Extending Ada to Assist Multiprocessor Embedded Development

Tony Lowe
Software Engineer
Rockwell Collins

1431 Opus Pl. Suite 200
Downers Grove, IL 60515
aalowe@collins.rockwell.com

Introduction

Multiple processor embedded system implementations are almost required for the expanding responsibilities of embedded systems, but modern interfaces for distributed processing can introduce an excessive amount of overhead for the current speed of embedded processors. CORBA, COM, TCP/IP and such interfaces may be appropriate for client/server based systems where the system can be plugged into the wall. For many embedded and low cost systems, they introduce a level of overhead which dramatically limits the functionality of the actual application software. A less overbearing solution for information sharing within an embedded multiprocessor solution is to use shared memory resources to communicate between the processes. This technique usually incurs very little overhead (depending on shared bus hardware speeds and memory access times), but requires a high level of responsibility and discipline from developers when coding these versions of systems. The Ada language has very good support for an individual process using shared memory, but very little built in support for multiple applications within a system. Thus, the responsibility falls once again on developers to ensure proper data transfer and access between processes on separate processors. This paper intends to propose an initial solution to the problems of data synchronization between processes.

Problem Discussion

Embedded software often uses hardware partitioning to separate different functionality for a variety of reasons. Sometimes, the partitions are for functional purposes: the I/O software would reside on a processor and communicate to a CPU to analyze data and communicate to the user interface software which resides on specialized display hardware. Partitioning also may occur to separate critical functions from less critical functions to ensure continuous operation of critical functions, especially in a safety critical environment where verification weighs heavily into the cost of developing software. Partitioning may just be a result of the growing needs for an application resulting in the need for more processing power. From a different angle, software must often specify data at exact locations to talk to special peripheral devices through registers or specialized memory. Whatever the reasons, the different partitions are required to communicate. When shared memory is used for communications, a process must declare data elements in the shared memory address range in order to pass data, state information, commands or whatever the application requires. Within the Ada language, two methods exist jump out to perform this synchronization:

- *Representation Clauses* - specification to the compiler as to the exact address in which a data item is going to be located.

- *Compiler Options* - specification to the compiler/linker as to the general data environment in which data is going to be located.

These techniques are very useful constructs, but are problematic for maintenance and simulation when using multiple executables.

The representation clause has the distinct advantage that the location of the data is static, and therefore the size of the code and data required by the application does not affect the placement of data in memory. This is very helpful for communicating with devices that have a set memory location according to the memory map and also for easy debugging when symbolic debugging is not available. For example, a memory register is very likely to have the same address for a piece of hardware for the lifetime of the application on that hardware. Thus, hard coding the address of the data object in memory is very positive for these purposes. Representation clauses require the developers to attach to the code the exact location in which the data is going to be stored. This means that if the separate applications have different relative data locations, separate code is needed for the multiple processes.

Process A
```
   IO_Process_State : State_Info;
   for IO_Proccess_State use at 16#60_0F3D#;
```

Process B
```
   IO_Process_State : State_Info;
   for IO_Proccess_State use at 16#90_0F3D#;
```

This means that two versions exist for each package that must be synchronized, and if you consider simulation, it may mean four! While representation clauses are much better for specific data structures, they are not a general solution to large amounts of shared data.

For data items in shared memory, the exact address of the item is not very important or even desirable. If you hard code a record structure and the size of the record changes, the address of the next element in memory may be required to shift to accommodate. As an alternative, open space may be left between data items, but this results in large amounts of unused memory locations and may still not solve this problem. A solution to exact addressing is the use of compiler options to specify the data environment in which the shared variable is to be located.

```
compile code_file.ada /DENV = 96
```

The compile option tells the compiler where the data from the package should be stored in memory by giving it the name of a data environment. This has the advantage that the compiler can modify the position of data items according to their size, and the data items are then adjusted accordingly within the desired address range. This is helpful for locating general data in the correct location, but once again, the position of data items is dynamic. The other executables will be required to synchronize their version of the data items to the exact mapping. This often is achieved by sharing the same compiler, linker and code for all processes, but this is a big limitation, and assumes the tool outputs are identical for the same data structures in different contexts. Similar to representation clauses, the compiler options must be tracked and maintained for each of the separate processes. Different from representation clauses, the required synchronization is normally maintained in a separate physical location from the code. It is very easy for a developer may forget to include the compiler options. This adds complexity to development, integration and maintenance of a system, by adding to the files and code items which must be controlled and maintained. An error in compiler options can be just as critical as one in code, and is often much more difficult to find for inexperienced developers. Each of these techniques have their advantages, but they also have distinct disadvantages in multiprocessor implementations.

Proposed Solution

In general, using shared memory means the developers need to know a great deal about the physical representation of data between the processes and may be required to maintain multiple address ranges to make the system perform correctly. When system performance depends on processor interaction, many of the benefits of using Ada are lost. If you are lucky, the processes will not function at all. If you are not, you will have an application that seems to function but is riddled with intermittent problems that are difficult and expensive to find, though most often very easy to solve with the correct address or compiler option. A better mechanism is to provide support within the language to utilize the benefits of strong typing and variable access even between separate processors. A very simple solution to these problems is to provide a method within the language to specify that data variables are going to be used in a shared area, but to let the tool support provide the actual mechanism for the synchronization. Since this is quite experimental, the best implementation at this point for synchronization would occur as a set of pragmas.

```
pragma Synchronize (<Object, (<Object2, <Object3 ...),
<Data_Environment);
```

- Object - declares the variable being synchronized.
- Data_Environment - declares the memory element in which the variable is being placed (more later).

```
pragma Shared_Access (<Object, <Application, <Access_Rules);
```

- Object - declares the variable being synchronized.
- Process - declares an executable in which the access rules are being defined.
- Access_Rules - declares the rule(s) which apply to access of the Object. These rules may include such items as read-only access.

The `Synchronize` pragma is used to notify the compiler that a data item should be located in a non-standard address location. The compiler will use the declared `Data_Environment`, and other tool information to be discussed later, to place the variable in the correct location. The `Shared_Access` pragma is an addition to the basic data placement, to limit the access to data items depending on the application at hand. This is not intended to provide mutual exclusion to data access at this point, but can be used to create read-only data within on application which is writable in another application, and share common source code. The pragmas can thus be used in a single, shared piece of code which, in turn, can be used in multiple applications.

```
package IO_State is
   IO_Process_State : State_Info;
   pragma Synchronize (IO_Process_State, IO_and_Master_Shared_Memory);
   pragma Shared_Access (IO_Process_State, Main_IO, Author);
   pragma Shared_Access (IO_Process_State, Main_Master, Read_Only);
end IO_State;
```

This implementation is not to imply that shared memory items should be made in packages specification, but is only to provide a simple example of usage. The shared element may actually be in multiple variables or package bodies, etc. They key to this implementation, all of the intent is located in one location. The variables are declared and the notion of alternative storage is coexistent in code, without the exact details of the storage location. The development environment tool set is responsible for determining the context and thus the location of the data structure, not the developer.

Tool Implementation

The pragma make code implementation very simple, but the tools required to support this are not necessarily trivial. The compiler and linker are given the responsibility for a system of executables, memory, devices, and processors and how they are allowed to interact, which is not information traditionally available to the tools. Therefore a method must be created to enable the developer to input the hardware layout and basic software architecture to the compiler/linker. The method of input is implementation dependent, but it seems reasonable to propose a base set of required information from the tool to enable the required functionality of the pragmas.

```
with System;

package Sync_Info is
-- Define potential types for compiler usage in pragma Sync.
-- This information most likely would be multiple packages, but done
-- here in one for presentation simplicity.

   type String_Access is access String;

-------------------------------------------------
   type Executable is tagged record
      Identifier    : String (1..100);
      Start_Address : System.Address;
      Add_Checksum  : Boolean;
      -- This could be extended to allow the user to define their checksum.
   end record;
   type Executable_Access is access Executable;

   type Executables;
   type Executables_Node is access Executables;
   type Executables is record
      Unit : Executable_Access;
      Next : Executables_Node;
   end record;

-------------------------------------------------
   type Memory is tagged record
      Identifier    : String (1..100);
      Start_Address : System.Address;
      End_Address   : System.Address;
   end record;
   type Memory_Access is access Memory;

   type RAM is new Memory with null record;
   type RAM_Access is access RAM;

   type ROM is new Memory with record
      Software : Executables;
   end record;
   type ROM_Access is access ROM;

   type Accessable_Memory;
   type Accessable_Memory_Node is access Accessable_Memory;
   type Accessable_Memory is record
      Unit : Memory_Access;
      Next : Accessable_Memory_Node;
   end record;

   type Accessable_RAM;
   type Accessable_RAM_Node is access Accessable_RAM;
   type Accessable_RAM is record
      Unit : RAM_Access;
      Next : Accessable_RAM_Node;
   end record;

   type Accessable_ROM;
   type Accessable_ROM_Node is access Accessable_ROM;
   type Accessable_ROM is record
      Unit : ROM_Access;
  end record;        Next : Accessable_ROM_Node;

   -- A list of ROM may or may not be sensible, but is not excluded.

--------------------------------------------------
   type Peripheral_Device is record
      Identifier    : String (1..100);
      Start_Address : String_Access; -- System.Address;
   end record;
   type Peripheral_Access is access Peripheral_Device;

   type Accessable_Peripherals;
   type Accessable_Peripheral_Node is access Accessable_Peripherals;
   type Accessable_Peripherals is record
      Unit : Peripheral_Access;
      Next : Accessable_Peripheral_Node;
   end record;

--------------------------------------------------
   type Processor is tagged record
      Identifier            : String (1..100);
      Available_RAM         : Accessable_RAM;
      Available_ROM         : Accessable_ROM;
      Available_Peripherals : Accessable_Peripherals;
   end record;

end Sync_Info;
```

The implementation takes on some fairly simple layers:

- The Executables which are in the system. The builds are identified through a unique textual name and are located in ROM at some given starting address. As an advanced feature, the user may also be given the option of adding a checksum to the end of the executable for purposes of health monitoring.

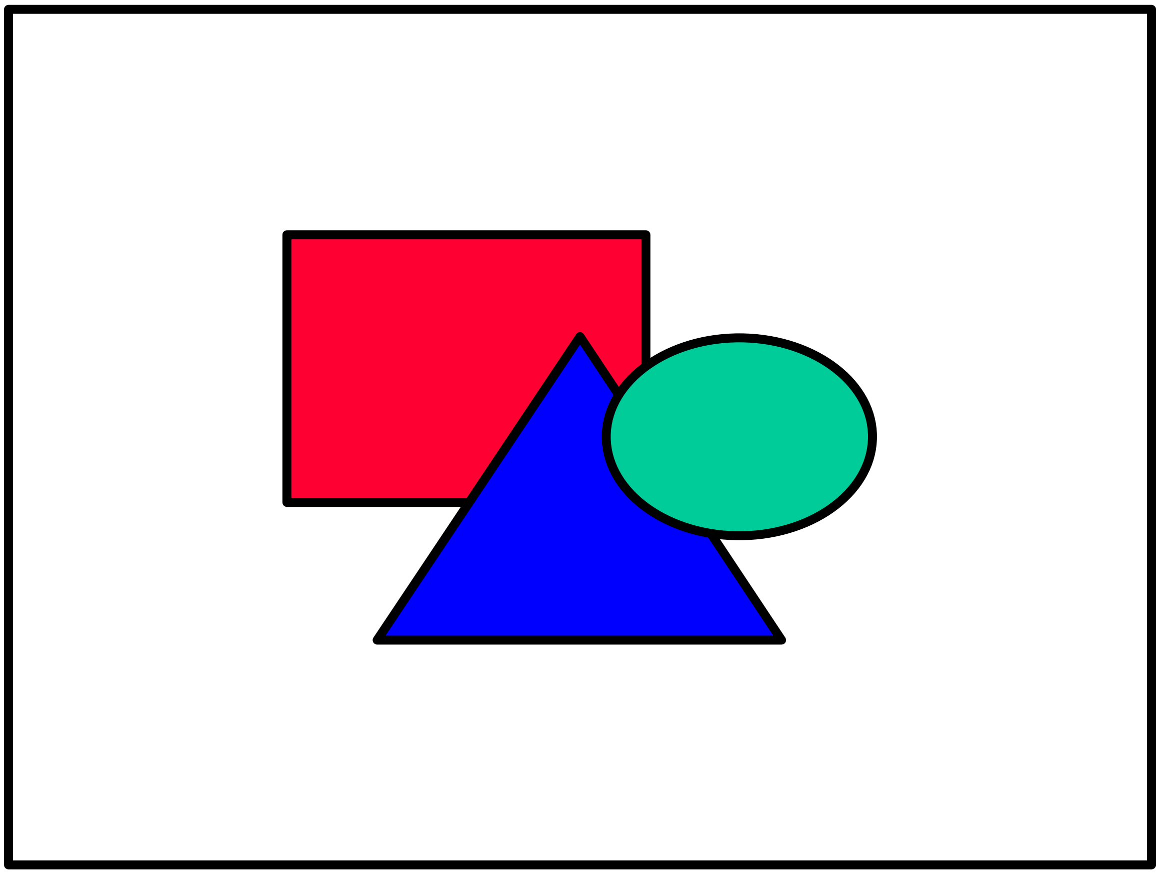
- The Memory devices which are available in the system. Memory units are identified through a unique textual name, and data is provided for the range addresses for that memory device. ROM memory units are extended to include a list of executables which are going to be resident on that ROM device.

- The Peripheral devices which are available in the system. The devices are identified through a unique textual name and a starting address for the device. The assumption is the user will create a data structure to represent the available features (registers, buffers, queues, etc.), and the tool will provide the placement of the data structure at the defined starting location for the device.

- The Processors which are available in the system. The devices are identified through a unique textual name and lists of access to other pieces in the system. The processor has a definition of which devices it has access to, which RAM devices can be written, which ROM devices are readable, and by default, the executables available on the ROM devices.

Given this information, the compiler should not be able to assure that all of the data structures in the series of executables are all synchronized to the same addresses, and assuming the same type is being used, the same representation of data. This could potentially have the side effect of reducing the need for user specification for the representation of data for inter-processor data items when common types are used!

The compiler still has to receive the data from the user to complete implementation however. The data ideally would be entered using some vendor provided GUI to draw the system such as:



The user can recreate the hardware layout in the tool and provide the appropriate information defined above via pop-up boxes. As an alternative a scripting mechanism could be provided (see the following example), but the GUI would provide a very concrete picture of data interactions and system capabilities.

Sample Code Implementation

To aid in the understanding and proposed scope of the change, a small example will be used to show the net effect of pragma Synchronize. Since the functionality being performed is not significant to this discussion, only the data items being synchronized are discussed. The data item is declared followed by the pragma.

```
Activity_Indicator : Integer;
pragma Synchronize (Activity_Indicator, "Health_Area");
```

Even without access to the inner workings of the compiler, the desired effect of `pragma Synchronize` can be simulated by manipulating the source code to replace the pragma with a representation clause. A small program has been provided to search code files and replace any occurrences of `pragma Synchronize` with the appropriate representation clause. Like the compiler, the tool must have access to the desired addresses in order to complete the representation clause. This information is entered by the user through a simple script enough to satisfy a trivial case. The format of the script is as follows:

```
Device : Start = 16#0000#
```

```
   Device : Health_Area = 16#0B0E#
   Device : Other = 16#6DA3_2BAD#
```

For this implementation, only the peripheral device mechanism is being used to provide an absolute address for the given data location. Since the device mechanism is only intend to place a single data structure at a specific item, it is the easiest implementation for this example. The tool reads the script and adds the address to the representation clause. After completing the code is modified to the following:

```
   Activity_Indicator : Integer;
   --     pragma Synchronize (Activity_Indicator, "Health_Area");
   -- The following line is tool generated.  The actual line
   -- is in the comment above.  Do not edit by hand!
   for Activity_Indicator use at 16#0B0E#;
```

This tool is overly simplified and not very flexible (the format of the code and script must be very exact). By implementing the synchronization as a pragma instead of a real code replacement, all of the build information (e.g. data structure sizes) is accessible greatly enhancing the flexibility and capability of data synchronization.

Conclusion

While it is preferable to use more modern approaches of distributed programming, it is undeniable that they are often too costly for implementation in embedded software applications. This does not mean, however, the language can not support a technique for distributed programming that is quick, efficient, and maintainable, even while limited. The use of a pragma enables vendors to selectively implement or ignore the requirements of the low level functionality, depending on the market serviced. The pragma enables those users which require low level programming assistance to create code which is portable between simulation and target environments through the assistance of tool support. The technology seems feasible and can be easily upgraded to more modern distributed programming methods, given the correct software architecture and programming techniques. By extending the available synchronization environment to include 'User defined' data elements could be synchronized through any of the distributed programming techniques, but still maintain the concept of remote data. This pragma provides a helpful step up from the historical techniques for multiprocessor data synchronization, and a possibly a link to modern methods. It is also helpful for handling specialized hardware which will continue regardless of distributed programming techniques. While the primary benefits of the pragma are not as much technical as they are managerial. Saving time by reducing the copies of source code, and enabling easier simulation environments and more reliable portability between simulation and target environments is a huge advantage. If a developer can produce source code which will be identical in simulation and target, and between different partitions, the development and verification costs can be dramatically reduced, as well as cycle times. The pragma seems to be a helpful addition to the language and welcome support for the beleaguered low level programmers.

References

1. Ada 95 Reference Manual. International Standard ANSI/ISO/IEC-8652:1995