Intro to

Embedded Systems Programming

Patrick Rogers

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

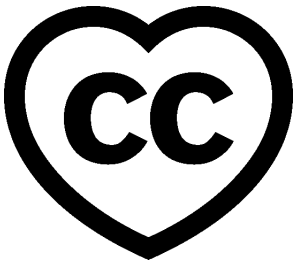
Introduction to Embedded Systems Programming

Release 2025-05

Patrick Rogers

May 31, 2025

CONTENTS:



This course will teach you the basics of the Embedded Systems Programming using Ada.

This document was written by Patrick Rogers, with review by Stephen Baird, Tucker Taft, Filip Gajowniczek, and Gustavo A. Hoffmann.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

[1] http://creativecommons.org/licenses/by-sa/4.0

CHAPTER

ONE

INTRODUCTION

This is a course about embedded systems programming. Embedded systems are everywhere today, including — just to name a few — the thermostats that control a building's temperature, the power-steering controller in modern automobiles, and the control systems in charge of jet engines.

Clearly, much can depend on these systems operating correctly. It might be only a matter of comfort if the thermostat fails. But imagine what might happen if one of the critical control systems in your car failed when you're out on the freeway. When a jet engine controller is designed to have absolute control, it is known as a Full Authority Digital Engine Controller, or FADEC for short. If a FADEC fails, the result can make international news.

Using Ada can help you get it right, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal is to get you started in using it well.

The course is based on the assumption that you know some of the Ada language already, preferably even some of the more advanced concepts. You don't need to know how to use Ada constructs for embedded systems, of course, but you do need to know at least the language basics. If you need that introduction, see the course Introduction to Ada.

We also assume that you already have some programming experience so we won't cover CS-101.

Ideally, you also have some experience with low-level programming, because we will focus on "how to do it in Ada." If you do, feel free to gloss over the introductory material. If not, don't worry. We will cover enough for the course to be of value in any case.

1.1 So, what will we actually cover?

We will introduce you to using Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data, e.g., data from sensors that are occasionally faulty.

We will discuss development using more than Ada alone, nowadays a quite common approach. Specifically, how to interface with code and data written in other languages, and how (and why) to work with assembly language.

Embedded systems interact with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. Frequently these devices are mapped into the target memory address space. We will cover how to define and interact with these memory-mapped devices.

Finally, we will show how to handle interrupts in Ada, using portable constructs.

1.2 Definitions

Before we go any further, what do we mean by "embedded system" anyway? It's time to be specific. We're talking about a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system: the enclosing thermostat controlling the temperature, the power steering controller in the enclosing automobile, and the FADEC embedded in the enclosing aircraft. So these are not stand-alone computers for general purpose application execution.

As such, embedded systems typically have reduced resources available, especially power, which means reduced processor speed and reduced memory on-board. For an example at the small end of the spectrum, consider the computer embedded in a wearable device: it must run for a long time on a very little battery, with comparatively little memory available. But that's often true of bigger systems too, such as systems on aircraft where power (and heat) are directly limiting factors.

As a result, developing embedded systems software can be more difficult than general application development, not to mention that this software is potentially safety-critical.

Ada is known for use in very large, very long-lived projects (e.g., deployed for decades), but it can also be used for very small systems with tight resource constraints. We'll show you how.

We used the term "computer" above. You already know what that means, but you may be thinking of your laptop or something like that, where the processor, memory, and devices are all distinct, separate components. That can be the case for embedded systems too, albeit in a different form-factor such as rack-mounted boards. However, be sure to expand your definition to include the notion of a system-on-chip (SoC), in which the processor, memory, and various useful devices are all on a single chip. Embedded systems don't necessarily involve SoC computers but they frequently do. The techniques and information in this course work on any of these kinds of computer.

1.3 Down To The Bare Metal

Ada has always had facilities designed specifically for embedded systems. The language includes constructs for directly manipulating hardware, for example, and direct interaction with assembly language. These constructs are as effective as those of any high-level programming language (yes, including C). These constructs are expressively powerful, well-specified (so there are few surprises), efficient, and portable (within reason).

We say "within reason" because portability is a difficult goal for embedded systems. That's because the hardware is so much a part of the application itself, rather than being abstracted away as in a general-purpose application. That said, the hardware details can be managed in Ada so that portability is maximized to the extent possible for the application.

But strictly speaking, not all software can or should be absolutely portable! If a specific device is required, well, the program won't work with some other device. But to the extent possible portability is obviously a good thing.

1.4 The Ada Drivers Library

Speaking of SoC computers, there is a library of freely-available device drivers in Ada. Known as the Ada Driver Library (ADL), it supports many devices on a number of vendors' products. Device drivers for timers, I2C, SPI, A/D and D/A converters, DMA, General Purpose I/O, LCD displays, sensors, and other devices are included. The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.

An extensive description of a project using the ADL is available here: https://blog.adacore.com/making-an-rc-car-with-ada-and-spark

We will refer to components of this library and use some of them as examples.

CHAPTER

TWO

LOW LEVEL PROGRAMMING

This section introduces a number of topics in low-level programming, in which the hardware and the compiler's representation choices are much more in view at the source code level. In comparatively high level code these topics are "abstracted away" in that the programmer can assume that the compiler does whatever is necessary on the current target machine so that their code executes as intended. That approach is not sufficient in low-level programming.

Note that we do not cover every possibility or language feature. Instead, we cover the necessary concepts, and also potential surprises or pitfalls, so that the parts not covered can be learned on your own.

2.1 Separation Principle

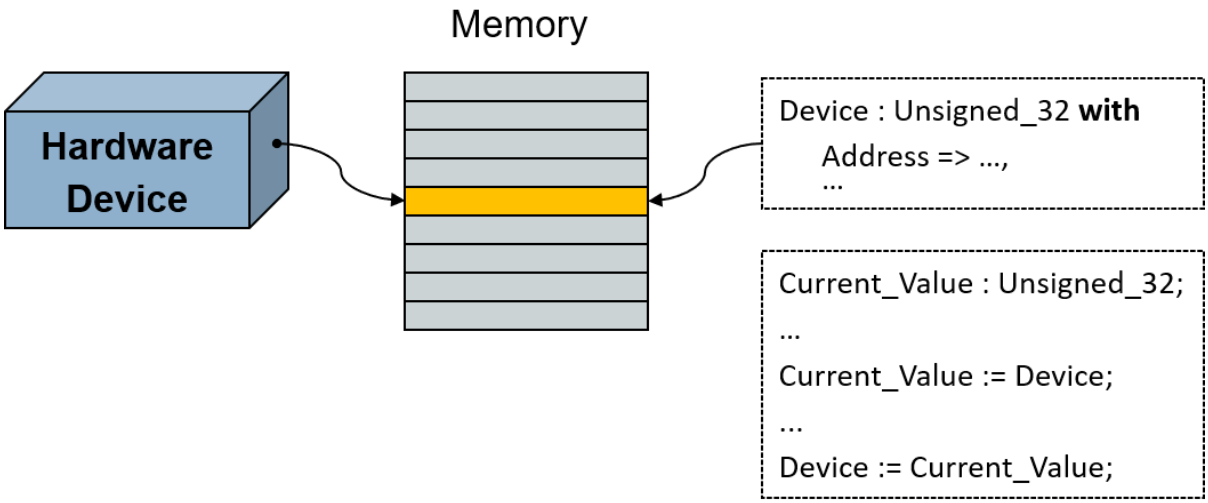
There is a language design principle underlying the Ada facilities intended for implementing embedded software. This design principle directly affects how the language is used, and therefore, the portability and readability of the resulting application code.

This language design principle is known as the "separation principle." What's being separated? The low-level, less portable aspects of some piece of code are separated from the usage of that piece of code.

Don't confuse this with hiding unnecessary implementation details via compile-time visibility control (i.e., information hiding and encapsulation). That certainly should be done too. Instead, because of the separation principle, we specify the low-level properties of something once, when we declare it. From then on, we can use regular Ada code to interact with it. That way the bulk of the code — the usage — is like any other Ada code, and doesn't propagate the low-level details all over the client code. This greatly simplifies usage and understandability as well as easing porting to new hardware-specific aspects. You change things in one place, rather than everywhere.

For example, consider a device mapped to the memory address space of the processor. To interact with the device we interact with one or more memory cells. Reading input from the device amounts to reading the value at the associated memory location. Likewise, sending output to the device amounts to writing to that location.

To represent this device mapping we declare a variable of an appropriate type and specify the starting address the object should occupy. (There are other ways too, but for a single, statically mapped object this is the simplest approach.) We'd want to specify some other characteristics as well, but let's focus on the address.



If the hardware presents an interface consisting of multiple fields within individual memory cells, we can use a record type instead of a single unsigned type representing a single word. Ada allows us to specify the exact record layout, down to the individual bit level, for any types we may need to use for the record components. When we declare the object we use that record type, again specifying the starting address. Then we can just refer to the object's record components as usual, having the compiler compute the address offsets required to access the components representing the individual hardware fields.

Note that we aren't saying that other languages cannot do this too. Many can, using good programming practices. What we're saying is that those practices are designed into the Ada way of doing it.

2.2 Guaranteed Level of Support

The Ada reference manual has an entire section dedicated to low-level programming. That's section 13, "Representation Issues," which provides facilities for developers to query and control aspects of various entities in their code, and for interfacing to hardware. Want to specify the exact layout for a record type's components? Easy, and the compiler will check your layout too. Want to specify the alignment of a type? That's easy too. And that's just the beginning. We'll talk about these facilities as we go, but there's another point to make about this section.

In particular, section 13 includes recommended levels of support to be provided by language implementations, i.e., compilers and other associated tools. Although the word "recommended" is used, the recommendations are meant to be followed.

For example, section 13.3 says that, for some entity named X, "X'`Address` should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose **`Address`** has been specified." So, for example, if the programmer specifies the address for a memory-mapped variable, the compiler cannot ignore that specification and instead, for the sake of performance, represent that variable using a register. The object must be represented as an addressable entity, as requested by the programmer. (Registers are not addressable.)

We mention this because, although the recommended levels of support are intended to be followed, those recommendations become **requirements** if the Systems Programming (SP) Annex is implemented by the vendor. In that case the vendor's implementation of section 13 must support at least the recommended levels. The SP Annex defines additional, optional functionality oriented toward this programming domain; you want it anyway. (Like all the annexes it adds no new syntax.) Almost all vendors, if not literally all, implement the Annex so you can rely on the recommended levels of support.

2.3 Querying Implementation Limits and Characteristics

Sometimes you need to know more about the underlying machine than is typical for general purpose applications. For example, your numerical analysis algorithm might need to know the maximum number of digits of precision that a floating-point number can have on this specific machine. For networking code, you will need to know the "endianness" of the machine so you can know whether to swap the bytes in an Ethernet packet. You'd go look in the `limits.h` file in C implementations, but in Ada we go to a package named `System` to get this information.

Clearly, these implementation values will vary with the hardware, so the package declares constants with implementation-defined values. The names of the constants are what's portable, you can count on them being the same in any Ada implementation.

However, vendors can add implementation-defined declarations to the language-defined content in package System. You might require some of those additions, but portability could then suffer when moving to a new vendor's compiler. Try not to use them unless it is unavoidable. Ideally these additions will appear in the private part of the package, so the implementation can use them but application code cannot.

For examples of the useful, language-defined constants, here are those for the numeric limits of an Ada compiler for an Arm 32-bit SoC:

```
Min_Int               : constant := Long_Long_Integer'First;
Max_Int               : constant := Long_Long_Integer'Last;

Max_Binary_Modulus    : constant := 2 ** Long_Long_Integer'Size;
Max_Nonbinary_Modulus : constant := 2 ** Integer'Size - 1;

Max_Base_Digits       : constant := Long_Long_Float'Digits;
Max_Digits            : constant := Long_Long_Float'Digits;

Max_Mantissa          : constant := 63;
Fine_Delta            : constant := 2.0 ** (-Max_Mantissa);
```

`Min_Int` and `Max_Int` supply the most-negative and most-positive integer values supported by the machine.

`Max_Binary_Modulus` is the largest power of two allowed as the modulus of a modular type definition.

But a modular type need not be defined in terms of powers of two. An arbitrary modulus is allowed, as long as it is not bigger than the machine can handle. That's specified by `Max_Nonbinary_Modulus`, the largest non-power-of-two value allowed as the modulus of a modular type definition.

`Max_Base_Digits` is the largest value allowed for the requested decimal precision in a floating-point type's definition.

We won't go over all of the above, you get the idea. Let's examine the more important contents.

Two of the most frequently referenced constants in `System` are the following, especially the first. (The values here are again for the Arm 32-bit SoC):

```
Storage_Unit : constant := 8;
Word_Size    : constant := 32;
```

`Storage_Unit` is the number of bits per memory storage element. Storage elements are the components of memory cells, and typically correspond to the individually addressable memory elements. A "byte" would correspond to a storage element with the above constant value.

Consider a typical idiom for determining the number of whole storage elements an object named X occupies:

```
Units : constant Integer := (X'Size + Storage_Unit - 1) / Storage_Unit;
```

Remember that 'Size returns a value in terms of bits. There are more direct ways to determine that size information but this will serve as an example of the sort of thing you might do with that constant.

A machine "word" is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of some number of storage elements, maybe one but typically more than one. As the unit the machine natively manipulates, words are expected to be independently addressable. (On some machines only words are independently addressable.)

Word_Size is the number of bits in the machine word. On a 32-bit machine we'd expect Word_Size to have a value of 32; on a 64-bit machine it would probably be 64, and so on.

Storage_Unit and Word_Size are obviously related.

Another frequently referenced declaration in package System is that of the type representing memory addresses, along with a constant for the null address designating no storage element.

```
type Address is private;
Null_Address : constant Address;
```

You may be wondering why type **Address** is a private type, since that choice means that we programmers cannot treat it like an ordinary (unsigned) integer value. Portability is of course the issue, because addressing, and thus address representation, varies among computer architectures. Not all architectures have a flat address space directly referenced by numeric values, although that is common. Some are represented by a base address plus an offset, for example. Therefore, the representation for type **Address** is hidden from us, the clients. Consequently we cannot simply treat address values as numeric values. Don't worry, though. The operations we need are provided.

Package *System* declares these comparison functions, for example:

```
function "<"  (Left, Right : Address) return Boolean;
function "<=" (Left, Right : Address) return Boolean;
function ">"  (Left, Right : Address) return Boolean;
function ">=" (Left, Right : Address) return Boolean;
function "="  (Left, Right : Address) return Boolean;
```

These functions are intrinsic, i.e., built-in, meaning that the compiler generates the code for them directly at the point of calls. There is no actual function body for any of them so there is no performance penalty.

Any private type directly supports the equality function, and consequently the inequality function, as well as assignment. What we don't get here is address arithmetic, again because we don't have a compile-time view of the actual representation. That functionality is provided by package System.Storage_Elements, a child package we will cover later. We should say though, that the need for address arithmetic in Ada is rare, especially compared to C.

Having type **Address** presented as a private type is not, strictly speaking, required by the language. Doing so is a good idea for the reasons given above, and is common among vendors. Not all vendors do, though.

Note that **Address** is the type of the result of the query attribute **Address**.

We mentioned potentially needing to swap bytes in networking communications software, due to the differences in the "endianness" of the machines communicating. That characteristic can be determined via a constant declared in package System as follows:

```
type Bit_Order is (High_Order_First, Low_Order_First);
Default_Bit_Order : constant Bit_Order := implementation-defined;
```

`High_Order_First` corresponds to "Big Endian" and `Low_Order_First` to "Little Endian." On a Big Endian machine, bit 0 is the most significant bit. On a Little Endian machine, bit 0 is the least significant bit.

Strictly speaking, this constant gives us the default order for bits within storage elements in record representation clauses, not the order of bytes within words. However, we can usually use it for the byte order too. In particular, if `Word_Size` is greater than `Storage_Unit`, a word necessarily consists of multiple storage elements, so the default bit ordering is the same as the ordering of storage elements in a word.

Let's take that example of swapping the bytes in a received Ethernet packet. The "wire" format is Big Endian so if we are running on a Little Endian machine we must swap the bytes received.

Suppose we want to retrieve typed values from a given buffer or bytes. We get the bytes from the buffer into a variable named `Value`, of the type of interest, and then swap those bytes within `Value` if necessary.

```
   ...
begin
   Value := ...

   if Default_Bit_Order /= High_Order_First then
      -- we're not on a Big Endian machine
      Value := Byte_Swapped (Value);
   end if;
end Retrieve_4_Bytes;
```

We have elided the code that gets the bytes into `Value`, for the sake of simplicity. How the bytes are actually swapped by function `Byte_Swapped` is also irrelevant. The point here is the if-statement: the expression compares the `Default_Bit_Order` constant to `High_Order_First` to see if this execution is on a Big Endian machine. If not, it swaps the bytes because the incoming bytes are always received in "wire-order," i.e., Big Endian order.

Another important set of declarations in package `System` define the values for priorities, including interrupt priorities. We will ignore them until we get to the section on interrupt handling.

Finally, and perhaps surprisingly, a few declarations in package `System` are almost always (if not actually always) ignored.

```
type Name is implementation-defined-enumeration-type;
System_Name : constant Name := implementation-defined;
```

Values of type `Name` are the names of alternative machine configurations supported by the implementation. `System_Name` represents the current machine configuration. We've never seen any actual use of this.

`Memory_Size` is an implementation-defined value that is intended to reflect the memory size of the configuration, in units of storage elements. What the value actually refers to is not specified. Is it the size of the address space, i.e., the amount possible, or is it the amount of physical memory actually on the machine, or what? In any case, the amount of memory available to a given computer is neither dependent upon, nor reflected by, this constant. Consequently, `Memory_Size` is not useful either.

Why have something defined in the language that nobody uses? In short, it seemed like a good idea at the time when Ada was first defined. Upward-compatibility concerns propagate these declarations forward as the language evolves, just in case somebody does use them.

2.4 Querying Representation Choices

As we mentioned in the introduction, in low-level programming the hardware and the compiler's representation choices can come to the forefront. You can, therefore, query many such choices.

For example, let's say we want to query the addresses of some objects because we are calling the imported C `memcpy` function. That function requires two addresses to be passed to the call: one for the source, and one for the destination. We can use the `'Address` attribute to get those values.

We will explore importing routines and objects implemented in other languages elsewhere. For now, just understand that we will have an Ada declaration for the imported routine that tells the compiler how it should be called. Let's assume we have an Ada function declared like so:

```
function MemCopy
  (Destination : System.Address;
   Source      : System.Address;
   Length      : Natural)
return Address
with
  Import,
  Convention => C,
  Link_Name => "memcpy",
  Pre  => Source /= Null_Address      and then
          Destination /= Null_Address and then
          not Overlapping (Destination, Source, Length),
  Post => MemCopy'Result = Destination;
--  Copies Length bytes from the object designated by Source to the object
--  designated by Destination.
```

The three aspects that do the importing are specified after the reserved word **`with`** but can be ignored for this discussion. We'll talk about them later. The preconditions make explicit the otherwise implicit requirements for the arguments passed to `memcpy`, and the postcondition specifies the expected result returned from a successful call. Neither the preconditions nor the postconditions are required for importing external entities but they are good "guard-rails" for using those entities. If we call it incorrectly the precondition will inform us, and likewise, if we misunderstand the result the postcondition will let us know (at least to the extent that the return value does that).

For a sample call to our imported routine, imagine that we have a procedure that copies the bytes of a **`String`** parameter into a `Buffer` parameter, which is just a contiguous array of bytes. We need to tell `MemCopy` the addresses of the arguments passed so we apply the `'Address` attribute accordingly:

```
procedure Put (This : in out Buffer; Start : Index; Value : String) is
   Result : System.Address with Unreferenced;
begin
   Result := MemCopy (Destination => This (Start)'Address,
                      Source      => Value'Address,
                      Length      => Value'Length);
end Put;
```

The order of the address parameters is easily confused so we use the named association format for specifying the actual parameters in the call.

Although we assign `Result` we don't otherwise use it, so we tell the compiler this is not a mistake via the `Unreferenced` aspect. And if we do turn around and reference it the compiler will complain, as it should. Note that `Unreferenced` is defined by GNAT, so usage is not necessarily portable. Other vendors may or may not implement something like it, perhaps with a different name.

(We don't show the preconditions for Put, but they would have specified that Start must be a valid index into this particular buffer, and that there must be room in the Buffer argument for the number of bytes in Value when starting at the Start index, so that we don't copy past the end of the Buffer argument.)

There are other characteristics we might want to query too.

We might want to ask the compiler what alignment it chose for a given object (or type, for all such objects).

For a type, when Alignment returns a non-zero value we can be sure that the compiler will allocate storage for objects of the type at correspondingly aligned addresses (unless we force it to do otherwise). Similarly, references to dynamically allocated objects of the type will be to properly aligned locations. Otherwise, an Alignment of zero means that the guarantee does not hold. That could happen if the type is packed down into a composite object, such as an array of Booleans. We'll discuss "packing" soon. More commonly, the smallest likely value is 1, meaning that any storage element's address will suffice. If the machine has no particular natural alignments, then all type alignments will probably be 1 by default. That would be somewhat rare today, though, because modern processors usually have comparatively strict alignment requirements.

We can ask for the amount of storage associated with various entities. For example, when applied to a task, 'Storage_Size tells us the number of storage elements reserved for the task's execution. The value includes the size of the task's stack, if it has one. We aren't told if other required storage, used internally in the implementation, is also included in this number. Often that other storage is not included in this number, but it could be.

Storage_Size is also defined for access types. The meaning is a little complicated. Access types can be classified into those that designate only variables and constants ("access-to-object") and those that can designate subprograms. Each access-to-object type has an associated storage pool. The storage allocated by **new** comes from the pool, and instances of Unchecked_Deallocation return storage to the pool.

When applied to an access-to-object type, Storage_Size gives us the number of storage elements reserved for the corresponding pool.

Note that Storage_Size doesn't tell us how much available, unallocated space remains in a pool. It includes both allocated and unallocated space. Note, too, that although each access-to-object type has an associated pool, that doesn't mean that each one has a distinct, dedicated pool. They might all share one, by default. On an operating system, such as Linux, the default shared pool might even be implicit, consisting merely of calls to the OS routines in C.

As a result, querying Storage_Size for access types and tasks is not necessarily all that useful. Specifying the sizes, on the other hand, definitely can be useful.

That said, we can create our own pool types and define precisely how they are sized and how allocation and deallocation work, so in that case querying the size for access types could be more useful.

For an array type or object, 'Component_Size provides the size in bits of the individual components.

More useful are the following two attributes that query a degree of memory sharing between objects.

Applied to an object, 'Has_Same_Storage is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations occupy exactly the same bits.

Applied to an object, 'Overlaps_Storage is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations share at least one bit.

Generally, though, we specify representation characteristics far more often than we query them. Rather than describe all the possibilities, we can just say that all the representation characteristics that can be specified can also be queried. We cover specifying representation characteristics next, so just assume the corresponding queries are available.

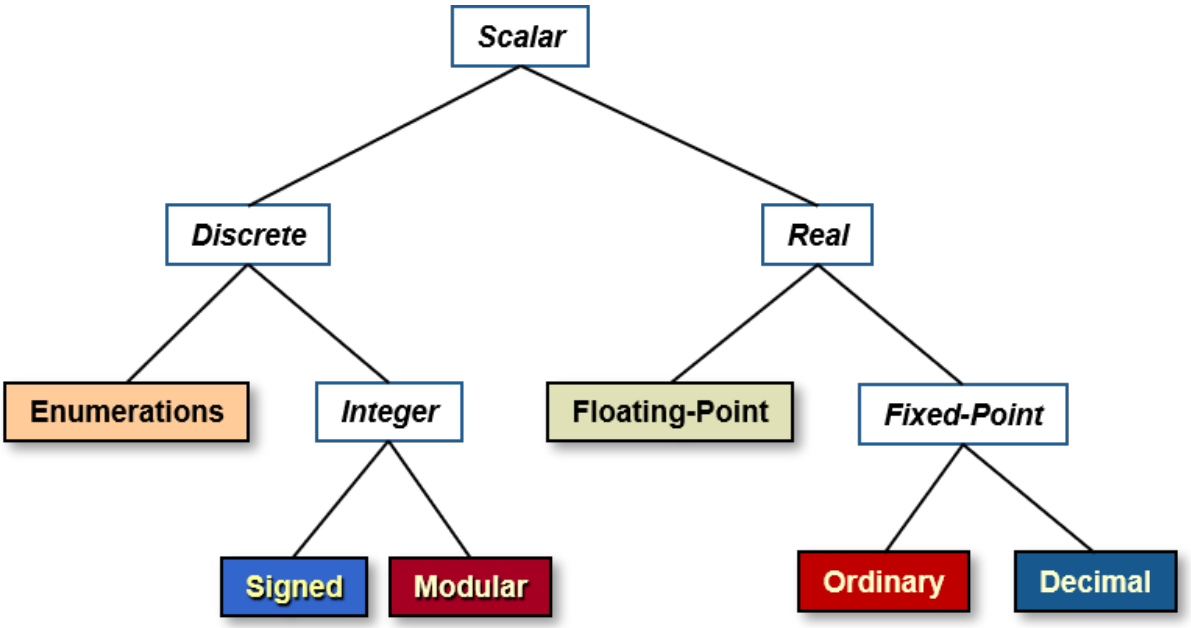
That said, there is one particular representation query we need to talk about explicitly, now, because there is a lot of confusion about it: the `'Size` attribute. The confusion stems from the fact that there are multiple contexts for applying the attribute, and multiple reasonable interpretations possible. We can apply the `'Size` attribute to a type, in an attempt to get information about all objects of the type, or we can apply it to individual objects to get specific information. In both cases, what actual information do we get? In the original version of Ada these questions weren't really answered so vendors did what they thought was correct. But they did not agree with each other, and portability became a problem.

For example, suppose you want to convert some value to a series of bytes in order to send the value over the wire. To do that you need to know how many bytes are required to represent the value. Many applications queried the size of the type to determine that, and then, when porting to a new vendor's compiler, found that their code no longer worked correctly. The new vendor's implementation wasn't wrong, it was just different.

Later versions of Ada answered these questions, where possible, so let's examine the contexts and meaning. Above all, though, remember that `'Size` returns values in terms of **bits**.

If we apply `'Size` to a type, the resulting value depends on the kind of type.

For scalar types, the attribute returns the *minimum* number of bits required to represent all the values of the type. Here's a diagram showing what the category "scalar types" includes:



Consider type `Boolean`, which has two possible values. One bit will suffice, and indeed the language standard requires `Boolean'Size` to be the value 1.

This meaning also applies to subtypes, which can constrain the number of values for a scalar type. Consider subtype `Natural`. That's a subtype defined by the language to be type `Integer` but with a range of `0 .. Integer'Last`. On a 32-bit machine we would expect `Integer` to be a native type, and thus 32-bits. On such a machine if we say `Integer'Size` we will indeed get 32. But if we say `Natural'Size` we will get 31, not 32, because only 31 bits are needed to represent that range on that machine.

The size of objects, on the other hand, cannot be just a matter of the possible values. Consider type `Boolean` again, where `Boolean'Size` is required to be 1. No compiler is likely to

allocate one bit to a **Boolean** variable, because typical machines don't support individually-addressable bits. Instead, addresses refer to storage elements, of a size indicated by the `Storage_Unit` constant. The compiler will allocate the smallest number of storage elements necessary, consistent with other considerations such as alignment. Therefore, for a machine that has `Storage_Unit` set to a value of eight, we can assume that a compiler for that machine will allocate an entire eight-bit storage element to a stand-alone **Boolean** variable. The other seven bits are simply not used by that variable. Moreover, those seven bits are not used by any other stand-alone object either, because access would be far less efficient, and such sharing would require some kind of locking to prevent tasks from interfering with each other when accessing those stand-alone objects. (Stand-alone objects are independently addressable; they wouldn't stand alone otherwise.)

By the same token (and still assuming a 32-bit machine), a compiler will allocate more than 31 bits to a variable of subtype Natural because there is no 31-bit addressable unit. The variable will get all 32-bits.

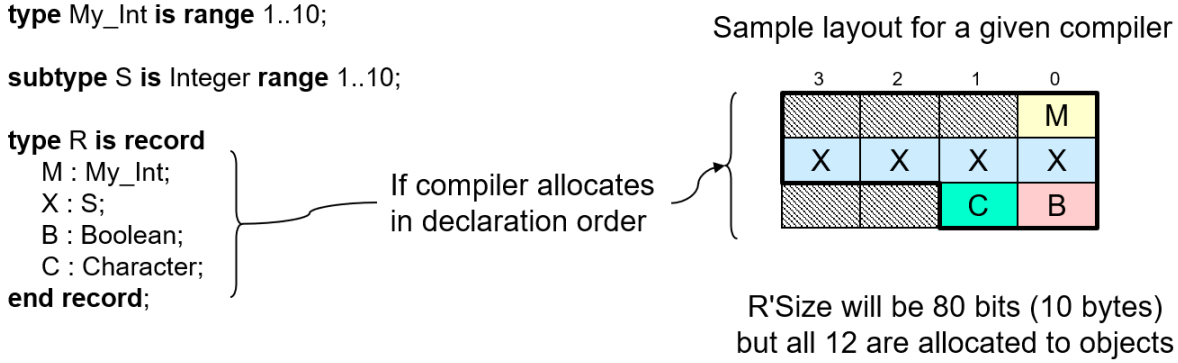
Note that we're talking about individual, stand-alone variables. Components of composite types, on the other hand, might indeed share bytes if the individual components don't require all the bits of their storage elements. You'd have to request that representation, though, with most implementations, because accessing the components at run-time would require more machine instructions. We'll go into the details of that later.

Let's talk further about sizes of types.

For record types, `'Size` gives the minimum number of bits required to represent the whole composite value. But again, that's not necessarily the number of bits required for the objects' in-memory representation. The order of the components within the record can make a difference, as well as their alignments. The compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. As a result the overall size could be larger.

Note that Ada compilers are allowed to reorder the components; the order in memory might not match the order in the source code.

For example, consider this record type and its components:



In the figure, we see a record type with some components, and a sample layout for that record type assuming the compiler does not reorder the components. Observe that some bytes allocated to objects of type R are unused (the darkly shaded ones). In this case that's because the alignment of subtype S happens to be 4 on this machine. The component X of that subtype S cannot start at byte offset 1, or 2, or 3, because those addresses would not satisfy the alignment constraint of S. (We're assuming byte 0 is at a word-aligned address.) Therefore, X starts at the object's starting address plus 4. Components B and C are of types that have an alignment of 1, so they can start at any storage element. They immediately follow the bytes allocated to component X. Therefore, `R'Size` is 80, or 10 bytes. The three bytes following component `M` are simply not used.

But what about the two bytes following the last component `C`? They could be allocated to

stand-alone objects if they would fit. More likely, though, the compiler will allocate those two bytes to objects of type R, that is, 12 bytes instead of 10 are allocated. As a result, 96 bits are actually used in memory. The extra, unused 16 bits are "padding."

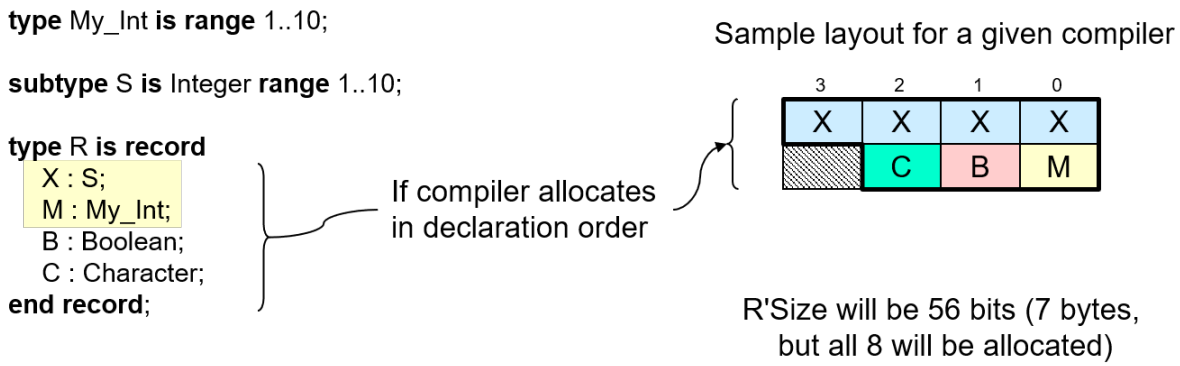
Why add unused padding? It simplifies the memory allocation of objects of type R. Suppose some array type has components of record type R. Assuming the first component is aligned properly, every following component will also be aligned properly, automatically, because the two padding bytes are considered parts of the components.

To make that work, the compiler takes the most stringent alignment of all the record type's components and uses that for the alignment of the overall record type. That way, any address that satisfies the record object's alignment will satisfy the components' alignment requirements. The alignment is component X, of subtype S, is 4. The other components have an alignment of 1, therefore R'`Alignment` is 4. An aligned address plus 12 will also be an aligned address.

This rounding up based on alignment is recommended behavior for the compiler, not a requirement, but is reasonable and typical among vendors. Although it can result in unused storage, that's the price paid for speed of access (or even correctness for machines that would fault on misaligned component accesses).

As you can see, alignment is a critical factor in the sizes of composite objects. If you care about the layout of the type you very likely need to care about the alignment of the components and overall record type.

Ada compilers are allowed to reorder the components of record types in order to minimize these gaps or satisfy the alignment requirements of the components. Some compilers do, some don't. Consider the type R again, this time with the first two components switched in the component declaration order:



Now R'`Size` will report 56 bits instead of 80. The one trailing byte will still be padding, but only that one.

What about unbounded types, for example type **`String`**? Querying the '`Size` in that case would provide an implementation-defined result. A somewhat silly thing to do, really, since the type — by definition — doesn't specify how many components are involved.

Usually, though, you don't want to query the size of a type. Most of the time what you want is the size of objects of the type. Going back to sending values over the wire, the code should query the size of the *parameter* holding the value to be sent. That will tell you how many bits are really needed.

One last point: GNAT, and now Ada 202x, define an attribute named `Object_Size`. It does just what the name suggests: what '`Size` does when applied to objects rather than types. GNAT also defines another attribute, named `Value_Size`, that does what '`Size` does when applied to types. The former is far more useful so Ada has standardized it.

2.5 Specifying Representation

Recall that we said `Boolean'Size` is required to be 1, and that stand-alone objects of type `Boolean` are very likely allocated some integral number of storage elements (e.g., bytes) in memory, typically one. What about arrays of Booleans? Suppose we have an array of 16 Boolean components. How big are objects of the array type? It depends on the machine. Continuing with our hypothetical (but typical) byte-addressable machine, for the sake of efficient access each component is almost certainly allocated an individual byte rather than a single bit, just like stand-alone objects. Consequently, our array of 16 Booleans will be reported by `'Size` to be 128 bits, i.e., 16 bytes. If you wanted a bit-mask, in which each Boolean component is allocated a single bit and the total array size is 16 bits, you'd have a problem. The compiler assumes you want speed of access rather than storage minimization, and normally that would be the right assumption.

Naturally there is a solution. Ada allows us to specify the representation characteristics of types, and thus objects of those types, including their bit-wise layouts. It also allows us to specify the representation of individual objects. You should understand, though, that the compiler is not required to do what you ask, because you might ask the impossible. For example, if you specify that the array of 16 Booleans is to be represented completely in 15 bits, what can the compiler do? Rejecting that specification is the only reasonable response. But if you specify something possible, the compiler must do what you ask, absent some compelling reason to the contrary.

With that in mind, let's examine setting the size for types.

So, how do we specify that we want our array of 16 Boolean components to be allocated one bit per component, for a total allocation of 16 bits? There are a couple of ways, one somewhat better than the other.

First, you can ask that the compiler "pack" the components into as small a number of bits as it can:

```
type Bits16 is array (0 .. 15) of Boolean with
  Pack;
```

That likely does what you want: `Bits16'Size` will probably be 16.

But realize that the `Pack` aspect (and corresponding pragma) is merely a request that the compiler do its best to minimize the number of bits allocated, not necessarily that it do exactly what you expected or required.

We could set the size of the entire array type:

```
type Bits16 is array (0 .. 15) of Boolean with
  Size => 16;
```

But the language standard says that a `Size` clause on array and record types should not affect the internal layout of their components. That's Implementation Advice, so not normative, but implementations are really expected to follow the advice, absent some compelling reason. That's what the `Pack` aspect, record representation clauses, and `Component_Size` clauses are for. (We'll talk about record representation clauses momentarily.) That said, at least one other vendor's compiler would have changed the size of the array type because of the `Size` clause, so GNAT defines a configuration pragma named `Implicit_Packing` that overrides the default behavior. With that pragma applied, the `Size` clause would compile and suffice to make the overall size be 16. That's a vendor-defined pragma though, so not portable.

Therefore, the best way to set the size for the array type is to set the size of the individual components, via the `Component_Size` aspect as the Implementation Advice indicates. That will say what we really want, rather than a "best effort" request for the compiler, and is portable:

```
type Bits16 is array (0 .. 15) of Boolean with
  Component_Size => 1;
```

With this approach the compiler must either use the specified size for each component or refuse to compile the code. If it compiles, objects of the array type will be 16 bits total (plus any padding bits required to make objects have a size that is a multiple of Storage_Unit, typically zero on modern machines).

Now that we have a bit-mask array type, let's put it to use.

Let's say that you have an object that is represented as a simple signed integer because, for most usage, that's the appropriate representation. Sometimes, though, let's say you need to access individual bits of the object instead of the whole numeric value. Signed integer types don't provide bit-level access. In Ada we'd say that the "view" presented by the object's type doesn't include bit-oriented operations. Therefore, we need to add a view to the object that does provide them. A different view will require an additional type for the same object.

Applying different types, and thus their operations, to the same object is known as type punning[2] in computer programming. Realize that doing so circumvents the static strong typing we harness to protect us from ourselves and from others. Use it with care! (For example, limit the compile-time visibility to such code.)

One way to add a view is to express an "overlay," in which an object of one type is placed at the same memory location as a distinct object of a different type, thus "overlaying" one object over the other in memory. The different types present different views, therefore different operations available for the shared memory cells. Our hypothetical example uses two views, but you can overlay as many different views as needed. (That said, requiring a large number of different views of the same object would be suspect.)

There are other ways in Ada to apply different views, some more flexible than others, but an overlay is a simple one that will often suffice.

Here is an implementation of the overlay approach, using our bit-mask array type:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

X : Integer;
Y : Bits32 with Address => X'Address;
```

We can query the addresses of objects, and other things too, but objects, especially variables, are the most common case. In the above, we say `X'Address` to query the starting address of object X. With that information we know what address to specify for our bit-mask overlay object Y. Now X and Y are aliases for the same memory cells, and therefore we can manipulate and query that memory as either a signed integer or as an array of bits. Reading or updating individual array components accesses the individual bits of the overlaid object.

Instead of the `Bits32` array type, we could have specified a modular type for the overlay Y to get a view providing bit-oriented operations. Overlaying such an array was a common idiom prior to the introduction of modular "unsigned" types in Ada, and remains useful for accessing individual bits. In other words, using a modular type for Y, you could indeed access an individual bit by passing a mask value to the **and** operator defined in any modular type's view. Using a bit array representation lets the compiler do that work for you, in the generated code. The source code will be both easier to read and more explicit about what it is doing when using the bit array overlay.

One final issue remains: in our specific overlay example the compiler would likely generate code that works. But strictly speaking it might not.

[2] https://en.wikipedia.org/wiki/Type_punning

The Ada language rules say that for such an overlaid object — Y in the example above — the compiler should not perform optimizations regarding Y that it would otherwise apply in the absence of aliases. That's necessary, functionally, but may imply degraded performance regarding Y, so keep it in mind. Aliasing precludes some desirable optimizations.

But what about X in the example above? We're querying that object's address, not specifying it, so the RM rule precluding optimizations doesn't apply to X. That can be problematic.

The compiler might very well place X in a register, for example, for the sake of the significant performance increase (another way of being friendly). But in that case `System.Null_Address` will be returned by the `X'Address` query and, consequently, the declaration for Y will not result in the desired overlaying.

Therefore, we should mark X as explicitly **`aliased`** to ensure that `X'Address` is well-defined:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

The only difference in the version above is the addition of **`aliased`** in the declaration of X. Now we can be certain that the optimizer will not represent X in some way incompatible with the idiom, and `X'Address` will be well-defined.

In our example X and Y are clearly declared in the same compilation unit. Most compilers will be friendly in this scenario, representing X in such a way that querying the address will return a non-null address value even if **`aliased`** is not applied. Indeed, **`aliased`** is relatively new to Ada, and earlier compilers typically emitted code that would handle the overlay as intended.

But suppose, instead of being declared in the same declarative part, that X was declared in some other compilation unit. Let's say it is in the visible part of a package declaration. (Assume X is visible to clients for some good reason.) That package declaration can be, and usually will be, compiled independently of clients, with the result that X might be represented in some way that cannot supporting querying the address meaningfully.

Therefore, the declaration of X in the package spec should be marked as aliased, explicitly:

```
package P is
   X : aliased Integer;
end P;
```

Then, in the client code declaring the overlay, we only declare Y, assuming a with-clause for P:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

Y : Bits32 with Address => P.X'Address;
```

All well and good, but how did the developer of the package know that some other unit, a client of the package, would query the address of X, such that it needed to be marked as aliased? Indeed, the package developer might not know. Yet the programmer is responsible for ensuring a valid and appropriate **`Address`** value is used in the declaration of Y. Execution is erroneous otherwise, so we can't say what would happen in that case. Maybe an exception is raised or a machine trap, maybe not.

Worse, the switches that were applied when compiling the spec for package P can make a difference: `P.X` might not be placed in a register unless the optimizer is enabled. Hence the client code using Y might work as expected when built for debugging, with the optimizer disabled, and then not do so when re-built for the final release. You'd probably have to solve this issue by debugging the application.

On a related note, you may be asking yourself how to know that type **Integer** is 32 bits wide, so that we know what size array to use for the bit-mask. The answer is that you just have to know the target well when doing low-level programming. The hardware becomes much more visible, as we mentioned.

That said, you could at least verify the assumption:

```
pragma Compile_Time_Error (Integer'Object_Size /= 32,
                           "Integers expected to be 32 bits");
X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

That's a vendor-defined pragma so this is not fully portable. It isn't an unusual pragma, though, so at least you can probably get the same functionality even if the pragma name varies.

Overlays aren't always structured like our example above, i.e., with two objects declared at the same time. We might apply a different type to the same memory locations at different times. Here's an example from the ADL to illustrate the idea. We'll elaborate on this example later, in another section.

First, a package declaration, with two functions that provide a device-specific unique identifier located in shared memory. Each function provides the same Id value in a distinct format. One format is a string of 12 characters, the other is a sequence of three 32-bit values. Hence both representations are the same size.

```
package STM32.Device_Id is

   subtype Device_Id_Image is String (1 .. 12);

   function Unique_Id return Device_Id_Image;

   type Device_Id_Tuple is array (1 .. 3) of UInt32
     with Component_Size => 32;

   function Unique_Id return Device_Id_Tuple;

end STM32.Device_Id;
```

In the package body we implement the functions as two ways to access the same shared memory, specified by `ID_Address`:

```
with System;

package body STM32.Device_Id is

   ID_Address : constant System.Address := System'To_Address (16#1FFF_7A10#);

   function Unique_Id return Device_Id_Image is
      Result : Device_Id_Image with Address => ID_Address, Import;
   begin
      return Result;
   end Unique_Id;

   function Unique_Id return Device_Id_Tuple is
      Result : Device_Id_Tuple with Address => ID_Address, Import;
   begin
      return Result;
   end Unique_Id;

end STM32.Device_Id;
```

`System'To_Address` is just a convenient way to convert a numeric value into an **Address**

value. The primary benefit is that the call is a static expression, but we can ignore that here. Using `Import` is a good idea to ensure that the Ada code does no initialization of the object, since the value is coming from the hardware via the shared memory. Doing so may not be necessary, depending on the type used, but is a good habit to develop.

The point of this example is that we have one object declaration per function, of a type corresponding to the intended function result type. Because each function places their local object at the same address, they are still overlaying the shared memory.

Now let's return, momentarily, to setting the size of entities, but now let's focus on setting the size of objects.

We've said that the size of an object is not necessarily the same as the size of the object's type. The object size won't be smaller, but it could be larger. Why? For a stand-alone object or a parameter, most implementations will round the size up to a storage element boundary, or more, so the object size might be greater than that of the type. Think back to **`Boolean`**, where `Size` is required to be 1, but stand-alone objects are probably allocated 8 bits, i.e., an entire storage element (on our hypothetical byte-addressed machine).

Likewise, recall that numeric type declarations are mapped to underlying hardware numeric types. These underlying numeric types provide at least the capabilities we request with our type declarations, e.g., the range or number of digits, perhaps more. But the mapped numeric hardware type cannot provide less than requested. If there is no underlying hardware type with at least our requested capabilities, our declarations won't compile. That mapping means that specifying the size of a numeric type doesn't necessarily affect the size of objects of the type. That numeric hardware type is the size that it is, and is fixed by the hardware.

For example, let's say we have this declaration:

```
type Device_Register is range 0 .. 2**5 - 1 with Size => 5;
```

That will compile successfully, because there will be a signed integer hardware type with at least that range. (Not necessarily, legally speaking, but realistically speaking, there will be such a hardware type.) Indeed, it may be an 8-bit signed integer, in which case `Device_Register'Size` will give us 5, but objects of the type will have a size of 8, unavoidably, even though we set `Size` to 5.

The difference between the type and object sizes can lead to potentially problematic code:

```
type Device_Register is range 0 .. 2**8 - 1 with Size => 8;

My_Device : Device_Register
  with Address => To_Address (...);
```

The code compiles successfully, and tries to map a byte to a hardware device that is physically connected to one storage element in the processor memory space. The actual address is elided as it is not important here.

That code might work too, but it might not. We might think that `My_Device'Size` is 8, and that `My_Device'Address` points at an 8-bit location. However, this isn't necessarily so, as we saw with the supposedly 5-bit example earlier. Maybe the smallest signed integer the hardware has is 16-bits wide. The code would compile because a 16-bit signed numeric type can certainly handle the 8-bit range requested. `My_Device'Size` would be then 16, and because `'Address` gives us the *starting* storage element, `My_Device'Address` might designate the high-order byte of the overall 16-bit object. When the compiler reads the two bytes for `My_Device` what will happen? One of the bytes will be the data presented by the hardware device mapped to the memory. The other byte will contain undefined junk, whatever happens to be in the memory cell at the time. We might have to debug the code a long time to identify that as the problem. More likely we'll conclude we have a failed device.

The correct way to write the code is to specify the size of the object instead of the type:

```
type Device_Register is range 0 .. 2**8 - 1;

My_Device : Device_Register with
  Size => 8,
  Address => To_Address (...);
```

If the compiler cannot support stand-alone 8-bit objects for the type, the code won't compile.

Alternatively, we could change the earlier `Size` clause on the type to apply `Object_Size` instead:

```
type Device_Register is range 0 .. 2**8 - 1 with Object_Size => 8;

My_Device : Device_Register with
  Address => To_Address (...);
```

The choice between the two approaches comes down to personal preference, at least if only a small number of stand-alone objects of the type are going to be declared. With either approach, if the implementation cannot support 8-bit stand-alone objects, we find out that there is a problem at compile-time. That's always cheaper than debugging.

You might conclude that setting the `Size` for a type serves no purpose. That's not an unreasonable conclusion, given what you've seen, but in fact there are reasons to do so. However, there are only a few specific cases so we will save the reasons for the discussions of the specific cases.

There is one general case, though, for setting the `'Size` of a type. Specifically, you may want to specify the size that you think is the minimum possible, and you want the compiler to confirm that belief. This would be one of the so-called "confirming" representation clauses, in which the representation detail is what the compiler would have chosen anyway, absent the specification. You're not actually changing anything, you're just getting confirmation via `Size` whether or not the compiler accepts the clause. Suppose, for example, that you have an enumeration type with 256 values. For enumeration types, the compiler allocates the smallest number of bits required to represent all the values, rounded up to the nearest storage element. (It's not like C, where enums are just named int values.) For 256 values, an eight-bit byte would suffice, so setting the size to 8 would be confirming. But suppose we actually had 257 enumerals, accidentally? Our size clause set to 8 would not compile, and we'd be told that something is amiss.

However, note that if your supposedly "confirming" size clause actually specifies a size larger than what the compiler would have chosen, you won't know, because the compiler will silently accept sizes larger than necessary. It just won't accept sizes that are too small.

There are other confirming representation clauses as well. Thinking again of enumeration types, the underlying numeric values are integers, starting with zero and consecutively increasing from there up to N-1, where N is the total number of enumeral values.

For example:

```
type Commands is (Off, On);

for Commands use (Off => 0, On => 1);
```

As a result, `Off` is encoded as 0 and `On` as 1. That specific underlying encoding is guaranteed by the language, as of Ada 95, so this is just a confirming representation clause nowadays. But it was not guaranteed in the original version of the language, so if you wanted to be sure of the encoding values you would have specified the above. It wasn't necessarily confirming before Ada 95, in other words.

But let's also say that the underlying numeric values are not what you want because you're interacting with some device and the commands are encoded with values other than 0 and

1. Maybe you want to use an enumeration type because you want to specify all the possible values actually used by clients. If you just used some numeric type instead and made up constants for `On` and `Off`, there's nothing to keep clients from using other numeric values in place of the two constants (absent some comparatively heavy code to prevent that from happening). Better to use the compiler to make that impossible in the first place, rather than debug the code to find the incorrect values used. Therefore, we could specify different encodings:

```
for Commands use (Off => 2, On => 4);
```

Now the compiler will use those encoding values instead of 0 and 1, transparently to client code.

The encoding values specified must maintain the relative ordering, otherwise the relational operators won't work correctly. For example, for type Commands above, Off is less than On, so the specified encoding value for Off must be less than that of On.

Note that the values given in the example no longer increase consecutively, i.e., there's a gap. That gap is OK, in itself. As long as we use the two enumerals the same way we'd use named constants, all is well. Otherwise, there is both a storage issue and a performance issue possible. Let's say that we use that enumeration type as the index for an array type. Perfectly legal, but how much storage is allocated to objects of this array type? Enough for exactly two components? Four, with two unused? The answer depends on the compiler, and is therefore not portable. The bigger the gaps, the bigger the overall storage difference possible. Likewise, imagine we have a for-loop iterating over the index values of one of these array objects. The for-loop parameter cannot be coded by the compiler to start at 0, clearly, because there is no index (enumeration) value corresponding to 0. Similarly, to get the next index, the compiler cannot have the code simply increment the current value. Working around that takes some extra code, and takes some extra time that would not be required if we did not have the gaps.

The performance degradation can be significant compared to the usual code generated for a for-loop. Some coding guidelines say that you shouldn't use an enumeration representation clause for this reason, with or without gaps. Now that Ada has type predicates we could limit the values used by clients for a numeric type, so an enumeration type is not the only way to get a restricted set of named, encoded values.

```
type Commands is new Integer with
   Static_Predicate => Commands in 2 | 4;

On    : constant Commands := 2;
Off   : constant Commands := 4;
```

The storage and performance issues bring us back to confirming clauses. We want the compiler to recognize them as such, so that it can generate the usual code, thereby avoiding the unnecessary portability and performance issues. Why would we have such a confirming clause now? It might be left over from the original version of the language, written before the Ada 95 change. Some projects have lifetimes of several decades, after all, and changing the code can be expensive (certified code, for example). Whether the compiler does recognize confirming clauses is a feature of the compiler implementation. We can expect a mature compiler to do so, but there's no guarantee.

Now let's turn to what is arguably the most common representation specification, that of record type layouts.

Recall from the discussion above that Ada compilers are allowed to reorder record components in physical memory. In other words, the textual order in the source code is not necessarily the physical order in memory. That's different from, say, C, where what you write is what you get, and you better know what you're doing. On some targets a misaligned **`struct`** component access will perform very poorly, or even trap and halt, but that's not the C compiler's fault. In Ada you'd have to explicitly specify the problematic layout. Otherwise, if compilation is successful, the Ada compiler must find a representation that will

work, either by reordering the components or by some other means. Otherwise it won't compile.

GNAT did not reorder components until relatively recently but does now, at least for the more egregious performance cases. It does this reordering silently, too, although there is a switch to have it warn you when it does. To prevent reordering, GNAT defines a pragma named `No_Component_Reorder` that does what the name suggests. You can apply it to individual record types, or globally, as a configuration pragma. But of course because the pragma is vendor defined it is not portable.

Therefore, if you care about the record components' layout in memory, the best approach is to specify the layout explicitly. For example, perhaps you are passing data to code written in C. In that case, you need the component order in memory to match the order given in the corresponding C struct declaration. That order in memory is not necessarily guaranteed from the order in the Ada source code. The Ada compiler is allowed to chose the representation unless you specify it, and it might chose a different layout from the one given. (Ordinarily, letting the compiler chose the layout is the most desirable approach, but in this case we have an external layout requirement.)

Fortunately, specifying a record type's layout is straightforward. The record layout specification consists of the storage places for some or all components, specified with a record representation clause. This clause specifies the order, position, and size of components (including discriminants, if any).

The approach is to first define the record type, as usual, using any component order you like — you're about to specify the physical layout explicitly, in the next step.

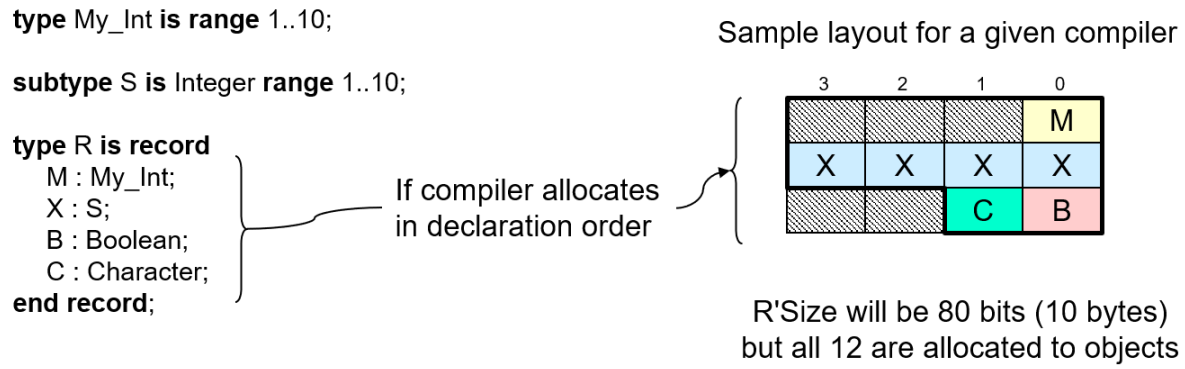
Let's reuse that record type from the earlier discussion:

```
type My_Int is range 1 .. 10;

subtype S is Integer range 1 .. 10;

type R is record
   M : My_Int;
   X : S;
   B : Boolean;
   C : Character;
end record;
```

The resulting layout might be like so, assuming the compiler doesn't reorder the components:



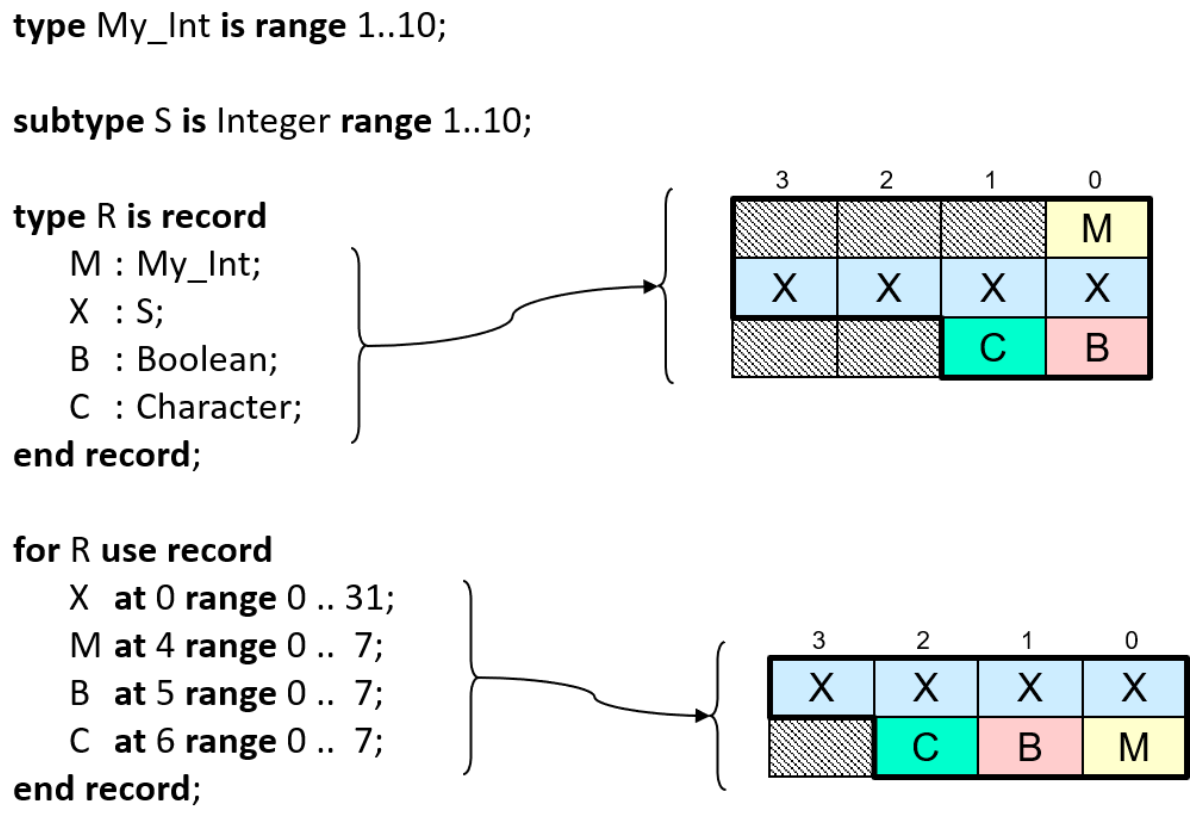
As a result, `R'Size` will be 80 bits (10 bytes), but those last two bytes will be will be allocated to objects, for an `Object_Size` of 96 bits (12 bytes). We'll change that with an explicit layout specification.

Having declared the record type, the second step consists of defining the corresponding record representation clause giving the components' layout. The clause uses syntax that somewhat mirrors that of a record type declaration. The components' names appear, as

in a record type declaration. But now, we don't repeat the components' types, instead we give their relative positions within the record, in terms of a relative offset that starts at zero. We also specify the bits we want them to occupy within the storage elements starting at that offset.

```
for R use record
   X at 0 range 0 .. 31;  -- note the order swap,
   M at 4 range 0 ..  7;  -- with this component
   B at 5 range 0 ..  7;
   C at 6 range 0 ..  7;
end record;
```

Now we'll get the optimized order, and we'll always get that order, or the layout specification won't compile in the first place. In the following diagram, both layouts, the default, and the one resulting from the record representation clause, are depicted for comparison:



`R'Size` will be 56 bits (7 bytes), but that last padding byte will also be allocated to objects, so the `Object_Size` will be 64 bits (8 bytes).

Notice how we gave each component an offset, after the reserved word **`at`**. These offsets are in terms of storage elements, and specify their positions within the record object as a whole. They are relative to the beginning of the memory allocated to the record object so they are numbered starting at zero. We want the `X` component to be the very first component in the allocated memory so the offset for that one is zero. The `M` component, in comparison, starts at an offset of 4 because we are allocating 4 bytes to the prior component `X`: bytes 0 through 3 specifically. `M` just occupies one storage element so the next component, `B`, starts at offset 5. Likewise, component `C` starts at offset 6.

Note that there is no requirement for the components in the record representation clause to be in any particular textual order. The offsets alone specify the components' order in memory. A good style, though, is to order the components in the representation clause so that their textual order corresponds to their order in memory. Doing so facilitates our verifying that the layout is correct because the offsets will be increasing as we read the

specification.

An individual component may occupy part of a single storage element, all of a single storage element, multiple contiguous storage elements, or a combination of those (i.e., some number of whole storage elements but also part of another). The bit "range" specifies this bit-specific layout, per component, by specifying the first and last bits occupied. The X component occupies 4 complete 8-bit storage elements, so the bit range is 0 through 31, for a total of 32 bits. All the other components each occupy an entire single storage element so their bit ranges are 0 through 7, for a total of 8 bits.

The text specifying the offset and bit range is known as a "component_clause" in the syntax productions. Not all components need be specified by component_clauses, but (not surprisingly) at most one clause is allowed per component. Really none are required but it would be strange not to have some. Typically, all the components are given positions. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation.

Components not otherwise given an explicit placement are given positions chosen by the compiler. We don't say that they "follow" those explicitly positioned because there's no requirement that the explicit positions start at offset 0, although it would be unusual not to start there.

Placements must not make components overlap, except for components of variant parts, a topic covered elsewhere. You can also specify the placement of implementation-defined components, as long as you have a name to refer to them. (In addition to the components listed in the source code, the implementation can add components to help implement what you wrote explicitly.) Such names are always attribute references but the specific attributes, if any, are implementation-defined. It would be a mistake for the compiler to define such implicit components without giving you a way to refer to them. Otherwise they might go exactly where you want some other component to be placed, or overlap that place.

The positions (offsets) and the bit numbers must be static, informally meaning that they are known at compile-time. They don't have to be numeric literals, though. Numeric constants would work, but literals are the most common by far.

Note that the language does not limit support for component clauses to specific component types. They need not be one of the integer types, in particular. For example, a position can be given for components that are themselves record types, or array types. Even task types are allowed as far as the language goes, although the implementation might require a specific representation, such as the component taking no bits whatsoever (`0 .. -1`). There are restrictions that keep things sane, for example rules about how a component name can be used within the overall record layout construct, but not restrictions on the types allowed for individual components. For example, here is a record layout containing a **`String`** component, arbitrarily set to contain 11 characters:

```
type R is record
   S : String (1 .. 11);
   B : Boolean;
end record;

for R use record
   S at 0  range 0 .. 87;
   B at 11 range 0 .. 7;
end record;
```

Component S is to be the first component in memory in this example, hence the position offset is 0, for the first byte of S. Next, S is 11 characters long, or 88 bits, so the bit range is 0 .. 87. That's 11 bytes of course, so S occupies storage elements 0 .. 10. Therefore, the next component position must be at least 11, unless there is to be a gap, in which case it would be greater than 11. We'll place B immediately after the last character of S, so B is at storage element offset 11 and occupying all that one byte's bits.

We'll have more to say about record type layouts but first we need to talk about alignment.

Modern target architectures are comparatively strict about the address alignments for some of their types. If the alignment is off, an access to the memory for objects of the type can have highly undesirable consequences. Some targets will experience seriously degraded performance. On others, the target will halt altogether. As you can see, getting the alignment correct is a low-level, but vital, part of correct code on these machines.

Normally the compiler does this work for us, choosing an alignment that is both possible for the target and also optimal for speed of access. You can, however, override the compiler's alignment choice using an attribute definition clause or the `Alignment` aspect. You can do so on types other than record types, but specifying it on record types is typical. Here's our example record type with the alignment specified via the aspect:

```
type My_Int is range 1 .. 10;

subtype S is Integer range 1 .. 10;

type R is record
   M : My_Int;
   X : S;
   B : Boolean;
   C : Character;
end record with
   Alignment => 1;
```

Alignment values are in terms of storage elements. The effect of the aspect or attribute clause is to ensure that the starting address of the memory allocated to objects of the type will be a multiple of the specified value.

In fact, whenever we specify a record type layout we really should also specify the record type's alignment, even though doing so is optional. Why? The alignment makes a difference in the overall record object's size. We've seen that already, with the padding bytes: the compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. The alignment also affects the size allocated to the record type even when the components are already aligned. As a result the overall size could be larger than we want for the sake of space. Additionally, when we pass such objects to code written in other languages, we want to ensure that the starting address of these objects is aligned as the external code expects. The compiler might not choose that required alignment by default.

Specifying alignment for record types is so useful that in the first version of Ada there was no syntax to specify alignment for anything other than record types (via the obsolete **`at mod`** clause on record representation clauses).

For that reason GNAT provides a pragma named `Optimize_Alignment`. This is a configuration pragma that affects the compiler's choice of default alignments where no alignment is explicitly specified. There is a time/space trade-off in the selection of these values, as we've seen. The normal choice tries to balance these two characteristics, but with an argument to the pragma you can give more weight to one or the other. The best approach is to specify the alignments explicitly, per type, for those that require specific alignment values. The pragma has the nice property of giving general guidance to the compiler for what should be done for the other types and objects not explicitly specified.

Now let's look into the details. We'll use a case study for this purpose, including specifying sizes as well as alignments.

The code for the case study is as follows. It uses `Size` clauses to specify the `Sizes`, instead of the `Size` aspect, just to emphasize that the `Size` clause approach is not obsolete.

```
package Some_Types is

   type Temperature is range -275 .. 1_000;
```

(continues on next page)

(continued from previous page)

```
   type Identity is range 1 .. 127;

   type Info is record
      T  : Temperature;
      Id : Identity;
   end record;

   for Info use record
      T  at 0 range 0 .. 15;
      Id at 2 range 0 .. 7;
   end record;

   for Info'Size use 24;

   type List is array (1 .. 3) of Info;
   for List'Size use 24 * 3;

end Some_Types;
```

When we compile this, the compiler will complain that the size for `List` is too small, i.e., that the minimum allowed is 96 bits instead of the 72 we specified. We specified 24 * 3 because we said the record size should be 24 bits, and we want our array to contain 3 record components of that size, so 72 seems right.

What's wrong? As we've shown earlier, specifying the record type size doesn't necessarily mean that objects (in this case array components) are that size. The object size could be bigger than we specified for the type. In this case, the compiler says we need 96 total bits for the array type, meaning that each of the 3 array components is 32 bits wide instead of 24.

Why is it 32 bits? Because the alignment for `Info` is 2 (on this machine). The record alignment is a multiple of the largest alignment of the enclosed components. The alignment for type `Temperature` (2), is larger than the alignment for type `Identity` (1), therefore the alignment for the whole record type is 2. We need to go from that number of storage elements to a number of bits for the size.

Here's where it gets subtle. The alignment is in terms of storage elements. Each storage element is of a size in bits given by `System.Storage_Unit`. We've said that on our hypothetical machine `Storage_Unit` is 8, so storage elements are 8 bits wide on this machine. Bytes, in other words. Therefore, to get the required size in bits, we have to find a multiple of the two 8-bit bytes (specified by the alignment) that has at least the number of bits we gave in the `Size` clause. Two bytes only provides 16 bits, so that's not big enough, we need at least 24 bits. The next multiple of 2 bytes is 4 bytes, providing 32 bits, which is indeed larger than 24. Therefore, the overall size of the record type, consistent with the alignment, is 4 bytes, or 32 bits. That's why the compiler says each array component is 32 bits wide.

But for our example let's say that we really want to use only 72 total bits for the array type (and that we want three array components). That's the size we specified, after all. So how do we get the record type to be 24 bits instead of 32? Yes, you guessed it, we change the alignment for the record type. If we change it from 2 to 1, the size of 24 bits will work. Adding this `Alignment` clause line will do that:

```
for Info'Alignment use 1;
```

An alignment of 1 means that any address will work, assuming that addresses refer to entire storage elements. (An alignment of 0 would mean that the address need not start on a storage element boundary, but we know of no such machines.)

We can even entirely replace the `Size` clause with the `Alignment` clause, because the `Size` clause specifying 24 bits is just confirming: it's the value that `'Size` would return anyway.

The problem is the object size.

Now, you may be wondering why an alignment of 1 would work, given that the alignment of the `Temperature` component is 2. Wouldn't it slow down the code, or even trap? Well, maybe. It depends on the machine. If it doesn't work we would just have to use 32 bits for the record type, with the original alignment of 2, for a larger total array size. Of course, if the compiler recognizes that a representation cannot be supported it must reject the code, but the compiler might not recognize the problem.

We said earlier that there are only a small number of reasons to specify `'Size` for a type. We can mention one of them now. Setting `'Size` can be useful to give the minimum number of bits to use for a component of a packed composite type, that is, within either a record type or an array type that is explicitly packed via the aspect or pragma Pack. It says that the compiler, when giving its best effort, shouldn't compress components of the type any smaller than the number of bits specified. No, it isn't earth-shattering, but other uses are more valuable, to be discussed soon.

One thing we will leave unaddressed (pun intended) is the question of bit ordering and byte ordering within our record layouts. In other words, the "endian-ness". That's a subject beyond the scope of this course. Suffice it to say that GNAT provides a way to specify record layouts that are independent of the endian-ness of the machine, within some implementation-oriented limits. That's obviously useful when the code might be compiled for a different ISA in the future. On the other hand, if your code is specifically for a single ISA, e.g. Arm, even if different boards and hardware vendors are involved, there's no need to be independent of the endian-ness. It will always be the same in that case. (Those are "famous last words" though.) For an overview of the GNAT facility, an attribute named attribute `Scalar_Storage_Order` see https://www.adacore.com/papers/lady-ada-mediates-peace-treaty-in-endianness-war.

Although specifying record type layouts and alignments are perhaps the most common representation characteristics expressed, there are a couple of other useful cases. Both involve storage allocation.

One useful scenario concerns tasking. We can specify the number of storage elements reserved for the execution of a task object, or all objects of a task type. You use the `Storage_Size` aspect to do so:

```
task Servo with
  Storage_Size => 1 * 1024,
  ...
```

Or the corresponding pragma:

```
task Servo is
   pragma Storage_Size (1 * 1024);
end Servo;
```

The aspect seems textually cleaner and lighter unless you have task entries to declare as well. In that case the line for the pragma wouldn't add all that much. That's a matter of personal aesthetics anyway.

The specified number of storage elements includes the size of the task's stack (GNAT does have one, per task). The language does not specify whether or not it includes other storage associated with the task used for implementing and managing the task execution. With GNAT, the extent of the primary stack size is the value returned, ignoring any other storage used internally in the run-time library for managing the task.

The GNAT run-time library allocates a default stack amount to each task, with different defaults depending on the underlying O.S., or lack thereof, and the target. You need to read the documentation to find the actual amount, or, with GNAT, read the code.

You would need to specify this amount in order to either increase or decrease the allocated storage. If the task won't run properly, perhaps crashing at strange and seemingly random

places, there's a decent chance it is running out of stack space. That might also be the reason if you have a really deep series of subprogram calls that fails. The correction is to increase the allocation, as shown above. How much? Depends on the application code. The quick-and-dirty approach is to iteratively increase the allocation until the task runs properly. Then, reverse the approach until it starts to fail again. Add a little back until it runs, and leave it there. We'll mention a much better approach momentarily (**GNATstack**).

Even if the task doesn't seem to run out of task stack, you might want to reduce it anyway, to the extent possible, because the total amount of storage on your target might be limited. Some of the GNAT bare-metal embedded targets have very small amounts of memory available, so much so that the default task stack allocations would exhaust the memory available quickly. That's what the example above does: empirical data showed that the `Servo` task could run with just 1K bytes allocated, so we reduced it from the default accordingly. (We specified the size with that expression for the sake of readability, relative to using literals directly.)

Notice we said "empirical data" above. How do we know that we exercised the task's thread of control exhaustively, such that the arrived-at allocation value covers the worst case? We don't, not with certainty. If we really must know the allocation will suffice for all cases, say because this is a high-integrity application, we would use **GNATstack**. GNATstack is an offline tool that exploits data generated by the compiler to compute worst-case stack requirements per subprogram and per task. As a static analysis tool, its computation is based on information known at compile time. It does not rely on empirical run-time information.

The other useful scenario for allocating storage concerns access types, specifically access types whose values designate objects, as opposed to designating subprograms. (Remember, objects are either variables or constants.) There is no notion of dynamically allocating procedures and functions in Ada so access-to-subprogram types are not relevant here. But objects can be of protected types (or task types), and protected objects can "contain" entries and protected subprograms, so there's a lot of expressive power available. You just don't dynamically allocate procedures or functions as such.

First, a little background on access types, to supplement what we said earlier.

By default, the implementation chooses a standard storage pool for each named access-to-object type. The storage allocated by an allocator (i.e., **new**) for such a type comes from the associated pool.

Several access types can share the same pool. By default, the implementation might choose to have a single global storage pool, used by all such access types. This global pool might consist merely of calls to operating system routines (e.g., `malloc`), or it might be a vendor-defined pool instead. Alternatively, the implementation might choose to create a new pool for each access-to-object type, reclaiming the pool's memory when the access type goes out of scope (if ever). Other schemes are possible.

Finally, users may define new pool types, and may override the choice of pool for an access-to-object type by specifying `Storage_Pool` for the type. In this case, allocation (via **new**) takes memory from the user-defined pool and deallocation puts it back into that pool, transparently.

With that said, here's how to specify the storage to be used for an access-to-object type. There are two ways to do it.

If you specify `Storage_Pool` for an access type, you indicate a specific pool object to be used (user-defined or vendor-defined). The pool object determines how much storage is available for allocation via **new** for that access type.

Alternatively, you can specify `Storage_Size` for the access type. In this case, an implementation-defined pool is used for the access type, and the storage available is at least the amount requested, maybe more (it might round up to some advantageous block size, for example). If the implementation cannot satisfy the request, `Storage_Error` is raised.

It should be clear that that the two alternatives are mutually exclusive. Therefore the compiler will not allow you to specify both.

Each alternative has advantages. If your only concern is the total number of allocations possible, use `Storage_Size` and let the implementation do the rest. However, maybe you also care about the behavior of the allocation and deallocation routines themselves, beyond just providing and reclaiming the storage. In that case, use `Storage_Pool` and specify a pool object of the appropriate type. For example, you (or the vendor, or someone else) might create a pool type in which the allocation routine performs in constant time, because you want to do **`new`** in a real-time application where predictability is essential.

Lastly, an idiom: when using `Storage_Size` you may want to specify a value of zero. That means you intend to do no allocations whatsoever, and want the compiler to reject the code if you try. Why would you want an access type that doesn't allow dynamically allocating objects? It isn't as unreasonable as it might sound. If you plan to use the access type strictly with aliased objects, never doing any allocations, you can have the compiler enforce your intent. There are application domains that prohibit dynamic allocations due to the difficulties in analyzing their behavior, including issues of fragmentation and exhaustion. Access types themselves are allowed in these domains. You'd simply use them to designate aliased objects alone. In addition, in this usage scenario, if the implementation associates an actual pool with each access type, the pool's storage would be wasted since you never intend to allocate any storage from it. Specifying a size of 0 tells the implementation not to waste that storage.

Before we end this section, there is a GNAT compiler switch you should know about. Th `-gnatR?` switch instructs the compiler to list the representation details for the types, objects and subprograms in the compiled file(s). Both implementation-defined and user-defined representation details are presented. The '?' is just a placeholder and can be one of the following characters:

```
[0|1|2|3|4][e][j][m][s]
```

Increasing numeric values provide increasing amounts of information. The default is '1' and usually will suffice. See the GNAT User's Guide for Native Platforms for the details of the switch in section 4.3.15 Debugging Control[3].

You'll have to scroll down some to find that specific switch but it is worth finding and remembering. When you cannot understand what the compiler is telling you about the representation of something, this switch is your best friend.

2.6 Unchecked Programming

Ada is designed to be a reliable language by default, based as it is on static strong typing and high-level semantics. Many of the pitfalls that a developer must keep in the back of their mind with other languages do not apply in Ada, and are typically impossible. That protection extends to low-level programming as well, e.g., the Separation Principle. Nevertheless, low-level programming occasionally does require mechanisms that allow us to go beyond the safety net provided by the type rules and high-level language constructs.

One such mechanism (unchecked conversion) provides a way to circumvent the type system, a system otherwise firmly enforced by the compiler on our behalf. Note that by "circumventing the type system" we do not include so-called "checked" conversions. These conversions have meaningful semantics, and are, therefore, allowed by the language using a specific syntax. This conversion syntax is known as "functional" syntax because it looks like a function call, except that the "function" name is a type name, and the parameter is the object or value being converted to that type. These conversions are said to be "checked" because only specific kinds of types are allowed, and the compiler checks that such conversions are indeed between these allowed types.

[3] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#debugging-control

Instead, this section discusses "unchecked" programming, so-called because the compiler does not check for meaningful semantics. There are multiple mechanisms for unchecked programming in Ada: in addition to circumventing the type system, we can also deallocate a previously-allocated object, and can create an access value without the usual checks. In all cases the responsibility for correct meaning and behavior rests on the developer. Very few, if any, checks are done by the compiler. If we convert a value to another type that generally makes no sense, for example a task object converted to a record type, we are on our own. If we deallocate an allocated object more than once, it is our fault and Bad Things inevitably result.

Likened to "escape hatches," the facilities for unchecked programming are explicit in Ada. Their use is very clear in the source code, and is relatively heavy: each mechanism is provided by the language in the form of a generic library subprogram that must be specified in a context clause ("with-clause") at the top of the file, and then instantiated prior to use, like any generic. For an introduction to generic units in Ada, see that section in the introductory Ada course: Introduction to Ada

You should understand that the explicitly unchecked facilities in Ada are no more unsafe than the implicitly unchecked facilities in other languages. There's no safety-oriented reason to "drop down" to C, for example, to do low-level programming. For that matter, the low-level programming facilities in Ada are at least as powerful as those in other languages, and probably more so.

We will explore unchecked storage deallocation in a separate book so let's focus on unchecked type conversions.

Unchecked type conversions are achieved by instantiating this language-defined generic library function, a "child" of the root package named "Ada":

```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion (S : Source) return Target
   with Pure, Nonblocking, Convention => Intrinsic;
```

The function, once instantiated and eventually invoked, returns the caller's value passed to S (of type `Source`) as if it is a value of type `Target`. That value can then be used in any way consistent with the `Target` type.

The two generic parameters, `Source` and `Target`, are defined in a manner that makes them very permissive in terms of the types they will accept when instantiated. To understand how, you need to understand a little bit of Ada's terminology and design for generic unit parameters. (If you are already familiar with generic formal types and how they are matched, feel free to skip this material.)

First, the terminology. The type parameters defined by a generic unit are known as "generic formal types," or "generic formals" for short. Types `Source` and `Target` are the generic formals in the unit above. When instantiating such a generic, clients must specify a type for each generic formal type. The types specified by the client are known as "generic actual types," or "generic actuals" for short. You can remember that by the fact that the actuals are the types "actually" given to the generic unit to work with when instantiated. (You may laugh, but that mnemonic works.)

Now we're ready to discuss the language design concept. The idea is that the syntax of a generic formal type indicates what kind of generic actual is required for a legal instantiation. This is known as the "Contract Model" because we can think of the formal parameters as expressing a contract between the generic unit's implementation and the client code that instantiates the generic. The contract is enforced by the compiler, in that it will reject any instantiation that attempts to specify some actual type that does not match the formal's requirements.

For example, if the generic computes some value for any floating point type, that floating-point type would be declared as a generic formal type, and would be defined so that only

some floating-point type could be used for the corresponding actual type:

```
generic
   type Real is digits <>;
```

The formal parameter syntax reflects the syntax of a floating-point type declaration, except that the <> (the "box") indicates that the generic does not care how many digits are available. The generic actual will be some floating point type and it will specify the number of decimal digits.

If instead we try to match that formal with some actual that is anything other than a floating-point type the compiler will reject the instantiation. Therefore, within the generic body, the implementation code can be written with the assurance that the characteristics and capabilities required of a floating point type will be available. That's the Contract Model in full: the requirements are a matter of the generic unit's purpose and implementation, so the formal parameters reflect those requirements and the compiler ensures they will be met.

Some generic units, though, do not require specifically numeric actual types. These generics can use less specific syntax for their formal types, and as a result, more kinds of actual types are permitted in the instantiations. Remember the Contract Model and this will make sense. The contract between the generic and the clients is, in this case, more permissive: it does not require a numeric type in order to implement whatever it does.

For illustration, suppose we want a generic procedure that will exchange two values of some type. What operations does the generic unit require in the implementation in order to swap two values? There are two: assignment, as you might expect, but also the ability to declare objects of the type (the "temporary" used to hold one of the values during the swap steps). As long as the body can do that, any type will suffice, so the generic formals are written to be that permissive. What is the syntax that expresses that permissiveness, you ask? To answer that, first consider simple, non-generic private types from the user's point of view. For example:

```
package P is
   type Foo is private;
   procedure Do_Something (This : Foo);
private
   type Foo is ...  -- whatever
end P;
```

There are two "views" associated with the package: one for the "visible" part of the package spec (declaration), known as the "partial" view, and one for the "private" part of the package spec and the package body, known as the "full" view. The differences between the two views are a function of compile-time visibility.

The partial view is what clients (i.e., users) of the package have: the ability to do things that a type name provides, such as declarations of objects, as well as some basic operations such as assignment, some functions for equality and inequality, some conversions, and whatever subprograms work on the type (the procedure `Do_Something` above). Practically speaking, that's about all that the partial view provides. That's quite a lot, in fact, and corresponds to the classic definition of an "abstract data type."

The code within the package private part and package body has the full view. This code has compile-time visibility to the full definition for type Foo, so there are additional capabilities available to this code. For example, if the full definition for `Foo` is as an array type, indexing will be available with the private part and body. If Foo is fully defined as some numeric type, arithmetic operations will be possible within the package, and so on.

Therefore, the full view provides capabilities for type `Foo` that users of the type cannot access via the partial view. Only the implementation for type `Foo` and procedure `Do_Something` have the potential to access them.

Now, back to the generic formal parameter. If the generic unit doesn't care what the actual type is, and just needs to be able do assignment and object declaration, a "generic formal private type" expresses exactly that:

```
generic
   type Item is private;
procedure Exchange( Left, Right : in out Item );

procedure Exchange( Left, Right : in out Item ) is
   Old_Left : Item;
begin
   Old_Left := Left;
   Left := Right;
   Right := Old_Left;
end Exchange;
```

Inside generic procedure Exchange, the view of type `Item` is as if `Item` were some private type declared in a package, with only the partial view available. But the operations provided by a partial view are sufficient to implement the body of Exchange: only assignment and object declaration are required. Any additional capabilities that the generic actual type may have — array indexing, arithmetic operators, whatever — are immaterial because they are not required. That's the Contract Model: only the specified view's required capabilities are important. Anything else the type can also do is not relevant.

But consider limited types. Those types don't allow assignment, by definition. Therefore, an instantiation that specified a limited actual type for the generic formal type Item above would be rejected by the compiler. The contract specifies the ability to do assignment so a limited type would violate the contract.

Finally, as mentioned, our Exchange generic needs to declare the "temporary" object `Old_Left`. A partial view of a private type allows that. But not all types are sufficient, by their name alone, to declare objects. Unconstrained array types, such as type **`String`**, are a familiar example: they require the bounds to be specified when declaring objects; the name **`String`** alone is insufficient. Therefore, such types would also violate the contract and, therefore, would be rejected by the compiler when attempting to instate generic procedure Exchange.

Suppose, however, that we have some other generic unit whose implementation does not need to declare objects of the formal type. In that case, a generic actual type that did not support object declaration (by the name alone) would be acceptable for an instantiation. The generic formal syntax for expressing that contract uses these tokens: (<>) in addition to the other syntax mentioned earlier:

```
generic
   type Foo(<>) is private;
```

In the above, the generic formal type Foo expresses the fact that it can allow unconstrained types — known as "indefinite types" — when instantiated because it will not attempt to use that type name to declare objects. Of course, the compiler will also allow constrained types (e.g., **`Integer`**, **`Boolean`**, etc.) in instantiations because it doesn't matter one way or the other inside the generic implementation. The Contract Model says that additional capabilities, declaring objects in this case, are allowed but not required. (There is a way to declare objects of indefinite types, but not using the type name alone. The unchecked facilities don't need to declare objects so we will not show how to do it.)

Now that you understand the Contract Model (perhaps more than you cared), we are ready to examine the generic formal type parameters for `Ada.Unchecked_Conversion`. Here's the declaration again:

```
generic
   type Source(<>) is limited private;
```

(continues on next page)

(continued from previous page)

```
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion (S : Source) return Target
   with Pure, Nonblocking, Convention => Intrinsic;
```

The two generic formal types, `Source`, and `Target`, are the types used for the incoming value and the returned value, respectively. Both formals are "indefinite, limited private types" in the jargon, but now you know what that means. Inside the implementation of the generic function, neither `Source` nor `Target` will be used to declare objects (the `(<>)` syntax). Likewise, neither type will be used in an assignment statement (the "limited" reserved word). And finally, no particular kind of type is required for `Source` or `Target` (the **`private`** reserved word). That's a fairly restricted usage within the generic implementation, but as a result the contract can be very permissive: the generic can be instantiated with almost any type. It doesn't matter if the actual is limited or not, private or not, and indefinite or not. The generic implementation doesn't need those capabilities to implement a conversion so they are not part of the contract expressed by the generic formal types.

What sort of type would be disallowed? Abstract types, and incomplete types. However, it is impossible to declare objects of those types, for good reasons, so unchecked conversion is never needed for them.

Note that the result value is returned by-reference whenever possible, in which case it is just a view of the `Source` bits in the formal parameter S and not a copy. For a `Source` type that is not a by-copy type, the result of an unchecked conversion will typically be returned by-reference (so that the result and the parameter S share the same storage); for a by-copy `Source` type, a copy is made.

The compiler can restrict instantiations but implementers are advised by the language standard to avoid them unless they are required by the target environment. For example, an instantiation for types for which unchecked conversion can't possibly make sense might be disallowed.

Clients can apply language- and vendor-defined restrictions as well, via pragma `Restrictions`. In particular, the language defines the `No_Dependence` restriction, meaning that no client's context clause can specify the unit specified. As a result no client can instantiate the generic for unchecked conversion:

```
pragma Restrictions (No_Dependence => Ada.Unchecked_Conversion);
```

hence there would be no use of unchecked conversion.

From the Contract Model's point of view most any type can be converted to some other type via this generic function. But practically speaking, some limitations are necessary. The following must all be true for the conversion effect to be defined by the language:

- `S'Size = Target'Size`
- `S'Alignment` is a multiple of `Target'Alignment`, or `Target'Alignment` is 0 (meaning no alignment required whatsoever)
- `Target` is not an unconstrained composite type
- `S` and `Target` both have a contiguous representation
- The representation of S is a representation of an object of the target subtype

We will examine these requirements in turn, but realize that they are not a matter of legality. Compilers can allow instantiations that violate these requirements. Rather, they are requirements for conversions to have the defined effect.

The first requirement is that the size (in bits) for the parameter S, of type `Source`, is the same as the size of the `Target` type. That's reasonable if you consider it. What would it mean to convert, for example, a 32-bit value to an 8-bit value? Which 8 bits should be used?

As a result, one of the few reasons for setting the size of a type (as opposed to the size of an object) is for the sake of well-defined unchecked conversions. We might make the size larger than it would need to be because we want to convert a value of that type to what would otherwise be a larger `Target` type.

Because converting between types that are not the same size is so open to interpretation, most compilers will issue a warning when the sizes are not the same. Some will even reject the instantiation. GNAT will issue a warning for these cases when the warnings are enabled, but will allow the instantiation. We're supposed to know what we are doing, after all. The warning is enabled via the specific `-gnatwz` switch or the more general `-gnatwa` switch. GNAT tries to be permissive. For example, in the case of discrete types, a shorter source is first zero or sign extended as necessary, and a shorter target is simply truncated on the left. See the GNAT RM for the other details.

The next requirement concerns alignment. As we mentioned earlier, modern architectures tend to have strict alignment requirements. We can meaningfully convert to a type with a stricter alignment, or to a type with no alignment requirement, but converting in the other direction would require a copy.

Next, recall that objects of unconstrained types, such as unconstrained array types or discriminated record types, must have their constraints specified when the objects are declared. We cannot just declare a **`String`** object, for example, we must also specify the lower and upper bounds. Those bounds are stored in memory, logically as part of the **`String`** object, since each object could have different bounds (that's the point, after all). What, then, would it mean to convert some value of a type that has no bounds to a type that requires bounds? The third requirement says that it is not meaningful to do so.

The next requirement is that the argument for S, and the conversion target type `Target`, have a contiguous representation in memory. In other words, each storage unit must be immediately adjacent, physically, to the next logical storage unit in the value. Such a representation for any given type is not required by the language, although on typical modern architectures it is common. (The type `System.Storage_Elements.Storage_Array` is an exception, in that a contiguous representation is guaranteed.) An instance of `Ada.Unchecked_Conversion` just takes the bits of S and treats them as if they are bits for a value of type `Target` (more or less), and does not handle issues of segmentation.

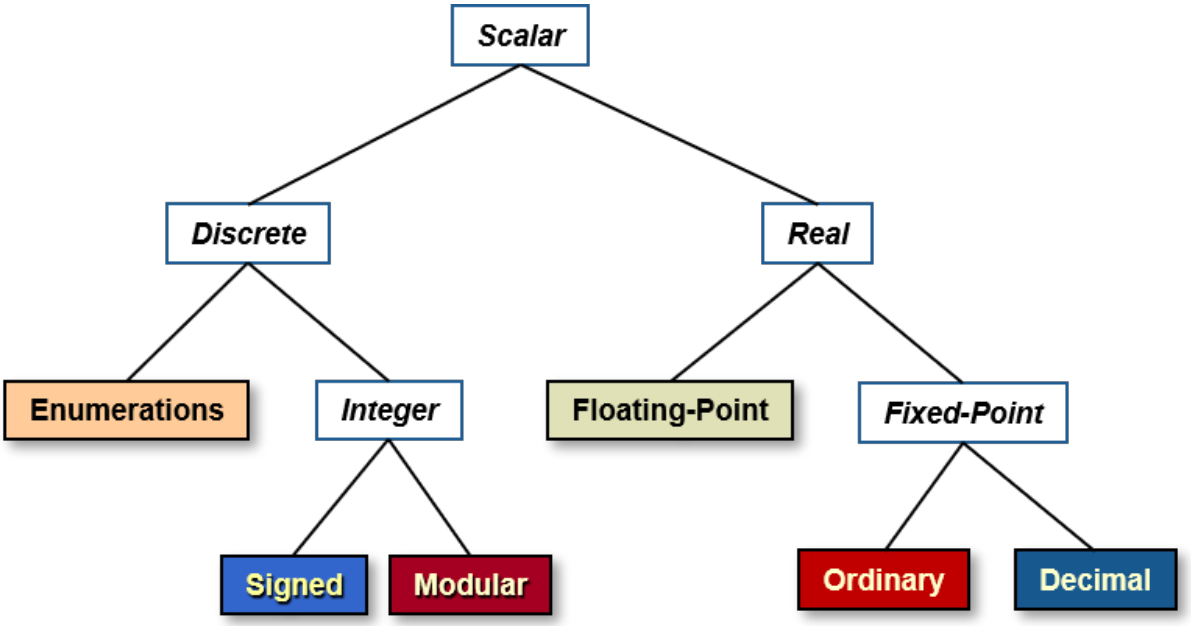
The last requirement merely states that the bits of the argument S, when treated as a value of type `Target`, must actually be a bit-pattern representing a value of type `Target` (strictly, the subtype). For example, with signed integers, any bit pattern (of the right size) represents a valid value for those types. In contrast, consider an enumeration type. By default, the underlying representational values are the same as the position values, i.e., starting at zero and increasing by one. But users can override that representation: they can start with any value and, although the values must increase, they need not increase by one:

```
type Toggle_Switch is (Off, On);
for Toggle_Switch use (Off => 0, On => 4);
```

If we covert an unsigned integer (of the right size) to a `Toggle_Switch` value, what would it mean if the `Source` value was neither 0 nor 4?

We've said that the instantiations are likely allowed, hence callable functions are created. If the above requirements are not met, what happens?

What happens depends on the `Target` type, that is, the result type for the conversion. Specifically, it depends on whether the target type is a "scalar" type. As we mentioned earlier, a scalar type is either a "discrete" type or a "real" type, which are themselves further defined, as the figure below indicates. Any other type is a non-scalar type, e.g., record types, access types, task types, and so on.



When the requirements for meaningful instantiations are not respected and the Target type is a scalar type, the result returned from the call is implementation defined and is potentially an invalid representation. For example, type Toggle_Switch is an enumeration type, hence it is a scalar type. Therefore, if we covert an unsigned integer (of the right size) to a Toggle_Switch value, and the Source value is neither 0 nor 4, the resulting value is an invalid representation. That's the same as an object of type Toggle_Switch that is never assigned a value. The random junk in the bits may or may not be a valid Toggle_Switch value. That's not a good situation, clearly, but it is well-defined: if it is detected, either Constraint_Error or Program_Error is raised. If the situation is not detected, execution continues using the invalid representation. In that case it may or may not be detected, near the call or later. For example:

```
with Ada.Unchecked_Conversion;
with Ada.Text_IO;  use Ada.Text_IO;
with Interfaces;   use Interfaces;

procedure Demo is

   type Toggle_Switch is (Off, On) with Size => 8;
   for Toggle_Switch use (Off => 1, On => 4);

   function As_Toggle_Switch is new Ada.Unchecked_Conversion
     (Source => Unsigned_8, Target => Toggle_Switch);

   T1 : Toggle_Switch;
   T2 : Toggle_Switch;
begin
   T1 := As_Toggle_Switch (12);  -- neither 1 nor 4
   if T1 = Off then
      Put_Line ("T1's off");
   else
      Put_Line ("T1's on");
   end if;
   T2 := T1;
   if T2 = Off then
      Put_Line ("T2's off");
   else
      Put_Line ("T2's on");
   end if;
```

(continues on next page)

(continued from previous page)

```
   Put_Line (T2'Image);
end Demo;
```

In the execution of the code above, the invalid representation value in T1 is not detected, except that it is copied into T2, where it is eventually detected when `'Image` is applied to T2. The invalid representation is not detected in the assignment statement or the comparison because we want the optimizer to be able to avoid emitting a check prior to every use of the value. Otherwise the generated code would be too slow. (The language explicitly allows this optimization.)

The evaluation of an object having an invalid representation value due to unchecked conversion is a so-called "bounded error" because the results at run-time are predictable and limited to one of those three possibilities: the two possible exceptions, or continued execution.

Continued execution might even work as hoped, but such code is not portable and should be avoided. A new vendor's compiler, or even a new version of a given vendor's compiler, might detect the situation and raise an exception. That happens, and it ends up costing developer time to make the required application code changes.

The possibilities get much worse when the result type is not a scalar type. In this case, the *effect* of the call — not the value returned by the call — is implementation defined. As a result, the possible run-time behavior is unpredictable and, consequently, from the language rules point of view anything is possible. Such execution is said to be "erroneous."

Why the difference based on scalar versus non-scalar types? Scalar types have a simple representation: their bits directly represent their values. Non-scalar types don't always have a simple representation that can be verified by examining their bits.

For example, we can have record types with discriminants that control the size of the corresponding objects because the record type contains an array component that uses the discriminant to set the upper bound. These record types might have multiple discriminants, and multiple dependent components. As a result, an implementation could have hidden, internal record components. These internal components might be used to store the starting address of the dependent components, for example, or might use pointers to provide a level of indirection. If an unchecked conversion did not provide correct values for these internal components, the effect of referencing the record object would be unpredictable.

Even a comparatively simple record type with one such dependent component is sufficient to illustrate the problem. There are no internal, hidden components involved:

```
with Ada.Unchecked_Conversion;
with Ada.Text_IO;              use Ada.Text_IO;
with System;                   use System;  -- for Storage_Unit
with System.Storage_Elements; use System.Storage_Elements;

procedure Demo_Erroneous is

   subtype Buffer_Size is Storage_Offset range 1 .. Storage_Offset'Last;

   type Bounded_Buffer (Capacity : Buffer_Size) is record
      Content : Storage_Array (1 .. Capacity);
      Length  : Storage_Offset := 0;
   end record;

   procedure Show_Capacity (This : Bounded_Buffer);

   subtype OneK_Bounded_Buffer is Bounded_Buffer (Capacity => 1 * 1024);

   function As_OneK_Bounded_Buffer is new Ada.Unchecked_Conversion
     (Source => Storage_Array, Target => OneK_Bounded_Buffer);
```

(continues on next page)

(continued from previous page)

```
   Buffer   : OneK_Bounded_Buffer;
   Sequence : Storage_Array (1 .. Buffer'Size / Storage_Unit);

   procedure Show_Capacity (This : Bounded_Buffer) is
   begin
      Put_line ("This.Capacity is" & This.Capacity'Image);
   end Show_Capacity;

begin
   Buffer := As_OneK_Bounded_Buffer (Sequence);
   Put_Line ("Buffer capacity is" & Buffer.Capacity'Image);
   Show_Capacity (Buffer);
   Put_Line ("Done");
end Demo_Erroneous;
```

In the above, the type Bounded_Buffer has an array component Content that depends on the discriminant Capacity for the number of array components. This is an extremely common idiom. However, unchecked conversion is only meaningful, as defined earlier, when converting to constrained target types. Bounded_Buffer is not constrained, so we define a constrained subtype (OneK_Bounded_Buffer) for the sake of the conversion.

The specific Buffer object is 8320 bits (1024 * 8, plus 2 * 64), as is the Sequence object, so the sizes are the same.

The alignment of OneK_Bounded_Buffer is 8, and Storage_Array's alignment is 1, so the Target type is a multiple of the Source type, as required.

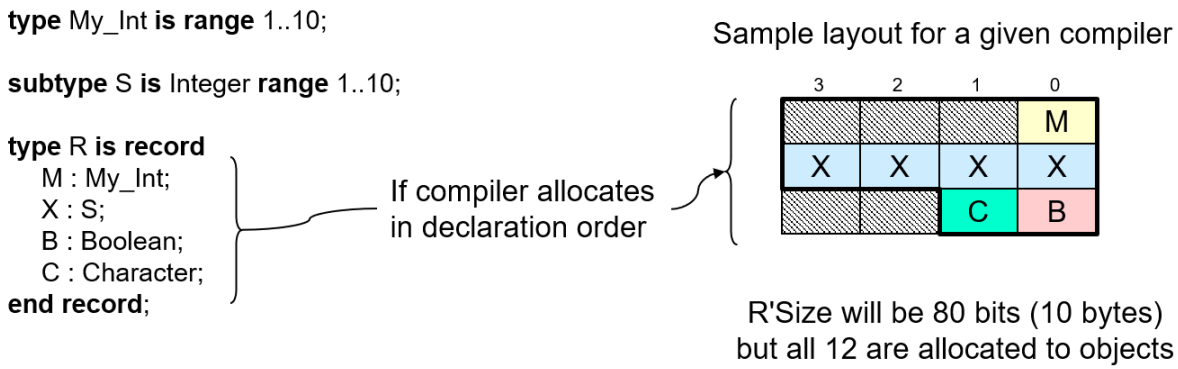
Both types have a contiguous representation, and the sequence of bytes can be a valid representation for the record type, although it certainly might not be valid. For example, if we change the discriminant from what the subtype specifies, we would have an invalid representation for that subtype.

So we can reasonably invoke an unchecked conversion between the array of bytes and the record type. However, as you can see in the code and as the compiler warns, we never assigned a value to the Sequence array object. The unchecked conversion from that Sequence of bytes includes the discriminant value, so it is very possible that we will get a discriminant value that is not 1K.

We can test that possibility by running the program. In the first call to Put_Line, the program prints the Capacity discriminant for the Buffer object. The compiler knew it was 1024, so it doesn't get the discriminant component from memory, it just directly prints 1024. However, we can force the compiler to query the discriminant in memory. We can pass Buffer to procedure Show_Capacity, which takes any Bounded_Buffer, and there query (print) the Capacity component under that different view. That works because the view inside the procedure Show_Capacity is as of Bounded_Buffer, in which the discriminant value is unknown at compile-time.

In the above examples, we are responsible for ensuring that the enumeration representation encoding and the record discriminant value are correct when converted from some other type. That's not too hard to recognize because we can literally see in the source code that there is something to be maintained by the conversions. However, there might be hidden implementation artifacts that we cannot see in the source code but that must be maintained nevertheless.

For example, the compiler's implementation for some record type might use dynamic memory allocations instead of directly representing some components. That would not appear in the source code. As a simpler example of invisible implementation issues, consider again our earlier record type:



As we discussed earlier, between the bytes that are allocated to the record components are some other bytes that are not used at all. As usual, the compiler must implement the language-defined equality operator for the record type. One way to implement that function would be to generate code that checks the equality for each component individually, ignoring any unused bytes. But suppose you have a large record type with many components. The code for checking record level equality will be extensive and inefficient. An alternative implementation for the compiler would be to use a "block compare" machine instruction to check the equality of the entire record at once, rather than component-by-component. That will be considerably more efficient because the block-compare instruction just compares the bits from one starting address to another ending address. But in that case the "unused" bytes are not skipped so the values within those bytes become significant. Comparison of those unused bytes will only work if their values are defined and assigned in each record object. Compilers that may use a block-comparison approach will, therefore, always set those unused bytes to a known value (typically zero). That is part of the valid representation for values of the type, and consequently must be maintained by our unchecked conversions. This being a non-scalar target type, failure to do so results in erroneous execution, i.e., undefined behavior. "There be dragons" as ancient maps of the unknown world once said.

As you can see, you should use unchecked conversions with considerable care and thought. Moreover, because unchecked programming is such a low-level activity, and has vendor-defined implementation issues, it is not only less portable than high-level coding, it is also less portable than other low-level programming. You will be well served if you limit the use of unchecked conversions overall. If your application code is performing unchecked conversions all over the code, something is very likely wrong, or at least very questionable. A well-designed Ada program should not need ubiquitous unchecked conversions.

That said, of course sometimes unchecked conversions are reasonable. But even then, it is better to isolate and hide their use via compile-time visibility controls. For example, instead of having clients invoke unchecked conversion instances many times, have a procedure that is invoked many times, and let the procedure body do the conversion. That way, the clients see a high-level specification of functionality, and, if the conversion needs to be changed later, there is only that one conversion usage (the procedure body) to change. This approach is really just another example of isolating and hiding code that might need to change in the future.

2.7 Data Validity

Our earlier demo program assigned an incorrect value via unchecked conversion into an object of an enumeration type that had non-standard representation values. The value assigned was not one of those representation values so the object had an invalid representation. Certain uses of an invalid representation value will be erroneous, and we saw that the effect of erroneous execution was unpredictable and unbounded.

That example was somewhat artificial, for the sake of illustration. But we might get an

invalid value in a real-world application. For example, we could get an invalid value from a sensor. Hardware sensors are frequently unreliable and noisy. We might get an invalid value from a call to an imported function implemented in some other language. Whenever an assignment is aborted, the target of the assignment might not be fully assigned, leading to so-called "abnormal" values. Other causes are also possible. The problem is not unusual in low-level programming.

How do we avoid the resulting bounded errors and erroneous execution?

In addition to assignment statements, we can safely apply the `Valid` attribute to the object. This language-defined attribute returns a Boolean value indicating whether or not the object's value is a valid representation for the object's subtype. (More details in a moment.) There is no portable alternative to check an object's validity. Here's an example:

```
with Ada.Unchecked_Conversion;
with Ada.Text_IO;  use Ada.Text_IO;
with Interfaces;   use Interfaces;
with System;

procedure Demo_Validity_Check is

   type Toggle_Switch is (Off, On) with Size => 8;
   for Toggle_Switch use (Off => 1, On => 4);

   T1 : Toggle_Switch;

   function Sensor_Reading (Default : Toggle_Switch) return Toggle_Switch is

      function As_Toggle_Switch is new Ada.Unchecked_Conversion
        (Source => Unsigned_8, Target => Toggle_Switch);

      Result : Toggle_Switch;
      Sensor : Unsigned_8;
      --  for Sensor'Address use System'To_Address (...);

   begin
      Result := As_Toggle_Switch (Sensor);
      return (if Result'Valid then Result else Default);
   end Sensor_Reading;

begin
   T1 := Sensor_Reading (Default => Off); -- arbitrary
   Put_Line (T1'Image);
end Demo_Validity_Check;
```

In the above, `Sensor_Reading` is the high-level, functional API provided to clients. The function hides the use of the unchecked conversion, and also hides the memory-mapped hardware interface named `Sensor`. We've commented out the address clause since we don't really have a memory mapped device available. You can experiment with this program by changing the code to assign a value to `Sensor` (e.g., when it is declared). It is an unsigned 8-bit quantity so any value in the corresponding range would be allowed.

In addition to checking for a valid representation, thus preventing the bounded error, Valid also checks that the object is not abnormal, so erroneous execution can be prevented too. (It also checks that any subtype predicate defined for the `Target` type is also satisfied, but that's a lesson for another day.)

However, the `Valid` attribute can be applied only to scalar objects. There is no language-defined attribute for checking objects of composite types. That's because it would be very hard to implement for some types, if not impossible. For example, given a typical run-time model, it is impossible to check the validity of an access value component. Therefore, you must individually check the validity of scalar record or array components.

At least, you would have to check them individually in standard Ada. GNAT defines another Boolean attribute, named `Valid_Scalars`, to check them all for us. This attribute returns **`True`** if the evaluation of `Valid` returns **`True`** for every scalar subcomponent of the enclosing composite type. It also returns **`True`** when there are no scalar subcomponents. See the GNAT RM for more information.

CHAPTER

THREE

MULTI-LANGUAGE DEVELOPMENT

Software projects often involve more than one programming language. Typically that's because there is existing code that already does something we need done and, for that specific code, it doesn't make economic sense to redevelop it in some other language. Consider the rotor blade model in a high-fidelity helicopter simulation. Nobody touches the code for that model except for a few specialists, because the code is extraordinarily complex. (This complexity is unavoidable because a rotor blade's dynamic behavior is so complex. You can't even model it as one physical piece because the tip is traveling so much faster than the other end.) Complex and expensive models like that are a simulator company's crown jewels; their cost is meant to be amortized over as many projects as possible. Nobody would imagine redeveloping it simply because a new project is to be written in a different language.

Therefore, Ada includes extensive facilities to "import" foreign entities into Ada code, and to "export" Ada entities to code in foreign languages. The facilities are so useful that Ada has been used purely as "glue code" to allow code written in two other programming languages to be used together.

You've already seen an introduction to Ada and C code working together in the "Interfacing" section of the Ada introductory course. If you have not seen that material, be sure to see it first. We will cover some further details not already discussed there, and then go into the details of the facilities not covered elsewhere, but we assume you're familiar with it.

The Ada foreign language interfacing facilities include both "general" and "language-specific" capabilities. The "general" facilities are known as such because they are not tied to any specific language. These pragmas and aspects work with any of the supported foreign languages. In contrast, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. For example, as you saw in that "Interfacing" section, there is a package with a number of declarations for C types, such as `int`, `float`, and `double`, as well as C "strings", with subprograms to convert back and forth between them and Ada's string type. Other languages are also supported, both by the Ada Standard and by vendor additions. You will frequently use both the "general" and the "language-specific" facilities together.

All these interfacing capabilities are defined in Annex B of the language standard. Note that Annex B is not a "Specialized Needs" annex, unlike some of the other annexes. The Specialized Needs annexes are wholly optional, whereas all Ada implementations must implement Annex B. However, some parts of Annex B are optional, so more precisely we should say that every implementation must support all the required features of Annex B. That comes down mainly to the package `Interfaces` (more on that package in a moment). However, if an implementation does implement any optional part of Annex B, it must be implemented as described by the standard, or with less functionality. An implementation cannot use the same name for some facility (aspect, etc.) but with different semantics. That's true of the Specialized Needs annexes too: not every part need be implemented, but any part that is implemented must conform to the standard. In practice, for Annex B, all implementations provide the required parts, but not all provide support for all the "language-specific" foreign languages' interfaces. The vendors make a business decision for the optional parts, just as they do regarding the Specialized Needs annexes.

3.1 General Interfacing

In the "Interfacing" section of the Ada introductory course you saw that Ada defines aspects and pragmas for working with foreign languages. These aspects and pragmas are functionally interchangeable, and we will use whichever one of the two that is most convenient in our discussion. The pragmas are officially "obsolescent," but that merely means that a newer approach is available, in this case the corresponding aspects. You can use either one without concern for future support because language constructs that are obsolescent are not removed from the language. Any compiler that supports such constructs will almost certainly support them forever, for the sake of not invalidating existing customers' code. The pragmas have been in the language since Ada 95 so there's a lot of existing code using them. Changing the compiler isn't cost-free, after all, so why spend the money to potentially lose a customer? Likewise, a brand new compiler will also probably support them, for the sake of potentially gaining a customer.

The general interfacing facility consists of these aspects and pragmas, specifically `Import`, `Export`, and `Convention`. As you saw in the Ada Introduction course, `Import` brings a foreign entity into Ada code, `Export` does the opposite, and `Convention` supplies additional information and directives to the compiler. We will go into the details of each.

Regardless of whether the Ada code is importing or exporting some entity, there will be an Ada declaration for that entity. That declaration tells the compiler how the entity can be used, as usual. The interfacing aspects and pragmas are then applied to these Ada declarations.

If we are exporting, then the entity is implemented in Ada. For a subprogram that means there will also be a subprogram body matching the declaration, and the compiler will enforce that requirement as usual. In contrast, if we are importing a subprogram, then it is not implemented in Ada, and therefore there will be no corresponding subprogram body for the Ada declaration. The compiler would not allow it if we tried. In that case the `Import` is the subprogram's completion.

Subprograms often have a separate declaration. Sometimes that's required, for example when we want to include a subprogram as part of a package's API, but at other times it is optional. Remember that a subprogram body acts as a corresponding declaration when there is no separate declaration defined. Thus, either way, we have a subprogram declaration available for the interfacing aspects and/or pragmas.

For data that are imported or exported, we'll have the declaration of the object in Ada to which we can apply the necessary interfacing aspects/pragmas. But we will also have the types for these objects, and as you will see, the types can be part of interfacing too.

3.1.1 Aspect/Pragma Convention

As you saw in the "Interfacing" section of the Ada introductory course, when importing and exporting you'll also specify the "convention" for the entity in question. The pragmas for importing and exporting include a parameter for this purpose. When using the aspects, you'll specify the `Convention` aspect too.

For types, though, you will specify the `Convention` aspect/pragma alone, without `Import` or `Export`. In this case the convention specifies the layout for objects of that type, presumably a layout different than the Ada compiler would normally use. You would need to specify this other layout either because you're going to later declare and export an object of the type, or because you are going to declare an object of the type and pass it as a argument to an imported subprogram.

For example, Ada specifies that multi-dimensional arrays are represented in memory in row-major order. In contrast, the Fortran standard specifies column-major order. If we want to define a type in Ada that can be used for passing parameters to Fortran routines, we need to specify that convention for the type. For example:

```
type Matrix is array (Rows, Columns) of Float
  with Convention => Fortran;
```

(Rows and Columns are user-defined discrete subtypes.)

As a result when we declare Matrix objects the Ada compiler will use the column-major layout. That makes it possible to pass objects of the type to imported Fortran subprograms because the formal parameter will also be of type Matrix. The imported Fortran routine will then see the parameter in memory as it expects to see it. So although you wouldn't need to import or export a type itself, you might very well import or export an object of the type, or pass it as a argument.

When Convention is applied to subprograms, a natural mistake is to think that we are specifying the programming language used to implement the subprogram. In reality, the convention indicates the subprogram calling convention, not the implementation language. The calling convention specifies how parameters are passed to and from subprogram calls, how result values for functions are returned, the order that parameters are pushed on the call stack, how dynamically-sized parameters are passed, and so on. Ordinarily these are matters you don't need to consider because you're working within a single convention automatically, in other words the one used by the Ada compiler you're using.

To illustrate that the convention is not the implementation language, consider a subprogram that we intend to import and call from Ada. This imported routine is implemented in assembly language, but, in addition, let's say it is written to use the same calling convention as the Ada compiler we are using for Ada code. Therefore, the calling convention would be Ada even though the implementation is in assembler.

```
procedure P (X : Integer) with
   ...
   Convention => Ada,
   ...
```

In the example above, Ada is known as a convention identifier, as is Fortran in the earlier example. Convention identifiers are defined by the Ada language standard, but also by Ada vendors.

The Ada standard defines two convention identifiers: Ada (the default), and Intrinsic. In addition, Annex B defines convention identifiers C, COBOL, and Fortran. Support for these Annex B conventions is optional.

GNAT supports the standard and Annex B conventions, as well as the following: Assembler, "C_PLUS_PLUS" (or CPP), Stdcall, WIN32, and a few others. C_PLUS_PLUS is the convention identifier required by the standard when C++ is supported. (Convention identifiers are actual identifiers, not strings, so they must obey the syntax rules for identifiers. "C++" would not be a valid identifier.) See the GNAT User Guide for those other GNAT-specific conventions.

Stdcall and WIN32 actually do specify a particular calling convention, but for those convention identifiers that are language names, how do we get from the name to a calling convention?

The ultimate requirement for any calling convention is compatibility with the Ada compiler we are using. Specifically, the Ada compiler must recognize what the calling convention specifies, and support importing and exporting subprograms with that convention applied.

For the Ada convention that's simple. There is no standard calling convention for Ada. Convention Ada simply means the calling convention applied by the Ada compiler we happen to be using. (We'll talk about Intrinsic shortly.)

So far, so good. But how to we get from those other language names to corresponding calling conventions? There is no standard calling convention for, say, C, any more than there is a standard calling convention for Ada.

In fact we don't get to the calling convention, at least not directly. What the language name in the convention identifier actually tells us is that, when that convention is supported, there is a compiler for that foreign language that uses a calling convention known to, and supported by, the Ada compiler we are using. The Ada compiler vendor defines which languages it supports, after all. For example, when supported, convention C means that there is a compatible C compiler known to the Ada compiler vendor. For GNAT you can guess which C compiler that might be.

It's actually pretty straightforward once you have the big picture. If the convention is supported, the Ada compiler in use knows of a compiler for that language with which it can work. Annex B just defines some convention identifiers for the sake of portability.

But suppose a given Ada compiler supports more than one vendor for a given programming language? In that case the Ada compiler would define and support multiple convention identifiers for the same programming language. Presumably these identifiers would be differentiated by the compiler vendors' names. Thus we might have available conventions `GNU_Fortran` and `Intel_Fortran` if both were supported. The `Fortran` convention identifier would then indicate the default vendor's compiler.

The `Intrinsic` calling convention represents subprograms that are "built in" to the compiler. When such a subprogram is called the compiler doesn't actually generate the code for an out-of-line call. Instead, the compiler emits the assembly code — often just a single instruction — corresponding to the intrinsic subprogram's name. There will be a separate declaration for the subprogram, but no actual subprogram body containing a sequence of statements. The compiler just knows what to emit in place of the call.

For example:

```
function Shift_Left
  (Value  : Unsigned_16;
   Amount : Natural)
  return Unsigned_16
    with ..., Convention => Intrinsic;
```

The effect is much like a subprogram call that is always in-lined, except that there's no body for the subprogram. In this example the compiler simply issues a shift-left instruction in assembly language.

You'll see the `Intrinsic` convention applied to many language-defined subprograms. For example:

```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target
   with ..., Convention => Intrinsic;
```

Thus when we call an instantiation of `Ada.Unchecked_Conversion` there is no actual call made to some subprogram. The compiler just treats the bits of S as a value of type `Target`.

Intrinsic subprograms are a good way to access interesting capabilities of the target hardware, without having to write the assembly language yourself (although we will show how to do that, later, directly in Ada). For example, some targets provide an instruction that atomically compares and swaps a value in memory. Ada 2022 just added a standard package for this, but before that we could use the following to access a gcc built-in:

```
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
-- Returns an indication of whether the swap took place.

function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
```

(continues on next page)

(continued from previous page)

```
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
   return Boolean
with Convention => Intrinsic,
     ...
```

We would specify additional aspects beyond that of `Convention` but these have not yet been discussed. That's what the ellipses indicate in the various examples above.

3.1.2 Aspect/Pragma Import and Export

You've already seen these aspects in the Ada Introduction course, but for completeness: `Import` brings a foreign entity into Ada code, and `Export` makes an Ada entity available to foreign code. In practice, these entities consist of objects and subprograms, but the language doesn't impose many restrictions. It is up to the vendor to decide what makes sense for their specific target.

The aspects `Import` and `Export` are so-called Boolean aspects because their value is either **`True`** or **`False`**. For example:

```
Obj : Matrix with
   Export => True,
   ...
```

For any Boolean-valued aspect the default is **`True`** so you only need to give the value explicitly if that value is **`False`**. There would be no point in doing that in these two cases, of course. Hence we just give the aspect name:

```
Obj : Matrix with
   Export,
   ...
```

Recall that objects of some types are initialized automatically during the objects' elaboration, unless they are explicitly initialized as part of their declarations. Access types are like that, for example. Objects of these types are default initialized to **`null`** as part of ensuring that their values are always meaningful (absent unchecked conversion).

```
type Reference is access Integer;

Obj : Reference;
```

In the above the value of `Obj` is **`null`**, just as if we had explicitly set it that way.

But that initialization is a problem if we are importing an object of an access type. Presumably the value is set by the foreign code, so automatic initialization to null would overwrite the incoming value. Therefore, the language guarantees that implicit initialization won't be applied to imported objects.

```
type Reference is access Integer;

Obj : Reference with Import;
```

Now the value of `Obj` is whatever the foreign code sets it to, and is not, in other words, overwritten during elaboration of the declaration.

3.1.3 Aspect/Pragma External_Name and Link_Name

For an entity with a `True` `Import` or `Export` aspect, we can also specify a so-called external name or link name. These names are specified via aspects `External_Name` and `Link_Name` respectively.

An external name is a string value indicating the name for some entity as known by foreign language code. For an entity that Ada code imports, this is the name that the foreign code declares it to be. For an entity that Ada code exports, this is the name that the foreign code is told to use. This string value is exactly the name to be used, so if you misspell the name the link will fail. For example:

```
function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean
with
  Import,
  Convention    => Intrinsic,
  External_Name => "__sync_bool_compare_and_swap_1";
```

The `External_Name` and `Link_Name` values are strings because the foreign unit names don't necessary follow the Ada rules for identifiers (the leading underscores in this case). Note that the ending digit in the name above is different from the declared Ada name.

Usually, the name of the imported or exported entity is precisely known and hence exactly specified by `External_Name`. Sometimes, however, a compilation system may have a linker "preprocessor" that augments the name actually used by the linkage step. For example, an implementation might always prepend "_" and then pass the result to the system linker. In that case we don't want to specify the exact name. Instead, we want to provide the "starting point" for the name modification. That's the purpose of the aspect `Link_Name`.

If you don't specify either `External_Name` or `Link_Name` the compilation system will choose one in some implementation-defined manner. Typically this would be the entity's defining name in the Ada declaration, or some simple transformation thereof. But usually we know the name exactly and so we use `External_Name` to give it.

As you can see, it really wouldn't make sense to specify both `External_Name` and `Link_Name` since the semantics of the two conflict. But if both are specified for some reason, the `External_Name` value is ignored.

Note that `Link_Name` cannot be specified for `Intrinsic` subprograms because there is no actual unit being linked into the executable, because intrinsics are built-in. In this case you must specify the `External_Name`.

Finally, because you will see a lot the pragma usage we should go into enough detail so that you know what you're looking at when you see them.

Pragma `Import` and pragma `Export` work almost like a subprogram call. Parameters cannot be omitted unless named notation is used. Reordering the parameters is not permitted, however, unlike subprogram calls.

The BNF syntax is as follows. We show Import, but Export has identical parameters:

```
pragma Import(
    [Convention =>] convention_identifier,
    [Entity =>] local_name
 [, [External_Name =>] external_name_string_expression]
 [, [Link_Name =>] link_name_string_expression]);
```

As you can see, the parameters correspond to the individual aspects `Convention`, `External_Name`, and `Link_Name`. When using aspects you don't need to say which Ada entity

you're applying the aspects to, because the aspects are part of the entity declaration syntax. In contrast, the pragma is distinct from the declaration so we must specify what's being imported or exported via the `Entity` parameter. That's the declared Ada name, in other words. Note that both the `External_Name` and `Link_Name` parameters are optional.

Here's that same built-in function, using the pragma to import it:

```
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
-- Returns an indication of whether the swap took place.

function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
   return Boolean;

pragma Import (Intrinsic,
               Sync_Val_Compare_And_Swap_Bool_8,
               "__sync_bool_compare_and_swap_1");
```

The first pragma parameter is for the convention. The next parameter, the `Entity`, is the Ada unit's declared name. The last parameter is the external name. The compiler either knows what we are referencing by that external name or it will reject the pragma. As we mentioned before, the string value for the name is not required to match the Ada unit name.

You will see later that there are other convention identifiers as well, but we will wait for the *Specific Interfacing section* (page 51) to introduce those.

3.1.4 Package Interfaces

Package `Interfaces` must be provided by all Ada implementations. The package is intended to provide types that reflect the actual numeric types provided by the target hardware. Of course, the standard has no way to know what hardware is involved, therefore the actual content is implementation-defined. But even so, it is possible to standardize the names for these types, and that is what the language standard does.

Specifically, the standard defines the format for the names for the hardware's signed and modular (unsigned) integer types, and for the floating-point types.

The signed integers have names of the form `Integer_n`, where *n* is the number of bits used by the machine-supported type. The type for an eight-bit signed integer would be named `Integer_8`, for example, and then `Integer_16` and so on for the larger types, for as many as the target machine supports.

Likewise, for the unsigned integers, the names are of the form `Unsigned_n`, with the same meaning for *n*. The colloquial eight-bit "byte" would be named `Unsigned_8`, with `Unsigned_16` for the 16-bit version, and so on, again for as many as the machine supports.

For floating-point types it is harder to talk about a format that is sufficiently common to standardize. The IEEE floating-point standard is well known and widely used, however, so if the machine does support the IEEE format that name can be used. Such types would be named `IEEE_Float_n`, again with the same meaning for *n*. Thus we might see declarations for types `IEEE_Float_32` and `IEEE_Float_64` and so on, for all the machine supported floating-point types.

In addition to these type declarations, for the unsigned integers only, there will be declarations for shift and rotate operations provided as intrinsic functions.

The resulting package declaration might look something like this:

```
package Interfaces is
```

(continues on next page)

(continued from previous page)

```
   type Integer_8  is range -2 ** 7 .. 2 ** 7 - 1;

   type Integer_16 is range -2 ** 15 .. 2 ** 15 - 1;

   type Integer_32 is range -2 ** 31 .. 2 ** 31 - 1;

   ...

   type Unsigned_8 is mod 2 ** 8;

   function Shift_Left   (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
   function Shift_Right  (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
   function Rotate_Left  (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
   function Rotate_Right (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
   function Shift_Right_Arithmetic (Value : Unsigned_8;  Amount : Natural)
     return Unsigned_8;

   type Unsigned_16 is mod 2 ** 16;

   function Shift_Left  (Value : Unsigned_16;  Amount : Natural)
     return Unsigned_16;
   function Shift_Right (Value : Unsigned_16;  Amount : Natural)
     return Unsigned_16;
   ...

   type Unsigned_32 is mod 2 ** 32;

   function Shift_Left  (Value : Unsigned_32;  Amount : Natural)
     return Unsigned_32;
   function Shift_Right (Value : Unsigned_32;  Amount : Natural)
     return Unsigned_32;
   ...

   type IEEE_Float_32 is digits 6;
   type IEEE_Float_64 is digits 15;
   ...

end Interfaces;
```

As you can see, when you need to write code in terms of the hardware's numeric types, this package is a great resource. There's no need to declare your own `UInt32` type, for example, although of course you could, trivially:

```
type UInt32 is mod 2 ** 32;
```

But if you do, realize that you won't get the shift and rotate operations for your type. Those are only defined for the types in package `Interfaces`. If you do need to declare such a type, and you do want the additional shift/rotate operations, use inheritance:

```
type UInt32 is new Interfaces.Unsigned_32;
```

GNAT also defines a pragma, as an alternative to inheritance:

```
type UInt32 is mod 2 ** 32;
pragma Provide_Shift_Operators (UInt32);
```

The approach using inheritance is preferable because it is portable, all other things being equal.

One reason to make up your own unsigned type is that you need one that does not in fact reflect the target hardware's numeric types. For example, a hardware device register might have gaps of bits that are currently not used by the device. Those gaps are frequently not

the size of a type declared in package `Interfaces`. We might need an `Unsigned_3` type, for example. That's a reasonable thing to do.

3.2 Language-Specific Interfacing

In addition to the aspects and pragmas for importing and exporting entities that work with any language, Ada also defines standard language-specific facilities for interfacing with a set of foreign languages. The standard defines which languages, but vendors can (and do) expand the set.

Specifically, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. Package `Interfaces` is the root package for a hierarchy of packages that organize these declarations by language, with one or more child packages per language.

Note that the declarations within package `Interfaces` are, by definition, compile-time visible to any child package in the subsystem. Thus whenever one of the language-specific packages needs to mention the machine types they are automatically available.

The standard defines specific support for foreign languages C, COBOL, and Fortran. Thus there are one or more child packages rooted at `Interfaces` that have those language names as their child package names: `Interfaces.C`, `Interfaces.COBOL`, and `Interfaces.Fortran`.

The material below will focus on C and, to a lesser extent, Fortran, ignoring altogether the support for COBOL. That's not because COBOL is unimportant. There is a lot of COBOL business software out there in use. Rather, we skip COBOL because it is not relevant to embedded systems. Similarly, although Fortran is extensively used, especially in high-performance computing, it is not used extensively in embedded systems. We will provide some information about the Fortran support but will not dwell on it.

Even though we do not consider C to be appropriate for large development projects, neither technically not economically, it has its place in small, low-criticality embedded systems. Ada developers can profit from existing device drivers and mature libraries coded in C, for example. Hence interfacing to it is important.

What about C++? Interfacing to C++ is tricky compared to C, because of the vendor-defined name-mangling, automatic invocations of constructors and destructors, exceptions, and so on. Generally, interfacing with C++ code can be facilitated by preventing much of those difficulties using the `extern "C" {... }` linkage-specification. Doing so then makes the bracketed C++ code look like C, so the C interfacing facilities then can be used.

3.2.1 Package Interfaces.C

The child package `Interfaces.C` supports interfacing with units written in the C programming language. Support is in the form of Ada constants and types, and some subprograms. The constants correspond to C's `limits.h` header file, and the Ada types correspond to types for C's `int`, `short`, `unsigned_short`, `unsigned_long`, `unsigned_char`, `size_t`, and so on. There is also support for converting Ada's type `String` to/from `char_array`, and similarly for type `Wide_String`, etc.

It's a large package so we will elide parts. The idea is to give you a feel for what's there. If you want the details, see either the Ada reference manual or bring up the source code in GNAT Studio.

```
package Interfaces.C is

   --  Declaration's based on C's <limits.h>

   CHAR_BIT  : constant := 8;
```

(continues on next page)

(continued from previous page)

```
   SCHAR_MIN : constant := -128;
   SCHAR_MAX : constant := 127;
   UCHAR_MAX : constant := 255;

   --  Signed and Unsigned Integers. Note that in GNAT, we have ensured that
   --  the standard predefined Ada types correspond to the standard C types

   type int   is new Integer;
   type short is new Short_Integer;
   type long  is range -(2 ** (System.Parameters.long_bits - Integer'(1)))
     .. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;
   type long_long is new Long_Long_Integer;

   type signed_char is range SCHAR_MIN .. SCHAR_MAX;
   for signed_char'Size use CHAR_BIT;

   type unsigned           is mod 2 ** int'Size;
   type unsigned_short     is mod 2 ** short'Size;
   type unsigned_long      is mod 2 ** long'Size;
   type unsigned_long_long is mod 2 ** long_long'Size;

   ...

   --  Floating-Point

   type C_float     is new Float;
   type double      is new Standard.Long_Float;
   type long_double is new Standard.Long_Long_Float;

   ----------------------------
   -- Characters and Strings --
   ----------------------------

   type char is new Character;

   nul : constant char := char'First;

   function To_C   (Item : Character) return char;
   function To_Ada (Item : char)      return Character;

   type char_array is array (size_t range <>) of aliased char;
   for char_array'Component_Size use CHAR_BIT;

   ...

end Interfaces.C;
```

The primary purpose of these types is for use in the formal parameters of Ada subprograms imported from C or exported to C. The various conversion functions can be called from within Ada to manipulate the actual parameters.

When writing the Ada subprogram declaration corresponding to a C function, an Ada procedure directly corresponds to a void function. An Ada procedure also corresponds to a C function if the return value is always to be ignored. Otherwise, the Ada declaration should be a function.

As we said, the types declared in this package can be used as the formal parameter types. That is the intended and recommended approach. However, some Ada types naturally correspond to C types, and you might see them used instead of those from `Interfaces.C`. Type `int` is the C native integer type for the target, for example, as is type **`Integer`** in Ada. Likewise, C's type **`float`** and type Ada's **`Float`** are likely compatible. GNAT goes to some lengths to maintain compatibility with C, since the two gcc compilers share so much internal

technology. Other vendors might not do so. Best practice is use the types in `Interfaces.C` for your parameters.

Of course, the types in `Interfaces.C` are not sufficient for all uses. You will often need to use user-defined types for the formal parameters, such as enumeration types and record types.

Ada enumeration types are compatible with C's enums but note that C requires enum values to be the size of an `int`, whereas Ada does not. The Ada compiler uses whatever sized machine type will support the specified number of enumeral values. It might therefore be smaller than an `int` but it might also be larger. (Declaring more enumeration values than would fit in an integer is unlikely except in tool-generated code, but it is possible.) For example:

```
type Small_Enum is (A, B, C);
```

If we printed the object size for `Small_Enum` we'd get 8 (on a typical machine with GNAT). Therefore, applying the aspect `Convention` to the Ada enumeration type declaration is a good idea:

```
type Small_Enum is (A, B, C) with Convention => C;
```

Now the object size will be 32, the same as `int`.

Speaking of enumeration types, note that Ada 2022 added a boolean type to `Interfaces.C` named `C_Bool` to match that of C99, so you should use it instead of Ada's **`Boolean`** type for formal parameters.

A simple Ada record type is compatible with a C struct, but remember that the Ada compiler is allowed to reorder the record components. The compiler would do that if it saw that the layout was inefficient, but the point here is that the compiler could do it silently. As a result, you should specify the record layout explicitly using a record representation clause, matching the layout of the C struct in question. Then there will be no question of the layouts matching. Once your record types get more complicated, for example with discriminants or tagged record extensions, things get tricky. Your best bet it to stick with the simple cases when interfacing to C.

Some types that you might think would correspond do not, at least not necessarily. For example, an Ada access type's value might be represented as a simple address, but it might not. In GNAT, an access value designating a value of some unconstrained array type (e.g., **`String`**) is comprised of two addresses, by default. One designates the characters and the other designates the bounds. You can override that with a pragma, but you must know to do so. For example, if we run the following program, we will see that the object size for the access type `Name` is twice the object size of `System.`**`Address`**:

```
with Ada.Text_IO;  use Ada.Text_IO;
with System;       use System;

procedure Demo is

   type Name is access String;

begin
   Put_Line (Address'Object_Size'Image);
   Put_Line (Name'Object_Size'Image);
end Demo;
```

Some Ada types simply have no corresponding type in C, such as record extensions, task types, and protected types. You'll have to pass those as an "opaque" type, usually as an address. It isn't clear that a C function would know what to do with values of these types, but the general notion of passing an opaque type as an address is useful and not uncommon. Of course, that approach forgoes all type safety, so avoid it when possible.

In addition to the types for the formal parameters, you'll also need to know how parameters are passed to and from C functions. That affects the parameter profiles on both sides, Ada and C. The text in Annex B for `Interfaces.C` specifies how parameters are to be passed back and forth between Ada and C so that your subprogram declarations can be portable. That's the approach for each supported programming language, i.e., in the discussion of the corresponding child package under `Interfaces`.

The rules are expressed in terms of scalar types, "elementary" types, array types, and record types. Remember that scalar types are composed of the discrete types and the real types, so we're talking about the signed and modular integers, enumerations, floating-point, and the two kinds of fixed-point types. The "elementary" types consist of the scalars and access types. The rules are fairly intuitive, but throw in Ada's access parameters and parameter modes and some subtleties arise. We won't cover all the various rules but will explore some of the subtleties.

First, the easy cases: mode **in** scalar parameters, such as `int`, as simply passed by copy. Scalar parameters are passed by copy anyway in Ada so the mechanism aligns with C in a straightforward manner. A record type T is passed by reference, so on the C side we'd see `t*` where `t` is a C struct corresponding to T. A constrained array type in Ada with a component type T would correspond to a C formal parameter `t*` where `t` corresponds to T. An Ada access parameter **access** T corresponds on the C side to `t*` where `t` corresponds to T. And finally, a private type is passed according to the full definition of the type; the fact that it is private is just a matter of controlling the client view, being private doesn't affect how it is passed. There are other simple cases, such as access-to-subprogram types, but we can leave that to the Annex.

Now to the more complicated cases. First, some C ABIs (application binary interfaces) pass small structs by copy instead of by reference. That can make sense, in particular when the struct is small, say the size of an address or smaller. In that case there's no performance benefit to be had by passing a reference. When that situation applies, there is another convention we have not yet mentioned: `C_Pass_By_Copy`. As a result the record parameter will be passed by copy instead of the default, by reference (i.e., T rather than *T), as long as the mode is **in**. For example:

```
type R2 is record
   V : int;
end record
with Convention => C_Pass_By_Copy;

procedure F2 (P : R2) with
   Import,
   Convention => C,
   External_Name => "f2";
```

```
struct R2 {
   int V;
};

void f2 (R2 p);
```

On the C side we expect that p is passed by copy and indeed that is how we find it. That said, passing record values to structs by reference is the more common programmer choice. Like arrays, records are typically larger than an address. The point here is that the Ada code can be configured easily to match the C code.

Next, consider passing array values, both to and from C. When passing an array value to C, remember that Ada array types have bounds. Those bounds are either specified at compile time when they are declared, or, for unconstrained array types, specified elsewhere, at run-time.

Array types are not first-class types in C, and C has no notion of unconstrained array types, or even of upper bounds. Therefore, passing an unconstrained array type value is interest-

ing. One approach is to avoid them. Instead, declare a sufficiently large constrained array as a subtype of the unconstrained array type, and then just pass the actual upper bound you want, along with the array object itself.

```
type List is array (Integer range <>) of Interfaces.C.int;

subtype Constrained_List is List (1 .. 100);

procedure P (V : Constrained_List;  Size : Interfaces.C.int);
pragma Import (C, P, "p");

Obj : Constrained_List := (others => 42);  -- arbitrary values
```

With that, we can just pass the value by reference as usual on the C side:

```
void p (int* v, int size) {
  // whatever
}
```

But that's assuming we know how many array components are sufficient from the C code's point of view. In the example above we'll pass a value up to 100 to the `Size` parameter and hope that is sufficient.

Really, it would work to use the unconstrained array type as the formal parameter type instead:

```
type List is array (Integer range <>) of Interfaces.C.int;

procedure P (V : List;  Size : Interfaces.C.int);
pragma Import (C, P, "p");
```

The C function parameter profile wouldn't change. But why does this work? With values of unconstrained array types, the bounds are stored with the value. Typically they are stored just ahead of the first component, but it is implementation-defined. So why doesn't the above accidentally pass the bounds instead of the first array component itself? It works because we are guaranteed by the Ada language that passing an array will pass (the address of) the components, not the bounds, even for Ada unconstrained array types.

Now for the other direction: passing an array from C to Ada. Here the lack of bounds information on the C side really makes a difference. We can't just pass the array by itself because that would not include the bounds, unlike an Ada call to an Ada routine. In this case the approach is the similar to the first alternative described above, in which we declare a very large array and then pass the bounds explicitly:

```
type List is array (Natural) of int;
--  DO NOT DECLARE AN OBJECT OF THIS TYPE

procedure P (V : List; Size : Interfaces.C.int);
pragma Export (C, P, "p");

procedure P (V : List; Size : Interfaces.C.int) is
begin
   for J in 0 .. Size - 1 loop
      -- whatever
   end loop;
end P;
```

```
extern void p (int* v, int size);

int x [100];

p (x, 100);  // call to Ada routine, passing x
```

The fundamental idea is to declare an Ada type big enough to handle anything conceivably needed on the C side. Subtype `Natural` means `0 .. Integer'Last` so `List` is quite large indeed. Just be sure never to declare an object of that type. You'll probably run out of storage on an embedded target.

Earlier we said that it is the Ada type that determines how parameters are passed, and that scalars and elementary types are always passed by copy. For mode `in` that's simple, the copy to the C formal parameter is done and that's all there is to it. But suppose the mode is instead `out` or `in out`? In that case the presumably updated value must be returned to the caller, but C doesn't do that by copy. Here the compiler will come to the rescue and make it work, transparently. Specifically, we just declare the Ada subprogram's formal parameter type as usual, but on the C formal we use a reference. We're talking about scalar and elementary types so let's use `int` arbitrarily. We make the mode `in out` but `out` would also serve:

```
procedure P (Formal : in out int);
```

```
void function p (int* formal);
```

Now the compiler does its magic: it generates code to make a copy of the actual parameter, but it makes that copy into a hidden temporary object. Then, when calling the C routine, it passes the address of the hidden object, which corresponds to the reference expected on the C side. The C code updates the value of the temporary object via the reference, and then, on return, the compiler copies the value back from the temporary to the actual parameter. Problem solved, if a bit circuitous.

There are other aspects to interfacing with C, such as variadic functions that take a varying number of arguments, but you can find these elsewhere in the learn courses.

Next, we examine the child packages under `Interfaces.C`. These packages are not used as much as the parent `Interfaces.C` package so we will provide an overview. You can look up the contents within GNAT Studio or the Ada language standard.

3.2.2 Package Interfaces.C.Strings

Package `Interfaces.C` declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type `chars_ptr` corresponds to a common use of `char *` in C programs, and an object of this type can be passed to imported subprograms for which `char *` is the type of the argument of the C function. A subset of the package content is as follows:

```
package Interfaces.C.Strings is

   type chars_ptr is private;
   ...

   function New_Char_Array (Chars : in char_array) return chars_ptr;

   function New_String (Str : in String) return chars_ptr;

   procedure Free (Item : in out chars_ptr);
   ...

   function Value (Item : in chars_ptr) return char_array;
   function Value (Item : in chars_ptr) return String;
   ...

   function Strlen (Item : in chars_ptr) return size_t;
```

(continues on next page)

(continued from previous page)

```
   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Chars  : in char_array;
                     Check  : in Boolean := True);

   ...

end Interfaces.C.Strings;
```

Note that allocation might be via malloc, or via Ada's allocator **new**. In either case, the returned value is guaranteed to be compatible with char*. Deallocation must be via the supplied procedure Free.

An amusing point is that you can overwrite the end of the char array just like you can in C, via procedure Update. The Check parameter indicates whether overwriting past the end is checked. The default is **True**, unlike in C, but you could pass an explicit **False** if you felt the need to do something questionable.

3.2.3 Package Interfaces.C.Pointers

The generic package Interfaces.C.Pointers allows us to perform C-style operations on pointers. It includes an access type named Pointer, various Value functions that dereference a Pointer value and deliver the designated array, several pointer arithmetic operations, and "copy" procedures that copy the contents of a source pointer into the array designated by a destination pointer.

We won't go into the details further. See the Ada RM for more.

3.2.4 Package Interfaces.Fortran

Like Interfaces.C, package Interfaces.Fortran defines Ada types to be used when working with subprograms using the Fortran calling convention. These types have representations that are identical to the default representations of the Fortran intrinsic types *Integer*, *Real*, *Double Precision*, *Complex*, *Logical*, and *Character* in some supported Fortran implementation. And like the C package, the ways that parameters of various types are passed are also specified.

We leave the details to you to look up in the language standard, if you find them needed in an embedded application.

3.2.5 Machine Code Insertions (MCI)

When working close to the hardware, especially when interacting with a device, it is not uncommon for the hardware to require a very specific set of assembly language instructions to be generated. There are two ways to achieve this: the right way and the wrong way.

The wrong way is to experiment with the source code and compiler switches until you get the exact assembly code you need generated (assuming it is possible at all). But what happens when the next compiler release arrives with a new optimization? And abandon all hope if you go to a new compiler vendor. This approach is both labor-intensive and very brittle.

The right way is to express the precise assembly code sequence explicitly within the Ada source code. (That's true to any high level language, not just Ada.) Or you can call an intrinsic function, if there is one that does exactly what you need. We will focus on inserting it directly, in what is known as "machine code insertion", or "inline assembler."

As an example of the need for this capability, consider the GPIO (General Purpose I/O) port on an STM32 Arm microcontroller. Each port contains 16 individual I/O pins, each of which can be configured as an independent discrete input or output, or as a control line for a

device, with pull-up or pull-down registers, with different clock speeds, and so on. Different on-chip devices use various collections of pins in ways specific to the devices, and require exclusive assignment of the pins. However, any given pin can be used by several different devices. For example, pin 11 on port A ("PA11") can be used by USART #1 as the clear-to-send ("CTS") line, or the CAN #1 bus Rx line, or Channel 4 of Timer 1, among others. Therefore, one of the responsibilities of the system designer is to allocate pins to devices, ensuring that they are allocated uniquely. It is difficult to debug the case in which a pin is accidentally configured for one device and then reconfigured for use with another device (assuming the first device remains in use). To help ensure exclusive allocations, every GPIO port on this Arm implementation has a way of locking the configuration of each I/O pin. That way, some other part of the software can't successfully change the configuration accidentally, for use with some other device. Even if the same configuration was to be used for another device, the lock prevents the accidental update so we find out about the unintentional sharing.

To lock a pin on a port requires a special sequence of reads and writes to a GPIO register for that port. A specific bit pattern is required during the reads and writes. The sequence and bit pattern is such that accidentally locking the pin is highly unlikely.

Once we see how to express assembly language sequences in general we will see how to get the necessary sequence to lock a port/pin pair. Unfortunately, although you can express exactly the code sequence required, such a sequence of assembly language instructions is clearly target hardware-specific. That means portability is inherently limited. Moreover, the syntax for expressing it varies with the vendor, even for the same target hardware. Being able to insert it at the Ada source level doesn't help with either portability issue. You should understand that the use-case for machine code insertion is for small, short sequences. Otherwise you would write the code in assembly language directly, in a separate file. That might obtain a degree of vendor independence, at least for the given target, but not necessarily. The use of inline assembler is intended for cases in which a separate file containing assembly language is not simpler.

With those caveats in place, let's first examine how to do it in general and then how to express it with GNAT specifically.

The right way to express an arbitrary sequence of one or more assembly language statements is to use so-called "code statements." A code statement is an Ada statement, but it is also a qualified expression of a type defined in package `System.Machine_Code`. The content of that package, and the details of code statements, are implementation-defined. Although that affects portability there really is no alternative because we are talking about machine instruction sets, which vary considerably and cannot be standardized at this level.

Package `System.Machine_Code` contains types whose values provide a way of expressing assembly instructions. For example, let's say that there is a "HLT" instruction that halts the processor for some target. There is no other parameter required, just that op-code. Let's also say that one of the types in `System.Machine_Code` is for these "short" instructions consisting only of an op-code. The syntax for the type declaration would then allow the following code statement:

```
Short_Instruction'(Command => HLT);
```

Each of `Short_Instruction`, `Command`, and `HLT` are defined by the vendor in this hypothetical version of package `System.Machine_Code`. You can see why we say that it is both a statement (note the semicolon) and a qualified expression (note the apostrophe).

Code statements must appear in a subprogram body, after the **`begin`**. Only code statements are allowed in such a body, only use-clauses can be in the declarative part, and no exception handlers are allowed. The complete example would be as follows:

```
procedure Halt -- stops processor
   with Inline;
```

(continues on next page)

(continued from previous page)

```
with System.Machine_Code; use System.Machine_Code;
procedure Halt is
begin
   Short_Instruction'(Command => HLT);
end Halt;
```

With that, to halt the processor the Ada code can simply call procedure `Halt`. When the optimizer is enabled there will be no code emitted to make the call, we'd simply see the halt instruction emitted directly in-line.

Package `System.Machine_Code` provides access to machine instructions but as we mentioned, the content is vendor-defined. In addition, the package itself is optional, but is required if Annex C, the Systems Programming Annex, is implemented by the vendor. In practice most all vendors provide this annex.

In GNAT, the content of `System.Machine_Code` looks something like this:

```
type Asm_Input_Operand  is ...
type Asm_Output_Operand is ...
type Asm_Input_Operand_List  is array (Integer range <>) of Asm_Input_Operand;
type Asm_Output_Operand_List is array (Integer range <>) of Asm_Output_Operand;

type Asm_Insn is private;

...

function Asm
   (Template : String;
    Outputs  : Asm_Output_Operand := No_Output_Operands;
    Inputs   : Asm_Input_Operand  := No_Input_Operands;
    Clobber  : String  := "";
    Volatile : Boolean := False) return Asm_Insn;
```

With this package content, the expression in a code statement is of type Asm_Insn, short for "assembly instruction." Multiple overloaded functions named `Asm` return values of that type.

The `Template` parameter in a string containing one or more assembly language instructions. These instructions are specific to the target machine. The parameter `Outputs` provides mappings from registers to source-level entities that are updated by the assembly statement(s). `Inputs` provides mappings from source-level entities to registers for inputs. `Volatile`, when True, tells the compiler not to optimize the call away, and `Clobber` tells the compiler which registers, or memory, if any, are altered by the instructions in `Template`. ("Clobber" is colloquial English for "destroy.") That last is important because the compiler was likely already using some of those registers so the compiler will need to restore them after the call.

We could say, for example, the following, taking all the defaults except for `Volatile`:

```
Asm ("nop", Volatile => True);
```

As you can imagine the full details are extensive, beyond the scope of this introduction. See the GNAT User Guide ("Inline Assembler") for all the gory details.

Now, back to our GPIO port/bin locking example. The port type is declared as follows:

```
type GPIO_Port is limited record
   ...
   LCKR : Word with Atomic;  -- lock register
   ...
end record with ...
```

We've elided all but the LCKR component representing the "lock register" within each port. We'd have a record representation clause to ensure the required layout but that's not important here. `Word` is an unsigned (modular) 32-bit integer type. One of the hardware requirements for accessing the lock register is that the entire register has to be read or written whenever any bits within it are accessed. The compiler must not, for example, write one of the bytes within the register in order to set or clear a bit within that part of the register. Therefore we mark the register as Atomic. If the compiler cannot honor that aspect the compilation will fail, so we would know there is a problem.

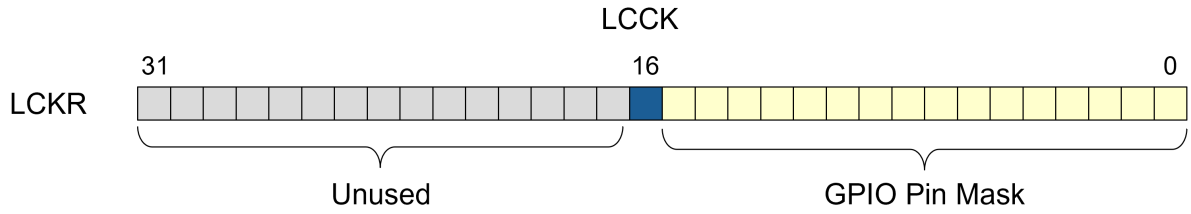
Per the ST Micro Reference Manual, the lock control bit is referred to as LCKK and is bit #16, i.e., the first in the upper half of the LCKR register word.

```
LCCK : constant Word := 16#0001_0000#;  -- the "lock control bit"
```

That bit is also known as the "Lock Key" (hence the abbreviation) because it is used to control the locking of port/pin configurations.

There are 16 GPIO pins per port, represented by the lower 16 bits of the register. Each one of these 16 bits corresponds to one of the 16 GPIO pins on a port. If any given bit reads as a 1 then the corresponding pin is locked.

Graphically that looks like this:



Therefore, the Ada types are:

```
type GPIO_Pin is
   (Pin_0, Pin_1, Pin_2,  Pin_3,  Pin_4,  Pin_5,  Pin_6,  Pin_7,
    Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);

for GPIO_Pin use (Pin_0  => 16#0001#,
                  Pin_1  => 16#0002#,
                  Pin_2  => 16#0004#,
                  ...
                  Pin_15 => 16#8000#);
```

Note that we had to override the default enumeration representation so that each pin — each enumeral value — would occupy a single dedicated bit in the bit-mask.

With that in place, let's lock a pin. A specific sequence is required to set a pin's lock bit. The sequence writes and reads values from the port's LCKR register. Remember that this 32-bit register has 16 bits for the pin mask (0 .. 15), with bit #16 used as the "lock control bit".

1. write a 1 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
2. write a 0 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
3. do step 1 again
4. read the entire LCKR register
5. read the entire LCKR register again (optional)

Throughout the sequence the same value for the lower 16 bits of the word must be maintained (i.e., the pin mask), including when clearing the LCCK bit in the upper half.

If we wrote this in Ada it would look like this:

```
procedure Lock (Port : in out GPIO_Port;  Pin : GPIO_Pin) is
   Temp : Word with Volatile;
begin
   --  set the lock control bit and the pin bit, clear the others
   Temp := LCCK or Pin'Enum_Rep;
   --  write the lock and pin bits
   Port.LCKR := Temp;
   --  clear the lock bit in the upper half
   Port.LCKR := Pin'Enum_Rep;
   --  write the lock bit again
   Port.LCKR := Temp;
   --  read the lock bit
   Temp := Port.LCKR;
   --  read the lock bit again
   Temp := Port.LCKR;
end Lock;
```

`Pin'Enum_Rep` gives us the underlying value for the enumeration value. We cannot use `'Pos` because that attribute provides the logical position number within the enumerated values, and as such always increases consecutively. We need the underlying representation value that we specified explicitly.

The Ada procedure works, but only if the optimizer is enabled (which also precludes debugging). But even so, there is no guarantee that the required assembly language instruction sequence would be generated, especially one that maintains that required bit mask value on each access. A machine-code insertion is appropriate for all the reasons presented earlier:

```
procedure Lock (Port : in out GPIO_Port;
                Pin  :        GPIO_Pin) is
   use System.Machine_Code, ASCII, System;
begin
   Asm ("orr  r3, %1, #65536" & LF & HT &  -- 0) Temp := LCCK or Pin'Enum_Rep
        "str  r3, [%0, #28]"  & LF & HT &  -- 1) Port.LCKR := Temp
        "str  %1, [%0, #28]"  & LF & HT &  -- 2) Port.LCKR := Pin'Enum_Rep
        "str  r3, [%0, #28]"  & LF & HT &  -- 3) Port.LCKR := Temp
        "ldr  r3, [%0, #28]"  & LF & HT &  -- 4) Temp := Port.LCKR
        "ldr  r3, [%0, #28]"  & LF & HT,   -- 5) Temp := Port.LCKR
        Inputs => (Address'Asm_Input ("r", This'Address), -- %0
                   (GPIO_Pin'Asm_Input ("r", Pin))),       -- %1
        Volatile => True,
        Clobber  => ("r3"));
end Lock;
```

We've combined the instructions into one `Asm` expression. As a result, we can use ASCII line-feed and horizontal tab characters to format the listing produced by the compiler so that each instruction is on a separate line and aligned with the previous instruction, as if we had written the sequence in assembly language directly. That enhances readability later, during examination of the compiler output to verify the required sequence was emitted.

In the above, `"%0"` is the first input, containing the address of the `Port` parameter. `"%1"` is the other input, the value of the `Pin` parameter. We're using register `r3` explicitly, as the "temporary" variable, so we tell the compiler that it has been "clobbered."

If we examine the assembly language output from compiling the file, we find the body of procedure `Lock` is as hoped:

```
ldr  r2, [r0, #4]
ldrh r1, [r0, #8]
.syntax unified
orr  r3, r1, #65536
str  r3, [r2, #28]
```

(continues on next page)

(continued from previous page)

```
str  r1, [r2, #28]
str  r3, [r2, #28]
ldr  r3, [r2, #28]
ldr  r3, [r2, #28]
```

The first two statements load register 2 (r2) and register 1 (r1) with the subprogram parameters, i.e., the port and pin, respectively. Register 2 gets the starting address of the port record, in particular. (Offset #28 is the location of the LCKR register. The port is passed by reference so that address is actually that of the hardware device.)

We will have separately declared procedure Lock with inlining enabled, so whenever we call the procedure we will get the exact assembly language sequence required to lock the indicated pin on the given port, without any additional code for a procedure call.

Note that we get the calling convention right automatically, because the subprogram is not a foreign entity written in some other language (such as assembly language). It's an Ada subprogram with special content so the Ada convention applies as usual.

3.3 When Ada Is Not the Main Language

When multiple programming languages are involved, the main procedure might not be implemented in Ada. Maybe the bulk of the program is written in C, for example, and this C code calls some Ada routines that have been exported (with the C convention).

That means the Ada builder does not create the executable image's entry point. In fact the Ada main procedure is never the entry point for the final executable image, it's just where the application code begins, like the C `main` function. There are setup and initialization steps that must happen before any program can execute on a target, and the entry point code is responsible for this functionality. For example, on a bare machine target, the hardware must be initialized, the trap vectors installed, the segments initialized, and so on. On a target running an operating system, the OS is responsible for that initialization but there will be OS-specific initialization steps too. For example, if command-line arguments are supported these may be gathered. All this initialization code is generated by the builder, regardless of the language, followed by a call to the main routine.

Some of the initialization is specific to Ada programming, and must occur before any calls occur to the exported Ada routines. In particular, the entry point code emitted by the Ada builder initializes the Ada run-time system and calls all the elaboration routines for the library units in the application code. Only then does the emitted code invoke the Ada main. If the Ada builder is not going to create the executable it has no chance to emit the code to do that prior initialization. A foreign language builder will not emit such code, so we have a problem.

You could learn enough about how the foreign builder works, and how your Ada builder works, to create a work-around. You could learn what the Ada builder would emit, in other words, and ensure those routines are called manually, either directly or by augmenting the builder scripts (assuming that's possible). But the work-around would be labor-intensive and not robust to changes by the tool vendors. It would be an ugly hack, in other words.

That work-around would not be portable either. The Ada standard can't address hardware- or OS-specific initialization, but it can standardize the name for a routine to do the Ada-specific initialization. Specifically, procedure `adainit` initializes the Ada application code and the Ada run-time library. Similarly, one might need to shut down the Ada code when no further calls will be made to the exported Ada routines. Procedure `adafinal` performs this shut-down functionality. Neither procedure has parameters.

The main function in the other language is intended to import these routines and manually call them each exactly once. `adainit` must be called prior to any calls to the Ada code, and `adafinal` is to be called after all the calls to the Ada code.

For example:

```
#include "stdio.h"

extern int checksum (char *input, int count);

extern void adainit (void);
extern void adafinal (void);

int main (int argc, char *argv[]) {
   char * Str = "Hello World!";
   int sum;
   adainit ();
   sum = checksum (Str, strlen (Str));
   adafinal ();
   printf ("checksum for '%s' is %d", Str, sum);
   return 0;
}
```

In the above, we have an Ada routine to compute a checksum, called by a C main function. Therefore, we use "extern" to tell the C compiler that the "checksum" function is defined elsewhere, i.e., in the Ada routine. Likewise, we tell the compiler that functions `adainit` and `adafinal` are defined elsewhere. The call to `adainit` is made before the call to any Ada code, thus all the elaboration code is guaranteed to happen before `checksum` needs it. Once the Ada code is not needed, the call to `adafinal` can be made.

Both `adainit` and `adafinal` have no effect after the first invocation. That means you cannot structure your foreign code to iteratively call the two routines whenever you want to invoke some Ada code. In practice you just call them once in the main and be done with it.

CHAPTER

FOUR

INTERACTING WITH DEVICES

Interacting with hardware devices is one of the more frequent activities in embedded systems programming. It is also one of the most enjoyable because you can make something happen in the physical world. There's a reason that making an LED blink is the "hello world" of embedded programming. Not only is it easy to do, it is surprisingly satisfying. I suspect that even the developers of "Full Authority Digital Engine Controllers" (FADEC) — the computers that are in complete, total control of commercial airline engines — have fond memories of making an LED blink early in their careers. And of course a blinking LED is a good way to indicate application status, especially if off-board I/O is limited, which is often the case.

Working at the device register level can be error prone and relatively slow, in terms of source-lines-of-code (SLOC) produced. That's partly because the hardware is in some cases complicated, and partly because of the way the software is written. Using bit masks for setting and clearing bits is not a readable approach, comparatively speaking. There's just not enough information transmitted to the reader. It might be clear enough when written, but will you see it that way months later? Readability is important because programs are read many more times than they are written. Also, an unreadable program is more difficult to maintain, and maintenance is where most money is spent in long-lived applications. Comments can help, until they are out of date. Then they are an active hindrance.

For example, what do you think the following code does? This is real code, where `temp` and `temp2` are unsigned 32-bit integers:

```
temp = ((uint32_t)(GPIO_AF) <<
          ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
          ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
```

That's unfair to ask, absent any context. The code configures a general purpose I/O (GPIO) pin on an Arm microcontroller for one of the "alternate functions". `GPIOx` is a pointer to a GPIO port, `GPIO_PinSource` is a GPIO pin number, and `GPIO_AF` is the alternate function number. But let's say you knew that. Is the code correct? The longer it takes to know, the less productive you are.

The fact that the code above is in C is beside the point. If we wrote it the same way in Ada it would be equally opaque, if not more so. There are simpler approaches. Judicious use of record and array types is one. We'll say more about that later, but the underlying idea is to let the compiler do as much work for us as possible. For example, the data structures used in the code above require explicit shifting whenever they are accessed. If we can avoid that at the source code level — by having the compiler do it for us — we will have simplified the code considerably. Furthermore, letting the compiler do the work for us makes the code more maintainable (which is where the money is). For example, if the code does the shifting explicitly and the data structures are changed, we'll have to change the number of bits to shift left or right. Constants will help there, but we still have to remember to change them;

the compiler won't complain if we forget. In contrast, if we let the compiler do this shifting for us, the amounts to shift will be changed automatically.

Some devices are very simple. In these cases the application may interact directly with the device without unduly affecting productivity. For example, there was a board that had a user-accessible rotary switch with sixteen distinct positions. Users could set the switch to whatever the application code required, e.g., to indicate some configuration information. The entire software interface to this device consisted of a single read-only 8-bit byte in memory. That's all there was to it: you read the memory and thus got the numeric setting of the switch.

More complex devices, however, usually rely on software abstraction to deal with the complexity. Just as abstraction is a fundamental way to combat complexity in software, abstraction also can be used to combat the complexity of driving sophisticated hardware. The abstraction is presented to users by a software "device driver" that exists as a layer between the application code and the hardware device. The layer hides the gory details of the hardware manipulation behind subprograms, types, and parameters.

We say that the device driver layer is an abstraction because, at the least, the names of the procedures and functions indicate what they do, so at the call site you can tell *what* is being done. That's the point of abstraction: it allows us to focus on what, rather than how. Consider that GPIO pin configuration code block again. Instead of writing that block every time we need to configure the alternate function for a pin, suppose we called a function:

```
GPIO_PinAFConfig(USARTx_TX_GPIO_PORT, USARTx_TX_SOURCE, USARTx_TX_AF);
```

The `GPIO_PinAFConfig` function is part of the GPIO device driver provided by the STM32 Standard Peripherals Library (SPL). Even though that's not the best function name conceivable, calls to the function will be far more readable than the code of the body, and we only have to make sure the function implementation is correct once. And assuming the device drivers' subprograms can be inlined, the subprogram call imposes no performance penalty.

Note the first parameter to the call above: `USARTx_TX_GPIO_PORT`. There are multiple GPIO ports on an Arm implementation; the vendor decides how many. In this case one of them has been connected to a USART (Universal Synchronous Asynchronous Receiver Transmitter), an external device for sending and receiving serial data. When there are multiple devices, good software engineering suggests that the device driver present a given device as one of a type. That's what an "abstract data type" (ADT) provides for software and so the device driver applies the same design. An ADT is essentially a class, in class-oriented languages. In Ada, an ADT is represented as a private type declared in a package, along with subprograms that take the type as a parameter.

The Ada Drivers Library (ADL) provided by AdaCore and the Ada community uses this design to supply Ada drivers for the timers, I2C, A/D and D/A converters, and other devices common to microcontrollers. Multiple devices are presented as instances of abstract data types. A variety of development platforms from various vendors are supported, including the STM32 series boards. The library is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library. We are going to use some of these drivers as illustrations in the following sections.

4.1 Non-Memory-Mapped Devices

Some devices are connected to the processor on a dedicated bus that is separate from the memory bus. The Intel processors, for example, used to have (and may still have) instructions for sending and receiving data on this bus. These are the "in" and "out" instructions, and their data-length specific variants.

The original version of Ada defined a package named `Low_Level_IO` for such architectures, but there were very few implementations (maybe just one, known to support the Intel processors). As a result, the package was actually removed from the language standard.

Implementations could still support the package, it just wouldn't be a standard package. That's different from constructs that are marked as "obsolescent" by the standard, e.g., the pragmas replaced by aspects, among other things. Obsolescent constructs are still part of the standard.

If a given target machine has such I/O instructions for the device bus, these can be invoked in Ada via machine-code insertions. For example:

```
procedure Send_Control (Device : Port;  Data : Unsigned_16) is
   pragma Suppress (All_Checks);
begin
   asm ("outw %1, (%0)",
        Inputs  => (Port'Asm_Input("dx",Device),
                    Unsigned_16'Asm_Input("ax",Data)),
        Clobber => "ax, dx");
end Send_Control;


procedure Receive_Control (Device : Port;  Data : out Unsigned_16) is
   pragma Suppress (All_Checks);
begin
   asm ("inw (%1), %0",
        Inputs   => (Port'Asm_Input("dx",Device)),
        Outputs  => (Unsigned_16'Asm_Output("=ax",Data)),
        Clobber  => "ax, dx",
        Volatile => True);
end Receive_Control;
```

Applications could use these subprograms to set the frequency of the Intel PC tone generator, for example, and to turn it on and off. (You can't do that any more in application code because modern operating systems don't give applications direct access to the hardware, at least not by default.)

Although the `Low_Level_IO` package is no longer part of the language, you can write this sort of thing yourself, or vendors can do it. That's possible because the Systems Programming Annex, when implemented, guarantees fully effective use of machine-code inserts. That means you can express anything the compiler could emit. The guarantee is important because otherwise the compiler might "get in the way." For example, absent the guarantee, the compiler would be allowed to insert additional assembly language statements in between yours. That can be a real problem, depending on what your statements do. For instance, if your MCI assembly statements do something and then check a resulting condition code, such as the overflow flag, those interleaved compiler-injected statements might clear that condition code before your code can check it. Fortunately, the annex guarantees that sort of thing cannot happen.

4.2 Memory-Mapped Devices

In *another earlier chapter* (page 7), we said that we could query the address of some object, and we also showed how to use that result to specify the address of some other object. We used that capability to create an "overlay," in which two objects are used to refer to the same memory locations. As we indicated in that discussion, you would not use the same type for each object — the point, after all, is to provide a view of the shared underlying memory cells that is not already available otherwise. Each distinct type would provide a distinct view of the memory values, that is, a set of operations providing some required functionality.

For example, here's an overlay composed of a 32-bit signed integer object and a 32-bit array object:

```
type Bits32 is array (0 .. 31) of Boolean
   with Component_Size => 1;

X : aliased Integer_32;
Y : Bits32 with Address => X'Address;
```

Because one view is as an integer and the other as an array, we can access that memory using the two different views' operations. Using the view as an array object (Y) we can access individual bits of the memory shared with X. Using the view as an integer (X), we can do arithmetic on the contents of that memory. (We could have used an unsigned integer instead of the signed type, and thereby gained the bit-oriented operations, but that's not the point.)

Very often, though, there is only one Ada object that we place at some specific address. That's because the Ada object is meant to be the software interface to some memory-mapped hardware device. In this scenario we don't have two overlaid Ada objects, we just have one. The other "object" is the hardware device mapped to that starting address. Since they are at the same memory location(s), accessing the Ada object accesses the hardware device.

For a real-world but nonetheless simple example, recall that example of a rotary switch on the front of our embedded computer that we mentioned in the introduction. This switch allows humans to provide some very simple input to the software running on the computer.

```
Rotary_Switch : Unsigned_8 with
  Address => System.Storage_Elements.To_Address (16#FFC0_0801#);
```

We declare the object and also specify the address, but not by querying some entity. We already know the address from the hardware documentation. But we cannot simply use an integer address literal from that documentation because type `System.Address` is almost always a private type. We need a way to compose an `Address` value from an integer value. The package `System.Storage_Elements` defines an integer representation for `Address` values, among other useful things, and a way to convert those integer values to `Address` values. The function `To_Address` does that conversion.

As a result, in the Ada code, reading the value of the variable `Rotary_Switch` reads the number on the actual hardware switch.

Note that if you specify the wrong address, it is hard to say what happens. Likewise, it is an error for an address clause to disobey the object's alignment. The error cannot be detected at compile time, in general, because the address is not necessarily known at compile time. There's no requirement for a run-time check for the sake of efficiency, since efficiency seems paramount here. Consequently, this misuse of address clauses is just like any other misuse of address clauses — execution of the code is erroneous, meaning all bets are off. You need to know what you're doing.

What about writing to the variable? Is that meaningful? In this particular example, no. It is effectively read-only memory. But for some other device it very well could be meaningful, certainly. It depends on the hardware. But in this case, assigning a value to the `Rotary_Switch` variable would have no effect, which could be confusing to programmers. It looks like a variable, after all. We wouldn't declare it as a constant because the human user could rotate the switch, resulting in a different value read. Therefore, we would hide the Ada variable behind a function, precluding the entire issue. Clients of the function can then use it for whatever purpose they require, e.g., as the unique identifier for a computer in a rack.

Let's talk more about the type we use to represent a memory-mapped device. As we said, that type defines the view we have for the object, and hence the operations we have available for accessing the underlying mapped device.

We choose the type for the representative Ada variable based on the interface of the hardware mapped to the memory. If the interface is a single monolithic register, for example,

then an integer (signed or unsigned) of the necessary size will suffice. But suppose the interface is several bytes wide, and some of the bytes have different purposes from the others? In that case, a record type is the obvious solution, with distinct record components dedicated to the different parts of the hardware interface. We could use individual bits too, of course, if that's what the hardware does. Ada is particularly good at this fine-degree of representation because record components of any types can be specified in the layout, down to the bit level, within the record.

In addition, we might want to apply more than one type, at any one time, to a given memory-mapped device. Doing so allows the client code some flexibility, or it might facilitate an internal implementation. For example, the STM32 boards from ST Microelectronics include a 96-bit device unique identifier on each board. The identifier starts at a fixed memory location. In this example we provide two different views — types — for the value. One type provides the identifier as a String containing twelve characters, whereas another type provides the value as an array of three 32-bit unsigned words (i.e., 12 bytes). The two types are applied by two overloaded functions that are distinguished by their return type:

```
package STM32.Device_Id is

   subtype Device_Id_Image is String (1 .. 12);

   function Unique_Id return Device_Id_Image;

   type Device_Id_Tuple is array (1 .. 3) of UInt32
     with Component_Size => 32;

   function Unique_Id return Device_Id_Tuple;

end STM32.Device_Id;
```

The subtype `Device_Id_Image` is the view of the 96-bits as an array of twelve 8-bit characters. (Using type **`String`** here isn't essential. We could have defined an array of bytes instead of **`Character`**.) Similarly, subtype `Device_Id_Tuple` is the view of the 96-bits as an array of three 32-bit unsigned integers. Clients can then choose how they want to view the unique id by choosing which function to call.

In the package body we implement the functions as two ways to access the same shared memory:

```
with System;

package body STM32.Device_Id is

   ID_Address : constant System.Address := System'To_Address (16#1FFF_7A10#);

   function Unique_Id return Device_Id_Image is
      Result : Device_Id_Image with Address => ID_Address, Import;
   begin
      return Result;
   end Unique_Id;

   function Unique_Id return Device_Id_Tuple is
      Result : Device_Id_Tuple with Address => ID_Address, Import;
   begin
      return Result;
   end Unique_Id;

end STM32.Device_Id;
```

The GNAT-defined attribute `System'To_Address` in the declaration of `ID_Address` is the same as the function `System.Storage_Elements.To_Address` except that, if the argument is static, the function result is static. This means that such an expression can be used in

contexts (e.g., preelaborable packages) which require a static expression and where the function call could not be used (because the function call is always non-static, even if its argument is static).

The only difference in the bodies is the return type and matching type for the local `Result` variable. Both functions read from the same location in memory.

Earlier we indicated that the bit-pattern implementation of the GPIO function could be expressed differently, resulting in more readable, therefore maintainable, code. The fact that the code is in C is irrelevant; the same approach in Ada would not be any better. Here's the complete code for the function body:

```
void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t      GPIO_PinSource,
                      uint8_t       GPIO_AF)
{
  uint32_t temp = 0x00;
  uint32_t temp_2 = 0x00;

  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
  assert_param(IS_GPIO_AF(GPIO_AF));

  temp = ((uint32_t)(GPIO_AF) <<
           ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
  GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
           ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
  temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
  GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}
```

The problem, other than the magic numbers (some named constants would have helped), is that the code is doing nearly all the work instead of off-loading it to the compiler. Partly that's because in C we cannot declare a numeric type representing a 4-bit quantity, so everything is done in terms of machine units, in this case 32-bit unsigned integers.

Why do we need 4-bit values? At the hardware level, each memory-mapped GPIO port has a sequence of 16 4-bit quantities, one for each of the 16 pins on the port. Those 4-bit quantities specify the "alternate functions" that the pin can take on, if needed. The alternate functions allow a given pin to do more than act as a single discrete I/O pin. For example, a pin could be connected to the incoming lines of a USART. We use the configuration routine to apply the specific 4-bit code representing the alternate function required for our application.

These 16 4-bit alternate function fields are contiguous in the register (hence memory) so we can represent them as an array with a total size of 64-bits (i.e., 16 times 4). In the C version this array has two components of type **`uint32_t`** so it must compute where the corresponding 4-bit value for the pin is located within those two words. In contrast, the Ada version of the array has components of the 4-bit type, rather than two 32-bit components, and simply uses the pin number as the index. The resulting Ada procedure body is extremely simple:

```
procedure Configure_Alternate_Function
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin;
   AF   : GPIO_Alternate_Function_Code)
is
begin
   Port.AFR (Pin) := AF;
end Configure_Alternate_Function;
```

In the Ada version, AFR is a component within the `GPIO_Port` record type, much like in the C code's struct. However, Ada allows us to declare a much more descriptive set of types,

and it is these types that allows the developer to off-load the work to the compiler.

First, in Ada we can declare a 4-bit numeric type:

```
type Bits_4 is mod 2**4 with Size => 4;
```

The Bits_4 type was already globally defined elsewhere so we just derive our 4-bit "alternate function code" type from it. Doing so allows the compiler to enforce simple strong typing so that the two value spaces are not accidentally mixed. This approach also increases understanding for the reader:

```
type GPIO_Alternate_Function_Code is new Bits_4;
--  We cannot use an enumeration type because there are duplicate binary
--  values
```

Hence type GPIO_Alternate_Function_Code is a copy of Bits_4 in terms of operations and values, but is not the same type as Bits_4 so the compiler will keep them separate for us.

We can then use that type as the array component type for the representation of the AFR:

```
type Alternate_Function_Fields is
  array (GPIO_Pin) of GPIO_Alternate_Function_Code
  with Component_Size => 4, Size => 64;  -- both in units of bits
```

Note that we can use the GPIO Pin parameter directly as the index into the array type, obviating any need to massage the Pin value in the procedure. That's possible because the type GPIO_Pin is an enumeration type:

```
type GPIO_Pin is
  (Pin_0, Pin_1, Pin_2,  Pin_3,  Pin_4,  Pin_5,  Pin_6,  Pin_7,
   Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);

for GPIO_Pin use
  (Pin_0  => 16#0001#,
   Pin_1  => 16#0002#,
   Pin_2  => 16#0004#,
   Pin_3  => 16#0008#,
   Pin_4  => 16#0010#,
   Pin_5  => 16#0020#,
   Pin_6  => 16#0040#,
   Pin_7  => 16#0080#,
   Pin_8  => 16#0100#,
   Pin_9  => 16#0200#,
   Pin_10 => 16#0400#,
   Pin_11 => 16#0800#,
   Pin_12 => 16#1000#,
   Pin_13 => 16#2000#,
   Pin_14 => 16#4000#,
   Pin_15 => 16#8000#);
```

In the hardware, the GPIO_Pin values don't start at zero and monotonically increase. Instead, the values are bit patterns, where one bit within each value is used. The enumeration representation clause allows us to express that representation.

Type Alternate_Function_Fields is then used to declare the AFR record component in the GPIO_Port record type:

```
type GPIO_Port is limited record
   MODER      : Pin_Modes_Register;
   OTYPER     : Output_Types_Register;
   Reserved_1 : Half_Word;
   OSPEEDR    : Output_Speeds_Register;
```

(continues on next page)

(continued from previous page)

```
   PUPDR      : Resistors_Register;
   IDR        : Half_Word;       --  input data register
   Reserved_2 : Half_Word;
   ODR        : Half_Word;       --  output data register
   Reserved_3 : Half_Word;
   BSRR_Set   : Half_Word;       --  bit set register
   BSRR_Reset : Half_Word;       --  bit reset register
   LCKR       : Word with Atomic;
   AFR        : Alternate_Function_Fields;
   Unused     : Unaccessed_Gap;
end record with
  Size => 16#400# * 8;

for GPIO_Port use record
   MODER      at 0  range 0 .. 31;
   OTYPER     at 4  range 0 .. 15;
   Reserved_1 at 6  range 0 .. 15;
   OSPEEDR    at 8  range 0 .. 31;
   PUPDR      at 12 range 0 .. 31;
   IDR        at 16 range 0 .. 15;
   Reserved_2 at 18 range 0 .. 15;
   ODR        at 20 range 0 .. 15;
   Reserved_3 at 22 range 0 .. 15;
   BSRR_Set   at 24 range 0 .. 15;
   BSRR_Reset at 26 range 0 .. 15;
   LCKR       at 28 range 0 .. 31;
   AFR        at 32 range 0 .. 63;
   Unused     at 40 range 0 .. 7871;
end record;
```

These declarations define a record type that matches the content and layout of the STM32 GPIO Port memory-mapped device.

Let's compare the two procedure implementations again. Here they are, for convenience:

```
void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t      GPIO_PinSource,
                      uint8_t       GPIO_AF)
{
  uint32_t temp = 0x00;
  uint32_t temp_2 = 0x00;

  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
  assert_param(IS_GPIO_AF(GPIO_AF));

  temp = ((uint32_t)(GPIO_AF) <<
            ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
  GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
            ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
  temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
  GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}
```

```
procedure Configure_Alternate_Function
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin;
   AF   : GPIO_Alternate_Function_Code)
is
begin
```

(continues on next page)

(continued from previous page)

```
   Port.AFR (Pin) := AF;
end Configure_Alternate_Function;
```

Which one is correct? Both. But clearly, the Ada version is far simpler, so much so that it is immediately obvious that it is correct. Not so for the coding approach used in the C version, comparatively speaking. It is true that the Ada version required a couple more type declarations, but those make the procedure body far simpler. That resulting simplicity is a reflection of the balance between data structures and executable statements that we should always try to achieve. Ada just makes that easier to achieve than in some other languages.

Of course, the underlying hardware likely has no machine-supported 4-bit unsigned type so larger hardware numeric types are used in the generated code. Hence there are shifts and masking being done in the Ada version as well, but they do not appear in the source code. The developer has let the compiler do that work. An additional benefit of this approach is that the compiler will change the shifting and masking code for us if we change the explicit type declarations.

Why is simplicity so important? Simplicity directly increases understandability, which directly affects correctness and maintainability, which greatly affects the economic cost of the software. In large, long-lived projects, maintenance is by far the largest economic cost driver. In high-integrity applications, correctness is essential. Therefore, doing anything reasonable to keep the code as simple as possible is usually worth the effort. In some projects the non-functional requirements, especially performance, can dictate less simple code, but that won't apply to all of the code. Where possible, simplicity rules.

One more point about the GPIO ports. There are as many of these ports as the Arm microcontroller vendor decides to implement. And as we said, they are memory-mapped, at addresses specified by the vendor. If the memory used by all the ports is contiguous, we can conveniently use an array of the `GPIO_Port` record type to represent all the ports implemented. We would just set the array object's address at the address specified for the first port object in memory. Then, normal array indexing will provide access to any given port in the memory-mapped hardware.

This array approach requires each array component — the `GPIO_Port` record type — to be the right size so that all the array components start on addresses corresponding to the start of the next port in hardware.

That starting address correspondence for the array components is obtained automatically as long as the record type includes all the memory used by any individual device. In that case the next array component will indeed start at an address matching the next device in hardware. Note that this assumes the first array component matches the address of the first hardware device in memory. The first array component is at the same address as the whole array object itself (a fact that is guaranteed by the language), so the array address must be set to whatever the vendor documentation specified for the first port.

However, in some cases the vendor will leave gaps of unused memory for complicated memory-mapped objects like these ports. They do so for the sake of future expansion of the implementation, e.g., to add new features or capacity. The gaps are thus between consecutive hardware devices.

These gaps are presumably (hopefully!) included in the memory layout documented for the device, but it won't be highlighted particularly. You should check, therefore, that the documented starting addresses of the second and subsequent array components are what you will get with a simple array object having components of that record type.

For example, the datasheet for the STM32F407 Arm implementation indicates that the GPIO ports start at address 16#4002_0000#. That's where GPIO_A begins. The next port, GPIO_B, starts at address 16#4002_0400#, or a byte offset of 1024 in decimal. In the STM32F4 Reference Manual, however, the GPIO port register layout indicates a size for any one port that is much less than 1024 bytes. As you saw earlier in the corresponding record

type declaration, on the STM32F4 each port only requires 40 (decimal) bytes. Hence there's a gap of unused memory between the ports, including after the last port, of 984 bytes (7872 bits).

To represent the gap, an "extra", unused record component was added, with the necessary location and size specified within the record type, so that the unused memory is included in the representation. As a result, each array component will start at the right address (again, as long as the first one does). Telling the compiler, and future maintainers, that this extra component is not meant to be referenced by the software would not hurt. You can use the pragma or aspect `Unreferenced` for that purpose. Here's the code again, for convenience:

```
type GPIO_Port is limited record
   MODER      : Pin_Modes_Register;
   OTYPER     : Output_Types_Register;
   Reserved_1 : Half_Word;
   OSPEEDR    : Output_Speeds_Register;
   PUPDR      : Resistors_Register;
   IDR        : Half_Word;       --  input data register
   Reserved_2 : Half_Word;
   ODR        : Half_Word;       --  output data register
   Reserved_3 : Half_Word;
   BSRR_Set   : Half_Word;       --  bit set register
   BSRR_Reset : Half_Word;       --  bit reset register
   LCKR       : Word with Atomic;
   AFR        : Alternate_Function_Fields;
   Unused     : Unaccessed_Gap with Unreferenced;
end record with
   Size => 16#400# * 8;

for GPIO_Port use record
   MODER      at 0  range 0 .. 31;
   OTYPER     at 4  range 0 .. 15;
   Reserved_1 at 6  range 0 .. 15;
   OSPEEDR    at 8  range 0 .. 31;
   PUPDR      at 12 range 0 .. 31;
   IDR        at 16 range 0 .. 15;
   Reserved_2 at 18 range 0 .. 15;
   ODR        at 20 range 0 .. 15;
   Reserved_3 at 22 range 0 .. 15;
   BSRR_Set   at 24 range 0 .. 15;
   BSRR_Reset at 26 range 0 .. 15;
   LCKR       at 28 range 0 .. 31;
   AFR        at 32 range 0 .. 63;
   Unused     at 40 range 0 .. 7871;
end record;
```

The type for the gap, `Unaccessed_Gap`, must represent 984 bytes so we declared an array like so:

```
Gap_Size : constant := 984;  -- bytes
--  There is a gap of unused, reserved memory after the end of the
--  bytes used by any given memory-mapped GPIO port. The size of the
--  gap is indicated in the STM32F405xx etc. Reference Manual, RM 0090.
--  Specifically, Table 1 shows the starting and ending addresses mapped
--  to the GPIO ports, for an allocated size of 16#400#, or 1024 (decimal)
--  bytes per port. However, in the same document, the register map for
--  these ports shows only 40 bytes currently in use. Presumably this gap is
--  for future expansion when additional functionality or capacity is added,
--  such as more pins per port.

type Unaccessed_Gap is array (1 .. Gap_Size) of Unsigned_8 with
   Component_Size => Unsigned_8'Size,
```

(continues on next page)

(continued from previous page)

```
   Size           => Gap_Size * Unsigned_8'Size;
--  This type is used to represent the necessary gaps between GPIO
--  ports in memory. We explicitly allocate a record component of
--  this type at the end of the record type for that purpose.
```

We also set the size of the entire record type to 16#400# bytes since that is the total of the required bytes plus the gap, as per the documentation. As such, this is a "confirming" size clause because the reserved gap component increases the required size to that value (which is the point). We don't really need to do both, i.e., declare the reserved gap component and also set the record type size to the larger value. We could have done either one alone. One could argue that setting the size alone would have been simpler, in that it would obviate the type declaration and corresponding record component declaration. Being doubly explicit seemed a good idea at the time.

4.3 Dynamic Address Conversion

In the overlay example there were two distinct Ada objects, of two different types, sharing one (starting) address. The overlay provides two views of the memory at that address because there are two types involved. In this idiom the address is known when the code is written, either because it is a literal value specified in some hardware spec, or it is simply the address of the other object (in which case the actual address value is neither known nor relevant).

When there are several views required, declaring multiple overlaid variables at the same address absolutely can work, but can be less convenient than an alternative idiom. The alternative is to convert an address value to a value of an access type. Dereferencing the resulting access value provides a view of the memory corresponding to the designated type, starting at the converted address value.

For example, perhaps a networking component is given a buffer — an array of bytes — representing a received message. A subprogram is called with the buffer as a parameter, or the parameter can be the address of the buffer. If the subprogram must interpret this array via different views, this alternative approach works well. We could have an access type designating a message preamble, for example, and convert the first byte's address into such an access value. Dereferencing the conversion gives the preamble value. Likewise, the subprogram might need to compute a checksum over some of the bytes, so a different view, one of an array of a certain set size, could be used. Again, we could do that with overlaid objects but the alternative can be more convenient.

Here's a simple concrete example to illustrate the approach. Suppose we want to have a utility to swap the two bytes at any arbitrary address. Here's the declaration:

```
procedure Swap2 (Location : System.Address);
```

Callers pass the address of an object intended to have its (first) two bytes swapped:

```
Swap2 (Z'Address);
```

In the call, Z is of type `Interfaces.Integer_16`, for example, or `Unsigned_16`, or even something bigger as long as you only care about swapping the first two bytes.

The incomplete implementation using the conversion idiom could be like so:

```
procedure Swap2 (Location : System.Address) is
   X : Word renames To_Pointer (Location).all;
begin
   X :=  Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;
```

The declaration of X is the pertinent part.

In the declaration, X is of type `Word`, a type (not yet shown) derived from `Interfaces.Unsigned_16`. Hence X can have the inherited shift and logical **`or`** operations applied.

The `To_Pointer (Location)` part of the declaration is a function call. The function returns the conversion of the incoming address value in `Location` into an access value designating `Word` values. We'll explain how to do that momentarily. The **`.all`** explicitly dereferences the access value resulting from the function call.

Finally, X renames the `Word` designated by the converted access value. The benefit of the renaming, in addition to the simpler name, is that the function is only called once, and the access value deference is only evaluated once.

Now for the rest of the implementation not shown earlier.

```
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;
```

`System.Address_To_Access_Conversions` is a language-defined generic package that provides just two functions: one to convert an address value to an access type, and one to convert in the opposite direction:

```
generic
   type Object (<>) is limited private;
package System.Address_To_Access_Conversions is

   type Object_Pointer is access all Object;

   function To_Pointer (Value : Address) return Object_Pointer;
   function To_Address (Value : Object_Pointer) return Address;

   pragma Convention (Intrinsic, To_Pointer);
   pragma Convention (Intrinsic, To_Address);

end System.Address_To_Access_Conversions;
```

`Object` is the generic formal type parameter, i.e., the type we want our converted addresses to designate via the type `Object_Pointer`. In the byte-swapping example, the type `Word` was passed to `Object` in the instantiation.

The access type used by the functions is `Object_Pointer`, declared along with the functions. `Object_Pointer` designates values of the type used for the generic actual parameter, in this case `Word`.

Note the pragma `Convention` applied to each function, indicating that there is no actual function call involved; the compiler emits the code directly, if any code is actually required. Otherwise the compiler just treats the incoming **`Address`** bits as a value of type `Object_Pointer`.

The instantiation specifies type `Word` as the generic actual type parameter, so now we have a set of functions for that type, in particular `To_Pointer`.

Let's look at the code again, this time with the additional declarations:

```
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;

procedure Swap2 (Location : System.Address) is
   X : Word renames To_Pointer(Location).all;
```

(continues on next page)

(continued from previous page)

```
begin
   X :=  Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;
```

`Word_Ops` is the generic instance, followed immediately by a **`use`** clause so that we can refer to the visible content of the package instance conveniently.

In the renaming expression, `To_Pointer (Location)` converts the incoming address in `Location` to a pointer designating the `Word` at that address. The **`.all`** dereferences the resulting access value to get the designated `Word` value. Hence `X` refers to that two-byte value in memory.

We could almost certainly achieve the same affect by replacing the call to the function in `To_Pointer` with a call to an instance of `Ada.Unchecked_Conversion`. The conversion would still be between an access type and a value of type `System.`**`Address`**, but the access type would require declaration by the user. In both cases there would be an instantiation of a language-defined facility, so there's not much saving in lines of source code, other than the access type declaration. Because `System.Address_To_Access_Conversions` is explicitly intended for this purpose, good style suggests its use in preference to unchecked conversion, but both approaches are common in production code.

In either case, the conversion is not required to work, although in practice it will, most of the time. Representing an access value as an address value is quite common because it matches the typical underlying hardware's memory model. But even so, a single address is not necessarily sufficient to represent an access value for any given designated type. In that case problems arise, and they are difficult to debug.

For example, in GNAT, access values designating values of unconstrained array types, such as **`String`**, are represented as two addresses, known as "fat pointers". One address points to the bounds for the specific array object, since they can vary. The other address designates the characters. Therefore, conversions of a single address to an access value requiring fat pointers will not work using unchecked conversions. (There is a way, however, to tell GNAT to use a single address value, but it is an explicit step in the code. Once done, though, unchecked conversions would then work correctly.)

You can alternatively use generic package `System.Address_To_Access_Conversions`. That generic is defined for the purpose of converting addresses to access values, and vice versa. But note that the implementation of the generic's routines must account for the representation their compiler uses for unbounded types like **`String`**.

4.4 Address Arithmetic

Part of "letting the compiler do the work for you" is not doing address arithmetic in the source code if you can avoid it. Instead, for instance, use the normal "dot notation" to reference components, and let the compiler compute the offsets to those components. The approach to implementing procedure `Configure_Alternate_Function` for a `GPIO_Port` is a good example.

That said, sometimes address arithmetic is the most direct expression of what you're trying to implement. For example, when implementing your own memory allocator, you'll need to do address arithmetic.

Earlier in this section we mentioned the package `System.Storage_Elements`, for the sake of the function that converts integer values to address values. The package also defines functions that provide address arithmetic. These functions work in terms of type `System.`**`Address`** and the package-defined type `Storage_Offset`. The type `Storage_Offset` is an integer type with an implementation-defined range. As a result you can have positive and negative offsets, as needed. Addition and subtraction of offsets to/from addresses is supported, as well as the **`mod`** operator.

Combined with package System (for type System.Address), the functions and types in this package provide the kinds of address arithmetic other languages provide. Nevertheless, you should prefer having the compiler do these computations for you, if possible.

Here's an example illustrating the facilities. The procedure defines an array of record values, then traverses the array, printing the array components as it goes. (This is not the way to really implement such code. It's just an illustration for address arithmetic.)

```
with Ada.Text_IO;               use Ada.Text_IO;
with System.Storage_Elements;   use System.Storage_Elements;
with System.Address_To_Access_Conversions;

procedure Demo_Address_Arithmetic is

   type R is record
      X : Integer;
      Y : Integer;
   end record;

   R_Size : constant Storage_Offset := R'Object_Size / System.Storage_Unit;

   Objects : aliased array (1 .. 10) of aliased R;     --  arbitrary bounds

   Objects_Base : constant System.Address := Objects'Address;

   Offset : Storage_Offset;

   --  display the object of type R at the address specified by Location
   procedure Display_R (Location : in System.Address) is

      package R_Pointers is new System.Address_To_Access_Conversions (R);
      use R_Pointers;

       Value : R renames To_Pointer (Location).all;
      --  The above converts the address to a pointer designating an R value
      --  and dereferences it, using the name Value to refer to the
      --  dereferenced R value.
   begin
      Put (Integer'Image (Value.X));
      Put (", ");
      Put (Integer'Image (Value.Y));
      New_Line;
   end Display_R;

begin
   Objects := ((0,0), (1,1), (2,2), (3,3), (4,4),
               (5,5), (6,6), (7,7), (8,8), (9,9));

   Offset := 0;

   --  walk the array of R objects, displaying each one individually by
   --  adding the offset to the base address of the array
   for K in Objects'Range loop
      Display_R (Objects_Base + Offset);
      Offset := Offset + R_Size;
   end loop;
end Demo_Address_Arithmetic;
```

Seriously, this is just for the purpose of illustration. It would be much better to just index into the array directly.

CHAPTER

FIVE

GENERAL-PURPOSE CODE GENERATORS

In *another chapter* (page 43), we mentioned that the best way to get a specific set of machine instructions emitted from the compiler is to write them ourselves, in the Ada source code, using machine-code insertions (MCI). The rationale was that the code generator will make reasonable assumptions, including the assumption that performance is of uppermost importance, but that these assumptions can conflict with device requirements.

For example, the code generator might not issue the specific sequence of machine code instructions required by the hardware. The GPIO pin "lock" sequence in that referenced chapter is a good example. Similarly, the optimizer might remove what would otherwise be "redundant" read/writes to a memory-mapped variable.

The code generator might issue instructions to read a small field in a memory-mapped record object using byte-sized accesses, when instead the device requires whole-word or half-word access instructions.

The code generator might decide to load a variable from memory into a register, accessing the register when the value is required. Typically that approach will yield far better performance than going to memory every time the value is read or updated. But suppose the variable is for a memory-mapped device? In that case we really need the generated code to go to memory every time.

As you can see, there are times when we cannot let the code generator make the usual assumptions. Therefore, Ada provides aspects and pragmas that developers can use to inform the compiler of facts that affect code generation in this regard.

These facilities are defined in the Systems Programming Annex, C.6, specifically. The title of that sub-clause is "Shared Variables" because the objects (memory) can be shared between tasks as well as between hardware devices and the host computer. We ignore the context of variables shared between tasks, focusing instead of shared memory-mapped devices, as this course is about embedded systems.

When describing these facilities we will use aspects, but remember that the corresponding pragmas are defined as well, except for one. (We'll mention it later.) For the other aspects, the pragmas existed first and, although obsolescent, remain part of the language and supported. There's no need to change your existing source code using the pragmas to use the aspects instead, unless you need to change it for some other reason.

As this is an introduction, we will not go into absolutely all the details, but will instead give a sense of what the language provides, and why.

5.1 Aspect Independent

To interface with a memory-mapped device, there will be an Ada object of an appropriate type that is mapped to one or more bytes of memory. The software interacts with the device by reading and/or writing to the memory locations mapped to the device, using the operations defined by the type in terms of normal Ada semantics.

Some memory-mapped devices can be directly represented by a single scalar value, usually of some signed or unsigned numeric type. More sophisticated devices almost always involve several distinct input and output fields. Therefore, representation in the software as a record object is very common. Ada record types have such extensive and flexible support for controlling their representation, down to the individual bit level, that using a record type makes sense. (And as mentioned, using normal record component access via the "dot notation" offloads to the compiler the address arithmetic needed to access individual memory locations mapped to the device.) And of course the components of the mapped record type can themselves be of scalar and composite types too, so an extensive descriptive capability exists with Ada.

Let's say that one of these record components is smaller than the size of the smallest addressable memory unit on the machine, which is to say, smaller than the machine instructions can read/write memory individually. A Boolean record component is a good example, and very common. The machine cannot usually read/write single bits in memory, so the generated code will almost certainly read or write a byte to get the enclosed single-bit Boolean component. It might use a larger sized access too, a half-word or word. Then the generated code masks off the bits that are not of interest and does some shifts to get the desired component.

Reading and writing the bytes surrounding the component accessed in the source code can cause a problem. In particular, some devices react to being read or written by doing something physical in the hardware. That's the device designer's intent for the software. But we don't want that to happen accidentally due to surrounding bytes being accessed.

Therefore, to prevent these "extra" bytes from being accessed, we need a way to tell the compiler that we need the read or write accesses for the given object to be independent of the surrounding memory. If the compiler cannot do so, we'll get an error and the compilation will fail. That beats debugging, every time.

Therefore, the aspect `Independent` specifies that the code generated by the compiler must be able to load and store the memory for the specified object without also accessing surrounding memory. More completely, it declares that a type, object, or component must be independently addressable by the hardware. If applied to a type, it applies to all objects of the type.

Likewise, aspect `Independent_Components` declares that the individual components of an array or record type must be independently addressable.

With either aspect the compiler will reject the declaration if independent access is not possible for the type/object in question.

For example, if we try to mark each Boolean component of a record type as `Independent` we can do so, either individually or via `Indepndent_Components`, but doing so will require that each component is a byte in size (or whatever the smallest addressable unit happens to be on this machine). We cannot make each Boolean component occupy one bit within a given byte if we want them to be independently accessed.

```
package P is

   type R is record
      B0 : Boolean;
      B1 : Boolean;
      B2 : Boolean;
      B3 : Boolean;
      B4 : Boolean;
      B5 : Boolean;
   end record with
     Size => 8,
     Independent_Components;

   for R use record
```

(continues on next page)

(continued from previous page)

```
      B0 at 0 range 0 .. 0;
      B1 at 0 range 1 .. 1;
      B2 at 0 range 2 .. 2;
      B3 at 0 range 3 .. 3;
      B4 at 0 range 4 .. 4;
      B5 at 0 range 5 .. 5;
   end record;

end P;
```

For a typical target machine the compiler will reject that code, complaining that the `Size` for R' must be at least 48 bits, i.e., 8 bits per component. That's because the smallest quantity this machine can independently address is an 8-bit byte.

But if we don't really need the individual bits to be independently accessed — and let's hope no hardware designer would define such a device — then we have more flexibility. We could, for example, require that objects of the entire record type be independently accessible:

```
package Q is

   type R is record
      B0 : Boolean;
      B1 : Boolean;
      B2 : Boolean;
      B3 : Boolean;
      B4 : Boolean;
      B5 : Boolean;
   end record with
     Size => 8,
     Independent;

   for R use record
      B0 at 0 range 0 .. 0;
      B1 at 0 range 1 .. 1;
      B2 at 0 range 2 .. 2;
      B3 at 0 range 3 .. 3;
      B4 at 0 range 4 .. 4;
      B5 at 0 range 5 .. 5;
   end record;

end Q;
```

This the compiler should accept, assuming a machine that can access bytes in memory individually, without having to read some number of other bytes.

But for another twist, suppose we need one of the components to be aliased, so that we can construct access values designating it via the **`Access`** attribute? For example, given the record type R above, and some object `Foo` of that type, suppose we want to say `Foo.B0'Access`? We'd need to mark the component as **`aliased`**:

```
package QQ is

   type R is record
      B0 : aliased Boolean;
      B1 : Boolean;
      B2 : Boolean;
      B3 : Boolean;
      B4 : Boolean;
      B5 : Boolean;
   end record with
     Size => 8,
```

(continues on next page)

(continued from previous page)

```
     Independent;

   for R use record
      B0 at 0 range 0 .. 0;
      B1 at 0 range 1 .. 1;
      B2 at 0 range 2 .. 2;
      B3 at 0 range 3 .. 3;
      B4 at 0 range 4 .. 4;
      B5 at 0 range 5 .. 5;
   end record;

end QQ;
```

The compiler will once again reject the code, complaining that the size of B0 must be a multiple of a `Storage_Unit`, in other words, the size of something independently accessible in memory on this machine.

Why? The issue here is that aliased objects, including components of composite types, must be represented in such a way that creating the designating access ("pointer") value is possible. The component B0, if allocated only one bit, would not allow an access value to be created due to the usual machine accessibility limitation we've been discussing.

Similarly, a record component that is of some by-reference type, such as any tagged type, introduces the same issues as an aliased component. That's because the underlying implementation of by-reference parameter passing is much like a `'Access` attribute reference.

As important as the effect of this aspect is, you probably won't see it specified. There are other aspects that are more typically required. However, the semantics of `Independent` are part of the semantics of some of these other aspects. Applying them applies `Independent` too, in effect. So even though you don't typically apply it directly, you need to understand the independent access semantics. We discuss these other, more commonly applied aspects next.

These representation aspects may be specified for an object declaration, a component declaration, a full type declaration, or a generic formal (complete) type declaration. If any of these aspects are specified True for a type, then the corresponding aspect is True for all objects of the type.

5.2 Aspect `Volatile`

Earlier we said that the compiler (specifically the optimizer) might decide to load a variable from memory into a register, accessing the register when the value is required or updated. Similarly, the compiler might reorder instructions, and remove instructions corresponding to redundant assignments in the source code. Ordinarily we'd want those optimizations, but in the context of embedded memory-mapped devices they can be problematic.

The hardware might indeed require the source code to read or write to the device in a way that the optimizer would consider redundant, and in order to interact with the device we need every read and write to go to the actual memory for the mapped device, rather than a register. As developers we have knowledge about the context that the compiler lacks.

The compiler is aware of the fact that the Ada object is memory-mapped because of the address clause placing the object at a specific address. But the compiler does not know we are interacting with an external hardware device. Perhaps, instead, the object is mapped to a specific location because some software written in another language expects to access it there. In that case redundant reads or writes of the same object really would be redundant. The fact that we are interacting with a hardware device makes a difference.

In terms of the language rules, we need reading from, and writing to, such devices to be part of what the language refers to as the "external effects" of the software. These effects

are what the code must actually produce. Anything else — the internal effects — could be removed by the optimizer.

For example, suppose you have a program that writes a value to some variable and also writes the string literal "42" to a file. That's is absolutely all that the program contains.

```
with Ada.Text_IO;  use Ada.Text_IO;

procedure Demo is
   Output : File_Type;
   Silly  : Integer;
begin
   Silly := 0;
   Create (Output, Out_File, "output.txt");
   Put (Output, "42");
   Close (Output);
end Demo;
```

The value of the variable `Silly` is not used in any way so there is no point in even declaring the variable, much less generating code to implement the assignment. The update to the variable has only an internal effect. With warnings enabled we'll receive notice from the compiler, but they're just warnings.

However, writing to the file is an external effect because the file persists beyond the end of the program's execution. The optimizer (when enabled) would be free to remove any access to the variable `Silly`, but not the write to the file.

We can make the compiler recognize that a software object is part of an external effect by applying the aspect `Volatile`. (Aspect `Atomic` is pertinent too. More in a moment.) As a result, the compiler will generate memory load or store instructions for every read or update to the object that occurs in the source code. Furthermore, it cannot generate any additional loads or stores to that variable, and it cannot reorder loads or stores from their order in the source code. "What You See Is What You Get" in other words.

```
with Ada.Text_IO;  use Ada.Text_IO;

procedure Demo is
   Output : File_Type;
   Silly  : Integer with Volatile;
begin
   Silly := 0;
   Create (Output, Out_File, "output.txt");
   Put (Output, "42");
   Close (Output);
end Demo;
```

If we compile the above, we won't get the warning we got earlier because the compiler is now required to generate the assignment for `Silly`.

The variable `Silly` is not even a memory-mapped object, but remember that we said these aspects are important to the tasking context too, for shared variables. We're ignoring that context in this course.

There is another reason to mark a variable as `Volatile`. Sometimes you want to have exactly the load and store instructions generated that match those of the Ada code, even though the volatile object is not a memory-mapped object. For example, *elsewhere* (page 43) we said that the best way to achieve exact assembly instruction sequences is the use of machine-code inserts (MCI). That's true, but for the moment let's say we want to write it in Ada without the MCIs. Our earlier example was the memory-mapped GPIO ports on Arm microcontrollers produced by ST Microelectronics. Specifically, these ports have a "lock" per GPIO pin that allows the developer to configure the pin and then lock it so that no other configuration can accidentally change the configuration of that pin. Doing so requires an exact sequence of loads and stores. If we wrote this in Ada it would look like this:

```
procedure Lock
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin)
is
   Temp : Word with Volatile;
begin
   --  set the lock control bit and the pin
   --  bit, clear the others
   Temp := LCCK or Pin'Enum_Rep;

   --  write the lock and pin bits
   Port.LCKR := Temp;

   --  clear the lock bit in the upper half
   Port.LCKR := Pin'Enum_Rep;

   --  write the lock bit again
   Port.LCKR := Temp;

   --  read the lock bit
   Temp := Port.LCKR;

   --  read the lock bit again
   Temp := Port.LCKR;
end Lock;
```

Temp is marked volatile for the sake of getting exactly the load and stores that we express in the source code, corresponding to the hardware locking protocol. It's true that Port is a memory-mapped object, so it too would be volatile, but we also need Temp to be volatile.

This high-level coding approach will work, and is simple enough that MCIs might not be needed. However, what really argues against it is that the correct sequence of emitted code requires the optimizer to remove all the other cruft that the code generator would otherwise include. (The gcc code generator used by the GNAT compiler generates initially poor code, by design, relying on the optimizer to clean it up.) In other words, we've told the optimizer not to change or add loads and stores for Temp, but without the optimizer enabled the code generator generates other code that gets in the way. That's OK in itself, as far as procedure Lock is concerned, but if the optimizer is sufficiently enabled we cannot debug the rest of the code. Using MCIs avoids these issues. The point, though, is that not all volatile objects are memory mapped.

So far we've been illustrating volatility with scalar objects, such as Lock.Temp above. What about objects of array and record types? (There are other "composite" types in Ada but they are not pertinent here.)

When aspect Volatile is applied to a record type or an object of such a type, all the record components are automatically volatile too.

For an array type (but not a record type), a related aspect Volatile_Components declares that the components of the array type — but not the array type itself — are volatile. However, if the Volatile aspect is specified, then the Volatile_Components aspect is automatically applied too, and vice versa. Thus components of array types are covered automatically.

If an object (of an array type or record type) is marked volatile then so are all of its subcomponents, even if the type itself is not marked volatile.

Therefore aspects Volatile and Volatile_Components are nearly equivalent. In fact, Volatile_Components is superfluous. The language provides the Volatile_Components aspect only to give symmetry with the Atomic_Components and Independent_Components aspects. You can simply apply Volatile and be done with it.

Finally, note that applying aspect Volatile does not implicitly apply Independent, although

you can specify it explicitly if need be.

5.3 Aspect Atomic

Consider the GPIO pin configuration lock we've mentioned a few times now, that freezes the configuration of a given pin on a given GPIO port. The register, named LCKR for "lock register", occupies 32-bits, but only uses 17 total bits (currently). The low-order 16 bits, [0:15], represent the 16 GPIO pins on the given port. Bit #16 is the lock bit. That bit is the first bit in the upper half of the entire word. To freeze the configuration of a given pin in [0:15], the lock bit must be set at the same time as the bit to be frozen. In other words, the lower half and the upper half of the 32-bit word representing the register must be written together, at the same time. That way, accidental (un)freezing is unlikely to occur, because the most efficient, hence typical way for the generated code to access individual bits is for the compiler to load or store just the single byte that contains the bit or bits in question.

This indivisibility effect can be specified via aspect `Atomic`. As a result, all reads and updates of such an object as a whole are indivisible. In practice that means that the entire object is accessed with one load or store instruction. For a 16-bit object, all 16-bits are loaded and stored at once. For a 32-bit object, all 32-bits at once, and so on. The upper limit is the size of the largest machine scalar that the processor can manipulate with one instruction, as defined by the target processor. The typical lower bound is 8, for a byte-addressable machine.

Therefore, within the record type representing a GPIO port, we include the lock register component and apply the aspect `Atomic`:

```
type GPIO_Port is limited record
   ...
   LCKR : UInt32 with Atomic;
   ...
end record with
  ...
  Size => 16#400# * 8;
```

Hence loads and stores to the LCKR component will be done atomically, otherwise the compiler will let us know that it is impossible. That's all we need to do for the lock register to be read and updated atomically.

You should understand that only accesses to the whole, entire object are atomic. In the case of the lock register, the entire object is a record component, but that causes no problems here.

There is, however, something we must keep in mind when manipulating the values of atomic objects. For the lock register we're using a scalar type to represent the register, an unsigned 32-bit integer. There are no sub-components because scalar types don't have components, by definition. We simply use the bit-level operations to set and clear the individual bits. But we cannot set the bits — the lock bit and the bit for the I/O pin to freeze — one at a time because the locking protocol requires all the bits to be written at the same time, and only the entire 32-bit load and stores are atomic. Likewise, if instead of a scalar we used a record type or an array type to represent the bits in the lock register, we could not write individual record or array components one at a time, for the same reason we could not write individual bits using the unsigned scalar. The `Atomic` aspect only applies to loads and stores of the entire register.

Therefore, to update or read individual parts of an atomic object we must use a coding idiom in which we explicitly read or write the entire object to get to the parts. For example, to read an individual record component, we'd first read the entire record object into a temporary variable, and then access the component of that temporary variable. Likewise, to update one or more individual components, we'd first read the record object into a temporary variable, update the component or components within that temporary, and then write the

temporary back to the mapped device object. This is known as the "read-modify-write" idiom. You'll see this idiom often, regardless of the programming language, because the hardware requirement is not unusual. Fortunately Ada defines another aspect that makes the compiler do this for us. We'll describe it in the next section.

Finally, there are issues to consider regarding the other aspects described in this section.

If you think about atomic behavior in the context of machine instructions, loading and storing from/to memory atomically can only be performed for quantities that are independently addressable. Consequently, all atomic objects are considered to be specified as independently addressable too. Aspect specifications and representation items cannot change that fact. You can expect the compiler to reject any aspect or representation choice that would prevent this from being true.

Likewise, atomic accesses only make sense on actual memory locations, not registers. Therefore all atomic objects are volatile objects too, automatically.

However, unlike volatile objects, the components of an atomic object are not automatically atomic themselves. You'd have to mark these types or objects explicitly, using aspect `Atomic_Components`. Unlike `Volatile_Components`, aspect `Atomic_Components` is thus useful.

As is usual with Ada programming, you can rely on the compiler to inform you of problems. The compiler will reject an attempt to specify `Atomic` or `Atomic_Components` for an object or type if the implementation cannot support the indivisible and independent reads and updates required.

5.4 Aspect `Full_Access_Only`

Many devices have single-bit flags in the hardware that are not allocated to distinct bytes. They're packed into bytes and words shared with other flags. It isn't just individual bits either. Multi-bit fields that are smaller than a byte, e.g., two 4-bit quantities packed into a byte, are common. We saw that with the GPIO alternate functions codes earlier.

Ordinarily in Ada we represent such composite hardware interfaces using a record type. (Sometimes an array type makes more sense. That doesn't change anything here.) Compared to using bit-patterns, and the resulting bit shifting and masking in the source code, a record type representation and the resulting "dot notation" for accessing components is far more readable. It is also more robust because the compiler does all the work of retrieving these individual bits and bit-fields for us, doing any shifting and masking required in the generated code. The loads and stores are done by the compiler in whatever manner the compiler thinks most efficient.

When the hardware device requires atomic accesses to the memory mapped to such flags, we cannot let the compiler generate whatever width load and store accesses it thinks best. If full-word access is required, for example, then only loads and stores for full words can work. Yet aspect `Atomic` only guarantees that the entire object, in this case the record object, is loaded and stored indivisibly, via one instruction. The aspect doesn't apply to reads and updates to individual record components.

In the section on `Atomic` above, we mentioned that proper access to individual components of atomic types/objects can be achieved by a "read-modify-write" idiom. In this idiom, to read a component you first read into a temporary the entire enclosing atomic object. Then you read the individual component from that temporary variable. Likewise, to update an individual component, you start with the same approach but then update the component(s) within the temporary, then store the entire temporary back into the mapped atomic object. Applying aspect `Atomic` to the enclosing object ensures that reading and writing the temporary will be atomic, as required.

Using bit masks and bit patterns to access logical components as an alternative to a record type doesn't change the requirement for the idiom.

Consider the STM32F4 DMA device. The device contains a 32-bit stream configuration register that requires 32-bit reads and writes. We can map that register to an Ada record type like so:

```
type Stream_Config_Register is record
   --  ...
   Direction         : DMA_Data_Transfer_Direction;
   P_Flow_Controller : Boolean;
   TCI_Enabled       : Boolean;  -- transfer complete
   HTI_Enabled       : Boolean;  -- half-transfer complete
   TEI_Enabled       : Boolean;  -- transfer error
   DMEI_Enabled      : Boolean;  -- direct mode error
   Stream_Enabled    : Boolean;
end record
   with Atomic, Size => 32;
```

The "confirming" size clause ensures we have declared the type correctly such that it will fit into 32-bits. There will also be a record representation clause to ensure the record components are located internally as required by the hardware. We don't show that part.

The aspect `Atomic` is applied to the entire record type, ensuring that the memory mapped to the hardware register is loaded and stored only as 32-bit quantities. In this example it isn't that we want the loads and stores to be indivisible. Rather, we want the generated machine instructions that load and store the object to use 32-bit word instructions, even if we are only reading or updating a component of the object. That's what the hardware requires for all accesses.

Next we'd use that type declaration to declare one of the components of an enclosing record type representing one entire DMA "stream":

```
type DMA_Stream is record
   CR   : Stream_Config_Register;
   NDTR : Word;    -- upper half must remain at reset value
   PAR  : Address; -- peripheral address register
   M0AR : Address; -- memory 0 address register
   M1AR : Address; -- memory 1 address register
   FCR  : FIFO_Control_Register;
end record
   with Volatile, Size => 192;  -- 24 bytes
```

Hence any individual DMA stream record object has a component named CR that represents the corresponding configuration register.

The DMA controllers have multiple streams per unit so we'd declare an array of `DMA_Stream` components. This array would then be part of another record type representing a DMA controller. Objects of the `DMA_Controller` type would be mapped to memory, thus mapping the stream configuration registers to memory.

Now, given all that, suppose we want to enable a stream on a given DMA controller. Using the read-modify-write idiom we would do it like so:

```
procedure Enable
   (Unit   : in out DMA_Controller;
    Stream : DMA_Stream_Selector)
is
   Temp : Stream_Config_Register;
   --  these registers require 32-bit accesses, hence the temporary
begin
   Temp := Unit.Streams (Stream).CR;  --  read entire CR register
   Temp.Stream_Enabled := True;
   Unit.Streams (Stream).CR := Temp;  --  write entire CR register
end Enable;
```

That works, and of course the procedural interface presented to clients hides the details, as it should.

To be fair, the bit-pattern approach can express the idiom concisely, as long as you're careful. Here's the C code to enable and disable a selected stream:

```
#define DMA_SxCR_EN    ((uint32_t)0x00000001)

/* Enable the selected DMAy Streamx by setting EN bit */
DMAy_Streamx->CR  |=  DMA_SxCR_EN;

/* Disable the selected DMAy Streamx by clearing EN bit */
DMAy_Streamx->CR  &=  ~DMA_SxCR_EN;
```

The code reads and writes the entire CR register each time it is referenced so the requirement is met.

Nevertheless, the idiom is error-prone. We might forget to use it at all, or we might get it wrong in one of the very many places where we need to access individual components.

Fortunately, Ada provides a way to have the compiler implement the idiom for us, in the generated code. Aspect `Full_Access_Only` specifies that all reads of, or writes to, a component are performed by reading and/or writing all of the nearest enclosing full access object. Hence we add this aspect to the declaration of `Stream_Config_Register` like so:

```
type Stream_Config_Register is record
   --  ...
   Direction         : DMA_Data_Transfer_Direction;
   P_Flow_Controller : Boolean;
   TCI_Enabled       : Boolean;  -- transfer complete interrupt
   HTI_Enabled       : Boolean;  -- half-transfer complete
   TEI_Enabled       : Boolean;  -- transfer error interrupt
   DMEI_Enabled      : Boolean;  -- direct mode error interrupt
   Stream_Enabled    : Boolean;
end record
   with Atomic, Full_Access_Only, Size => 32;
```

Everything else in the declaration remains unchanged.

Note that `Full_Access_Only` can only be applied to `Volatile` types or objects. `Atomic` types are automatically `Volatile` too, so either one is allowed. You'd need one of those aspects anyway because `Full_Access_Only` just specifies the accessing instruction requirements for the generated code when accessing components.

The big benefit comes in the source code accessing the components. Procedure `Enable` is now merely:

```
procedure Enable
   (Unit   : in out DMA_Controller;
    Stream : DMA_Stream_Selector)
is
begin
   Unit.Streams (Stream).CR.Stream_Enabled := True;
end Enable;
```

This code works because the compiler implements the read-modify-write idiom for us in the generated code.

The aspect `Full_Access_Only` is new in Ada 2022, and is based on an implementation-defined aspect that GNAT first defined named `Volatile_Full_Access`. You'll see that GNAT aspect throughout the Arm device drivers in the Ada Drivers Library, available here: https://github.com/AdaCore/Ada_Drivers_Library. Those drivers were the motivation for the GNAT aspect.

Unlike the other aspects above, there is no pragma corresponding to the aspect `Full_Access_Only` defined by Ada 2022. (There is such a pragma for the GNAT-specific version named `Volatile_Full_Access`, as well as an aspect.)

CHAPTER

SIX

HANDLING INTERRUPTS

6.1 Background

Embedded systems developers offload functionality from the application processor onto external devices whenever possible. These external devices may be on the same "chip" as the central processor (e.g., within a System-on-Chip) or they may just be on the same board, but the point here is that they are not the processor executing the application. Offloading work to these other devices enables us to get more functionality implemented in a target platform that is usually very limited in resources. If the processor has to implement everything we might miss deadlines or perhaps not fit into the available code space. And, of course, some specialized functionality may simply require an external device, such as a sensor.

For a simple example, a motor encoder is a device attached to a motor shaft that can be used to count the number of full or partial rotations that the shaft has completed. When the shaft is rotating quickly, the application would need to interact with the encoder frequently to get an up-to-date count, representing a non-trivial load on the application processor. There are ways to reduce that load, which we discuss shortly, but by far the simplest and most efficient approach is to do it all in hardware: use a timer device driven directly by the encoder. The timer is connected to the encoder such that the encoder signals act like an external clock driving the timer's internal counter. All the application processor must do to get the encoder count is query the timer's counter. The timer is almost certainly memory-mapped, so querying the timer amounts to a memory access.

In some cases, we even offload communication with these external devices onto other external devices. For example, the I2C[4] (Inter-Integrated Circuit) protocol is a popular two-wire serial protocol for communicating between low-level hardware devices. Individual bits of the data are sent by driving the data line high and low in time with the clock signal on the other line. The protocol has been around for a long time and many embedded devices use it to communicate. We could have the application drive the data line for each individual bit in the protocol. Known as "bit-banging," that would be a significant load on the processor when the overall traffic volume is non-trivial. Fortunately, there are dedicated devices — I2C transceivers — that will implement the protocol for us. To send application data to another device using the I2C protocol, we just give the transceiver the data and destination address. The rest is done in the transceiver hardware. Receiving data is of course also possible. I2C transceivers are ubiquitous because the protocol is so common among device implementations. A USART[5] / UART[6] is a similar example.

Having offloaded some of the work, the application must have some way to interact with the device in order to know what is happening. Maybe the application has requested the external device perform some service — an analog-to-digital conversion, say — and must know when that function has completed. Maybe a communications device is receiving incoming data for the application to process. Or maybe that communications device has completed sending outgoing data and is ready for more to send.

[4] https://en.wikipedia.org/wiki/I%C2%B2C
[5] https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter
[6] https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Ultimately, interaction with the external device will be either synchronous or asynchronous, and has system-level design implications.

For synchronous interaction, the application periodically queries the device, typically a status flag or function on the device. Known as "polling," this approach is simple to implement but wastes cycles when the external device has not yet completed the request. After all, the point of offloading the work is to allow the application processor to execute other functionality. Polling negates that benefit. On the other hand, if the expected time to completion is extremely short, polling can be sufficiently efficient to make sense.

Usually, there's enough time involved so that polling is undesirable. The external environment takes time to respond and change state. Maybe a sensor has been designed to wait passively for something to happen in the external world, and only on the infrequent occurrence of that event should the application be notified. Perhaps a switch is to be toggled in certain circumstances, or an intruder detected. In this case, nothing happens for extended intervals.

As a consequence of all this, there's a very good chance that the internal processor should not poll these external devices.

Before we discuss the asynchronous alternative, there's another issue to consider. However the notification from the external device is implemented, a very quick response from the internal processor may be required. Think back to that serial port with a USART again. The USART is responsible for composing the arriving characters (or bytes) from their individual incoming bits on the receiving line. When all the bits for a single character have arrived, what happens next depends on the software design. In the simplest case, the internal processor copies the single character from the USART to an internal buffer and then goes back to doing something else while the next full character arrives in the USART. The response to the USART must be fairly quick because the next incoming character's bits are arriving. The internal processor must get the current character before it is overwritten by the next arriving character, otherwise we'll lose data. So we can say that the response to the notification from the external device must often be very quick.

Now, ideally in the USART case, we would further offload the work from the internal processor. Instead of having the processor copy each arriving character from the USART into an application buffer, we would have another external hardware device — a direct memory access (DMA)[7] device — copy each arriving character from the USART to the buffer. A DMA device copies data from one location to another, in this case from the address of the USART's one-character memory-mapped register to the address of the application buffer in memory. The copy is performed by the DMA hardware so it is extremely fast and costs the main processor no cycles. But even with this approach, we need to notify the application that a complete message is ready for processing. We might need to do that quickly so that enough time remains for the application to process the message content prior to the arrival of the next message.

Therefore, the general requirement is for an external device to be able to asynchronously notify the internal processor, and for the notification to be implemented in such a way that the beginning of the response can be sufficiently and predictably quick.

Fortunately, computers already have such a mechanism: interrupts. The details vary considerably with the hardware architecture, but the overall idea is independent of the ISA[8]: an external event can trigger a response from the processor by becoming "active." The current state of the application is temporarily stored, and then an interrupt response routine, known as an "interrupt handler" is executed. Upon completion of the handler, the original state of the application is restored and the application continues execution. The time between the interrupt becoming active and the start of the responding handler execution is known as the "interrupt latency."

Hardware interrupts typically have priorities assigned, depending on the hardware. These priorities are applied when multiple interrupts are triggered at the same time, to define

[7] https://en.wikipedia.org/wiki/Direct_memory_access
[8] https://en.wikipedia.org/wiki/Instruction_set_architecture

the order in which the interrupts are presented and the handlers invoked. The canonical model is that only higher-priority interrupts can preempt handlers executing in response to interrupts with lower or equal priority.

Ada defines a model for hardware interrupts and interrupt handling that closely adheres to the conceptual model described above. If you have experience with interrupt handling, you will recognize them in the Ada model. One very important point to make about the Ada facilities is that they are highly portable, so they don't require extensive changes when moving to an new target computer. Part of that portability is due to the language-defined model.

Before we go into the Ada facility details, there's a final point. Sometimes we *do* want the application to wait for the external device. When would that be the case? To answer that, we need to introduce another term. The act of saving and restoring the state of the interrupted application software is known as "interrupt context switching." If the time for the device to complete the application request is approximately that of the context switching, the application might as well wait for the device after issuing the request.

Another reason to consider polling is that the architectural complexity of interrupt handling is greater than that of polling. If your system has some number of devices to control and polling them would be fast enough for the application to meet requirements, it is simpler to do so. But that will likely only work for a few devices, or at least a few that have short response time requirements.

The application code can wait for the device by simply entering a loop, exiting only when some external device status flag indicates completion of the function. The loop itself, in its simplest form, would contain only the test for exiting. As mentioned earlier, polling in a tight loop like this only makes sense for very fast device interactions. That's not the usual situation though, so polling should not be your default design assumption. Besides, active polling consumes power. On an embedded platform, conserving power is often important.

That loop polling the device will never exit if the device can fail to signal completion. Or maybe it might take too long in some odd case. If you don't want to be potentially stuck in the loop indefinitely, chewing up cycles and power, you can add an upper bound on the number of attempts, i.e., loop iterations. For example:

```
procedure Await_Data_Ready (This : in out Three_Axis_Gyroscope) is
   Max_Status_Attempts : constant := 10_000;
   --  This upper bound is arbitrary but must be sufficient for the
   --  slower gyro data rate options and higher clock rates.  It need
   --  not be as small as possible, the point is not to hang forever.
begin
   Polling: for K in 1 .. Max_Status_Attempts loop
      if Data_Status (This).ZYX_Available then
         return;
      end if;
   end loop Polling;
   raise Gyro_Failure;
end Await_Data_Ready;
```

In the above, `Data_Status` is a function that returns a record object containing Boolean flags. The if-statement queries one of those flags. Thus the loop either detects the desired device status or raises an exception after the maximum number of attempts have been made. In this version, the maximum is a known upper bound so a local constant will suffice. The maximum could be passed as a parameter instead, or declared in a global "configuration" package containing such constants.

Presumably, the upper bound on the attempts is either specified by the device documentation or empirically determined. Sometimes, however, the documentation will instead specify a maximum possible response time, for instance 30 milliseconds. Any time beyond that maximum indicates a device failure.

In the code above, the number of iterations indirectly defines the amount of elapsed time

the caller waits. That time varies with the target's system clock and the generated instructions' required clock cycles, hence the approach is not portable. Alternatively, we can work in terms of actual time, which will be portable across all targets with a sufficiently precise clock.

You can use the facilities in package `Ada.Real_Time` to work with time values. That package defines a type `Time_Span` representing time intervals, useful for expressing relative values such as elapsed time. There is also type `Time` representing an absolute value on the timeline. A function `Clock` returns a value of type `Time` representing "now," along with overloaded addition and subtraction operators taking `Time` and `Time_Span` parameters. The package also provides operators for comparing `Time` values. (The value returned by `Clock` is monotonically increasing so you don't need to handle time zone jumps and other such things, unlike the function provided by `Ada.Calendar`.)

If the timeout is not context-specific then we'd use a constant as we did above, otherwise we'd allow the caller to specify the timeout. For example, here's a polling routine included with the DMA device driver we've mentioned a few times now. Some device-specific parts have been removed to keep the example simple. The appropriate timeout varies, so it is a parameter to the call:

```
procedure Poll_For_Completion
  (This    : in out DMA_Controller;
   Stream  : DMA_Stream_Selector;
   Timeout : Time_Span;
   Result  : out DMA_Error_Code)
is
   Deadline : constant Time := Clock + Timeout;
begin
   Result := DMA_No_Error;  -- initially
   Polling : loop
      exit Polling when Status (This, Stream, Transfer_Complete_Indicated);
      if Clock >= Deadline then
         Result := DMA_Timeout_Error;
         return;
      end if;
   end loop Polling;
   Clear_Status (This, Stream, Transfer_Complete_Indicated);
end Poll_For_Completion;
```

In this approach, we compute the deadline as a point on the timeline by adding the value returned from the `Clock` function (i.e., "now") to the time interval specified by the parameter. Then, within the loop, we compare the value of the `Clock` to that deadline.

Finally, with another design approach we can reduce the processor cycles "wasted" when the polled device is not yet ready. Specifically, in the polling loop, when the device has not yet completed the requested function, we can temporarily relinquish the processor so that other tasks within the application can execute. That isn't perfect because we're still checking the device status even though we cannot exit the loop. And it requires other tasks to exist in your design, although that's probably a good idea for other reasons (e.g., logical threads having different, non-harmonic periods). This approach would look like this (an incomplete example):

```
procedure Poll_With_Delay is
   Next_Release : Time;
   Period       : constant Time_Span := Milliseconds (30); -- let's say
begin
   Next_Release := Clock;
   loop
      exit when Status (...);
      Next_Release := Next_Release + Period;
      delay until Next_Release;
```

(continues on next page)

(continued from previous page)

```
   end loop;
end Poll_With_Delay;
```

The code above will check the status of some device every 30 milliseconds (an arbitrary period just for illustration) until the `Status` function result allows the loop to exit. If the device "hangs" the loop is never exited, but as you saw there are ways to address that possibility. When the code does not exit the loop, the next point on the timeline is computed and the task executing the code then suspends, allowing the other tasks in the application to execute. Eventually, the next release point is reached and so the task becomes ready to execute again (and will, subject to priorities).

But how long should the polling task suspend when awaiting the device? We need to suspend long enough for the other tasks to get something done, but not so long that the device isn't handled fast enough. Finding the right balance is often not simple, and is further complicated by the "task switching" time. That's the time it takes to switch the execution context from one task to another, in this case in response to the "delay until" statement suspending the polling task. And it must be considered in both directions: when the delay expires we'll eventually switch back to the polling task.

As you can see, polling is easily expressed but has potentially significant drawbacks and architectural ramifications so it should be avoided as a default approach.

Now let's explore the Ada interrupt facilities.

6.2 Language-Defined Interrupt Model

The Ada language standard defines a model for hardware interrupts, as well as language-defined mechanisms for handling interrupts consistent with that model. The model is defined in Annex C, the "Systems Programming" annex, section 3 "Interrupt Support." The following is the text of that section with only a few simplifications and elisions.

- Interrupts are said to occur. An occurrence of an interrupt is separable into generation and delivery.
  - Generation of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program.
  - Delivery is the action that invokes part of the program as response to the interrupt occurrence.
- Between generation and delivery, the interrupt occurrence is pending.
- Some or all interrupts may be blocked. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.
- Certain interrupts are reserved. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other RTL-defined means. The set of reserved interrupts is determined by the hardware and run-time library (RTL).
- Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be attached to that interrupt. The execution of that program unit, the interrupt handler, is invoked upon delivery of the interrupt occurrence.
- While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt.
- The corresponding interrupt is blocked while the handler executes. While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is determined by the hardware and the RTL.

- Each interrupt has a default treatment which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is defined by the RTL.
- An exception propagated from a handler that is invoked by an interrupt has no effect. In particular, it is not propagated out of the handler, in the same way that exceptions do not propagate outside of task bodies.
- If the `Ceiling_Locking` policy is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object. ("Protected object" is abbreviated as "PO" for convenience).
- If the hardware or the underlying system holds pending interrupt occurrences, the RTL must provide for later delivery of these occurrences to the program.

(The above is not everything in the model but we can ignore the rest in this introduction.)

Because interrupt occurrences are generated by the hardware and delivered by the underlying system software (run-time library or real-time operating system), the application code is mainly responsible for responding to occurrences. Of course, the application must first configure the relevant external devices so that they generate the expected interrupts.

The actual response is application-specific but is also hardware-specific. The latter often (but not always) requires clearing the interrupt status within the generating device so that the same occurrence is not delivered again.

Furthermore, the standard model requires the underlying software to block further occurrences while the handler executes, and only allow preemption by higher-priory interrupt occurrences (if any). The application handlers are not responsible for these semantics either. As you will see, the choice of program unit used for expressing handlers makes this all very convenient for the developer.

As a consequence, in terms of the response, the application developer must write the specific handlers and attach those handlers to the corresponding interrupts. Attaching the handlers is implemented in the underlying system software, and it is this same underlying software that delivers the occurrences.

We will now explore the Ada facilities in detail. At the end of this chapter we will explore some common idioms using these mechanisms, especially with regard to the handlers' interaction with the rest of the application.

6.3 Interrupt Handlers

Interrupt handling is, by definition, asynchronous: some event occurs that causes the processor to suspend the application, respond to the event, and then resume application execution.

Because these events are asynchronous, the actions performed by the interrupt handler and the application are subject to the same sorts of race conditions as multiple tasks acting on shared data.

For example, a "reader" task may be in the act of reading (copying) the value of some shared variable, only to be preempted by a "writer" task that updates the value of the variable. In that case, when the "reader" task resumes execution, it will finish the read operation but will, as a result, have a value that is partly from the old value and partly from the new value. The effect is unpredictable. An interrupt handler can have the same effect on shared data as the preempting "writer" task that interrupts the "reader" task. This problem is possible for shared data of any type that is not atomically read or written. You can think of large record objects if that helps, but it even applies to some scalars.

That scenario applies even if no explicit tasks are declared in the application. That's because an implicit "environment task" is executing the main subprogram. In that case, the

main subprogram is the entire application, but more typically some non-null application code is actively executing in one or more tasks.

But it's not just a matter of tasks. We said that interrupts usually have priorities. Typically that means a higher-priority interrupt will preempt the execution of the handler for a lower-priority interrupt. It's the same issue.

Furthermore, the fact that an interrupt has occurred needs to be communicated to the application, for example to say that updated data are available, perhaps a sensor reading or characters from a serial port. As we said above, we usually don't want to poll for that fact, so the application must be able to suspend until the event has occurred. Often we'll have a dedicated task within the application that suspends, rather than the entire application, but that's an application detail.

Ada's protected objects address all these asynchronous issues. Shared data declared within a protected object can be accessed only via protected procedures or protected entries, both of which execute with mutually exclusive access. Hence no race conditions are possible.

Here is an extremely simple, but realistic, example of a PO. This is not an interrupt handler example — we'll get to that — but it does show a shared variable and a protected procedure that executes with mutually exclusive access no matter how many tasks concurrently call it. The PO provides unique serial numbers.

```
protected Serial_Number is
   procedure Get_Next (Number : out Positive);
private
   Value : Positive := 1;
end Serial_Number;


protected body Serial_Number is

   procedure Get_Next (Number : out Positive) is
   begin
      Number := Value;
      Value := Value + 1;
   end Get_Next;

end Serial_Number;
```

Imagine there are multiple assembly lines creating devices of various sorts. Each device gets a unique serial number. These assembly lines run concurrently, so the calls to `Get_Next` occur concurrently. Without mutually exclusive access to the `Value` variable, multiple devices could get the same serial number.

Protected entries can suspend a caller until some condition is true; in this case, the fact that an interrupt has occurred and been handled. (As we will see, a protected entry is not the only way to synchronize with an accessing task, but it is the most robust and general.)

Here's an example of a PO with a protected entry:

```
protected type Persistent_Signal is
   entry Wait;
   procedure Send;
private
   Signal_Arrived : Boolean := False;
end Persistent_Signal;


protected body Persistent_Signal is

   entry Wait when Signal_Arrived is
   begin
```

(continues on next page)

(continued from previous page)

```
      Signal_Arrived := False;
   end Wait;

   procedure Send is
   begin
      Signal_Arrived := True;
   end Send;

end Persistent_Signal;
```

This is a PO providing a "Persistent Signal" abstraction. It allows a task to wait for a "signal" from another task. The signal is not lost if the receiving task is not already waiting, hence the term "persistent." Specifically, if `Signal_Arrived` is **`False`**, a caller to `Wait` will be suspended until `Signal_Arrived` becomes **`True`**. A caller to Send sets `Signal_Arrived` to **`True`**. If a caller to `Wait` was already present, suspended, it will be allowed to continue execution. If no caller was waiting, eventually some caller will arrive, find `Signal_Arrived` **`True`**, and will be allowed to continue. In either case, the `Signal_Arrived` flag will be set back to **`False`** before the `Wait` caller is released. Protected objects can have a priority assigned, similar to tasks, so they are integrated into the global priority semantics including interrupt priorities.

Therefore, in Ada an interrupt handler is a protected procedure declared within some protected object (PO). A given PO may handle more than one interrupt, and if so, may use one or more protected procedures to do so.

Interrupts can be attached to a protected procedure handler using a mechanism we'll discuss shortly. When the corresponding interrupt occurs, the attached handler is invoked. Any exceptions propagated by the handler's execution are ignored and do not go past the procedure.

While the protected procedure handler executes, the corresponding interrupt is blocked. As a consequence, another occurrence of that same interrupt will not preempt the handler's execution. However, if the hardware does not allow interrupts to be blocked, no blocking occurs and a subsequent occurrence would preempt the current execution of the handler. In that case, your handlers must be written with that possibility in mind. Most targets do block interrupts so we will assume that behavior in the following descriptions.

The standard mutually exclusive access provided to the execution of protected procedures and entries is enforced whether the "call" originates in hardware, via an interrupt, or in the application software, via some task. While any protected action in the PO executes, the corresponding interrupt is blocked, such that another occurrence will not preempt the execution of that actions' procedure or entry body execution in the PO.

On some processors blocked interrupts are lost, they do not persist. However, if the hardware can deliver an interrupt that had been blocked, the Systems Programming Annex requires the handler to be invoked again later, subject to the PO semantics described above.

The default treatment for a given interrupt depends on the RTL implementation. The default may be to jump immediately to system-defined handler that merely loops forever, thereby "hanging" the system and preventing any further execution of the application. On a bare-board target that would be a very common approach. Alternatively the default could be to ignore the interrupt entirely.

As mentioned earlier, some interrupts may be reserved, meaning that the application cannot install a replacement handler. For instance, most bare-board systems include a clock that is driven by a dedicated interrupt. The application cannot (or at least should not) override the interrupt handler for that interrupt. The determination of which interrupts are reserved is RTL-defined. Attempting to attach a user-defined handler for a reserved interrupt raises `Program_Error`, and the existing treatment is unchanged.

6.4 Interrupt Management

Ada defines a standard package that provides a primary type for identifying individual interrupts, as well as subprograms that take a parameter of that type in order to manage the system's interrupts and handlers. The package is named `Ada.Interrupts`, appropriately.

The primary type in that package is named `Interrupt_Id` and is an compiler-defined discrete type, meaning that it is either an integer type (signed or not) or an enumeration type. That representation is guaranteed so you can be sure that `Interrupt_Id` can be used, for example, as the index for an array type.

Package `Ada.Interrupts` provides functions to query whether a given interrupt is reserved, or if an interrupt has a handler attached. Procedures are defined to allow the application to attach and detach handlers, among other things. These procedures allow the application to dynamically manage interrupts. For example, when a new external device is added, perhaps as a "hot spare" replacing a damaged device, or when a new external device is simply connected to the target, the application can arrange to handle the new interrupts without having to recompile the application or restart application execution.

However, typically you will not use these procedures or functions to manage interrupts. In part that's because the architecture is usually static, i.e., the handlers are set up once and then never changed. In that case you won't need to query whether a given exception is reserved at run-time, or to check whether a handler is attached. You'd know that already, as part of the system architecture choices. For the same reasons, another mechanism for attaching handlers is more commonly used, and will be explained in that section. The package's type `Interrupt_Id`, however, will be used extensively.

A child package `Ada.Interrupts.Names` defines a target-dependent set of constants providing meaningful names for the `Interrupt_Id` values the target supports. Both the number of constants and their names are defined by the compiler, reflecting the variations in hardware available. This package and the enclosed constants are used all the time. For the sake of illustration, here is part of the package declaration for a Cortex M4F microcontroller supported by GNAT:

```
package Ada.Interrupts.Names is
   Sys_Tick_Interrupt     : constant Interrupt_ID := 1;
   ...
   EXTI0_Interrupt        : constant Interrupt_ID := 8;
   ....
   DMA1_Stream0_Interrupt : constant Interrupt_ID := 13;
   ...
   HASH_RNG_Interrupt     : constant Interrupt_ID := 80;
   ...
end Ada.Interrupts.Names;
```

Notice `HASH_RNG_Interrupt`, the name for `Interrupt_Id` value 80 on this target. That is the interrupt that the on-chip random number generator hardware uses to signal that a new value is available. We will use this interrupt in an example at the end of this chapter.

The representation chosen by the compiler for `Interrupt_Id` is very likely an integer, as in the above package, so the child package provides readable names for the numeric values. If `Interrupt_Id` is represented as an enumeration type the enumeral values are probably sufficiently readable, but the child package must be provided by the vendor nonetheless.

6.5 Associating Handlers With Interrupts

As we mentioned above, the Ada standard provides two ways to attach handlers to interrupts. One is procedural, described earlier. The other mechanism is automatic, achieved during elaboration of the protected object enclosing the handler procedure. The behavior is not unlike the activation of tasks: declared tasks are activated automatically as a result

of their elaboration, whereas dynamically allocated tasks are activated as a result of their allocations.

We will focus exclusively on the automatic, elaboration-driven attachment model because that is the more common usage, and as a result, that is what GNAT supports on bare-board targets. It is also the mechanism that the standard Ravenscar and Jorvik profiles require. Our examples are consistent with those targets.

In the elaboration-based attachment model, we specify the interrupt to be attached to a given protected procedure within a protected object. This interrupt specification occurs within the enclosing protected object declaration. (Details in a moment.) When the enclosing PO is elaborated, the run-time library installs that procedure as the handler for that interrupt. A given PO may contain one or more interrupt handler procedures, as well as any other protected subprograms and entries.

In particular, we can associate an interrupt with a protected procedure by applying the aspect `Attach_Handler` to that procedure as part of its declaration, with the `Interrupt_Id` value as the aspect parameter. The association can also be achieved via a pragma with the same name as the aspect. Strictly speaking, the pragma `Attach_Handler` is obsolescent, but that just means that there is a newer way to make the association (i.e., the aspect). The pragma is not illegal and will remain supported. Because the pragma existed in a version of Ada prior to aspects you will see a lot of existing code using the pragma. You should become familiar with it. There's no language-driven reason to change the source code to use the aspect. New code should arguably use the aspect, but there's no technical reason to prefer one over the other.

Here is an example of a protected object with one protected procedure interrupt handler. It uses the `Attach_Handler` aspect to tie a random number generator interrupt to the `RNG_Controller.Interrupt_Handler` procedure:

```
protected RNG_Controller is
   ...
   entry Get_Random (Value : out UInt32);
private

   Last_Sample    : UInt32 := 0;
   Buffer         : Ring_Buffer;
   Data_Available : Boolean := False;

   procedure Interrupt_Handler with
      Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;

end RNG_Controller;
```

That's all that the developer must do to install the handler. The compiler and run-time library do the rest, automatically.

The local variables are declared in the private part, as required by the language, because they are shared data meant to be protected from race conditions. Therefore, the only compile-time access possible is via visible subprograms and entries declare in the visible part. Those subprograms and entries execute with mutually exclusive access so no race conditions are possible, as guaranteed by the language.

Note that procedure `Interrupt_Handler` is declared in the private part of `RNG_Controller`, rather than the visible part. That location is purely a matter of choice (unlike the variables), but there is a good reason to hide it: application software can call an interrupt handler procedure too. If you don't ever intend for that to happen, have the compiler enforce your intent. An alert code reader will then recognize that clients cannot call that procedure. If, on the other hand, the handler is declared in the visible part, the reader must examine more of the code to determine whether there are any callers in the application code. Granted, a software call to an interrupt handler is rare, but not illegal, so you should state your intent in the code in an enforceable manner.

Be aware that the Ada compiler is allowed to place restrictions on protected procedure handlers. The compiler can restrict the content of the procedure body, for example, or it might forbid calls to the handler from the application software. The rationale is to allow direct invocation by the hardware, to minimize interrupt latency to the extent possible.

For completeness, here's the same RNG_Controller protected object using the pragma instead of the aspect to attach the interrupt to the handler procedure:

```
protected RNG_Controller is
   ...
   entry Get_Random (Value : out UInt32);
private

   Last_Sample    : UInt32 := 0;
   Buffer         : Ring_Buffer;
   Data_Available : Boolean := False;

   procedure Interrupt_Handler;
   pragma Attach_Handler (Interrupt_Handler,
                          Ada.Interrupts.Names.HASH_RNG_Interrupt;

end RNG_Controller;
```

As you can see, there isn't much difference. The aspect is somewhat more succinct. (The choice of where to declare the procedure remains the same.)

In this attachment model, protected declarations containing interrupt handlers must be declared at the library level. That means they must be declared in library packages. (Protected objects cannot be library units themselves, just as tasks cannot. They must be declared within some other unit.) Here is the full declaration for the RNG_Controller PO declared within a package — in this case within a package body:

```
with Ada.Interrupts.Names;
with Bounded_Ring_Buffers;

package body STM32.RNG.Interrupts is

   package UInt32_Buffers is new Bounded_Ring_Buffers (Content => UInt32);
   use UInt32_Buffers;


   protected RNG_Controller is
      ...
      entry Get_Random (Value : out UInt32);
   private

      Last_Sample    : UInt32 := 0;
      Samples        : Ring_Buffer (Upper_Bound => 9);  -- arbitrary
      Data_Available : Boolean := False;

      procedure Interrupt_Handler with
        Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;

   end RNG_Controller;

   ...

end STM32.RNG.Interrupts;
```

But note that we're talking about protected declarations, a technical term that encompasses not only protected types but also anonymously-typed protected objects. In the RNG_Controller example, the PO does not have an explicit type declared; it is anonymously-typed. (Task objects can also be anonymously-typed.) You don't have to

use a two-step process of first declaring the type and then an object of the type. If you only need one, no explicit type is required.

Although interrupt handler protected types must be declared at library level, the Ada model allows you to have an object of the type declared elsewhere, not necessarily at library level. However, note that the Ravenscar and Jorvik profiles require protected interrupt handler objects — anonymously-typed or not — to be declared at the library level too, for the sake of analysis. The profiles also require the elaboration-based attachment mechanism we have shown. For the sake of the widest applicability, and because with GNAT the most likely use-case involves either Ravenscar or Jorvik, we are following those restrictions in our examples.

6.6 Interrupt Priorities

Many (but not all) processors assign priorities to interrupts, with blocking and preemption among priorities of different levels, much like preemptive priority-based task semantics. Consequently, the priority semantics for interrupt handlers are as if a hardware "task," executing at an interrupt level priority, calls the protected procedure handler.

Interrupt handlers in Ada are protected procedures, which do not have priorities individually, but the enclosing protected object can be assigned a priority that will apply to the handler(s) when executing.

Therefore, protected objects can have priorities assigned using values of subtype `System.Interrupt_Priority`, which are high enough to require the blocking of one or more interrupts. The specific values among the priority subtypes are not standardized but the intent is that interrupt priorities are higher (more urgent) than non-interrupt priorities, as if they are declared like so in package `System`:

```
subtype Any_Priority is Integer range compiler-defined;

subtype Priority is Any_Priority
   range Any_Priority'First .. compiler-defined;

subtype Interrupt_Priority is Any_Priority
   range Priority'Last + 1 .. Any_Priority'Last;
```

For example, here are the subtype declarations in the GNAT compiler for an Arm Cortex M4 target:

```
subtype Any_Priority       is Integer      range 0 .. 255;
subtype Priority           is Any_Priority range Any_Priority'First .. 240;
subtype Interrupt_Priority is Any_Priority range
   Priority'Last + 1  .. Any_Priority'Last;
```

Although the ranges are compiler-defined, when the Systems Programming Annex is implemented the range of `System.Interrupt_Priority` must include at least one value. Vendors are not required to have a distinct priority value in `Interrupt_Priority` for each hardware interrupt possible on a given target. On a bare-metal target, they probably will have a one-to-one correspondence, but might not in a target with an RTOS or host OS.

A PO containing an interrupt handler procedure must be given a priority within the `Interrupt_Priority` subtype's range. To do so, we apply the aspect `Interrupt_Priority` to the PO. Perhaps confusingly, the aspect and the value's required subtype have the same name.

```
with Ada.Interrupts.Names;  use Ada.Interrupts.Names;
with System;                use System;

package Gyro_Interrupts is
```

(continues on next page)

(continued from previous page)

```
   protected Handler with
      Interrupt_Priority => Interrupt_Priority'Last
   is
   private
      procedure IRQ_Handler;
      pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
   end Handler;

end Gyro_Interrupts;
```

The code above uses the highest (most urgent) interrupt priority value but some other value could be used instead, as long as it is in the `Interrupt_Priority` subtype's range. `Constraint_Error` is raised otherwise.

There is also an alternative pragma, now obsolescent, with the same name as the aspect and subtype. Here is an example:

```
with Ada.Interrupts.Names;  use Ada.Interrupts.Names;

package Gyro_Interrupts is

   protected Handler is
      pragma Interrupt_Priority (245);
   private
      procedure IRQ_Handler;
      pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
   end Handler;

end Gyro_Interrupts;
```

In the above we set the interrupt priority to 245, presumably a value conformant with this specific target. You should be familiar with this pragma too, because there is some much existing code using it. New code should use the aspect, ideally.

If we don't specify the priority for some protected object containing an interrupt handler (using either the pragma or the aspect), the initial priority of protected objects of that type is compiler-defined, but within the range of the subtype `Interrupt_Priority`. Generally speaking, you should specify the priorities per those of the interrupts handled, assuming they have distinct values, so that you can reason concretely about the relative blocking behavior at run-time.

Note that the parameter specifying the priority is optional for the `Interrupt_Priority` pragma. When none is given, the effect is as if the value `Interrupt_Priority'Last` was specified.

```
with Ada.Interrupts.Names;  use Ada.Interrupts.Names;

package Gyro_Interrupts is

   protected Handler is
      pragma Interrupt_Priority;
   private
      ...
   end Handler;

end Gyro_Interrupts;
```

No pragma parameter is given in the above, therefore `Gyro_Interrupts.Handler` executes at `Interrupt_Priority'Last` when invoked.

While an interrupt handler is executing, the corresponding interrupt is blocked. Therefore,

the same interrupt will not be delivered again while the handler is executing. Plus, the protected object semantics mean that no software caller is also concurrently executing within the protected object. So no data race conditions are possible. If the system does not support blocking, however, the interrupt is not blocked when the handler executes.

In addition, when interrupt priorities are involved, hardware blocking typically extends to interrupts of equal or lower priority.

You should understand that a higher-priority interrupt could preempt the execution of a lower-priority interrupt's handler. Handlers do not define "critical sections" in which the processor cannot be preempted at all (other than the case of the highest priority interrupt).

Preemption does not cause data races, usually, because the typical case is to have a given protected object handle only one interrupt. It follows that only that one interrupt handler has visibility to the protected data in any given protected object, therefore only that one handler can update it. Any preempting handler would be in a different protected object, hence the preempting handler could not possibly update the data in the preempted handler's PO. No data race condition is possible.

However, protected objects can contain handlers for more than one interrupt. In that case, depending on the priorities, the execution of a higher-priority handler could preempt the execution of a lower priority handler in that same PO. Because each handler in the PO can update the local protected data, these data are effectively shared among asynchronous writers. Data race conditions are, as a result, possible.

The solution to the case of multiple handlers in a single PO is to assign the PO a priority not less than the highest of the interrupt priorities for which it contains handlers. That's known as the "ceiling priority" and works the same as when applying the ceiling for the priorities of caller tasks in the software. Then, whenever any interrupt handled by that PO is delivered, the handler executes at the ceiling priority, not necessarily the priority of the specific interrupt handled. All interrupts at a priority equal or lower than the PO priority are blocked, so no preemption by another handler within that same PO is possible. As a result, a handler for a higher priority interrupt must be in a different PO. If that higher priority handler is invoked, it can indeed preempt the execution of the handler for the lower priority interrupt in another PO. But because these two handlers will not be in the same PO, they will not share the data, so again no race condition is possible.

Note also that software callers will execute at the PO priority as well, so their priority may be increased during that execution. As you can see, the Ceiling Priority Protocol integrates application-level priorities, for tasks and protected objects, with interrupt-level priorities for interrupt handlers.

The Ceiling Locking Protocol is requested by specifying the `Ceiling_Locking` policy (see ARM D.3) to the pragma `Locking_Policy`. Both Ravenscar and Jorvik do so, automatically.

6.7 Common Design Idioms

In this section we explore some of the common idioms used when writing interrupt handlers in Ada.

6.7.1 Parameterizing Handlers

Suppose we have more than one instance of a kind of device. For example, multiple DMA controllers are often available on a System-on-Chip such as an Arm microcontroller. We can simplify our code by defining a device driver **type**, with one object of the type per supported hardware device. This is the same abstract data type (ADT) approach we'd take for software objects in application code, and in general for device drivers when multiple hardware instances are available.

We can also apply the ADT approach to interrupt handlers when we have multiple devices of a given kind that can generate interrupts. In this case, the type will be fully implemented

as a protected type containing at least one interrupt handling procedure, with or without additional protected procedures or entries.

As is the case with abstract data types in general, we can tailor each object with discriminants defined with the type, in order to "parameterize" the type and thus allow distinct objects to have different characteristics. For example, we might define a bounded buffer ADT with a discriminant specifying the upper bound, so that distinct objects of the single type could have different bounds. In the case of hardware device instances, one of these parameters will often specify the device being driven, but we can also specify other device-specific characteristics. In particular, for interrupt handler types both the interrupt to handle and the interrupt priority can be discriminants. That's possible because the aspects/pragmas do not require their values to be specified via literals, unlike what was done in the `RNG_Controller` example above.

For example, here is the declaration for an interrupt handler ADT named `DMA_Interrupt_Controller`. This type manages the interrupts for a given DMA device, known as a `DMA_Controller`. Type `DMA_Controller` is itself an abstract data type, declared elsewhere.

```
protected type DMA_Interrupt_Controller
  (Controller   : not null access DMA_Controller;
   Stream       : DMA_Stream_Selector;
   IRQ          : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System.Interrupt_Priority)
with
   Interrupt_Priority => IRQ_Priority
is

   procedure Start_Transfer
     (Source      : Address;
      Destination : Address;
      Data_Count  : UInt16);

   procedure Abort_Transfer (Result : out DMA_Error_Code);

   procedure Clear_Transfer_State;

   function Buffer_Error return Boolean;

   entry Wait_For_Completion (Status : out DMA_Error_Code);

private

   procedure Interrupt_Handler with Attach_Handler => IRQ;

   No_Transfer_In_Progess : Boolean := True;
   Last_Status            : DMA_Error_Code := DMA_No_Error;
   Had_Buffer_Error       : Boolean := False;

end DMA_Interrupt_Controller;
```

In the above, the `Controller` discriminant provides an access value designating the specific `DMA_Controller` device instance to be managed. Each DMA device supports multiple independent conversion "streams" so the `Stream` discriminant specifies that characteristic. The `IRQ` and `IRQ_Priority` discriminants specify the handler values for that specific device and stream. These discriminant values are then used in the `Interrupt_Priority` pragma and the `Attach_Handler` aspect in the private part. ("IRQ" is a command handler name across programming languages, and is an abbreviation for "interrupt request.")

Here then are the declarations for two instances of the interrupt handler type:

```
DMA2_Stream0 : DMA_Interrupt_Controller
   (Controller   => DMA_2'Access,
    Stream       => Stream_0,
    IRQ          => DMA2_Stream0_Interrupt,
    IRQ_Priority => Interrupt_Priority'Last);

DMA2_Stream5 : DMA_Interrupt_Controller
   (Controller   => DMA_2'Access,
    Stream       => Stream_5,
    IRQ          => DMA2_Stream5_Interrupt,
    IRQ_Priority => Interrupt_Priority'Last);
```

In the above, both objects `DMA2_Stream0` and `DMA2_Stream5` are associated with the same object named `DMA2`, an instance of the `DMA_Controller` type. The difference in the objects is the stream that generates the interrupts they handle. One object handles `Stream_0` interrupts and the other handles those from `Stream_5`. Package `Ada.Interrupts.Names` for this target (for GNAT) declares distinct names for the streams and devices generating the interrupts, hence `DMA2_Stream0_Interrupt` and `DMA2_Stream5_Interrupt`.

On both objects the priority is the highest interrupt priority (and hence the highest overall), `Interrupt_Priority'Last`. That will work, but of course all interrupts will be blocked during the execution of the handler, as well as the execution of any other subprogram or entry in the same PO. That means that the clock interrupt is blocked for that interval, for example. We use that interrupt value in our demonstrations for expedience, but in a real application you'd almost certainly use a lower value specific to the interrupt handled.

We could reduce the number of discriminants, and also make the code more robust, by taking advantage of the requirement that type `Interrupt_Id` be a discrete type. As such, it can be used as the index type into arrays. Here is a driver example with only the `Interrupt_Id` discriminant required:

```
Device_Priority : constant array (Interrupt_Id) of Interrupt_Priority := ( ... );

protected type Device_Interface
  (IRQ : Interrupt_Id)
with
   Interrupt_Priority => Device_Priority (IRQ)
is
   procedure Handler with Attach_Handler => IRQ;
   ...
end Device_Interface;
```

Now we use the one `IRQ` discriminant both to assign the priorities for distinct objects and to attach their handler procedures.

6.7.2 Multi-Level Handlers

Interrupt handlers are intended to be very brief, in part because they prevent lower priority interrupts and application tasks from executing.

However, complete interrupt processing may require more than just the short protected procedure handler's activity. Therefore, two levels of handling are common: the protected procedure interrupt handler and a task. The handler does the least possible and then signals the task to do the rest.

Of course, sometimes the handler does everything required and just needs to signal the application. In that case, the awakened task does no further "interrupt processing" but simply uses the result.

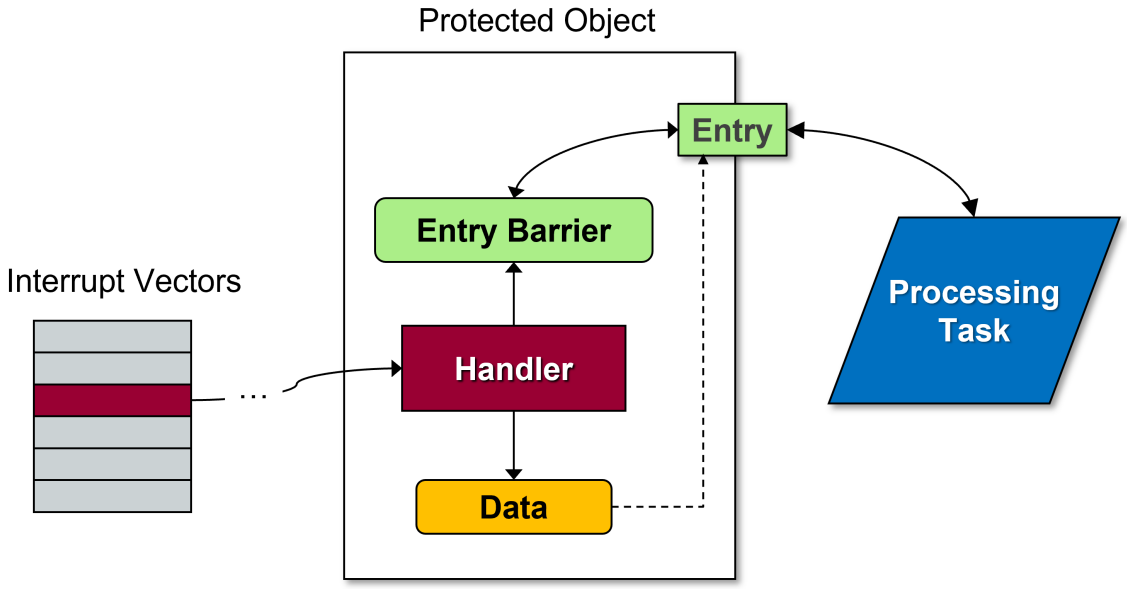
Regardless, the same issues apply: 1) How do application tasks synchronize with the handlers? Assuming the task is not polling the event, at some point the task must stop what

it was doing and suspend, waiting for the handler to signal it. 2) Once synchronized, how can the handlers pass data to the tasks?

Using protected objects for interrupt handling provides an efficient mechanism that elegantly addresses both issues. In addition, when data communication is not required, another standard language mechanism is available. These give rise to two design idioms. We will explore both.

In the first idiom, the protected object contains a protected entry as well as the interrupt handler procedure. The task suspends on the entry when ready for the handler results, controlled by the barrier condition as usual. The protected handler procedure responds to interrupts, managing data (if any) as required. When ready, based on what the handler does, the handler sets the entry barrier to `True`. That allows the suspended task to execute the entry body. The entry body can do whatever is required, possibly just copying the local protected data to the entry parameters. Of course, the entry may be used purely for synchronizing with the handler, i.e., suspending and resuming the task, in which case there would be no parameters passed.

The image below depicts this design.



The `DMA_Interrupt_Controller` described earlier actually uses this design.

```
protected type DMA_Interrupt_Controller
  (Controller   : not null access DMA_Controller;
   Stream       : DMA_Stream_Selector;
   IRQ          : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System.Interrupt_Priority)
with
   Interrupt_Priority => IRQ_Priority
is

   procedure Start_Transfer
     (Source      : Address;
      Destination : Address;
      Data_Count  : UInt16);

   procedure Abort_Transfer (Result : out DMA_Error_Code);
```

(continues on next page)

(continued from previous page)

```
   procedure Clear_Transfer_State;

   function Buffer_Error return Boolean;

   entry Wait_For_Completion (Status : out DMA_Error_Code);

private

   procedure Interrupt_Handler with Attach_Handler => IRQ;

   No_Transfer_In_Progess : Boolean := True;
   Last_Status            : DMA_Error_Code := DMA_No_Error;
   Had_Buffer_Error       : Boolean := False;

end DMA_Interrupt_Controller;
```

The client application code (task) calls procedure `Start_Transfer` to initiate the DMA transaction, then presumably goes off to accomplish something else, and eventually calls the `Wait_For_Completion` entry. That call blocks the task if the device has not yet completed the DMA transfer. The interrupt handler procedure, cleverly named `Interrupt_Handler`, handles the interrupts, one of which indicates that the transfer has completed. Device errors also generate interrupts so the handler detects them and acts accordingly. Eventually, the handler sets the barrier to **`True`** and the task can get the status via the entry parameter.

```
procedure Start_Transfer
   (Source      : Address;
    Destination : Address;
    Data_Count  : UInt16)
is
begin
   No_Transfer_In_Progess := False;
   Had_Buffer_Error := False;
   Clear_All_Status (Controller.all, Stream);
   Start_Transfer_with_Interrupts
     (Controller.all,
      Stream,
      Source,
      ...,
      Enabled_Interrupts =>
        (Half_Transfer_Complete_Interrupt => False,
         others  => True));
end Start_Transfer;


entry Wait_For_Completion
   (Status : out DMA_Error_Code)
when
   No_Transfer_In_Progress
is
begin
   Status := Last_Status;
end Wait_For_Completion;
```

In the above, the entry barrier consists of the Boolean variable `No_Transfer_In_Progress`. Procedure `Start_Transfer` first sets that variable to **`False`** so that a caller to `Wait_For_Completion` will suspend until the transaction completes one way or the other. Eventually, the handler sets `No_Transfer_In_Progress` to **`True`**.

```
procedure Interrupt_Handler is
```

(continues on next page)

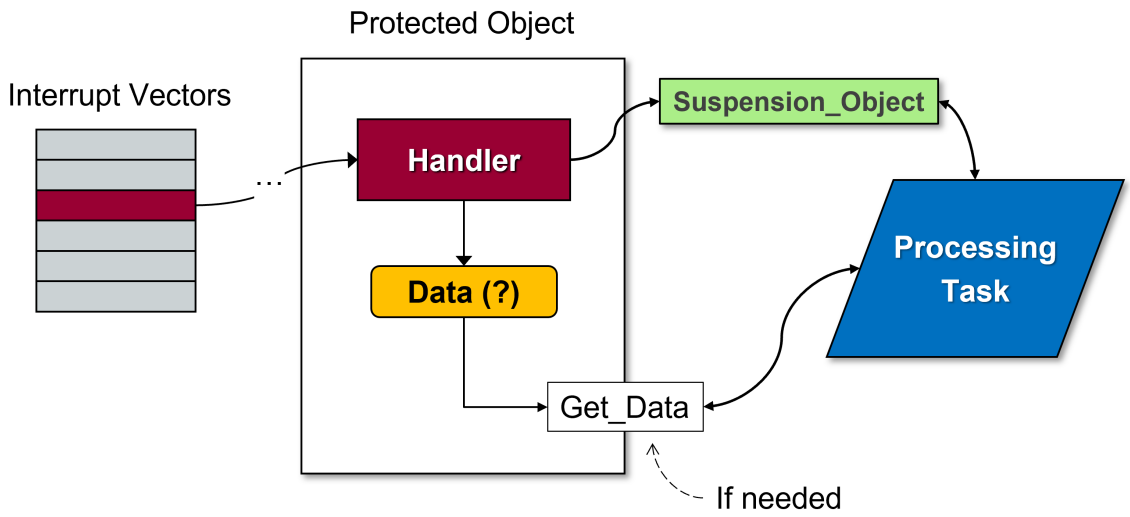
(continued from previous page)

```
   subtype Checked_Status_Flag is DMA_Status_Flag with
      Static_Predicate => Checked_Status_Flag /= Half_Transfer_Complete_Indicated;
begin
   for Flag in Checked_Status_Flag loop
      if Status (Controller.all, Stream, Flag) then
         case Flag is
            when FIFO_Error_Indicated =>
               Last_Status := DMA_FIFO_Error;
               Had_Buffer_Error := True;
               No_Transfer_In_Progess := not Enabled (Controller.all, Stream);
            when Direct_Mode_Error_Indicated =>
               Last_Status := DMA_Direct_Mode_Error;
               No_Transfer_In_Progess := not Enabled (Controller.all, Stream);
            when Transfer_Error_Indicated =>
               Last_Status := DMA_Transfer_Error;
               No_Transfer_In_Progess := True;
            when Transfer_Complete_Indicated =>
               Last_Status := DMA_No_Error;
               No_Transfer_In_Progess := True;
         end case;
         Clear_Status (Controller.all, Stream, Flag);
      end if;
   end loop;
end Interrupt_Handler;
```

This device driver doesn't bother with interrupts indicating that transfers are half-way complete so that specific status flag is ignored. In response to an interrupt, the handler checks each status flag to determine what happened. Note the resulting assignments for both the protected variables `Last_Status` and `No_Transfer_In_Progess`. The variable `No_Transfer_In_Progess` controls the entry, and `Last_Status` is passed to the caller via the entry formal parameter. When the interrupt handler exits, the resulting protected action allows the now-enabled entry call to execute.

In the second design idiom, the handler again synchronizes with the application task, but not using a protected entry.

The image below depicts this design.



In this approach, the task synchronizes with the handler using a `Suspension_Object` variable. The type `Suspension_Object` is defined in the language standard package `Ada.Synchronous_Task_Control`. Essentially, the type provides a thread-safe Boolean flag. Callers can suspend themselves (hence the package name) until another task resumes them by setting the flag to **`True`**. Here's the package declaration, somewhat elided:

```
package Ada.Synchronous_Task_Control is

   type Suspension_Object is limited private;

   procedure Set_True (S : in out Suspension_Object);

   procedure Set_False (S : in out Suspension_Object);

   function Current_State (S : Suspension_Object) return Boolean;

   procedure Suspend_Until_True (S : in out Suspension_Object);

private
   ...
end Ada.Synchronous_Task_Control;
```

Tasks call `Suspend_Until_True` to suspend themselves on some object of the type passed as the parameter. The call suspends the caller until that object becomes **`True`**. If it is already **`True`**, the caller continues immediately. Objects of type `Suspension_Object` are automatically set to **`False`** initially, and become **`True`** via a call to `Set_True`. As part of the return from a call to `Suspend_Until_True`, the flag is set back to **`False`**. As a result, you probably only need those two subprograms.

The interrupt handler procedure responds to interrupts, eventually setting some visible `Suspension_Object` to **`True`** so that the caller will be signaled and resume. Here's an example showing both the protected object, with handler, and a `Suspension_Object` declaration:

```
with Ada.Interrupts.Names;          use Ada.Interrupts.Names;
with Ada.Synchronous_Task_Control;  use Ada.Synchronous_Task_Control;

package Gyro_Interrupts is

   Data_Available : Suspension_Object;

   protected Handler is
      pragma Interrupt_Priority;
   private
      procedure IRQ_Handler
        with Attach_Handler => EXTI2_Interrupt;
   end Handler;

end Gyro_Interrupts;
```

In the code above, `Gyro_Interrupts.Data_Available` is the `Suspension_Object` variable visible both to the interrupt handler PO and the client task.

`EXTI2_Interrupt` is "external interrupt number 2" on this particular microcontroller. It is connected to an external device, not on the SoC itself. Specifically, it is connected to a L3GD20 MEMS motion sensor[9], a three-axis digital output gyroscope. This gyroscope can be either polled or generate interrupts when ever data are available. The handler is very simple:

[9] https://www.st.com/en/mems-and-sensors/l3gd20.html

```
with STM32.EXTI;  use STM32.EXTI;

package body Gyro_Interrupts is

   protected body Handler is

      procedure IRQ_Handler is
      begin
         if External_Interrupt_Pending (EXTI_Line_2) then
            Clear_External_Interrupt (EXTI_Line_2);
            Set_True (Data_Available);
         end if;
      end IRQ_Handler;

   end Handler;

end Gyro_Interrupts;
```

The handler simply clears the interrupt and resumes the caller task via a call to Set_True on the variable declared in the package spec.

The lack of an entry means that no data can be passed to the task via entry parameters. It is possible to pass data to the task but doing so would require an additional protected procedure or function.

The gyroscope hardware device interface is in package L3GD20. Here are the pertinent parts:

```
package L3GD20 is

   type Three_Axis_Gyroscope is tagged limited private;

   procedure Initialize
      (This        : in out Three_Axis_Gyroscope;
       Port        : Any_SPI_Port;
       Chip_Select : Any_GPIO_Point);

   ...

   procedure Enable_Data_Ready_Interrupt  (This : in out Three_Axis_Gyroscope);

   ...

   type Angle_Rate is new Integer_16;

   type Angle_Rates is record
      X : Angle_Rate;  -- pitch, per Figure 2, pg 7 of the Datasheet
      Y : Angle_Rate;  -- roll
      Z : Angle_Rate;  -- yaw
   end record with Size => 3 * 16;

   ...

   procedure Get_Raw_Angle_Rates
      (This  : Three_Axis_Gyroscope;
       Rates : out Angle_Rates);

   ...

end L3GD20;
```

With those packages available, we can write a simple main program to use the gyro. The real demo displayed the readings on an LCD but we've elided all those irrelevant details:

```
with Gyro_Interrupts;
with Ada.Synchronous_Task_Control;  use Ada.Synchronous_Task_Control;
with L3GD20;                        use L3GD20;
with STM32.Board;
...

procedure Demo_L3GD20 is

   Axes : L3GD20.Angle_Rates;

   ...

   procedure Await_Raw_Angle_Rates (Rates : out L3GD20.Angle_Rates) is
   begin
      Suspend_Until_True (Gyro_Interrupts.Data_Available);
      L3GD20.Get_Raw_Angle_Rates (STM32.Board.Gyro, Rates);
   end Await_Raw_Angle_Rates;

   ...

begin
   Configure_Gyro;
   Configure_Gyro_Interrupt;
   ...
   loop
      Await_Raw_Angle_Rates (Axes);
      ...
   end loop;
end Demo_L3GD20;
```

The demo is a main procedure, even though we've been describing the client application code in terms of tasks. The main procedure is executed by the implicit "environment task" so it all still works. `Await_Raw_Angle_Rates` suspends (if necessary) on `Gyro_Interrupts.Data_Available` and then calls `L3GD20.Get_Raw_Angle_Rates` to get the rate values.

The operations provided by `Suspension_Object` are faster than protected entries, and noticeably so. However, that performance difference is due to the fact that `Suspension_Object` provides so much less capability than entries. In particular, there is no notion of protected actions, nor expressive entry barriers for condition synchronization, nor parameters to pass data while synchronized. Most importantly, there is no caller queue, so at most one caller can be waiting at a time on any given `Suspension_Object` variable. You'll get `Program_Error` if you try. Protected entries should be your first design choice. Note that the Ravenscar restrictions can make use of `Suspension_Object` much more likely.

6.8 Final Points

As you can see, the semantics of protected objects are a good fit for interrupt handling. However, other forms of handlers are allowed to be supported. For example, the compiler and RTL for a specific target may include support for interrupts generated by a device known to be available with that target. For illustration, let's imagine the target always has a serial port backed by a UART. In addition to handlers as protected procedure without parameters, perhaps the compiler and RTL support interrupt handlers with a single parameter of type `Unsigned_8` (or larger) as supported by the UART.

Overall, the interrupt model defined and supported by Ada is quite close to the canonical model presented by most programming languages, in part because it matches the model presented by typical hardware.

CHAPTER

SEVEN

CONCLUSION

In the introduction to this course, we defined an "embedded system" as a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system. That, in itself, sets this kind of programming apart from the more typical host-oriented programming. But the context also implies fewer resources are available, especially memory and electrical power, as well as processor power. Add to those limitations a frequent reliability requirement and you have a demanding context for development.

Using Ada can help you in this context, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal in this course was to get you started in using it well.

To that end, we spent a lot of time talking about how to use Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data. Ada has a lot of support for this activity so there was much to explore.

Likewise, we examined development using Ada in combination with other languages, a not uncommon approach. Specifically, we saw how to interface with code and data written in other languages, and how (and why) to work with assembly language. Development in just one language is becoming less common over time so these were important aspects to know.

One of the more distinctive activities of embedded programming involves interacting with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. (This can be one of the more entertaining activities as well.) We covered how to interact with these memory-mapped devices using representation specifications, data structures that simplified the functional code, and time-honored aspects of software engineering, including abstract data types.

Finally, we explored how to handle interrupts in Ada, another distinctive part of embedded systems programming. As we saw, Ada has extensive support for handling interrupts, using the same building blocks — protected objects — used in concurrent programming. These constructs provide a way to handle interrupts that is as portable as possible, in what is otherwise a very hardware-specific endeavor.

In the course, we mentioned a library of freely-available device drivers in Ada known as the Ada Driver Library (ADL). The ADL is a good resource for learning how Ada can be used to develop software for embedded systems using real-world devices and processors. Becoming familiar with it would be a good place to go next. Contributing to it would be even better! The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.