Named Memory Pool for Ada

Luke Wong
CMC Electronics Inc.
Ottawa, Ontario, Canada
luke.wong@cmcelectronics.ca

Stephen Michell
Maurya Software Inc
Ottawa, Ontario, Canada
stephen.michell@maurya.on.ca

Brad Moore
General Dynamics Canada
Calgary, Alberta, Canada
brad.moore@gdcanada.ca

Abstract

This paper proposes a new paradigm in Ada to access different types of memory in single or multi-partitions real-time, safety-critical computer systems. The approach recognizes that simple allocation and assignment statements are often insufficient. The named memory pool model compliments the storage pools in Ada.

1 Introduction

Specialized computer systems often contain multiple kinds of memory. Some of these kinds of memory cannot be supported simply by mapping the memory region. Because of this, the standard Ada paradigm of creation of storage pools, simple allocation and assignment to memory via allocators is insufficient. We propose a method of named memory spaces and a library package to reliably access and update objects in memory.

2 Issues with Memory Types

There are many types of memory in a modern computer system, especially one that targets embedded systems. Examples are SRAM, EPROM, EEPROM, and Flash. Different types of memory may be on multiple buses accessible by multiple processors. A memory device may be on several buses to reduce bus contention.

POSIX supports a typed memory API. It is versatile enough to support diverse memory types with multiple buses, which cannot be provided by the memory mapped files or shared memory APIs.

For Ada Storage_Pools, or the POSIX memory models, the memory is accessed as a variable once the memory is allocated from the memory pool and mapped to the application. The benefit of this approach is that this access method is simple and is sufficient for most applications.

There are several issues related to these memory access models:

1. The read/write speed of the memory device may lead to performance issues. Some devices e.g. NAND flash, may be slow and the application cannot afford to spend a long time in memory access operations.
2. In safety-critical applications, the application may need to know whether an error occurs when the memory is accessed, especially with those memory devices that have a limited life span. e.g. Flash.
3. Some memory types require additional control when accessing memory locations, e.g. writing to a control register may be needed prior to an EEPROM write. This cannot be easily addressed with the APIs of these memory models.
4. In some partitioned operating systems, where each partition has a different assurance level, such as in avionics, access from different partitions will be subject to different rules.[1]

1. Access to different areas of the same typed memory from partitions with different assurance levels leads to excessive work upgrading the assurance on partitions with otherwise lower assurance needs. For ex-

5. There may be driver already written for the memory device in other programming languages like C, the driver may not necessarily address all the application requirements such as partitioning. This model provides a simple "middleware" API to supplement those capabilities not provided by the underlying device driver.

We therefore propose a common memory model similar to direct file I/O to address these issues. This model is independent of the underlying memory device structure, it also allows multiple buses configurations similar to typed memory described in [POSIX2009].

3 Named Memory Pool

We define a package called *Named_Memory_Pool* that specifies a *Root_Memory_Pool* abstract tagged type and the operations to operate on objects of this type. The implementation defines new memory pool types, and overrides the abstract subprograms in this package.

```
with System.Storage_Elements;
package System.Named_Memory_Pool is
    type Root_Memory_Pool is abstract tagged limited private;
    type Memory_Type is (Read_Only, Read_Write);
    type Access_Mode is (Read_Only, Write_Only, Read_Write);
    type Delete_Mode is (Unallocated_Mode, Force_Mode);

    type Status_Type is (
       No_Error,
       Device_Error,
       Name_Error,
       Allocate_Error,
       Access_Error,
       End_Of_Region_Error,
       Configuration_Error);

    procedure Create_Memory_Pool (
      Pool             : in out Root_Memory_Pool;
      Pool_Id          : in     Natural;
      Device_Name      : in     String;
      Memory_Name      : in     String;
      Port_Name        : in     String;
      Memory_Offset    : in     System.Storage_Elements.Storage_Count;
      Memory_Length    : in     System.Storage_Elements.Storage_Count;
      Memory_Word_Size : in     System.Storage_Elements.Storage_Count;
      Pool_Type        : in     Memory_Type;
      Status           :    out Status_Type)
    is abstract;

    procedure Create (
      Pool          : in out Root_Memory_Pool;
      Pool_Id       : in     Natural;
      Region_Name   : in     String;
      Mode          : in     Access_Mode;
      Memory_Length : in     System.Storage_Elements.Storage_Count;
      Status        :    out Status_Type)
```

ample, flash memory code in the application does not need to be certified at a high assurance level. However, accessing the flash memory directly from a high assurance level may need to certify the flash memory code at the same high assurance level because a memory failure may potentially bring the high assurance partition down. It is therefore desirable in this case to allocate the actual memory read/write to a separate partition.

```
   is abstract;

   procedure Open (
     Pool        : in out Root_Memory_Pool;
     Pool_Id     : in     Natural;
     Region_Name : in     String;
     Mode        : in     Access_Mode;
     Status      :    out Status_Type)
   is abstract;

   procedure Read (
     Pool           : in out Root_Memory_Pool;
     Out_Addr       : in     System.Address;
     Length_To_Read : in     System.Storage_Elements.Storage_Count;
     Length_Read    :    out System.Storage_Elements.Storage_Count;
     Status         :    out Status_Type)
   is abstract;

   procedure Write (
     Pool            : in out Root_Memory_Pool;
     In_Addr         : in     System.Address;
     Length_To_Write : in     System.Storage_Elements.Storage_Count;
     Status          :    out Status_Type)
   is abstract;

   procedure Close (
     Pool   : in out Root_Memory_Pool;
     Status :    out Status_Type)
   is abstract;

   procedure Delete (
     Pool        : in out Root_Memory_Pool;
     Region_Name : in     String;
     Mode        : in     Delete_Mode;
     Status      :    out Status_Type)
   is abstract;

   function Get_Region_Length (
      Pool        : in Root_Memory_Pool;
      Region_Name : in String)
     return System.Storage_Elements.Storage_Count
   is abstract;

private
   type Root_Memory_Pool is abstract tagged limited
   record
      Pool_Id : Natural;
      Region_Id : Natural;
      Start_Address : System.Address;
   end record

end Named_Memory_Pool;
```

The proposal given above is written with “out” *Status* parameters instead of using exceptions. The justification is that many of the environments needing such a package also disallow the use of exception handlers. We acknowledge that a version of the package that uses exceptions may also be required. One approach is to have a child *System.Named_Memory_Pool.Error_Handing* package defining abstract procedures to handle errors, and remove the "out" *Status* parameters from the API:

```
package System.Named_Memory_Pool.Error_Handling is
         type Root_Memory_Pool_Error is abstract tagged limited private;

   procedure Raise_Error
     (Pool_Error : in out Root_Memory_Pool_Error;
      Status     : in     Status_Type)
    is abstract;

   function Get_Status
     (Pool_Error : in Root_Memory_Pool_Error) return Status_Type
   is abstract;

private
   type Root_Memory_Pool_Error is abstract tagged limited null record;
end System.Named_Memory_Pool.Error_Handling;
```

An application that allows exception handlers may implement *Raise_Error()* to raise an exception, the exception handler may call *Get_Status()* to get the error status. For *No_Exception_Handler* implementations, *Raise_Error()* is a null operation, the application calls *Get_Status()* to get the completion status.

3.1 Defining the Memory Pool

Create_Memory_Pool() defines a named memory pool on a memory device. The memory pool is characterized by

- its *Device_Name,*
- *Memory_Name*, and
- the port (bus) specified by *Port_Name* the memory is connected to.

There is no restriction on the number of ports connected to a memory pool. The *Device_Name*, *Memory_Name*, *Port_Name* are unique in the system. To simplify implementation, an identifier *Pool_Id* is used to refer to this memory pool in the *Named_Memory_Pool* API.

Memory_Word_Size is the storage units in bits, as in the Ada convention.

All or part of the memory on a memory device may be assigned to the memory pool. *Memory_Offset* and *Memory_Length* define that portion of physical memory that is assigned to the named memory pool. These areas should not overlap. *Memory_Length* is restricted to multiples of *Memory_Word_Size*.

This example shows two memory devices - MemDev1 and MemDev2 - both connected to Bus A and Bus B. There is 1 memory pool MemPool1 defined on MemDev1. This memory pool can be accessed from memory ports PortA1 and PortB1. There are 2 memory pools MemPool2.1 and MemPool2.2 defined on MemDev2. Each pool is assigned to part of MemDev2. MemPool2.1 can only be accessed from PortA2, MemPool2.2 can only be accessed from PortB2.

Memory_Type can be *Read_Only* or *Read_Write*.

Create_Memory_Pool() returns these error status:

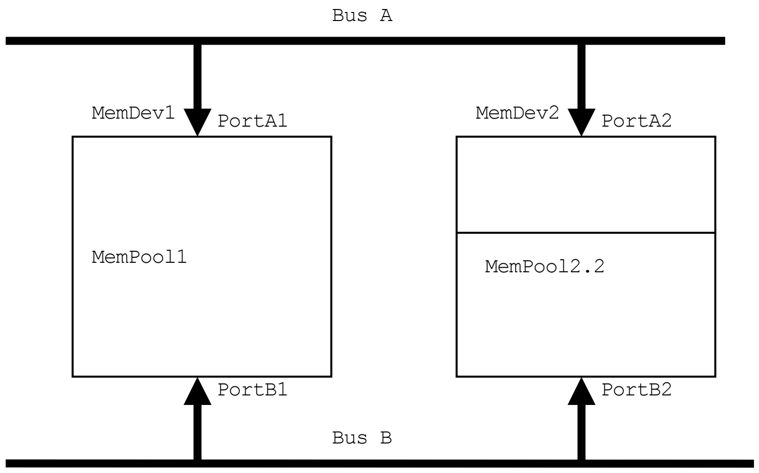

Figure 1: Example of shared access to Partitioned Memory

Configuration_Error if an memory pool has already been defined with the same *Memory_Name* in the system, or the address range of the memory pool overlaps with another on the same memory device.

3.2 Allocating Memory

Create() allocates a memory region of the specified *Memory_Length* in the named memory pool identified by *Pool_Id*, and maps it to the process's address space. The region is identified by the *Region_Name*. The region may be contiguous or non-contiguous. Contiguous memory region simplifies implementation.

Create() translates the *Region_Name* to a *Region_Id* that identifies the allocated memory region in the named memory pool. It returns the *Start_Address* of the memory region, and the *Region_Id* in the *Root_Memory_Pool* type for subsequent named memory pool operations.

Create() also saves the *Pool_Id* in the *Root_Memory_Pool* type. (*Pool_Id*, *Region_Id*) uniquely identifies the allocated memory region in the system.

Create() returns these error status:

Device_Error for a hardware or underlying driver failure and the allocation fails.

Name_Error if a memory region of the same Region_Name has been allocated in the named memory pool. Note that the same Region_Name may be used in a different named memory pool.

Allocate_Error if there is not enough resource to allocate *Memory_Length* storage units in the named memory pool.

Access_Error if the memory type does not support the access Mode, e.g. *Read_Write* on ROM.

On successful completion, the memory region is left open for subsequent read/write operations. The specified access *Mode* applies between a *Create()* or *Open()* and a *Close()*, and not to the life span of the region, the application may *Close()* and re-*Open()* the memory region with a different access Mode.

3.3 Mapping and Unmapping Memory

Open() maps an existing memory region with *Region_Name* in the named memory pool identified by *Pool_Id* to the process's address space. It allows another process to share an allocated memory region.

Open() translates the Region_Name to a *Region_Id* that identifies the allocated memory region in the named memory pool. It returns the *Start_Address* of the memory region, and the *Region_Id* in the *Root_Memory_Pool* type for subsequent named memory pool operations. The *Region_Id* and *Start_Address* are usually values read from some internal table calculated by the last *Create()* call.

Open() returns these error status:

> *Device_Error* if there is a hardware or underlying driver failure and the open fails.
>
> *Name_Error* if there is no memory region with the *Region_Name* in the named memory pool.
>
> *Allocate_Error* if the memory region has been deallocated by *Delete()*.
>
> *Access_Error* if the memory type does not support the access *Mode*, e.g. *Read_Write* on ROM.
>
> *Close()* unmaps the memory region mapped by *Create()* or *Open()* from the process's address space, thereby severing access to the memory region.
>
> *Close()* sets *Start_Address* in the *Root_Memory_Pool* type to *System.Null_Address*. It is not necessary to specify *Region_Id* and *Pool_Id*, they are not relevant.
>
> *Close()* returns these error status:
>
> *Allocate_Error* if the memory region has been deallocated by Delete().

3.4 Deallocating Memory

Delete() deallocates the memory region with *Region_Name* in the named memory pool. If the delete *Mode* is *Unallocated_Mode*, the region is deallocated if no process is mapped to it. If delete *Mode* is *Force_Mode*, the region is unallocated regardless.

Delete() returns these error status:

> *Device_Error* if there is a hardware or underlying driver failure and the deallocation fails.
>
> *Name_Error* if there is no memory region with the *Region_Name* in the named memory pool.

3.5 Reading and Writing Memory

Read() requests to read *Length_To_Read* storage units from the opened memory region to the address at *Out_Addr*. The actual number of storage units read is returned in *Length_Read*.

Read() is thread-safe.

Read() returns these error status:

> *Device_Error* for a hardware or underlying driver failure and the open fails.
>
> *Allocate_Error* if the memory region has been deallocated by *Delete()*.
>
> *Access_Error* if the region was opened with *Write_Only* access mode.
>
> *End_Of_Region_Error* if *Length_To_Read* is larger than the memory region length. In this case, memory region length storage units are read.

Write() writes *Length_To_Write* storage units at the address *In_Addr* to the opened memory region.

Write() is thread-safe.

Write() returns these error status:

> *Device_Error* for a hardware or underlying driver failure and the open fails.
>
> *Allocate_Error* for a memory region has been deallocated by *Delete()*.
>
> *Access_Error* for a memory region opened in *Read_Only* access mode.
>
> *End_Of_Region_Error* if *Length_To_Write* is larger than the memory region length. In this case, no data is written.

4. Use Cases

One example is to implement the *Named_Memory_Pool* API for accessing an NAND flash memory device in a single partition system and a multi-partition system for a safety-critical application. NAND flash has limited life span, the application requires every flash operation to return a status code to check the integrity of the operation. The underlying flash driver, which may be written in another programming language, is implemented as a simple file system. The model maps the file system to the memory pool, and a file is a mapped region.

In a single partition system, the named memory pool operations may be implemented as a library and linked in with the partition. Thread safety may be implemented by protected objects in *Read()* and *Write()*.

In a multi-partition system, the named memory pool operations may be implemented as a message API. The actual memory services are implemented by a memory server/driver running in another partition. The partitions may have different assurance levels. Clients use the message API to send requests to the memory server using inter-partition communications. Thread safety is achieved by single threading the requests using partition-specific and partition-aware approaches.

Another example is to implement the *Named_Memory_Pool* API for accessing EEPROM device in an embedded system. The model maps the device's I/O address space to a mapped region. EEPROM read/write usually require accessing a control register before the operation, this register access can be coded in the *Read()* and *Write()* operations.

5. Conclusion

This paper provides a solution to interface with different types of memory in a real-time computer system. The requirements of the API are discussed. We believe it is appropriate to further investigate reasonable approaches and alternatives to refine the model proposed herein and given the Ada Working Group as proposals for additional capability in a future revision of Ada.

6. Bibliography

[Ada05] ISO/IEC 8652:2007, The Ada Programming Language,

[POSIX2009] IEEE 1003.1 and IS9945-1:2009, The Portable Operating System Interface