

STM32WL MBMUX メールボックス・マルチプレクサ

レビジョン 1.0

STM32WLのメールボックス・マルチプレクサのプレゼンテーションへようこそ

- MBMUX の目的は 2 つの STM32WL コアである CM4 (CPU1) と CM0PLUS (CPU2) 間の通信の容易化
- MBMUX は、デュアル・コア動作の STM32WL デバイス上で実行する Sub-GHz のサンプルとアプリケーションのニーズに特化したソフトウェア・フレームワーク
- すべての種類のデバイスとアプリケーションで一般的なミドルウェアは対象外
- MBMUX の設計は簡潔性と柔軟性とのトレードオフからの帰結

MBMUX の目的は 2 つの STM32WL コアである CM4 (CPU1) と CM0PLUS (CPU2) 間の通信を容易にすることにあります。

MBMUX は、デュアル・コア動作の STM32WL デバイス上で実行する SubGHz のサンプルとアプリケーションのニーズに特化したソフトウェア・フレームワークです。

すべての種類のデバイスとアプリケーションで一般的なミドルウェアは対象外です。

MBMUX の設計は簡潔性と柔軟性とのトレードオフから得られています。

シングル・コアとデュアル・コア

- シングル・コアとデュアル・コアの相違点

- 2つの独立したバイナリ
 - →共通のリンクなし
- 通信を可能にするように2つのコア間のインタフェースを構築(プロセッサ間通信コントローラ(IPCC)と共有メモリ(SH-mem))
- コア間で共通の言語を確立
 - 構造体型の共通化、交換する構造体へのポインタなど

- 影響

- 2つのファームウェア(CM4 と CM0PLUS)に相互の互換性が必要
- CM0PLUS バイナリの機能(サポート対象機能)を CM4 で認識することが必要
 - 異なるコンパイル・オプションを持つ同じリリース・バージョンのビルドが可能

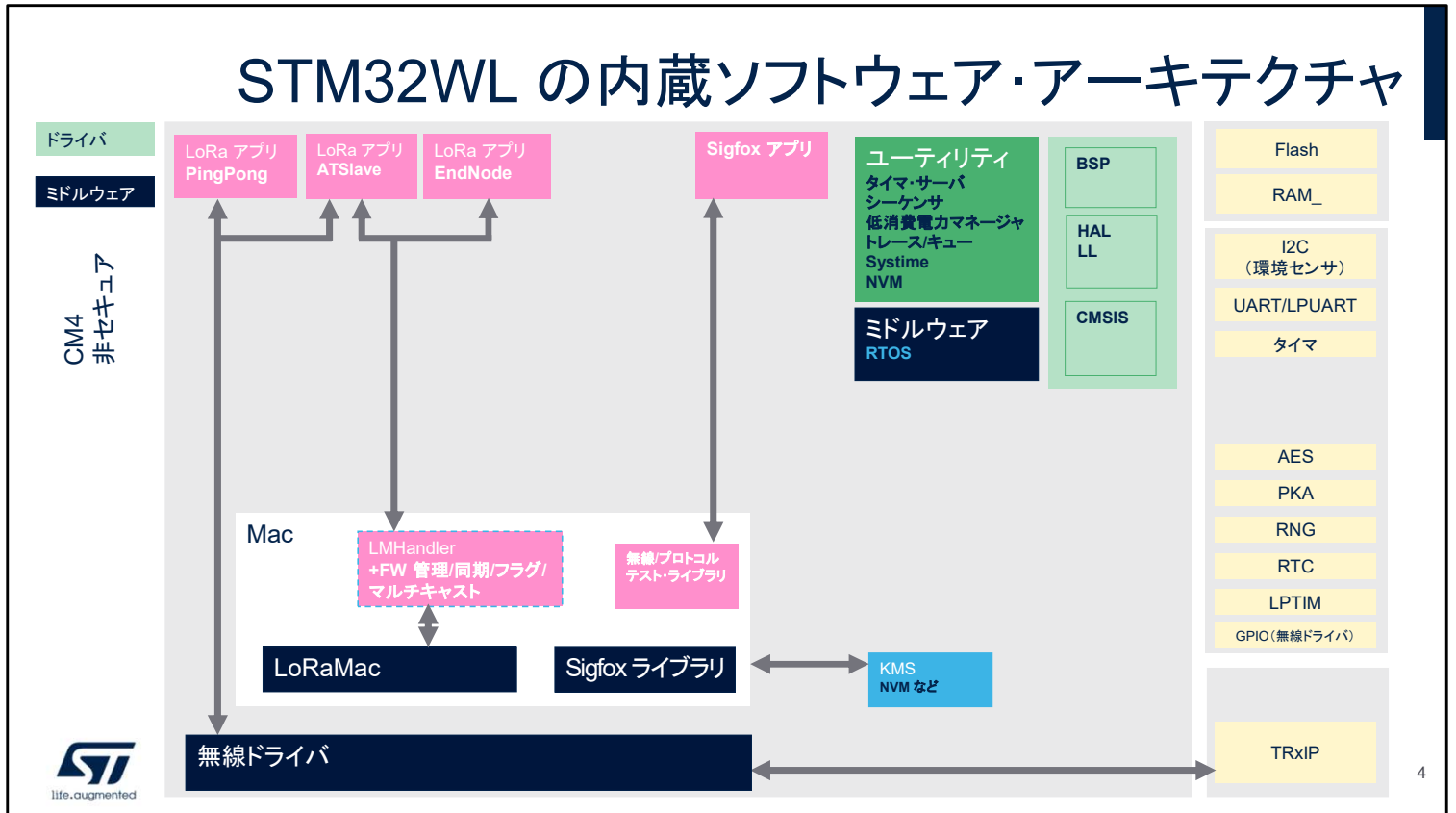


デュアル・コア環境にこのようなフレームワークが必要な理由は何でしょうか。それは、2つのコアが全面的に独立していて、2つのバイナリで同じリンク・ファイルを共有していないからです(FOUTA プロジェクトでの例外あり)。

MBMUX は IPCC と共有メモリを介して通信チャネルを提供し、2つのコア間で情報を交換できるようにする共通言語を確立します。

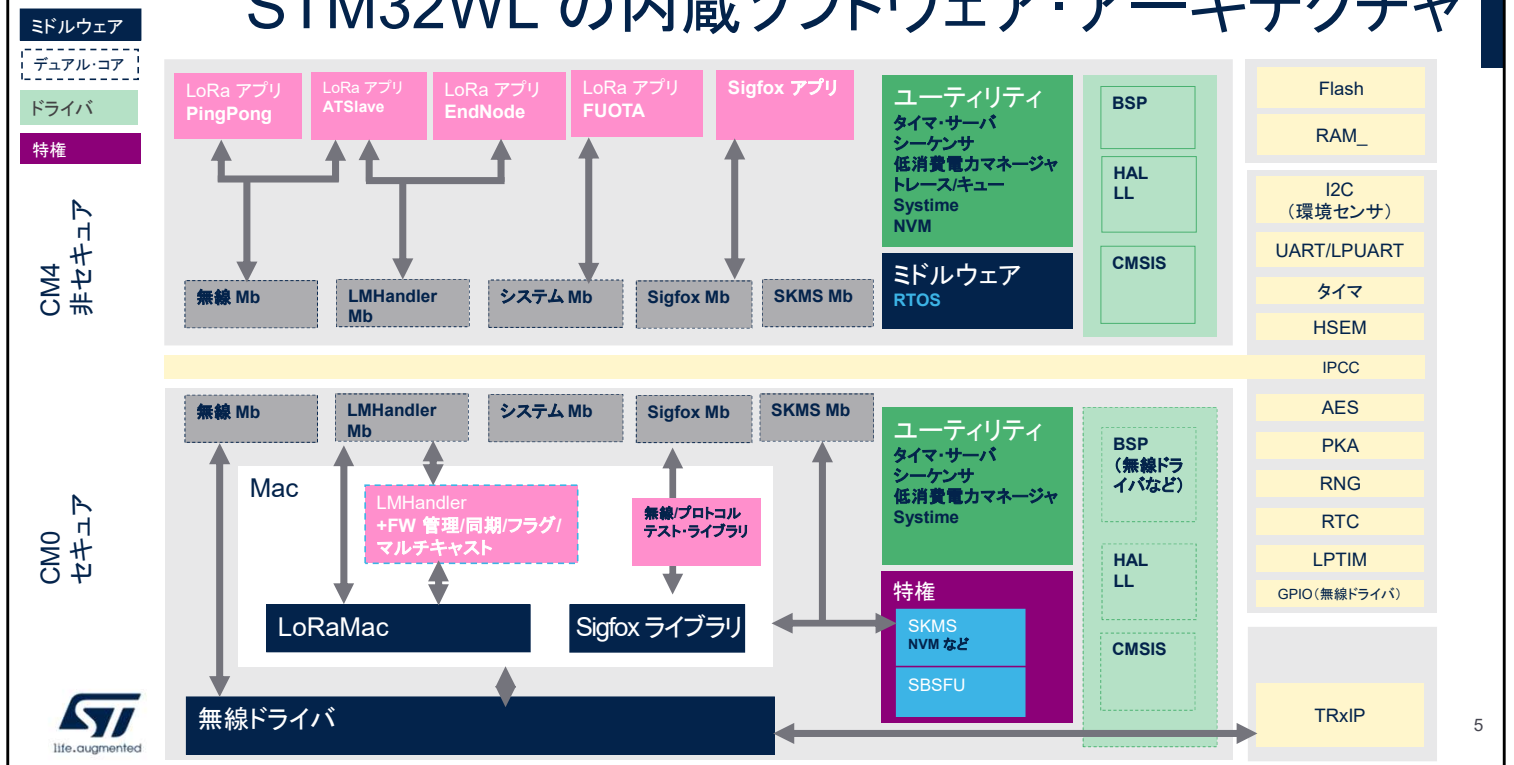
これは、2つのコアのファームウェア・バージョンに相互の互換性が必要であることを意味します。2つのコアのそれぞれには、相手コアでどの機能がサポートされているかを知るための手段が必要です(Sigfox ではなく、LoRa など)。

STM32WL の内蔵ソフトウェア・アーキテクチャ



このスライドは、シングル・コア・プロジェクトの一般的なアーキテクチャを示しています。アプリケーション (LoRaWAN または Sigfox) は、同じコアに用意されているミドルウェアと無線ドライバを使用します。

STM32WL の内蔵ソフトウェア・アーキテクチャ



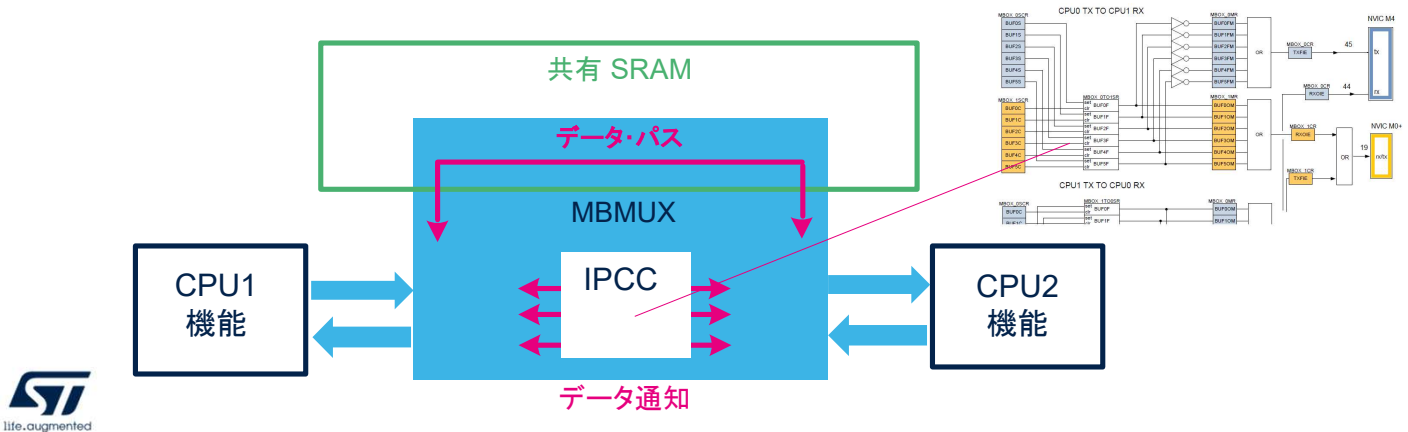
このスライドは、デュアル・コア・プロジェクトのより複雑なアーキテクチャを示しています。この例では、実用的な部分を CM4 コアに実装し、各種スタックと無線ドライバを CM0PLUS コアで実行しています。

このモデルの目的は、CM0PLUS コアでのスタックの動作に影響することなく、CM4 コア上のアプリケーションを変更できるようにすることです。

灰色のボックス(無線 Mb、Sigfox Mb など)は、MBMUX フレームワークを構成する各モジュールです。これらのモジュールは、アプリケーションとスタックまたは無線ドライバとの相互作用を実現します。

MBMUX の意味

- **MBMUX**は、IPCC チャンネルの IRQ を共有メモリ・バッファに結び付け、2 つの CPU の間でメッセージを交換できるようにするメールボックス(MailBox)レイヤ
- **MBMUX** は、2x6 の IPCC チャンネルに各種機能(LoRa、Sigfox、無線、Mwbus、KMS (SKS)、トレースなど)をマッピングできるマルチプレクサとしても機能



MBMUX は、メールボックス (Mailbox) とマルチプレクサ (Multiplexer) の略です。

メールボックスは、2 つの物理的手段である IPCC と共有メモリを介して 2 つのコア間の通信を提供します。

マルチプレクサは、各機能と IPCC チャンネル間の動的マッピングを提供します。

次のスライドでは、IPCC、メモリ・バッファ、チャンネル、機能などをもう少し詳しく説明します。

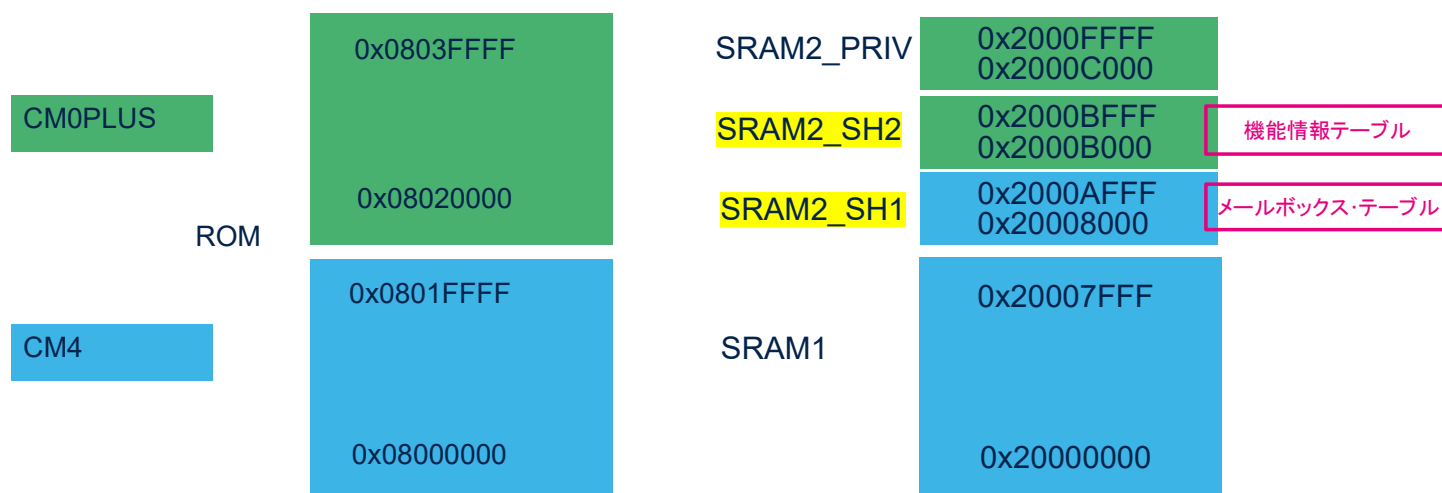
- IPCC: プロセッサ間通信コントローラ (Inter-Processor communication controller)
- 多くの STM32 ファミリ (STM32MP1、STM32WB、STM32WL など) にこの IP を用意
- 次の 2 つのチャンネル・グループを扱うように IPCC を設計
 - IPCC_1TO2: CPU1 の Tx チャンネルと CPU2 の Rx チャンネルに対応
 - IPCC_2TO1: CPU2 の Tx チャンネルと CPU1 の Rx チャンネルに対応
- チャンネル数はチップセットに応じて固定 (STM32MP1 では 2x2、STM32WB では 2x6、STM32WL では 2x6)
- バッファとメモリは IPCC の対象外
- STM32WL では、CPU1 は CM4、CPU2 は CM0PLUS



life.augmented

IPCC (プロセッサ間通信コントローラ) から説明を始めます。
この IP は、STM32MP1、STM32WB、STM32WL をはじめとする多数の STM32 ファミリに用意されています。
デバイス・ファミリに応じて、複数の通信チャンネルを基本としています (STM32MP1 では 2x2、STM32WB では 2x6、STM32WL では 2x6)。
IPCC ではバッファもメモリも扱いませんが、この両方に対する割込みをトリガします。

- v1.0.0 リリースでの EndNode 向けリンカ・ファイルの例



情報を交換するには共有メモリが必要です。

一般的に、ST の納品に付属する SubGHz アプリケーションでは、リンカ・ファイルが次のようになっています。

主に CM4 で使用する SRAM1:C スタック、ヒープ、cm4 グローバル変数など

CM0PLUS で使用する SRAM2_PRIV(セキュリティが有効な場合に保護されます):C スタック、ヒープ、CM4 のグローバル変数、セキュア・コードなど

SubGHz のリンカ・ファイルでは、STANDBY モードで保持する必要があるプロパティが存在することから、MBMUX の共有メモリが SRAM2 に置かれます。また、アドレス 0x20008000 は、IPCCDBA オプション・バイトのデフォルト値と一致しています(詳細は後ほど説明します)。

共有メモリは SH1 および SH2 の 2 つで構成されます。

SRAM2_SH1 は、CM4 で割り当てられる項目の共有メモリです(メールボックス・テーブルなど)。このテーブルは、Cmd/Resp と Notif/Ack の両方向での情報交換に使用します。

SRAM2_SH2 は、CM0PLUS で割り当てられる項目の共有メモリです(機能情報テーブルなど)。このテーブルは、CM0PLUS の機能を公開するために初期化でのみ使用します。

MUX: 機能とチャンネルとの関係

機能 (CPU で用意される、アプリ、サービス、ファームウェアのエンティティ)

システム

Radio_SubGh

Sigfox

KMS

トレース

LoRaWAN

サンプル ID:
LORAMAC_INITIALIZATION_ID
LORAMAC_START_ID
LORAMAC_STOP_ID
LORAMAC_MIB_GET_REQUEST_CONFIRM_ID
LORAMAC_MIB_SET_REQUEST_CONFIRM_ID
LORAMAC_MLME_REQUEST_ID
LORAMAC_MCPS_REQUEST_ID

MUX

チャンネル

MB

CmdResp_0

CmdResp_1

CmdResp_2

CmdResp_3

CmdResp_4

CmdResp_5

NotifAck_0

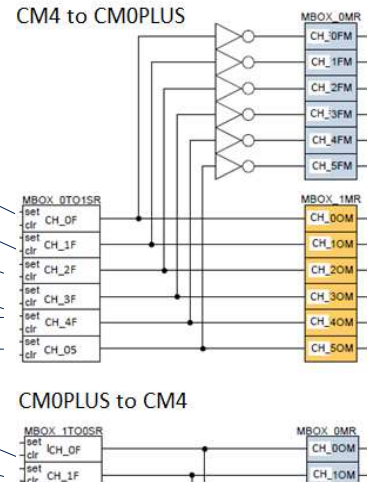
NotifAck_1

NotifAck_2

NotifAck_3

NotifAck_4

NotifAck_5



life.augmented

9

概念説明の最後は、機能とチャンネルとの関係です。

チャンネルは、IPCC のハードウェアで用意される物理エンティティです。

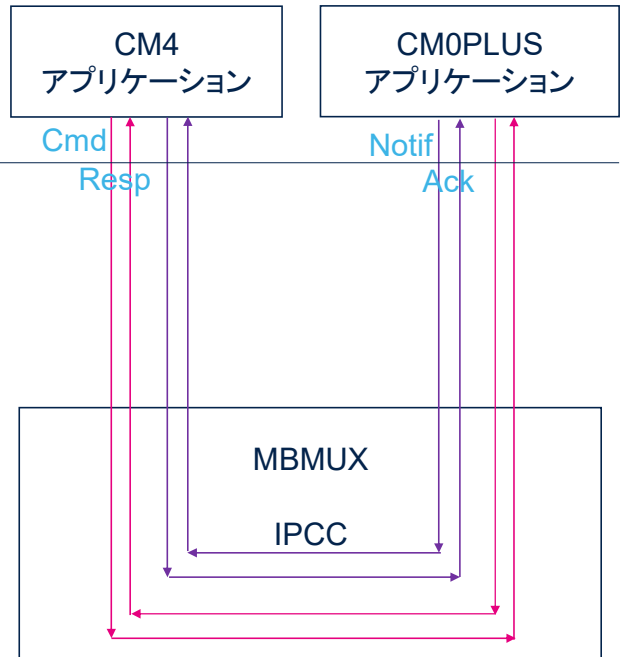
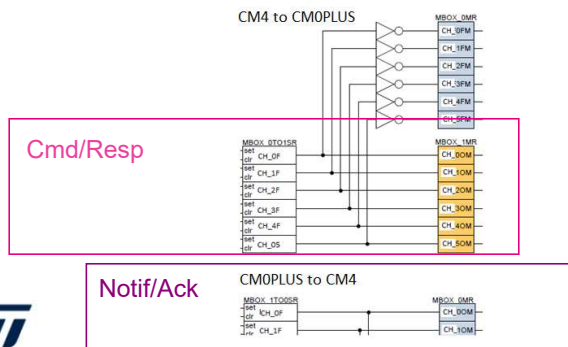
機能は、IPCC の各ハードウェア・チャンネルにマッピングされるモジュールやエンティティです (LoRa、Sigfox、KMS、トレースなど)。

このマッピングは動的で、ソフトウェアの登録を通じて実行されます。マッピングを扱うモジュールはマルチプレクサ (MUX) です。

この図の右側には、6 つのコマンドレスポンス・チャンネルと 6 つの通知/確認応答チャンネルを示しています。これらのチャンネルは、左側の各機能にマッピングできます。

ファームウェア・ディクショナリ: Cmd、Resp、Notif、Ack

- **Cmd**: IPCCのTxチャンネル上でCM4からCM0PLUSへ送信
- **Resp**: IPCCのTxチャンネル上でCM0PLUSからCM4へ送信
- **Notif**: IPCCのRxチャンネル上でCM0PLUSからCM4へ送信
- **Ack**: IPCCのRxチャンネル上でCM4からCM0PLUSへ送信



IPCC のチャンネルは、Tx と Rx のペアを複数使用して編成されています。MBMUX では、Tx チャンネルをコマンドレスポンス、Rx チャンネルを通知/確認応答に使用しています。

- **システム**: システムに関連するすべての通信をサポート
 - システムの初期化
 - IPCC チャンネルと機能との関係の登録
 - 機能の属性/能力に関する情報の交換
 - RTC 通知などの優先度が高い動作向けに RTC 追加システム・チャンネルを追加可能
- **トレース**: CM0PLUS では、IPCC を通じて CM4 に送信するログを循環キューに配置。CM4 では、従来の CM4 ログの場合と同じ方法 (USART など) でこの情報を出力して処理
- **KMS/SKS**: セキュア・キー・ストレージ
- **無線ドライバ**: SubGHz 無線と直接のインタフェースが可能
- **LoRaWANスタック**: このチャンネルを使用して、LoRaWANコマンド (初期化やリクエストなど) および LoRaWAN プロトコルに関連するイベント (レスポンスや指示) すべてとのインタフェースを構成
- **Sigfoxスタック**: このチャンネルを使用して、Sigfoxコマンド (初期化やリクエストなど) および Sigfox プロトコルに関連するイベント (レスポンスや指示) のすべてとのインタフェースを構成
- これら以外の機能を追加可能



ここでは、現在利用可能な機能について簡単に説明します。システムは実際には機能ではなく、メールボックスのコアを扱い、必ずチャンネル 0 を使用します。CM0PLUS のコアで何らかのデータをログに記録する必要がある場合にトレースを使用します。このログは、メールボックスを介して CM4 に送信されます。LoRaWAN、Sigfox、KMS などの他の機能については、他のプレゼンテーションで取り上げます。Wmbus など、ここにはない機能を追加できます。追加した機能は動的に各チャンネルにマッピングされます。

チャンネルでの機能の登録

- MBMUX は、利用できる IPCC チャンネルの数よりも "アプリケーション機能" の数が多い状況を考慮して実装済み。したがって、マッピングはハードコードではない。
- MBMUX では、各 "機能" を 1 つ以上のチャンネルに関連付け。どの機能がメッセージを送信するかが上位レイヤから MBMUX に通知され、MBMUX は関連するチャンネルを透過的に使用。
- 各機能は、登録とキャンセルによってチャンネルを予約または解放。MBMUX は、利用できるチャンネルに応じてチャンネルを決定。
- MBMUX の "システム" チェック:
 - 登録を要求している機能を登録できるチャンネルがまだ十分にあるかどうか
 - 登録を要求している機能が 2 回登録されていないかどうか
 - 登録を要求している機能がリモート CPU (CM0PLUS) に存在するかどうか
- IPCC の Tx チャンネルと Rx チャンネルは方向が独立しているので、ある 1 つの方向には機能がチャンネルを必要としないことが考えられるが、その他の方向に 2 つのチャンネルを必要とすることもあり得る
 - その例として Tx 方向を使用しないトレース機能がある。



MBMUX は、利用できる IPCC チャンネルの数よりも "アプリケーション機能" の数が多い状況を考慮して実装されています。したがって、マッピングはハードコードではありません。MBMUX では、各 "機能" を 1 つ以上のチャンネルに関連付けます。どの機能がメッセージを送信するかが上位レイヤから MBMUX に通知され、MBMUX は関連するチャンネルを透過的に使用します。各機能は、登録とキャンセルによってチャンネルを予約または解放し、MBMUX は、利用できるチャンネルに応じてチャンネルを決定します。

MBMUX: 機能のマッピング

- 登録とキャンセルによって自動的に機能をチャンネルに割当て

機能	IPCC の Tx チャンネル (CPU1 → CPU2)	IPCC の Rx チャンネル (CPU2 → CPU1)
システム	0	0
LoRaWAN	登録による自動割当て	登録による自動割当て
無線	登録による自動割当て	登録による自動割当て
Sigfox	登録による自動割当て	登録による自動割当て
KMS*	登録による自動割当て	該当なし
トレース	該当なし	登録による自動割当て
RTC	該当なし	登録による自動割当て



* CubeWL v1.0.0 では KMS が完全には実装されていない

13

この表は、これまでのスライドで説明したマッピングをまとめたものです。

機能によるチャネルの使用

- 原則として、2つのコア間で共通の言語が定義済みであれば、チャネルと共有バッファを使用してあらゆる種類のメッセージを交換可能
- 実際上、"言語"には、"一方のコアから他方のコアで機能を実行できるようにして、その戻り値または処理済みバッファを取得する"という固有の目標がある
- メールボックスの共有メモリは、この目標を達成できるように設計
 - CM4 で Cmd を使用して、CM0PLUS に実装されている関数を呼び出し
 - CM0PLUS で Resp を使用して、関数の実行が完了して戻り値が得られていることを通知
 - CM0PLUS で Notif を使用して、CM4 に実装されている機能呼び出し
 - CM4 で Ack を使用して、関数の実行が完了して戻り値が得られていることを通知




2つのコア間で共通の言語が定義済みであれば、チャネルと共有バッファを使用してあらゆる種類のメッセージを交換できます。

メールボックスの共有メモリは、この目標を達成できるように設計されています。

- CM4 で Cmd を使用して、CM0PLUS に実装されている関数を呼び出し
- CM0PLUS で Resp を使用して、関数の実行が完了して戻り値が得られていることを通知
- CM0PLUS で Notif を使用して、CM4 に実装されている機能呼び出し
- CM4 で Ack を使用して、関数の実行が完了して戻り値が得られていることを通知

- 言語の目的に応じてメッセージを書式設定
 - 識別子 (MsgId) を使用して呼び出す関数を特定する
 - バッファと構造体に関数のパラメータまたはポインタを渡す
 - 関数の実行が完了した通知を受け、戻り値を取得する

```
typedef struct {
    uint32_t MsgId;
    void (*MsgCpu1Cb)(void* ComObj);
    void (*MsgCpu2Cb)(void* ComObj);
    uint16_t BufSize;    /*!< アプリケーションから渡される配列のサイズ */
    uint16_t ParamCnt;  /*!< メッセージを構成する単語の数 */
    uint32_t* ParamBuf;
    uint32_t ReturnVal;
} MBMUX_ComParam_t;
```



- 緑色の値は登録時に設定 (機能ごとに 1 回)
- 青色の値はメッセージ (Cmd、Resp、Notif、Ack) 送信のたびに設定



したがって、言語の目的に応じてメッセージが書式設定されています。

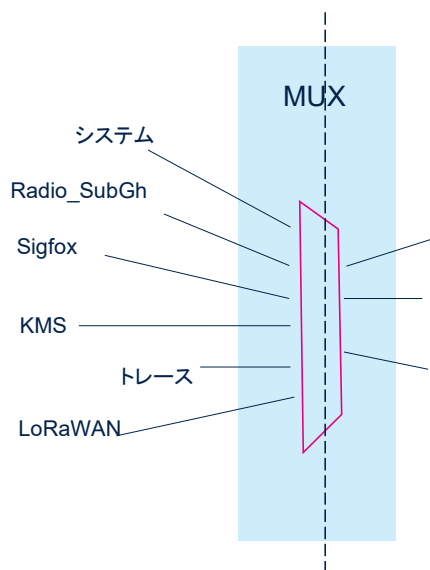
- 識別子 (MsgId) を使用して呼び出す関数を特定する
 - バッファと構造体にパラメータまたはポインタを渡す
 - 関数の実行が完了した通知を受け、戻り値を取得する
- ここに挙げた構造体は、コードから直接抜粋したものです。後ほど詳しく説明します。

```
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;
```

各 IPCC チャンネルには、通信のチャンネル専用インスタンスがあり、これはすべての IPCC チャンネル(Tx と Rx)と同様です。したがって、前のスライドでは MBMUX_ComParam_t が 12 回繰り返されています。

次のスライドの図は直感的でわかりやすく、C コードも示されています。

通信テーブル(1/5)

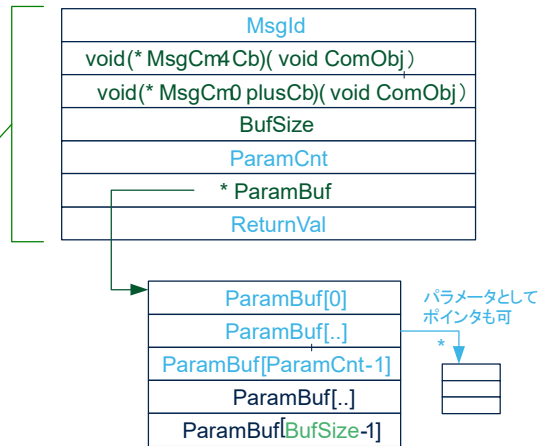


MBMUX_ComTable_t

MBCmdRespParam[0]
MBCmdRespParam[1]
MBCmdRespParam[2]
MBCmdRespParam[3]
MBCmdRespParam[4]
MBCmdRespParam[5]
MBNotifAckParam[0]
MBNotifAckParam[1]
MBNotifAckParam[2]
MBNotifAckParam[3]
MBNotifAckParam[4]
MBNotifAckParam[5]
MBMUXMapping [FEAT_INFO_CNT][2];
SynchronizeCpusAtBoot
ChipRevId

MBMUX_ComParam_t を 12 回繰り返し

緑 : 初期化と機能登録のときに設定
青 : メッセージ交換のために設定

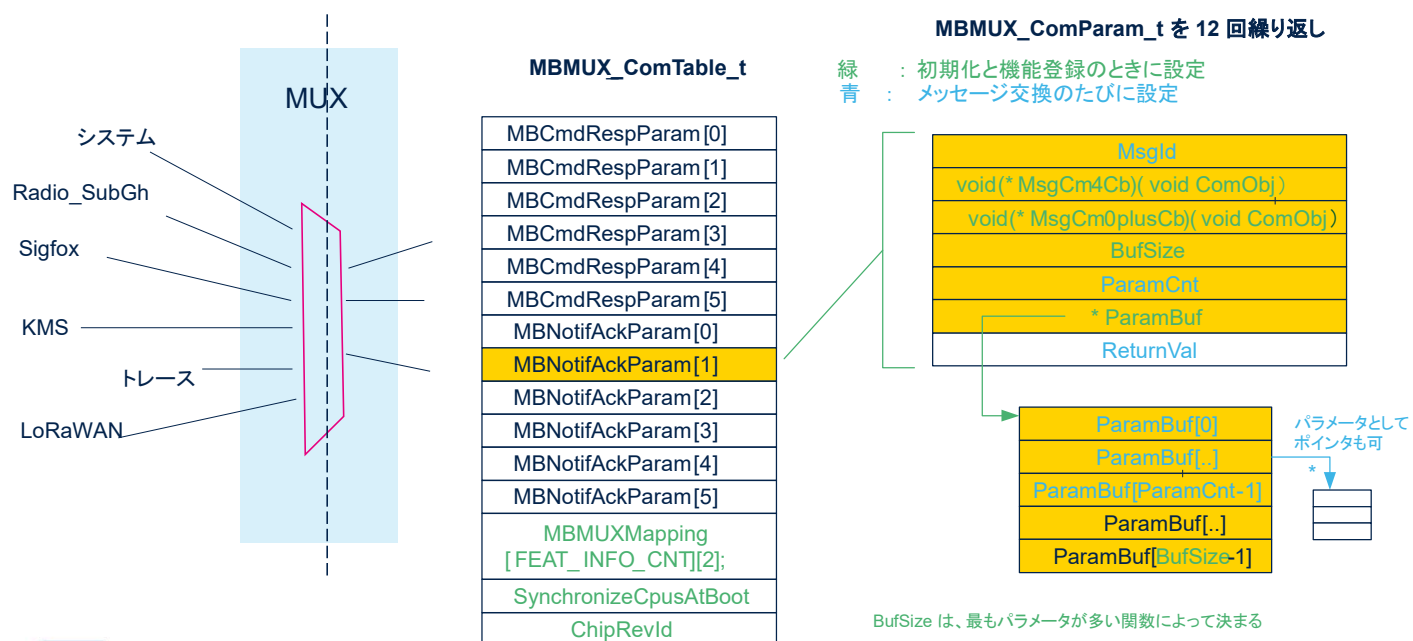


BufSize は、最もパラメータが多い関数によって決まる



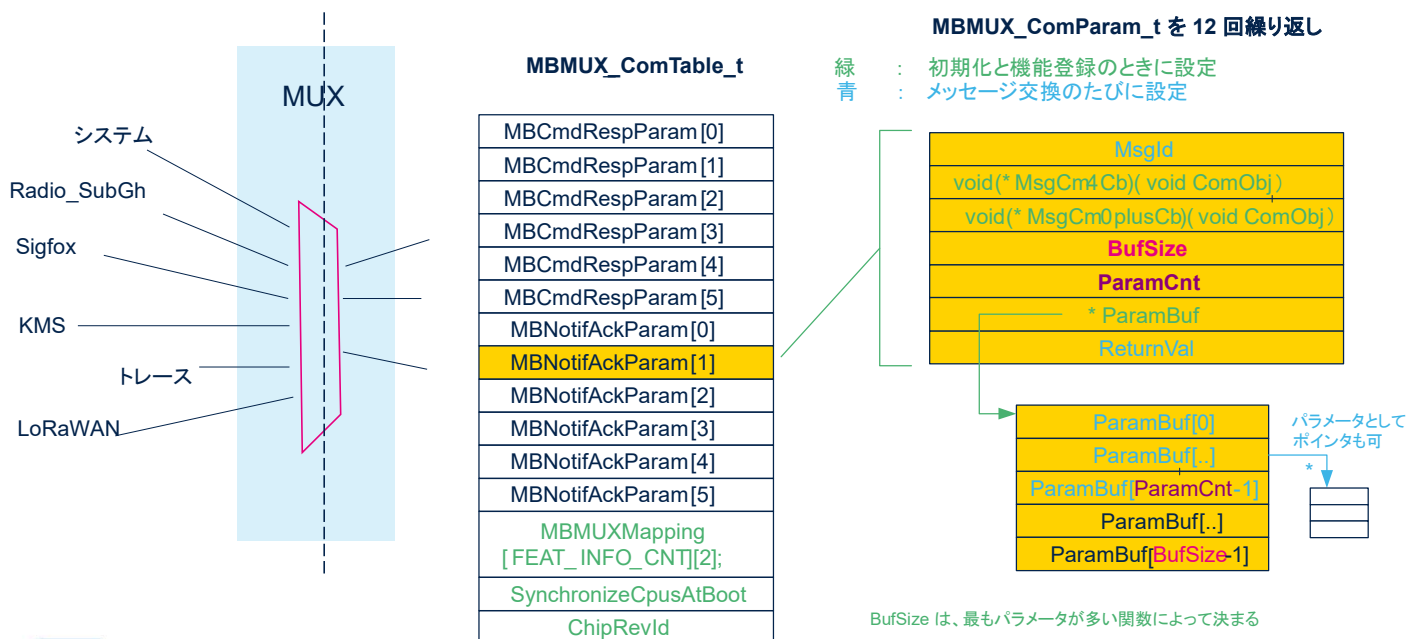
ここでは、MBMUX_ComTable_t で MBMUX_ComParameter_t が 12 回インスタンス化されていることがわかります。
機能で登録をリクエストすると、空いているチャンネルがその機能に割り当てられ、この登録が MBMUX のマッピング・テーブルに保存されます。

通信テーブル(2/5)



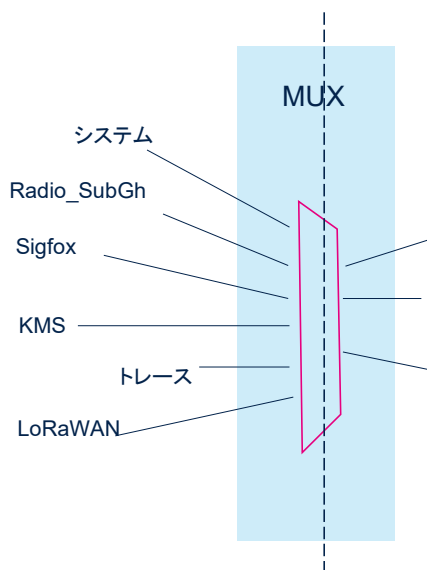
たとえば、トレース機能が登録を求め、それに MBNotifAckParam[1] が割り当てられたとします。トレース機能は Rx チャンネルのみを必要とするので MBCmdRespParam インスタンスの登録は不要です。CM4 に対して CM0PLUS のみがトレース関数を呼び出し、他の呼び出し形態はないからです。2 つのコア間でのトレース関数の呼び出しごとに MsgID があります。トレースの ParamBuf は、パラメータが多いほうのトレース関数から計算されます。パラメータが最も多い関数のパラメータ数が 5 であれば、BufSize パラメータは 5 に設定されます。

通信テーブル(3/5)



MBMUX に登録された機能は、IPCC チャネルとそれに関連付けられたバッファを予約します。このバッファのサイズは登録された機能によって決まります。一般的には、その機能の関数のうち、パラメータが最も多い関数に基づきます。たとえば、"トレースの ParamBuf" は、パラメータが多いほうの "トレース関数" に応じて決まります。パラメータが 3 つのみのトレース関数を呼び出すと、トレースの ParamBuf は部分的に使用され、その関数の実行中は ParamCnt が 3 に設定されています。ParamBuf は int32_t 型フィールドで構成され、これらのフィールドには、値、構造体へのポインタ、またはデータ・バッファへのポインタが格納されます。

通信テーブル(4/5)

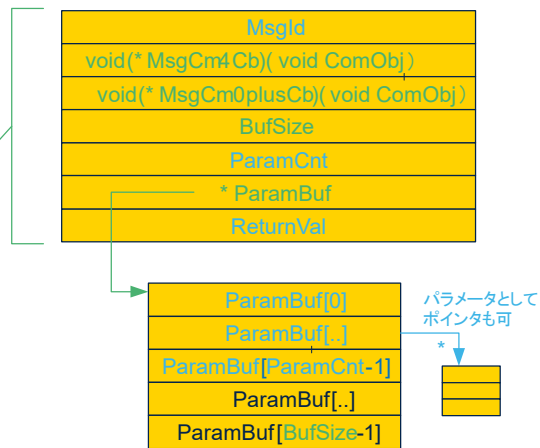


MBMUX_ComTable_t

MBCmdRespParam[0]
MBCmdRespParam[1]
MBCmdRespParam[2]
MBCmdRespParam[3]
MBCmdRespParam[4]
MBCmdRespParam[5]
MBNotifAckParam[0]
MBNotifAckParam[1]
MBNotifAckParam[2]
MBNotifAckParam[3]
MBNotifAckParam[4]
MBNotifAckParam[5]
MBMUXMapping [FEAT_INFO_CNT][2];
SynchronizeCpusAtBoot
ChipRevId

MBMUX_ComParam_t を 12 回繰り返し

緑 : 初期化と機能登録のときに設定
青 : メッセージ交換のたびに設定



BufSize は、最もパラメータが多い関数によって決まる

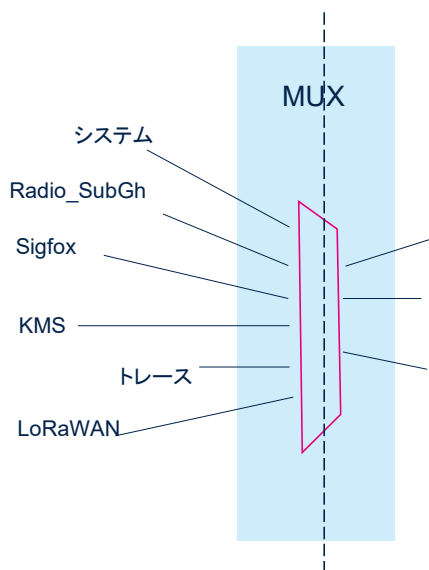


交換するデータまたは 2 つの CPU からアクセスするデータを取り
めるバッファ(メールボックス・バッファ、パラメータ・バッファ、各パ
ラメータが指すバッファ)は、すべて共有メモリとする必要があります。

通信テーブル全体および ParamBuf になる可能性がある 12 個
のバッファは CM4 によって割り当てます。

パラメータ・バッファ引数が指すバッファは、CM4 または
CM0PLUS のどちらからでも割り当てることができます。

通信テーブル(5/5)

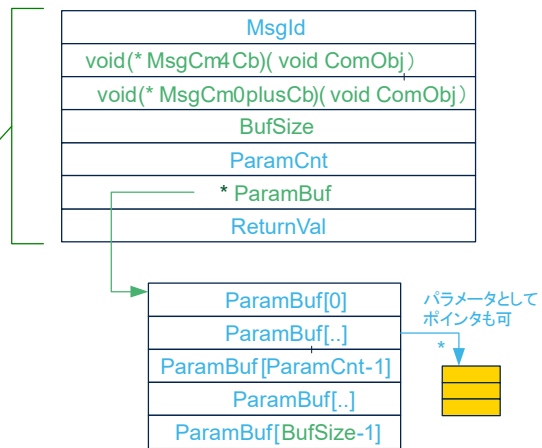


MBMUX_ComTable_t

MBCmdRespParam[0]
MBCmdRespParam[1]
MBCmdRespParam[2]
MBCmdRespParam[3]
MBCmdRespParam[4]
MBCmdRespParam[5]
MBNotifAckParam[0]
MBNotifAckParam[1]
MBNotifAckParam[2]
MBNotifAckParam[3]
MBNotifAckParam[4]
MBNotifAckParam[5]
MBMUXMapping [FEAT_INFO_CNT][2];
SynchronizeCpusAtBoot
ChipRevId

MBMUX_ComParam_t を 12 回繰り返し

緑 : 初期化と機能登録のときに設定
 青 : メッセージ交換のたびに設定



BufSize は、最もパラメータが多い関数によって決まる



トレースのサンプルに戻ります。ログ・メッセージは、パラメータの buffer 引数で指す数バイトのバッファです。トレース機能では、このバッファは CM0PLUS によって割り当てられます。

プライベート・バッファの共有

SH1 メモリ (CM4 によって割り当てられたテーブル)

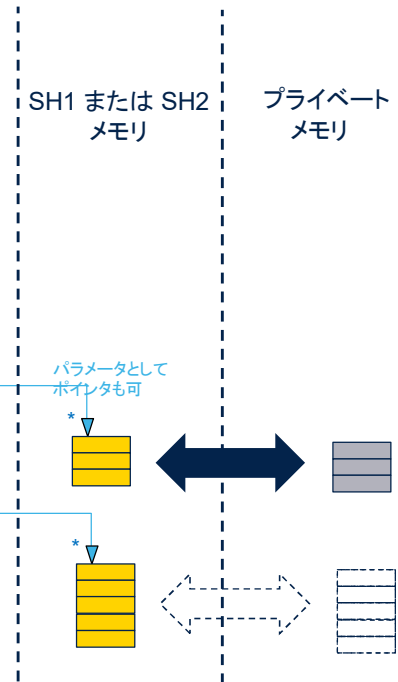
MBMUX_ComTable_t

MBCmdRespParam [0]
MBCmdRespParam [1]
MBCmdRespParam [2]
MBCmdRespParam [3]
MBCmdRespParam [4]
MBCmdRespParam [5]
MBNotifAckParam [0]
MBNotifAckParam [1]
MBNotifAckParam [2]
MBNotifAckParam [3]
MBNotifAckParam [4]
MBNotifAckParam [5]
MBMUXMapping [FEAT_INFO_CNT][2];
SynchronizeCpusAtBoot
ChipRevId

MBMUX_ComParam_t を 12 回繰り返す

MsgId
void (* MsgCm4Cb)(void ComObj)
void (* MsgCm0plusCb)(void ComObj)
BufSize
ParamCnt
* ParamBuf
RetVal

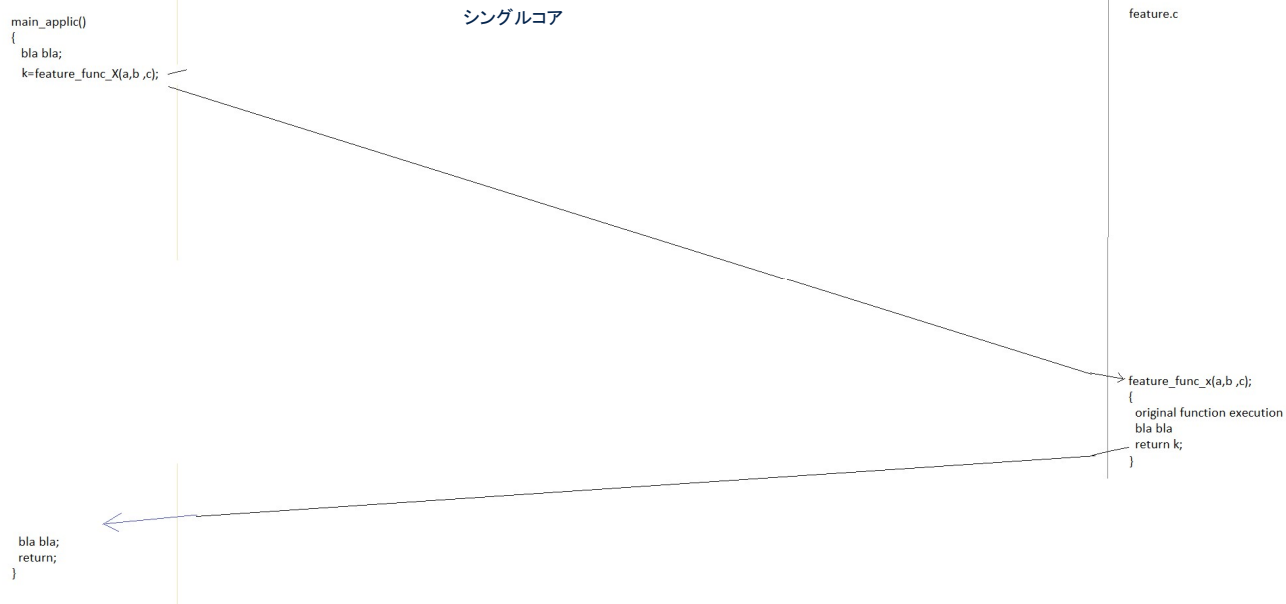
ParamBuf[0]
ParamBuf[.]
ParamBuf[ParamCnt-1]
ParamBuf[.]
ParamBuf[BufSize-1]



プライベート・メモリ領域にある "メモリ・スタック" または変数やバッファを関数パラメータのポインタが指している場合、そのバッファには特段の注意が必要です。
 このような場合、"機能の関数のラッパー" で "プライベート変数やバッファ" の一時コピーを作成し、他の CPU がその関数処理する前に共有メモリとする必要があります。他のコアの関数によってバッファが変更されている場合は、同様のラッパーでそのバッファを取得してプライベート・メモリに戻します。

シングル・コア・デバイスでの関数の実行

CM4



life.augmented

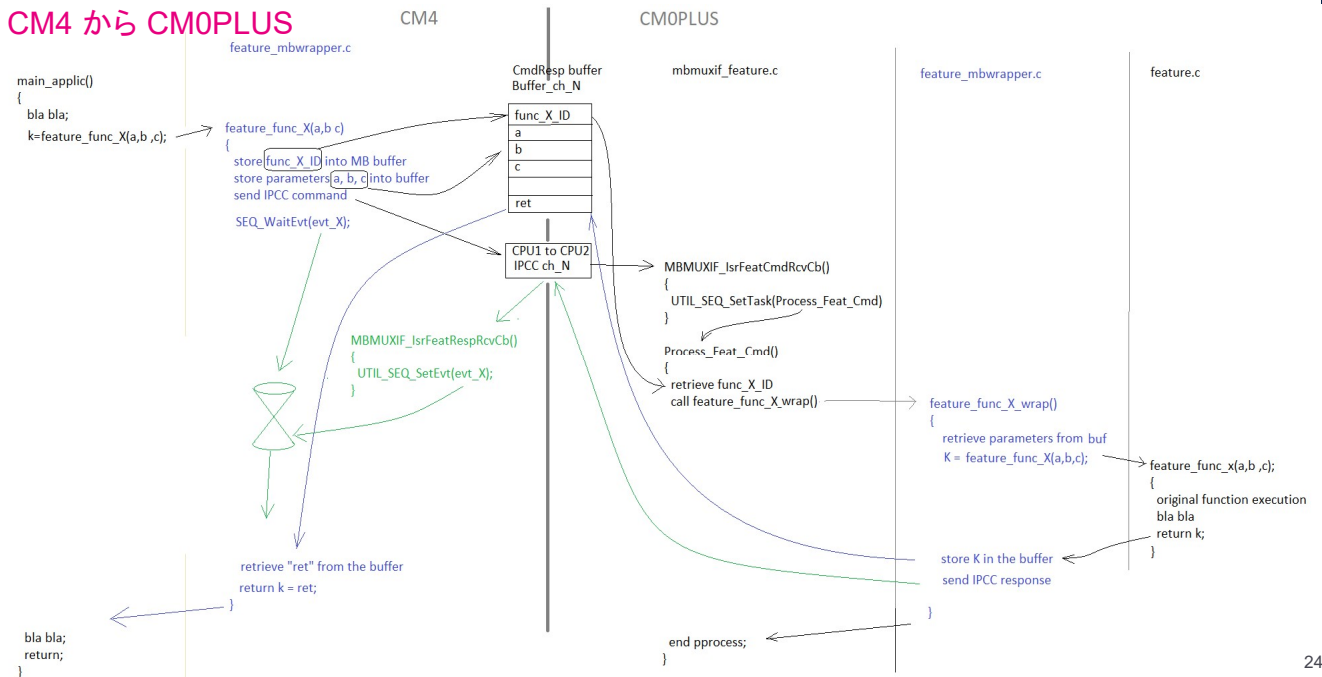
23

シングル・コアの場合は、すべての関数が同じコア上に存在します。

特定の機能に属する関数 "X" をプログラムで実行する必要がある場合、そのプログラムから a、b、c の各パラメータを渡すことでその関数を呼び出します。

メールボックスを使用する必要はありません。

ある CPU 上の関数を別の CPU から実行できるようにする メールボックス



CM0PLUS コア上の関数を CM4 コアから呼び出す必要があるとします。この関数は、チャンネルで登録された機能に属しており、CmdRespParam バッファが関連付けられています。

各機能にはラッパーが実装され、各関数にはその関数名を名前とするラッパー関数が存在します。

このラッパーは、次の値を CmdRespParam バッファに設定します。

- MsgId
- ParamCnt
- 関数 "X" のパラメータ(名前を a、b、c とします)

CM4 から IPCC を介して CM0PLUS に割り込みをトリガし、その応答を待ちます。この呼び出しはブロッキング呼び出しです。

CM0PLUS が IPCC 割り込みの処理を開始します。MsgId をデコードするアンラッパー関数を呼び出し、そのパラメータを抽出して、実際の関数 "X" を呼び出します。

CM0PLUS によって関数 "X" が実行され、その戻り値が CmdRespParam バッファに置かれます。

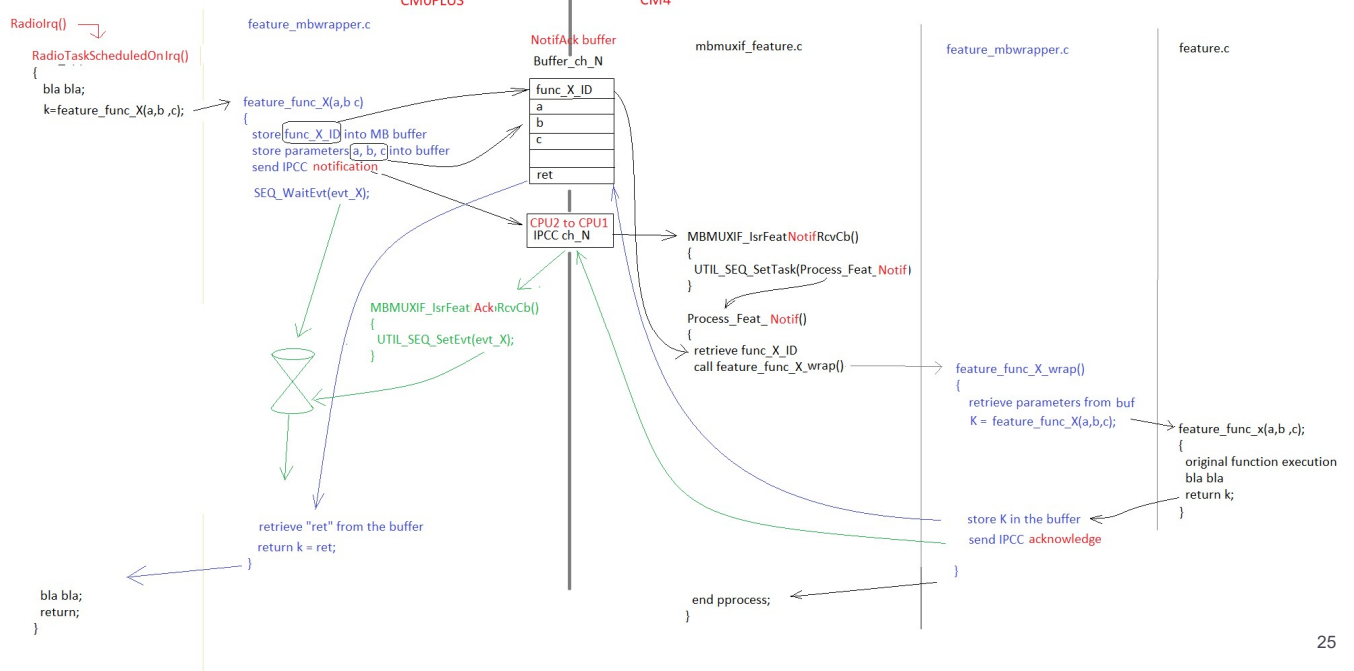
これで、CM0PLUS から CM4 に応答 (IPCC フラグのクリアなど) を返送できるようになります。

CM4 は、自身をウェイト・ステートからウェイクアップする割り込みを受け取り、CmdRespParam バッファから戻り値を取得して、`feature_func_x()` 関数の実行を終了します。

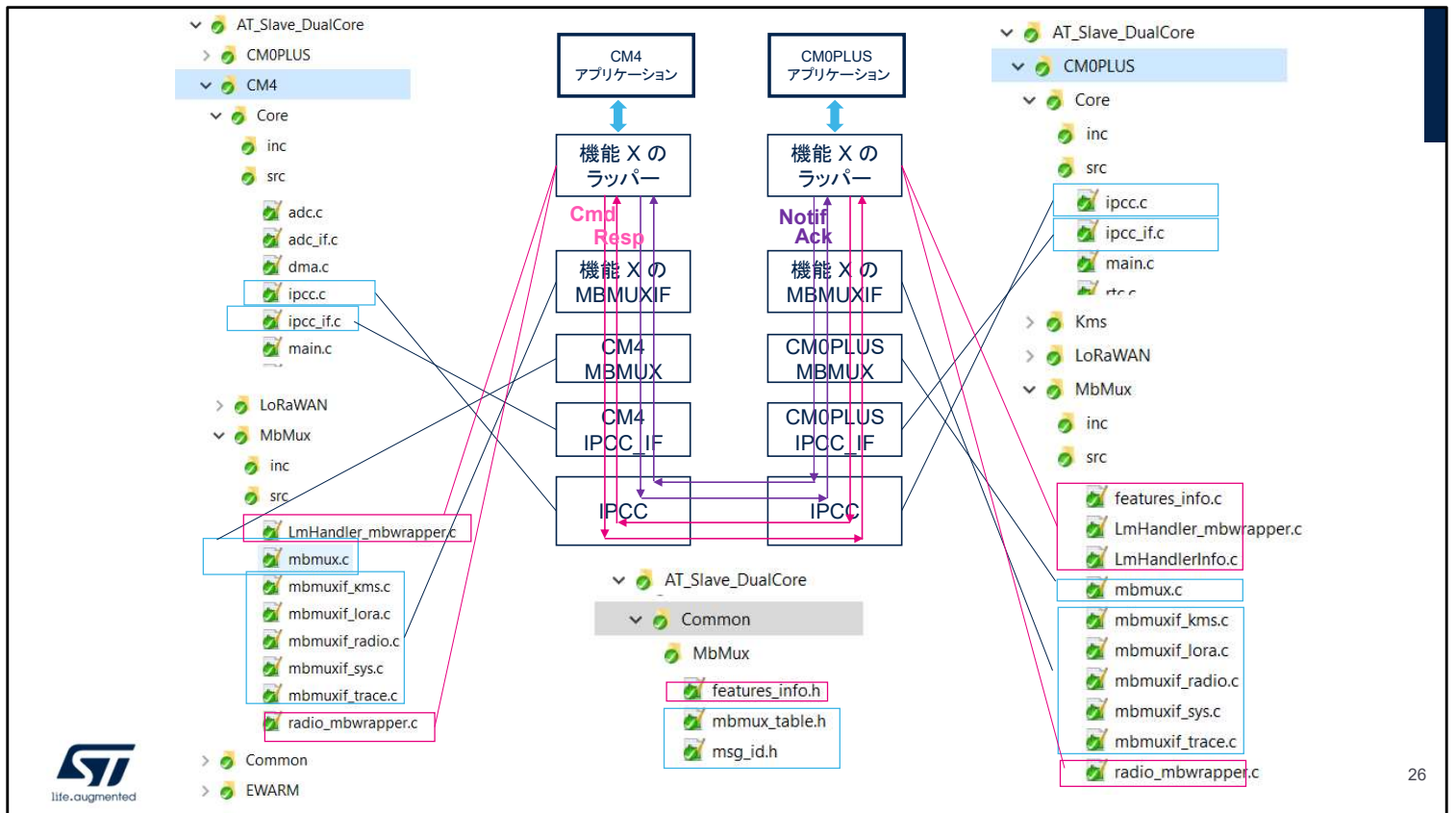
このような "ブロッキング呼び出し" 手法により、シングル・コア実行とデュアル・コア実行のどちらでも確実に同じ動作シーケンスが実現します。

ある CPU 上の関数を別の CPU から実行できるようにする メールボックス

CM0PLUS から CM4



CM4 コア上の関数を CM0PLUS から実行する必要がある場合、その手順は前回の場合と全面的に対称です。ただし、値が置かれる場所は `CmdRespParam` バッファではなく、`NotifAckParam` バッファです。関連する関数の名前に使用する文字も、`Cmd` が `Notif`、`Resp` が `Ack` になります。



次のスライドは、ここまで説明した概念とコード、ファイル、レイヤとの対応関係を理解するうえで効果的です。

この図では、例として挙げた LoRaWAN AT_Slave プロジェクトのディレクトリ・ツリーを示しています。

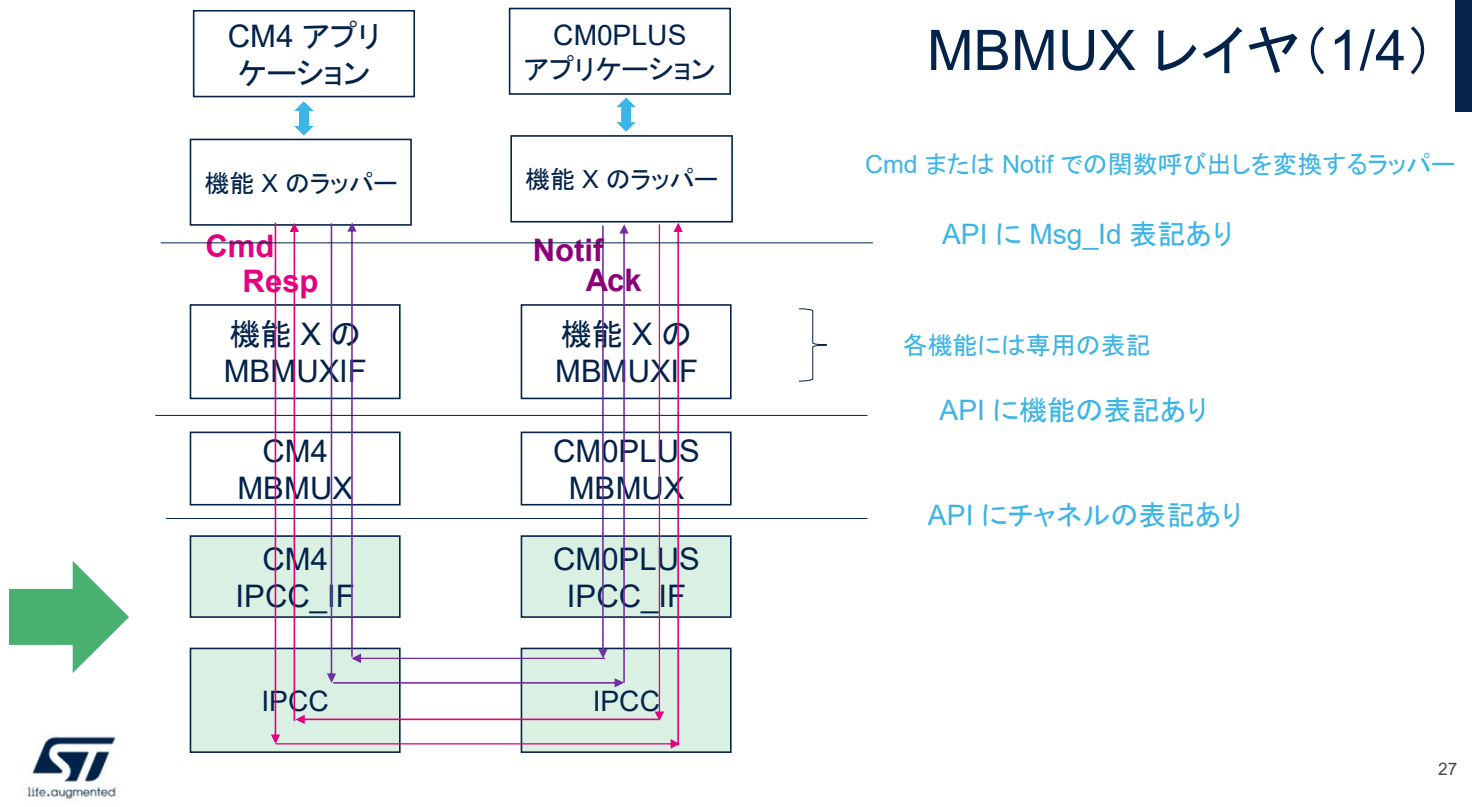
左側は、CM4 側で実行するプロジェクト部分を構成するファイルであり、右側は、プロジェクトの CM0PLUS 部分を構成するファイルです。

MBMUX は、このアプリケーションの一部ですが、自身専用のディレクトリ MbMux を備えています。

ファイルはアプリケーションごとに複製されます(すべてのアプリケーションやサンプルで横断的に使用されるミドルウェア、ユーティリティ、ドライバとはこの点が異なります)。

MBMUX はいくつかの方法で実装されます。このコードを理解するために、各レイヤについて説明します。

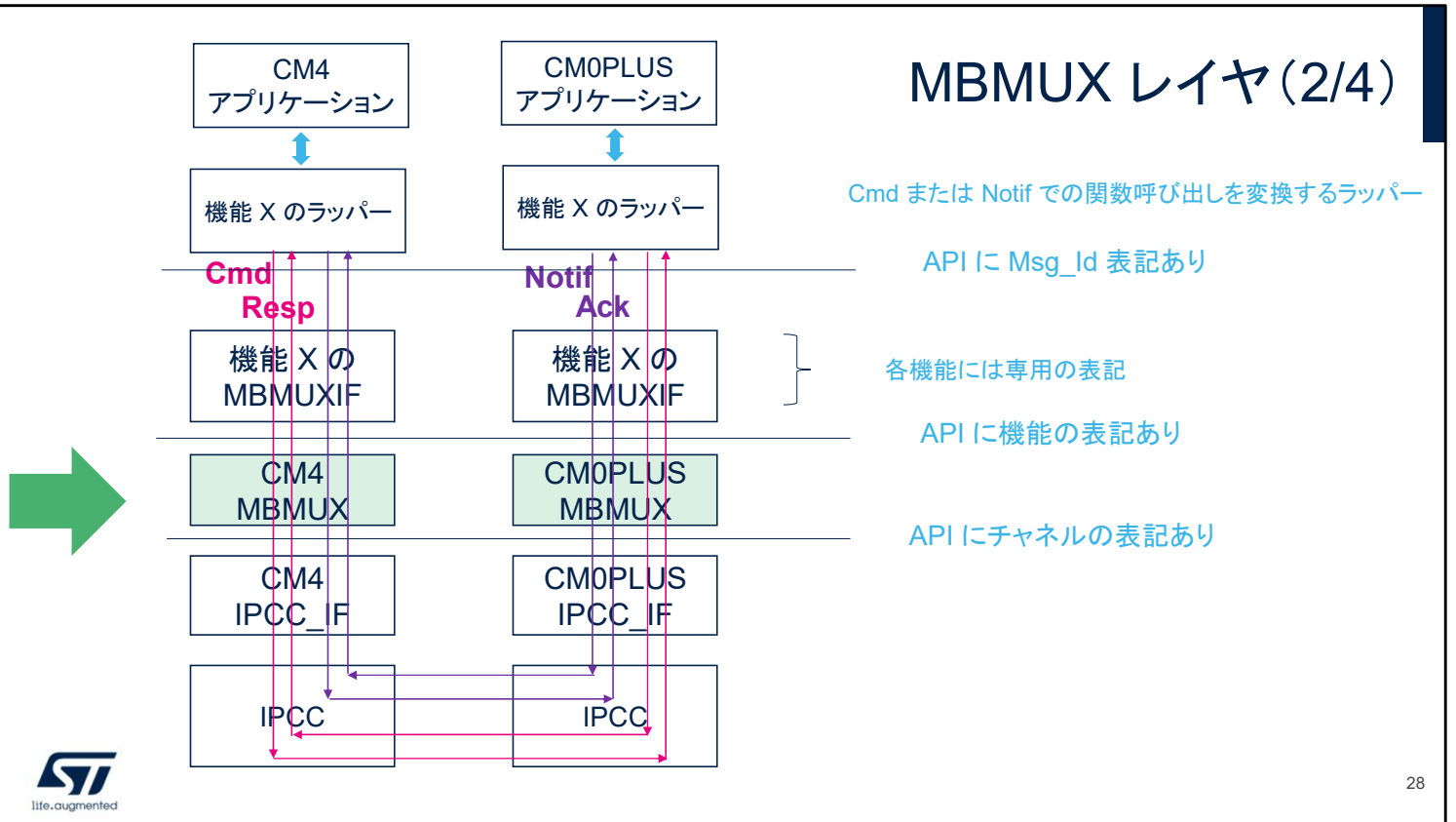
MBMUX レイヤ (1/4)



27

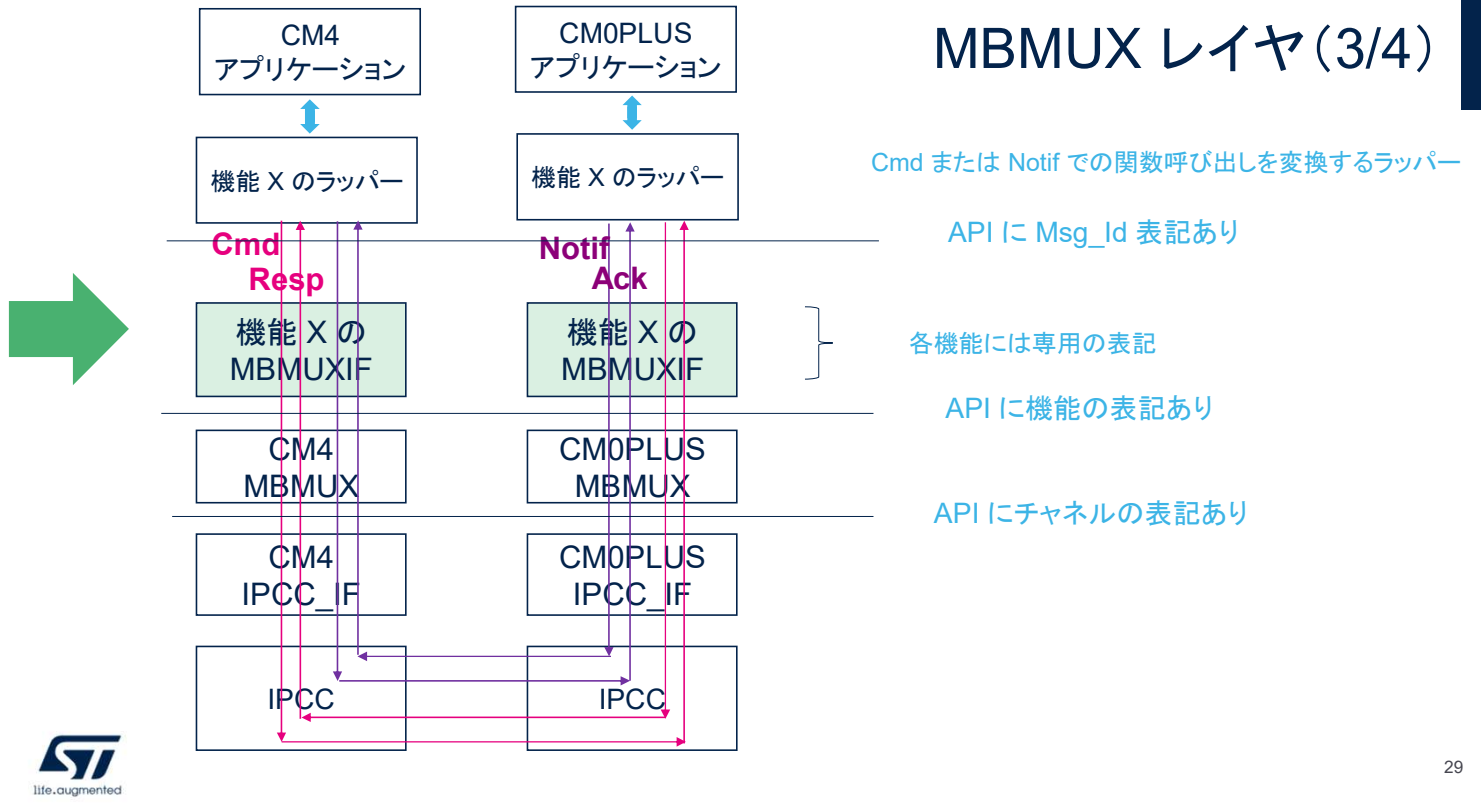
MBMUX は 3 つのレイヤで構成されます。この図の最下部は、基本的に物理レイヤである IPCC を示しています。IPCC は HAL ドライバの一部です。IPCC_IF はインタフェース・レイヤです (IP ごとに存在) CM4 コアと CM0PLUS コア両方のこれらのインタフェースは MBMUX の一部ではありません。

MBMUX レイヤ (2/4)



MBMUX の下位レイヤは IPCC の直上に位置し、そのタスクはチャンネルへの機能のマッピングです(登録と多重化を提供)。IPCC_IF および MBMUX の下位レイヤは汎用サービスです(ユーザによる変更は想定されていません)。

MBMUX レイヤ (3/4)



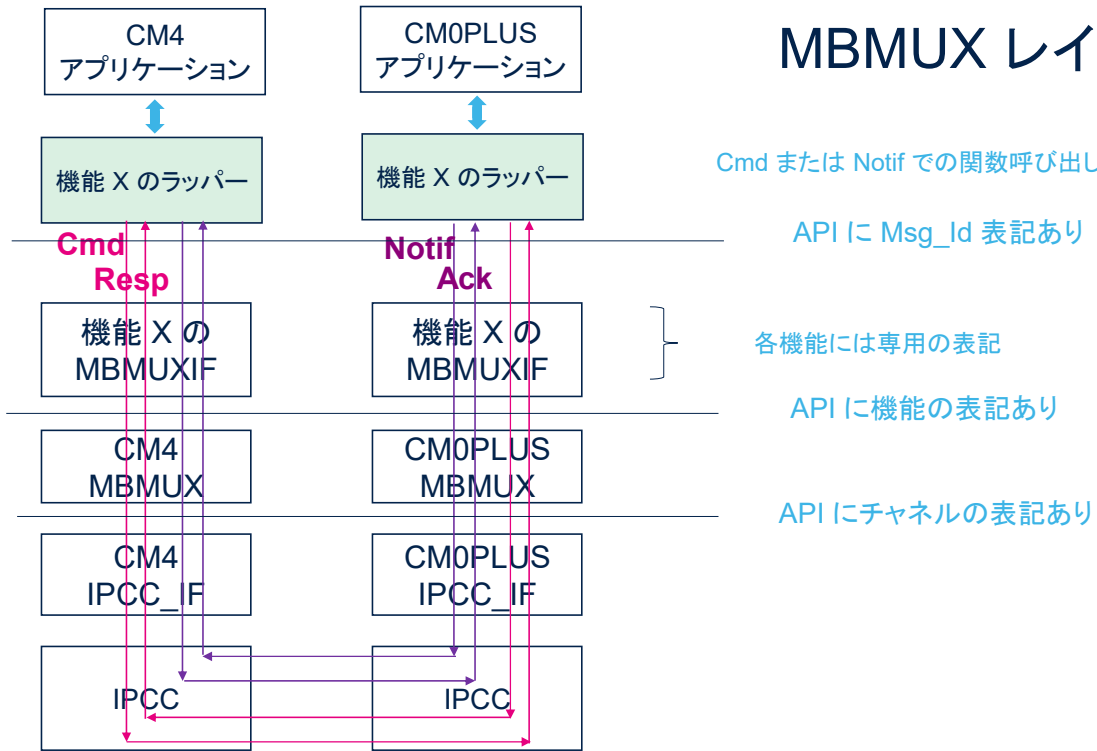
各機能は、それに専用の MBMUXIF によって MBMUX とインタフェース接続します。新しい機能を追加するには、該当の MBMUXIF を実装する必要があります。

パラメータ・バッファ・サイズやコールバックに対する反応性(タスクと割り込みのサービス・ルーチン間の関係)などのカスタマイズを、目的の機能で必要とする場合、既存の MBMUXIF の適応が求められることがあります。

MBMUXIF は次の各処理を実行します。

- CM4 と CM0PLUS との通信に使用する、機能専用バッファの割り当て
- MBMUX バッファでの機能登録の開始(通信バッファの登録、コールバックの登録)
- feature_ID とメールボックス方向からの上位レイヤの抽象化
- メッセージ ID スイッチの実装
- シーケンサの取り扱い(他のドキュメントで取り上げている "カーネル/スケジューラ" の一種)

MBMUX レイヤ (4/4)



Cmd または Notif での関数呼び出しを変換するラッパー

API に Msg_Id 表記あり

各機能には専用の表記

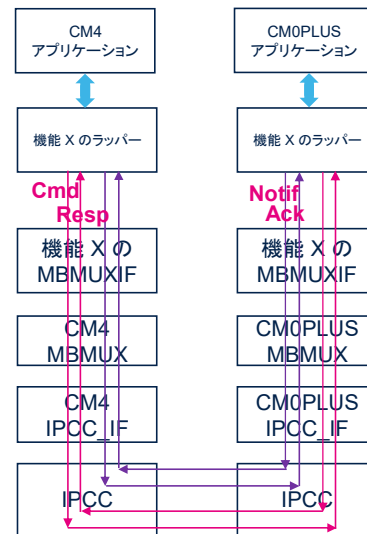
API に機能の表記あり

API にチャンネルの表記あり



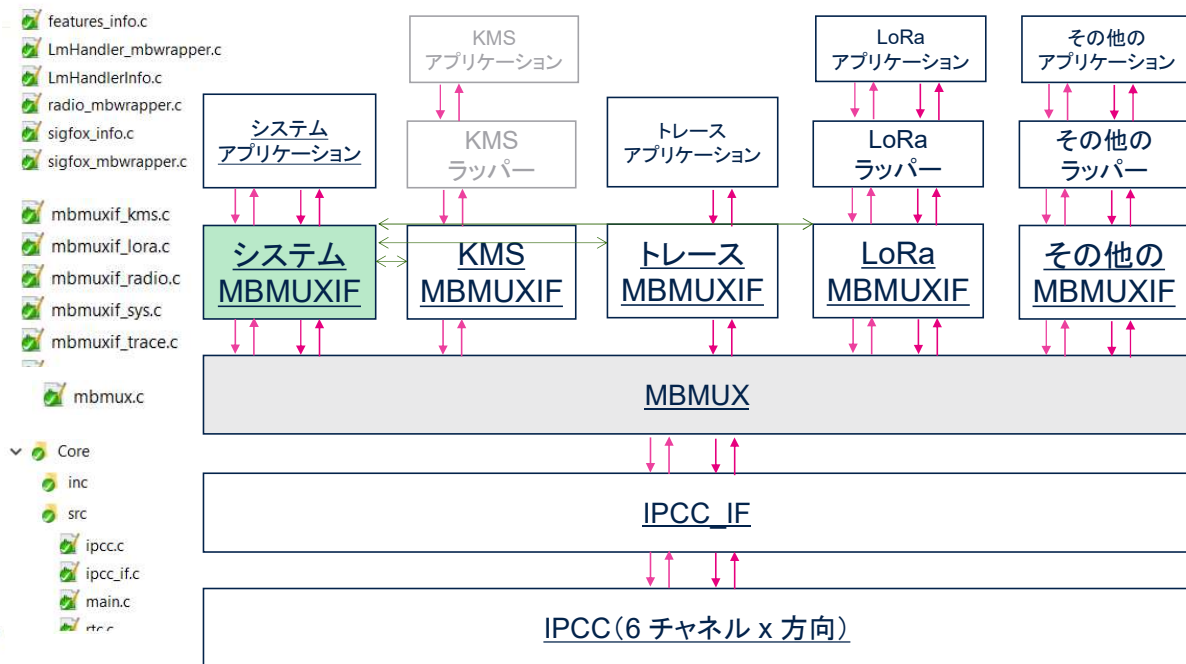
機能のラッパーは関数をメッセージに変換します。

- CM4 の IPCC_IF は CM0PLUS の IPCC_IF に対して鏡面関係
 - CM4 の IPCC_IF は、Cmd と Ack を送信するための API、および Notif と Resp を扱うコールバックを提供
 - CM0PLUS の IPCC_IF は、Notif と Resp を送信するための API、および Cmd と Ack を扱うコールバックを提供
- CM4 の MBMUX は CM0PLUS の MBMUX に対して鏡面関係
 - CM4 の MBMUX は、Cmd と Ack を送信するための API、および Notif と Resp を扱うコールバックを提供
 - CM0PLUS の MBMUX は、Notif と Resp を送信するための API、および Cmd と Ack を扱うコールバックを提供
- CM4 の MBMUXIF は CM0PLUS の MBMUXIF に対して鏡面関係
 - CM4 の MBMUXIF は、Cmd と Ack を送信するための API、および Notif と Resp を扱うコールバックを提供
 - CM0PLUS の MBMUXIF は、Notif と Resp を送信するための API、および Cmd と Ack を扱うコールバックを提供



CM4 コアの各ブロックには、CM0PLUS コアの該当ブロックに対する鏡面性がありますが、ブロック自体は同一ではありません。したがって、CM4/Mbmux/mbmux.c と CM0PLUS/Mbmux/mbmux.c のコードは異なります。CM4 の MBMUX ファイルでコマンドと確認応答を送信し、通知と応答を扱うコールバックを提供する場合、CM0PLUS 側でそれと鏡面性を持つファイルでは、通知と応答を送信し、コマンドと確認応答を扱うコールバックを提供します。

1つのCPU上のグローバル・スタック (他のCPUにミラーリング)



これは、前の図にある2つの列の1つを詳しく表した図です。すべてのMBMUXIFと相互作用する"システム MBMUXIF"を除き、各MBMUXIFブロックは機能間で独立しています。MBMUXとシステム MBMUXIFはメールボックスの中核的存在です。

注意: KMSへのNotif/Ackの矢印と、トレースへのCmd/Respの矢印がないのは誤植ではありません。これらの機能は次のように一方向性です。

- KMSは、CM0PLUSコアに暗号化を要求するためにCM4コアから呼び出されます。コールバックは不要です。
- トレースは、出力するCM4コア文字列をIPCCを介して送信するためにCM0PLUSコアで使用されます。

KMSのラッパーは実装されていません。SubGHzのサンプルでは、KMS同様にCM0PLUSコア上に存在するSubGHzスタックからのみKMSが呼び出されます。

CM4からCM0PLUSコア上のKMS APIを呼び出すには、専用のラッパーを実装する必要があります。

スライドの左側は、コード間の移動に便利なファイル・ツリーのスクリーン・ショットです。

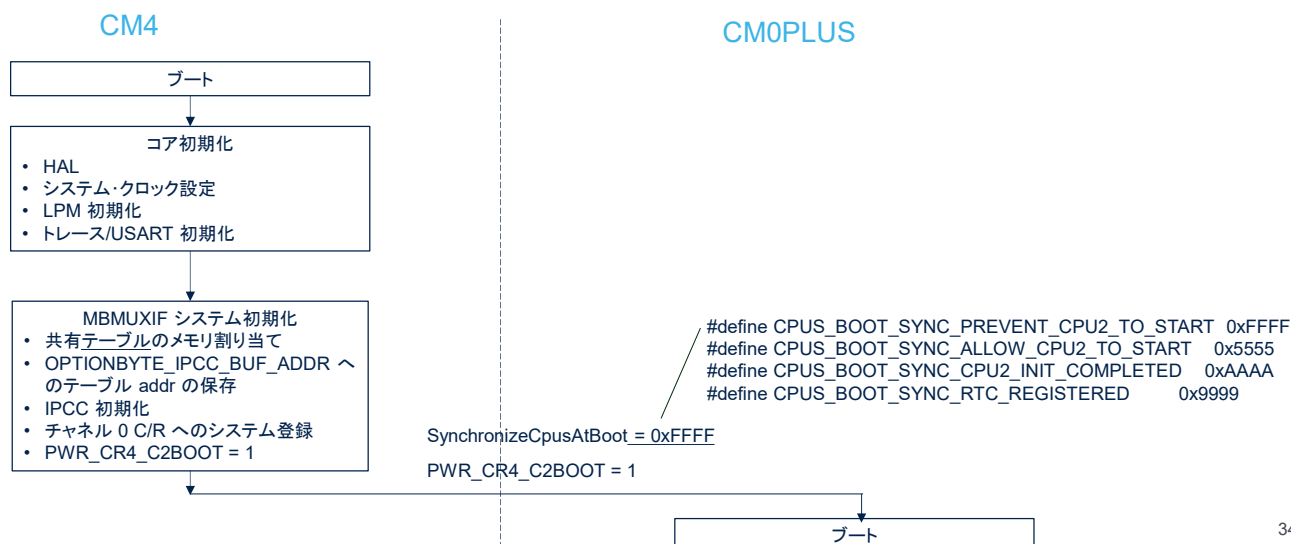
初期化シーケンスと機能情報テーブル

次のスライドでは、MBMUX がどのように初期化されるかを説明するタイムラインシーケンスを提供します。これは、コードを変更する必要がある人にも役立ちます

デュアル・コア通信の初期化(1/2)

• 初期化シーケンス

- 共有メモリのアドレスを CM4 で設定(オプション・バイト)
- CM4 から CM0PLUS をブート



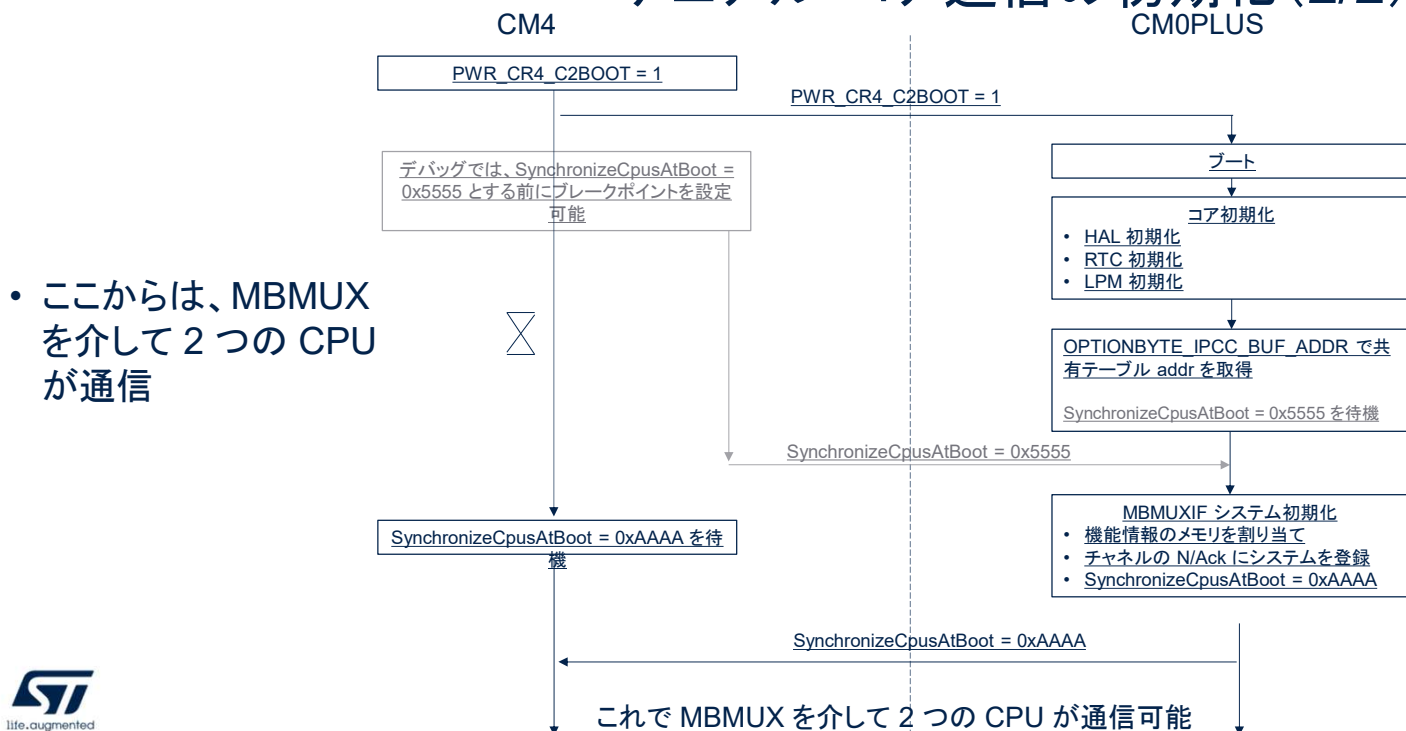
34

SubGHz のサンプルで最初にブートするコアは必ず CM4 コアです。CM0PLUS コアは、PWR_CR4 レジスタの C2BOOT ビット部分を通じて保留状態になっています。

MBMUX テーブルや通信バッファなどの MBMUXIF システム・ブロックが CM4 コアによって初期化されてから、CM0PLUS コアが解放されます。

このプレゼンテーションの最後で、OPTIONBYTE_IPCC_BUF_ADDR の初期化を詳しく取り上げます。

デュアル・コア通信の初期化 (2/2)



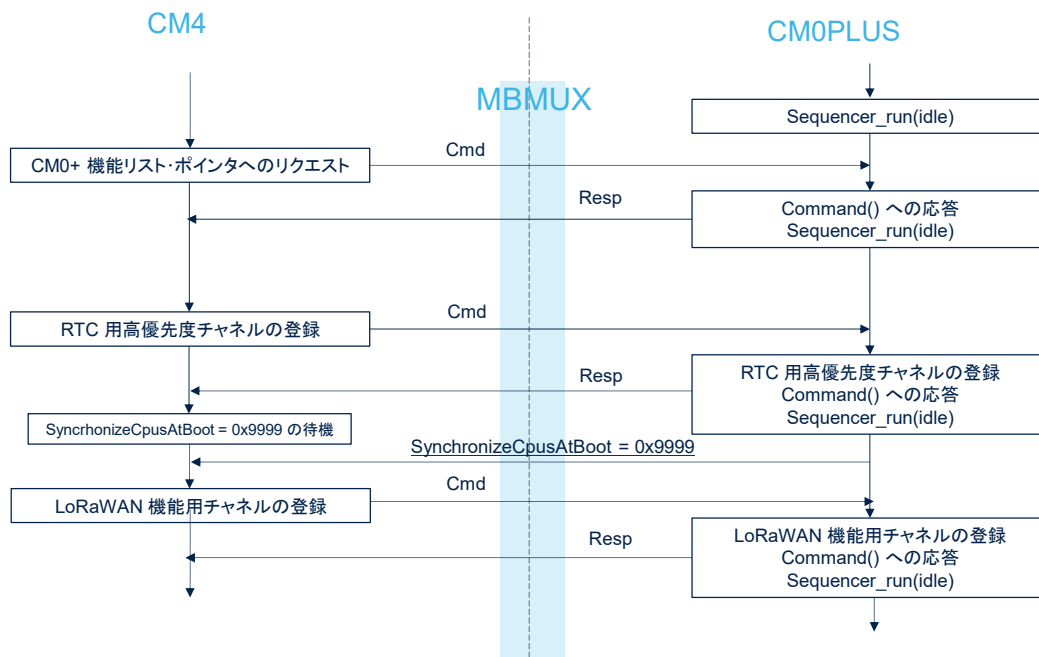
35

SynchronizeCpusAtBoot フラグが 0xFFFF で初期化されます。CM4 でこのフラグを使用して、CM0PLUS で MBMUNIX の初期化が完了したことを検知します (このフラグの値が 0xAAAA になります)。

必要に応じ、このフラグをデバッグの際に使用して、MBMUNIX_System_Init() 関数が実行されないように CM0PLUS コアを保持できます。CM4 にブレークポイントを配置しないので、この動作は透過的です。

MBMUNIXIF システムが初期化されると、MBMUNIX を介して 2 つの CPU が通信できます。

MBMUX を介した初期化の継続



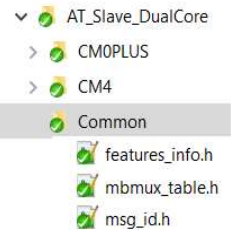
どの機能についても登録のマスタは必ず CM4 コアです。CM4 コアでは、機能を登録する前に、CM0PLUS コアにダウンロードされたバイナリ・ファームウェアがその機能をサポートしているかどうかを知る必要があります。CM0PLUS バイナリが Sigfox プロトコルのみをサポートしているとすると、LoRaWAN プロトコルを登録できないことを検知する手段が MBMUX に必要です。これは、FEATURE_INFO_LIST テーブルによって実現できます。後ほど詳しく取り上げます。インストールする必要があるシステム要素として、ほかに時間サーバのタイム・ベースがあります。SubGHz のサンプルでは、このタイム・ベースが実時間コントローラ(RTC)から提供されます。両方のコアでタイム・ベースが必要なので、RTC 専用の IPCC チャンネルがあります。ソフトウェア・アーキテクチャ設計の選択により、RTC 割り込みは CM0PLUS コアでのみ扱われ、IPCC を介して CM4 に RCC 割り込みが転送されます。SynchronizeCpusAtBoot フラグが 0x9999 になると、RTC チャンネルも登録されます。この時点から、MBMUX フレームワークの中核部が初期化されます。トレース、LoRaWAN、Sigfox をはじめとする他のあらゆる機能を登録できます。

- このプレゼンテーションの冒頭でも取り上げたように、CM4 と CM0PLUS のバイナリは互いに相違
- CM4 で以下の点を知る方法
 - CM0PLUS にどの無線スタックが実装されているか。(LoRaWan、Sigfox、その他)
 - 各スタックと各機能のバージョンは何か
 - 目的のスタックと機能がどのように設定されているか(LoRaMAC がすべての領域、すべてのクラスをサポートしているかなど)
- この情報は、CM0PLUS から FEAT_INFO_List_t テーブルで提供
- このテーブルは、CM0PLUS によって SRAM2_SH2 メモリに割り当て
- CM4 は、システム機能が提供するサービスを通じて、このテーブルのアドレスを要求(前のスライドの左上部分)



CM4 コアと CM0PLUS コアのバイナリは互いに異なります。
CM4 コアでは、どの無線スタックが CM0PLUS コアに実装されているか(LoRaWAN、Sigfox、その他)、各スタックと各機能の設定のバージョンは何かを知る必要があります。
この情報は、CM0PLUS コアから FEAT_INFO_List_t テーブルで提供されます。このテーブルは、CM0PLUS コアによって SRAM2_SH2 メモリに割り当てられます。
CM4 コアでは、システム機能が提供するサービスを通じて、このテーブルのアドレスを取得します。

- 構造体を定義するヘッダ・ファイルは両方の CPU に共通
- リストにある構造体はバージョンが変わっても変更なし
- 各機能のパラメータはバージョンが変わると変更される可能性あり



```
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(LoralInfo_t),
        .Feat_Info_Config_Ptr = (void *) loralInfo
    },
    {
        .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
    },
    ...
}
```



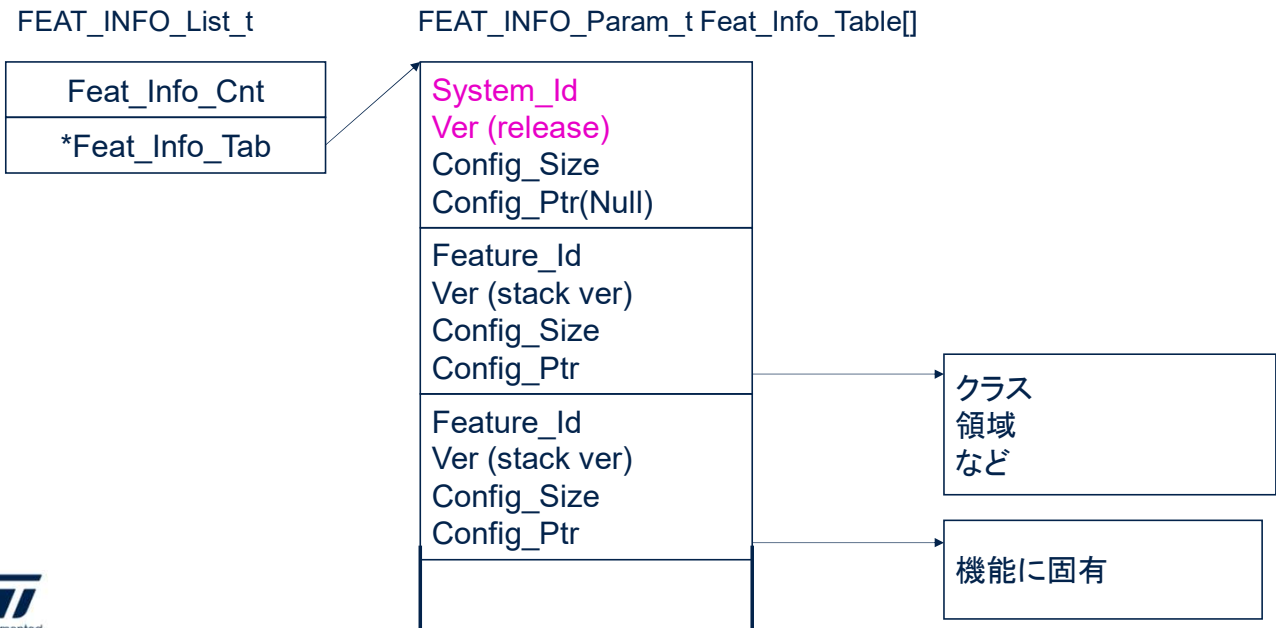
これは、CM0PLUS のバイナリがその機能をどのように公開しているかに関するコードの詳細です。

FEAT_INFO_List_t のサイズは機能の数に応じて変化します。機能の数は Feat_Info_Cnt パラメータとして設定します。

この構造体と FEAT_INFO_Param_t 型は、レビジョンが変わっても変更しないようにします。

FEAT_INFO_Param_t 型には、各機能の説明が記述されています。この説明として、機能の ID、バージョン、設定テーブルのサイズ、そのテーブルへのポインタなどがあります。各機能に関連する設定テーブルには事前定義の構造体がありません。各機能は自身を実装できます。LoRaWAN 機能を例に挙げると、設定テーブル名は LoralInfo_t で、サポートしている領域、クラス、モード (OTAA、ABP)、KMS サポートなどを記述しています。

視覚的に見た Feat_info_list_t



これは、前のスライドで紹介したコードを視覚的に表現した図です。

このテーブルに最初に記述されている機能はシステム機能です。グローバル・リリースのバージョン(v1.0.0 など)を示し、設定テーブルはありません(Null)。

他のタブはバイナリによって異なります。CM0PLUS のバイナリでサポートされている機能のみがテーブルに記述されています。LoraWAN_AT_Slave プロジェクトには、LoRaWAN、無線、トレースなどの各機能が属します。これらの機能ごとに、スタック・バージョンとそれに対応する設定テーブルへのポインタが指定されています。

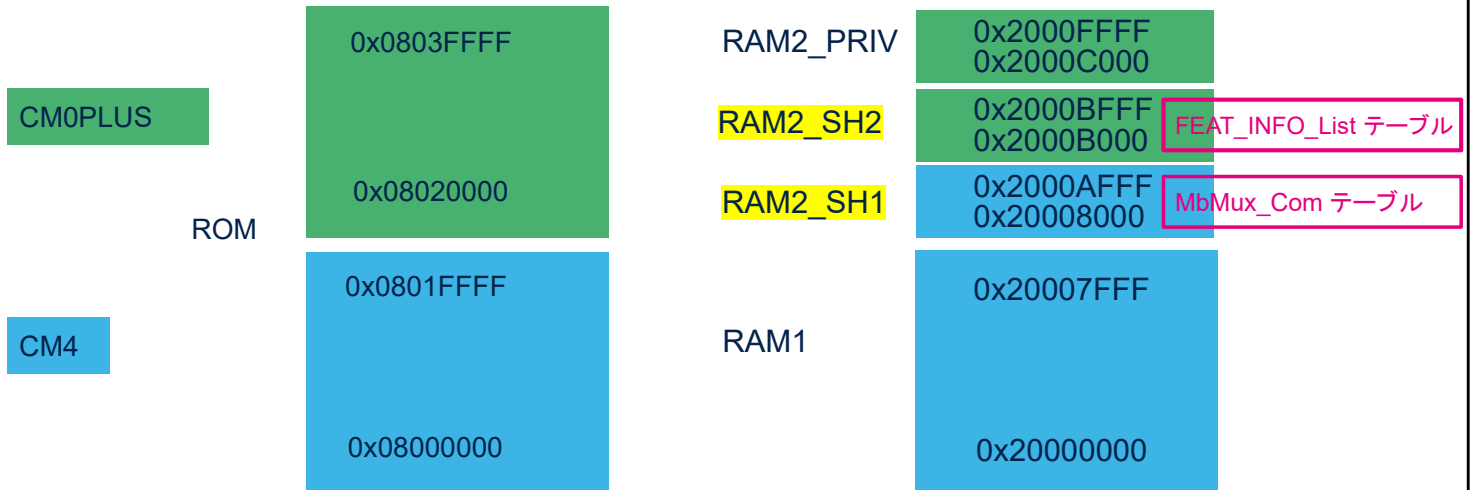
共有メモリとIPCCDBA オプション・バイト



次の付録は、FW パックによって提供されるリンカー ファイルを使用しないユーザーに特に役立ちます。実際、FW パックによって提供されるリンカー ファイルを使用しないコードは無限ループによる保護状態に入る可能性があります。

メモリ・マッピング

- v1.0.0 リリースでの EndNode 向けリンカ・ファイルの例



要約すると、MbMux_Com テーブルが CM4 コアによって割り当てられ、SRAM2_SH1 メモリに置かれます (CM4 のリンカ・ファイル)。

Feat_Info_List テーブルが CM0PLUS コアによって割り当てられ、SRAM2_SH2 メモリに置かれます (CM0PLUS のリンカ・ファイル)。

これら 2 つのコアは、相互に通信するためにこれらのテーブルのアドレスを知る必要があります。

CM0PLUS は、IPCCDBA オプション・バイトを通じて MbMux_Com テーブルのアドレスを取得します。

CM4 は、MbMux 通信を通じて Feat_Info_List テーブルのアドレスを取得します。

IPCCDBA オプション・バイト

- "SubGHz サンプル" では、CM4 のリンカ・ファイルによって MbMux_Com テーブルが 0x20008000 に置かれる
- CM4 によって置かれた MbMux_Com テーブルのアドレスを CM0PLUS が次のように取得
 - OptionsBytesStruct.IPCCdataBufAddr = SRAM1_BASE + IPCCDBA <<4;
- STM32WL デバイスでは、オプション・バイト IPCCDBA はデフォルトで 0x800
 - したがって、OptionsBytesStruct.IPCCdataBufAddr = 0x20008000 は CM4 のリンカ・ファイルと一致
- リンカ・ファイルで MbMux_Com テーブルを他の場所に置いた場合の動作
- 初期化の際に、CM4 は IPCCDBA の値を読み取り、リンカ・ファイルに記述された MbMux_Com テーブルのアドレスと比較。これらの値が一致しない場合、CM4 は IPCCDBA オプション・バイトを上書きしてボードをリブート。次回の実行ではこれらの値が一致し、CM0PLUS が新しいアドレスを取得。



life.augmented

42

IPCCDBA オプション・バイト・レジスタには MbMux_Com テーブルのアドレスが記述されています。

新しいボードでは、このレジスタのデフォルト値と、SubGHz サンプルのリンカ・ファイルに用意されている値が一致します。

それ以外のボードでは、CM4 コアによってアドレス値が更新されていることが想定できます。CM0PLUS コアは、このテーブルを使用して MbMux_Com テーブルのアドレスを取得します。

このスライドは、この処理手順を詳しく示しています。

オプション・バイトに関するプログラミング上の問題

- 現実的に、CM4 コアで IPCCDBA オプション・バイトを上書きするには、CM4/mbmuxif_sys.c ファイルのコードをユーザが若干修正する必要あり
- 実際、保護手段として無限ループが置かれているので、IPCCDBA とリンカ・ファイルの不一致があると、コードがこの無限ループを開始
- この保護手段は次の 2 つの理由で存在
 - カット 1.1では、MSI のシステム・クロック周波数が 16 MHz を超えると、このオプション・バイトのプログラミングによってバグが発生 (SubGHz アプリケーションの場合)。したがって、OBProgram 関数と OBLaunch 関数を呼び出す前に MSI 周波数が低くなるようにコードを修正する必要あり。この問題はカット 1.2 で修正。
 - リンカ・ファイルで MAPPING_TABLE を定義していないと、リンカから警告が発行されずに、SRAM1 のどこかに MbMux_Com テーブルが置かれ、それに応じて OB が設定される。この保護手段がないと、ユーザが気付かないまま、IPCCDBA オプション・バイトをプログラムが上書き。



IPCCDBA オプション・バイトが CM4 によって変更されないように、現在は保護手段が置かれています。これは、このスライドで説明している 2 つの理由によります。
ファームウェアの次回パッケージ WL_1.1.0 では、この保護手段が撤廃される予定です。

CubeMX とリンカ・ファイル

- CubeMXを使用して白紙からコードを生成する場合に使用されるスキヤッタ・ファイルは、導入パッケージ・アプリケーションに用意されているものではなく、CMSISのスキヤッタ・ファイル
 - CMSIS のデフォルトのスキヤッタ・ファイルでは MAPPING_TABLE が未定義
- これによって IPCCDBA と MbMux_Com テーブルのアドレスが一致せず、while(1) {} による保護状態に入る。この保護は、リンクとコード実行の前にリンカ・ファイルを設定することをユーザに促すもの
- リンカ・ファイルを更新すると、MX でコードを再生成しても更新済みのリンカ・ファイルは消去されない。実際、MX で CMSIS デフォルトのスキヤッタ・ファイルが使用されるのは、関係する IDE ディレクトリにスキヤッタ・ファイルが存在しない場合のみ (USER CODE SESSION と同様の原則)
- 保護の除去は、その影響を把握しているユーザのみが実施のこと



CUBEMX では MBMUX の表記がない汎用のリンカ・ファイルのみが生成される点に注意します。

したがって、CUBEMX を使用して生成したコードは、このスライドで説明しているようなリンカ・ファイルの更新やコードの修正を適用しないと、無限ループ保護のトラップに入ります。