UNCLASSIFIED

AD NUMBER

ADB120259

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies and their contractors; Critical Technology; MAR 1988. Other requests shall be referred to Air Force Armament Lab., Eglin AFB, FL. This document contains export-controlled technical data.

AUTHORITY

AFSC/MNOL Wright Lab ltr dtd 13 Feb 1992

THIS PAGE IS UNCLASSIFIED
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 Documents; and, Volumes 7-12 contain the Detail Design Documents.
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)
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

Distribution authorized to U.S. Government agencies and their contractors only; this report documents test and evaluation; distribution limitation applied July 1987. Other requests for this document must be referred to the Air Force Armament Laboratory (FXG) Eglin Air Force Base, Florida 32542-5434.

DESTRUCTION NOTICE - For classified documents, follow the procedures in DoD 5220.22-M, Industrial Security Manual, Section II - 19 or DoD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

WARNING: This document contains technical data whose export is restricted by the Arms Export Control Act (Title 22, U.S.C., Sec. 2751, et seq.) or the Export Administration Act of 1979, as amended (Title 50, U.S.C., App. 2401, et seq.). Violations of these export laws are subject to severe criminal penalties. Disseminate in accordance with the provisions of AFR 80-34.

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

88 4 6 131
3.3.7 ABSTRACT MECHANISMS
3.3.7.1 ABSTRACT_DATA_STRUCTURES TLCSC P691 (CATALOG #P330-0)

This package contains the bodies of the generic packages required to define and manipulate the following abstract data structures:

- bounded FIFO buffer
- unbounded FIFO buffer
- nonblocking circular buffer
- unbounded priority queue
- bounded stack
- unbounded stack

It also contains the package required by the unbounded parts to handle the manipulation of their available space lists.

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

3.3.7.1.1 REQUIREMENTS ALLOCATION

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Requirements Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded_FIFO_Buffer</td>
<td>R125</td>
</tr>
<tr>
<td>Unbounded_FIFO_Buffer</td>
<td>R164</td>
</tr>
<tr>
<td>Nonblocking_Circular_Buffer</td>
<td>R126</td>
</tr>
<tr>
<td>Unbounded_Priority_Queue</td>
<td>R165</td>
</tr>
<tr>
<td>Bounded_Stack</td>
<td>R166</td>
</tr>
<tr>
<td>Unbounded_Stack</td>
<td>R167</td>
</tr>
</tbody>
</table>

3.3.7.1.2 LOCAL ENTITIES DESIGN

Packages:

The following table describes the packages maintained local to this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Space_List_Operations</td>
<td>generic</td>
<td>Contains a set of functions to retrieve a node from and add a node to an available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.3 INPUT/OUTPUT

None.

3.3.7.1.4 LOCAL DATA

None.
3.3.7.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.6 PROCESSING

The following describes the processing performed by this part:

package body Abstract_Data_Structures is

-- ----------------------------------
-- --separate package bodies
-- ----------------------------------

  package body Bounded_FIFO_Buffer is separate;
  package body Unbounded_FIFO_Buffer is separate;
  package body Nonblocking_Circular_Buffer is separate;
  package body Unbounded_Priority_Queue is separate;
  package body BoundedStack is separate;
  package body Unbounded_Stack is separate;

  package body Available_Space_List_Operations is separate;

end Abstract_Data_Structures;

3.3.7.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.8 LIMITATIONS

None.

3.3.7.1.9 LLCSC DESIGN

3.3.7.1.9.1 AVAILABLE_SPACE_LIST_OPERATIONS PACKAGE DESIGN

This package contains a set of routines used to manipulate an available space list which is maintained local to the part instantiating this package.

The first routine, New Node, will return a node to the calling routine. If a node is available in the available space list, the node will be retrieved from there. If not, a new node will be dynamically allocated. If no memory is available for the allocation, a STORAGE_ERROR exception is raised.
The second routine, Save_Node, places a node in the available space list.
The third routine, Save_List, places a list of nodes in the available space list.
The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.1.1 REQUIREMENTS ALLOCATION

This part helps meet CAMP requirements R164, R165, R167.

3.3.7.1.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>limited private</td>
<td>A single element in the available space list</td>
</tr>
<tr>
<td>Pointers</td>
<td>access Nodes</td>
<td>A pointer to an element in the available space list</td>
</tr>
</tbody>
</table>

Data objects:
The following table summarizes the generic formal objects required by this part. All of these objects are in/out parameters and are changed by calls to the enclosed routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Length</td>
<td>INTEGER</td>
<td>N/A</td>
<td>Length of the available space list</td>
</tr>
<tr>
<td>Available_Head</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the first element in the available space list</td>
</tr>
<tr>
<td>Available_Tail</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the last element in the available space list</td>
</tr>
</tbody>
</table>

Subprograms:
The following table describes the generic formal subprograms required by this part:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot_Next</td>
<td>function</td>
<td>Given a pointer to a node, this function returns a pointer to the next node in the list</td>
</tr>
<tr>
<td>Set_Next</td>
<td>procedure</td>
<td>Given two points, A and B, sets A.Next equal to B</td>
</tr>
</tbody>
</table>

3.3.7.1.9.1.4 LOCAL DATA

None.

3.3.7.1.9.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.1.6 PROCESSING

The following describes the processing performed by this part:

```haskell
generic
  type Nodes is limited private;
  type Pointers is access Nodes;
  Available_Length : in out INTEGER;
  Available_Head   : in out Pointers;
  Available_Tail   : in out Pointers;
  with function Dot_Next (Ptr : in Pointers) return Pointers is <>;
  with procedure Set_Next (Ptr       : in Pointers;
                             Ptr_dot_Next : in Pointers) is <>;
package Available_Space_List_Operations is

  function New_Node return Pointers;

  procedure Save_Node (Saved_Node : in Pointers);

  procedure Save_List (Saved_Head : in Pointers;
                       Saved_Tail : in Pointers;
                       Node_Count : in POSITIVE);

end Available_Space_List_Operations;
```

3.3.7.1.9.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.9.1.8 LIMITATIONS

The following table describes the exceptions raised by this part:
<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD STORAGE_ERROR</td>
<td>Raised during elaboration of this package if an attempt is made to allocate memory when no more is available</td>
</tr>
</tbody>
</table>

3.3.7.1.9.1.9 LLCSC DESIGN

None.

3.3.7.1.9.1.10 UNIT DESIGN

None.

3.3.7.1.9.2 AVAILABLE_SPACE_LIST_OPERATIONS PACKAGE DESIGN

This package contains a set of routines used to manipulate an available space list which is maintained local to the part instantiating this package.

The first routine, New_Node, will return a node to the calling routine. If a node is available in the available space list, the node will be retrieved from there. If not, a new node will be dynamically allocated. If no memory is available for the allocation, a STORAGE_ERROR exception is raised.

The second routine, Save_Node, places a node in the available space list.

The third routine, Save_List, places a list of nodes in the available space list.

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

3.3.7.1.9.2.1 REQUIREMENTS ALLOCATION

This part helps meet CAMP requirements R164, R165, R167.

3.3.7.1.9.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this was specified in the package body of Abstract_Data_Structures.
Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>limited private</td>
<td>A single element in the available space list</td>
</tr>
<tr>
<td>Pointers</td>
<td>access Nodes</td>
<td>A pointer to an element in the available space list</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the generic formal objects required by this part. All of these objects are in/out parameters and are changed by calls to the enclosed routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Length</td>
<td>INTEGER</td>
<td>N/A</td>
<td>Length of the available space list</td>
</tr>
<tr>
<td>Available_Head</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the first element in the available space list</td>
</tr>
<tr>
<td>Available_Tail</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the last element in the available space list</td>
</tr>
</tbody>
</table>

Subprograms:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dct_Next</td>
<td>function</td>
<td>Given a pointer to a node, this function returns a pointer to the next node in the list</td>
</tr>
<tr>
<td>Set_Next</td>
<td>procedure</td>
<td>Given two points, A and B, sets A.Next equal to B</td>
</tr>
</tbody>
</table>

3.3.7.1.9.2.4 LOCAL DATA

None.

3.3.7.1.9.2.5 PROCESS CONTROL

Not applicable.
3.3.7.1.9.2.6  PROCESSING

The following describes the processing performed by this part:

separate (Abstract_Data_Structures)
package body Available_Space_List_Operations is
end Available_Space_List_Operations;

3.3.7.1.9.2.7  UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.9.2.8  LIMITATIONS

None.

3.3.7.1.9.2.9  LLCSC DESIGN

None.

3.3.7.1.9.2.10  UNIT DESIGN

3.3.7.1.9.2.10.1  NEW_NODE UNIT DESIGN

This function returns a node to the calling routine. If nodes are available in the space list, the node returned will be from there. If the available space list is empty, this routine will attempt to dynamically allocate memory. If no more memory is available on the system, a STORAGE_ERROR exception will be raised.

3.3.7.1.9.2.10.1.1  REQUIREMENTS ALLOCATION

This part helps meets CAMP requirements R164, R164, R176.

3.3.7.1.9.2.10.1.2  LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.10.1.3  INPUT/OUTPUT

None.

3.3.7.1.9.2.10.1.4  LOCAL DATA

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the node being returned</td>
</tr>
<tr>
<td>New_Available_Head</td>
<td>Pointers</td>
<td>N/A</td>
<td>Temporary variable used to mark where Available_Head will point</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>when this routine is exited</td>
</tr>
</tbody>
</table>

3.3.7.1.9.2.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.10.1.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
function New_Node return Pointers is

-- declaration section

Ptr : Pointers;
New_Available_Head : Pointers;

-- begin function New_Node

begin

if Available_Length > 0 then

  -- get the node from the available space list and mark the node
  -- that will now be the head of the available space list
  Ptr := Available_Head;
  New_Available_Head := Dot_Next(Available_Head);

  -- initialize node being returned
  Set_Next (Ptr => Ptr,
             Ptr_dot_Next => NULL);

  -- adjust the available space list
  Available_Head := New_Available_Head;
  Available_Length := Available_Length - 1;

else

  -- allocate space to get the node
  Ptr := NEW Nodes;

end if;
```
3.3.7.1.9.2.10.1.7 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 describes the subroutines required by this part and defined as generic formal subprograms to the Available_Space_List_Operations package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oot_Next</td>
<td>function</td>
<td>Given a pointer to a node, this function returns a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pointer to the next node in the list</td>
</tr>
<tr>
<td>Set_Next</td>
<td>procedure</td>
<td>Given two points, A and B, sets A.Next equal to B</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_List_Operations package.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>limited private</td>
<td>A single element in the available space list</td>
</tr>
<tr>
<td>Pointers</td>
<td>access Nodes</td>
<td>A pointer to an element in the available space list</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_List_Operations package.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Length</td>
<td>INTEGER</td>
<td>N/A</td>
<td>Length of the available space list</td>
</tr>
<tr>
<td>Available_Head</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the first element in the available space list</td>
</tr>
</tbody>
</table>
3.3.7.1.9.2.10.1.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD.STORAGE_ERROR</td>
<td>Raised if an attempt is made to allocate memory when no more is available</td>
</tr>
</tbody>
</table>

3.3.7.1.9.2.10.2 SAVE_NODE UNIT DESIGN

This procedure returns a node to the available space list. The node returned to the list is the one pointed to by Saved_Node.

3.3.7.1.9.2.10.2.1 REQUIREMENTS ALLOCATION

This part helps meet CAMP requirements R164, R164, R176.

3.3.7.1.9.2.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved_Node</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node which is to be placed in the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.2.10.2.4 LOCAL DATA

None.

3.3.7.1.9.2.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.10.2.6 PROCESSING

The following describes the processing performed by this part:

```haskell
procedure Save_Node(Saved_Node : In Pointers) is
```
begin
  Set_Next (Ptr        => Available_Tail,
            Ptr_dot_Next => Saved_Node);  
  Available_Tail := Saved_Node;
  Set_Next (Ptr        => Available_Tail,
            Ptr_dot_Next => NULL);
  Available_Length := Available_Length + 1;
end Save_Node;

3.3.7.1.9.2.10.2.7  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 describes the subroutines required by this part and defined as generic formal subprograms to the Available_Space_List_Operations package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetNext</td>
<td>procedure</td>
<td>Given two points, A and B, sets A.Next equal to B</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_List_Operations package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>limited</td>
<td>A single element in the available space list</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>A pointer to an element in the available space list</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_List_Operations package.
### NAME | TYPE | VALUE | DESCRIPTION
--- | --- | --- | ---
Available Length | INTEGER | N/A | Length of the available space list
Available Tail | Pointers | N/A | Points to the last element in the available space list

#### 3.3.7.1.9.2.10.2.8 LIMITATIONS
None.

#### 3.3.7.1.9.2.10.3 SAVE_LIST UNIT DESIGN
This procedure places a linked list of nodes in the available space list.

#### 3.3.7.1.9.2.10.3.1 REQUIREMENTS ALLOCATION
None.

#### 3.3.7.1.9.2.10.3.2 LOCAL ENTITIES DESIGN
None.

#### 3.3.7.1.9.2.10.3.3 INPUT/OUTPUT

**FORMAL PARAMETERS:**

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved_Head</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the first node to be placed in the available space list</td>
</tr>
<tr>
<td>Saved_Tail</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the last node to be placed in the available space list</td>
</tr>
<tr>
<td>Node_Count</td>
<td>POSITIVE</td>
<td>in</td>
<td>Number of nodes to be placed in the available space list</td>
</tr>
</tbody>
</table>

#### 3.3.7.1.9.2.10.3.4 LOCAL DATA
None.
3.3.7.1.9.2.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.10.3.6 PROCESSING

The following describes the processing performed by this part:

```pascal
procedure Save_List (Saved_Head : in Pointers;
                    Saved_Tail : in Pointers;
                    Node_Count : in POSITIVE) is
begin
  Set_Next (Ptr              => Available_Tail,
             Ptr_dot_Next    => Saved_Head);
  Available_Tail := Saved_Tail;
  Set_Next (Ptr              => Available_Tail,
             Ptr_dot_Next    => Saved_Head);
  Available_Length := Available_Length + Node_Count;
end Save_List;
```

3.3.7.1.9.2.10.3.7 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 describes the subroutines required by this part and defined as generic formal subprograms to the Available_Space_List_Operations package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set_Next</td>
<td>procedure</td>
<td>Given two points, A and B, sets A.Next equal to B</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Available_Space_List_Operations LLCSC:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers</td>
<td>access Nodes</td>
<td>A pointer to an element in the available space list</td>
</tr>
</tbody>
</table>
Data objects:

The following table summarizes the objects required by this part and defined as generic formal parameters to the Available_Space_List_Operations LLCSC:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Len</td>
<td>INTEGER</td>
<td>N/A</td>
<td>Length of the available space list</td>
</tr>
<tr>
<td>Available_Tail</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the last element in the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.2.10.3.8 LIMITATIONS

None.

3.3.7.1.9.3 BOUNDED_FIFO_BUFFER PACKAGE DESIGN (CATALOG #P331-0)

This generic package defines the data type and contains the operations required to perform first-in-first-out buffering operations on incoming data. The head always points to a dummy node. The first node following the dummy node contains the next piece of data to be retrieved. The tail always points to where the next element should be added. If the tail points to the element immediately in front of the head, the buffer is empty. If the tail points to the same element as the head, the buffer is full. Since the buffer is implemented as an array, the head and tail will advance through the array in a circular fashion, but no overwriting of data currently in the buffer will be permitted.

Empty FIFO buffer:   +-+ <--------Head ++ ++ ++ +++++<--------Tail ++ ++ ++ +++ +++

Full FIFO buffer:    Tail----->+++ <--------Head ++ ++ ++ ++ ++ ++ ++ +++ +++

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

3.3.7.1.9.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP required R125.

3.3.7.1.9.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:
The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial_Buffer_Size</td>
<td>POSITIVE</td>
<td>N/A</td>
<td>Maximum number of elements which can be in the buffer at any given time</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.4 LOCAL DATA

None.

3.3.7.1.9.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.6 PROCESSING

The following describes the processing performed by this part:

separate (Abstract_Data_Structures)
package body Bounded_FIFO_Buffer is
end Bounded_FIFO_Buffer;

3.3.7.1.9.3.7 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:

Data types:
The following table describes the data types which were previously defined in this part's specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Buffer_Statuses</td>
<td>discrete</td>
<td>Buffer_Size</td>
<td>Used to indicate the status of the buffer</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>Available, Full</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of this part's specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant</td>
</tr>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>Array of elements</td>
</tr>
</tbody>
</table>

Data objects:

The following table describes the data objects which were previously defined in this part's specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Size</td>
<td>POSITIVE</td>
<td>Initial Buffer_Size</td>
<td>Number of usable elements in a buffer</td>
</tr>
</tbody>
</table>

Exceptions:

The following table describes the exceptions which were previously defined in this part's specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
<tr>
<td>Buffer_Full</td>
<td>Error condition raised if an attempt is made to add elements to a full buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.8 LIMITATIONS

None.
3.3.7.1.9.3.9 LLCSC DESIGN
None.

3.3.7.1.9.3.10 UNIT DESIGN
3.3.7.1.9.3.10.1 CLEAR_BUFFER UNIT DESIGN
This procedure clears an input buffer by setting its length to 0 and resetting its head and tail to 0 and 1, respectively.

3.3.7.1.9.3.10.1.1 REQUIREMENTS ALLOCATION
This part meets CAMP requirement R125.

3.3.7.1.9.3.10.1.2 LOCAL ENTITIES DESIGN
None.

3.3.7.1.9.3.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:
The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.1.4 LOCAL DATA
None.

3.3.7.1.9.3.10.1.5 PROCESS CONTROL
Not applicable.

3.3.7.1.9.3.10.1.6 PROCESSING
The following describes the processing performed by this part:

    procedure Clear_Buffer (Buffer : out Buffers) is
    -- declaration section
    --
    Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
Head        : Buffer_Range renames Buffer.Head;
Tail        : Buffer_Range renames Buffer.Tail;

begin
  Buffer_Length := 0;
  Head := 0;
  Tail := 1;
end Clear_Buffer;

3.3.7.1.9.3.10.1.7 UTILIZATION OF OTHER ELEMENTS
None.

3.3.7.1.9.3.10.1.8 LIMITATIONS
None.

3.3.7.1.9.3.10.2 ADD_ELEMENT UNIT DESIGN
This procedure adds an element to an input buffer if the buffer is not already full. After the element is added, the tail is advanced one place in the buffer and the length counter is incremented by 1.

The exception Buffer_Full is raised if an attempt is made to add an element to an already full buffer.

3.3.7.1.9.3.10.2.1 REQUIREMENTS ALLOCATION
This part meets CAMP requirement R125.

3.3.7.1.9.3.10.2.2 LOCAL ENTITIES DESIGN
None.

3.3.7.1.9.3.10.2.3 INPUT/OUTPUT
FORMAL PARAMETERS:
The following table describes this part's formal parameters:
3.3.7.1.9.3.10.2.4 LOCAL DATA
None.

3.3.7.1.9.3.10.2.5 PROCESS CONTROL
Not applicable.

3.3.7.1.9.3.10.2.6 PROCESSING
The following describes the processing performed by this part:

    procedure Add_Element (New Element : in   Elements;  
                        Buffer    : in out Buffers) is

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

    List       : Lists      renames Buffer.List;  
    Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;  
    Head        : Buffer_Range renames Buffer.Head;  
    Tail        : Buffer_Range renames Buffer.Tail;  

--                     --begin procedure Add_Element
--                     ----------------------

begin

--                     --make sure buffer isn’t full
   if Head = Tail then
      raise Buffer_Full;
   end if;

List(Tail) := New Element;
Buffer Length := Buffer Length + 1;
if Tail = Buffer_Size then
    Tail := 0;
else
    Tail := Tail +.1;
end if;

end Add_Element;
3.3.7.1.9.3.10.2.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Abstract_Data_Structures. Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Buffer_Size</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>Array of elements</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Size</td>
<td>POSITIVE</td>
<td>Initial Buffer_Size</td>
<td>Number of usable elements in a buffer</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:
### 3.3.7.1.9.3.10.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Full</td>
<td>Error condition raised if an attempt is made to add elements to a full buffer</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.3.10.3 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the top element in the buffer if the buffer is not empty. The head is advanced through the buffer by 1 before the element is retrieved and the size of the buffer is decremented by 1 after the element is retrieved.

If the buffer is empty before calling this routine, the exception Buffer_Empty is raised.

### 3.3.7.1.9.3.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

### 3.3.7.1.9.3.10.3.2 LOCAL ENTITIES DESIGN

None.

### 3.3.7.1.9.3.10.3.3 INPUT/OUTPUT

**FORMAL PARAMETERS:**

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
<tr>
<td>Old_Element</td>
<td>Elements</td>
<td>out</td>
<td>Element retrieved from the buffer</td>
</tr>
</tbody>
</table>
3.3.7.1.9.3.10.3.4 LOCAL DATA

None.

3.3.7.1.9.3.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.3.6 PROCESSING

The following describes the processing performed by this part:

```haskell
procedure Retrieve_Element (Buffer     : in out Buffers;
                           Old_Element :   out Elements) is

   -- declaration section
   --
   -- Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
   -- Head           : Buffer_Range renames Buffer.Head;
   -- List           : Lists      renames Buffer.List;
   -- Tail           : Buffer_Range renames Buffer.Tail;

   -- begin procedure Retrieve_Element
   --
   begin

      -- make sure don't have an empty buffer
      if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
         raise Buffer_Empty;
      end if;

      if Head = Buffer_Size then
         Head := 0;
      else
         Head := Head + 1;
      end if;
      Old_Element := List(Head);
      Buffer_Length := Buffer_Length - 1;

      end Retrieve_Element;

   end Retrieve_Element;

3.3.7.1.9.3.10.3.7 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:
Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Abstract_Data_Structures. Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Buffer_Size</td>
<td>subtype</td>
<td>List of data along with relevant information</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>Array of elements</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_JEmpty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.3.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>
3.3.7.1.9.3.10.4 PEEK UNIT DESIGN

This function returns the first element of the buffer if the buffer is not empty. The status of the buffer is not changed, however, and the element itself remains in the buffer.

The Buffer_Empty exception is raised if an attempt is made to look at an empty buffer.

3.3.7.1.9.3.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:
The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.4.4 LOCAL DATA

Data objects:
The following table describes the data objects maintained local to this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spot</td>
<td>Buffer_Range</td>
<td>Marks location of element to be looked at</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.4.6 PROCESSING

The following describes the processing performed by this part:

    function Peek (Buffer : in Buffers) return Elements is
-- declaration section

Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
Head           : Buffer_Range renames Buffer.Head;
Tail           : Buffer_Range renames Buffer.Tail;
List           : Lists renames Buffer.List;
Spot           : Buffer_Range;

-- --begin function Peek
--

begin

begin

-- make sure don’t have an empty buffer
if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
    raise Buffer_Empty;
end if;

if Head = Buffer_Size then
    Spot := 0;
else
    Spot := Head + 1;
end if;

return List(Spot);

end Peek;

3.3.7.1.9.3.10.4.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Abstract_Data_Structures. Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.
The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 .. Buffer_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
</tbody>
</table>

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures.Bounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.5 BUFFER_STATUS UNIT DESIGN

This function returns the status of the buffer. If there are no elements in the buffer, the status is empty; if there is no room for additional elements, the status is full; otherwise, the status is available.

3.3.7.1.9.3.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.
3.3.7.1.9.3.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.5.4 LOCAL DATA

Data objects:

The following objects are maintained local to this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Buffer_Statuses</td>
<td>Status of the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.5.6 PROCESSING

The following describes the processing performed by this part:

```vhdl
function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

  -- declaration section
  --

  Head : Buffer_Range renames Buffer.Head;
  Tail : Buffer_Range renames Buffer.Tail;

  Status : Buffer_Statuses;

  -- begin function Buffer_Status

  begin
```
if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
    Status := Empty;
elsif Head = Tail then
    Status := Full;
else
    Status := Available;
end if;

return Status;
end Buffer_Status;

3.3.7.1.9.3.10.5.7 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:

Data types:

The following table summarizes the types required by this part and defined in the package specification of AbstractDataStructures. Bounded_FIFO_Buffer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 .. Buffer_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Buffer_Statuses</td>
<td>subtype</td>
<td>Buffer_Size</td>
<td>Used to indicate the status of the buffer</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Size</td>
<td>POSITIVE</td>
<td>Initial Buffer_Size</td>
<td>Number of usable elements in a buffer</td>
</tr>
</tbody>
</table>
3.3.7.1.9.3.10.5.8 LIMITATIONS

None.

3.3.7.1.9.3.10.6 BUFFER_LENGTH UNIT DESIGN

This function returns the length of the current buffer.

3.3.7.1.9.3.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.6.4 LOCAL DATA

None.

3.3.7.1.9.3.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.6.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Buffer_Length (Buffer : in Buffers) return Buffer_Range is
begin
    return Buffer.Buffer_Length;
end Buffer_Length;
```
3.3.7.1.9.3.10.6.7 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:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>subtype</td>
<td>Buffer_Size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant info</td>
</tr>
</tbody>
</table>

3.3.7.1.9.3.10.6.8 LIMITATIONS

None.

3.3.7.1.9.4 UNBOUNDED_FIFO_BUFFER PACKAGE DESIGN (CATALOG #P332-0)

This generic package defines the data type and contains the operations required to perform first-in-first-out buffering operations on incoming data. The head of the buffer always points to a dummy node. The first node following the dummy node contains the next piece of data to be retrieved. The tail always points to the node containing the last element added to the buffer. If the tail points to the same node as the head, the buffer is empty.

A buffer must be initialized before it is used. If an attempt is made to use an uninitialized buffer, the exception Buffer_Not_Initaialized will be raised. The Initialized_Buffer procedure returns an initialized buffer. The Clear_Buffer procedure returns the nodes of a buffer to the available space list and then returns an initialized buffer.

An available space list is maintained local to this part. When this part is elaborated the available space list will have a dummy node plus Initial_Available_Space_Size nodes. When nodes are added to the buffer, the Add_Element routine will try to get a node from the available space list before attempting to allocate more memory. When the Retrieve Element routine is called, the unused node will be returned to the available space list for later use. The memory committed to the available space may be deallocated by calling
the Free_Memory procedure.

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

3.3.7.1.9.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.2 LOCAL ENTITIES DESIGN

Data structures:

An available space list is maintained local to this part’s package body.

Subprograms:

The following subprograms are contained local to this body:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free_Node</td>
<td>procedure</td>
<td>Instantiation of UNCHECKED DEALLOCATION</td>
</tr>
<tr>
<td>Dot_Next</td>
<td>function</td>
<td>Given a pointer P, this function returns the value of P.Next</td>
</tr>
<tr>
<td>Set_Next</td>
<td>procedure</td>
<td>Given two points P &amp; Q, this procedure sets P.Next = Q</td>
</tr>
</tbody>
</table>

The following subprograms are contained in this part as a result of renaming operations on identically named routines contained in the locally instantiated Available_Space_Operations package.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
<tr>
<td>Save_Node</td>
<td>procedure</td>
<td>Handles placing a node in the available space list</td>
</tr>
<tr>
<td>Save_List</td>
<td>procedure</td>
<td>Handles placing a list of nodes in the available space list</td>
</tr>
</tbody>
</table>

This package body contains code to initialize the Available_Space_List. This code is executed when the package is elaborated. At a minimum, this code calls the Initialize Buffer procedure to initialize the Available_Space_List so it contains a dummy node. If the generic formal object Initial_Available_Space_Size is greater than or equal to 1, this routine then places the requested number of nodes (in addition to the dummy node) in the available space list.
3.3.7.1.9.4.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial_Available_</td>
<td>NATURAL</td>
<td>Number of nodes to be initially placed in</td>
</tr>
<tr>
<td>Space_Size</td>
<td></td>
<td>the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.4 LOCAL DATA

None.

3.3.7.1.9.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.6 PROCESSING

The following describes the processing performed by this part:

```with UNCHECKED_DEALLOCATION;
separate (Abstract_Data_Structures)
package body Unbounded_FIFO_Buffer is

-- declaration section

-- this variable is accessed ONLY when setting up the available space list
Initial_Head : Pointers := new Nodes;
Available_Space : Buffers := (Current_Length => 0,
                           Head       => Initial_Head,
                           ... )
```
procedure Free is new UNCHECKED_DEALLOCATION
  (Object  => Nodes,
   Name    => Pointers);

procedure Free_Node (Which_Node : in out Pointers)
  renames Free;

function Dot_Next (Ptr : in Pointers) return Pointers;

procedure Set_Next (Ptr        : in Pointers;
                    Ptr_dot_Next : in Pointers);

package Available_Space_Operations is new
  Available_Space_Llst_Operations
  (Nodes => Nodes,
   Pointers => Pointers,
   Available_Length => Available_Length,
   Available_Head => Available_Head,
   Available_Tail => Available_Tail);

function New_Node return Pointers
  renames Available_Space_Operations.New_Node;

procedure Save_Node (Saved_Node : in Pointers)
  renames Available_Space_Operations.Save_Node;

procedure Save_List (Saved_Head : in Pointers;
                     Saved_Tail : in Pointers;
                     Node_Count : in POSITIVE)
  renames Available_Space_Operations.Save_List;

------------------------------------------------------------
--begin package Unbounded_FIFO_Buffer
--(see header for package body for details)
------------------------------------------------------------
begin
  --set up available space list if one is desired
  if Initial_Available_Space_Size > 0 then
    Add Nodes To Available_Space_List:
    for I in 1..Initial_Available_Space_Size loop
      Available_Tail.Next := NEW Nodes;
      Available_Tail := Available_Tail.Next;
    end loop Add_Nodes_to_Available_Space_List;

    Available_Length := Initial_Available_Space_Size;
  end if;
end Unbounded_FIFO_Buffer;

3.3.7.1.9.4.7 UTILIZATION OF OTHER ELEMENTS

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

1. Unchecked_Deallocation

Subprograms and task entries:

The following table describes the subroutines required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNCHECKED</td>
<td>generic</td>
<td>N/A</td>
<td>Used to deallocate memory</td>
</tr>
<tr>
<td>DEALLOCATION</td>
<td>function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exceptions:

The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE_ERROR</td>
<td>Raised when an attempt is made to dynamically allocate more memory than is available</td>
</tr>
</tbody>
</table>

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:

Packages:

The following table describes the packages required by this part and specified in the package body of the Abstract_Data_Structures package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Space_List_Operations</td>
<td>generic</td>
<td>Contains the routines required to retrieve a node from and place a node in the available space list</td>
</tr>
</tbody>
</table>

Data types:

The following data types were previously defined in this part's package specification:
The following data types were previously defined in the private portion of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the buffer; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the buffer</td>
</tr>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

Exceptions:

The following exceptions were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
<tr>
<td>Buffer_Not_Initialized</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage_Error</td>
<td>Raised during elaboration of this package if an attempt is made to allocate memory when no more is available</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.9 LLCSC DESIGN

None.
3.3.7.1.9.4.10  UNIT DESIGN

3.3.7.1.9.4.10.1  INITIALIZE_BUFFER UNIT DESIGN

This procedure initializes a buffer. It does this in the following manner:

1) If the buffer has never been initialized then:
   o places a dummy node in the buffer and
   o initializes the length to 0

2) else if the buffer has elements in it then:
   o calls the Clear_Buffer procedure

3) else if the buffer has a length of 0 then o does nothing

3.3.7.1.9.4.10.1.1  REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.1.2  LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.1.3  INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being initialized</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.1.4  LOCAL DATA

None.

3.3.7.1.9.4.10.1.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.1.6  PROCESSING

The following describes the processing performed by this part:

    procedure Initialize_Buffer (Buffer : in out Buffers) is
--    declaration section
--
Current_Length : INTEGER renames Buffer.Current_Length;
Head : Pointers renames Buffer.Head;
Tail : Pointers renames Buffer.Tail;

begin procedure Initialize_Buffer
end

begin
if Current_Length = -1 then

-- handle an uninitialized buffer
  Head := New Node;
  Tail := Head;
  Current_Length := 0;

elsif Current_Length > 0 then

-- handle a buffer that has something in it
  Clear_Buffer(Buffer => Buffer);
else

-- current length = 0 so it is already initialized
  NULL;

end if;

end Initialize_Buffer;

3.3.7.1.9.4.10.1.7 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 in the package specification of Unbounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear_Buffer</td>
<td>procedure</td>
<td>Returns all the nodes in a buffer to the available space list</td>
</tr>
</tbody>
</table>

Data types:
The following table summarizes the types required by this part and defined in the private portion of the part's package specification:
### 3.3.7.1.9.4.10.18 LIMITATIONS

None.

### 3.3.7.1.9.4.10.2 CLEAR_BUFFER UNIT DESIGN

This procedure returns all the elements in a buffer, except for the dummy node, to the available space list. If this routine is sent an uninitialized buffer, a Buffer_Not_Initialed exception is raised.

#### 3.3.7.1.9.4.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

#### 3.3.7.1.9.4.10.2.2 LOCAL ENTITIES DESIGN

None.

#### 3.3.7.1.9.4.10.2.3 INPUT/OUTPUT

**FORMAL PARAMETERS:**

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being cleared</td>
</tr>
</tbody>
</table>

#### 3.3.7.1.9.4.10.2.4 LOCAL DATA

**Data objects:**

The following table describes the objects maintained local to this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This_Node</td>
<td>Pointers</td>
<td>Node to be placed in the available space list</td>
</tr>
</tbody>
</table>
3.3.7.1.9.4.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Clear_Buffer (Buffer : in out Buffers) is

   -- declaration section

   Current_Length : INTEGER  renames Buffer.Current_Length;
   Head            : Pointers renames Buffer.Head;
   Tail            : Pointers renames Buffer.Tail;
   This_Node       : Pointers;

   begin

      -- make sure this is an initialized buffer
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      end if;

      -- placed nodes in the available space list
      Save_List (Saved_Head => Head.Next,
                 Saved_Tail  => Tail,
                 Node_Count  => Current_Length);

      -- reinitialize buffer variables
      Current_Length := 0;
      Head.Next      := NULL;
      Tail           := Head;

      end Clear_Buffer;

```

3.3.7.1.9.4.10.2.7 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 in the package body of Unbounded_FIFO_Buffer:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save_List</td>
<td>procedure</td>
<td></td>
<td>Handles placing a list of nodes in the available space list</td>
</tr>
</tbody>
</table>

Data types:
The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the buffer; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the buffer</td>
</tr>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

Exceptions:
The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Not_Init</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Not_Initiated</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.3 FREE_MEMORY UNIT DESIGN

This procedure deallocates the memory occupied by the available space list.
3.3.7.1.9.4.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.3.3 INPUT/OUTPUT

None.

3.3.7.1.9.4.10.3.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node_to_be_Freed</td>
<td>Pointers</td>
<td>N/A</td>
<td>Pointer to the node to be deallocated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.3.6 PROCESSING

The following describes the processing performed by this part:

    procedure Free_Memory is
    -- declaration section
    --

    Node_to_be_Freed : Pointers;
    --
    -- begin procedure Free_Memory
    --
    begin
    Clear_Out_Available_Space_List:
    while Available_Head /= Available_Tail loop
      Node_To_Be_Free := Available_Head;
      Available_Header := Available_Head.Next;
Free_Node (Which_Node => Node_to_be_Freed);
end loop Clear_Out_Available_Space_List;
Available_Length := 0;
end Free_Memory;

3.3.7.1.9.4.10.3.7 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 in the package body of Unbounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeNode</td>
<td>procedure</td>
<td>Instantiation of UNCHECKED_DEALLOCATION</td>
</tr>
</tbody>
</table>

Data types:
The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the private portion of the part’s package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the buffer; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the buffer</td>
</tr>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Space</td>
<td>Buffers</td>
<td>List of available nodes; nodes will be added to list when Retrieve_Element is called and retrieved from the list when Add_Element is called; the nodes in the list are deallocated when Clear_Memory is called</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Length</td>
<td>INTEGER</td>
<td>Available_Space. Current_Length</td>
<td>Indicates the current length of the available space list</td>
</tr>
<tr>
<td>Available_Head</td>
<td>Pointers</td>
<td>Available_Space. Head</td>
<td>Points to the head node in the available space list</td>
</tr>
<tr>
<td>Available_Tail</td>
<td>Pointers</td>
<td>Available_Space. Tail</td>
<td>Points to the tail node in the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.3.8 LIMITATIONS

None.

3.3.7.1.9.4.10.4 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the end of the FIFO buffer.

If the buffer has not been initialized, the exception Buffer_Not_Initialized is raised.

The Storage Error exception is raised if a call to this routine requires memory to be dynamically allocated when no more memory is available.

3.3.7.1.9.4.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.4.2 LOCAL ENTITIES DESIGN

None.
3.3.7.1.9.4.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
<tr>
<td>New_Element</td>
<td>Elements</td>
<td>in</td>
<td>Element to be added to the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.4.4 LOCAL DATA

None.

3.3.7.1.9.4.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.4.6 PROCESSING

The following describes the processing performed by this part:

procedure Add_Element (New Element : in Elements; Buffer : in out Buffers) is

-- declaration section

Current_Length : INTEGER renames Buffer.Current_Length;
Tail : Pointers renames Buffer.Tail;
New_Tail : Pointers;

-- begin procedure Add_Element

begin

-- make sure buffer has been initialized
if Current_Length = -1 then
  raise Buffer_Not_Initialized;
end if;

-- now get a node
New_Tail := New_Node;

-- now adjust the buffer
Tail.Next := New_Tail;
Tail := New_Tail;
Tail.Data := New_Element;
Current_Length := Current_Length + 1;
end Add_Element;

3.3.7.1.9.4.10.4.7 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 in the package body of Unbounded_FIFO_Buffer:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
</tbody>
</table>
```

Data types:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>
```

The following table summarizes the types required by this part and defined in the private portion of the part’s package specification:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the buffer; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the buffer</td>
</tr>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>
```

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_Buffer:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE_ERROR</td>
<td>Raised when an attempt is made to dynamically allocate more memory than is available</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.8 LIMITATIONS
The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage_Error</td>
<td>Raised if an attempt is made to allocate memory when no more is available</td>
</tr>
<tr>
<td>Buffer_NotInitialized</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.5 RETRIEVE_ELEMENT UNIT DESIGN
This procedure retrieves the oldest element from the FIFO buffer, places the spare node on the available space list, and updates the status of the FIFO buffer.

If the buffer has not been initialized, a Buffer_Not Initialized exception is raised.

If the buffer is empty, a Buffer Empty exception is raised.

3.3.7.1.9.4.10.5.1 REQUIREMENTS ALLOCATION
This part meets CAMP requirement R164.

3.3.7.1.9.4.10.5.2 LOCAL ENTITIES DESIGN
None.
3.3.7.1.9.4.10.5.3  INPUT/OUTPUT

FORMAL PARAMETERS:
The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
<tr>
<td>Old_Element</td>
<td>Elements</td>
<td>out</td>
<td>Element retrieved from the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.5.4  LOCAL DATA

Data objects:
The following table describes the objects maintained local to this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This_Node</td>
<td>Pointers</td>
<td>Node to be placed in the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.5.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.5.6  PROCESSING

The following describes the processing performed by this part:

```pl
procedure Retrieve_Element (Buffer     : in out Buffers;
   Old_Element :   out Elements) is

   -- ---declaration section
   -- ---begin procedure Retrieve_Element
   -- ---begin procedure Retrieve_Element

   begin
   -- --make sure an element is available
   if Current_Length = -1 then
      raise Buffer_NotInitialized;
```

elsif Current_Length = 0 then
    raise Buffer_Empty;
end if;

-- --save dummy node in the available space list
This_Node := Head;
Head := Head.Next;
Save_Node (Saved_Node => This_Node);

-- --retrieve element (its node becomes the new dummy node)
Old_Element := Head.Data;

-- --update buffer status
Current_Length := Current_Length - 1;

end Retrieve_Element;

3.3.7.1.9.4.10.5.7 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 in the package body of Unbounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save_Node</td>
<td>procedure</td>
<td>Handles placing a node in the available space list</td>
</tr>
</tbody>
</table>

Data type:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:
### Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of `Abstract_Data_Structures`. 

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer.</td>
</tr>
<tr>
<td>Buffer_Not_Initilaized</td>
<td>Raised if an attempt is made to use an uninitialized buffer.</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.4.10.5.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Raised if an attempt is made to access an empty buffer.</td>
</tr>
<tr>
<td>Buffer_Not_Initilaized</td>
<td>Raised if an attempt is made to use an uninitialized buffer.</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.4.10.6 PEEK UNIT DESIGN

This function returns the oldest element in the FIFO buffer.

If the buffer has not been initialized, a `Buffer_Not_Initilaized` exception is raised.

If the buffer is empty, a `Buffer_Empty` exception is raised.

### 3.3.7.1.9.4.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.
3.3.7.1.9.4.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.6.4 LOCAL DATA

None.

3.3.7.1.9.4.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.6.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Peek (Buffer : in Buffers) return Elements is

   --- declaration section
   ---
   Current_Length : INTEGER renames Buffer.Current_Length;
   Head : Pointers renames Buffer.Head;

   ---
   begin function Peek
   ---

   begin
      --- make sure something is there to look at
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      elsif Current_Length = 0 then
         raise Buffer_Empty;
      end if;
      return Head.Next.Data;
   end Peek;
```

end Peek ;
3.3.7.1.9.4.10.6.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the private portion of the part’s package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the buffer; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the buffer</td>
</tr>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_Buffer:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
<tr>
<td>Buffer_Not_Initiaized</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:
3.3.7.1.9.4.10.7 BUFFER_STATUS UNIT DESIGN

This function returns the status of the buffer based on the following algorithm:

if buffer has never been initialized then status is uninitialized
elsif buffer has no nodes in it then status is empty
else status is available

3.3.7.1.9.4.10.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.7.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.7.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.7.4 LOCAL DATA

None.

3.3.7.1.9.4.10.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.7.6 PROCESSING

The following describes the processing performed by this part:

    function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is
-- declaration section

Current_Length : INTEGER renames Buffer.Current_Length;
Status : Buffer_Statuses;

-- begin function Buffer_Status

begin

if Current_Length = -1 then
    Status := Uninitialized;
elsif Current_Length = 0 then
    Status := Empty;
else
    Status := Available;
end if;

return Status;

end Buffer_Status;

3.3.7.1.9.4.10.7.7 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:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Statuses</td>
<td>discrete type</td>
<td>Empty, Available, Uninitialized</td>
<td>Used to indicate the status of the buffer</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.7.8 LIMITATIONS

None.

3.3.7.1.9.4.10.8 BUFFER_LENGTH UNIT DESIGN

This function returns the length of the current buffer.

If the buffer has not been initialized, a Buffer_Not_Initialized exception is raised.

3.3.7.1.9.4.10.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.8.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>in out</td>
<td>FIFO buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.8.4 LOCAL DATA

None.

3.3.7.1.9.4.10.8.5 PROCESS CONTROL

Not applicable.
3.3.7.1.9.4.10.8.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Buffer_Length (Buffer : in Buffers) return NATURAL is
-- ---------
-- --declaration section
-- ---------

Current_Length : INTEGER renames Buffer.Current_Length;

-- ---------
-- --begin function Buffer_Length
-- ---------

begin

-- --make sure the buffer has a length
if Current_Length = -1 then
  raise Buffer_Not_Initialized;
end if;

return Current_Length;

end Buffer_Length;
```

3.3.7.1.9.4.10.8.7 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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_Buffer:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_NotInitialized</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_NotInitialized</td>
<td>Raised if an attempt is made to use an uninitialized buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.9 DOT_NEXT UNIT DESIGN

Given an input pointer P, this function returns the value of P.Next.

3.3.7.1.9.4.10.9.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.4.10.9.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.9.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node whose &quot;next&quot; entry is to be returned</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.9.4 LOCAL DATA

None.
3.3.7.1.9.4.10.9.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.9.6  PROCESSING

The following describes the processing performed by this part:

```plaintext
function Dot_Next (Ptr : in Pointers) return Pointers is
begin
  return Ptr.Next;
end Dot_Next;
```

3.3.7.1.9.4.10.9.7  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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the buffer; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the buffer</td>
</tr>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.9.8  LIMITATIONS

None.

3.3.7.1.9.4.10.10  SET_NEXT UNIT DESIGN

Given an two input pointers, P and Q, this procedure sets P.Next equal to Q.

3.3.7.1.9.4.10.10.1  REQUIREMENTS ALLOCATION

None.
3.3.7.1.9.4.10.10.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.10.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node whose &quot;next&quot; entry is to be modified</td>
</tr>
<tr>
<td>Ptr dot Next</td>
<td>Pointers</td>
<td>in</td>
<td>Value to which Ptr.Next is to be set</td>
</tr>
</tbody>
</table>

3.3.7.1.9.4.10.10.4 LOCAL DATA

None.

3.3.7.1.9.4.10.10.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.10.6 PROCESSING

The following describes the processing performed by this part:

```pascal
procedure Set_Next (Ptr : in Pointers;
                   Ptr dot Next : in Pointers) is
begin
    Ptr.Next := Ptr dot Next;
end Set_Next;
```

3.3.7.1.9.4.10.10.7 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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:
### 3.3.7.1.9.4.10.10.8 LIMITATIONS

None.

### 3.3.7.1.9.5 NONBLOCKING_CIRCULAR_BUFFER PACKAGE DESIGN (CATALOG #:P333-0)

This generic package defines the data type and contains the operations required to perform circular buffering operations on incoming data. These operations are performed in a non-blocking fashion such that if the buffer is full, incoming data will overwrite old data. The head of the buffer always points to a dummy node. The first node following the dummy node contains the next piece of data to be retrieved. The tail always points to where the next element should be added. If the tail points to the element immediately in front of the head, the buffer is empty. If the tail points to the same element as the head, the buffer is full. This is illustrated below.

**Empty circular buffer:**  
+-+ <----->Head +-+ +-+ +-+ +-+ +-+ +

**Full circular buffer:**  
Tail----->+-+ <----->Head +-+ +-+ +-+ +-+ +-+ +

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

#### 3.3.7.1.9.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

#### 3.3.7.1.9.5.2 LOCAL ENTITIES DESIGN

None.

#### 3.3.7.1.9.5.3 INPUT/OUTPUT

**GENERIC PARAMETERS:**

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial_Buffe r_Size</td>
<td>POSITIVE</td>
<td>N/A</td>
<td>Maximum number of elements which can be in the buffer at any given time</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.4 LOCAL DATA

None.

3.3.7.1.9.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.6 PROCESSING

The following describes the processing performed by this part:

separate (Abstract_Data_Structures)
package body Nonblocking_Circular_Buffer is

end Nonblocking_Circular_Buffer;

3.3.7.1.9.5.7 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:

Data types:

The following data types were previously defined in this part's package specification:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL subtype</td>
<td>0 .. Buffer_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Buffer_Statuses</td>
<td>discrete type</td>
<td>Empty, Available, Full</td>
<td>Used to indicate the status of the buffer</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists Buffers</td>
<td>array record</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data objects:

The following data objects were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Size</td>
<td>POSITIVE</td>
<td>Initial_Buffer_Size</td>
<td>Number of usable elements in a buffer</td>
</tr>
</tbody>
</table>

Exceptions:

The following exceptions were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.8 LIMITATIONS
None.

3.3.7.1.9.5.9 LLCSC DESIGN
None.
3.3.7.1.9.5.10  UNIT DESIGN

3.3.7.1.9.5.10.1  CLEAR_BUFFER UNIT DESIGN

This procedure clears a buffer by setting the Head to 0, the Tail to 1, and the length to 0.

3.3.7.1.9.5.10.1.1  REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.1.2  LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.1.3  INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>Nonblocking circular buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.1.4  LOCAL DATA

None.

3.3.7.1.9.5.10.1.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.1.6  PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Clear_Buffer (Buffer : out Buffers) is

-- declaration section

Head        : Buffer.Range renames Buffer.Head;
Tail         : Buffer.Range renames Buffer.Tail;
```

-- -----------------------------------
-- --begin procedure Clear_Buffer
-- -----------------------------------

begin

  Head  := 0;
  Tail  := 1;
  Current_Length := 0;

end Clear_Buffer;

3.3.7.1.9.5.10.1.7 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:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferRange</td>
<td>NATURAL</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Buffer_Size</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.1.8 LIMITATIONS

None.

3.3.7.1.9.5.10.2 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the end of the buffer, overwriting old data if the buffer is full. If data was overwritten, both the head and tail of the buffer are adjusted to reflect the current status of the buffer. If data was not overwritten, only the tail of the buffer is adjusted.
3.3.7.1.9.5.10.2.1 REQUIREMENTS ALLOCATION
This part meets CAMP requirement R126.

3.3.7.1.9.5.10.2.2 LOCAL ENTITIES DESIGN
None.

3.3.7.1.9.5.10.2.3 INPUT/OUTPUT
FORMAL PARAMETERS:
The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>Circular buffer being accessed</td>
</tr>
<tr>
<td>New_Element</td>
<td>Elements</td>
<td>in</td>
<td>Element to be added to the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.2.4 LOCAL DATA
None.

3.3.7.1.9.5.10.2.5 PROCESS CONTROL
Not applicable.

3.3.7.1.9.5.10.2.6 PROCESSING
The following describes the processing performed by this part:

```
procedure Add_Element (New_Element : in Elements;
                      Buffer      : in out Buffers) is

-- declaration section
--

    Head       : Buffer_Range renames Buffer.Head;
    Tail       : Buffer_Range renames Buffer.Tail;
    Current_Length : Buffer_Range renames Buffer.Current_Length;
    List       : Lists renames Buffer.List;

-- begin procedure Add_Element
--

begin

    List(Tail) := New_Element;
```
if Head = Tail then

-- buffer was already full and an element was overwritten; therefore,
-- both head and tail need to be advanced, but Current_Length does
-- not need to be changed

if Tail = Buffer_Size then
  Head := 0;
  Tail := 0;
else
  Head := Head + 1;
  Tail := Tail + 1;
end if;
else

-- buffer was not already full; therefore, the Current_Length needs
-- to be increment and only the tail needs to be advanced

if Tail = Buffer_Size then
  Tail := 0;
else
  Tail := Tail + 1;
end if;

Current_Length := Current_Length + 1;
end if;

end Add_Element;

3.3.7.1.9.5.10.2.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:
The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td></td>
<td>subtype</td>
<td>Buffer_Size</td>
<td></td>
</tr>
</tbody>
</table>

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferSize</td>
<td>POSITIVE</td>
<td>Initial</td>
<td>Number of usable elements in a Buffer_Size</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.2.8 LIMITATIONS

None.

3.3.7.1.9.5.10.3 RETRIEVE_ELEMENT UNIT DESIGN

This procedure returns the first element in the circular buffer. If there are no elements in the buffer, a Buffer_Empty exception is raised.

3.3.7.1.9.5.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.3.2 LOCAL ENTITIES DESIGN

None.
3.3.7.1.9.5.10.3.3  INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>Circular buffer being accessed</td>
</tr>
<tr>
<td>Old_Element</td>
<td>Elements</td>
<td>out</td>
<td>Element retrieved from the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.3.4  LOCAL DATA

None.

3.3.7.1.9.5.10.3.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.3.6  PROCESSING

The following describes the processing performed by this part:

```ada
procedure Retrieve_Element (Buffer : in out Buffers;
                           Old_Element : out Elements) is

-- declaration section
--

Head : Buffer_Range renames Buffer.Head;
Tail  : Buffer_Range renames Buffer.Tail;
Current_Length : Buffer_Range renames Buffer.Current_Length;
List : Lists renames Buffer.List;

--

begin

-- make sure there is something there to retrieve
if Current_Length = 0 then
  raise Buffer_Empty;
end if;

-- advance the head to get to the next element to go out
if Head = Buffer_Size then
  Head := 0;
else
  Head := Head + 1;
end if;
```


-- now retrieve the element and update the state of the buffer
   Old_Element := List(Head);
   Current_Length := Current_Length - 1;

   end Retrieve_Element;

3.3.7.1.9.5.10.3.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>subtype Buffer_Size</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>information</td>
</tr>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>Array of elements</td>
</tr>
</tbody>
</table>

Data objects:

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

The following table summarizes the exceptions required by this part and defined in the package specification of `Nonblocking_Circular_Buffer`:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

#### 3.3.7.1.9.5.10.3.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

#### 3.3.7.1.9.5.10.4 PEEK UNIT DESIGN

This function returns the data contained in the first element in the buffer without changing the state of the buffer (i.e., the element is not removed from the buffer).

If there are no elements in the buffer, a `Buffer_Empty` exception is raised.

#### 3.3.7.1.9.5.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

#### 3.3.7.1.9.5.10.4.2 LOCAL ENTITIES DESIGN

None.

#### 3.3.7.1.9.5.10.4.3 INPUT/OUTPUT

**FORMAL PARAMETERS:**
The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>Circular buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.4.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spot</td>
<td>Buffer_Range</td>
<td>N/A</td>
<td>Marks the spot in the buffer containing the element to be looked at</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.4.6 PROCESSING

The following describes the processing performed by this part:

```haskell
function Peek (Buffer : in Buffers) return Elements is

-- declaration section
--
Head        : Buffer_Range renames Buffer.Head;
Current_Length : Buffer_Range renames Buffer.Current_Length;
List        : Lists renames Buffer.List;
Spot        : Buffer_Range;

-- begin function Peek
--

begin

-- make sure there is something to peek at
if Current_Length = 0 then
   raise Buffer_Empty;
end if;

-- determine location of desired element
if Head = Buffer_Size then

```

```
Spot := 0;
else
    Spot := Head + 1;
end if;

-- --return requested element
return List(Spot);

end Peek;

3.3.7.1.9.5.10.4.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the buffer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 .. Buffer_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>Array of elements</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the types required by this part and defined in the package specification of Nonblocking_Circular_Buffer:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Size</td>
<td>POSITIVE</td>
<td>Initial_Buffer_Size</td>
<td>Number of usable elements in a buffer</td>
</tr>
</tbody>
</table>

Exceptions:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.5 BUFFER_STATUS UNIT DESIGN

This function returns the current status of the buffer according to the following algorithm:

if there are no elements in the buffer then buffer status is empty
else
if the buffer contains the maximum number of elements buffer status is full
else buffer status is available
end if;

3.3.7.1.9.5.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.5.2 LOCAL ENTITIES DESIGN

None.
3.3.7.1.9.5.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:
The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>Circular buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.5.4 LOCAL DATA

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Buffer_Statuses</td>
<td>N/A</td>
<td>Current status of the buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.5.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is
  -- declaration section
  Status : Buffer.Statuses;

  begin
    if Current_Length = 0 then
      Status := Empty;
    elsif Current_Length = Buffer_Size then
      Status := Full;
    else
      Status := Available;
```
end if;
return Status;
end Buffer_Status;

3.3.7.1.9.5.10.5.7 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:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL subtype</td>
<td>0 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Buffer_Statuses</td>
<td>discrete subtype</td>
<td>Buffer_Size</td>
<td>Used to indicate the status of the buffer</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>Empty,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Available,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Full</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Size</td>
<td>POSITIVE</td>
<td>Initial</td>
<td>Number of usable elements in a Buffer_Size buffer</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.5.8 LIMITATIONS

None.
3.3.7.1.9.5.10.6 BUFFER_LENGTH UNIT DESIGN

This function returns the current length of the buffer.

3.3.7.1.9.5.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffers</td>
<td>out</td>
<td>Circular buffer being accessed</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.6.4 LOCAL DATA

None.

3.3.7.1.9.5.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.6.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
function Buffer_Length (Buffer : in Buffers) return Buffer_Range is
begin
    return Buffer.Current_Length;
end Buffer_Length;
```

3.3.7.1.9.5.10.6.7 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:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer_Range</td>
<td>NATURAL</td>
<td>0 .. Buffer_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.5.10.6.8 LIMITATIONS

None.

3.3.7.1.9.6 UNBOUNDED_PRIORITY_QUEUE PACKAGE DESIGN (CATALOG #F334-0)

This generic package defines the data type and contains the operations required to perform priority queueing operations on incoming data. The head of the queue always points to a dummy node. The node following the dummy node contains the element with the highest priority. The tail always points to the element with the lowest priority.

The elements will be ordered in the queue such that: 1) Elements with higher priorities are placed before those with lower priorities. 2) Elements with the same priority are arranged in the queue in a first-in-first-out manner.

A queue must be initialized before it is used. If an attempt is made to use an uninitialized queue, the exception Queue_Not_Initialized will be raised. The Initialized_Queue procedure returns an initialized queue. The Clear_Queue procedure returns the nodes of a queue to the available space list and then returns an initialized queue.

An available space list is maintained local to this part. When this part is elaborated the available space list will have a dummy node plus Initial_Available_Space_Size nodes. When nodes are added to the queue, the Add_Element routine will try to get a node from the available space list before attempting to allocate more memory. When the Retrieve_Element routine is called, the unused node will be returned to the available space list for later use. The memory committed to the available space may be deallocated by calling the Free_Memory procedure.
The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.2 LOCAL ENTITIES DESIGN

Data structures:

An available space list is maintain local to this part's package body.

Subprograms:

The following subprograms are contained local to this body:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free_Node</td>
<td>procedure</td>
<td>Instantiation of UNCHECKED DEALLOCATION</td>
</tr>
<tr>
<td>Dot_Next</td>
<td>function</td>
<td>Given a pointer P, this function returns the value of P.Next</td>
</tr>
<tr>
<td>Set_Next</td>
<td>procedure</td>
<td>Given two points P &amp; Q, this procedure sets P.Next = Q</td>
</tr>
</tbody>
</table>

The following subprograms are contained in this part as a result of renaming operations on identically named routines contained in the locally instantiated Available_Space_Operations package.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
<tr>
<td>Save_Node</td>
<td>procedure</td>
<td>Handles placing a node in the available space list</td>
</tr>
<tr>
<td>Save_List</td>
<td>procedure</td>
<td>Handles placing a list of nodes in the available space list</td>
</tr>
</tbody>
</table>

This package body contains code to initialize the Available_Space_List. This code is executed when the package is elaborated. If the generic formal object Initial_Available_Space_Size is greater than or equal to 1, this routine then places the requested number of nodes (in addition to the dummy node) in the available space list.

3.3.7.1.9.6.3 INPUT/OUTPUT

GENERIC PARAMETERS:
The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the queue</td>
</tr>
<tr>
<td>Priorities</td>
<td>private</td>
<td>User defined type determining the priority of the node</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial_Available_Space_Size</td>
<td>NATURAL</td>
<td>Number of available nodes to be initially placed in the available space list</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>function</td>
<td>Used to determine ordering of priorities</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.4 LOCAL DATA

Data objects:
The following table summarizes the data objects defined by this part as the result of renames:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Len</td>
<td>INTEGER</td>
<td>Current_Length</td>
<td>Indicates the current length of the available space list</td>
</tr>
<tr>
<td>Available_Pointers</td>
<td>available_Space.</td>
<td>Head</td>
<td>Points to the head node in the available space list</td>
</tr>
<tr>
<td>Available_Pointers</td>
<td>available_Space.</td>
<td>Tail</td>
<td>Points to the tail node in the available space list</td>
</tr>
</tbody>
</table>
3.3.7.1.9.6.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.6  PROCESSING

The following describes the processing performed by this part:

with UNCHECKED_DEALLOCATION;
separate (Abstract_Data_Structures)
package body Unbounded_Priority_Queue is

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

-- -- this pointers is accessed ONLY when setting up the Available_Space
Initial_Head   : Pointers := new Nodes;

Available_Space : Queues := (Current_Length => 0,
    Head   => Initial_Head,
    Tail   => Initial_Head);

Available_Length : INTEGER renames Available_Space.Current_Length;
Available_Head   : Pointers renames Available_Space.Head;
Available_Tail   : Pointers renames Available_Space.Tail;

procedure Free is new UNCHECKED_DEALLOCATION
  (Object  => Nodes,
   Name   => Pointers);

procedure Free_Node (Which_Node : in out Pointers)
  renames Free;

function Dot_Next (Ptr : in Pointers) return Pointers;

procedure Set_Next (Ptr       : in Pointers;
   Ptr_dot_Next : in Pointers);

package Available_Space_Operations is new
Available_Space_List_Operations
  (Nodes   => Nodes,
   Pointers => Pointers,
   Available_Length => Available_Length,
   Available_Head   => Available_Head,
   Available_Tail   => Available_Tail);

function New_Node return Pointers
  renames Available_Space_Operations.New_Node;

procedure Save_Node (Saved_Node : in Pointers)
  renames Available_Space_Operations.Save_Node;

procedure Save_List (Saved_Head : in Pointers;
   Saved_Tail : in Pointers;
   Node_Count : in POSITIVE)
renames Available_Space_Operations.Save_List;

--begin package Unbounded_Priority_Queue
--(see header for package body for details)
begin
  --set up available space list if one is desired
  if Initial_Available_Space_Size > 0 then
    Add Nodes To Available_Space_List:
    for I In 1..Initial_Available_Space_Size loop
      Available_Tail.Next := NEW Nodes;
      Available_Tail := Available_Tail.Next;
    end loop Add_Nodes_to_Available_Space_List;
    Available_Length := Initial_Available_Space_Size;
  end if;
end Unbounded_Priority_Queue;

3.3.7.1.9.6.7 UTILIZATION OF OTHER ELEMENTS

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

Subprograms and task entries:

The following table describes the subroutines required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unchecked_Deallocation</td>
<td>generic</td>
<td>N/A</td>
<td>Used to deallocate memory</td>
</tr>
</tbody>
</table>

Exceptions:

The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage_Error</td>
<td>Raised when an attempt is made to dynamically allocate more memory than is available</td>
</tr>
</tbody>
</table>

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:

Packages:

The following table describes the packages required by this part and specified in the package body of the Abstract_Data_Structures package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Space_List_Operations</td>
<td>generic</td>
<td>Contains the routines required to retrieve a node from and place a node in the available space list</td>
</tr>
</tbody>
</table>

Data types:

The following data types were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Statuses</td>
<td>discrete type</td>
<td>Empty, Available, Uninitialized</td>
<td>Used to indicate the status of the queue</td>
</tr>
</tbody>
</table>

The following data types were previously defined in the private portion of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

Exceptions:

The following exceptions were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty queue</td>
</tr>
<tr>
<td>Queue_NotInitialized</td>
<td>Indicates an attempt was made to use an uninitialized queue</td>
</tr>
</tbody>
</table>
3.3.7.1.9.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>StorageError</td>
<td>Raised during elaboration of this package if an attempt is made to allocate memory when no more is available</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.9 LLCSC DESIGN

None.

3.3.7.1.9.6.10 UNIT DESIGN

3.3.7.1.9.6.10.1 INITIALIZE UNIT DESIGN

This procedure initializes a queue by placing a dummy node in it, pointing the head and the tail to the dummy node, and setting the length to 0.

3.3.7.1.9.6.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>Queues</td>
<td>in out</td>
<td>Unbounded priority queue being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.1.4 LOCAL DATA

None.
3.3.7.1.9.6.10.1.5  PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.1.6  PROCESSING

The following describes the processing performed by this part:

procedure Initialize (Queue : in out Queues) is

-- declare section

Current-Length : INTEGER    renames Queue.Current-Length;
Head            : Pointers renames Queue.Head;
Tail            : Pointers renames Queue.Tail;

begin
    if Current-Length = -1 then
        -- handle an uninitialized queue
        Head := New Node;
        Tail := Head;
        Current-Length := 0;
    elsif Current-Length > 0 then
        -- handle a queue that has something in it
        Clear_Queue(Queue => Queue);
    else
        -- current length = 0 so it is already initialized
        NULL;
    end if;
end Initialize;

3.3.7.1.9.6.10.1.7  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 in the package specification of Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear_Queue</td>
<td>procedure</td>
<td>Returns all the nodes in a queue to the available space list</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
</tbody>
</table>

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.1.8 LIMITATIONS

None.

3.3.7.1.9.6.10.2 CLEAR_QUEUE UNIT DESIGN

This procedure removes the nodes from a queue and places them in an available space list.

The Queue_Not_Initiajized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.
3.3.7.1.9.6.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>Queues</td>
<td>in out</td>
<td>Unbounded priority queue being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.2.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained local to this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThisNode</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the node to be returned to the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.2.5 PROCESS CONTROL

Not applicable

3.3.7.1.9.6.10.2.6 PROCESSING

The following describes the processing performed by this part:

```literate
procedure Clear_Queue (Queue : in out Queues) is

  ----declaration section

  Current_Length : INTEGER renames Queue.Current_Length;
  Head           : Pointers renames Queue.Head;
  Tail           : Pointers renames Queue.Tail;
  This_Node      : Pointers;

  -- --begin procedure Clear_Queue
```
begin

-- --make sure this is an initialized queue
if Current_Length = -1 then
    raise Queue_Not_Initialized;
elsif Current_Length > 0 then
      --placed nodes in the available space list
    Save_List (Saved_Head => Head.Next,
               Saved_Tail => Tail,
               Node_Count => Current_Length);

-- --reinitialize queue variables
    Current_Length := 0;
    Head.Next := NULL;
    Tail := Head;
end if;

end Clear_Queue;

3.3.7.1.9.6.10.2.7 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 in the package body of Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Save_List</th>
<th>procedure</th>
<th>Handles placing a list of nodes in the available space list</th>
</tr>
</thead>
</table>

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>
Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Not_Initialed</td>
<td>Raised if an attempt is made to manipulate an uninitialized queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.3 FREE_MEMORY UNIT DESIGN

This procedure deallocates the memory taken up by the available space list.

3.3.7.1.9.6.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.3.3 INPUT/OUTPUT

None.

3.3.7.1.9.6.10.3.4 LOCAL DATA

Data objects:
The following table describes the data objects maintained by this part:
### 3.3.7.1.9.6.10.3.5 PROCESS CONTROL

Not applicable.

### 3.3.7.1.9.6.10.3.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Free_Memory is
-- ------------------
-- --declaration section
-- ------------------

    Node_to_be_Freed : Pointers;

-- ------------------
-- --begin procedure Free_Memory
-- ------------------

begin

    Clear_Out_Available_Space_List:
        while Available_Head /= Available_Tail loop
            Node_To_Be_Freed := Available_Head;
            Available_Head := Available_Head.Next;
            Free_Node (Which_Node => Node_to_be_Freed);
        end loop
    Clear_Out_Available_Space_List;

    Available_Length := 0;

end Free_Memory;
```

### 3.3.7.1.9.6.10.3.7 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 in the package body of Unbounded_Priority_Queue:
### Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

### Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Length</td>
<td>INTEGER</td>
<td>Available_Space.</td>
<td>Indicates the current length of the available space list</td>
</tr>
<tr>
<td>Available_Head</td>
<td>Pointers</td>
<td>Available_Space.</td>
<td>Points to the head node in the available space list</td>
</tr>
<tr>
<td>Available_Tail</td>
<td>Pointers</td>
<td>Available_Space.</td>
<td>Points to the tail node in the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.3.8 LIMITATIONS

None.

3.3.7.1.9.6.10.4 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the queue. The elements are added such that the new element is added before the first element which has a smaller priority and after all other elements which a greater or equal priority.

The Queue_Empty exception is raised if this routine is called with an empty queue.

The Queue_Not_InitiaIized exception is raised if this routine is called with an uninitialized queue.
The Storage_Error exception is raised if a call to this routine requires memory to be dynamically allocated when no more memory is available.

3.3.7.1.9.6.10.4.1 REQUIREMENTS ALLOCATION
This part meets CAMP requirement R165.

3.3.7.1.9.6.10.4.2 LOCAL ENTITIES DESIGN
None.

3.3.7.1.9.6.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:
The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Element</td>
<td>Elements</td>
<td>in</td>
<td>Element to be placed in the queue</td>
</tr>
<tr>
<td>New_Priority</td>
<td>Priorities</td>
<td>in</td>
<td>Priority of the element to be placed in the queue</td>
</tr>
<tr>
<td>Queue</td>
<td>Queues</td>
<td>in out</td>
<td>Unbounded priority queue being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.4.4 LOCAL DATA

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the element which will go before the new element</td>
</tr>
<tr>
<td>Here</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the node to be added to the queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.4.6 PROCESSING

The following describes the processing performed by this part:

```pascal
procedure Add_Element (New_Element : in Elements;
                        New_Priority : in Priorities;
```
CAMP Software Detailed Design Document

Queue : in out Queues) is

-- declaration section

Current_Length : INTEGER renames Queue.Current_Length;
Head : Pointers renames Queue.Head;
Tail : Pointers renames Queue.Tail;

Before : Pointers;
Here : Pointers;

-- begin procedure Add_Element
--

begin

-- make sure queue has been initialized
if Current_Length = -1 then
    raise Queue_Not_Initiaized;
end if;

-- find the nodes which are go before and after the new element
Before := Head;
loop
    exit when (Before = Tail) or else
        (New_Priority > Before.Next.Priority);
    Before := Before.Next;
end loop;

-- now get a new node
Here := New_Node;

-- set up the new node
Here.Priority := New_Priority;
Here.Data := New_Element;
Here.Next := Before.Next;
Before.Next := Here;

-- readjust the tail, if required
if Before = Tail then
    Tail := Here;
end if;

-- now adjust the queue
Current_Length := Current_Length + 1;
end Add_Element;

3.3.7.1.9.6.10.4.7 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 in the package body of Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free_Node</td>
<td>procedure</td>
<td>Instantiation of UNCHECKED DEALLOCATION</td>
</tr>
</tbody>
</table>

The following table describes the subroutines required by this part and defined as generic formal subroutines to the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&gt;&quot;</td>
<td>function</td>
<td>Used to determine ordering of priorities</td>
</tr>
</tbody>
</table>

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
### 3.3.7.1.9.6.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage_Error</td>
<td>Raised if an attempt is raised to allocate memory when no more is available</td>
</tr>
<tr>
<td>Queue_NotInitialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized queue</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.6.10.5 RETRIEVE_ELEMENT UNIT DESIGN

This procedure returns the first element in the queue.

The QueueEmpty exception is raised if this routine is called with an empty queue.

The Queue_NotInitialized exception is raised if this routine is called with an uninitialized queue.

#### 3.3.7.1.9.6.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

#### 3.3.7.1.9.6.10.5.2 LOCAL ENTITIES DESIGN

None.

#### 3.3.7.1.9.6.10.5.3 INPUT/OUTPUT:

**FORMAL PARAMETERS:**

The following table describes this part's formal parameters:
3.3.7.1.9.6.10.5.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This_Node</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the node to be returned to the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.5.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Retrieve_Element (Queue : in out Queues;
                           Old_Element :   out Elements) is

-- ---- declaration section

Current_Length : INTEGER renames Queue.Current_Length;
Head            : Pointers renames Queue.Head;

This_Node : Pointers;

-- ---- begin procedure Retrieve_Element

begin

-- make sure an element is available
if Current_Length = -1 then
  raise Queue_NotInitialized;
elsif Current_Length = 0 then
  raise Queue_Empty;
end if;
```
CAMP Software Detailed Design Document

-- save dummy node in the available space list
This_Node := Head;
Head := Head.Next;
Save_Node (Saved_Node => This_Node);

-- retrieve element (its node becomes the new dummy node)
Old_Element := Head.Data;

-- update queue status
Current_Length := Current_Length - 1;

end Retrieve_Element;

3.3.7.1.9.6.10.5.7 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 in the package body of Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save_Node</td>
<td>procedure</td>
<td>Handles placing a node in the available space list</td>
</tr>
</tbody>
</table>

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:
3.3.7.1.9.6.10.5.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Empty</td>
<td>Raised if an attempt is made to look at or retrieve elements from an empty queue</td>
</tr>
<tr>
<td>Queue_Not_Initialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.6 PEEK UNIT DESIGN

This function returns the value of the first element in the queue, but does not change the state of the queue (i.e., the node is not actually removed from the queue).

The Queue_Empty exception is raised if this routine is called with an empty queue.

The Queue_Not_Initialized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:
3.3.7.1.9.6.10.6.4 LOCAL DATA

None.

3.3.7.1.9.6.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.6.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
function Peek (Queue : in Queues) return Elements is
--- declaration section
---

  Current_Length : INTEGER renames Queue.Current Length;
  Head : Pointers renames Queue.Head;

--- begin function Peek
---

begin

  -- make sure something is there to look at
  if Current_Length = -1 then
    raise Queue Not Initialized;
  elsif Current_Length = 0 then
    raise Queue Empty;
  end if;

  return Head.Next.Data;

end Peek;
```

3.3.7.1.9.6.10.6.7 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:
Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty queue</td>
</tr>
<tr>
<td>Queue_Not_Initiated</td>
<td>Indicates an attempt was made to use an uninitialized queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Empty</td>
<td>Raised if an attempt is made to look at or retrieve from an empty queue</td>
</tr>
<tr>
<td>Queue_Not_Initiated</td>
<td>Raised if an attempt is made to manipulate an uninitialized queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.7 QUEUE_STATUS UNIT DESIGN

This function returns the status of the queue based on the following algorithm:

if the queue has not been initialized then queue status is uninitialized else if no elements are in the queue then queue status is empty else queue status is available end if;
3.3.7.1.9.6.10.7.1 REQUIREMENTS ALLOCATION
This part meets CAMP requirement R165.

3.3.7.1.9.6.10.7.2 LOCAL ENTITIES DESIGN
None.

3.3.7.1.9.6.10.7.3 INPUT/OUTPUT

FORMAL PARAMETERS:
The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>Queues</td>
<td>in out</td>
<td>Unbounded priority queue being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.7.4 LOCAL DATA

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Queue_Statuses</td>
<td>N/A</td>
<td>Status of the queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.7.6 PROCESSING

The following describes the processing performed by this part:

    function Queue_Status (Queue : in Queues) return Queue_Statuses is
    -- declaration section
    --
    Current_Length : INTEGER renames Queue.Current_Length;
    Status        : Queue_Statuses;
    --
    -- begin function Queue_Status
begin
    if Current_Length = -1 then
        Status := Uninitialized;
    elsif Current_Length = 0 then
        Status := Empty;
    else
        Status := Available;
    end if;

    return Status;
end Queue_Status;

3.3.7.1.9.6.10.7.7 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:

Data types:

The following table describes the data types required by this part and defined in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Statuses</td>
<td>discrete</td>
<td>Empty, Available, Uninitialized</td>
<td>Used to indicate the status of the queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.7.8 LIMITATIONS

None.

3.3.7.1.9.6.10.8 QUEUE_LENGTH UNIT DESIGN

This function returns the length of a queue. The Queue Not Initialized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.
3.3.7.1.9.6.10.8.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>Queues</td>
<td>in</td>
<td>Unbounded priority queue being</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.8.4 LOCAL DATA

None.

3.3.7.1.9.6.10.8.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.8.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Queue_Length (Queue : in Queues) return NATURAL is

begin
    if CurrentLength = -1 then
        raise Queue_Not_Initlized;
    end if;

    return CurrentLength;
end Queue_Length;
```

3.3.7.1.9.6.10.8.7 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:

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Not_Initialized</td>
<td>Indicates an attempt was made to use an uninitialized queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue_Not_Initialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.9 DOT_NEXT UNIT DESIGN

Given an input pointer P, this function returns the value of P.Next.
3.3.7.1.9.6.10.9.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.6.10.9.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.9.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node whose &quot;next&quot; entry is to be returned</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.9.4 LOCAL DATA

None.

3.3.7.1.9.6.10.9.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.9.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Dot_Next (Ptr : in Pointers) return Pointers is
begin
    return Ptr.Next;
end Dot_Next;
```

3.3.7.1.9.6.10.9.7 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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.9.8 LIMITATIONS

None.

3.3.7.1.9.6.10.10 SET_NEXT UNIT DESIGN

Given an two input pointers, P and Q, this procedure sets P.Next equal to Q.

3.3.7.1.9.6.10.10.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.6.10.10.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.10.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node whose &quot;next&quot; entry is to be modified</td>
</tr>
<tr>
<td>Ptr_dot_Next</td>
<td>Pointers</td>
<td>in</td>
<td>Value to which Ptr.Next is to be set</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.10.4 LOCAL DATA

None.

3.3.7.1.9.6.10.10.5 PROCESS CONTROL

Not applicable.
3.3.7.1.9.6.10.10.6 PROCESSING

The following describes the processing performed by this part:

```haskell
procedure Set_Next (Ptr        : in Pointers;
                    Ptr_dot_Next : in Pointers) is
begin
  Ptr.Next := Ptr_dot_Next;
end Set_Next;
```

3.3.7.1.9.6.10.10.7 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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the queue; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the queue</td>
</tr>
<tr>
<td>Queues</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the queue</td>
</tr>
</tbody>
</table>

3.3.7.1.9.6.10.10.8 LIMITATIONS

None.

3.3.7.1.9.7 BOUNDED_STACK PACKAGE DESIGN (CATALOG #P335-0)

This generic package defines the data type and contains the operations required to perform last-in-first-out stacking operations on incoming data. The top of the stack always points to the last element added to the stack and the next element to be removed. When top equals 0, the stack is empty. When top equals Stack_Size, the stack is full.

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

3.3.7.1.9.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP require R166.
3.3.7.1.9.7.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial_Scalar</td>
<td>POSITIVE</td>
<td>N/A</td>
<td>Maximum number of elements which can be in the stack at any given time</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.4 LOCAL DATA

None.

3.3.7.1.9.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.6 PROCESSING

The following describes the processing performed by this part:

separate (Abstract_Data_structures)
package body Bounded_Stack is
end Bounded_Stack;
### 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:

#### Data types:

The following data types were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Length</td>
<td>POSITIVE</td>
<td>1 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Range Stacks</td>
<td>subtype</td>
<td>Stack Size</td>
<td></td>
</tr>
<tr>
<td>Stacks</td>
<td>limited</td>
<td>N/A</td>
<td>List of data along with relevant</td>
</tr>
<tr>
<td>Stack Statuses</td>
<td>discrete</td>
<td>Empty, Available, Full</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td>Used to indicate the status of the stack</td>
</tr>
</tbody>
</table>

The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Dimensions</td>
<td>Stack Dimensions</td>
<td>1 .. 'LAST</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Range Stacks</td>
<td>subtype</td>
<td>N/A</td>
<td>List of data along with relevant</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td></td>
<td>information</td>
</tr>
</tbody>
</table>

#### Data objects:

The following data objects were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Size</td>
<td>POSITIVE</td>
<td>Initial Stack Size</td>
<td>Number of elements in the stack</td>
</tr>
</tbody>
</table>

#### Exceptions:

The following exceptions were previously defined in this part's package specification:
3.3.7.1.9.7.8 LIMITATIONS

None.

3.3.7.1.9.7.9 LLCSC DESIGN

None.

3.3.7.1.9.7.10 UNIT DESIGN

3.3.7.1.9.7.10.1 CLEAR_STACK UNIT DESIGN

This procedure clears a stack by setting the top to 0.

3.3.7.1.9.7.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Bounded stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.1.4 LOCAL DATA

None.
3.3.7.1.9.7.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.1.6 PROCESSING

The following describes the processing performed by this part:

procedure Clear_Stack (Stack : out Stacks) is
begin
  Stack.Top := 0;
end Clear_Stack;

3.3.7.1.9.7.10.1.7 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:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_</td>
<td>POSITIVE</td>
<td>1 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Length_</td>
<td>subtype</td>
<td>Stack_Size</td>
<td></td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.1.8 LIMITATIONS

None.
3.3.7.1.9.7.10.2 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the top of the stack.

A Stack_Full exception is raised if this routine is called with a full stack.

3.3.7.1.9.7.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Element</td>
<td>Elements</td>
<td>in</td>
<td>Element to be added to the stack</td>
</tr>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Bounded stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.2.4 LOCAL DATA

None.

3.3.7.1.9.7.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.2.6 PROCESSING

The following describes the processing performed by this part:

```vhdl
procedure Add_Element (New_Element : in Elements; Stack : in out Stacks) is
```

```vhdl
-- declaration section
```

```vhdl
List : Lists    renames Stack.List;
Top  : Stack_Length_Range renames Stack.Top;
```

```vhdl
-- --begin procedure Add_Element
```
begin

begin

-- make sure the stack is not already full
if Top = Stack Size then
    raise Stack_Full;
end if;

-- add element to the stack
Top := Top + 1;
List(Top) := New_Element;

end Add_Element;

3.3.7.1.9.7.10.2.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures.Bounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Length</td>
<td>POSITIVE</td>
<td>1 .. Stack_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Range</td>
<td>subtype</td>
<td>Stack_Size</td>
<td></td>
</tr>
</tbody>
</table>

The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>List of elements</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>
Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>POSITIVE</td>
<td>Initial</td>
<td>Number of elements in the stack</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td>Stack</td>
<td>Size</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Full</td>
<td>Error condition raised if an attempt is made to add elements to a full stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Full</td>
<td>Raised if an attempt is made to add elements to a full stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.3 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the top element from the stack and returns it to the calling routine.

A Stack_Error exception is raised if this routine is called with an empty stack.

3.3.7.1.9.7.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.3.2 LOCAL ENTITIES DESIGN

None.
3.3.7.1.9.7.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Bounded stack being manipulated</td>
</tr>
<tr>
<td>Old_Element</td>
<td>Elements</td>
<td>out</td>
<td>Element retrieved from the stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.3.4 LOCAL DATA

None.

3.3.7.1.9.7.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.3.6 PROCESSING

The following describes the processing performed by this part:

```pascal
procedure Retrieve_Element (Stack : in out Stacks;
                          Old_Element : out Elements) is

-- declaration section

List : Lists renames Stack.List;
Top : Stack_Length_Range renames Stack.Top;

-- begin procedure Retrieve_Element

begin
    -- make sure there is something in the stack to retrieve
    if Top = 0 then
        raise StackEmpty;
    end if;

    -- retrieve and remove the top element from the stack
    Old_Element := List(Top);
    Top       := Top - 1;

end Retrieve_Element;
```
3.3.7.1.9.7.10.3.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures.Bounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Length_</td>
<td>POSITIVE</td>
<td>1 .. Stack_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
</tbody>
</table>

The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>List of elements</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Size</td>
<td>POSITIVE</td>
<td>Initial Stack_Size</td>
<td>Number of elements in the stack</td>
</tr>
</tbody>
</table>

Exceptions:
The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Strip package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.3.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Empty</td>
<td>Raised if an attempt is made to look at or retrieve elements from an empty stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.4 PEEK UNIT DESIGN

This function returns the data in the top element of the stack, but does not remove the element from the stack.

A Stack_Empty exception is raised if this routine is called with an empty stack.

3.3.7.1.9.7.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Bounded stack being manipulated</td>
</tr>
</tbody>
</table>
3.3.7.1.9.7.10.4.4 LOCAL DATA

None.

3.3.7.1.9.7.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.4.6 PROCESSING

The following describes the processing performed by this part:

function Peek (Stack : in Stacks) return Elements is

-- declaration section

List : Lists renames Stack.List;
Top : Stack_Length_Range renames Stack.Top;

-- begin function Peek

begin

-- make sure there is something in the stack
if Top = 0 then
  raise Stack_Empty;
end if;

-- return value in top element of the stack
return List(Top);

end Peek;

3.3.7.1.9.7.10.4.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures.Bounded_Stack package:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

The following table summarizes the types required by this part and defined in the package specification of the `Abstract_Data_Structures.Bounded_Stack` package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_</td>
<td>POSITIVE</td>
<td>1 .. Stack_Size</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Length_</td>
<td>subtype</td>
<td>Stack_Size</td>
<td></td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>array</td>
<td>N/A</td>
<td>List of elements</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of `Abstract_Data_Structures. Bounded_Stack` package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_</td>
<td>POSITIVE</td>
<td>Initial</td>
<td>Number of elements in the stack</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td>Stack_Size</td>
<td></td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of `Abstract_Data_Structures. Bounded_Stack` package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackEmpty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty stack</td>
</tr>
</tbody>
</table>
3.3.7.1.9.7.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Empty</td>
<td>Raised if an attempt is made to look at or retrieve elements from an empty stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.5 STACK_STATUS UNIT DESIGN

This function returns the status of the stack based on the following algorithm:

if no elements are in the stack then stack status is empty
elsif the maximum number of elements are in the stack then stack status is full
else stack status is available
end if

3.3.7.1.9.7.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Bounded stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.5.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Stack_Statuses</td>
<td>N/A</td>
<td>Status of the stack</td>
</tr>
</tbody>
</table>
3.3.7.1.9.7.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.5.6 PROCESSING

The following describes the processing performed by this part:

```vhdl
function Stack_Status (Stack : in Stacks) return Stack_Statuses is

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

Top   : Stack_Length_Range renames Stack.Top;
Status : Stack_Statuses;

-- ------------------------------
-- --begin function Stack_Status
-- ------------------------------

begin

if Top = 0 then
    Status := Empty;
elsif Top = Stack_Size then
    Status := Full;
else
    Status := Available;
end if;

return Status;

end Stack_Status;
```

3.3.7.1.9.7.10.5.7 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:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:
### Name | Type | Range | Description
--- | --- | --- | ---
Stack Length Range | POSITIVE subtype Stack Size | 1 .. | Used to dimension the list of elements
Stack Statuses | discrete type Empty, Available, Full | | Used to indicate the status of the stack

The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Size</td>
<td>POSITIVE</td>
<td>Initial Stack Size</td>
<td>Number of elements in the stack</td>
</tr>
</tbody>
</table>

#### 3.3.7.1.9.7.10.5.8 LIMITATIONS

None.

#### 3.3.7.1.9.7.10.6 STACK_LENGTH UNIT DESIGN

This function returns the length of the stack.

#### 3.3.7.1.9.7.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

#### 3.3.7.1.9.7.10.6.2 LOCAL ENTITIES DESIGN

None.
3.3.7.1.9.7.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Bounded stack being manipulated</td>
</tr>
</tbody>
</table>
```

3.3.7.1.9.7.10.6.4 LOCAL DATA

None.

3.3.7.1.9.7.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.6.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Stack_Length (Stack : in Stacks) return Stack_Length_Range is
begin
    return Stack.Top;
end Stack_Length;
```

3.3.7.1.9.7.10.6.7 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:

Data types:

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

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>POSITIVE</td>
<td>1 ..</td>
<td>Used to dimension the list of elements</td>
</tr>
<tr>
<td>Length</td>
<td>subtype</td>
<td>Stack_Size</td>
<td></td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The following data types were previously defined in the private part of this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.7.10.6.8 LIMITATIONS

None.

3.3.7.1.9.8 UNBOUNDED_STACK PACKAGE DESIGN (CATALOG #P336-0)

This generic package performs last-in-first-out stacking operations on incoming data. The head of the stack always points to the last element added to the stack and the next element to be removed. The tail always points to a dummy node located below the oldest element on the stack. If head and tail point to the same node, the stack is empty.

An available space list is maintained local to this part. When this part is elaborated the available space list will have a dummy node plus Initial_AVAILABLE_Space_Size nodes. When nodes are added to the stack, the Add_Element routine will try to get a node from the available space list before attempting to allocate more memory. When the Retrieve_Element routine is called, the unused node will be returned to the available space list for later use. The memory committed to the available space may be deallocated by calling the Free_Memory procedure.

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

3.3.7.1.9.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.2 LOCAL ENTITIES DESIGN

Data structures:

This part maintains an available space list local to the package body.

Subprograms:

The following subprograms are contained local to this body:
### Name | Type | Description
---|---|---
| **Free Node** | procedure | Instantiation of UNCHECKED_DEALLOCATION
| **Dot_Next** | function | Given a pointer P, this function returns the value of P.Next
| **Set_Next** | procedure | Given two points P & Q, this procedure sets P.Next = Q

The following subprograms are contained in this part as a result of renaming operations on identically named routines contained in the locally instantiated Available_Space_Operations package.

### Name | Type | Description
---|---|---
| **New_Node** | function | Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node
| **Save_Node** | procedure | Handles placing a node in the available space list
| **Save_List** | procedure | Handles placing a list of nodes in the available space list

This package body contains code to initialize the Available_Space_List. This code is executed when the package is elaborated. At a minimum, this code calls the Initialize procedure to initialize the Available_Space_List so it contains a dummy node. If the generic formal object Initial_Available_Space_Size is greater than or equal to 1, this routine then places the requested number of nodes (in addition to the dummy node) in the available space list.

### 3.3.7.1.9.8.3 INPUT/OUTPUT

#### GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

**Data types:**

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

| Name   | Type  | Description
|---|---|---
| Elements | private | User defined type of data contained in the stack

**Data objects:**

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

None.

### Process Control

Not applicable.

### Processing

The following describes the processing performed by this part:

```with UNCHECKED_DEALLOCATION;
separate (Abstract_Data_Structures)
package body Unbounded_Stack is
```

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

```-- this pointer is accessed ONLY when setting up the Available_Space
Initial_Head : Pointers := new Nodes;
```

```Available_Space : Stacks := (Current_Length => 0,
   Top             => Initial_Head,
   Bottom          => Initial_Head);
```

```Available_Length : INTEGER renames Available_Space.Current_Length;
Available_Top    : Pointers renames Available_Space.Top;
Available_Bottom : Pointers renames Available_Space.Bottom;
```

```procedure Free is new UNCHECKED_DEALLOCATION
  (Object => Nodes,
   Name  => Pointers);
```

```procedure Free_Node (Which_Node : in out Pointers)
  renames Free;
```

```function Dot_Next (Ptr : in Pointers) return Pointers;
```

```procedure Set_Next (Ptr    : in Pointers;
        Ptr_dot_Next : in Pointers);
```

```package Available_Space_Operations is new
  Available_Space_List_Operations
    (Nodes        => Nodes,
     Pointers     => Pointers,
```
Available Length => Available Length,
Available Head => Available Top,
Available Tail => Available Bottom);

function New_Node return Pointers
renames Available_Space_Operations.New_Node;

procedure Save_Node (Saved_Node : in Pointers)
renames Available_Space_Operations.Save_Node;

procedure Save_List (Saved_Head : in Pointers;
Saved_Tail : in Pointers;
Node_Count : in POSITIVE)
renames Available_Space_Operations.Save_List;

--begin package Unbounded_Stack
(see header for package body for details)
--begin package Unbounded_Stack

begin
begin
-- set up available space list if one is desired
if Initial_Available_Space_Size > 0 then
Add Nodes to Available Space List:
for I in 1..Initial_Available_Space_Size loop
Available_Bottom.Next := NEW Nodes;
Available_Bottom := Available_Bottom.Next;
end loop Add_Nodes_to_Available_Space_List;

Available_Length := Initial_Available_Space_Size;

end if;

end if;

end Unbounded_Stack;

3.3.7.1.9.8.7 UTILIZATION OF OTHER ELEMENTS

Subprograms and task entries:

The following table describes the subroutines required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNCHECKED DEALLOCATION</td>
<td>generic</td>
<td>N/A</td>
<td>Used to deallocate memory</td>
</tr>
</tbody>
</table>

Exceptions:

The following table describes the exceptions required by this part and defined
in the Ada predefined package STANDARD:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE_ERROR</td>
<td>Raised when an attempt is made to dynamically allocate more memory than is available</td>
</tr>
</tbody>
</table>

**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:

**Packages:**

The following table describes the packages required by this part and specified in the package body of the Abstract_Data_Structures package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available_Space_List_Operations</td>
<td>generic</td>
<td>Contains the routines required to retrieve a node from and place a node in the available space list</td>
</tr>
</tbody>
</table>

**Data types:**

The following data types were previously defined in this part's package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Statuses</td>
<td>discrete</td>
<td>Empty, Available, Uninitialized</td>
<td>Indicates the current status of the stack</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

**Exceptions:**

The following exceptions were previously defined in this part's package specification:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty stack</td>
</tr>
<tr>
<td>Stack Not Initialized</td>
<td>Raised if an attempt is made to use an uninitialized stack</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD STORAGE ERROR</td>
<td>Raised during elaboration of this package if an attempt is made to allocate memory when no more is available</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.8.9 LLCSC DESIGN

None.

### 3.3.7.1.9.8.10 UNIT DESIGN

#### 3.3.7.1.9.8.10.1 INITIALIZE UNIT DESIGN

This procedure initializes a stack by placing a dummy node in the stack, pointing the top and bottom to the dummy node, and setting the length to 0. If this routine is called with a stack containing elements, then the stack is cleared of all but the dummy node.

#### 3.3.7.1.9.8.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

#### 3.3.7.1.9.8.10.1.2 LOCAL ENTITIES DESIGN

None.

#### 3.3.7.1.9.8.10.1.3 INPUT/OUTPUT

**FORMAL PARAMETERS:**

The following table describes this part's formal parameters:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.8.10.1.4 LOCAL DATA
None.

### 3.3.7.1.9.8.10.1.5 PROCESS CONTROL
Not applicable.

### 3.3.7.1.9.8.10.1.6 PROCESSING
The following describes the processing performed by this part:

```plaintext
procedure Initialize (Stack : in out Stacks) is
  -- declaration section
  Current_Length : INTEGER    renames Stack.Current_Length;
  Top             : Pointers  renames Stack.Top;
  Bottom          : Pointers  renames Stack.Bottom;

  -- begin procedure Initialize

  begin
    if Current_Length = -1 then
      -- handle an uninitialized stack
      Top     := New Node;
      Bottom  := Top;
      Current_Length := 0;
    elsif Current_Length > 0 then
      -- handle a stack that has elements in it
      Clear_Stack (Stack => Stack);
    else
      -- current length = 0, so do nothing
      NULL;
    end if;
  end Initialize;
```
3.3.7.1.9.8.10.1.7 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 required by this part and defined in the package specification of Abstract_Data_Structures.Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear_Stack</td>
<td>procedure</td>
<td>Clears a stack by returning all of its nodes to the available space list</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
</tbody>
</table>

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.1.8 LIMITATIONS

The following table describes the exceptions raised by this part:
3.3.7.1.9.8.10.2 CLEAR_STACK UNIT DESIGN

This procedure removes nodes from a stack, leaving only the dummy node. The nodes removed are placed in the available space list.

A Stack_NotInitialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.2.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This_Node</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the node to be placed in the available space list</td>
</tr>
</tbody>
</table>
3.3.7.1.9.8.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.2.6 PROCESSING

The following describes the processing performed by this part:

```pascal
procedure Clear_Stack (Stack : in out Stacks) is

   — declaration section
   Current_Length : INTEGER renames Stack.Current_Length;
   Top            : Pointers renames Stack.Top;
   Bottom         : Pointers renames Stack.Bottom;
   This_Node      : Pointers;

   — begin procedure Clear_Stack

   begin
      — make sure stack has been initialized
      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      end if;

      — make sure there is something in the stack
      elsif Current_Length /= 0 then

         — placed nodes in the available space list
         Save_List (Saved_Head => Top.Next,
                     Saved_Tail  => Bottom,
                     Node_Count  => Current_Length);

         — reinitialize stack variables
         Top.Next      := NULL;
         Bottom       := Top;
         Current_Length := 0;

      end if;

   end Clear_Stack;
```

3.3.7.1.9.8.10.2.7 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 in the package body of Abstract_Data_Structures.Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save_List</td>
<td>procedure</td>
<td>Handles placing a list of nodes in the available space list</td>
</tr>
</tbody>
</table>

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures.Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Not_Initialized</td>
<td>Raised if an attempt is made to use an uninitialized stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Not_Initialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized stack</td>
</tr>
</tbody>
</table>
3.3.7.1.9.8.10.3 FREE_MEMORY UNIT DESIGN

This procedure deallocates the memory occupied by the nodes in the available space list. Only a dummy node will be left in the list.

3.3.7.1.9.8.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.3.3 INPUT/OUTPUT

None.

3.3.7.1.9.8.10.3.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThisNode</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the node to be deallocated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Free_Memory is

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

     This_Node : Pointers;

-- -------------------
-- --begin procedure Free_Memory
-- -------------------

    begin
```
Deallocation of Nodes in Available Space List:
while Available_Top /= Available_Bottom loop

   This_Node := Available_Top;
   Available_Top := Available_Top.Next;
   Free_Node (Which_Node => This_Node);

end loop Deallocation_Nodes_in_Available_Space_List;

Available_Length := 0;
Available_Top.Next := NULL;

end Free_Memory;

3.3.7.1.9.8.10.3.7 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 in the package body of Abstract_Data_ Structures.Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free_Node</td>
<td>procedure</td>
<td>Instantiation of UNCHECKED_DEALLOCATION</td>
</tr>
</tbody>
</table>

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.3.8 LIMITATIONS

None.
3.3.7.1.9.8.10.4 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the top of the stack.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewElement</td>
<td>Elements</td>
<td>in</td>
<td>Element to be added to the stack</td>
</tr>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.4.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>N/A</td>
<td>Points to the new node to be placed in the stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.4.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Add_Element (New_Element : in   Elements;
                       Stack     : in out Stacks) is
```
-- declaration section
--

Current_Length : INTEGER renames Stack.Current_Length;
Top : Pointers renames Stack.Top;

Ptr : Pointers;

-- begin procedure Add_Element
--

begin

if Current_Length = -1 then
    raise Stack_Not_Initialized;
end if;

-- get a node and initialize it
Ptr := New_Node;
Ptr.Data := New_Element;

-- place the node on the stack
Ptr.Next := Top;
Top := Ptr;
Current_Length := Current_Length + 1;

end Add_Element ;

3.3.7.1.9.8.10.4.7 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 in the package body of Abstract_Data_Structures.Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Node</td>
<td>function</td>
<td>Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node</td>
</tr>
</tbody>
</table>

Data types:
The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_NotInitialized</td>
<td>Raised if an attempt is made to use an uninitialized stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD_STORAGE_ERROR</td>
<td>Raised if an attempt is made to allocate more memory than is available</td>
</tr>
<tr>
<td>Stack_NotInitialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.5 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the top element of the stack and returns the data in it to the calling routine. The node is then placed in the available space list.
A Stack Empty exception is raised if this routine is called with an empty stack.

A Stack Not Initialized exception is raised if this routine is called with an uninitializesd stack.

3.3.7.1.9.8.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
<tr>
<td>Old_Element</td>
<td>Elements</td>
<td>out</td>
<td>Elements retrieved from the stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.5.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This_Node</td>
<td>Pointers</td>
<td>N/A</td>
<td>Node to be returned to the available space list</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.5.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Retrieve_Element (Stack        : in out Stacks;
                           Old_Element : out Elements) is
```
3.3.7.1.9.8.10.5.7 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 in the package body of Abstract_Data_Structures.Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save_Node</td>
<td>procedure</td>
<td>Handles placing a node in the available space list</td>
</tr>
</tbody>
</table>

Data types:
The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Empty</td>
<td>Error condition raised if an attempt is made to look at or retrieve elements from an empty stack</td>
</tr>
<tr>
<td>Stack_NotInitialized</td>
<td>Raised if an attempt is made to use an uninitialized stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.5.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_NotInitialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized stack</td>
</tr>
<tr>
<td>Stack_Empty</td>
<td>Raised if an attempt is made to retrieve or look at elements in an empty stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.6 PEEK UNIT DESIGN

This function returns the data contained in the top element of the stack, but does not remove the element from the stack.
A Stack_Empty exception is raised if this routine is called with an empty stack.

A Stack_Not_INITIALIZed exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.6.4 LOCAL DATA

None.

3.3.7.1.9.8.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.6.6 PROCESSING

The following describes the processing performed by this part:

        function Peek (Stack : in Stacks) return Elements is
        begin
            
            Current_Length : INTEGER    renames Stack.Current_Length;
            Top              : Pointers  renames Stack.Top;

            -- --begin function Peek
            
        end function Peek;
begin

-- make sure there is something to peek at
if Current Length = -1 then
    raise Stack_Not_Initialed;
elsif Current Length = 0 then
    raise Stack_Empty;
end if;

-- returned desired element
return Top.Data;
end Peek;

3.3.7.1.9.8.10.6.7 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:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>private</td>
<td>User defined type of data contained in the stack</td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>Contains a single element and a pointer to another node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:
### 3.3.7.1.9.8.10.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Not_Initiated</td>
<td>Raised if an attempt is made to manipulate an uninitialized stack</td>
</tr>
<tr>
<td>Stack_Empty</td>
<td>Raised if an attempt is made to retrieve or look at elements in an empty stack</td>
</tr>
</tbody>
</table>

### 3.3.7.1.9.8.10.7 STACK_STATUS UNIT DESIGN

This function returns the status of the stack according to the following algorithm:

if stack has never been initialized then stack status is uninitialized
elsif stack has no elements in it then stack status in empty
else stack status is available
end if

### 3.3.7.1.9.8.10.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

### 3.3.7.1.9.8.10.7.2 LOCAL ENTITIES DESIGN

None.

### 3.3.7.1.9.8.10.7.3 INPUT/OUTPUT

#### FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
</tbody>
</table>
3.3.7.1.9.8.10.7.4 LOCAL DATA

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Stack_Statuses</td>
<td>N/A</td>
<td>Status of the stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.7.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
function Stack_Status (Stack : in Stacks) return Stack_Statuses is

-- declaration section

Current_Length : INTEGER    renames Stack.Current_Length;
Status        : Stack_Statuses;

-- begin function Stack_Status

begin
    if Current_Length = -1 then
        Status := Uninitialized;
    elsif Current_Length = 0 then
        Status := Empty;
    else
        Status := Available;
    end if;
    return Status;
end Stack_Status;
```
3.3.7.1.9.8.10.7.7 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:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_</td>
<td>discrete</td>
<td>Empty,</td>
<td>Indicates the current status of the stack</td>
</tr>
<tr>
<td>Statuses</td>
<td>type</td>
<td>Uninitialized</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.7.8 LIMITATIONS

None.

3.3.7.1.9.8.10.8 STACK LENGTH UNIT DESIGN

This function returns the length of the stack.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.8.2 LOCAL ENTITIES DESIGN

None.
3.3.7.1.9.8.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stacks</td>
<td>in out</td>
<td>Stack being manipulated</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.4 LOCAL DATA

None.

3.3.7.1.9.8.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Stack_Length (Stack : in Stacks) return NATURAL is
    ——declaration section
    ——begin function Stack_Length
    begin
        ——make sure stack has been initialized
        if Current_Length = -1 then
            raise Stack_Not_Initialized;
        end if;

        return Current_Length;
    end Stack_Length;
```

3.3.7.1.9.8.7 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:

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>List of data along with relevant information</td>
</tr>
</tbody>
</table>

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Not</td>
<td>Raised if an attempt is made to use an uninitialized stack</td>
</tr>
<tr>
<td>Initialized</td>
<td></td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_Not_initialized</td>
<td>Raised if an attempt is made to manipulate an uninitialized stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.9 DOT_NEXT UNIT DESIGN

Given an input pointer P, this function returns the value of P.Next.

3.3.7.1.9.8.10.9.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.8.10.9.2 LOCAL ENTRIES DESIGN

None.
3.3.7.1.9.8.10.9.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node whose &quot;next&quot; entry is to be returned</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.9.4 LOCAL DATA

None.

3.3.7.1.9.8.10.9.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.9.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Dot_Next (Ptr : in Pointers) return Pointers is
begin
    return Ptr.Next;
end Dot_Next;
```

3.3.7.1.9.8.10.9.7 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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part’s package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the stack; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the stack</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the stack</td>
</tr>
</tbody>
</table>
3.3.7.1.9.8.10.9.8 LIMITATIONS

None.

3.3.7.1.9.8.10.10 SET_NEXT UNIT DESIGN

Given an two input pointers, P and Q, this procedure sets P.Next equal to Q.

3.3.7.1.9.8.10.10.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.8.10.10.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.10.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr</td>
<td>Pointers</td>
<td>in</td>
<td>Pointer to the node whose &quot;next&quot; entry is to be modified</td>
</tr>
<tr>
<td>Ptr_dot_Next</td>
<td>Pointers</td>
<td>in</td>
<td>Value to which Ptr.Next is to be set</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.10.4 LOCAL DATA

None.

3.3.7.1.9.8.10.10.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.10.6 PROCESSING

The following describes the processing performed by this part:

```pascal
procedure Set_Next (Ptr        : in Pointers;
                    Ptr_dot_Next : in Pointers) is
begin
    Ptr.Next := Ptr_dot_Next;
end Set_Next;
```
3.3.7.1.9.8.10.10.7 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:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part’s package specification:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>record</td>
<td>N/A</td>
<td>A single entity in the stack; contains data and a pointer to the next node</td>
</tr>
<tr>
<td>Pointers</td>
<td>access</td>
<td>N/A</td>
<td>Points to a node in the stack</td>
</tr>
<tr>
<td>Stacks</td>
<td>record</td>
<td>N/A</td>
<td>Record containing the value of the current length, head, and tail of the stack</td>
</tr>
</tbody>
</table>

3.3.7.1.9.8.10.8 LIMITATIONS

None.

3.3.7.1.10 UNIT DESIGN

None.
package body Abstract_Data_Structures is

pragma PAGE;
generic
    type Nodes is limited private;
    type Pointers is access Nodes;
    Available_Length : in out INTEGER;
    Available_Head   : in out Pointers;
    Available_Tail   : in out Pointers;
    with function Dot_Next (Ptr : in Pointers) return Pointers is <>;
    with procedure Set_Next (Ptr : in Pointers; Ptr_Dot_Next : in Pointers) is <>;

package Available_Space_List_Operations is

    function New_Node return Pointers;
    procedure Save_Node (Saved_Node : in Pointers);
    procedure Save_List (Saved_Head : in Pointers;
                         Saved_Tail : in Pointers;
                         Node_Count : in POSITIVE);

end Available_Space_List_Operations;

pragma PAGE;

-- -----------------------------
-- -- separate package bodies
-- -----------------------------

package body Bounded_Fifo_Buffer is separate;

package body Unbounded_Fifo_Buffer is separate;

package body Nonblocking_Circular_Buffer is separate;

package body Unbounded_Priority_Queue is separate;

package body Bounded_Stack is separate;

package body Unbounded_Stack is separate;

package body Available_Space_List_Operations is separate;

end Abstract_Data_Structures;
package body Available_Space_List_Operations is
pragma PAGE;

function New_Node return Pointers is
begin
  Junction New Node
  if Available_Length > 0 then
    — get the node from the available space list and mark the node
    — that will now be the head of the available space list
    Ptr := Available_Head;
    New_Available_Head := Dot_Next(Available_Head);

    — initialize node being returned
    Set_Next (Ptr => Ptr,
      Ptr_Dot_Next => null);

    — adjust the available space list
    Available_Head := New_Available_Head;
    Available_Length := Available_Length - 1;
  else
    — allocate space to get the node
    Ptr := new Nodes;
  end if;

  return Ptr;
end New_Node;

pragma PAGE;
procedure Save_Node(Saved_Node : in Pointers) is
begin
  Set_Next (Ptr => Available_Tail,
    Ptr_Dot_Next => Saved_Node);

  Available_Tail := Saved_Node;

  Set_Next (Ptr            => Available_Tail,
    Ptr_Dot_Next => null);
Available_Length := Available_Length + 1;
end Save_Node;

pragma PAGE;
procedure Save_List (Saved_Head : in Pointers;
Saved_Tail : in Pointers;
Node_Count : in POSITIVE) is
begin
Set_Next (Ptr => Available_Tail,
Ptr Dot Next => Saved_Head);
Available_Tail := Saved_Tail;
Set_Next (Ptr => Available_Tail,
Ptr Dot Next => Saved_Head);
Available_Length := Available_Length + Node_Count;
end Save_List;
end Available_Space_List_Operations;
separate (Abstract_Data_Structures)
package body Bounded_Fifo_Buffer is
pragma PAGE;

procedure Clear_Buffer (Buffer : out Buffers) is
--- declaration section
---
begin
Buffer_Length := 0;
Head := 0;
Tail := 1;
end Clear_Buffer;

pragma PAGE;
procedure Add_Element (New Element : in Elements;
                      Buffer : in out Buffers) is
--- declaration section
---
begin
--- make sure buffer isn't full
if Head = Tail then
  raise Buffer_Full;
end if;
LIST(Tail) := New Element;
Buffer_Length := Buffer_Size + 1;
if Tail = Buffer_Size then
  Tail := 0;
else
  Tail := Tail + 1;
end if;
end Add_Element;

pragma PAGE;
procedure Retrieve_Element (Buffer : in out Buffers;
   Old_Eleme: out Elements) is

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

   Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
   Head           : Buffer_Range renames Buffer.Head;
   LIST           : Lists renames Buffer_.LIST;
   Tail           : Buffer_Range renames Buffer.Tail;

   -- --------------------------
   -- -- begin procedure Retrieve_Element
   -- --------------------------

   begin

   -- -- make sure don't have an empty buffer
   if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
     raise Buffer_Empty;
   end if;

   if Head = Buffer_Size then
     Head := 0;
   else
     Head := Head + 1;
   end if;
   Old_Eleme := LIST(Head);
   Buffer_Length := Buffer_Length - 1;

   end Retrieve_Element;

pragma PAGE;
function Peek (Buffer : in Buffers) return Elements is

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

   Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
   Head           : Buffer_Range renames Buffer.Head;
   Tail           : Buffer_Range renames Buffer.Tail;
   LIST           : Lists renames Buffer_.LIST;
   Spot           : Buffer_Range;

   -- --------------------------
   -- -- begin function Peek
   -- --------------------------

   begin

   -- -- make sure don't have an empty buffer
if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
    raise Buffer_Empty;
end if;

if Head = Buffer_Size then
    Spot := 0;
else
    Spot := Head + 1;
end if;

return LIST(Spot);
end Peek;

pragma PAGE;
function Buffer_Status (Buffer : in Buffers) return Buffer_statuses is
    — declaration section
    — —
    Head : Buffer_Range renames Buffer.Head;
    Tail : Buffer_Range renames Buffer.Tail;
    Status : Buffer_Statuses;
    — —
    — —begin function Buffer_Status
    — —
    begin
    if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
        Status := Empty;
    elsif Head = Tail then
        Status := Full;
    else
        Status := Available;
    end if;
    return Status;
end Buffer_Status;

pragma PAGE;
function Buffer_Length (Buffer : in Buffers) return Buffer_Range is
    begin
    return Buffer.Buffer_Length;
end Buffer_Length;
end Bounded_Fifo_Buffer;
package body Unbounded_FIFO_Buffer is

-- declaration section

-- this variable is accessed ONLY when setting up the available space list.

Initial_Head       : Pointers := new Nodes;

Available_Space : Buffers := (Current_Length => 0,
Head        => Initial_Head,
Tail        => Initial_Head);

Available_Length : INTEGER renames Available_Space.Current_Length;
Available_Head  : Pointers renames Available_Space.Head;
Available_Tail  : Pointers renames Available_Space.Tail;

procedure Free is new Unchecked_Deallocation
(Object => Nodes,
NAME  => Pointers);

procedure Free_Node (Which_Node : in out Pointers)
renames Free;

function Dot_Next (Ptr : in Pointers) return Pointers;

procedure Set_Next (Ptr       : in Pointers;
Ptr_Dot_Next : in Pointers);

package Available_Space_Operations is new
Available_Space_List_Operations
(Nodes       => Nodes,
Pointers     => Pointers,
Available_Length => Available_Length,
Available_Head    => Available_Head,
Available_Tail    => Available_Tail);

function New_Node return Pointers
renames Available_Space_Operations.New_Node;

procedure Save_Node (Saved_Node : in Pointers)
renames Available_Space_Operations.Save_Node;

procedure Save_List (Saved_Head : in Pointers;
Saved_Tail : in Pointers;
Node_Count : in POSITIVE)
renames Available_Space_Operations.Save_List;

pragma PAGE;

procedure Initialize_Buffer (Buffer : in out Buffers) is

-- declaration section


```vhdl
Current_Length : INTEGER  renames Buffer.Current_Length;
Head            : Pointers renames Buffer.Head;
Tail            : Pointers renames Buffer.Tail;

-- --begin procedure Initialize_Buffer
begin
  if Current_Length = -1 then
    -- handle an uninitialized buffer
    Head  := NewNode;
    Tail  := Head;
    Current_Length := 0;
  elsif Current_Length > 0 then
    -- handle a buffer that has something in it
    Clear_Buffer(Buffer => Buffer);
  else
    -- current length = 0 so it is already initialized
    null;
  end if;
end Initialize_Buffer;
```

```
pragma PAGE;
procedure Clear_Buffer (Buffer : in out Buffers) is

-- declaration section
--
Current_Length : INTEGER  renames Buffer.Current_Length;
Head            : Pointers renames Buffer.Head;
Tail            : Pointers renames Buffer.Tail;

This_Node       : Pointers;

-- --begin procedure Clear_Buffer
begin
  -- make sure this is an initialized buffer
  if Current_Length = -1 then
    raise Buffer_Not_Initiaized;
  end if;

  -- placed nodes in the available space list
  Save_List (Saved_Head => Head.Next,
```
Saved Tail => Tail,
Node_Count => Current_Length);

-- reinitialize buffer variables
Current_Length := 0;
Head.Next := null;
Tail := Head;

end Clear_Buffer;

pragma PAGE;
procedure Free_Memory is

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

Node_To_Be_Freed : Pointers;

-- ------------------------------
-- -- begin procedure Free_Memory
-- ------------------------------

begin

Clear_Out_Available_Space_List:
while Available_Head /= Available_Tail loop

Node_To_Be_Freed := Available_Head;
Available_Head := Available_Head.Next;

Free_Node (Which_Node => Node_To_Be_Freed);

end loop Clear_Out_Available_Space_List;

Available_Length := 0;

end Free_Memory;

pragma PAGE;
procedure Add_Element (New Element : in Elements;
Buffer : in out Buffers) is

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

Current_Length : INTEGER renames Buffer.Current_Length;
Tail : Pointers renames Buffer.Tail;

New_Tail : Pointers;

-- ------------------------------
-- -- begin procedure Add_Element
-- ------------------------------

begin
-- -- make sure buffer has been initialized
if Current_Length = -1 then
  raise Buffer_NotInitialized;
end if;

-- -- now get a node
New_Tail := New_Node;

-- -- now adjust the buffer
Tail.Next := New_Tail;
Tail := New_Tail;
Tail.Data := New_Element;
Current_Length := Current_Length + 1;

end Add_Element;

pragma PAGE;

procedure Retrieve_Element (Buffer: in out Buffers;
Old_Element: out Elements) is

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

Current_Length : INTEGER renames Buffer.Current_Length;
Head : Pointers renames Buffer.Head;
This_Node : Pointers;

-- ---------------
-- -- begin procedure Retrieve_Element
-- ---------------

begin

-- -- make sure an element is available
if Current_Length = -1 then
  raise Buffer_NotInitialized;
elsif Current_Length = 0 then
  raise Buffer_Empty;
end if;

-- -- save dummy node in the available space list
This_Node := Head;
Head := Head.Next;
Save_Node (Saved_Node => This_Node);

-- -- retrieve element (its node becomes the new dummy node)
Old_Element := Head.Data;

-- -- update buffer status
Current_Length := Current_Length - 1;

end Retrieve_Element;

pragma PAGE;
function Peek (Buffer : in Buffers) return Elements is

--- declaration section
---

Current_Length : INTEGER renames Buffer.Current_Length;
Head : Pointers renames Buffer.Head;

--- begin function Peek
---

begin

--- make sure something is there to look at
if Current_Length = -1 then
  raise Buffer_Not_InitIALIZED;
elsif Current_Length = 0 then
  raise Buffer_Empty;
end if;

return Head.Next.Data;

end Peek;

pragma PAGE;

function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

--- declaration section
---

Current_Length : INTEGER renames Buffer.Current_Length;
Status : Buffer_Statuses;

--- begin function Buffer_Status
---

begin

if Current_Length = -1 then
  Status := Uninitialized;
elsif Current_Length = 0 then
  Status := Empty;
else
  Status := Available;
end if;

return Status;

end Buffer_Status;
pragma PAGE;

function Buffer_Length (Buffer : in Buffers) return NATURAL is

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

Current_Length : INTEGER renames Buffer.Current_Length;

-- ------------------------------
-- --begin function Buffer_Length
-- ------------------------------

begin

-- --make sure the buffer has a length
if Current_Length = -1 then
  raise Buffer_Not_Initilaized;
end if;

return Current_Length;

end Buffer_Length;

pragma PAGE;

function Dot_Next (Ptr : in Pointers) return Pointers is

begin
  return Ptr.Next;

end Dot_Next;

pragma PAGE;

procedure Set_Next (Ptr       : in Pointers;
                    Ptr_Dot_Next : in Pointers) is

begin
  Ptr.Next := Ptr_Dot_Next;

end Set_Next;

pragma PAGE;

begin

-- set up available space list if one is desired
if Initial_Available_Space_Size > 0 then

  Add Nodes To Available_Space_List:
  for i in 1..Initial_Available_Space_Size loop
    Available_Tail.Next := new Nodes;
    Available_Tail := Available_Tail.Next;
  end loop Add_NODES_To_Available_Space_List;

  Available_Length := Initial_Available_Space_Size;

end if;
end Unbounded_Fifo_Buffer;
package body Nonblocking_Circular_Buffer is

pragma PAGE;
procedure Clear_Buffer (Buffer : out Buffers) is

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

Head : Buffer_Range renames Buffer.Head;
Tail : Buffer_Range renames Buffer.Tail;
Current_Length : Buffer_Range renames Buffer.Current_Length;

-- --------------------------
-- -- begin procedure Clear_Buffer
-- --------------------------

begin

Head := 0;
Tail := 1;
Current_Length := 0;

end Clear_Buffer;

pragma PAGE;
procedure Add_Element (New Element : in Elements;
                      Buffer    : in out Buffers) is

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

Head : Buffer_Range renames Buffer.Head;
Tail : Buffer_Range renames Buffer.Tail;
Current_Length : Buffer_Range renames Buffer.Current_Length;
LIST : Lists renames Buffer.LIST;

-- --------------------------
-- -- begin procedure Add_Element
-- --------------------------

begin

LIST(Tail) := New_Element;

if Head = Tail then

-- -- buffer was already full and an element was overwritten; therefore,
-- -- both head and tail need to be advanced, but Current_Length does
-- -- not need to be changed

if Tail = Buffer_Size then
  Head := 0;
  Tail := 0;
else
```plaintext
Head := Head + 1;
Tail := Tail + 1;
end if;

else

-- buffer was not already full; therefore, the Current Length needs
-- to be incremented and only the tail needs to be advanced

if Tail = Buffer_Size then
   Tail := 0;
else
   Tail := Tail + 1;
end if;

Current_Length := Current_Length + 1;
end if;

end Add_Element;

pragma PAGE;

procedure Retrieve_Element (Buffer    : in out Buffers;
                           Old_Element :   out Elements) is

   -- declaration section

   Head        : Buffer_Range renames Buffer.Head;
   Tail        : Buffer_Range renames Buffer.Tail;
   Current_Length : Buffer_Range renames Buffer.Current_Length;
   LIST        : Lists    renames Buffer.LIST;

-- begin procedure Retrieve_Element

begin

   -- make sure there is something there to retrieve
   if Current_Length = 0 then
      raise Buffer_Empty;
   end if;

   -- advance the head to get to the next element to go out
   if Head = Buffer_Size then
      Head := 0;
   else
      Head := Head + 1;
   end if;

   -- now retrieve the element and update the state of the buffer
   Old_Element := LIST(Head);
   Current_Length := Current_Length - 1;

end Retrieve_Element;
```
pragma PAGE;

function Peek (Buffer : in Buffers) return Elements is

-- declaration section
--

Head : Buffer_Range renames Buffer.Head;
Current_Length : Buffer_Range renames Buffer.Current_Length;
LIST : Lists renames Buffer.LIST;
Spot : Buffer_Range;

-- begin function Peek
--

begin
-- make sure there is something to peek at
if Current_Length = 0 then
  raise Buffer_Empty;
end if;
-- determine location of desired element
if Head = Buffer_Size then
  Spot := 0;
else
  Spot := Head + 1;
end if;
-- return requested element
return LIST(Spot);
end Peek;

pragma PAGE;

function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

-- declaration section
--

Current_Length : Buffer_Range renames Buffer.Current_Length;
Status : Buffer_Statuses;

-- begin function Buffer_Status
--

begin
if Current_Length = 0 then
  Status := Empty;
elsif Current_Length = Buffer_Size then
Status := Full;
else
    Status := Available;
end if;

return Status;

end Buffer_Status;

pragma PAGE;
function Buffer_Length (Buffer : in Buffers) return Buffer_Range is
begin

    return Buffer.Current_Length;

end Buffer_Length;

end Nonblocking_Circular_Buffer;
with Unchecked_Deallocation;
separate (Abstract_Data_Structures)
package body Unbounded_Priority_Queue is

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

-- this pointers is accessed ONLY when setting up the Available_Space
Initial_Head : Pointers := new Nodes;

Available_Space : Queues := (Current_Length \to 0,
Head \to Initial_Head,
Tail \to Initial_Head);

Available_Length : INTEGER renames Available_Space.Current_Length;
Available_Head : Pointers renames Available_Space.Head;
Available_Tail : Pointers renames Available_Space.Tail;

procedure Free is new Unchecked_Deallocation
  (Object \to Nodes,
   NAME \to Pointers);

procedure Free_Node (Which_Node : in out Pointers)
  renames Free;

function Dot_Next (Ptr : in Pointers) return Pointers;

procedure Set_Next (Ptr : in Pointers;
  Ptr_Dot_Next : in Pointers);

package Available_Space_Operations is new
  Available_Space_List_Operations
  (Nodes \to Nodes,
   Pointers \to Pointers,
   Available_Length \to Available_Length,
   Available_Head \to Available_Head,
   Available_Tail \to Available_Tail);

function New_Node return Pointers
  renames Available_Space_Operations.New_Node;

procedure Save_Node (Saved_Node : in Pointers)
  renames Available_Space_Operations.Save_Node;

procedure Save_List (Saved_Head : in Pointers;
  Saved_Tail : in Pointers;
  Node_Count : in POSITIVE)
  renames Available_Space_Operations.Save_List;

pragma PAGE;
procedure Initialize (Queue : in out Queues) is

-- ---------------------
-- -- declaration section
-- ---------------------
Current_Length : INTEGER renames Queue.Current_Length;
Head : Pointers renames Queue.Head;
Tail : Pointers renames Queue.Tail;

-- begin procedure Initialize
--

begin

if Current_Length = -1 then
  -- handle an uninitialized queue
  Head := New Node;
  Tail := Head;
  Current_Length := 0;

elsif Current_Length > 0 then
  -- handle a queue that has something in it
  Clear_Queue(Queue => Queue);

else
  -- current length = 0 so it is already initialized
  null;
end if;

end Initialize;

pragma PAGE;
procedure Clear_Queue (Queue : in out Queues) is

-- declaration section
begin

-- begin procedure Clear_Queue

begin

-- make sure this is an initialized queue
if Current_Length = -1 then
  raise Queue_NotInitialized;

elsif Current_Length > 0 then

-- placed nodes in the available space list
Save_List (Savedjdead => Head.Next,
SavedTail => Tail,
Node_Count => Current_Length);

-- reinitialize queue variables
Current_Length := 0;
Head.Next := null;
Tail := Head;

end if;
end Clear Queue;

pragma PAGE;
procedure Free_Memory is

-- declaration section
begin procedure Free Memory

begin
Clear Out Available Space List:
while Available Head /= Available Tail loop
Node To Be Freed := Available Head;
Available Head := Available Head.Next;
Free_Node (Which Node => Node To Be Freed);
end loop Clear Out Available Space List;
Available_Length := 0;
end Free_Memory;

pragma PAGE;
procedure Add_Element (New_Element : in Elements;
New_Priority : in Priorities;
Queue : in out Queues) is

-- declaration section
begin procedure Add Element

Current_Length : INTEGER renames Queue.Current_Length;
Head : Pointers renames Queue.Head;
Tail : Pointers renames Queue.Tail;
Before : Pointers;
Here : Pointers;

--
-- --begin procedure Add Element
-- --------------------------------

begin

-- --make sure queue has been initialized
if Current_Length = -1 then
   raise Queue_Not_Initialized;
end if;

-- --find the nodes which are to go before and after the new element
Before := Head;
loop
   exit when (Before = Tail) or else
      (New_Priority > Before.Next.PRIORITY);
   Before := Before.Next;
end loop;

-- --now get a new node
Here := New_Node;

-- --set up the new node
Here.PRIORITY := New_Priority;
Here.Data := New_Element;
Here.Next := Before.Next;
Before.Next := Here;

-- --readjust the tail, if required
if Before = Tail then
   Tail := Here;
end if;

-- --now adjust the queue
Current_Length := Current_Length + 1;

end AddElement;

pragma PAGE;
procedure Retrieve_Element (Queue : in out Queues;
   Old_Element : out Elements) is

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

Current_Length : INTEGER renames Queue.Current_Length;
Head : Pointers renames Queue.Head;
This_Node : Pointers;

-- --begin procedure Retrieve Element
-- --------------------------------

begin

-- --make sure an element is available
if Current_Length = -1 then
    raise Queue_Not_Initialized;
elsif Current_Length = 0 then
    raise Queue_Empty;
end if;

-- -- save dummy node in the available space list
This_Node := Head;
Head.Next := Head.Next;
Save_Node (Saved_Node => This_Node);

-- -- retrieve element (its node becomes the new dummy node)
Old_Element := Head.Data;

-- -- update queue status
Current_Length := Current_Length - 1;

end Retrieve_Element;

pragma PAGE;

function Peek (Queue : in Queues) return Elements is

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

Current_Length : INTEGER renames Queue.Current_Length;
Head : Pointers renames Queue.Head;

-- ---------------
-- begin function Peek
-- ---------------

begin

-- -- make sure something is there to look at
if Current_Length = -1 then
    raise Queue_Not_Initialized;
elsif Current_Length = 0 then
    raise Queue_Empty;
end if;

return Head.Next.Data;

end Peek;

pragma PAGE;

function Queue_Status (Queue : in Queues) return Queue_Statuses is

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

Current_Length : INTEGER renames Queue.Current_Length;
Status : Queue_Statuses;

-- ---------------
begin function Queue_Status
begin
    if Current_Length = -1 then
        Status := Uninitialized;
    elsif Current_Length = 0 then
        Status := Empty;
    else
        Status := Available;
    end if;
    return Status;
end Queue_Status;
pragma PAGE;

function Queue_Length (Queue : in Queues) return NATURAL is
begin
    if Current_Length = -1 then
        raise Queue_Not_Initiaized;
    end if;
    return Current_Length;
end Queue_Length;
pragma PAGE;

function Dot_Next (Ptr : in Pointers) return Pointers is
begin
    return Ptr.Next;
end Dot_Next;
pragma PAGE;

procedure Set_Next (Ptr : in Pointers; Ptr_Dot_Next : in Pointers) is
begin
    Ptr.Next := Ptr_Dot_Next;
end Set_Next;
pragma PAGE;

begin package Unbounded_Priority_Queue
begin

-- set up available space list if one is desired
if Initial_Available_Space_Size > 0 then

    Add Nodes To Available_Space_List:
        For I In 1..Initial_Available_Space_Size loop
            Available_Tail.Next := new Nodes;
            Available_Tail := Available_Tail.Next;
        end loop Add_Nodes_To_Available_Space_List;

    Available_Length := Initial_Available_Space_Size;

end if;

end Unbounded_Priority_Queue;
separate (Abstract_Data_Structures)
package body Bounded_Stack is

pragma PAGE;
procedure Clear_Stack (Stack : out Stacks) is
  begin
    Stack.Top := 0;
  end Clear_Stack;

pragma PAGE;
procedure Add_Element (New_Element : in Elements;
  Stack : in out Stacks) is

  -- declaration section
  LIST : Lists renames Stack.LIST;
  Top : Stack_Length_Range renames Stack.Top;

  -- begin procedure Add_Element

  begin
    -- make sure the stack is not already full
    if Top = Stack_Size then
      raise Stack_Full;
    end if;

    -- add element to the stack
    Top := Top + 1;
    LIST(Top) := New_Element;

  end Add_Element;

pragma PAGE;
procedure Retrieve_Element (Stack : in out Stacks;
  Old_Element : out Elements) is

  -- declaration section
  LIST : Lists renames Stack.LIST;
  Top : Stack_Length_Range renames Stack.Top;

  -- begin procedure Retrieve_Element

  begin
-- make sure there is something in the stack to retrieve
if Top = 0 then
    raise Stack_Empty;
end if;

-- retrieve and remove the top element from the stack
Old_Element := LIST(Top);
Top := Top - 1;

end Retrieve_Element;

pragma PAGE;
function Peek (Stack : in Stacks) return Elements is

-- declaration section
LIST : Lists renames Stack.LIST;
Top : Stack_Length_Range renames Stack.Top;

-- begin function Peek
begin

-- make sure there is something in the stack
if Top = 0 then
    raise Stack_Empty;
end if;

-- return value in top element of the stack
return LIST(Top);

end Peek;

pragma PAGE;
function Stack_Status (Stack : in Stacks) return Stack_Statuses is

-- declaration section
Top : Stack_Length_Range renames Stack.Top;

-- begin function Stack Status
begin

if Top = 0 then
    Status := Empty;
elseif Top = Stack_Size then
Status := Full;
else
  Status := Available;
end if;

return Status;
end Stack_Status;

pragma PAGE;
function Stack_Length (Stack : in Stacks) return Stack_Length_Range is
begin
  return Stack.Top;
end Stack_Length;
end Bounded_Stack;
with Unchecked_Deallocation;
separate (Abstract Data Structures)
package body Unbounded_Stack is

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

-- --this pointer is accessed ONLY when setting up the Available_Space
Initial_Head : Pointers := new Nodes;

Available_Space : Stacks := (Current_Length => 0,
  Top       => Initial_Head,
  Bottom    => Initial_Head);

Available_Length : INTEGER renames Available_Space.Current_Length;
Available_Top    : Pointers renames Available_Space.Top;
Available_Bottom : Pointers renames Available_Space.Bottom;

procedure Free is new Unchecked_Deallocation
  (Object  => Nodes,
   NAME   => Pointers);

procedure Free_Node (Which_Node : in out Pointers)
  renames Free;

function Dot_Next (Ptr : in Pointers) return Pointers;

procedure Set_Next (Ptr : in Pointers; Ptr_Dot_Next : in Pointers);

package Available_Space_Operations is new
  Available_Space_List_Operations
  (Nodes     => Nodes,
   Pointers  => Pointers,
   Available_Length => Available_Length,
   Available_Head   => Available_Top,
   Available_Tail   => Available_Bottom);

function New_Node return Pointers
  renames Available_Space_Operations.New_Node;

procedure Save_Node (Saved_Node : in Pointers)
  renames Available_Space_Operations.Save_Node;

procedure Save_List (Saved_Head : in Pointers;
  Saved_Tail : in Pointers;
  Node_Count : in POSITIVE)
  renames Available_Space_Operations.Save_List;

pragma PAGE;
procedure Initialize (Stack : in out Stacks) is

-- ---------------
-- --declaration section
-- ---------------
Current_Length : INTEGER renames Stack.Current_Length;
Top : Pointers renames Stack.Top;
Bottom : Pointers renames Stack.Bottom;

-- -----------------------------
-- --begin procedure Initialize
-- -----------------------------

begin
  if Current_Length = -1 then
    -- handle an uninitialized stack
    Top := New_Node;
    Bottom := Top;
    Current_Length := 0;
  elsif Current_Length > 0 then
    -- handle a stack that has elements in it
    Clear_Stack (Stack => Stack);
  else
    -- current length = 0, so do nothing
    null;
  end if;
end Initialize;

pragma PAGE;
procedure Clear_Stack (Stack : in out Stacks) is

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

Current_Length : INTEGER renames Stack.Current_Length;
Top : Pointers renames Stack.Top;
Bottom : Pointers renames Stack.Bottom;
This_Node : Pointers;

-- -----------------------------
-- --begin procedure Clear_Stack
-- -----------------------------

begin
  -- make sure stack has been initialized
  if Current_Length = -1 then
    raise Stack_Not_Initiaized;
  -- make sure there is something in the stack
  elsif Current_Length /= 0 then
-- placed nodes in the available space list
Save_List (Saved_Head => Top.Next,
Saved_Tail => Bottom,
Node_Count => Current_Length);

-- reinitialize stack variables
Top.Next := null;
Bottom := Top;
Current_Length := 0;

end if;

end Clear_Stack;

pragma PAGE;
procedure Free_Memory is
pragma PAGE;

procedure Free_Memory is

begin

Deallocate_Nodes_In_Available_Space_List:
while Available_Top /= Available_Bottom loop

This_Node := Available_Top;
Available_Top := Available_Top.Next;
Free_Node (Which_Node => This_Node);

end loop Deallocate_Nodes_In_Available_Space_List;

Available_Length := 0;
Available_Top.Next := null;

end Free_Memory;

pragma PAGE;
procedure Add_Element (New_Element : in Elements;
Stack : in out Stacks) is

declaration section

Current_Length : INTEGER renames Stack.Current_Length;
Top : Pointers renames Stack.Top;
Ptr : Pointers;
-- -- begin procedure Add_Element

begin

if Current_Length = -1 then
  raise Stack_NotInitialized;
end if;

-- get a node and initialize it
Ptr := NewNode;
Ptr.Data := NewElement;

-- place the node on the stack
Ptr.Next := Top;
Top := Ptr;
Current_Length := Current_Length + 1;
end Add_Element;

pragma PAGE;
procedure Retrieve_Element (Stack : in out Stacks;
  Old Element : out Elements) is

-- declaration section
--

Current_Length : INTEGER renames Stack.Current_Length;
Top : Pointers renames Stack.Top;

This_Node : Pointers;

-- begin procedure Retrieve_Element

begin

-- make sure there is something to retrieve
if Current_Length = -1 then
  raise Stack_NotInitialized;
elsif Current_Length = 0 then
  raise Stack_Empty;
end if;

-- retrieve data in the top node
Old_Element := Top.Data;

-- dispose of top node and adjust the stack
This_Node := Top;
Top := Top.Next;
Save_Node (Saved_Node => This_Node);
Current_Length := Current_Length - 1;
end Retrieve_Element;
pragma PAGE;

function Peek (Stack : in Stacks) return Elements is

logdeneration section
Current_Length : INTEGER renames Stack.Current_Length;
Top            : Pointers renames Stack.Top;

begin

begin

-- make sure there is something to peek at
if Current_Length = -1 then
    raise Stack_Not_Initialized;
elsif Current_Length = 0 then
    raise Stack_Empty;
end if;

-- returned desired element
return Top.Data;

end Peek;

pragma PAGE;

function Stack_Status (Stack : in Stacks) return Stack_Statuses is

logdeneration section
Current_Length : INTEGER renames Stack.Current_Length;

Status : Stack_Statuses;

begin

if Current_Length = -1 then
    Status := Uninitialized;
elseif Current_Length = 0 then
    Status := Empty;
else
    Status := Available;
end if;
return Status;
end Stack_Status;

pragma PAGE;
function Stack_Length (Stack : in Stacks) return NATURAL is

— declaration section
Current_Length : INTEGER renames Stack.Current_Length;

begin

— make sure stack has been initialized
if Current_Length = -1 then
    raise Stack_Not_Initialized;
end if;

return Current_Length;
end Stack_Length;

pragma PAGE;
function Dot_Next (Ptr : in Pointers) return Pointers is

begin
    return Ptr.Next;
end Dot_Next;

pragma PAGE;
procedure Set_Next (Ptr       : in Pointers;
                   Ptr_Dot_Next : in Pointers) is

begin
    Ptr.Next := Ptr_Dot_Next;
end Set_Next;

pragma PAGE;

begin package Unbounded Stack
(see header for package body for details)

begin

— set up available space list if one is desired
if Initial_Available_Space_Size > 0 then

Add Nodes To Available_Space_List:
    for I in 1..Initial_Available_Space_Size loop
        Available_Bottom.Next := new Nodes;
        Available_Bottom := Available_Bottom.Next;
    end loop;

end if;

end Unbounded Stack;

begin

...
Available_Length := Initial_Available_Space_Size;

end if;

end Unbounded_Stack;
3.3.8 GENERAL UTILITIES
THIS REPORT HAS BEEN DELIMITED AND CLEARED FOR PUBLIC RELEASE UNDER DOD DIRECTIVE 5200.20 AND NO RESTRICTIONS ARE IMPOSED UPON ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.
3.3.8.1 GENERAL_UTILITIES TLCSC P361 (CATALOG #P267-0)

This package provides a group of general utility routines used in a missile system.

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

3.3.8.1.1 REQUIREMENTS ALLOCATION

This part meets requirement R141.

3.3.8.1.2 LOCAL ENTITIES DESIGN

None.

3.3.8.1.3 INPUT/OUTPUT

None.

3.3.8.1.4 LOCAL DATA

None.

3.3.8.1.5 PROCESS CONTROL

Not applicable.

3.3.8.1.6 PROCESSING

The following describes the processing performed by this part:

package body General_Utilities is
end General_Utilities;

3.3.8.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.1.8 LIMITATIONS

None.
3.3.8.1.9 LLCSC DESIGN

None.

3.3.8.1.10 UNIT DESIGN

3.3.8.1.10.1 INSTRUCTION_SET_TEST UNIT DESIGN (CATALOG #P268-0)

This part is a generic function which checks for proper processor operation by executing a function and comparing the result to the expected result. If the expected and derived values match, "True" is returned. The part's generic parameter may be any type, but a Test function must be supplied which matches the parameter defined in the specification.

3.3.8.1.10.1.1 REQUIREMENTS ALLOCATION

This part meets requirement R141.

3.3.8.1.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.8.1.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return_Values</td>
<td>private</td>
<td>May be any type. The type which the included function must return.</td>
</tr>
</tbody>
</table>

Subprograms:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>function</td>
<td>the function to be tested, it must return a value of Return_Values type.</td>
</tr>
</tbody>
</table>

FORMAL PARAMETERS:
The following table describes this part’s formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct_Answer</td>
<td>Return_Values</td>
<td>in</td>
<td>The answer which is to be compared to what the function returns.</td>
</tr>
</tbody>
</table>

3.3.8.1.10.1.4 LOCAL DATA

None.

3.3.8.1.10.1.5 PROCESS CONTROL

Not applicable.

3.3.8.1.10.1.6 PROCESSING

The following describes the processing performed by this part:

```haskell
function Instruction_Set_Test( Correct_Answer : Return_Values )
return BOOLEAN is
begin
  return Test = Correct_Answer;
  -- returns true if function and answer are the same
  -- false if they are not
end Instruction_Set_Test;
```

3.3.8.1.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.1.10.1.8 LIMITATIONS

None.
package body General Utilities is

  function Instruction_Set_Test( Correct_Answer : Return_Values )
  return BOOLEAN is

    begin
      return Test = Correct_Answer;
      —— returns true if function and answer are the same
      —— false if they are not
    end Instruction_Set_Test;

end General Utilities;
(This page left intentionally blank.)
3.3.8.2 COMMUNICATION_PARTS TLCSC P602 (CATALOG #P691-0)

This package provides a group of communication routines used in a missile system.

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

3.3.8.2.1 REQUIREMENTS ALLOCATION

This part meets requirement R137.

3.3.8.2.2 LOCAL ENTITIES DESIGN

None.

3.3.8.2.3 INPUT/OUTPUT

None.

3.3.8.2.4 LOCAL DATA

None.

3.3.8.2.5 PROCESS CONTROL

Not applicable.

3.3.8.2.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
package body Communication_Parts is
end Communication_Parts;
```

3.3.8.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.2.8 LIMITATIONS

None.
3.3.8.2.9 LLCSC DESIGN

3.3.8.2.9.1 UPDATE_EXCLUSION PACKAGE DESIGN (CATALOG #P692-0)

This part is a generic package containing a task providing a mechanism for ensuring that data accessed by more than one asynchormous task is properly protected for such accesses. The part's generic parameter can be any type.

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

3.3.8.2.9.1.1 REQUIREMENTS ALLOCATION

This part meets requirement R137.

3.3.8.2.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.8.2.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element_Type</td>
<td>private</td>
<td>Allows any type to be protected</td>
</tr>
</tbody>
</table>

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial_Value</td>
<td>Element_Type</td>
<td>Allows the data type to be initialized so that the first time Start_Update_Request is called a constraint error is not raised by some uninitialized value.</td>
</tr>
</tbody>
</table>

FORMAL PARAMETERS:

The following table describes the formal parameters for the task entries in the task contained in this part.
<table>
<thead>
<tr>
<th>Task</th>
<th>Name</th>
<th>Mode</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_Update</td>
<td>Read_Request</td>
<td>Output</td>
<td>Element_Type</td>
<td>Contains the value of the returned data.</td>
</tr>
<tr>
<td></td>
<td>Start_Update_</td>
<td>Output</td>
<td>Element_Type</td>
<td>Contains the value of the returned data.</td>
</tr>
<tr>
<td>Request</td>
<td>Complete_</td>
<td>Input</td>
<td>Element_Type</td>
<td>Contains the new value of the data to replace the protected data.</td>
</tr>
<tr>
<td></td>
<td>Update_Request</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3.8.2.9.1.4 LOCAL DATA

None.

3.3.8.2.9.1.5 PROCESS CONTROL

Not applicable.

3.3.8.2.9.1.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
definitions for packages and type declarations

package body Update_Exclusion is

    task Read_Update is
        entry Task_Read_Request( Requested_Data : out Element_Type );
        entry Task_Start_Update_Request( Old_Data : out Element_Type );
        entry Task_Complete_Update_Request( New_Data : in Element_Type );
    end Read_Update;

    procedure Attempt_Read( Requested_Data : in out Element_Type;
                            Result       : out Rendezvous_Flags ) is
      begin
        select
          Read_Update.Task_Read_Request( Requested_Data );
        Result := Success;
        else
          Result := Failure;
        end select;
      end Attempt_Read;

    procedure Attempt_Read_Wait( Requested_Data : in out Element_Type;
                                  Result       : out Rendezvous_Flags ) is
      begin
        Read_Update.Task_Read_Request( Requested_Data );
        Result := Success;
      end Attempt_Read_Wait;

    procedure Attempt_Read_Delay( Requested_Data : in out Element_Type;
                                  Result       : out Rendezvous_Flags;
                                  Delay_Time   : in DURATION ) is
```

```plaintext
end Update_Exclusion;
```
begin
    Result := Failure;
    select
        Read_Update.Task_Read_Request(Removed_Data);
        Result := Success;
    or
        DELAY Delay_Time;
        end select;
end Attempt_Read_Delay;

procedure Attempt_Start_Update( Old_Data : in out Element_Type;
                               New_Id : out Rendezvous_Ids;
                               Result : out Rendezvous_Flags) is
begin
    select
        Read_Update.Task_Start_Update_Request(Old_Data => Old_Data);
        New_Id := Id;
        Result := Success;
    else
        Result := Failure;
        New_Id := 0;
        end select;
end Attempt_Start_Update;

procedure Attempt_Start_Update_Wait( Old_Data : in out Element_Type;
                                     New_Id : out Rendezvous_Ids;
                                     Result : out Rendezvous_Flags) is
begin
    Read_Update.Task_Start_Update_Request(Old_Data => Old_Data);
    New_Id := Id;
    Result := Success;
end Attempt_Start_Update_Wait;

procedure Attempt_Start_Update_Delay( Old_Data : in out Element_Type;
                                     New_Id : out Rendezvous_Ids;
                                     Result : out Rendezvous_Flags;
                                     Time : in DURATION) is
begin
    Result := Failure;
    select
        Read_Update.Task_Start_Update_Request(Old_Data => Old_Data);
        New_Id := Id;
        Result := Success;
    or
        DELAY Time;
        end select;
end Attempt_Start_Update_Delay;

procedure Attempt_Complete_Update( New_Data : in Element_Type;
                                   Passed_Id : in Rendezvous_Ids;
                                   Result : out Rendezvous_Flags) is
begin
    if Passed_Id = Id then
        select
            Read_Update.Task_Complete_Update_Request(New_Data);
            Result := Success;
        else
            Result := Failure;
            New_Id := 0;
            end select;
end Attempt_Complete_Update;
Result := Failure;
end select;
else
Result := Bad_Id;
end if;
end Attempt_Complete_Update;

task body Read_Update is

Protected_Data : Element_Type := Initial_Value;

begin
process_continually:

loop

select

accept Task_Read_Request (Requested_Data : out Element_Type)
do
Requested_Data := Protected_Data;
end Task_Read_Request;
or
accept Task_Start_Update_Request (Old_Data : out Element_Type)
do
Old_Data := Protected_Data;
end Task_Start_Update_Request;

accept Task_Complete_Update_Request (New_Data : in Element_Type)
do
Protected_Data := New_Data;
end Task_Complete_Update_Request;

if Id = Rendezvous_Ids'LAST then
Id := Rendezvous_Ids'FIRST + 1;
else
Id := Rendezvous_Ids'SUCC(Id);
end if;
or
terminate;
end select;
end loop process_continually;
end Read_Update;

end Update_Exclusion;

3.3.8.2.9.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.2.9.1.8 LIMITATIONS

None.
3.3.8.2.9.1.9 LLCSC DESIGN
None.

3.3.8.2.9.1.10 UNIT DESIGN
None.

3.3.8.2.10 UNIT DESIGN
None.
package body Communication_Parts is

package body Update_Exclusion is

task Read_Update is
  entry Task_Read_Request( Requested Data : out Element_Type );
  entry Task_Start_Update_Request( Old_Data : out Element_Type );
  entry Task_Complete_Update_Request( New_Data : in Element_Type );
end Read_Update;

procedure Attempt_Read( Requested_Data : in out Element_Type;
                        Result       : out Rendezvous_Flags ) is
begin
  select
    Read_Update.Task_Read_Request( Requested_Data );
    Result := Success;
  else
    Result := Failure;
  end select;
end Attempt_Read;

procedure Attempt_Read_Wait( Requested_Data : in out Element_Type;
                            Result       : out Rendezvous_Flags ) is
begin
  Read_Update.Task_Read_Request( Requested_Data );
  Result := Success;
end Attempt_Read_Wait;

procedure Attempt_Read_Delay( Requested_Data : in out Element_Type;
                             Result       : out Rendezvous_Flags;
                             Delay_Time   : in DURATION ) is
begin
  Result := Failure;
  select
    Read_Update.Task_Read_Request( Requested_Data );
    Result := Success;
  or
    delay Delay_Time;
  end select;
end Attempt_Read_Delay;

procedure Attempt_Start_Update( Old_Data : in out Element_Type;
                                New_Id    : out Rendezvous_Id;
                                Result     : out Rendezvous_Flags ) is
begin
  select
    Read_Update.Task_Start_Update_Request( Old_Data -> Old_Data );
    New_Id := Id;
    Result := Success;
  else
    Result := Failure;
    New_Id := 0;
  end select;
end Attempt_Start_Update;

procedure Attempt_Start_Update_Wait( Old_Data : in out Element_Type;
                                     New_Id    : out Rendezvous_Id;
                                     Result     : out Rendezvous_Flags );
Result : out Rendezvous_Flags ) is

begin
    Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
    New_Id := Id;
    Result := Success;
end Attempt_Start_Update_Wait;

procedure Attempt_Start_Update_Delay( Old_Data : in out ElementJType;
                                         New_Id  : out Rendezvous_Ids;
                                         Result  : out Rendezvous_Flags;
                                         Time    : in DURATION ) is

begin
    Result := Failure;
    select
        Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
        New_Id := Id;
        Result := Success;
    or
        delay Time;
    end select;
end Attempt_Start_Update_Delay;

procedure Attempt_Complete_Update( New_Data : in Element_Type;
                                          Passed_Id : in Rendezvous_Ids;
                                          Result   : out Rendezvous_Flags ) is

begin
    if Passed_Id = Id then
        select
            Read_Update.Task_Complete_Update_Request( New_Data );
            Result := Success;
        else
            Result := Failure;
        end select;
    else
        Result := Bad_Id;
    end if;
end Attempt_Complete_Update;

task body Read_Update is

    Protected_Data : Element_Type := Initial_Value;

begin
    Process_Continually:
        loop
            select
                accept Task_Read_Request (Requested_Data : out Element_Type)
                do
                    Requested_Data := Protected_Data;
                end Task_Read_Request;
            or
                accept Task_Start_Update_Request (Old_Data : out Element_Type)
                do
                    Old_Data := Protected_Data;
                end Task_Start_Update_Request;
            accept Task_Complete_Update_Request (New_Data : in Element_Type)
do
  Protected_Data := New_Data;
end Task_Complete_Update_Request;

if Id = Rendezvous_Ids'LAST then
  Id := Rendezvous_Ids'FIRST + 1;
else
  Id := Rendezvous_Ids'SUCC( Id );
end if;
or
terminate;
end select;
end loop Process_Continually;
end Read_Update;

end Update_Exclusion;

end Communication_Parts;
3.3.9 EQUIPMENT INTERFACES
3.3.9.1 CLOCK_HANDLER TLCSC P634 (CATALOG #P270-0)

This package contains the routines required to maintain an internal clock.

The following routines are provided to manipulate the clock:
- Reset clock (effectively zeroes out the clock)
- Synchronize clock (effectively sets the clock to the specified time)
- Current time (effectively reads the internal clock)

In addition, a Converted Time routine is provided to convert a CALENDAR. TIME to the "local time zone".

An Elapsed Time routine is provided to act as a stopwatch. It returns the elapsed time between successive calls to the function. This function is not affected by resetting or synchronizing the clock.

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

3.3.9.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R046.

3.3.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

This part is a parameterless generic.

3.3.9.1.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference_Time</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Internal reference clock maintained by this part</td>
</tr>
<tr>
<td>Time_Last_Called</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Last time the Elapsed Time function was called</td>
</tr>
</tbody>
</table>
3.3.9.1.5 PROCESS CONTROL

Not applicable.

3.3.9.1.6 PROCESSING

The following describes the processing performed by this part:

package body Clock_Handler is

    use CALENDAR;

    -- local declarations

    Reference_Time     : CALENDAR.TIME := CALENDAR.CLOCK;
    Time_Last_Called   : CALENDAR.TIME := CALENDAR.CLOCK;

end Clock_Handler;

3.3.9.1.7 UTILIZATION OF OTHER ELEMENTS

The following library units were with'd by the package specification of this part:

1. CALENDAR

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Function</td>
<td>Calendar</td>
<td>Returns the internal system time</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the external types required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>private</td>
<td>CALENDAR</td>
<td>Implementation-dependent representation of time</td>
</tr>
<tr>
<td>DURATION</td>
<td>fixed</td>
<td>STANDARD</td>
<td>Represents a length of time</td>
</tr>
</tbody>
</table>
3.3.9.1.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD.TIME_ERROR</td>
<td>Raised by the following routines if a difference in times does not fit within the range of type STANDARD.DURATION:</td>
</tr>
<tr>
<td></td>
<td>o Current_Time</td>
</tr>
<tr>
<td></td>
<td>o Converted_Time</td>
</tr>
<tr>
<td></td>
<td>o Elapsed_Time</td>
</tr>
<tr>
<td></td>
<td>o Synchronize_Time</td>
</tr>
</tbody>
</table>

3.3.9.1.9 LLCSC DESIGN

None.

3.3.9.1.10 UNIT DESIGN

3.3.9.1.10.1 CURRENT_TIME (FUNCTION BODY) UNIT DESIGN

This function returns the time of the current time of the clock. The current time is the time which has passed since the last time the internal clock was reset or since the time specified when the clock was synchronized.

3.3.9.1.10.1.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.1.3 INPUT/OUTPUT

None.

3.3.9.1.10.1.4 LOCAL DATA

None.

3.3.9.1.10.1.5 PROCESS CONTROL

Not applicable.
3.3.9.1.10.1.6 PROCESSING

The following describes the processing performed by this part:

```plsql
function Current_Time return DURATION is
begin
    return CALENDAR.CLOCK - Reference_Time;
end Current_Time;
```

3.3.9.1.10.1.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with’d and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Function</td>
<td>Calendar</td>
<td>Returns the internal system time</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the external types required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>private</td>
<td>CALENDAR</td>
<td>Implementation-dependent representation of time</td>
</tr>
<tr>
<td>DURATION</td>
<td>fixed</td>
<td>STANDARD</td>
<td>Represents a length of time</td>
</tr>
</tbody>
</table>

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:

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Clock_Handler:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference_Time</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Internal reference clock maintained by this part</td>
</tr>
</tbody>
</table>

3.3.9.1.10.1.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD.</td>
<td>Raised if the elapsed time does not fit within the range of</td>
</tr>
<tr>
<td>TIME_ERROR</td>
<td>type STANDARD.DURATION</td>
</tr>
</tbody>
</table>

3.3.9.1.10.2 CONVERTED_TIME (FUNCTION BODY) UNIT DESIGN

This function converts an input time to a local time (i.e., converts it to the "local time zone"). A local time is defined as the difference between the input time and the internal reference time.

3.3.9.1.10.2.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock_Time</td>
<td>CALENDAR.TIME</td>
<td>In</td>
<td>Time to be converted to a local time</td>
</tr>
</tbody>
</table>

3.3.9.1.10.2.4 LOCAL DATA

None.
3.3.9.1.10.2.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.2.6 PROCESSING

The following describes the processing performed by this part:

function Converted_Time (Clock_Time : in CALENDAR.TIME)
  return DURATION is
begin
  return Clock_Time - Reference_Time;
end Converted_Time;

3.3.9.1.10.2.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:
1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Data types:

The following table summarizes the external types required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>private</td>
<td>CALENDAR</td>
<td>Implementation-dependent representation of time</td>
</tr>
<tr>
<td>DURATION</td>
<td>fixed</td>
<td>STANDARD</td>
<td>Represents a length of time</td>
</tr>
</tbody>
</table>

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:

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference_Time</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Internal reference clock maintained by this part</td>
</tr>
</tbody>
</table>
3.3.9.1.10.2.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD.</td>
<td>Raised if the elapsed time does not fit within the range of</td>
</tr>
<tr>
<td>TIME_ERROR</td>
<td>type STANDARD.DURATION</td>
</tr>
</tbody>
</table>

3.3.9.1.10.3 RESET_CLOCK (PROCEDURE BODY) UNIT DESIGN

This procedure effectively zeroes out the internal clock by setting the internal reference time equal to the system time.

3.3.9.1.10.3.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.3.3 INPUT/OUTPUT

None.

3.3.9.1.10.3.4 LOCAL DATA

None.

3.3.9.1.10.3.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.3.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Reset_Clock is
begin
    Reference_Time := CALENDAR.CLOCK;
end Reset_Clock;
```
3.3.9.1.10.3.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with’d and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Function</td>
<td>Calendar</td>
<td>Returns the internal system time</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the external types required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>private</td>
<td>CALENDAR</td>
<td>Implementation-dependent representation of time</td>
</tr>
<tr>
<td>DURATION</td>
<td>fixed</td>
<td>STANDARD</td>
<td>Represents a length of time</td>
</tr>
</tbody>
</table>

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:

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference_Time</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Internal reference clock maintained by this part</td>
</tr>
</tbody>
</table>

3.3.9.1.10.3.8 LIMITATIONS

None.
3.3.9.1.10.4 SYNCHRONIZE_CLOCK (PROCEDURE BODY) UNIT DESIGN

This procedure effectively sets the internal clock to a user-specified time. It does this by setting the reference time to a system (CALENDAR) time - the desired time. By default, the system time used is CALENDAR.CLOCK by the user may supply his own "system" time.

3.3.9.1.10.4.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New_Time</td>
<td>STANDARD.DURATION</td>
<td>In</td>
<td>Time to which the internal clock should be set</td>
</tr>
<tr>
<td>Clock_Time</td>
<td>CALENDAR.TIME</td>
<td>In</td>
<td>System time</td>
</tr>
</tbody>
</table>

3.3.9.1.10.4.4 LOCAL DATA

None.

3.3.9.1.10.4.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.4.6 PROCESSING

The following describes the processing performed by this part:

```plaintext
procedure Synchronize_Clock
    (New_Time : in STANDARD.DURATION;
     Clock_Time : in CALENDAR.TIME := CALENDAR.CLOCK) is

begin
    Reference_Time := Clock_Time - New_Time;

end Synchronize_Clock;
```
3.3.9.1.10.4.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Data types:

The following table summarizes the external types required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>private</td>
<td>CALENDAR</td>
<td>Implementation-dependent representation of time</td>
</tr>
<tr>
<td>DURATION</td>
<td>fixed</td>
<td>STANDARD</td>
<td>Represents a length of time</td>
</tr>
</tbody>
</table>

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:

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference_Time</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Internal reference clock maintained by this part</td>
</tr>
</tbody>
</table>

3.3.9.1.10.8 LIMITATIONS

None.

3.3.9.1.10.5 ELAPSED_TIME (FUNCTION BODY) UNIT DESIGN

This function returns the time since the last call to this function. The first call to this function will result in the time since the package was elaborated. This function is not affected by calls to Reset_Clock or Synchronize_Clock.

3.3.9.1.10.5.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP RO46.
3.3.9.1.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.5.3 INPUT/OUTPUT

None.

3.3.9.1.10.5.4 LOCAL DATA

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer</td>
<td>STANDARD.DURATION</td>
<td>N/A</td>
<td>Amount of time which has elapsed since the last call to this function</td>
</tr>
<tr>
<td>New_Time</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>System time</td>
</tr>
</tbody>
</table>

3.3.9.1.10.5.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.5.6 PROCESSING

The following describes the processing performed by this part:

```pascal
function Elapsed_Time return STANDARD.DURATION is
  -- declaration section
  Answer : STANDARD.DURATION;
  New_Time : CALENDAR.TIME := CALENDAR.CLOCK;

  -- begin function Elapsed_Time
  begin
    Answer := New_Time - Time_Last_Called;
    Time_Last_Called := New_Time;
    return Answer;
  end Elapsed_Time;
```
3.3.9.1.10.5.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:
   1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Function</td>
<td>Calendar</td>
<td>Returns the internal system time</td>
</tr>
</tbody>
</table>

Data types:

The following table summarizes the external types required by this part:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>private</td>
<td>CALENDAR</td>
<td>Implementation-dependent representation of time</td>
</tr>
<tr>
<td>DURATION</td>
<td>fixed</td>
<td>STANDARD</td>
<td>Represents a length of time</td>
</tr>
</tbody>
</table>

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:

Data objects:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time_Last_Called</td>
<td>CALENDAR.TIME</td>
<td>N/A</td>
<td>Last time the Elapsed Time function was called</td>
</tr>
</tbody>
</table>

3.3.9.1.10.5.8 LIMITATIONS

The following table describes the exceptions propagated by this part:
<table>
<thead>
<tr>
<th>Name</th>
<th>When/Why Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD.</td>
<td>Raised if the elapsed time does not fit within the range of type STANDARD.DURATION</td>
</tr>
<tr>
<td>TIME_ERROR</td>
<td></td>
</tr>
</tbody>
</table>
package body Clock_Handler is

use CALENDAR;

-- local declarations

Reference_Time     : CALENDAR.Time := CALENDAR.Clock;
Time_Last_Called    : CALENDAR.Time := CALENDAR.Clock;

pragma PAGE;
function Current_Time return DURATION is
begin
  return CALENDAR.Clock - Reference_Time;
end Current_Time;

pragma PAGE;
function Converted_Time (Clock_Time : in CALENDAR.Time) return DURATION is
begin
  return Clock_Time - Reference_Time;
end Converted_Time;

pragma PAGE;
procedure Reset_Clock is
begin
  Reference_Time := CALENDAR.Clock;
end Reset_Clock;

pragma PAGE;
procedure Synchronize_Clock
  (New_Time      : in STANDARD.DURATION;
   Clock_Time : in CALENDAR.Time := CALENDAR.Clock) is
begin
  Reference_Time := Clock_Time - New_Time;
end Synchronize_Clock;

pragma PAGE;
function Elapsed_Time return STANDARD.DURATION is

-- declaration section

Answer  : STANDARD.DURATION;
New_Time : CALENDAR.Time := CALENDAR.Clock;
— —
begin function Elapsed_Time
— —
begin
  Answer := New_Time - Time_Last_Called;
  Time_Last_Called := New_Time;
  return Answer;
end Elapsed_Time;
end Clock_Handler;
4  (NOT USED)
5 (NOT USED)
6 NOTES

This paragraph does not apply to this DDD.
ERRATA
AD-8129 2.59
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).

<table>
<thead>
<tr>
<th>Technical Report Number</th>
<th>AD Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 88-18-Vol-4</td>
<td>ADB 120 251</td>
</tr>
<tr>
<td>2. 88-18-Vol-5</td>
<td>ADB 120 252</td>
</tr>
<tr>
<td>3. 88-18-Vol-6</td>
<td>ADB 120 253</td>
</tr>
<tr>
<td>4. 88-25-Vol-1</td>
<td>ADB 120 309</td>
</tr>
<tr>
<td>5. 88-25-Vol-2</td>
<td>ADB 120 310</td>
</tr>
<tr>
<td>6. 88-62-Vol-1</td>
<td>ADB 129 568</td>
</tr>
<tr>
<td>7. 88-62-Vol-2</td>
<td>ADB 129 569</td>
</tr>
<tr>
<td>8. 88-62-Vol-3</td>
<td>ADB 129-570</td>
</tr>
<tr>
<td>9. 85-93-Vol-1</td>
<td>ADB 102-654</td>
</tr>
<tr>
<td>10. 85-93-Vol-2</td>
<td>ADB 102-655</td>
</tr>
<tr>
<td>11. 85-93-Vol-3</td>
<td>ADB 102-656</td>
</tr>
<tr>
<td>12. 88-18-Vol-1</td>
<td>ADB 120 248</td>
</tr>
<tr>
<td>13. 88-18-Vol-2</td>
<td>ADB 120 249</td>
</tr>
<tr>
<td>14. 88-18-Vol-7</td>
<td>ADB 120 254</td>
</tr>
<tr>
<td>15. 88-18-Vol-8</td>
<td>ADB 120 255</td>
</tr>
<tr>
<td>16. 88-18-Vol-9</td>
<td>ADB 120 256</td>
</tr>
<tr>
<td>17. 88-18-Vol-10</td>
<td>ADB 120 257</td>
</tr>
<tr>
<td>18. 88-18-Vol-11</td>
<td>ADB 120 258</td>
</tr>
<tr>
<td>19. 88-18-Vol-12</td>
<td>ADB 120 259</td>
</tr>
</tbody>
</table>

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
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, Lt Col, USAF
Chief of Public Affairs

AFDIC/PA 92-039