REFERENCES

1) Boehm [1981]

2) Boehm [1988]

3) Lehman and Belady [1985]

4) Humphrey [1988]

5) DeMarco and Lister [1987]
   Tom DeMarco and Timothy Lister, "Peopleware", *Dorset House Publishing*, 1987

---

Practical Ada Memory Management Strategies for Real-Time Systems

Jay Johnson
Honeywell
Glendale, Ariz.

Jay Johnson has spent most of his career since receiving his BSCS from BYU in 1983 working on real-time embedded systems. He was a member of the Boeing Ada Design-Build Team while at General Electric in 1986, where he began designing embedded systems using Ada. He now works as a senior project engineer at Honeywell, where he teaches advanced Ada classes and designs embedded Ada software.
If an embedded-systems programmer uses Ada long enough, he or she will undoubtedly need to represent a variable-sized data structure, possibly at an address determined at runtime. Links between such structures may also be necessary, and the programmer may finally break down and attempt to use pointers.

Many embedded-systems programmers who use Ada have very little experience with pointers (access types) in Ada. This is largely because the performance of most real-time embedded systems cannot tolerate the overhead of the memory management afforded by the usually-provided Ada runtime. This system is generally non-deterministic due to garbage collection considerations, and can be wasteful of both memory and clock cycles. However, the LRM clearly states that an access object (pointer) can only be assigned the result of the NEW allocator, the contents of another access object of the same type, or the NULL constant. Therefore, software engineers often give up on the standard Ada runtime and resort to simple compile-time allocated arrays.

There are other options, unknown to many Ada programmers. For the moment, however, consider the array. Even with compile-time allocated arrays, programmers with a modicum of imagination can conceive of one array of records which has indices referring to another array of records. One may also see an advantage in using a parallel array of booleans to indicate when a record is “in use.” Using arrays in this way can simulate a linked list, but is potentially quite wasteful of memory.

At this juncture, one might ask about the value of conserving memory when it can be purchased in cheap bushells of megabytes. True, memory seems to get a bit less expensive every day. However, if one thinks back to the old days (pre 1980), it is obvious that while memory was an order of magnitude more expensive then, embedded systems were an order of magnitude smaller. Today, with embedded systems weighing in at up to millions of lines of source code, there seems to be an unlimited demand for memory in a box, but the fact remains than only a finite amount of memory will fit.

To conserve memory, there are deterministic, pseudo-dynamic allocation schemes one might use. For example, if the location of a memory pool to be allocated is known at runtime, an unconstrained array of bytes may be coupled with an address clause if one is lucky enough to use a compiler which supports using an unconstrained object in an address clause. In some cases, these may provide just what is needed:

```ada
large_pool : array (<> of byte); for large_pool use at 16#0010_FACE#;
```

In other cases, something a little more clever is called for. The Ada Language Reference Manual states that a value (other than null) may only be assigned to an access type by means of an allocator (new), or assigned from another access object of the same type. In most cases, this is not strictly true. One of the more important tricks an embedded systems Ada programmer can know is the following: An access type can often be safely converted into a type system.address, and vice versa. Access types and addresses look essentially the same to most Ada compilers and runtime systems. Therefore, the following is possible:

```ada
temp_address : system.address;
temp_ptr : block_pointer;

function address_to_bptr is new unchecked_conversion( system.address, block_pointer);
function bptr_to_address is new unchecked_conversion( block_pointer, system.address);

temp_address := bptr_to_address(temp_ptr);
temp_ptr := address_to_bptr(temp_address);
```

If `temp_ptr` were defined as follows:

```ada
a_pointer is access a_record;
temp_ptr : a_pointer;
temp_record : a_record;
```

then this assigns the contents of an `a_record` to the address contained in `temp_pointer`:

```ada
temp_ptr.all := temp_element;
```

If this sort of unchecked conversion is not possible, the compiler will usually provide an alternative method for converting an address to a pointer and vice versa. This idea, combined with some good Ada and a knowledge of system memory requirements allows great flexibility in designing information structures for embedded systems.

For example, consider the idea of a dynamic array which must start at a runtime-determined address. Memory will be sequentially templated with one type and the index to each element will be an offset into memory. Noting that pointer dereferencing in Ada is automatic, and that and that a generic can be used to good advantage, examine the code which performs this feat:

```ada
with System;
with Unchecked_Conversion;
```
If an embedded-systems programmer uses Ada long enough, he or she will undoubtedly need to represent a variable-sized data structure, possibly at an address determined at runtime. Links between such structures may also be necessary, and the programmer may finally break down and attempt to use pointers.

Many embedded-systems programmers who use Ada have very little experience with pointers (access types) in Ada. This is largely because the performance of most real-time embedded systems cannot tolerate the overhead of the memory management afforded by the usually-provided Ada runtime. This system is generally non-deterministic due to garbage collection considerations, and can be wasteful of both memory and clock cycles. However, the LRM clearly states that an access object (pointer) can only be assigned the result of the NEW allocator, the contents of another access object of the same type, or the NULL constant. Therefore, software engineers often give up on the standard Ada runtime and resort to simple compile-time allocated arrays.

There are other options, unknown to many Ada programmers. For the moment, however, consider the array. Even with compile-time allocated arrays, programmers with a modicum of imagination can conceive of one array of records which has indices referring to another array of records. One may also see an advantage in using a parallel array of booleans to indicate when a record is "in use." Using arrays in this way can simulate a linked list, but is potentially quite wasteful of memory.

At this juncture, one might ask about the value of conserving memory when it can be purchased in cheap bushells of megabytes. True, memory seems to get a bit less expensive every day. However, if one thinks back to the old days (pre 1980), it is obvious that while memory was an order of magnitude more expensive then, embedded systems were an order of magnitude smaller. Today, with embedded systems weighing in at up to millions of lines of source code, there seems to be an unlimited demand for memory in a box, but the fact remains that only a finite amount of memory will fit.

To conserve memory, there are deterministic, pseudo-dynamic allocation schemes one might use. For example, if the location of a memory pool to be allocated is known at runtime, an unconstrained array of bytes may be coupled with an address clause if one is lucky enough to use a compiler which supports using an unconstrained object in an address clause. In some cases, these may provide just what is needed:

```ada
large_pool : array (<> ) of byte;
for large_pool use at 16#0010_FACE#;
```

In other cases, something a little more clever is called for. The Ada Language Reference Manual states that a value (other than null) may only be assigned to an access type by means of an allocator (new), or assigned from another access object of the same type. In most cases, this is not strictly true. One of the more important tricks an embedded systems Ada programmer can know is the following: An access type can often be safely converted into a type system.address, and vice versa. Access types and addresses look essentially the same to most Ada compilers and runtime systems. Therefore, the following is possible:

```ada
temp_address : system.address;
temp_ptr : block_pointer;

function address_to_bptr is new unchecked_conversion( system.address, block_pointer);
function bptr_to_address is new unchecked_conversion( block_pointer, system.address);

temp_address := bptr_to_address(temp_ptr);
temp_ptr := address_to_bptr(temp_address);
```

If `temp_ptr` were defined as follows:

```ada
a_pointer is access a_record;
temp_ptr : a_pointer;
temp_record : a_record;
```

then this assigns the contents of an a_record to the address contained in `temp_pointer`:

```ada
temp_ptr.all := temp_element;
```

If this sort of unchecked conversion is not possible, the compiler will usually provide an alternative method for converting an address to a pointer and vice versa. This idea, combined with some good Ada and a knowledge of system memory requirements allows great flexibility in designing information structures for embedded systems.

For example, consider the idea of a dynamic array which must start at a runtime-determined address. Memory will be sequentially templated with one type and the index to each element will be an offset into memory. Noting that pointer dereferencing in Ada is automatic, and that a generic can be used to good advantage, examine the code which performs this feat:

```ada
with System;
with Unchecked_Cons version;
generic
type Table_Element_Type is private;
with function "+" (right : natural; left : natural) return natural;

package Template_Table is
  type Element_Ptr is access Table_Element_Type;

  function Address_To_Access is new Unchecked_Conversion( System.Address, Element_Ptr );
  function Natural_To_Address is new Unchecked_Conversion( Natural, System.Address );

  function Get_Element ( Table_Start: in System.Address; Index in Natural )
    return Table_Element_Type;

  procedure Put_Element ( Table_Start in System.Address; Index in Natural; Element in Table_Element_Type );

end Template_Table;

package body Template_Table is

  function Locate_Slot ( Table_Start in System.Address; Index in Natural )
    return Element_Ptr is
    begin
      Element_Ptr := Address_To_Access ( Element_Address );
      return Element_Ptr;
    end Locate_Slot;

  function Get_Element ( Table_Start : in System.Address; Index : in Natural )
    return Table_Element_Type is
  begin
    Element_Ptr := Locate_Slot ( Table_Start, Index );
    return Table_Element_Type is
      Element_Ptr := Element_Ptr;
    end Get_Element;
    procedure Put_Element ( Table_Start : in System.Address; Index : in Natural; Element : in Table_Element_Type ) is
      Element_Ptr := Element_Ptr;
    begin
      Element_Ptr := Locate_Slot ( Table_Start, Index );
      Element_Ptr.all := Element;
      end Put_Element;
    end Template_Table;

Please note that this package, and all other code included in this paper are provided for illustrative purposes only and may not have been fully tested and validated. Note further that complex error-case handling has been eliminated from all examples so that the general concepts presented may be more easily understood.

Table_Element_Type is determined in the package specification via the instantiation of the generic. Element_Ptr is based on Table_Element_Type. The generic package may be instantiated as follows:

  package Test_Table is new Template_Table ( Test_Record );

After the package has been instantiated, subprograms within the package may be used as shown here:

  A_Record := Test_Table.Get_Element ( Temp_Address, I );

Here is a summary of the subprograms in the package body, TABLE.ADB:

The Locate_Slot function works as follows:

1. The base address for the table and the offset in number of table elements are provided as input parameters.
2. Determine the number of bytes (the minimum is one) needed to store one element.
3. The offset is the number of bytes into the table as given by Index * Element_Size.
type Table_Element_Type is private;
with function "+" (right : natural; left : natural) return natural;

package Template_Table is
  type Element_Ptr is access Table_Element_Type;
  function Address_To_Access is new
    Unchecked_Conversion( System.Address, Element_Ptr );
  function Natural_To_Address is new
    Unchecked_Conversion( Natural, System.Address );
  function Get_Element ( Table_Start: in System.Address; Index in Natural )
    return Table_Element_Type;
  procedure Put_Element ( Table_Start in System.address;
    Index in Natural; Element in Table_Element_Type );
end Template_Table;

package body Template_Table is
  function Locate_Slot ( Table_Start in System.Address; Index in Natural )
    return Element_Ptr is
    Element_Pointer := Address_To_Access( Element_Address);
    return Element_Pointer;
end Locate_Slot;
  function Get_Element( Table_Start : in System.Address; Index : in Natural )
    return Table_Element_Type is
    Element_Pointer := Locate_Slot( Table_Start, Index );
    return Element_Pointer.all; end Get_Element;
  procedure Put_Element( Table_Start : in System.Address; Index : in Natural;
    Element : in Table_Element_Type ) is
    begin          % Element_Pointer := Locate_Slot ( Table_Start, Index );
      Element_Pointer := Element_Ptr;
    end Put_Element;
end Template_Table;

Please note that this package, and all other code included in this paper are provided for
illustrative purposes only and may not have been fully tested and validated. Note further that
complex error-case handling has been eliminated from all examples so that the general
concepts presented may be more easily understood.

Table_Element_Type is determined in the package specification via the instantiation of the
generic. Element_Ptr is based on Table_Element_Type. The generic package may be
instantiated as follows:

package Test_Table is new Template_Table ( Test_Record );

After the package has been instantiated, Subprograms within the package may be used as
shown here:

A_Record := Test_Table.Get_Element ( Temp_Address, I );

Here is a summary of the subprograms in the package body, TABLE.ADB:

The Locate_Slot function works as follows:

1. The base address for the table and the offset in number of table elements are provided
   as input parameters.
2. Determine the number of bytes (the minimum is one) needed to store one element.
3. The offset is the number of bytes into the table as given by Index * Element_Size.
Note that the result must be converted into an address.
4. Add base and offset together to get the address of the element.
5. Convert the address to an access type.
6. Return an access type.

The Get_Element function:
1. Locate the slot from which to retrieve the element.
2. Return Element_Pointer.all. This automatically dereferences the pointer and yields the contents.

The Put_Element procedure:
1. Locate the slot in which to store the element.
2. Assign Element to Element_Pointer.all. This places the value of Element in memory at the address indicated by Element_Pointer.

This kind of memory-templated table is useful for many things, including reading uploaded configuration tables or dumping the contents of memory, but by itself does not really solve many memory management problems. However, one simple application of this idea puts two of these together in a free pool and allows them to grow toward and shrink back from each other. This allows two dynamic arrays to exist in memory which would have been used by only one fixed-size array.

Another application of the template table uses one as a bit map to keep track of the status of memory blocks, and another as a free pool of blocks. This idea is illustrated in the following simple generic which can only allocate a single size of block per instantiation, yet can be quite useful in a multi-user system, or where only a few sizes of blocks are required. Using this method, allocation and deallocation of memory is generally faster than with any variable-sized block approach, but memory can be wasted if too many different block sizes are needed.

with template_table;
with system;
generic
Block_Size : in positive := 640; — in bits
Memory_Start : in system.address;
Memory_End : in system.address;

package Fixed_Allocator is
    type Byte is range 0 .. 255;
    for Byte’size use 8;

type Block is array (1 .. block_size) of Byte;
package Bit_Map is new template_table(boolean,“+”);

package Memory_Pool is new template_table(Block,“+”);
Number_of_Blocks : positive :=
    ((Memory_Pool.Address_To_Natural(Memory_End) - Memory_Pool.Address_To_Natural(Memory_Start) + 1)*8/Block_Size + 1;
Bit_Map_Size : positive := Number_of_Blocks;
Bit_Map_End : positive :=
    Memory_Pool.Address_To_Natural(Memory_Start) + Number_Of_Blocks;

function Alloc return system.address;
procedure Free (node : in system.address);
end Fixed_Allocator;
package body Fixed_Allocator is
    function Alloc return system.address is
        begin
            for I in 1 .. Number_Of_Blocks loop
                if Bit_Map.Get_Element(Memory_Start, I) = FALSE then
                    Bit_Map.Put_Element(Memory_Start, I, TRUE);
                    return Memory_Pool.Natural_To_Address((I - 1)*Block_Size + Bit_Map_End + 1);
                end if;
            end loop;
        end;

    procedure Free (Node : in system.address) is
        Index : positive := (Memory_Pool.Address_To_Natural(Node) - Bit_Map_End - 1)/Block_Size + 1;
        begin
            Bit_Map.Put_Element(Memory_Start, Index, FALSE);
        end;
    end;

end Fixed_Allocator;

Function Alloc works as follows:
1. Loop through bitmap to find a free block.
2. Return the address of the free block.

Procedure Free:
Note that the result must be converted into an address.
4. Add base and offset together to get the address of the element.
5. Convert the address to an access type.
6. Return an access type.

The Get_Element function:
1. Locate the slot from which to retrieve the element.
2. Return Element_Pointer.all. This automatically dereferences the pointer and yields the contents.

The Put_Element procedure:
1. Locate the slot in which to store the element.
2. Assign Element to Element_Pointer.all. This places the value of Element in memory at the address indicated by Element_Pointer.

This kind of memory-templated table is useful for many things, including reading uploaded configuration tables or dumping the contents of memory, but by itself does not really solve many memory management problems. However, one simple application of this idea puts two of these together in a free pool and allows them to grow toward and shrink back from each other. This allows two dynamic arrays to exist in memory which would have been used by only one fixed-size array.

Another application of the template table uses one as a bit map to keep track of the status of memory blocks, and another as a free pool of blocks. This idea is illustrated in the following simple generic which can only allocate a single size of block per instantiation, yet can be quite useful in a multi-user system, or where only a few sizes of blocks are required. Using this method, allocation and deallocation of memory is generally faster than with any variable-sized block approach, but memory can be wasted if too many different block sizes are needed.

```
with template_table;
with system;
generic
  Block_Size in positive := 640; -- in bits
  Memory_Start in system.address;
  Memory_End in system.address;
package Fixed_Allocator is
  type Byte is range 0 .. 255;
  for Byte's size use 8;
  type Block is array (1 .. block_size) of Byte;
  package Bit_Map is new template_table(boolean,"+");
package Memory_Pool is new template_table(Block,"+");

  Number_of_Blocks : positive :=
    (Memory_Pool.Address_To_Natural(Memory_End)
    - Memory_Pool.Address_To_Natural(Memory_Start)) \ 8/Block_Size+1;
  Bit_Map_Size : positive := Number_of_Blocks;
  Bit_Map_End : positive :=
    Memory_Pool.Address_To_Natural(Memory_Start)+
    Number_of_Blocks;

  function Alloc return system.address;
  procedure Free (node : in system.address);
end Fixed_Allocator;
```

Function Alloc works as follows:
1. Loop through bitmap to find a free block.
2. Return the address of the free block.

Procedure Free:

```
1. Convert the address to an index into the bit map.
2. Set the bit at the index to FALSE

Often, capabilities beyond those provided in a fixed-sized allocator are needed. Some of these can be found in a variable-sized block allocator. Such an allocator has its own set of limitations, however. The strategy illustrated in the following package works best when there is only a narrow (< 30%) variability in requested block sizes, with the requests distributed randomly by size. Where X is the median, the worst case is for all of the memory pool to be divided up into small (1/2X) blocks immediately prior to multiple requests for large (2X) blocks. The clever designer will be able to come up with multiple variations on this theme, better suited to the requirements for his or her system. The code included to illustrate this approach, for simplicity sake, does not address the action to be taken when no memory is available. Depending on system needs, this can range from returning an error code to re-allocating parts of the memory pool. In between are methods such as calling the requestor back when a block of the requested size is available, or re-packing the status table.

```haskell
package Allocate Variable Block is

type byte is range 0 .. 255;
for byte' size use 8;

type status_type is (in_use, available);

type status_record is

record
  status : status_type;
  node_size : natural;
  node_location : system.address;
end record;

a_status : status_record;

package status_template is

new
  template_table(status_record,"-"),

function integer_to_address is new
  unchecked_conversion(integer,system.address);

function address_to_integer is new
  unchecked_conversion(system.address,integer);

end status_template;

bottom_of_memory : integer := 16#FACE_FADE#;

end_of_allocated_memory : integer := address_to_integer (top_of_memory);
end_of_status_table : integer := bottom_of_memory;
number_of_status_nodes : integer := 0;

function Alloc (s : natural) return system.address;

end Allocate Variable Block;

-- allocate or free a variable-size block of memory package

body Allocate Variable Block is

function Alloc (s : natural) return system.address is
  a_status : status_record;
  an_address : system.address;

  -- allocate a new block of unused memory
  function allocate_new_node return system.address is
    temp : natural;
    begin
      if ((end_of_status_table-end_of_allocated_memory))*8 >= s then
        temp := end_of_allocated_memory + 1;
        end_of_allocated_memory := end_of_allocated_memory + s/8;
      end if;
      return status_template.natural_to_address(temp);
    end allocate_new_node;

  -- get an available status node address;
  begin alloc
    recycle an old one if possible
    for i in 1..number_of_status_nodes loop
      a_status := status_template.get_element (memory'address,i);
      if a_status.node_size = s and a_status.status = available then
        a_status.status := in_use;
        a_status.node_size := s;
        return a_status.node_location;
      end if;
    end loop;
  end alloc;
end Allocate Variable Block;
```
1. Convert the address to an index into the bit map.
2. Set the bit at the index to FALSE

Often, capabilities beyond those provided in a fixed-sized allocator are needed. Some of these can be found in a variable-sized block allocator. Such an allocator has its own set of limitations, however. The strategy illustrated in the following package works best when there is only a narrow (< 30%) variability in requested block sizes, with the requests distributed randomly by size. Where X is the median, the worst case is for all of the memory pool to be divided up into small (1/2X) blocks immediately prior to multiple requests for large (2X) blocks. The clever designer will be able to come up with multiple variations on this theme, better suited to the requirements for his or her system. The code included to illustrate this approach, for simplicity sake, does not address the action to be taken when no memory is available. Depending on system needs, this can range from returning an error code to re-allocating parts of the memory pool. In between are methods such as calling the requestor back when a block of the requested size is available, or re-packing the status table.

```pascal
with template_table;
with system;
with text_io; use text_io;
with unchecked_conversion;

package Allocate Variable Block is

  type byte is range 0 .. 255;
  for byte' size use 8;

  type status_type is (in_use, available);

  type status_record is
    record
      status : status_type;
      node_size : natural;
      node_location : system.address;
    end record;

  a_status : status_record;

  package status_template is new
    template_table(status_record,"-");

  function integer_to_address is new
    unchecked_conversion(integer,system.address);
  function address_to_integer is new
    unchecked_conversion(system.address,integer);

  top_of_memory : system.address := 16#BEEF_CAFE#;

  bottom_of_memory : integer := 16#FACE_FADE#;
  end_of_allocated_memory : integer := address_to_integer(top_of_memory);
  end_of_status_table : integer := bottom_of_memory;
  number_of_status_nodes : integer := 0;

  function Alloc (s : natural) return system.address;
  procedure Free (location : system.address);

end Allocate Variable Block;

-- allocate or free a variable-size block of memory package

function Alloc (s : natural) return system.address is
  a_status : status_record;
  an_address : system.address;

  -- allocate a new block of unused memory
  function allocate new block return system.address is
    temp : natural;
    begin
      if ((end_of_status_table-end_of_allocated_memory)/8 >= s
          then
              temp := end_of_allocated_memory + 1;
            end_of_allocated_memory := end_of_allocated_memory + s/8;
          end if;
        return status_template.natural_to_address(temp);
      end allocate new node;

  -- get an available status node address;
  begin - alloc
    recycle an old one if possible
    for i in 1..number_of_status_nodes loop
      a_status := status_template.get_element(memory'address,i);
      if a_status.node_size = s and a_status.status = available
        then
            a_status.status := in_use;
            a_status.node_size := s;
            return a_status.node_location;
        end if;
      end loop;
```
otherwise, get a new one, if enough space is left
if ((end_of_status_table-end_of_allocated_memory)*8 >=
a status'size then
an_address := allocate_new_block;
an_status := in_use;
an_status.node_size := s;
an_status.node_location := an_address;
number_of_status_nodes := number_of_status_nodes + 1;
end_of_status_table := end_of_status_table -
a_status'size/8;
status_template.put_element
(top_of_memory,number_of_status_nodes,a_status);
end if;
end Alloc;

procedure Free (location : system.address) is
begin
- search list for address, set status to available
for i in 1 .. Number of Status_Nodes loop
  a_status :=
  status_template.get_element(integer_to_address
    (Bottom Of_Memory),i);
  if status_template.address_to_natural
    (a_status.node_location) =
    status_template.address_to_natural(location) then
    a_status.status := available;
    end if;
end loop;
- could add a procedure to re-pack status table. end
Allocate_Variable_Block;

Function Alloc works as follows:
1. Search the status table for a free block of at least the size requested.
2. If one can't be found, and un-allocated memory is available, move the bottom of
allocated memory down according to the size requested.
3. Return the previous end of allocated memory + 1.
4. If enough space remains, create a new status node using the returned address.

Procedure Free:
1. Search the status table for a node containing the specified address.
2. When the node is found, set the status to available.

The above variable-sized alloc and free, as well as the fixed-size block allocation package do
not show how multi-user resource-sharing would work using either Ada tasking or some
other embedded operating system. This is a topic which is beyond the scope of this paper,
and since it is completely system-dependent, is best left to individual implementations.

The following illustrates the use of the Allocate_Variable_Block:

with Allocate_Variable_Block; use Allocate_Variable_Block;
with system;
with unchecked_conversion;
procedure v_test is
  record_count : integer := 0;
type a_record;   -- define a linked-list node
type link is access a_record;
type a_record is
  record
    I : integer;
    A : string(1..140);
    J : integer;
    L : link;
  end record;

function Address_to_Link new unchecked_conversion
  (system.address,Link);
function Link_to_Address new unchecked_conversion
  (Link,system.address);

a : a_record;
head : Link;  -- pointers to the list
tail : Link;
current : Link;
begin
  for i in 1 .. 3 loop
    record_count := record_count + 1;
    if record_count = 1 then
      Current := Address_to_Link(Alloc(A'size));
      allocate an address
      Head := Current;
    else
      Current.L := Address_to_Link(Alloc(A'size));
      Current := Current.L;
    end if;
    Current.L := null;
otherwise, get a new one, if enough space is left
if ((end_of_status_table-end_of_allocated_memory)*8 >=
a_status'size then
  an_address := allocate_new_block;
a_status := in_use;
a_status.node_size := s;
number_of_status_nodes := number_of_status_nodes + 1;
end_of_status_table := end_of_status_table -
a_status'size/8;
status_template.put_element
  (top_of_memory,number_of_status_nodes,a_status);
end Alloc;

procedure Free (location : system.address) is
begin
  - search list for address, set status to available
  for i in 1 .. Number_of_Status_Nodes loop
    a_status :=
      status_template.get_element(integer_to_address
        (Bottom_of_Memory),i);
    if status_template.address_to_natural
      (a_status.node_location) =
        status_template.address_to_natural(location) then
      a_status := available;
    end if;
  end loop;
end Free;
- could add a procedure to re-pack status table. end
Allocate_Variable_Block;

Function Alloc works as follows:
1. Search the status table for a free block of at least the size requested.
2. If one can't be found, and un-allocated memory is available, move the bottom of
   allocated memory down according to the size requested.
3. Return the previous end of allocated memory + 1.
4. If enough space remains, create a new status node using the returned address.

Procedure Free:
1. Search the status table for a node containing the specified address.
2. When the node is found, set the status to available.

The above variable-sized alloc and free, as well as the fixed-size block allocation package do
not show how multi-user resource-sharing would work using either Ada tasking or some
other embedded operating system. This is a topic which is beyond the scope of this paper,
and since it is completely system-dependent, is best left to individual implementations.

The following illustrates the use of the Allocate_Variable_Block:

with Allocate_Variable_Block; use Allocate_Variable_Block;
with system;
with unchecked_conversion; procedure v_test is
record_count : integer := 0;
type a_record; - define a linked-list node
type link is access a_record;
type a_record is
  record
    I : integer;
    A : string(1..140);
    J : integer;
    L : link;
  end record;
function Address_to_Link is new unchecked_conversion
  (system.address,Link);
function Link_to_Address is new unchecked_conversion
  (Link,system.address);

a : a_record;
head : Link; - pointers to the list
tail : Link;
current : Link;
begin
  for i in 1..3 loop
    record_count := record_count + 1;
    if record_count = 1 then
      Current := Address_to_Link(Alloc(A'size));
      - allocate an address
      Head := Current;
    else
      Current.L := Address_to_Link(Alloc(A'size));
      Current := Current.L;
    end if;
    Current.L := null;
It should now be apparent that unchecked conversion combined with type system.address, along with automatic dereferencing provide more ways to use access types than simply assigning the result of a NEW allocator from the standard Ada memory management system. In addition, the tools needed for developing a memory manager tailored to a specific embedded system have now been introduced.

Now, a brief explanation of how Ada handles memory. Ada recognizes a "Heap" area wherein all dynamically allocated (using the New allocator) and all unconstrained objects are stored. All objects created during runtime without an allocator —— i.e local variables, parameters, etc. are stored on the "Stack." Some Ada linkers allow, linking without a heap. This may be a good solution if the memory needs of a system are simple, and one does not want to waste memory on a heap.

There are also some linkers which allow the programmer to specify where in memory the heap will be. Using this feature, it may be possible to pre-allocate a variety of node sizes at initialization using the built-in NEW allocator. The addresses and sizes of these nodes could then be stored in a status table, and the nodes allocated and deallocated at runtime as needed.

Alternatively, a designer may chose to specify a range for the heap, and then trust the built-in runtime allocator. This strategy can be a good one when the runtime allocator does not involve garbage collection. In this case, it may be advisable to use the built-in unchecked_deallocation generic procedure to make sure that memory is returned to the heap when it is no longer needed. Depending upon the implementation of built-in memory allocation, this may be redundant since the runtime may already make an effort to recycle unused memory. Using this procedure often involves danger and complexity. There is danger in the fact that one might deallocate a node without resolving all of the pointers to it, and complexity in determining what pointers reference the node. Since the deallocation is unchecked, the runtime storage manager does not know about it, and may allow a pointer to reference a node which has been deallocated, without generating an error, thus leading to disastrous consequences.

The memory management system provided in the standard runtime system is of necessity highly general. It must be ready to allocate or free data blocks of anywhere from one bit in size to several megabytes or more, in any sequence and at any moment. This means that the garbage collection it employs must necessarily be complex, nondeterministic, and often slow and sometimes even wasteful of memory. This may not be noticed in a non-embedded system, but in an embedded system it can be intolerable. When the standard Ada runtime encounters an unconstrained object, only two general actions are possible: either allocate the maximum amount of memory which may possibly be needed for the object (waste of memory,) or allocate only what is needed each time the object is encountered, thus continually allocating and reallocating memory (slow and leads to fragmentation.)

The analysts and designers of an embedded system know the memory requirements of their system far better than any standard memory management system, and can tailor a memory management capability which will be sufficiently flexible yet be fast and efficient. This capability may consist of several memory management utilities, or one central manager, and may or may not allow shared memory resources between processes.

For example, suppose requirements analysis determines that buffers sized within a narrow, specific range must be allocated. If an average size of 1024 bytes would fill the bill, a simple a simple fixed-size allocator may be in order.

An Ada or assembly language purist may balk at using a sort of "back door" feature to do the kind of memory allocation normally done in assembly language. One might well answer such a critic by explaining that using the features of Ada to manage memory instead of resorting to assembly language has the advantage of maintainability, in addition to the fact that good compilers can optimize RISC code far better that the average assembly- language programmer.

Good object-oriented analysis and design should isolate data structures into packages wherein selectors and constructors specific to the class are provided. The fact that using the above allocation methods causes a client package to use a constructor (Get Element, Alloc, etc.) in instead of directly accessing the data is right in line with the object-oriented idea of protecting data objects with constructors and selectors. Another good Object-Oriented principle is that the implementation connected to the data constructor should remain hidden from packages which with the data-structure package. Therefore, it should make no difference to client packages whether the data is managed as a linked list or a template table. However, if as far as detailed design is concerned, an object oriented designer must take care that the data isolation mandated in the methodology does not result in copying blocks of data back and forth between objects. As it turns out, what has been discussed here about memory and pointers can help with these implementation details.

Quite the opposite from the sometimes-stated fear that using access types within an embedded system can cause a serious decrease in performance, careful use of access types can actually increase the throughput and conserve memory. Since Ada does guarantee that a object will not be copied even when it is an IN OUT parameter, it can be preferable to use unchecked conversion and an address clause set up a pointer to an object and pass the pointer, as in the following:

\[
\text{x_ptr} := \text{address_to_x_ptr(x_object'address)}
\]

To ensure memory efficiency, there is one other feature of Ada which should be considered: representation clauses, also known as rep specs or rep clauses. The Ada Language Reference Manual states that rep specs are provided to allow a direct mapping of Ada types onto the underlying machine, thus increasing the efficiency of memory storage. Most embedded systems programmers who have used Ada are familiar with rep specs as in the following:
It should now be apparent that unchecked conversion combined with type system address, along with automatic dereferencing provide more ways to use access types than simply assigning the result of a NEW allocation from the standard Ada memory management system. In addition, tools needed for developing a memory manager tailored to a specific embedded system have now been introduced.

Now, a brief explanation of how Ada handles memory. Ada recognizes a “Heap” area wherein all dynamically allocated (using the New allocator) and all unconstrained objects are stored. All objects created during runtime without an allocator — i.e. local variables, parameters, etc. are stored on the “Stack.” Some Ada linkers allow, linking without a heap. This may be a good solution if the memory needs of a system are simple, and one does not want to waste memory on a heap.

There are also some linkers which allow the programmer to specify where in memory the heap will be. Using this feature, it may be possible to pre-allocate a variety of node sizes at initialization using the built-in NEW allocator. The addresses and sizes of these nodes could then be stored in a status table, and the nodes allocated and deallocated at runtime as needed.

Alternatively, a designer may choose to specify a range for the heap, and then trust the built-in runtime allocator. This strategy can be a good one when the runtime allocator does not involve garbage collection. In this case, it may be advisable to use the built-in unchecked_deallocation generic procedure to make sure that memory is returned to the heap when it is no longer needed. Depending upon the implementation of built-in memory allocation, this may be redundant since the runtime may already make an effort to recycle unused memory. Using this procedure often involves danger and complexity. There is danger in the fact that one might deallocate a node without resolving all of the pointers to it, and complexity in determining what pointers reference the node. Since the deallocation is unchecked, the runtime storage manager does not know about it, and may allow a pointer to reference a node which has been deallocated, without generating an error, thus leading to disastrous consequences.

The memory management system provided in the standard runtime system is of necessity highly general. It must be ready to allocate or free data blocks of anywhere from one bit in size to several megabytes or more, in any sequence and at any moment. This means that the garbage collection it employs must necessarily be complex, nondeterministic, and often slow and sometimes even wasteful of memory. This may not be noticed in a non-embedded system, but in an embedded system it can be intolerable. When the standard Ada runtime encounters an unconstrained object, only two general actions are possible: either allocate the maximum amount of memory which may possibly be needed for the object (waste of memory,) or allocate only what is needed each time the object is encountered, thus continually allocating and deallocating memory (slow and leads to fragmentation.)