AD NUMBER

ADB120257

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies and their contractors; Critical Technology; JUL 1987. Other requests shall be referred to Air Force Armament Lab., Eglin AFB, FL. This document contains export-controlled technical data.

AUTHORITY

afsc/mnoi's wright lab ltr dtd 13 Feb 1991

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Distribution authorized to U.S. Government Agencies and their contractors; CT (over)
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFATL-TR-88-18, Vol 10
6a. NAME OF PERFORMING ORGANIZATION McDonnell Douglas Astronautics Company 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Aeromechanics Division
6c. ADDRESS (City, State, and ZIP Code) P.O. Box 516 St. Louis, MO 63166	7b. ADDRESS (City, State, and ZIP Code) Air Force Armament Laboratory Eglin AFB, FL 32542-5434
8a. NAME OF FUNDING/SPONSORING ORGANIZATION STARS Joint Program Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F08635-86-C-0025
8c. ADDRESS (City, State, and ZIP Code) Room 3D139 (1211 Fern St) The Pentagon Washington DC 20301-3081	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63756D; PROJECT NO. 921C; TASK NO. GZ; WORK UNIT ACCESSION NO. 57

AD-B120 257

11. TITLE (Include Security Classification)
Common Ada Missile Package (CAMP) Project: Missile Software Parts, Vol 10: Detail Design Documents (Vol 7-12)

12. PERSONAL AUTHOR(S)
D. McNicholl, S. Cohen, C. Palmer, et al.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Note	FROM Sep 85 TO Mar 88	March 1988	316

16. SUPPLEMENTARY NOTATION
SUBJECT TO EXPORT CONTROL LAWS.
Availability of this report is specified on verso of front cover. (over)

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Reusable Software, Missile Software, Software Generators, Ada, Parts Composition Systems, Software Parts

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The objective of the CAMP program is to demonstrate the feasibility of reusable Ada software parts in a real-time embedded application area; the domain chosen for the demonstration was that of missile flight software systems. This required that the existence of commonality within that domain be verified (in order to justify the development of parts for that domain), and that software parts be designed which address those areas identified. An associated parts system was developed to support parts usage. Volume 1 of this document is the User's Guide to the CAMP Software parts; Volume 2 is the Version Description Document; Volume 3 is the Software Product Specification; Volumes 4-6 contain the Top-Level Design Document; and, Volumes 7-12 contain the Detail Design Documents.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Christine Anderson	22b. TELEPHONE (Include Area Code) (904) 882-2961 22c. OFFICE SYMBOL AFATL/FXG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

3. DISTRIBUTION/AVAILABILITY OF REPORT (CONCLUDED)

~~this report documents test and evaluation~~; distribution limitation applied March 1988. Other requests for this document must be referred to AFATL/FXG, Eglin AFB, Florida 32542-5434.

16. SUPPLEMENTARY NOTATION (CONCLUDED)

These technical notes accompany the CAMP final report AFATL-TR-85-93 (3 Vols)

AFATL-TR-88-18, Vol 10

SOFTWARE DETAILED DESIGN DOCUMENT

FOR THE

MISSILE SOFTWARE PARTS

OF THE

COMMON ADA MISSILE PACKAGE (CAMP) PROJECT

CONTRACT F08635-86-C-0025

CDRL SEQUENCE NO. C007

30 OCTOBER 1987

AIR FORCE ARMAMENT LABORATORY
Air Force Systems Command ▮ United States Air Force ▮ Eglin Air Force Base, Florida

3.3.6.3 STANDARD_TRIG (BODY) TLCSC P683 (CATALOG #P2-0)

This package provides a set of standard trigonometric functions. The functions provided are sine, cosine, sine-cosine, tangent, arcsine, arccosine, arcsine-arccosine, and arctangent.

This package body implements the above functions by instantiating the packages contained in the Polynomials.System_Functions package.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.3.1 REQUIREMENTS ALLOCATION

The following table illustrates the allocation of requirements to units in this part:

Name	Type	Requirements Allocation
Sin	function	R086, R092, R098
Cos	function	R087, R093, R099
Sin_Cos	procedure	R086, R087, R092 R093, R098, R099
Tan	function	R088, R094, R100
ArcSin	function	R089, R095, R101
ArcCos	function	R090, R096, R102
ArcSin_ArcCos	procedure	R089, R090, R095 R096, R101, R102
ArcTan	function	R091, R097, R103
ArcTan2	function	N/A

3.3.6.3.2 LOCAL ENTITIES DESIGN

Subprograms:

The following table describes the operators defined by this part:

Name	Left Input Type	Right Input Type	Result Type
"*"	Semicircles	Radians	Radians
"*"	Radians	Radians	Semicircles
"*"	Sin_Cos_Ratio	Sin_Cos_Ratio	Sin_Cos_Ratio_Squared

Packages:

The following table describes the packages contained in this part:

Name	Description
Radian_Opns	Locally instantiated version of Polynomials.System_Functions.Radian_Operations
Semicircle_Opns	Locally instantiated version of Polynomials.System_Functions.Semicircle_Operations
Degree_Opns	Locally instantiated version of Polynomials.System_Functions.Degree_Operations
Radian_Sqrt	Locally instantiated version of Polynomials.System_Functions.Square_Root
Semicircle_Sqrt	Locally instantiated version of Polynomials.System_Functions.Square_Root
Degree_Sqrt	Locally instantiated version of Polynomials.System_Functions.Square_Root

3.3.6.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined in this part's specification:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Angle	floating point type	Used to determine precision of type Radians, Semicircles, and Degrees
Trig_Ratio	floating point type	Used to determine precision of type Sin_Cos_Ratio and Tan_Ratio

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Value	Description
pi_value	Angle	N/A	Value to be used for pi

3.3.6.3.4 LOCAL DATA

Data types:

The following table describes the data types previously defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function
Tan_Ratio	Trig_Ratio	-oo .. +oo	Result of a tangent function function

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Pi	Radians	Pi_Value	Value of pi with a data type of Radians
Pi_Over_2	Radians	Pi/2	Value of pi/2
Two_Pi	Radians	Pi * 2	Value of pi * 2

3.3.6.3.5 PROCESS CONTROL

Not applicable.

3.3.6.3.6 PROCESSING

The following describes the processing performed by this part:

```
with Polynomials;
package body Standard_Trig is

-- ---------------------
-- --local declarations-
-- ---------------------

   type Sin_Cos_Ratio_Squared is new Sin_Cos_Ratio;

   Pi        : constant Radians := Radians(Pi_Value);
   Pi_Over_2 : constant Radians := Pi / 2.0;
   Two_Pi    : constant Radians := Pi * 2.0;

-- -----------------
-- --local operators
-- -----------------

-- --the following operators are required for the instantiation of
-- --the Polynomials.System_Functions.Semicircle_Operations package

   function "*" (Left  : Semicircles;
                 Right : Radians) return Radians;
```

```
   function "*" (Left  : Radians;
                 Right : Radians) return Semicircles;

-- --the following operators are required for the Sin_Cos procedures

   function "*" (Left  : Sin_Cos_Ratio;
                 Right : Sin_Cos_Ratio) return Sin_Cos_Ratio_Squared;

-- ------------------------------------------------------------------
-- --required instantiations of Polynomial.System_Functions packages
-- ------------------------------------------------------------------

   package SysFns renames Polynomials.System_Functions;

-- --radian functions

   package Radian_Opns is new
           SysFns.Radian_Operations (Radians       => Radians,
                                     Sin_Cos_Ratio => Sin_Cos_Ratio,
                                     Tan_Ratio     => Tan_Ratio);

   function Sin_R    (Input : Radians)
                     return Sin_Cos_Ratio renames Radian_Opns.Sin;
   function Cos_R    (Input : Radians)
                     return Sin_Cos_Ratio renames Radian_Opns.Cos;
   function Tan_R    (Input : Radians)
                     return Tan_Ratio     renames Radian_Opns.Tan;
   function Arcsin_R (Input : Sin_Cos_Ratio)
                     return Radians       renames Radian_Opns.Arcsin;
   function Arccos_R (Input : Sin_Cos_Ratio)
                     return Radians       renames Radian_Opns.Arccos;
   function Arctan_R (Input : Tan_Ratio)
                     return Radians       renames Radian_Opns.Arctan;


-- --semicircle functions

   package Semicircle_Opns is new
           SysFns.Semicircle_Operations (Scalars => Radians,
                                         Semicircles => Semicircles,
                                         Sin_Cos_Ratio => Sin_Cos_Ratio,
                                         Tan_Ratio     => Tan_Ratio,
                                         pi            => pi);

   function Sin_S    (Input   Semicircles)
                     return Sin_Cos_Ratio renames Semicircle_Opns.Sin;
   function Cos_S    (Input : Semicircles)
                     return Sin_Cos_Ratio renames Semicircle_Opns.Cos;
   function Tan_S    (Input : Semicircles)
                     return Tan_Ratio     renames Semicircle_Opns.Tan;
   function Arcsin_S (Input : Sin_Cos_Ratio)
                     return Semicircles   renames Semicircle_Opns.Arcsin;
   function Arccos_S (Input : Sin_Cos_Ratio)
                     return Semicircles   renames Semicircle_Opns.Arccos;
   function Arctan_S (Input : Tan_Ratio)
                     return Semicircles   renames Semicircle_Opns.Arctan;
```

```
-- --degree functions

   package Degree_Opns is new
           SysFns.Degree_Operations (Degrees       => Degrees,
                                     Sin_Cos_Ratio => Sin_Cos_Ratio,
                                     Tan_Ratio     => Tan_Ratio);

   function Sin_D    (Input : Degrees)
                     return Sin_Cos_Ratio renames Degree_Opns.Sin;
   function Cos_D    (Input : Degrees)
                     return Sin_Cos_Ratio renames Degree_Opns.Cos;
   function Tan_D    (Input : Degrees)
                     return Tan_Ratio     renames Degree_Opns.Tan;
   function Arcsin_D (Input : Sin_Cos_Ratio)
                     return Degrees       renames Degree_Opns.Arcsin;
   function Arccos_D (Input : Sin_Cos_Ratio)
                     return Degrees       renames Degree_Opns.Arccos;
   function Arctan_D (Input : Tan_Ratio)
                     return Degrees       renames Degree_Opns.Arctan;

-- --square root function

   package Square_Root is new
           SysFns.Square_Root (Inputs  => Sin_Cos_Ratio_Squared,
                               Outputs => Sin_Cos_Ratio);

   function Sqrt (Input : Sin_Cos_Ratio_Squared)
                  return Sin_Cos_Ratio renames Square_Root.Sqrt;


-- ---------------------------
-- --bodies of local operators
-- ---------------------------

   function "*" (Left  : Semicircles;
                 Right : Radians) return Radians is
   begin
      return Radians(Left) * Right;
   end "*";

   function "*" (Left  : Radians;
                 Right : Radians) return Semicircles is
   begin
      return Semicircles(Left) * Semicircles(Right);
   end "*";

   function "*" (Left  : Sin_Cos_Ratio;
                 Right : Sin_Cos_Ratio) return Sin_Cos_Ratio_Squared is
   begin
      return Sin_Cos_Ratio_Squared(Left) * Sin_Cos_Ratio_Squared(Right);
   end "*";

end Standard_Trig;
```

3.3.6.3.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:
1. Polynomials

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms:

The following table summarizes the external subroutines required by this part and located in the Polynomials.System_Functions. Radian_Operations package:

Name	Type	Description
Sin	function	Sine function
Cos	function	Cosine function
Tan	function	Tangent function
Arcsin	function	Arcsine function
Arccos	function	Arccosine function
Arctan	function	Arctangent function

The following table summarizes the external subroutines required by this part and located in the Polynomials.System_Functions. Semicircle_Operations package:

Name	Type	Description
Sin	function	Sine function
Cos	function	Cosine function
Tan	function	Tangent function
Arcsin	function	Arcsine function
Arccos	function	Arccosine function
Arctan	function	Arctangent function

The following table summarizes the external subroutines required by this part and located in the Polynomials.System_Functions. Degree_Operations package:

Name	Type	Description
Sin	function	Sine function
Cos	function	Cosine function
Tan	function	Tangent function
Arcsin	function	Arcsine function
Arccos	function	Arccosine function
Arctan	function	Arctangent function

The following table summarizes the subroutines required by this part and located in the Polynomials.System_Functions.Square_Root package:

Name	Type	Description
Sqrt	function	Square root function

3.3.6.3.8 LIMITATIONS

None.

3.3.6.3.9 LLCSC DESIGN

None.

3.3.6.3.10 UNIT DESIGN

3.3.6.3.10.1 SIN (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P3-0 {RADIANS},P538-0 {SEMICIRCLES},P539-0 {DEGREES})

This set of functions calculate the sine of an input angle. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.1.1 REQUIREMENTS ALLOCATION

These functions meet CAMP requirements R086, R092, and R098.

3.3.6.3.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Radians	N/A	Value for which a sine is desired
Input	Semicircles	N/A	Value for which a sine is desired
Input	Degrees	N/A	Value for which a sine is desired

3.3.6.3.10.1.4 LOCAL DATA

None.

3.3.6.3.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
function Sin (Input : Radians) return Sin_Cos_Ratio is
begin
   return Sin_R(Input);
end Sin;

function Sin (Input : Semicircles) return Sin_Cos_Ratio is
begin
   return Sin_S(Input);
end Sin;

function Sin (Input : Degrees) return Sin_Cos_Ratio is
begin
   return Sin_D(Input);
end Sin;
```

3.3.6.3.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Sin_R	function	Sine function handling units of radians
Sin_S	function	Sine function handling units of semicircles
Sin_D	function	Sine function handling units of degrees

3.3.6.3.10.1.8 LIMITATIONS

None.

3.3.6.3.10.2 COS (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P4-0 (RADIANS),P540-0 (SEMICIRCLES),P541-0 (DEGREES))

This set of functions calculate the cosine of an input angle. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.2.1 REQUIREMENTS ALLOCATION

These functions meet CAMP requirements R087, R093, and R099.

3.3.6.3.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Radians	N/A	Value for which a cosine is desired
Input	Semicircles	N/A	Value for which a cosine is desired
Input	Degrees	N/A	Value for which a cosine is desired

3.3.6.3.10.2.4 LOCAL DATA

None.

3.3.6.3.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Cos (Input : Radians) return Sin_Cos_Ratio is
begin
   return Cos_R(Input);
end Cos;

function Cos (Input : Semicircles) return Sin_Cos_Ratio is
begin
   return Cos_S(Input);
end Cos;

function Cos (Input : Degrees) return Sin_Cos_Ratio is
begin
   return Cos_D(Input);
end Cos;
```

3.3.6.3.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Cos_R	function	Cosine function handling units of radians
Cos_S	function	Cosine function handling units of semicircles
Cos_D	function	Cosine function handling units of degrees

3.3.6.3.10.2.8 LIMITATIONS

None.

3.3.6.3.10.3 SIN_COS (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P5-0 {RADIANS},P542-0 {SEMICIRCLES},P543-0 {DEGREES})

This set of procedures calculate the sine and cosine of an input angle. Three procedures have been set up to handle angles with units of radians, semicircles, and degrees.

The sine values will be obtained by calling the appropriate Sin function contained in one of the instantiated packages from Polynomials.System_Functions. The cosine values will then be calculated based on the sine values obtained.

The cosine values will be calculated as follows:

Cos(x) := +/- Sqrt(1.0 - sin**2(x))

where the sign of the cosine depends upon the quadrant in which the angle lies:

```
II  |   I
  - | +
---------
  - | +
III |  IV
```

3.3.6.3.10.3.1 REQUIREMENTS ALLOCATION

These procedures meet CAMP requirements R086, R087, R092, R093, R098, and R099.

3.3.6.3.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Radians	N/A	Value for which a sine and cosine are desired
Input	Semicircles	N/A	Value for which a sine and cosine are desired
Input	Degrees	N/A	Value for which a sine and cosine are desired

3.3.6.3.10.3.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Reduced_Angle	Radians	N/A	Input angle after its range has been reduced to +/- pi
Reduced_Angle	Semicircles	N/A	Input angle after its range has been reduced to 0 .. 2.0
Reduced_Angle	Degrees	N/A	Input angle after its range has been reduced to 0 .. 360.0
Temp_Cos	Sin_Cos_Ratio	N/A	Temporary cosine value
Temp_Sin	Sin_Cos_Ratio	N/A	Temporary sine value

3.3.6.3.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Sin_Cos (Input      : in  Radians;
                      Sin_Result : out Sin_Cos_Ratio;
                      Cos_Result : out Sin_Cos_Ratio) is

--    ---------------------
--    --declaration section
--    ---------------------

      Reduced_Angle : Radians;
      Temp_Cos      : Sin_Cos_Ratio;
      Temp_Sin      : Sin_Cos_Ratio;

-- -------------------------
-- --begin procedure Sin_Cos
-- -------------------------

   begin

--    ----------get sine value----------

      Temp_Sin   := Sin_R(Input);
      Sin_Result := Temp_Sin;

--    ----------reduce input angle to +/- pi----------

      Reduced_Angle := Input;
```

```
      if ABS(Reduced_Angle) > Pi then

--       --handle positive angles

         if Reduced_Angle > Pi then
            Reduce_Input_Angle_Through_Subtraction:
               while Reduced_Angle > Pi loop
                  Reduced_Angle := Reduced_Angle - Two_Pi;
               end loop Reduce_Input_Angle_Through_Subtraction;
         end if;

--       --handle negative angles

         if Reduced_Angle < -Pi then
            Reduce_Input_Angle_Through_Addition:
               while Reduced_Angle < -Pi loop
                  Reduced_Angle := Reduced_Angle + Two_Pi;
               end loop Reduce_Input_Angle_Through_Addition;
         end if;

      end if;

--    ----------calculate cosine value----------

      Temp_Cos := Sqrt(1.0 - Temp_Sin*Temp_Sin);

--    --check for angles in Quadrant II or III since they would be negative
      if ABS(Reduced_Angle) > Pi_Over_2 then
         Cos_Result := - Temp_Cos;
      else
         Cos_Result := Temp_Cos;
      end if;

   end Sin_Cos;


   procedure Sin_Cos (Input      : in  Semicircles;
                      Sin_Result : out Sin_Cos_Ratio;
                      Cos_Result : out Sin_Cos_Ratio) is

--    --------------------
--    --declaration section
--    --------------------

      Reduced_Angle : Semicircles;
      Temp_Cos      : Sin_Cos_Ratio;
      Temp_Sin      : Sin_Cos_Ratio;

-- ------------------------
-- --begin function Sin_Cos
-- ------------------------

   begin

--    ----------calculate sine value----------

      Temp_Sin   := Sin_S(Input);
```

```
      Sin_Result := Temp_Sin;

--    ----------reduce input angle to 0..2----------

      Reduced_Angle := Input;

      Put_Negative_Angles_In_Correct_Range:
         while Reduced_Angle < 0.0 loop
            Reduced_Angle := Reduced_Angle + 2.0;
         end loop Put_Negative_Angles_In_Correct_Range;

      Put_Positive_Angles_In_Correct_Range:
         while Reduced_Angle > 2.0 loop
            Reduced_Angle := Reduced_Angle - 2.0;
         end loop Put_Positive_Angles_In_Correct_Range;

--    ----------calculate cosine value----------

      Temp_Cos := Sqrt(1.0 - Temp_Sin*Temp_Sin);

--    --see if angle is in quadrants II or III since these cosines should
--    --be negative
      if Reduced_Angle > 0.5 and then
         Reduced_Angle < 1.5 then
         Cos_Result := - Temp_Cos;
      else
         Cos_Result := Temp_Cos;
      end if;

   end Sin_Cos;


   procedure Sin_Cos (Input      : in  Degrees;
                      Sin_Result : out Sin_Cos_Ratio;
                      Cos_Result : out Sin_Cos_Ratio) is

--    ---------------------
--    --declaration section
--    ---------------------

      Reduced_Angle : Degrees;
      Temp_Cos      : Sin_Cos_Ratio;
      Temp_Sin      : Sin_Cos_Ratio;

-- -------------------------
-- --begin procedure Sin_Cos
-- -------------------------

   begin

--    ----------calculate sine value----------

      Temp_Sin := Sin_D(Input);
      Sin_Result := Temp_Sin;

--    ----------reduce input angle to 0 .. 360----------
```

```
      Reduced_Angle := Input;

      Reduce_Positive_Angles_To_Appropriate_Range:
         while Reduced_Angle > 360.0 loop
            Reduced_Angle := Reduced_Angle - 360.0;
         end loop Reduce_Positive_Angles_To_Appropriate_Range;

      Reduce_Negative_Angles_to_Appropriate_Range:
         while Reduced_Angle < 0.0 loop
            Reduced_Angle := Reduced_Angle + 360.0;
         end loop Reduce_Negative_Angles_to_Appropriate_Range;

--    ----------calculate cosine value

      Temp_Cos   := Sqrt(1.0 - Temp_Sin*Temp_Sin);

--    --check for angles in quadrants II or III since these values need to
--    --be negative
      if Reduced_Angle > 90.0 and then
         Reduced_Angle < 270.0 then
         Cos_Result := - Temp_Cos;
      else
         Cos_Result := Temp_Cos;
      end if;

   end Sin_Cos;
```

3.3.6.3.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Sin_R	function	Sine function handling units of radians
Sin_S	function	Sine function handling units of semicircles
Sin_D	function	Sine function handling units of degrees

3.3.6.3.10.3.8 LIMITATIONS

None.

3.3.6.3.10.4 TAN (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P6-0 {RADIANS},P544-0 {SEMICIRCLES},P545-0 {DEGREES})

This set of functions calculates the tangent of an input angle. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.4.1 REQUIREMENTS ALLOCATION

These functions meet CAMP requirements R088, R094, and R100.

3.3.6.3.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Radians	N/A	Value for which a tangent is desired
Input	Semicircles	N/A	Value for which a tangent is desired
Input	Degrees	N/A	Value for which a tangent is desired

3.3.6.3.10.4.4 LOCAL DATA

None.

3.3.6.3.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
function Tan (Input : Radians) return Tan_Ratio is
begin
   return Tan_R(Input);
end Tan;

function Tan (Input : Semicircles) return Tan_Ratio is
begin
   return Tan_S(Input);
end Tan;

function Tan (Input : Degrees) return Tan_Ratio is
begin
   return Tan_D(Input);
end Tan;
```

3.3.6.3.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Tan_Ratio	Trig_Ratio	-oo .. +oo	Result of a tangent function function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Tan_R	function	Tangent function handling units of radians
Tan_S	function	Tangent function handling units of semicircles
Tan_D	function	Tangent function handling units of degrees

3.3.6.3.10.4.8 LIMITATIONS

None.

3.3.6.3.10.5 ARCSIN (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P7-0 {RADIANS},P546-0 {SEMICIRCLES},P547-0 {DEGREES})

This set of functions calculates the arcsine of an input value. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.5.1 REQUIREMENTS ALLOCATION

These functions meet CAMP requirements R089, R095, and R101.

3.3.6.3.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Sin_Cos_Ratio	N/A	Value for which an arcsine is desired

3.3.6.3.10.5.4 LOCAL DATA

None.

3.3.6.3.10.5.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
function Arcsin (Input : Sin_Cos_Ratio) return Radians is
begin
   return Arcsin_R(Input);
end Arcsin;

function Arcsin (Input : Sin_Cos_Ratio) return Semicircles is
begin
```

```
      return Arcsin_S(Input);
   end Arcsin;

   function Arcsin (Input : Sin_Cos_Ratio) return Degrees is
   begin
      return Arcsin_D(Input);
   end Arcsin;
```

3.3.6.3.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Arcsin_R	function	Arcsine function handling units of radians
Arcsin_S	function	Arcsine function handling units of semicircles
Arcsin_D	function	Arcsine function handling units of degrees

3.3.6.3.10.5.8 LIMITATIONS

None.

3.3.6.3.10.6 ARCCOS (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P8-0 {RADIANS},P548-0 {SEMICIRCLES},P549-0 {DEGREES})

This set of functions calculates the arccosine of an input value. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.6.1 REQUIREMENTS ALLOCATION

These parts meet CAMP requirements R090, R096, and R102.

3.3.6.3.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Sin_Cos_Ratio	N/A	Value for which an arccosine is desired

3.3.6.3.10.6.4 LOCAL DATA

None.

3.3.6.3.10.6.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
function Arccos (Input : Sin_Cos_Ratio) return Radians is
begin
   return Arccos_R(Input);
end Arccos;

function Arccos (Input : Sin_Cos_Ratio) return Semicircles is
begin
   return Arccos_S(Input);
end Arccos;

function Arccos (Input : Sin_Cos_Ratio) return Degrees is
begin
   return Arccos_D(Input);
end Arccos;
```

3.3.6.3.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Arccos_R	function	Arccosine function handling units of radians
Arccos_S	function	Arccosine function handling units of semicircles
Arccos_D	function	Arccosine function handling units of degrees

3.3.6.3.10.6.8 LIMITATIONS

None.

3.3.6.3.10.7 ARCSIN ARCCOS (FOR RADIANS, SEMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P9-0 {RADIANS},P550-0 {SEMICIRCLES},P551-0 {DEGREES})

This set of procedures calculates the arcsine and arccosine of an input value. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.7.1 REQUIREMENTS ALLOCATION

These procedures meet CAMP requirements R089, R090, R095, R096, R101, and R102.

3.3.6.3.10.7.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.7.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Sin_Cos_Ratio	N/A	Value for which an arcsine and arccosine is desired

3.3.6.3.10.7.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by these parts:

Name	Type	Value	Description
Temp_Asin	Radians	N/A	Temporary arcsine value
Temp_Asin	Semicircles	N/A	Temporary arcsine value
Temp_Asin	Degrees	N/A	Temporary arcsine value

3.3.6.3.10.7.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.7.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Arcsin_Arccos (Input         : in  Sin_Cos_Ratio;
                            Arcsin_Result : out Radians;
                            Arccos_Result : out Radians) is

--    ---------------------
--    --declaration section
--    ---------------------

      Temp_Asin : Radians;

-- ------------------------------
-- --begin procedure Arcsin_Arccos
-- ------------------------------
```

```
   begin
      Temp_Asin     := Arcsin_R(Input);
      Arcsin_Result := Temp_ASin;
      Arccos_Result := Pi_Over_2 - Temp_ASin;
   end Arcsin_Arccos;


   procedure Arcsin_Arccos (Input         : in  Sin_Cos_Ratio;
                            Arcsin_Result : out Semicircles;
                            Arccos_Result : out Semicircles) is

--    ---------------------
--    --declaration section
--    ---------------------

      Temp_Asin : Semicircles;

-- -------------------------------
-- --begin procedure Arcsin_Arccos
-- -------------------------------

   begin
      Temp_Asin     := Arcsin_S(Input);
      Arcsin_Result := Temp_Asin;
      Arccos_Result := 0.5 - Temp_Asin;
   end Arcsin_Arccos;


   procedure Arcsin_Arccos (Input         : in  Sin_Cos_Ratio;
                            Arcsin_Result : out Degrees;
                            Arccos_Result : out Degrees) is

--    ---------------------
--    --declaration section
--    ---------------------

      Temp_Asin : Degrees;

-- -------------------------------
-- --begin procedure Arcsin_Arccos
-- -------------------------------

   begin
      Temp_Asin     := Arcsin_D(Input);
      Arcsin_Result := Temp_Asin;
      Arccos_Result := 90.0 - Temp_Asin;
   end Arcsin_Arccos;
```

3.3.6.3.10.7.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Sin_Cos_Ratio	Trig_Ratio	-1 .. +1	Result of a sine or cosine function

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Standard_Trig:

Name	Type	Value	Description
Pi_Over_2	Radians	Pi/2	Value of pi/2

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Arcsin_R	function	Arcsine function handling units of radians
Arcsin_S	function	Arcsine function handling units of semicircles
Arcsin_D	function	Arcsine function handling units of degrees

3.3.6.3.10.7.8 LIMITATIONS

None.

3.3.6.3.10.8 ARCTAN (FOR RADIANS, EMICIRCLES, AND DEGREES) UNIT DESIGN (CATALOG #P10-0 {RADIANS},P552-0 {SEMICIRCLES},P553-0 {DEGREES})

This set of functions calculates the arctangent of an input value. Three functions have been set up to handle angles with units of radians, semicircles, and degrees.

3.3.6.3.10.8.1 REQUIREMENTS ALLOCATION

The functions meet CAMP requirements R091, R097, and R103.

3.3.6.3.10.8.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Value	Description
Input	Tan_Ratio	N/A	Value for which an arctangent is desired

3.3.6.3.10.8.4 LOCAL DATA

None.

3.3.6.3.10.8.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.8.6 PROCESSING

The following describes the processing performed by this part:

```
function Arctan (Input : Tan_Ratio) return Radians is
begin
   return Arctan_R(Input);
end Arctan;

function Arctan (Input : Tan_Ratio) return Semicircles is
begin
   return Arctan_S(Input);
end Arctan;

function Arctan (Input : Tan_Ratio) return Degrees is
begin
   return Arctan_D(Input);
end Arctan;
```

3.3.6.3.10.8.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF ANCESTRAL ELEMENTS:

The following tables describe the elements used by this part but defined in one or more ancestral units:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Standard_Trig:

Name	Base Type	Range	Description
Radians	Angle	-oo .. +oo	Radian unit of measurement
Semicircles	Angle	-oo .. +oo	Semicircle unit of measurement
Degrees	Angle	-oo .. +oo	Degree unit of measurement
Tan_Ratio	Trig_Ratio	-oo .. +oo	Result of a tangent function function

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Standard_Trig:

Name	Type	Description
Arctan_R	function	Arctangent function handling units of radians
Arctan_S	function	Arctangent function handling units of semicircles
Arctan_D	function	Arctangent function handling units of degrees

3.3.6.3.10.8.8 LIMITATIONS

None.

3.3.6.3.10.9 ARCTAN2 (FUNCTION BODY) UNIT DESIGN (CATALOG #P554-0)

This function calculates the arctangent of two input values defining the endpoint of a vector. The result of this function is the angle between the vector and the positive x-axis and is in the range equivalent to +/- pi.

If both X and Y equal 0, this function will return a value of 0.

3.3.6.3.10.9.1 REQUIREMENTS ALLOCATION

This part meets requirement R.

3.3.6.3.10.9.2 LOCAL ENTITIES DESIGN

None.

3.3.6.3.10.9.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this function was specified:

Data types:

The following table describes the generic formal types required by this part:

Name	Type	Description
Angles	floating point	Data type defining angular measurements
Measurements	floating point	Data type defining function input types

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Description
Cycle_over_2	Angles	Number of angular units of measurement in half a circle (e.g., pi/2 for Radians, 180 for Degrees, and 1.0 for Semicircles)
Cycle_over_4	Angles	Number of angular units of measurement in 1/4 of a circle (e.g., pi/4 for Radians, 90 for degrees, and .5 for Semicircles)

Subprograms:

The following table describes the generic formal subroutines required by this part:

Name	Type	Description
"/"	function	Division operator defining the operation: Measurements / Measurements => Tan_Ratio
Arctan	function	Arctangent function

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
X	Measurements	in	First element of the coordinate pair
Y	Measurements	in	Second element of the coordinate pair

3.3.6.3.10.9.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Description
Answer	Angles	Arctangent-2 of input values

3.3.6.3.10.9.5 PROCESS CONTROL

Not applicable.

3.3.6.3.10.9.6 PROCESSING

The following describes the processing performed by this part:

```
   function Arctan2 (X : Measurements;
                     Y : Measurements) return Angles is

--    ---------------------
--    --declaration section
--    ---------------------

      Answer    : Angles;
      Tan_Value : Tan_Ratio;

-- ---------------------
-- --begin function body
-- ---------------------

   begin

      if X = 0.0 then

         if Y = 0.0 then
            Answer := 0.0;
         elsif Y > 0.0 then
            Answer := Cycle_over_4;
         else
            Answer := -Cycle_over_4;
         end if;

      else
```

```
         if abs(X) >= abs(Y) then

            Tan_Value := Y / X;
            Answer := Arctan (Tan_Value);

            if X < 0.0 then
               if Y >= 0.0 then
                  Answer := Answer + Cycle_over_2;
               else
                  Answer := Answer - Cycle_over_2;
               end if;
            end if;

         else  --  abs(X) < abs(Y)

            Tan_Value := X / Y;
            Answer := Arctan (Tan_Value);

            if Y >= 0.0 then
               Answer :=  Cycle_over_4 - Answer;
            else
               Answer := -Cycle_over_4 - Answer;
            end if;

         end if;

      end if;

      return Answer;

   end Arctan2;
```

3.3.6.3.10.9.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.3.10.9.8 LIMITATIONS

None.

(This page left intentionally blank.)

```
with Polynomials;
package body Standard_Trig is

-- ---------------------
-- --local declarations-
-- ---------------------

   type Sin_Cos_Ratio_Squared is new Sin_Cos_Ratio;

   Pi        : constant Radians := Radians(Pi_Value);
   Pi_Over_2 : constant Radians := Pi / 2.0;
   Two_Pi    : constant Radians := Pi * 2.0;

-- -----------------
-- --local operators
-- -----------------

-- --the following operators are required for the instantiation of
-- --the Polynomials.System_Functions.Semicircle_Operations package

   function "*" (Left  : Semicircles;
                 Right : Radians) return Radians;

   function "*" (Left  : Radians;
                 Right : Radians) return Semicircles;

-- --the following operators are required for the Sin_Cos procedures

   function "*" (Left  : Sin_Cos_Ratio;
                 Right : Sin_Cos_Ratio) return Sin_Cos_Ratio_Squared;

-- ------------------------------------------------------------------
-- --required instantiations of Polynomial.System_Functions packages
-- ------------------------------------------------------------------

   package Sysfns renames Polynomials.System_Functions;

-- --radian functions

   package Radian_Opns is new
           Sysfns.Radian_Operations (Radians       => Radians,
                                     Sin_Cos_Ratio => Sin_Cos_Ratio,
                                     Tan_Ratio     => Tan_Ratio);

   function Sin_R    (Input : Radians)
                      return Sin_Cos_Ratio renames Radian_Opns.Sin;
   function Cos_R    (Input : Radians)
                      return Sin_Cos_Ratio renames Radian_Opns.Cos;
   function Tan_R    (Input : Radians)
                      return Tan_Ratio     renames Radian_Opns.Tan;
   function Arcsin_R (Input : Sin_Cos_Ratio)
                      return Radians       renames Radian_Opns.Arcsin;
   function Arccos_R (Input : Sin_Cos_Ratio)
                      return Radians       renames Radian_Opns.Arccos;
   function Arctan_R (Input : Tan_Ratio)
                      return Radians       renames Radian_Opns.Arctan;
```

```
pragma PAGE;
-- --semicircle functions

   package Semicircle_Opns is new
           Sysfns.Semicircle_Operations (Scalars => Radians,
                                         Semicircles => Semicircles,
                                         Sin_Cos_Ratio => Sin_Cos_Ratio,
                                         Tan_Ratio     => Tan_Ratio,
                                         Pi            => Pi);

   function Sin_S    (Input : Semicircles)
                     return Sin_Cos_Ratio renames Semicircle_Opns.Sin;
   function Cos_S    (Input : Semicircles)
                     return Sin_Cos_Ratio renames Semicircle_Opns.Cos;
   function Tan_S    (Input : Semicircles)
                     return Tan_Ratio     renames Semicircle_Opns.Tan;
   function Arcsin_S (Input : Sin_Cos_Ratio)
                     return Semicircles   renames Semicircle_Opns.Arcsin;
   function Arccos_S (Input : Sin_Cos_Ratio)
                     return Semicircles   renames Semicircle_Opns.Arccos;
   function Arctan_S (Input : Tan_Ratio)
                     return Semicircles   renames Semicircle_Opns.Arctan;

-- --degree functions

   package Degree_Opns is new
           Sysfns.Degree_Operations (Degrees       => Degrees,
                                     Sin_Cos_Ratio => Sin_Cos_Ratio,
                                     Tan_Ratio     => Tan_Ratio);

   function Sin_D    (Input : Degrees)
                     return Sin_Cos_Ratio renames Degree_Opns.Sin;
   function Cos_D    (Input : Degrees)
                     return Sin_Cos_Ratio renames Degree_Opns.Cos;
   function Tan_D    (Input : Degrees)
                     return Tan_Ratio     renames Degree_Opns.Tan;
   function Arcsin_D (Input : Sin_Cos_Ratio)
                     return Degrees       renames Degree_Opns.Arcsin;
   function Arccos_D (Input : Sin_Cos_Ratio)
                     return Degrees       renames Degree_Opns.Arccos;
   function Arctan_D (Input : Tan_Ratio)
                     return Degrees       renames Degree_Opns.Arctan;

-- --square root function

   package Square_Root is new
           Sysfns.Square_Root (Inputs  => Sin_Cos_Ratio_Squared,
                               Outputs => Sin_Cos_Ratio);

   function Sqrt (Input : Sin_Cos_Ratio_Squared)
                  return Sin_Cos_Ratio renames Square_Root.Sqrt;

pragma PAGE;
-- --------------------------
-- --bodies of local operators
-- --------------------------
```

```
   function "*" (Left  : Semicircles;
                 Right : Radians) return Radians is
   begin
      return Radians(Left) * Right;
   end "*";

   function "*" (Left  : Radians;
                 Right : Radians) return Semicircles is
   begin
      return Semicircles(Left) * Semicircles(Right);
   end "*";

   function "*" (Left  : Sin_Cos_Ratio;
                 Right : Sin_Cos_Ratio) return Sin_Cos_Ratio_Squared is
   begin
      return Sin_Cos_Ratio_Squared(Left) * Sin_Cos_Ratio_Squared(Right);
   end "*";

pragma PAGE;
   function Sin (Input : Radians) return Sin_Cos_Ratio is
   begin
      return Sin_R(Input);
   end Sin;

   function Sin (Input : Semicircles) return Sin_Cos_Ratio is
   begin
      return Sin_S(Input);
   end Sin;

   function Sin (Input : Degrees) return Sin_Cos_Ratio is
   begin
      return Sin_D(Input);
   end Sin;

pragma PAGE;
   function Cos (Input : Radians) return Sin_Cos_Ratio is
   begin
      return Cos_R(Input);
   end Cos;

   function Cos (Input : Semicircles) return Sin_Cos_Ratio is
   begin
      return Cos_S(Input);
   end Cos;

   function Cos (Input : Degrees) return Sin_Cos_Ratio is
   begin
      return Cos_D(Input);
   end Cos;

pragma PAGE;
   procedure Sin_Cos (Input      : in  Radians;
                      Sin_Result : out Sin_Cos_Ratio;
                      Cos_Result : out Sin_Cos_Ratio) is

--    ---------------------
--    --declaration section
```

```
--     ---------------------

       Reduced_Angle : Radians;
       Temp_Cos      : Sin_Cos_Ratio;
       Temp_Sin      : Sin_Cos_Ratio;

-- -------------------------
-- --begin procedure Sin_Cos
-- -------------------------

   begin

--     ----------get sine value----------

       Temp_Sin   := Sin_R(Input);
       Sin_Result := Temp_Sin;

--     ----------reduce input angle to +/- pi----------

       Reduced_Angle := Input;
       if abs(Reduced_Angle) > Pi then

--        --handle positive angles

          if Reduced_Angle > Pi then
             Reduce_Input_Angle_Through_Subtraction:
                while Reduced_Angle > PI loop
                   Reduced_Angle := Reduced_Angle - Two_Pi;
                end loop Reduce_Input_Angle_Through_Subtraction;
          end if;

--        --handle negative angles

          if Reduced_Angle < -PI then
             Reduce_Input_Angle_Through_Addition:
                while Reduced_Angle < -PI loop
                   Reduced_Angle := Reduced_Angle + Two_Pi;
                end loop Reduce_Input_Angle_Through_Addition;
          end if;

       end if;

--     ----------calculate cosine value----------

       Temp_Cos := Sqrt(1.0 - Temp_Sin*Temp_Sin);

--     --check for angles in Quadrant II or III since they would be negative
       if abs(Reduced_Angle) > Pi_Over_2 then
          Cos_Result := - Temp_Cos;
       else
          Cos_Result := Temp_Cos;
       end if;

   end Sin_Cos;

pragma PAGE;
   procedure Sin_Cos (Input      : in  Semicircles;
```

```
                        Sin_Result : out Sin_Cos_Ratio;
                        Cos_Result : out Sin_Cos_Ratio) is

--      ---------------------
--      --declaration section
--      ---------------------

        Reduced_Angle : Semicircles;
        Temp_Cos      : Sin_Cos_Ratio;
        Temp_Sin      : Sin_Cos_Ratio;

-- ------------------------
-- --begin function Sin_Cos
-- ------------------------

    begin

--      ----------calculate sine value----------

        Temp_Sin   := Sin_S(Input);
        Sin_Result := Temp_Sin;

--      ----------reduce input angle to 0..2---------

        Reduced_Angle := Input;

        Put_Negative_Angles_In_Correct_Range:
           while Reduced_Angle < 0.0 loop
              Reduced_Angle := Reduced_Angle + 2.0;
           end loop Put_Negative_Angles_In_Correct_Range;

        Put_Positive_Angles_In_Correct_Range:
           while Reduced_Angle > 2.0 loop
              Reduced_Angle := Reduced_Angle - 2.0;
           end loop Put_Positive_Angles_In_Correct_Range;

--      ----------calculate cosine value----------

        Temp_Cos := Sqrt(1.0 - Temp_Sin*Temp_Sin);

--      --see if angle is in quadrants II or III since these cosines should
--      --be negative
        if Reduced_Angle > 0.5 and then
           Reduced_Angle < 1.5 then
           Cos_Result := - Temp_Cos;
        else
           Cos_Result := Temp_Cos;
        end if;

    end Sin_Cos;

pragma PAGE;
    procedure Sin_Cos (Input      : in  Degrees;
                       Sin_Result : out Sin_Cos_Ratio;
                       Cos_Result : out Sin_Cos_Ratio) is

--      ---------------------
```

```
--      --declaration section
--      ---------------------

        Reduced_Angle : Degrees;
        Temp_Cos      : Sin_Cos_Ratio;
        Temp_Sin      : Sin_Cos_Ratio;

-- -------------------------
-- --begin procedure Sin_Cos
-- -------------------------

   begin

--      ----------calculate sine value----------

        Temp_Sin := Sin_D(Input);
        Sin_Result := Temp_Sin;

--      ----------reduce input angle to 0 .. 360----------

        Reduced_Angle := Input;

        Reduce_Positive_Angles_To_Appropriate_Range:
           while Reduced_Angle > 360.0 loop
              Reduced_Angle := Reduced_Angle - 360.0;
           end loop Reduce_Positive_Angles_To_Appropriate_Range;

        Reduce_Negative_Angles_To_Appropriate_Range:
           while Reduced_Angle < 0.0 loop
              Reduced_Angle := Reduced_Angle + 360.0;
           end loop Reduce_Negative_Angles_To_Appropriate_Range;

--      ----------calculate cosine value

        Temp_Cos    := Sqrt(1.0 - Temp_Sin*Temp_Sin);

--      --check for angles in quadrants II or III since these values need to
--      --be negative
        if Reduced_Angle > 90.0 and then
           Reduced_Angle < 270.0 then
           Cos_Result := - Temp_Cos;
        else
           Cos_Result := Temp_Cos;
        end if;

   end Sin_Cos;

pragma PAGE;
   function Tan (Input : Radians) return Tan_Ratio is
   begin
      return Tan_R(Input);
   end Tan;

   function Tan (Input : Semicircles) return Tan_Ratio is
   begin
      return Tan_S(Input);
   end Tan;
```

```
   function Tan (Input : Degrees) return Tan_Ratio is
   begin
      return Tan_D(Input);
   end Tan;

pragma PAGE;
   function Arcsin (Input : Sin_Cos_Ratio) return Radians is
   begin
      return Arcsin_R(Input);
   end Arcsin;

   function Arcsin (Input : Sin_Cos_Ratio) return Semicircles is
   begin
      return Arcsin_S(Input);
   end Arcsin;

   function Arcsin (Input : Sin_Cos_Ratio) return Degrees is
   begin
      return Arcsin_D(Input);
   end Arcsin;

pragma PAGE;
   function Arccos (Input : Sin_Cos_Ratio) return Radians is
   begin
      return Arccos_R(Input);
   end Arccos;

   function Arccos (Input : Sin_Cos_Ratio) return Semicircles is
   begin
      return Arccos_S(Input);
   end Arccos;

   function Arccos (Input : Sin_Cos_Ratio) return Degrees is
   begin
      return Arccos_D(Input);
   end Arccos;

pragma PAGE;
   procedure Arcsin_Arccos (Input         : in  Sin_Cos_Ratio;
                            Arcsin_Result : out Radians;
                            Arccos_Result : out Radians) is

--    ---------------------
--    --declaration section
--    ---------------------

      Temp_Asin : Radians;

-- -------------------------------
-- --begin procedure Arcsin_Arccos
-- -------------------------------

   begin
      Temp_Asin     := Arcsin_R(Input);
      Arcsin_Result := Temp_Asin;
      Arccos_Result := Pi_Over_2 - Temp_Asin;
```

```
   end Arcsin_Arccos;

pragma PAGE;
   procedure Arcsin_Arccos (Input         : in  Sin_Cos_Ratio;
                            Arcsin_Result : out Semicircles;
                            Arccos_Result : out Semicircles) is

--     ---------------------
--     --declaration section
--     ---------------------

       Temp_Asin : Semicircles;

-- -------------------------------
-- --begin procedure Arcsin_Arccos
-- -------------------------------

   begin
      Temp_Asin      := Arcsin_S(Input);
      Arcsin_Result := Temp_Asin;
      Arccos_Result := 0.5 - Temp_Asin;
   end Arcsin_Arccos;

pragma PAGE;
   procedure Arcsin_Arccos (Input         : in  Sin_Cos_Ratio;
                            Arcsin_Result : out Degrees;
                            Arccos_Result : out Degrees) is

--     ---------------------
--     --declaration section
--     ---------------------

       Temp_Asin : Degrees;

-- -------------------------------
-- --begin procedure Arcsin_Arccos
-- -------------------------------

   begin
      Temp_Asin      := Arcsin_D(Input);
      Arcsin_Result := Temp_Asin;
      Arccos_Result := 90.0 - Temp_Asin;
   end Arcsin_Arccos;

pragma PAGE;
   function Arctan (Input : Tan_Ratio) return Radians is
   begin
      retur Arctan_R(Input);
   end Arctan;

   function Arctan (Input : Tan_Ratio) return Semicircles is
   begin
      return Arctan_S(Input);
   end Arctan;

   function Arctan (Input : Tan_Ratio) return Degrees is
   begin
```

```
      return Arctan_D(Input);
   end Arctan;

pragma PAGE;
   function Arctan2 (X : Measurements;
                     Y : Measurements) return Angles is

--    ---------------------
--    --declaration section
--    ---------------------

      Answer    : Angles;
      Tan_Value : Tan_Ratio;

-- ---------------------
-- --begin function body
-- ---------------------

   begin

      if X = 0.0 then

         if Y = 0.0 then
            Answer := 0.0;
         elsif Y > 0.0 then
            Answer := Cycle_Over_4;
         else
            Answer := -CYcle_Over_4;
         end if;

      else

         if abs(X) >= abs(Y) then

            Tan_Value := Y / X;
            Answer := Arctan (Tan_Value);

            if X < 0.0 then
               if Y >= 0.0 then
                  Answer := Answer + Cycle_Over_2;
               else
                  Answer := Answer - Cycle_Over_2;
               end if;
            end if;

         else  --   abs(X) < abs(Y)

            Tan_Value := X / Y;
            Answer := Arctan (Tan_Value);

            if Y >= 0.0 then
               Answer :=  Cycle_Over_4 - Answer;
            else
               Answer := -CYcle_Over_4 - Answer;
            end if;

         end if;
```

```
      end if;
      return Answer;
   end Arctan2;
end Standard_Trig;
```

3.3.6.4 GEOMETRIC_OPERATIONS TLCSC P684 (CATALOG #P118-0)

This part contains the CAMP routines which perform geometric functions relative to the Earth frame.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.4.1 REQUIREMENTS ALLOCATION

The following chart summarizes the allocation of CAMP requirements to this part:

Name	Requirements Allocation
Unit_Radial_Vector	R168
Unit_Normal_Vector	N/A
Compute_Segment_and_Unit_Normal_Vector	R169
Compute_Segment_and_Unit_Normal_ with_Arcsin	N/A
Great_Circle_Arc_Length	R082

3.3.6.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.4.3 INPUT/OUTPUT

None.

3.3.6.4.4 LOCAL DATA

None.

3.3.6.4.5 PROCESS CONTROL

Not applicable.

3.3.6.4.6 PROCESSING

The following describes the processing performed by this part:

```
package body Geometric_Operations is

   function Unit_Radial_Vector
              (Lat_of_Point  : Earth_Positions;
               Long_of_Point : Earth_Positions)
              return Unit_Vectors is separate;
```

```
   function Unit_Normal_Vector
              (Unit_Radial_A : Unit_Vectors;
               Unit_Radial_B : Unit_Vectors) return Unit_Vectors is separate;

   procedure Compute_Segment_and_Unit_Normal_Vector
              (Unit_Radial1     : in     Unit_Vectors;
               Unit_Radial2     : in     Unit_Vectors;
               Unit_Normal2     :    out Unit_Vectors;
               Segment_Distance :    out Segment_Distances) is separate;

   procedure Compute_Segment_and_Unit_Normal_Vector_with_Arcsin
              (Unit_Radial1     : in     Unit_Vectors;
               Unit_Radial2     : in     Unit_Vectors;
               Unit_Normal2     :    out Unit_Vectors;
               Segment_Distance :    out Segment_Distances) is separate;

   package body Great_Circle_Arc_Length is separate;

end Geometric_Operations;
```

3.3.6.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.4.8 LIMITATIONS

None.

3.3.6.4.9 LLCSC DESIGN

3.3.6.4.9.1 GREAT_CIRCLE_ARC_LENGTH PACKAGE DESIGN (CATALOG #P122-0)

This package contains the function required to compute the great circle arc length of a course segment given the latitude and longitude of the two endpoints.

The circle arc length equals: circle radius * angle subtended by the arc

To define the angle subtended by the arc this part calculates the unit radial vectors to the end points of the arc. Since the radials vectors have a length of 1 and since it is assumed the angle subtended by the arc is "relatively small", the following is true:

```
    arc length = radius * angle
               = radius * sin(angle)
               = radius * (1)*(1)*sin(angle)
               = radius * (length of UR_A) * (length of UR_B) * sin(angle)
               = radius * length(URB X UR_A)
```

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.4.9.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirements R082.

3.3.6.4.9.1.2 LOCAL ENTITIES DESIGN

Packages:

The following table describes the package contained local to this part:

Name	Type	Description
V_Opns	package	Defines a vector type and provides operations on that type

Subprograms:

The following subprograms are defined local to this part:

Name	Type	Description
Unit_Rad_Vector	function	Computes the unit radial vector of a point

3.3.6.4.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified.

Data types:

The following table describes the generic formal types required by this part:

Name	Type	Description
Earth_Distances	floating point type	Data type used to define distance measurements involving the Earth's radius
Segment_Distances	floating point type	Data type used to define distance measure-menets involving nagivation segments
Earth_Positions	floating point type	Data type of longitude/latitude measurements
Sin_Cos_Ratio	floating point type	Data type of results of sine/cosine routines

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Value	Description
Earth_Radius	Earth_Distances	N/A	Radius of the Earth

Subprograms:

The following table describes the generic formal subroutines required by this part:

Name	Type	Description
"*"	function	Operator defining the operation: Earth_Distances * Sin_Cos_Ratio => Segment_Distances
Sqrt	function	Square root function
Sin_Cos	procedure	Returns the sine and cosine of an input value

FORMAL PARAMETERS:

The following table defines the formal parameters of the function contained in this package:

Name	Type	Value	Description
Latitude_A	Earth_Positions	\	Latitude and longitude of point A
Latitude_B	Earth_Positions	/	
Longitude_A	Earth_Positions	\	Latitude and longitude of point B
Longitude_B	Earth_Positions	/	

3.3.6.4.9.1.4 LOCAL DATA

Data types:

The following table describes the data types defined by this part:

Name	Description
Unit_Vectors	Subtype of Vector_Operations.Vectors (one-dimensional array of Sin_Cos_Ratio)

Data objects:

The following table describes the data objects maintained by the function contained in this part:

Name	Type	Description
Unit_Radial_A	Unit_Vectors	Unit radial vector pointing to point A
Unit_Radial_B	Unit_Vectors	Unit radial vector pointing to point B
Unit_Normal_B	Unit_Vectors	Segment AB unit normal vector
V_Length	Sin_Cos_Ratio	Length of a vector resulting from a cross product of UR_A and UR_B which, because of the geometry, equals the angle between the two radial vectors

3.3.6.4.9.1.5 PROCESS CONTROL

Not applicable.

3.3.6.4.9.1.6 PROCESSING

The following describes the processing performed by this part:

```
with Coordinate_Vector_Matrix_Algebra;
separate (Geometric_Operations)
package body Great_Circle_Arc_Length is

   package CVMA renames Coordinate_Vector_Matrix_Algebra;

-- ---------------------
-- --local declarations-
-- ---------------------

   type Indices is (X, Y, Z);

-- --instantiation to get a vector type and a vector length function

   package V_Opns is new CVMA.Vector_Operations
                           (Axes             => Indices,
                            Elements         => Sin_Cos_Ratio,
                            Elements_Squared => Sin_Cos_Ratio);

   subtype Unit_Vectors is V_Opns.Vectors;

   function Vector_Length (Input : Unit_Vectors)
                          return Sin_Cos_Ratio
               renames V_Opns.Vector_Length;

-- --instantiations to obtain required operations

   function Cross_Prod is new
               CVMA.Cross_Product (Axes           => Indices,
                                   Left_Elements  => Sin_Cos_Ratio,
                                   Right_Elements => Sin_Cos_Ratio,
                                   Result_Elements => Sin_Cos_Ratio,
                                   Left_Vectors   => Unit_Vectors,
                                   Right_Vectors  => Unit_Vectors,
                                   Result_Vectors => Unit_Vectors);
```

```
   function Unit_Rad_Vector is new
               Unit_Radial_Vector (Indices        => Indices,
                                   Earth_Positions => Earth_Positions,
                                   Sin_Cos_Ratio   => Sin_Cos_Ratio,
                                   Unit_Vectors    => Unit_Vectors);


-- ----------------
-- --function body-
-- ----------------

   function Compute (Latitude_A  : Earth_Positions;
                     Latitude_B  : Earth_Positions;
                     Longitude_A : Earth_Positions;
                     Longitude_B : Earth_Positions) return Segment_Distances is

--    ----------------------
--    --declaration section-
--    ----------------------

      Unit_Radial_A : Unit_Vectors;
      Unit_Radial_B : Unit_Vectors;
      Unit_Normal_B : Unit_Vectors;
      V_Length      : Sin_Cos_Ratio;

-- -------------------------
-- --begin function Compute-
-- -------------------------

   begin

      Unit_Radial_A := Unit_Rad_Vector(Lat_of_Point  => Latitude_A,
                                       Long_of_Point => Longitude_A);
      Unit_Radial_B := Unit_Rad_Vector(Lat_of_Point  => Latitude_B,
                                       Long_of_Point => Longitude_B);

      Unit_Normal_B := Cross_Prod (Left  => Unit_Radial_B,
                                   Right => Unit_Radial_A);

--    --because of the geometry, the length of vector UN_B equals
--    --the angle between the two radial vectors
      V_Length := Vector_Length(Unit_Normal_B);

      return Earth_Radius * V_Length;

   end Compute;

end Great_Circle_Arc_Length;
```

3.3.6.4.9.1.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:

1. Coordinate_Vector_Matrix_Algebra (CVMA)

UTILIZATION OF EXTERNAL ELEMENTS:

Packages:

The following table describes the packages required by this part:

Name	Type	Source	Description
Vector_ Operations	generic package	CVMA	Defines a vector type and provides operations on that type

Subprograms and task entries:

The following table describes the subprograms required by this part and contained in CVMA.Vector_Operations:

Name	Type	Description
Vector_Length	function	Calculates the length of a vector

The following table describes the subprograms required by this part and contained in CVMA:

Name	Type	Description
Cross_Product	generic function	Calculates the cross product of two vectors, returning the resultant vector

Data types:

The following table describes the data types used by this part and defined in CVMA.Vector_Operations:

Name	Type	Range	Description
Vectors	array	N/A	One-dimensional array

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table describes the subprograms required by the function contained in this part and defined in the package specification of Geometric_-Operations.

Name	Type	Description
Unit_Radial_Vector	generic function	Computes the unit radial vector of a point

3.3.6.4.9.1.8 LIMITATIONS

None.

3.3.6.4.9.1.9 LLCSC DESIGN

None.

3.3.6.4.9.1.10 UNIT DESIGN

None.

3.3.6.4.10 UNIT DESIGN

3.3.6.4.10.1 UNIT_RADIAL_VECTOR UNIT DESIGN (CATALOG #P119-0)

This part computes the unit radial vector of a point given the point's latitude and longitude. It extends outward from the origin of the Earth- centered reference frame towards the point whose latitude and longitude are given.

The computations performed by this part are as follows:

```
UR(X) := Cos(Lat) * Cos(Long)
UR(Y) := Cos(Lat) * Sin(Long)
UR(Z) := Sin(Lat)
```

3.3.6.4.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R168.

3.3.6.4.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.4.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified:

Data types:

The following table describes the generic formal types required by this part:

Name	Type	Description
Indices	discrete type	Used to dimension arrays
Earth_Positions	floating point	Data type of longitude/latitude values
Sin_Cos_Ratio	floating point	Data type of results of sine/cosine routines
Unit_Vectors	array	One-dimensional, 3-element array of Sin_Cos_Ratio dimensioned by Indices

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Value	Description
X	Indices	'FIRST	Index into first element of array
Y	Indices	'SUCC(X)	Index into second element of array
Z	Indices	'LAST	Index into third element of array

Subprograms:

The following table describes the generic formal subroutines required by this part:

Name	Type	Description
Sin_Cos	procedure	Returns the sine and cosine of an input value

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Lat_of_Point	Earth_Positions	In	Latitude of point for which a unit radial vector is desired
Long_of_Point	Earth_Positions	In	Latitude of point for which a unit radial vector is desired

3.3.6.4.10.1.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Sin_of_Lat	Sin_Cos_Ratio	N/A	Sine of latitude of point
Cos_of_Lat	Sin_Cos_Ratio	N/A	Cosine of latitude of point
Sin_of_Long	Sin_Cos_Ratio	N/A	Sine of longitude of point
Cos_of_Long	Sin_Cos_Ratio	N/A	Cosine of longitude of point
Radial_Vector	Unit_Vectors	N/A	Radial vector being calculated and returned

3.3.6.4.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.4.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Geometric_Operations)
function Unit_Radial_Vector
            (Lat_of_Point  : Earth_Positions;
             Long_of_Point : Earth_Positions) return Unit_Vectors is

-- ----------------------
-- --declaration section-
-- ----------------------

   Sin_of_Lat    : Sin_Cos_Ratio;
   Cos_of_Lat    : Sin_Cos_Ratio;
   Sin_of_Long   : Sin_Cos_Ratio;
   Cos_of_Long   : Sin_Cos_Ratio;
   Radial_Vector : Unit_Vectors;

-----------------------------------
--begin function Unit_Radial_Vector-
-----------------------------------

begin

   Sin_Cos (Input  => Lat_of_Point,
            Sine   => Sin_of_Lat,
            Cosine => Cos_of_Lat);

   Sin_Cos (Input  => Long_of_Point,
            Sine   => Sin_of_Long,
            Cosine => Cos_of_Long);

   Radial_Vector(X) := Cos_of_Lat * Cos_of_Long;
```

```
   Radial_Vector(Y) := Cos_of_Lat * Sin_of_Long;
   Radial_Vector(Z) := Sin_of_Lat;

   return Radial_Vector;

end Unit_Radial_Vector;
```

3.3.6.4.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.4.10.1.8 LIMITATIONS

None.

3.3.6.4.10.2 UNIT_NORMAL_VECTOR UNIT DESIGN (CATALOG #P120-0)

This function computes the segment unit normal vector for a course segment given the unit radial vectors for the two points defining the course segment.

The computations performed by this part are as follows:

```
    UN_B := UR_B X UR_A / Length(UR_B X UR_A)
    where UN_B :== unit normal vector
          UR_B :== unit radial vector to point B
          UR_A :== unit radial vector to point A
```

3.3.6.4.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R169.

3.3.6.4.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.4.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified:

Data types:

The following table describes the generic formal types required by this part:

Name	Type	Description
Unit_Vectors	private	One-dimensional, 3-element array of Sin_Cos_Ratio
Sin_Cos_Ratio	floating point type	Data type of results of sine/cosine routines

Subprograms:

The following table describes the generic formal subroutines required by this part:

Name	Type	Description
"/"	function	Division operator defining the operation: Unit_Vectors / Sin_Cos_Ratio => Unit_Vectors
Cross_Product	function	Cross product function
Vector_Length	function	Vector length function

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Unit_Radial_A	Unit_Vectors	In	Unit radial vector defining one endpoint of the course segment
Unit_Radial_B	Unit_Vectors	In	Unit radial vector defining one endpoint of the course segment

3.3.6.4.10.2.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Unit_Normal_B	Unit_Vectors	N/A	Vector being calculated and returned
V_Length	Sin_Cos_Ratio	N/A	Vector length

3.3.6.4.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.4.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Geometric_Operations)
function Unit_Normal_Vector
            (Unit_Radial_A : Unit_Vectors;
             Unit_Radial_B : Unit_Vectors) return Unit_Vectors is

-- ----------------------
-- --declaration section-
-- ----------------------

   Unit_Normal_B : Unit_Vectors;
   V_Length      : Sin_Cos_Ratio;

-------------------------------------
--begin function Unit_Normal_Vector-
-------------------------------------

begin

   Unit_Normal_B := Cross_Product(Left  => Unit_Radial_B,
                                  Right => Unit_Radial_A);
   V_Length      := Vector_Length(Input => Unit_Normal_B);

   return (Unit_Normal_B / V_Length);

end Unit_Normal_Vector;
```

3.3.6.4.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.4.10.2.8 LIMITATIONS

None.

3.3.6.4.10.3 COMPUTE_SEGMENT_AND_UNIT_NORMAL_VECTOR UNIT DESIGN (CATALOG #P121-0)

This procedure computes the segment unit normal vector for a course segment and the length of the course segment, given the unit radial vectors for the 2 points defining the course segment.

The computations performed by this part are as follows:

```
UN_2     := UR_2 X UR_1 / Length(UR_2 X UR_1)
Seg_Dist := Earth_Radius * Length(UR_2 X UR_1)
where UN_2     :== unit normal vector
      UR_2     :== unit radial vector to point 2
      UR_1     :== unit radial vector to point 1
      Seg_Dist :== great circle arc length between points 1 and 2
```

3.3.6.4.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R169.

3.3.6.4.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.4.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified:

Data types:

The following table describes the generic formal types required by this part:

Name	Type	Description
Indices	discrete type	Used to dimension Unit_Vectors; this type should have a length of 3
Earth_Distances	floating point type	Data type used to define distance measurements involving the Earth's radius
Segment_Distances	floating point type	Data type used to define distance measuremenets involving nagivation segments
Sin_Cos_Ratio	floating point type	Data type used to define results of sine/cosine operations
Unit_Vectors	array	One-dimensional, 3-element array indexed by Indices and containing Sin_Cos_Ratio elements

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Value	Description
Earth_Radius	Earth_Distances	N/A	Radius of the Earth

Subprograms:

The following table describes the generic formal subroutines required by this part:

Name	Type	Description
"*"	function	Operator defining the operation: Earth_Distances * Sin_Cos_Ratio => Segment_Distances
"/"	function	Operator defining the operation: Unit_Vectors / Sin_Cos_Ratio => Unit_Vectors
Cross_Product	function	Calculates the cross product of two units
Vector_Length	function	Calculates the length of a vector

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Unit_Radial1	Unit_Vectors	in	Unit radial vector to waypoint B
Unit_Radial2	Unit_Vectors	in	Unit radial vector to waypoint C
Unit_Normal2	Unit_Vectors	out	Segment unit normal vector
Segment_Distance	Segment_Distances	out	Great circle arc length between points 1 and 2

3.3.6.4.10.3.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Temp_Unit_Vector	Unit_Vectors	N/A	Temporary unit vector
V_Length	Sin_Cos_Ratio	N/A	Vector length

3.3.6.4.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.4.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Geometric_Operations)
procedure Compute_Segment_and_Unit_Normal_Vector
            (Unit_Radial1     : in  Unit_Vectors;
             Unit_Radial2     : in  Unit_Vectors;
             Unit_Normal2     : out Unit_Vectors;
             Segment_Distance : out Segment_Distances) is
```

```
-- ----------------------
-- --declaration section-
-- ----------------------

   Temp_Unit_Vector : Unit_Vectors;
   V_Length         : Sin_Cos_Ratio;

-------------------------------------------------------
--begin procedure Compute_Segment_and_Unit_Normal_Vectcr-
-------------------------------------------------------

begin

   Temp_Unit_Vector := Cross_Product(Left  => Unit_Radial2,
                                     Right => Unit_Radial1);
   V_Length         := Vector_Length(Input => Temp_Unit_Vector);

   Unit_Normal2     := Temp_Unit_Vector / V_Length;

   Segment_Distance := Earth_Radius * V_Length;

end Compute_Segment_and_Unit_Normal_Vector;
```

3.3.6.4.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.4.10.3.8 LIMITATIONS

None.

3.3.6.4.10.4 COMPUTE_SEGMENT_AND_UNIT_NORMAL_VECTOR_WITH_ARCSIN UNIT DESIGN (CATALOG #P1050-0)

This procedure computes the segment unit normal vector for a course segment and the length of the course segment, given the unit radial vectors for the 2 points defining the course segment.

The computations performed by this part are as follows:

```
UN_2     := UR_2 X UR_1 / Length(UR_2 X UR_1)
Seg_Dist := Earth_Radius * Arcsin(Length(UR_2 X UR_1))
where UN_2     :== unit normal vector
      UR_2     :== unit radial vector to point 2
      UR_1     :== unit radial vector to point 1
      Seg_Dist :== great circle arc length between points 1 and 2
```

3.3.6.4.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R169.

3.3.6.4.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.4.10.4.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified:

Data types:

The following table describes the generic formal types required by this part:

Name	Type	Description
Indices	discrete type	Used to dimension Unit_Vectors; this type should have a length of 3
Earth_Distances	floating point type	Data type used to define distance measurements involving the Earth's radius
Segment_Distances	floating point type	Data type used to define distance measuremenets involving nagivation segments
Radians	floating point type	Data type used to define angular measurement
Sin_Cos_Ratio	floating point type	Data type used to define results of sine/cosine operations
Unit_Vectors	array	One-dimensional, 3-element array indexed by Indices and containing Sin_Cos_Ratio elements

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Value	Description
Earth_Radius	Earth_Distances	N/A	Radius of the Earth

Subprograms:

The following table describes the generic formal subroutines required by this part:

Name	Type	Description
"*"	function	Operator defining the operation: Earth_Distances * Sin_Cos_Ratio => Segment_Distances
"/"	function	Operator defining the operation: Unit_Vectors / Sin_Cos_Ratio => Unit_Vectors
Cross_Product	function	Calculates the cross product of two units
Vector_Length	function	Calculates the length of a vector

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Unit_Radial1	Unit_Vectors	in	Unit radial vector to waypoint B
Unit_Radial2	Unit_Vectors	in	Unit radial vector to waypoint C
Unit_Normal2	Unit_Vectors	out	Segment unit normal vector
Segment_Distance	Segment_Distances	out	Great circle arc length between points 1 and 2

3.3.6.4.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Temp_Unit_Vector	Unit_Vectors	N/A	Temporary unit vector
V_Length	Sin_Cos_Ratio	N/A	Vector length

3.3.6.4.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.4.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Geometric_Operations)
procedure Compute_Segment_and_Unit_Normal_Vector_with_Arcsin
            (Unit_Radial1     : in  Unit_Vectors;
             Unit_Radial2     : in  Unit_Vectors;
             Unit_Normal2     : out Unit_Vectors;
             Segment_Distance : out Segment_Distances) is
```

```
-- ----------------------
-- --declaration section-
-- ----------------------

   Temp_Unit_Vector : Unit_Vectors;
   V_Length         : Sin_Cos_Ratio;

------------------------------------------------------------
--begin procedure Compute_Segment_and_Unit_Normal_Vector-
------------------------------------------------------------

begin

   Temp_Unit_Vector := Cross_Product(Left  => Unit_Radial2,
                                     Right => Unit_Radial1);
   V_Length         := Vector_Length(Input => Temp_Unit_Vector);

   Unit_Normal2     := Temp_Unit_Vector / V_Length;

   Segment_Distance := Earth_Radius * Arcsin(V_Length);

end Compute_Segment_and_Unit_Normal_Vector_with_Arcsin;
```

3.3.6.4.10.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.4.10.4.8 LIMITATIONS

None.

(This page left intentionally blank.)

```
package body Geometric_Operations is

   function Unit_Radial_Vector
               (Lat_Of_Point  : Earth_Positions;
                Long_Of_Point : Earth_Positions)
               return Unit_Vectors is separate;

   function Unit_Normal_Vector
               (Unit_Radial_A : Unit_Vectors;
                Unit_Radial_B : Unit_Vectors) return Unit_Vectors is separate;

   procedure Compute_Segment_And_Unit_Normal_Vector
               (Unit_Radial1     : in     Unit_Vectors;
                Unit_Radial2     : in     Unit_Vectors;
                Unit_Normal2     :    out Unit_Vectors;
                Segment_Distance :    out Segment_Distances) is separate;

   procedure Compute_Segment_And_Unit_Normal_Vector_With_Arcsin
               (Unit_Radial1     : in     Unit_Vectors;
                Unit_Radial2     : in     Unit_Vectors;
                Unit_Normal2     :    out Unit_Vectors;
                Segment_Distance :    out Segment_Distances) is separate;

   package body Great_Circle_Arc_Length is separate;

end Geometric_Operations;
```

```
separate (Geometric_Operations)
function Unit_Radial_Vector
            (Lat_Of_Point  : Earth_Positions;
             Long_Of_Point : Earth_Positions) return Unit_Vectors is

-- ----------------------
-- --declaration section-
-- ----------------------

   Sin_Of_Lat    : Sin_Cos_Ratio;
   Cos_Of_Lat    : Sin_Cos_Ratio;
   Sin_Of_Long   : Sin_Cos_Ratio;
   Cos_Of_Long   : Sin_Cos_Ratio;
   Radial_Vector : Unit_Vectors;

-------------------------------------
--begin function Unit_Radial_Vector-
-------------------------------------

begin

   Sin_Cos (Input  => Lat_Of_Point,
            Sine   => Sin_Of_Lat,
            Cosine => Cos_Of_Lat);

   Sin_Cos (Input  => Long_Of_Point,
            Sine   => Sin_Of_Long,
            Cosine => Cos_Of_Long);

   Radial_Vector(X) := Cos_Of_Lat * Cos_Of_Long;
   Radial_Vector(Y) := Cos_Of_Lat * Sin_Of_Long;
   Radial_Vector(Z) := Sin_Of_Lat;

   return Radial_Vector;

end Unit_Radial_Vector;
```

```
separate (Geometric_Operations)
function Unit_Normal_Vector
            (Unit_Radial_A : Unit_Vectors;
             Unit_Radial_B : Unit_Vectors) return Unit_Vectors is

-- ----------------------
-- --declaration section-
-- ----------------------

   Unit_Normal_B : Unit_Vectors;
   V_Length      : Sin_Cos_Ratio;

-----------------------------------
--begin function Unit_Normal_Vector-
-----------------------------------

begin

   Unit_Normal_B := Cross_Product(Left  => Unit_Radial_B,
                                  Right => Unit_Radial_A);
   V_Length      := Vector_Length(Input => Unit_Normal_B);

   return (Unit_Normal_B / V_Length);

end Unit_Normal_Vector;
```

```
separate (Geometric_Operations)
procedure Compute_Segment_And_Unit_Normal_Vector
             (Unit_Radial1     : in  Unit_Vectors;
              Unit_Radial2     : in  Unit_Vectors;
              Unit_Normal2     : out Unit_Vectors;
              Segment_Distance : out Segment_Distances) is

-- ----------------------
-- --declaration section-
-- ----------------------

   Temp_Unit_Vector : Unit_Vectors;
   V_Length         : Sin_Cos_Ratio;

-------------------------------------------------------------
--begin procedure Compute_Segment_and_Unit_Normal_Vector-
-------------------------------------------------------------

begin

   Temp_Unit_Vector := Cross_Product(Left  => Unit_Radial2,
                                     Right => Unit_Radial1);
   V_Length         := Vector_Length(Input => Temp_Unit_Vector);

   Unit_Normal2     := Temp_Unit_Vector / V_Length;

   Segment_Distance := Earth_Radius * V_Length;

end Compute_Segment_And_Unit_Normal_Vector;
```

```
with Coordinate_Vector_Matrix_Algebra;
separate (Geometric_Operations)
package body Great_Circle_Arc_Length is

   package Cvma renames Coordinate_Vector_Matrix_Algebra;

-- ---------------------
-- --local declarations-
-- ---------------------

   type Indices is (X, Y, Z);

-- --instantiation to get a vector type and a vector length function

   package V_Opns is new Cvma.Vector_Operations
                         (Axes             => Indices,
                          Elements         => Sin_Cos_Ratio,
                          Elements_Squared => Sin_Cos_Ratio);

   subtype Unit_Vectors is V_Opns.Vectors;

   function Vector_Length (Input : Unit_Vectors)
                          return Sin_Cos_Ratio
            renames V_Opns.Vector_Length;

-- --instantiations to obtain required operations

   function Cross_Prod is new
            Cvma.Cross_Product (Axes           => Indices,
                                Left_Elements  => Sin_Cos_Ratio,
                                Right_Elements => Sin_Cos_Ratio,
                                Result_Elements => Sin_Cos_Ratio,
                                Left_Vectors   => Unit_Vectors,
                                Right_Vectors  => Unit_Vectors,
                                Result_Vectors => Unit_Vectors);

   function Unit_Rad_Vector is new
            Unit_Radial_Vector (Indices         => Indices,
                                Earth_Positions => Earth_Positions,
                                Sin_Cos_Ratio   => Sin_Cos_Ratio,
                                Unit_Vectors    => Unit_Vectors);

pragma PAGE;
-- ----------------
-- --function body-
-- ----------------

   function Compute (Latitude_A  : Earth_Positions;
                     Latitude_B  : Earth_Positions;
                     Longitude_A : Earth_Positions;
                     Longitude_B : Earth_Positions) return Segment_Distances is

--    ----------------------
--    --declaration section-
--    ----------------------

      Unit_Radial_A : Unit_Vectors;
```

```
      Unit_Radial_B : Unit_Vectors;
      Unit_Normal_B : Unit_Vectors;
      V_Length      : Sin_Cos_Ratio;

-- -------------------------
-- --begin function Compute-
-- -------------------------

   begin

      Unit_Radial_A := Unit_Rad_Vector(Lat_Of_Point  => Latitude_A,
                                       Long_Of_Point => Longitude_A);
      Unit_Radial_B := Unit_Rad_Vector(Lat_Of_Point  => Latitude_B,
                                       Long_Of_Point => Longitude_B);

      Unit_Normal_B := Cross_Prod (Left  => Unit_Radial_B,
                                   Right => Unit_Radial_A);

--      --because of the geometry, the length of vector UN_B equals
--      --the angle between the two radial vectors
      V_Length := Vector_Length(Unit_Normal_B);

      return Earth_Radius * V_Length;

   end Compute;

end Great_Circle_Arc_Length;
```

```
separate (Geometric_Operations)
procedure Compute_Segment_And_Unit_Normal_Vector_With_Arcsin
             (Unit_Radial1     : in  Unit_Vectors;
              Unit_Radial2     : in  Unit_Vectors;
              Unit_Normal2     : out Unit_Vectors;
              Segment_Distance : out Segment_Distances) is

-- ----------------------
-- --declaration section-
-- ----------------------

   Temp_Unit_Vector : Unit_Vectors;
   V_Length         : Sin_Cos_Ratio;

-- ------------------------------------------------------------
--begin procedure Compute_Segment_and_Unit_Normal_Vector-
-- ------------------------------------------------------------

begin

   Temp_Unit_Vector := Cross_Product(Left  => Unit_Radial2,
                                     Right => Unit_Radial1);
   V_Length         := Vector_Length(Input => Temp_Unit_Vector);

   Unit_Normal2     := Temp_Unit_Vector / V_Length;

   Segment_Distance := Earth_Radius * Arcsin(V_Length);

end Compute_Segment_And_Unit_Normal_Vector_With_Arcsin;
```

(This page left intentionally blank.)

3.3.6.5 DATA_CONVERSION

(This page intentionally left blank.)

3.3.6.5.1 UNIT_CONVERSIONS (PACKAGE BODY) TLCSC P851 (CATALOG #P631-0)

This part, which is a package of generic packages, provides a set of functions which convert data objects from one unit of measurement to another.

The table below shows which package contains the functions to perform the unit conversions between which units:

Name	Type A	Type B
Meters_and_Feet	Meters	Feet
Meters_and_Feet_per_Second	Meters_per_Second	Feet_per_Second
Meters_and_Feet_per_Second_Squared	Meters_per_Second_Squared	Feet_per_Second_Squared
Gees_and_Meters_per_Second_Squared	Gees	Meters_per_Second_Squared
Gees_and_Feet_per_Second_Squared	Gees	Feet_per_Second_Squared
Radians_and_Degrees	Radians	Degrees
Radians_and_Degrees_per_Second	Radians_per_Second	Degrees_per_Second
Radians_and_Semicircles	Radians	Semicircles
Radians_and_Semicircles_per_Second	Radians_per_Second	Semicircles_per_Second
Degrees_and_Semicircles	Degrees	Semicircles
Degrees_and_Semicircles_per_Second	Degrees_per_Second	Semicircles_per_Second
Seconds_and_Minutes	Seconds	Minutes
Centigrade_and_Fahrenheit	Centigrade	Fahrenheit
Centigrade_and_Kelvin	Centigrade	Kelvin
Fahrenheit_and_Kelvin	Fahrenheit	Kelvin
Kilograms_and_Pounds	Kilograms	Pounds
Kilograms_per_Meter_Squared_and_Pounds_per_Foot_Squared	Kilograms_per_Meters_Squared	Pounds_per_Foot_Squared

The following table lists the catalog numbers for subunits contained in this part:

Name	Catalog _#
Kilograms_per_Meter_Squared_and_ Pounds_per_Foot_Squared	P632-0
Conversion_to_Pounds_per_Foot2	P633-0
Conversion_to_Kilograms_per_Meter2	P634-0
Meters_and_Feet	P635-0
Conversion_to_Feet	P636-0
Conversion_to_Meters	P637-0
Meters_and_Feet_per_Second	P638-0
Conversion_to_Feet_per_Second	P639-0
Conversion_to_Meters_per_Second	P640-0
Meters_and_Feet_per_Second_Squared	P641-0
Conversion_to_Feet_per_Second2	P642-0
Conversion_to_Meters_per_Second2	P643-0
Gees_and_Meters_per_Second_Squared	P644-0
Conversion_to_Meters_per_Second2	P645-0
Conversion_to_Gees	P646-0
Gees_and_Feet_per_Second_Squared	P647-0
Conversion_to_Feet_per_Second2	P648-0
Conversion_to_Gees	P649-0
Radians_and_Degrees	P650-0
Conversion_to_Degrees	P651-0
Conversion_to_Radians	P652-0
Radians_and_Degrees_per_Second	P653-0
Conversion_to_Degrees_per_Second	P654-0
Conversion_to_Radians_per_Second	P655-0
Radians_and_Semicircles	P656-0
Conversion_to_Semicircles	P657-0
Conversion_to_Radians	P658-0
Radians_and_Semicircles_per_Second	P659-0
Conversion_to_Semicircles_per_Second	P660-0
Conversion_to_Radians_per_Second	P661-0
Degrees_and_Semicircles	P662-0
Conversion_to_Semicircles	P663-0
Conversion_to_Degrees	P664-0
Degrees_and_Semicircles_per_Second	P665-0
Conversion_to_Semicircles_per_Second	P666-0
Conversion_to_Degrees_per_Second	P667-0
Seconds_and_Minutes	P668-0
Conversion_to_Minutes	P669-0
Conversion_to_Seconds	P670-0
Centigrade_and_Fahrenheit	P671-0
Conversion_to_Fahrenheit	P672-0
Conversion_to_Centigrade	P673-0
Centigrade_and_Kelvin	P674-0
Conversion_to_Kelvin	P675-0
Conversion_to_Centigrade	P676-0
Fahrenheit_and_Kelvin	P677-0
Conversion_to_Kelvin	P678-0
Conversion_to_Fahrenheit	P679-0
Kilograms_and_Pounds	P680-0
Conversion_to_Kilograms	P681-0
Conversion_to_Pounds	P682-0

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.5.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R105.

3.3.6.5.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.5.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

All of the packages contained in this TLCSC are generic packages. The types defining the units between which the conversions are taking place have been defined as generic formal parameters to the individual packages.

FORMAL PARAMETERS:

Each of the units contained in this part's LLCSC's require one input parameter. That parameter is of the type from which the conversion is to take place.

3.3.6.5.1.4 LOCAL DATA

None.

3.3.6.5.1.5 PROCESS CONTROL

Not applicable.

3.3.6.5.1.6 PROCESSING

The following describes the processing performed by this part:

```
with Conversion_Factors;
package body Unit_Conversions is

   package CF renames Conversion_Factors;


-- ------------------------------------------------------------
-- --package containing subroutines to convert Meters <==> Feet
-- ------------------------------------------------------------

   package body Meters_and_Feet is

      function Conversion_to_Feet (Input : Meters) return Feet is
```

```
      begin
         return Feet(Input) * CF.Feet_per_Meter;
      end Conversion_to_Feet;

      function Conversion_to_Meters (Input : Feet) return Meters is
      begin
         return Meters(Input) * CF.Meters_per_Foot;
      end Conversion_to_Meters;

   end Meters_and_Feet;


-- ------------------------------------------------------------------------
-- --package containing routines to convert Meters/Second <==> Feet/Second
-- ------------------------------------------------------------------------

   package body Meters_and_Feet_per_Second is

      function Conversion_to_Feet_per_Second
                  (Input : Meters_per_Second) return Feet_per_Second is
      begin
         return Feet_per_Second(Input) * CF.Feet_per_Meter;
      end Conversion_to_Feet_per_Second;

      function Conversion_to_Meters_per_Second
                  (Input : Feet_per_Second) return Meters_per_Second is
      begin
         return Meters_per_Second(Input) * CF.Meters_per_Foot;
      end Conversion_to_Meters_per_Second;

   end Meters_and_Feet_per_Second;


-- ---------------------------------------------------------------------------
-- --package containing routines to convert Meters/Second**2 <==> Feet/Second**2
-- ---------------------------------------------------------------------------

   package body Meters_and_Feet_per_Second_Squared is

      function Conversion_to_Feet_per_Second2
                  (Input : Meters_per_Second_Squared)
                  return Feet_per_Second_Squared is
      begin
         return Feet_per_Second_Squared(Input) * CF.Feet_per_Meter;
      end Conversion_to_Feet_per_Second2 ;

      function Conversion_to_Meters_per_Second2
                  (Input : Feet_per_Second_Squared)
                 return Meters_per_Second_Squared is
      begin
         return Meters_per_Second_Squared(Input) * CF.Meters_per_Foot;
      end Conversion_to_Meters_per_Second2;

   end Meters_and_Feet_per_Second_Squared;


-- ------------------------------------------------------------------
```

```
-- --package containing routines to convert Gees <==> Meters/Second**2
-- ------------------------------------------------------------------

   package body Gees_and_Meters_per_Second_Squared is

      function Conversion_to_Meters_per_Second2
                  (Input : Gees) return Meters_per_Second_Squared is
      begin
         return Meters_per_Second_Squared(Input) * CF.Meters_per_Sec2_per_Gee;
      end Conversion_to_Meters_per_Second2;

      function Conversion_to_Gees
                  (Input : Meters_per_Second_Squared) return Gees is
      begin
         return Gees(Input) * CF.Gees_per_Meter_per_Sec2;
      end Conversion_to_Gees;

   end Gees_and_Meters_per_Second_Squared;


-- ------------------------------------------------------------------
-- --package containing routines to convert Gees <==> Feet/Second**2
-- ------------------------------------------------------------------

   package body Gees_and_Feet_per_Second_Squared is

      function Conversion_to_Feet_per_Second2
                  (Input : Gees) return Feet_per_Second_Squared is
      begin
         return Feet_per_Second_Squared(Input) * CF.Feet_per_Sec2_per_Gee;
      end Conversion_to_Feet_per_Second2 ;

      function Conversion_to_Gees
                  (Input : Feet_per_Second_Squared) return Gees is
      begin
         return Gees(Input) * CF.Gees_per_Foot_per_Sec2;
      end Conversion_to_Gees;

   end Gees_and_Feet_per_Second_Squared;


-- ------------------------------------------------------------
-- --package containing routines to convert Radians <==> Degrees
-- ------------------------------------------------------------

   package body Radians_and_Degrees is

      function Conversion_to_Degrees (Input : Radians) return Degrees is
      begin
         return Degrees(Input) * CF.Degrees_per_Radian;
      end Conversion_to_Degrees ;

      function Conversion_to_Radians (Input : Degrees) return Radians is
      begin
         return Radians(Input) * CF.Radians_per_Degree;
      end Conversion_to_Radians ;
```

```
   end Radians_and_Degrees;


-- ------------------------------------------------------------------------
-- --package containing routines to convert Radians/Second <==> Degrees/Second
-- ------------------------------------------------------------------------

   package body Radians_and_Degrees_per_Second is

      function Conversion_to_Degrees_per_Second
                  (Input : Radians_per_Second)
                  return Degrees_per_Second is
      begin
         return Degrees_per_Second(Input) * CF.Degrees_per_Radian;
      end Conversion_to_Degrees_per_Second;

      function Conversion_to_Radians_per_Second
                  (Input : Degrees_per_Second)
                  return Radians_per_Second is
      begin
         return Radians_per_Second(Input) * CF.Radians_per_Degree;
      end Conversion_to_Radians_per_Second;

   end Radians_and_Degrees_per_Second;


-- ---------------------------------------------------------------
-- --package containing routines to convert Radians <==> Semicircles
-- ---------------------------------------------------------------

   package body Radians_and_Semicircles is

      function Conversion_to_Semicircles (Input : Radians) return Semicircles is
      begin
         return Semicircles(Input) * CF.Semi_Circles_per_Radian;
      end Conversion_to_Semicircles ;

      function Conversion_to_Radians (Input : Semicircles) return Radians is
      begin
         return Radians(Input) * CF.Radians_per_Semi_Circle;
      end Conversion_to_Radians ;

   end Radians_and_Semicircles;


-- ---------------------------------------
-- --package containing routines to convert
-- --Radians/Second <==> Semicircles/Second
-- ---------------------------------------

   package body Radians_and_Semicircles_per_Second is

      function Conversion_to_Semicircles_per_Second
                  (Input : Radians_per_Second) return Semicircles_per_Second is
      begin
         return Semicircles_per_Second(Input) * CF.Semi_Circles_per_Radian;
      end Conversion_to_Semicircles_per_Second ;
```

```
      function Conversion_to_Radians_per_Second
                  (Input : Semicircles_per_Second) return Radians_per_Second is
      begin
         return Radians_per_Second(Input) * CF.Radians_per_Semi_Circle;
      end Conversion_to_Radians_per_Second;

   end Radians_and_Semicircles_per_Second;


-- ------------------------------------------------------------
-- --package containing routines to convert Degrees <==> Semicircles
-- ------------------------------------------------------------

   package body Degrees_and_Semicircles is

      function Conversion_to_Semicircles (Input : Degrees) return Semicircles is
      begin
         return Semicircles(Input) * CF.Semi_Circles_per_Degree;
      end Conversion_to_Semicircles ;

      function Conversion_to_Degrees (Input : Semicircles) return Degrees is
      begin
         return Degrees(Input) * CF.Degrees_per_Semi_Circle;
      end Conversion_to_Degrees ;

   end Degrees_and_Semicircles;


-- ---------------------------------------
-- --package containing routines to convert
-- --Degrees/Second <==> Semicircles/Second
-- ---------------------------------------

   package body Degrees_and_Semicircles_per_Second is

      function Conversion_to_Semicircles_per_Second
                  (Input : Degrees_per_Second)
                  return Semicircles_per_Second is
      begin
         return Semicircles_per_Second(Input) * CF.Semi_Circles_per_Degree;
      end Conversion_to_Semicircles_per_Second ;

      function Conversion_to_Degrees_per_Second
                  (Input : Semicircles_per_Second)
                 return Degrees_per_Second is
      begin
         return Degrees_per_Second(Input) * CF.Degrees_per_Semi_Circle;
      end Conversion_to_Degrees_per_Second;

   end Degrees_and_Semicircles_per_Second;


-- ------------------------------------------------------------
-- --package containing routines to convert Seconds <==> Minutes
-- ------------------------------------------------------------
```

```
   package body Seconds_and_Minutes is

      function Conversion_to_Minutes (Input : Seconds) return Minutes is
      begin
         return Minutes(Input) * CF.Minutes_per_Second;
      end Conversion_to_Minutes ;

      function Conversion_to_Seconds (Input : Minutes) return Seconds is
      begin
         return Seconds(Input) * CF.Seconds_per_Minute;
      end Conversion_to_Seconds ;

   end Seconds_and_Minutes;


-- ------------------------------------------------------------------
-- --package containing routines to convert Centigrade <==> Fahrenheit
-- ------------------------------------------------------------------

   package body Centigrade_and_Fahrenheit is

      function Conversion_to_Fahrenheit
                  (Input : Centigrade) return Fahrenheit is
      begin
         return Fahrenheit(Input) * CF.Degrees_Fahrenheit_per_Centigrade +
                CF.Centigrade_to_Fahrenheit_Bias;
      end Conversion_to_Fahrenheit ;

      function Conversion_to_Centigrade
                  (Input : Fahrenheit) return Centigrade is
      begin
         return Centigrade(Input + CF.Fahrenheit_to_Centigrade_Bias) *
                CF.Degrees_Centigrade_per_Fahrenheit;
      end Conversion_to_Centigrade;

   end Centigrade_and_Fahrenheit;


-- ---------------------------------------------------------------
-- --package containing routines to convert Centigrade <==> Kelvin
-- ---------------------------------------------------------------

   package body Centigrade_and_Kelvin is

      function Conversion_to_Kelvin (Input : Centigrade) return Kelvin is
      begin
         return Kelvin(Input + CF.Centigrade_to_Kelvin_Bias);
      end Conversion_to_Kelvin ;

      function Conversion_to_Centigrade (Input : Kelvin) return Centigrade is
      begin
         return Centigrade(Input + CF.Kelvin_to_Centigrade_Bias);
      end Conversion_to_Centigrade ;

   end Centigrade_and_Kelvin;
```

```
-- ------------------------------------------------------------
-- --package containing routines to convert Fahrenheit <==> Kelvin
-- ------------------------------------------------------------

   package body Fahrenheit_and_Kelvin is

      function Conversion_to_Kelvin (Input : Fahrenheit) return Kelvin is
      begin
         return Kelvin(Input + CF.Fahrenheit_to_Kelvin_Bias) *
                CF.Degrees_Kelvin_per_Fahrenheit ;
      end Conversion_to_Kelvin ;

      function Conversion_to_Fahrenheit (Input : Kelvin) return Fahrenheit is
      begin
         return Fahrenheit(Input) * CF.Degrees_Fahrenheit_per_Kelvin +
                CF.Kelvin_to_Fahrenheit_Bias;
      end Conversion_to_Fahrenheit ;

   end Fahrenheit_and_Kelvin;


-- ------------------------------------------------------------
-- --package containing routines to convert Kilograms <==> Pounds
-- ------------------------------------------------------------

   package body Kilograms_and_Pounds is

      function Conversion_to_Pounds (Input : Kilograms) return Pounds is
      begin
         return Pounds(Input) * CF.Pounds_per_Kilogram;
      end Conversion_to_Pounds ;

      function Conversion_to_Kilograms (Input : Pounds) return Kilograms is
      begin
         return Kilograms(Input) * CF.Kilograms_per_Pound;
      end Conversion_to_Kilograms ;

   end Kilograms_and_Pounds;


-- ---------------------------------------
-- --package containing routines to convert
-- --Kilograms/Meter**2 <==> Pounds/Foot**2
-- ---------------------------------------

   package body Kilograms_per_Meter_Squared_and_Pounds_per_Foot_Squared is

      function Conversion_to_Pounds_per_Foot2
                  (Input : Kilograms_per_Meter_Squared)
                  return Pounds_per_Foot_Squared is
      begin
         return Pounds_per_Foot_Squared(Input) *
                CF.Pounds_per_Foot2_per_Kilograms_per_Meter2;
      end Conversion_to_Pounds_per_Foot2;

      function Conversion_to_Kilograms_per_Meter2
                  (Input : Pounds_per_Foot_Squared)
```

```
                return Kilograms_per_Meter_Squared is
      begin
         return Kilograms_per_Meter_Squared(Input) *
                CF.Kilograms_per_Meter2_per_Pounds_per_Foot2;
      end Conversion_to_Kilograms_per_Meter2;

   end Kilograms_per_Meter_Squared_and_Pounds_per_Foot_Squared;

end Unit_Conversions;
```

3.3.6.5.1.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:
1. Conversion_Factors (CF)

UTILIZATION OF EXTERNAL ELEMENTS:

Data objects:

All of the conversion factors required by the units nested in this part are contained in the Conversion_Factors TLCSC.

3.3.6.5.1.8 LIMITATIONS

None.

3.3.6.5.1.9 LLCSC DESIGN

None.

3.3.6.5.1.10 UNIT DESIGN

None.

```
with Conversion_Factors;
package body Unit_Conversions is

   package Cf renames Conversion_Factors;

pragma PAGE;
-- ------------------------------------------------------------
-- --package containing subroutines to convert Meters <==> Feet
-- ------------------------------------------------------------

   package body Meters_And_Feet is

      function Conversion_To_Feet (Input : Meters) return Feet is
      begin
         return Feet(Input) * Cf.Feet_Per_Meter;
      end Conversion_To_Feet;

      function Conversion_To_Meters (Input : Feet) return Meters is
      begin
         return Meters(Input) * Cf.Meters_Per_Foot;
      end Conversion_To_Meters;

   end Meters_And_Feet;


-- ------------------------------------------------------------------------
-- --package containing routines to convert Meters/Second <==> Feet/Second
-- ------------------------------------------------------------------------

   package body Meters_And_Feet_Per_Second is

      function Conversion_To_Feet_Per_Second
                  (Input : Meters_Per_Second) return Feet_Per_Second is
      begin
         return Feet_Per_Second(Input) * Cf.Feet_Per_Meter;
      end Conversion_To_Feet_Per_Second;

      function Conversion_To_Meters_Per_Second
                  (Input : Feet_Per_Second) return Meters_Per_Second is
      begin
         return Meters_Per_Second(Input) * Cf.Meters_Per_Foot;
      end Conversion_To_Meters_Per_Second;

   end Meters_And_Feet_Per_Second;

pragma PAGE;
-- ------------------------------------------------------------------------------
-- --package containing routines to convert Meters/Second**2 <==> Feet/Second**2
-- ------------------------------------------------------------------------------

   package body Meters_And_Feet_Per_Second_Squared is

      function Conversion_To_Feet_Per_Second2
                  (Input : Meters_Per_Second_Squared)
                  return Feet_Per_Second_Squared is
      begin
         return Feet_Per_Second_Squared(Input) * Cf.Feet_Per_Meter;
```

```
      end Conversion_To_Feet_Per_Second2 ;

      function Conversion_To_Meters_Per_Second2
                . (Input : Feet_Per_Second_Squared)
                 return Meters_Per_Second_Squared is
      begin
         return Meters_Per_Second_Squared(Input) * Cf.Meters_Per_Foot;
      end Conversion_To_Meters_Per_Second2;

   end Meters_And_Feet_Per_Second_Squared;


-- ------------------------------------------------------------------
-- --package containing routines to convert Gees <==> Meters/Second**2
-- ------------------------------------------------------------------

   package body Gees_And_Meters_Per_Second_Squared is

      function Conversion_To_Meters_Per_Second2
                  (Input : Gees) return Meters_Per_Second_Squared is
      begin
         return Meters_Per_Second_Squared(Input) * Cf.Meters_Per_Sec2_Per_Gee;
      end Conversion_To_Meters_Per_Second2;

      function Conversion_To_Gees
                  (Input : Meters_Per_Second_Squared) return Gees is
      begin
         return Gees(Input) * Cf.Gees_Per_Meter_Per_Sec2;
      end Conversion_To_Gees;

   end Gees_And_Meters_Per_Second_Squared;

pragma PAGE;
-- ------------------------------------------------------------------
-- --package containing routines to convert Gees <==> Feet/Second**2
-- ------------------------------------------------------------------

   package body Gees_And_Feet_Per_Second_Squared is

      function Conversion_To_Feet_Per_Second2
                  (Input : Gees) return Feet_Per_Second_Squared is
      begin
         return Feet_Per_Second_Squared(Input) * Cf.Feet_Per_Sec2_Per_Gee;
      end Conversion_To_Feet_Per_Second2 ;

      function Conversion_To_Gees
                  (Input : Feet_Per_Second_Squared) return Gees is
      begin
         return Gees(Input) * Cf.Gees_Per_Foot_Per_Sec2;
      end Conversion_To_Gees;

   end Gees_And_Feet_Per_Second_Squared;

pragma PAGE;
-- ------------------------------------------------------------
-- --package containing routines to convert Radians <==> Degrees
-- ------------------------------------------------------------
```

```
   package body Radians_And_Degrees is

      function Conversion_To_Degrees (Input : Radians) return Degrees is
      begin
         return Degrees(Input) * Cf.Degrees_Per_Radian;
      end Conversion_To_Degrees ;

      function Conversion_To_Radians (Input : Degrees) return Radians is
      begin
         return Radians(Input) * Cf.Radians_Per_Degree;
      end Conversion_To_Radians ;

   end Radians_And_Degrees;


-- ------------------------------------------------------------------------
-- --package containing routines to convert Radians/Second <==> Degrees/Second
-- ------------------------------------------------------------------------

   package body Radians_And_Degrees_Per_Second is

      function Conversion_To_Degrees_Per_Second
                  (Input : Radians_Per_Second)
                  return Degrees_Per_Second is
      begin
         return Degrees_Per_Second(Input) * Cf.Degrees_Per_Radian;
      end Conversion_To_Degrees_Per_Second;

      function Conversion_To_Radians_Per_Second
                  (Input : Degrees_Per_Second)
                  return Radians_Per_Second is
      begin
         return Radians_Per_Second(Input) * Cf.Radians_Per_Degree;
      end Conversion_To_Radians_Per_Second;

   end Radians_And_Degrees_Per_Second;

pragma PAGE;
-- ------------------------------------------------------------------
-- --package containing routines to convert Radians <==> Semicircles
-- ------------------------------------------------------------------

   package body Radians_And_Semicircles is

      function Conversion_To_Semicircles (Input : Radians) return Semicircles is
      begin
         return Semicircles(Input) * Cf.Semi_Circles_Per_Radian;
      end Conversion_To_Semicircles ;

      function Conversion_To_Radians (Input : Semicircles) return Radians is
      begin
         return Radians(Input) * Cf.Radians_Per_Semi_Circle;
      end Conversion_To_Radians ;

   end Radians_And_Semicircles;
```

```
-- ----------------------------------------
-- --package containing routines to convert
-- --Radians/Second <==> Semicircles/Second
-- ----------------------------------------

   package body Radians_And_Semicircles_Per_Second is

      function Conversion_To_Semicircles_Per_Second
                  (Input : Radians_Per_Second) return Semicircles_Per_Second is
      begin
         return Semicircles_Per_Second(Input) * Cf.Semi_Circles_Per_Radian;
      end Conversion_To_Semicircles_Per_Second ;

      function Conversion_To_Radians_Per_Second
                  (Input : Semicircles_Per_Second) return Radians_Per_Second is
      begin
         return Radians_Per_Second(Input) * Cf.Radians_Per_Semi_Circle;
      end Conversion_To_Radians_Per_Second;

   end Radians_And_Semicircles_Per_Second;

pragma PAGE;
-- ------------------------------------------------------------------
-- --package containing routines to convert Degrees <==> Semicircles
-- ------------------------------------------------------------------

   package body Degrees_And_Semicircles is

      function Conversion_To_Semicircles (Input : Degrees) return Semicircles is
      begin
         return Semicircles(Input) * Cf.Semi_Circles_Per_Degree;
      end Conversion_To_Semicircles ;

      function Conversion_To_Degrees (Input : Semicircles) return Degrees is
      begin
         return Degrees(Input) * Cf.Degrees_Per_Semi_Circle;
      end Conversion_To_Degrees ;

   end Degrees_And_Semicircles;


-- ----------------------------------------
-- --package containing routines to convert
-- --Degrees/Second <==> Semicircles/Second
-- ----------------------------------------

   package body Degrees_And_Semicircles_Per_Second is

      function Conversion_To_Semicircles_Per_Second
                  (Input : Degrees_Per_Second)
                  return Semicircles_Per_Second is
      begin
         return Semicircles_Per_Second(Input) * Cf.Semi_Circles_Per_Degree;
      end Conversion_To_Semicircles_Per_Second ;

      function Conversion_To_Degrees_Per_Second
```

```
                (Input : Semicircles_Per_Second)
               return Degrees_Per_Second is
      begin
         return Degrees_Per_Second(Input) * Cf.Degrees_Per_Semi_Circle;
      end Conversion_To_Degrees_Per_Second;

   end Degrees_And_Semicircles_Per_Second;

pragma PAGE;
-- ------------------------------------------------------------
-- --package containing routines to convert Seconds <==> Minutes
-- ------------------------------------------------------------

   package body Seconds_And_Minutes is

      function Conversion_To_Minutes (Input : Seconds) return Minutes is
      begin
         return Minutes(Input) * Cf.Minutes_Per_Second;
      end Conversion_To_Minutes ;

      function Conversion_To_Seconds (Input : Minutes) return Seconds is
      begin
         return Seconds(Input) * Cf.Seconds_Per_Minute;
      end Conversion_To_Seconds ;

   end Seconds_And_Minutes;

pragma PAGE;
-- ---------------------------------------------------------------------
-- --package containing routines to convert Centigrade <==> Fahrenheit
-- ---------------------------------------------------------------------

   package body Centigrade_And_Fahrenheit is

      function Conversion_To_Fahrenheit
                  (Input : Centigrade) return Fahrenheit is
      begin
         return Fahrenheit(Input) * Cf.Degrees_Fahrenheit_Per_Centigrade +
                Cf.Centigrade_To_Fahrenheit_Bias;
      end Conversion_To_Fahrenheit ;

      function Conversion_To_Centigrade
                  (Input : Fahrenheit) return Centigrade is
      begin
         return Centigrade(Input + Cf.Fahrenheit_To_Centigrade_Bias) *
                Cf.Degrees_Centigrade_Per_Fahrenheit;
      end Conversion_To_Centigrade;

   end Centigrade_And_Fahrenheit;


-- ------------------------------------------------------------
-- --package containing routines to convert Centigrade <==> Kelvin
-- ------------------------------------------------------------

   package body Centigrade_And_Kelvin is
```

```
      function Conversion_To_Kelvin (Input : Centigrade) return Kelvin is
      begin
         return Kelvin(Input + Cf.Centigrade_To_Kelvin_Bias);
      end Conversion_To_Kelvin ;

      function Conversion_To_Centigrade (Input : Kelvin) return Centigrade is
      begin
         return Centigrade(Input + Cf.Kelvin_To_Centigrade_Bias);
      end Conversion_To_Centigrade ;

   end Centigrade_And_Kelvin;

pragma PAGE;
-- ------------------------------------------------------------------
-- --package containing routines to convert Fahrenheit <==> Kelvin
-- ------------------------------------------------------------------

   package body Fahrenheit_And_Kelvin is

      function Conversion_To_Kelvin (Input : Fahrenheit) return Kelvin is
      begin
         return Kelvin(Input + Cf.Fahrenheit_To_Kelvin_Bias) *
                Cf.Degrees_Kelvin_Per_Fahrenheit ;
      end Conversion_To_Kelvin ;

      function Conversion_To_Fahrenheit (Input : Kelvin) return Fahrenheit is
      begin
         return Fahrenheit(Input) * Cf.Degrees_Fahrenheit_Per_Kelvin +
                Cf.Kelvin_To_Fahrenheit_Bias;
      end Conversion_To_Fahrenheit ;

   end Fahrenheit_And_Kelvin;

pragma PAGE;
-- ------------------------------------------------------------------
-- --package containing routines to convert Kilograms <==> Pounds
-- ------------------------------------------------------------------

   package body Kilograms_And_Pounds is

      function Conversion_To_Pounds (Input : Kilograms) return Pounds is
      begin
         return Pounds(Input) * Cf.Pounds_Per_Kilogram;
      end Conversion_To_Pounds ;

      function Conversion_To_Kilograms (Input : Pounds) return Kilograms is
      begin
         return Kilograms(Input) * Cf.Kilograms_Per_Pound;
      end Conversion_To_Kilograms ;

   end Kilograms_And_Pounds;


-- ----------------------------------------
-- --package containing routines to convert
-- --Kilograms/Meter**2 <==> Pounds/Foot**2
-- ----------------------------------------
```

```
   package body Kilograms_Per_Meter_Squared_And_Pounds_Per_Foot_Squared is

      function Conversion_To_Pounds_Per_Foot2
                  (Input : Kilograms_Per_Meter_Squared)
                  return Pounds_Per_Foot_Squared is
      begin
         return Pounds_Per_Foot_Squared(Input) *
                Cf.Pounds_Per_Foot2_Per_Kilograms_Per_Meter2;
      end Conversion_To_Pounds_Per_Foot2;

      function Conversion_To_Kilograms_Per_Meter2
                  (Input : Pounds_Per_Foot_Squared)
                  return Kilograms_Per_Meter_Squared is
      begin
         return Kilograms_Per_Meter_Squared(Input) *
                Cf.Kilograms_Per_Meter2_Per_Pounds_Per_Foot2;
      end Conversion_To_Kilograms_Per_Meter2;

   end Kilograms_Per_Meter_Squared_And_Pounds_Per_Foot_Squared;

end Unit_Conversions;
```

(This page left intentionally blank.)

3.3.6.5.2 EXTERNAL_FORM_CONVERSION_TWOS_COMPLEMENT (PACKAGE BODY) TLCSC P852 (CATALOG #P684-0)

This generic package performs scaling operations on input values. It is able to convert two's complement engineering units to floating point representations and to convert floating point to engineering units.

NOTE: The scaled values, while representing two's complement values, are themselves one's complement values and, therefore, are always positive.

The calculations to go from a scaled integer value to an unscaled floating point value are as follows:

```
unscaled_output := unscaled_bias +
                   ((scaled_value - scale_factor_2) *
                    unscaled_range / scale_factor_1)
```

and the calculations to go from an unscaled floating point value to a scaled integer are as follows:

```
scaled_output := (unscaled_value - unscaled_bias) *
                 (scale_factor_1 / unscaled_range)
                 + scale_factor_2
```

where:

```
scale_factor_1 := 2 ** initial_engineering_units_bits - 1
   (represents the value range which may be assumed by the
    scaled, integer values)

scale_factor_2 := 2 ** (initial_engineering_units_bits - 1)
   (represents the scaled bias; i.e., the amount by which
    the minumum scaled, integer value is negatively offset
    from 0)

unscaled_bias  := [(unscaled_max - unscaled_min + lsb_value)
                   / 2] + unscaled_min
   (represents the offset from 0 of the median unscaled value)

unscaled_range := unscaled_max - unscaled_min
   (represents the value range which may be assumed by the
    unscaled, floating point values)
```

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.5.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R106.

3.3.6.5.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.5.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously declared when this was specified.

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Scaled_Integers	integer type	Defines scaled variables stored in engineering units
Unscaled_Floats	floating point type	Defines unscaled variables stored in floating point format

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Bits_In_Unscaled_Values	POSITIVE	Number of significant bits in the engineering units representation
Initial_Min_Unscaled_Value	Unscaled_Floats	Minimum value which the unscaled values may assume
Initial_Max_Unscaled_Value	Unscaled_Floats	Maximum value which the unscaled values may assume

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
"*"	function	Multiplication operator defining the operation: Scaled_Integers * Unscaled_Floats => Unscaled_Floats
"*"	function	Multiplication operator defining the operation: Unscaled_Floats * Unscaled_Floats => Scaled_Integers
"/"	function	Division operator defining the operation: Unscaled_Floats / Scaled_Integers => Unscaled_Floats
"/"	function	Division operator defining the operation: Unscaled_Floats / Unscaled_Floats => Scaled_Integers

3.3.6.5.2.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
One_Over_LSB_Value	Unscaled_Floats	1/LSB_Value	Inverse of the value of the least significant bit
Scaled_Bias	Scaled_Integers	2**(Bits_in_Scaled_Values-1)	Amount by which the median scaled value is negatively offset from 0
Unscaled_Bias	Unscaled_Floats	Initial_Unscaled_Bias	Amount by which the median unscaled value is negatively offset from 0

3.3.6.5.2.5 PROCESS CONTROL

Not applicable.

3.3.6.5.2.6 PROCESSING

The following describes the processing performed by this part:

```
package body External_Form_Conversion_Twos_Complement is

-- ------------------
-- --local variables-
-- ------------------

   One_Over_LSB_Value : constant Unscaled_Floats := 1.0 / LSB_Value;

   Scaled_Bias   : constant Scaled_Integers := 2**(Bits_in_Scaled_Values-1);
   Unscaled_Bias : constant Unscaled_Floats
                     := ((Max_Unscaled_Value - Min_Unscaled_Value +
                          LSB_Value) / 2 ) +
                        Min_Unscaled_Value;


begin

-- --make sure min < max

   if Initial_Min_Unscaled_Value > Initial_Max_Unscaled_Value then
      raise NUMERIC_ERROR;
   end if;

end External_Form_Conversion_Twos_Complement;
```

3.3.6.5.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the top level component:

Data types:

The following table describes the data types defined in the package specification of the TLCSC:

Name	Range	Description
Positive_Scaled_Integers	0 .. Max_Scaled_Value	Subtype of generic formal type Scaled_Integers; used to ensure the values of the scaled input parameters are within the allowable range
Valid_Unscaled_Floats	Min_Unscaled_Value .. Max_Unscaled_Value	Subtype of generic formal type Unscaled_Floats; used to ensure the values of the scaled input parameters are within the allowable range

Data objects:

The following table describes the data objects which were defined in the package specification of the TLCSC:

Name	Type	Value	Description
Max_Scaled_Value	Scaled_Integers	2**Bits_in_Scaled_Values - 1	Maximum scaled value
Value_Range	Unscaled_Floats	Initial_Max_Unscaled_Value - Initial_Min_Unscaled_Value	Range of values which may be assumed by the unscaled values
LSB_Value	Unscaled_Floats	Initial_Value_Range / Max_Scaled_Value	Value of the least significant bit in the scaled values
Min_Unscaled_Value	Unscaled_Floats	Initial_Min_Unscaled_Value	Minimum unscaled value
Max_Unscaled_Value	Unscaled_Floats	Initial_Max_Unscaled_Value	Maximum unscaled value

3.3.6.5.2.8 LIMITATIONS

This part raises a NUMERIC_ERROR exception if Initial_Min_Unscaled_Value is greater than Initial_Max_Unscaled_Value.

3.3.6.5.2.9 LLCSC DESIGN

None.

3.3.6.5.2.10 UNIT DESIGN

3.3.6.5.2.10.1 SCALE (FUNCTION BODY) UNIT DESIGN (CATALOG #P685-0)

This function accepts a floating point value and performs a scaling operation on it to convert it to engineering units representation.

NOTE: The scaled values, while representing two's complement values, are themselves one's complement values and, therefore, are always positive.

The calculations to go from an unscaled floating point value to a scaled integer are as follows:

```
scaled_output := (unscaled_value - unscaled_bias) *
                 (scale_factor_1 / unscaled_range) + scale_factor_2
```

where:

```
scale_factor_1 := 2 ** initial_engineering_units_bits - 1
   (represents the value range which may be assumed by the
    scaled, integer values)

scale_factor_2 := 2 ** (initial_engineering_units_bits - 1)
   (represents the scaled bias; i.e., the amount by which
    the minumum scaled, integer value is negatively offset
    from 0)

unscaled_bias  := initial_bias
   (represents the amount by which the minimum unscaled,
    floating point value is negatively offset from 0)

unscaled_range := initial_range
   (represents the value range which may be assumed by the
    unscaled, floating point values; i.e., equals the
    maximum unscaled value - minumum unscaled value)
```

3.3.6.5.2.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R106.

3.3.6.5.2.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.5.2.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Unscaled_ Value	Valid_ Unscaled_ Floats	In	Floating point value which is to be scaled so it may be represented in an integer format

3.3.6.5.2.10.1.4 LOCAL DATA

None.

3.3.6.5.2.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.5.2.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
function Scale (Unscaled_Value : Valid_Unscaled_Floats)
                return Positive_Scaled_Integers is

   Answer : Scaled_Integers;

begin

   Answer := (Unscaled_Value - Unscaled_Bias) * One_Over_LSB_Value +
             Scaled_Bias;
   return Answer;

end Scale;
```

3.3.6.5.2.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined as generic formal subroutines to the External_Form_-Conversion_Twos_Complement package:

Name	Type	Description
"*"	function	Multiplication operator defining the operation: Unscaled_Floats * Unscaled_Floats => Scaled_Integers

Data types:

The following table summarizes the types required by this part and defined in the package specification of the External_Form_Conversion_Twos_Complement package:

Name	Range	Description
Positive_Scaled_Integers	0 .. Max_Scaled_Value	Subtype of generic formal type Scaled_Integers; used to ensure the values of the scaled input parameters are within the allowable range
Valid_Unscaled_Floats	Min_Unscaled_Value .. Max_Unscaled_Value	Subtype of generic formal type Unscaled_Floats; used to ensure the values of the scaled input parameters are within the allowable range

Data objects:

The following table summarizes the objects required by this part and defined in the package body of the External_Form_Conversion_Twos_Complement package:

Name	Type	Value	Description
One_Over_LSB_Value	Unscaled_Floats	1/LSB_Value	Inverse of the value of the least significant bit
Scaled_Bias	Scaled_Integers	2**(Bits_in_Scaled_Values-1)	Amount by which the median scaled value is negatively offset from 0
Unscaled_Bias	Unscaled_Floats	Initial_Unscaled_Bias	Amount by which the median unscaled value negatively offset from 0

3.3.6.5.2.10.1.8 LIMITATIONS

None.

3.3.6.5.2.10.2 UNSCALE UNIT DESIGN (CATALOG #P686-0)

This function accepts a value in engineering units representation and perform an unscaling operation on it to convert it to a floating point representation.

NOTE: The scaled values, while representing two's complement values, are themselves one's complement values and, therefore, are always positive.

The calculations to go from a scaled integer value to an unscaled floating point value are as follows:

```
unscaled_output := unscaled_bias +
                   ((scaled_value - scale_factor_2) *
                    unscaled_range /
                    scale_factor_1)
```

where:

```
scale_factor_1 := 2 ** initial_engineering_units_bits - 1
   (represents the value range which may be assumed by the
    scaled, integer values)

scale_factor_2 := 2 ** (initial_engineering_units_bits - 1)
   (represents the scaled bias; i.e., the amount by which
    the minumum scaled, integer value is negatively offset
    from 0)

unscaled_bias  := initial_bias
   (represents the amount by which the minimum unscaled,
    floating point value is negatively offset from 0)

unscaled_range := initial_range
   (represents the value range which may be assumed by the
    unscaled, floating point values; i.e., equals the
    maximum unscaled value - minumum unscaled value)
```

3.3.6.5.2.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R106.

3.3.6.5.2.10.2.2 LOCAL ENTITIES DESIGN

Subprograms:

This package body contains code which is executed when the part is elaborated. This code checks to ensure that Initial_Min_Unscaled_Value > Initial_Max_-Unscaled_Value. If it is not, a NUMERIC_ERROR exception is raised.

3.3.6.5.2.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The fcllowing table describes this part's formal parameters:

Name	Type	Mode	Description
Scaled_ Value	Scaled_ Integers	In	Integer scaled value for which an unscaled, floating point representation is desired

3.3.6.5.2.10.2.4 LOCAL DATA

Data objects:

The following tables describes the objects maintained by this part:

Name	Type	Description
Answer	Scaled_Integers	Value being calculated and returned

3.3.6.5.2.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.5.2.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Unscale (Scaled_Value : Positive_Scaled_Integers)
                 return Valid_Unscaled_Floats is

   Answer : Unscaled_Floats;

begin

   Answer := Unscaled_Bias + (Scaled_Value - Scaled_Bias) * LSB_Value;
   return Answer;

end Unscale;
```

3.3.6.5.2.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined as generic formal subroutines to the External_Form_-Conversion_Twos_Complement package:

Name	Type	Description
"*"	function	Multiplication operator defining the operation: Scaled_Integers * Unscaled_Floats => Unscaled_Floats

Data types:

The following table summarizes the types required by this part and defined in the package specification of the External_Form_Conversion_Twos_Complement package:

Name	Range	Description
Positive_Scaled_Integers	0 .. Max_Scaled_Value	Subtype of generic formal type Scaled_Integers; used to ensure the values of the scaled input parameters are within the allowable range
Valid_Unscaled_Floats	Min_Unscaled_Value .. Max_Unscaled_Value	Subtype of generic formal type Unscaled_Floats; used to ensure the values of the scaled input parameters are within the allowable range

Data objects:

The following table summarizes the objects required by this part and defined in the package body of the External_Form_Conversion_Twos_Complement package:

Name	Type	Value	Description
LSB_Value	Unscaled_Floats	N/A	Value of the least significant bit
Scaled_Bias	Scaled_Integers	2**(Bits_in_Scaled_Values-1)	Amount by which the median scaled value is negatively offset from 0
Unscaled_Bias	Unscaled_Floats	Initial_Unscaled_Bias	Amount by which the median unscaled value negatively offset from 0

3.3.6.5.2.10.2.8 LIMITATIONS

None.

```
package body External_Form_Conversion_Twos_Complement is

-- ------------------
-- --local variables-
-- ------------------

   One_Over_Lsb_Value : constant Unscaled_Floats := 1.0 / Lsb_Value;

   Scaled_Bias   : constant Scaled_Integers := 2**(Bits_In_Scaled_Values-1);
   Unscaled_Bias : constant Unscaled_Floats
                     := ((Max_Unscaled_Value - Min_Unscaled_Value +
                          Lsb_Value) / 2 ) +
                        Min_Unscaled_Value;

pragma PAGE;
   function Scale (Unscaled_Value : Valid_Unscaled_Floats)
                  return Positive_Scaled_Integers is

      Answer : Scaled_Integers;

   begin

      Answer := (Unscaled_Value - Unscaled_Bias) * One_Over_Lsb_Value +
                Scaled_Bias;
      return Answer;

   end Scale;

pragma PAGE;
   function Unscale (Scaled_Value : Positive_Scaled_Integers)
                    return Valid_Unscaled_Floats is

      Answer : Unscaled_Floats;

   begin

      Answer := Unscaled_Bias + (Scaled_Value - Scaled_Bias) * Lsb_Value;
      return Answer;

   end Unscale;

pragma PAGE;
begin

-- --make sure min < max

   if Initial_Min_Unscaled_Value > Initial_Max_Unscaled_Value then
      raise NUMERIC_ERROR;
   end if;

end External_Form_Conversion_Twos_Complement;
```

(This page left intentionally blank.)

3.3.6.6 SIGNAL_PROCESSING (BODY) TLCSC P686 (BODY) (CATALOG #P81-0)

This package provides signal processing parts. Each part is designed as an Ada generic package, where the generic parameters will specify the data types of the input and output signals and the values for coefficients used in performing the signal processing functions.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.1 REQUIREMENTS ALLOCATION

The following diagram summarizes the allocation of CAMP requirements to this part's LLCSC's.

Name	Type	Req. Allocation
Limiter (Upper & Lower Bounds)	generic package	R108
Limiter (Upper Bound)	generic package	R037
Limiter (Lower Bound)	generic package	R038
Absolute limiter	generic package	R160
Absolute limiter with flag	generic package	R202
General First Order Filter	generic package	R109
Tustin Lag Filter	generic package	R162
Tustin Lead-Lag Filter	generic package	R161
Second Order (Notch) Filter	generic package	R110, R111
Tustin Integrator with Limit	generic package	R203
Tustin Integrator with Asymmetric Limit	generic package	N/A

3.3.6.6.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.3 INPUT/OUTPUT

None.

3.3.6.6.4 LOCAL DATA

None.

3.3.6.6.5 PROCESS CONTROL

Not applicable.

3.3.6.6.6 PROCESSING

The following describes the processing performed by this part:

```
package body Signal_Processing is
    package body Upper_Lower_Limiter is separate;
    package body Upper_Limiter is separate;
    package body Lower_Limiter is separate;
    package body Absolute_Limiter is separate;
    package body Absolute_Limiter_With_Flag is separate;
    package body General_First_Order_Filter is separate;
    package body Tustin_Lead_Lag_Filter is separate;
    package body Tustin_Lag_Filter is separate;
    package body Second_Order_Filter is separate;
    package body Tustin_Integrator_With_Limit is separate;
    package body Tustin_Integrator_With_Asymmetric_Limit is separate;
begin
   null;
exception
   when others => raise;
end Signal_Processing;
```

3.3.6.6.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.8 LIMITATIONS

None.

3.3.6.6.9 LLCSC DESIGN

3.3.6.6.9.1 UPPER_LOWER_LIMITER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P82-0)

This package exports operations to perform a limiter function on an input signal (with both upper and lower bounds) and to update the values of the bounds. The package initializes the limits as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.1.1 REQUIREMENTS ALLOCATION

R108

3.3.6.6.9.1.2 LOCAL ENTITIES DESIGN

Subprograms:

This package contains a sequence of statements at the end of the package body which are executed when this part is elaborated. The code initializes the upper and lower bounds by calling the Update_Limits procedure from this package. Update_Limits raises the Limit_Exception if the upper bound is less than or equal to the lower bound.

3.3.6.6.9.1.3 INPUT/OUTPUT

None.

3.3.6.6.9.1.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Upper_Limit	Signal_Type	Current value of upper limit to limiter.
Lower_Limit	Signal_Type	Current value of lower limit to limiter.

3.3.6.6.9.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.1.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body UPPER_LOWER_LIMITER is

-------------------------
-- -- Local Data Objects-
-------------------------

   Upper_Limit,
   Lower_Limit : Signal_Type;


-----------------------------------
--begin processing for package body
-----------------------------------

begin

   Update_Limits (New_Upper_Limit => Initial_Upper_Limit,
                  New_Lower_Limit => Initial_Lower_Limit);

end Upper_Lower_Limiter;
```

3.3.6.6.9.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.1.8 LIMITATIONS

Name	When/Why Raised
Limit_Exception	This exception is raised if the value of the Upper_Limit <= Lower_Limit

3.3.6.6.9.1.9 LLCSC DESIGN

None.

3.3.6.6.9.1.10 UNIT DESIGN

3.3.6.6.9.1.10.1 UPDATE_LIMITS (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the values of the bounds which were initialized at the elaboration of the instantiation. It will raise an exception if the new lower limit is greater than the new upper limit.

3.3.6.6.9.1.10.1.1 REQUIREMENTS ALLOCATION

R108

3.3.6.6.9.1.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.1.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Upper_Limit	Signal_Type	New value for upper limit to limiter.
New_Lower_Limit	Signal_Type	New value for lower limit to limiter.

Global parameters:

Name	Type	Description
Upper_Limit	Signal_Type	Current value of upper limit to limiter.
Lower_Limit	Signal_Type	Current value of lower limit to limiter.

3.3.6.6.9.1.10.1.4 LOCAL DATA

None.

3.3.6.6.9.1.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.1.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limits (New_Upper_Limit : in Signal_Type;
                         New_Lower_Limit : in Signal_Type) is
begin

   if New_Upper_Limit > New_Lower_Limit then
      Upper_Limit := New_Upper_Limit;
      Lower_Limit := New_Lower_Limit;
   else
       raise Limit_Exception;
   end if;

end Update_Limits;
```

3.3.6.6.9.1.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.1.10.1.8 LIMITATIONS

Name	When/Why Raised
Limit_Exception	This exception is raised if the value of the New_Upper_Limit <= New_Lower_Limit

3.3.6.6.9.1.10.2 LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the limit operation on the input signal.

3.3.6.6.9.1.10.2.1 REQUIREMENTS ALLOCATION

R108

3.3.6.6.9.1.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.1.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Input value of signal to be limited.

Global parameters:

Name	Type	Description
Upper_Limit	Signal_Type	Current value of upper limit to limiter.
Lower_Limit	Signal_Type	Current value of lower limit to limiter.

3.3.6.6.9.1.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Limited_Signal	Signal_Type	Value of signal after limit is applied.

3.3.6.6.9.1.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.1.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit (Signal : Signal_Type) return Signal_Type is

   Limited_Signal: Signal_Type;

begin

   if Signal > Upper_Limit then
      Limited_Signal := Upper_Limit;
   elsif Signal < Lower_Limit then
      Limited_Signal := Lower_Limit;
   else
      Limited_Signal := Signal;
   end if;

   return Limited_Signal;

end LIMIT;
```

3.3.6.6.9.1.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.1.10.2.8 LIMITATIONS

None.

3.3.6.6.9.2 UPPER_LIMITER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P83-0)

This package exports operations to perform a limiter function on an input signal (with upper bounds) and to update the value of the bounds. The package initializes the limits as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.2.1 REQUIREMENTS ALLOCATION

R037

3.3.6.6.9.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

Data objects:

Name	Type	Description
Initial_Upper_Limit	Signal_Type	Initial value of upper limit signals to limiter.

3.3.6.6.9.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Upper_Limit	Signal_Type	Current value of upper limit to limiter. Initialized to Initial_Upper_Limit.

3.3.6.6.9.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.2.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Signal_Processing)
package body Upper_Limiter is
```

```
------------------------
-- -- Local Data Object-
------------------------

   Upper_Limit: Signal_Type := Initial_Upper_Limit;

end Upper_Limiter;
```

3.3.6.6.9.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.2.8 LIMITATICNS

None.

3.3.6.6.9.2.9 LLCSC DESIGN

None.

3.3.6.6.9.2.10 UNIT DESIGN

3.3.6.6.9.2.10.1 UPDATE_LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the values of the bounds which were initialized at the elaboration of the instantiation.

3.3.6.6.9.2.10.1.1 REQUIREMENTS ALLOCATION

R037

3.3.6.6.9.2.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.2.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Upper_Limit	Signal_Type	New value for upper limit to limiter.

Global parameters:

Name	Type	Description
Upper_Limit	Signal_Type	Current value of upper limit to limiter.

3.3.6.6.9.2.10.1.4 LOCAL DATA

None.

3.3.6.6.9.2.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.2.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limit (New_Upper_Limit : in Signal_Type) is
begin

   Upper_Limit := New_Upper_Limit;

end Update_Limit;
```

3.3.6.6.9.2.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.2.10.1.8 LIMITATIONS

None.

3.3.6.6.9.2.10.2 LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the limit operation on the input signal.

3.3.6.6.9.2.10.2.1 REQUIREMENTS ALLOCATION

R037

3.3.6.6.9.2.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.2.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Input value of signal to be limited.

Global parameters:

Name	Type	Description
Upper_Limit	Signal_Type	Current value of upper limit to limiter.

3.3.6.6.9.2.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Limited_Signal	Signal_Type	Value of signal after limit is applied.

3.3.6.6.9.2.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.2.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit (Signal : Signal_Type) return Signal_Type is

   Limited_Signal: Signal_Type;

begin

   if Signal > Upper_Limit then
      Limited_Signal := Upper_Limit;
   else
      Limited_Signal := Signal;
   end if;

   return Limited_Signal;

end LIMIT;
```

3.3.6.6.9.2.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.2.10.2.8 LIMITATIONS

None.

3.3.6.6.9.3 LOWER_LIMITER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P84-0)

This package exports operations to perform a limiter function on an input signal (with lower bound) and to update the value of the bound. The package initializes the limits as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.3.1 REQUIREMENTS ALLOCATION

R038

3.3.6.6.9.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

Data objects:

Name	Type	Description
Initial_Lower_Limit	Signal_Type	Initial value of lower limit signals to limiter.

3.3.6.6.9.3.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Lower_Limit	Signal_Type	Current value of lower limit to limiter. Initialized to Initial_Lower_Limit.

3.3.6.6.9.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.3.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body LOWER_LIMITER is

------------------------
-- -- Local data objects
------------------------

   Lower_Limit : Signal_Type := Initial_Lower_Limit;

end Lower_Limiter;
```

3.3.6.6.9.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.3.8 LIMITATIONS

None.

3.3.6.6.9.3.9 LLCSC DESIGN

None.

3.3.6.6.9.3.10 UNIT DESIGN

3.3.6.6.9.3.10.1 UPDATE_LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the value of the bound which was initialized at the elaboration of the instantiation.

3.3.6.6.9.3.10.1.1 REQUIREMENTS ALLOCATION

R038

3.3.6.6.9.3.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.3.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Lower_Limit	Signal_Type	New value for lower limit to limiter.

Global parameters:

Name	Type	Description
Lower_Limit	Signal_Type	Current value of lower limit to limiter.

3.3.6.6.9.3.10.1.4 LOCAL DATA

None.

3.3.6.6.9.3.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.3.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limit (New_Lower_Limit : in Signal_Type) is

begin
   Lower_Limit := New_Lower_Limit;
end Update_Limit;
```

3.3.6.6.9.3.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.3.10.1.8 LIMITATIONS

None.

3.3.6.6.9.3.10.2 LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the limit operation on the input signal.

3.3.6.6.9.3.10.2.1 REQUIREMENTS ALLOCATION

R037

3.3.6.6.9.3.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.3.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Input value of signal to be limited.

Global parameters:

Name	Type	Description
Lower_Limit	Signal_Type	Current value of lower limit to limiter.

3.3.6.6.9.3.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Limited_Signal	Signal_Type	Value of signal after limit is applied.

3.3.6.6.9.3.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.3.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit (Signal : Signal_Type) return Signal_Type is

   Limited_Signal: Signal_Type;

begin

   if Signal < Lower_Limit then
      Limited_Signal := Lower_Limit;
   else
      Limited_Signal := Signal;
   end if;

   return Limited_Signal;

end LIMIT;
```

3.3.6.6.9.3.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.3.10.2.8 LIMITATIONS

None.

3.3.6.6.9.4 ABSOLUTE_LIMITER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P85-0)

This package exports operations to perform a limiter function on an input signal (with absolute bounds) and to update the value of the bounds. The package initializes the limits as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.4.1 REQUIREMENTS ALLOCATION

R160

3.3.6.6.9.4.2 LOCAL ENTITIES DESIGN

Name	Type	Description
Signal_Sign	Function	Instantiation of Sign function from General_Purpose_Math.

3.3.6.6.9.4.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

Data objects:

Name	Type	Description
Initial_Absolute Limit	Signal_Type	Initial value of absolute limit of signals to limiter.

3.3.6.6.9.4.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Absolute_Limit	Signal_Type	Current value of abslute limit to limiter.

3.3.6.6.9.4.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.4.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body Absolute_Limiter is

-- --------------------
-- --Local Data Object-
-- --------------------

   Absolute_Limit: Signal_Type := abs (Initial_Absolute_Limit);

end Absolute_Limiter;
```

3.3.6.6.9.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.4.8 LIMITATIONS

None.

3.3.6.6.9.4.9 LLCSC DESIGN

None.

3.3.6.6.9.4.10 UNIT DESIGN

3.3.6.6.9.4.10.1 UPDATE_LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the value of the absolute bound which was initialized at the elaboration of the instantiation.

3.3.6.6.9.4.10.1.1 REQUIREMENTS ALLOCATION

R160

3.3.6.6.9.4.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.4.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Absolute_Limit	Signal_Type	New value for absolute limit to limiter.

Global parameters:

Name	Type	Source	Description
Absolute_Limit	Signal_Type	Package Body	Current value of absolute limit to limiter.

3.3.6.6.9.4.10.1.4 LOCAL DATA

None.

3.3.6.6.9.4.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.4.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limit (New_Absolute_Limit : in Signal_Type) is
begin

   Absolute_Limit := abs (New_Absolute_Limit);

end Update_Limit;
```

3.3.6.6.9.4.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.4.10.1.8 LIMITATIONS

None.

3.3.6.6.9.4.10.2 LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the limit operation on the input signal.

3.3.6.6.9.4.10.2.1 REQUIREMENTS ALLOCATION

R160

3.3.6.6.9.4.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.4.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Input value of signal to be limited.

Global parameters:

Name	Type	Source	Description
Absolute_limit	Signal_Type	Package Body	Current value of absolute limit to limiter.

3.3.6.6.9.4.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Limited_Signal	Signal_Type	Value of signal after limit is applied.

3.3.6.6.9.4.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.4.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit (Signal : Signal_Type) return Signal_Type is
   Limited_Signal: Signal_Type;
begin
   if abs (Signal) > Absolute_Limit then
      if Signal > 0.0 then
         Limited_Signal := Absolute_Limit;
      else
         Limited_Signal := - Absolute_Limit;
      end if;
   else
      Limited_Signal := Signal;
   end if;
   return Limited_Signal;
```

```
end LIMIT;
```

3.3.6.6.9.4.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.4.10.2.8 LIMITATIONS

None.

3.3.6.6.9.5 ABSOLUTE_LIMITER_WITH_FLAG (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P86-0)

This package exports operations to perform a limiter function on an input signal (with absolute bounds) and to update the value of the bounds. The limit operation will set a flag to represent the relation- ship of the signal to the limit: Within_Limit, At_Positive_Limit, At_Negative_Limit. The package initializes the limits as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.5.1 REQUIREMENTS ALLOCATION

R202

3.3.6.6.9.5.2 LOCAL ENTITIES DESIGN

Name	Type	Description
Signal_Sign	Function	Instantiation of Sign function from General_Purpose_Math.

3.3.6.6.9.5.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

Data objects:

Name	Type	Description
Initial_Absolute_Limit	Signal_Type	Initial value of absolute limit of signals to limiter.

3.3.6.6.9.5.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Absolute_Limit	Signal_Type	Current value of absolute limit to limiter.
Limit_Relation	Limit_Relations	Represents relationship of most recent signal to limit. Initialized to Within_Limit.

3.3.6.6.9.5.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.5.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body Absolute_Limiter_With_Flag is

-----------------------
-- --Local Data Object-
-----------------------

   Absolute_Limit: Signal_Type := abs (Initial_Absolute_Limit);

   Limit_Relation: Limit_Relations := Within_Limit;


end Absolute_Limiter_with_Flag;
```

3.3.6.6.9.5.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.5.8 LIMITATIONS

None.

3.3.6.6.9.5.9 LLCSC DESIGN

None.

3.3.6.6.9.5.10 UNIT DESIGN

3.3.6.6.9.5.10.1 UPDATE_LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the value of the absolute bound which was initialized at the elaboration of the instantiation.

3.3.6.6.9.5.10.1.1 REQUIREMENTS ALLOCATION

R202

3.3.6.6.9.5.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.5.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Absolute_Limit	Signal_Type	New value for absolute limit to limiter.

Global parameters:

Name	Type	Source	Description
Absolute_Limit	Signal_Type	Package Body	Current value of absolute limit to limiter.

3.3.6.6.9.5.10.1.4 LOCAL DATA

None.

3.3.6.6.9.5.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.5.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limit (New_Absolute_Limit : in Signal_Type) is
begin

   Absolute_Limit := abs (New_Absolute_Limit);

end Update_Limit;
```

3.3.6.6.9.5.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.5.10.1.8 LIMITATIONS

None.

3.3.6.6.9.5.10.2 LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the limit operation on the input signal.

3.3.6.6.9.5.10.2.1 REQUIREMENTS ALLOCATION

R160

3.3.6.6.9.5.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.5.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to limiter.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Input value of signal to be limited.

Global parameters:

Name	Type	Source	Description
Absolute_limit	Signal_Type	Package Body	Current value of absolute limit to limiter.

3.3.6.6.9.5.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Limited_Signal	Signal_Type	Value of signal after limit is applied.

3.3.6.6.9.5.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.5.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit (Signal : Signal_Type) return Signal_Type is

   Limited_Signal: Signal_Type;

begin

   if abs (Signal) >= Absolute_Limit then

      if Signal >= 0.0 then

         Limited_Signal := Absolute_Limit;
         Limit_Relation := At_Positive_Limit;

      else

         Limited_Signal := - Absolute_Limit;
         Limit_Relation := At_Negative_Limit;

      end if;

   else

      Limited_Signal := Signal;
      Limit_Relation := Within_Limit;

   end if;

   return Limited_Signal;

end LIMIT;
```

3.3.6.6.9.5.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.5.10.2.8 LIMITATIONS

None.

3.3.6.6.9.5.10.3 LIMIT_FLAG_SETTING (SUBPROGRAM BODY) UNIT DESIGN

This subprogram returns the current setting of the limit relation flag.

3.3.6.6.9.5.10.3.1 REQUIREMENTS ALLOCATION

R202

3.3.6.6.9.5.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.5.10.3.3 INPUT/OUTPUT

None.

3.3.6.6.9.5.10.3.4 LOCAL DATA

None.

3.3.6.6.9.5.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.5.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit_Flag_Setting return Limit_Relations is
begin

   return Limit_Relation;

end Limit_Flag_Setting;
```

3.3.6.6.9.5.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.5.10.3.8 LIMITATIONS

None.

3.3.6.6.9.6 GENERAL_FIRST_ORDER_FILTER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P87-0)

This package exports operations to perform a first order filter operation on an input signal. The package also allows updating of the filter coefficients. The package initializes the coefficients and initial filter state as part of the elaboration of the instantiation.

The form of the filter operations is as follows:

```
X           := (c1 * Input_Signal) + (c2 * Prev_Input) +
               (c3 * Prev_Output)
Prev_Input  := Input_Signal;
Prev_Output := X;
```

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.6.1 REQUIREMENTS ALLOCATION

R109

3.3.6.6.9.6.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.6.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to filter.
Coefficient_Type	generic float	Defines data type for defining filter coefficients.

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Previous_Input_Signal	Signal_Type	Initial value of previous input signal.
Initial_Coefficient_1	Coefficient_Type	Initial value of first filter Coefficient.
Initial_Coefficient_2	Coefficient_Type	Initial value of second filter Coefficient.
Initial_Coefficient_3	Coefficient_Type	Initial value of third filter Coefficient.

Subprograms:

The following table summarizes the generic formal subprograms required by this part:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type
"/"	Function	Signal_Type / Coefficient_Type return Signal_Type

3.3.6.6.9.6.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Input_Signal	Signal_Type	Value of signal input to filter on previous pass.
Previous_Output_Signal	Signal_Type	Value of output signal from previous pass.
Coefficient_1	Coefficient_Type	Value of first coefficient
Coefficient_2	Coefficient_Type	Value of second coefficient
Coefficient_3	Coefficient_Type	Value of third coefficient

3.3.6.6.9.6.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.6.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body General_First_Order_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Coefficient_1         : Coefficient_Type := Initial_Coefficient_1;
   Coefficient_2         : Coefficient_Type := Initial_Coefficient_2;
   Coefficient_3         : Coefficient_Type := Initial_Coefficient_3;
   Previous_Input_Signal:  Signal_Type := Initial_Previous_Input_Signal;
   Previous_Output_Signal: Signal_Type :=
      ( Previous_Input_Signal * (Coefficient_1 + Coefficient_2) ) /
      ( 1.0 - Coefficient_3);

end General_First_Order_Filter;
```

3.3.6.6.9.6.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.6.8 LIMITATIONS

None.

3.3.6.6.9.6.9 LLCSC DESIGN

None.

3.3.6.6.9.6.10 UNIT DESIGN

3.3.6.6.9.6.10.1 UPDATE_COEFFICIENTS (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the values of the coefficients which were initialized at the elaboration of the instantiation.

3.3.6.6.9.6.10.1.1 REQUIREMENTS ALLOCATION

R109

3.3.6.6.9.6.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.6.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Coefficient_Type	generic float	Defines data type for incoming coefficients.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Coefficient_1	Coefficient_ Type	New value for first coefficient
New_Coefficient_2	"	New value for second coefficient
New_Coefficient_3	"	New value for third coefficient

Global parameters:

The following parameters used by this procedure are stored in the package body.

Name	Type	Description
Coefficient_1	Coefficient_Type	Current value of first coefficient
Coefficient_2	Coefficient_Type	Current value of secont coefficient
Coefficient_3	Coefficient_Type	Current value of third coefficient

3.3.6.6.9.6.10.1.4 LOCAL DATA

None.

3.3.6.6.9.6.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.6.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure UPDATE_COEFFICIENTS (NEW_COEFFICIENT_1 : in COEFFICIENT_TYPE;
                               NEW_COEFFICIENT_2 : in COEFFICIENT_TYPE;
                               NEW_COEFFICIENT_3 : in COEFFICIENT_TYPE) is
begin

   Coefficient_1 := New_Coefficient_1;
   Coefficient_2 := New_Coefficient_2;
   Coefficient_3 := New_Coefficient_3;

end Update_Coefficients;
```

3.3.6.6.9.6.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.6.10.1.8 LIMITATIONS

None.

3.3.6.6.9.6.10.2 FILTER (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the filter operation on the input signal. The form for this filter is as follows:

```
X           := (c1 * Input_Signal) + (c2 * Prev_Input) +
               (c3 * Prev_Output)
Prev_Input  := Input_Signal;
Prev_Output := X;
```

3.3.6.6.9.6.10.2.1 REQUIREMENTS ALLOCATION

R109

3.3.6.6.9.6.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.6.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals.
Coefficient_Type	generic float	Defines data type for incoming coefficients.

Subprograms:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	New value for input signal.

Global parameters:

Name	Type	Description
Previous_Input_Signal	Signal_Type	Value of input signal to previous pass.
Previous_Output_Signal	Signal_Type	Value of output signal from previous pass.
Coefficient_1	Coefficient_Type	Current value of first coefficient
Coefficient_2	Coefficient_Type	Current value of secont coefficient
Coefficient_3	Coefficient_Type	Current value of third coefficient

3.3.6.6.9.6.10.2.4 LOCAL DATA

None.

3.3.6.6.9.6.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.6.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function FILTER (SIGNAL : SIGNAL_TYPE) return SIGNAL_TYPE is

   Output_Signal :  Signal_Type;

begin

   Output_Signal := ( Signal * Coefficient_1) +
                    ( Previous_Input_Signal * Coefficient_2) +
                    ( Previous_Output_Signal * Coefficient_3);

   Previous_Input_Signal  := Signal;
   Previous_Output_Signal := Output_Signal;

   return Output_Signal;

end Filter;
```

3.3.6.6.9.6.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.6.10.2.8 LIMITATIONS

None.

3.3.6.6.9.6.10.3 REINITIALIZE (SUBPROGRAM BODY) UNIT DESIGN

This subprogram reinitializes the state of the filter by setting

```
Previous_Input_Signal  := Signal;
Previous_Output_Signal := ( Previous_Input_Signal *
                            (Coefficient_1 + Coefficient_2) ) /
                          ( 1.0 - Coefficient_3);
```

3.3.6.6.9.6.10.3.1 REQUIREMENTS ALLOCATION

R109

3.3.6.6.9.6.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.6.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming reinitialization signal.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Value used for reinitializing filter state

Global parameters:

The following parameters used by this procedure are stored in the package body.

Name	Type	Description
Previous_Input_Value	Signal_Type	Current value of previous input
Previous_Output_Value	Signal_Type	Current value of previous output

3.3.6.6.9.6.10.3.4 LOCAL DATA

None.

3.3.6.6.9.6.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.6.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Reinitialize (Signal : in Signal_Type) is

   begin

         Previous_Input_Signal := Signal;

         Previous_Output_Signal :=
                  ( Previous_Input_Signal * (Coefficient_1 + Coefficient_2) ) /
                  ( 1.0 - Coefficient_3);
   end Reinitialize;
```

3.3.6.6.9.6.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.6.10.3.8 LIMITATIONS

None.

3.3.6.6.9.7 TUSTIN_LAG_FILTER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P88-0)

This package exports operations to perform a Tustin Lag filter operation on an input signal. The package also allows updating of the filter coefficients. The package initializes the coefficients and initial filter state as part of the elaboration of the instantiation.

The form of the filter operations is as follows:

```
    X           := (c1 * (Input_Signal +  Prev_Input) +
                   (c2 * Prev_Output)
    Prev_Input  := Input_Signal;
    Prev_Output := X;
```

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.7.1 REQUIREMENTS ALLOCATION

R162

3.3.6.6.9.7.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.7.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to filter.
Coefficient_Type	generic float	Defines data type for defining filter coefficients.

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Previous_Input_Signal	Signal_Type	Initial value of previous input signal.
Initial_Coefficient_1	Coefficient_Type	Initial value of first filter Coefficient.
Initial_Coefficient_2	Coefficient_Type	Initial value of second filter Coefficient.

Subprograms:

The following table summarizes the generic formal subprograms required by this part:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type

3.3.6.6.9.7.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Input_Signal	Signal_Type	Value of signal input to filter on previous pass.
Previous_Output_Signal	Signal_Type	Value of output signal from previous pass. Initialized as per SRS 3.4.5.7.10.2
Coefficient_1	Coefficient_Type	Value of first coefficient
Coefficient_2	Coefficient_Type	Value of second coefficient

3.3.6.6.9.7.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.7.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body Tustin_Lag_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Coefficient_1         : Coefficient_Type := Initial_Coefficient_1;
   Coefficient_2         : Coefficient_Type := Initial_Coefficient_2;

   Previous_Input_Signal : Signal_Type := Initial_Previous_Input_Signal;

   Previous_Output_Signal: Signal_Type := Previous_Input_Signal *
                             ( (2.0 * Coefficient_1) / (1.0 - Coefficient_2) );

end Tustin_Lag_Filter;
```

3.3.6.6.9.7.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.7.8 LIMITATIONS

None.

3.3.6.6.9.7.9 LLCSC DESIGN

None.

3.3.6.6.9.7.10 UNIT DESIGN

3.3.6.6.9.7.10.1 UPDATE_COEFFICIENTS (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the values of the coefficients which were initialized at the elaboration of the instantiation.

3.3.6.6.9.7.10.1.1 REQUIREMENTS ALLOCATION

R162

3.3.6.6.9.7.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.7.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Coefficient_Type	generic float	Defines data type for incoming coefficients.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Coefficient_1	Coefficient_	New value for first coefficient
New_Coefficient_2	Type	New value for second coefficient

Global parameters:

Name	Type	Source	Description
Coefficient_1	Coefficient_Type	Package body	Current value of first coefficient
Coefficient_2	Coefficient_Type	Package body	Current value of second coefficient

3.3.6.6.9.7.10.1.4 LOCAL DATA

None.

3.3.6.6.9.7.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.7.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure UPDATE_COEFFICIENTS (NEW_COEFFICIENT_1 : in COEFFICIENT_TYPE;
                               NEW_COEFFICIENT_2 : in COEFFICIENT_TYPE) is

begin

   Coefficient_1 := New_Coefficient_1;
   Coefficient_2 := New_Coefficient_2;

end Update_Coefficients;
```

3.3.6.6.9.7.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.7.10.1.8 LIMITATIONS

None.

3.3.6.6.9.7.10.2 FILTER (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the filter operation on the input signal. The form for this filter is as follows:

```
X           := (c1 * (Input_Signal + Prev_Input) +
               (c2 * (Prev_Output)
Prev_Input  := Input_Signal;
Prev_Output := X;
```

3.3.6.6.9.7.10.2.1 REQUIREMENTS ALLOCATION

R162

3.3.6.6.9.7.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.7.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals.
Coefficient_Type	generic float	Defines data type for incoming coefficients.

Subprograms:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	New value for input signal.

Global parameters:

The following global data objects are stored in the package body:

Name	Type	Source	Description
Previous_Input_Signal	Signal_Type	Package Body	Value of input signal to previous pass.
Previous_Output_Signal	Signal_Type	Package Body	Value of output signal from previous pass.
Coefficient_1	Coefficient_Type	Package Body	Current value of first coefficient
Coefficient_2	Coefficient_Type	Package Body	Current value of second coefficient

3.3.6.6.9.7.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Output_Signal	Signal_Type	Output from filter operation.

3.3.6.6.9.7.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.7.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function FILTER (SIGNAL : SIGNAL_TYPE) return SIGNAL_TYPE is

   Output_Signal :  Signal_Type;

begin

   Output_Signal := ( (Signal + Previous_Input_Signal) * Coefficient_1) +
                    ( Previous_Output_Signal * Coefficient_2);

   Previous_Input_Signal  := Signal;
   Previous_Output_Signal := Output_Signal;

   return Output_Signal;

end Filter;
```

3.3.6.6.9.7.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.7.10.2.8 LIMITATIONS

None.

3.3.6.6.9.7.10.3 REINITIALIZE (SUBPROGRAM BODY) UNIT DESIGN

This subprogram reinitializes the state of the filter according to following processing:

```
Previous_Input_Signal  := Signal;

Previous_Output_Signal := Previous_Input_Signal *
                          ( (2.0 * Coefficient_1) / (1.0 - Coefficient_2) );
```

3.3.6.6.9.7.10.3.1 REQUIREMENTS ALLOCATION

R162

3.3.6.6.9.7.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.7.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming reinitialization signal

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Value for reinitializing filter state

Global parameters:

Name	Type	Source	Description
Previous_Input_Signal	Signal_Type	Package body	Current value of previous input signal
Previous_Output_Signal	Signal_Type	Package body	Current value of previous output signal

3.3.6.6.9.7.10.3.4 LOCAL DATA

None.

3.3.6.6.9.7.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.7.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Signal: in Signal_Type) is

begin

   Previous_Input_Signal  := Signal;

   Previous_Output_Signal :=
     Previous_Input_Signal * ( (2.0 * Coefficient_1) / (1.0 - Coefficient_2) );

end Reinitialize;
```

3.3.6.6.9.7.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.7.10.3.8 LIMITATIONS

None.

3.3.6.6.9.8 TUSTIN_LEAD_LAG_FILTER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P89-0)

This package exports operations to perform a Tustin Lead Lag filter operation on an input signal. The package also allows updating of the filter coefficients. The package initializes the coefficients and initial filter state as part of the elaboration of the instantiation.

The form of the filter operations is as follows:

```
X           := ( (Input_Signal - Prev_Input) * c1 ) +
               ( (Prev_Output - Prev_Input)  * c2 ) +
               Prev_Input
Prev_Input  := Input_Signal;
Prev_Output := X;
```

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.8.1 REQUIREMENTS ALLOCATION

R161

3.3.6.6.9.8.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.8.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to filter.
Coefficient_Type	generic float	Defines data type for defining filter coefficients.

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Previous_Input_Signal	Signal_Type	Initial value of previous input signal.
Initial_Coefficient_1	Coefficient_Type	Initial value of first filter Coefficient.
Initial_Coefficient_2	Coefficient_Type	Initial value of second filter Coefficient.

Subprograms:

The following table summarizes the generic formal subprograms required by this part:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type

3.3.6.6.9.8.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Input_Signal	Signal_Type	Value of signal input to filter on previous pass.
Previous_Output_Signal	Signal_Type	Value of output signal from previous pass.
Coefficient_1	Coefficient_Type	Value of first coefficient
Coefficient_2	Coefficient_Type	Value of second coefficient

3.3.6.6.9.8.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.8.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body Tustin_Lead_Lag_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Previous_Input_Signal : Signal_Type := Initial_Previous_Input_Signal;
   Previous_Output_Signal: Signal_Type := Initial_Previous_Input_Signal;

   Coefficient_1         : Coefficient_Type := Initial_Coefficient_1;
   Coefficient_2         : Coefficient_Type := Initial_Coefficient_2;

end Tustin_Lead_Lag_Filter;
```

3.3.6.6.9.8.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.8.8 LIMITATIONS

None.

3.3.6.6.9.8.9 LLCSC DESIGN

None.

3.3.6.6.9.8.10 UNIT DESIGN

3.3.6.6.9.8.10.1 UPDATE_COEFFICIENTS (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the values of the coefficients which were initialized at the elaboration of the instantiation.

3.3.6.6.9.8.10.1.1 REQUIREMENTS ALLOCATION

R161

3.3.6.6.9.8.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.8.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Coefficient_Type	generic float	Defines data type for incoming coefficients.

Name	Type	Description
New_Coefficient_1	Coefficient_	New value for first coefficient
New_Coefficient_2	Type	New value for second coefficient

Global parameters:

Name	Type	Source	Description
Coefficient_1	Coefficient_Type	Package Body	Current value of first coefficient
Coefficient_2	Coefficient_Type	Package Body	Current value of second coefficient

3.3.6.6.9.8.10.1.4 LOCAL DATA

None.

3.3.6.6.9.8.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.8.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure UPDATE_COEFFICIENTS (NEW_COEFFICIENT_1 : in COEFFICIENT_TYPE;
                               NEW_COEFFICIENT_2 : in COEFFICIENT_TYPE) is
begin

   Coefficient_1 := New_Coefficient_1;
   Coefficient_2 := New_Coefficient_2;

end Update_Coefficients;
```

3.3.6.6.9.8.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.8.10.1.8 LIMITATIONS

None.

3.3.6.6.9.8.10.2 FILTER (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the filter operation on the input signal. The form for this filter is as follows:

```
X           := (c1 * (Input_Signal - Prev_Input) +
               (c2 * (Prev_Output - Prev_Input) +
               Prev_Input)
Prev_Input  := Input_Signal;
Prev_Output := X;
```

3.3.6.6.9.8.10.2.1 REQUIREMENTS ALLOCATION

R161

3.3.6.6.9.8.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.8.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals.
Coefficient_Type	generic float	Defines data type for incoming coefficients.

Subprograms:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	New value for input signal.

Global parameters:

Name	Type	Source	Description
Previous_Input_Signal	Signal_Type	Package Body	Value of input signal to previous pass.
Previous_Output_Signal	Signal_Type	Package Body	Value of output signal from previous pass.
Coefficient_1	Coefficient_Type	Package Body	Current value of first coefficient
Coefficient_2	Coefficient_Type	Package Body	Current value of second coefficient

3.3.6.6.9.8.10.2.4 LOCAL DATA

None.

3.3.6.6.9.8.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.8.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function FILTER (SIGNAL : SIGNAL_TYPE) return SIGNAL_TYPE is

   Output_Signal :  Signal_Type;

begin

   Output_Signal := ( (Signal - Previous_Input_Signal) * Coefficient_1) +
                    ( (Previous_Output_Signal - Previous_Input_Signal) *
                       Coefficient_2) +
                    Previous_Input_Signal;

   Previous_Input_Signal  := Signal;
   Previous_Output_Signal := Output_Signal;

   return Output_Signal;

end Filter;
```

3.3.6.6.9.8.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.8.10.2.8 LIMITATIONS

None.

3.3.6.6.9.8.10.3 REINITIALIZE (SUBPROGRAM BODY) UNIT DESIGN

This subprogram reinitializes the state of the filter as follows:

```
Previous_Input_Signal  := Signal;
Previous_Output_Signal := Signal;
```

3.3.6.6.9.8.10.3.1 REQUIREMENTS ALLOCATION

R161

3.3.6.6.9.8.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.8.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming reinitialization signal

Name	Type	Description
Signal	Signal_Type	Value for reinitializing filter state

Global parameters:

Name	Type	Source	Description
Previous_Input_Signal	Signal_Type	Package body	Current value of previous input signal
Previous_Output_Signal	Signal_Type	Package body	Current value of previous output signal

3.3.6.6.9.8.10.3.4 LOCAL DATA

None.

3.3.6.6.9.8.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.8.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Signal : in Signal_Type) is

begin

   Previous_Input_Signal  := Signal;
   Previous_Output_Signal := Previous_Input_Signal;

end Reinitialize;
```

3.3.6.6.9.8.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.8.10.3.8 LIMITATIONS

None.

3.3.6.6.9.9 SECOND_ORDER_FILTER (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P90-0)

This package exports operations to perform a Second Order filter operation on an input signal. The package also allows updating of the filter coefficients. The package initializes the coefficients and initial filter state as part of the elaboration of the instantiation.

The form of the filter operations is as follows:

```
X               := (c1 * (Input_Signal - 2nd_Prev_Input) +
                   (c2 * (Prev_Input - Prev_Output) ) -
                   (c3 * 2nd_Prev_Output);
2nd_Prev_Input  := Prev_Input;
Prev_Input      := Input_Signal;
2nd_Prev_Output := Prev_Output;
Prev_Output     := X;
```

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.9.1 REQUIREMENTS ALLOCATION

R110

3.3.6.6.9.9.2 LOCAL ENTITIES DESIGN

This package contains a sequence of statements at the end of the package body which are executed when this part is elaborated. This code initializes the previous input signal and coefficients. It also sets the value for the previous output signal. It calls Redefine_Coefficients procedure from this package to initialize the values of the coefficients.

3.3.6.6.9.9.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals to filter.
Coefficient_Type	generic float	Defines data type for defining filter coefficients.

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Previous_Input_Signal	Signal_Type	Initial value of previous input signal.
Initial_Coefficient_Defining_Parameter_1	Coefficient_Type	Initial value of first defining parameter of coefficients.
Initial_Coefficient_Defining_Parameter_2	Coefficient_Type	Initial value of second defining parameter of coefficients.

Subprograms:

The following table summarizes the generic formal subprograms required by this part:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type

3.3.6.6.9.9.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Input_Signal	Signal_Type	Value of signal input to filter on previous pass.
Second_Previous_Input_Signal	Signal_Type	Value of input signal on pass before previous.
Previous_Output_Signal	Signal_Type	Value of signal output from filter on previous pass.
Second_Previous_Output_Signal	Signal_Type	Value of output signal from filter on pass before last.
Coefficient_1	Coefficient_Type	Value of first coefficient
Coefficient_2	Coefficient_Type	Value of second coefficient
Coefficient_3	Coefficient_Type	Value of third coefficient

3.3.6.6.9.9.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.9.6 PROCESSING

The following describes the processing performed by this part:

```
separate (SIGNAL_PROCESSING)
package body Second_Order_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Previous_Input_Signal:          Signal_Type;
   Second_Previous_Input_Signal:   Signal_Type;

   Previous_Output_Signal:         Signal_Type;
   Second_Previous_Output_Signal:  Signal_Type;

   Coefficient_1          : Coefficient_Type;
   Coefficient_2          : Coefficient_Type;
   Coefficient_3          : Coefficient_Type;


-----------------------------------
--begin processing for package body
-----------------------------------

begin

   Previous_Input_Signal         := Initial_Previous_Input_Signal;
```

```
   Second_Previous_Input_Signal := Initial_Previous_Input_Signal;

   Previous_Output_Signal        := Initial_Previous_Input_Signal;
   Second_Previous_Output_Signal := Initial_Previous_Input_Signal;

   Redefine_Coefficients (New_Coefficient_Defining_Parameter_1 =>
                             Initial_Coefficient_Defining_Parameter_1,
                          New_Coefficient_Defining_Parameter_2 =>
                             Initial_Coefficient_Defining_Parameter_2);

end Second_Order_Filter;
```

3.3.6.6.9.9.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.9.8 LIMITATIONS

None.

3.3.6.6.9.9.9 LLCSC DESIGN

None.

3.3.6.6.9.9.10 UNIT DESIGN

3.3.6.6.9.9.10.1 REDEFINE_COEFFICIENTS (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the values of the coefficients which were initialized at the elaboration of the instantiation.

3.3.6.6.9.9.10.1.1 REQUIREMENTS ALLOCATION

R111

3.3.6.6.9.9.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.9.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Coefficient_Type	generic float	Defines data type for incoming coefficients.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Coefficient_Defining_Parameter_1	Coefficient_Type	New value for first defining paramter of coefficients
New_Coefficient_Defining_Parameter_2	Coefficient_Type	New value for second defining paramter of coefficients

Global parameters:

Name	Type	Description
Coefficient_1	Coefficient_Type	Current value of first coefficient
Coefficient_2	Coefficient_Type	Current value of second coefficient
Coefficient_3	Coefficient_Type	Current value of third coefficient

3.3.6.6.9.9.10.1.4 LOCAL DATA

None.

3.3.6.6.9.9.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.9.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Redefine_COEFFICIENTS
            (NEW_COEFFICIENT_Defining_Parameter_1 : in COEFFICIENT_TYPE;
             NEW_COEFFICIENT_Defining_Parameter_2 : in COEFFICIENT_TYPE) is

      Coefficient_Defining_Ratio:  Coefficient_Type;

   begin

      Coefficient_Defining_Ratio := 1.0 /
                                    (1.0 + NEW_COEFFICIENT_Defining_Parameter_1 +
```

```
                                    NEW_COEFFICIENT_Defining_Parameter_2);

      Coefficient_1 := (1.0 + New_Coefficient_Defining_Parameter_1) *
                       Coefficient_Defining_Ratio;

      Coefficient_2 := (2.0 * (1.0 - New_Coefficient_Defining_Parameter_2) ) *
                       Coefficient_Defining_Ratio;

      Coefficient_3 := (1.0 - (NEW_COEFFICIENT_Defining_Parameter_1 +
                               NEW_COEFFICIENT_Defining_Parameter_2)) *
                        Coefficient_Defining_Ratio;

   end Redefine_Coefficients;
```

3.3.6.6.9.9.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.9.10.1.8 LIMITATIONS

None.

3.3.6.6.9.9.10.2 FILTER (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the filter operation on the input signal. The form for this filter is as follows:

```
    X              := (c1 * (Input_Signal - 2nd_Prev_Input) +
                       (c2 * (Prev_Input - Prev_Output) ) -
                       (c3 * 2nd_Prev_Output);
    2nd_Prev_Input := Prev_Input;
    Prev_Input     := Input_Signal;
    2nd_Prev_Output := Prev_Output;
    Prev_Output    := X;
```

3.3.6.6.9.9.10.2.1 REQUIREMENTS ALLOCATION

R110

3.3.6.6.9.9.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.9.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming signals.
Coefficient_Type	generic float	Defines data type for incoming coefficients.

Subprograms:

Name	Type	Description
"*"	Function	Signal_Type * Coefficient_Type return Signal_Type

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	New value for input signal.

Global parameters:

Name	Type	Description
Previous_Input_Signal	Signal_Type	Value of input signal to previous pass.
Second_Previous_Input_Signal	Signal_Type	Value of input signal on pass before last.
Previous_Output_Signal	Signal_Type	Value of output signal from previous pass.
Second_Previous_Output_Signal	Signal_Type	Value of output signal from pass before last.
Coefficient_1	Coefficient_Type	Current value of first coefficient
Coefficient_2	Coefficient_Type	Current value of second coefficient
Coefficient_3	Coefficient_Type	Current value of third coefficient

3.3.6.6.9.9.10.2.4 LOCAL DATA

None.

3.3.6.6.9.9.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.9.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function FILTER (SIGNAL : SIGNAL_TYPE) return SIGNAL_TYPE is

   Output_Signal :  Signal_Type := 0.0;

begin

   Output_Signal := ( (Signal - Second_Previous_Input_Signal) *
                                                  Coefficient_1) +
                    ( (Previous_Input_Signal - Previous_Output_Signal) *
                                                  Coefficient_2) -
                    ( Second_Previous_Output_Signal * Coefficient_3);

   Second_Previous_Input_Signal  := Previous_Input_Signal;
   Previous_Input_Signal         := Signal;
   Second_Previous_Output_Signal := Previous_Output_Signal;
   Previous_Output_Signal        := Output_Signal;

   return Output_Signal;

end Filter;
```

3.3.6.6.9.9.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.9.10.2.8 LIMITATIONS

None.

3.3.6.6.9.9.10.3 REINITIALIZE (SUBPROGRAM BODY) UNIT DESIGN

This subprogram reinitializes the state of the filter by the following processing:

```
Previous_Input_Signal         := Initial_Previous_Input_Signal;
Second_Previous_Input_Signal  := Initial_Previous_Input_Signal;

Previous_Output_Signal        := Initial_Previous_Input_Signal;
Second_Previous_Output_Signal := Initial_Previous_Input_Signal;
```

3.3.6.6.9.9.10.3.1 REQUIREMENTS ALLOCATION

R111

3.3.6.6.9.9.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.9.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signal_Type	generic float	Defines data type for incoming reinitialization signal

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signal_Type	Value for reinitializing filter state

Global parameters:

Name	Type	Source	Description
Previous_Input_Signal	Signal_Type	Package body	Current value of previous input signal
Previous_Output_Signal	Signal_Type	Package body	Current value of previous output signal

3.3.6.6.9.9.10.3.4 LOCAL DATA

None.

3.3.6.6.9.9.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.9.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Signal : in Signal_Type) is

begin

   Previous_Input_Signal        := Signal;
   Second_Previous_Input_Signal := Signal;

   Previous_Output_Signal        := Signal;
   Second_Previous_Output_Signal := Signal;

end Reinitialize;
```

3.3.6.6.9.9.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.9.10.3.8 LIMITATIONS

None.

3.3.6.6.9.10 TUSTIN_INTEGRATOR_WITH_LIMIT (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P91-0)

This package exports operations to perform a Tustin Integrator operation on an input signal. The form of the integration will be: Y = Y_prev + (X + X_prev) * gain * 0.5 * integration_time_interval.

The package will set a flag when the result of the integration reaches an established limit and will not perform further integration if the next integration will not move the current state below the limit. The package also allows updating of the gain on the input signal and limit and reading of the flag setting. The package initializes the gain, limit and initial integrator state as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.10.1 REQUIREMENTS ALLOCATION

R203

3.3.6.6.9.10.2 LOCAL ENTITIES DESIGN

Name	Type	Description
Local_Integrate	Package	Instantiated from General Math Parts to perform integration
Local_Limit	Package	Instantiated from Absolute_ Limit_With_Flag to perform limit and clamp operation

3.3.6.6.9.10.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signals	generic float	Defines data type for incoming signals to integrator.
States	generic float	Defines data type for signals output from integrator
Gained_Signals	generic float	Defines data type for incoming signal after gain applied
Gains	generic float	Defines data type for gains

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Signal_ Level	Signals	Initial value of input signal for first pass.
Initial_Output_ Level	States	Initial values of output signal after first pass.
Initial_Signal_ Limit	States	Initial value of limit on integrator output.
Initial_Time_ Inteval	Times	Initial value of time interval for integration
Initial_ Tustin_Gain	Gains	Initial value of gain used in Tustin integration

Subprograms:

The following table summarizes the generic formal subprograms required by this part:

Name	Type	Description
"*"	Function	Signals * Gains return Gained_Signals
"*"	Function	Gained_Signals * States return States

Global parameters:

Data types:

Name	Source	Type	Description
Limit_Relations	Package Spec.	Enumeration	Establishes the relationship between a signal and the limit imposed on that signal

3.3.6.6.9.10.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Gained_Input	Signals	Value of signal input to filter on previous pass with gain applied.
Previous_Output_Signal	States	Value of output signal from previous pass.
Signal_Gain	Gains	Value of gain applied to input signal.

3.3.6.6.9.10.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.10.6 PROCESSING

The following describes the processing performed by this part:

```
with General_Purpose_Math;
separate (SIGNAL_PROCESSING)

package body Tustin_Integrator_With_Limit is

------------------------
-- -- local entities   -
```

```
-----------------------

   package Local_Limit is new Signal_Processing.Absolute_Limiter_With_Flag
             (Signal_Type           => States,
              Initial_Absolute_Limit => Initial_Signal_Limit);

------------------------
-- -- Local Data Object-
------------------------

   Signal_Gain              : Gains         := Initial_Tustin_Gain;

   Previous_Gained_Input    : Gained_Signals :=
                                 Initial_Signal_Level * Signal_Gain;

   Previous_Output_Signal   : States        :=
                                 Local_Limit.Limit (Initial_State);

------------------------
-- -- local entities   -
------------------------

   package Local_Integrator is new General_Purpose_Math.Integrator
             (Dependent_Type           => States,
              Independent_Type         => Gained_Signals,
              Time_Interval            => States,
              Initial_Dependent_Value  => Previous_Output_Signal,
              Initial_Independent_Value => Previous_Gained_Input,
              Default_Delta_Time       => 1.0,
              "*"                      => "*");

end Tustin_Integrator_With_Limit;
```

3.3.6.6.9.10.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:
1. General_Purpose_Math

UTILIZATION OF EXTERNAL ELEMENTS:

Packages:

The following table summarizes the external packages required by this part:

Name	Type	Source	Description
Integrator	Generic Package	General Purpose Math	Performs Integrate operations.

UTILIZATION OF OTHER TLC ELEMENTS:

The following tables describe the elements used by this part but defined in one or more units from the top level component.

Packages:

The following table summarizes the TLC packages required by this part:

Name	Type	Description
Absolute_Limiter_with_Flag	Generic Package	Performs limit operation on integrator output.

3.3.6.6.9.10.8 LIMITATIONS

None.

3.3.6.6.9.10.9 LLCSC DESIGN

None.

3.3.6.6.9.10.10 UNIT DESIGN

3.3.6.6.9.10.10.1 UPDATE_LIMIT (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the limit on the integrator output as initialized at the elaboration of the instantiation. It does this by calling the Update_Limit function of the Local_Limit package.

3.3.6.6.9.10.10.1.1 REQUIREMENTS ALLOCATION

R203

3.3.6.6.9.10.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.10.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
States	generic float	Defines data type for limit on output signal.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Absolute_Limit	Output_Type	New value for absolute limit

3.3.6.6.9.10.10.1.4 LOCAL DATA

None.

3.3.6.6.9.10.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.10.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limit (New_Absolute_Limit: in States) is
begin
   Local_Limit.Update_Limit (New_Absolute_Limit => New_Absolute_Limit);
end Update_Limit;
```

3.3.6.6.9.10.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.10.10.1.8 LIMITATIONS

None.

3.3.6.6.9.10.10.2 UPDATE_GAIN (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the gain to applied applied to the input signal prior to the integration operation.

3.3.6.6.9.10.10.2.1 REQUIREMENTS ALLOCATION

R203

3.3.6.6.9.10.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.10.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Gains	generic float	Defines data type for gains to input signal.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Gain	Gains	New value for gain

Global parameters:

Name	Type	Source	Description
Signal_Gain	Gains	Pack. body	Current value of gain

3.3.6.6.9.10.10.2.4 LOCAL DATA

None.

3.3.6.6.9.10.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.10.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Gain (New_Gain: in Gains) is
begin
```

```
      Signal_Gain := New_Gain;

   end Update_Gain;
```

3.3.6.6.9.10.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.10.10.2.8 LIMITATIONS

None.

3.3.6.6.9.10.10.3 INTEGRATE (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the integrator operation on the input signal. The form of this integration is:

```
    If Prev_Output within limit then
        Output      := Prev_Output +
            (Input_Signal + Prev_Input) * Gain * Time_Interval/2;
        Limit Output and set flag if appropriate
    Elsif Prev_Output at positive limit and
      Integration forces output below limit then
        Output      := Prev_Output +
            (Input_Signal + Prev_Input) * Gain * Time_Interval/2;
    Elsif Prev_Output at negative limit and
      Integration forces output above limit then
        Output      := Prev_Output +
            (Input_Signal + Prev_Input) * Gain * Time_Interval/2;
        Prev_Input  := Input_Signal * Gain;
        Prev_Output := Output;
```

3.3.6.6.9.10.10.3.1 REQUIREMENTS ALLOCATION

R203

3.3.6.6.9.10.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.10.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signals	generic float	Defines data type for incoming signals.
States	generic float	Defines data type for output from integrator.
Gained_Signals	generic float	Defines data type for intermediate result from applying gain to input signal

Subprograms:

Name	Type	Description
"*"	Function	Signals * Coefficient_Type return Signal_Type

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signals	New value for input signal.

3.3.6.6.9.10.10.3.4 LOCAL DATA

None.

3.3.6.6.9.10.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.10.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   function Integrate (Signal : Signals) return States is

      New_Gained_Signal : Gained_Signals := Signal * Signal_Gain;
      New_Gained_Total  : Gained_Signals :=
                             New_Gained_Signal + Previous_Gained_Input;
      Output_Signal     : States;

   begin

-------------------------------------------------------------------
--    if    within limit or
```

```
--          (beyond limit and new input brings output within limit) then
--       integrate new output
-------------------------------------------------------------------------

      if
         (Local_Limit.Limit_Flag_Setting = Within_Limit or else
         (Local_Limit.Limit_Flag_Setting = At_Positive_Limit and then
             New_Gained_Total < 0.0) or else
         (Local_Limit.Limit_Flag_Setting = At_Negative_Limit and then
             New_Gained_Total > 0.0) )
                                        then

         Output_Signal := Local_Integrator.Integrate (New_Gained_Signal);
         Output_Signal := Local_Limit.Limit (Output_Signal);

         if Local_Limit.Limit_Flag_Setting /= Within_Limit then

             Reset (Integrator_State => Output_Signal,
                    Signal           => Signal);

         end if;

         Previous_Output_Signal := Output_Signal;

      else

         Output_Signal := Previous_Output_Signal;
         Local_Integrator.Update
                           (Current_Independent_Value => New_Gained_Signal);

      end if;


      Previous_Gained_Input  := New_Gained_Signal;

      return Output_Signal;

   end Integrate;
```

3.3.6.6.9.10.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.10.10.3.8 LIMITATIONS

None.

3.3.6.6.9.10.10.4 RESET (INTEGRATOR_STATE) (SUBPROGRAM BODY) UNIT DESIGN

This procedure calls the Local_Integrator.Reinitialize procedure to reinitialize to integrator state.

3.3.6.6.9.10.10.4.1 REQUIREMENTS ALLOCATION

R203

3.3.6.6.9.10.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.10.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Integrator_State	States	In	Unlimited value of new integrator state
Signal	Signals	In	New value of previous input signal

3.3.6.6.9.10.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Description
Limited_State	States	State after local limit is applied

3.3.6.6.9.10.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.10.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reset (Integrator_State : in States;
                 Signal           : in Signals) is

   Limited_State: States := Local_Limit.Limit (Integrator_State);

begin

   Previous_Gained_Input := Signal * Signal_Gain;
```

```
      Local_Integrator.Reinitialize
                        (Initial_Dependent_Value   => Integrator_State,
                         Initial_Independent_Value => Previous_Gained_Input);

   end Reset;
```

3.3.6.6.9.10.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined elsewhere in the parent top level component:

Name	Type	Source	Description
Reinitialize	Procedure	Local_Integrator	Reinitializatiof of integrator state

Data types:

The following table summarizes the TLC types required by this part:

Name	Type	Source	Description
States	Generic float	Package Spec	State of integrator
Signals	Generic float	Package Spec	Represents type of input to integrator

3.3.6.6.9.10.10.4.8 LIMITATIONS

None.

3.3.6.6.9.10.10.5 LIMIT_FLAG_SETTING (SUBPROGRAM BODY) UNIT DESIGN

This procedure calls the Local_Limit.Limit_Flag_Setting function to return the relation of the current output signal to the limit.

3.3.6.6.9.10.10.5.1 REQUIREMENTS ALLOCATION

R203

3.3.6.6.9.10.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.10.10.5.3 INPUT/OUTPUT

None.

3.3.6.6.9.10.10.5.4 LOCAL DATA

None.

3.3.6.6.9.10.10.5.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.10.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit_Flag_Setting return Limit_Relations is
begin
   return (Local_Limit.Limit_Flag_Setting);
end Limit_Flag_Setting;
```

3.3.6.6.9.10.10.5.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.10.10.5.8 LIMITATIONS

None.

3.3.6.6.9.11 TUSTIN_INTEGRATOR_WITH_ASYMMETRIC_LIMIT (PACKAGE BODY) PACKAGE DESIGN (CATALOG #P1054-0)

This package exports operations to perform a Tustin Integrator operation on an input signal. The form of the integration will be: Y = Y_prev + (X + X_prev) * gain * 0.5 * integration_time_interval.

The package will set a flag when the result of the integration reaches an established limit and will not perform further integration if the next integration will not move the current state below the limit. The package also allows updating of the gain on the input signal and limit and reading of the flag setting. The package initializes the gain, limits and initial integrator state as part of the elaboration of the instantiation.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.6.9.11.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.6.9.11.2 LOCAL ENTITIES DESIGN

Name	Type	Description
Local_Integrate	Package	Instantiated from General Math Parts to perform integration
Local_Limit	Package	Instantiated from Upper_Lower_ Limiter to perform limiting

3.3.6.6.9.11.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signals	generic float	Defines data type for incoming signals to integrator.
States	generic float	Defines data type for signals output from integrator
Gained_Signals	generic float	Defines data type for incoming signal after gain applied
Gains	generic float	Defines data type for gains

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Signal_Level	Signals	Initial value of input signal for first pass.
Initial_Output_Level	States	Initial values of output signal after first pass.
Initial_Signal_Lower_Limit	States	Initial value of lower limit on integrator output.
Initial_Signal_Upper_Limit	States	Initial value of upper limit on integrator output.
Initial_Tustin_Gain	Gains	Initial value of gain used in Tustin integration

Subprograms:

The following table summarizes the generic formal subprograms required by this part:

Name	Type	Description
"*"	Function	Signals * Gains return Gained_Signals
"*"	Function	Gained_Signals * States return States

Global parameters:

Data types:

Name	Source	Type	Description
Limit_Relations	Package Spec.	Enumeration	Establishes the relationship between a signal and the limit imposed on that signal

3.3.6.6.9.11.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Gained_Input	Signals	Value of signal input to filter on previous pass with gain applied.
Previous_Output_Signal	States	Value of output signal from previous pass.
Signal_Gain	Gains	Value of gain applied to input signal.
Local_Limit_Flag_Setting	Limit_Relations	Current state of the limited integrator state
Local_Lower_Limit	States	Local copy of the limiter's lower limit
Local_Upper_Limit	States	Local copy of the limiter's upper limit

3.3.6.6.9.11.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.11.6 PROCESSING

The following describes the processing performed by this part:

```
with General_Purpose_Math;
separate (SIGNAL_PROCESSING)

package body Tustin_Integrator_With_Asymmetric_Limit is

------------------------
-- -- local entities   -
------------------------

   package Local_Limit is new Signal_Processing.Upper_Lower_Limiter
             (Signal_Type         => States,
              Initial_Lower_Limit => Initial_Signal_Lower_Limit,
              Initial_Upper_Limit => Initial_Signal_Upper_Limit);

-------------------------
-- -- Local Data Objects-
-------------------------

   Signal_Gain              : Gains          := Initial_Tustin_Gain;

   Previous_Gained_Input    : Gained_Signals :=
                                 Initial_Signal_Level * Signal_Gain;

   Previous_Output_Signal   : States         :=
                                 Local_Limit.Limit (Initial_State);

   Local_Limit_Flag_Setting : Limit_Relations := Within_Limit;
```

```
   Local_Lower_Limit        : States := Initial_Signal_Lower_Limit;
   Local_Upper_Limit        : States := Initial_Signal_Upper_Limit;


------------------------
-- -- local entities   -
------------------------

   package Local_Integrator is new General_Purpose_Math.Integrator
             (Dependent_Type           => States,
              Independent_Type         => Gained_Signals,
              Time_Interval            => States,
              Initial_Dependent_Value  => Previous_Output_Signal,
              Initial_Independent_Value => Previous_Gained_Input,
              Default_Delta_Time       => 1.0);

end Tustin_Integrator_With_Asymmetric_Limit;
```

3.3.6.6.9.11.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:
1. General_Purpose_Math

UTILIZATION OF EXTERNAL ELEMENTS:

Packages:

The following table summarizes the external packages required by this part:

Name	Type	Source	Description
Integrator	Generic Package	General Purpose Math	Performs Integrate operations.

UTILIZATION OF OTHER TLC ELEMENTS:

The following tables describe the elements used by this part but defined in one or more units from the top level component.

Packages: The following table summarizes the TLC packages required by this part:

Name	Type	Description
Upper_Lower_Limiter	Generic Package	Performs limit operation on integrator output.

3.3.6.6.9.11.8 LIMITATIONS

None.

3.3.6.6.9.11.9 LLCSC DESIGN

None.

3.3.6.6.9.11.10 UNIT DESIGN

3.3.6.6.9.11.10.1 UPDATE_LIMITS (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the limits on the integrator output as initialized at the elaboration of the instantiation. It does this by calling the Update_Limit function of the Local_Limit package.

3.3.6.6.9.11.10.1.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.6.9.11.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.11.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
States	generic float	Defines data type for limit on output signal.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Upper_Limit	Output_Type	New value for upper limit
New_Lower_Limit	Output_Type	New value for lower limit

3.3.6.6.9.11.10.1.4 LOCAL DATA

None.

3.3.6.6.9.11.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.11.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update_Limits (New_Lower_Limit: in States;
                         New_Upper_Limit: in States) is

begin

   Local_Limit.Update_Limits (New_Lower_Limit => New_Lower_Limit,
                              New_Upper_Limit => New_Upper_Limit);

   Local_Lower_Limit := New_Lower_Limit;
   Local_Upper_Limit := New_Upper_Limit;

end Update_Limits;
```

3.3.6.6.9.11.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.11.10.1.8 LIMITATIONS

None.

3.3.6.6.9.11.10.2 UPDATE_GAIN (SUBPROGRAM BODY) UNIT DESIGN

This subprogram updates the gain to applied applied to the input signal prior to the integration operation.

3.3.6.6.9.11.10.2.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.6.9.11.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.11.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Gains	generic float	Defines data type for gains to input signal.

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
New_Gain	Gains	New value for gain

Global parameters:

Name	Type	Source	Description
Signal_Gain	Gains	Pack. body	Current value of gain

3.3.6.6.9.11.10.2.4 LOCAL DATA

None.

3.3.6.6.9.11.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.11.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure UPDATE_Gain (New_Gain: in Gains) is
begin
   Signal_Gain := New_Gain;
end Update_Gain;
```

3.3.6.6.9.11.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.11.10.2.8 LIMITATIONS

None.

3.3.6.6.9.11.10.3 INTEGRATE (SUBPROGRAM BODY) UNIT DESIGN

This subprogram performs the integrator operation on the input signal. The form of this integration is:

```
If Prev_Output within limit then
    Output      := Prev_Output +
        (Input_Signal + Prev_Input) * Gain * Time_Interval/2;
    Limit Output and set flag if appropriate
Elsif Prev_Output at positive limit and
  Integration forces output below limit then
    Output      := Prev_Output +
        (Input_Signal + Prev_Input) * Gain * Time_Interval/2;
Elsif Prev_Output at negative limit and
  Integration forces output above limit then
    Output      := Prev_Output +
        (Input_Signal + Prev_Input) * Gain * Time_Interval/2;
    Prev_Input  := Input_Signal * Gain;
    Prev_Output := Output;
```

3.3.6.6.9.11.10.3.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.6.9.11.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.11.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Signals	generic float	Defines data type for incoming signals.
States	generic float	Defines data type for output from integrator.
Gained_Signals	generic float	Defines data type for intermediate result from applying gain to input signal

Subprograms:

Name	Type	Description
"*"	Function	Signals * Coefficient_Type return Signal_Type

FORMAL PARAMETERS:

The following table summarizes the formal parameters to this part:

Name	Type	Description
Signal	Signals	New value for input signal.

3.3.6.6.9.11.10.3.4 LOCAL DATA

None.

3.3.6.6.9.11.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.11.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   function Integrate (Signal : Signals) return States is

      New_Gained_Signal : Gained_Signals := Signal * Signal_Gain;
      New_Gained_Total  : Gained_Signals :=
                             New_Gained_Signal + Previous_Gained_Input;
      Output_Signal     : States;

   begin

--    ------------------------------------------------------------------
--    --if    within limit or
```

```
--    --      (beyond limit and new input brings output within limit) then
--    --   integrate new output
--    ------------------------------------------------------------------------

      if
         (Local_Limit_Flag_Setting = Within_Limit or else
         (Local_Limit_Flag_Setting = At_Positive_Limit and then
             New_Gained_Total < 0.0) or else
         (Local_Limit_Flag_Setting = At_Negative_Limit and then
             New_Gained_Total > 0.0) )
                                        then

         Output_Signal := Local_Integrator.Integrate (New_Gained_Signal);
         Output_Signal := Local_Limit.Limit (Output_Signal);

         if Output_Signal >= Local_Upper_Limit then
            Local_Limit_Flag_Setting := At_Positive_Limit;
         elsif Output_Signal <= Local_Lower_Limit then
            Local_Limit_Flag_Setting := At_Negative_Limit;
         else
            Local_Limit_Flag_Setting := Within_Limit;
         end if;

         if Local_Limit_Flag_Setting /= Within_Limit then

             Reset (Integrator_State => Output_Signal,
                    Signal           => Signal);

         end if;

         Previous_Output_Signal := Output_Signal;

      else

         Output_Signal := Previous_Output_Signal;
         Local_Integrator.Update
                           (Current_Independent_Value => New_Gained_Signal);

      end if;


      Previous_Gained_Input  := New_Gained_Signal;

      return Output_Signal;

   end Integrate;
```

3.3.6.6.9.11.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.11.10.3.8 LIMITATIONS

None.

3.3.6.6.9.11.10.4 RESET (SUBPROGRAM BODY) UNIT DESIGN

This procedure calls the Local_Integrator.Reinitialize procedure to reinitialize the integrator state.

3.3.6.6.9.11.10.4.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.6.9.11.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.11.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Integrator_State	States	In	Unlimited value of new integrator state
Signal	Signals	In	New value of previous input signal

3.3.6.6.9.11.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Description
Limited_State	States	State after local limit is applied

3.3.6.6.9.11.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.11.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Reset (Integrator_State : in States;
                    Signal           : in Signals) is

      Limited_State: States := Local_Limit.Limit (Integrator_State);

   begin

      Previous_Gained_Input := Signal * Signal_Gain;

      Local_Integrator.Reinitialize
                         (Initial_Dependent_Value   => Integrator_State,
                          Initial_Independent_Value => Previous_Gained_Input);

   end Reset;
```

3.3.6.6.9.11.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined elsewhere in the parent top level component:

Name	Type	Source	Description
Reinitialize	Procedure	Local_Integrator	Reinitializatiof of integrator state

Data types:

The following table summarizes the TLC types required by this part:

Name	Type	Source	Description
States	Generic float	Package Spec	State of integrator
Signals	Generic float	Package Spec	Represents type of input to integrator

3.3.6.6.9.11.10.4.8 LIMITATIONS

None.

3.3.6.6.9.11.10.5 LIMIT_FLAG_SETTING (SUBPROGRAM BODY) UNIT DESIGN

This procedure queries the flag setting and returns the relation of the current output signal to the limit.

3.3.6.6.9.11.10.5.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.6.9.11.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.6.6.9.11.10.5.3 INPUT/OUTPUT

None.

3.3.6.6.9.11.10.5.4 LOCAL DATA

None.

3.3.6.6.9.11.10.5.5 PROCESS CONTROL

Not applicable.

3.3.6.6.9.11.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
function Limit_Flag_Setting return Limit_Relations is
begin
   return Local_Limit_Flag_Setting;
end Limit_Flag_Setting;
```

3.3.6.6.9.11.10.5.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.6.9.11.10.5.8 LIMITATIONS

None.

3.3.6.6.10 UNIT DESIGN

None.

(This page left intentionally blank.)

```
package body Signal_Processing is

    package body Upper_Lower_Limiter is separate;

    package body Upper_Limiter is separate;

    package body Lower_Limiter is separate;

    package body Absolute_Limiter is separate;

    package body Absolute_Limiter_With_Flag is separate;

    package body General_First_Order_Filter is separate;

    package body Tustin_Lead_Lag_Filter is separate;

    package body Tustin_Lag_Filter is separate;

    package body Second_Order_Filter is separate;

    package body Tustin_Integrator_With_Limit is separate;

    package body Tustin_Integrator_With_Asymmetric_Limit is separate;
begin
   null;
exception
   when others => raise;
end Signal_Processing;
```

```
separate (Signal_Processing)
package body Upper_Lower_Limiter is

-------------------------
-- -- Local Data Objects-
-------------------------

   Upper_Limit,
   Lower_Limit : Signal_Type;

pragma PAGE;
   procedure Update_Limits (New_Upper_Limit : in Signal_Type;
                            New_Lower_Limit : in Signal_Type) is
   begin

      if New_Upper_Limit > New_Lower_Limit then
         Upper_Limit := New_Upper_Limit;
         Lower_Limit := New_Lower_Limit;
      else
          raise Limit_Exception;
      end if;

   end Update_Limits;

pragma PAGE;
   function Limit (Signal : Signal_Type) return Signal_Type is

      Limited_Signal: Signal_Type;

   begin

      if Signal > Upper_Limit then
         Limited_Signal := Upper_Limit;
      elsif Signal < Lower_Limit then
         Limited_Signal := Lower_Limit;
      else
         Limited_Signal := Signal;
      end if;

      return Limited_Signal;

   end Limit;

pragma PAGE;
-----------------------------------
--begin processing for package body
-----------------------------------

begin

   Update_Limits (New_Upper_Limit => Initial_Upper_Limit,
                  New_Lower_Limit => Initial_Lower_Limit);

end Upper_Lower_Limiter;
```

```
separate (Signal_Processing)
package body Upper_Limiter is

------------------------
-- -- Local Data Object-
------------------------

   Upper_Limit: Signal_Type := Initial_Upper_Limit;

pragma PAGE;
   procedure Update_Limit (New_Upper_Limit : in Signal_Type) is
   begin

      Upper_Limit := New_Upper_Limit;

   end Update_Limit;

pragma PAGE;
   function Limit (Signal : Signal_Type) return Signal_Type is

      Limited_Signal: Signal_Type;

   begin

      if Signal > Upper_Limit then
         Limited_Signal := Upper_Limit;
      else
         Limited_Signal := Signal;
      end if;

      return Limited_Signal;

   end Limit;

end Upper_Limiter;
```

```
separate (Signal_Processing)
package body Lower_Limiter is

------------------------
-- -- Local data objects
------------------------

   Lower_Limit : Signal_Type := Initial_Lower_Limit;

pragma PAGE;
   procedure Update_Limit (New_Lower_Limit : in Signal_Type) is

   begin
      Lower_Limit := New_Lower_Limit;
   end Update_Limit;

pragma PAGE;
   function Limit (Signal : Signal_Type) return Signal_Type is

      Limited_Signal: Signal_Type;

   begin

      if Signal < Lower_Limit then
         Limited_Signal := Lower_Limit;
      else
         Limited_Signal := Signal;
      end if;

      return Limited_Signal;

   end Limit;

end Lower_Limiter;
```

```
separate (Signal_Processing)
package body Absolute_Limiter is

-- --------------------
-- --Local Data Object-
-- --------------------

   Absolute_Limit: Signal_Type := abs (Initial_Absolute_Limit);

pragma PAGE;
   procedure Update_Limit (New_Absolute_Limit : in Signal_Type) is
   begin

      Absolute_Limit := abs (New_Absolute_Limit);

   end Update_Limit;

pragma PAGE;
   function Limit (Signal : Signal_Type) return Signal_Type is

      Limited_Signal: Signal_Type;

   begin

      if abs (Signal) > Absolute_Limit then

         if Signal > 0.0 then

            Limited_Signal := Absolute_Limit;

         else

            Limited_Signal := - Absolute_Limit;

         end if;

      else

         Limited_Signal := Signal;

      end if;

      return Limited_Signal;

   end Limit;

end Absolute_Limiter;
```

```
separate (Signal_Processing)
package body Absolute_Limiter_With_Flag is

-----------------------
-- --Local Data Object-
-----------------------

   Absolute_Limit: Signal_Type := abs (Initial_Absolute_Limit);

   Limit_Relation: Limit_Relations := Within_Limit;

pragma PAGE;
   procedure Update_Limit (New_Absolute_Limit : in Signal_Type) is
   begin

      Absolute_Limit := abs (New_Absolute_Limit);

   end Update_Limit;

pragma PAGE;
   function Limit (Signal : Signal_Type) return Signal_Type is

      Limited_Signal: Signal_Type;

   begin

      if abs (Signal) >= Absolute_Limit then

         if Signal >= 0.0 then

            Limited_Signal := Absolute_Limit;
            Limit_Relation := At_Positive_Limit;

         else

            Limited_Signal := - Absolute_Limit;
            Limit_Relation := At_Negative_Limit;

         end if;

      else

         Limited_Signal := Signal;
         Limit_Relation := Within_Limit;

      end if;

      return Limited_Signal;

   end Limit;
pragma PAGE;
   function Limit_Flag_Setting return Limit_Relations is
   begin

      return Limit_Relation;

   end Limit_Flag_Setting;
```

```
end Absolute_Limiter_With_Flag;
```

```
separate (Signal_Processing)
package body General_First_Order_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Coefficient_1         : Coefficient_Type := Initial_Coefficient_1;
   Coefficient_2         : Coefficient_Type := Initial_Coefficient_2;
   Coefficient_3         : Coefficient_Type := Initial_Coefficient_3;
   Previous_Input_Signal:  Signal_Type := Initial_Previous_Input_Signal;
   Previous_Output_Signal: Signal_Type :=
      ( Previous_Input_Signal * (Coefficient_1 + Coefficient_2) ) /
      ( 1.0 - Coefficient_3);

pragma PAGE;
   procedure Update_Coefficients (New_Coefficient_1 : in Coefficient_Type;
                                  New_Coefficient_2 : in Coefficient_Type;
                                  New_Coefficient_3 : in Coefficient_Type) is
   begin

      Coefficient_1 := New_Coefficient_1;
      Coefficient_2 := New_Coefficient_2;
      Coefficient_3 := New_Coefficient_3;

   end Update_Coefficients;

pragma PAGE;
   function Filter (Signal : Signal_Type) return Signal_Type is

      Output_Signal :  Signal_Type;

   begin

      Output_Signal := ( Signal * Coefficient_1) +
                       ( Previous_Input_Signal * Coefficient_2) +
                       ( Previous_Output_Signal * Coefficient_3);

      Previous_Input_Signal  := Signal;
      Previous_Output_Signal := Output_Signal;

      return Output_Signal;

   end Filter;

pragma PAGE;
   procedure Reinitialize (Signal : in Signal_Type) is

   begin

         Previous_Input_Signal := Signal;

         Previous_Output_Signal :=
                  ( Previous_Input_Signal * (Coefficient_1 + Coefficient_2) ) /
                  ( 1.0 - Coefficient_3);
   end Reinitialize;
```

```
end General_First_Order_Filter;
```

```
separate (Signal_Processing)
package body Tustin_Lag_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Coefficient_1         : Coefficient_Type := Initial_Coefficient_1;
   Coefficient_2         : Coefficient_Type := Initial_Coefficient_2;

   Previous_Input_Signal : Signal_Type := Initial_Previous_Input_Signal;

   Previous_Output_Signal: Signal_Type := Previous_Input_Signal *
                            ( (2.0 * Coefficient_1) / (1.0 - Coefficient_2) );

pragma PAGE;
   procedure Update_Coefficients (New_Coefficient_1 : in Coefficient_Type;
                                  New_Coefficient_2 : in Coefficient_Type) is

   begin

      Coefficient_1 := New_Coefficient_1;
      Coefficient_2 := New_Coefficient_2;

   end Update_Coefficients;
pragma PAGE;
   function Filter (Signal : Signal_Type) return Signal_Type is

      Output_Signal :  Signal_Type;

   begin

      Output_Signal := ( (Signal + Previous_Input_Signal) * Coefficient_1) +
                       ( Previous_Output_Signal * Coefficient_2);

      Previous_Input_Signal  := Signal;
      Previous_Output_Signal := Output_Signal;

      return Output_Signal;

   end Filter;

pragma PAGE;
   procedure Reinitialize (Signal: in Signal_Type) is

   begin

      Previous_Input_Signal := Signal;

      Previous_Output_Signal :=
        Previous_Input_Signal * ( (2.0 * Coefficient_1) / (1.0 - Coefficient_2) );

   end Reinitialize;

end Tustin_Lag_Filter;
```

```
separate (Signal_Processing)
package body Tustin_Lead_Lag_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Previous_Input_Signal : Signal_Type := Initial_Previous_Input_Signal;
   Previous_Output_Signal: Signal_Type := Initial_Previous_Input_Signal;

   Coefficient_1         : Coefficient_Type := Initial_Coefficient_1;
   Coefficient_2         : Coefficient_Type := Initial_Coefficient_2;

pragma PAGE;
   procedure Update_Coefficients (New_Coefficient_1 : in Coefficient_Type;
                                  New_Coefficient_2 : in Coefficient_Type) is
   begin

      Coefficient_1 := New_Coefficient_1;
      Coefficient_2 := New_Coefficient_2;

   end Update_Coefficients;

pragma PAGE;
   function Filter (Signal : Signal_Type) return Signal_Type is

      Output_Signal :  Signal_Type;

   begin

      Output_Signal := ( (Signal - Previous_Input_Signal) * Coefficient_1) +
                       ( (Previous_Output_Signal - Previous_Input_Signal) *
                          Coefficient_2) +
                       Previous_Input_Signal;

      Previous_Input_Signal  := Signal;
      Previous_Output_Signal := Output_Signal;

      return Output_Signal;

   end Filter;

pragma PAGE;
   procedure Reinitialize (Signal : in Signal_Type) is

   begin

      Previous_Input_Signal  := Signal;
      Previous_Output_Signal := Previous_Input_Signal;

   end Reinitialize;

end Tustin_Lead_Lag_Filter;
```

```
separate (Signal_Processing)
package body Second_Order_Filter is

------------------------
-- -- Local Data Object-
------------------------

   Previous_Input_Signal:         Signal_Type;
   Second_Previous_Input_Signal:  Signal_Type;

   Previous_Output_Signal:        Signal_Type;
   Second_Previous_Output_Signal: Signal_Type;

   Coefficient_1          : Coefficient_Type;
   Coefficient_2          : Coefficient_Type;
   Coefficient_3          : Coefficient_Type;

pragma PAGE;
   procedure Redefine_Coefficients
            (New_Coefficient_Defining_Parameter_1 : in Coefficient_Type;
             New_Coefficient_Defining_Parameter_2 : in Coefficient_Type) is

      Coefficient_Defining_Ratio:  Coefficient_Type;

   begin

      Coefficient_Defining_Ratio := 1.0 /
                                    (1.0 + New_Coefficient_Defining_Parameter_1 +
                                     New_Coefficient_Defining_Parameter_2);

      Coefficient_1 := (1.0 + New_Coefficient_Defining_Parameter_1) *
                       Coefficient_Defining_Ratio;

      Coefficient_2 := (2.0 * (1.0 - New_Coefficient_Defining_Parameter_2) ) *
                       Coefficient_Defining_Ratio;

      Coefficient_3 := (1.0 - (New_Coefficient_Defining_Parameter_1 +
                               New_Coefficient_Defining_Parameter_2)) *
                        Coefficient_Defining_Ratio;

   end Redefine_Coefficients;

pragma PAGE;
   function Filter (Signal : Signal_Type) return Signal_Type is

      Output_Signal :  Signal_Type := 0.0;

   begin

      Output_Signal := ( (Signal - Second_Previous_Input_Signal) *
                                                    Coefficient_1) +
                       ( (Previous_Input_Signal - Previous_Output_Signal) *
                                                    Coefficient_2) -
                       ( Second_Previous_Output_Signal * Coefficient_3);

      Second_Previous_Input_Signal  := Previous_Input_Signal;
      Previous_Input_Signal         := Signal;
```

```
      Second_Previous_Output_Signal := Previous_Output_Signal;
      Previous_Output_Signal        := Output_Signal;

      return Output_Signal;

   end Filter;


pragma PAGE;
   procedure Reinitialize (Signal : in Signal_Type) is

   begin

      Previous_Input_Signal        := Signal;
      Second_Previous_Input_Signal := Signal;

      Previous_Output_Signal        := Signal;
      Second_Previous_Output_Signal := Signal;

   end Reinitialize;

pragma PAGE;
-----------------------------------
--begin processing for package body
-----------------------------------

begin

   Previous_Input_Signal        := Initial_Previous_Input_Signal;
   Second_Previous_Input_Signal := Initial_Previous_Input_Signal;

   Previous_Output_Signal        := Initial_Previous_Input_Signal;
   Second_Previous_Output_Signal := Initial_Previous_Input_Signal;

   Redefine_Coefficients (New_Coefficient_Defining_Parameter_1 =>
                             Initial_Coefficient_Defining_Parameter_1,
                          New_Coefficient_Defining_Parameter_2 =>
                             Initial_Coefficient_Defining_Parameter_2);

end Second_Order_Filter;
```

```
with General_Purpose_Math;
separate (Signal_Processing)

package body Tustin_Integrator_With_Limit is

------------------------
-- -- local entities  -
------------------------

   package Local_Limit is new Signal_Processing.Absolute_Limiter_With_Flag
             (Signal_Type           => States,
              Initial_Absolute_Limit => Initial_Signal_Limit);

------------------------
-- -- Local Data Object-
------------------------

   Signal_Gain              : Gains          := Initial_Tustin_Gain;

   Previous_Gained_Input    : Gained_Signals :=
                                 Initial_Signal_Level * Signal_Gain;

   Previous_Output_Signal   : States         :=
                                 Local_Limit.Limit (Initial_State);

------------------------
-- -- local entities  -
------------------------

   package Local_Integrator is new General_Purpose_Math.Integrator
             (Dependent_Type           => States,
              Independent_Type         => Gained_Signals,
              Time_Interval            => States,
              Initial_Dependent_Value  => Previous_Output_Signal,
              Initial_Independent_Value => Previous_Gained_Input,
              Default_Delta_Time       => 1.0,
              "*"                      => "*");

pragma PAGE;
   procedure Update_Limit (New_Absolute_Limit: in States) is

   begin

      Local_Limit.Update_Limit (New_Absolute_Limit => New_Absolute_Limit);

   end Update_Limit;

pragma PAGE;
   procedure Update_Gain (New_Gain: in Gains) is

   begin

      Signal_Gain := New_Gain;

   end Update_Gain;

pragma PAGE;
```

```
   function Integrate (Signal : Signals) return States is

      New_Gained_Signal : Gained_Signals := Signal * Signal_Gain;
      New_Gained_Total  : Gained_Signals :=
                             New_Gained_Signal + Previous_Gained_Input;
      Output_Signal     : States;

   begin

-------------------------------------------------------------------------
--     if  within limit or
--           (beyond limit and new input brings output within limit) then
--         integrate new output
-------------------------------------------------------------------------

      if
         (Local_Limit.Limit_Flag_Setting = Within_Limit or else
         (Local_Limit.Limit_Flag_Setting = At_Positive_Limit and then
             New_Gained_Total < 0.0) or else
         (Local_Limit.Limit_Flag_Setting = At_Negative_Limit and then
             New_Gained_Total > 0.0) )
                                        then

         Output_Signal := Local_Integrator.Integrate (New_Gained_Signal);
         Output_Signal := Local_Limit.Limit (Output_Signal);

         if Local_Limit.Limit_Flag_Setting /= Within_Limit then

             RESET (Integrator_State => Output_Signal,
                    Signal           => Signal);

         end if;

         Previous_Output_Signal := Output_Signal;

      else

         Output_Signal := Previous_Output_Signal;
         Local_Integrator.Update
                             (Current_Independent_Value => New_Gained_Signal);

      end if;


      Previous_Gained_Input  := New_Gained_Signal;

      return Output_Signal;

   end Integrate;

pragma PAGE;
   procedure RESET (Integrator_State : in States;
                    Signal           : in Signals) is

      Limited_State: States := Local_Limit.Limit (Integrator_State);

   begin
```

```
      Previous_Gained_Input := Signal * Signal_Gain;

      Local_Integrator.Reinitialize
                          (Initial_Dependent_Value   => Integrator_State,
                           Initial_Independent_Value => Previous_Gained_Input);

   end RESET;

pragma PAGE;
   function Limit_Flag_Setting return Limit_Relations is

   begin

      return (Local_Limit.Limit_Flag_Setting);

   end Limit_Flag_Setting;

end Tustin_Integrator_With_Limit;
```

```
with General_Purpose_Math;
separate (Signal_Processing)

package body Tustin_Integrator_With_Asymmetric_Limit is

------------------------
-- -- local entities -
------------------------

   package Local_Limit is new Signal_Processing.Upper_Lower_Limiter
             (Signal_Type         => States,
              Initial_Lower_Limit => Initial_Signal_Lower_Limit,
              Initial_Upper_Limit => Initial_Signal_Upper_Limit);

-------------------------
-- -- Local Data Objects-
-------------------------

   Signal_Gain              : Gains         := Initial_Tustin_Gain;

   Previous_Gained_Input    : Gained_Signals :=
                                 Initial_Signal_Level * Signal_Gain;

   Previous_Output_Signal   : States        :=
                                 Local_Limit.Limit (Initial_State);

   Local_Limit_Flag_Setting : Limit_Relations := Within_Limit;

   Local_Lower_Limit        : States := Initial_Signal_Lower_Limit;
   Local_Upper_Limit        : States := Initial_Signal_Upper_Limit;


------------------------
-- -- local entities -
------------------------

   package Local_Integrator is new General_Purpose_Math.Integrator
             (Dependent_Type           => States,
              Independent_Type         => Gained_Signals,
              Time_Interval            => States,
              Initial_Dependent_Value  => Previous_Output_Signal,
              Initial_Independent_Value => Previous_Gained_Input,
              Default_Delta_Time       => 1.0);
pragma PAGE;
   procedure Update_Limits (New_Lower_Limit: in States;
                            New_Upper_Limit: in States) is

   begin

      Local_Limit.Update_Limits (New_Lower_Limit => New_Lower_Limit,
                                 New_Upper_Limit => New_Upper_Limit);

      Local_Lower_Limit := New_Lower_Limit;
      Local_Upper_Limit := New_Upper_Limit;

   end Update_Limits;
```

```
pragma PAGE;
   procedure Update_Gain (New_Gain: in Gains) is

   begin

      Signal_Gain := New_Gain;

   end Update_Gain;

pragma PAGE;
   function Integrate (Signal : Signals) return States is

      New_Gained_Signal : Gained_Signals := Signal * Signal_Gain;
      New_Gained_Total  : Gained_Signals :=
                              New_Gained_Signal + Previous_Gained_Input;
      Output_Signal     : States;

   begin

--    ------------------------------------------------------------------------
--    --if  within limit or
--    --       (beyond limit and new input brings output within limit) then
--    --    integrate new output
--    ------------------------------------------------------------------------

      if
         (Local_Limit_Flag_Setting = Within_Limit or else
         (Local_Limit_Flag_Setting = At_Positive_Limit and then
             New_Gained_Total < 0.0) or else
         (Local_Limit_Flag_Setting = At_Negative_Limit and then
             New_Gained_Total > 0.0) )
                                        then

         Output_Signal := Local_Integrator.Integrate (New_Gained_Signal);
         Output_Signal := Local_Limit.Limit (Output_Signal);

         if Output_Signal >= Local_Upper_Limit then
            Local_Limit_Flag_Setting := At_Positive_Limit;
         elsif Output_Signal <= Local_Lower_Limit then
            Local_Limit_Flag_Setting := At_Negative_Limit;
         else
            Local_Limit_Flag_Setting := Within_Limit;
         end if;

         if Local_Limit_Flag_Setting /= Within_Limit then

             RESET (Integrator_State => Output_Signal,
                    Signal           => Signal);

         end if;

         Previous_Output_Signal := Output_Signal;

      else

         Output_Signal := Previous_Output_Signal;
         Local_Integrator.Update
```

```
                              (Current_Independent_Value => New_Gained_Signal);

      end if;


      Previous_Gained_Input  := New_Gained_Signal;

      return Output_Signal;

   end Integrate;

pragma PAGE;
   procedure RESET (Integrator_State : in States;
                    Signal           : in Signals) is

      Limited_State: States := Local_Limit.Limit (Integrator_State);

   begin

      Previous_Gained_Input := Signal * Signal_Gain;

      Local_Integrator.Reinitialize
                         (Initial_Dependent_Value   => Integrator_State,
                          Initial_Independent_Value => Previous_Gained_Input);

   end RESET;

pragma PAGE;
   function Limit_Flag_Setting return Limit_Relations is

   begin

      return Local_Limit_Flag_Setting;

   end Limit_Flag_Setting;

end Tustin_Integrator_With_Asymmetric_Limit;
```

(This page left intentionally blank.)

3.3.6.7 GENERAL PURPOSE MATH (BODY) TLCSC P687 (CATALOG #P28-0)

This TLCSC is a package which consists of two type of subpackages: generic packages and simple packages which contain generic functions. As a group, the subpackages provide the general purpose math routines required by the rest of the math parts.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.1 REQUIREMENTS ALLOCATION

The following chart summarizes the allocation of CAMP requirements to this

Tlcsc:

Name	Requirements Allocation
Lookup_Table_Even_Spacing	R118
Lookup_Table_Uneven_Spacing	R119
Incrementor	R120
Decrementor	R121
Running_Average	R142
Change_Calculator	R113
Accumulator	R114
Change_Accumulator	R115
Integrator	R124
Interpolate_or_Extrapolate	R116, R117
Square_Root	R123
Root_Sum_Of_Squares	R122
Sign	R224
Mean_Value	k144
Mean_Absolute_Difference	R143

3.3.6.7.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.3 INPUT/OUTPUT

None.

3.3.6.7.4 LOCAL DATA

None.

3.3.6.7.5 PROCESS CONTROL

Not applicable.

3.3.6.7.6 PROCESSING

The following describes the processing performed by this part:

```
with Polynomials;
package body General_Purpose_Math is

   package body Lookup_Table_Even_Spacing   is separate;

   package body Lookup_Table_Uneven_Spacing is separate;

   package body Two_Way_Table_Lookup        is separate;

   package body Incrementor                 is separate;

   package body Decrementor                 is separate;

   package body Running_Average             is separate;

   package body Change_Calculator           is separate;

   package body Accumulator                 is separate;

   package body Change_Accumulator          is separate;

   package body Integrator                  is separate;

   function Interpolate_or_Extrapolate
               (Input             : in Independent_Type;
                Lower_Independent : in Independent_Type;
                Higher_Independent: in Independent_Type;
                Lower_Dependent   : in Dependent_Type;
                Higher_Dependent  : in Dependent_Type)
               return Dependent_Type is separate;

   package body Square_Root is separate;

   function Root_Sum_Of_Squares (X : Real_Type;
                                 Y : Real_Type;
                                 Z : Real_Type)
                                return Real_Type is separate;

   function Sign (Input_Variable : Real_Type) return INTEGER is separate;

   function Mean_Value (Value_Vector : Vector_Type)
                        return Element_Type is separate;

   function Mean_Absolute_Difference (Value_Vector : Vector_Type)
                                      return Element_Type is separate;

end General_Purpose_Math;
```

3.3.6.7.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.8 LIMITATIONS

None.

3.3.6.7.9 LLCSC DESIGN

3.3.6.7.9.1 LOOKUP_TABLE_EVEN_SPACING (BODY) PACKAGE DESIGN (CATALOG #P29-0)

This LLCSC, which is designed as an Ada generic package, provides the ability to initialize and search through a table of independent and dependent values which are evenly spaced.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R118.

3.3.6.7.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Independent_Type	generic float	Type for the independent variable
Dependent_Type	generic float	Type for the dependent variable
Index_Type	generic float	Type for the independent variable

Data objects:

The following objects were defined in the package specification:

Name	Type	Mode	Description
Minimum_Independnt_Value	Independent_Type	in	value of the first independent table value
Maximum_Independnt_Value	Independent_Type	in	value of the last independent table value

3.3.6.7.9.1.4 LOCAL DATA

None.

3.3.6.7.9.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.1.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

package body Lookup_Table_Even_Spacing is

-- -- Declare table step variable and initialize it to the value of the step
-- -- between each successive independent entry


   Internal_Min_Independent_Value : constant Independent_Type
                                          := Minimum_Independent_Value;
   Internal_Max_Independent_Value : constant Independent_Type
                                          := Maximum_Independent_Value;

   Table_Step : Independent_Type :=
                   (Internal_Max_Independent_Value - Internal_Min_Independent_Value)
                       / Independent_Type (Tables'LENGTH - 1);

end Lookup_Table_Even_Spacing;
```

3.3.6.7.9.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.1.8 LIMITATIONS

The exception "Value_Out_Of_Range" is raised when Lookup is called with a value outside of the table range.

3.3.6.7.9.1.9 LLCSC DESIGN

None.

3.3.6.7.9.1.10 UNIT DESIGN

3.3.6.7.9.1.10.1 INITIALIZE UNIT DESIGN

This procedure initializes one row of the table.

3.3.6.7.9.1.10.1.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.1.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.1.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Table	Tables	out	Table to be searched
Index	Index_Type	in	Row of the lookup table
Dependent_Value	Dependent_Type	in	The dependent value to be inserted into this row

3.3.6.7.9.1.10.1.4 LOCAL DATA

None.

3.3.6.7.9.1.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.1.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Initialize (Table           : out Tables;
                      Index           : in  Index_Type;
                      Dependent_Value : in  Dependent_Type) is
begin
```

```
    Table (Index) := Dependent_Value;

  end Initialize;
```

3.3.6.7.9.1.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.1.10.1.8 LIMITATIONS

None.

3.3.6.7.9.1.10.2 LOOKUP UNIT DESIGN

This procedure does a lookup of the table and returns the two closest sets of dependent and independent values; if the key is not in the table range, an exception is raised.

3.3.6.7.9.1.10.2.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.1.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.1.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Table	Tables	in	Table to be searched
Key	Independent_Type	in	The input independent value be inserted into this row
Lower_Independent	Independent_Type	out	The lower independent value returned
Higher_Independent	Independent_Type	out	The higher independent value returned
Lower_Dependent	Dependent_Type	out	The lower dependent value returned
Higher_Dependent	Dependent_Type	out	The higher dependent value returned

3.3.6.7.9.1.10.2.4 LOCAL DATA

None.

3.3.6.7.9.1.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.1.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Lookup
                (Table              : in  Tables;
                 Key                : in  Independent_Type;
                 Lower_Independent  : out Independent_Type;
                 Higher_Independent : out Independent_Type;
                 Lower_Dependent    : out Dependent_Type;
                 Higher_Dependent   : out Dependent_Type) is

      Index                  : Index_Type;
      Temp_Lower_Independent : Independent_Type;
      Index_Float_Var        : Independent_Type;
      Index_Integer_Var      : INTEGER;

   begin

--    -- Raise exception if Key is outside of table range

      if Key < Internal_Min_Independent_Value or Key > Internal_Max_Independent_Value th
         raise Value_Out_Of_Range;
      end if;

--    -- Map the key to an index value

      Index_Float_Var := (Key - Internal_Min_Independent_Value) / Table_Step;
      Index_Integer_Var := INTEGER (Index_Float_Var);

      if Index_Float_Var >= Independent_Type (Index_Integer_Var) then
        Index_Integer_Var := Index_Integer_Var + 1;
      end if;

      Index := Index_Type'VAL (Index_Integer_Var);


--    -- Adjust the index value if the key was mapped to the last entry

      if Index = Index_Type'LAST then
         Index := Index_Type'PRED (Index);
      end if;


--    -- Return the table values

      Temp_Lower_Independent := Internal_Min_Independent_Value +
```

```
                                    (Independent_Type (Index_Type'POS (Index) - Index_Type'F
(Index_Type'FIRST)) *
                                     Table_Step);

        Lower_Independent  := Temp_Lower_Independent;
        Higher_Independent := Temp_Lower_Independent + Table_Step;

        Lower_Dependent  := Table (Index);
        Higher_Dependent := Table (Index_Type'SUCC(Index));

     end Lookup;
```

3.3.6.7.9.1.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.1.10.2.8 LIMITATIONS

The exception "Value_Out_Of_Range" is raised if the input key is outside the table range.

3.3.6.7.9.1.10.3 LOOKUP UNIT DESIGN

This procedure does a lookup of the table and returns the two closest sets of dependent and independent values and a flag specifying whether the key is below, within, or above the table range.

3.3.6.7.9.1.10.3.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.1.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.1.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Table	Tables	in	The table to be searched
Key	Independent_Type	in	The input independent value be inserted into this row
Lower_Independent	Independent_Type	out	The lower independent value returned
Higher_Independent	Independent_Type	out	The higher independent value returned
Lower_Dependent	Dependent_Type	out	The lower dependent value returned
Higher_Dependent	Dependent_Type	out	The higher dependent value returned
Key_Location	Key_Range_Flag	out	Specifies whether the key is above, in, or below the table range

3.3.6.7.9.1.10.3.4 LOCAL DATA

None.

3.3.6.7.9.1.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.1.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Lookup
               (Table              : in  Tables;
                Key                : in  Independent_Type;
                Lower_Independent  : out Independent_Type;
                Higher_Independent : out Independent_Type;
                Lower_Dependent    : out Dependent_Type;
                Higher_Dependent   : out Dependent_Type;
                Key_Location       : out Key_Range_Flag) is

      Index                  : Index_Type;
      Temp_Lower_Independent : Independent_Type;
      Index_Float_Var        : Independent_Type;
      Index_Integer_Var      : INTEGER;

   begin

--    -- Return the first two rows of the lookup table if Key is below table range

      if Key < Internal_Min_Independent_Value then

         Key_Location := Below_Table_Range;

         Lower_Independent  := Internal_Min_Independent_Value;
```

```
            Higher_Independent := Internal_Min_Independent_Value + Table_Step;

            Lower_Dependent  := Table (Index_Type'FIRST);
            Higher_Dependent := Table (Index_Type'SUCC(Index_Type'FIRST));

--      -- Return the last two rows of the lookup table if Key is above table range

         elsif Key > Internal_Max_Independent_Value then

            Key_Location := Above_Table_Range;

            Lower_Independent  := Internal_Max_Independent_Value - Table_Step;
            Higher_Independent := Internal_Max_Independent_Value;

            Lower_Dependent  := Table (Index_Type'PRED(Index_Type'LAST));
            Higher_Dependent := Table (Index_Type'LAST);


--      -- Key is in table range, so map the key to an index value and return the
--      -- corresponding values

         else

            Key_Location := In_Table_Range;


--          -- Map the Key to an Index value

            Index_Float_Var := (Key - Internal_Min_Independent_Value) / Table_Step;
            Index_Integer_Var := INTEGER (Index_Float_Var);

            if Index_Float_Var >= Independent_Type (Index_Integer_Var) then
              Index_Integer_Var := Index_Integer_Var + 1;
            end if;

            Index := Index_Type'VAL (Index_Integer_Var);


--          -- Adjust the Index if it has been mapped to the last entry

            if Index = Index_Type'LAST then
               Index := Index_Type'PRED (Index);
            end if;

--          -- Return the table values

            Temp_Lower_Independent := Internal_Min_Independent_Value +
                                      (Independent_Type (Index_Type'POS (Index) - Index_Ty
POS(Index_Type'FIRST)) *
                                       Table_Step);

            Lower_Independent  := Temp_Lower_Independent;
            Higher_Independent := Temp_Lower_Independent + Table_Step;

            Lower_Dependent  := Table (Index);
            Higher_Dependent := Table (Index_Type'SUCC(Index));

         end if;
```

```
end Lookup;
```

3.3.6.7.9.1.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.1.10.3.8 LIMITATIONS

None.

3.3.6.7.9.2 LOOKUP_TABLE_UNEVEN_SPACING (BODY) PACKAGE DESIGN (CATALOG #P30-0)

This LLCSC, which is designed as an Ada generic package, provides the ability to initialize and search through a table of independent and dependent values which are unevenly spaced.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R119.

3.3.6.7.9.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table describes the generic formal types required by this part:

Name	Base Type	Description
Dependent_Type	generic float	Type for the dependent variable
Independent_Type	generic float	Type for the independent variable
Index_Type	generic float	Type for the independent variable

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Mode	Description
Minimum_Independnt_Value	Independent_Type	in	value of the first independent table value
Maximum_Independnt_Value	Independent_Type	in	value of the last independent table value

3.3.6.7.9.2.4 LOCAL DATA

None.

3.3.6.7.9.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.2.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)
package body Lookup_Table_Uneven_Spacing is
end Lookup_Table_Uneven_Spacing;
```

3.3.6.7.9.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.2.8 LIMITATIONS

None.

3.3.6.7.9.2.9 LLCSC DESIGN

None.

3.3.6.7.9.2.10 UNIT DESIGN

3.3.6.7.9.2.10.1 INITIALIZE UNIT DESIGN

This procedure initializes one row of the table.

3.3.6.7.9.2.10.1.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.2.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.2.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Table	Tables	out	The lookup table
Index	Index_Type	in	Row of the lookup table
Independent_Val	Independent_ Type	in	The independent value to be inserted into this row
Dependent_Value	Dependent_ Type	in	The dependent value to be inserted into this row

3.3.6.7.9.2.10.1.4 LOCAL DATA

None.

3.3.6.7.9.2.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.2.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Initialize
             (Table             : out Tables;
              Index             : in  Index_Type;
              Independent_Value : in  Independent_Type;
              Dependent_Value   : in  Dependent_Type) is
begin

  Table (Index).Independent_Entry := Independent_Value;
  Table (Index).Dependent_Entry   := Dependent_Value;

end Initialize;
```

3.3.6.7.9.2.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.2.10.1.8 LIMITATIONS

None.

3.3.6.7.9.2.10.2 LOOKUP UNIT DESIGN

This procedure does a lookup of the table and returns the two closest sets of dependent and independent values; if the input key is outside the table range, an exception will be raised.

3.3.6.7.9.2.10.2.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.2.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.2.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Table	Tables	in	Lookup table to be searched
Key	Independent_Type	in	The input independent value be inserted into this row
Lower_Independent	Independent_Type	out	The lower independent value returned
Higher_Independent	Independent_Type	out	The higher independent value returned
Lower_Dependent	Dependent_Type	out	The lower dependent value returned
Higher_Dependent	Dependent_Type	out	The higher dependent value returned

3.3.6.7.9.2.10.2.4 LOCAL DATA

None.

3.3.6.7.9.2.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.2.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Lookup
                (Table              : in  Tables;
                 Key                : in  Independent_Type;
                 Lower_Independent  : out Independent_Type;
                 Higher_Independent : out Independent_Type;
                 Lower_Dependent    : out Dependent_Type;
                 Higher_Dependent   : out Dependent_Type) is

      Index : Index_Type;

   begin

--    -- Raise an exception if the input key is outside of the table range

      if Key < Table (Index_Type'FIRST).Independent_Entry or
         Key > Table (Index_Type'LAST).Independent_Entry then

         raise Value_Out_Of_Range;

      end if;


--    -- Search through the table until the independent entry is greater than
--    -- or equal to the Key or have gone through the entire table (actually
--    -- go through one less row than the whole table since we return two rows)

      Index := Index_Type'SUCC(Index_Type'FIRST);

      loop

         exit when (Table (Index).Independent_Entry >= Key);

         Index := Index_Type'SUCC (Index);

      end loop;


--    -- Return the table values

      Lower_Independent := Table (Index_Type'PRED(Index)).Independent_Entry;
      Lower_Dependent   := Table (Index_Type'PRED(Index)).Dependent_Entry;

      Higher_Independent := Table (Index).Independent_Entry;
      Higher_Dependent   := Table (Index).Dependent_Entry;

   end Lookup;
```

3.3.6.7.9.2.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.2.10.2.8 LIMITATIONS

The exception "Value_Out_Of_Range" is created if Key flag) is outside of the Table range

3.3.6.7.9.2.10.3 LOOKUP UNIT DESIGN

This procedure does a lookup of the table and returns the two closest sets of dependent and independent values along with a flag which the table range, an exception will be raised.

3.3.6.7.9.2.10.3.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.2.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.2.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Table	Tables	in	Lookup table to be searched
Key	Independent_Type	in	The input independent value be inserted into this row
Lower_Independent	Independent_Type	out	The lower independent value returned
Higher_Independent	Independent_Type	out	The higher independent value returned
Lower_Dependent	Dependent_Type	out	The lower dependent value returned
Higher_Dependent	Dependent_Type	out	The higher dependent value returned
Key_Location	Key_Range_Flag	out	Specifies whether the key is above, in, or below the table range

3.3.6.7.9.2.10.3.4 LOCAL DATA

None.

3.3.6.7.9.2.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.2.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Lookup
                (Table              : in  Tables;
                 Key                : in  Independent_Type;
                 Lower_Independent  : out Independent_Type;
                 Higher_Independent : out Independent_Type;
                 Lower_Dependent    : out Dependent_Type;
                 Higher_Dependent   : out Dependent_Type;
                 Key_Location       : out Key_Range_Flag) is

      Index : Index_Type;

   begin

--    -- This routine first checks to make sure that the Key is within the Table
--    -- range; if it is below, return the first two rows; if it is above, return
--    -- the last two rows; if it is within the range, find which rows it is between
--    -- and return those two rows

      if Key < Table (Index_Type'FIRST).Independent_Entry then

--       -- Return the first two rows of the table if key is less than first entry

         Key_Location := Below_Table_Range;

         Lower_Independent  := Table (Index_Type'FIRST). Independent_Entry;
         Higher_Independent := Table (Index_Type'SUCC (Index_Type'FIRST)). Independent_E
ry;


         Lower_Dependent  := Table (Index_Type'FIRST). Dependent_Entry;
         Higher_Dependent := Table (Index_Type'SUCC (Index_Type'FIRST)). Dependent_Entry

      elsif Key > Table (Index_Type'LAST).Independent_Entry then

--       -- Return the last two rows of the table if key is greater than last entry

         Key_Location := Above_Table_Range;

         Lower_Independent  := Table (Index_Type'PRED (Index_Type'LAST)). Independent_En
y;
         Higher_Independent := Table (Index_Type'LAST). Independent_Entry;

         Lower_Dependent  := Table (Index_Type'PRED (Index_Type'LAST)). Dependent_Entry;
         Higher_Dependent := Table (Index_Type'LAST). Dependent_Entry;

      else
```

```
--          -- Search through the table until the independent entry is greater than
--          -- or equal to the Key or have gone through the entire table

            Key_Location := In_Table_Range;

            Index := Index_Type'SUCC(Index_Type'FIRST);

            loop

               exit when (Table (Index).Independent_Entry >= Key);

               Index := Index_Type'SUCC (Index);

            end loop;


--          -- Return the table values

            Lower_Independent := Table (Index_Type'PRED(Index)).Independent_Entry;
            Lower_Dependent   := Table (Index_Type'PRED(Index)).Dependent_Entry;

            Higher_Independent := Table (Index).Independent_Entry;
            Higher_Dependent   := Table (Index).Dependent_Entry;

         end if;

      end Lookup;
```

3.3.6.7.9.2.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.2.10.3.8 LIMITATIONS

None

3.3.6.7.9.3 INCREMENTOR (BODY) PACKAGE DESIGN (CATALOG #P31-0)

This generic package provides the capability to initialize and/or reinitialize a variable to be incremented, select a value to be used as an incrementor, and increment the variable accordingly.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.3.1 REQUIREMENTS ALLOCATION

This LLCSC meets CAMP requirement R120.

3.3.6.7.9.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Real_Type	gener float	Type of the incrementor variable

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Mode	Description
Initial_Value	Real_Type	in	Initial incrementor value
Increment_Amount	Real_Type	in	Amount by which to increment

3.3.6.7.9.3.4 LOCAL DATA

Data objects:

The following data objects are maintained by this package:

Name	Type	Description
Increment_Variable	Real_Type	Storage of the incremented variable
Amount_of_Increment	Real_Type	Storage of increment amount

3.3.6.7.9.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.3.6 PROCESSING

The following describes the processing performed by this part:

separate (General_Purpose_Math)

```
package body Incrementor is

-- -- Declare the Incrementor and amount of increment object and initialize them
-- -- to values given during instantiation

   Amount_of_Increment  : Real_Type := Increment_Amount;
   Increment_Variable   : Real_Type := Initial_Value;

end Incrementor;
```

3.3.6.7.9.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.3.8 LIMITATIONS

None.

3.3.6.7.9.3.9 LLCSC DESIGN

None.

3.3.6.7.9.3.10 UNIT DESIGN

3.3.6.7.9.3.10.1 REINITIALIZE UNIT DESIGN

This procedure reinitializes the increment variable and the increment amount.

3.3.6.7.9.3.10.1.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.3.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.3.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Value	Real_Type	in	Initial value for incrementor
Increment_Amount	Real_Type	in	Amount by which to increment

3.3.6.7.9.3.10.1.4 LOCAL DATA

None.

3.3.6.7.9.3.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.3.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Value    : in Real_Type;
                        Increment_Amount : in Real_Type) is
begin

   Increment_Variable := Initial_Value;

   Amount_of_Increment := Increment_Amount;

end Reinitialize;
```

3.3.6.7.9.3.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.3.10.1.8 LIMITATIONS

None.

3.3.6.7.9.3.10.2 INCREMENT UNIT DESIGN

This procedure increments the stored increment variable by the increment amount and returns the new value of the increment variable.

3.3.6.7.9.3.10.2.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.3.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.3.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
<returned value>	Real_Type	out	new value of increment var.

3.3.6.7.9.3.10.2.4 LOCAL DATA

None.

3.3.6.7.9.3.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.3.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   function Increment return Real_Type is
   begin

--    -- Increment and return new value

      Increment_Variable := Increment_Variable + Amount_of_Increment;
      return Increment_Variable;

   end Increment;
```

3.3.6.7.9.3.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements defined in this part's parent component and used by this part:

Data types:

The following table summarizes the LLC types required by this part:

Name	Description
Element_Type	Type of the Incrementor variable

Data objects:

The following table summarizes the LLC objects required by this part:

Name	Type	Description
Increment_Variable	Real_Type	Variable being incremented
Amount_of_Increment	Real_Type	Amount by which incrementor is increased each time

3.3.6.7.9.3.10.2.8 LIMITATIONS

None.

3.3.6.7.9.4 DECREMENTOR (BODY) PACKAGE DESIGN (CATALOG #P32-0)

This generic package provides the capability to initialize and/or reinitialize a variable that is to be decremented, select a value to be used as an decrementor, and decrement the variable accordingly.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.4.1 REQUIREMENTS ALLOCATION

This LLCSC meets CAMP requirement R121.

3.3.6.7.9.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.4.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table describes the generic formal types required by this part:

Name	Base Type	Description
Real_Type	gener float	Type of the decrementor variable

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Mode	Description
Initial_Value	Real_Type	in	Initial decrementor value
Decrement_Amount	Real_Type	in	Amount by which to decrement

3.3.6.7.9.4.4 LOCAL DATA

Data objects:

The following table describes the object used local to this package:

Name	Type	Description
Decrement Variable	Real_Type	Storage of the decremented variable
Amount_of_ Decrement	Real_Type	Storage of decrement amount

3.3.6.7.9.4.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.4.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

package body Decrementor is

-- -------------
-- -Data Objects-
-- -------------

-- -- Each of these objects is initialized to the value given during instantiation

   Amount_of_Decrement : Real_Type := Decrement_Amount;
   Decrement_Variable  : Real_Type := Initial_Value;


end Decrementor;
```

3.3.6.7.9.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.4.8 LIMITATIONS

None.

3.3.6.7.9.4.9 LLCSC DESIGN

None.

3.3.6.7.9.4.10 UNIT DESIGN

3.3.6.7.9.4.10.1 REINITIALIZE UNIT DESIGN

This procedure reinitializes the decrement variable and the decrement amount.

3.3.6.7.9.4.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R121.

3.3.6.7.9.4.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.4.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Value	Real_Type	in	Initial value for decrementor
Decrement_Amount	Real_Type	in	Amount by which to decrement

3.3.6.7.9.4.10.1.4 LOCAL DATA

None.

3.3.6.7.9.4.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.4.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Value    : in Real_Type;
                        Decrement_Amount : in Real_Type) is
```

```
   begin

      Decrement_Variable := Initial_Value;

      Amount_of_Decrement := Decrement_Amount;

   end Reinitialize;
```

3.3.6.7.9.4.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF LLC ELEMENTS:

The following tables describe the elements defined in this part's parent component and used by this part:

Data types:

The following table summarizes the LLC types required by this part:

Name	Description
Element_Type	Type of the Decrementor variable

Data objects:

The following table summarizes the LLC objects required by this part:

Name	Type	Description
Decrement_Variable	Real_Type	Variable being decremented
Amount_of_Decrement	Real_Type	Amount by which decrementor is decreased each time

3.3.6.7.9.4.10.1.8 LIMITATIONS

None.

3.3.6.7.9.4.10.2 DECREMENT UNIT DESIGN

This procedure decrements the stored decrement variable by the decrement amount and returns the new value of the decrement variable.

3.3.6.7.9.4.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R121.

3.3.6.7.9.4.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.4.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
<returned value>	Real_Type	out	new value of decrement var.

3.3.6.7.9.4.10.2.4 LOCAL DATA

None.

3.3.6.7.9.4.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.4.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   function Decrement return Real_Type is
   begin

--     -- Decrement and return new value

       Decrement_Variable := Decrement_Variable - Amount_of_Decrement;
       return Decrement_Variable;

   end Decrement;
```

3.3.6.7.9.4.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.4.10.2.8 LIMITATIONS

None.

3.3.6.7.9.5 RUNNING_AVERAGE (BODY) PACKAGE DESIGN (CATALOG #P33-0)

This generic package provides the capability to initialize a sum and/or a count and to maintain a running average. A reinitialization routine is provided to reinitialize the sum and count. An averaging routine is provided to perform the running sum.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R142

3.3.6.7.9.5.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.5.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following generic formal parameters were defined in this LLC's package specification:

Name	Base Type	Description
Real_Type	gen. float	Type of the running average var.

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Mode	Description
Initial_Sum	Real_Type	in	Initial running sum
Initial_Count	INTEGER	in	Initial _# of data points

Subprograms:

The following table describes the generic formal subprograms required by this LLCSC:

Name	Type	Description
"/"	function	Real_Type := Real_Type / Integer

3.3.6.7.9.5.4 LOCAL DATA

Data objects:

The following data objects are defined local to this part:

Name	Type	Description
Sum	Real_Type	Sum of all measurements to this point
Count	Real_Type	Number of measurements to this point

3.3.6.7.9.5.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.5.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

package body Running_Average is

-- -- Declare Sum and Count variable and initialize them to values given at
-- -- instantiation

   Sum                    : Real_Type := Initial_Sum;
   Count                  : INTEGER := Initial_Count;


end Running_Average;
```

3.3.6.7.9.5.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.5.8 LIMITATIONS

None.

3.3.6.7.9.5.9 LLCSC DESIGN

None.

3.3.6.7.9.5.10 UNIT DESIGN

3.3.6.7.9.5.10.1 REINITIALIZE UNIT DESIGN

This procedure reinitializes the running sum and count along with the type of actions that will be requested (i.e., Sum, Count, or both)

3.3.6.7.9.5.10.1.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.5.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.5.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Sum	Real_Type	in	Starting sum
Initial_Count	Real_Type	in	Starting count

3.3.6.7.9.5.10.1.4 LOCAL DATA

None.

3.3.6.7.9.5.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.5.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Sum   : in Real_Type;
                        Initial_Count : in INTEGER) is
begin

   Sum   := Initial_Sum;
   Count := Initial_Count;

end Reinitialize;
```

3.3.6.7.9.5.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.5.10.1.8 LIMITATIONS

None.

3.3.6.7.9.5.10.2 REINITIALIZE UNIT DESIGN

This procedure reinitializes the count.

3.3.6.7.9.5.10.2.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.5.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.5.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Count	Real_Type	in	Starting count

3.3.6.7.9.5.10.2.4 LOCAL DATA

None.

3.3.6.7.9.5.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.5.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Count : in INTEGER) is
begin

   Count := Initial_Count;
```

```
   end Reinitialize;
```

3.3.6.7.9.5.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.5.10.2.8 LIMITATIONS

None.

3.3.6.7.9.5.10.3 CURRENT_AVERAGE UNIT DESIGN

This unit calculates the current running average given input of a new value.

3.3.6.7.9.5.10.3.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.5.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.5.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
New_Value	Real_Type	in	New value to sum
<returned value>	Real_Type	out	New running average

3.3.6.7.9.5.10.3.4 LOCAL DATA

None.

3.3.6.7.9.5.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.5.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   function Current_Average (New_Value : Real_Type) return Real_Type is
```

```
   begin
--    -- Do average calculation

      Sum   := Sum + New_Value;
      Count := Count + 1;

      return Sum / Count;

   end Current_Average;
```

3.3.6.7.9.5.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.5.10.3.8 LIMITATIONS

None.

3.3.6.7.9.6 ACCUMULATOR PACKAGE DESIGN (CATALOG #P34-0)

This generic package provides a set of operations for maintaining an accumulation of a subject variable.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R114.

3.3.6.7.9.6.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.6.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following generic formal types were defined in this LLCSC's package specification:

Name	Base Type	Description
Element_Type	generic float	Type of the variable being accumulated.

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Mode	Description
Initial_Value	Real_Type	in	Initial accumulator value

3.3.6.7.9.6.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Accumulated_Value	Element_Type	Current value of the accumulator

3.3.6.7.9.6.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.6.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

package body Accumulator is

-- -- Declare accumulator variable and set it to the value given during
-- -- instantiation

   Accumulated_Value    : Element_Type := Initial_Value;

end Accumulator;
```

3.3.6.7.9.6.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.6.8 LIMITATIONS

None.

3.3.6.7.9.6.9 LLCSC DESIGN

None.

3.3.6.7.9.6.10 UNIT DESIGN

3.3.6.7.9.6.10.1 REINITIALIZE UNIT DESIGN

This procedure is used to reinitialize the accumulator variable.

3.3.6.7.9.6.10.1.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.6.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.6.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Value	Element_Type	in	Value to which accumulator will be initialized

3.3.6.7 9.6.10.1.4 LOCAL DATA

None.

3.3.6.7.9.6.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.6.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Value : in Element_Type) is
begin
   Accumulated_Value    := Initial_Value;
end Reinitialize;
```

3.3.6.7.9.6.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.6.10.1.8 LIMITATIONS

None.

3.3.6.7.9.6.10.2 ACCUMULATE UNIT DESIGN

This unit is a function which adds an input value to the Accumulator

3.3.6.7.9.6.10.2.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.6.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.6.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
New_Value	Element_Type	in	Value to which accumulator is set

3.3.6.7.9.6.10.2.4 LOCAL DATA

None.

3.3.6.7.9.6.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.6.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Accumulate (New_Value : in Element_Type) is
   begin

--     -- Add input value to accumulator
```

```
      Accumulated_Value := Accumulated_Value + New_Value;

   end Accumulate;
```

3.3.6.7.9.6.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.6.10.2.8 LIMITATIONS

None.

3.3.6.7.9.6.10.3 ACCUMULATE UNIT DESIGN

This unit is a function which adds an input value to the Accumulator and returns the new accumulator value.

3.3.6.7.9.6.10.3.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.6.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.6.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
New_Value	Element_Type	in	Value to which accumulator is set

3.3.6.7.9.6.10.3.4 LOCAL DATA

None.

3.3.6.7.9.6.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.6.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Accumulate
               (New_Value       : in  Element_Type;
                Retrieved_Value : out Element_Type) is
   begin

--     -- Add input value to accumulator

       Accumulated_Value := Accumulated_Value + New_Value;

--     -- return new accumulator value

       Retrieved_Value := Accumulated_Value;

   end Accumulate;
```

3.3.6.7.9.6.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.6.10.3.8 LIMITATIONS

None.

3.3.6.7.9.6.10.4 RETRIEVE UNIT DESIGN

This unit is a function which returns the current value of the accumulator.

3.3.6.7.9.6.10.4.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.6.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.6.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
<returned value>	Element_Type	out	Current value of Accumulator

3.3.6.7.9.6.10.4.4 LOCAL DATA

None.

3.3.6.7.9.6.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.6.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   function Retrieve return Element_Type is
   begin
--     -- Retrieve Accumulator Value;
       return Accumulated_Value;
   end Retrieve;
```

3.3.6.7.9.6.10.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.6.10.4.8 LIMITATIONS

None.

3.3.6.7.9.7 CHANGE ACCUMULATOR PACKAGE DESIGN (CATALOG #P36-0)

This generic package combines the operations of a change calculator and an accumulator; i.e., it provides a set of operations for maintaining the running accumulation of a change to a subject variable.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R115.

3.3.6.7.9.7.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following generic formal types were defined in this LLCSC's package specification:

Name	Base Type	Description
Element_Type	generic float	Type of the variable being accumulated and tracked

Data objects:

The following table describes the generic formal objects required by this part:

Name	Type	Mode	Description
Initial_PV	Element_Typ	in	Initial previous value
Initial_Accumulator_Value	Element_Typ	in	Initial accumulator value

3.3.6.7.9.7.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Value	Element_Type	Change calculator variable
Accumulator	Element_Type	Accumulator variable
PV_Is_Initialized	BOOLEAN	Flag determining if the Previous Value variable is initialized
Accumulator_is_Initialized	BOOLEAN	Flag determining if the Accumulator variable is initialized

3.3.6.7.9.7.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.6 PROCESSING

The following describes the processing performed by this part:

separate (General_Purpose_Math)

```
package body Change_Accumulator is

-- -- Declare accumulator and previous value objects and initialize them to
-- -- values given during instantiation

   Accumulator                : Element_Type := Initial_Accumulator_Value;
   Previous_Value             : Element_Type := Initial_Previous_Value;


end Change_Accumulator;
```

3.3.6.7.9.7.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.8 LIMITATIONS

None.

3.3.6.7.9.7.9 LLCSC DESIGN

None.

3.3.6.7.9.7.10 UNIT DESIGN

3.3.6.7.9.7.10.1 REINITIALIZE UNIT DESIGN

This procedure is used to initialize the accumulator

3.3.6.7.9.7.10.1.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.7.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Accumulatr _Value	Element_Type	in	Value to which accumulator will be initialized

3.3.6.7.9.7.10.1.4 LOCAL DATA

None.

3.3.6.7.9.7.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Accumulator_Value : in Element_Type) is
begin
   Accumulator := Initial_Accumulator_Value;
end Reinitialize;
```

3.3.6.7.9.7.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.10.1.8 LIMITATIONS

None.

3.3.6.7.9.7.10.2 REINITIALIZE UNIT DESIGN

This procedure is used to initialize the accumulator and previous value variable.

3.3.6.7.9.7.10.2.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.7.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Accumulatr_Value	Element_Type	in	Value to which accumulator will be initialized
Initial_Previous_Value	Element_Type	in	Value to which previous value will be initialized

3.3.6.7.9.7.10.2.4 LOCAL DATA

None.

3.3.6.7.9.7.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Accumulator_Value : in Element_Type;
                        Initial_Previous_Value    : in Element_Type) is
begin

   Previous_Value := Initial_Previous_Value;

   Accumulator := Initial_Accumulator_Value;

end Reinitialize;
```

3.3.6.7.9.7.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.10.2.8 LIMITATIONS

None.

3.3.6.7.9.7.10.3 ACCUMULATE_CHANGE UNIT DESIGN

This unit is a function which accumulates the change in the subject variable.

3.3.6.7.9.7.10.3.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.7.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
New_Value	Element_Type	in	New value of the subject variable

3.3.6.7.9.7.10.3.4 LOCAL DATA

None.

3.3.6.7.9.7.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Accumulate_Change (New_Value : in Element_Type) is
   begin

--     -- Add change to accumulator and reset PV

       Accumulator := Accumulator + New_Value - Previous_Value;
       Previous_Value := New_Value;

   end Accumulate_Change;
```

3.3.6.7.9.7.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.10.3.8 LIMITATIONS

None.

3.3.6.7.9.7.10.4 ACCUMULATE_CHANGE UNIT DESIGN

This unit is a procedure which accumulates the change in the subject variable and returns the new value of the accumulator.

3.3.6.7.9.7.10.4.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.7.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
New_Value	Element_Type	in	New value of the subject variable
Retrieved_Accumulator_Value	Element_Type	out	New accumulator value

3.3.6.7.9.7.10.4.4 LOCAL DATA

None.

3.3.6.7.9.7.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Accumulate_Change
               (New_Value                    : in  Element_Type;
                Retrieved_Accumulator_Value : out Element_Type) is
   begin

--     -- Add change to accumulator and reset PV

       Accumulator    := Accumulator + New_Value - Previous_Value;
       Previous_Value := New_Value;

--     -- Return new accumulated value
```

```
      Retrieved_Accumulator_Value := Accumulator;

   end Accumulate_Change;
```

3.3.6.7.9.7.10.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.10.4.8 LIMITATIONS

None.

3.3.6.7.9.7.10.5 RETRIEVE_ACCUMULATION UNIT DESIGN

This unit is a function which returns the current value of the accumulated change to the subject variable.

3.3.6.7.9.7.10.5.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.7.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
<returned value>	Element_Type	out	Current value of Accumulator

3.3.6.7.9.7.10.5.4 LOCAL DATA

None.

3.3.6.7.9.7.10.5.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
function Retrieve_Accumulation return Element_Type is
begin
   return Accumulator;
end Retrieve_Accumulation;
```

3.3.6.7.9.7.10.5.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.10.5.8 LIMITATIONS

None.

3.3.6.7.9.7.10.6 RETRIEVE_PREVIOUS_VALUE UNIT DESIGN

This unit is a function which returns the value of the Previous Value variable.

3.3.6.7.9.7.10.6.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.7.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.7.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
<returned value>	Element_Type	out	Current value of PV variable

3.3.6.7.9.7.10.6.4 LOCAL DATA

None.

3.3.6.7.9.7.10.6.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.7.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
function Retrieve_Previous_Value return Element_Type is
begin
   return Previous_Value;
end Retrieve_Previous_Value;
```

3.3.6.7.9.7.10.6.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.7.10.6.8 LIMITATIONS

None.

3.3.6.7.9.8 CHANGE_CALCULATOR PACKAGE DESIGN (CATALOG #P35-0)

This generic package provides a set of operations for tracking the change in a given variable.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R113.

3.3.6.7.9.8.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.8.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following data types are defined in the LLCSC specification:

Name	Base Type	Description
Element_Type	generic float	Type of the variable being trackd

3.3.6.7.9.8.4 LOCAL DATA

Data objects:

The following data objects are maintained by this part:

Name	Type	Description
Previous_Value	Element_Type	Value which is stored for comparison later on

3.3.6.7.9.8.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.8.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

package body Change_Calculator is

-- -- Declare previous value object and initialize it to value specified
-- -- during instantiation

   Previous_Value       : Element_Type := Initial_Value;


end Change_Calculator;
```

3.3.6.7.9.8.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.8.8 LIMITATIONS

None.

3.3.6.7.9.8.9 LLCSC DESIGN

None.

3.3.6.7.9.8.10 UNIT DESIGN

3.3.6.7.9.8.10.1 REINITIALIZE UNIT DESIGN

This procedure is used to reinitialize the previous value variable.

3.3.6.7.9.8.10.1.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.8.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.8.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Value	Element_Type	in	Value to which Prev_Value will be initialized

3.3.6.7.9.8.10.1.4 LOCAL DATA

None.

3.3.6.7.9.8.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.8.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Value : in Element_Type) is
begin
   Previous_Value := Initial_Value;
end Reinitialize;
```

3.3.6.7.9.8.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.8.10.1.8 LIMITATIONS

None.

3.3.6.7.9.8.10.2 CHANGE UNIT DESIGN

This unit is a function which -- computes and returns the change in the Previous_Value variable since the last call to this function; and -- set the value of Previous_Value to the input value.

3.3.6.7.9.8.10.2.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.8.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.8.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
New_Value	Element_Type	in	Value to which Prev_Value is compared and set to
<returned value>	Element_Type	out	Change in value since the last call to this functn

3.3.6.7.9.8.10.2.4 LOCAL DATA

Data objects:

The following data objects are maintained by this unit.

Name	Type	Description
Change_Amount	Element_Type	Calculated change since Previous_Val was last updated.

3.3.6.7.9.8.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.8.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   function Change (New_Value : Element_Type) return Element_Type is

      Change_Amount : Element_Type;

   begin

--    -- Calculate change and update Previous_Value

      Change_Amount  := New_Value - Previous_Value;
      Previous_Value := New_Value;

--    -- return Change

      return Change_Amount;

   end Change;
```

3.3.6.7.9.8.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.8.10.2.8 LIMITATIONS

None.

3.3.6.7.9.8.10.3 RETRIEVE_VALUE UNIT DESIGN

This unit is a function which returns the current value of the Previous_Value variable.

3.3.6.7.9.8.10.3.1 REQUIREMENTS ALLOCATION

None.

3.3.6.7.9.8.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.8.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
<returned value>	Element_Type	out	Current value of Prev_Value

3.3.6.7.9.8.10.3.4 LOCAL DATA

None.

3.3.6.7.9.8.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.8.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
function Retrieve_Value return Element_Type is
begin
   return Previous_Value;
end Retrieve_Value;
```

3.3.6.7.9.8.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.8.10.3.8 LIMITATIONS

None.

3.3.6.7.9.9 INTEGRATOR (BODY) PACKAGE DESIGN (CATALOG #P37-0)

This generic package provides the capability to manage a data value and integrate it across time.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.9.1 REQUIREMENTS ALLOCATION

This LLCSC meets CAMP requirement R124.

3.3.6.7.9.9.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.9.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Dependent_Type	generic float	Type of the dependent variable
Independent_Type	generic float	Type of the independent variable
Time_Interval	generic float	Type of the delta time variable

Data objects:

The following objects were defined in the package specification:

Name	Type	Description
Initial_Independent_Value	Independent_Type	Initial value for independent variable
Initial_Dependent_Value	Dependent_Type	Initial value for dependent variable
Default_Delta_Time	Time_Interval	Default time between integration

3.3.6.7.9.9.4 LOCAL DATA

Data objects:

The following data objects are maintained by this package:

Name	Type	Description
Dependent_Value	Dependent_Type	Last value of the dependent variable
Previous_Independent_Value	Independent_Type	Last value of the independent var.

3.3.6.7.9.9.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.9.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

package body Integrator is

-- -- Declare the dependent and independent objects and initialize them to the
-- -- values specified at instantiation

   Dependent_Value            : Dependent_Type   := Initial_Dependent_Value;
   Previous_Independent_Value : Independent_Type := Initial_Independent_Value;

end Integrator;
```

3.3.6.7.9.9.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.9.8 LIMITATIONS

None.

3.3.6.7.9.9.9 LLCSC DESIGN

None.

3.3.6.7.9.9.10 UNIT DESIGN

3.3.6.7.9.9.10.1 REINITIALIZE UNIT DESIGN

This procedure reinitializes the integrator.

3.3.6.7.9.9.10.1.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.9.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.9.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Initial_Dependent_Value	Dependent_Type	in	Initial value for dependent variable
Initial_Independent_Value	Independent_Type	in	Initial value for independent variable

3.3.6.7.9.9.10.1.4 LOCAL DATA

None.

3.3.6.7.9.9.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.9.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reinitialize (Initial_Dependent_Value   : in Dependent_Type;
                        Initial_Independent_Value : in Independent_Type) is
begin

   Dependent_Value   := Initial_Dependent_Value;
   Previous_Independent_Value := Initial_Independent_Value;

end Reinitialize;
```

3.3.6.7.9.9.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.9.10.1.8 LIMITATIONS

None.

3.3.6.7.9.9.10.2 UPDATE UNIT DESIGN

This procedure updates the value of the independent value without integrating the dependent value across time.

3.3.6.7.9.9.10.2.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.9.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.9.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Current_Independent_Value	Independent_Type	in	Initial value for independent variable

3.3.6.7.9.9.10.2.4 LOCAL DATA

None.

3.3.6.7.9.9.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.9.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Update (Current_Independent_Value : in Independent_Type) is
begin

   Previous_Independent_Value := Current_Independent_Value;

end Update;
```

3.3.6.7.9.9.10.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.9.9.10.2.8 LIMITATIONS

None.

3.3.6.7.9.9.10.3 INTEGRATE UNIT DESIGN

This function performs a trapezoidal integration across the specified time interval (the time interval can default).

3.3.6.7.9.9.10.3.1 REQUIREMENTS ALLOCATION

See package specification.

3.3.6.7.9.9.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.9.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Current_Independent_Value	Independent_Type	in	New value of independent variable
Delta_Time	Time_Interval	in	Time interval to integrate
<returned value>	Dependent_Type	out	New value of dependent variable

3.3.6.7.9.9.10.3.4 LOCAL DATA

None.

3.3.6.7.9.9.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.9.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   function Integrate (Current_Independent_Value : Independent_Type;
                       Delta_Time  : Time_Interval := Default_Delta_Time)
                      return Dependent_Type is
   begin

--     -- Integrate new independent value across specified time interval

       Dependent_Value := Dependent_Value
               + ((Previous_Independent_Value + Current_Independent_Value) * 0.5)
               * Delta_Time;
```

```
--      -- Update previous independent value and return new dependent value

        Previous_Independent_Value := Current_Independent_Value;
        return Dependent_Value;

    end Integrate;
```

3.3.6.7.9.9.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements defined in this part's parent component and used by this part:

Data types:

The following table summarizes the LLC types required by this part:

Name	Description
Dependent_Type	Type of the dependent variable
Independent_Type	Type of the independent variable
Time_Interval	Type of the delta time variable

Data objects:

The following table summarizes the LLC objects required by this part:

Name	Type	Description
Default_Delta_Time	Time_Interval	Default time interval sepecified during package instantiation
Dependent_Value	Dependent_Type	Last value of the dependent variable
Previous_Independent_Value	Independent_Type	Last value of the independent var.

3.3.6.7.9.9.10.3.8 LIMITATIONS

None.

3.3.6.7.9.10 TWO_WAY_TABLE_LOOKUP (BODY) PACKAGE DESIGN (CATALOG #P1078-0)

This package provides a general two way table lookup. These routines allow the table to be created and initialized, or an already existing table may be used. Either variable type may be looked up in the table. The routines return a single value, interpolated or extrapolated as necessary.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.6.7.9.10.1 REQUIREMENTS ALLOCATION

N/A

3.3.6.7.9.10.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.9.10.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Indices	INTEGER or Enumerated	Type to index the table
X_Values	FLOAT	Type of 1 table value
Y_Values	FLOAT	Type of other table value
Real	FLOAT	Type for intermediate calculations

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
"/"	function	Divide operator for X_Values / Y_Values => Real
"/"	function	Divide operator for Y_Values / X_Values => Real
"/"	function	Divide operator for X_Values / X_Values => Y_Values
"/"	function	Divide operator for Y_Values / Y_Values => X_Values
"*"	function	Multiply operator for X_Values * Y_Values => Real
"*"	function	Multiply operator for Y_Values * X_Values => Real

3.3.6.7.9.10.4 LOCAL DATA

Data types:

The following table describes the data types defined by this part:

Name	Range	Description
X_Arrays	Indices	Array type for 1 type of values in table
Y_Arrays	Indices	Array type for other type of values in table
Tables	Indices	Type for Table of X and Y arrays

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Description
Table	record	Table of X_Array and Y_Array to be operated on

3.3.6.7.9.10.5 PROCESS CONTROL

Not applicable.

3.3.6.7.9.10.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)
package body Two_Way_Table_Lookup is

   function Inter_or_Extra_X is new General_Purpose_Math.
                                     Interpolate_or_Extrapolate
                                        (Independent_Type => Y_Values,
                                         Dependent_Type   => X_Values);

   function Inter_or_Extra_Y is new General_Purpose_Math.
                                     Interpolate_or_Extrapolate
                                        (Independent_Type => X_Values,
                                         Dependent_Type   => Y_Values);

   procedure Initialize( Table : out Tables;
                         Index : in  Indices;
                         X     : in  X_Values;
                         Y     : in  Y_Values ) is

   begin
      Table.Table_X( Index ) := X;
      Table.Table_Y( Index ) := Y;
   end Initialize;

   function Lookup_X ( Table : Tables;
                       Input : Y_Values ) return X_Values is

      Higher_Dept   : X_Values;
      Higher_Indept : Y_Values;
      Index         : Indices;
```

```
      Lower_Dept     : X_Values;
      Lower_Indept   : Y_Values;
      Result         : X_Values;

   begin

--    -- if the input value is out of bounds, then extrapolate for the
--    -- required value

      if Input < Table.Table_Y( Indices'FIRST) then
         Lower_Indept:= Table.Table_Y( Indices'FIRST );
         Higher_Indept := Table.Table_Y( Indices'SUCC(Indices'FIRST) );
         Lower_Dept:= Table.Table_X( Indices'FIRST );
         Higher_Dept := Table.Table_X( Indices'SUCC(Indices'FIRST) );
         Result := Inter_or_Extra_X( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );
      elsif Input > Table.Table_Y( Indices'LAST) then
         Higher_Indept := Table.Table_Y( Indices'LAST );
         Lower_Indept := Table.Table_Y( Indices'PRED(Indices'LAST) );
         Higher_Dept := Table.Table_X( Indices'LAST );
         Lower_Dept := Table.Table_X( Indices'PRED(Indices'LAST) );
         Result := Inter_or_Extra_X( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );

      else      -- the input value is in the bounds of the table

--       -- Search through the table until the independent entry is greater than
--       -- or equal to the Key (Input) value.  Note that the value will be
--       -- found in the table since out-of-bounds values have already been
--       -- checked for.  Start with the successor of the first element so that
--       -- PRED will be defined if Input is equal to the first element.

         Index := Indices'SUCC( Indices'FIRST );
         Find_Value:
            loop
               exit when (Table.Table_Y( Index ) >= Input);
               Index := Indices'SUCC( Index );
            end loop Find_Value;

--       -- Save the table values.

         Lower_Indept := Table.Table_Y( Indices'PRED(Index) );
         Lower_Dept   := Table.Table_X( Indices'PRED(Index) );

         Higher_Indept := Table.Table_Y( Index );
         Higher_Dept   := Table.Table_X( Index );

--       -- Check to see if the value needs to be interpolated - that is,
--       -- if the value is not equal to one of the boundary values.

         if (Input = Higher_Indept) then
```

```
            Result := Higher_Dept;
         elsif (Input = Lower_Indept) then
            Result := Lower_Dept;
         else                -- Value is between, so interpolate
            Result := Inter_or_Extra_X( Input             => Input,
                                        Lower_Independent  => Lower_Indept,
                                        Higher_Independent => Higher_Indept,
                                        Lower_Dependent    => Lower_Dept,
                                        Higher_Dependent   => Higher_Dept );
         end if;             -- Value is between, so interpolate
      end if;       -- the input value is in the bounds of the table
      return Result;
   end Lookup_X;

   function Lookup_Y ( Table : Tables;
                       Input : X_Values ) return Y_Values is

      Higher_Dept   : Y_Values;
      Higher_Indept : X_Values;
      Index         : Indices;
      Lower_Dept    : Y_Values;
      Lower_Indept  : X_Values;
      Result        : Y_Values;

   begin

--    -- if the input value is out of bounds, then extrapolate for the
--    -- required value

      if Input < Table.Table_X( Indices'FIRST) then
         Lower_Indept:= Table.Table_X( Indices'FIRST );
         Higher_Indept := Table.Table_X( Indices'SUCC(Indices'FIRST) );
         Lower_Dept:= Table.Table_Y( Indices'FIRST );
         Higher_Dept := Table.Table_Y( Indices'SUCC(Indices'FIRST) );
         Result := Inter_or_Extra_Y( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );
      elsif Input > Table.Table_X( Indices'LAST) then
         Higher_Indept := Table.Table_X( Indices'LAST );
         Lower_Indept := Table.Table_X( Indices'PRED(Indices'LAST) );
         Higher_Dept := Table.Table_Y( Indices'LAST );
         Lower_Dept := Table.Table_Y( Indices'PRED(Indices'LAST) );
         Result := Inter_or_Extra_Y( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );

      else      -- the input value is in the bounds of the table

--       -- Search through the table until the independent entry is greater than
--       -- or equal to the Key (Input) value.  Note that the value will be
--       -- found in the table since out-of-bounds values have already been
--       -- checked for.  Start with the successor of the first element so that
--       -- PRED will be defined if Input is equal to the first element.
```

```
         Index := Indices'SUCC( Indices'FIRST );
         Find_Value:
            loop
               exit when (Table.Table_X( Index ) >= Input);
               Index := Indices'SUCC( Index );
            end loop Find_Value;

--       -- Save the table values.

         Lower_Indept := Table.Table_X( Indices'PRED(Index) );
         Lower_Dept   := Table.Table_Y( Indices'PRED(Index) );

         Higher_Indept := Table.Table_X( Index );
         Higher_Dept   := Table.Table_Y( Index );

--       -- Check to see if the value needs to be interpolated - that is,
--       -- if the value is not equal to one of the boundary values.

         if (Input = Higher_Indept) then
            Result := Higher_Dept;
         elsif (Input = Lower_Indept) then
            Result := Lower_Dept;
         else                -- Value is between, so interpolate
            Result := Inter_or_Extra_Y( Input             => Input,
                                        Lower_Independent  => Lower_Indept,
                                        Higher_Independent => Higher_Indept,
                                        Lower_Dependent    => Lower_Dept,
                                        Higher_Dependent   => Higher_Dept );
         end if;             -- Value is between, so interpolate
      end if;       -- the input value is in the bounds of the table
      return Result;
   end Lookup_Y;

end Two_Way_Table_Lookup;
```

3.3.6.7.9.10.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined elsewhere in the parent top level component:

Name	Type	Source	Description
Interpolate_or_Extrapolate	function	General_ Purpose_ Math	Allows interpolation or extrapolation to return a single value

3.3.6.7.9.10.8 LIMITATIONS

None.

3.3.6.7.9.10.9 LLCSC DESIGN

None.

3.3.6.7.9.10.10 UNIT DESIGN

None.

3.3.6.7.10 UNIT DESIGN

3.3.6.7.10.1 INTERPOLATE_OR_EXTRAPOLATE (BODY) UNIT DESIGN (CATALOG #P39-0)

This part is a generic function which computes the linear interpolation between two values.

3.3.6.7.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R116.

3.3.6.7.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Independent_Type	generic float	Type of the independent variables
Dependent_Type	generic float	Type of the dependent variable
Dependent_over_Independent_Type	generic float	Result of Dependent / Independent

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
"/"	function	Dependent_Over_Independent_Type := Dependent_Type / Independent_Type
"*"	function	Dependent_Type := Dependent_Over_Independent_Type * Independent_Type

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Input	Independent	in	Independent value for which a dependent value is returned
Lower_Independent	Independent	in	Lower independent value
Higher_Independent	Independent	in	Higher independent value
Lower_Dependent	Dependent	in	Lower dependent value
Higher_Dependent	Dependent	in	Higher dependent value
<return value>	Dependent	out	Computed interpolated value

3.3.6.7.10.1.4 LOCAL DATA

None.

3.3.6.7.10.1.5 PROCESS CONTROL

Not applicable.

3.3.6.7.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

function Interpolate_or_Extrapolate
            (Input              : in Independent_Type;
             Lower_Independent  : in Independent_Type;
             Higher_Independent : in Independent_Type;
             Lower_Dependent    : in Dependent_Type;
             Higher_Dependent   : in Dependent_Type)
            return Dependent_Type is

begin

   return Lower_Dependent +
          ((Input - Lower_Independent) / (Higher_Independent - Lower_Independent)
          * (Higher_Dependent - Lower_Dependent));

end Interpolate_or_Extrapolate;
```

3.3.6.7.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.10.1.8 LIMITATIONS

None.

3.3.6.7.10.2 SQUARE_ROOT (BODY) UNIT DESIGN (CATALOG #P40-0)

This generic function computes the Square Root of a value.

3.3.6.7.10.2.1 REQUIREMENTS ALLOCATION

This LLCSC meets CAMP requirement R123.

3.3.6.7.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.10.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Type	Description
Inputs	floating point	Data type of input values
Outputs	floating point	Data type of output values
Real	floating point	Unconstrained type for intermediate calculations

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Input	Inputs	in	Input value to square root operation
<return value>	Outputs	out	Result of square root operation

3.3.6.7.10.2.4 LOCAL DATA

Data objects:

The following table summarizes the data objects maintained by the Sqrt function contained in this package:

Name	Type	Description
Answer	Outputs	Square root of input value

3.3.6.7.10.2.5 PROCESS CONTROL

Not applicable.

3.3.6.7.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
with Polynomials;
separate (General_Purpose_Math)
package body Square_Root Is

   package System is new Polynomials.System_Functions.Square_Root
                            (Inputs  => Real,
                             Outputs => Real);

   function Sqrt (Input : Inputs) return Outputs is
      Answer : Outputs;
   begin
      Answer := Outputs(System.Sqrt(Real(Input)));
      return Answer;
   exception
      when others => raise Negative_Input;
   end Sqrt;

end Square_Root;
```

3.3.6.7.10.2.7 UTILIZATION OF OTHER ELEMENTS

The following Library Units are with'ed by this unit:

1. Polynomials (Poly). contains Square Root function

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

Name	Type	Source	Description
Sqrt	function	Poly	Gives Square Root of an input value

3.3.6.7.10.2.8 LIMITATIONS

The exception "Negative_Input" is raised if "Input" is negative.

3.3.6.7.10.3 ROOT_SUM_OF_SQUARES (BODY) UNIT DESIGN (CATALOG #P41-0)

This generic function computes the root sum of squares of three input input values.

3.3.6.7.10.3.1 REQUIREMENTS ALLOCATION

This LLCSC meets CAMP requirement R228.

3.3.6.7.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.10.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Real_Type	generic float	Type of the resultant variable
Squared_Type	generic float	Type of the input variable

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
Sqrt	function	computes the square root of an input variable

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
X	Real_Type	in	First of the three input vars
Y	Real_Type	in	Second of the three input vars
Z	Real_Type	in	Third of the three input vars
<return value>	Real_Type	out	Resultant root sum of squares

3.3.6.7.10.3.4 LOCAL DATA

None.

3.3.6.7.10.3.5 PROCESS CONTROL

Not applicable.

3.3.6.7.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

function Root_Sum_Of_Squares (X : Real_Type;
                              Y : Real_Type;
                              Z : Real_Type)
                             return Real_Type is
begin

   return Sqrt (X*X + Y*Y + Z*Z);

end Root_Sum_Of_Squares;
```

3.3.6.7.10.3.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.10.3.8 LIMITATIONS

None.

3.3.6.7.10.4 SIGN (BODY) UNIT DESIGN (CATALOG #P42-0)

This unit is a generic function which determines the sign of an input value; it returns -1 if input is negative, 1 if non-negative

3.3.6.7.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R224.

3.3.6.7.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.10.4.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Real_Type	generic float	Type of the input variable

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Input	Real_Type	in	Number for which absolute value is to be computed
<return value>	INTEGER	out	-1 if negative; 1 otherwise

3.3.6.7.10.4.4 LOCAL DATA

Data objects:

The following data objects are maintained by this unit:

Name	Type	Description
Result_Value	INTEGER	Stores result

3.3.6.7.10.4.5 PROCESS CONTROL

Not applicable.

3.3.6.7.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)
function Sign (Input_Variable : Real_Type) return INTEGER is
   Result_Value : INTEGER;
begin
   if Input_Variable >= 0.0 then
      Result_Value := 1;
   else
      Result_Value := -1;
   end if;
   return Result_Value;
end Sign;
```

3.3.6.7.10.4.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.10.4.8 LIMITATIONS

None.

3.3.6.7.10.5 MEAN_VALUE (BODY) UNIT DESIGN (CATALOG #P43-0)

This unit is a generic function which computes the average value of a vector of numbers. The vector is unconstrained; i.e., a different number of elements can be averaged each time that the function is called.

3.3.6.7.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R144.

3.3.6.7.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.10.5.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Element_Type	generic float	Type of the elements averaged
Index_Type	Discrete	Type of index to vector
Vector_Type	ARRAY	Array of "Element_Type" with "Index_Type" as the index

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
"/"	function	Element_Type := Element_Type / INTEGER

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Value_Vector	Vector_Type	in	Values to be averaged
<return value>	Element_Type	in	Average of input values

3.3.6.7.10.5.4 LOCAL DATA

Data objects:

The following data objects are maintained by this unit:

Name	Type	Description
Running_Sum	Element_Type	Running sum of the vector's values

3.3.6.7.10.5.5 PROCESS CONTROL

Not applicable.

3.3.6.7.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)
function Mean_Value (Value_Vector : Vector_Type) return Element_Type is
   Running_Sum : Element_Type := 0.0;
begin
-- -- Sum the values in the input vector
   for Index in Value_Vector'RANGE loop
      Running_Sum := Running_Sum + Value_Vector (Index);
   end loop;
-- -- Calculate and return the average value
   return Running_Sum / Element_Type (Value_Vector'LENGTH);
end Mean_Value;
```

3.3.6.7.10.5.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.10.5.8 LIMITATIONS

None.

3.3.6.7.10.6 MEAN_ABSOLUTE_DIFFERENCE (BODY) UNIT DESIGN (CATALOG #P44-0)

This unit is a generic function which computes the mean absolute difference (MAD) of a vector, i.e., Avg (Abs (Xi - Xavg))

3.3.6.7.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R143.

3.3.6.7.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.6.7.10.6.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following types were defined in the package specification:

Name	Base Type	Description
Element_Type	generic float	Type of the elements averaged
Index_Type	Discrete	Type of index to vector
Vector_Type	ARRAY	Array of "Element_Type" with "Index_Type" as the index

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
"/"	function	Element_Type := Element_Type / INTEGER

FORMAL PARAMETERS:

The following table describes this unit's formal parameters:

Name	Type	Mode	Description
Value_Vector	Vector_Type	in	Input values
<return value>	Element_Type	in	MAD of input vector

3.3.6.7.10.6.4 LOCAL DATA

Data objects:

The following data objects are maintained by this unit:

Name	Type	Description
Average	Element_Type	Average value of the input vector
One_Over_N	Element_Type	1 / N (N = _# of values in the input vector)
Running_Sum	Element_Type	Running sum of the vector's values
Total_Difference	Element_Type	Running sum of the absolute difference between Xi and Xavg

3.3.6.7.10.6.5 PROCESS CONTROL

Not applicable.

3.3.6.7.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
separate (General_Purpose_Math)

function Mean_Absolute_Difference (Value_Vector : Vector_Type)
                                   return Element_Type is

   Average            : Element_Type;
   One_Over_N         : Element_Type;
   Running_Sum        : Element_Type := 0.0;
   Total_Difference   : Element_Type := 0.0;

begin

-- -- Set up static value used twice

   One_Over_N := 1.0 / Element_Type (Value_Vector'Length);


-- -- Calculate the average vector value

   for Index in Value_Vector'RANGE loop

      Running_Sum := Running_Sum + Value_Vector (Index);

   end loop;

   Average := Running_Sum * One_Over_N;


-- -- Calculate and return the MAD

   for Index in Value_Vector'RANGE loop

      Total_Difference := Total_Difference + Abs (Value_Vector (Index) - Average);

   end loop;
```

```
   return Total_Difference * One_Over_N;
end Mean_Absolute_Difference;
```

3.3.6.7.10.6.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.6.7.10.6.8 LIMITATIONS

None.

(This page left intentionally blank.)

```
with Polynomials;
package body General_Purpose_Math is

   package body Lookup_Table_Even_Spacing   is separate;

   package body Lookup_Table_Uneven_Spacing is separate;

   package body Two_Way_Table_Lookup        is separate;

   package body Incrementor                 is separate;

   package body Decrementor                 is separate;

   package body Running_Average             is separate;

   package body Change_Calculator           is separate;

   package body Accumulator                 is separate;

   package body Change_Accumulator          is separate;

   package body Integrator                  is separate;

   function Interpolate_Or_Extrapolate
               (Input             : in Independent_Type;
                Lower_Independent : in Independent_Type;
                Higher_Independent: in Independent_Type;
                Lower_Dependent   : in Dependent_Type;
                Higher_Dependent  : in Dependent_Type)
               return Dependent_Type is separate;

   package body Square_Root is separate;

   function Root_Sum_Of_Squares (X : Real_Type;
                                 Y : Real_Type;
                                 Z : Real_Type)
                                return Real_Type is separate;

   function Sign (Input_Variable : Real_Type) return INTEGER is separate;

   function Mean_Value (Value_Vector : Vector_Type)
                        return Element_Type is separate;

   function Mean_Absolute_Difference (Value_Vector : Vector_Type)
                                      return Element_Type is separate;

end General_Purpose_Math;
```

```
separate (General_Purpose_Math)

package body Lookup_Table_Even_Spacing is

-- -- Declare table step variable and initialize it to the value of the step
-- -- between each successive independent entry


   Internal_Min_Independent_Value : constant Independent_Type
                                          := Minimum_Independent_Value;
   Internal_Max_Independent_Value : constant Independent_Type
                                          := Maximum_Independent_Value;

   Table_Step : Independent_Type :=
                  (Internal_Max_Independent_Value - Internal_Min_Independent_Value)
                      / Independent_Type (Tables'LENGTH - 1);
                                                                    pragma PAGE;
   procedure Initialize (Table           : out Tables;
                         INDEX           : in  Index_Type;
                         Dependent_Value : in  Dependent_Type) is
   begin

     Table (INDEX) := Dependent_Value;

   end Initialize;
                                                                    pragma PAGE;
   procedure Lookup
               (Table              : in  Tables;
                Key                : in  Independent_Type;
                Lower_Independent  : out Independent_Type;
                Higher_Independent : out Independent_Type;
                Lower_Dependent    : out Dependent_Type;
                Higher_Dependent   : out Dependent_Type) is

      INDEX                  : Index_Type;
      Temp_Lower_Independent : Independent_Type;
      Index_Float_Var        : Independent_Type;
      Index_Integer_Var      : INTEGER;

   begin

--    -- Raise exception if Key is outside of table range

      if Key < Internal_Min_Independent_Value or Key > Internal_Max_Independent_Value the
         raise Value_Out_Of_Range;
      end if;

--    -- Map the key to an index value

      Index_Float_Var := (Key - Internal_Min_Independent_Value) / Table_Step;
      Index_Integer_Var := INTEGER (Index_Float_Var);

      if Index_Float_Var >= Independent_Type (Index_Integer_Var) then
        Index_Integer_Var := Index_Integer_Var + 1;
      end if;

      INDEX := Index_Type'VAL (Index_Integer_Var);
```

```
--      -- Adjust the index value if the key was mapped to the last entry

        if INDEX = Index_Type'LAST then
           INDEX := Index_Type'PRED (INDEX);
        end if;


--      -- Return the table values

        Temp_Lower_Independent := Internal_Min_Independent_Value +
                                  (Independent_Type (Index_Type'POS (INDEX) - Index_Type'P(
(Index_Type'FIRST)) *
                                   Table_Step);

        Lower_Independent  := Temp_Lower_Independent;
        Higher_Independent := Temp_Lower_Independent + Table_Step;

        Lower_Dependent  := Table (INDEX);
        Higher_Dependent := Table (Index_Type'SUCC(INDEX));

     end Lookup;
                                                                    pragma PAGE;
     procedure Lookup
                 (Table              : in  Tables;
                  Key                : in  Independent_Type;
                  Lower_Independent  : out Independent_Type;
                  Higher_Independent : out Independent_Type;
                  Lower_Dependent    : out Dependent_Type;
                  Higher_Dependent   : out Dependent_Type;
                  Key_Location       : out Key_Range_Flag) is

        INDEX                : Index_Type;
        Temp_Lower_Independent : Independent_Type;
        Index_Float_Var      : Independent_Type;
        Index_Integer_Var    : INTEGER;

     begin

--      -- Return the first two rows of the lookup table if Key is below table range

        if Key < Internal_Min_Independent_Value then

           Key_Location := Below_Table_Range;

           Lower_Independent  := Internal_Min_Independent_Value;
           Higher_Independent := Internal_Min_Independent_Value + Table_Step;

           Lower_Dependent  := Table (Index_Type'FIRST);
           Higher_Dependent := Table (Index_Type'SUCC(Index_Type'FIRST));

--      -- Return the last two rows of the lookup table if Key is above table range

        elsif Key > Internal_Max_Independent_Value then

           Key_Location := Above_Table_Range;
```

```
            Lower_Independent  := Internal_Max_Independent_Value - Table_Step;
            Higher_Independent := Internal_Max_Independent_Value;

            Lower_Dependent  := Table (Index_Type'PRED(Index_Type'LAST));
            Higher_Dependent := Table (Index_Type'LAST);


--      -- Key is in table range, so map the key to an index value and return the
--      -- corresponding values

         else

            Key_Location := In_Table_Range;


--          -- Map the Key to an Index value

            Index_Float_Var := (Key - Internal_Min_Independent_Value) / Table_Step;
            Index_Integer_Var := INTEGER (Index_Float_Var);

            if Index_Float_Var >= Independent_Type (Index_Integer_Var) then
              Index_Integer_Var := Index_Integer_Var + 1;
            end if;

            INDEX := Index_Type'VAL (Index_Integer_Var);


--          -- Adjust the Index if it has been mapped to the last entry

            if INDEX = Index_Type'LAST then
               INDEX := Index_Type'PRED (INDEX);
            end if;

--          -- Return the table values

            Temp_Lower_Independent := Internal_Min_Independent_Value +
                                      (Independent_Type (Index_Type'POS (INDEX) - Index_Type
POS(Index_Type'FIRST)) *
                                       Table_Step);

            Lower_Independent  := Temp_Lower_Independent;
            Higher_Independent := Temp_Lower_Independent + Table_Step;

            Lower_Dependent  := Table (INDEX);
            Higher_Dependent := Table (Index_Type'SUCC(INDEX));

         end if;

      end Lookup;

   end Lookup_Table_Even_Spacing;
```

```
separate (General_Purpose_Math)

package body Lookup_Table_Uneven_Spacing is
                                                                    pragma PAGE;
   procedure Initialize
               (Table             : out Tables;
                INDEX             : in  Index_Type;
                Independent_Value : in  Independent_Type;
                Dependent_Value   : in  Dependent_Type) is
   begin

     Table (INDEX).Independent_Entry := Independent_Value;
     Table (INDEX).Dependent_Entry   := Dependent_Value;

   end Initialize;
                                                                    pragma PAGE;
   procedure Lookup
               (Table              : in  Tables;
                Key                : in  Independent_Type;
                Lower_Independent  : out Independent_Type;
                Higher_Independent : out Independent_Type;
                Lower_Dependent    : out Dependent_Type;
                Higher_Dependent   : out Dependent_Type) is

      INDEX : Index_Type;

   begin

--    -- Raise an exception if the input key is outside of the table range

      if Key < Table (Index_Type'FIRST).Independent_Entry or
         Key > Table (Index_Type'LAST).Independent_Entry then

         raise Value_Out_Of_Range;

      end if;


--    -- Search through the table until the independent entry is greater than
--    -- or equal to the Key or have gone through the entire table (actually
--    -- go through one less row than the whole table since we return two rows)

      INDEX := Index_Type'SUCC(Index_Type'FIRST);

      loop

         exit when (Table (INDEX).Independent_Entry >= Key);

         INDEX := Index_Type'SUCC (INDEX);

      end loop;


--    -- Return the table values

      Lower_Independent := Table (Index_Type'PRED(INDEX)).Independent_Entry;
      Lower_Dependent   := Table (Index_Type'PRED(INDEX)).Dependent_Entry;
```

```
      Higher_Independent := Table (INDEX).Independent_Entry;
      Higher_Dependent   := Table (INDEX).Dependent_Entry;

   end Lookup;

                                                                  pragma PAGE;
   procedure Lookup
               (Table              : in  Tables;
                Key                : in  Independent_Type;
                Lower_Independent  : out Independent_Type;
                Higher_Independent : out Independent_Type;
                Lower_Dependent    : out Dependent_Type;
                Higher_Dependent   : out Dependent_Type;
                Key_Location       : out Key_Range_Flag) is

      INDEX : Index_Type;

   begin

--       -- This routine first checks to make sure that the Key is within the Table
--       -- range; if it is below, return the first two rows; if it is above, return
--       -- the last two rows; if it is within the range, find which rows it is between
--       -- and return those two rows

      if Key < Table (Index_Type'FIRST).Independent_Entry then

--          -- Return the first two rows of the table if key is less than first entry

         Key_Location := Below_Table_Range;

         Lower_Independent  := Table (Index_Type'FIRST). Independent_Entry;
         Higher_Independent := Table (Index_Type'SUCC (Index_Type'FIRST)). Independent_Er
ry;

         Lower_Dependent  := Table (Index_Type'FIRST). Dependent_Entry;
         Higher_Dependent := Table (Index_Type'SUCC (Index_Type'FIRST)). Dependent_Entry;

      elsif Key > Table (Index_Type'LAST).Independent_Entry then

--          -- Return the last two rows of the table if key is greater than last entry

         Key_Location := Above_Table_Range;

         Lower_Independent  := Table (Index_Type'PRED (Index_Type'LAST)). Independent_Ent
y;
         Higher_Independent := Table (Index_Type'LAST). Independent_Entry;

         Lower_Dependent  := Table (Index_Type'PRED (Index_Type'LAST)). Dependent_Entry;
         Higher_Dependent := Table (Index_Type'LAST). Dependent_Entry;

      else

--          -- Search through the table until the independent entry is greater than
--          -- or equal to the Key or have gone through the entire table

         Key_Location := In_Table_Range;

         INDEX := Index_Type'SUCC(Index_Type'FIRST);
```

```
         loop

            exit when (Table (INDEX).Independent_Entry >= Key);

            INDEX := Index_Type'SUCC (INDEX);

         end loop;


--       -- Return the table values

         Lower_Independent := Table (Index_Type'PRED(INDEX)).Independent_Entry;
         Lower_Dependent   := Table (Index_Type'PRED(INDEX)).Dependent_Entry;

         Higher_Independent := Table (INDEX).Independent_Entry;
         Higher_Dependent   := Table (INDEX).Dependent_Entry;

      end if;

   end Lookup;

end Lookup_Table_Uneven_Spacing;
```

```
separate (General_Purpose_Math)

package body Incrementor is

-- -- Declare the Incrementor and amount of increment object and initialize them
-- -- to values given during instantiation

   Amount_Of_Increment  : Real_Type := Increment_Amount;
   Increment_Variable   : Real_Type := Initial_Value;
                                                                    pragma PAGE;
   procedure Reinitialize (Initial_Value    : in Real_Type;
                           Increment_Amount : in Real_Type) is
   begin

      Increment_Variable := Initial_Value;

      Amount_Of_Increment := Increment_Amount;

   end Reinitialize;
                                                                    pragma PAGE;
   function Increment return Real_Type is
   begin

--    -- Increment and return new value

      Increment_Variable := Increment_Variable + Amount_Of_Increment;
      return Increment_Variable;

   end Increment;

end Incrementor;
```

```
separate (General_Purpose_Math)

package body Decrementor is

-- --------------
-- -Data Objects-
-- --------------

-- -- Each of these objects is initialized to the value given during instantiation

   Amount_Of_Decrement : Real_Type := Decrement_Amount;
   Decrement_Variable  : Real_Type := Initial_Value;
                                                                    pragma PAGE;
   procedure Reinitialize (Initial_Value    : in Real_Type;
                           Decrement_Amount : in Real_Type) is
   begin

      Decrement_Variable := Initial_Value;

      Amount_Of_Decrement := Decrement_Amount;

   end Reinitialize;
                                                                    pragma PAGE;
   function Decrement return Real_Type is
   begin

--    -- Decrement and return new value

      Decrement_Variable := Decrement_Variable - Amount_Of_Decrement;
      return Decrement_Variable;

   end Decrement;


end Decrementor;
```

```
separate (General_Purpose_Math)

package body Running_Average is

-- -- Declare Sum and Count variable and initialize them to values given at
-- -- instantiation

   Sum                    : Real_Type := Initial_Sum;
   COUNT                  : INTEGER := Initial_Count;
                                                                  pragma PAGE;
   procedure Reinitialize (Initial_Sum   : in Real_Type;
                           Initial_Count : in INTEGER) is
   begin

      Sum   := Initial_Sum;
      COUNT := Initial_Count;

   end Reinitialize;
                                                                  pragma PAGE;
   procedure Reinitialize (Initial_Count : in INTEGER) is
   begin

      COUNT := Initial_Count;

   end Reinitialize;
                                                                  pragma PAGE;
   function Current_Average (New_Value : Real_Type) return Real_Type is

   begin

--    -- Do average calculation

      Sum   := Sum + New_Value;
      COUNT := COUNT + 1;

      return Sum / COUNT;

   end Current_Average;


end Running_Average;
```

```
separate (General_Purpose_Math)

package body Accumulator is


-- -- Declare accumulator variable and set it to the value given during
-- -- instantiation

   Accumulated_Value     : Element_Type := Initial_Value;
                                                                    pragma PAGE;
   procedure Reinitialize (Initial_Value : in Element_Type) is

   begin

      Accumulated_Value     := Initial_Value;

   end Reinitialize;
                                                                    pragma PAGE;
   procedure Accumulate (New_Value : in Element_Type) is
   begin

--    -- Add input value to accumulator

      Accumulated_Value := Accumulated_Value + New_Value;

   end Accumulate;
                                                                    pragma PAGE;
   procedure Accumulate
           .   (New_Value       : in  Element_Type;
                Retrieved_Value : out Element_Type) is
   begin

--    -- Add input value to accumulator

      Accumulated_Value := Accumulated_Value + New_Value;

--    -- return new accumulator value

      Retrieved_Value := Accumulated_Value;

   end Accumulate;
                                                                    pragma PAGE;
   function Retrieve return Element_Type is

   begin

--    -- Retrieve Accumulator Value;

      return Accumulated_Value;

   end Retrieve;


end Accumulator;
```

```
separate (General_Purpose_Math)

package body Change_Accumulator is

-- -- Declare accumulator and previous value objects and initialize them to
-- -- values given during instantiation

   Accumulator                : Element_Type := Initial_Accumulator_Value;
   Previous_Value             : Element_Type := Initial_Previous_Value;
                                                                  pragma PAGE;
   procedure Reinitialize (Initial_Accumulator_Value : in Element_Type) is

   begin

      Accumulator := Initial_Accumulator_Value;

   end Reinitialize;
                                                                  pragma PAGE;
   procedure Reinitialize (Initial_Accumulator_Value : in Element_Type;
                           Initial_Previous_Value    : in Element_Type) is
   begin

      Previous_Value := Initial_Previous_Value;

      Accumulator := Initial_Accumulator_Value;

   end Reinitialize;
                                                                  pragma PAGE;
   procedure Accumulate_Change (New_Value : in Element_Type) is
   begin

--    -- Add change to accumulator and reset PV

      Accumulator := Accumulator + New_Value - Previous_Value;
      Previous_Value := New_Value;

   end Accumulate_Change;
                                                                  pragma PAGE;
   procedure Accumulate_Change
               (New_Value                  : in  Element_Type;
                Retrieved_Accumulator_Value : out Element_Type) is
   begin

--    -- Add change to accumulator and reset PV

      Accumulator    := Accumulator + New_Value - Previous_Value;
      Previous_Value := New_Value;

--    -- Return new accumulated value

      Retrieved_Accumulator_Value := Accumulator;

   end Accumulate_Change;
                                                                  pragma PAGE;
   function Retrieve_Accumulation return Element_Type is

   begin
```

```
      return Accumulator;
   end Retrieve_Accumulation;
                                                          pragma PAGE;
   function Retrieve_Previous_Value return Element_Type is
   begin
      return Previous_Value;
   end Retrieve_Previous_Value;

end Change_Accumulator;
```

```
separate (General_Purpose_Math)

package body Change_Calculator is

-- -- Declare previous value object and initialize it to value specified
-- -- during instantiation

   Previous_Value       : Element_Type := Initial_Value;
                                                                    pragma PAGE;
   procedure Reinitialize (Initial_Value : in Element_Type) is

   begin

      Previous_Value := Initial_Value;

   end Reinitialize;
                                                                    pragma PAGE;
   function Change (New_Value : Element_Type) return Element_Type is

      Change_Amount : Element_Type;

   begin

--    -- Calculate change and update Previous_Value

      Change_Amount  := New_Value - Previous_Value;
      Previous_Value := New_Value;

--    -- return Change

      return Change_Amount;

   end Change;
                                                                    pragma PAGE;
   function Retrieve_Value return Element_Type is

   begin

      return Previous_Value;

   end Retrieve_Value;


end Change_Calculator;
```

```
separate (General_Purpose_Math)

package body Integrator is

-- -- Declare the dependent and independent objects and initialize them to the
-- -- values specified at instantiation

   Dependent_Value            : Dependent_Type   := Initial_Dependent_Value;
   Previous_Independent_Value : Independent_Type := Initial_Independent_Value;
                                                                    pragma PAGE;
   procedure Reinitialize (Initial_Dependent_Value   : in Dependent_Type;
                           Initial_Independent_Value : in Independent_Type) is
   begin

      Dependent_Value   := Initial_Dependent_Value;
      Previous_Independent_Value := Initial_Independent_Value;

   end Reinitialize;
                                                                    pragma PAGE;
   procedure Update (Current_Independent_Value : in Independent_Type) is
   begin

      Previous_Independent_Value := Current_Independent_Value;

   end Update;
                                                                    pragma PAGE;
   function Integrate (Current_Independent_Value : Independent_Type;
                       Delta_Time  : Time_Interval := Default_Delta_Time)
                      return Dependent_Type is
   begin

--    -- Integrate new independent value across specified time interval

      Dependent_Value := Dependent_Value
             + ((Previous_Independent_Value + Current_Independent_Value) * 0.5)
             * Delta_Time;

--    -- Update previous independent value and return new dependent value

      Previous_Independent_Value := Current_Independent_Value;
      return Dependent_Value;

   end Integrate;

end Integrator;
```

```
separate (General_Purpose_Math)

function Interpolate_Or_Extrapolate
            (Input              : in Independent_Type;
             Lower_Independent  : in Independent_Type;
             Higher_Independent : in Independent_Type;
             Lower_Dependent    : in Dependent_Type;
             Higher_Dependent   : in Dependent_Type)
            return Dependent_Type is

begin

   return Lower_Dependent +
          ((Input - Lower_Independent) / (Higher_Independent - Lower_Independent)
          * (Higher_Dependent - Lower_Dependent));

end Interpolate_Or_Extrapolate;
```

```
with Polynomials;
separate (General_Purpose_Math)
package body Square_Root is

   package SYSTEM is new Polynomials.System_Functions.Square_Root
                            (Inputs  => Real,
                             Outputs => Real);

   function Sqrt (Input : Inputs) return Outputs is
      Answer : Outputs;
   begin
      Answer := Outputs(SYSTEM.Sqrt(Real(Input)));
      return Answer;
   exception
      when others => raise Negative_Input;
   end Sqrt;

end Square_Root;
```

```
separate (General_Purpose_Math)

function Root_Sum_Of_Squares (X : Real_Type;
                              Y : Real_Type;
                              Z : Real_Type)
                             return Real_Type is
begin

   return Sqrt (X*X + Y*Y + Z*Z);

end Root_Sum_Of_Squares;
```

```
separate (General_Purpose_Math)
function Sign (Input_Variable : Real_Type) return INTEGER is
   Result_Value : INTEGER;
begin
   if Input_Variable >= 0.0 then
      Result_Value := 1;
   else
      Result_Value := -1;
   end if;
   return Result_Value;
end Sign;
```

```
separate (General_Purpose_Math)
function Mean_Value (Value_Vector : Vector_Type) return Element_Type is
   Running_Sum : Element_Type := 0.0;
begin
-- -- Sum the values in the input vector
   for INDEX in Value_Vector'range loop
      Running_Sum := Running_Sum + Value_Vector (INDEX);
   end loop;
-- -- Calculate and return the average value
   return Running_Sum / Element_Type (Value_Vector'LENGTH);
end Mean_Value;
```

```
separate (General_Purpose_Math)

function Mean_Absolute_Difference (Value_Vector : Vector_Type)
                                   return Element_Type is

   Average            : Element_Type;
   One_Over_N         : Element_Type;
   Running_Sum        : Element_Type := 0.0;
   Total_Difference   : Element_Type := 0.0;

begin

-- -- Set up static value used twice

   One_Over_N := 1.0 / Element_Type (Value_Vector'LENGTH);


-- -- Calculate the average vector value

   for INDEX in Value_Vector'range loop

      Running_Sum := Running_Sum + Value_Vector (INDEX);

   end loop;

   Average := Running_Sum * One_Over_N;


-- -- Calculate and return the MAD

   for INDEX in Value_Vector'range loop

      Total_Difference := Total_Difference + abs (Value_Vector (INDEX) - Average);

   end loop;

   return Total_Difference * One_Over_N;

end Mean_Absolute_Difference;
```

```
separate (General_Purpose_Math)
package body Two_Way_Table_Lookup is

   function Inter_Or_Extra_X is new General_Purpose_Math.
                                     Interpolate_Or_Extrapolate
                                        (Independent_Type => Y_Values,
                                         Dependent_Type   => X_Values);

   function Inter_Or_Extra_Y is new General_Purpose_Math.
                                     Interpolate_Or_Extrapolate
                                        (Independent_Type => X_Values,
                                         Dependent_Type   => Y_Values);

   procedure Initialize( Table : out Tables;
                         INDEX : in  Indices;
                         X     : in  X_Values;
                         Y     : in  Y_Values ) is

   begin
      Table.Table_X( INDEX ) := X;
      Table.Table_Y( INDEX ) := Y;
   end Initialize;

   function Lookup_X ( Table : Tables;
                       Input : Y_Values ) return X_Values is

      Higher_Dept   : X_Values;
      Higher_Indept : Y_Values;
      INDEX         : Indices;
      Lower_Dept    : X_Values;
      Lower_Indept  : Y_Values;
      Result        : X_Values;

   begin

--       -- if the input value is out of bounds, then extrapolate for the
--       -- required value

      if Input < Table.Table_Y( Indices'FIRST) then
         Lower_Indept:= Table.Table_Y( Indices'FIRST );
         Higher_Indept := Table.Table_Y( Indices'SUCC(Indices'FIRST) );
         Lower_Dept:= Table.Table_X( Indices'FIRST );
         Higher_Dept := Table.Table_X( Indices'SUCC(Indices'FIRST) );
         Result := Inter_Or_Extra_X( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );
      elsif Input > Table.Table_Y( Indices'LAST) then
         Higher_Indept := Table.Table_Y( Indices'LAST );
         Lower_Indept := Table.Table_Y( Indices'PRED(Indices'LAST) );
         Higher_Dept := Table.Table_X( Indices'LAST );
         Lower_Dept := Table.Table_X( Indices'PRED(Indices'LAST) );
         Result := Inter_Or_Extra_X( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
```

```
                                        Higher_Dependent   => Higher_Dept );

      else        -- the input value is in the bounds of the table

--        -- Search through the table until the independent entry is greater than
--        -- or equal to the Key (Input) value.  Note that the value will be
--        -- found in the table since out-of-bounds values have already been
--        -- checked for.  Start with the successor of the first element so that
--        -- PRED will be defined if Input is equal to the first element.

         INDEX := Indices'SUCC( Indices'FIRST );
         Find_Value:
            loop
               exit when (Table.Table_Y( INDEX ) >= Input);
               INDEX := Indices'SUCC( INDEX );
            end loop Find_Value;

--        -- Save the table values.

         Lower_Indept := Table.Table_Y( Indices'PRED(INDEX) );
         Lower_Dept   := Table.Table_X( Indices'PRED(INDEX) );

         Higher_Indept := Table.Table_Y( INDEX );
         Higher_Dept   := Table.Table_X( INDEX );

--        -- Check to see if the value needs to be interpolated - that is,
--        -- if the value is not equal to one of the boundary values.

         if (Input = Higher_Indept) then
            Result := Higher_Dept;
         elsif (Input = Lower_Indept) then
            Result := Lower_Dept;
         else                -- Value is between, so interpolate
            Result := Inter_Or_Extra_X( Input              => Input,
                                        Lower_Independent  => Lower_Indept,
                                        Higher_Independent => Higher_Indept,
                                        Lower_Dependent    => Lower_Dept,
                                        Higher_Dependent   => Higher_Dept );
         end if;             -- Value is between, so interpolate
      end if;       -- the input value is in the bounds of the table
      return Result;
   end Lookup_X;

   function Lookup_Y ( Table : Tables;
                       Input : X_Values ) return Y_Values is

      Higher_Dept   : Y_Values;
      Higher_Indept : X_Values;
      INDEX         : Indices;
      Lower_Dept    : Y_Values;
      Lower_Indept  : X_Values;
      Result        : Y_Values;

   begin

--       -- if the input value is out of bounds, then extrapolate for the
--       -- required value
```

```
      if Input < Table.Table_X( Indices'FIRST) then
         Lower_Indept:= Table.Table_X( Indices'FIRST );
         Higher_Indept := Table.Table_X( Indices'SUCC(Indices'FIRST) );
         Lower_Dept:= Table.Table_Y( Indices'FIRST );
         Higher_Dept := Table.Table_Y( Indices'SUCC(Indices'FIRST) );
         Result := Inter_Or_Extra_Y( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );
      elsif Input > Table.Table_X( Indices'LAST) then
         Higher_Indept := Table.Table_X( Indices'LAST );
         Lower_Indept := Table.Table_X( Indices'PRED(Indices'LAST) );
         Higher_Dept := Table.Table_Y( Indices'LAST );
         Lower_Dept := Table.Table_Y( Indices'PRED(Indices'LAST) );
         Result := Inter_Or_Extra_Y( Input             => Input,
                                     Lower_Independent => Lower_Indept,
                                     Higher_Independent => Higher_Indept,
                                     Lower_Dependent   => Lower_Dept,
                                     Higher_Dependent  => Higher_Dept );

      else       -- the input value is in the bounds of the table

--       -- Search through the table until the independent entry is greater than
--       -- or equal to the Key (Input) value.  Note that the value will be
--       -- found in the table since out-of-bounds values have already been
--       -- checked for.  Start with the successor of the first element so that
--       -- PRED will be defined if Input is equal to the first element.

         INDEX := Indices'SUCC( Indices'FIRST );
         Find_Value:
            loop
               exit when (Table.Table_X( INDEX ) >= Input);
               INDEX := Indices'SUCC( INDEX );
            end loop Find_Value;

--       -- Save the table values.

         Lower_Indept := Table.Table_X( Indices'PRED(INDEX) );
         Lower_Dept   := Table.Table_Y( Indices'PRED(INDEX) );

         Higher_Indept := Table.Table_X( INDEX );
         Higher_Dept   := Table.Table_Y( INDEX );

--       -- Check to see if the value needs to be interpolated - that is,
--       -- if the value is not equal to one of the boundary values.

         if (Input = Higher_Indept) then
            Result := Higher_Dept;
         elsif (Input = Lower_Indept) then
            Result := Lower_Dept;
         else                -- Value is between, so interpolate
            Result := Inter_Or_Extra_Y( Input             => Input,
                                        Lower_Independent => Lower_Indept,
                                        Higher_Independent => Higher_Indept,
                                        Lower_Dependent   => Lower_Dept,
```

```
                                                Higher_Dependent    => Higher_Dept );
         end if;                -- Value is between, so interpolate
      end if;          -- the input value is in the bounds of the table
      return Result;
   end Lookup_Y;

end Two_Way_Table_Lookup;
```

(This page left intentionally blank.)

SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE AIR FORCE
WRIGHT LABORATORY (AFSC)
EGLIN AIR FORCE BASE, FLORIDA, 32542-5434

ERRATA
AD-B120257

REPLY TO ATTN OF: MNOI

13 Feb 92

SUBJECT: Removal of Distribution Statement and Export-Control Warning Notices

TO: Defense Technical Information Center
ATTN: DTIC/HAR (Mr William Bush)
Bldg 5, Cameron Station
Alexandria, VA 22304-6145

1. The following technical reports have been approved for public release by the local Public Affairs Office (copy attached).

	Technical Report Number	AD Number
1.	88-18-Vol-4	ADB 120 251
2.	88-18-Vol-5	ADB 120 252
3.	88-18-Vol-6	ADB 120 253
4.	88-25-Vol-1	ADB 120 309
5.	88-25-Vol-2	ADB 120 310
6.	88-62-Vol-1	ADB 129 568
7.	88-62-Vol-2	ADB 129 569
8.	88-62-Vol-3	ADB 129-570
9.	85-93-Vol-1	ADB 102-654
10.	85-93-Vol-2	ADB 102-655
11.	85-93-Vol-3	ADB 102-656
12.	88-18-Vol-1	ADB 120 248
13.	88-18-Vol-2	ADB 120 249
14.	88-18-Vol-7	ADB 120 254
15.	88-18-Vol-8	ADB 120 255
16.	88-18-Vol-9	ADB 120 256
17.	88-18-Vol-10	ADB 120 257
18.	88-18-Vol-11	ADB 120 258
19.	88-18-Vol-12	ADB 120 259

2. If you have any questions regarding this request call me at DSN 872-4620.

LYNN S. WARGO
Chief, Scientific and Technical
Information Branch

1 Atch
AFDTC/PA Ltr, dtd 30 Jan 92

ERRATA

DEPARTMENT OF THE AIR FORCE
HEADQUARTERS AIR FORCE DEVELOPMENT TEST CENTER (AFSC)
EGLIN AIR FORCE BASE, FLORIDA 32542-6000

REPLY TO
ATTN OF: PA (Jim Swinson, 882-3931) 30 January 1992

SUBJECT: Clearance for Public Release

TO: WL/MNA

The following technical reports have been reviewed and are approved for public release: AFATL-TR-88-18 (Volumes 1 & 2), AFATL-TR-88-18 (Volumes 4 thru 12), AFATL-TR-88-25 (Volumes 1 & 2), AFATL-TR-88-62 (Volumes 1 thru 3) and AFATL-TR-85-93 (Volumes 1 thru 3).

Virginia N. Pribyla

VIRGINIA N. PRIBYLA, Lt Col, USAF
Chief of Public Affairs

AFDTC/PA 92-039