressions at that point in the program, e.g. the range, if the expression is of integer type, whether the expression is static, and so on.

Likewise, pragma `describe_locals` can appear wherever a statement can appear in a procedure or function, or in a procedure or function's declaration list.

```
pragma describe_locals;
```

It emits information about local variables and formal parameters.

Pragma `describe_scope` can appear at any point that a statement can appear.

```
pragma describe_scope;
```

It lists the entities that are in scope at that point.

27 Other pragmas, exemptions and representations

Pragma `hide` causes a variable to become invisible from the point of the pragma on until the end of the enclosing procedure or function. This can be used when the value of an input parameter is only used to generate an intermediate value that is then used in all subsequent computations, in order to detect accidental use of the parameter.

```
pragma hide(var1, var2, …);
```

Pragma `note` displays a note. The compiler will display the note (which is a string or a series of strings), and a count of such notes, when it translates the program. The note has no effect on the program itself.

```
pragma note(“Check this”);

pragma note(“Check this”, “then retest procedure Main”);

pragma note;  //- displays an empty note
```

28 Unit testing

The compiler supports unit testing. The model is to build a C program that can be compiled and run on the compiler host rather than on the target.

Unit tests can be embedded in procedures and functions:

```
procedure proc(formal_parameter_list) is
  local_declaration1;
   …
  local_declarationm;
begin
  statement1;
   …
  statementn;
unit_test
  statementu1
   …
  statementun
end proc_name;

function fn(formal_parameter_list) : return_type is
  local_declaration1;
   …
  local_declarationm;
begin
  statement1;
   …
  statementn;
unit_test
  statementu1
   …
  statementun
end fn_name;
```

The unit test statement sequence *statement*$_{u1}$ through *statement*$_{un}$ should call the procedure *proc* or function *fn* with appropriate parameters and, if appropriate, settings of global variables, so as to test every code path through it. If at any point in the unit test statements an error is detected, then `pragma fail` should be used as a statement in order to report a unit test failure. If all tests pass then `pragma success` can be used. Alternatively, if the end of the unit test statements is reached without a failure, then success will be reported. However, if not every path was exercised then incomplete coverage will be reported instead of success.

The program is built for unit testing if the command line parameter `-tu` is given to the compiler. In that case, the generated program is initialised and finalised as usual but, instead of running the application, all routine unit tests are run in order in which those units are encountered when reading the source from the top, regarding separate packages as appearing at the point of the `separate` statements.

When the system is built for unit testing, it is also built for coverage testing, and unexercised code branches will be listed.

Under unit test builds, variables that are declared with access mode `mapped_device` are converted to ordinary variables that require initialisation before use, although use-before-reinitialisation and global usage checks still do not apply to them. Mapped device variables that are declared with address or external reference clauses will be converted to ordinary variable declarations. Mapped device variables remain volatile, however. Within the main body of the routine, mapped device variables can be manipulated by specially marked embedded code in order to generate values. Such code is delineated by the syntax:

```
begin when unit_test
  statement_list
end;
```

These blocks are only compiled in unit test builds. This approach can be used for testing routines

that manipulate hardware registers, including interrupt handlers. Within `unit_test` sections (i.e. the section at the end of the routine, not these blocks), the mapped devices can be read and written as ordinary variables without using this special syntax.

Conversely, some code is awkward to deal with under unit tests. An example might be a block of embedded C code that performs manipulation of hardware (as an alternative to defining the appropriate mapped devices etc.). This code can be excluded from unit test builds with:

```
begin when not unit_test
  statement_list
end;
```

Procedures and functions that are intended only as helper routines in unit test builds can be marked as such by applying the prefix `unit_test` to them:

```
unit_test procedure p(...) is
begin
...
end p;

unit_test function f(...) : t is
begin
...
end f;
```

These routines will only be compiled under unit test builds.

A `unit_test` procedure can itself contain a `unit_test` section, in which case that section will also be called as a unit test when the compiled program is run. If that is the only purpose of the procedure, then its main body can be set to a `null` statement:

```
unit_test procedure p(...) is
begin
  null;
unit_test
  statement_u1
  …
  statement_un
end p;
```

The `cprefix` and `cprefix_line` attributes are not applied under unit test builds, on the basis that they are likely to be specific to the target and hence not appropriate for the compiler host.

Routines that differ between unit test and non unit test builds can usually be handled by internal blocks that are conditional on `unit_test` as described above. However, in some cases this is not possible, typically when the non unit test version is a reference to an external C function, or it might be cumbersome. In these cases a non unit test version can be declared with the prefix `not unit_test`. Routines with this prefix are only compiled under non unit test builds.

```
not unit_test procedure p(...) is
begin
...
end p;

not unit_test function f(...) : t is
begin
...
end f;
```

Occasionally a unit test build needs additional global variables and other objects that are not required under non unit test builds (for test data, for example). This can be accomplished with a sub-package that has a `unit_test` prefix. Such a package will only be compiled under unit test builds.

```
unit_test package p is
...
end p;
```

As with blocks, procedures and functions, a corresponding non unit test version exists:

```
not unit_test package p is
...
end p;
```

Under unit test builds, the C symbol `__UNIT_TEST` is defined at the start of the application header file.

Procedures that contain a loop that needs to be supplied with different values as the test proceeds are dealt with as follows. The procedure must be declared with `using (unit_test_with_resume)`[29], `pragma unit_test_resume` is placed at the top of the loop, and `pragma unit_test_return` is placed at the end of the loop.

```
procedure p using (unit_test_with_resume) is
  local variable declarations
begin
  initialisation
  loop
    pragma unit_test_resume;
    loop body with possible exit
    pragma unit_test_return;
  end loop;
  more_statements
unit_test
  p;
  …
  p;
  …
  etc.
end p;
```

The first time that *p* is called from the unit test section, the code starting from *initialisation* is run, and continues up to the point where `unit_test_return` appears. On subsequent calls to *p*, the code starts running at the position of `unit_test_resume` and runs up to `unit_test_return`. The values of the local variables of *p* are preserved between these calls. This allows the unit test to simulate several loops of *p*, modifying its global environment in between if necessary, but with *p* maintaining its local state as it would in a running system. If the loop exits, then *more_statements* are run as usual and the procedure returns. In this case, the loop control is reset so that the next time the procedure is called it will start from *initialisation*. This allows it to be used in subsequent tests.

`Unit_test_with_resume`, `unit_test_resume` and `unit_test_return` have no effect when the system is not built for unit testing.

[29]This is an abbreviation for 'using (unit_test_with_resume => true)'

29 Predefined packages

29.1 Package SystemInformation

This library package can be imported in order to give some information about the state of the running system.

```
package SystemInformation using (usage => library) is

    public unit SubSystemIdentity;

    MAX_SUBSYSTEMS : constant := implementation-defined;

    public type SubSystemID is public range 0..MAX_SUBSYSTEMS - 1
      unit SubSystemIdentity;

    public type LineNumber is public range 0..implementation-defined;

    // Return an identifier for the system or subsystem of the caller
    public function GetSubsystemID : SubSystemID is implementation-defined;

    // Return the line number of the most recent restart of the caller's
    // system or subsystem, or 0 if never restarted.
    public function GetRestartLine : LineNumber is implementation-defined;

    // Return a pointer to the null-terminated file name of the
    // restart of the caller's system or subsystem, or NULL if never restarted.
    public function GetRestartFile : string is implementation-defined;

end SystemInformation;
```

29.2 Package unchecked_conversions

Generic package `unchecked_conversions` converts from one type to another and compiles to a C cast.

```
generic
  type SourceType;
  type TargetType;
package unchecked_conversions using (usage => library) is

  public function convert_from(Source : in SourceType) : TargetType
    using (code => macro) is
  begin
  ! (`TargetType`)(`Source`)
  end convert_from;

  public procedure convert(SourceVar : in SourceType; TargetVar : out TargetType)
    using (require_named_parameters, code => macro) is
  begin
  ! *(`TargetVar`) = (`TargetType`)(`SourceVar`);
  end convert;

end unchecked_conversions;
```

Example

```
package unchecked_conversions is separate;
…
```

```
package T1toT2 is new unchecked_conversions (
  SourceType => T1,
  TargetType => T2
);
…
var1 : T1;
var2 : T2;
…
var2 := T1toT2.convert_from(var1);
…
T1toT2.convert(SourceVar => var1, TargetVar => var2);
```

30 Summary of pragmas

`unchecked_initialisation` `unchecked_initialization`	Declare that a variable should be regarded as initialised	73
`unchecked_finalisation` `unchecked_finalization`	Declare that a variable should be regarded as finalised	75
`unchecked_use`	Declare that a variable should be regarded as accessed Declare that a procedure or function should be regarded as used	74, 74 53
`finalised` `finalized`	Declare that an object is finalised	75
`hide`	Hide a variable	83
`runtime_check`	Insert a runtime check	49
`unchecked_reset`	Forget range information for a variable - reset it to the full range of its type	49
`unchecked_deadlock`	Declare that deadlock checking between locks of two specific variables should be suppressed	55
`test_assert`	Include a runtime check in test builds	49
`fail`	Declare that a unit test has failed	83
`succeed`	Declare that a unit test has passed	83
`incomplete`	Declare that a routine is not yet complete in order to suppress certain checks on it during program construction	82
`no_main`	Declare that the system has no main procedure	53
`no_c_main`	Declare that the programmer will supply a separate C main routine	53
`include`	Include C header files	42
`contiki_autostart`	Autostart C-based Contiki threads (contiki task model only)	57
`set_default_allocate`	Set the default memory allocator at package level	70
`set_default_free`	Set the default memory free routine at package level	70
`pre_dispatch`	Set C code to be emitted immediately before dispatches under the longjmp task model	57

post_dispatch	Set C code to be emitted immediately after dispatches under the longjmp task model	57
dispatch	Insert a subsystem dispatch call	57
unit_test_resume	Set the resume point for unit testing of loops	83
unit_test_return	Return from a loop under unit testing	83
stack_size	Specify subsystem stack size	58
stack_overhead	Specify the stack overhead for procedure calls or for subsystems	58
call_overhead	Specify the fixed stack overhead of procedure and function calls	58
interrupt_overhead	Specify the additional stack overhead of interrupts	58
maximum_object_size	Specify the maximum size of any object	78
target_integer_type	Declare an integer type that is available on the target	78
boolean_type	Declare which target integer type should be used for booleans	78
character_type	Declare which target integer type should be used for characters	78
universal_index_type	Declare the target integer type and the range of a universal array index	78
target_float_type	Declare a floating point type that is available on the target	78
target_stack_unit	Specify the target type to be used as the unit of stack size and overhead pragmas	78
clear_target_stack	Indicate that subsystem stacks should be zeroed prior to use under the longjmp task implementation	78
memory_move	Name the C function to be used for memory block moves	78
inline_prefix inline_prefix_line	Specify the prefix to be applied to C routines in order to inline them with `code => inline`	78
system_header	Write lines to the generated C header file of the system	42
describe	Write a description of the analysis of given expressions at a point in the program into the generated C code	83
describe_locals	Write a description of local variables at a point in the program into the generated C code	83
describe_scope	Write a summary of the scope at a point in the program into the generated C code	83
note	Emit a note during compilation	83

31 Summary of exemptions

`unchecked_side_effect`	Allow side-effects where they would otherwise be forbidden	44
`unchecked_ambiguous_order`	Allow a potentially ambiguous positional notation for parameters where named notation would otherwise be required	39
`unchecked_share`	Allow entities to be accessed across subsystem boundaries where this would otherwise be forbidden	55
`shared_atomic_access`	Allow variables and components of variables to be accessed across subsystem boundaries where this would otherwise be forbidden, but only if they are of atomic types	55
`unchecked_access_conversion`	Allow assignments between access types that would otherwise be forbidden	25
`unchecked_range`	Suppress range checking for assignments or for individual `in` parameters	44, 39
`unchecked_alias`	Allow aliasing of `renames` declarations and of procedure and function parameters	31, 39

32 Summary of access modes for variables

`volatile`	Access without intermediate storage	68
`mapped_device`	Volatile and exempt from the usual initialisation and usage order rules	68
`image`	A constant object as part of the program image	68
`shared_atomic`	An atomic variable that can be accessed directly across a subsystem boundary (implies volatile)	68, 52, 55
`virtual`	A variable without storage that is used for analysis and integrity checking	68