STM32WL MBMUX mailbox multiplexer

Revision 1.0

Hello, and welcome to this presentation of the STM32WL's Mailbox Multiplexer .

Scope

- MbMux aims to facilitate communications between the two STM32WL cores: CM4 (CPU1) and CM0PLUS (CPU2).
- MbMux is a software framework specifically tailored for the needs of SubGhz examples/applications running on STM32WL devices operating in Dual Core.
- It does not aim to be a middleware generic to all kind of devices and applications.
- MbMux design results from a trade-off between simplicity versus flexibility.

life.augmented

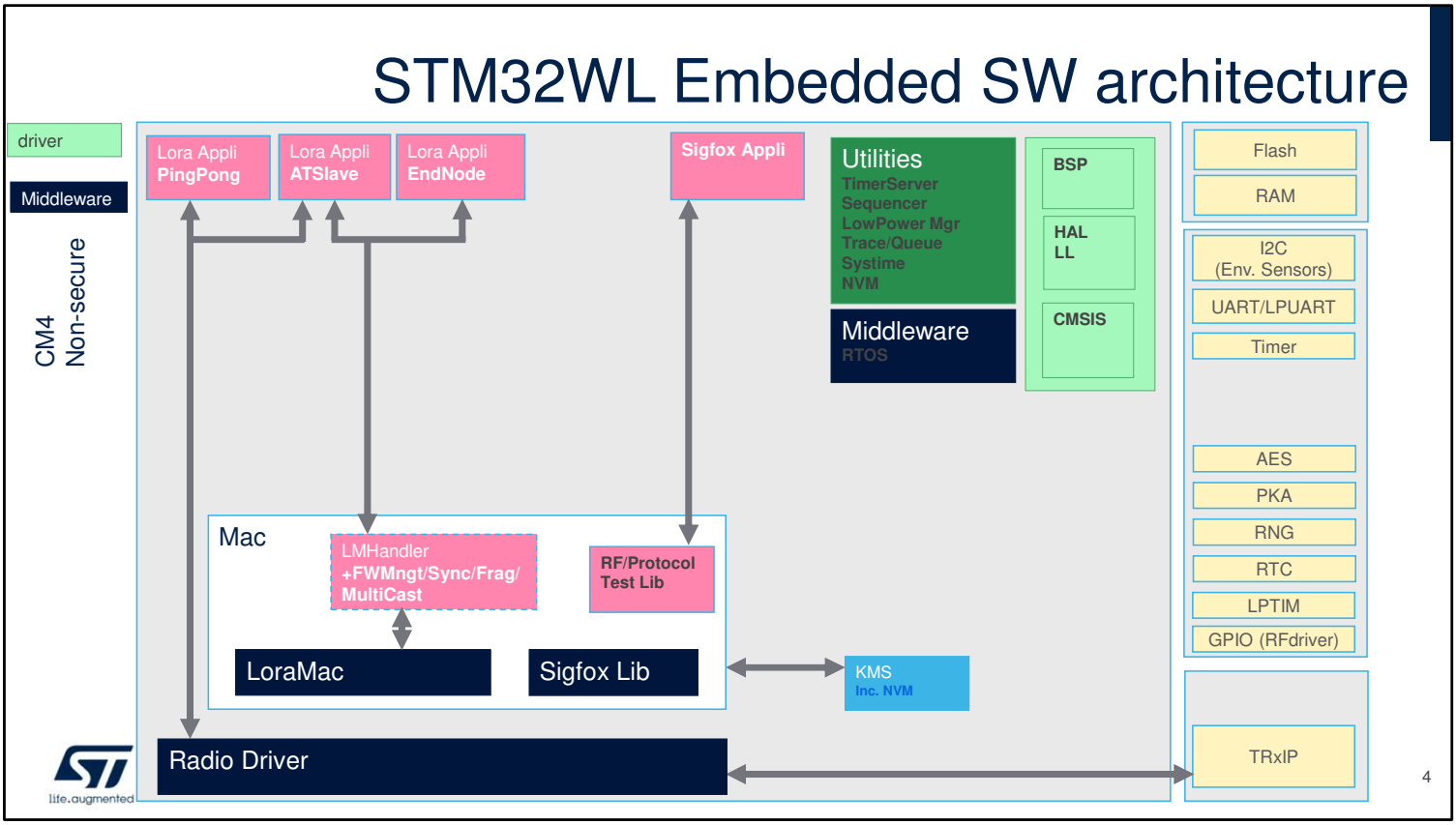
MbMux aims to facilitate communications between the two STM32WL cores: CM4 (CPU1) and CM0PLUS (CPU2).
MbMux is a software framework specifically tailored for the needs of SubGhz examples/applications running on STM32WL devices operating in Dual Core.
It does not aim to be a middleware generic to all kind of devices and applications.
MbMux design results from a trade-off between simplicity versus flexibility.

Singlecore vs dualcore

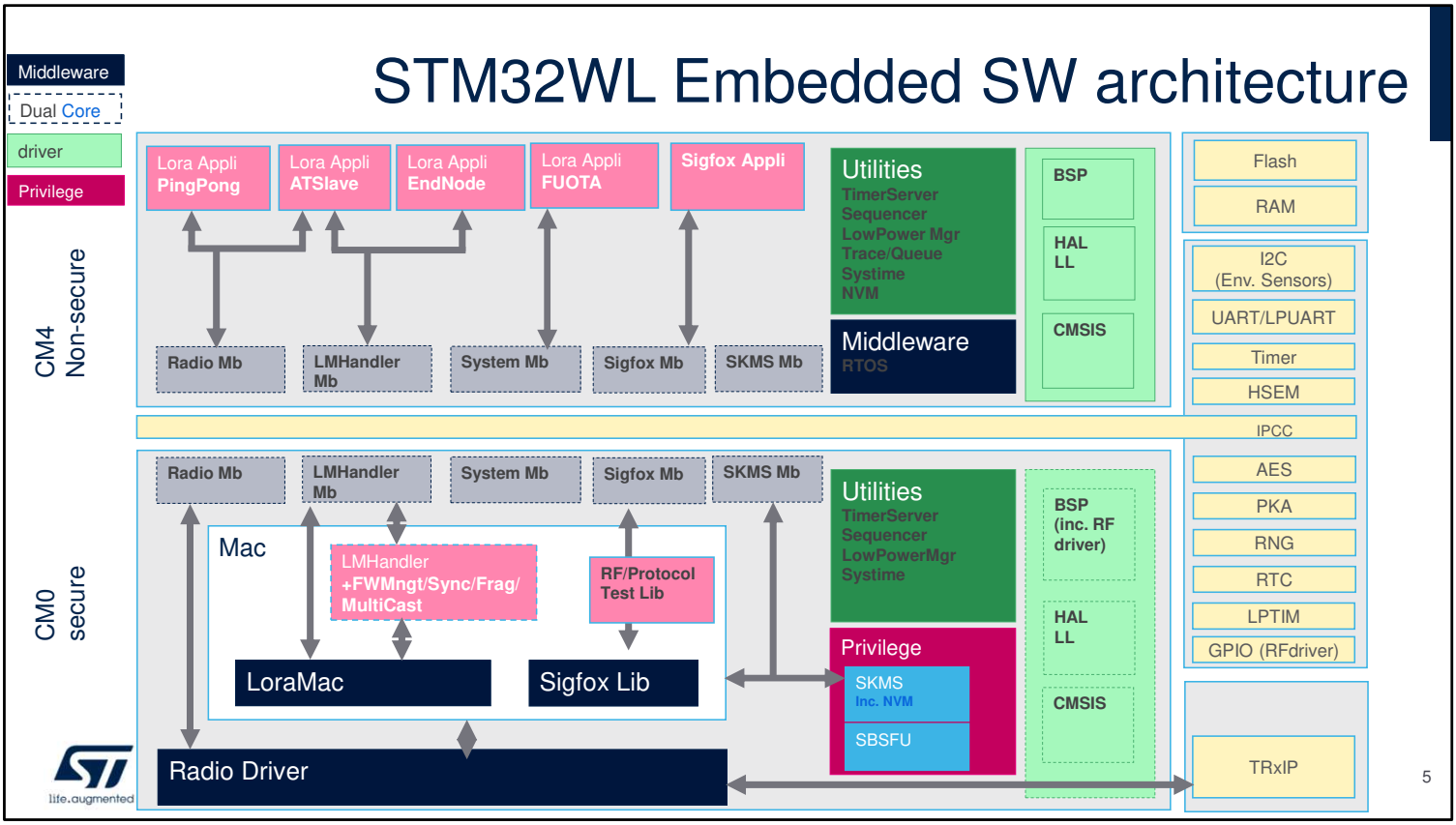
- Differences between Single Core & Dual Core
  - Two separated binaries
    - ->no common linker
  - Interface between the 2 cores built to allow communication (Inter Processor Communication Controller (IPCC) + Shared memory (SH-mem))
  - Common language between Cores to establish
    - Structure type in common, pointer to the structure to be exchanged, etc.
- Implications
  - The two firmware (CM4 and CM0PLUS) have to be compatible
  - CM0PLUS binaries capabilities (supported features) need to be known by CM4
    - Same release version can be built with different compilation options (e.g. regions)

life.augmented

Why Dual Core environment needs such a framework?
Because the two cores are fully independent, and the two binaries don't share a common linker file (exception made for FOUTA project).
MbMux provides communication channels via IPCC and shared memory and it establishes a common language that enables the two cores to exchange information.
This implies that the firmware version on the two cores must be compatible. The two cores must have a mean to know which features are supported by the other core (e.g. Lora rather than Sigfox).



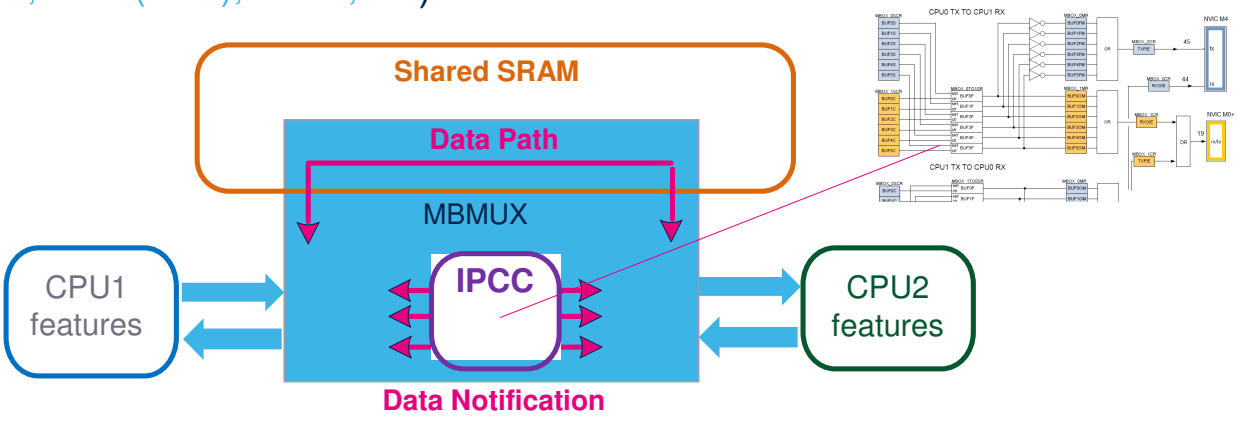
This slide shows the usual architecture for single core projects. Applications (LoraWan and/or Sigfox) make use of the middleware and the radio drivers available on the same core.



This slide exposes the more complex architecture for dual core projects. The provided examples are such that the applicative part is implemented on the CM4 core, while the stacks and the radio drivers are running on the CM0PLUS core.
The aim of this model is to allow customers to change the application on CM4 core without impacting the stack behavior on CM0PLUS core.
The grey boxes (Radio Mb, Sigfox Mb, etc) are the modules composing the MbMux framework. They allow interaction between applications and stacks or radio drivers.

What MBMUX stands for?

- **MB**MUX is a **M**ail**B**ox layer that combines IPCC channels irqs with share memory buffers to allow message exchange between the two CPUs
- MB**MUX** is also a **Mux** that allows the mapping of features (Lora, Sigfox, Radio, Mwbus, Kms (SKS), Trace, etc) on the 2x6 IPCC channels



life.augmented

MbMux stands for Mailbox plus Multiplexer.
The Mailbox allows communication in between the two cores via two physical means: IPCC and shared memory.
The Mux allows dynamic mapping between features and IPCC channels.
The next slides give a bit more details about these elements: IPCC, memory buffers, channels, features, etc.

Ipcc

- IPCC: Inter-Processor Communication Controller
- This IP is available on many STM32 families (STM32MP1, STM32WB, STM32WL, etc)
- The IPCC is designed to handle two groups of channels:
  - IPCC_1TO2 corresponds to the Tx channels of the CPU1 and the Rx channels of the CPU2.
  - IPCC_2TO1 corresponds the Tx channels of the CPU2 and the Rx channels of the CPU1.
- For a given chipset, the number of channels is fixed (2x2 on STM32MP1, 2x6 on STM32WB, 2x6 for STM32WL)
- It does not handle buffers nor memories
- On STM32WL, CPU1 is CM4 and CPU2 is CM0plus

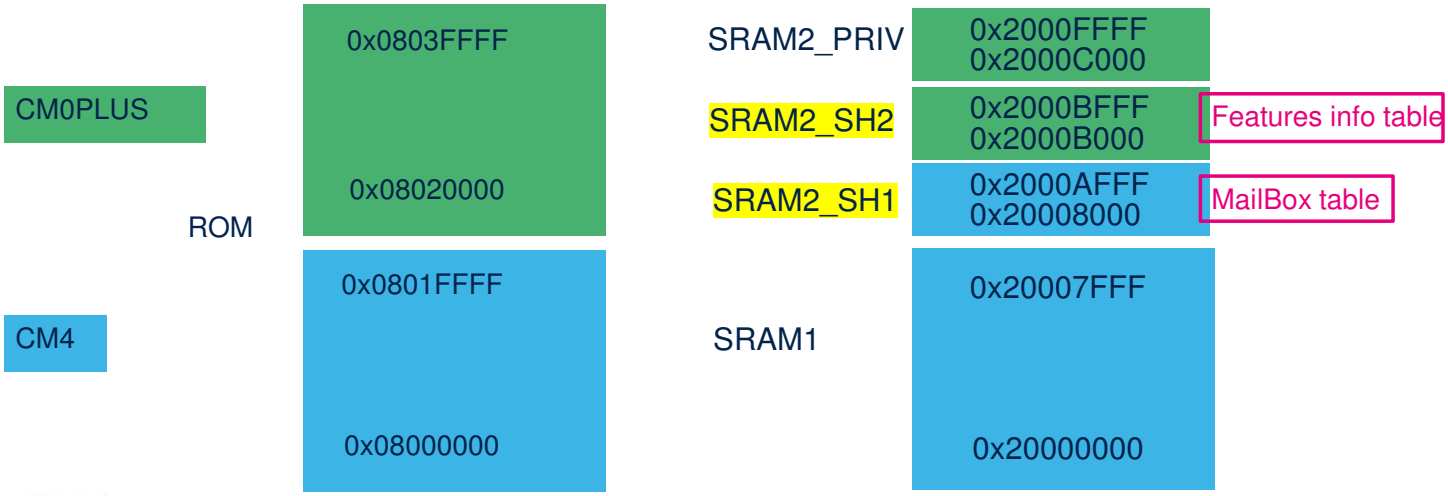
life.augmented

7

Let's start by IPCC, which stands for Inter-Processor Communication Controller. This IP is available on different STM32 families (STM32MP1, STM32WB, STM32WL, etc).
It is based on several communication channels depending on the device family (2x2 on STM32MP1, 2x6 on STM32WB, 2x6 for STM32WL).
IPCC does not handle buffers nor memories, but it triggers interrupts to both cores.

Shared memory

- Example of linker file for EndNode in v1.0.0 release

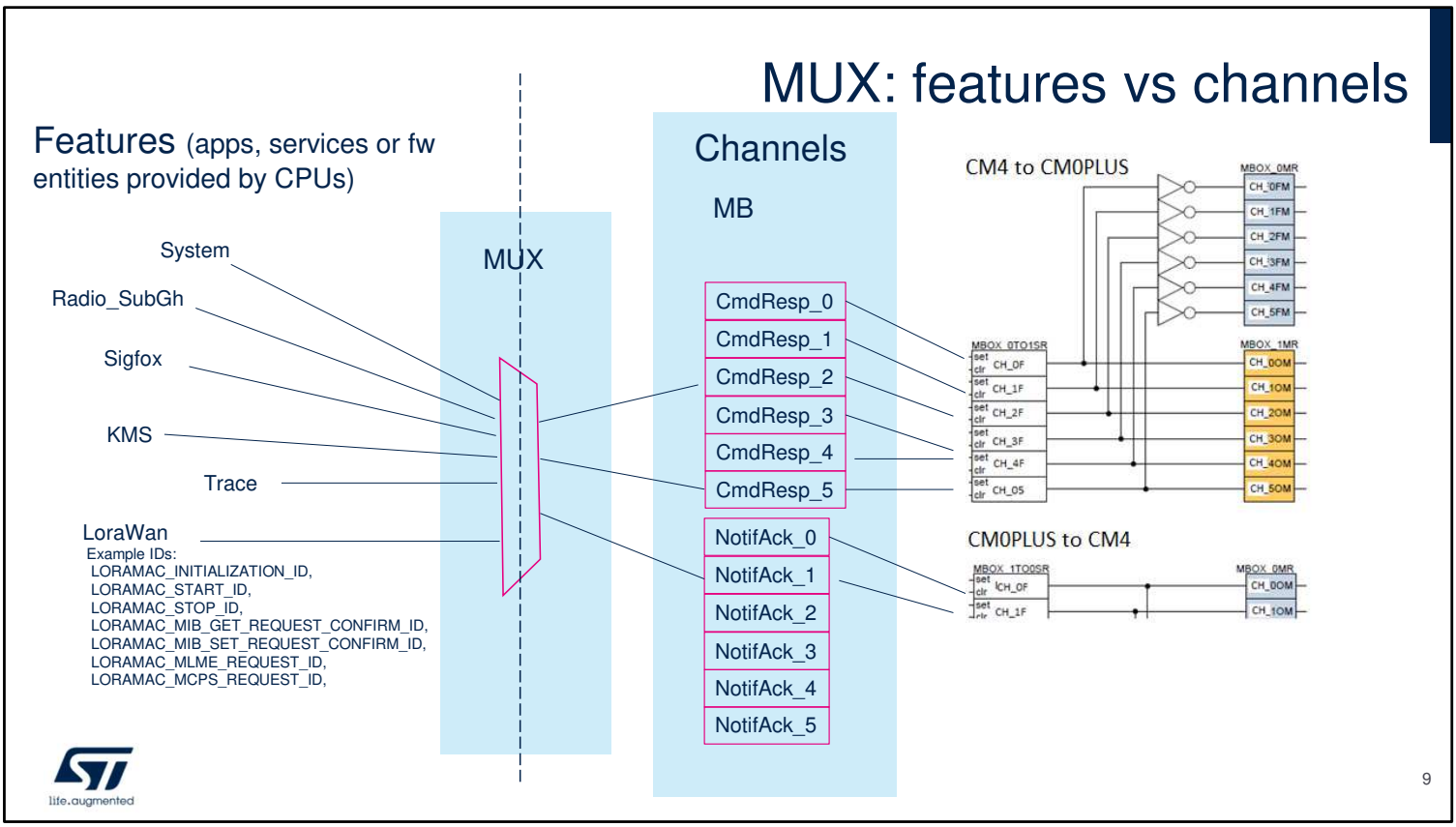


life.augmented

8

To exchange information, a shared memory is required. Typically, in SubGhz applications provided in the ST delivery the linker files are such that:
SRAM1 in entirely used by CM4: c-stack, heap, cm4 global variable, etc.
SRAM2_PRIV is used by CM0PLUS (It it is protected when security is enabled): c-stack, heap, cm4 global variable, secure code, etc.
In SubGhz linker files, MBMUX shared memory is placed in SRAM2 because of the retention properties in Standby mode. Furthermore, the address 0x20008000 matches the default value of the IPCCDBA option byte (more details will be explained later).
The Shared memory is divided into 2 parts: SH1 and SH2
SRAM2_SH1 is the shared memory for items allocated by CM4 (e.g. Mailbox table). This table is used for

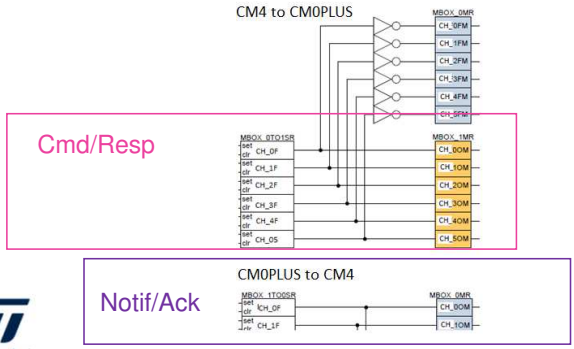
message exchange in both directions (Cmd/Resp & Notif/Ack).
SRAM2_SH2 is the shared memory for items allocated by CM0PLUS (e.g. Features-Information table). This table is used only at initialization by CM0PLUS to expose its capabilities.

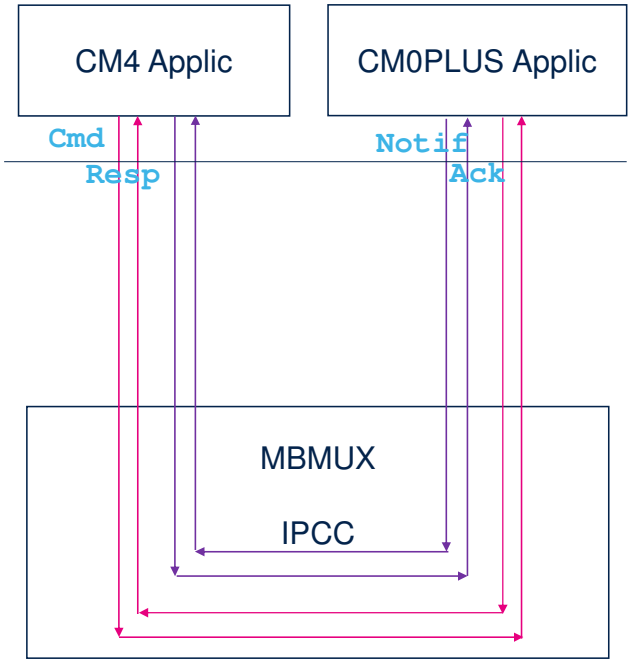


Last concepts to describe are the features and the channels.
The channels are physical entities provided by IPCC hardware.
The features are firmware modules/entities mapped on the IPCC hardware channels (Lora, Sigfox, Kms, Trace, etc)
The mapping is dynamic and done via SW registration.
The module which handles the mapping is the multiplexer (MUX).
The right of the picture shows the 6 Command/Response channels and 6 Notif/Ack channels that can be mapped to the features listed on the left.

Firmware dictionary: cmd, rsp, notif & ack

- **Cmd**: sent by CM4 to CM0PLUS on IPCC Tx-channels
- **Resp**: sent by CM0PLUS to CM4 on IPCC Tx-channels
- **Notif**: sent by CM0PLUS to CM4 on IPCC Rx-channels
- **Ack**: sent by CM4 to CM0PLUS on IPCC Rx-channels





10

IPCC channels are organized by pairs (so called Tx and Rx). MbMux uses Tx channels for Commands/Responses and Rx channels for Notifications/Acknowledgements.

Features list

- **System**: It supports all communications related to the system.
  - System initialization
  - IPCC channels vs features registration
  - To exchange features attributes/capabilities information
  - RTC Additional system channels can be added for high priority operations (e.g. RTC notifications)
- **Trace**: CM0plus fills a circular queue with its logs which are send to CM4 via IPCC. Cm4 is in charge to handle this information by output it on the same means used for classical Cm4 logs (E.g. USART)
- **KMS/SKS**: Secure Key Storage
- **Radio drivers**: It is possible to interface directly with the sub-ghz radio
- **LoraWan stack**.: this channel is used to interface all LoraWan Commands (Init, request, etc) and Events (response/indication) related to LoraWan protocol.
- **Sigfox stack**: this channel is used to interface all Sigfox Commands (Init, request, etc) and Events (response/indication) related to Sigfox protocol
- More features can be added

life.augmented

11

The features currently available are shortly described here.
System is not really a feature, it handles the core of the Mailbox, it always uses channel zero.
Trace is used when the CM0PLUS core needs to log some data. It sends the log to the CM4 core via the mailbox.
The description of other features as LoraWan, Sigfox, KMS are treated by other presentations.
Additional features could be added (e.g. Wmbus). They will be dynamically mapped on channels.

Features registration on channels

- The MbMux has been implemented considering cases where the number of “application features” might be superior to the available IPCC channels. Therefore the mapping is not hardcoded.
- MbMux associates each “feature” with one or more channel(s). The upper layer just tells MbMux which feature is sending a Msg, and the MbMux uses the associated channel transparently.
- Each feature can reserve or release a channel via registration/cancellation; MbMux decides which channel to associate depending on the availability.
- MbMux “system” checks :
  - If enough channels are still available to register the feature requesting registration.
  - The requesting features does not get registered twice
  - The requesting feature is available on the remote CPU (CM0PLUS)
- The IPCC TX and RX channel directions are independent, so it is possible a feature doesn’t need to use the channel in one direction; but it’s also possible that it needs two channels in the other directions
  - Example is TRACE feature which doesn’t use TX direction.

life.augmented

12

The MbMux has been implemented considering cases where the number of “application features” might be superior to the available IPCC channels. Therefore, the mapping is not hardcoded.
MbMux associates each “feature” with one or more channel(s). The upper layer just tells the MbMux which feature is sending a message, and the MbMux uses the associated channel transparently.
Each feature can reserve or release a channel via registration/cancellation; MbMux decides which channel to associate depending on the availability.

MBMUX: feature mapping

- Automatically allocates features on channels by registration/cancellation

feature	Ipcc channel Tx (CPU1->CPU2)	Ipcc channel Rx (CPU2->CPU1)
system	0	0
LoraWan	Auto By Registration	Auto By Registration
Radio	Auto By Registration	Auto By Registration
Sigfox	Auto By Registration	Auto By Registration
KMS*	Auto By Registration	NA
Trace	NA	Auto By Registration
RTC	NA	Auto By Registration

life.augmented

* KMS not fully implemented in CubeWL v1.0.0

This table summarizes the mapping expressed in previous slides.

How features use the channels

- In principle the channels and the shared buffers can be used to exchange any kind of message as far as a common language between the two cores is defined.
- In practice the “language” has a very specific goal: “Allowing one core to execute functions on the other core and retrieve the return value or the processed buffers”.
- The mailbox shared buffers are designed to reach that goal:
  - Cmd are used by CM4 to call functions implemented on CM0PLUS
  - Rsp are used by CM0PLUS to tell that the function has been executed and a return value is available
  - Notif are used by CM0PLUS to call functions implemented on CM4
  - Ack are used by CM4 to tell that the function has been executed and a return value is available

life.augmented

The channels and the shared buffers can be used to exchange any kind of message as far as a common language between the two cores is defined.
The mailbox shared buffers are designed to reach that goal:
- Cmd are used by CM4 to call functions implemented on CM0PLUS
- Rsp are used by CM0PLUS to tell that the function has been executed and a return value is available
- Notif are used by CM0PLUS to call functions implemented on CM4
- Ack are used by CM4 to tell that the function has been executed and a return value is available.

Communication language

- Messages are formatted according to the aim of the language:
  - To identify the function to call via an identifier (MsgId)
  - To pass function parameters and/or pointers to buffers and structures
  - To get alerted when a function has been executed and to retrieve the return value

```
typedef struct {
  uint32_t MsgId;
  void  (*MsgCpu1Cb)(void* ComObj);
  void  (*MsgCpu2Cb)(void* ComObj);
  uint16_t BufSize;      /*!<  size of the array given by the applic */
  uint16_t ParamCnt;     /*!<  nr of words composing the message */
  uint32_t* ParamBuf;
  uint32_t ReturnVal;
} MBMUX_ComParam_t;
```

- In green values set at registration (once for each feature)
- In blue values set each time a Msg (Cmd/Resp/Notif/Ack) is sent

life.augmented

15

Hence the messages have been formatted according to the aim of the language:

- To identify the function to call via an identifier (MsgId)
- To pass parameters and/or pointers to buffers and structures
- To get alerted when a function has been executed and to retrieve the return value

The structure shown here comes directly from the code. We will see later more details about it.

Communication table

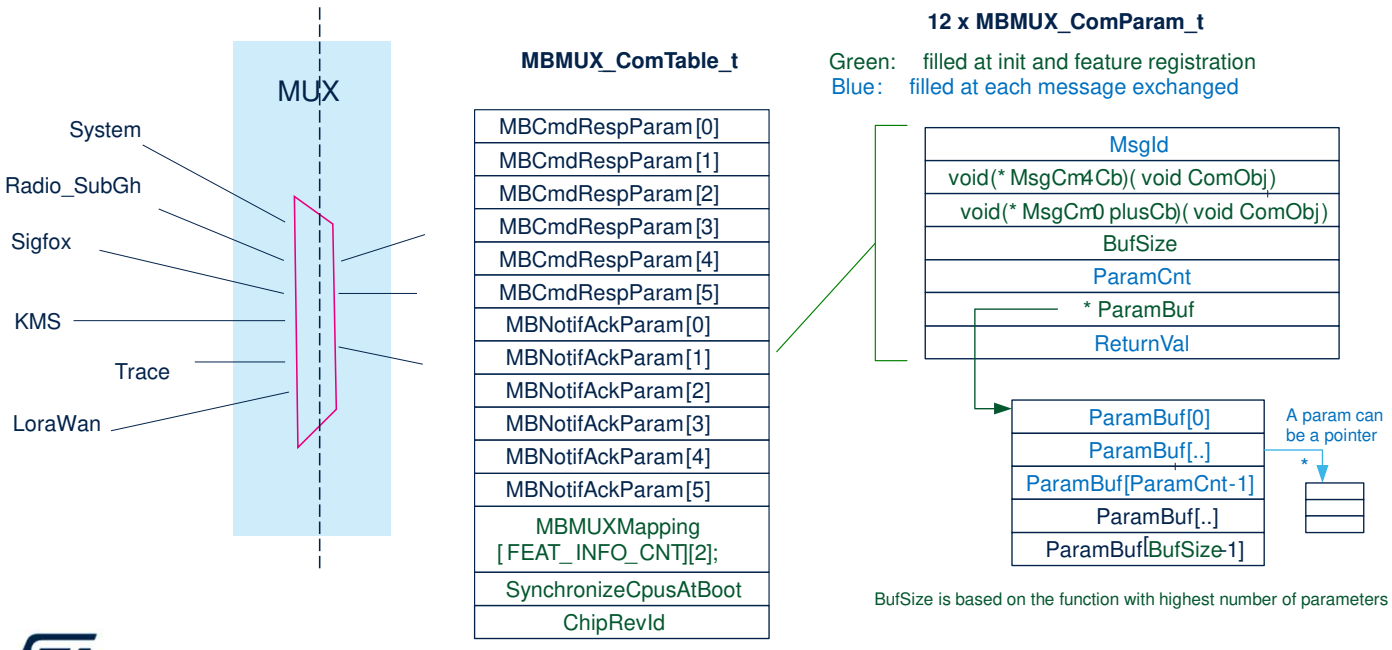
```
typedef struct  {
  MBMUX_ComParam_t        MBCmdRespParam[IPCC_CHANNEL_NUMBER];
  MBMUX_ComParam_t        MBNotifAckParam[IPCC_CHANNEL_NUMBER];
  uint8_t                 MBMUXMapping[FEAT_INFO_CNT][2];
  __IO uint16_t           SynchronizeCpusAtBoot;
  uint16_t                ChipRevId;
} MBMUX_ComTable_t;
```

life.augmented

16

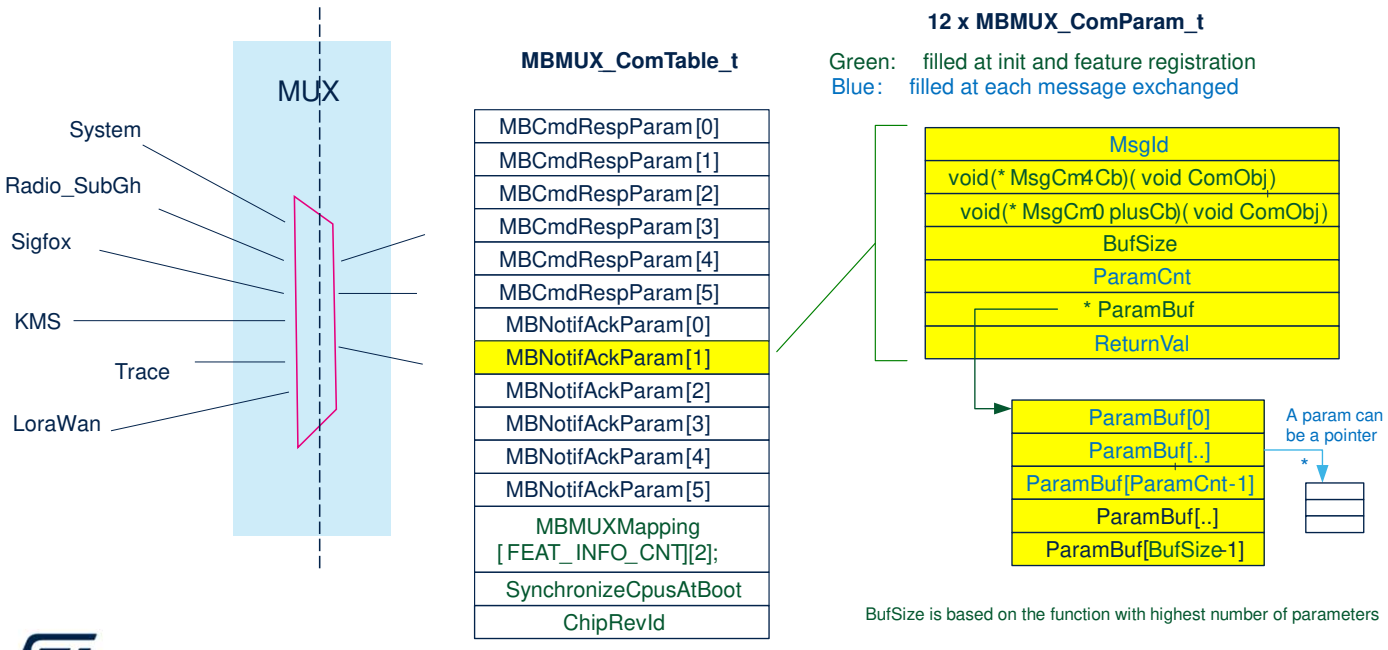
Each IPCC channel has its dedicated instance of communication structure, which is similar for all IPCC channels (TX and RX).
So MBMUX_ComParam_t seen in previous slide is duplicated 12 times.
Let's see the picture in the next slide, which is more intuitive, and then the C code.

Communication table (1/5)



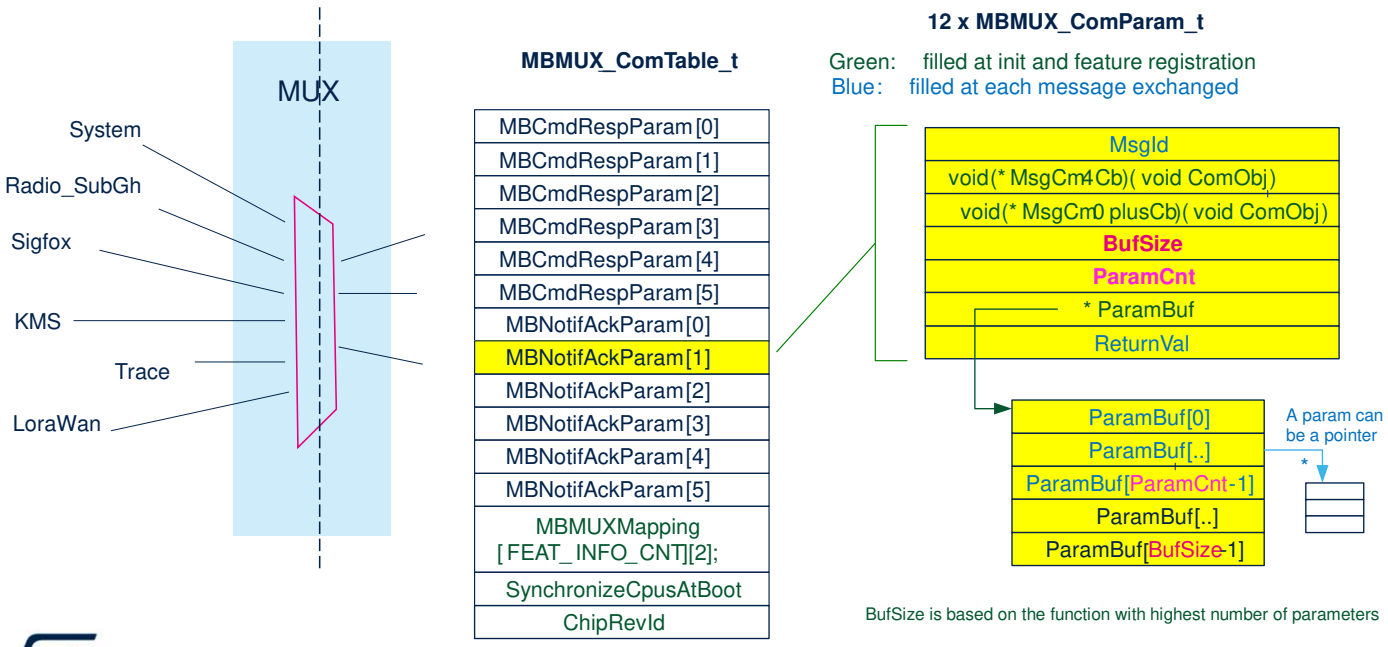
Now we can visualize that the MBMUX_ComParameter_t is instantiated 12 times within the MBMUX_ComTable_t
Once a feature requests to register, the available channels are assigned to it, and the registration is stored in the MBMUX mapping table.

Communication table (2/5)



For example, let's suppose that Trace feature asks for registration and MBNotifAckParam[1] is assigned to it. Trace feature only needs a RX channel, so no need to register an MBCmdRespParam instance, because only CM0PLUS calls trace functions on CM4 and not the other way. Each trace functions call between the two cores have a MsgID. The size of Trace ParamBuf is calculated according to the Trace function that has the bigger number of parameters. Would the function with the highest number of parameters have 5 parameters, then the BufSize parameter would be set to 5.

Communication table (3/5)



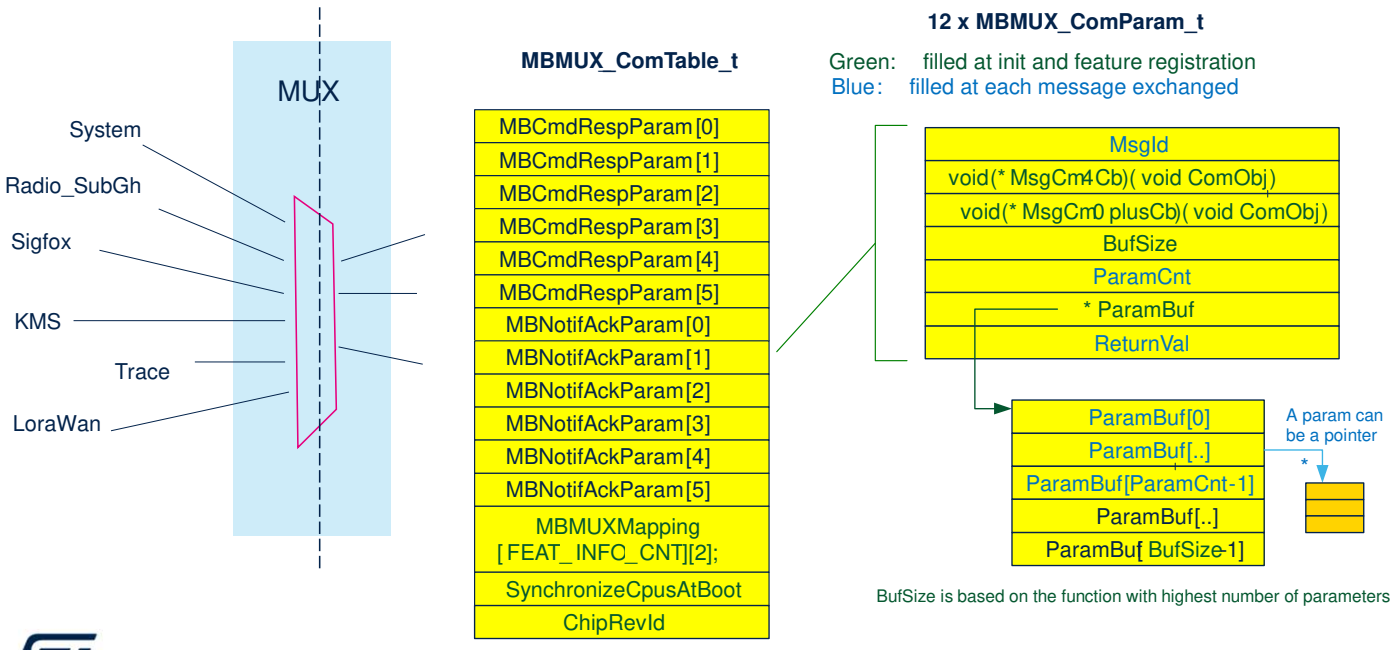
When a feature registers to the MbMux, it reserves the IPCC channel and the associated buffer. The buffer size is decided by the feature, typically it sizes the buffer based on the feature function with the highest number of parameters.

For example: the “Trace ParamBuf” is sized according to the “Trace function” that has the bigger number of parameters.

When calling a Trace function with only 3 parameters, the Trace ParamBuf is partially filled and ParamCnt is set to 3 during that function execution.

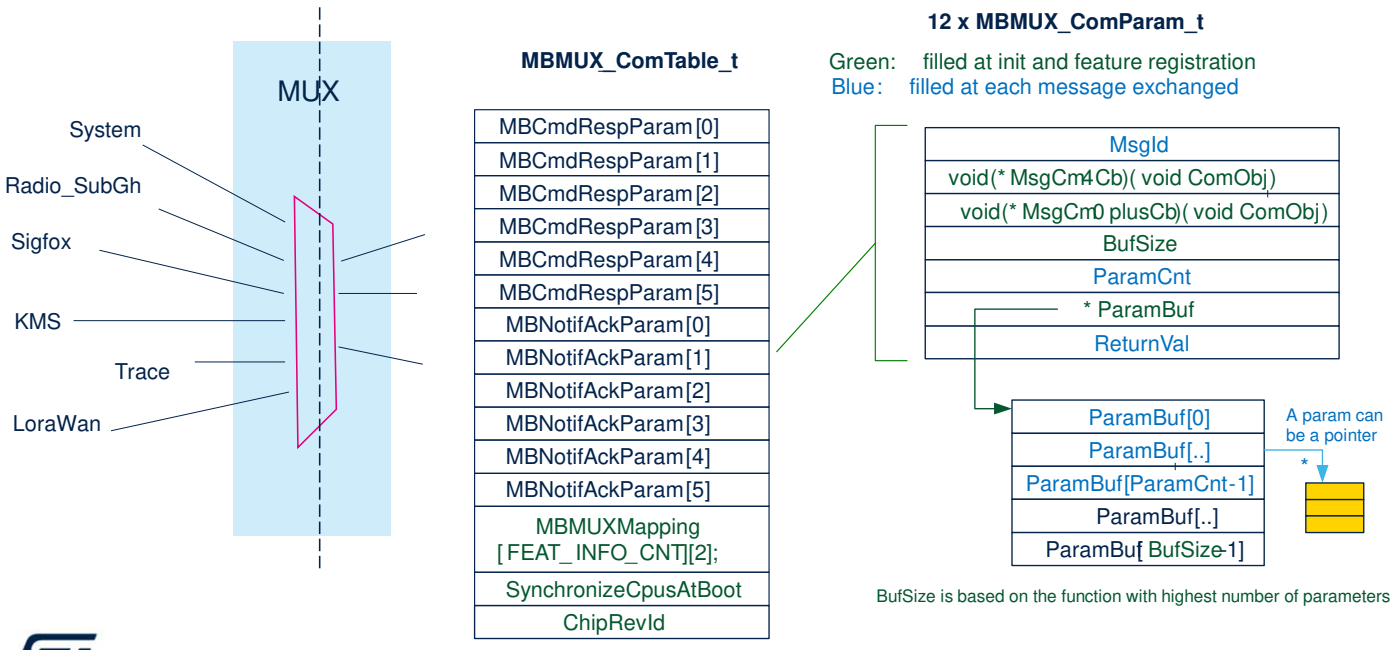
The ParamBuf is composed of int32_t fields being either values, pointer to structures or pointer to data buffers.

Communication table (4/5)



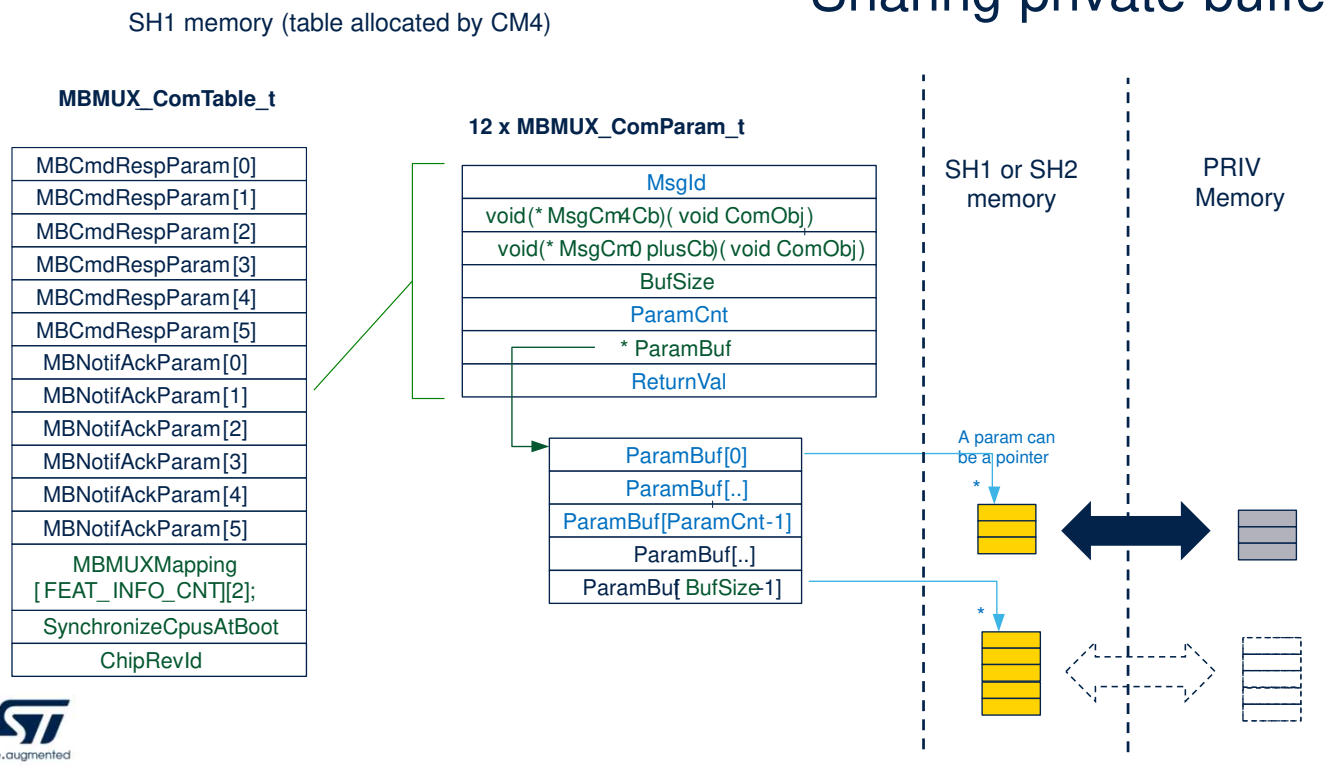
All the buffers that contain data exchanged or accessed by the two CPUs shall be in SHARED memory (Mailbox buffers, ParamsBuffers and buffers where the parameters point).
The entire communication table and the twelve potential ParamBuf are allocated by CM4.
The buffers pointed by the parameter buffer arguments might be allocated either by CM4 or CM0Plus.

Communication table (5/5)



Back to the trace example: The log message is a buffer of several bytes that is pointed by the parameter buffer arguments. For the trace feature, this buffer is allocated by CM0PLUS.

Sharing private buffers



User should take particular care concerning the buffers where the function parameter pointer points to the “memory stack” or variables/buffers in the private memory area.
In these cases, it is up to the “feature functions wrapper” make a temporary copy of the “private variable/buffer” to share memory before the other CPUs processes the function, and to retrieve the buffer back in private memory in case the other core function has modified it.

Execute functions on SingleCore device

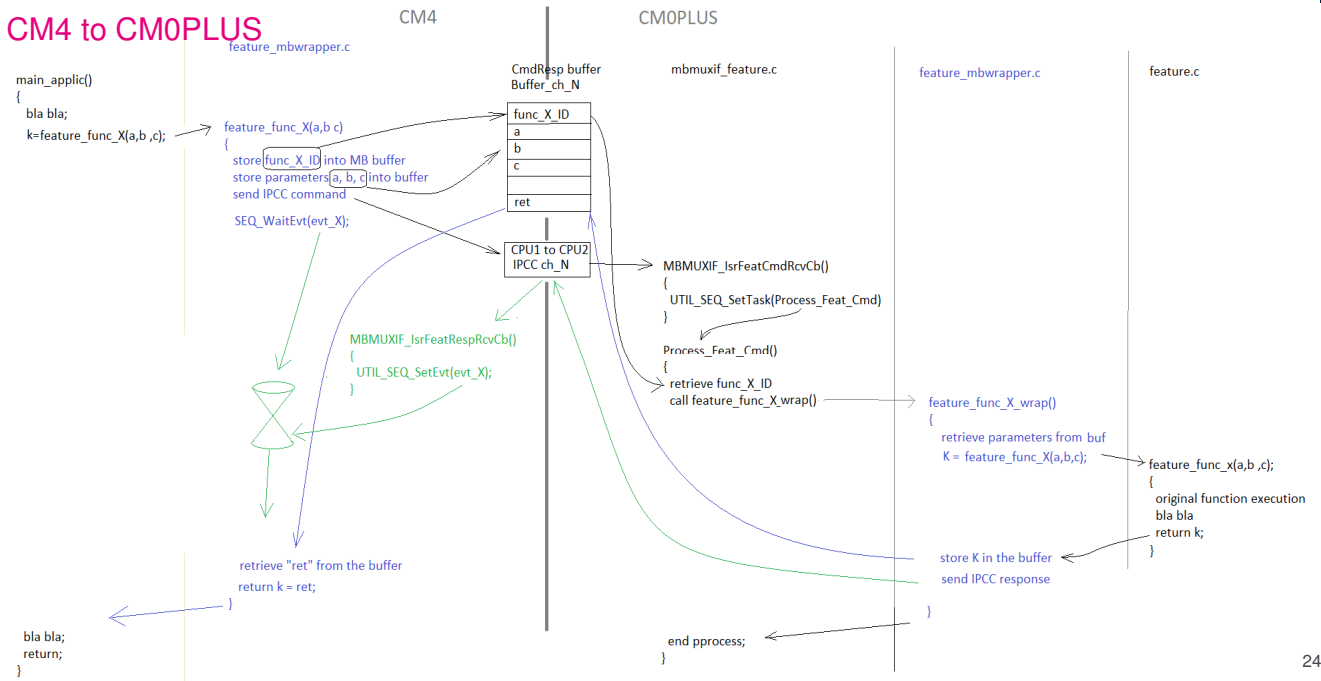
CM4

life.augmented

23

In a single core context, all functions are on the same core.
If the program needs to execute a function “X” belonging to a given feature, it just calls that function by passing its a,b,c parameters.
There is no need to use a mailbox.

Mailbox to allow one CPU to execute functions on the other CPU



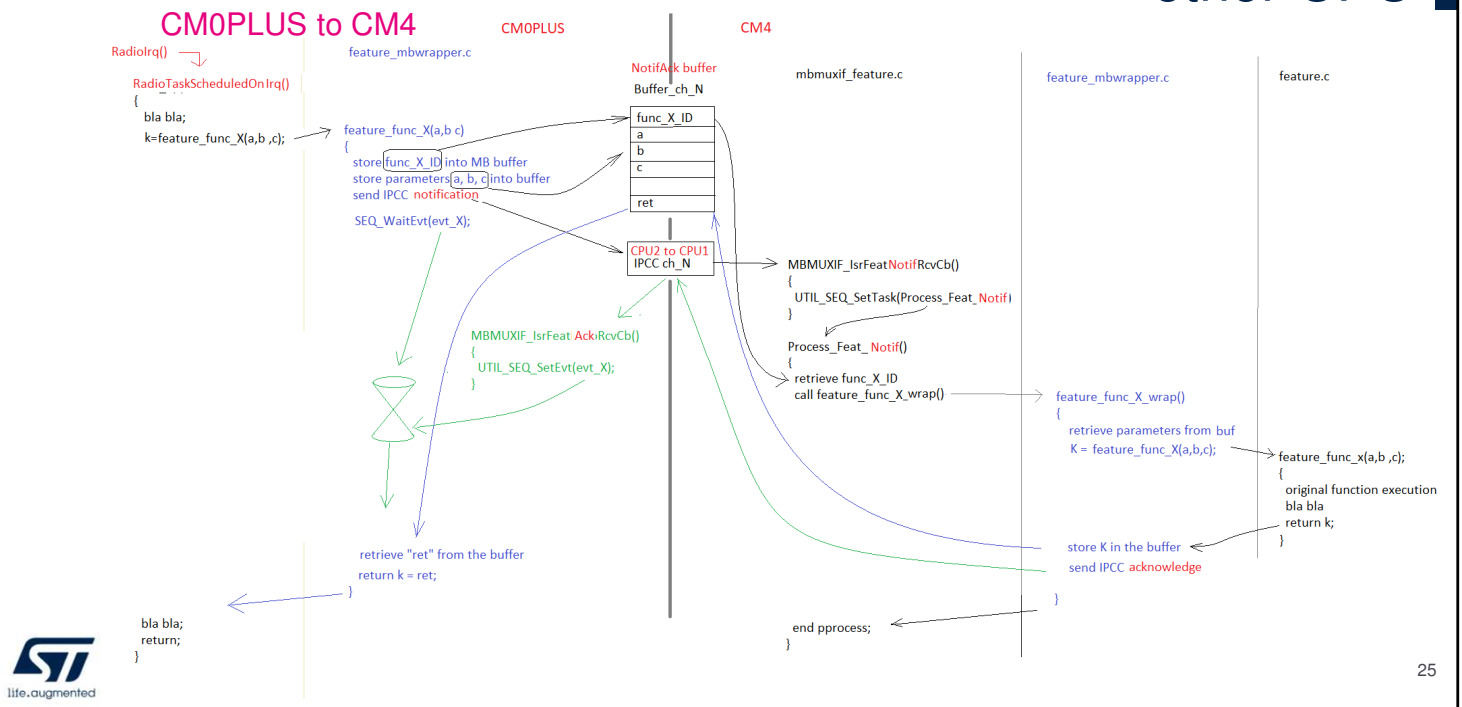
Let's consider the CM4 core needs to call a function on the CM0PLUS core.
The function belongs to a feature, which has been registered on a channel, and it has an associated CmdRespParam buffer.
A wrapper is implemented for each feature and each function has its own wrapper function with the name of the function itself.
The wrapper fills the CmdRespParam buffer, by setting

- the MsgId,
- the ParamCnt
- and the parameters of that function "X", let's call them a,b,c.

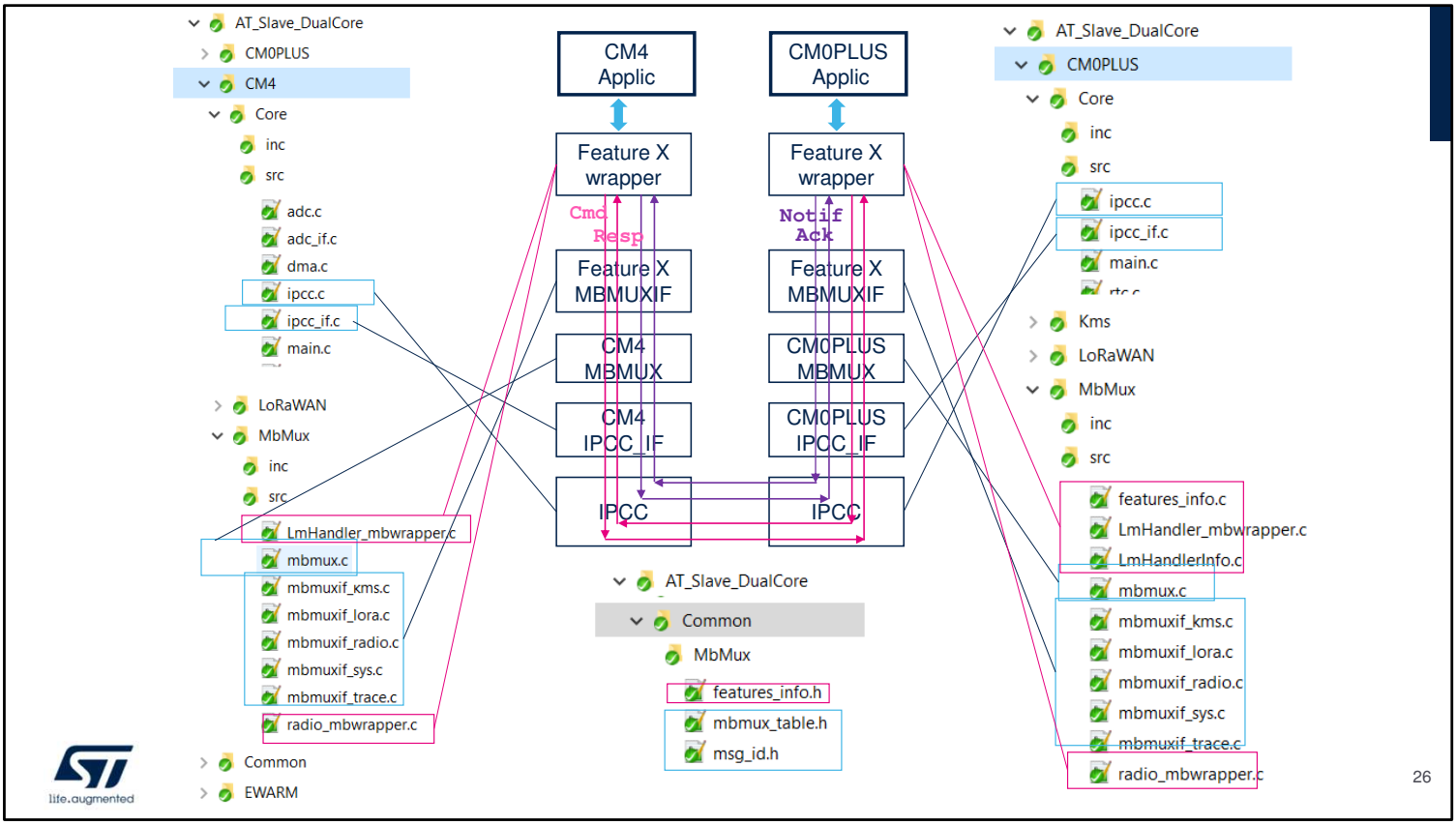
The CM4 core triggers an interrupt to the CM0PLUS core via IPCC and waits for its response. Note the call is blocking.
CM0PLUS proceeds with the IPCC interrupt: It calls an unwrapper function that decodes the MsgId, extracts the parameters, and finally calls the actual function "X".

At last, function “X” is executed by CM0PLUS which puts the return value in the CmdRespParam buffer.
CM0PLUS can now send the response (i.e. clearing the IPCC flag) back to CM4.
CM4 receives the related interrupt that wakes it up from the wait state, retrieves the return value from the CmdRespParam buffer, and then exits from the feature_func_x() function.
With such kind of “blocking call” strategy, we assure the same sequence of operation between a single core and a dual core execution.

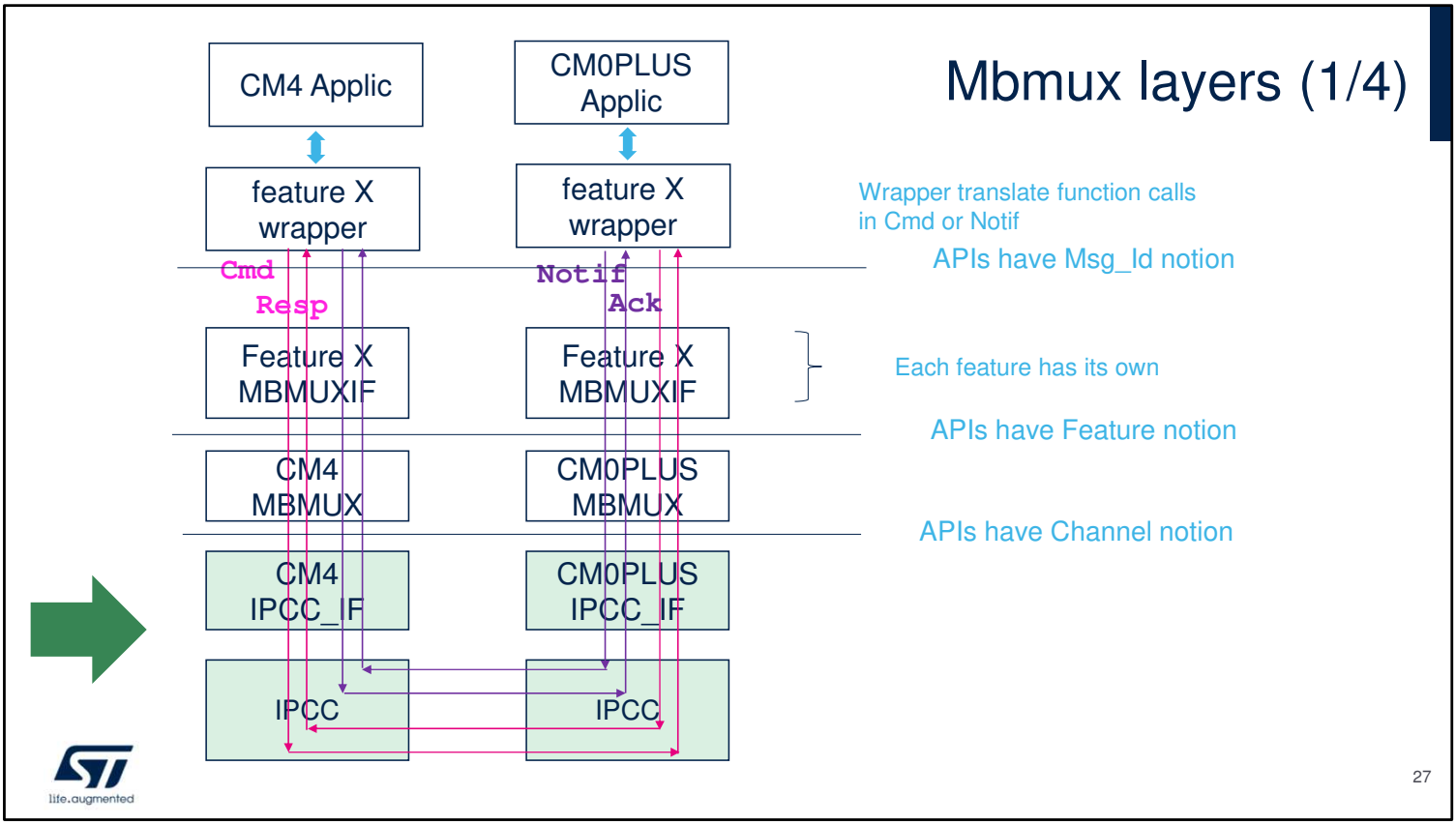
Mailbox to allow one CPU to execute functions on the other CPU



When the CM0PLUS core needs to execute a function on the CM4 core, the procedure is perfectly symmetric.
But instead of filling the CmdRespParam buffer, it uses a NotifAckParam buffer.
The naming of the function involved is also adapted: Cmd vs Notif and Resp vs Ack.

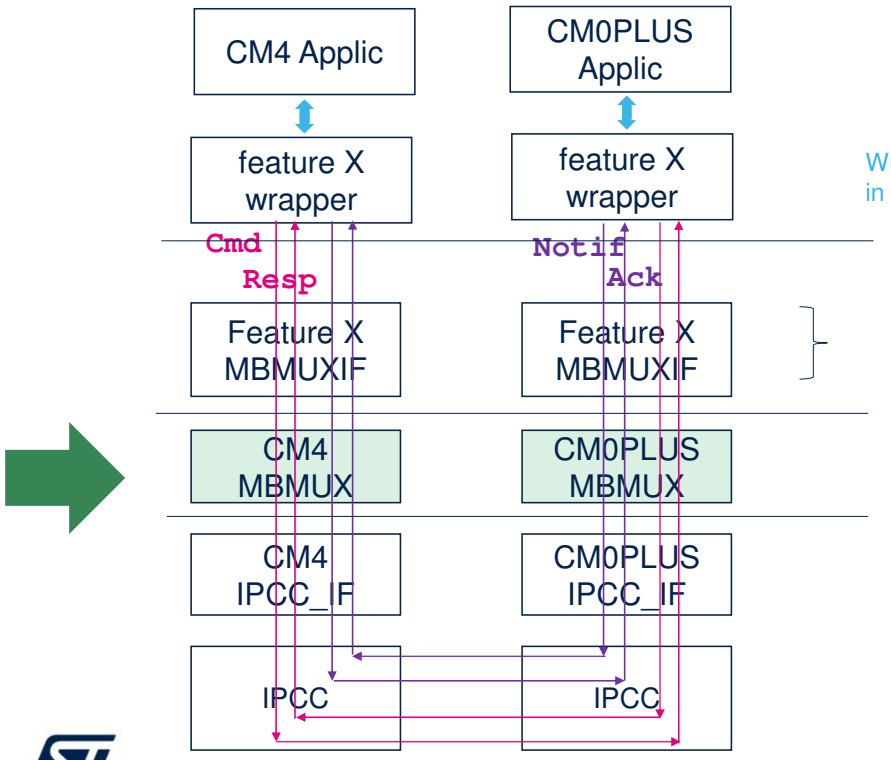


The next slides help to understand how to match the concepts discussed so far with code, files and layers.
This figure exposes the directory tree of the LoraWAN AT_Slave project taken as example.
On the left side, the files building the part of the project running on the CM4 side and on the right side, the files constituting the CM0PLUS part of the project..
MbMux is part of the application, but it has its own directory MbMux.
Files are duplicated for each application (on the contrary of middleware, utilities, and drivers which are transversal to all applications/examples).
MbMux is implemented in several layers. In order to understand the code, let's talk about these layers.



MbMux is divided in three layers. The bottom of the picture shows the IPCC which is basically the physical layer. IPCC is part of the HAL drivers. IPCC_IF is the interface layer (as for each IP). These interfaces for both CM4 and CM0PLUS cores are not part of MbMux.

Mbmux layers (2/4)



Wrapper translate function calls in Cmd or Notif

APIs have Msg_Id notion

Each feature has its own

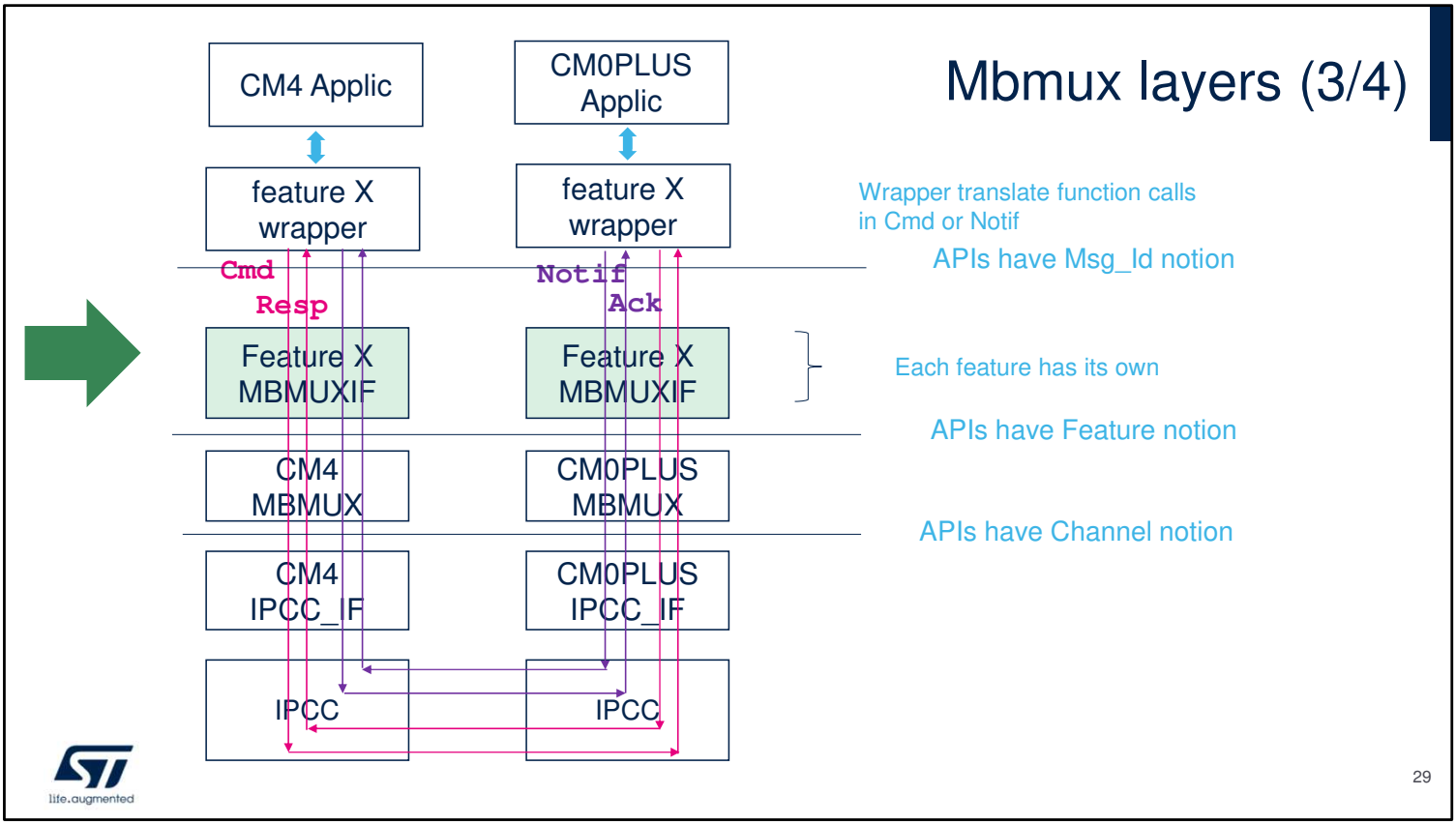
APIs have Feature notion

APIs have Channel notion

life.augmented

28

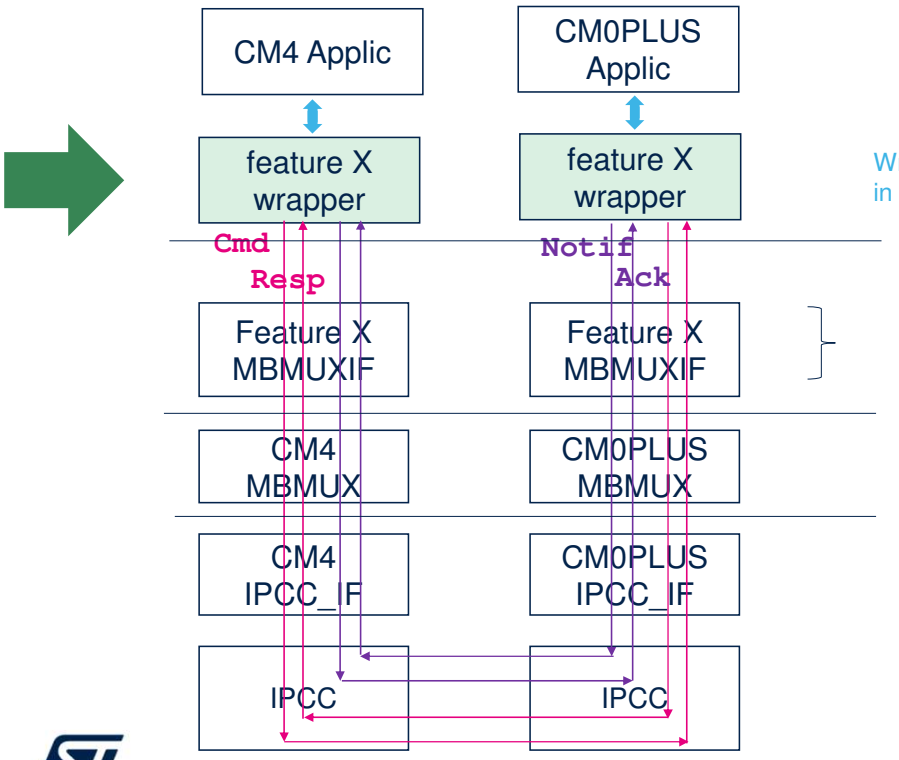
The MbMux lower layer resides just above IPCC and its task consists in mapping features on channels, i.e. it provides registration and muxing.
IPCC_IF and MbMux lower layer are rather generic services (the user is never supposed to modify them).



Each feature interfaces to MbMux with its own MBMUXIF. In order to add a new feature, the correspondent MBMUXIF needs to be implemented.
And existing MBMUXIF might require to be adapted if a given feature needs customization for parameter buffer size or callback reactivity i.e. task vs interrupt service routines.
MBMUXIF is in charge of:

- Allocating feature dedicated buffers for CM4-CM0PLUS communication
- Initiating the feature registration on MBMUX (com buffers registration, callbacks registration)
- Abstracting upper layer from the feature_ID and the Mailbox direction
- Implementing message ID switch
- Handling the sequencer (kind of “kernel/scheduler” presented in other documentation).

Mbmux layers (4/4)



Wrapper translate function calls in Cmd or Notif

APIs have Msg_Id notion

Each feature has its own

APIs have Feature notion

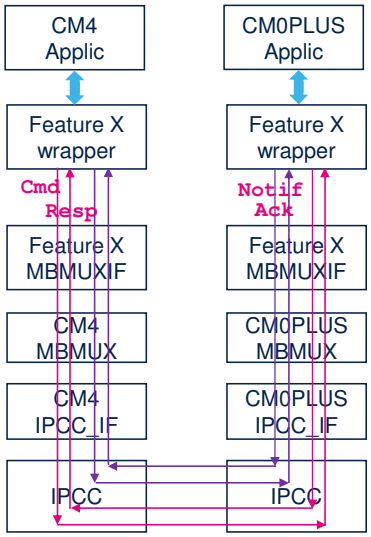
APIs have Channel notion

life.augmented

The feature wrapper translates functions to messages.

Specularity

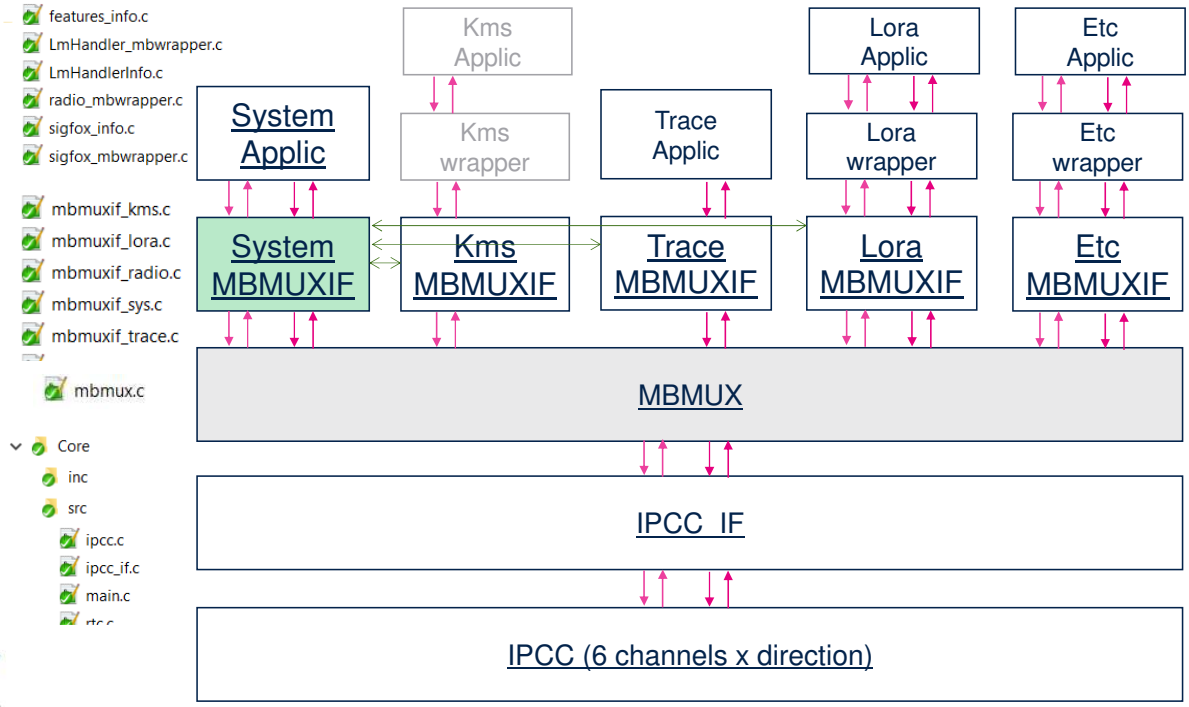
- CM4 IPCC_IF is specular to CM0PLUS IPCC_IF
  - CM4 IPCC_IF provides API for sending Cmd and Ack and provides callbacks to handle Notif and Resp
  - CM0PLUS IPCC_IF provides API for sending Notif and Resp and provides callbacks to handle Cmd and Ack
- CM4 MBMUX is specular to CM0PLUS MBMUX
  - CM4 MBMUX provides API for sending Cmd and Ack and provides callbacks to handle Notif and Resp
  - CM0PLUS MBMUX provides API for sending Notif and Resp and provides callbacks to handle Cmd and Ack
- CM4 MBMUXIF is specular to CM0PLUS MBMUXIF
  - CM4 MBMUXIF provides API for sending Cmd and Ack and provides callbacks to handle Notif and Resp
  - CM0PLUS MBMUXIF provides API for sending Notif and Resp and provides callbacks to handle Cmd and Ack



life.augmented

The blocks on CM4 core are specular to the one on CM0PLUS core, but not the same.
So CM4/Mbmux/mbmux.c is not the same code as CM0PLUS/Mbmux/mbmux.c
If a CM4 MBMUX file sends commands and acknowledgements and provides callbacks to handle notifications and responses, its specular file on CM0PLUS side sends notifications and responses and provides callbacks to handle commands acknowledgments.

Global stack on one CPU (mirrored on other CPU)



This is an expanded view of one of the two columns described in previous picture.
The MBMUXIF blocks are independent between features except “System MBMUXIF” which interacts with all other MBMUXIFs.
MBMUX + System MBMUXIF are the core of the mailbox.
Notice: Missing Notif/Ack arrows for KMS and missing Cmd/Resp arrows for TRACE is not a mistake: these features are unidirectional:

- KMS is called from CM4 core to ask CM0PLUS core for encryption, no callback is required.
- TRACE is used by CM0PLUS core to send CM4 core (via IPCC) strings to be printed.

Kms wrapper is not implemented. In SubGhz examples, KMS is only called by the SubGhz stacks which reside as well on CM0PLUS core.
To call KMS API (on CM0PLUS core) from CM4, a specific wrapper must be implemented.

On the left side of the slide is a print shot of the tree files that should help navigate the code.

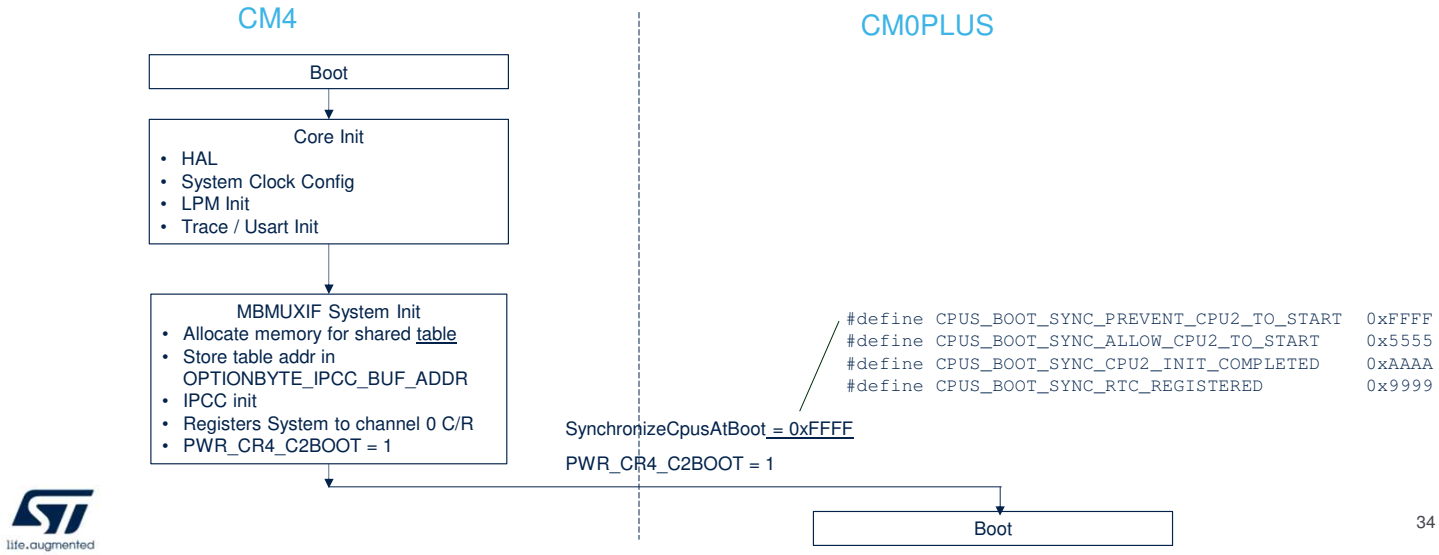
Initialization sequence and feature information table

life.augmented

Next slides give a timeline sequence describing how MBMUX is initialized.
This is also helpful for whom needs to modify the code.

Dual core communication initialization (1/2)

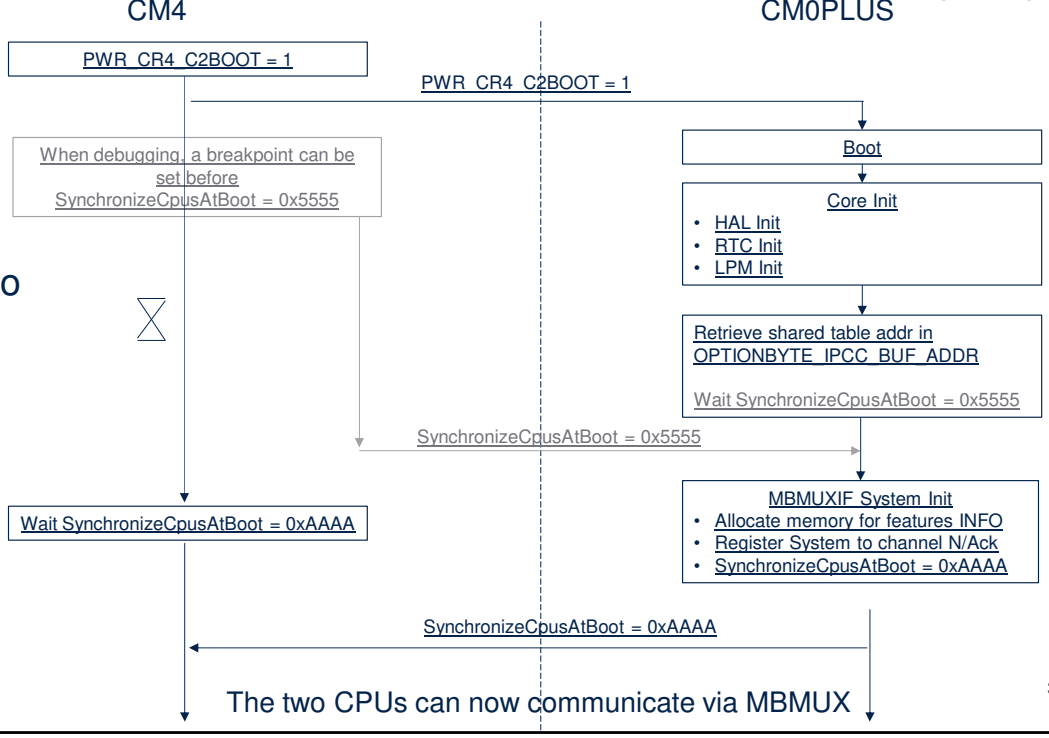
- Init sequence
  - CM4 set address of the shared memory (option byte)
  - CM4 boots CM0PLUS



In SubGhz examples, the first core to boot is always the CM4 core, the CM0PLUS core is hold via the C2BOOT bit part of the PWR_CR4 register.
Before releasing the CM0PLUS core, the CM4 core initializes the MBMUXIF System blocks such as MBMUX table and communication buffers.
More details about initializing the OPTIONBYTE_IPCC_BUF_ADDR at the end of this presentation.

Dual core communication initialization (2/2)

- From now on the two CPUs will communicate via MBMUX

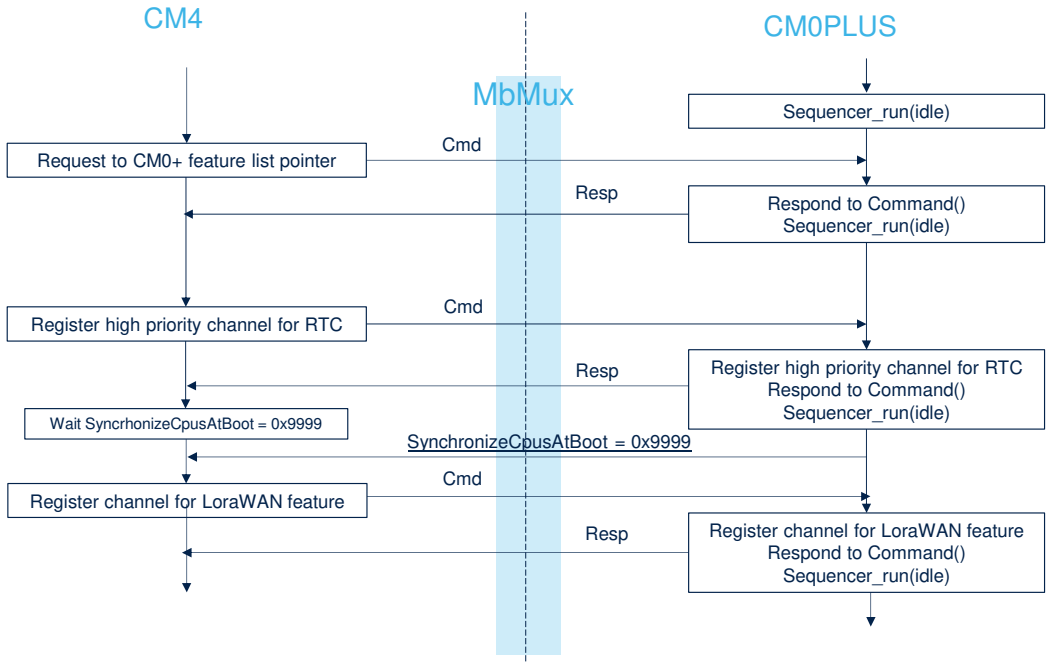


life.augmented

35

The SynchronizeCpusAtBoot flag is initialized at 0xFFFF and is used by the CM4 core to know when the CM0PLUS core completes the MBMUX initialization (i.e. its value becomes 0xAAAA).
Optionally this flag can be used for debugging to hold the CM0PLUS core from executing MBMUX_System_Init() function. Without putting a breakpoint on CM4 this is transparent.
Once the MBMUXIF System initialization is done, the two CPUs can communicate via MBMUX.

Initialization goes on via MBMUX



life.augmented

The CM4 core is always the master of the registration for any feature.
Before registering a feature, the CM4 core shall know if the binary firmware downloaded on the CM0PLUS core supports that feature.
Assuming the CM0PLUS binary supports only the Sigfox protocol, the MbMux needs a means to detect that LoraWAN protocol cannot be registered.
This is possible thanks to the FEATURE_INFO_LIST table, more details to come.
Another system brick that needs to be installed is a time base for the timeserver. In a subghz example, this time base is provided by the real Time Controller (RTC).
An IPCC channel is dedicated to RTC as both cores need a time base and, from a software architecture design choice, RTC interrupts are only handled by the CM0PLUS core, which then forward the RCC interrupts to the CM4 core via IPCC.

When the SynchronizeCpusAtBoot flag becomes 0x9999, the RTC channel is also registered.
From now on, the core of the MBMUX framework is initialized. Any other feature can be registered (Trace, LoraWAN, Sigfox, etc).

Capabilities information exchange

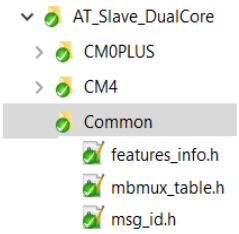
- As said at the beginning of the presentation, CM4 and CM0PLUS have different binaries.
- How CM4 knows
  - which radio stacks are implemented on Cm0plus? (Lorawan? Sigfox? Other?)
  - which version of each stack/feature?
  - How is a given stack/feature configured (e.g. LoraMAC supports all regions, all classes)?
- CM0PLUS provides this information via a table named FEAT_INFO_List_t.
- This table is allocated by CM0PLUS in SRAM2_SH2 memory
- CM4 asks for the address of this table via a service offered by the System feature (as shown in the top left corner of the previous slide)

life.augmented

CM4 and CM0PLUS cores have different binaries.
CM4 core needs to know which radio stacks are implemented on the CM0PLUS core (LoraWan? Sigfox? Other?), which version and configuration of each stack/feature?
CM0PLUS core provides this information via a table named FEAT_INFO_List_t. This table is allocated by the CM0PLUS core in the SRAM2_SH2 memory.
CM4 core gets the address of this table via a service offered by the System feature.

Feat_info_list_t

- The header files that defines the structure are common to both CPUs
- The structure of the list shall not change between versions
- The parameters of a given feature might change between versions.




```
typedef struct{
  FEAT_INFO_IdTypeDef Feat_Info_Feature_Id;
  uint32_t Feat_Info_Feature_Version;
  uint32_t Feat_Info_Config_Size;
  void *Feat_Info_Config_Ptr;
} FEAT_INFO_Param_t;

typedef struct{
  uint32_t Feat_Info_Cnt;
  FEAT_INFO_Param_t *Feat_Info_TableAddress;
} FEAT_INFO_List_t;
```

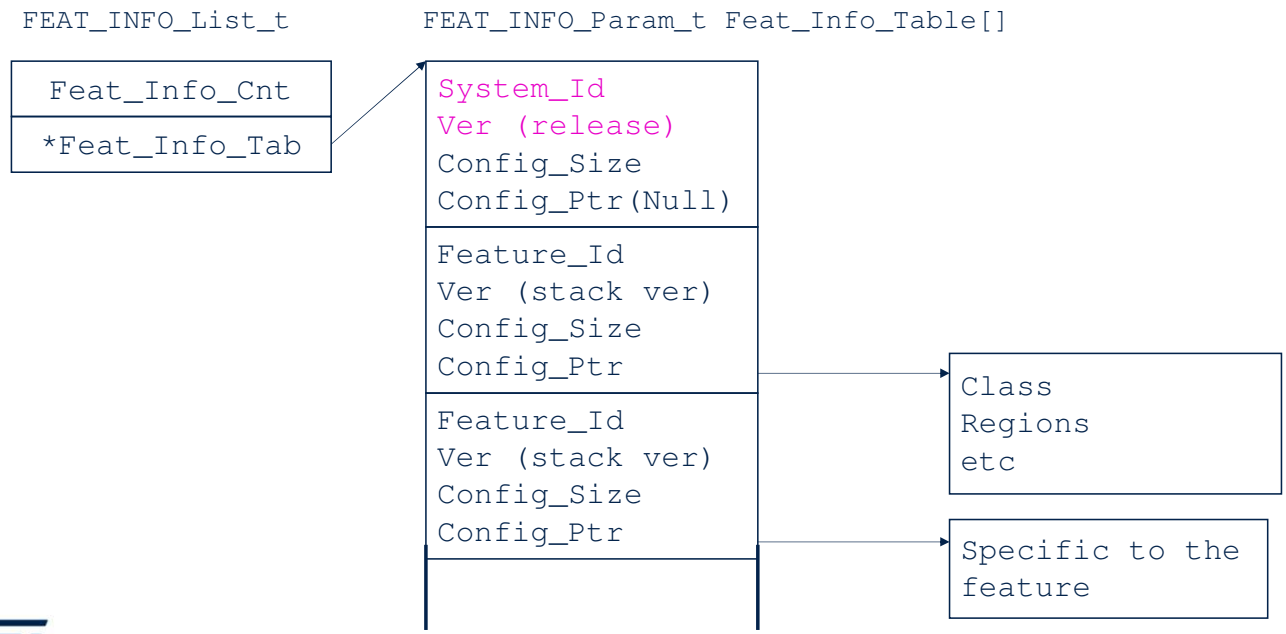
```
FEAT_INFO_Param_t Feat_Info_Table[] =
{
  {
    .Feat_Info_Feature_Id      = FEAT_INFO_SYSTEM_ID,
    .Feat_Info_Feature_Version = FEAT_INFO_SYSTEM_VER,
    .Feat_Info_Config_Size     = 0,
    .Feat_Info_Config_Ptr = (void *) NULL
  },
  {
    .Feat_Info_Feature_Id      = FEAT_INFO_LORAWAN_ID,
    .Feat_Info_Feature_Version = __LORA_CM0_APP_VERSION,
    .Feat_Info_Config_Size     = sizeof(LoraInfo_t),
    .Feat_Info_Config_Ptr = (void *) loraInfo
  },
  {
    .Feat_Info_Feature_Id      = FEAT_INFO_RADIO_ID,
    .Feat_Info_Feature_Version = __SUBGH_MW_VERSION,
    .Feat_Info_Config_Size     = 0,
    .Feat_Info_Config_Ptr = (void *) NULL
  }, ...
```

life.augmented

38

Here are some code details on how CM0PLUS binary exposes its capability.
The FEAT_INFO_List_t has variable size, the size depends on the number of features, which is set as parameter: Feat_Info_Cnt.
This structure and the FEAT_INFO_Param_t type should not change between revisions.
The FEAT_INFO_Param_t type contains the description of each feature such as its ID, its version, the size of its configuration table, and a pointer to such table. The config table related to each feature has not a predefined structure. Each feature can implement its own. As example for LoraWAN feature, the config table is named LoraInfo_t and contains the supported regions, classes, mode (OTAA, ABP) and KMS support.

Feat_info_list_t graphically



life.augmented

39

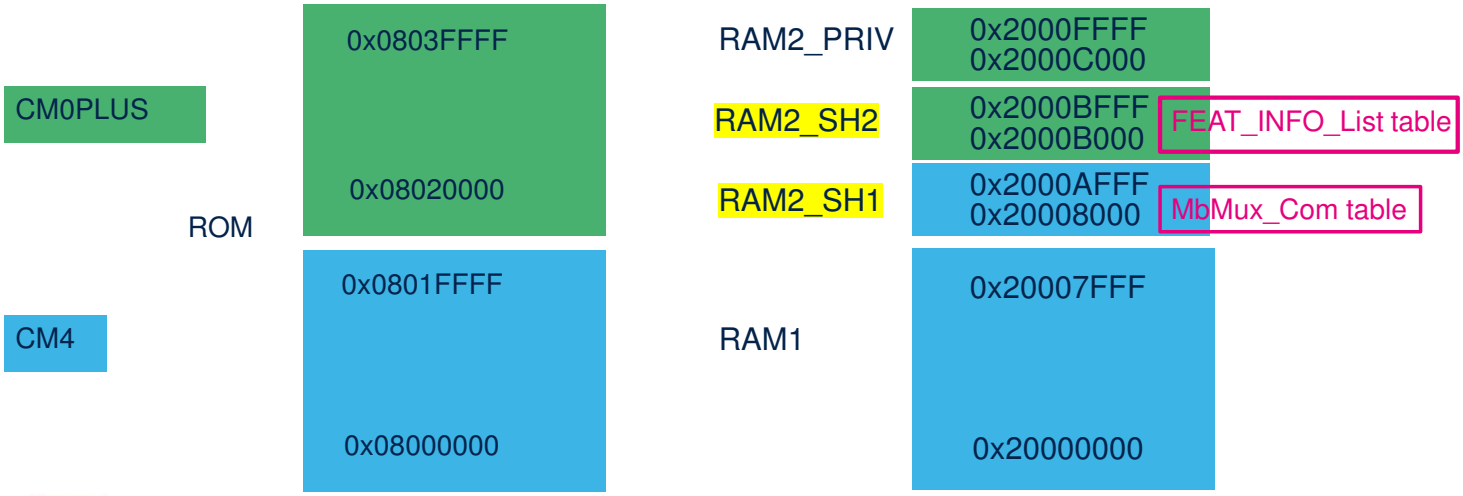
Here a graphical view that describes the code previously exposed.
The first feature in the table is the system feature; it tells the version of the global release (e.g. v1.0.0) and it has no Config table (Null).
The other tab depends on the binary; only the features supported by that CM0PLUS binary are part of the table.
LoraWAN_AT_Slave project includes LoraWan, Radio and Trace features. For each feature, the stack version is given as well as a pointer to the corresponding configuration table.

Shared memory and IPCCDBA Option byte

The following appendix is especially useful for user that don’t use the linker files provided by the FW packs. In fact their code will enter in an infinite loop as protection.

Memory mapping

- Example of linker file for EndNode in v1.0.0 release



life.augmented

41

To recapitulate, the MbMux_ComTable is allocated by the CM4 core and placed in SRAM2_SH1 memory (CM4 linker file).
Feat_Info_List_Table is allocated by the CM0PLUS core and put in SRAM2_SH2 memory (CM0PLUS linker file).
In order to communicate, the two cores need to know the addresses of these tables.
CM0PLUS retrieves the address of MbMux_ComTable via the IPCCDBA option byte.
CM4 retrieves the address of Feat_Info_List_Table via MbMux communication.

IPCCDBA option byte

- In “Subghz examples” the CM4 linker file places the MbMux_ComTable at 0x20008000
- CM0PLUS retrieves the MbMux_ComTable address (placed by CM4) as follow:
  - OptionsBytesStruct.IPCCdataBufAddr = SRAM1_BASE + IPCCDBA <<4;
- By default, on STM32WL devices the option byte IPCCDBA is 0x800
  - So OptionsBytesStruct.IPCCdataBufAddr = 0x20008000 matches the Cm4 linker file
- What’s happen if the linker file places the MbMux_ComTable somewhere else?
- During initialization, CM4 reads the IPCCDBA value and compare it with the MbMux_ComTable address from the linker file. If the values are not compatible, CM4 overwrites the IPCCDBA option byte and reboots the board. On next run, the values will be matching, and CM0PLUS will retrieve the new address.

life.augmented

The IPCCBDA option byte register contains the address of the MbMux_ComTable.
The default register value in a new board matches the value provided in Subghz examples linker files.
Otherwise, It is expected to be updated by the CM4 core.
The CMOPLUS core uses this table to retrieve the address of the MbMux_ComTable.
This slide shows details about the calculation.

Option byte programming issue

- Actually, to allow the CM4 core to overwrite the IPCCDBA option byte, the user has to do a little modification in the code, in the CM4/mbmuxif_sys.c file.
- In fact, an infinite loop is put as protection, so currently if there is a mismatch between IPCCDBA and the linker file, the code enters this infinite loop.
- This protection is in place for two reasons:
  - On Cut 1.1 the option byte programming is buggy when using MSI system clock frequency > 16Mhz (which is the case for SubGhz applications). So before calling OBProgram and OBLaunch functions the user should patch the code to reduce MSI frequency. Problem is fixed in Cut 1.2.
  - If users forget to define MAPPING_TABLE in the linker file, the linker doesn't give warnings, it just places the MbMux_ComTable somewhere in SRAM1 and set the OB accordingly. With no protection, the program would overwrite the IPCCDBA Option Byte without even the user realize it.

life.augmented

A protection is currently in place to prevent the CM4 from changing the IPCCBDA option byte. This is for two reasons explained in the slide.
Protection will be removed in the next FW pack WL_1.1.0.

Cubemx and linker files

- When generating code with CubeMX starting from scratch, it will use the CMSIS scatter files instead of the one provided by Intropack applications
  - CMSIS default scatter file does not define MAPPING_TABLE
- This means there will be a mismatch between IPCCDBA and MbMux_ComTable, the code will enter in while(1) {} protection. Such protection reminds to users to configure the linker file before linking and running the code
- Notice that once the linker file is updated, regenerating the code with MX will not erase the updated linker file. In fact, MX uses CMSIS default scatter files only in case there is no scatter file present in the concerned IDE directory (similar principle of USER CODE SESSION)
- Protection can be removed by user who has understood the implication

life.augmented

Be aware that CUBEMX only generates GENERIC linker files that do not contain the notion of MBMUX.
so, the code generated with CUBEMX will enter the infinite loop protection trap, unless updating the linker file or modifying the code as explained in the slide.