Hardware-Based Data Protection/Isolation at Runtime in Ada Code for Microcontrollers

J. Germán Rivera
(jgrivera67@gmail.com)

Agenda

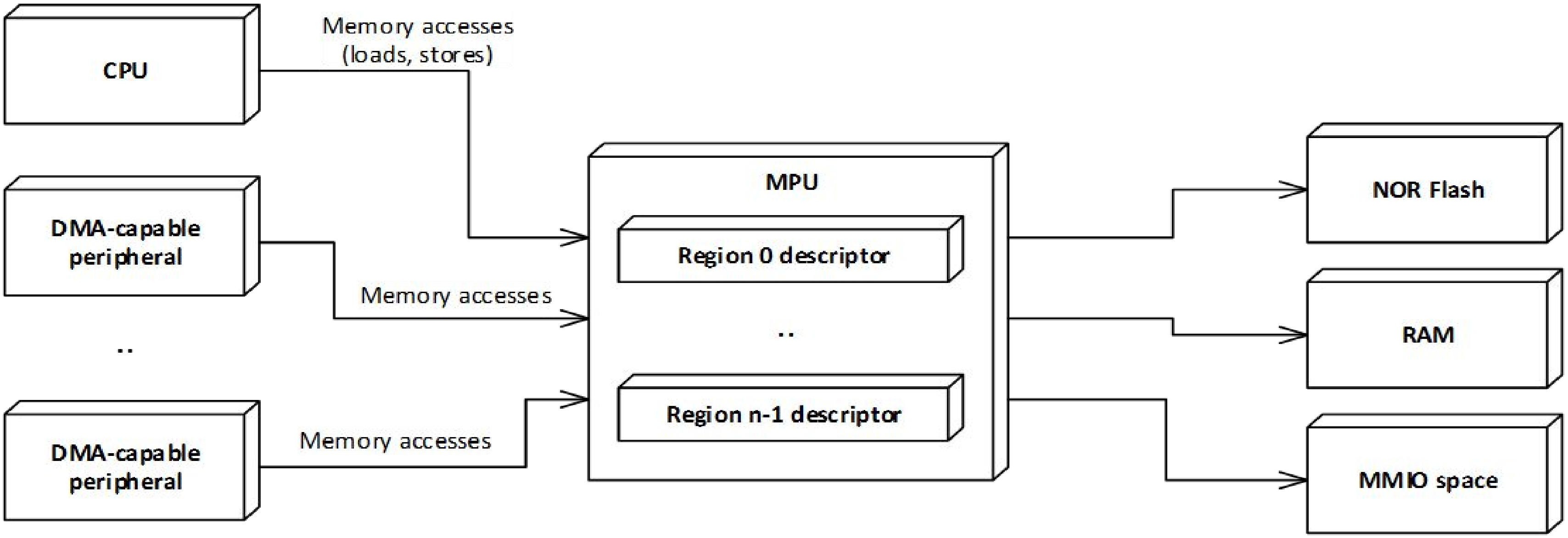
- Motivation
- Using an MPU to Enforce Data Protection
- An MPU-based Data Protection Architecture for Ada Programs
- Ada Code Design Implications
- Changes Required in the Ada RTS
- Variations of the Basic Data Protection Architecture
- Portable Implementation Available in GitHub
- Conclusions

Motivation

- Although data corruption is less likely in code written in Ada than in code written in C/C++, it is still possible (e.g., Ada 2012 **Address** aspect buggy uses).
- Ada programs that call libraries written in C/C++ need to protect themselves from buggy or malicious C/C++ code that can corrupt the Ada data structures.
- Ada programs need to protect themselves from data corruption caused by buggy or malicious DMA transfers that may write (or read) to memory that they are not supposed to.

Using an MPU to Enforce Data Protection

- Many modern small embedded processors (microcontrollers) come with a memory protection unit (MPU) as an alternative to the memory management unit (MMU) from larger processors.



Using an MPU to Enforce Data Protection (2)

- The MPU enables software to control access to areas of the physical memory address space, known as regions
  - Number of regions is limited by the number of region descriptors in the MPU (typically 8-16)
  - Each region descriptor defines a region as an address range along with read/write/execute permissions to access it
  - Each region can be of different size
  - Modern MPUs support regions <u>as small as 32 bytes</u> long.

Using an MPU to Enforce Data Protection (3)

- With this level of protection granularity, it becomes possible to control access at the individual data object or data structure level.
  - This protection granularity can be used to protect an Ada package's private data structures from being corrupted by an errant pointer in another Ada package, C/C++ module or assembly module.
  - A device's MMIO registers can be protected in a similar way.

Using an MPU to Enforce Data Protection (4)

- The MPU can also be used to ensure that security-sensitive/privacy-sensitive data be accessed only by code it is supposed to (even for read-only access).
- Finally, the MPU can be used to ensure that safety-critical code is only invoked from the expected callers, and not accidentally via an invalid function pointer or from a malicious attack.
- MPU region access violations trigger a hardware exception (e.g. Bus fault, MemManage fault)

Using an MPU to Enforce Data Protection (5)

- ARMv7-M architecture MPU:
  - Region size can be as small as 32 bytes but needs to be a power of two
  - A region’s starting address needs to be a multiple of its size
  - A region can have up to 8 sub-regions to compensate for these limitations
  - No DMA access control and only one CPU is supported
- New ARMv8-M architecture MPU is more flexible:
  - Region size does not need to be a power of two, but just a multiple of 32 bytes
  - A region’s starting address does not need to be a multiple of its size, but just 32-byte aligned

Using an MPU to Enforce Data Protection (6)

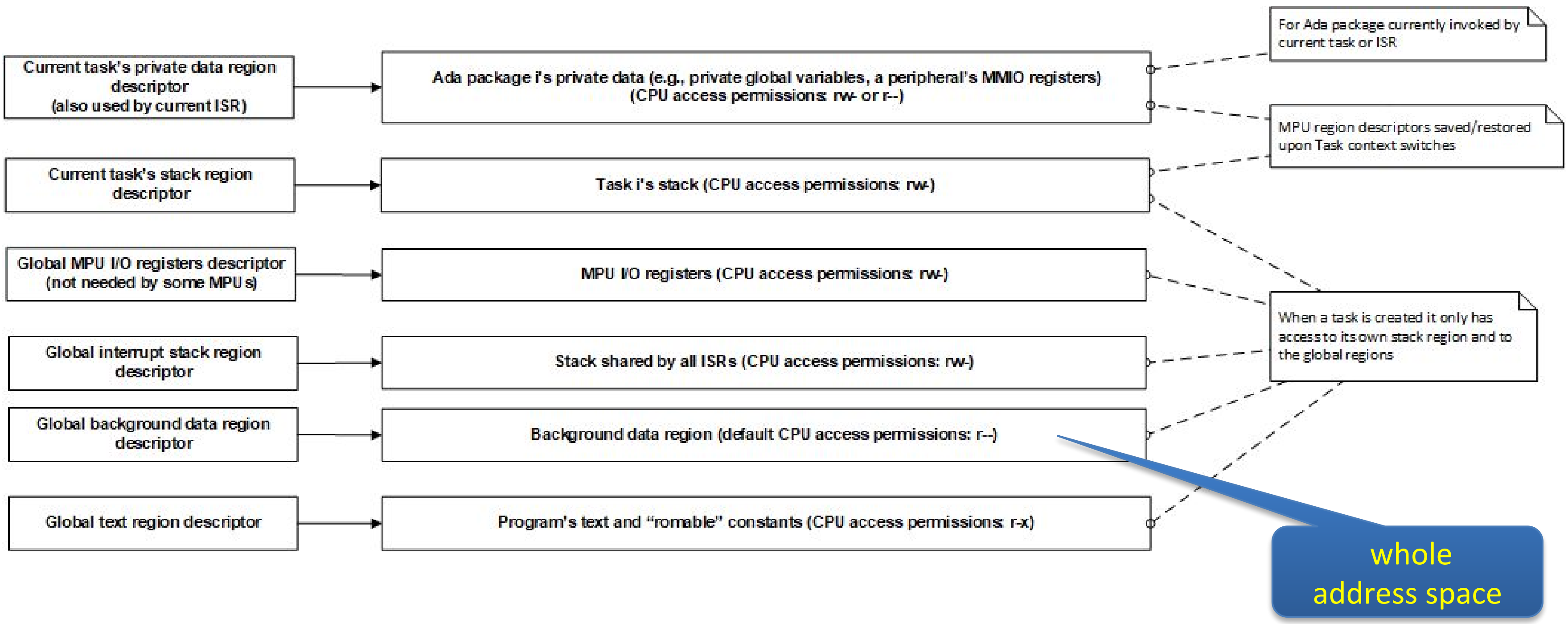
- NXP Kinetis microcontrollers MPU:
  - Region size just needs to be a multiple of 32 bytes
  - Region starting address just needs to be a multiple of 32 (32-byte aligned)
  - Supports multiple bus masters
    - DMA access control for multiple DMA-capable devices and/or more than one CPU

An MPU-based Data Protection Architecture for Ada Programs

- The MPU is programmed so that all RAM data that is not a local variable is read-only by default.
  - By default, the only writable area for an Ada task is its own stack, nothing else. (Same for an interrupt service routine).
  - Non-local variables (statically and dynamically allocated globals) and MMIO registers are not writable by default.
- An Ada package needs to ask permission to the MPU to be able to modify its own private global variables.
  - This may sound a little inconvenient, but it is a small price to pay to ensure the package's data integrity at runtime.

An MPU-based Data Protection Architecture for Ada Programs (2)

- Allocation of MPU Region Descriptors



Ada Code Design Implications

- The private global data of each Ada package must be grouped into a contiguous area of memory
  - A simple way is to use a global **record** data type
  - For maximum protection, this record must occupy a whole MPU region. Example:

```
type My_Global_Data_Type is record
   Field_1 : Type_1
   ..
   Field_N : Type_N;
end record with
   Alignment => MPU_Region_Alignment,
   Size => MPU_Region_Alignment * Byte'Size;
```

32 bytes

Ada Code Design Implications (2)

- Example for a record larger than the smallest region:
  - For ARMv8-M and Kinetis MPUs:

```
type My_Global_Data_Type is record
   ...
end record with
   Alignment => MPU_Region_Alignment,
   Size => 3 * MPU_Region_Alignment * Byte'Size;
```

32 bytes

size in bytes must be multiple of 32

  - For ARMv7-M MPU:

```
type My_Global_Data_Type is record
   ...
end record with
   Alignment => 4 * MPU_Region_Alignment,
   Size => 4 * MPU_Region_Alignment * Byte'Size;
```

alignment must match size in bytes

size in bytes must be a power of 2

Ada Code Design Implications (3)

- A package's public subprograms that modify non-local variables need to:
  - Call **Set_Private_Data_Region**, upon entry, to set the private data region descriptor in the MPU, to point to the package's private global variables or to point to output parameters, with read-write permissions.

```
procedure Set_Private_Data_Region (
     Start_Address : System.Address;
     Size_In_Bits : Integer_Address;
     Permissions : Data_Permissions_Type;
     Old_Region : out MPU_Region_Descriptor_Type);
```

Read_Write

Ada Code Design Implications (4)

- A package's public subprograms that modify non-local variables need to (cont.):
  - Then, call **Set_Private_Data_Region** without saving the previous private region, to switch to other private areas:

```
procedure Set_Private_Data_Region (
     Start_Address : System.Address;
     Size_In_Bits : Integer_Address;
     Permissions : Data_Permissions_Type);
```

Read_Write

  - Call **Restore_Private_Data_Region**, upon exit, to restore the caller's private data region descriptor in the MPU.

```
procedure Restore_Private_Data_Region (
     Saved_Region : MPU_Region_Descriptor_Type);
```

Ada Code Design Implications (6)

- Example 1:

```
procedure My_Public_Proc1(Out_Arg : out Arg_Type) is
   Old_Region : MPU_Region_descriptor_Type;
 begin
   Set_Private_Data_Region (My_Globals'Address,
                            My_Globals'Size,
                            Read_Write, Old_Region);
   ...
   Set_Private_Data_Region (Out_Arg'Address,
                            Out_Arg'Size,
                            Read_Write);
   ...
   Restore_Private_Data_Region (Old_Region);
 end My_Public_Proc1;
```

No need to save the previous private data region again

Ada Code Design Implications (6)

- Example 2:

```
procedure My_Public_Proc2(arg : Arg_Type) is
   Old_Region : MPU_Region_descriptor_Type;
 begin
   Set_Private_Data_Region (My_Data'Address,
                            My_Data'Size,
                            Read_Write, Old_Region);
   ...
   Set_Private_Data_Region (MMIO_registers'Address,
                            MMIO_Registers'Size,
                            Read_Write);
   ...
   Restore_Private_Data_Region (Old_Region);
 end My_Public_Proc1;
```

Input arguments do not need to use the private data region, since read-only access is always allowed

Memory-mapped I/O registers are treated as another form of non-local variables

Changes Required in the Ada RTS

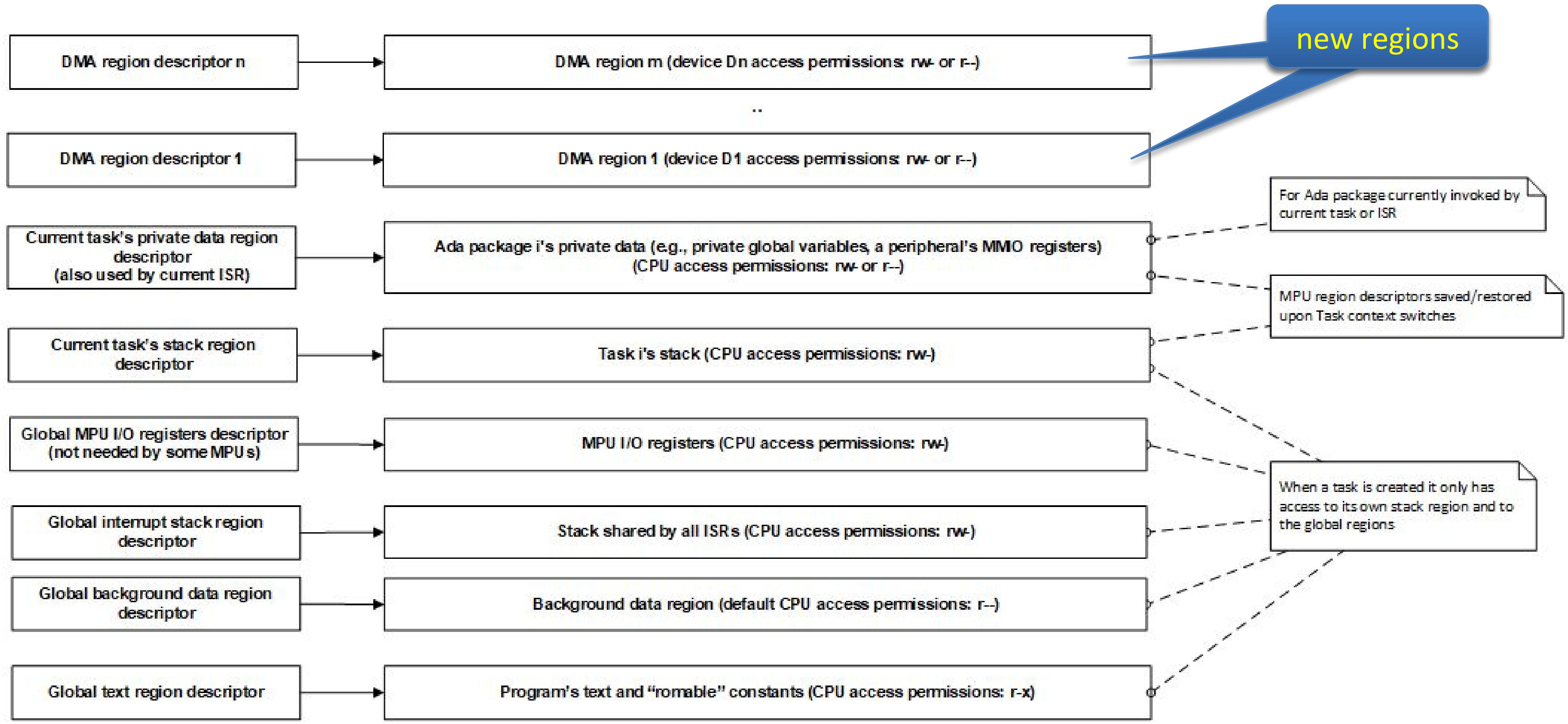
- Task control block:
  - Fields for the following MPU region descriptors need to be added:
    - Task stack region descriptor
    - Private data region descriptor
    - Private code region descriptor (if "hidden" code areas, see later)
- Task creation:
  - Initialize task stack MPU region
- All RTS code that writes to non-local variables:
  - Needs to temporarily make the global background data region writable (or use the private data region)

Changes Required in the Ada RTS (2)

- Startup code (reset exception handler):
  - Add MPU initialization and configuration of global regions
    - MPU initialized but left disabled
    - It needs to be enabled in the application's main subprogram
- Task context switch:
  - The following MPU region descriptors need to be saved/restored:
    - Task stack region descriptor
    - Private data region descriptor
    - Private code region descriptor
  - Since the "writable" on/off state of the global background data region is per task, the permissions for this region need to be saved/restored as well

Variations of the Basic Data Protection Architecture

- Support for DMA access control



Variations of the Basic Data Protection Architecture (2)

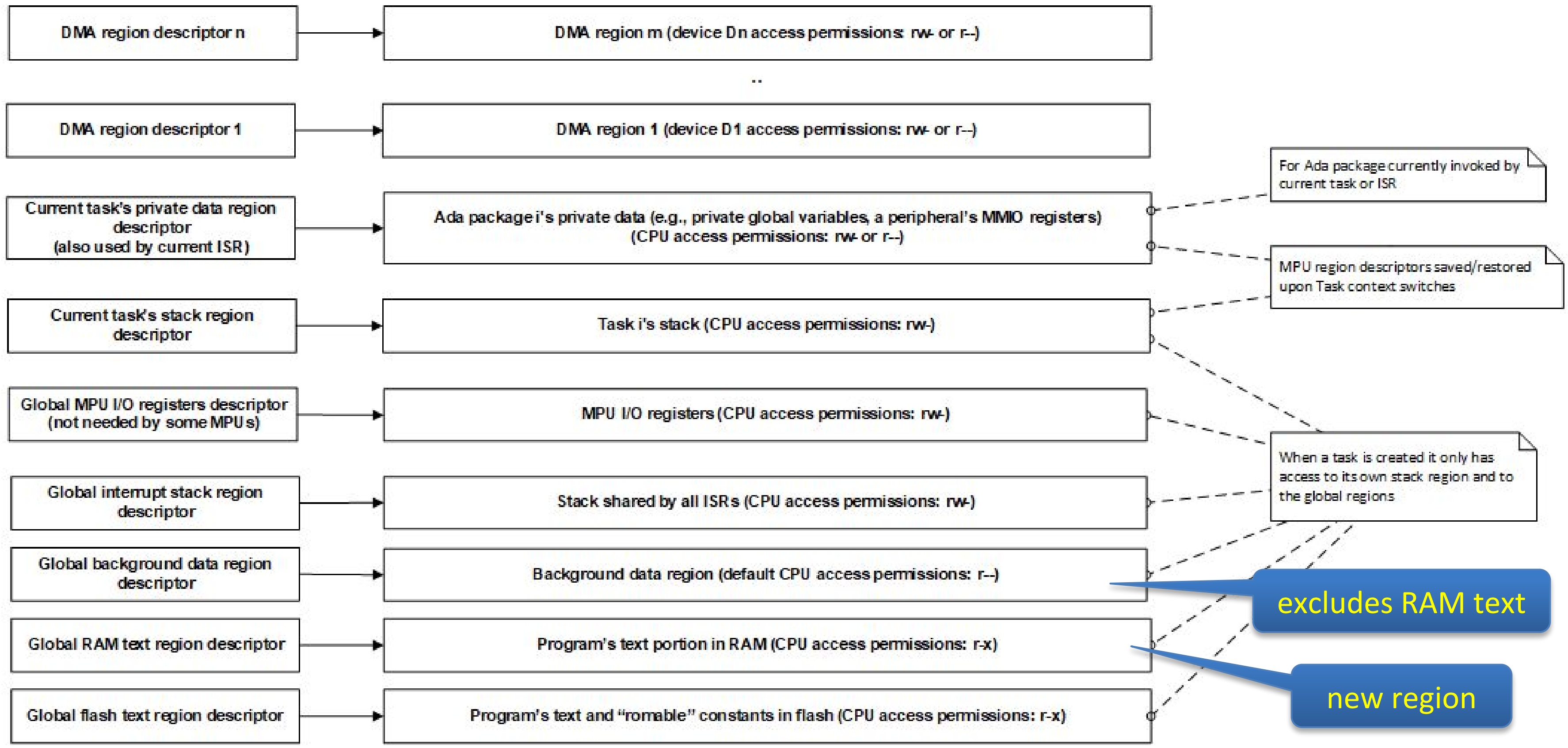
- Support for DMA access control (cont.)
  - Devices driver needs to call **Set_DMA_Region** during device initialization time:

```
procedure Set_DMA_Region (
   Region_Id : MPU_Region_Id_Type;
   DMA_Master : Bus_Master_Type;
   Start_Address : System.Address;
   Size_In_Bits : Integer_Address;
   Permissions : Data_Permissions_Type);
```

Read_Write
or Read_Only

Variations of the Basic Data Protection Architecture (3)

- Support for code in RAM



Variations of the Basic Data Protection Architecture (4)

- Support for code in RAM (cont.)
  - Linker script changes:

```
.data : AT (__rom_end) {
   ...
   __ram_text_start = .;
   *(.ram_text)
   . = ALIGN(MPU_REGION_ALIGNMENT);
   __ram_text_end = .;
   __background_data_region_start = .;
   ...
}
```

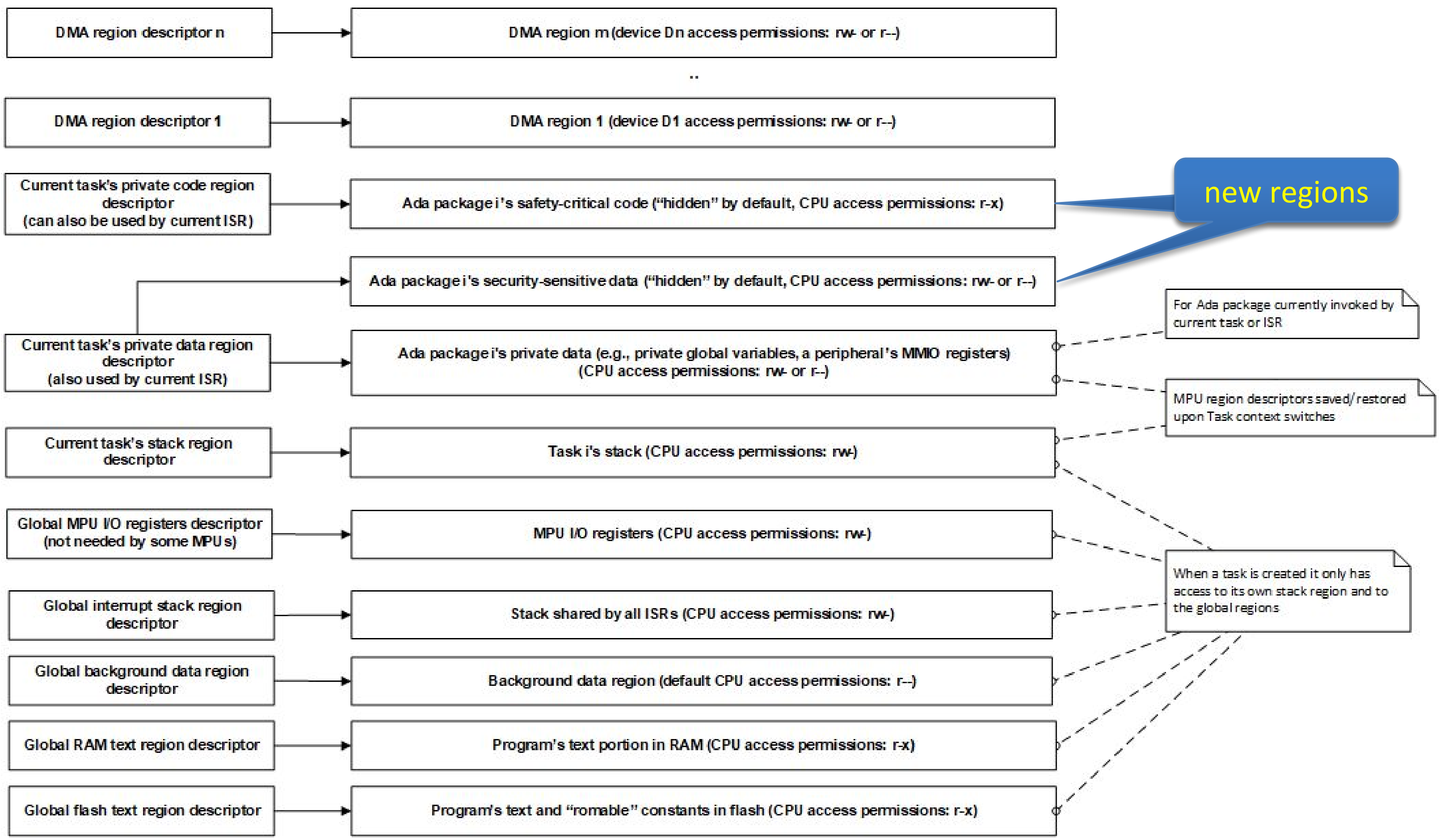
Code placed in RAM

  - Subprogram declaration:

```
procedure My_Public_RAM_Code
   with Linker_Section => ".ram_text";
```

Variations of the Basic Data Protection Architecture (5)

- Support for "hidden" (secret) regions



Variations of the Basic Data Protection Architecture (6)

- Support for "hidden" regions (cont.)
  - To make "hidden" data areas visible, application code needs to call **Set_Private_Data_Region**, with Read_Only or Read_Write permissions.
  - To make "hidden" code visible, application code needs to call **Set_Private_Code_Region:**

```
procedure Set_Private_Code_Region (
     First_Address : System.Address;
     Last_Address : System.Address;
     Old_Region : out MPU_Region_Descriptor_Type);
```

Variations of the Basic Data Protection Architecture (7)

- Support for "hidden" regions (cont.)
  - Linker script changes:

```
.text : {
    . = ALIGN(MPU_REGION_ALIGNMENT);
    __secret_flash_text_start = .;
    *(.secret_flash_text)
    . = ALIGN(MPU_REGION_ALIGNMENT);
    __secret_flash_text_end = .;
    __flash_text_start = .;
    *(.text .text.* .gnu.linkonce.t*)
    *(.gnu.warning)
    . = ALIGN(MPU_REGION_ALIGNMENT);
    __flash_text_end = .;
} > flash_text
```

secret code in flash

public code in flash

Variations of the Basic Data Protection Architecture (8)

- Support for "hidden" regions (cont.)
  - Linker script changes (cont.):

```
.data : AT (__rom_end) {
    . = ALIGN(MPU_REGION_ALIGNMENT);
    __data_start = .;
    __secret_ram_text_start = .;
    *(.secret_ram_text)
    . = ALIGN(MPU_REGION_ALIGNMENT);
    __secret_ram_text_end = .;
    ...
    __secret_data_area_start = .;
    *(.secret_data)
    . = ALIGN(MPU_REGION_ALIGNMENT);
    __secret_data_area_end = .;
    __background_data_region_start = .;
```

secret code in RAM

secret data in RAM

Variations of the Basic Data Protection Architecture (9)

- Support for "hidden" regions (cont.)
  - Ada declarations:

```
My_Secret_Data : My_Secret_Data_Type
   with Linker_Section => ".secret_data";

procedure My_Secret_Flash_Code
   with Linker_Section => ".secret_flash_text";

procedure My_Secret_RAM_Code
   with Linker_Section => ".secret_ram_text";
```

Portable Implementation Available in GitHub

- Support for the Kinetis MPU
  - https://github.com/jgrivera67/embedded-runtimes/blob/master/bsps/kinetis_k64f_common/bsp/memory_protection.ads
  - https://github.com/jgrivera67/embedded-runtimes/blob/master/bsps/kinetis_k64f_common/bsp/memory_protection.adb
- Support for the ARMv7-M MPU
  - https://github.com/jgrivera67/embedded-runtimes/blob/master/bsps/kinetis_kl28z_frdm/bsp/memory_protection.ads
  - https://github.com/jgrivera67/embedded-runtimes/blob/master/bsps/kinetis_kl28z_frdm/bsp/memory_protection.adb
- Usage Examples
  - https://github.com/jgrivera67/make-with-ada/blob/master/hexiwear_iot_stack/mpu_tests.adb

Conclusions

- Data protection at the individual data object level is a novel approach of using a Memory Protection Unit
  - For bare-metal single-address-space embedded platforms, code modules can be protected from corrupting each other’s data structures, even in single-threaded programs.
  - Same approach could be used to protect code modules inside of a process running in an MMU-based operating system (e.g., Linux), if fine-grained MPU functionality for virtual addresses were available as part of the MMU.
  - For object-oriented code, data protection can be done at the individual object instance.

Conclusions (2)

- Ideally, application code should be architected from the beginning to use MPU-based data protection, as opposed to adding it as an afterthought.
- However, MPU-based data protection does not have to be all or nothing
  - Ada tasks executing trusted or legacy code can set the global background data region as writable (as if the MPU was not being used), for the lifetime of the task
  - Only some untrusted components (e.g., third-party libraries or C/C++ code invoked from Ada code) may need to be wrapped in a data protection layer.