ISO/IEC JTC1/SC22
Languages
Secretariat: CANADA (SCC)

ISO/IEC JTC1/SC22
N 1172

JUNE 1992

TITLE:	WG2 Proposal for a NWI on: Extended Pascal Binding to Language-Independent Arithmetic
SOURCE:	Secretariat ISO/IEC JTC1/SC22
WORK ITEM:	N/A
STATUS:	New
CROSS REFERENCE:	N/A
DOCUMENT TYPE:	WG2 Proposal for a NWI for SC22
ACTION:	For review by SC22 Member Bodies. This proposal will be discussed at the forthcoming Plenary meeting.

PROPOSAL FOR A NEW WORK ITEM	
date of presentation of proposal 1992 – 06 – 10	proposer JTC1/SC22/WG2
secretariat BSI	ISO/IEC JTC

A **proposal for a new work item** shall be submitted to the secretariat of the ISO/IEC joint technical committee concerned, with a copy to the ISO Central Secretariat.

Presentation of the proposal — to be completed by the proposer

Guidelines for proposing and justifying a new work item are given in ISO Guide 26. For ease of reference an extract is given overleaf.

Title (subject to be covered and type of standard, e.g. terminology, method of test, performance requirements, etc.)

EXTENDED PASCAL BINDING TO LIA-1

Scope (and field of application) ACCESS TO LIA-1 ARITHMETIC FROM PASCAL PROGRAMS

Purpose and justification — attach a separate page as annex, if necessary

SEE ATTACHED SHEET

Programme of work

If the proposed new work item is approved, which of the following document(s) is (are) expected to be developed?

☒ a single International Standard

□ more than one International Standard (expected number: ......)

□ a multi-part International Standard consisting of ...... parts

□ an addendum or addenda to the following International Standard(s): ....

□ a technical report, type ......

Relevant documents to be considered

ISO/IEC 10206: 1991 (EXTENDED PASCAL)
ISO/IEC 7185: 1990 (PASCAL)
DIS 10967-1 (LANGUAGE-INDEPENDENT ARITHMETIC, PART 1)

Co-operation and liaison

THERE WILL BE CLOSE COOPERATION WITH JTC1/SC22/WG11 (LANGUAGE BINDINGS)

Preparatory work offered with target date(s)

DOCUMENT JTC1/SC22/WG2 N318 ATTACHED

Signature (CONVENOR)

	yes	no
Will the services of a maintenance agency or registration authority be required?	□	☒
If yes, have you identified a potential candidate	□	□
If yes, indicate name:		
Are there any known requirements for coding? If yes, please specify on a separate page	□	☒
Does the proposed standard concern known patented items? If yes, please provide full information at annex	□	☒

Comments and recommendations of the JTC secretariat — attach a separate page as annex, if necessary

Comments with respect to the proposal in general, and recommendations thereon
It is proposed to assign this new item to SC /WG

Voting on the proposal

Each P-member of the ISO/IEC joint technical committee has an obligation to vote within the time limits laid down (normally thre months after the date of circulation)

Date of circulation	Closing date for voting	Signature of the JTC secretary

NEW WORK ITEM PROPOSAL

LIA-1, part 1 of ISO/IEC 10967, Language-Independent Arithmetic, was formerly known as LCAS, the "Language-Compatible Arithmetic Standard". It defines integer and real computer arithmetic such that a program can obtain consistent and reliable results. The 2nd CD 10967 will be ready in mid-1992 for SC22 ballot. Subsequent parts of ISO/IEC 10967 will be:

Part 2: "Language-Compatible Mathematical Procedure Standard"

Part 3: "Language-Compatible Complex Arithmetic and Procedure Standard"

This work item is to produce a binding to LIA-1. Preparatory work on this binding has been done, as ISO/IEC 10967-1 has reached a sufficiently stable state. Work items to produce bindings to subsequent parts of ISO/IEC 10967 may be proposed in due course.

The Extended Pascal standard was published in 1991 as ISO/IEC 10206.

The Extended Pascal binding to LIA-1 will specify:

(1) requirements on an Extended Pascal processor so that its arithmetic will conform to LIA-1;

(2) an interface to an Extended Pascal module LIA_1 which will provide those LIA-1 facilities which are not available directly in the Extended Pascal language;

(3) the manner in which the user shall specify the basic arithmetic constants used by LIA_1;

(4) alternative requirements which will permit a (classic) Pascal processor which conforms to ISO/IEC 7185 but not necessarily to ISO/IEC 10206, to conform to LIA-1.

It will also include as informative annexes:

(a) an Extended Pascal implementation of the LIA_1 module;

(b) a sample basic arithmetic constants module, applicable to IEEE 754 double precision arithmetic;

(c) an Extended Pascal LIA-1 conformity checker, utilising the proposed binding.

Use of this binding will enable the Extended Pascal programmer to obtain all the benefits of ISO/IEC 10967-1.

The preparatory work done by ISO/IEC JTC1/SC22/WG2 on this binding is attached. Note: this is in no sense the complete standard, it is not even a Working Draft. It is circulated now for information and comment.

Contents

Introduction

LIA-1, part 1 of ISO/IEC 10967, Language-Independent Arithmetic, was formerly known as LCAS, the "Language-Compatible Arithmetic Standard". It is currently under development by X3T2 and ISO/IEC JTC1/SC22/WG11. Version 3.1 (document X3T2/91-073, WG11/N229) will be updated in mid-1992 for the 2nd CD 10967-1 SC22 ballot.

The LIA-1 Extended Pascal Binding consists of a module LIA_1 which exports an interface LIA_1 (to be imported by any program using LIA-1 arithmetic).

A possible implementation of LIA_1 is specified.

Module LIA_1 imports interface ARITH_CONSTS, which sets the values of the basic arithmetic constants (integer and floating point). A sample module is listed, which applies to IEEE 754 double precision arithmetic.

As an example of the use of the Binding, the LCAS Conformity Checker from Brian Wichmann's NPL report DITC 167/90 "Getting the Correct Answers" has been modified to import interface LIA_1 for its LIA-1 arithmetic.

The LIA-1 Extended Pascal Binding module and the LCAS Conformity Checker are available from the WG2 Convenor, either by email (daj@tees.ac.uk) or on PC/DOS floppy disk (state size and format required).

```
module LIA_1 interface;

  export
    LIA_1 = (
        integer,            { = integer }
        real,               { = real }

        maxint,             { = maxint }
        minint,
        bounded,
        radix,
        places,
        maxexp,
        minexp,
        denorm,

        protected maxreal,
        protected minrealn,
        protected minreal,
        protected epsreal,

        remi,

        signf,
        exponf,
        fractionf,
        scalef,
        succf,
        predf,
        ulpf,
        truncf,
        roundf,
        intpartf,           { intpartf(x) = ToReal(trunc(x)) }
        fractpartf          { fractpartf(x) = ToReal(x - trunc(x)) }
        );

  import ARITH_CONSTS;

  const
    bounded = true;

  var
    maxreal: real;
    minrealn: real;
    minreal: real;
    epsreal: real;

  function remi(i, j: integer): integer;
  function signf(x: real): real;
  function exponf(x: real): integer;
  function fractionf(x: real): real;
  function scalef(x: real; n: integer): real;
  function succf(x: real): real;
  function predf(x: real): real;
  function ulpf(x: real) = result: real;
  function truncf(x: real; n: integer): real;
  function roundf(x: real; n: integer): real;
  function intpartf(x: real): real;
  function fractpartf(x: real): real;

end { LIA_1 interface } .
```

```
module LIA_1 implementation;

  { hidden stuff }
  var
    MaxMantissa: real;  { Largest integer-valued real. }
    minreald: real;

  function RadixPower(i: integer) = temp: real;
    var j: integer;
    begin
      if i = 0 then temp := 1.0
      else if i > 0 then
        begin
          temp := radix;
          for j := 1 to i-1 do temp := temp * radix
        end
      else
        begin
          temp := 1.0 / radix;
          for j := 1 to abs(i)-1 do temp := temp / radix
        end
    end;

  function floor(x: real): real;
    var
      intpart: real;
      p: integer;
      negative: Boolean;
    begin
      if x = 0.0 then floor := 0.0
      else if exponf(x) >= places then floor := x
      else if exponf(x) < 0 then
        begin
          if x < 0.0 then floor := -1.0 else floor := 1.0
        end
      else
        begin
          negative := x < 0.0;
          x := abs(x);
          intpart := 0.0;
          while x >= 1.0 do
            if x <> 0.0 then
              begin
                p := 0;
                while x - p * RadixPower(exponf(x)) >= 0.0 do
                  p := p + 1;
                intpart := intpart + (p-1) * RadixPower(exponf(x));
                x := x - (p-1) * RadixPower(exponf(x))
              end;
          if negative then
            begin
              if x <> 0.0 then floor := - intpart - 1.0
              else floor := - intpart
            end
          else floor := intpart
        end
    end;


  {exported stuff }

  function remi { (i, j: integer): integer } ;
    begin
      if j = -1 then remi := 0 { to avoid overflow }
      else remi := i - (i div j) * j
    end;

  function signf { (x: real): real } ;
    begin
      if x >= 0.0 then signf := 1.0 else signf := -1.0
    end;

  function exponf { (x: real): integer } ;
    var e: integer;
    begin
      if x = 0.0 then halt; { Undefined }
      e := 0;
      x := abs(x);
      while (x >= radix) or (x < 1.0) do
        begin
          if x >= radix then
            begin x := x / radix; e := e + 1 end
          else
            begin x := x * radix; e := e - 1 end
        end;
      exponf := e
    end;

  function fractionf { (x: real): real } ;
    begin
      if x = 0.0 then fractionf := 0.0
      else fractionf := x / RadixPower(exponf(x))
    end;

  function scalef { (x: real; n: integer): real } ;
    begin
      if x = 0.0 then scalef := 0.0
      else scalef := fractionf(x) * RadixPower(exponf(x) + n)
    end;

  function succf { (x: real): real } ;
    begin
      if x = -minreal then succf := 0.0
      else if x = 0.0 then succf := minreal
      else if x < 0.0 then
        if (fractionf(-x) = fractionf(1.0)) and (abs(x) > minrealn) then
          succf := -fractionf(maxreal) * RadixPower(exponf(x)-1)
        else if denorm or (exponf(x) > minexp + places) then
        succf := x + ulpf(x)
      else
        succf := succf(RadixPower(places) * x) / RadixPower(places)
        { this case is to avoid underflow for machines without
          gradual underflow }
    else if denorm or (exponf(x) > minexp + places) then
      succf := x + ulpf(x)
    else
      succf := succf(RadixPower(places) * x) / RadixPower(places)
      { This case is to avoid underflow for machines without
        gradual underflow }
  end;

function predf { (x: real): real } ;
  begin predf := - succf(-x) end;

function ulpf { (x: real) = result: real } ;
  begin
    if x = 0.0 then halt; { undefined }
    if abs(x) >= minrealn then
      result := RadixPower(exponf(x) + 1 - places)
    else
      result := RadixPower(minexp - places);
    if result = 0.0 then halt { underflow }
  end;

function truncf { (x: real; n: integer): real } ;
  begin
    if x < 0.0 then truncf := - truncf(-x,n)
    else if x < minrealn then
      truncf := floor(x/RadixPower(minexp-n))*RadixPower(minexp-n)
    else
      truncf := floor(x/RadixPower(exponf(x)+1-n))*RadixPower(exponf(x)+1-n)
  end;

function roundf { (x: real; n: integer): real } ;

  function rnf(x: real; n: integer): real;
    begin
      if abs(x) < minrealn then
        rnf := signf(x) * floor(abs(x) / RadixPower(minexp-n) + 0.5) *
                          RadixPower(minexp-n)

      else
        rnf := signf(x) * floor(abs(x) / radixPower(exponf(x)+1-n) + 0.5) *
                          RadixPower(exponf(x)+1-n)

    end;

  begin
    if n <= 0 then halt; { undefined }
    if n >= places then roundf := x
    else roundf := rnf(x,n)
  end;

function intpartf { (x: real): real } ;
```

```
    begin
      if abs(x) < 1.0 then intpartf := 0.0
      else intpartf := truncf(x,exponf(x)+1)
    end;

  function fractpartf { (x: real): real } ;
    begin fractpartf := x - intpartf(x) end;


  { initialization }

  procedure ComputeConstants;
    var i: integer;
    begin
      { compute maxreal }
      { maxreal := (RadixPower(places)-1.0)*RadixPower(maxexp-places) }
      MaxMantissa := radix - 1;
      for i := 1 to places-1 do
        MaxMantissa := radix * MaxMantissa + (radix-1);
      maxreal := MaxMantissa * RadixPower(maxexp-places);
      { compute epsilon }
      epsreal := RadixPower(1-places);
      { compute minrealn }
      minrealn := RadixPower(minexp-1);
      { compute minreald }
      if denorm then minreald := RadixPower(minexp-places);
      { compute minreal }
      if denorm then minreal := minreald else minreal := minrealn;
    end;

  to begin do ComputeConstants;

end { LIA_1 implementation } .
```

```
module ARITH_CONSTS;
  export
    ARITH_CONSTS = (
        maxint,          { = maxint }
        minint,

        radix,
        places,
        maxexp,
        minexp,
        denorm
        );

  const

    { integer characterization -- assumes LIA_1.maxint = maxint }
    TwosComplement = true;  { not an LIA-1 property }
    minint = -maxint - ord(TwosComplement) ;
    { WARNING:  an implementation may reject numbers less than -maxint }

    { floating point -- the following are set for IEEE Double format }
    radix = 2;
    places = 53;
    maxexp = 1024;
    minexp = -1021;
    denorm = true;

  end { ARITH_CONSTS interface } ;

  end { ARITH_CONSTS implementation } .
```

```
{ Crown Copyright 1989 }
{ Author: B A Wichmann
          National Physical Laboratory
          Teddington, Middlesex,
          TW11 0LW.
  e-mail: baw@seg.npl.co.uk
  }

{ 1991-03-14  Jim Miner
              Modified to import LIA_1 interface and use imported
              entities in place of those previously defined in this
              program. }


{ ****************************************************************** }

{ This program is a 'Model Implementation' of the
      Language Compatible Arithmetic Standard
  in ISO-Pascal. The purpose of this implementation is to

  1) Provide a version of LCAS which can be easily converted to
     any programming language;
  2) Provide a tool for checking an existing implementation of
     LCAS (although the checks are rather minimal);
  3) Provide an executable version of LCAS in order to demonstrate
     how the standard works.

  This model implementation does not aim to be efficient, since
  that would require knowledge of the machine representation in
  order to do such things as masking out the exponent of a floating
  point number.

  This program assumes that the underlying implementation of the
  following facilities already conform to LCAS:

  +, -, *, div  for type integer
  - (negate)    for type integer
  abs           for type integer
  +, -, *, /    for type real
  - (negate)    for type real
  abs, sqrt     for type real

  Rudimentary checks are included in this program in case the above
  assumption proves to be false. Although this program is useful for
  determining conformity with LCAS, more detailed testing is needed,
  such as that provided by the NAG FPV package. (Enquires to NAG Ltd
  Wilkinson House, Jordan Hill Road, OXFORD OX2 8DR, UK or NAG Inc,
  1400 Opus Place, Suite 200, Downers Grove, IL 60515-5702, USA.)

  Some machines, particularly those with IEEE hardware, can perform
  computations with higher precision than the main floating point data
  types. If this facility is visible to the Pascal program, then this
  program will report non conformity with the LCAS. This statement is
  not strictly correct, since conformity can be claimed by stating how
```

each Pascal expression is converted into the LCAS primitive operations. For these machines, implicit conversions between the register type and the Pascal type real would be needed. It is not feasible to allow for this within a (simple) Pascal program.

The ISO-Pascal standard requires that if an error is detected, there is a mode in which this detection causes the program to halt. Hence this program assumes that the LCAS concept of 'notification' causes the program to halt. This implies that only one form of notification can be checked for each program execution. To overcome this difficulty, the program uses the file parameter input which has one integer on it. Each test for notification increments this number, provided the program is halted. Hence this program is run 27 times by setting the integer on the file to the values from 1 to 27. The combined output from all the executions gives the results of testing a system. You should be able to run the program 27 times automatically without interactive execution (it depends upon the compiler environment).

Rather than attempt to compute the characteristics of the underlying machine by various tricks, the program requires that the main properties are inserted as constants in the source text. These are given a zero value in the distributed version to ensure that they are reset for each machine.

The effect of incorrectly setting one constant is as follows:

Problem		Result (comment in brackets)
Radix	incorrect	(unpredictable) (N hex places is rather similar to 4*N binary places)
Places	too large:	Floating Point Overflow (before any output)
	too small:	Extended Accuracy Reals - not LCAS Rounding Inconsistent
ExpoMin	too large:	Exponent Range not roughly symetric fmin not correct -- check ExpoMin
	too small:	Notification: undefined (after output of parameter values)
ExpoMax	too large:	Exponent Range not roughly symetric then Floating Point Overflow (before output of parameter values)
	too small:	Notification does not occur on test 21 (test for succF(fmax))
denorm	true and should be false:	fmin is zero, check ExpoMin and denorm
	false and should be true:	fmin not correct -- check ExpoMin

An unfortunate fact is that some compilers perform optimizations which can prevent this program from testing what was intended. For instance, perhaps an expression is written as 0.0 * expr, with the intension of seeing if a notification arises from an overflow in the expression. Clearly an optimizer can avoid evaluation of the expression completely, so no notification would be observed. In general, optimising compilers which assume that the program is legal Pascal

```
  (and hence will not cause notification) cannot be safely used with
  this program. Hence, one should term off any optimization switch,
  although it is more important to test the compiler/system in the mode
  in which it is being used.

  Acknowledgement:
  Many thanks to Martha Jaffe from DEC who commented extensively on
  Version 2.2a; most of the points arising have been handled.
  }

{           ******** Operating Instructions ************
  Read the manual for the version of Pascal being tested in order to
  set the first six constants listed below. If problems arise, one
  may need to set the constant Trace to true. Values of these constants
  for common implementations appear in the annex to LCAS.
  Compile the program -- warnings may be issued due to the presence of
  code which will not be executed for specific values of the constants.
  Execute the program 27 times with the integer data values 1..27,
  putting the output into a file. This file constitutes a test report on the
  implementation. Edit the file to include vital information such as
  details of the computer and compiler. See the example after listing.
  Please review the report careful: for instance, on many machimes there
  are vital options to include such as hardware co-processor, compiling
  options, etc.

  The execution time is only significant for the data value of 1, taking
  about two minutes on a PC level machine.
  Report the results to NPL. }

program LCASimpl(input, output);

import LIA_1 qualified;

label
   13; { for internal notifications }
const
   Radix   = LIA_1.radix;
   Places  = LIA_1.places;
   ExpoMin = LIA_1.minexp;
   ExpoMax = LIA_1.maxexp;
   denorm  = LIA_1.denorm;

   TwosComplement = true; { This value is used to compute minint since
                            Pascal does not allow minint to be given as
                            a literal value. }
                          { NOT USED. }
   Version = '2.2c';      { The version of this program (preceded by version
                            of LCAS). }
   Trace = false;
   Digits = 25; { set to width for printing, if needed }

type
   notify = (ZeroDivide, Overflow, Underflow, Undefined);
```

```
var
   { The values of the derived constants are computed }
   fmax, fminN, fminD, fmin, epsilon : real;
     { fminD NOT USED. }  { OTHERS OBTAINED FROM LIA_1 MODULE }

   minint: integer; { Computed, while maxint is predefined in Pascal }

   MaxMantissa: real; { Largest integer-valued real. }

   CaseValue: integer; { Value for notification check }

 { The LCAS operations are provided as follows:

   addI(x,y)     x + y   (underlying system)
   subI(x,y)     x - y   (underlying system)
   mulI(x,y)     x * y   (underlying system)
   divI(x,y)     x div y (underlying system)
   remI(x,y)     remI(x,y)
   modI(x,y)     x mod y (underlying system)
   negI(x)       - x     (underlying system)
   absI(x)       abs(x)  (underlying system)
   eqI(x,y)      x = y   (underlying system)
   neqI(x,y)     x <> y  (underlying system)
   lssI(x,y)     x < y   (underlying system)
   leqI(x,y)     x <= y  (underlying system)
   gtrI(x,y)     x > y   (underlying system)
   geqI(x,y)     x >= y  (underlying system)

   addF(x,y)     x + y   (underlying system)
   subF(x,y)     x - y   (underlying system)
   mulF(x,y)     x * y   (underlying system)
   divF(x,y)     x / y   (underlying system)
   negF(x)       - x     (underlying system)
   absF(x)       abs(x)  (underlying system)
   sqrtF(x)      sqrt(x) (underlying system)
   signF(x)      signF(x)
   exponF(x)     exponF(x)
   signifF(x)    signifF(x)
   scaleF(x,n)   scaleF(x,n)
   succF(x)      succF(x)
   predF(x)      predF(x)
   ulpF(x)       ulpF(x)
   truncF(x,n)   truncF(x,n)
   roundF(x,n)   roundF(x,n)
   intF(x)       intF(x)
   fractF(x)     fractF(x)
   eqF(x,y)      x = y   (underlying system)
   neqF(x,y)     x <> y  (underlying system)
   lssF(x,y)     x < y   (underlying system)
   leqF(x,y)     x <= y  (underlying system)
   gtrF(x,y)     x > y   (underlying system)
   geqF(x,y)     x >= y  (underlying system)
   cvtIF(x)      (implicit in underlying system)
   cvtFI(x)      round(x) (underlying system)
```

```
  cvtFI(x)     trunc(x) (underlying system)
}

function RadixPower(i: integer): real;
 { = Radix ** i }
   var
      temp: real;
      j: integer;
   begin
   if i = 0 then
      RadixPower := 1.0
   else if i > 0 then
      begin
      if Trace and (i >= ExpoMax) then
         writeln('Exponent too large');
      temp := Radix;
      for j := 1 to i-1 do
         temp := temp * Radix;
      RadixPower := temp
      end
   else
      begin
      temp := 1.0/Radix;
      for j := 1 to abs(i)-1 do
         temp := temp/Radix;
      RadixPower := temp
      end;
   end;

procedure InitialChecks;
   begin
   if odd(Radix) or (Radix < 0) then
      writeln('Radix value is not positive even integer');
   if (Places-1)*ln(Radix) < ln(1.0e6) then
      writeln('Precision less than six decimal places');
   if (ExpoMin-1) >= -2*(Radix-1) then
      writeln('Exponent minimum too large');
   if ExpoMax <= 2*(Radix-1) then
      writeln('Exponent maximum not large enough');
   if (-2 > ExpoMin-1+ExpoMax) or (ExpoMin-1+ExpoMax > 2) then
      writeln('Exponent range not roughly symmetric');
   end;

procedure ComputeConstants;
   var
      i: integer;
   begin
   if (Radix=0) or (Places=0) or (ExpoMin=0) or (ExpoMax=0) then
      writeln('Set constants and recompile');
   { compute minint }
   minint := LIA_1.minint;
 {   if TwosComplement then
      minint := -maxint - trunc(1.0)  }
      { The call of trunc has been added to ensure compilation on
```

```
        systems which fold constant integer expressions and do compile
        time range checking. }
  { else
      minint := -maxint; } { One's complement machine }
    { compute fmax }
    { fmax := (RadixPower(Places)-1.0)*RadixPower(ExpoMax-Places); }
    MaxMantissa := Radix - 1;
    for i := 1 to Places - 1 do
       MaxMantissa := Radix * MaxMantissa + (Radix-1);
    fmax := LIA_1.maxreal;
    { compute epsilon }
    epsilon := LIA_1.epsreal;
    { compute fminN }
    fminN := LIA_1.minrealn;
    { compute fminD }
    if denorm then
       fminD := RadixPower(ExpoMin-Places);
    { compute fmin }
    fmin := LIA_1.minreal;
  {   if denorm then
        fmin := fminD
     else
        fmin := fminN; }
     if fmin = 0.0 then
        writeln('fmin is zero, check ExpoMin and denorm');
     if fmin / Radix <> 0.0 then
        writeln('fmin not correct -- check ExpoMin')
        { This test can fail on Extended register machines (see above),
          since fmin/Radix will be computed with greater precision/range. }
     end;

  procedure InternalNotification(N: notify);
     begin
     case N of
      ZeroDivide: writeln('Notification: zero divide');
      Overflow:   writeln('Notification: overflow');
      Underflow:  writeln('Notification: underflow');
      Undefined:  writeln('Notification: undefined');
     end;
     { If global goto are not permitted, the next statement should be
       replaced by a call of 'halt' or similar extension to halt the
       program. The test report should indicate that this change was
       necessary. }
   { goto 13; }  halt;
     end;

  function ulpF(x: real): real; forward;

  function exponF(x: real): integer; forward;

  function floor(x: real): real;
     var
        intpart: real;
        p: integer;
      negative: boolean;
   begin
   if x = 0.0 then
      floor := 0.0
   else if exponF(x) >= Places then
      floor := x
   else if exponF(x) < 0 then
      begin
      if x < 0.0 then
         floor := -1.0
      else
         floor := 0.0
      end
   else
      begin
      negative := x < 0.0;
      x := abs(x);
      intpart := 0.0;
      while x >= 1.0 do
         if x <> 0.0 then
            begin
            p := 0;
            while x - p*RadixPower(exponF(x)) >= 0.0 do
               p := p + 1;
            intpart := intpart + (p-1)*RadixPower(exponF(x));
            x := x - (p-1)*RadixPower(exponF(x))
            end;
      if negative then
         begin
            if x <> 0.0 then
               floor := - intpart - 1.0
            else
               floor := - intpart
         end
      else
         floor := intpart
      end;
   end;

{ The main optional functions }

function remI(x, y: integer): integer;
   begin  remI := LIA_1.remi(x,y)  end;

function signF(x: real): real;
   begin  signF := LIA_1.signf(x)  end;

function exponF { (x: real): integer };
   begin  exponF := LIA_1.exponf(x)  end;

function signifF(x: real): real;
   begin  signifF := LIA_1.fractionf(x)  end;

function scaleF(x: real; n: integer): real;
```

```
   begin  scaleF := LIA_1.scalef(x,n)  end;

function succF(x: real): real;
   begin  succF := LIA_1.succf(x)  end;

function predF(x: real): real;
   begin  predF := LIA_1.predf(x)  end;

function ulpF { (x: real): real } ;
   begin  ulpF := LIA_1.ulpf(x)  end;

function truncF(x: real; n: integer): real;
   begin  truncF := LIA_1.truncf(x,n)  end;

function roundF(x: real; n: integer): real;
   begin  roundF := LIA_1.roundf(x,n)  end;

function intF(x: real): real;
   begin  intF := LIA_1.intpartf(x)  end;

function fractF(x: real): real;
   begin  fractF := LIA_1.fractpartf(x)  end;

{ These two procedures check the functions }

procedure FinalChecks;
   procedure EqualI(I,J: integer; TestNumber: integer);
      begin
      if I <> J then
         writeln('Integer operation check fails number ',
                  TestNumber:1, ' ', I:1, ' ', J:1)
      else if Trace then
         writeln('Test OK for ', TestNumber)
      end;
   procedure EqualF(X,Y: real; TestNumber: integer);
      begin
      if X <> Y then
         begin
         writeln('Floating point operation check fails number ',
                  TestNumber:1);
         writeln('   ',X:Digits, ' ', Y:Digits)
         end
      else if Trace then
         writeln('Test OK for ', TestNumber)
      end;
   procedure TestTrue(B: boolean; TestNumber: integer);
      begin
      if not B then
         writeln('Predicate fails, number ', TestNumber:1)
      else if Trace then
         writeln('Test OK for ', TestNumber)
      end;
   procedure CheckPowers;
      { This procedure ensures that RadixPower calculates powers of
```

```
        the radix correctly. }
      var
         max, min, step, a, b, TestNumber: integer;
         temp, temp1: real;
      begin
      max := ExpoMax - 1;
      if denorm then
         min := ExpoMin - Places
      else
         min := ExpoMin - 1;
      TestNumber := 100;
      step := (max-min) div 10 + 1; { To reduce tests to a reasonable number}
      a := min;
      while a < max do
         begin
         temp := RadixPower(a);
         EqualI(exponF(temp), a, TestNumber);
         TestNumber := TestNumber + 1;
         temp1 := temp;
         for b := 1 to Radix-1 do
            temp := temp + temp1;
         EqualF(temp, RadixPower(a+1), TestNumber);
         TestNumber := TestNumber + 1;
         TestTrue(temp1 < temp, TestNumber);
         TestNumber := TestNumber + 1;
         b := a;
         while (a + b >= min) and (a + b <= max) and (b <= max) do
            begin
            EqualF(RadixPower(a)*RadixPower(b), RadixPower(a+b), TestNumber);
            TestNumber := TestNumber + 1;
            b := b + step
            end;
         a := a + step
         end;
      if Trace then
         writeln('Radix power checks ', TestNumber-100)
      end;
   procedure CheckExactSquares;
      { This procedure checks that sqrt(x*x) = x when x*x is exact }
      var
         x, y: real;
         count: integer;
         fail: boolean;
      begin
      fail := false;
      x := 10.0;
      count := 0;
      while exponF(x) <= Places div 2 do
         begin
         y := floor(x*x);
         if sqrt(y) <> x then
            if not fail or Trace then
               begin
               writeln('square root not exact for a square', x:Digits);
            fail := true
            end;
      count := count + 1;
      x := floor(1.2*x)
      end;
   if Trace then
      writeln(count, ' equality tests for square root done')
   end;
procedure CheckConversions;
   { This procedure checks integer-real and real-integer conversions. }
   var
      MaxCommon, i, j, last: integer;
      x: real;
      sig: boolean;
   begin
   if maxint < MaxMantissa then
      MaxCommon := maxint
   else
      MaxCommon := trunc(MaxMantissa);
   last := 1;
   while last < MaxCommon div 2 do
      begin
      for i := -1 to 1 do
         if 2*last - MaxCommon < -i then
            for sig := false to true do
               begin
               if sig then
                  j := 2 * last + i
               else
                  j := -2 * last - i;
               x := j;
               { The above integer to real conversion is implicit in
                 Pascal. The 1990 revision of ISO-Pascal notes explicitly
                 that this conversion is approximate. Prior to this
                 revision, implementations may have chosen a precision
                 for real to ensure this conversion is exact. Hence we
                 check that the conversion is exact over the common
                 integer/real range. }
               if (j <> x) or (j <> trunc(x)) or (j <> round(x)) then
                  writeln('Error with equality for conversions')
               end;
      last := 2 * last
      end;
   end;
begin
if CaseValue = 1 then
   begin
   CheckPowers;
   EqualI(-(-maxint), maxint, 1);
   EqualI(2+2, 2*2, 2);
   EqualI(remI(minint,-1), 0, 3);
   EqualF(1.0+1.0, 2.0, 4);
   EqualF(fmax-1.0, fmax, 5);
   EqualF(fmax/2.0+fmax/2.0, fmax, 6);
EqualF(fmax/fmax, 1.0, 7);
EqualF(fmax/Radix*Radix, fmax, 7);
EqualF(fmin/fmin, 1.0, 8);
EqualF(-(-1.1), 1.1, 9);
EqualF(abs(-fmax), fmax, 10);
EqualF(abs(-fminN), fminN, 11);
EqualF(signF(-fmin), -1.0, 12);
EqualF(signF(0.0), 1.0, 13);
EqualF(signF(fmin), 1.0, 14);
EqualI(exponF(1.0), 0, 15);
EqualI(exponF(1.6), 0, 16);
EqualI(exponF(Radix), 1, 17);
EqualI(exponF(fmax), ExpoMax-1, 18);
EqualI(exponF(fminN), ExpoMin-1, 19);
if denorm then
   EqualI(exponF(fmin), ExpoMin-Places, 20);
EqualF(signifF(1.1), 1.1, 21);
EqualF(signifF(1.0), 1.0, 22);
EqualF(signifF(fmax), predF(Radix), 23);
EqualF(signifF(-fmin), -1.0, 24);
EqualF(scaleF(1.1, 1), 1.1*Radix, 25);
EqualF(scaleF(scaleF(1.7, 11), -11), 1.7, 26);
EqualF(succF(1.0), 1.0+epsilon, 27);
EqualF(succF(signifF(fmax)), Radix, 28);
EqualF(succF(-fmin), 0.0, 29);
EqualF(succF(0.0), fmin, 30);
EqualF(predF(succF(fmin)), fmin, 31);
TestTrue(predF(Radix) < Radix, 32);
TestTrue(predF(1.1) < 1.1, 33);
EqualF(predF(succF(1.2)), 1.2, 34);
EqualF(ulpF(1.0), epsilon, 35);
EqualF(Radix*ulpF(predF(1.0)), epsilon, 36);
EqualF(succF(predF(fmax)), fmax, 37);
EqualF(truncF(1.0 + 3*epsilon, Places), 1.0 + 3*epsilon, 38);
EqualF(truncF(1.0 + 3*epsilon, Places-1), 1.0 + 2*epsilon, 39);
EqualF(truncF(1.0 + 3*epsilon, Places-2), 1.0, 40);
EqualF(roundF(1.0 + 3*epsilon, Places), 1.0 + 3*epsilon, 41);
EqualF(roundF(1.0 + 3*epsilon, Places-1), 1.0 + 4*epsilon, 42);
EqualF(roundF(1.0 + 3*epsilon, Places-2), 1.0 + 4*epsilon, 43);
EqualF(intF(1.0), 1.0, 44);
EqualF(intF(succF(1.0)), 1.0, 45);
EqualF(intF(predF(2.0)), 1.0, 46);
EqualF(intF(-fmin), 0.0, 47);
EqualF(intF(fmin), 0.0, 48);
EqualF(fractF(fmax), 0.0, 49);
EqualF(fractF(fmin), fmin, 50);
EqualF(fractF(succF(1.0)), epsilon, 51);
EqualF(fractF(Radix), 0.0, 52);
EqualF(fractF(-fmin), -fmin, 53);
TestTrue(fmin > 0.0, 54);
TestTrue(-fmax < -fmin, 55);
CheckExactSquares;
EqualI(trunc(3.5), 3, 56);
EqualI(round(3.5), 4, 57);
```

```
      EqualI(round(-3.5), -4, 58);
      EqualF(floor(-5.0), -5.0, 59);
      EqualF(floor(-5.5), -6.0, 60);
      EqualF(scaleF(fminN, ExpoMax+1), RadixPower(ExpoMax+ExpoMin), 61);
      EqualF(scaleF(fmax, ExpoMin-2),
             signifF(fmax) * RadixPower(ExpoMax+ExpoMin-3), 62);
      CheckConversions;
      end
   end;

procedure FindRoundingMode;
   { Use multiplication of values near 1.0 to determine the
     rounding mode, assuming it is one of the conventional modes. }
   type
      Round = (down, up);
      Mode = (ToZero, Minus, Plus, Unbiased, Nearest);
   var
      a, b, count, tests: integer;
      res: Round;
      NotThis : array [Mode] of integer;
      ShouldBe, M, ResultM: Mode;
      positive, failure: boolean;
   function Actual(a, b: integer; positive: boolean): Round;
      var
         x, y, p, rem: real;
         low, high: integer;
      begin
      x := 1.0 + a*RadixPower(-(Places div 2));
      y := 1.0 + b*RadixPower(-Places - 1 + (Places div 2));
      if not positive then
         y := - y;
      p := abs(x * y) - 1.0;
      rem := (p - a*RadixPower(-(Places div 2))
                - b*RadixPower(-Places - 1 + (Places div 2)))/epsilon;
      low := a*b div (Radix*Radix);
      high := low + 1;
      if ((rem < low) or (rem > high)) and not failure then
         begin
         failure := true;
         writeln('Multiply does not round',
                  rem, positive, a:3, b:3);
         { This failure indicates that multiply does not round. This
         can happen on high performance machines which produce an
         approximate result fast rather than a properly rounded result. }
         end;
      if ((rem <> low) and (rem <> high)) and not failure then
         begin
         failure := true;
         writeln('Extended accuracy reals - not LCAS',
                  rem, positive, a:3, b:3);
         { This failure will arise with machines having extended
         registers. The value 'rem' printed will be a proper fraction,
         which should be truncated/rounded for LCAS comformance. }
         end;
      if (rem = low) and positive then
         Actual := down
      else if (rem = high) and not positive then
         Actual := down
      else
         Actual := up
      end;
   function Predict(a, b: integer; positive: boolean; M: Mode): Round;
      var
         rem: integer;
         temp: Round;
      begin
      case M of
      ToZero:   if positive then
                   Predict := up
                else
                   Predict := down;
      Minus:    Predict := down;
      Plus:     Predict := up;
      Unbiased: begin
                rem := a*b mod (Radix*Radix);
                if rem < Radix*(Radix div 2) then
                   temp := down
                else if rem > Radix*(Radix div 2) then
                   temp := up
                else if odd(a*b div (Radix*Radix)) then
                   temp := up
                else
                   temp := down;
                if not positive then
                   if temp=up then
                      Predict := down
                   else
                      Predict := up
                else
                   Predict := temp
                end;
      Nearest:  begin
                rem := a*b mod (Radix*Radix);
                if rem < Radix*(Radix div 2) then
                   temp := down
                else
                   temp := up;
                if not positive then
                   if temp=up then
                      Predict := down
                   else
                      Predict := up
                else
                   Predict := temp
                end;
      end { case }
      end;
   procedure PrintArray;
```

```
      begin
      writeln('Rounding   counter-examples');
      writeln('ToZero   ',NotThis[ToZero]);
      writeln('Minus    ',NotThis[Minus]);
      writeln('Plus     ',NotThis[Plus]);
      writeln('Unbiased ',NotThis[Unbiased]);
      writeln('Nearest  ',NotThis[Nearest])
      end;
   begin
   failure := false;
   ShouldBe := Nearest; {alter to expected rounding for diagnostics.}
   if CaseValue = 1 then
      begin
      for M := ToZero to Nearest do
         NotThis[M] := 0;
      tests := Radix*Radix*Radix;
      if tests > 30 then
         tests := 30;
      for a := 1 to tests do
      for b := a to tests + 1 do
      for positive := false to true do
         begin
         res := Actual(a, b, positive);
         for M := ToZero to Nearest do
            if Predict(a, b, positive, M) <> res then
               begin
               NotThis[M] := NotThis[M] + 1;
               if (M = ShouldBe) and Trace then
                  writeln('Unexpected rounding',
                           positive, a:3, b:3, ord(res):2);

               end
         end;
      count := 0;
      for M := ToZero to Nearest do
         if NotThis[M] = 0 then
            begin
            ResultM := M;
            count := count + 1
            end;
      writeln('Rounding on multiplication appears to be:');
      if count = 1 then
         case ResultM of
         ToZero:   writeln('ToZero   ');
         Minus:    writeln('Minus    ');
         Plus:     writeln('Plus     ');
         Unbiased: writeln('Unbiased ');
         Nearest:  writeln('Nearest  ')
         end
      else
         writeln('Inconsistent');
      if Trace or (count <> 1) then
         PrintArray
      end
   end;
```

```
procedure Notification(CaseValue: integer);
   const
      CaseMax = 27;
   var
      I, tempI1, tempI2: integer;
      MyMaxint: integer; { Introduced to prevent compile-time detection
                           of overflow etc }
      F, tempF1: real;
   begin
   if fmax > 0.0 then
      MyMaxint := maxint
   else
      MyMaxint := 5;
   if CaseValue > CaseMax then
      writeln('No test for this case');
   if (CaseValue > 0) and (CaseValue <= CaseMax) then
      begin
      case CaseValue of
      1:
         begin
         writeln(' 1   addI overflow pos  Overf');
         I := MyMaxint + 1;
         end;
      2: begin
         writeln(' 2   addI overflow neg  Overf');
         tempI1 := -minint; tempI2 := -1;
         I := tempI1 + tempI2
         end;
      3:
         begin
         writeln(' 3   subI overflow neg  Overf');
         I := minint - 1
         end;
      4: begin
         writeln(' 4   subI overflow pos  Overf');
         tempI1 := maxint; tempI2 := -1;
         I := tempI1 - tempI2
         end;
      5: begin
         writeln(' 5   mulI overflow pos  Overf');
         tempI1 := MyMaxint div 2 + 1; tempI2 := 2;
         I := tempI1 * tempI2
         end;
      6: begin
         writeln(' 6   mulI overflow neg  Overf');
         tempI1 := -2; tempI2 := MyMaxint div 2 + 2;
         I := tempI1 * tempI2
         end;
      7: begin
         writeln(' 7   int divide by zero ZeroD');
         tempI1 := 1; tempI2 := MyMaxint - maxint;
         I := tempI1 div tempI2
         end;
```

```
 8:
   if TwosComplement then
      begin
      writeln(' 8   divI overflow       Overf');
      I := minint div (-1)
      end
   else
      begin
      writeln(' 8   int divide by zero ZeroD');
      I := 1 div (MyMaxint-maxint)
      end;
 9:
   begin
   writeln(' 9   remI divide by 0   ZeroD');
   I := remI(1, MyMaxint - maxint)
   end;
10: begin
   writeln('10   modI divide by 0   ZeroD');
   tempI1 := 1; tempI2 := MyMaxint - maxint;
   I := tempI1 mod tempI2
   end;
11: begin
   writeln('11   modI by -maxint    Overf');
   tempI1 := 1; tempI2 := -MyMaxint;
   I := tempI1 mod tempI2
   end;
12:
   if TwosComplement then
      begin
      writeln('12   negI overflow       Overf');
      I := - minint
      end
   else
      begin
      writeln('12  divide by zero       ZeroD');
      I := 1 div (MyMaxint-maxint);
      end;
13:
   if TwosComplement then
      begin
      writeln('13   absI overflow       Overf');
      I := abs(minint)
      end
   else
      begin
      writeln('13  divide by zero       ZeroD');
      I := 1 div (MyMaxint-maxint)
      end;
14:
   begin
   writeln('14   addF overflow      Overf');
   F := fmax + RadixPower(ExpoMax-Places+1)
   end;
15:
    begin
    writeln('15   subF overflow       Overf');
    F := -fmax - RadixPower(ExpoMax-Places+1)
    end;
16:
    begin
    writeln('16   mulF overflow       Overf');
    F := fmax * 1.001
    end;
17:
    begin
    writeln('17   divF overflow       Overf');
    F := fmax / 0.7
    end;
18: begin
    writeln('18   divF by zero        ZeroD');
    tempF1 := MyMaxint-maxint;
    F := 1.0 / tempF1
    end;
19:
    begin
    writeln('19  sqrt of tiny neg     Undef');
    F := sqrt(-fmin)
    end;
20: begin
    writeln('20   exponF(zero)        Undef');
    tempF1 := MyMaxint-maxint;
    I := exponF(tempF1)
    end;
21:
    begin
    writeln('21   succF of fmax       Overf');
    F := succF(fmax)
    end;
22:
    begin
    writeln('22   predF of -fmax      Overf');
    F := predF(-fmax)
    end;
23:
    begin
    writeln('23   ulpF(zero)          Undef');
    F := ulpF(0.0)
    end;
24:
    begin
    writeln('24  roundF to 0 places  Undef');
    F := roundF(1.0, 0)
    end;
25:
    begin
    writeln('25   roundF  overflow   Overf');
    F := roundF(fmax, 2)
    end;
    26:
        begin
        writeln('26   trunc  overflow    Overf');
        if maxint < MaxMantissa then
           I := trunc(maxint+1.0)
        else
           I := trunc(succF(maxint))
        end;
    27:
        begin
        writeln('27  round overflow      Overf');
        if maxint < MaxMantissa then
           I := round(-maxint-1.0)
        else
           I := round(predF(-maxint))
        end;

      end
      end
   end;

  begin
  { Read and increment CaseValue }
  read(CaseValue);
  InitialChecks;
  ComputeConstants;
  if CaseValue = 1 then
     begin
     writeln('LCAS Model Implementation ', Version);
     writeln;
     writeln('Test results');
     writeln('Computer: ');
     writeln('Compiler: ');
     writeln('Options used: ');
     writeln('Program modifications (with reasons): ');
     writeln('Date tested: ');
     writeln('Tested by: ');
     writeln;
     writeln('Parameter values');
     writeln('minint,  maxint');
     writeln(minint, ' ',  maxint);
     writeln(' r,  p,   emin,   emax, denorm');
     writeln(Radix:3, Places:4, ExpoMin:8, ExpoMax:8, ' ', denorm);
     writeln('fmax,      fmin,      fminN');
     writeln(fmax:Digits, fmin:Digits, fminN:Digits)
     end;
  FindRoundingMode;
  FinalChecks;
  if CaseValue = 1 then
     begin
     writeln;
     writeln('Does the output distinguish between the four types of');
     writeln('   notification?');
     writeln;
   writeln('Test  Condition Tested Notify  Result(=yes/post mortem/NO)')
   end;
Notification(CaseValue);

{ Program should not get here! }
writeln('Notification did not occur, test number ', CaseValue);
13:
end.
```