ENTER.
tech with a heartbeat

LOW-END EMBEDDED SYSTEMS FOR HIGH-END PRODUCTS

INSIGHTS FROM A REAL PRODUCTS DEVELOPMENT

CAIO OLIVEIRA

TODAY'S STORY

LOW-END EMBEDDED SYSTEMS AND CODE SIZE:

GETTING BLOOD OUT OF STONE

PART I

INTRODUCTION

BACKGROUND
OVER ME

Caio Souza Oliveira (30)
Brazilian

- **Profile**
  - Working for **Enter** since 2017;
  - Embedded Software Engineer;
  - Involved in several projects within **Philips** ever since;
- **Education**
  - Master in Electrical Engineering (UFMG-Brazil);
    Major: **Embedded System Design**
    (Heterogeneous Computing);
  - Bachelor in Computer Engineering (UNIFEI-Brazil);
- **General**
  - Sweet spot for **C/C++** and **Assembly**;
  - Working on homebrew for retro-video games (SNES/GB);
  - Deep interest
    - Quantum Computing
    - Compiler Theory;
    - Languages;
    - Astronomy;
    - Exercicing;

BACKGROUND
PROJECT'S OVERVIEW

- **Embedded Sofware Engineer in Philips Drachten**
  - Wi-Fi connected device, with rich user interface, mobile app and back-end connectivity;
  - Original firmware was developed by an external partner;
    - Not the best development practices;
    - Not maintainable in the long run;

- **Rewrite the Firmware!**
  - The firmware is meant to run on legacy hardware;
  - It must be written in C++ and fully tested (integration + unit);
  - Must be compatible with product's ecosystem (FW+CLOUD+APP);
  - Must be continuously updated over the span of several years;

BACKGROUND
HARDWARE OVERVIEW

- The hardware was designed in 2016 based on an external partner's specification. The system is quite complex, and it can be summarized as:
    - End-of-life **Cypress' 32-bit ARM Cortex-M0+** based microcontroller;
    - 40.5MHz MCU, **128KB internal Flash and 16KB RAM**;
    - **8MB External (*off-board*) Flash chip** for data storage;
    - Multiple connectivity and multi-media components not relevant for this presentation;

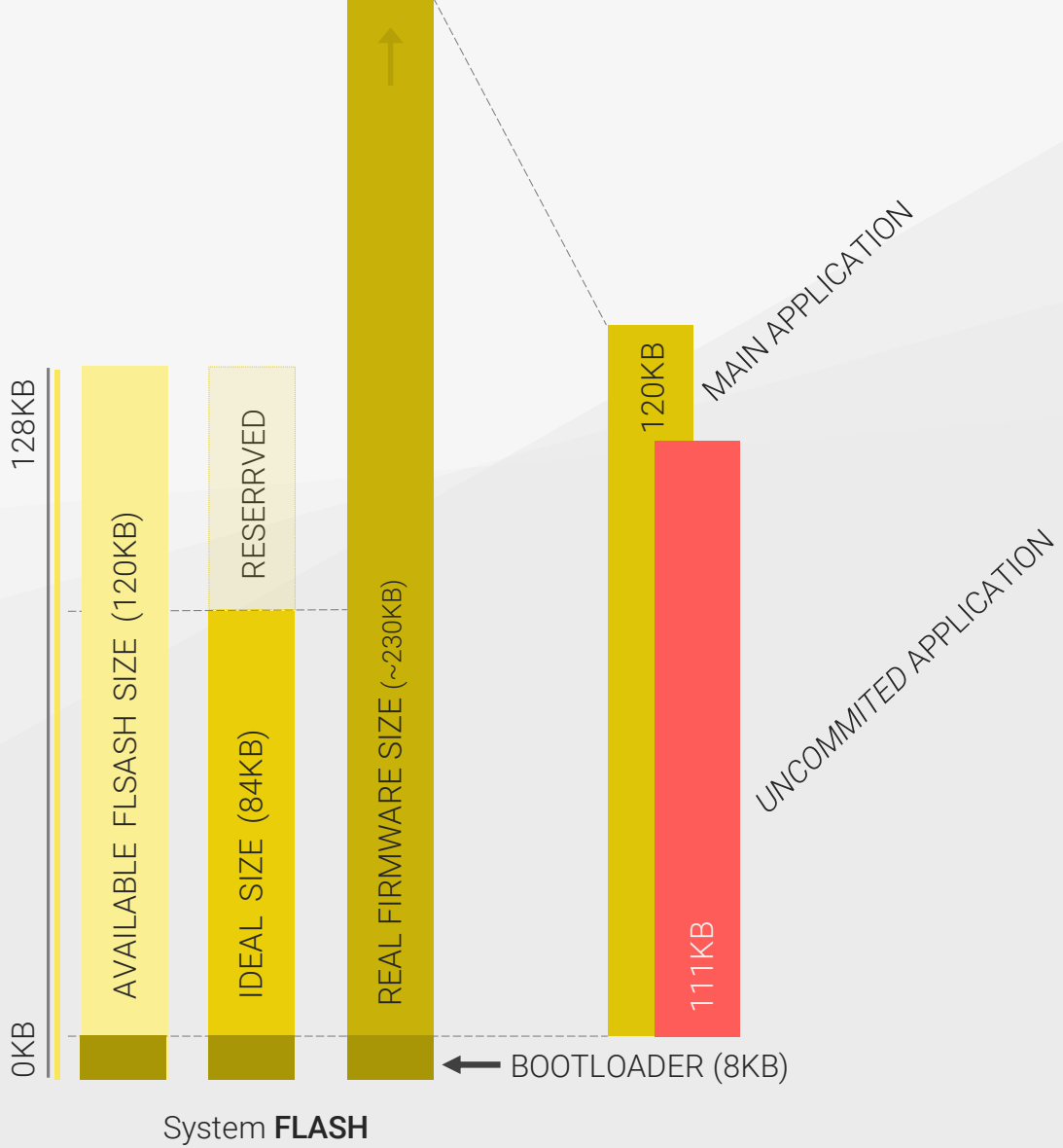
BACKGROUND
SOFTWARE OVERVIEW

- Bare Metal;
- Fully **layered architecture** following OOP paradigm;
- *Plug-and-Play* components architecture (for services and feature abstractions);
- Rich *template-based* UI rendering;
  - Almost **one hundred different** screens;
  - Also provides the concept of UI Controls/Views for *tighter* software development;
  - Event-driven user-input handling;
- REST-API host;
  - Provides **hundreds of properties and methods** to allow remote control;
  - Including *transactional* memory manager;
- More **than a dozen user-level features**;
- **Command Line Interface** for advanced automated testing;
- Services, Infrastructure, HAL layers;
- And lots more…

BACKGROUND
THE PROBLEM

- *It will **not fit** in 128 KB;*

BACKGROUND
THE PROBLEM



BACKGROUND
THE PROBLEM

BACKGROUND
THE PROBLEM

- Changing the hardware is not an option;
- We could not find a commercial/open source solution that would fit our needs;
  - Even IAR has been officially contacted but they had no appropriate solution;

BACKGROUND
THE SOLUTION

CODE BANKS
BANK SWITCHING

Formally:

*"Bank Switching (or code banking) is a technique where one can **increase the amount of usable memory** without directly changing addressable space reachable by the microprocessor" (Wikipedia, modified)*

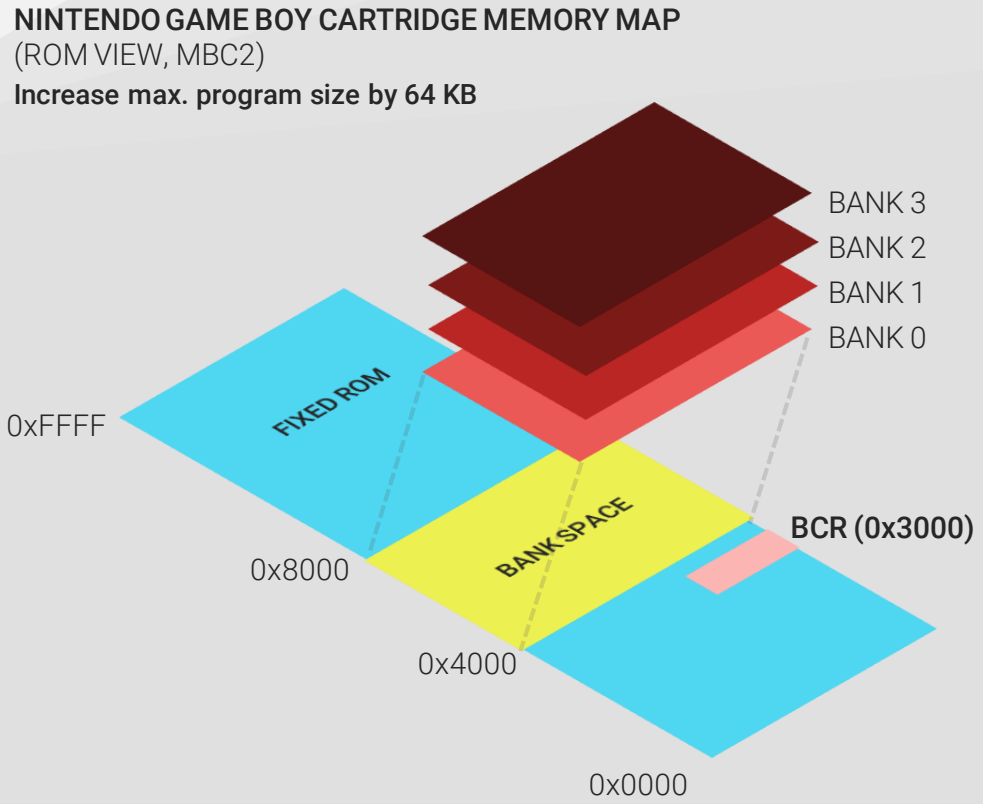
CODE BANKS

BANK SWITCHING

- **Idea**: use a single addressing space, and **dynamically switch code/data segments in-and-out** as needed.

- Very common technique in older computer, where addressing space would be from 10, 12 or 16-bits while the programs were much larger than what would fit in those address-ranges.

    - Used in old computer and video games (8080, Z80, 6502, 6809, etc.) using special hardware registers/instructions;
    - Switching *floppies* or changing CDs can also be considered a type of bank switching.



CODE BANKS
IMPLEMENTATION CHALLENGES

- **Cortex-M0+** has no hardware for bank switching. **Software is the only option!**
    - Expand CPU's addressing space…?
        - No, **increase the amount of usable code memory** seen by the CPU instead;
    - How dynamically swap code segments?
    - How to **properly build** code segments so that:
        - *Data sharing* among the multiple slices is allowed;
        - *Code sharing* is possible;
    - How to split the program into code banks?
    - Which programming model should be used to allow code banks to be used along with any C++ program?
    - How good would this solution perform?

CODE BANKS

IMPLEMENTING THE CONCEPT **IN SOFTWARE**

- What did I need:
  1. **Readily available** bank's storage location;
  2. ***Non-degradable*, exactable** memory resource;
  3. **A software architecture** to abstract (and somehow *hide*) the banking concept;
     - In other words: banks must be *invisible* from the programmer's and CPU perspective
  4. **Very low latency** switching and execution

CODE BANKS

IMPLEMENTING THE CONCEPT **IN SOFTWARE**

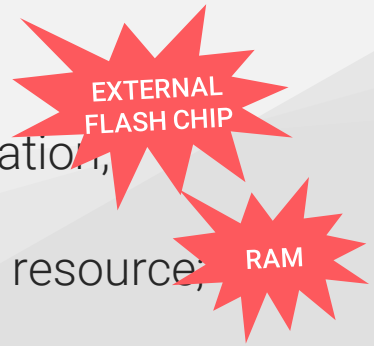
- What did I need:
  1. **Readily available** bank's storage location;

  2. ***Non-degradable*, exactable** memory resource;
  3. **A software architecture** to abstract (and somehow *hide*) the banking concept;
     - In other words: banks must be *invisible* from the programmer's and CPU perspective
  4. **Very low latency** switching and execution

CODE BANKS

IMPLEMENTING THE CONCEPT **IN SOFTWARE**

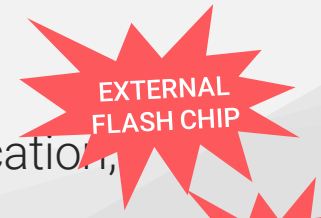
- What did I need:
  1. **Readily available** bank's storage location;
  2. ***Non-degradable*, exactable** memory resource;
  3. **A software architecture** to abstract (and somehow *hide*) the banking concept;
     - In other words: banks must be *invisible* from the programmer's and CPU perspective
  4. **Very low latency** switching and execution



CODE BANKS

IMPLEMENTING THE CONCEPT **IN SOFTWARE**

- What did I need:
  1. **Readily available** bank's storage location,
  2. ***Non-degradable*, exactable** memory resource;
  3. **A software architecture** to abstract (and somehow *hide*) the banking concept
     - In other words: banks must be *invisible* from the programmer's and CPU pe
  4. **Very low latency** switching and execution



CODE BANKS

IMPLEMENTING THE CONCEPT **IN SOFTWARE**

- What did I need:
  1. **Readily available** bank's storage location,
  2. ***Non-degradable*, exactable** memory resource,
  3. **A software architecture** to abstract (and somehow *hide*) the banking concept
     - In other words: banks must be *invisible* from the programmer's and CPU pe
  4. **Very low latency** switching and exe

EXTERNAL FLASH CHIP

RAM

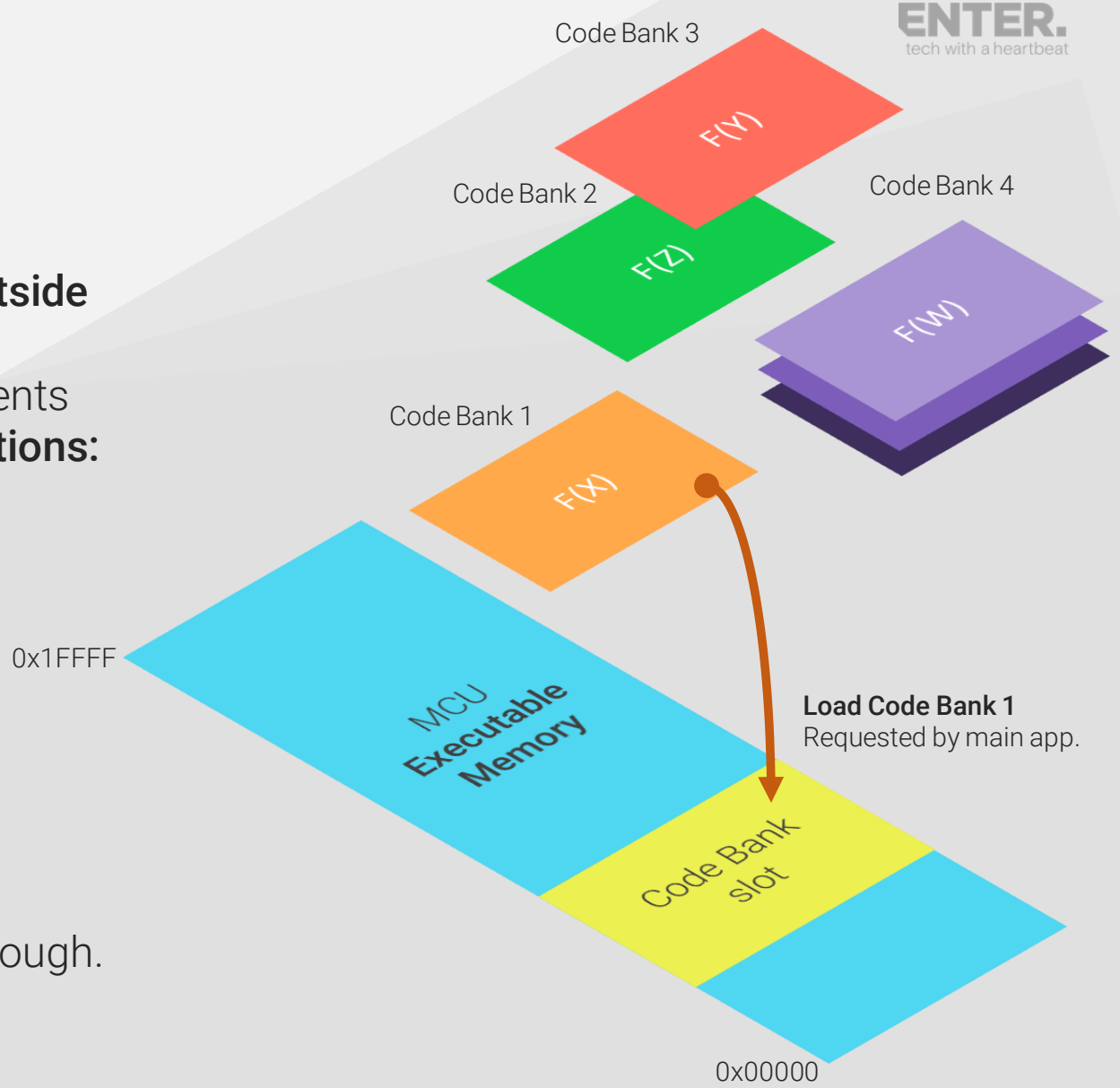
CUSTOM LIBRARY AND BUILD TOOLS

SIMPLE ARCH. & VIRTUAL BANKS

CODE BANKS

SIMULATING THE BEHAVIOR OF CODE BANKS

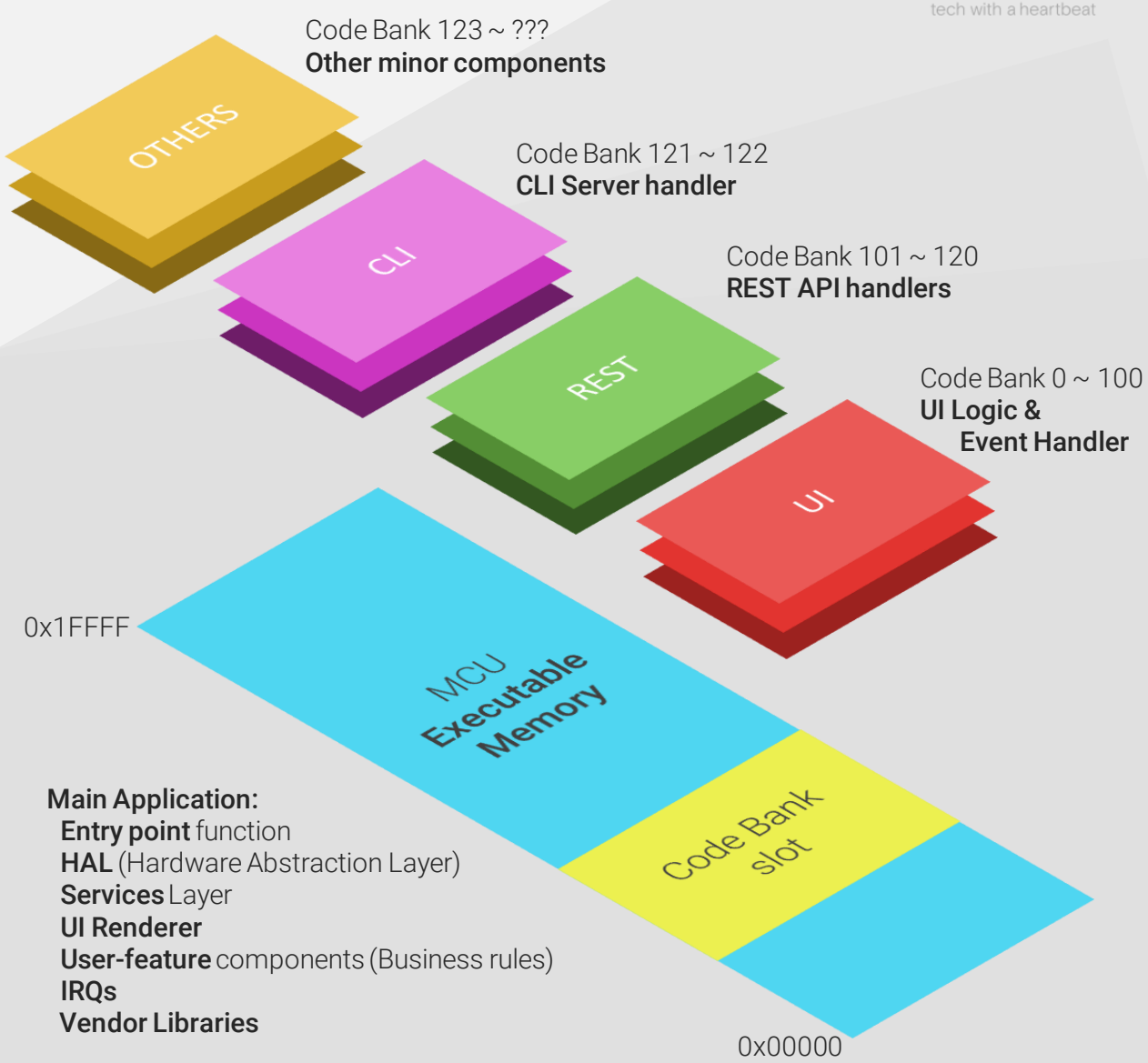
- How would it work?
  1. Code Segments (*AKA code banks*) are **stored outside of the microcontroller's** Flash;
  2. Upon **request from the application**, those segments would be **loaded and executed as ordinary functions:**
     a) It should be possible to **pass arguments** to it;
     b) It should be possible get **return values** from it;
  3. When the code is no longer needed, other code segments **could take its place**;
- This would effectively allow the application to be considerably bigger!
  - The application should be designed differently, though.



CODE BANKS

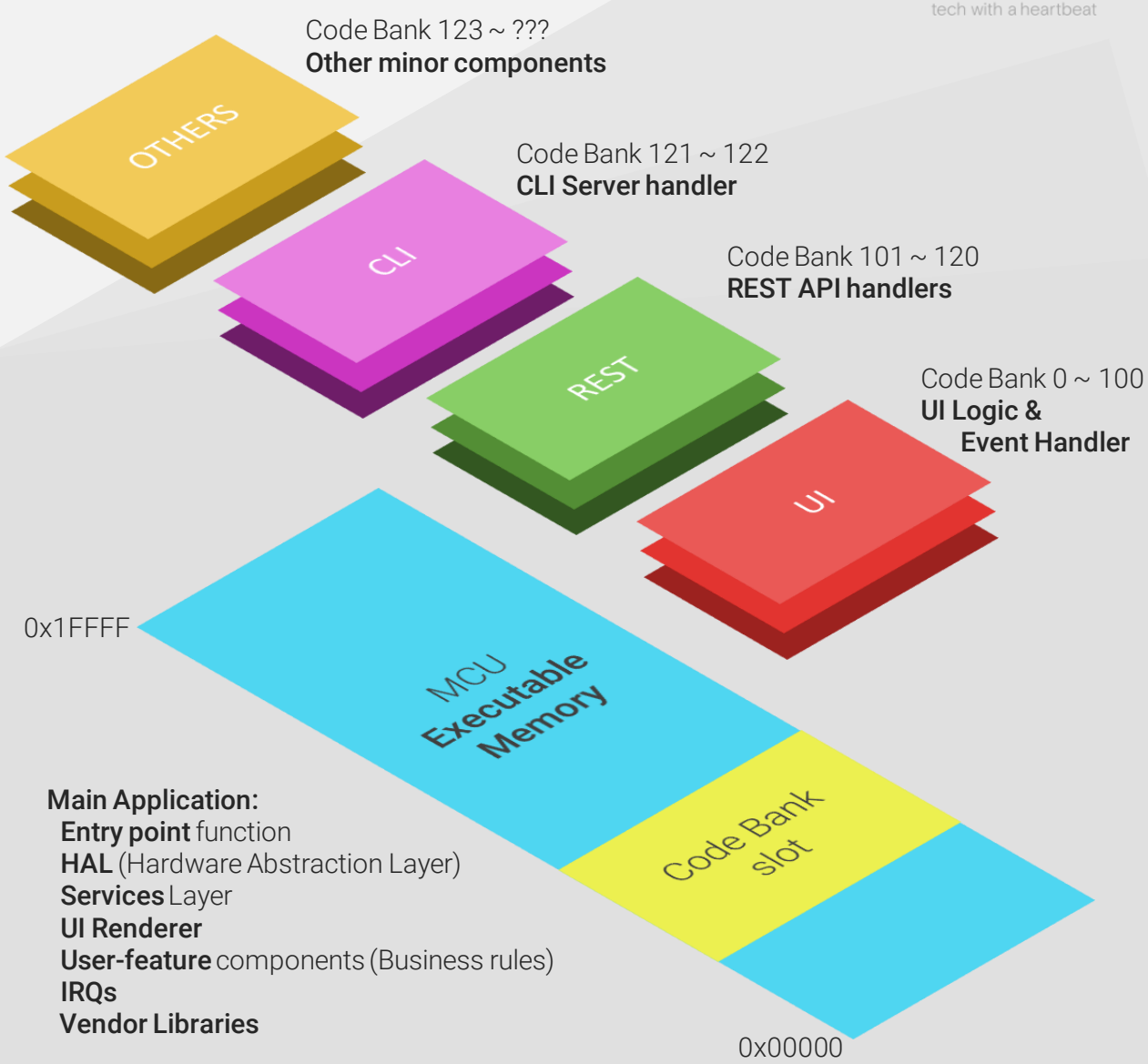
SOFTWARE PARTITION

- To simplify development (and debugging) the software should be split into *two* groups: ***bankable*/*non-bankable***;
- General functionality (HAL, Services, User Features) was considered ***non-bankable*** would remain in the CPU's internal Flash;
- Those functionalities are **constantly used throughout the entire code**, and if set in a code bank, it would require continuous bank switching (possibly degrading performance);



CODE BANKS
SOFTWARE PARTITION

- To simplify development (and debugging) the software should be split into *two* groups: ***bankable*/*non-bankable***;
- Most of the *stateless* logic has been considered ***bankable***;
  - Little to no data sharing;
- UI Logic, REST API and CLI are **predictable and context-sensitive components** that only need to be executed based on user/system generated events – good candidate to be ***banked***;



CODE BANKS

IMPLEMENTING THE CONCEPT IN SOFTWARE

- **SPOILERS**: It works!

- On the bright side:
  - It provides maximum program size of **2,5MB (up to 256 different banks)**;
  - It is *mostly* compatible with the existing code base, requiring minor adjustments;
  - It is performant enough to be indistinguishable from code running directly from the microcontroller's Flash;

- However,
  - It took about 1.5 months of intensive work to be fully implemented and integrated;
  - It requires understanding of low-level programming and Arm assembly (Thumb-2 to be more specific);
  - It reduces the amount of usable RAM in the system;

CODE BANKS

SYSTEM COMPONENTS OVERVIEW

Development time
(programming and building)

Annotated source code

Pre-build

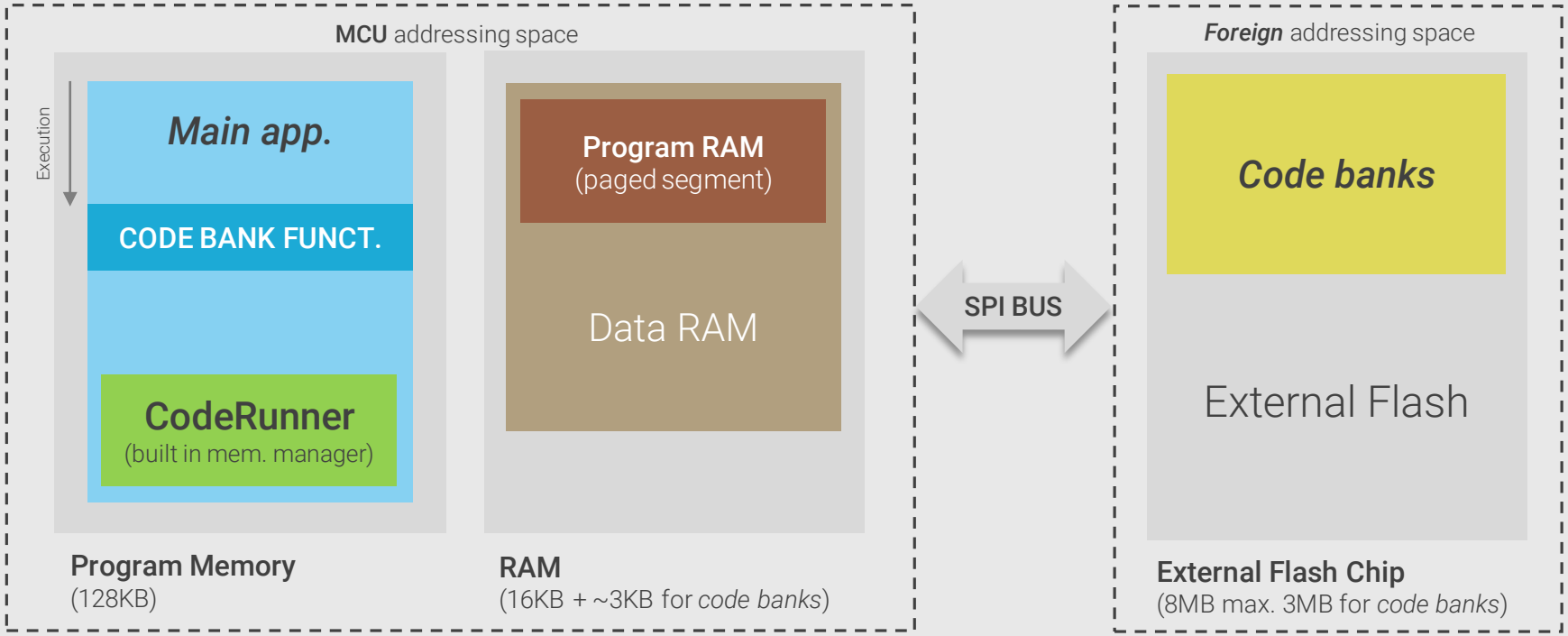
Build

Post-build

Code banks Image

Main app. Image

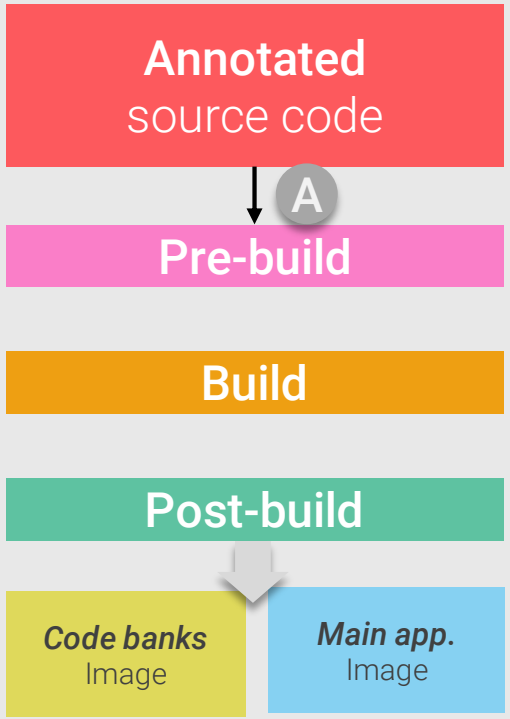
Runtime
(code banks execution flow)



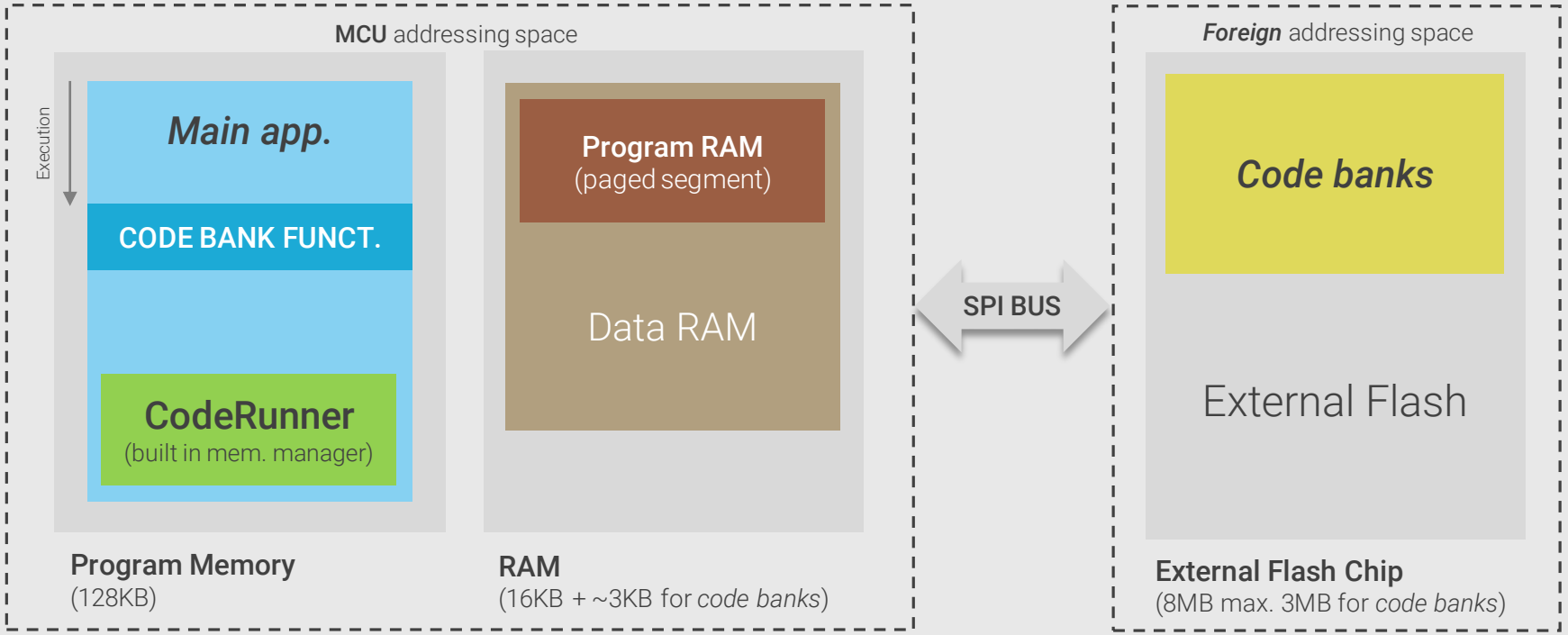
CODE BANKS

SYSTEM COMPONENTS OVERVIEW

Development time
(programming and building)



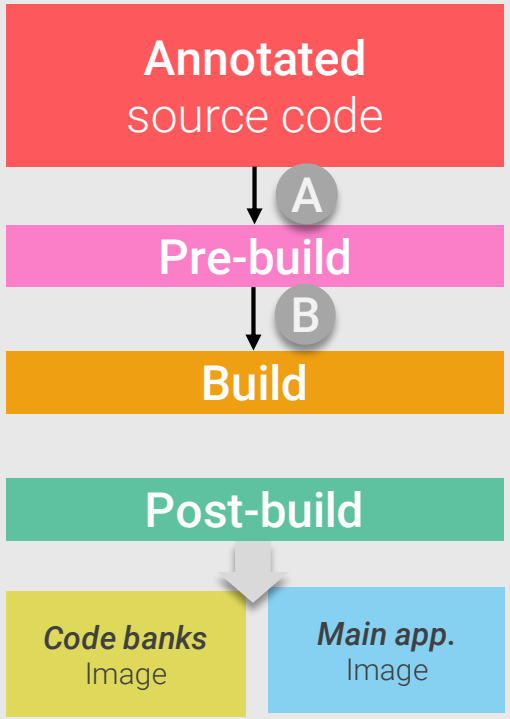
Runtime
(code banks execution flow)



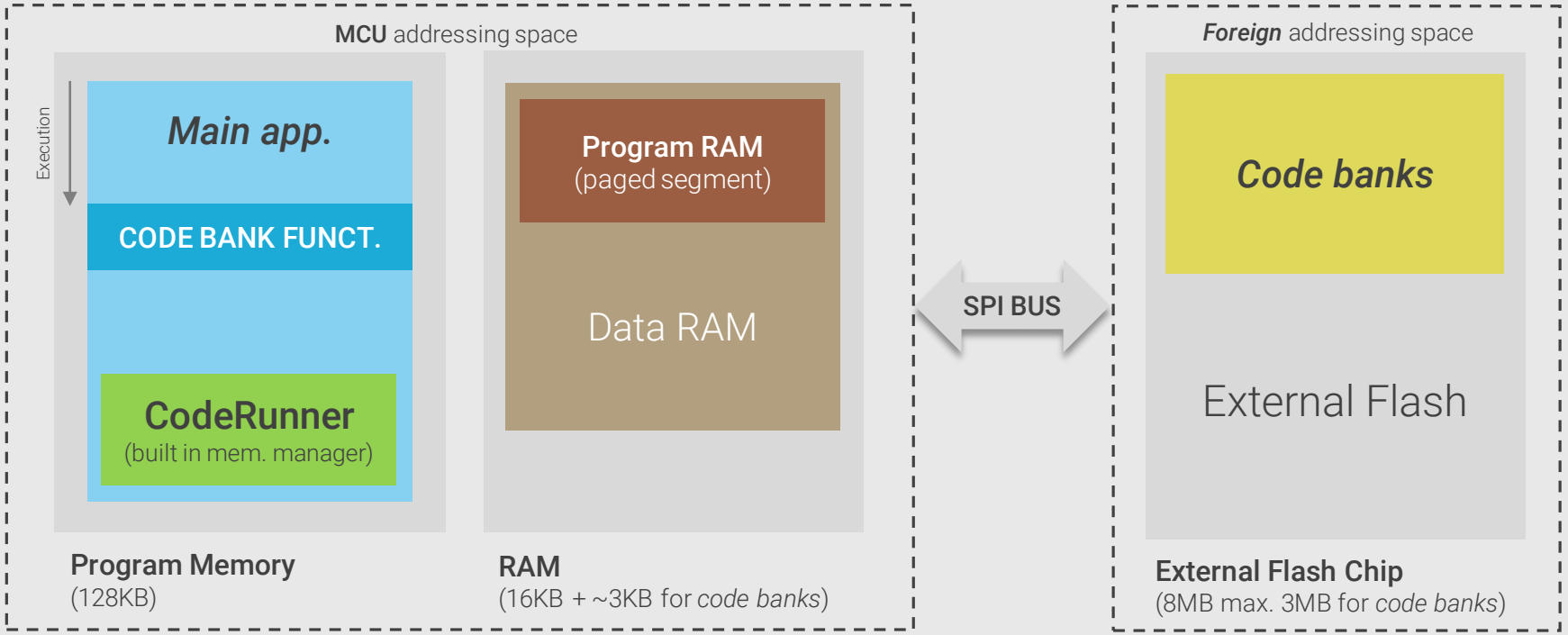
CODE BANKS

SYSTEM COMPONENTS OVERVIEW

Development time
(programming and building)



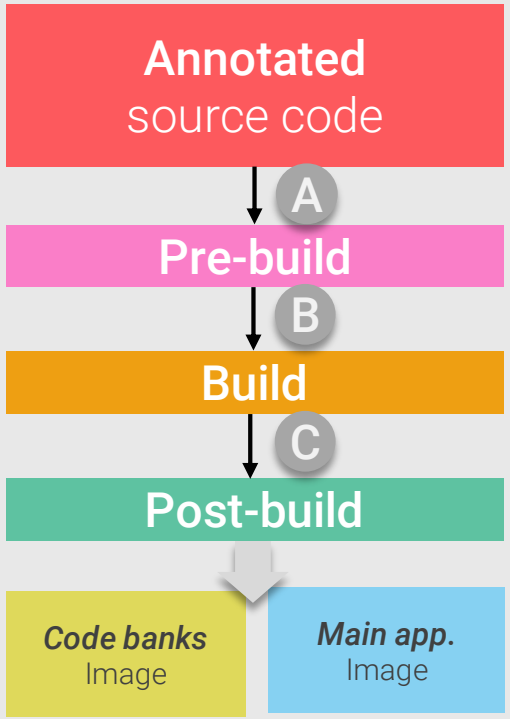
Runtime
(code banks execution flow)



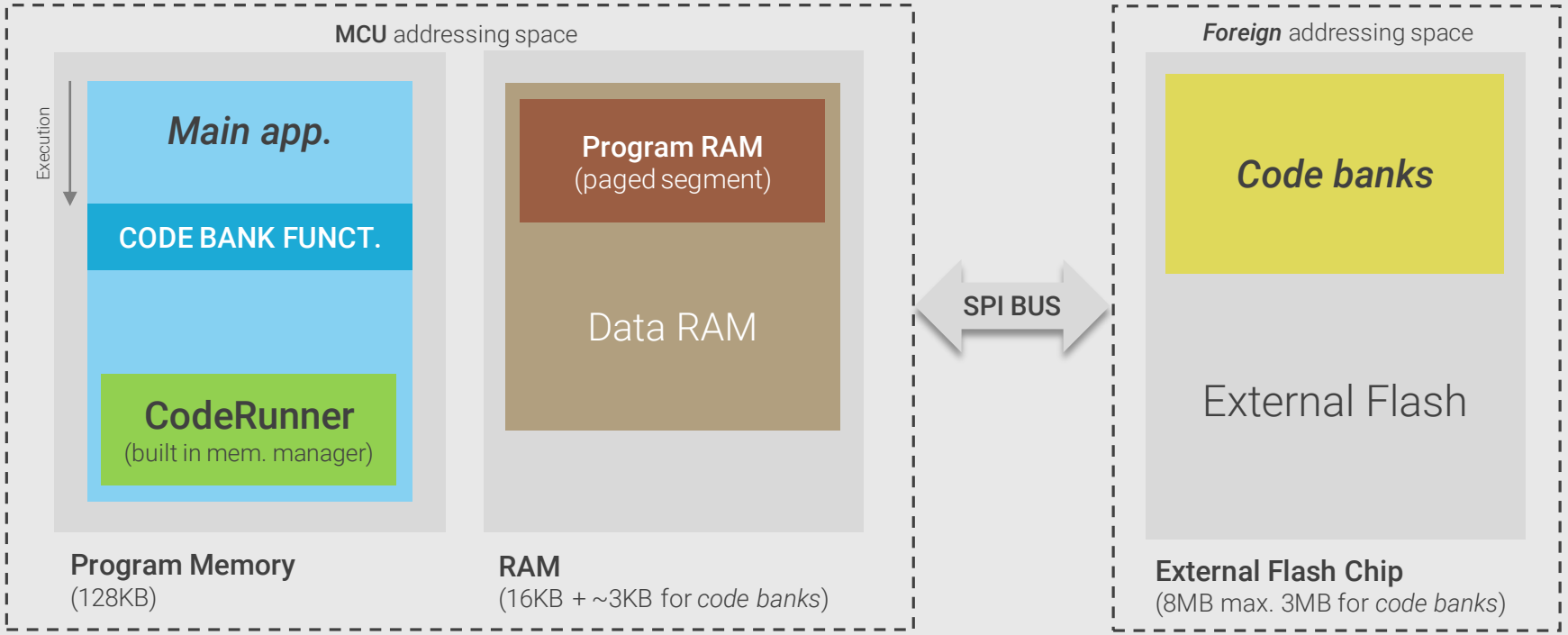
CODE BANKS

SYSTEM COMPONENTS OVERVIEW

Development time
(programming and building)

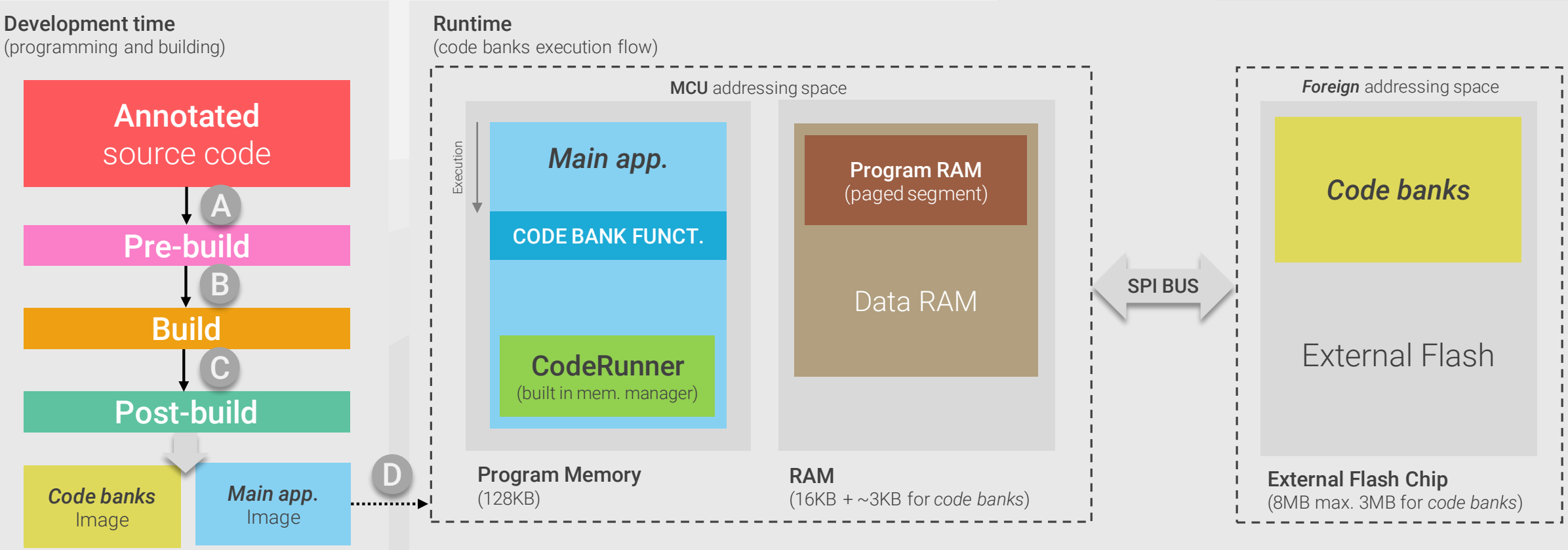


Runtime
(code banks execution flow)



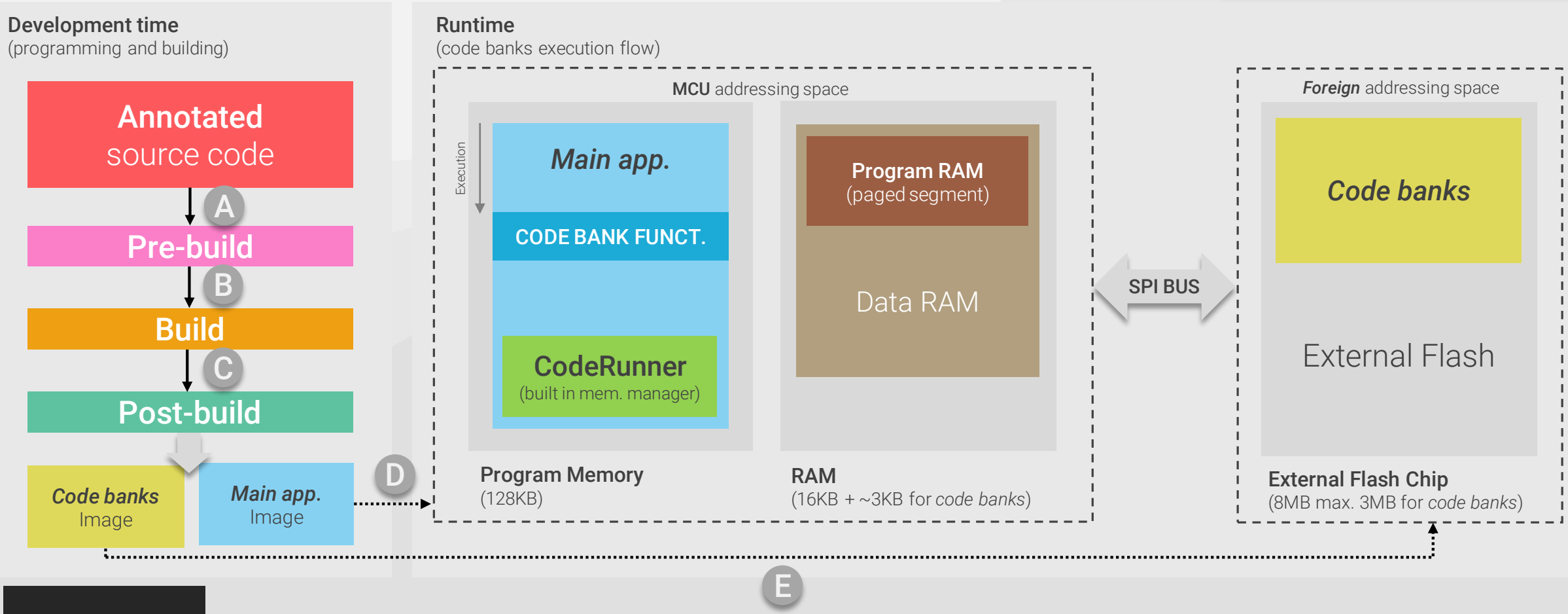
CODE BANKS

SYSTEM COMPONENTS OVERVIEW



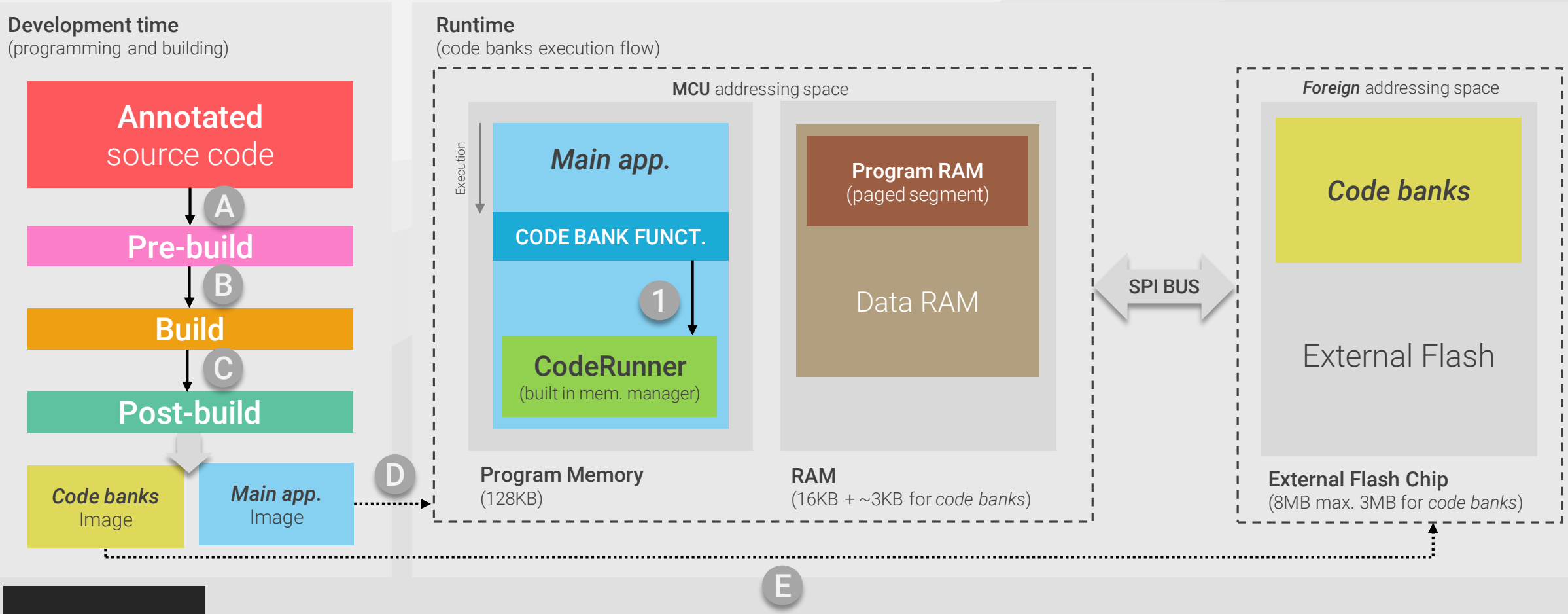
CODE BANKS

SYSTEM COMPONENTS OVERVIEW



CODE BANKS

SYSTEM COMPONENTS OVERVIEW

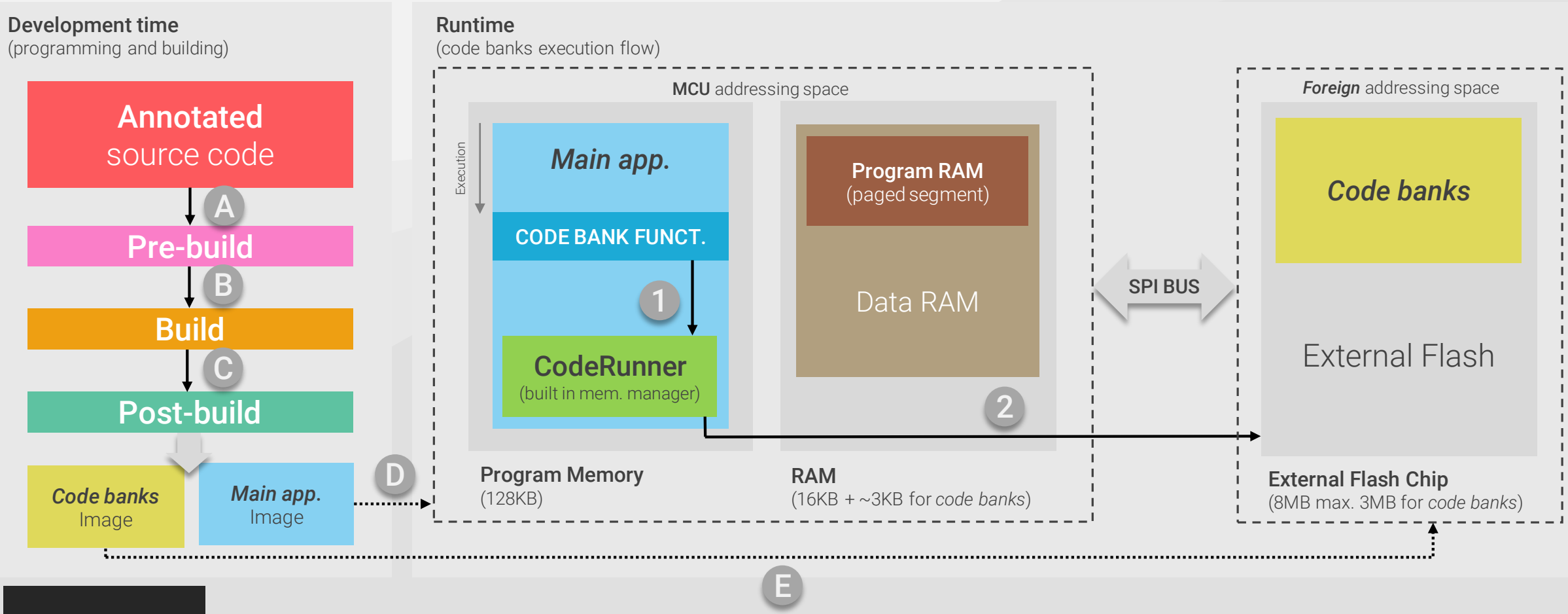


$ {040}_

040coders.nl

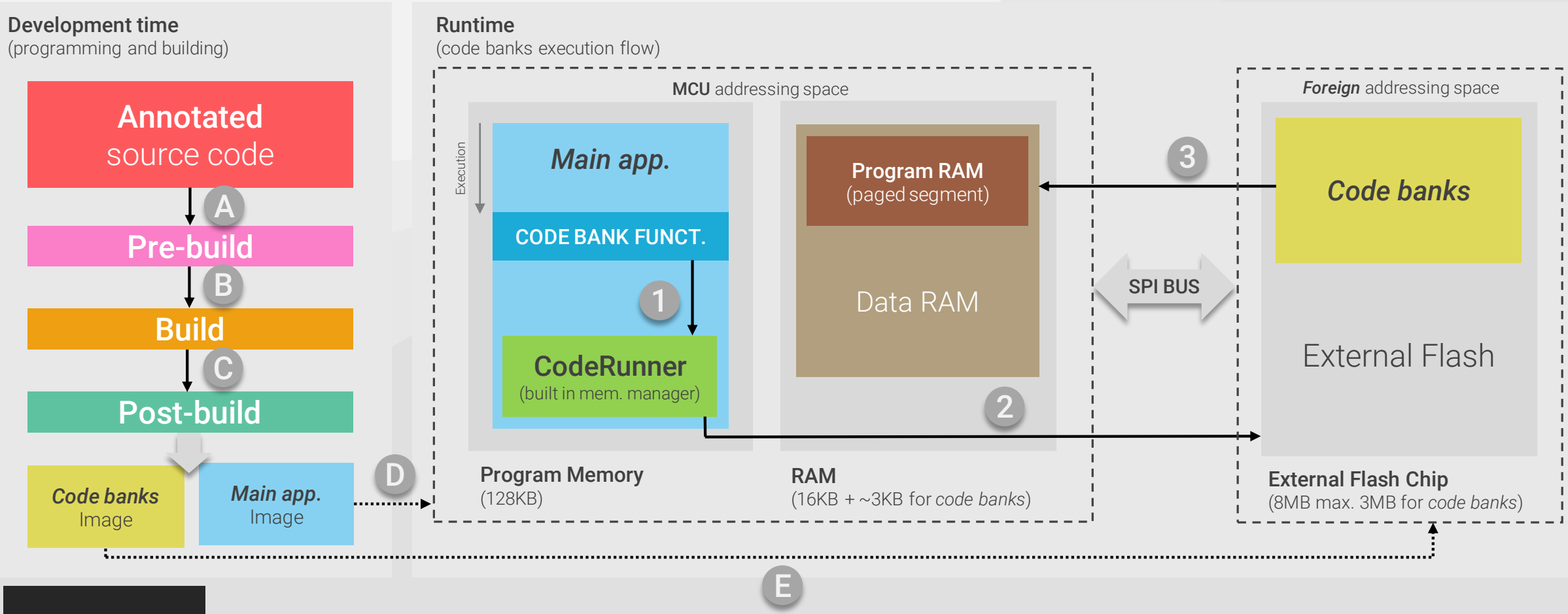
CODE BANKS

SYSTEM COMPONENTS OVERVIEW



CODE BANKS

SYSTEM COMPONENTS OVERVIEW

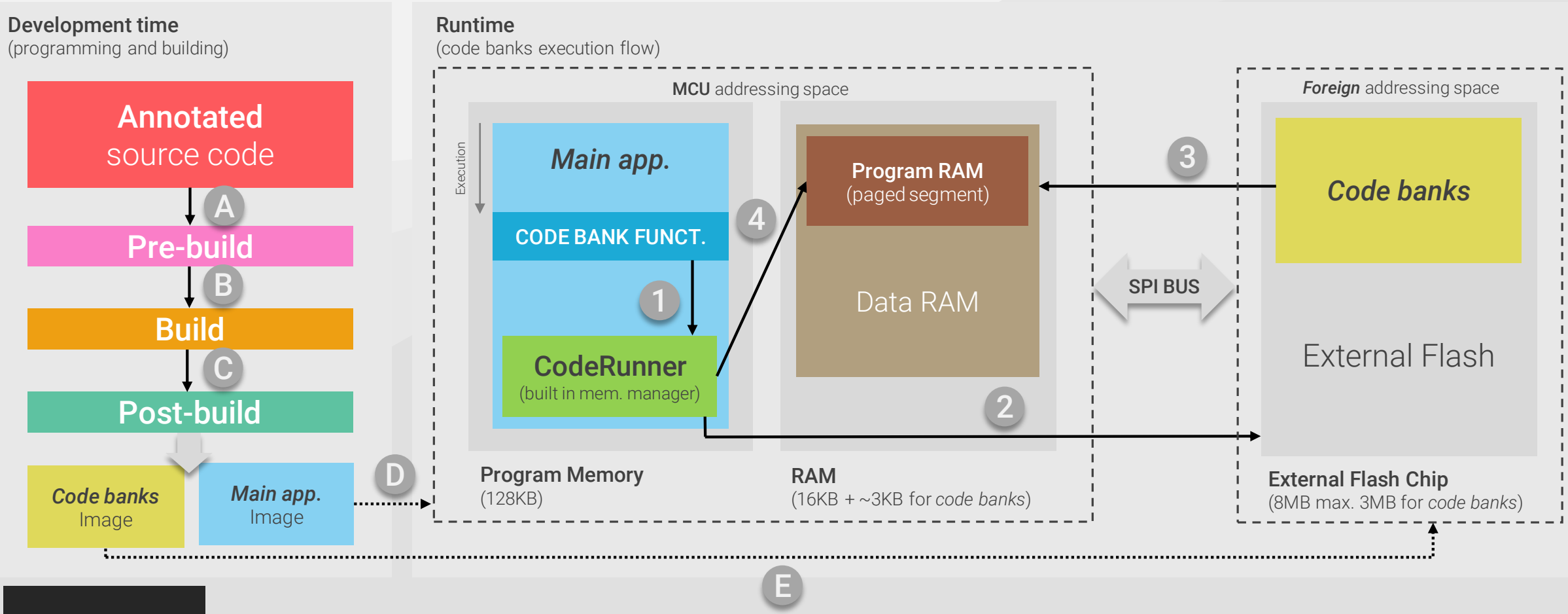


$ {040}_

040coders.nl

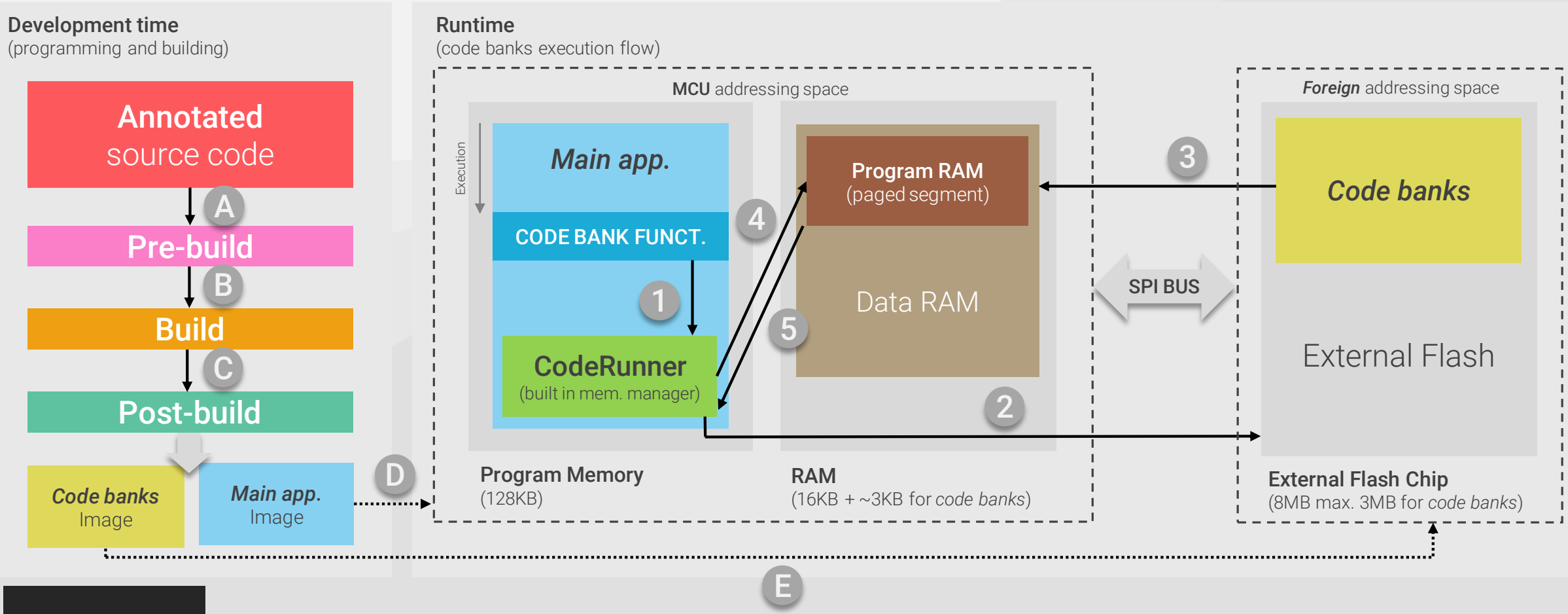
CODE BANKS

SYSTEM COMPONENTS OVERVIEW



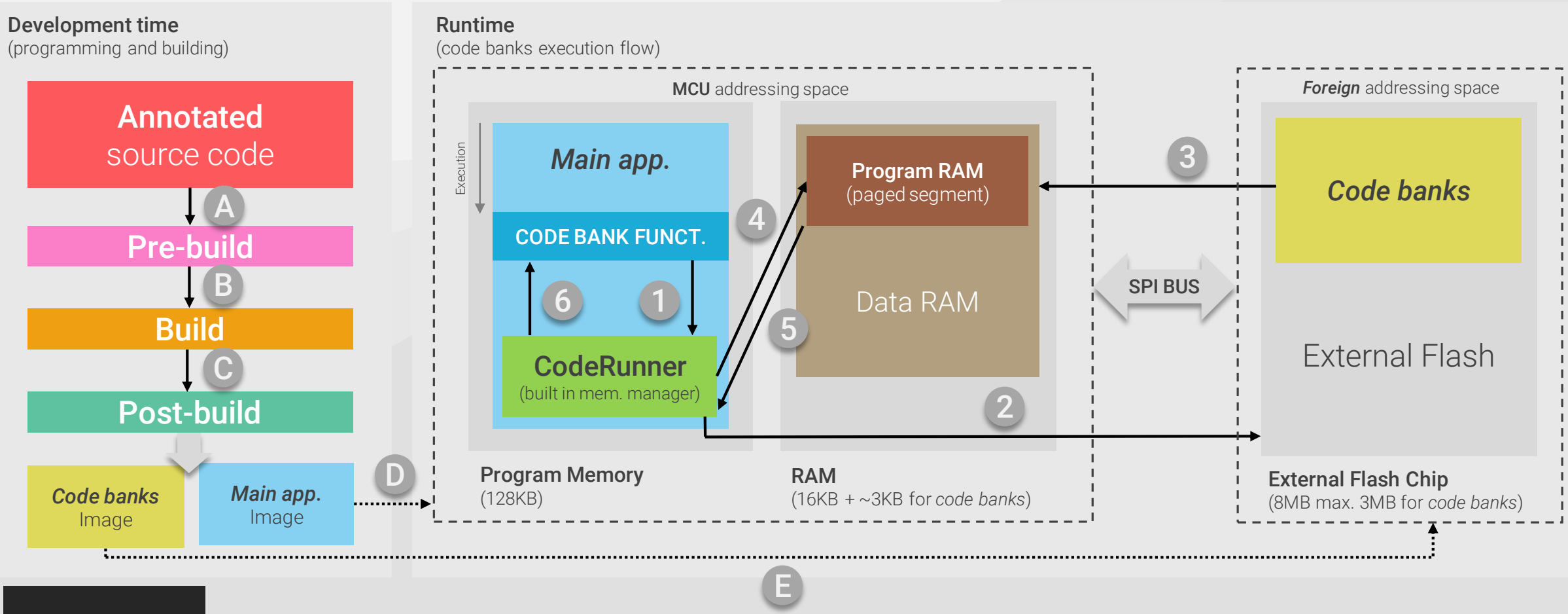
CODE BANKS

SYSTEM COMPONENTS OVERVIEW



CODE BANKS

SYSTEM COMPONENTS OVERVIEW



CODE BANKS
IN SUMMARY

Code banks **are**...

- A way to **allow bigger program sizes** in restricted embedded environment.
  It allow **cost reduction** by being compatible with lower-end embedded platforms.
- Meant to provide a flexible framework to **store code in any storage medium** available in the embedded system.
  Not in the CPU addressing space.
- Designed to be **easily integrated** in existing code base.
  There is **no significant difference in development times** when using *code banks*.

Code banks **are not**...

- NOT
  Platform agnostic
  currently compatible with ARM on IAR toolchain.
- NOT
  Meant for **low-power applications**.
  Code banks make **intensive use of external memory access** (typically through SPI) which might be prohibitive for certain applications;
- NOT
  Open source
  The code is currently not disclosed.

PART II

HANDS ON

This section covers:

WRITE

BUILD

RUN

PART II.A

WRITING CODE BANKS

```
#ifndef BANK_HPP_
#define BANK_HPP_

#include <cstdint>
#include <limits>
#include <ctime>
// *REQUIRED* custom macros are defined here
#include "BankedCodeUtils.hpp"


//Declare code bank and places it at a speci
namespace Banks
{
CODE_BANK Bank : public SchedulableBank
{
   // From: SchedulableBank class
   EXPOSED: virtual void OnLoaded()    overr
   EXPOSED: virtual void OnStarted()   overr
   EXPOSED: virtual void OnScheduled() overr
   EXPOSED: virtual void OnFinished()  overr

   // Bank methods
   EXPOSED: void SetInternalValue(uint8_t va
   EXPOSED: uint8_t GetInternalValue();

   // Internal method
   RESTRICTED: uint8_t InternalCalculation()

    private:
   // Member attribute (always private from
   uint8_t _internalValue;
};

#endif // BANK_HPP_
} // namespace Banks
```

WRITING CODE BANKS

WHAT DO YOU NEED?

- Code banks are simple C++ Classes
  - A class that is marked to be banked is said to have a ***banked-type*** .
  - Two differences:
    - After building, a bank is surrounded by a **metadata block** (relevant for the `CodeRunner`);
    - There is no concept of an **instance of a code bank**;

- New keywords and operators were designed to extend the C++ Language:
  - These keywords and operators should have **no impact in the general compilation/linking steps** and are mostly used during **pre- and post-build.**

WRITING CODE BANKS

WHAT DO YOU NEED?

- Definition keywords and usage operators:

Type	Keyw./Oprt.	C++ Equivalence	Example
Definition	`CODE_BANK`	`class`	`CODE_BANK MyBank`
	`EXPOSED`	`public`	`EXPOSED: void MyMethod();`
	`RESTRICTED`	`private`	`RESTRICTED: void Secret();`
Usage	`of <BankedType>`	`[sizeof(MyBank)]`	`BankContext context of MyBank;`
	`@`	Expands to: `CodeRunner.RunXXXArgYYYRet`	`@MyBank.MyMethod(context);`

WRITING CODE BANKS

WHAT DO YOU NEED?

Bank.hpp

```
#ifndef BANK_HPP_
#define BANK_HPP_

#include <cstdint>
// *REQUIRED* custom macros are defined here.
#include "BankedCodeUtils.hpp"

//Declare code bank and places it at a specific address.
CODE_BANK Bank : public SchedulableBank
{
      // From: SchedulableBank class
      EXPOSED: virtual void OnLoaded()    override;
      EXPOSED: virtual void OnStarted()   override;
      EXPOSED: virtual void OnScheduled() override;
      EXPOSED: virtual void OnFinished()  override;

      // Bank methods
      EXPOSED: void SetInternalValue(uint8_t value);
      EXPOSED: uint8_t GetInternalValue();

      // Internal method
      RESTRICTED: uint8_t InternalCalculation();

    private:
      // Member attribute (always private from banks persp.).
      uint8_t _internalValue;
};

#endif // BANK_HPP_
```

Bank.cpp

```
#include "Bank.hpp“

// Implementation comes here.
void Bank::OnLoaded()
{...}
void Bank::OnStarted()
{...}
void Bank::OnScheduled()
{...}
void Bank::OnFinished()
{...}
void Bank::SetInternalValue(uint8_t value)
{...}
uint8_t Bank::GetInternalValue()
{...}
uint8_t Bank::InternalCalculation()
{...}
```

Somewhere.cpp

```
// *REQUIRED* to run any method from a code bank.
#include "BankDefinitions.hpp"

// Implementation comes here.
void DoSomethingWithBanks()
{
      BankContext context of Bank;
      @Bank.SetInternalValue(context, 0x10);
}
```

$ {040}_

WRITING CODE BANKS

WHAT DO YOU NEED TO KNOW?

- *Banked-type* members can be `public`, `private` or `protected`
  - Access modifiers only apply in the context of inheritance, though;
  - Member types (`struct` and `class`) are also allowed;

- `virtual` and `override` are fully allowed.
  - **Be very careful with them, though!**

- `static` attributes are **not recommended**
  - That's due to the volatile nature of the bank's context
    - Could be added in a future implementation.
  - Specifiers such as `const`, `mutable`, `extern`, `volatile`, etc. are allowed.

- Use of `this` is allowed, although with some caveats.

WRITING CODE BANKS

CODE BANKS CONTEXT

- A *banked-type* is in essence a class type:
  - For classes, the C++ compiler allocate and generate code to manage a valid (and `const`) reference to the object instance's context (seen as `this` pointer).
    - A reference to that context is automatically passed as the first function argument for each member method call.
  - For *banked-types*, `this` is a **mutable context reference** that must be explicitly provided during an `EXPOSED` method call.
    - *Banked-types* do **not manage its context**, and have **no responsibility over its life-cycle**: the **caller is responsible** to allocate, handle and destroy any context data used across call.

PART II.B

BUILDING CODE BANKS

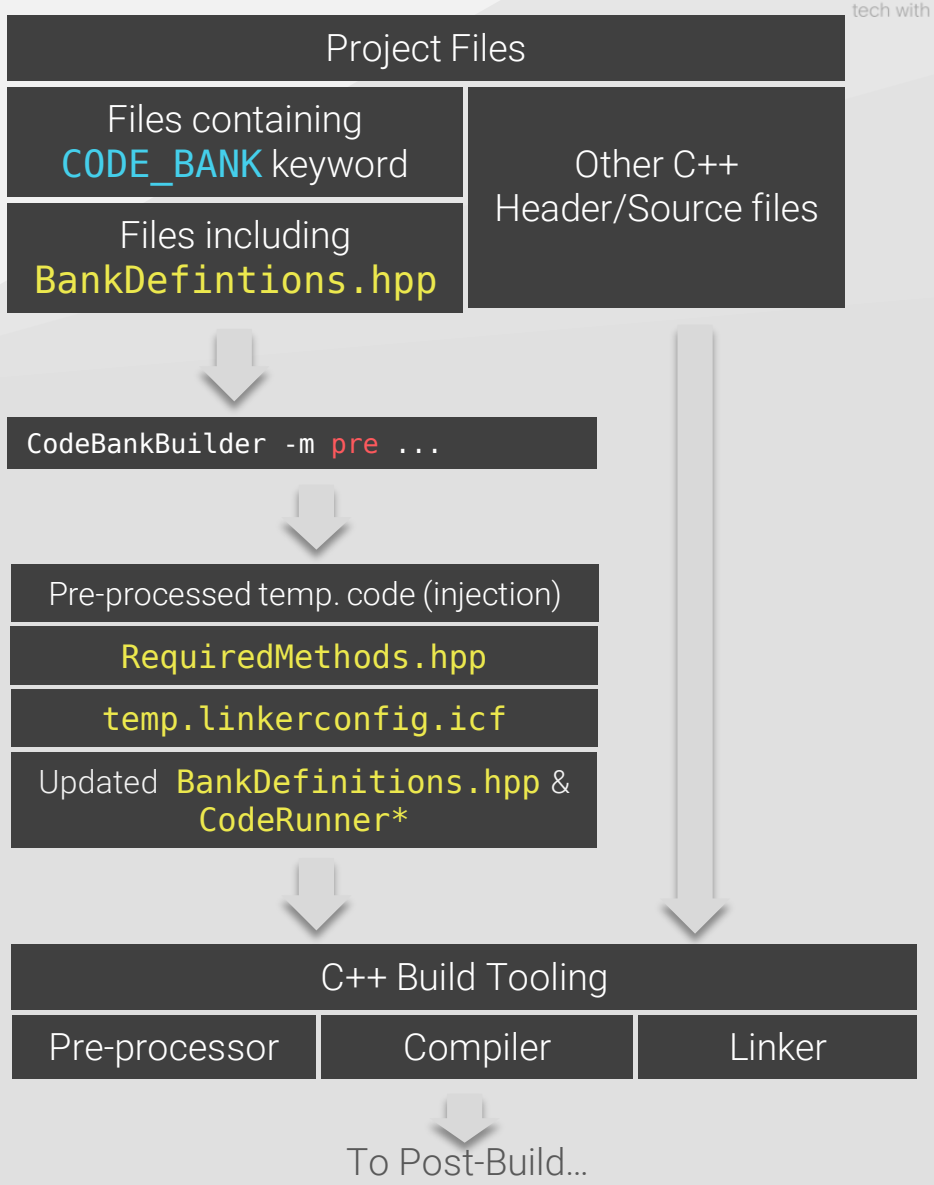
```
$ IarBuild.exe project.ewp -build Debug
$ Running pre-build script at $PROJECT_ROOT
$ CodeBankBuilder -m pre -c $BANKED_CONFIG -v
$ Code Bank Builder v1.0.0
$ > Opening Configuration File (banks.json)
$ > Configurations loaded
$         ObjDump located.
$         $IAR_ROOT updated.
$         658 files detected
$ > Fetching Code Banks
$   Bank 'MyBank' found. Extracting symbols...
$   Parsing 'MyBank.hpp'... Done!
$         Exposed Method Detected: EXP1 (0 arg
$         Exposed Method Detected: EXP2 (1 arg
$         Restricted Method Detected: RES1 (0
$   Region 'RegionMyBank' allocated at 0x10000
$   Symbol table updated with 4 new tokens
$   Bank 'ThatBank' found. Extracting symbols.
$   Parsing 'ThatBank.hpp'... Done!
$         Exposed Method Detected: Exposed (2
$   Region 'RegionThatBank' allocated at 0x110
$   Symbol table updated with 2 new tokens
$ > Writing RequiredMethods.hpp...
$         12288 bytes written
$ > Backing up linker file (s6e1c12_rom.icf)..
$         63602 bytes written
$ > Patching linker file... Done!
$ > Backing up bank files...
$         Two new files created at $PROJECT_RO
$         639124 bytes written
$ > Expanding header files...
$         Expanding 'MyBank.hpp'... Done!
$         Expanding 'ThatBank.hpp'... Done!
$ > Backing up 'BankDefinitions.hpp'... Done!
```

BUILDING CODE BANKS

UNDER THE HOOD

- To allow proper bank's code generation, the project's build must be expanded with extra pre- and post- build steps.

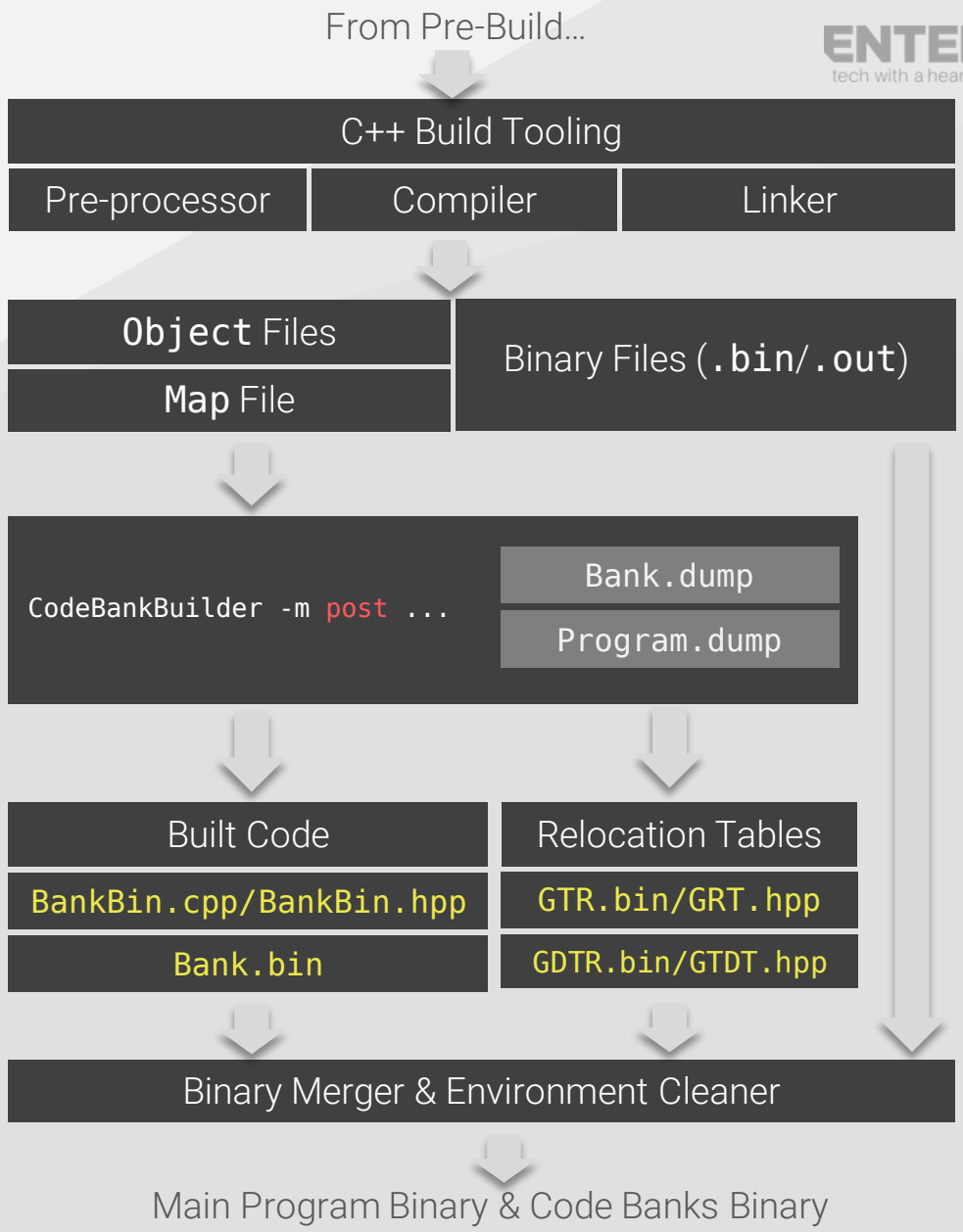
- Both build steps are processed by the **BankedCodeBuilder** tool, developed for the project using C#;
  - The tool requires a JSON input file, describing the project's structure and some output files.
  - Multiple temporary (source, object etc.) files are generated during the build process.



BUILDING CODE BANKS

UNDER THE HOOD

- To allow proper bank's code generation, the project's build must be expanded with extra pre- and post- build steps.
- Both build steps are processed by the **BankedCodeBuilder** tool, developed for the project using C#;
  - The tool requires a JSON input file, describing the project's structure and some output files.
  - Multiple temporary (source, object etc.) files are generated during the build process.



BUILDING CODE BANKS

UNDER THE HOOD: **PRE-BUILD**

- Remove the keywords and operators are processed during the pre-build step
  - Extra code information is acquired during this stage.
  - **In a nutshell**: adjust the code to be properly built.

BUILDING CODE BANKS

UNDER THE HOOD: **PRE-BUILD**

- `BankedCodeBuilder` has to:
  1. Generate a ***symbol table*** containing all *banked-types*, as well as `RESTRICTED` and `EXPOSED` methods;
  2. Update the linker file, to place the *code banks* **outside** of the initial 128KB of flash, and ensure that no PC-relative branches are used;
  3. Generates code for `CodeRunner`;
  4. Injects code wherever `CODE_BANK`, `@` or `of` operators are used;

BUILDING CODE BANKS

UNDER THE HOOD: **STANDARD COMPILATION**

- With all **non-standard structures out of the code**, and with all the appropriate source/configuration adjustments, the C++ build tools are free to do their job:
  - The build follows that standard flow of pre-processing, compilation, (multiple) optimizations and linking;

- If the build is successful, the resulting **object files** and **binaries** will be used in the post-build steps;

BUILDING CODE BANKS

UNDER THE HOOD: **POST-BUILD**

- Generate the *code banks* binary code to create the infrastructure needed for proper execution;
  - It makes use of the **fully built** *main application* assembly code and the **pre-linked** *code banks* code.

- **In a nutshell**: **resolves *main application* dependencies** found in *the code bank* and **outputs the bank's binary code** including relevant metadata.

BUILDING CODE BANKS

UNDER THE HOOD: **POST-BUILD**

- `BankedCodeBuilder` has to:
  1. Locate all *main application* dependencies used in the *code banks;*
  2. Map the dependencies found step 1 with symbols defined in the *main application;*
  3. Generate the **relocation tables** (GRT/GRDT);
  4. Patch the *code banks* code to make use of the tabled from step 3;
  5. Emit the binary code of each *code bank*

BUILDING CODE BANKS

UNDER THE HOOD: **WRAPPING UP BUILD**

- A few extra steps are performed during the post-build:
  - The new content of GRT and GRDT gets re-injected into the binary of the *main application;*
  - Any temporary file is deleted and the original source code is restored.
    - This is specially valid for the pre-build artifacts

PART II.C

RUNNING CODE BANKS

```
0x20d8: 0x4e18          LDR.N R6, [PC, #0x60]  ; .text_8
0x20da: 0x9801          LDR   R0, [SP, #0x4]
0x20dc: 0x42b0          CMP   R0, R6
0x20de: 0xd00e          BEQ.N @20fe
0x20e0: 0x2280          MOVS  R2, #128
0x20e2: 0x0152          LSLS  R2, R2, #5
0x20e4: 0x2004          MOVS  R0, #4
0x20e6: 0x9000          STR   R0, [SP]
0x20e8: 0xab01          ADD   R3, SP, #0x4
0x20ea: 0x2112          MOVS  R1, #18
0x20ec: 0x6868          LDR   R0, [R5, #0x4]
0x20ee: 0xf00c 0xfc45   BL ; 0xe97c
0x20f2: 0x9801          LDR   R0, [SP, #0x4]
0x20f4: 0x42b0          CMP   R0, R6
0x20f6: 0xd102          BNE.N @20fe
0x20f8: 0x2001          MOVS  R0, #1
0x20fa: 0x8028          STRH  R0, [R5]
0x20fc: 0xe000          B.N   @2100
        @20fe:
0x20fe: 0x802c          STRH  R4, [R5]
        @2100:
0x2100: 0x2280          MOVS  R2, #128
0x2102: 0x0192          LSLS  R2, R2, #6
0x2104: 0x2004          MOVS  R0, #4
0x2106: 0x9000          STR   R0, [SP]
0x2108: 0xab01          ADD   R3, SP, #0x4
0x210a: 0x2112          MOVS  R1, #18
0x210c: 0x6868          LDR   R0, [R5, #0x4]
0x210e: 0xf00c 0xfc35   BL ; 0xe97c
0x2112: 0x2402          MOVS  R4, #2
0x2114: 0x9801          LDR   R0, [SP, #0x4]
0x2116: 0x42b0          CMP   R0, R6
0x2118: 0xd00c          BEQ.N @2134
0x211a: 0x22c0          MOVS  R2, #192
```

RUNNING CODE BANKS
DYNAMIC COMPONENTS

- The `CodeRunner` is the essential component when dealing with *code banks* in runtime.
  - It is responsible to **allocate and deallocate** resources in the **program RAM**;
  - It must **properly branch to any** `EXPOSED` method;
  - It must properly **return** from the *code bank;*
- All operations performed by the `CodeRunner` should be executed as efficiently as possible and must appear seamless to the programmer;

RUNNING CODE BANKS

BRANCHING TO/FROM CODE BANKS

- When the execution moment comes, the `CodeRunner` will use a **function pointer** to perform the branch **to** the *code bank* in RAM;
  - Registers `R0` through `R3` are used to pass the arguments and the return address is stored in the **link register**;
  - Braches are performed with the `BLX` instruction, forcing **Thumb-mode** (using a 32-bit address);
  - Return values are stored in `R0`;

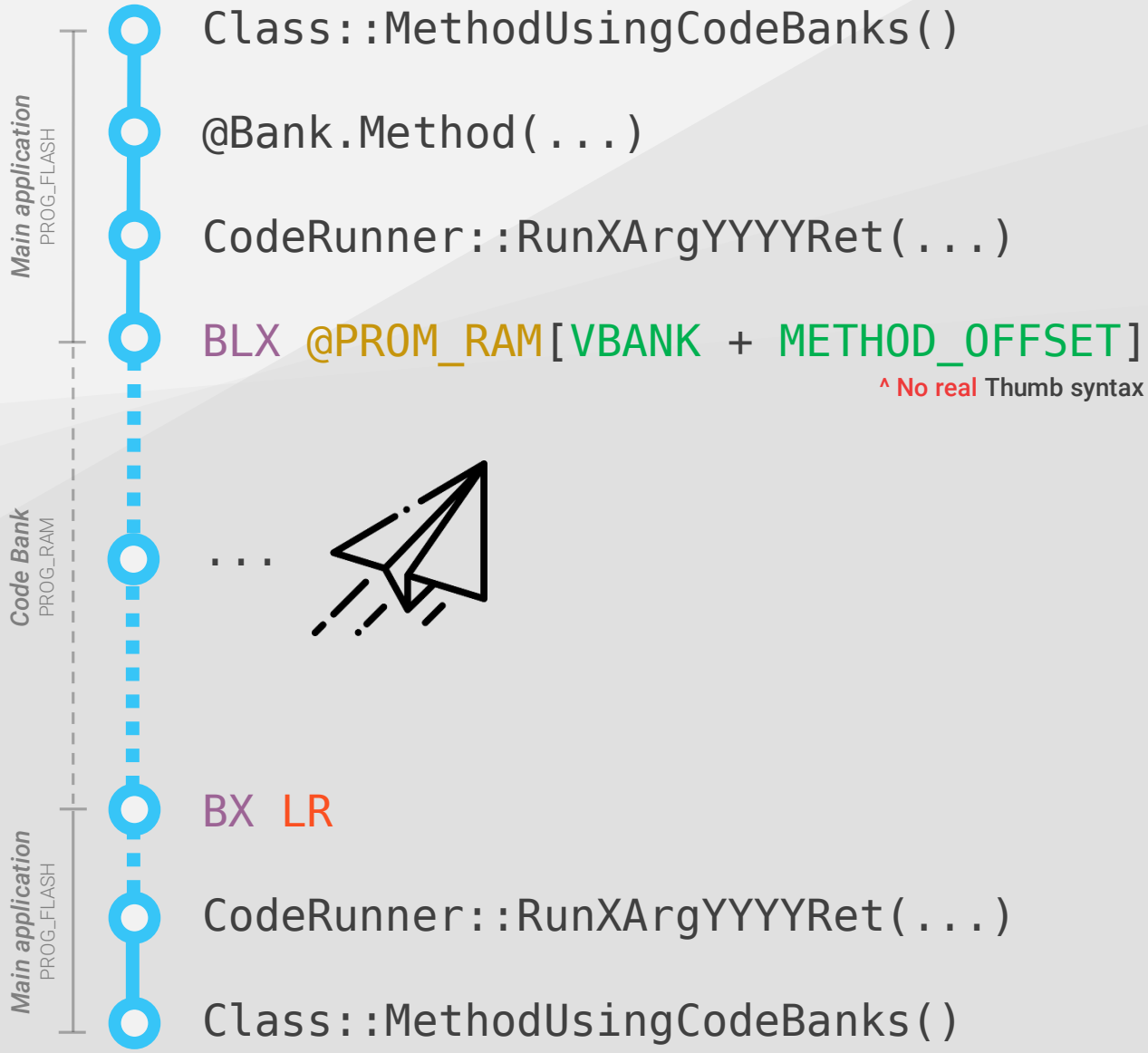
- Branches **from** the *code bank* to the *main application* use **veneering** with absolute 32-bit addresses (as discussed in the previous section);

- Branches **from** *code banks* **to** *other code* banks are indirectly performed through the `CodeRunner` (which ends up as a special case of veneering branch);

RUNNING CODE BANKS

JOURNEY TO CODE BANKS

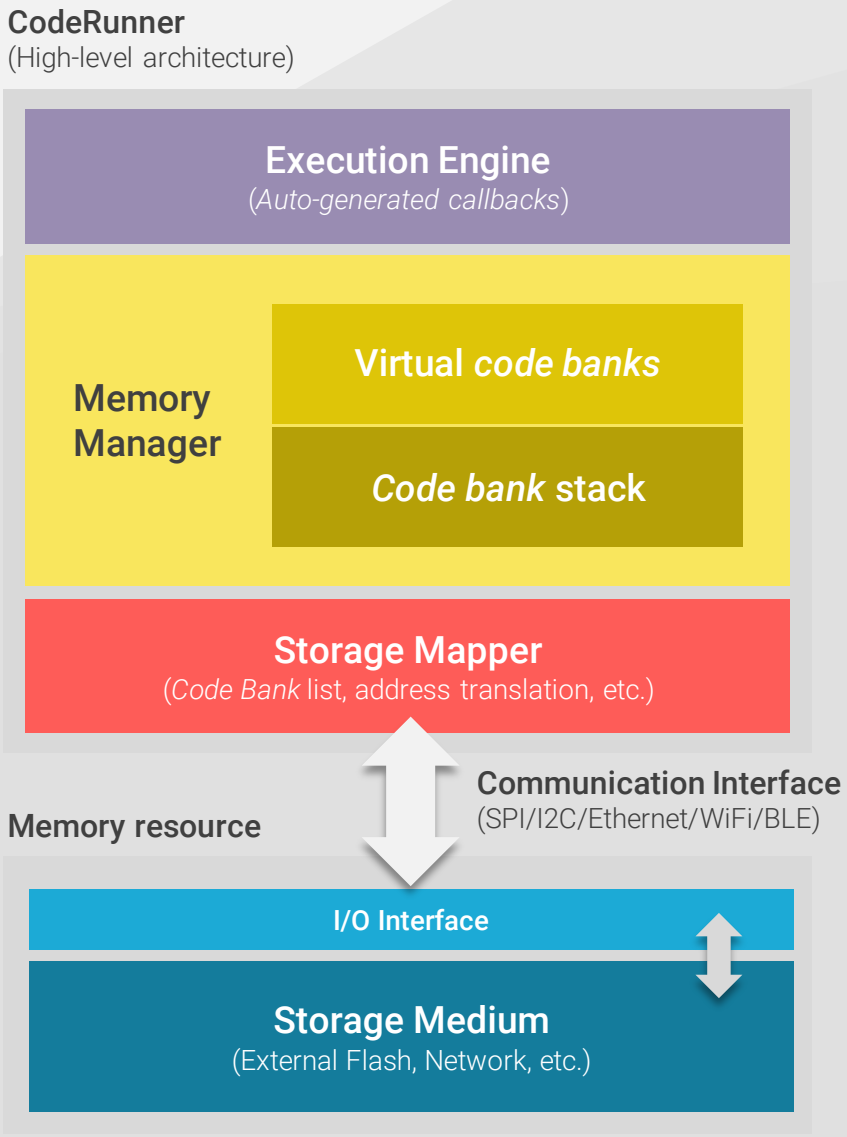
- Within the code bank, anything can happen:
  - Branches to ***the main application***;
  - Branches to the **internal** `RESTRICTED` methods;
  - Indirect **branches to other *code banks***;
  - Execution of **IRQs**



RUNNING CODE BANKS

CODERUNNER INTERNAL ORGANIZATION

- Internally, the code runner is subdivided into three major components:
  - The `ExecutionEngine` is responsible to prepare and perform branches to *code banks*;
  - The `MemoryManager` keeps track of the program RAM resources and virtual banks;
  - The `StorageMapper` is responsible to locate and load *code banks* from an external memory resource;



RUNNING CODE BANKS

CODERUNNER'S EXECUTION ENGINE

- The `ExecutionEngine` is a collection of *callbacks* used to jump to *code banks* loaded in the program RAM;
  - It is composed of **<<N>>** *callbacks* where: each *callback* **<<N_i>>** maps to a **distinct** `EXPOSED` **method signature** (`i ⊆ [0, N)`);
  - Each *callback* **<<N_i>>** will have its unique **input arguments list** or **return types** (`i ⊆ [0, N)`).

Generated based on (example):

```
// Code bank MyBank1
EXPOSED: bool IsComplete();
```

CodeRunner.hpp

```
#ifndef CODE_RUNNER_HPP_
#define CODE_RUNNER_HPP_

#include <cstdint>
#include "BankedCodeUtils.hpp"

class CodeRunner
{
  ...
  // Auto-generated methds
public:
  void Run0ArgVoidRet(uint8_t bankID, uint8_t methodID, void* ctx);
  bool Run0ArgBoolRet(uint8_t bankID, uint8_t methodID, void* ctx);
  char Run0ArgCharRet(uint8_t bankID, uint8_t methodID, void* ctx);
  uint8_t Run0ArgCharRet(uint8_t bankID, uint8_t methodID, void* ctx);

  void Run1ArgVoidRet(uint8_t bankID, uint8_t methodID, void* ctx, void* arg1);
  bool Run1ArgBoolRet(uint8_t bankID, uint8_t methodID, void* ctx, void* arg1);
  char Run1ArgCharRet(uint8_t bankID, uint8_t methodID, void* ctx, void* arg1);
  uint8_t Run1ArgUint8Ret(uint8_t bankID, uint8_t methodID, void* ctx, void* arg1);

  ...
};

#endif
```

Generated based on (example):

```
// Code bank MyBankX
EXPOSED: uint8_t NumberOfEntries(bool update);
```

RUNNING CODE BANKS

MEMORY MANAGEMENT

- Originally, banks were *monolithic* chunks of code taking ownership of the entire program RAM resources when loaded;
  - Only one *code bank* was allowed to be loaded at a time and no *reentrant calls* were allowed.

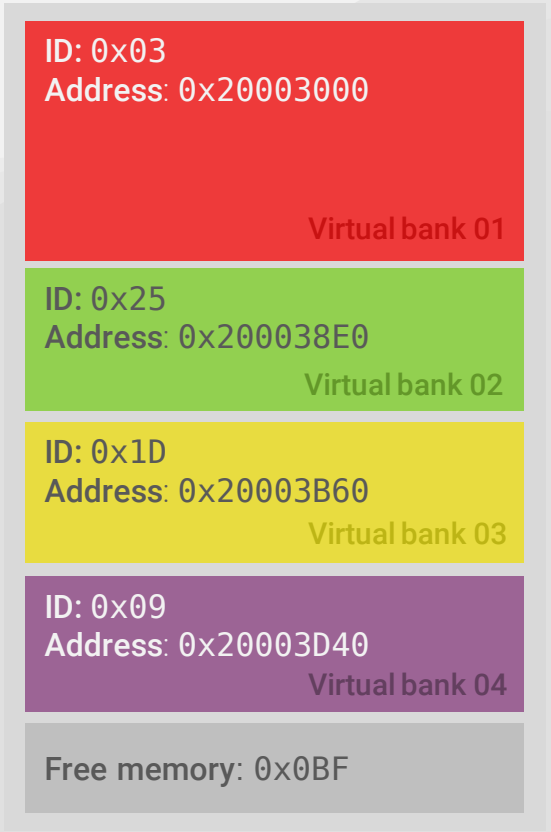
- That's a not very efficient approach:
  - As *code banks* were more frequently used in the project, it was clear that **the average bank size** was way **smaller than program RAM buffer**;
    - In general, banks were between 500 Bytes to 1.5KB while the buffer was as big as 3KB;
    - That lead to a sub-allocation of the program RAM (of about 1/3 to 1/2) leading to low-performance in some cases.
      - That was particularly noticeable in UI-driven *code banks;*

  - *Code banks* **were not allowed to call other *code banks***, or even **to call functions in the *main application* that would make use of *code banks***;

RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

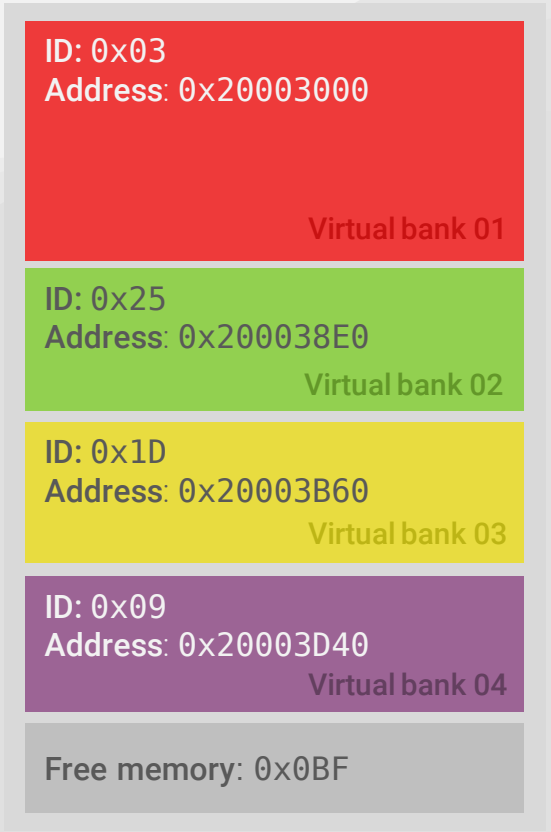
- To solve that problem, the concept of **virtual *code banks*** was developed:
  - A virtual *code bank* is a slice of the Program RAM, holding one *code bank*;
  - Virtual *code banks* can be loaded at any **contiguous memory range** where they fit;



RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

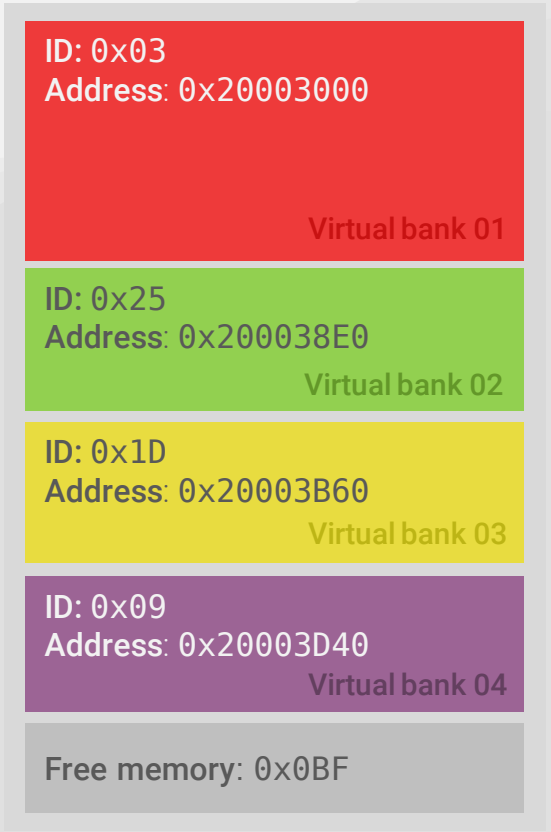
- The stack is used when **suspending** or **resuming** *code banks*:
  - Upon requests, banks can be suspended (i.e. lack of memory) and later resumed, in an operation called **context switching**;



RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

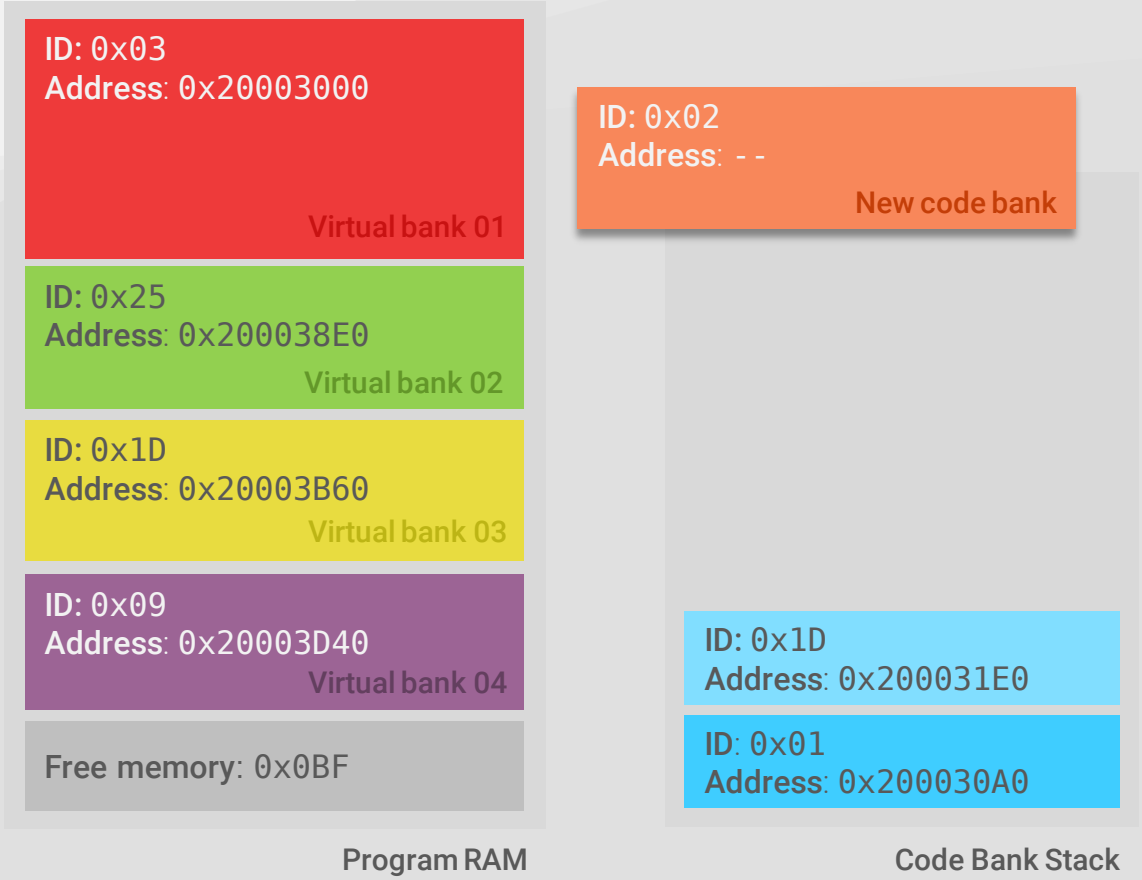
- **4 banks** loaded at the same time;
  - 2 banks were suspended at some point in the past.



RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

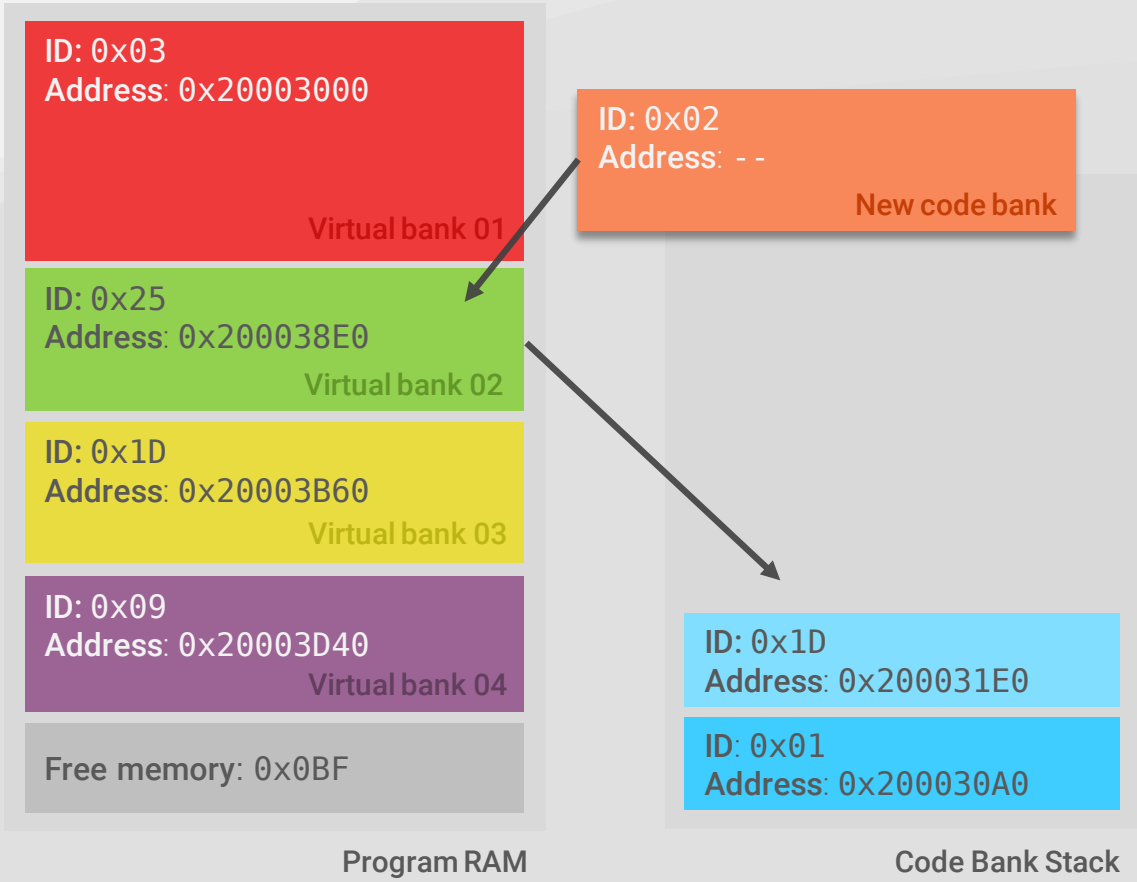
- A new *code bank* is must be **loaded in memory**;
  - But there is **no room** for it!



RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

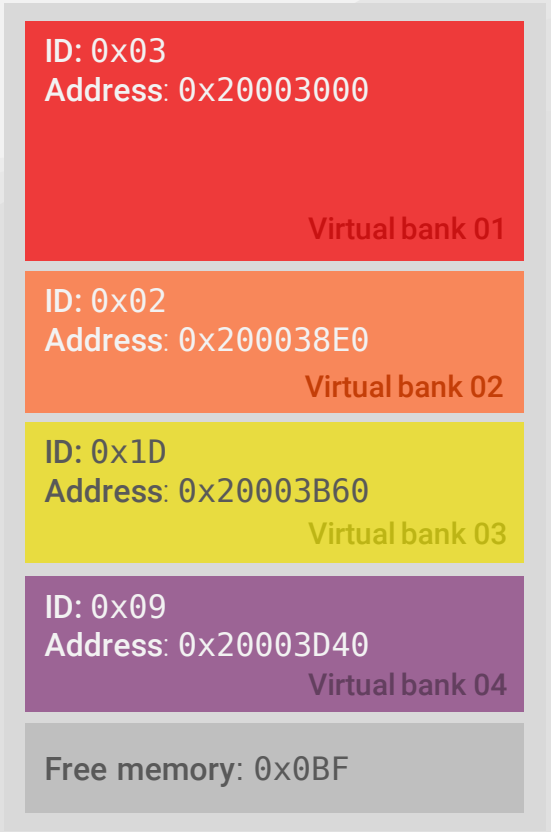
- No problem! **Unload `0x25`** and **load `0x02`** in its place;
  - `0x25` goes to the *code bank* stack for a while;
  - We say that `0x25` has been **suspended**;

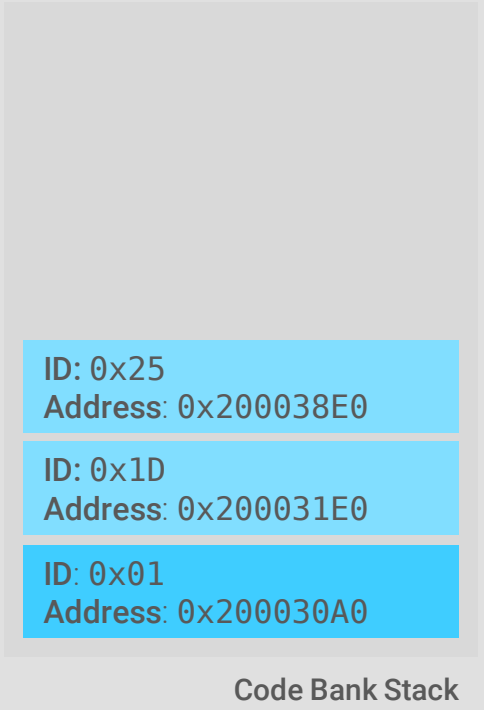


RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

- Now we can run `0x02`!

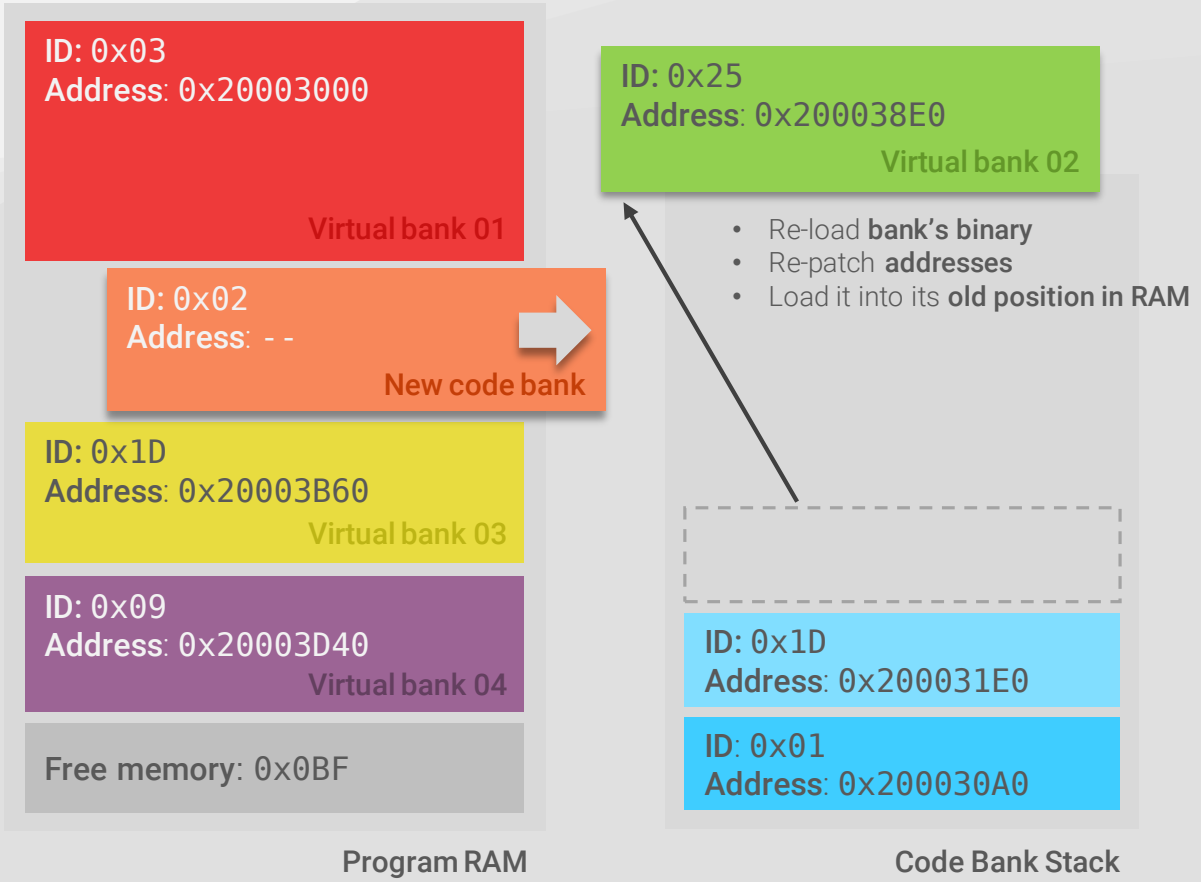




RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

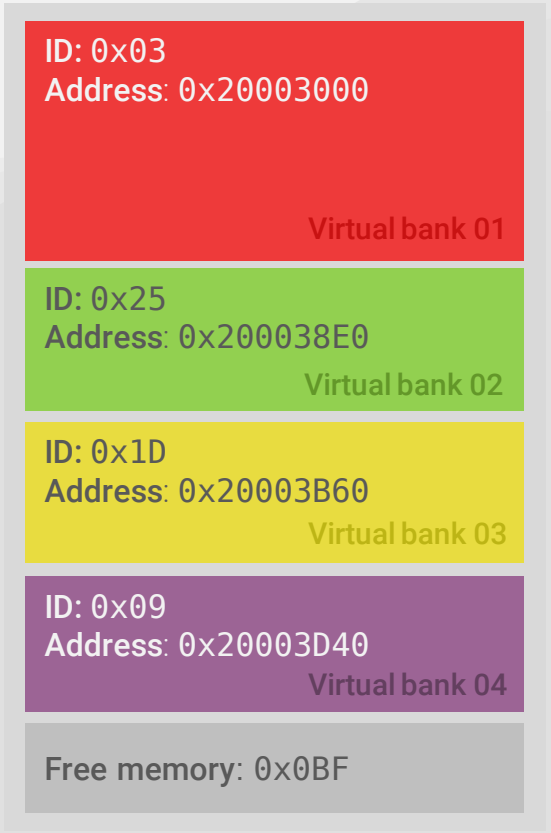
- Done `0x02`! `0x25` can be **resumed**. A full **context switching** is needed.
  - Request and reload `0x25`'s binary code;
  - Re-patch `0x25`'s code
  - Reload it into RAM and update the virtual code bank's house keeping;



RUNNING CODE BANKS

MEMORY MANAGEMENT: VIRTUAL CODE BANKS AND CODE BANK STACK

- Back to the beginning…



RUNNING CODE BANKS

CODE EXECUTION: INS-AND-OUTS

- The execution of a *code bank* is carried over exactly as you would expect:
  - **Stack** and **Heap** usage is shared among the *main application* and the *code banks;*
  - **Exceptions** will trigger a call stack unrolling as they would normally do;
  - *(local)* Variable scope follows the usual rules;
    - Member attributes depend on the life-cycle of the bank's context!

- Some points should be taken into account:
  - Running **virtual methods require proper initialization** of VPTR table;
    - The bank's **constructor** is responsible for doing so: remember that your code must do it explicitly!
  - Similarly, **destructors** need to be called explicitly (if needed);
  - There is no real use of **RTTI** (thus, no use for `dynamic_cast`) with *banked-types*;

RUNNING CODE BANKS

CODE EXECUTION: INS-AND-OUTS

- The `CodeRunner`'s house keeping must be taken into account when integrating it in your application:
  - All algorithms in the `CodeRunner` are of complexity $O(n)$, **mostly I/O-bound**;
  - That means, *the bigger the number of banks, and the size of those banks, the slower the* `CodeRunner` *becomes*;
    - The **speed of the interface** between the CPU and the storage medium, *plays a huge role* in the overall runtime performance;

- *Code banks* place some **pressure over stack size**: every call to a *code bank* requires intermediate steps which can get up to 5 levels (worst case scenario);
  - *Furthermore, re-entrant code banks* **multiply that cost by the amount of nesting calls** *they perform;*
  - Careful stack management is advised!
    - This is also valid when dimensioning the *code bank*'s stack;

PART III

GENERAL TOPICS

This section *quickly* covers:

DEBUG

TESTING

SECURITY

WHAT'S NEXT

DEBUGGING CODE BANKS

- Your success on debugging directly depends on the building/debug tools you are using:
  - In any case, there is **no** *C++ level* debug support;
  - Your tool must be able to **disassemble ARM/Thumb code in RAM** space (IAR can't, btw);
    - Set breakpoints in RAM are also desirable;
- **Forget about debug symbols**:
  - The code loaded in RAM **has been patched** (both in build and runtime);
  - The header will also ruin any addresses mentioned in the dbs files, making the **binary inconsistent** with the compiler-generated debug data;

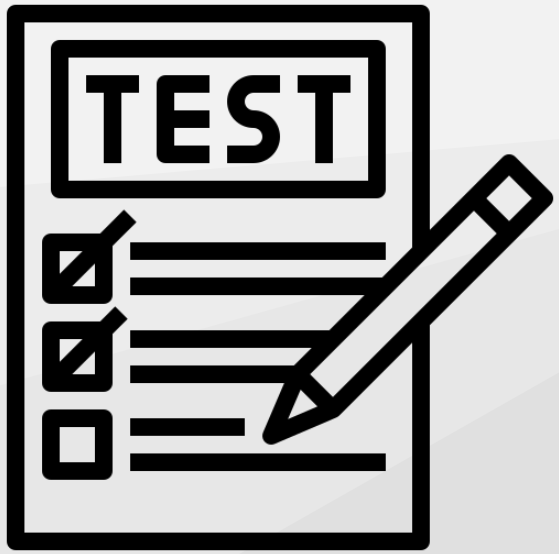
DEBUGGING CODE BANKS

- Be ready to learn ARM binary code…
- That's what I would usually debug with:

```
20000420: 8c3e 0020 f8b5 0400 0d00 a068 0028 05d1  .>. .......h.(..
20000430: fff7 1afe fff7 28fe a060 f1bd fff7 14fe  ......(..`......
20000440: fa21 4900 a068 fff7 17fe 0028 35d0 fff7  .!I..h.....(5...
20000450: 0bfe fff7 19fe a060 794e 3068 2a21 405c  .......`yN0h*!@\
20000460: 0700 0321 0328 08d1 207b 421c 2273 0022  ...!.(.. {B."s."
20000470: 0092 2b00 0322 0240 1ce0 3800 01d0 022f  ..+.."."@..8..../
20000480: 09d1 207b 421c 2273 0022 0092 fff7 e4fd  .. {B."s."......
20000490: 2b00 cab2 0de0 042f 07d1 3068 fff7 0cfe  +....../..0h....
200004a0: 0090 2b00 0022 0421 04e0 0020 0090 2b00  ..+.."."!... ..+.
200004b0: 0022 3900 2000 00f0 11f8 f8bd 6c2e 0020  ."9. .......l..
200004c0: c42e 0020 f81e 0020 2564 0020 242f 0020  ... ... %d..$/.
200004d0: 3b28 0020 3928 0020 3c28 0020 10b5 1400  ;(. 9(. <(. ....
200004e0: 1a00 0529 01d1 2421 33e0 0729 00d0 0329  ...)..$!3..)...)
200004f0: 0bd1 2100 28d0 012c 01d1 2621 29e0 022c  ..!.(..,..&!)..,
20000500: 01d1 2721 25e0 2e21 23e0 0b00 01d0 0229  ..'!%..!#......)
20000510: 09d1 2100 01d1 1821 1be0 012c 01d1 1a21  ..!....!...,...!
```

You are here!

TESTING CODE BANKS



- **Unit testing *code banks* is possible** using code banks, but require some *getting-used to:*
- The new syntax elements **must not be included**. That means:
  - Include a **special testing header** to your code-bank (`BankTestUtils.hpp`);
  - Surround special code (using `@` and `of`) with `#ifdef/#ifndef` conditional compilation statements;
  - Use a `CodeRunner` mock object in the test body;

- Admittedly needs more work to get to an acceptable level;
  - Only *toyed* around with this idea;

SECURITY & CODE BANKS

- *Code banks are usually stored* in *(potentially)* **exposed storage media**;
- That may open room for hackers to **extract the application code** and **reverse-engineer your product**;
  - This could lead to IP leaks;
  - Code injection, leading to misuse of the host hardware;

- Code **encryption is a *must have***:
  - Secure Flash (in the case of external Flash chips);
  - Encrypted communication channels (i.e. Https, in the case of remote code storage);
  - Discuss this with your hardware/production teams;

WHAT'S NEXT

- **Port** to other platforms/compilers;
  - GCC and Keil would be a good start;
- **Simplify** the build flow;
  - Remove the lots of temporary files;
  - Reduce build times (multithread support)
- **Empower** debugging:
  - Generate new debug symbols based on the information acquired during build;
- **Improve** testing infrastructure;

QUESTIONS?

```
// Include required information to allow us
#include "BankDefinitions.hpp"
#include "infrastructure/CodeRunner.hpp"

//Declare code bank and places it at a spec
int main()
{
    ...
    // The code runner should be initialized
    CodeRunner::Initialize(&storageControlle
    ...

    BankContext context of FactorialCalculat

    uint32_t factorial = @FactorialCalculato
);

    std::cout<<"Factorial of 5: "<<factorial
    ...
}

#endif // BANK_HPP_
```

```
#ifndef FACTORIAL_CALCULATOR_HPP_
#define FACTORIAL_CALCULATOR_HPP_

#include <cstdint>
#include "BankedCodeUtils.hpp"

CODE_BANK FactorialCalculator
{
   // Only one Exposed Method
   EXPOSED: uint32_t Calculate(uint8_t number)
     {
       uint32_t result = 1;

       for(auto i = 1; i <= number; i++)
       {
         result = result * i;
       }
```

$ {040}_

FACTORIAL.BANK

FactorialCalculator.hpp

```
#ifndef FACTORIAL_CALCULATOR_HPP_
#define FACTORIAL_CALCULATOR_HPP_

#include <cstdint>
#include "BankedCodeUtils.hpp"

CODE_BANK FactorialCalculator
{
      // Only one Exposed Method
      EXPOSED: uint32_t Calculate(uint8_t number)
      {
          uint32_t result = 1;

          for(auto i = 1; i <= number; i++)
          {
              result = result * i;
          }

          return result;
      }
};

#endif // FACTORIAL_CALCULATOR_HPP_
```

Example Program

```
// C++ std headers
#include <iostream>
#include <cstdint>

// Include required information to allow use of code banks
#include "BankDefinitions.hpp"
#include "infrastructure/CodeRunner.hpp"

//Declare code bank and places it at a specific address.
int main()
{
      ...
      // The code runner should be initialized at some point
      CodeRunner::Initialize(&storageController);
      ...

      BankContext context of FactorialCalculator;
      uint32_t factorial = @FactorialCalculator.Calculate(context, 5);

      std::cout<<"Factorial of 5: "<<factorial<<std::endl;
      ...
}

#endif // BANK_HPP_
```

After properly building:

Output

```
cso@op040 /projects/programs
$ ./Factorial
$ Factorial of 5: 120
```

$ {040}_

040coders.nl

BACKUP

CODERUNNER

CodeRunner.cpp

```
bool CodeRunner::Run2ArgBoolRet(uint8_t bankID, uint8_t methodID, void* ctx, void* arg1, void* arg2)
{
        // Check whether code bank with ID bankID is in memory; If not, load it.
        auto virtualBank = LoadCodeBankIfNeeded(bankID);

        // Get a pointer to the first bank's address in the program RAM and get the Exposed method Offset offset from the bank's header.
        uint8_t *bankPointer = reinterpret_cast<uint8_t*>(reinterpret_cast<uint32_t>(executableMemory) + virtualBank->bankAddress);
        uint16_t methodOffset = GetExposedMethodOffset(bankPointer, methodID);

        // Sanity Check.
        if (methodOffset == 0xFFFF) { SetFailure(UnknownMethod); return; }

        // Create a function pointer to the exact address of the Exposed method and force it to be taken in Thumb mode.
        uint32_t exposedMethodAddress = methodOffset + reinterpret_cast<uint32_t> (bankPointer) + GetHeaderSize(bankPointer);
        bool (*method)(void *, void *, void *) = reinterpret_cast<bool(*)(void *, void *, void *)>(exposedMethodAddress | 0x01);

        // Ensure this bank cannot be unloaded.
        virtualBank->Lock();

        // Branch and acquire return value
        bool returnValue = (*method)(ctx, arg1, arg2);

        // Restore any possible bank suspended during the execution of the Exposed method.
        RestoreSuspendedBanksIfNeeded();
        virtualBank->Unlock();

        return returnValue;
}
```

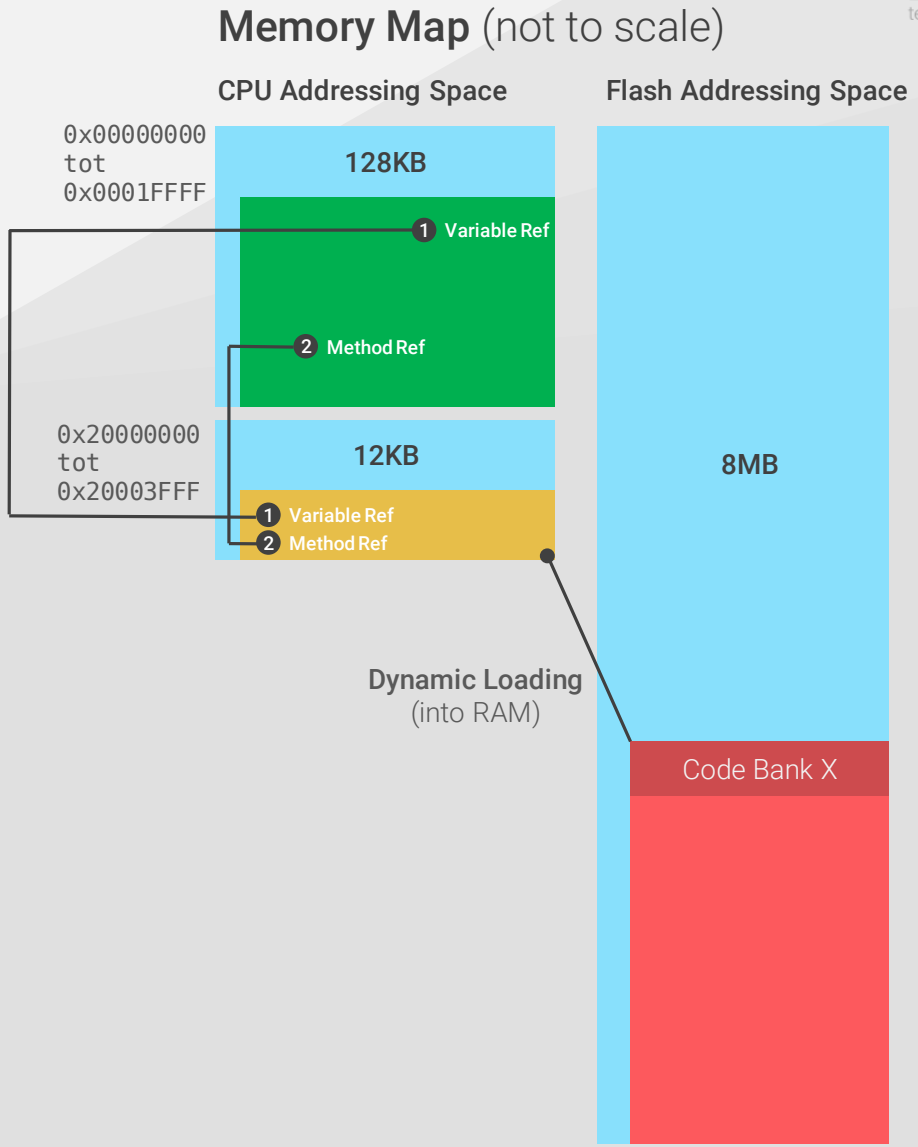
PRE-BUILD::SYMBOL TABLE

- The `BankedCodeBuilder` symbol table is a simple data structure holding the name of each *code bank*, its header file path and a list of each `RESTRICTED` or `EXPOSED` methods.
  - The tool will sweep every file in the project containing `CODE_BANK`, looking for *code banks* declarations.

PRE-BUILD::LINKER FILE UPDATE

- Ensuring that the *code bank* gets built, is just one part of the riddle:
  - *It must be **built correctly***!

1. It places the code **outside the first 128KB** of the CPU addressing space;
   1. Give an unique code region for each code bank
2. It ensures that all references to *main application* variables/functions are **made using absolute addresses**;



PRE-BUILD::CODE GENERATION

- The second-last stage of pre-building consists of `CodeRunner` **generation**;
  - `CodeRunner` is the component capable of branching to *code banks;*
  - It needs **one function for each different *code bank* method signatur**e
    - That's due to the way function pointers and Arm's Procedure Call convention is defined;

PRE-BUILD::CODE GENERATION

```
<RET_TYPE> Run<N_ARGS>Arg<RET_TYPE>Ret(uint8_t bank_id, uint8_t method_id, void* context, void * arg0, ..., void * argN);
```

- The second-last stage of pre-building consists of `CodeRunner` **generation**;
  - `CodeRunner` is the component capable of branching to *code banks;*
  - It needs **one function for each different *code bank* method signature**
    - That's due to the way function pointers and Arm's Procedure Call convention is defined;

PRE-BUILD::CODE GENERATION

- Since `@` is no standard C++ operator, it must be *patched-out* before the code is delivered to the compiler;
  - This is done during the **code injection** phase.
- The `@` operator is ultimately translated into a call to the `CodeRunner` providing:
  - the bank's location in the Flash;
  - the method location in the bank's header;
  - A reference to the context and the arguments;

- Likewise, The `of` operator is translated into the declaration of a `BankContext` array, used as the bank's context used during a method call;
  - `BankContext` is in fact an alias of `uint8_t`;

@ operator transformation

Original Source Code

```
@SampleBank.SampleMethod(NoContext, 0x10);
```

Pass 1: pre-build injection

```
CodeRunner::Instance().Run1ArgVoidRet
(
  BankedTypes.SampleBank,
  SampleBankMethods.SampleMethod,
  reinterpret_cast<void*>(&_noThis),
  reinterpret_cast<void*>(&_arg1)
);
```

of operator transformation

Original Source Code

```
BankContext context of SampleBank;
```

Pass 1: pre-build injection

```
BankContext context[sizeof(SampleBank)];
```

POST-BUILD::LOCATING DEPENDENCIES

- *Code banks unlinked assembly* are used to acquire all *main application* variables and functions referred in the *code bank*;
  - In the object file *dumps*, those references appear as unresolved symbols (such as `_ZN6ExternalCall1` or `_ZN8ExterbakVariable`).
  - A list of all symbols of every *code bank* is held by the `BankedCodeBuilder`.
    - Those symbol must be found in the *main application* (where their addresses can be extracted);

Code bank object dump

```
      $t:
...
0x3e: 0x6869         LDR   R1, [R5, #0x4]
0x40: 0x7d4a         LDRB  R2, [R1, #0x15]
0x42: 0x2a29         CMP   R2, #41              ; 0x29
0x44: 0xd111         BNE.N @6a
0x46: 0x1c48         ADDS  R0, R1, #1
0x48: 0xf7ff 0xfffe  BL    atoi
0x4c: 0x0001         MOVS  R1, R0
0x4e: 0x0600         LSLS  R0, R0, #24
0x50: 0xd020         BEQ.N @94
0x52: 0x48ff         LDR.N R0, REGION_BANK_21   ; _ZN8ExternalVariable
...
0x54: 0x6800         LDR   R0, [R0]
0x56: 0x22bd         MOVS  R2, #189             ; 0xbd
0x58: 0x5c82         LDRB  R2, [R0, R2]
0x5a: 0xb2cb         UXTB  R3, R1
0x5c: 0x429a         CMP   R2, R3
0x5e: 0xd319         BCC.N @94
0x60: 0x1e49         SUBS  R1, R1, #1
0x62: 0xb2c9         UXTB  R1, R1
0x64: 0xf7ff 0xfffe  BL    _ZN6ExternalCall1
...
      $d:
      REGION_BANK_21:
0x148: 0x00000000     DC32  _ZN8ExternalVariable
```

This step is specially interested in **static relocation symbols** such as those labeled with `R_ARM_THM_CALL` or `R_ARM_ABS32`. **No dynamic relocation** is expected to appear throughout the code.

POST-BUILD::ADDRESS MAPPING

- Mapping consists of **locating the symbols in the *main application dump*** (by name) and **extracting their 32-bit addresses**;
  - **Thumb-class relocation** symbols such as `R_ARM_THM_CALL` will map into *main application* function addresses and will be used as part of the **veneering patching** stage;
    - Those addresses will ultimately be added to **GRT**;
  - **Data-class relocation** symbols such as `R_ARM_ABS32` will map into *main application* variables and will be inserted in the *code bank*'s header to be used during the **address patching**;
    - Those addresses will ultimately be added to **GRDT**;

POST-BUILD::GLOBAL RELOCATION TABLES

- The Global Relocation Tables (GRT & GRDT) are a way to be **flexible with the dependencies** shared among the *main application* and the *code banks*;
  - It creates truly *position-independent* code, **allowing the main application to independently change**, without requiring a full re-build of the code banks.

- The content of the tables are used in two situations:
  - **Static Address Relocation** (SAR);
  - **Dynamic Address Patching** (DAP);

POST-BUILD::GLOBAL RELOCATION TABLES

- In the current implementation, GRT/GRDT is list of **128 32-bit void pointers** to methods in the *main application*.
  - GRT starts at address `0x1F8E0`;
  - GRDT starts at address `0x1F6E0`;
  - Each entry has a **well known address** (`base + (entry_index*4)`) and an appropriate label `GRT_ENTRY_XXX/GRDT_ENTRY_XXX`;

GRDT

```
...

// _variable1
#pragma location = 0x0001F6E0
__root const uint32_t GRDT_ENTRY_0001 = 0x200024e0;

// _variable2
#pragma location = 0x0001F6E4
__root const uint32_t GRDT_ENTRY_0002 = 0x20003e8c;

// _variable3
#pragma location = 0x0001F6E8
__root const uint32_t GRDT_ENTRY_0003 = 0x20002e6c;

// _variable4
#pragma locati
__root const u

// _variable5
#pragma locati
__root const u

...
```

GRT

```
...

// __aeabi_memcpy
#pragma location = 0x0001F8E0
__root const uint32_t GRT_ENTRY_0001 = 0x000059E9;

// __iar_small_idiv
#pragma location = 0x0001F8E4
__root const uint32_t GRT_ENTRY_0002 = 0x00006BD5;

// snprintf
#pragma location = 0x0001F8E8
__root const uint32_t GRT_ENTRY_0003 = 0x0000C6FB;

// _Z6ito02diPc
#pragma location = 0x0001F8EC
__root const uint32_t GRT_ENTRY_0004 = 0x0000C719;

...
```

POST-BUILD::VENEERS

- Veneers are small sections of code generated by the linker to **allow branching outside of the range of** BL instructions;
  - Veneers are required because the runtime address of a *code bank lies about 64MB away from the main application* (starting at `0x20003000`);
  - All veneers generated in *code banks* are thus **long branch veneers**;

Memcpy veneeer

```
; Linker-generated code
            $t:
            `?Veneer 14 (3) for __aeabi_memcpy`:
0x10000000: 0xb408          PUSH  {R3}
0x10000002: 0x4b02          LDR.N R3, [PC, #0x8] ; __aeabi_memcpy
0x10000004: 0x469c          MOV   R12, R3
0x10000006: 0xbc08          POP   {R3}
0x10000008: 0x4760          BX    R12
0x1000000a: 0x46c0          MOV   R8, R8
            $d:
0x1000000c: 0x00000000      DC32  __aeabi_memcpy
```

In-depth:

1. Push **R3** to (for temporary calculation)
2. Load the absolute address of **memcpy** into **R3** located 8 bytes away of the current PC value
3. Save the address into **R12** (note that **R12** content is allowed to be modified according to **APCS**)
4. Pop **R3** (restore argument)
5. Branch to the address in **R12** (in Thumb mode)
6. NOP

POST-BUILD::VENEER PATCHING

- After the Relocation Tables are generated, each veneer detected in the *code banks* **must branch to an address held by a GRT entry**;
- This is done by ***directly changing the absolute address stored in the veneer, by the address of the GRT entry holding the pointer to the to the function targeted by the veneer***;
  - Furthermore, due to the indirection introduced by GRT, the assembly code itself must be updated, to use the **indirect addressing mode** when loading the function address;

Original Veneer

```
; Linker-generated code
            $t:
            `?Veneer 14 (3) for __aeabi_memcpy`:
 0x10000000: 0xb408          PUSH  {R3}
 0x10000002: 0x4b02          LDR.N R3, [PC, #0x8]
 0x10000004: 0x469c          MOV   R12, R3
 0x10000006: 0xbc08          POP   {R3}
 0x10000008: 0x4760          BX    R12
 0x1000000a: 0x46c0          MOV   R8, R8
            $d:
 0x1000000c: 0x00000000      DC32  __aeabi_memcpy
```

Patched Veneer

```
; Patched veneer code
            $t:
            `?Veneer 14 (3) for __aeabi_memcpy`:
 0x10000000: 0xb408          PUSH  {R3}
 0x10000002: 0x4b02          LDR.N R3, [PC, #0x8]
 0x10000004: 0x1b68          LDR.N R3, [R3]
 0x10000004: 0x9c46          MOV   R12, R3
 0x10000008: 0xbc08          POP   {R3}
 0x1000000a: 0x4760          BX    R12
            $d:
 0x1000000c: 0x0001f8e0      DC32  GRT_ENTRY_0001
```

POST-BUILD::BINARY GENERATION

- The last post-build step is the generation of the binary code of each *code bank;*
  - The binaries generated in this state are ready to be programmed in the device's external flash and have the proper structure to be compatible with the `CodeRunner`;

- Each *code bank* binary holds an header providing metadata used by the `CodeRunner` to locate `EXPOSED` methods and to perform **address patching**;