All you need to know about Position-Independent Code and Data (PIC and PID)

Agenda

- Position-independent code and data
- Systems with multiple images
- Challenges in this environment
- Demonstration
- Q&A

Position-independent code and data

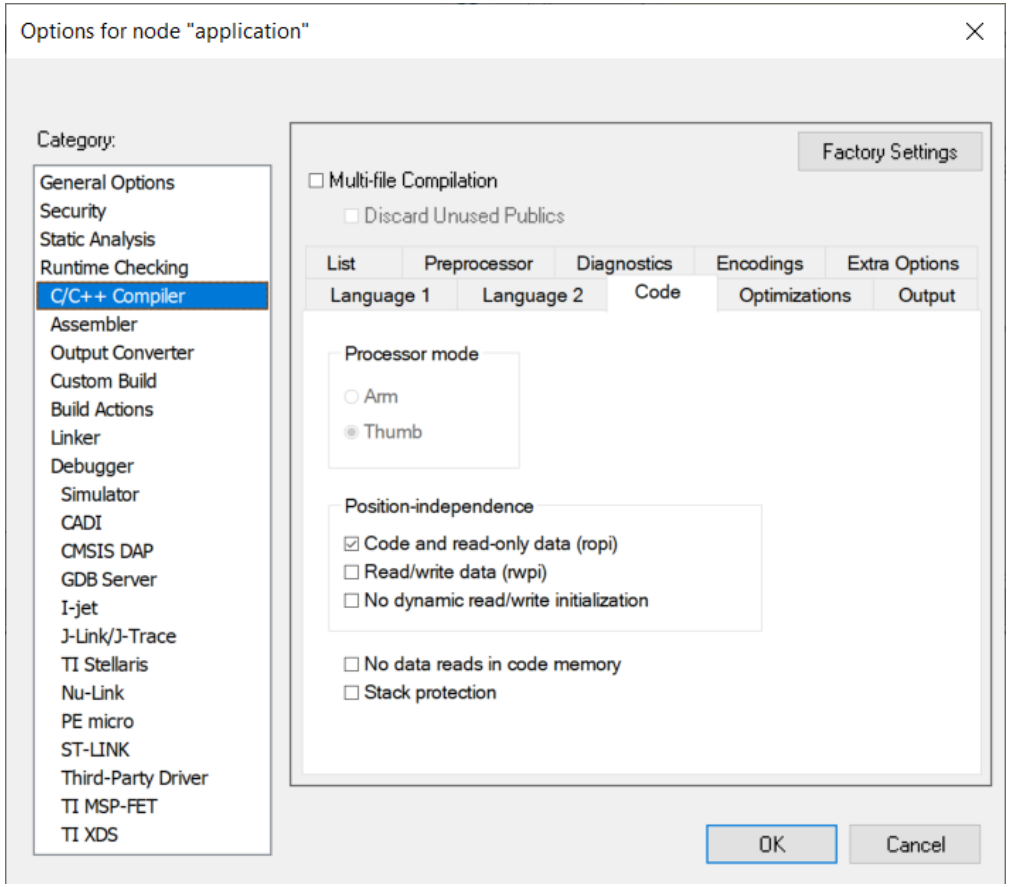
The Basics

Position independent code

- Absolute code
  - Code placed at specific location in memory
- Position independent code (PIC) or position independedent executables + Position independent data (PID)
  - Code placed somewhere in memory
  - Can be executed from any address
  - Binary will not contain jumps to absolute address but rather it will use program counter relative address

Support for PIC and PID

- ROPI = Read-Only Position Independence.
  - This concerns everything that is readonly in the ELF output from the linker.
- RWPI = Read-Write Position Independence.
  - This concerns everything that is readwrite in the ELF output from the linker.
- --pi_veneers (position independent veneers, see the Development Guide for further information)



Limitations with PIC and PID

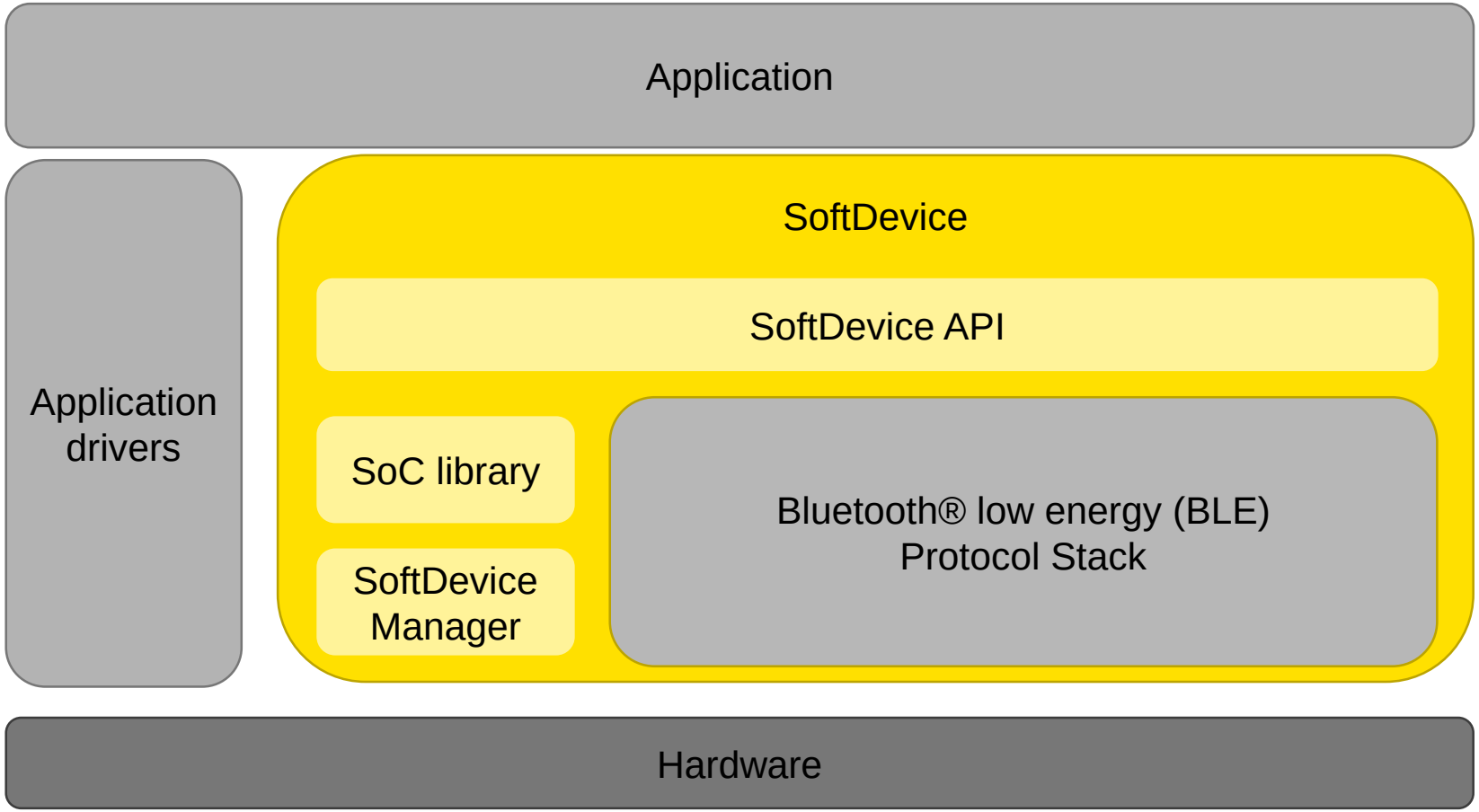
- When PIC/ROPI is used, these limitations apply:
  - C++ constructions cannot be used
  - The object attribute __ramfunc cannot be used
  - Pointer constants cannot be initialized with the address of another constant, a string literal, or a function. However, writable variables can be initialized to constant addresses at runtime.
- When PID/RWPI is used, these limitations apply:
  - The object attribute __ramfunc cannot be used
  - Pointer constants cannot be initialized with the address of a writable variable.

Systems with multiple images

Separately built Protocol Stacks

- Protocol stacks may be provided by silicon vendors for their microcontrollers.
  - Pre-certified communications stacks for easy product certification
  - No need to re-compile in project
- Can be extensive source bases.
- Silicon vendors may consider this proprietary IP.

Example use case



What is a SoftDevice?

- Precompiled hex file
- Programmed separately
- No link time dependency

It is located in reserved memory space, which ensures run time protection.

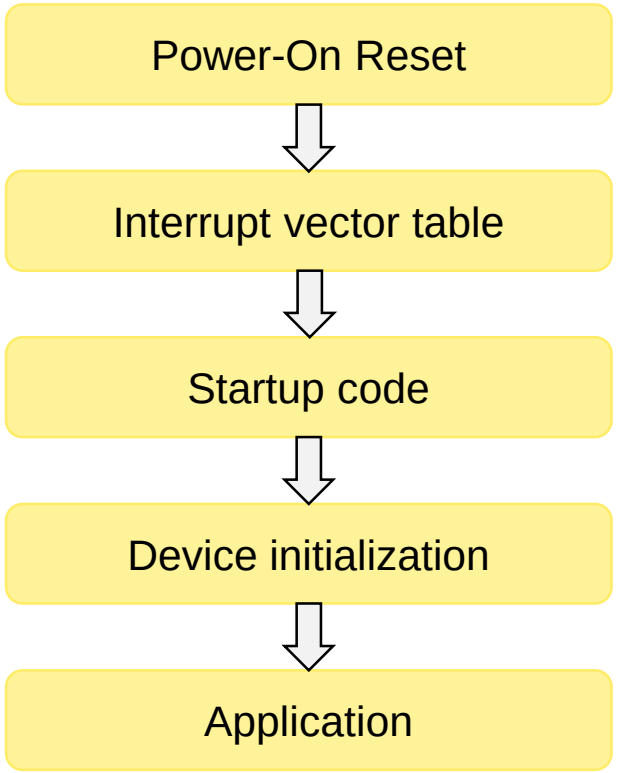
Changing data sets

- An application is kept stable while new data sets may be updated or added
- Many applications may behave this way
  - System that uses Map data may want to update the map data set keeping the access mechanisms the same while the map data is updated
  - Media players may accept new media databases while player itself is not changed
  - These all fit model of simulating an attached file system where none exists in a simple embedded system

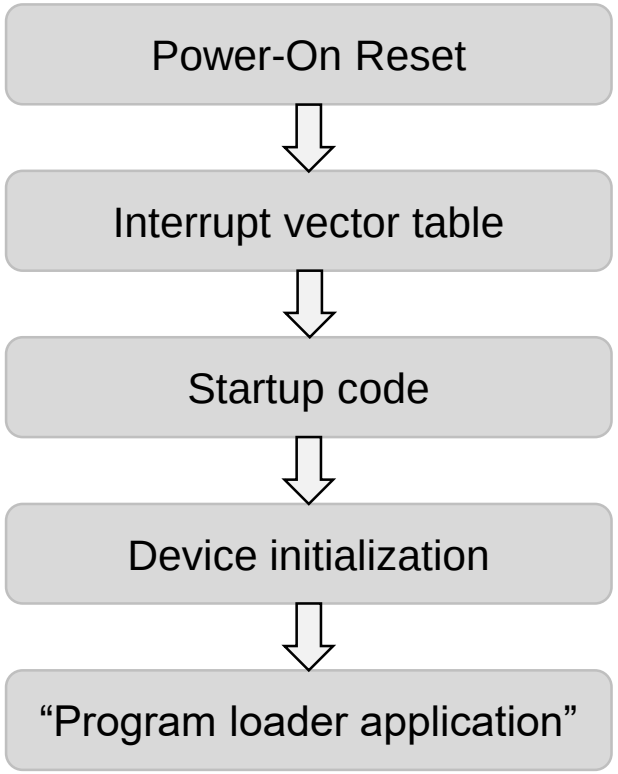
Program loader

- Many systems designed to accept program modifications after shipment
  - Initial system software loading
    - All systems may use same program loader
    - Different loaded application determines system personality
  - Updating system firmware for many reasons
    - Fix basic operating bugs
    - Fix security holes
    - Add new functionality
      - Possibly on a fee-for-service basis
    - Update loader firmware as well

Typical application flow



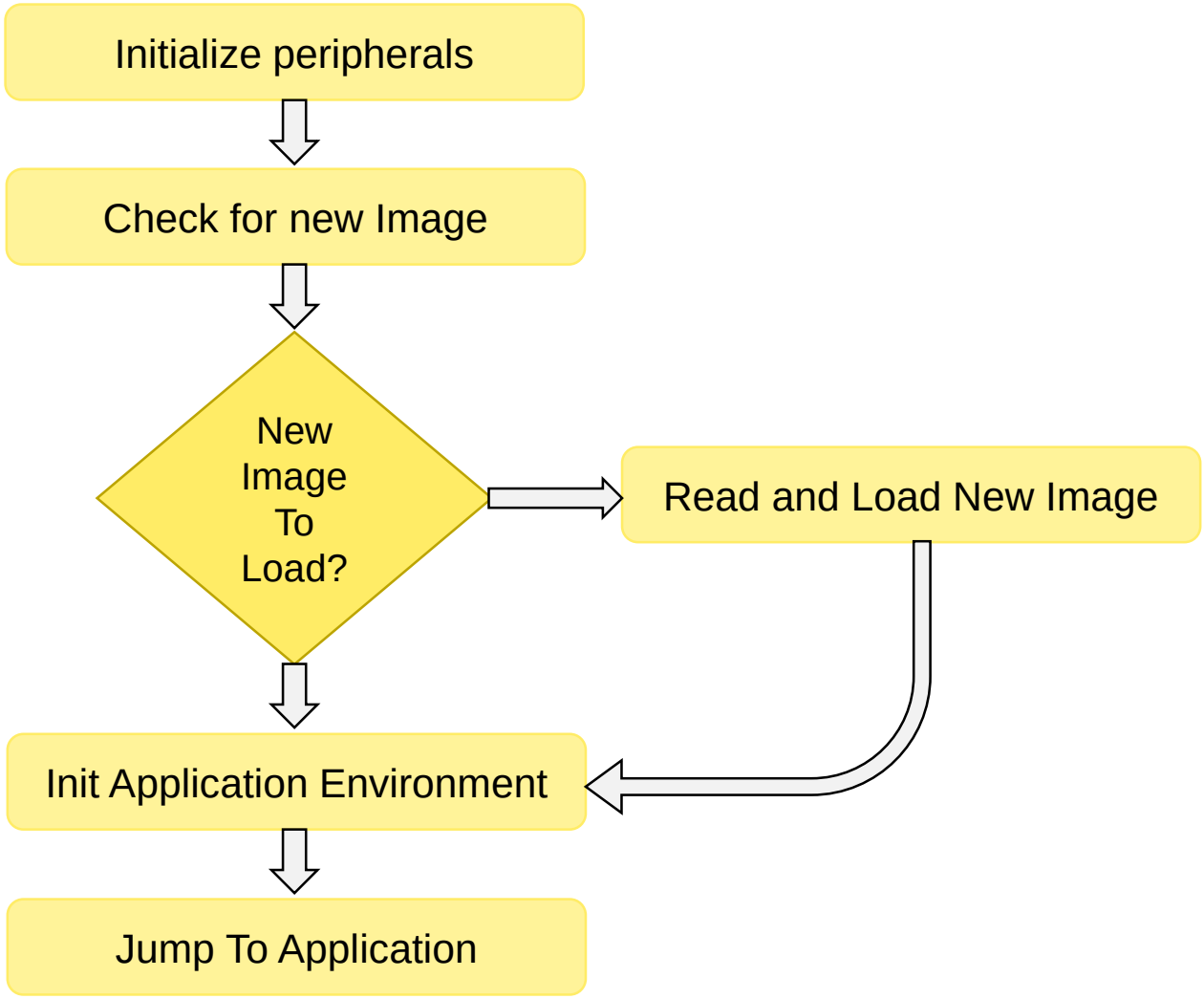
Program loader flow



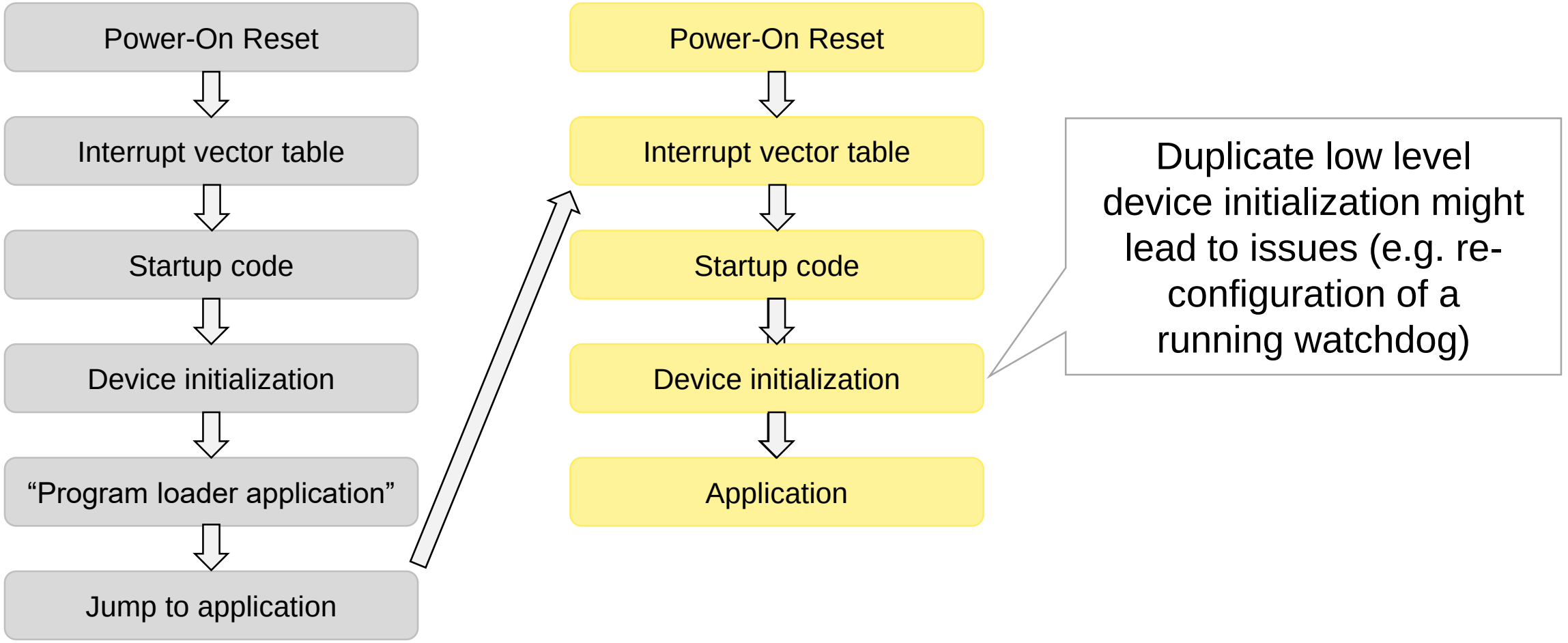
Possible use-cases for a program loader:

- Receive a new firmware package and perform an update of the application (e.g. OTA update)
- Switch between different applications based on device settings (e.g. country variants, …)
- …

Program loader application

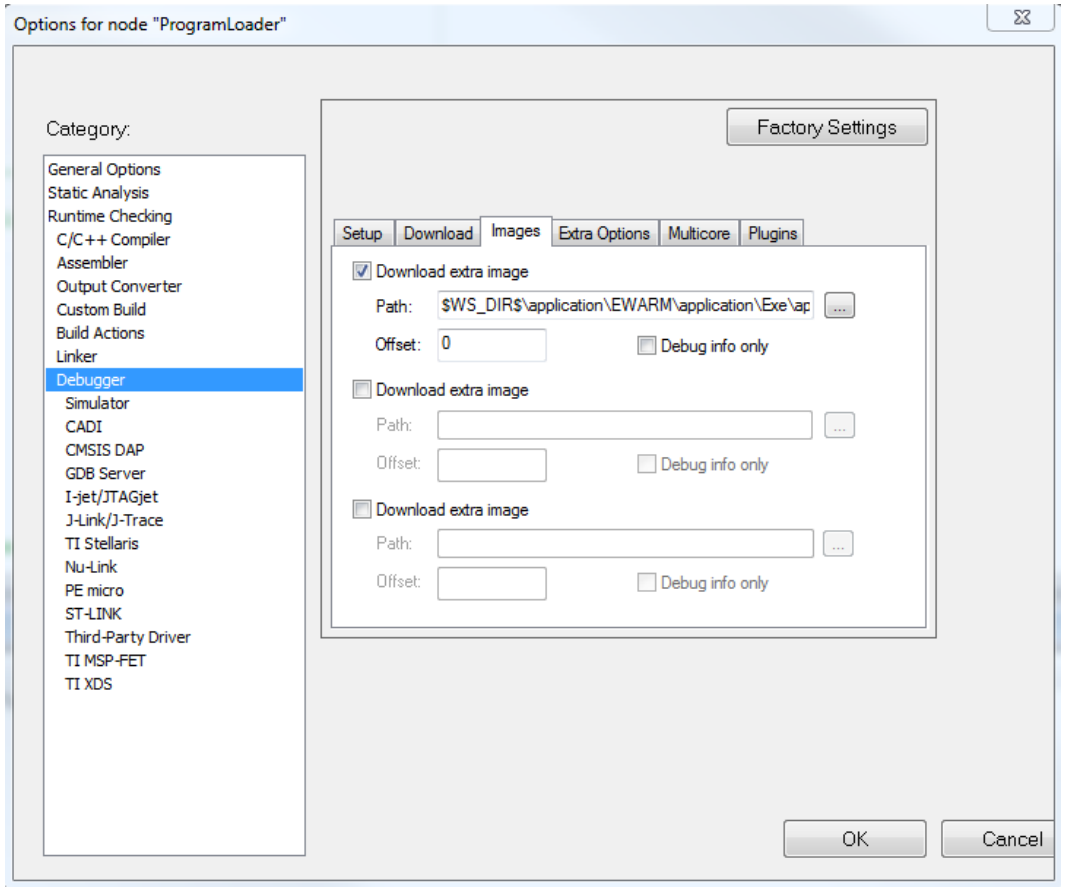


Program loader + Application flow



Debugging multiple image loading

- Debugger optionally loads multiple images
- Specify file location and optional offset
- Must be a recognized program image file
- Optionally only load debug information
  - If image already in Flash
  - Only want symbols for debug purposes



Things that user MUST manage

- Vector table location
  - Loader has its own Vector Table
    - Interrupts used for program loading etc.
    - Likely located at the “default” vector table location for this chip
  - Application must have its own vector table located in application Flash memory range
    - May use some of same interrupts and exceptions as loader
      - Needs its own handlers. Conditions may not be the same
      - SysTick, NMI, Hard Fault etc.
  - __iar_data_init3() must be called for proper RTL initialization

Loader setup to “see” application

```
43  /* USER CODE BEGIN Includes */
44  #include <intrinsics.h>
45
46  typedef void (application_t) (void);
47
48  typedef struct vector
49  {
50      uint32_t        stack_addr;     // intvec[0] is initial Stack Pointer
51      application_t   *func_p;        // intvec[1] is initial Program Counter
52  } vector_t;
53
54  extern const uint32_t app_vector;   // Application vector address symbol from
55                                      // the linker configuration file
56
57  /* USER CODE END Includes */
58
59  /* Private variables ---------------------------------------------------------*/
60
61  /* USER CODE BEGIN PV */
62  /* Private variables ---------------------------------------------------------*/
63  const vector_t *vector_p            = (vector_t*) &app_vector;
64  volatile uint32_t stack_arr[100]    = {0}; // Allocate some stack
65                                             // just to show that
66                                             // the SP should be reset
67                                             // before the jump - or the
68                                             // stack won't be configured
69                                             // correctly.
70
71  /* USER CODE END PV */
```

```
15
16  define memory mem with size = 4G;
17  define region ROM_region    = mem:[from __ICFEDIT_regic
18  define region RAM_region    = mem:[from __ICFEDIT_regic
19
20  define block CSTACK    with alignment = 8, size = __IC
21  define block HEAP      with alignment = 8, size = __IC
22
23  define exported symbol app_vector = 0x08008000;
24
```

Application Setup

- Application linker config file.
  - Edited in Linker configuration options in IDE for Vector table start address and Flash memory range
- Vector table set at start of application Flash Block
- Flash block set at 0x08008000 rather than 0x08000000

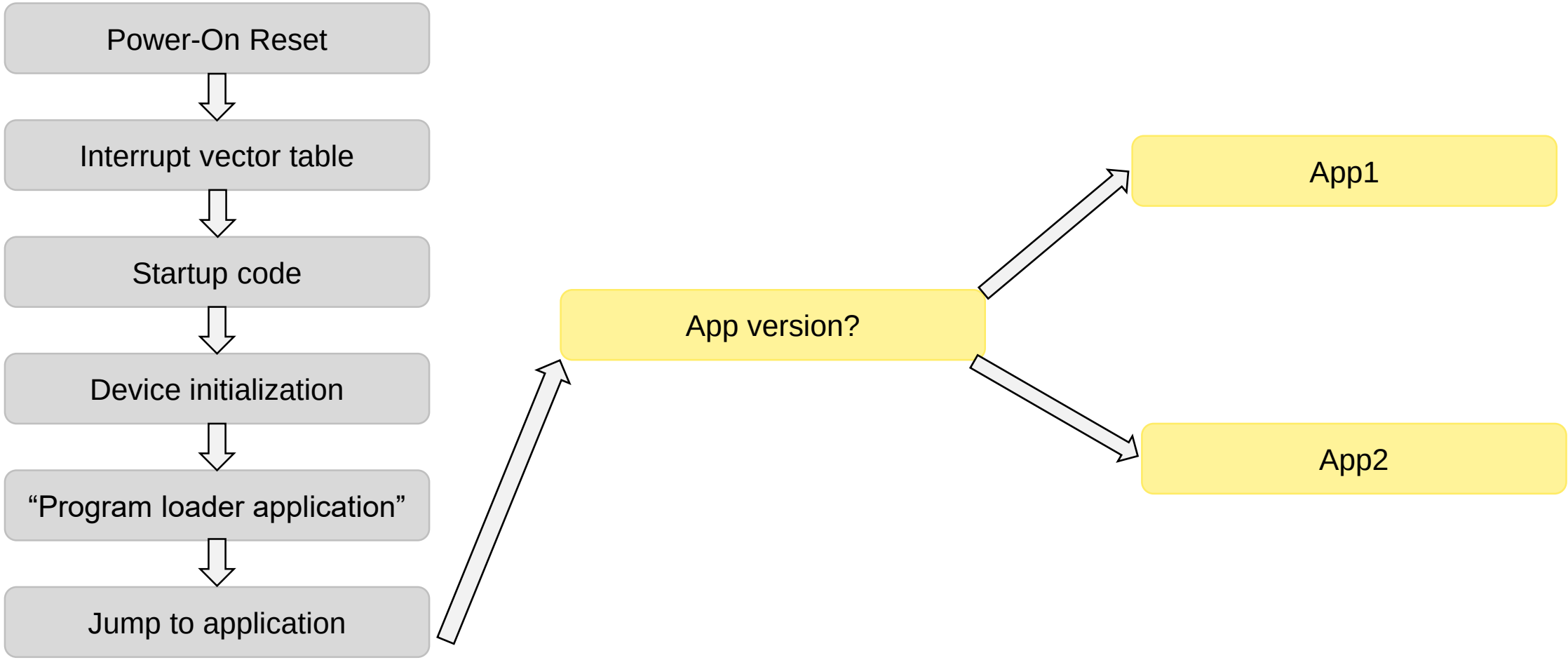
```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08008000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08008000;
define symbol __ICFEDIT_region_ROM_end__   = 0x081FFFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x2002FFFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x800;
define symbol __ICFEDIT_size_heap__   = 0x800;
/**** End of ICF editor section. ###ICF###*/

define memory mem with size = 4G;
```

```
1037
1038 /** @addtogroup Peripheral_memory_map
1039   * @{
1040   */
1041 #define FLASH_BASE            0x08008000U /*!< FLASH(up to 2 MB) base address in the alias
1042 #define CCMDATARAM_BASE       0x10000000U /*!< CCM(core coupled memory) data RAM(64 KB) ba
1043 #define SRAM1_BASE            0x20000000U /*!< SRAM1(112 KB) base address in the alias reg
```

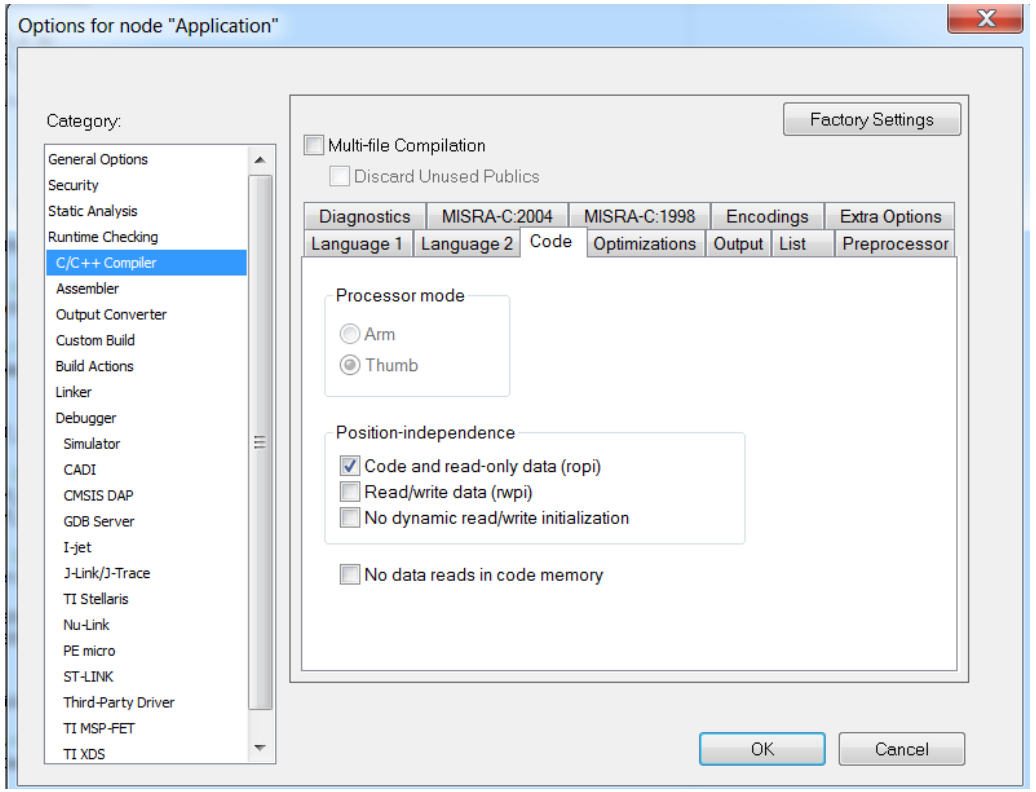
Challenges in this environment

What if we have multiple app versions?



What if we have multiple app versions?

- Instruct the compiler to generate position independent application binary.
- Build and install different versions of the same app?
- Bootloader can decide which one to jump to.



Demonstration

Summary

Summary

- Loading multiple project images for debugging can be a powerful tool
- Several factors must be understood and considered to make the environment work correctly
- Once configured correctly, it can be used over as a template for many projects

Thank you for your attention!

Questions? Reach out to fae@iar.com

More information is also available at iar.com