RTA-OS

リファレンスガイド

著作権

Document: 10671-RG-5.5.1 JP-1-2016

目次

1 はじめに

RTA-OS は、静的に設定されるプリエンプティブなリアルタイムオペレーティングシステム（RTOS: Real-Time Operating System）で、制約されたリソースを最大限に活用したハイパフォーマンスなアプリケーションを実現することができます。RTA- OS には、オープン規格である AUTOSAR OS の R3.x と R4.x がフル実装され、OSEK/VDX OS 規格の V2.2.3 に完全準拠の機能が含まれています。

本書は RTA-OS の完全なテクニカルリファレンスで、おおまかに以下の 2 つの内容に分かれています。

- 本書の主要部分は RTA-OS の OS カーネル（API、型、マクロなど）についての説明で、すべてのターゲットハードウェアに共通な内容となっています。
- 本書の後方部分は RTA-OS に付属する PC ツールについての説明です。コマンドラインインターフェース、入出力ファイルフォーマットなどについての情報がまとめられています。この部分も、すべてのターゲットハードウェアに共通です。

サポートされている各ターゲットごとのポーティング情報は、『Target/Compiler Port Guide』に記載されています。

1.1 対象ユーザー

本書は、プリエンプティブなオペレーティングシステムを使用するリアルタイムアプリケーションを構築しようとする組み込みシステム開発者を対象としています。C プログラミング言語の知識は必須で、要件に合わせて選択されたツールチェーンを使用して組み込みアプリケーション用の C コードのコンパイル、アセンブル、リンクを行えることが前提条件となります。またターゲットマイクロコントローラについての基礎知識（先頭アドレス、メモリレイアウト、周辺機器のロケーションなど）も不可欠です。

さらに、Microsoft Windows オペレーティングシステムの一般的用法（ソフトウェアのインストール、メニューアイテムの選択、ボタンクリックの操作、ファイルやフォルダのナビゲートなど）も熟知している必要があります。

1.2 表記上の規約

本書では以下の表記を使用しています。

File > Open を選択します。	メニューオプションは**青い太字（bold）**で表記しています。
OK をクリックします。	ボタンのラベルは**太字（bold）**で表記しています。
<Enter>を押します。	キーボードコマンドは山括弧で括って表記しています。
“Open file”ダイアログボックスが開きます。	GUI エレメント名（ウィンドウのタイトルやフィールドなど）は、二重引用符で括って表記しています。
Activate(Task1)	プログラムコードは、すべて Typewriter フォントで表記しています。

1.3 項を参照してください。	文書内の別の箇所にジャンプするハイパーリンクは青字で表記しています。
	RTA-OS の機能のうち、他の AUTOSAR OS 実装に移植できない可能性のあるものには、RTA-OS アイコンを付けています。
	RTA-OS を正しく機能させるために必ず従わなければならない重要な指示には、警告標識を付けています。

1.3 参考資料

OSEK は欧州の自動車産業界の標準規格策定を目標とするプロジェクトで、自動車エレクトロニクス用のオープンシステムインターフェースを作成しています。OSEK 規格についての詳しい情報は以下のサイトを参照してください。

http://www.osek-vdx.org

AUTOSAR（AUTomotive Open System ARchitecture）は、標準化されたオープンな自動車用ソフトウェアアーキテクチャで、自動車メーカー、サプライヤ、ツールメーカーにより共同開発されたものです。仕様についての詳しい情報は以下のサイトを参照してください。

http://www.autosar.org

2 RTA-OS の API 関数

2.1 説明の形式

各API関数の情報は、すべて以下の構成で記述されています。

構文

```
/* C function prototype for the API call （API 関数の C 関数プロトタイプ）*/
Return Value NameOfAPICall(Parameter Type, ...)
```

引数

API 関数の引数とそのモードが示されています。

in API関数に渡される引数

out API関数に引数への参照（ポインタ）を渡し、API関数がそこに値をセットします。

inout API 関数に引数を渡し、API 関数がそれを更新します。

戻り値

API 関数が StatusType 型の値を返す場合は、値の一覧と、それぞれの値が示すエラー／警告の内容が記載されています。「ビルド」列は、標準（standard）ステータスと拡張（extended）ステータスの両方において値が戻るか（「All」）、または拡張ステータスでのみ戻るか（「拡張」）を示しています。

説明

API関数の挙動の詳細説明です。

可搬性

RTA-OS の API 関数は以下の 6 つのクラスに分かれています。

OSEK: OSEK OS の仕様に準拠する関数です。他の OSEK OS や AUTOSAR OS にポーティング可能です。

R3.x: AUTOSAR R3.x の仕様に準拠する関数です。他の AUTOSAR OS R3.x にポーティング可能です。OSEK OS にも準拠している場合に限り、他の OSEK OS にポーティング可能です。

R4.x: AUTOSAR R4.0 および R4.1 の仕様に準拠する関数です。他の AUTOSAR OS R4.x にポーティング可能です。OSEK OS にも準拠している場合に限り、他の OSEK OS にポーティング可能です。

MultiCore: AUTOSAR R4.0 multicore OS の仕様に準拠する関数です。他の AUTOSAR OS R4.x にポーティング可能です。複数のコアの構成が設定され、ターゲットバリアントが複数のコアをサポートしている場合に限り、使用可能です。

RTA-TRACE: RTA-OS が RTA-TRACE ランタイム監視ツールに関連する関数です。RTA-TRACE のサポートが設定されている場合に限り、使用可能です。

RTA-OS: 他のすべてのクラスの関数とAUTOSAR OSの拡張機能を提供する関数です。RTA-OS独自のもので、他の OS にはポーティングできません。

コーティング例

```
A C code listing showing how to use the API calls
```
（API 関数呼び出しのコーディング例）

呼び出し元コンテキスト

API 関数を呼び出せる環境が示されています。✓ の付いた環境からの呼び出しが可能で、✗ の付いた環境からは呼び出せません。

参照

関連する API 関数の一覧（リンク）です。

2.2 ActivateTask

タスクを起動します。

構文

```
StatusType ActivateTask(
   TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

この関数は StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_LIMIT	All	要求されている起動数が、コンフィギュレーションに定義されている最大起動要求キューイング数を超えてしまうため、今回の起動要求は無視されました。
E_OS_CORE	All	このタスクは、Shutdown により停止されたコアに属しています。
E_OS_ID	拡張	TaskID は有効な TaskType ではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

TaskID がサスペンド（休止）状態である場合は、レディ状態に遷移します。

TaskID がレディまたは実行状態であり、キューイングされている起動要求の合計数が最大起動数より小さい場合は、要求された起動はキューイングされます。

リスケジューリングの実行タイミングは、呼び出し元のタスクによって以下のように異なります。

- ノンプリエンプティブタスクから呼び出された場合は、呼び出し元がターミネートするか Schedule() を呼び出した後にリスケジューリングが行われます。
- プリエンプティブタスクから呼び出され、TaskID の方が優先度が高い場合は、直ちにリスケジューリングが行われます。
- カテゴリ 2 ISR から呼び出された場合は、そのカテゴリ 2 ISR がターミネートした後にリスケジューリングが行われます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
   ...
   ActivateTask(YourTask);
   ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

ChainTask
DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.3 AllowAccess

呼び出し元の OS アプリケーションの状態を APPLICATION_RESTARTING から APPLICATION_ACCESSIBLE に変更します。

構文

```
StatusType AllowAccess(void)
```

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_STATE	All	アプリケーションの状態が APPLICATION_RESTARTING ではありません。

説明

呼び出し元の OS アプリケーションの状態が APPLICATION_RESTARTING であった場合は、それを APPLICATION_ACCESSIBLE に変更します。

一般的に、再起動タスク（restart task）がこの API 関数を使用して、ターミネートされた OS アプリケーションに再びアクセスできるようにします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

例

```
TASK(AppRestarter) {
  AllowAccess();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetApplicationState
TerminateApplication

2.4 CallAndProtectFunction

タイムリミット付きで OS アプリケーション関数を呼び出します。

構文

```
StatusType CallAndProtectFunction(
    TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams,
    Os_TimeLimitType TimeLimit
)
```

引数

引数	モード	説明
FunctionIndex	in	TrustedFunctionIndexType 呼び出す関数のインデックス（宣言された関数の名前）
FunctionParams	in	TrustedFunctionParameterRefType 関数の引数へのポインタ。NULL ポインタも使用できます。
TimeLimit	in	Os_TimeLimitType 関数の実行時間のタイムミットに相当するストップウォッチのチック数。この値が 1 より小さい場合は、タイムリミットは適用されません。

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_SERVICEID	All	FunctionIndex が無効です。
E_OS_PROTECTION_TIME	All	関数がタイムアウトになったため、ProtectionHook が PRO_TERMINATETASKISR を返しました。
E_OS_PROTECTION_LOCKED	All	関数がリソースまたは割り込みをロックしていた時間が長すぎたため、ProtectionHook が PRO_TERMINATETASKISR を返しました。
E_OS_PROTECTION_MEMORY	All	関数のメモリ保護違反が発生したため、ProtectionHook が PRO_TERMINATETASKISR を返しました。
E_OS_PROTECTION_EXCEPTION	All	関数において予期しない例外が発生したため、ProtectionHook が PRO_TERMINATETASKISR を返しました。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ACCESS	拡張	この関数は別のコア上の OS アプリケーションの関数です。

説明

CallTrustedFunction と同様の機能ですが、タイミング保護（Timing Protection）の実行制限と、メモリ保護違反からの回復機能が追加されています。

関数の実行時間が所定のタイムリミットに達すると、ProtectionHook が理由 'E_OS_PROTECTION_TIME'で呼び出されます。ユーザーは ProtectionHook 内で OS アプリケーションをシャットダウンするか停止させる（kill）かを選択できます。あるいは、PRO_TERMINATETASKISR を返すと、この関数の残り部分の実行が中断され、CallAndProtectFunction は E_OS_PROTECTION_TIME というステータスを返します。

同様に、リソースや割り込みのロック違反が発生すると、この関数は 'E_OS_PROTECTION_LOCKED'を返してターミネートする可能性があり、メモリ違反が発生すると'E_OS_PROTECTION_MEMORY'または'E_OS_PROTECTION_EXCEPTION'を返してターミネートする可能性があります。

CallAndProtectFunction を使用するには、OS オプション'Function Protection'を設定しておく必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  struct func3_params {
    uint32 val1;
    uint32 val2;
  } data = {1U, 2U};
  ...
  CallAndProtectFunction(Func3,
      (TrustedFunctionParameterRefType)&data, (0.001 *
      OSSWTICKSPERSECOND)); /* Limit 1ms */
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

CallTrustedFunction
Os_Cbk_SetMemoryAccess
Os_TimeLimitType
ProtectionHook

2.5 CallTrustedFunction

OS アプリケーションの関数を呼び出します。

構文

```
StatusType CallTrustedFunction(
    TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams
)
```

引数

引数	モード	説明
FunctionIndex	in	TrustedFunctionIndexType 呼び出す関数のインデックス（宣言された関数の名前）
FunctionParams	in	TrustedFunctionParameterRefType 関数の引数へのポインタ。NULL ポインタも使用できます。

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_SERVICEID	All	FunctionIndex が無効です。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ACCESS	拡張	この関数は別のコア上の OS アプリケーションの関数です。

説明

OS アプリケーション提供の関数を呼び出します。通常このサービスは 、ノントラステッド OS アプリケーションからトラステッド OS アプリケーションの関数を呼び出す際に使用します。

RTA-OS では、ユーザーはアントラステッド（*untrusted*）OS アプリケーションに属する関数を呼び出すことができます。つまり、一連のアントラステッド関数を呼び出すトラステッドタスクを定義することができます。

呼び出された関数は、それが属している OS アプリケーションのアクセス許可に従って実行されます。

注記

AUTOSAR V4.0.3 の場合は、以下の要件が適用されます。

「タイミング保護に対応する処理を定義して、OS アプリケーションをターミネートさせることが可能です。あるタスクが CallTrustedFunction()を実行しているときに、その OS アプリケーション内でタスクのリスケジューリングが発生すると、新しく実行状態になったタスクの方が優先度が高い場合は、タイミング保護が発生して OS アプリケーションをターミネートしてしまい、間接的にトラステッド関数をアボートしてしまう可能性があります。これを防ぐため、呼び出し元と同じ OS アプリケーションに属する他のタスクのスケジューリングと、同じ OS アプリケーションの割り込みを制限する必要があります。」

ランタイムパフォーマンスへの影響を考慮して、RTA-OS はこの挙動を実装していません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask) {
  struct func3_params {
    uint32 val1;
    uint32 val2;
  } data = {1U, 2U};
  ...
  CallTrustedFunction(Func1, (TrustedFunctionParameterRefType)0U);
  CallTrustedFunction(Func2,
      (TrustedFunctionParameterRefType)&value);
  CallTrustedFunction(Func3,
      (TrustedFunctionParameterRefType)&data);
  ...
}
...
void TRUSTED_Func3(TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams) {
    struct func3_params *params = (struct func3_params
        *)FunctionParams;
    if (params->val2 < 100U) {
        ...
    }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

CallAndProtectFunction
Os_Cbk_SetMemoryAccess

2.6 CancelAlarm

アラームをキャンセルします。

構文

```
StatusType CancelAlarm(
    AlarmType AlarmID
)
```

引数

引数	モード	説明
AlarmID	in	AlarmType アラームの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_NOFUNC	All	AlarmID は実行中ではありません。
E_OS_ID	拡張	AlarmID は有効なアラームではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから AlarmID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっているときに呼び出されました（サービス保護が設定されている場合のみ）。

説明

この関数は指定されたアラームをキャンセル（停止）します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyExtendedTask){
  ...
  CancelAlarm(TimeOutAlarm);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

CancelAlarm
DeclareAlarm
TickRefTypeGetAlarm
GetAlarmBase
SetRelAlarm

2.7 ChainTask

呼び出し元タスクをターミネートして、別のタスクを起動します。

構文

```
StatusType ChainTask(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OS_LIMIT	All	要求されている起動の数が、コンフィギュレーションに定義されている最大起動要求キューイング数を超えてしまうため、今回の起動要求は無視されました。
E_OS_CORE	All	このタスクは、Shutdown により停止されたコアに属しています。
E_OS_ID	拡張	TaskID は有効な TaskType ではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。
E_OS_RESOURCE	拡張	呼び出し元のタスクがまだリソースを保持しています。
E_OS_CALLEVEL	拡張	割り込みレベルで呼び出されました。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

このサービスは呼び出し元のタスクをターミネートさせてから TaskID を起動します。ChainTask()の呼び出しが成功した場合は、呼び出し元に戻りません。

呼び出し元のタスクが保持していた内部リソースは自動的に解放されます。

呼び出し元のタスクが保持していた標準リソースまたはリンクリソースも自動的に解放され、これは拡張ステータスではエラー条件としてレポートされます。

タスクからそのタスク自体へのチェーニングは、キューイングされた起動数に影響を与えません。

ChainTask()が実行されると必ずリスケジューリングが発生しますが、レディキュー内にはTaskIDよりも優先して実行されるタスク（たとえば内部リソースをTaskIDと共有し、TaskIDよりも優先度の高いタスクなど）が存在する可能性があるため、TaskID が直ちに実行されるとは限りません。

RTA-OS 用の最適化（Optimizations）で 'Fast Terminate' が有効になっている場合は、ChainTask()は必ずタスクのエントリ関数から呼び出されなければならず、リターンステータスはチェックされません。エラーの場合は、ErrorHook が設定されていればそれが呼び出されます。この最適化によりメモリと実行時間が節約されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ... ChainTask(YourTask);
  /* Any code here will not execute if the call is successful */
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

ActivateTask
DeclareAlarm
GetTaskID
GetTaskState
TerminateTask

2.8 CheckISRMemoryAccess

指定された ISR からメモリ領域への各種アクセス（読み取り、書き込み、実行、スタック）が可能かどうかを調べます。

構文

```
AccessType CheckISRMemoryAccess(
    ISRType ISRID,
    MemoryStartAddressType Address,
    MemorySizeType Size
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID
Address	in	MemoryStartAddressType メモリ領域の先頭アドレスへのポインタ
Size	in	MemorySizeType メモリ領域のサイズ（バイト数）

戻り値

AccessType 型の値を返します。

説明

ISRID が有効な ISR を表している場合は、CheckISRMemoryAccess()は Address から (Address+Size) までのアドレス範囲のメモリ領域について、以下のタイプのアクセスの可否を判定します。

- ISRID による読み取り
- ISRID による書き込み
- ISRID による実行
- スタック空間としてのアクセス

あるタイプのメモリアクセス条件が、指定されたメモリ領域の全体について有効でない場合は、そのタイプについて「アクセスなし」をレポートします。たとえば指定された範囲内に書き込み可能でない領域のアドレスが含まれている場合は、「その範囲は書き込み不可」とレポートします。

このサービスを呼び出すと、OS が Os_Cbk_CheckMemoryAccess()を呼び出します。

この呼び出しの結果は AccessType に符号化され、AccessType は OSMEMORY_IS_*マクロにより復号できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ISR(MyISR){
  if (OSMEMORY_IS_WRITEABLE(CheckISRMemoryAccess(MyISR, &datum,
      sizeof(datum)))) {
    datum = ...
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

CheckTaskMemoryAccess
Os_Cbk_CheckMemoryAccess

2.9 CheckObjectAccess

OS アプリケーションが OS オブジェクトにアクセスできるかどうかを判定します。

構文

```
ObjectAccessType CheckObjectAccess(
    ApplicationType ApplID,
    ObjectTypeType ObjectType,
    Os_AnyType Object
)
```

引数

引数	モード	説明
ApplID	in	ApplicationType アプリケーションの ID
ObjectType	in	ObjectTypeType オブジェクトのタイプ（OBJECT_TASK、OBJECT_ISR、OBJECT_ALARM、OBJECT_RESOURCE、OBJECT_COUNTER または OBJECT_SCHEDULETABLE）
Object	in	Os_AnyType オブジェクトの ID

戻り値

この関数はObjectAccessType型の値を返します。

値	ビルド	説明
NO_ACCESS	All	OS アプリケーションはこのオブジェクトにアクセスできません。または、Object か ObjectType が無効です。
ACCESS	All	OS アプリケーションはこのオブジェクトにアクセスできます。

説明

指定された OS オブジェクトに AppID がアクセスできる場合は ACCESS を返し、そうでない場合は NO_ACCESS を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
if (CheckObjectAccess(GetApplicationID(), OBJECT_TASK, Task1) ==
    ACCESS) {
  ActivateTask(Task1);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

CheckObjectOwnership

2.10 CheckObjectOwnership

オブジェクトを所有している OS アプリケーションを取得します。

構文

```
ApplicationType CheckObjectOwnership(
    ObjectTypeType ObjectType,
    Os_AnyType Object
)
```

引数

引数	モード	説明
ObjectType	in	ObjectTypeType オブジェクトのタイプ（OBJECT_TASK、OBJECT_ISR、OBJECT_ALARM、OBJECT_RESOURCE、OBJECT_COUNTER または OBJECT_SCHEDULETABLE）
Object	in	Os_AnyType オブジェクトの ID

戻り値

ApplicationType 型の値を返します。

値	ビルド	説明
INVALID_OSAPPLICATION	All	無効な Object または ObjectType

説明

この関数は Object を所有している OS アプリケーションの識別子を返しますが、OS アプリケーションが所有しているオブジェクトと ObjectType および Object とがマッチしない場合は INVALID_OSAPPLICATION を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ApplicationType OwningApplication =
    CheckObjectOwnership(OBJECT_TASK, Task1);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

CheckObjectAccess

2.11 CheckTaskMemoryAccess

指定されたタスクからメモリ領域への各種アクセス（読み取り、書き込み、実行、スタック）が可能かどうかを調べます。

構文

```
AccessType CheckTaskMemoryAccess(
    TaskType TaskID,
    MemoryStartAddressType Address,
    MemorySizeType Size
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID
Address	in	MemoryStartAddressType メモリ領域の先頭アドレスへのポインタ
Size	in	MemorySizeType メモリ領域のサイズ（バイト数）

戻り値

AccessType 型の値を返します。

説明

TaskID が有効な ISR を表している場合は、Address から (Address+Size) までの範囲のメモリ領域について、以下のタイプのアクセスの可否を判定します。

- TaskID による読み取り
- TaskID による書き込み
- TaskID による実行
- スタック空間としてのアクセス

あるタイプのメモリアクセス条件が、指定されたメモリ領域の全体について有効でない場合は、そのタイプについて「アクセスなし」をレポートします。たとえば指定された範囲内に書き込み可能でない領域のアドレスが含まれている場合は、「その範囲は書き込み不可」とレポートします。

このサービスを呼び出すと、OS が Os_Cbk_CheckMemoryAccess()を呼び出します。

この呼び出しの結果は AccessType に符号化され、AccessType は OSMEMORY_IS_*マクロにより復号できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  if (OSMEMORY_IS_WRITEABLE(CheckTaskMemoryAccess(MyTask, &datum,
      sizeof(datum)))) {
    datum = ...
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

CheckISRMemoryAccess
OSMEMORY_IS_EXECUTABLE
OSMEMORY_IS_READABLE
OSMEMORY_IS_STACKSPACE
OSMEMORY_IS_WRITEABLE
Os_Cbk_CheckMemoryAccess
Os_Cbk_SetMemoryAccess

2.12 ClearEvent

1 つまたは複数のイベントを、タスクのイベントマスクに従ってクリアします。

構文

```
StatusType ClearEvent(
    EventMaskType Mask
)
```

引数

引数	モード	説明
Mask	in	EventMaskType クリアするイベントを表すマスク

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ACCESS	拡張	拡張タスクから呼び出されていません。
E_OS_CALLEVEL	拡張	割り込みレベルから呼び出されました。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

ClearEvent を呼び出した拡張タスクのイベントが、イベントマスク Mask に従ってクリアされます。

イベントマスクにセットされていないイベントの状態は変更されません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyExtendedTask){
  EventMaskType WhatHappened;
  while (WaitEvent(Event1 | Event2 | Event3) == E_OK ) {
    GetEvent(MyExtendedTask, &WhatHappened);
    if (WhatHappened & Event1) {
      ClearEvent(Event1);
      /* Take action on Event1 */
      ...
```

```
    } else if (WhatHappened & (Event2 | Event3) {
      ClearEvent(Event2 | Event3);
      /* Take action on Event2 or Event3 */
      ...
    }
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareEvent
GetEvent
SetEvent
WaitEvent

2.13 ControlIdle

コアをアイドルモードにします。

構文

```
StatusType ControlIdle(
    CoreIdType CoreID,
    IdleModeType IdleMode
)
```

引数

引数	モード	説明
CoreID	in	CoreIdType コアの ID
IdleMode	in	IdleModeType セットされるモード

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	コアが無効です。

説明

コアをアイドルモードにします。現在実行中のコアにセットされている値は、OS_CURRENT_IDLEMODE()マクロにより読み取ることができます。このAPI関数ControlIdleは AUTOSAR V4.1.x 用に追加されたものですが、RTA-OS ではそれより前のバージョンにも使用できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
/* Set user-defined value */
ControlIdle(OS_CORE_ID_MASTER, 3);
/* Set AUTOSAR-defined value */
ControlIdle(OS_CORE_ID_1, IDLE_NO_HALT);

/* Read mode for this core */
idle_mode = OS_CURRENT_IDLEMODE();
```

参照

OS_CURRENT_IDLEMODE

2.14 DisableAllInterrupts

ハードウェアがディセーブルすることができるすべての割り込みをマスクして、ディセーブル状態にします。

構文

```
void DisableAllInterrupts(void)
```

説明

コード内の「クリティカルセクション」を実行する際に呼び出す API 関数です。クリティカルセクションは EnableAllInterrupts()の呼び出しによって終了します。クリティカルセクション内では API 関数を呼び出せません。

この API 関数はネスティングをサポートしていません。ライブラリ用などでクリティカルセクションのネスティングが必要な場合は、SuspendAllInterrupts()/ResumeAllInterrupts()を使用してください。

マルチコア環境では、この API 関数は自身の呼び出しが行われたコア上の割り込みだけに影響します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ... DisableAllInterrupts();
    /* Critical section */
    /* No RTA-OS API calls allowed */
  EnableAllInterrupts();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

EnableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.15 EnableAllInterrupts

すべての割り込みのマスクをクリアして、イネーブル状態にします。

構文

```
void EnableAllInterrupts(void)
```

説明

マスク可能な割り込みがすべてマスクされたクリティカルセクションを終了します。このクリティカルセクションは、必ず DisableAllInterrupts()の呼び出しによって開始されていなければなりません。

このAPI関数はDisableAllInterrupts()において保存されていた割り込みマスクの状態を復元します。

マルチコア環境では、この API 関数は自身の呼び出しが行われたコア上の割り込みだけに影響します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ... DisableAllInterrupts();
  /* Critical section */
  /* No RTA-OS API calls allowed */
  EnableAllInterrupts();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

DisableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.16 GetActiveApplicationMode

アクティブなアプリケーションモードを取得します。

構文

```
AppModeType GetActiveApplicationMode(void)
```

戻り値

AppModeType 型の値を返します。

説明

現在アクティブになっているアプリケーションモード（StartOS()に渡された引数の値）を返します。アプリケーションモードに依存するコードを作成する際にこの API 関数を使用します。

OS が実行中でない場合は、OS_NOAPPMODE を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  if (GetActiveApplicationMode() == DiagnosticsMode) {
    /* Send diagnostic message */
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

StartOS

2.17 TickRefTypeGetAlarm

アラームが満了するまでのチック数を取得します。

構文

```
StatusType GetAlarm(
    AlarmType AlarmID,
    TickRefType Tick
)
```

引数

引数	モード	説明
AlarmID	in	AlarmType アラームの ID
Tick	out	TickRefType TickType 変数への参照

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_NOFUNC	All	AlarmID は現在セットされていません。
E_OS_ID	拡張	AlarmID は有効なアラームではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから AlarmID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	Tick が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

呼び出された時点からアラーム AlarmID が満了するまでのチック数を返します。

このAPI関数を呼び出した後、出力引数Tickを評価する前にタスクがプリエンプトされると、評価の時点においてアラームがすでに満了している可能性があります。Tick の値に基づいて処理を決定する際には注意が必要です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
```

```
  TickType TicksToExpiry;
  ...
  GetAlarm(MyAlarm, &TicksToExpiry);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

CancelAlarm
DeclareAlarm
GetAlarmBase
SetAbsAlarm
SetRelAlarm

2.18 GetAlarmBase

アラームに割り当てられているカウンタのプロパティを取得します。

構文

```
StatusType GetAlarmBase(
    AlarmType AlarmID,
    AlarmBaseRefType Info
)
```

引数

引数	モード	説明
AlarmID	in	AlarmType 当該アラームの ID
Info	out	AlarmBaseRefType AlarmBaseType 構造体への参照

戻り値

StatusType型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	AlarmID は有効なアラームではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから AlarmID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	Info が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

アラームの基本特性を読み取ります。基本特性は AlarmID に割り当てられているカウンタの静的特性です。

出力引数 Info は、AlarmBaseType データ型の情報が格納されている構造体を参照します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  AlarmBaseType Info;
  TickType maxallowedvalue;
  TickType ticksperbase;
  TickType mincycle;

  GetAlarmBase(MyAlarm, &Info);
  maxallowedvalue = Info.maxallowedvalue;
  ticksperbase = Info.ticksperbase;
  mincycle = Info.mincycle;
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

CancelAlarm
DeclareAlarm
TickRefTypeGetAlarm
SetAbsAlarm
SetRelAlarm

2.19 GetApplicationID

現在実行中のタスク、ISR、フックを所有している OS アプリケーションの識別子を取得します。

構文

```
ApplicationType GetApplicationID(void)
```

戻り値

ApplicationType 型の値を返します。

説明

現在実行中のフック、タスク、カテゴリ 2 ISR を所有している OS アプリケーションを返します。

アクティブな OS アプリケーションがない場合は INVALID_OSAPPLICATION を返します。

返される値は、CallTrustedFunction 実行中は変わりません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
if (GetApplicationID() == App1) {
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

CallTrustedFunction
GetCurrentApplicationID
GetISRID
GetTaskID

2.20 GetApplicationState

指定された OS アプリケーションの状態を取得します。

構文

```
StatusType GetApplicationState(
    ApplicationType Application,
    ApplicationStateRefType Value
)
```

引数

引数	モード	説明
Application	in	ApplicationType 状態がリクエストされる OS アプリケーション。
Value	out	ApplicationStateRefType アプリケーションの現在の状態。

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	All	Application は有効な OS アプリケーションではありません。

説明

指定された OS アプリケーションの状態を返します。状態は、StartOS の処理の中で APPLICATION_ACCESSIBLE になります。

アプリケーションがターミネートされて再起動されないと、状態は APPLICATION_TERMINATED に変わります。

ProtectionHook、または RESTART 指定の TerminateApplication の後は、状態は APPLICATION_RESTARTING になっている可能性があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

例

```
ApplicationStateType appState;
if (GetApplicationState(App1, &appState) == E_OK) {
  switch (appState) {
    case APPLICATION_ACCESSIBLE:
      ...
    case APPLICATION_RESTARTING:
      ...o
    case APPLICATION_TERMINATED:
      ...
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

AllowAccess
TerminateApplication

2.21 GetCoreID

呼び出し元のコアの一意の論理 CoreID を返します。

構文

```
CoreIdType GetCoreID(void)
```

戻り値

CoreIdType 型の値を返します。

説明

この関数を呼び出したコアの一意の論理 CoreID を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
CoreIdType core_id = GetCoreID();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

StartCore
StartNonAutosarCore

2.22 GetCounterValue

カウンタの値を取得します。

構文

```
StatusType GetCounterValue(
    CounterType CounterID,
    TickRefType Value
)
```

引数

引数	モード	説明
CounterID	in	CounterType カウンタのＩＤ
Value	out	TickRefType カウンタのカレント値

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	CounterID は有効なカウンタではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから CounterID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	値が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）

説明

指定されたカウンタ（CounterID）のカレント値を返します。

オペレーティングシステムは、カウンタ値の最小値をゼロとし、ラップアラウンドするまでインクリメントされる値を返します。

CounterID がハードウェアカウンタである場合は、ユーザーコールバック関数 Os_Cbk_Now_<CounterID>が呼び出されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
Task(MyTask){
  TickType Value;
  ...
  GetCounterValue(MyCounter,&Value);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetElapsedCounterValue

GetElapsedValue
IncrementCounter
Os_AdvanceCounter
Os_AdvanceCounter_<CounterID>
Os_IncrementCounter_<CounterID>

2.23 GetCurrentApplicationID

現在実行中の OS アプリケーションの ID（識別子）を取得します。

構文

```
ApplicationType GetCurrentApplicationID(void)
```

戻り値

ApplicationType 型の値を返します。

説明

現在実行中の OS アプリケーション、つまり現在実行中のフック、タスク、カテゴリ 2 ISR、トラステッド関数を所有している OS アプリケーションを返します。

アクティブな OS アプリケーションがない場合は INVALID_OSAPPLICATION を返します。

返される値は CallTrustedFunction の実行に影響されます。つまり CallTrustedFunction により呼び出されたトラステッド関数を所有する OS アプリケーションが対象となります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
if (GetCurrentApplicationID() == App1) {
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

CallTrustedFunction
GetApplicationID
GetISRID
GetTaskID

2.24 GetElapsedCounterValue

<Value>以降に経過したチック数を<ElapsedValue>で返します。

構文

```
StatusType
    GetElapsedCounterValue(
    CounterType CounterID,
    TickRefType Value,
    TickRefType ElapsedValue
)
```

引数

引数	モード	説明
CounterID	in	CounterType カウンタの ID
Value	in	TickRefType 前回のカウンタ値
Value	out	TickRefType 現在のカウンタ値
ElapsedValue	out	TickRefType 入力されたカウンタ値との差

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	CounterID は有効なカウンタではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから CounterID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	Value または ElapsedValue が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

前回の Value から現在のカウンタ値までの経過チック数を返します。

この API 関数から戻ると、Value は現在のカウンタ値に更新されています。

この API 関数は maxallowedvalue チック以下の値しか返せません。

前回の Value からの経過チック数が maxallowedvalue チックより大きい場合は、ElapsedValue の値は現在の値を maxallowedvalue で割った後の余りになります。

この API 関数の名前は AUTOSAR V4 において GetElapsedValue に変更されましたが、互換性の確保のため、RTA-OS では AUTOSAR V4 のアプリケーションにはどちらの名前も使用でき、実装された関数名へのマッピングマクロが用意されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
Task(MyTask){
  TickType Value;
  TickType ElapsedValue;
  ...
  GetCounterValue(MyCounterID,&Value);
  /* Value => current count */
  ...
  GetElapsedCounterValue(MyCounter,&Value,&ElapsedValue);
  /* ElapsedValue => ticks since original Value, Value => current
      count */
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetCounterValue

GetElapsedValue

2.25 GetElapsedValue

入力された<Value>以降に経過したチック数を<ElapsedValue>で返します。

構文

```
StatusType GetElapsedValue(
    CounterType CounterID,
    TickRefType Value,
    TickRefType ElapsedValue
)
```

引数

引数	モード	説明
CounterID	in	CounterType カウンタの名前
Value	in	TickRefType 前回のカウンタ値
Value	out	TickRefType 現在のカウンタ値
ElapsedValue	out	TickRefType 入力されたカウンタ値との差

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	CounterID は有効なカウンタではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから CounterID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	Value または ElapsedValue が、カレント OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

前回の Value から現在のカウンタ値までの経過チック数を返します。

この API 関数から戻ると、Value は現在のカウンタ値に更新されています。

この API 関数は maxallowedvalue チック以下の値しか返せません。

前回の Value からの経過チック数が maxallowedvalue チックより大きい場合は、ElapsedValue の値は現在の値を maxallowedvalue で割った後の余りになります。

この API 関数の名前は AUTOSAR V3 では GetElapsedCounterValue でしたが、互換性の確保のため、RTA-OS では AUTOSAR V4 のアプリケーションにはどちらの名前も使用でき、実装された関数名へのマッピングマクロが用意されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
Task(MyTask){
  TickType Value;
  TickType ElapsedValue;
  ...
  GetCounterValue(MyCounterID,&Value);
  /* Value => current count */
  ...
  GetElapsedValue(MyCounter,&Value,&ElapsedValue);
  /* ElapsedValue => ticks since original Value, Value => current
      count */
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetCounterValue
GetElapsedCounterValue

2.26 GetEvent

タスクのイベントビットの状態を取得します。

構文

```
StatusType GetEvent(
    TaskType TaskID,
    EventMaskRefType Event
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID
Event	out	EventMaskRefType イベントマスクへの参照

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	TaskID は有効なタスクではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。
E_OS_ACCESS	拡張	TaskID は拡張タスクではありません。
E_OS_STATE	拡張	TaskID はサスペンド状態になっています。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	Event が、カレント OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

このAPI関数は拡張タスクTaskIDに対してセットされているすべてのイベントを返します。

当該タスクが現在どのイベントを待っているかに関わらず、セットされているすべてのイベントが返されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyExtendedTask){
  EventMaskType WhatHappened;
  while (WaitEvent(Event1 | Event2 | Event3) == E_OK ) {
    GetEvent(MyExtendedTask, &WhatHappened);
    if(WhatHappened & Event1) {
      ClearEvent(Event1);
      /* Take action on Event1 */
      ...
    } else if (WhatHappened & (Event2 | Event3) {
      ClearEvent(Event2 | Event3);
      /* Take action on Event2 or Event3 */
      ...
    }
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

ClearEvent
DeclareEvent
SetEvent
WaitEvent

2.27 GetISRID

現在実行中の ISR の識別子を取得します。

構文

```
ISRType GetISRID(void)
```

戻り値

ISRType 型の値を返します。

説明

現在実行中のカテゴリ 2 ISR の ID を返します。実行中の ISR がない場合は INVALID_ISR を返します。

一般的に、フック関数内でどの ISR が実行されているかを特定する目的で使用します。

マルチコア環境では、この API 関数は呼び出し元のコアだけを調べます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  ISRType ISRInError;
  TaskType TaskInError;

  ISRInError = GetISRID();
  if (ISRInError != INVALID_ISR) {
    /* Must be an ISR in error */
  } else {
    /* Maybe it's a task in error */
    GetTaskID(&TaskInError);
    ...
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

GetTaskID
OS_ISRTYPE_TO_INDEX

2.28 GetNumberOfActivatedCores

API 関数 StartCore により起動されたコアの数を取得します。

構文

```
uint32 GetNumberOfActivatedCores(void)
```

戻り値

uint32 型の値を返します。

説明

StartCore()を使用して起動されたコア（OS を実行するように設定されているコアに限られます）の数を返します。

RTA-OS では、StartOS()より前にこの GetNumberOfActivatedCores()を呼び出すことができます。このタイミングで呼び出された場合に限り、GetNumberOfActivatedCores()の戻り値がコンフィギュレーションに設定されている数値より小さくなる可能性があります。

この API 関数は、マルチコア構成のプロジェクトにしか提供されません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
OS_MAIN(){
  StatusType status;
  CoreIdType core_id = GetCoreID()
  ...
  if (core_id == OS_CORE_ID_MASTER) {
    while (GetNumberOfActivatedCores() < OS_NUM_CORES) {
      StartCore(++core_id, &status);
    }
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

StartCore
StartOS

2.29 GetResource

クリティカルセクションを開始する目的でリソースを取得（ロック）します。

構文

```
StatusType GetResource(
    ResourceType ResID
)
```

引数

引数	モード	説明
ResID	in	ResourceType リソースの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ResID は有効なリソースではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ResID にアクセスできません。
E_OS_ACCESS	拡張	以下のいずれかです。 (a) すでに別のタスク／ISR によってロックされているリソースを取得しようとしました。 (b) 呼び出し元のタスクまたは割り込みルーチンの優先度が ResID の実際の優先度よりも高くなっています。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

名前付きのクリティカルセクション（リソース）を開始し、そのリソースにアクセスできるように設定されている他のタスクとISRによるアクセスからクリティカルセクション内のコードを保護します。

クリティカルセクションを終了するには ReleaseResource()を使用します。

リソース占有をネストさせることも可能です。その場合は、例に示すように、外側のクリティカルセクションの中で内側のクリティカルセクションの実行が完結しなければなりません。

同じリソースの占有をネストさせることはできませんが、リンクリソースを使用すればそれと同様の効果を得ることができます。

クリティカルセクション内では、実行中のタスクの状態を別の状態に遷移させる API 関数（ChainTask()、Schedule()、TerminateTask()、WaitEvent()など）は使用できません。

カテゴリ 2 ISR がリソースをロックできるシステムでは、タスクだけがリソースをロックできるシステムよりもランタイムオーバーヘッドが若干大きくなります。

リソースは、複数のコアにまたがって使用できるように設定することはできません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ... GetResource(Outer);
    /* Outer Critical Section */
    ...
    GetResource(Inner);
      /* Inner Critical Section */
    ReleaseResource(Inner);
    ...
  ReleaseResource(Outer);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareResource
ReleaseResource

2.30 GetScheduleTableStatus

スケジュールテーブルの状態を取得します。

構文

```
StatusType GetScheduleTableStatus(
    ScheduleTableType ScheduleTableID,
    ScheduleTableStatusRefType ScheduleStatus
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID
ScheduleStatus	out	ScheduleTableStatusRefType スケジュールテーブルの状態への参照

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ScheduleTableID は有効なスケジュールテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	ScheduleStatus が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

ScheduleTableID の状態を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  ScheduleTableStatusType Status;

  GetScheduleTableStatus(MyScheduleTable, &Status);
  if (Status != SCHEDULETABLE_RUNNING){
      StartScheduleTableAbs(MyScheduleTable,42);
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
NextScheduleTable
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.31 GetSpinlock

スピンロック変数の占有を試みます。

構文

```
GetSpinlock(
    SpinlockIdType SpinlockId
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	SpinlockId は有効なスピンロックを参照していません。
E_OS_STATE	拡張	スピンロックは呼び出し元のタスク／ISR によりすでに占有されています（AUTOSAR V4.0.3 より前）。
E_OS_INTERFERENCE_DEADLOCK	拡張	スピンロックは同じコア上のタスク／ISR によりすでに占有されているため、デッドロックが発生します。
E_OS_NESTING_DEADLOCK	拡張	別のスピンロックをすでに保持しているときにこのスピンロックを占有しようとしました。デッドロックを招く可能性があります。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから SpinlockId にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。

説明

この API 関数は、共有リソース（通常は書き込み可能領域）への排他的アクセスを 1 つのコアに与えるためのものです。

1 つのスピンロックは一度に 1 つのコアだけが占有することができます。別のコアに占有されているスピンロックの占有をリクエストすると、現在占有しているコアがロックを解除するまでは、後からリクエストした側はビジーウェイト状態になります。

ロックの解除には ReleaseSpinlock を使用します。

スピンロックを占有しているコードが、より優先度の高いタスクまたは ISR によりプリエンプトされると、他のコア上のコードがそのスピンロックを取得するには、プリエンプトされ

たコードが完了するのを待つ必要があります。これではシステムの応答に悪影響が出る可能性があるので、そのような挙動を防ぐため、リソースを使用するか、スピンロックの前後にSuspendOSInterrupts/ResumeOSInterrupts を使用することができます。

また、プリエンプトする側のコードがそれより優先度の低いタスク／ISR によってすでに占有されているスピンロックを占有しようとすると、OS はその処理を行わずにエラーを発行します。これは、プリエンプトする側によるロックは絶対に成功しないので、恒久的にビジーウェイト状態になってしまうためです。

スピンロックへのアクセスがネストする場合は注意が必要です。ネストしてロックされる順序を正しく設定してデッドロックを防ぐようにしなければなりません。

GetSpinlock は、スピンロックの占有に成功した場合にだけリターンします。リターンしない場合はエラーが発生していることになります。

スピンロックのロックメソッド（コンテナ: OsSpinlock）のタイプに応じて、この API 関数の挙動は以下のように変わります。

- LOCK_ALL_INTERRUPTS ― SuspendAllInterrupts()を呼び出して終了します。SuspendAllInterrupts()に適用される使用制限が GetSpinlock()にも適用されます。
- CK_CAT2_INTERRUPTS ― SuspendOSInterrupts()を呼び出して終了します。SuspendOSInterrupts()に適用される使用制限が GetSpinlock()にも適用されます。
- LOCK_WITH_RES_SCHEDULER ― GetResource(RES_SCHEDULER) を呼び出して終了します。GetResource(RES_SCHEDULER)に適用される使用制限が GetSpinlock()にも適用されます。
- NESTABLE ― 同じコア上のタスク／ISR によってスピンロックがすでに保持されていると、「ロック成功」を知らせます。ロックの解除は、そのロックを行ったコードが、実行した Get コールと同じ数の Release コールを実行したときに限り行われます。
- COMMONABLE ― スピンロックの後に、独自のサクセッサを持たず COMMONABLE でもない任意のスピンロックを実行することができます。このスピンロックは、通常のロックの後に、そのサクセッサリストに載らずに実行することができます。

OS コンフィギュレーションオプション'Force spinlock error checks'を使用すると、拡張ステータスのビルド以外に標準ステータスのビルドでもエラーチェックが行われます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
TASK(MyTask){
  ...
  GetSpinlock(Spinlock1);
  ...
  ReleaseSpinlock(Spinlock1);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetResource
ReleaseSpinlock
SuspendAllInterrupts
SuspendOSInterrupts
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.32 GetSpinlockInfo

スピンロックのランタイム統計情報を取得します。

構文

```
StatusType GetSpinlockInfo(
    SpinlockIdType SpinlockId,
    Os_SpinlockInfoRefType Info
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの識別子
Info	out	Os_SpinlockInfo スピンロックの統計情報への参照

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	SpinlockId は有効なスピンロックを参照していません。
E_OS_ILLEGAL_ADDRESS	拡張	Info が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

GetSpinlockInfo はオプションの API 関数で、アプリケーション内にスピンロックがあり、'Provide spinlock statistics'という OS オプションが有効になっている場合に限り使用できます。

この API 関数は、指定の SpinlockId について、ロックに関するランタイム統計情報を取得するためのものです。

この API 関数を呼び出したときにスピンロックがロックされていると、Info の各フィールドに以下のような情報がセットされます。

- CurrentLockTime フィールド ― このスピンロックがロックされてから経過したチック数が格納されます。ロックされていない場合はゼロになります。
- CurrentLocker フィールド ― このスピンロックをロックした TaskType のタスクまたは ISRType の ISR が格納されます。ロックされていない場合は INVALID_TASK になります。
- CurrentLockingCore フィールド ― このスピンロックをロックしたコアの番号が格納されます。ロックされていない場合は OS_NUM_CORES になります。

- LockAttemptsフィールド ー このスピンロックについて行われたTry/Get試行数(コア当たりの数）が格納されます。
- LockSucceeds フィールド ー ロックに成功した Try/Get 試行数（コア当たりの数）が格納されます。
- LockFails フィールド ー ロックに失敗した Try 試行数と 1 回の Get に対するリトライ数とを加えたもの（コア当たりの数）が格納されます。
- MaxLockTime フィールド ー 各コアについて、このスピンロックがロックされていた最長時間に相当するチック数が格納されます。
- MaxLockTimeLocker フィールド ー MaxLockTime に該当するロックを行った TaskType のタスクまたは ISRType の ISR が格納されます。
- MaxSpinTime フィールド ー 各コアについて、このスピンロックの取得にかかった最長時間に相当するチック数を示します。一般的にこの時間は、他のコアがそのロックを解除するのを GetSpinlock 内で待っていた時間に相当します。
- MaxSpinTimeLocker フィールド ー MaxSpinTime に該当するロック試行を行った TaskType のタスクまたは ISRType の ISR が格納されます。

OS コンフィギュレーションオプション'Force spinlock error checks'を使用すると、拡張ステータスのビルド以外に標準ステータスのビルドでもエラーチェックが行われます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.CurrentLocker == MyTask) {
    ...
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetSpinlock
Os_SpinlockInfo
ReleaseSpinlock

ResetSpinlockInfo
TryToGetSpinlock

2.33 GetTaskID

現在実行中のタスクを識別します。

構文

```
StatusType GetTaskID(
    TaskRefType TaskID
)
```

引数

引数	モード	説明
TaskID	out	TaskRefType 実行中のタスクへの参照

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	TaskID が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

この API 関数は現在実行中のタスクへの参照を返します。

タスクから呼び出された場合は、そのタスクの識別子を返します。

ISR から実行された場合は、その割り込みが発生したときに実行中だったタスクの識別子を返します。

この API 関数は主に、実行中のタスクをフック関数内で識別する際に使用します。

マルチコア環境では、この API 関数を呼び出したコアだけが処理対象となります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  TaskType TaskInError;
  GetTaskID(&TaskInError);
```

```
  if (TaskInError == INVALID_TASK) {
    /* Must be an ISR in error */
  } else if (TaskInError == MyTask) {
    /* Do something */
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

DeclareTask
GetISRID
GetTaskID
GetTaskState
OS_TASKTYPE_TO_INDEX
TerminateTask

2.34 GetTaskState

指定されたタスクの現在の状態（サスペンド、レディ、実行中、待ち）を取得します。

構文

```
StatusType GetTaskState(
    TaskType TaskID,
    TaskStateRefType State
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID
State	out	TaskStateRefType タスクの状態への参照

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	TaskID は有効な TaskType ではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_ILLEGAL_ADDRESS	拡張	State が現在の OS アプリケーションから書き込めない領域にあります（アントラステッド OS アプリケーションがある場合のみ）。

説明

この API 関数が呼び出された時点における、指定のタスクの状態を返します。

この API 関数は主に、イベントをセットする前に拡張タスクがサスペンド状態ではないことを確認する際に使用します。

ISR によってプリエンプトされているタスクは「実行中」となります。

プリエンプティブタスクまたは ISR から呼び出された場合は、この API 関数がリターンしてから結果が評価されるまでの間にプリエンプションが発生する可能性があるため、取得された「状態」は、評価の時点において正しくなくなっている場合があります。

マルチコア環境では、別のコアに割り当てられているタスクも含むすべてのタスクの状態を調べることができます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  TaskStateType CurrentState;
  ...
  GetTaskState(YourTask, &CurrentState);
  switch (CurrrentState) {
    case SUSPENDED:
      /* YourTask is suspended */
    case READY:
      /* YourTask is ready to run */
    case WAITING:
      /* YourTask is waiting (for an event) */
    case RUNNING:
      /* YourTask is running. Not possible as MyTask must be running to
          make the call */
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.35 IncrementCounter

ソフトウェアカウンタをインクリメントします。

構文

```
StatusType IncrementCounter(
    CounterType CounterID
)
```

引数

引数	モード	説明
CounterID	in	CounterType カウンタの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	CounterID はソフトウェアカウンタではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから CounterID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_CORE	拡張	CounterID を所有しているコア以外のコアから呼び出されました。

説明

ソフトウェアカウンタ CounterID をインクリメントします。つまり CounterID のカウンタ値に 1 を加えます。

インクリメントにより指定のカウンタ上のアラームがトリガされると、この API 関数がリターンする前に、トリガされたすべてのアラームに対応するアラームアクションが実行されます。

アラームアクションによってエラーが発生した場合（たとえば、タスク起動で E_OS_LIMIT が発行された場合など）は、発生した各エラーについてエラーフックが呼び出され、IncrementCounter()自体は E_OK を返します。

この API 関数によりリスケジューリングが発生する可能性があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ISR(MillisecondTimerInterrupt){
  ...
  IncrementCounter(MillisecondCounter);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_AdvanceCounter
Os_AdvanceCounter_<CounterID>
Os_IncrementCounter_<CounterID>

2.36 NextScheduleTable

実行パターンを、1 つのスケジュールテーブルから別のスケジュールテーブルに切り替えます。

構文

```
StatusType NextScheduleTable(
    ScheduleTableType ScheduleTableID_From,
    ScheduleTableType ScheduleTableID_To
)
```

引数

引数	モード	説明
ScheduleTableID_From	in	ScheduleTableType 切り替え前のスケジュールテーブルの ID
ScheduleTableID_To	in	ScheduleTableType 切り替え後のスケジュールテーブルの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_NOFUNC	All	ScheduleTableID_From が起動されていません。
E_OS_ID	拡張	ScheduleTableID_From または ScheduleTableID_To は有効なスケジュールテーブルではありません。
E_OS_ID	拡張	ScheduleTableID_From と ScheduleTableID_To の同期化ストラテジーが異なります。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID_From または ScheduleTableID_To にアクセスできません。
E_OS_STATE	拡張	ScheduleTableID_To はすでに起動されているか、次に実行されることになっています。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

この API 関数は、ScheduleTableID_From の最終満了ポイントが処理されてから ScheduleTableID_From.FinalDelay チック後に、ScheduleTableID_To のスケジュールテーブルの処理を開始します。

ScheduleTableID_To の最初の満了ポイントは、ScheduleTableID_To が起動されてから ScheduleTable_To.InitialOffset チック後に処理されます。

すでにScheduleTableID_Fromの次に実行が予定されているスケジュールテーブルがある場合は、その代わりにScheduleTableID_Toが実行されることになり、予定されていた方のテーブルはSCHEDULETABLE_STOPPED状態になります。

ScheduleTableID_FromとScheduleTableID_Toのいずれかのスケジュールテーブルが有効でないか、または両者が互いに異なるカウンタにより駆動される場合は、 各テーブルの状態は変わりません。

OSがScheduleTableID_Toの最初の満了ポイントを処理するときに、ScheduleTableID_Toの同期化ストラテジーが発効されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
    /* Stop MyScheduleTable at the end and start
       YourScheduleTable */
    NextScheduleTableAbs(MyScheduleTable, YourScheduleTable);
    ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.37 Os_AddDelayedTasks

この API 関数を呼び出したコアに関して、実行を遅延させるタスクのセットに 1 つまたは複数のタスクを追加します。

構文

```
StatusType Os_AddDelayedTasks(
    Os_TasksetType Taskset
)
```

引数

引数	モード	説明
Taskset	in	Os_TasksetType 一連のタスク

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ACCESS	拡張	遅延させるタスクのセットをアントラステッドコードが変更することはできません。
E_OS_ID	拡張	同じコア上の他のタスクと同じ優先度のタスクを追加しようとしました。その優先度を共有するすべてのタスクを追加する必要があります。

説明

プロジェクトにタスクの実行遅延が設定されている場合、あるタスクのセットについて、実行の遅延を RTA-OS に指示することができます。これらのタスクは、起動することはできますが、遅延実行タスクのセットから除外されるまでは実行されません。

この API 関数は、既存の実行遅延タスクのセットにタスクを追加します。1 つのタスクを 2 回以上追加することは可能ですが、これによる効果は何もありません。

1 つのコア上で複数のタスクが優先度を共有している場合は、それらのタスクをすべて追加するか、または 1 つも追加しないか、いずれかにするに必要があります。

この API 関数で追加できるタスクは、呼び出し元のコアで実行されるタスクに限られます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LowPrioTask){
  Os_AddDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 can be activated, but will not run */

  Os_SetDelayedTasks(OS_NO_TASKS);
  ... /* Tasks 1 and 3 can run */
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

OS_ADD_TASK
OS_NO_TASKS
Os_RemoveDelayedTasks
Os_SetDelayedTasks
Os_TasksetType
TASK_MASK

2.38 Os_AdvanceCounter

ハードウェアカウンタが、最後にセットされた値に到達したことを OS に通知します。

構文

```
StatusType Os_AdvanceCounter(
    CounterType CounterID
)
```

引数

引数	モード	説明
CounterID	in	CounterType カウンタの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	CounterID はハードウェアカウンタではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから CounterID にアクセスできません。
E_OS_STATE	拡張	CounterID は実行中ではありません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

この API 関数は、Os_Cbk_Set_<CounterID>コールバック関数により最後にセットされた値とカウンタ値が一致したことを OS に通知するものです。当該カウンタを所有しているコアでしか実行できません。

一般的にこの API 関数は、当該カウンタがマッチ値に達することによって発生する割り込みの処理から呼び出します。

この API 関数が呼び出されると、OS は期限に達したすべてのアラームまたは満了ポイントアクションを処理します。その後、Os_Cbk_Set_<CounterID>を使用して新しいマッチ値をセットするか、または Os_Cbk_Cancel_<CounterID>を使用して当該カウンタをキャンセルします。

なおここでは、カウンタ駆動用の割り込みの処理が終了する前にそのカウンタの新しいマッチ値に到達してしまう、という可能性があるので、注意が必要です。これを見落とすと、ベースとなるハードウェアカウンタがラップアラウンドするまで、そのカウンタは起動しなくなってしまいます。

ただし一部のハードウェアプラットフォームでは、既存インスタンスの終了時に割り込みが再びアサートされるので、特別な対策は必要ありません。

その他のプラットフォームでは、割り込みをソフトウェアで再びアサートするか、それができない場合は、下の例のようにコードの処理をループさせる必要があります。どちらの場合も、ドライバ実行中に発生するマッチ値への到達を見落とさないように細心の注意を払う必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
/* For systems where the interrupt will be re-entered
    automatically if the match occurs before leaving the ISR: */
ISR(SimpleCounterDriver){
  Os_AdvanceCounter(MyHWCounter);
}
/* For systems where the software can force the interrupt to get
    re-entered if the match occurs before leaving the ISR: */
ISR(RetriggeringCounterDriver){
  Os_CounterStatusRefType CurrentState;
  Os_AdvanceCounter(MyHWCounter);
  Os_Cbk_State_MyHWCounter(&CurrentState);
  if (CurrentState.Running && CurrentState.Pending) {
    /* Retrigger this interrupt */
  }
}
/* For systems where the software has to loop if the match occurs
    before leaving the ISR: */
ISR(LoopingCounterDriver){
  Os_CounterStatusRefType CurrentState;
  do {
    Os_AdvanceCounter(MyHWCounter);
    Os_Cbk_State_MyHWCounter(&CurrentState);
  } while (CurrentState.Running && CurrentState.Pending);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

IncrementCounter
Os_AdvanceCounter_<CounterID>
Os_Cbk_Cancel_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>
Os_IncrementCounter_<CounterID>

2.39 Os_AdvanceCounter_<CounterID>

特定のハードウェアカウンタが前回セットされた値に到達したことを OS に通知します。

構文

```
StatusType Os_AdvanceCounter_CounterID(void)
```

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_STATE	拡張	CounterID は実行中ではありません。

説明

この API 関数の挙動は Os_AdvanceCounter(CounterID)と同様ですが、特定のカウンタ用にカスタマイズされているので、処理が高速化され、割り込みハンドラ内での使用に適しています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
/* For systems where the interrupt will be re-entered
    automatically if the match occurs before leaving the ISR: */
ISR(SimpleCounterDriver){
  Os_AdvanceCounter_MyHWCounter();
}
/* For systems where the software can force the interrupt to get
    re-entered if the match occurs before leaving the ISR: */
ISR(RetriggeringCounterDriver){
  Os_CounterStatusRefType CurrentState;
  Os_AdvanceCounter_MyHWCounter();
  Os_Cbk_State_MyHWCounter(&CurrentState);
  if (CurrentState.Running && CurrentState.Pending) {
    /* Retrigger this interrupt */
  }
}
/* For systems where the software has to loop if the match occurs before
    leaving the ISR: */
ISR(LoopingCounterDriver){
  Os_CounterStatusRefType CurrentState;
  do {
    Os_AdvanceCounter_MyHWCounter();
    Os_Cbk_State_MyHWCounter(&CurrentState);
  } while (CurrentState.Running && CurrentState.Pending);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

IncrementCounter
Os_AdvanceCounter
Os_Cbk_Cancel_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>
Os_IncrementCounter_<CounterID>

2.40 Os_GetCurrentIMask

現在の割り込み優先度／マスクを取得します。

構文

```
Os_imaskType Os_GetCurrentIMask(void)
```

戻り値

Os_imaskType 型の値を返します。

説明

現在の割り込み優先度（一部のターゲットの場合は割り込みマスク）を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_imaskType imask = Os_GetCurrentIMask();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

OS_IMASK_FOR_<TaskOrISR>
OS_IMASK_FOR_ISR
OS_IMASK_FOR_TASK
Os_GetCurrentTPL

2.41 Os_GetCurrentTPL

実行中のタスクの内部優先度を取得します。

構文

```
uint32 Os_GetCurrentTPL(void)
```

戻り値

uint32 型の値を返します。

説明

現在実行中のタスクの内部優先度を返します。

1 つのコアの各タスクには、内部優先度に基づいて一意の TPL が割り当てられます。その値は宣言されている優先度の値と同じではありませんが、順序は似たものになります。

実行中のタスクがない場合は OS_INVALID_TPL が返ります。

タスクが実行中で、リソースをロックしているか、他のタスクと優先度を共有している場合は、OS_TPL の値より大きくなる可能性があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 tpl = Os_GetCurrentTPL();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

OS_INVALID_TPL
OS_TPL_FOR_<Task>
OS_TPL_FOR_TASK
Os_GetCurrentIMask

2.42 Os_GetElapsedTime

呼び出し元のタスク／ISR または Idle コールバック関数の累積実行時間を取得します。

構文

```
Os_StopwatchTickType Os_GetElapsedTime(void)
```

戻り値

Os_StopwatchTickType 型の値を返します。

説明

StartOS()、または累積時間の明示的リセット以降に呼び出し元（タスク、ISR、Idle コールバック関数）の実行に消費された累積時間を返します。

累積時間が Os_StopwatchTickType の範囲を超える場合は、レポートされる値は Os_StopwatchTickType に格納できる最大値（通常は 0xffffffff）に飽和されます。

レポートされる時間には、現在のタスク／ISR／Idle コールバック関数の呼び出しに消費された時間も含まれます。

この API 関数から適切な結果を得るには、時間監視（Time Monitoring）と経過時間記録（Elapsed Time Recording）をイネーブルにしておく必要があります。時間監視がイネーブルになっていないと、この API 関数はゼロを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType Total_Idle_Time;
Os_StopwatchTickType Total_Task_Time;
Os_StopwatchTickType Total_ISR_Time;

FUNC(boolean, {memclass}) Os_Cbk_Idle(void) {
  Total_Idle_Time = Os_GetElapsedTime();
}

TASK(MyTask){
  Total_Task_Time = Os_GetElapsedTime();
}

ISR(MyISR){
  Total_ISR_Time = Os_GetElapsedTime();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.43 Os_GetExecutionTime

呼び出し元のタスク／ISR の実行時間を取得します。

構文

```
Os_StopwatchTickType Os_GetExecutionTime(void)
```

戻り値

Os_StopwatchTickType 型の値を返します。

説明

当該タスク／ISR の開始以降に消費された正味の実行時間（プリエンプションされた期間を除外した時間）を返します。

拡張タスクの場合は、WaitEvent()の呼び出しからのリターン時に実行時間がリセットされ、カウントが開始されます。

この値は PreTaskHook()内では有効ではありませんが、Os_Cbk_TaskStart()内と Os_Cbk_ISRStart()内では有効です。

PostTaskHook()内で読み取られる値は有効ですが、その値は、タスクの最大実行時間を決定するために用いられる値より大きくなります。

実行時間がオーバーフローした場合は、ラップアラウンドした値を返します。

この API 関数から適切な結果を得るには、時間監視をイネーブルにしておく必要があります。イネーブルになっていないと、この API 関数はゼロを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  Os_StopwatchTickType Start, Finish, Used, APICallCorrection;
  Start = GetExecutionTime();
  Finish = GetExecutionTime();
  APICallCorrection = Finish - Start;  /* Get time for
      GetExecutionTime() call itself. */
  Start = GetExecutionTime();
  Call3rdPartyLibraryFunction();             /* Measure 3rd Party
      Library Code Execution Time */
  Finish = GetExecutionTime();
  Used = Finish - Start - APICallCorrection; /* Calculate the
      amount of time used. */
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

OS_ISRTYPE_TO_INDEX
OS_TASKTYPE_TO_INDEX
Os_Cbk_ISREnd
Os_Cbk_ISRStart
Os_Cbk_TaskEnd
Os_Cbk_TaskStart
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.44 Os_GetISRElapsedTime

指定された ISR の累積実行時間を取得します。

構文

```
Os_StopwatchTickType Os_GetISRElapsedTime(
      ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID

戻り値

Os_StopwatchTickType 型の値を返します。

説明

StartOS()、または累積時間の明示的リセット以降、指定されたカテゴリ 2 ISR の実行に消費された累積時間を返します。

累積時間が Os_StopwatchTickType の範囲を超えると、ラップアラウンドされた値を返すので、実際の値よりも小さい値となります。

当該 ISR が実行中である場合は、現在の呼び出し以降の時間は累積時間に含まれません。

ISRID が有効でない場合はゼロを返します。

この API 関数から適切な結果を得るには、時間監視（Time Monitoring）と経過時間記録（Elapsed Time Recording）をイネーブルにしておく必要があります。時間監視がイネーブルになっていないと、この API 関数はゼロを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType Total_ISR_Time;
Total_ISR_Time = Os_GetISRMaxExecutionTime(MyISR);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.45 Os_GetISRMaxExecutionTime

ISR の最大実行時間を取得します。

構文

```
Os_StopwatchTickType Os_GetISRMaxExecutionTime(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID

戻り値

Os_StopwatchTickType 型の値を返します。

説明

ISRID で指定されたカテゴリ 2 ISR が消費した最大実行時間を返します。

この値は、ISRID に対する前回の ResetISRMaxExecutionTime()の呼び出し以降、または前回の StartOS()の呼び出し以降に完了した ISRID の実行時間のうちの最大値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LoggingTask){
  Os_StopwatchTickType ExecutionTimes[MAXISRS];
  ...
  ExecutionTimes[0] = GetISRMaxExecutionTime(ISR1);
  ExecutionTimes[1] = GetISRMaxExecutionTime(ISR2);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

OS_ISRTYPE_TO_INDEX
Os_GetExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.46 Os_GetISRMaxStackUsage

ISR が使用したスタックの最大量を取得します。

構文

```
Os_StackSizeType Os_GetISRMaxStackUsage(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID

戻り値

Os_StackSizeType 型の値を返します。

説明

ISRID で指定されたカテゴリ 2 ISR が使用したスタックの最大量を返します。

この値は、ISRID に対する前回の ResetISRMaxStackUsage()の呼び出し以降、または前回の StartOS()の呼び出し以降に ISRID の呼び出しで使用されたスタック使用量の最大値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LoggingTask){
  Os_StackSizeType StackUsages[MAXISRS];
  ...
  StackUsages[0] = GetISRMaxStackUsage(ISR1);
  StackUsages[1] = GetISRMaxStackUsage(ISR2);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

OS_ISRTYPE_TO_INDEX
Os_GetStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage

2.47 Os_GetIdleElapsedTime

指定されたコアの累積アイドル時間を取得します。

構文

```
Os_StopwatchTickType Os_GetIdleElapsedTime(
    Os_IdleType IdleID
)
```

引数

引数	モード	説明
OS_CORE_CURRENT	in	Os_IdleType 呼び出しが実行されたコア上の Idle ID
OS_CORE_ID_0	in	Os_IdleType コア 0 上の Idle ID
OS_CORE_ID_n	in	Os_IdleType コア n 上の Idle ID

戻り値

Os_StopwatchTickType 型の値を返します。

説明

StartOS()、または累積時間の明示的リセット以降に「アイドル状態」として消費された累積時間を返します。

合計時間が Os_StopwatchTickType の範囲を超えると、ラップアラウンドされた値を返すので、実際の値よりも小さい値となります。

この API 関数を呼び出したコア上で、現在 Idle コールバック関数が呼び出されていた場合は、その Idle コールバック関数の呼び出しに消費された時間も含まれます。

IdleID が有効でない場合はゼロを返します。

この API 関数から適切な結果を得るには、時間監視（Time Monitoring）と経過時間記録（Elapsed Time Recording）をイネーブルにしておく必要があります。時間監視がイネーブルになっていないと、この API 関数はゼロを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType Total_Idle_Time;
Total_Idle_Time = Os_IdleType IdleID(OS_CORE_CURRENT);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.48 Os_GetStackSize

2 つのスタック値の差を取得します。

構文

```
Os_StackSizeType Os_GetStackSize(
      Os_StackValueType Base,
      Os_StackValueType Sample
)
```

引数

引数	モード	説明
Base	in	Os_StackValueType 基準となるスタック位置
Sample	in	Os_StackValueType 基準位置との差を求めるスタック位置

戻り値

Os_StackValueType 型の値を返します。

説明

2 つの Os_StackValueType の値の差を返します。正しい値を得るためには、'Base' が、'Sample' が読み取られた時点よりもスタックサイズが小さかった時点（または 'Sample' が読み取られたのと同じ時点）の位置を表している必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StackValueType start_position;
Os_StackValueType end_position;
Os_StackSizeType stack_size;
TASK(MyTask){
  start_position = Os_GetStackValue();
  nested_call();
  stack_size = Os_GetStackSize(start_position, end_position);
}
void nested_call(void) {
  end_position = Os_GetStackValue();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetStackUsage
Os_GetStackValue

2.49 Os_GetStackUsage

呼び出し元のタスク／ISR のスタック使用量を取得します。

構文

```
Os_StackSizeType Os_GetStackUsage(void)
```

戻り値

Os_StackSizeType 型の値を返します。

説明

この呼び出しの時点で呼び出し元のタスク／ISR が使用しているスタックの量を返します。

スタック使用量は OS カーネルがそのタスク／ISR の実行を開始した時点から測定され、カーネル内のオーバーヘッドも含まれるので、タスク／ISR のスタックアロケーションバジェットを設定する際に役立ちます。

この API 関数の呼び出しには、呼び出し元のタスク／ISR について記録されている最大スタック使用量が更新される、という副作用が伴います。

そのタスク／ISR にスタックアロケーションバジェットが定義されている場合は、この API 関数がリターンする前にスタックオーバーランがレポートされる場合があります。

この API 関数から適切な結果を得るには、OS コンフィギュレーションでスタック監視をイネーブルにしておく必要があります。スタック監視がイネーブルになっていないと、この API 関数はゼロを返します。

OS オプションの'Stack Sampling'が有効になっている場合は、この API 関数の中で Os_Cbk_CheckStackDepth コールバック関数が呼び出されます。このコールバック関数はスタック監視がイネーブルになっていなくても呼び出されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  Os_StackSizeType stack_size;
  stack_size = Os_GetStackUsage();
  nested_call();
}
void nested_call(void) {
  Os_GetStackUsage(); /* Identifies a possible max stack usage location
      */
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_Cbk_CheckStackDepth
Os_Cbk_StackOverrunHook
Os_GetISRMaxStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage

2.50 Os_GetStackValue

現在のスタック値を取得します。

構文

```
Os_StackValueType Os_GetStackValue(void)
```

戻り値

Os_StackValueType 型の値を返します。

説明

スタックポインタ（1 つまたは複数）の現在位置を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StackValueType start_position;
Os_StackValueType end_position;
Os_StackSizeType stack_size;
TASK(MyTask){
  start_position = Os_GetStackValue();
  nested_call();
  stack_size = Os_GetStackSize(start_position, end_position);
}
void nested_call(void) {
  end_position = Os_GetStackValue();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetStackSize
Os_GetStackUsage

2.51 Os_GetTaskActivationTime

タスクが最後に起動された時点のタイムスタンプを取得します。

構文

```
StatusType Os_GetTaskActivationTime(
    TaskType TaskID,
    Os_StopwatchTickRefType Value
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID
Value	out	Os_StopwatchTickRefType タスクの最後の起動時のタイムスタンプ

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	TaskID は有効なタスクではありません。
E_OS_NOFUN	拡張	TaskID はまだ一度も起動されていません。

説明

TaskID の最後に起動された時点のタイムスタンプを返します。まだ一度も起動されていない場合はゼロを返します。

起動は、ActivateTask、SetEvent、またはアラーム／満了ポイントにより行われる可能性があります。

タイムスタンプは、タスクが起動された時点で呼び出されるユーザー関数 Os_Cbk_GetStopwatch()が返す値です。

注記： この API 関数と起動時刻の記録機能は、Activation Monitoring という OS オプションが有効になっているのみ使用可能です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(Task1){
  Os_StopwatchTickType MyLastActivation;
  ...
  GetTaskActivationTime(Task1, &MyLastActivation);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

OS_TASKTYPE_TO_INDEX
Os_Cbk_GetStopwatch

2.52 Os_GetTaskElapsedTime

タスクの累積実行時間を取得します。

構文

```
Os_StopwatchTickType Os_GetTaskElapsedTime(
      TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

Os_StopwatchTickType 型の値を返します。

説明

StartOS()（または累積時間の明示的リセット）以降、指定されたタスクの実行のために消費された累積時間を返します。

合計時間が Os_StopwatchTickType の範囲を超えると、ラップアラウンドされた値を返すので、実際の値よりも小さい値となります。

タスクが実行中の場合は、レポートされる時間には現在の呼び出しからの時間は含まれません。

TaskID が有効でない場合はゼロを返します。

この API 関数から適切な結果を得るには、時間監視（Time Monitoring）と経過時間記録（Elapsed Time Recording）をイネーブルにしておく必要があります。時間監視がイネーブルになっていないと、この API 関数はゼロを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType Total_Task_Time;
Total_Task_Time = Os_GetTaskElapsedTime(MyTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.53 Os_GetTaskMaxExecutionTime

タスクの最大実行時間を取得します。

構文

```
Os_StopwatchTickType Os_GetTaskMaxExecutionTime(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

Os_StopwatchTickType 型の値を返します。

説明

TaskID で指定されたタスクが消費した最大実行時間を返します。

この値は、TaskID に対する前回の ResetTaskMaxExecutionTime()の呼び出し以降、または前回の StartOS()の呼び出し以降に完了した TaskID の実行時間のうちの最大値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LoggingTask){
  Os_StopwatchTickType ExecutionTimes[MAXTASKS];
  ...
  ExecutionTimes[0] = GetTaskMaxExecutionTime(Task1);
  ExecutionTimes[1] = GetTaskMaxExecutionTime(Task2);
  ExecutionTimes[2] = GetTaskMaxExecutionTime(Task3);
  ExecutionTimes[3] = GetTaskMaxExecutionTime(Task4);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

OS_TASKTYPE_TO_INDEX
Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxStackUsage

2.54 Os_GetTaskMaxStackUsage

タスクが使用したスタックの最大量を取得します。

構文

```
Os_StackSizeType Os_GetTaskMaxStackUsage(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

Os_StackSizeType 型の値を返します。

説明

TaskID で指定されたタスクが使用したスタックの最大量を返します。

この値は、TaskID に対する前回の ResetTaskMaxStackUsage()の呼び出し以降、または前回の StartOS()の呼び出し以降に TaskID 呼び出しで使用されたスタック使用量の最大値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LoggingTask){
  Os_StackSizeType StackUsages[MAXTASKS];
  ...
  StackUsages[0] = GetTaskMaxStackUsage(Task1);
  StackUsages[1] = GetTaskMaxStackUsage(Task2);
  StackUsages[2] = GetTaskMaxStackUsage(Task3);
  StackUsages[3] = GetTaskMaxStackUsage(Task4);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

OS_ISRTYPE_TO_INDEX
Os_GetISRMaxStackUsage
Os_GetStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage

2.55 Os_GetVersionInfo

OS のバージョン情報を取得します。

構文

```
void Os_GetVersionInfo(
    Std_VersionInfoType *versioninfo
)
```

引数

引数	モード	説明
versioninfo	out	Std_VersionInfoType OS のバージョン情報を格納する変数へのポインタ

説明

構造体 Std_VersionInfoType は Std_Types.h に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
Std_VersionInfoType ver;
Os_GetVersionInfo(&ver);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

なし。

2.56 Os_IncrementCounter_<CounterID>

ソフトウェアカウンタをインクリメントします。

構文

```
StatusType IncrementCounter_<CounterID>(void)
```

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。

説明

この API 関数の挙動は IncrementCounter(CounterID) と同様ですが、特定のカウンタ用にカスタマイズされているので、処理が高速化され、割り込みハンドラ内での使用に適しています。ただしエラーチェックは行われないので注意してください。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
ISR(MillisecondTimerInterrupt){
  ...
  Os_IncrementCounter_MillisecondCounter();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

IncrementCounter
Os_AdvanceCounter
Os_AdvanceCounter_<CounterID>

2.57 Os_Metrics_Reset

この API 関数を呼び出したコアの OS メトリクス（計測値）をすべてゼロにリセットします。

構文

```
void Os_Metrics_Reset(void)
```

説明

OS オプション'Collect OS usage metrics'がイネーブルになっていると、OS はランタイムに'Os_Metrics'という構造体のデータを収集します。この構造体は Os_Metrics.h に定義されています。この情報は、ユーザーが任意に使用することができ、そのダンプ情報を ETAS サポートに送ってコンフィギュレーションのパフォーマンス調整に役立てることもできます。

OS は、各 OS API 関数の呼び出し、各タスクの起動と実行開始、各 ISR の実行開始、スピンロックアクセス、クロスコア割り込み、クロスコアのタスク起動、クロスコアの IOC 起動、カウンタのインクリメント／アドバンス、ECC タスクのデータ／ユーザーカウンタ／（必要に応じて）ターゲット固有値待ちの数を収集します。

これらの値は、Os_Metrics_Reset() API 関数を呼び出すことにより、いつでもゼロにリセットすることができます。

マルチコア環境では、各コアにつき 1 つの'Os_Metrics'があり、コア自身の'Os_Metrics'のデータだけを更新します。

ランタイムオーバーヘッドを抑えるため、Os_Metrics 内の値をインクリメントしたりリセットしたりする際には相互排除や範囲チェックのためのコードが含まれないことに注意してください。また、OS のメトリクスの収集は試験工程においてのみ実施し、量産システムでは絶対に行わないでください。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  ...
#ifdef OS_METRICS_ENABLED
  Os_Metrics_Reset();
#endif /* OS_METRICS_ENABLED */
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

OS_COUNT_USER_n
OS_COUNT_cat1isrname

2.58 Os_RemoveDelayedTasks

実行を遅延させるタスクのセットから 1 つまたは複数のタスクを除外します。

構文

```
StatusType Os_RemoveDelayedTasks(
    Os_TasksetType Taskset
)
```

引数

引数	モード	説明
Taskset	in	Os_TasksetType タスクの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ACCESS	拡張	遅延させるタスクのセットをアントラステッドコードが変更することはできません。
E_OS_ID	拡張	同じコア上の他のタスクと同じ優先度のタスクを除外しようとしました。同じ優先度を共有するタスクは、すべて除外する必要があります。

説明

プロジェクトにタスクの実行遅延が設定されている場合、あるタスクのセットについて、実行の遅延を RTA-OS に指示することができます。これらのタスクは、起動することはできますが、遅延実行タスクのセットから除外されるまでは実行されません。

Os_RemoveDelayedTasks は、実行を遅延させるタスクのセットからタスクを除外するためのものです。

除外されたタスクのうち、優先度が呼び出し元のタスクより高いものは、この API 関数がリターンする前に実行されます。

1 つのコア上で複数のタスクが優先度を共有している場合は、それらのタスクをすべて追加するか、または 1 つも追加しないか、いずれかにするに必要があります。

実行遅延のタスクセットに追加できるタスクは、呼び出し元のコアで実行されるタスクに限られます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LowPrioTask){
  Os_SetDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 (only) can be activated, but will not run */

  Os_RemoveDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 can run */
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

OS_ADD_TASK
OS_NO_TASKS
Os_AddDelayedTasks
Os_SetDelayedTasks
Os_TasksetType
TASK_MASK

2.59 Os_ResetISRElapsedTime

ISR の累積実行時間をリセットします。

構文

```
StatusType Os_ResetISRElapsedTime(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ISRID は有効な ISR ではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ISRID にアクセスできません。

説明

ISRID で指定された ISR が消費した累積実行時間をゼロにリセットします。

マルチコアアプリケーションにおいては、信頼できる形でこの呼び出しを行えるのは ISRID を所有しているコアに限られます。効率性の確保のため、完全なクロスコア保護は実装されていません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_ResetISRElapsedTime(MyISR);
```

呼び出し元

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetISRElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.60 Os_ResetISRMaxExecutionTime

ISR の最大実行時間をリセットします。

構文

```
StatusType Os_ResetISRMaxExecutionTime(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ISRID は有効なカテゴリ 2 ISR ではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ISRID にアクセスできません。

説明

ISRID で指定されたカテゴリ 2 ISR が消費した最大実行時間をゼロにリセットします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(ProfilingTask){
  Os_StopwatchTickType ExecutionTimeLog[SAMPLES];
  ...
  ExecutionTimeLog[index++] = Os_GetISRMaxExecutionTime(ISR1);
  Os_ResetISRMaxExecutionTime(ISR1);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.61 Os_ResetISRMaxStackUsage

ISR が使用したスタックの最大量をリセットします。

構文

```
StatusType Os_ResetISRMaxStackUsage(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType ISR の ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ISRID は有効なカテゴリ 2 ISR ではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ISRID にアクセスできません。

説明

ISRID が使用したスタックの最大量をゼロにリセットします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(ProfilingTask){
  Os_StackSizeType StackUsageLog[SAMPLES];
  ...
  StackUsageLog[index++] = Os_GetISRMaxStackUsage(ISR1);
  Os_ResetISRMaxStackUsage(ISR1);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

Os_GetISRMaxStackUsage
Os_GetStackUsage
Os_GetTaskMaxStackUsage
Os_ResetTaskMaxStackUsage

2.62 Os_ResetIdleElapsedTime

コアの累積アイドル時間をリセットします。

構文

```
StatusType Os_ResetIdleElapsedTime(
    Os_IdleType IdleID
)
```

引数

引数	モード	説明
OS_CORE_CURRENT	in	Os_IdleType 呼び出しが実行されたコア上の Idle ID
OS_CORE_ID_0	in	Os_IdleType コア 0 上の Idle ID
OS_CORE_ID_n	in	Os_IdleType コア n 上の Idle ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	IdleID が無効です。

説明

IdleID で指定されたコアが消費した累積アイドル実行時間をゼロにリセットします。

マルチコアアプリケーションにおいては、信頼できる形でこの呼び出しを行えるのは IdleID を所有しているコアに限られます。効率性の確保のため、完全なクロスコア保護は実装されていません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_ResetIdleElapsedTime(OS_CORE_CURRENT);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetTaskElapsedTime

2.63 Os_ResetTaskElapsedTime

タスクの累積実行時間をリセットします。

構文

```
StatusType Os_ResetTaskElapsedTime(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	TaskID は有効なタスクではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。

説明

TaskID で指定されたタスクが消費した累積実行時間をゼロにリセットします。

マルチコアアプリケーションでは、信頼できる形でこの呼び出しを行えるのは TaskID を所有しているコアからだけです。効率性を確保するために、完全なクロスコア保護は実装されていません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_ResetTaskElapsedTime(MyTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime

2.64 Os_ResetTaskMaxExecutionTime

タスクの最大実行時間をリセットします。

構文

```
StatusType Os_ResetTaskMaxExecutionTime(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	TaskID は有効なタスクではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。

説明

TaskID で指定されたタスクが消費した最大実行時間をゼロにリセットします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(ProfilingTask){
  Os_StopwatchTickType ExecutionTimeLog[SAMPLES];
  ...
  ExecutionTimeLog[index++] = Os_GetTaskMaxExecutionTime(Task1);
  Os_ResetTaskMaxExecutionTime(Task1);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime

2.65 Os_ResetTaskMaxStackUsage

タスクが使用したスタックの最大量をリセットします。

構文

```
StatusType Os_ResetTaskMaxStackUsage(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	TaskID は有効なタスクではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。

説明

TaskID で指定されたタスクが使用したスタックの最大量をゼロにリセットします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(ProfilingTask){
  Os_StackSizeType StackUsageLog[SAMPLES];
  ...
  StackUsageLog[index++] = Os_GetTaskMaxStackUsage(Task1);
  Os_ResetTaskMaxStackUsage(Task1);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

Os_GetISRMaxStackUsage
Os_GetStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage

2.66 Os_Restart

定義済みの再起動ポイントにジャンプして、OS を再起動します。

構文

```
StatusType Os_Restart(void)
```

戻り値

StatusType 型の値を返します。

値	ビル	説明
E_OS_SYS_RESTART	All	ShutdownHook 以外から呼び出されました。
E_OS_SYS_NO_RESTART	All	再起動ポイントがセットされていません。

説明

この API 関数は、必要に応じてコンテキストを再初期化し、Os_SetRestartPoint でセットされた再起動ポイントにジャンプします。この API 関数は、呼び出し元には戻りません。

これに続く StartOS()の呼び出しによって OS 再初期化が行われるようにするため、StartOS()より前に再起動ポイントがセットされている必要があります。

Os_Restart()はマルチコアシステムでも使用できます。ただし StartOS()内の同期化コードは各コアの再起動を妨げるので、すべての AUTOSAR コアについてそれぞれ Os_Restart()を呼び出す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) ShutdownHook(StatusType Error){
  ...
  Os_Restart();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_SetRestartPoint
ShutdownAllCores
ShutdownOS
StartOS

2.67 Os_SetDelayedTasks

実行を遅延させるタスクを指定します。

構文

```
StatusType Os_SetDelayedTasks(
    Os_TasksetType Taskset
)
```

引数

引数	モード	説明
Taskset	in	Os_TasksetType 呼び出し元のコア上のタスクのセット

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ACCESS	拡張	遅延させるタスクのセットをアントラステッドコードが変更することはできません。
E_OS_ID	拡張	同じコア上の他のタスクと同じ優先度のタスクを追加しようとしました。同じ優先度を共有するタスクは、すべて追加する必要があります。

説明

プロジェクトにタスクの実行遅延が設定されている場合は、あるタスクのセットについて、実行の遅延を RTA-OS に指示することができます。これらのタスクは、起動することはできますが、遅延実行タスクのセットから除外されるまでは実行されません。

Os_SetDelayedTasks は、実行を遅延するタスクを指定するためのものです。

あるタスクの実行が、この API 関数の呼び出し前にすでに遅らされていて、遅延実行タスクのセットにそのタスクが含まれていない場合は、この API 関数から戻る前にそのタスクが実行されます（そのタスクの優先度が Os_SetDelayedTasks の呼び出し元よりも高い場合）。

1 つのコア上で複数のタスクが優先度を共有している場合は、それらのタスクをすべて追加するか、または 1 つも追加しないか、いずれかにするに必要があります。

実行遅延のタスクセットに追加できるタスクは、呼び出し元のコアで実行されるタスクに限られます。

可搬性

可搬性	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(LowPrioTask){
  Os_SetDelayedTasks(TASK_MASK(Task1)  | TASK_MASK(Task3));
  ... /* Tasks  1 and 3 (only) can be activated, but will not run*/

  Os_SetDelayedTasks(OS_NO_TASKS);
  ... /* Tasks 1 and 3 can run*/
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

OS_ADD_TASK
OS_NO_TASKS
Os_AddDelayedTasks
Os_RemoveDelayedTasks
Os_TasksetType
TASK_MASK

2.68 Os_SetRestartPoint

コード内の StartOS()より前の位置で呼び出し、OS の再起動ポイントをマークします。

構文

```
StatusType Os_SetRestartPoint(void)
```

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OS_SYS_NO_RESTART	All	StartOS より後に呼び出されました。

説明

この API 関数は、Os_Restart()の呼び出しの後に実行を再開（レジューム）するコード位置をマークするものです。この位置は OS による制御の範囲外である必要があるため、StartOS()の前に呼び出す必要があります。再起動ポイントがすでにセットされている状態においてこの API 関数を呼び出すと、再起動ポイントは新しい位置に再セットされます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
OS_MAIN() {
  ...
  Os_SetRestartPoint();
  ...
  StartOS(OSDEFAULTAPPMODE);

}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_Restart
ShutdownAllCores
ShutdownOS
StartOS

2.69 Os_SyncScheduleTableRel

明示的に同期化されるスケジュールテーブル用の調整値を提供します。

構文

```
StatusType Os_SyncScheduleTableRel(
    ScheduleTableType ScheduleTableID,
    SignedTickType RelativeValue
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID
RelativeValue	in	SignedTickType 同期化カウンタの調整量

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	All	ScheduleTableID は明示的に同期化されているテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_VALUE	All	値がテーブルの期間より大きくなっています。
E_OS_STATE	All	ScheduleTableID の状態が SCHEDULETABLE_WAITING、SCHEDULETABLE_STOPPED または SCHEDULETABLE_NEXT です。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

明示的に同期化されているテーブル ScheduleTableID に同期化のための値 RelativeValue を提供します。ScheduleTableID は実行中である必要があります。ScheduleTableID の実行開始には、API 関数 SyncScheduleTable を使用します。

同期化カウンタの制御と同期化カウンタについての情報は、OS の領域外になります。OS は、同期化カウンタの期間が ScheduleTableID と等しいこと、および同期化カウンタの分解能と ScheduleTableID の駆動に使用される OS カウンタの分解能とが等しいことを想定しています。そのため、ユーザーの責任において、アプリケーションがこれらの制約条件を満足していることを確認する必要があります。

ScheduleTableID の状態が SCHEDULETABLE_RUNNING または SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS である場合は、RelativeValue は ScheduleTableID 上の概念位置と望ましい値との現在の差であると解釈されます。

ScheduleTableID の状態は、静的に定義された精度（precision）に応じて以下のようにセットされます。

- RelativeValue <= precision の場合は、状態は SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS になります。
- RelativeValue > precision の場合は、状態は SCHEDULETABLE_RUNNING になります。

RelativeValue が正値で、定義済みの精度を超えている場合は、OS_SYNC_RETARDING 状態になります（チックが削除されます）。

RelativeValue が負値で、定義済みの精度を超えている場合は、OS_SYNC_ADVANCING 状態になります（チックが追加されます）。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  StartScheduleTableSynchron(MyScheduleTable);
  ...
  Os_SyncScheduleTable(MyScheduleTable, 0); /* Start */
  ...
  Os_SyncScheduleTableRel(MyScheduleTable, -2); /* Adjust drift */
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.70 Os_TimingFaultDetected

検出されたタイミング保護障害をレポートします。

構文

```
void Os_TimingFaultDetected(void)
```

説明

タイミング保護が設定されていて、タイミング割り込みがタイムリミットの実施に使用される場合、発行されたタイミング割り込みはこの API 関数を呼び出す必要があります。

最後のOs_Cbk_SetTimeLimit()の呼び出しによってセットされたタイムリミットに到達した場合は、その後に Os_Cbk_SuspendTimeLimit()が呼び出されてこのタイミング割り込みをキャンセルしない限り、必ずこのタイミング割り込みが実行されなければなりません。

このタイミング割り込みはカテゴリ 1 ISR で、カテゴリ 2 ISR の最高の優先度より高い優先度を割り当てます。他のカテゴリ 1 ISR は使用しないことが望ましいですが、使用する場合は、それらがこのタイミング割り込みによってプリエンプトされないようにする注意が必要です。

この API 関数を呼び出すと、OS は ProtectionHook を呼び出します。つまり、通常はこのタイミング割り込みには戻らないため、Os_TimingFaultDetected()を呼び出す前に必要な割り込みクリーンアップコードを実行しておく必要があります。

マルチコアシステムでは、タイミングリミットを持つコアごとに 1 つのタイミング割り込みが必要です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
CAT1_ISR(timing_interrupt) {
  /* Reset pending interrupt flags here if needed */
  Os_TimingFaultDetected();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_Cbk_SetTimeLimit
Os_Cbk_SuspendTimeLimit
ProtectionHook

2.71 ReleaseResource

占有されているリソースを解放（アンロック）してクリティカルセクションを終了します。

構文

```
StatusType ReleaseResource(
    ResourceType ResID
)
```

引数

引数	モード	説明
ResID	in	ResourceType リソースの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ResID は有効なリソースではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ResID にアクセスできません。
E_OS_ACCESS	拡張	呼び出し元のタスク／ISR について設定されている優先度よりも低い上限優先度のリソースを解放しようとしました。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_RESOURCE	拡張	このリソースがロックされた後に新しいスピンロックが取得されたところで呼び出されました。スピンロックやリソースのロックとアンロックは厳密に LIFO 順でしか行えません。
E_OS_NOFUNC	拡張	指定されたリソースは占有されていません。

説明

ReleaseResource は GetResource に対応する呼び出しで、コード内のクリティカルセクションを終了する機能があります。

リソースを、複数のコアで共有することはできません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  GetResource(Outer);
    /* Outer Critical Section */
    ... GetResource(Inner);
      /* Inner Critical Section */
    ReleaseResource(Inner);
    ...
  ReleaseResource(Outer);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareResource
GetResource

2.72 ReleaseSpinlock

スピンロック変数を解放します。

構文

```
ReleaseSpinlock(
    SpinlockIdType SpinlockId
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	SpinlockId は有効なスピンロックを参照していません。
E_OS_STATE	拡張	呼び出し元のタスク／ISR はこのスピンロックを占有していません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから SpinlockId にアクセスできません。
E_OS_NOFUNC	拡張	別のスピンロックを先に解放する必要があります。スピンロックの GetSpinlock/ReleaseSpinlock のネスト順序が守られていません。
E_OS_RESOURCE	拡張	先にリソースを解放する必要があります。スピンロックやリソースのネスト順序が守られていません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_RESOURCE	拡張	このスピンロックが取得されてから新しいリソースがロックされたところで呼び出されました。スピンロックやリソースのロックとアンロックは厳密に LIFO 順でしか行えません。

説明

ReleaseSpinlock は、GetSpinLock または TryToGetSpinLock を使用して占有されているスピンロックを解放します。

すべてのスピンロックは、タスクまたは ISR がターミネートする前に解放する必要があります。

すべてのスピンロックは、ECC タスクが WaitEvent を呼び出す前に解放する必要があります。

スピンロックのロックメソッド（コンテナ: OsSpinlock）のモードに応じて、この API 関数の挙動は以下のように変わります。

- LOCK_ALL_INTERRUPTS — ResumeAllInterrupts()の呼び出しで終了します。
- LOCK_CAT2_INTERRUPTS — ResumeOSInterrupts()の呼び出しで終了します。
- LOCK_WITH_RES_SCHEDULER — ReleaseResource(RES_SCHEDULER)の呼び出しで終了します。
- NESTABLE — スピンロックをロックした側により呼び出され、実行した Get コールと同じ数の Release コールを実行したときに限り、アンロックを行います。

OS コンフィギュレーションオプション'Force spinlock error checks'を使用すると、拡張ステータスのビルド以外に標準ステータスのビルドでもエラーチェックが行われます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
TASK(MyTask){
  ...
  GetSpinlock(Spinlock1);
  ...
  ReleaseSpinlock(Spinlock1);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetSpinlock
ReleaseResource
ResumeAllInterrupts
ResumeOSInterrupts
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.73 ResetSpinlockInfo

スピンロックのランタイム統計情報をリセットします。

構文

```
StatusType ResetSpinlockInfo(
    SpinlockIdType SpinlockId
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	SpinlockId は有効なスピンロックを参照していません。

説明

ResetSpinlockInfo はオプションの API 関数です。アプリケーション内にスピンロックがあり、'Provide spinlock statistics'という OS オプションが有効になっている場合に限り使用できます。

この API 関数は、指定されたスピンロックのすべての統計情報をゼロにリセットします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  ResetSpinlockInfo(Spinlock1);
  ...
  GetSpinlock(Spinlock1);
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.LockFails > 0) {
    ...
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetSpinlockInfo
Os_SpinlockInfo

2.74 ResumeAllInterrupts

カテゴリ 1 とカテゴリ 2 の割り込みの受け付けを再開（レジューム）します。

構文

```
void ResumeAllInterrupts(void)
```

説明

マスク可能なすべての割り込みが発生しないように保護されているクリティカルセクションを終了します。このクリティカルセクションは SuspendAllInterrupts()の呼び出しで開始されたものである必要があります。

このクリティカルセクションの中では、SuspendAllInterrupts()/ResumeAllInterrupts()のペアと SuspendOSInterrupts()/ResumeOSInterrupts()のペア以外の API 関数は使用できません。

割り込みの状態は、直前の SuspendAllInterrupts()の呼び出し前の状態に戻ります。

SuspendAllInterrupts()と ResumeAllInterrupts()の呼び出しがネストしている場合は、最初の SuspendAllInterrupts()の呼び出しによって保存された割り込みマスクの状態が、最後の ResumeAllInterrupts()の呼び出しによって復元されます。

マルチコア環境では、この API 関数が呼び出されたコア上の割り込みにのみ影響します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  SuspendAllInterrupts();
    /* Critical Section 1 */
    FunctionWithNestedCriticalSection();
  ResumeAllInterrupts();
  ...
}
void FunctionWithNestedCriticalSection(void) {
  ...
  SuspendAllInterrupts();
  /* Critical Section 2 */
  ResumeAllInterrupts();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

DisableAllInterrupts
EnableAllInterrupts
ResumeOSInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.75 ResumeOSInterrupts

カテゴリ 2 割り込みの受け付けを再開（レジューム）します。

構文

```
void ResumeOSInterrupts(void)
```

説明

この API 関数は、マスク可能なすべてのカテゴリ 2（OS レベル）割り込みの発生から保護されるクリティカルセクションを終了します。このクリティカルセクションは SuspendOSInterrupts()の呼び出しで開始されたものである必要があります。

このクリティカルセクションの中では、SuspendAllInterrupts()/ResumeAllInterrupts()のペアと SuspendOSInterrupts()/ResumeOSInterrupts()のペア以外の API 関数は使用できません。

割り込みの状態は、直前の SuspendOSInterrupts()呼び出しの前の状態に戻ります。

SuspendOSInterrupts()と ResumeOSInterrupts()の呼び出しがネストしている場合は、最初の SuspendOSInterrupts()の呼び出しによって保存された割り込み認識ステータスが、最後の ResumeOSInterrupts()の呼び出しによって復元されます。

マルチコア環境では、この API 関数が呼び出されたコア上の割り込みだけに影響します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  SuspendOSInterrupts():
    /* Longer Critical Section */
    SuspendAllInterrupts();
      /* Shorter Critical Section*/
    ResumeAllInterrupts();
  ResumeOSInterrupts():
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

DisableAllInterrupts
EnableAllInterrupts
ResumeAllInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.76 Schedule

優先度が高いタスクを実行できるかどうかを OS に問い合わせます。

構文

```
StatusType Schedule(void)
```

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_RESOURCE	拡張	呼び出し元タスクがまだリソースを保持しています。
E_OS_CALLEVEL	拡張	割り込みレベルで呼び出されました。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_SPINLOCK	拡張	タスクがスピンロックを保持しているときに呼び出されました。

説明

この API 関数を呼び出すと、ノンプリエンプティブタスクまたは内部リソースを使用するタスク／ISR がプリエンプションポイントを提示することができます。

以下の場合、リスケジューリングが発生します。

1. 呼び出し元タスクがノンプリエンプティブで、そのタスクが実行状態のときにそれより優先度の高いタスクが起動された場合。

2. 呼び出し元タスク／ISR がそれより優先度の高いタスク／ISR と内部リソースを共有していて、後者のタスク／ISR がすでに起動されている場合。

呼び出し元タスク／ISR より優先度の高い、レディ状態のタスク／ISR がない場合は、呼び出し元のタスク／ISR がレジュームします。

この API 関数は、内部リソースを使用しないプリエンプティブタスクや ISR には影響しません。

ISR に内部リソースを共有させることができるのは、RTA-OS 固有の機能です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  CooperativeProcessA();
  Schedule();
  CooperativeProcessB();
  Schedule();
  CooperativeProcessC();
  Schedule();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareTask
GetISRID
GetTaskState
TerminateTask

2.77 SetAbsAlarm

アラームに「絶対カウンタ値」をセットします。

構文

```
StatusType SetAbsAlarm(
    AlarmType AlarmID,
    TickType start,
    TickType cycle
)
```

引数

引数	モード	説明
AlarmID	in	AlarmType アラームの ID
start	in	TickType 最初にこのアラームがトリガされる時の絶対チック値
cycle	in	TickType その後、このアラームが繰り返しトリガされる間隔（チック数）

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_STATE	All	AlarmID はすでに実行されています。
E_OS_ID	拡張	AlarmID は有効なアラームではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから AlarmID にアクセスできません。
E_OS_VALUE	拡張	start または cycle の値が許容範囲外です。 許容範囲： 0 <= start <= maxallowedvalue、cycle = 0 または mincycle <= cycle <= maxallowedvalue
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

この API 関数はアラームを始動し、そのアラームをトリガするカウンタの満了値をセットします。

cycle がゼロの場合は、アラームは 1 回だけトリガされ、ゼロ以外の場合は start 以降も cycle チックごとにトリガされます。

アラームがトリガされると、静的に設定されているアクション（タスクの起動、イベントのセット、アラームコールバック関数の実行、カウンタのインクリメント）が実行されます。

アラームの実行中に、別の値でそのアラームを再度開始する場合は、先にそのアラームをキャンセルする必要があります。

start の値が現在のカウンタ値と等しいかそれより小さい場合は、このアラームはカウンタがフルラップした後でないとトリガされません。

開始時に絶対アラームが start 値ゼロでセットされると（SetAbsAlarm(MyAlarm,0,x) ）、このアラームはカウンタの maxallowedvalue+1 チックが経過しないとトリガされません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  /* SingleShotAlarm at tick 42 */
  SetAbsAlarm(SingleShotAlarm, 42, 0);
  ...
  /* PeriodicAlarm at 10, 60, 110, 160,... */
  SetAbsAlarm(PeriodicAlarm, 10, 50);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

CancelAlarm
DeclareAlarm
TickRefTypeGetAlarm
GetAlarmBase
SetRelAlarm

2.78 SetEvent

タスクに対して 1 つ以上のイベントをセットします。

構文

```
StatusType SetEvent(
      TaskType TaskID,
      EventMaskType Mask
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID
Mask	in	EventMaskType セットするイベントのマスク

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_CORE	All	このタスクは Shutdown で停止しているコアに属しています。
E_OS_ID	拡張	TaskID は有効なタスクではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから TaskID にアクセスできません。
E_OS_ACCESS	拡張	TaskID は拡張タスクではありません。
E_OS_STATE	拡張	TaskID はサスペンド状態です。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

この API 関数は、タスク TaskID に対して、Mask で指定された一連のイベントをセットします。

そのタスクが WaitEvent を呼び出してイベント待ち状態になっていた場合は、直ちにレディ状態になり、リスケジューリングが発生する可能性があります。

ビットごとのイベントを論理 OR で統合したイベントマスクを使用することにより、一度に複数のイベントをセットすることができます。

イベントマスク内でセットされていないイベントの状態は変化しません。

サスペンド状態の拡張タスクに対してはイベントをセットすることはできません。拡張ステータスの場合は、この呼び出しの結果はエラー（E_OS_STATE）になります。標準ステータスの場合は、サスペンド状態のタスクについてイベントをセットしても何の効果もありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask) {
  ...
  /* Set a single event */
  SetEvent(MyExtendedTask, Event1);
  ...
  /* Set multiple events */
  SetEvent(MyOtherExtendedTask, Event1 | Event2 | Event3);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

ClearEvent
DeclareEvent
SetEvent
WaitEvent

2.79 SetRelAlarm

アラームに「相対カウンタ値」をセットします。

構文

```
StatusType SetRelAlarm(
    AlarmType AlarmID,
    TickType increment,
    TickType cycle
)
```

引数

引数	モード	説明
AlarmID	in	AlarmType アラームの ID
Increment	in	TickType 最初にこのアラームがトリガされるまでの相対チック数
Cycle	in	TickType その後、このアラームが繰り返しトリガされる間隔（チック数）

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_STATE	All	AlarmID はすでに実行されています。
E_OS_ID	拡張	AlarmID は有効なアラームではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから AlarmID にアクセスできません。
E_OS_VALUE	拡張	increment または cycle の値が許容範囲外です。 許容範囲： 0 < increment <= maxallowedvalue、cycle = 0 または mincycle <= cycle <= maxallowedvalue
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました(サービス保護が設定されている場合のみ)。

説明

この API 関数はアラームを始動し、そのアラームをトリガするカウンタの満了値をセットします。現在のカウンタ値に Increment を加えた値が満了値となります。

Cycle がゼロの場合は、アラームは 1 回だけトリガされ、ゼロ以外の場合はそれ以降も Cycle チックごとにトリガされます。

アラームがトリガされると、静的に設定されているアクション（タスクの起動、イベントのセット、アラームコールバック関数の実行、カウンタのインクリメント）が実行されます。

アラームの実行中に、別の値でそのアラームを再度開始する場合は、先にそのアラームをキャンセルする必要があります。

Increment の値が小さい場合は注意が必要です。これは、この API 関数の処理が完了する前にカウンタが満了値を超えたか超えていないかによって、間もなくアラームが満了するか、またはカウンタのラップアラウンド後に満了するかのいずれかになります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  /* SingleShotAlarm in Now+123 ticks */
  SetRelAlarm(SingleShotAlarm, 123, 0);
  ...
  /* PeriodicAlarm at Now+42, Now+142, Now+242... */
  SetRelAlarm(PeriodicAlarm, 42, 100);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

CancelAlarm
DeclareAlarm
TickRefTypeGetAlarm
GetAlarmBase
SetAbsAlarm

2.80 SetScheduleTableAsync

スケジュールテーブル上の同期化をキャンセルします。

構文

```
StatusType SetScheduleTableAsync(
    ScheduleTableType ScheduleTableID
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ScheduleTableID は無効です。または、ScheduleTableID は明示的に同期化されるテーブルではありません。
E_OS_STATE	拡張	ScheduleTableID の現在の状態が SCHEDULETABLE_STOPPED、SCHEDULETABLE_NEXT、SCHEDULETABLE_WAITING のいずれでもありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

この関数は、ScheduleTableID が実行中で、しかも「明示的に同期化される」と設定されている場合に限り、ScheduleTableID の状態を SCHEDULETABLE_RUNNING にします。

OS は ScheduleTableID 上の満了ポイントの処理を続けますが、後に SyncScheduleTable() または SyncScheduleTableRel()が呼び出されるまでは満了ポイントの同期化を停止します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){ StartScheduleTableRel(MyScheduleTable, 2U);
  ...
  SyncScheduleTable(MyScheduleTable, 12U);
  ...
  SetScheduleTableAsync(MyScheduleTable);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.81 ShutdownAllCores

すべての AUTOSAR コアをシャットダウンします。

構文

```
void ShutdownAllCores(
    StatusType Error
)
```

引数

引数	モード	説明
Error	in	StatusType シャットダウンの理由

説明

マルチコア構成では、ShutdownAllCores が実行されるとすべての AUTOSAR コアが同期的にシャットダウンされます。

この API 関数は各 AUTOSAR コア上で強制的に ShutdownOS を実行させます。

これらのコアは、グローバルの ShutdownHook を呼び出す直前に同期化されます。

各コアにおいてグローバルの ShutdownHook 内で Os_Restart() が呼び出された場合は、システムを再起動できます。

ShutdownAllCores は、アントラステッドコードから呼び出された場合には無効となり、何の処理も行わずに呼び出し元に戻ります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
TASK(MyTask){
  ...
  if (ErrorCondition != E_OK) {
    ShutdownAllCores(ErrorCondition);
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

Os_Restart
Os_SetRestartPoint
ShutdownOS
StartOS

2.82 ShutdownOS

オペレーティングシステムをシャットダウンします。

構文

```
void ShutdownOS(
    StatusType Error
)
```

引数

引数	モード	説明
Error	in	StatusType シャットダウンの理由

説明

この API 関数が実行されると OS がシャットダウンします。タスクのスケジューリング、すべてのカテゴリ 2 割り込み、アラーム、スケジュールテーブルはすべて直ちに停止します。

PostTaskHook（設定されている場合）は、ShutdownOS()が発生した場合は呼び出されません。

ShutdownHook（設定されている場合）が呼び出され、 OS がシャットダウンするときに ShutdownHook に Error 引数が渡されます。

ShutdownHook()がリターンした場合や設定されていない場合は、オペレーティングシステムはすべてのカテゴリ 1 割り込みをディセーブルにして無限ループに入ります。

ShutdownOS()は、回復不能なエラーへの対応としてオペレーティングシステムから内部的に呼び出すことができます。

ShutdownOS()は、アントラステッドコードから呼び出された場合は無効となり、何も行わずに呼び出し元に戻ります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  if (ErrorCondition != E_OK) {
    ShutdownOS(ErrorCondition);
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

Os_Restart
Os_SetRestartPoint
ShutdownAllCores
StartOS

2.83 StartCore

AUTOSAR 用として設定されているコアを起動します。

構文

```
void StartCore(
    CoreIdType CoreID,
    StatusType* Status
)
```

引数

引数	モード	説明
CoreID	in	CoreIdType コアの ID
Status	out	StatusType リターンステータス

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	このコアは存在していません。または、このコアは AUTOSAR 用として設定されていません。
E_OS_ACCESS	拡張	StartOS()より後に呼び出されました。
E_OS_STATE	拡張	このコアはすでに起動されています。

説明

マルチコア構成において StartCore は、AUTOSAR OS を実行するように設定されているコアを起動します。この API 関数は、マスタコア（マスタコアとして設定されているコアがある場合）を含む AUTOSAR コアのそれぞれについて呼び出す必要があります。

すべてのコアが OS_MAIN()から実行を開始するので、同じコアを 2 回以上起動しないように注意してください。

StartCore はマスタコアとスレーブコアのいずれからでも呼び出せます。

その後、StartOS()を各 AUTOSAR OS コアについて呼び出す必要があり、各呼び出しにおいて、同じ AppModeType、または DONOTCARE を渡す必要があります。

この API 関数は、マルチコアプロジェクト構成の場合のみ提供されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
OS_MAIN(){
  StatusType status;
  ...
  if (GetCoreID() == OS_CORE_ID_MASTER) {
    StartCore(OS_CORE_ID_0, &status);
    StartCore(OS_CORE_ID_1, &status);
  }
  StartOS(OSDEFAULTAPPMODE);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetCoreID
GetNumberOfActivatedCores
StartNonAutosarCore
StartOS

2.84 StartNonAutosarCore

AUTOSAR 用として設定されていないコアを起動します。

構文

```
void StartNonAutosarCore(
    CoreIdType CoreID,
    StatusType* Status
)
```

引数

引数	モード	説明
CoreID	in	CoreIdType コアの ID
Status	out	StatusType リターンステータス

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	このコアは存在していません。または、このコアは AUTOSAR を実行するように設定されています。
E_OS_ACCESS	拡張	StartOS()より後に呼び出されました。
E_OS_STATE	拡張	このコアはすでに起動されています。

説明

マルチコア構成において StartNonAutosarCore は、AUTOSAR OS を実行するように設定されていないコアを起動します。この API 関数は、マスタコア（マスタコアとして設定されているコアがある場合）を含む非 AUTOSAR コアのそれぞれについて呼び出す必要があります。

すべてのコアが OS_MAIN()から実行を開始するので、同じコアを 2 回以上起動しないように注意してください。

StartNonAutosarCore はマスタコアとスレーブコアのいずれからでも呼び出せます。

この API 関数は、マルチコアプロジェクト構成の場合のみ提供されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
OS_MAIN(){
  StatusType status;
  ...
  if (GetCoreID() == OS_CORE_ID_MASTER) {
    StartNonAutosarCore(OS_CORE_ID_0, &status);
    StartNonAutosarCore(OS_CORE_ID_1, &status);
  }
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetCoreID
StartCore

2.85 StartOS

指定されたモードでオペレーティングシステムを起動します。

構文

```
void StartOS(
    AppModeType Mode
)
```

引数

引数	モード	説明
Mode	in	AppModeType 起動時のアプリケーションモード

説明

StartOS()は OS のすべての内部データ構造を初期化し、Mode により指定されたモードで OS を起動します。

指定されたモードで自動起動されるタスクは、すべてレディ状態になります。

指定されたモードで自動起動されるアラームやスケジュールテーブルは、すべて適宜に初期化されます。

ソフトウェアカウンタにはゼロが初期設定されます。

OSDEFAULTAPPMODE というモードは必須ですが、それ以外のモード名は必要に応じて設定可能です。

StartOS()は、OS のコンテキストの外側からしか呼び出せません。OS が実行中のときに StartOS()が呼び出されても何の効果もありません。

OS の再起動は、StartOS()の呼び出し前に置かれた Os_SetRestartPoint()を呼び出して再起動ポイントをマークし、Os_Restart()でそのポイントにジャンプすることにより実現できます。

前提条件が満たされていない状態（CPU が間違ったモードになっている、スタックが正しくセットアップされていない、など）で呼び出されると、StartOS()は ShutdownOS(E_OS_STATE)を呼び出す可能性があります。前提条件はポートにより異なるので、ポートのユーザーガイドを参照してください。

2 つ以上のコアが AUTOSAR OS 用に設定されている場合は、各 OS コアが StartOS を呼び出し、そのうち 1 つ以上のコアが DONOTCARE 以外の Mode 値を提供する必要があります。DONOTCARE 以外の場合、すべての StartOS の呼び出しに同じ Mode 値を使用する必要があります。

OS コアから正しく呼び出されると、StartOS()は呼び出し元にリターンしません。

非 OS コアから呼び出されると、StartOS()は何の処理も行わずにリターンします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
OS_MAIN() {
  /* Initialize target hardware before starting OS */
  StartOS(OSDEFAULTAPPMODE);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DONOTCARE
Os_Cbk_Idle
Os_Restart
Os_SetRestartPoint
ShutdownAllCores
ShutdownOS

2.86 StartScheduleTableAbs

スケジュールテーブルが始動されるカウンタチック値を設定します。

構文

```
StatusType StartScheduleTableAbs(
    ScheduleTableType ScheduleTableID,
    TickType Start
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID
Start	in	TickType スケジュールテーブルが始動される絶対チック値

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_STATE	All	ScheduleTableID はすでに実行されています。
E_OS_ID	拡張	ScheduleTableID は有効なスケジュールテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_VALUE	拡張	カウンタの Start > maxallowedvalue。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました(サービス保護が設定されている場合のみ)。

説明

引数が有効であれば、ScheduleTableID を始動し、ScheduleTableID の状態を SCHEDULETABLE_RUNNING にします。

ScheduleTableID の最初の満了ポイントは Start+InitialOffset チックの時点で処理されます。InitialOffset は、ScheduleTableID の満了ポイント用に静的に定義されているオフセットのうち、最小のものです。

このオフセットが現在のカウンタ値と等しいかそれより小さい場合は、初回の満了はカウンタがラップアラウンドした後でないと発生しません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  /* Start MyScheduleTable when the associated counter reaches
     100 */
  StartScheduleTableAbs(MyScheduleTable, 100);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.87 StartScheduleTableRel

スケジュールテーブルが始動されるまでのカウンタチック数を設定します。

構文

```
StatusType StartScheduleTableRel(
    ScheduleTableType ScheduleTableID,
    TickType Offset
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID
Offset	in	TickType スケジュールテーブルが起動されるまでの相対チック数

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_STATE	All	ScheduleTableID は SCHEDULETABLE_STOPPED 状態ではありません。
E_OS_ID	拡張	ScheduleTableIDは有効なスケジュールテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_VALUE	拡張	Offset == ゼロ、または Offset > maxallowed- value - InitialOffset。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました(サービス保護が設定されている場合のみ)。

説明

引数が有効であれば、ScheduleTableID の実行を開始し、ScheduleTableID の状態を SCHEDULETABLE_RUNNING にします。

ScheduleTableID の最初の満了ポイントは Offset+InitialOffset チック経過後に処理されます。InitialOffset は、ScheduleTableID の満了ポイント用に静的に定義されているオフセットのうち、最小のものです。

この API 関数は、暗黙的に同期化されるものとして設定されているスケジュールテーブルについて呼び出すことはできません。ScheduleTableID が暗黙的に同期化されるスケジュールテーブルであった場合は、この API 関数は E_OS_ID を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  /* Start MyScheduleTable at Now+42 ticks */
  StartScheduleTableRel(MyScheduleTable, 42);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
StartScheduleTableAbs
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.88 StartScheduleTableSynchron

明示的に同期化されるスケジュールテーブルを始動して、同期化呼び出しを待ちます。

構文

```
StatusType StartScheduleTableSynchron(
    ScheduleTableType ScheduleTableID
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	ScheduleTableID は有効なスケジュールテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_STATE	拡張	ScheduleTableID の状態が SCHEDULETABLE_STOPPED ではありません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

明示的に同期化される ScheduleTableID を準備し、SyncScheduleTable()の呼び出しによって同期化カウントが提供された時に同期的に起動されるようにします。ScheduleTableID が明示的に同期化されるスケジュールテーブルでない場合は、E_OS_ID を返します。

この API 関数の呼び出しが成功すると ScheduleTableID は SCHEDULETABLE_WAITING 状態になります。ScheduleTableID の満了ポイントの処理は、このスケジュールテーブルが SCHEDULETABLE_WAITING 状態である間に SyncScheduleTable()の呼び出しが実行されるまでは開始されません。

ScheduleTableID が同期的に起動された後は、SyncScheduleTable()または StopScheduleTable()が呼び出されるまで、このスケジュールテーブルは SCHEDULETABLE_WAITING 状態のままになります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  StartScheduleTableSynchron(MyScheduleTable);
  ...
  SyncScheduleTable(MyScheduleTable, 12U);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
StartScheduleTableAbs
StartScheduleTableRel
StopScheduleTable
SyncScheduleTable

2.89 StopScheduleTable

スケジュールテーブルを停止します。

構文

```
StatusType StopScheduleTable(
    ScheduleTableType ScheduleTableID
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_NOFUNC	All	ScheduleTableID は実行中ではありません。
E_OS_ID	拡張	ScheduleTableID は有効なスケジュールテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました(サービス保護が設定されている場合のみ)。

説明

この API 関数は ScheduleTableID を直ちに停止します。StartScheduleTableAbs()、StartScheduleTableRel()、または場合に応じて StartScheduleTableSynchron()の呼び出しが ScheduleTableID を最初から再起動します。

この呼び出しが実行されると、ScheduleTableID の次に実行されることになっていたスケジュールテーブルは起動されず、SCHEDULETABLE_NEXT 状態のままになってしまうので、そのようなテーブルについても StopScheduleTable()を呼び出してテーブルの状態をリセットする必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  StopScheduleTable(MyScheduleTable);
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron

2.90 SuspendAllInterrupts

カテゴリ 1 とカテゴリ 2 の割り込みの受け付けを一時的にサスペンド（休止）します。

構文

```
void SuspendAllInterrupts(void)
```

説明

マスク可能なすべての割り込みが発生しないように保護されているクリティカルセクションを開始します。このクリティカルセクションから抜け出すには ResumeAllInterrupts()を呼び出します。

このクリティカルセクションの中では、SuspendAllInterrupts()／ResumeAllInterrupts()のペアと SuspendOSInterrupts()／ResumeOSInterrupts()のペア以外の API 関数は使用できません。

この API 関数は現在の割り込みマスクを保存し、ResumeAllInterrupts()がマスクの状態を復元できるようにします。

SuspendAllInterrupts()と ResumeAllInterrupts()の呼び出しがネストしている場合は、最初の SuspendAllInterrupts()の呼び出しによって保存された割り込みマスクの状態が、最後の ResumeAllInterrupts()の呼び出しによって復元されます。

マルチコア環境では、この API 関数が呼び出されたコア上の割り込みにのみ影響します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  SuspendAllInterrupts();
  ...
  ResumeAllInterrupts();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

DisableAllInterrupts
EnableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendOSInterrupts

2.91 SuspendOSInterrupts

カテゴリ 2 割り込みの受け付けを一時的にサスペンド（休止）します。

構文

```
void SuspendOSInterrupts(void)
```

説明

すべてのカテゴリ 2 割り込みが発生しないように保護されるクリティカルセクションを開始します。クリティカルセクション内ではカテゴリ 1 割り込みは発生する可能性があります。クリティカルセクションから抜け出すには ResumeOSInterrupts ()を呼び出します。

クリティカルセクションの中では、SuspendAllInterrupts()／ResumeAllInterrupts()のペアと SuspendOSInterrupts()／ResumeOSInterrupts()のペア以外の API 関数は使用できません。

この API 関数は現在の割り込みマスクを保存し、ResumeOSInterrupts ()がマスクの状態を復元できるようにします。

SuspendOSInterrupts()と ResumeOSInterrupts()の呼び出しがネストしている場合は、最初の SuspendOSInterrupts()の呼び出しによって保存された割り込みマスクの状態が、最後の ResumeOSInterrupts()の呼び出しによって復元されます。

マルチコア環境では、この API 関数が呼び出されたコア上の割り込みにのみ影響します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ...
  SuspendOSInterrupts();
    /* Longer Critical Section */
    ...
    SuspendAllInterrupts();
    /* Shorter Critical Section*/
    ResumeAllInterrupts();
    ...
  ResumeOSInterrupts();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

DisableAllInterrupts
EnableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendOSInterrupts

2.92 SyncScheduleTable

明示的に同期化されるスケジュールテーブル用の同期化カウントを提供します。

構文

```
StatusType SyncScheduleTable(
    ScheduleTableType ScheduleTableID,
    TickType Value
)
```

引数

引数	モード	説明
ScheduleTableID	in	ScheduleTableType スケジュールテーブルの ID
Value	in	TickType 同期化カウンタの絶対値

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	All	ScheduleTableID は明示的に同期化されるテーブルではありません。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから ScheduleTableID にアクセスできません。
E_OS_VALUE	All	値がテーブルの期間より大きいです。
E_OS_STATE	All	ScheduleTableID の状態が SCHEDULETABLE_STOPPED または SCHEDULETABLE_NEXT です。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。

説明

明示的に同期化されるテーブル ScheduleTableID に同期化カウント値を提供します。ScheduleTableID は同期化の値を待っている状態、または実行中である必要があります。

同期化カウンタの制御と同期化カウンタについての情報は、OS の領域外になります。OS は、同期化カウンタの期間が ScheduleTableID と等しいこと、および同期化カウンタの分解能と ScheduleTableID の駆動に使用される OS カウンタの分解能とが等しいことを想定しています。そのため、ユーザーの責任において、アプリケーションがこれらの制約条件を満足していることを確認する必要があります。

ScheduleTableID の状態が SCHEDULETABLE_WAITING である場合は、SyncScheduleTable() が呼び出されると ScheduleTableID の状態が

SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS に変わり、OS は満了ポイントの処理を開始します。ScheduleTableID と同期化カウントとの差はゼロになります。

処理される最初の満了ポイントは、静的に設定されているオフセットの中で値が最小のものを使用して決められます。この最小のオフセットは InitialOffset（初期オフセット）と呼ばれます。最初の満了ポイントが処理されるポイントは以下のようにして決められます。

- Value が InitialOffset より小さい場合は、ScheduleTableID を駆動するカウンタの InitialOffset- Value チックが経過したときに最初の満了ポイントが処理されます。
- Value が InitialOffset と等しいかそれより大きい場合は、(Duration-Value)+InitialOffset チックが経過したときに最初の満了ポイントが処理されます。この場合、駆動カウンタがフルラップしてからでないと最初の満了ポイントが処理されない可能性があります。

つまり、ScheduleTableID が SCHEDULETABLE_WAITING 状態のときに SyncScheduleTable()が呼び出された場合の挙動は、オフセットが InitialOffset- Value と等しいか(Duration- Value)+InitialOffset の状態で StartScheduleTableRel() が呼び出された場合の挙動と論理的に同等になります。

ScheduleTableID の状態が SCHEDULETABLE_RUNNING または SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS である場合は、SyncScheduleTable() は ScheduleTableID 上の概念位置と Value との差を求めます。この差は P-Value mod Duration（P 値を Duration で割った剰余）と等しくなります。ScheduleTableID は、計算された差と静的に設定されている精度との差に応じて以下の状態になります。

- 「差 <= 精度」の場合は、SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS
- 「差 > 精度」の場合は、SCHEDULETABLE_RUNNING

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
TASK(MyTask){
  StartScheduleTableSynchron(MyScheduleTable);
  ...
  SyncScheduleTable(MyScheduleTable, 12U);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.93 TerminateApplication

OS アプリケーションをターミネートします。

構文

```
StatusType TerminateApplication(
    ApplicationType Application /* AUTOSAR R4.x only */,
    RestartType RestartOption
)
```

引数

引数	モード	説明
Application	in	ApplicationType OS アプリケーション（AUTOSAR R4.x のみ）
RestartOption	in	RestartType RESTART または NO_RESTART

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_VALUE	All	RestartOption が RESTART でも NO_RESTART でもありません。
E_OS_CALLEVEL	All	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_STATE	All	（AUTOSAR R4.x のみ）この OS アプリケーションがすでに APPLICATION_TERMINATED 状態のとき、または APPLICATION_RESTARTING 状態で RestartOption が RESTART のときに呼び出されました。または、この OS アプリケーションが APPLICATION_RESTARTING 状態で、呼び出し元がこの OS アプリケーションに属していません。
E_OS_ACCESS	All	（AUTOSAR R4.x のみ）アントラステッドである別の OS アプリケーションから呼び出されました。
E_OS_ACCESS	All	最適化'Only Terminate Untrusted Applica- tions'が設定されている状態で、トラステッド OS アプリケーションについて呼び出されました。
E_OS_ID	All	（AUTOSAR R4.x のみ）Application は有効な OS アプリケーションではありません。

説明

注記： AUTOSAR R4.0 より前のバージョンでは、この関数は呼び出し元のタスク／ISR を所有している OS アプリケーションについてのみ機能します。

注記： AUTOSAR R4.0 以降では、どの OS アプリケーションをターミネートするかを、トラステッドのタスク／ISR が指定できます。

この API 関数は OS アプリケーションをターミネートします。その OS アプリケーションの実行中のタスク／ISR は強制的にターミネートされ、レディ状態のタスク／ISR はレジュームする前に強制的にターミネートされます。

その OS アプリケーションが所有するすべてのカテゴリ 2 ISR の割り込みソースは、その OS アプリケーションが所有する各 ISRName について OS が Os_Cbk_Disable_<ISRName>()を呼び出すことによりディセーブルになります。

そのOSアプリケーションが所有するすべてのアラームはキャンセルされ、すべてのスケジュールテーブルは停止されます。

標準（Standard）、リンク（Linked）、内部（Internal）の種別を問わずいずれかのリソースを保持しているタスク／ISR がある場合は、すべてのリソースが解放されます。同様に、Suspend[All|OS]Interrupts()またはDisableAllInterrupts()サービスコールを使用して割り込みをマスクしていたタスク／ISR がある場合は、OS は必要な Resume[All|OS]Interrupts()または EnableAllInterrupts()のサービスを自動的に呼び出します。

スピンロックを保持しているタスク／ISR がある場合は、それらのスピンロックは解放されます（マルチコアの場合のみ）。

RestartOption が RESTART である場合は、この OS アプリケーションの再起動タスクが起動されます。

TerminateApplication()を使用すると、アプリケーションにおいて以下の競合状態が発生する可能性があるので、注意してください。

- OS アプリケーション間で共有されているクリティカルセクションを保護する目的でリソースのロックや割り込みのマスクが行われていた場合は、タスク／ISR が強制的にターミネートされることにより、クリティカルセクション内で扱われていたデータが未知の状態のままになってしまう可能性があります。アプリケーション側の責任においてこのような状態を回避してください。
- 再起動タスクの優先度が当該 OS アプリケーション内のタスクの中で最も高くなっていない限り、再起動タスクが当該 OS アプリケーション内の他のどのタスクよりも先に実行される、という保証はありません。
- 他のOSアプリケーションからアクセスされているカウンタは、ターミネーション後、そのカウンタを駆動する割り込みが再度イネーブルになるまでは使用できなくなり、この障害が他の OS アプリケーションに伝播する可能性があります。
- 他の OS アプリケーションのタスク（およびそれらに関連するイベントセット）のうち、ターミネートされた OS アプリケーション内のアラームまたはスケジュールによりトリガされるものは、起動（またはセット）されなくなります。そのため、他の OS アプリケーションが正常に動作しなくなる可能性があります。
- （AUTOSAR R3.x のみ）この OS アプリケーションがターミネートされた後も、その OS アプリケーションのいずれかのオブジェクトにアクセスできる他の OS アプリケーションが、そのオブジェクトを使用する可能性があります。たとえば、ターミネートされた OS アプリケーション内のタスクを別の OS アプリケーションが起動する可能性があります。

TerminateApplicationを使用して他のコアのアプリケーションをターミネートすることができますが、それによってタイミングに関する微妙な問題が生じる可能性があるので、

TerminateApplication は、ターミネートされるアプリケーションが含まれているコアから呼び出すことをお勧めします。

この API 関数は、OS オプション'Omit TerminateApplication'が TRUE の場合、または OS オプション'Only Terminate Untrusted Applications'が TRUE でアントラステッド OS アプリケーションが存在しない場合は、提供されません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

例

```
TASK(MyTask){
  ...
  if (ErrorDetected == TRUE) {
#if (OS_AR_MAJOR_VERSION < (4U))
    TerminateApplication(RESTART);
#else
    TerminateApplication(GetApplicationID(), RESTART);
#endif
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

参照

AllowAccess
GetApplicationState
Os_Cbk_Disable_<ISRName>
Os_Cbk_Terminated_<ISRName>
TerminateTask

2.94 TerminateTask

呼び出し元のタスクをターミネートします。

構文

StatusType TerminateTask(**void**)

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_RESOURCE	拡張	呼び出し元のタスクがまだリソースを保持しています。
E_OS_CALLEVEL	拡張	割り込みレベルで呼び出されました。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました(サービス保護が設定されている場合のみ)。
E_OS_SPINLOCK	拡張	このタスクがスピンロックを保持している間に呼び出されました。
E_OS_MISSINGEND	拡張	あるタスクが TerminateTask または ChainTask を呼び出さずに終了しました（保護が設定されている場合）

説明

呼び出し元のタスクをターミネートします。つまり呼び出し元のタスクを実行状態からサスペンド状態に移行させます。成功した場合は呼び出し元に戻りません。

呼び出し元のタスクの起動がキューイングされてペンディング状態になっている場合は、そのタスクの次のインスタンスが自動的にレディ状態に移行します。

内部リソースは自動的に解放されます。

標準またはリンクリソースも自動的に解放され、これは拡張ステータスにおいてはエラー条件としてレポートされます。

TerminateTask()を呼び出すと、必ずリスケジューリングが行われます。

RTA-OSの最適化オプションで'Fast Terminate'が有効になっている場合は、TerminateTask()は必ずタスクのエントリ関数から呼び出す必要があり、リターンステータスはチェックしません。エラーが発生した場合に限り、ErrorHook が設定されていればそれが呼び出されます。この最適化によりメモリと実行時間が節約され、さらに節約が必要な場合は、SC1 および SC2 において TerminateTask()の呼び出しを実際に割愛することができます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask){
  ... TerminateTask();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.95 TryToGetSpinlock

スピンロック変数の占有を試みます。

構文

```
TryToGetSpinlock(
    SpinlockIdType SpinlockId,
    TryToGetSpinlockType* Success
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID
Success	out	Os_TryToGetSpinlockRefType 実行結果: TRYTOGETSPINLOCK_SUCCESS または TRYTOGETSPINLOCK_NOSUCCESS

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ID	拡張	SpinlockId は有効なスピンロックを参照していません。
E_OS_STATE	拡張	このスピンロックは呼び出し元のタスク／ISR によりすでに占有されています（4.0.3 より前）。
E_OS_INTERFERENCE_DEADLOCK	拡張	このスピンロックは同じコアのタスク／ISR によりすでに占有されています。これはデッドロックの原因になります。
E_OS_NESTING_DEADLOCK	拡張	別のスピンロックをすでに保持しているときに、デッドロックを招く可能性のある形でこのスピンロックを占有しようとしました。
E_OS_ACCESS	拡張	呼び出し元の OS アプリケーションから SpinlockId にアクセスできません。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。

説明

TryToGetSpinlock の挙動は GetSpinlock と同様ですが、要求されたスピンロックがすでに別のコアにより占有されている場合は、このスピンロックを待たずに E_OK を返し、*Success に TRYTOGETSPINLOCK_NOSUCCESS をセットします。

このスピンロックを占有できる場合は、E_OK を返して*Success に TRYTOGETSPINLOCK_SUCCESS を設定します。

スピンロックのロックメソッド（コンテナ: OsSpinlock）のタイプに応じて、この API 関数の挙動は以下のように変わります。

- LOCK_ALL_INTERRUPTS、LOCK_CAT2_INTERRUPTS、LOCK_WITH_RES_SCHEDULER ― ロックが成功する場合だけ有効です。ロックが成功しない場合は、システムのスケジューラビリティは変更されません。
- NESTABLE ―同じコア上のタスク／ISR によってスピンロックがすでに保持されていると、「ロック成功」を知らせます。ロックの解除は、そのロックを行ったコードが、実行した Get コールと同じ数の Release コールを実行したときに限り行われます。
- COMMONABLE ― そのスピンロックの後に、独自のサクセッサを持たず COMMONABLE でもない任意のスピンロックを実行することができます。このスピンロックは、通常のロックの後に、そのサクセッサリストに載らずに実行することができます。

OS コンフィギュレーションオプション'Force spinlock error checks'を使用すると、拡張ステータスのビルド以外に標準ステータスのビルドでもエラーチェックが行われます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
TASK(MyTask){
  TryToGetSpinlockType try_result;
  ...
  TryToGetSpinlock(Spinlock1, &try_result);
  if (TRYTOGETSPINLOCK_SUCCESS == try_result) {
    ...
    ReleaseSpinlock(Spinlock1);
  }
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

GetSpinlock
ReleaseSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.96 UncheckedGetSpinlock

スピンロック変数の占有を試みます。

構文

```
UncheckedGetSpinlock(
    SpinlockIdType SpinlockId
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。

説明

カテゴリ 1 ISR から呼び出すことのできるバージョンの GetSpinlock ですが、注意が必要です。カテゴリ 1 ISR の中では十分なチェックが行えないので、LOCK_ALL タイプのスピンロック以外には使用しないことをお勧めします。

この API 関数は OS オプション'Add Spinlock APIs for CAT1 ISRs'が有効に設定されている場合のみ使用できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
CAT1ISR(MyISR){
  ...
  UncheckedGetSpinlock(Spinlock1);
  ...
  UncheckedReleaseSpinlock(Spinlock1);
}
```

呼び出し元コンテキストコンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

GetSpinlock
ReleaseSpinlock
TryToGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.97 UncheckedReleaseSpinlock

スピンロック変数を解放します。

構文

```
UncheckedReleaseSpinlock(
    SpinlockIdType SpinlockId
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。

説明

カテゴリ 1 ISR から呼び出すことのできるバージョンの ReleaseSpinlock ですが、注意が必要です。カテゴリ 1 ISR の中では十分なチェックが行えないので、LOCK_ALL タイプのスピンロック以外には使用しないことをお勧めします。

この API 関数は OS オプション'Add Spinlock APIs for CAT1 ISRs'が有効に設定されている場合のみ使用できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
CAT1ISR(MyISR){
  ...
  UncheckedGetSpinlock(Spinlock1);
  ...
  UncheckedReleaseSpinlock(Spinlock1);
}
```

呼び出し元コンテキストコンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

GetSpinlock
ReleaseResource
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedTryToGetSpinlock

2.98 UncheckedTryToGetSpinlock

スピンロック変数の占有を試みます。

構文

```
UncheckedTryToGetSpinlock(
    SpinlockIdType SpinlockId,
    TryToGetSpinlockType* Success
)
```

引数

引数	モード	説明
SpinlockId	in	SpinlockIdType スピンロックの ID
Success	out	Os-TryToGetSpinlockRefType 実行結果: TRYTOGETSPINLOCK_SUCCESS または TRYTOGETSPINLOCK_NOSUCCESS

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。

説明

カテゴリ 1 ISR から呼び出すことのできるバージョンの TryToGetSpinlock ですが、注意が必要です。カテゴリ 1 ISR の中では十分なチェックが行えないので、LOCK_ALL タイプのスピンロック以外には使用しないことをお勧めします。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
CAT1ISR(MyISR){
  UncheckedTryToGetSpinlockType try_result;
  ...
  UncheckedTryToGetSpinlock(Spinlock1, &try_result);
  if (TRYTOGETSPINLOCK_SUCCESS == try_result) {
    ...
    UncheckedReleaseSpinlock(Spinlock1);
  }
}
```

呼び出し元コンテキストコンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

参照

GetSpinlock
ReleaseSpinlock
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock

2.99 WaitEvent

1 つまたは複数のイベントを待ちます。

構文

```
StatusType WaitEvent(
    EventMaskType Mask
)
```

引数

引数	モード	説明
Mask	in	EventMaskType イベントの ID（1 つまたは複数）

戻り値

StatusType 型の値を返します。

値	ビルド	説明
E_OK	All	エラーなし。
E_OS_ACCESS	拡張	拡張タスクから呼び出されていません。
E_OS_CALLEVEL	拡張	割り込みレベルから呼び出されました。
E_OS_RESOURCE	拡張	呼び出し元のタスクはリソースを保持しています。
E_OS_CALLEVEL	拡張	無効なコンテキストから呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_DISABLEDINT	拡張	割り込みがディセーブルになっている状態において呼び出されました（サービス保護が設定されている場合のみ）。
E_OS_SPINLOCK	拡張	タスクがスピンロックを保持しているときに呼び出されました。

説明

指定されたイベントのいずれかがセットされるまで、呼び出し元のタスクを待ち状態にします。

指定されたイベントのいずれかがすでにセットされている場合は、呼び出し元のタスクは実行状態のままになります。

この API 関数の呼び出しによってリスケジューリングが行われる場合があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyExtendedTask){
  ...
  WaitEvent(Event1);
  /* Task resumes here when Event1 is set */
```

```
  ...
}
```

呼び出し元コンテキストコンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

ClearEvent
DeclareEvent
GetEvent
SetEvent

3 RTA-OS のコールバック関数

3.1 説明の形式

コールバック関数は、ユーザーが提供する OS 用関数です。本章では RTA-OS が使用するすべてのコールバック関数について説明します。各コールバック関数の情報は、すべて以下の構成で記述されています。

構文

```
/* C function prototype for the callback （コールバック関数の C 関数プロトタイプ）*/
Return Value NameOfCallback(Parameter Type, ...)
```

引数

コールバック関数の引数とそのモードが示されています。

in　OSがコールバック関数に渡す引数

out　OSがコールバック関数に引数への参照（ポインタ）を渡し、コールバック関数がそこに値をセットして返します。

inout OS がコールバック関数に引数を渡し、コールバック関数がそれを更新して返します。

戻り値

コールバック関数からの戻り値の説明です。

説明

コールバック関数に要求される挙動の詳細説明です。

可搬性

他の環境（OSEK OS、AUTOSAR OS、RTA-OS、RTA- TRACE）への可搬性

コーティング例

```
A C code listing showing how to use the API calls
```
（コールバック関数呼び出しのコーディング例）

RTA-OS コンフィギュレーション

当該コールバック関数が必要となる RTA-OS のコンフィギュレーション設定が示されています。

参照

関連するコールバック関数の一覧（リンク）です。

3.2 ErrorHook

OS API関数の間違った使い方が原因で発生したエラーをトラップするためのコールバック関数です。

構文

```
FUNC(void, {memclass})
    ErrorHook( StatusType Error
)
```

引数

引数	モード	説明
Error	in	StatusType エラーのID

説明

API関数がE_OK以外のStatusTypeを返す際に呼び出され、StatusTypeがErrorHook()に渡されます。

エラーの原因についての情報を取得するマクロを使用するには、OSの設定において当該マクロが生成されるように設定する必要があります。

これらのマクロは必ずErrorHook()内で使用してください。

(1) マクロOSErrorGetServiceId()は、エラーが発生したAPI関数を示すOSServiceIdTypeを返します。この値は、OSServiceId_xxx（"xxx"はAPI関数名）という形を取ります（例: OSServiceId_ActivateTask）。

(2) OSError_<APIName>_<ParameterName>()というマクロは、API関数に渡された引数の値を返します（例: OSError_ActivateTask_TaskID()）。

ErrorHookはOSレベルで実行されるので、タスクやカテゴリ2 ISRがこれをプリエンプトすることはありません。

サンプルのErrorHookをrtaosgenで自動的に生成することができます。詳細は『RTA-OSユーザーズガイド』を参照してください。

注記: 下記のコード内のmemclassは、AUTOSAR 3.xの場合はOS_APPL_CODE、AUTOSAR 4.0の場合はOS_CALLOUT_CODE、AUTOSAR 4.1の場合はOS_ERRORHOOK_CODEです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  switch (Error){
    case E_OS_ID:
      /* Handle illegal identifier error */
      break;
    case E_OS_VALUE:
      /* Handle illegal value error */
      break;
    case E_OS_STATE:
      /* Handle illegal state error */
      break;
    default:
      /* Handle all other types of error */
      break;
  }
}
```

RTA-OS コンフィギュレーション

ErrorHook がセットされている場合は必須です。

3.3 Os_Cbk_Cancel_<CounterID>

ハードウェアカウンタからの割り込みをキャンセルするコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_Cancel_<CounterID>(void)
```

説明

このコールバック関数は、実行中のアラームまたはスケジュールテーブルがない状態において OS によって Os_AdvanceCounter 内から呼び出され、ハードウェアカウンタがマッチをアサートするのを阻止するものです。

そのためには割り込みソースをディセーブルにします。さらに、関連するすべての割り込みペンディングビットもクリアする必要もあります。

ハードウェアカウンタ自体はカウント動作を続けます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_CANCEL_<COUNTERID>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_Cancel_MyCounter(void) {
  DISABLE_HW_COUNTER_INTERRUPT_SOURCE;
  CLEAR_HW_COUNTER_PENDING_INTERRUPT;
}
```

RTA-OS コンフィギュレーション

コンフィギュレーションに含まれるハードウェアカウンタごとに必須です。

参照

Os_Cbk_Now_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>
Os_AdvanceCounter

3.4 Os_Cbk_CheckMemoryAccess

指定された OS アプリケーションからメモリ領域への各種アクセス（読み取り、書き込み、実行、スタック）が可能かどうかを調べます。

構文

```
FUNC(AccessType, {memclass}) Os_Cbk_CheckMemoryAccess(
    ApplicationType Application,
    TaskType TaskID,
    ISRType ISRID,
    MemoryStartAddressType Address,
    MemorySizeType Size
)
```

引数

引数	モード	説明
Application	in	ApplicationType タスク／ISR が属している OS アプリケーション
TaskID	in	TaskType INVALID_TASK 以外の場合： 調べる対象のタスク
ISRID	in	ISRType INVALID_ISR 以外の場合：調べる対象の ISR
Address	in	MemoryStartAddressType メモリ領域の先頭アドレス
Size	in	MemorySizeType メモリ領域のサイズ（バイト数）

戻り値

AccessType 型の値を返します。

説明

アントラステッド OS アプリケーションがコンフィギュレーション内に存在し、API 関数'Get'に渡されるアドレスが書き込み先アドレスとして正当かどうかを調べる際に、OS はこのコールバック関数を呼び出します。たとえば、GetAlarm()には値の書き込み先として TickRefType 型の引数が 1 つ渡されますが、そのアドレスに値を書き込む前に Os_Cbk_CheckMemoryAccess を呼び出して正当性が確認されます。

API 関数 CheckTaskMemoryAccess()または CheckISRMemoryAccess()を呼び出した際にも、OS は Os_Cbk_CheckMemoryAccess を呼び出します。

これは、プロジェクトに適用されたアクセス許可をユーザーがコントロールするためのものです。たとえば、アントラステッドコードに対して書き込みアクセスを制限し、読み取りと実行のアクセスはすべて許可する、という選択が可能です。あるいは、アントラステッドコードについて読み取り、書き込み、実行のアクセスを制限することも可能です。

このコールバック関数では、Address から (Address + Size) までのメモリ領域に対するこの OS アプリケーション（実行されているタスク／ISR）からの各種アクセス（読み取り、書き込み、実行、スタック）が可能かどうかを明らかにする必要があります。

API 関数 CheckTaskMemoryAccess()に対応してこのコールバック関数を呼び出す場合は、OS は ISRID の値を INVALID_ISR にセットし、CheckISRMemoryAccess()に対応して呼び出す場合は、TaskID の値を INVALID_TASK にセットします。

返される AccessType には、以下の定数を使用できます。

- OS_ACCESS_READ - このメモリ領域は読み取り可能です。
- OS_ACCESS_EXECUTE - このメモリ領域は実行可能です。
- OS_ACCESS_WRITE - このメモリ領域は書き込み可能です。
- OS_ACCESS_STACK - このメモリ領域はスタックです。

注記：下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_CHECKMEMORYACCESS_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(AccessType, {memclass}) Os_Cbk_CheckMemoryAccess(ApplicationType
    Application, TaskType
    TaskID, ISRType ISRID, MemoryStartAddressType Address,
    MemorySizeType Size) {
  AccessType Access = OS_ACCESS_EXECUTE;
  /* Check for stack space in address range */
  if ((Address >= STACK_BASE) && (Address + Size < STACK_BASE +
      STACK_SIZE) ) {
      Access |= OS_ACCESS_STACK;
  }
  /* Address range is read/write if it is in RAM */
  if ((Address >= RAM_BASE) && (Address + Size < RAM_BASE +
      RAM_SIZE) ) {
      Access |= (OS_ACCESS_WRITE | OS_ACCESS_READ);
  }
  switch (Application) {
    case APP1:
        /* Trusted application - no further restrictions */
        break;
      case APP2:
        /* Untrusted application - write restrictions */
        if ((Address <= APP2_BASE) || (Address + Size > APP2_BASE +
            APP2_SIZE) ) {
          Access &= ~OS_ACCESS_WRITE;
        }
        break;
    }
    return Access;
  }
```

RTA-OSコンフィギュレーション

メモリアクセスチェックが使用される場合は必須です。

参照

CheckISRMemoryAccess
CheckTaskMemoryAccess
Os_Cbk_SetMemoryAccess

3.5 Os_Cbk_CheckStackDepth

ランタイムにシステムのスタック使用量を監視するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_CheckStackDepth(
    CoreIdTⅴype Core_id,
    Os_StackSizeType Depth,
    Os_StackSizeType CurrentPos
)
```

引数

引数	モード	説明
Core_id	in	CoreIdType 呼び出し元コアの ID（シングルコア構成の場合は OS_CORE_ID_MASTER）
Depth	in	Os_StackSizeType StartOS 以降に使用されたスタックの量
CurrentPos	in	Os_StackOverrunType 呼び出しの時点のスタックポインタの値

説明

OS オプションの'Stack Sampling'が有効になっていると、各タスク／ISR が起動する直前にこのコールバック関数が呼び出されます。

このコールバック関数は、個々のタスクや ISR のスタック使用量ではなく、システム全体のスタック使用量を監視するためのものです。

API 関数 Os_GetStackUsage からも呼び出されるので、アプリケーションコード内に監視ポイントを置くことがことができます。

このコールバック関数の中で API 関数 GetISRID および GetTaskID を呼び出すことにより、どのタスク／ISR が実行中か（または実行されようとしているか）を調べることができます。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_CHECKSTACKDEPTH_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

RTA-OSコンフィギュレーション

OS オプションの'Stack Sampling'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

Os_GetStackUsage
Os_GetISRMaxStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage
GetISRID
GetTaskID

3.6 Os_Cbk_CrosscoreISREnd

クロスコア ISR の終了を示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_CrosscoreISREnd(
    CoreIdType core_id
)
```

説明

マルチコア構成 においてOSオプション'Additional ISR Hooks'が有効になっている場合は、Os_Cbk_CrosscoreISREnd()を実装する必要があります。このコールバック関数は、クロスコア ISR ハンドラが実行を終了する時点で呼び出されます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_CROSSCOREISREND_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 CrosscoreEndCount[OS_NUM_CORES];
FUNC(void, {memclass}) Os_Cbk_CrosscoreISREnd(CoreIdType core_id)
    {
  CrosscoreEndCount[core_id] += 1;
}
```

RTA-OS コンフィギュレーション

マルチコア構成 においてOSオプション'Additional ISR Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_CrosscoreISRStart

3.7 Os_Cbk_CrosscoreISRStart

クロスコア ISR の開始を示すコールバック関数です。

構文

```
FUNC(void,{memclass})Os_Cbk_CrosscoreISRStart(
    CoreIdTypecore_id
)
```

説明

マルチコア構成において OS オプション'Additional ISR Hooks'が有効になっている場合は、Os_Cbk_CrosscoreISRStart()を実装する必要があります。このコールバック関数は、クロスコア ISR ハンドラが実行を開始する時点で呼び出されます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_CROSSCOREISRSTART_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32CrosscoreStartCount[OS_NUM_CORES];
FUNC(void,{memclass})Os_Cbk_CrosscoreISRStart(CoreIdType core_id){
  CrosscoreStartCount[core_id]+=1;
}
```

RTA-OS コンフィギュレーション

マルチコア構成 においてOS オプション'Additional ISR Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_CrosscoreISREnd

3.8 Os_Cbk_Disable_<ISRName>

ISR <ISRName>に関連する割り込みソースをディセーブルするコールバック関数です。

構文

```
FUNC(void,{memclass})Os_Cbk_Disable_<ISRName>(void)
```

説明

OS は TerminateApplication の処理中にこの関数を呼び出し、指定された ISR に関連付けられている割り込みソースをディセーブルにすることを要求します。

AUTOSAR では、ある 1 つの OS アプリケーションがターミネートされるときにはその OS アプリケーションに属するすべての割り込みをディセーブルにすることが要件となっています。

OS アプリケーションの割り込みは、一般的に、その再起動タスクの中で再びイネーブルにされます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_DISABLE_<ISRNAME>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_Disable_App2Isr1(void) {
    disable_interrupt_source(_App2Isr1_);
}
```

RTA-OS コンフィギュレーション

TerminateApplication がサポートされている場合は、ISR ごとに必須です。

参照

ProtectionHook
TerminateApplication

3.9 Os_Cbk_GetStopwatch

フリーランニングカウンタのカレント値を取得するコールバック関数です。

構文

```
FUNC(Os_StopwatchTickType,{memclass})Os_Cbk_GetStopwatch(void)
```

戻り値

Os_StopwatchTickType 型の値を返します。

説明

Os_Cbk_GetStopwatch()は、インクリメントされレンジの終わりでオーバーフローするフリーランニングタイマのカレント値を返します。

このタイマは、実行時間とトレース測定のためのタイムベースを提供するものです。

このコールバック関数は、OS がタイミング測定を行っている際に非常に頻繁にアクセスされますが、コード内にインライン化して埋め込むことによりパフォーマンスを高めることができます。これを行うには、ヘッダファイル内に C プリプロセッサマクロとして Os_Cbk_GetStopwatch を実装し、-using というコマンドラインオプションを使用してそのヘッダファイルをビルドに含めます（下のコード例の場合では -using:MyStopwatch.h）。

-using コマンドラインオプションを使用してビルドに差し込まれるファイルは他のどの OS ヘッダファイルよりも前に読み取られるので、各ペリフェラルの指定は、名前ではなく、ハードコードされたアドレスを使用する必要があります。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_GETSTOPWATCH_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(Os_StopwatchTickType,{memclass})Os_Cbk_GetStopwatch(void){
  return(Os_StopwatchTickType)HARDWARE_TIMER_CHANNEL;
}
```

または、ファイル MyStopwatch.h 内に以下のマクロを定義:

```
#define Os_Cbk_GetStopwatch()
    ((Os_StopwatchTickType)HARDWARE_TIMER_CHANNEL)
```

RTA-OSコンフィギュレーション

OS に時間監視またはトレースが設定されている場合は、このコールバック関数を実装する必要があります。

参照

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

3.10 Os_Cbk_ISREnd

カテゴリ 2 ISR の終了を示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_ISREnd(
    ISRType isr
)
```

説明

OS オプション'Additional ISR Hooks'が有効になっている場合は、Os_Cbk_ISREnd() を実装する必要があります。このコールバック関数はカテゴリ 2 ISR が実行を終了する時点で呼び出されます。

ISR がプリエンプトされる際には、このコールバック関数は呼び出されません。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_ISREND_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType isr_exe_time[OS_NUM_ISRS];
FUNC(void, {memclass}) Os_Cbk_ISREnd(ISRType isr) {
  isr_exe_time[OS_ISRTYPE_TO_INDEX(isr)] = GetExecutionTime();
}
```

RTA-OS コンフィギュレーション

OS オプション'Additional ISR Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

OS_ISRTYPE_TO_INDEX
Os_Cbk_TaskStart
Os_Cbk_ISRStart
Os_Cbk_TaskEnd
Os_Cbk_CrosscoreISRStart
Os_Cbk_CrosscoreISREnd

3.11 Os_Cbk_ISRStart

カテゴリ 2 ISR の開始を示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_ISRStart(
    ISRType isr
)
```

説明

OS オプション'Additional ISR Hooks'が有効になっている場合は、Os_Cbk_ISRStart()を実装する必要があります。このコールバック関数はカテゴリ 2 ISR が実行を開始しようとする時点で呼び出されます。

ISR がプリエンプトされる際には、このコールバック関数は呼び出されません。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_ISRSTART_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 isr_starts[OS_NUM_ISRS];
FUNC(void, {memclass}) Os_Cbk_ISRStart(ISRType isr) {
  isr_starts[OS_ISRTYPE_TO_INDEX(isr)] += 1;
}
```

RTA-OS コンフィギュレーション

OS オプション'Additional ISR Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

OS_ISRTYPE_TO_INDEX
Os_Cbk_TaskEnd
Os_Cbk_TaskStart
Os_Cbk_ISREnd
Os_Cbk_CrosscoreISRStart
Os_Cbk_CrosscoreISREnd

3.12 Os_Cbk_Idle

OS がアイドル状態になる際に実行されます。

構文

```
FUNC(boolean, {memclass}) Os_Cbk_Idle(void)
```

戻り値

boolean 型の値を返します。

説明

Os_Cbk_Idle()は、OS が起動後初めてアイドル状態になる際に呼び出されます。このコールバック関数は、自動起動されるタスクがすべて実行された後に呼び出されます。

Os_Cbk_Idle()は、戻り値 TRUE で終了した場合は直ちに再び呼び出されます。FALSE の場合は、再び呼び出されることはなく、レディ状態のタスク／ISR がなければ、OS はビジーウェイト状態になります。

FALSE を返すデフォルトのコードがライブラリに含まれています。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_IDLE_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(boolean, {memclass}) Os_Cbk_Idle(void) {
  sleep();
  return TRUE;
}
```

RTA-OS コンフィギュレーション

ユーザーコード内では省略可能です。設定は必要ありません。

参照

StartOS
ShutdownOS
ShutdownAllCores

3.13 Os_Cbk_IsSystemTrapAllowed

呼び出し元からシステムトラップにアクセスしてもよいかどうかを判断するコールバック関数です。

構文

```
FUNC(boolean, {memclass}) Os_Cbk_IsSystemTrapAllowed(
    MemoryStartAddressType Caller
)
```

戻り値

boolean 型の値を返します。

説明

一般的なRTA-OSターゲットは、システムトラップを使用してプロセッサをトラステッドモードからアントラステッドモードに切り替えます。'Guard supervisor access'ターゲットオプションを使用してシステムトラップへのアクセスを制限しても、アントラステッドコードがシステムトラップを呼び出して、不当な許可を得てしまう可能性があります。

これは、Os_Cbk_IsSystemTrapAllowed()を使用して防ぐことができます。このコールバック関数は、OSがトラステッドモードへの切り替えを実行する直前にOSから呼び出されます。このコールバック関数には、トラップコールを行ったコードのリターンアドレスが渡されるので、それにより、このコードのトラステッドモードへの切り替えが許されるかどうかをユーザーが判断することができます。

このコールバック関数が TRUE を返すとトラステッドへの切り替えが許されます。FALSE を返すと切り替えは行われず、トラップは何も行わずにリターンします。

このコールバック関数は、トラップへの有効なアクセスを規制することのできる、システムのセキュアなパートに実装されることが想定されています。このコードは通常は呼び出し元のアドレスを参照して、そのトラップが許されるものかどうかを決定します。その際、GetTaskID／GetISRID のみを信頼することはできません。これは、アントラステッドのタスク／ISR が完了する際には、OS はトラップを使用してトラステッドモードに戻る必要があるためです。OS はこれを行ってからでないと、実行中のタスク／ISR の ID を変更できません。

一般的なアプリケーションでは、このようなモード切り替えを行う必要があるのは OS コードだけです。シンプルなコード実装を行ってすべての OS API 関数が隣接する小さなメモリブロックに配置されるようにすれば、その範囲内に呼び出し元があるかどうかを調べるだけで済みます。

「ヘルパー関数」（システムトラップを使用してトラステッドモードへの切り替えを行う関数）は実装しないでください。アントラステッドコードからヘルパー関数が呼び出されてしまうと、その呼び出し元はアントラステッドコードではなくヘルパー関数ということになるので、防御が破られてしまいます。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_ISSYSTEMTRAPALLOWED_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(boolean, {memclass})
  Os_Cbk_IsSystemTrapAllowed(MemoryStartAddressType Caller) {
    return ( (Caller >= MyTrustedCodeStartAddress) && (Caller <=
  MyTrustedCodeEndAddress));
}
```

RTA-OS コンフィギュレーション

ターゲットオプション'Enhanced Isolation'が有効になっている場合は、このコールバック関数を実装する必要があります。ターゲットによってはこのオプションをサポートしていないものがあるので、注意が必要です。

参照

Os_Cbk_RestoreGlobalRegisters
Os_Cbk_IsUntrustedCodeOK
Os_Cbk_IsUntrustedTrapOK

3.14 Os_Cbk_IsUntrustedCodeOK

アントラステッドコードの有効性を調べるコールバック関数です。

構文

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedCodeOK(
    { parameters are target-dependent}
)
```

戻り値

ProtectionReturnType 型の値を返します。

説明

'Enhanced Isolation'ターゲットオプションが有効になっていると、OS は、アントラステッドコードが何らかの形でそのレジスタやアクセス可能なすべてのシステムコンテキストを破壊する可能性があると想定します。

この場合、アントラステッドコードをプリエンプトするすべての割り込みは OS によってインターセプトされます。コールバック関数には、アントラステッドコードによって使用されているレジスタの詳細が渡されるので、その内容が、アントラステッドコードが不正な挙動を取ったことを示しているかどうかを判断する必要があります。

戻り値の意味は以下のとおりです。

- PRO_IGNORE — 割り込み完了後もコードの実行を続けることが許されます。
- PRO_TERMINATETASKISR — アントラステッドコードは正しくターミネートされ、割り込みが実行されます。その後システムは、アントラステッドコードが正しく完了したかのように処理を続行します。
- PRO_TERMINATEAPPL — PRO_TERMINATETASKISR の場合と同じ処理を行い、さらに、当該のアントラステッドを所有している OS アプリケーションを構成している他のすべてのタスクや ISR もターミネートします。
- PRO_TERMINATEAPPL_RESTART — PRO_TERMINATEAPPL の場合と同じ処理を行い、さらに、適切な再起動タスクの起動も行います。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_ISUNTRUSTEDCODEOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedCodeOK({
    parameters are target-dependent} ) {
    return PRO_IGNORE;
}
```

RTA-OS コンフィギュレーション

'Enhanced Isolation'ターゲットオプションが有効になっている場合は、このコールバック関数を実装する必要があります。ターゲットによってはこのオプションをサポートしていないものがあるので、注意が必要です。

参照

Os_Cbk_RestoreGlobalRegisters
Os_Cbk_IsSystemTrapAllowed
ProtectionHook
Os_Cbk_IsUntrustedTrapOK

3.15 Os_Cbk_IsUntrustedTrapOK

アントラステッドコードが原因でプロセッサトラップ/例外が発生する場合に何をするべきかを調べるために使用されるコールバック関数です。

構文

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedTrapOK(
    { parameters are target-dependent}
)
```

戻り値

`ProtectionReturnType` 型の値を返します。

説明

'Enhanced Isolation'ターゲットオプションが有効になっている場合は、アントラステッドコードが原因で発生したすべてのトラップは OS によりインターセプトされます。トラップについての詳細情報がこのコールバック関数に渡されるので、このコールバック関数では、それに対応して実行すべき処理を決定します。

戻り値の意味は以下のとおりです。

- PRO_IGNORE ― コードが通常のトラップ処理コードの実行を続けることが許されます。
- PRO_TERMINATETASKISR ― アントラステッドコードは正しくターミネートされ、システムの残りの処理は何も影響を受けずに続行されます。
- PRO_TERMINATEAPPL ― PRO_TERMINATETASKISR の場合と同じ処理を行い、さらに、当該のアントラステッドコードを所有している OS アプリケーションを構成する他のすべてのタスクや ISR もターミネートします。
- PRO_TERMINATEAPPL_RESTART ― PRO_TERMINATEAPPL の場合と同じ処理を行い、さらに適切な再起動タスクの起動も行います。

このコールバック関数は'Enhanced Isolation'スタック上で実行されます。割り込みはグローバルにディセーブルにされるので、このコールバックでは一切の OS API 関数を実行しないことになっています。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_ISUNTRUSTEDTRAPOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedTrapOK({
    parameters are target-dependent} ) {
    return PRO_IGNORE;
}
```

RTA-OSコンフィギュレーション

'Enhanced Isolation'ターゲットオプションが有効になっている場合は、このコールバック関数を実装する必要があります。ターゲットによってはこのオプションをサポートしていないものがあるので、注意が必要です。

参照

Os_Cbk_RestoreGlobalRegisters
Os_Cbk_IsSystemTrapAllowed
ProtectionHook
Os_Cbk_IsUntrustedCodeOK

3.16 Os_Cbk_Now_<CounterID>

カウンタの現在のチック値を返すコールバック関数です。

構文

```
FUNC(TickType, {memclass}) Os_Cbk_Now_<CounterID>(void)
```

戻り値

この関数は TickType 型の値を返します。

説明

このコールバック関数はハードウェアカウンタのカレント値を返すもので、マルチコアシステム内のどのコアでも呼び出すことができます。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_NOW_<COUNTERID>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(TickType, {memclass}) Os_Cbk_Now_MyCounter(void){
  return (TickType) HW_COUNTER_NOW_VALUE;
}
```

RTA-OS コンフィギュレーション

設定されているハードウェアカウンタごとに必須です。

参照

Os_Cbk_Cancel_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>

3.17 Os_Cbk_RegSetRestore_<RegisterSetID>

レジスタセット<RegisterSetID>のコンテキストを復元するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_RegSetRestore_<RegisterSetID>(
    Os_RegSetDepthType Depth
)
```

説明

このコールバック関数は、アプリケーションからの要求によってレジスタセット<RegisterSetID>のコンテキストを復元するものです。

Depth はアプリケーションが提供する保存バッファ内の位置を示し、このバッファからコンテキストを読み取ります。値の範囲はゼロから(OS_REGSET_<RegisterSetID>_SIZE - 1)までです。

<RegisterSetID>の実装はマルチコアに対応し、2 つのコアで同時にこのコールバック関数を実行することができます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_REGSETRESTORE_<REGISTERSETID>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_REGSET_FP_SIZE
static fp_context_save_area fpsave[OS_REGSET_FP_SIZE];
FUNC(void, {memclass}) Os_Cbk_RegSetRestore_FP(Os_RegSetDepthType
    Depth){
  ... = fpsave[Depth];
}
#endif /* OS_REGSET_FP_SIZE */
```

RTA-OS コンフィギュレーション

レジスタセット<RegisterSetID>が存在し、プリエンプションの際にそのコンテキストを復元する必要が生じる可能性がある場合は、このコールバック関数を実装する必要があります。

参照

OS_REGSET_<RegisterSetID>_SIZE
Os_Cbk_RegSetSave_<RegisterSetID>

3.18 Os_Cbk_RegSetSave_<RegisterSetID>

レジスタセット<RegisterSetID>のコンテキストを保存するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_RegSetSave_<RegisterSetID>(
    Os_RegSetDepthType Depth
)
```

説明

このコールバック関数は、アプリケーションからの要求によってレジスタセット<RegisterSetID>のカレントコンテキストを保存するものです。

Depth はアプリケーションが提供する保存バッファ内の位置を示し、このバッファにコンテキストを保存します。値の範囲はゼロから(OS_REGSET_<RegisterSetID>_SIZE - 1)までです。

<RegisterSetID>の実装はマルチコアに対応し、2 つのコアで同時にこのコールバック関数を実行することができます。このコールバック関数は複数のコアのコンテキストをそれぞれ異なる場所に保存しますが、レジスタセットはコア間で共有されるコンテキストを保護することはできません。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_REGSETSAVE_<REGISTERSETID>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_REGSET_FP_SIZE
static fp_context_save_area fpsave[OS_REGSET_FP_SIZE];
FUNC(void, {memclass}) Os_Cbk_RegSetSave_FP(Os_RegSetDepthType
    Depth){
  fpsave[Depth] = ...;
}
#endif /* OS_REGSET_FP_SIZE */
```

RTA-OS コンフィギュレーション

レジスタセット<RegisterSetID>が存在し、プリエンプションの際にそのコンテキストを保存する必要が生じる可能性がある場合は、このコールバック関数を実装する必要があります。

参照

OS_REGSET_<RegisterSetID>_SIZE
Os_Cbk_RegSetRestore_<RegisterSetID>

3.19 Os_Cbk_RestoreGlobalRegisters

すべてのシステムグローバルレジスタに正しい値を確実に格納するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_RestoreGlobalRegisters(void)
```

戻り値

boolean 型の値を返します。

説明

'Enhanced Isolation'ターゲットオプションが有効になっていると、OS は、アントラステッドコードが何らかの形でそのレジスタやアクセス可能なすべてのシステムコンテキストを破壊する可能性があると想定します。

このコールバック関数は、OS 内の、アントラステッドコードがターミネートするかプリエンプトされるすべてのポイントにチェック／リカバリコードを挿入するためのものです。このコードは専用の安全なスタックを使用し、すべての割り込みがディセーブルになっている状態で実行されます。

このコールバック関数はチェック中の非常に早い時期に Os_Cbk_RestoreGlobalRegisters() を呼び出して、システムグローバルレジスタやその他のシステムコンテキストを復元します。これは一般的に、スモールデータ領域用ベースレジスタとして使用されるすべてのレジスタが復元されることを意味します。

このコールバック関数は'Enhanced Isolation'スタック上で実行されます。生成されるプロセッサ命令がシステムグローバルレジスタを復元前に使用しない場合は、このコールバック関数を C で実装することができます。割り込みはグローバルにディセーブルされるので、このコールバック関数から OS API 関数を呼び出すことはできません。

このコールバック関数によって値をリセットすることにより、値が壊れなかったかどうかを調べる必要がなくなります。

注記： 一部のターゲットでは、グローバルレジスタのうちのいずれかが Enhanced Isolation コードによってスタックスワッピングコードの一部として使用されるため、意図的に壊される場合があります。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_RESTOREGLOBALREGISTERS_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_RestoreGlobalRegisters(void) {
  ASM("mov A0, _sda_RAM_base");
  ASM("mov A1, _sda_ROM_base");
}
```

RTA-OSコンフィギュレーション

'Enhanced Isolation'ターゲットオプションが有効になっている場合は、このコールバック関数を実装する必要があります。ターゲットによってはこのオプションをサポートしていないものがあるので、注意が必要です。

参照

Os_Cbk_IsSystemTrapAllowed
Os_Cbk_IsUntrustedCodeOK
Os_Cbk_IsUntrustedTrapOK

3.20 Os_Cbk_SetMemoryAccess

トラステッドモードからアントラステッドモードへの切り替えのためのメモリ保護システムを準備するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_SetMemoryAccess(
    Os_UntrustedContextRefType ApplicationContext
)
```

引数

引数	モード	説明
ApplicationContext	in	Os_UntrustedContextRefType アントラステッドコンテキストが記述されている Os_UntrustedContextType への参照

説明

このコールバック関数の機能は、デバイス上のメモリ保護ハードウェアをユーザーが完全に制御できるようにすることと、プロジェクトに適用する保護のレベルをユーザーが決定できるようにすることです。たとえば、アントラステッドコードについては書き込みアクセスを制限し、読み取りと実行のアクセスはすべて許可する、という選択が可能です。あるいは、アントラステッドコードについて読み取り／書き込み、実行のアクセスを制限することもできます。

AUTOSAR OS では、トラステッド OS アプリケーションのコンテキストで実行されるコードは、使用可能な任意の領域（RAM、ROM 、IO スペース）に対する完全なアクセス権を持つと想定され、そのようなコードは特権モードで実行されます。 一方、アントラステッド OS アプリケーションのコンテキストで実行されるコードは、所定の領域にアクセスできないように制限がかけられている場合があります。このようなコードは通常'user'モードで実行されます。

RTA-OS は、トラステッドコードからアントラステッドコードに切り替わろうとするときには必ずこの Os_Cbk_SetMemoryAccess を呼び出し、その際、アントラステッドコードについてどのような許可を設定するかを決定するためにユーザーが使用できる Os_UntrustedContextType 型のデータ構造体への参照を渡します。この Os_UntrustedContextType 型の構造体には、OS アプリケーション、タスク／ISR、実行されようとしているコードに適用されるスタック領域に関する情報が含まれています。切り替えのコンテキストによっては、情報の一部に NULL 値が含まれている可能性があります。このコールバック関数はトラステッドコードからしか呼び出されません。

このコールバック関数が呼び出されるのは以下のいずれかの場合です。

- アントラステッド OS アプリケーションに属するタスクを呼び出す前
- アントラステッド OS アプリケーションに属するカテゴリ 2 ISR を呼び出す前
- アントラステッド OS アプリケーションのスタートアップ、シャットダウン、エラーフックを呼び出す前

- アントラステッド OS アプリケーションに属する'TrustedFunction'を呼び出す前（これはAUTOSARのコンセプトを拡大するものであり、あるコアのトラステッドタスクにサードパーティ供給のアントラステッドコードの呼び出しを許可するものです）

メモリ保護機能を使用する場合は、StartOS()を呼び出す前にメモリ保護ハードウェアを初期化する必要があります。ユーザーは、使用するハードウェア、保護する領域の数、適用する制限の種類を選択することができます。

すべてのアントラステッドコードを同じメモリ保護設定で実行するには、OS オプション'Single Memory Protection Zone'を設定にします。この場合、Os_Cbk_SetMemoryAccess は呼び出されません。アントラステッドコードを実行する前に、MPU をセットアップする必要があります。

すべてのアントラステッドコードを同じ基本メモリ保護設定で実行し、スタックに保護を適用したい場合は、OS オプション'Stack Only Memory Protection' を使用します。その場合は、スタック関連のフィールド（Address と Size）と Application だけが渡されます。ユーザーは。メモリ保護設定がスタックを所定の範囲だけに制限していることを確認する必要があります。

* 注記 *

RTA-OS がサポートしている一部のターゲットプロセッサには、MPU 保護領域の設定に使用できるアドレスに関する制約（アドレスが 64 バイト境界上でなければならない、など）があります。このような状況でスタックを完全に保護するため、RTA-OS は Os_UntrustedContextType 内の'AlignedAddress'という名前の予備のフィールドを使用します。

'AlignedAddress'が存在する場合、そこには'Address'と同じ値が初期設定されます。ユーザーはその値を、MPU の適切なスタック上の次のアドレスを反映するように変更することができます。たとえば、スタック領域が 256 バイト境界から始まり、アドレスの小さくなる方へスタックされていく場合は、この値を 0x580 から 0x500 に変更するようなことが必要になります。

'AlignedAddress'をサポートするターゲットでは、ユーザーがセットアップしたメモリ保護領域内で処理が行われるように、RTA-OS は'AlignedAddress'の変化を検知して、アントラステッドコードが実行される直前にスタックが確実にこの位置に移動するようにします。

これらの調整は、宣言するすべてのスタックバジェット内で行う必要があります。通常のスタック上ではない位置にスタックを移動してしまうと、RTA-OS 内の前提条件や最適化のほとんどが無効になってしまいます。

このメカニズムは、それをサポートしていて、かつコマンドラインオプション'Enable stack repositioning'（ARXML の'AlignUntrustedStacks'）を提供している RTA-OS ターゲットでのみ使用できます。

一般に、スタック再配置が有効な状態では、ユーザーが割り当てる必要のあるスタックバジェットは、通常よりも配置粒度に等しい値の分だけ大きくなります。

* 注記 *

'FunctionID'と'FunctionParams'が存在するのは、アントラステッド関数が存在している場合だけです。このコールバック関数がアントラステッド関数について使用された場合以外は、'FunctionID'の値は INVALID_FUNCTION になります。アントラステッド関数について使用された場合は、'FunctionID'には関数の識別子が格納され、'FunctionParams'は関数の引数へのポインタのコピーです。

* 注記 *

'CoreID'は、AUTOSAR コアが複数個存在する場合だけ存在し、そこにはカレントコアの番号が格納されています。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_SETMEMORYACCESS_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass})
    Os_Cbk_SetMemoryAccess(Os_UntrustedContextRefType
    ApplicationContext) {
  /*
   * When called for an Untrusted TASK:
   *  ApplicationContext->Application contains the ID of the OS
       Application that the TASK belongs to.
   *  ApplicationContext->TaskID is the ID of the TASK
   *  ApplicationContext->ISRID is INVALID_ISR
   *  ApplicationContext->Address is the starting address for the
       TASK's stack.
   *  ApplicationContext->Size is the stack budget configured for
       the TASK. (Zero if no budget or if stack monitoring is
       disabled.)
```

```
*
* When called for an Untrusted ISR:
*  ApplicationContext->Application contains the ID of the OS
    Application that the ISR belongs to.
*  ApplicationContext->TaskID is INVALID_TASK
*  ApplicationContext->ISRID is the ID of the ISR
*  ApplicationContext->Address is the starting address for the
    ISR's stack.
*  ApplicationContext->Size is the stack budget configured for the
    ISR. (Zero if no budget or if stack monitoring is
    disabled.)
*
* When called for:
*    - an Untrusted Function
*    - an Untrusted OS Application error hook
*    - an Untrusted OS Application startup hook
*    - an Untrusted OS Application shutdown hook
*  ApplicationContext->Application contains the ID of the OS
    Application that the function/hook belongs to.
*  ApplicationContext->TaskID is INVALID_TASK
*  ApplicationContext->ISRID is INVALID_ISR
*  ApplicationContext->Address is the value of the stack
    pointer just before the untrusted code gets called.
*  ApplicationContext->Size is zero
*
* Where there are Untrusted Functions, there are two more
    fields:
*  ApplicationContext->FunctionID contains the ID of the
    function (INVALID_FUNCTION unless being called for an Untrusted
    Function)
*  ApplicationContext->FunctionParams contains FunctionParams for
    the Untrusted Function call (undefined for INVALID_FUNCTION)
*
* On systems where the hardware does not allow protection regions
    to be set at any address/size combination,
* it may be necessary to adjust the stack to a position that
    can be protected efficiently.
* For example, the protection region may have to be aligned on
    a 64-byte address boundary.
* In these cases, RTA-OS may provide the
    'AlignUntrustedStacks' configuration option.
* When this is set, a further field
    'ApplicationContext->AlignedAddress' becomes available.
* Its initial value will be the same as
    ApplicationContext->Address. However you can
* change its value to signal to the OS that the untrusted code should
    start at a different location.
```

```
 * For the earlier example, if
    ApplicationContext->AlignedAddress initially has value
    0x1020, you
 * might change it to 0x1000 before returning so that the OS
    will start running the code at an
 * address that is a multiple of 64. (This example assumes that the stack
    grows towards lower addresses.)
 * You will have set the stack protection region to start from
    0x1000.
 *
 * Be aware that on some target devices (Power PC, for example)
    the EABI might specify that a
 * back link will be written before the stack pointer on entry.
 * You will have to account for this in your calculations.
 *
 * For a multicore system, ApplicationContext->CoreID contains
    the ID of the calling core.
 *  (This is omitted if the OS is only running on one core)
 */

/*  Force AlignedAddress to the the next 64-byte value below
   Address */
(uint32)ApplicationContext->AlignedAddress &=
   ((uint32)ApplicationContext->Address % 64U);
SET_STACK_RANGE(ApplicationContext->AlignedAddress,
   STACK_ALLOWANCE);

if (ApplicationContext->Application == App2) {
  /* Set memory protection regions that apply for the overall
     application 'App2' */
  SET_UNTRUSTED_WRITE_RANGE(App2_BASE, App2_SIZE); /* 例 */
  if (ApplicationContext->TaskID == App2TaskB) {
    /* Extend or restrict ranges as desired for Task
       'App2TaskB' */
  }
  if (ApplicationContext->ISRID == App2ISR1) {
    /* Extend or restrict ranges as desired for ISR 'App2ISR1'
       */
  }
  if (ApplicationContext->FunctionID == UTF1) {
    /* Extend or restrict ranges as desired for Untrusted
       Function 'tf1' */
  }
}
if (ApplicationContext->Application == App3) {
  /* Set memory protection regions that apply for the overall
     application 'App3' */
  SET_UNTRUSTED_WRITE_RANGE(App3_BASE, App3_SIZE); /* Example */
  if (ApplicationContext->TaskID == App3TaskB) {
```

```
      /* Extend or restrict ranges as desired for Task
          'App3TaskB' */
    }
    if (ApplicationContext->FunctionID == UTF2) {
      /* Extend or restrict ranges as desired for Untrusted
          Function 'tf2' */
    }
    if (ApplicationContext->FunctionID == UTF3) {
      /* Extend or restrict ranges as desired for Untrusted
          Function 'tf3' */
    }
  }
  ...
}
OS_MAIN() {
  ...
  InitializeMemoryProtectionHardware();
  ...
  StartOS(OSDEFAULTAPPMODE);

}
```

RTA-OS コンフィギュレーション

メモリ保護が有効になっていて、さらにアントラステッド OS アプリケーションが存在している場合は、このコールバック関数を実装する必要があります。

参照

Os_Cbk_CheckMemoryAccess
Os_UntrustedContextType
CallTrustedFunction
CallAndProtectFunction

3.21 Os_Cbk_SetTimeLimit

タイミング割り込みをイネーブルにし、そのためのタイムリミットを設定するコールバック関数です。

構文

```
FUNC(void,{memclass}) Os_Cbk_SetTimeLimit(
    Os_TimeLimitType Limit
)
```

戻り値

Os_TimeLimitType 型の値を返します。

説明

タイミング保護が設定されていて、タイムリミットを実施するためにタイミング割り込みが使用されている場合は、 この API 関数を実装する必要があります。

この API 関数は、タイミング割り込みを確実にイネーブルにし、Os_Cbk_SuspendTimeLimit() によってキャンセルされない限りは現時点から'Limit'チック後にそのタイミング割り込みがイネーブルされるようにする必要があります。

Os_TimeLimitType の 1 チックはストップウォッチの 1 チックと同じ長さであると想定されます。

値ゼロで呼び出された場合は、直ちに Os_TimingFaultDetected()を呼び出して、割り込みをイネーブルにする処理をスキップすることができます。

マルチコアシステムでは、タイミングリミットを持つコアごとに 1 つのタイミング割り込みが必要です。この API 関数は、自身が呼び出されたコアのタイミング割り込みだけに影響を与えます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_SETTIMELIMIT_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void,{memclass}) Os_Cbk_SetTimeLimit(Os_TimeLimitType Limit)
    {
  Os_TimeLimitType now = <read current counter value>;
  if (Limit == 0) {
    Os_TimingFaultDetected();
  }
  <set current counter compare value>(now + Limit + 1);
}
```

RTA-OSコンフィギュレーション

タイミング保護が設定されていて、タイムリミットを実施するためにタイミング割り込みが使用されている場合には、このコールバック関数を実装する必要があります。

参照

Os_TimingFaultDetected
Os_Cbk_SuspendTimeLimit
ProtectionHook
Os_TimeLimitType

3.22 Os_Cbk_Set_<CounterID>

ハードウェアカウンタ用に次の目標値（マッチ値）をセットするコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_Set_<CounterID>(
    TickType Match
)
```

引数

引数	モード	説明
Match	in	TickType 次の絶対マッチ値

説明

このコールバック関数はハードウェアカウンタをセットアップし、新しいマッチ値に到達したら適切な割り込みが発行されるようにします。

Match は、次のカウンタアクションが処理されるべき時点を示す絶対値です。

このコールバック関数は Os_AdvanceCounter 内から呼び出され、次のアラーム、またはスケジュールテーブルの次の満了ポイントのマッチ値を設定します。

また、アラームまたはスケジュールテーブルを起動した結果としてこのコールバック関数を呼び出すこともできます。

このコールバック関数はカウンタを所有するコア上でだけ呼び出されます。また、リエントラントにする必要はありません。

以下の点に注意してください。

- インターバルが短い場合、このコールバック関数が呼び出された時点でハードウェアカウントがすでにマッチ値を過ぎてしまっている可能性があります。 その場合は、割り込みペンディングビットをソフトウェア内で確実にセットすることが重要です。
- 設定されているインバーバルよりも短いインターバルでアラームまたはスケジュールテーブルを起動できるようにするには、そのコードは、マッチ値を小さくできること、さらに、そのポイントをハードウェアカウントがすでに通過してしまっていないかを検知できることが必要です。

一般的に、このコールバック関数はカウンタハードウェアを初期化しません。初期化は、OS が起動される前の初期化コード内で行われるのが一般的です。

注記：下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_SET_<COUNTERID>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_Set_MyCounter(TickType Match){
  /* Prevent match interrupts for maxallowedvalue+1 ticks*/
  HW_OUTPUT_COMPARE_VALUE = COUNTER - 1u;
  dismiss_interrupt();
  HW_OUTPUT_COMPARE = Match;
  enable_interrupt();
}
```

RTA-OS コンフィギュレーション

設定されているハードウェアカウンタごとに必須です。

参照

Os_AdvanceCounter
SetAbsAlarm
SetRelAlarm
Os_Cbk_Cancel_<CounterID>
Os_Cbk_Now_<CounterID>
Os_Cbk_State_<CounterID>

3.23 Os_Cbk_StackOverrunHook

スタック関連のエラーをトラップするコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_StackOverrunHook(
    Os_StackSizeType Overrun,
    Os_StackOverrunType Reason
)
```

引数

引数	モード	説明
Overrun	in	Os_StackSizeType オーバーランの量
Reason	in	Os_StackOverrunType オーバーランの原因

説明

このコールバック関数は、以下のいずれかのイベントが発生した際に呼び出されるフックルーチンです。

- タスク／ISR のスタックアロケーションバジェットが定義されていて、 このバジェットを超えた場合
- スタック上にスペースがなかったために ECC タスクが起動できなかった場合
- スタック上にスペースがなかったために ECC タスクがウェイト状態からレジュームできなかった場合
- スタックを多く使いすぎて、タスクの状態を安全に保存できなかったために、ECC タスクがウェイト状態に入れなかった場合

GetTaskID()と GetISRID()を呼び出すことにより、どのタスク／ISR が関連しているかが分かります。

ProtectionHook() が設定されている場合は、E_OS_STACKFAULT 状態で ProtectionHook()（ProtectionHook() が設定されていない場合は ShutdownOS()）を呼び出すカーネル内にこのフックのデフォルトバージョンが含まれていますが、ユーザーがアプリケーション内に Os_Cbk_StackOverrunHook を実装することにより、この挙動をオーバーライドすることができます。

バジェットオーバーランはプリエンプションポイントにおいて（または Os_GetStackUsage() が呼び出される際に）検出されます。ただし、検出されたバジェットオーバーランが報告されるのは、各実行の中で初めて検出されたときだけです。

バジェットオーバーランが発生しても、タスク／ISR が強制的にターミネートされることはありません。このフック内で TerminateTask を呼び出すことは許されません。

ECC関連のオーバーランは、優先度の低いタスクが自身のスタックバジェットを超えてしまった場合や、スタックのプリエンプションオーバーヘッドとして設定されている値が小さすぎる場合に発生します。

ECCオーバーランが発生すると、タスクが強制的にターミネートさせられることになります。

OS_BUDGETは、スタック監視が設定されている場合に限り発生する可能性があります。

OS_ECC_START、OS_ECC_RESUME、OS_ECC_WAITは、スタック監視が設定されていてもいなくても発生する可能性があります。

注記: 下記のコード内のmemclassは、AUTOSAR 3.xの場合はOS_APPL_CODE、AUTOSAR 4.0の場合はOS_CALLOUT_CODE、AUTOSAR 4.1の場合はOS_OS_CBK_STACKOVERRUNHOOK_CODEです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_StackOverrunHook(Os_StackSizeType
    Overrun, Os_StackOverrunType Reason) {
  switch (Reason) {
    case OS_BUDGET:
      /* The currently running task or ISR has exceeded its stack budget
          */
      break;
    case OS_ECC_START:
      /* An ECC task has failed to start because there is
          insufficient room on the stack */
      break;
    case OS_ECC_RESUME:
      /* An ECC task has failed to resume from wait because there
          is insufficient room on the stack */
      break;
    case OS_ECC_WAIT:
      /* An ECC task has failed to enter the waiting state
          because it is exceeding its waiting stack budget */
      break;
  }
}
```

RTA-OSコンフィギュレーション

スタック監視が設定されていてバジェットが割り当てられている場合、またはECCタスクが存在する場合は、必要に応じて実装できます。

参照

Os_GetStackUsage
Os_GetISRMaxStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage
GetISRID
GetTaskID

3.24 Os_Cbk_State_<CounterID>

ハードウェアカウンタの現在の状態を読み取るコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_State_<CounterID>(
    Os_CounterStatusRefType State
)
```

引数

引数	モード	説明
State	out	Os_CounterStatusRefType カウンタの状態

説明

OS は、次のマッチが到来する時を判断する際に Os_AdvanceCounter 内からこのコールバック関数を呼び出します。また、API 関数 GetAlarm と GetScheduleTableStatus からも呼び出される可能性があります。

このコールバック関数は State 構造体を更新し、カウンタが実行されているかどうか、マッチ割り込みがペンディングされているかどうか、さらに次のマッチまでのインターバルの長さをセットする必要があります。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_STATE_<COUNTERID>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass})
    Os_Cbk_State_MyCounter(Os_CounterStatusRefType State) {
  State.Delay = HW_OUTPUT_COMPARE_VALUE - HW_COUNTER_NOW_VALUE;
  State.Pending = counter_interrupt_pending();
  State.Running = counter_interrupt_enabled();
}
```

RTA-OS コンフィギュレーション

設定されているハードウェアカウンタごとに必須です。

参照

Os_Cbk_Cancel_<CounterID>
Os_Cbk_Now_<CounterID>
Os_Cbk_Set_<CounterID>
Os_AdvanceCounter

3.25 Os_Cbk_SuspendTimeLimit

タイミング割り込みをキャンセルして、残り時間を調べるコールバック関数です。

構文

```
FUNC(Os_TimeLimitType,{memclass}) Os_Cbk_SuspendTimeLimit(void)
```

戻り値

Os_TimeLimitType 型の値を返します。

説明

タイミング保護が設定され、タイミング割り込みを使用してタイムリミットを適用する場合は、このコールバック関数を実装する必要があります。

OS はこのコールバック関数を呼び出すことにより、前回の Os_Cbk_SetTimeLimit()の呼び出しをキャンセルします。これにより、タイムリミットに到達してもタイミング割り込みが発行されないことをユーザーが確認することができ、現在ペンディング状態である場合は、そのペンディング状態が確実にクリアされるようにすることができます。

戻り値は、この呼び出しが行われた時点からリミットまでの残りのチック数です。

マルチコアシステムでは、タイミングリミットを持つコアごとに 1 つのタイミング割り込みが必要です。このコールバック関数は、自身が呼び出されたコアのタイミング割り込みだけに影響を与えます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_SUSPENDTIMELIMIT_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(Os_TimeLimitType,{memclass}) Os_Cbk_SuspendTimeLimit(void) {
  Os_TimeLimitType now = <read current counter value>;
  <disable timing interrupt>;
  <clear timing interrupt pending flag>;
  return now - <read current counter compare value>;
}
```

RTA-OS コンフィギュレーション

タイミング保護が設定されていて、タイムリミットを実施するためにタイミング割り込みが使用されている場合には、このコールバック関数を実装する必要があります。

参照

Os_TimingFaultDetected
Os_Cbk_SetTimeLimit
ProtectionHook
Os_TimeLimitType

3.26 Os_Cbk_TaskActivated

タスクの起動を示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TaskActivated(
    TaskType task
)
```

説明

OS オプション'Task Activation Hook'が有効になっている場合は、このコールバック関数を実装する必要があります。このコールバック関数は、タスクの起動が成功した時点で呼び出されます。

起動は ActivateTask、ChainTask、アラーム、スケジュールテーブルのいずれかによって行われます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TASKACTIVATED_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 task_activations[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskActivated(TaskType task) {
  task_activations[OS_TASKTYPE_TO_INDEX(task)] += 1;
}
```

RTA-OS コンフィギュレーション

OS オプション'Task Activation Hook'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

PreTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskEnd
Os_Cbk_TaskStart
Os_Cbk_ISREnd
Os_Cbk_ISRStart

3.27 Os_Cbk_TaskEnd

タスクの終了を示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TaskEnd(
    TaskType task
)
```

説明

OS オプション'Additional Task Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。このコールバック関数は、タスクが実行を終了する時点で呼び出されます。

ECC タスクの場合、このコールバック関数はウェイト状態に入るときにも呼び出されます。

PostTaskHook とは異なり、タスクがプリエンプトされているときには呼び出されません。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TASKEND_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType task_exe_time[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskEnd(TaskType task) {
  task_exe_time[OS_TASKTYPE_TO_INDEX(task)] = GetExecutionTime();
}
```

RTA-OS コンフィギュレーション

OS オプション'Additional Task Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

PostTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskStart
Os_Cbk_ISREnd
Os_Cbk_ISRStart

3.28 Os_Cbk_TaskStart

タスクの開始を示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TaskStart(
    TaskType task
)
```

説明

OS オプション'Additional Task Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。このコールバック関数は、タスクが実行開始される直前に呼び出されます。

ECC タスクの場合、このコールバック関数はウェイト状態からレジュームするときにも呼び出されます。

PreTaskHook とは異なり、タスクがプリエンプトされているときには呼び出されません。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TASKSTART_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 task_starts[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskStart(TaskType task) {
  task_starts[OS_TASKTYPE_TO_INDEX(task)] += 1;
}
```

RTA-OS コンフィギュレーション

OS オプション'Additional Task Hooks'が有効になっている場合は、このコールバック関数を実装する必要があります。

参照

PreTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskActivated
Os_Cbk_TaskEnd
Os_Cbk_ISREnd
Os_Cbk_ISRStart

3.29 Os_Cbk_Terminated_<ISRName>

カテゴリ 2 ISR <ISRName>が強制的にターミネートされたことを示すコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_Terminated_<ISRName>(void)
```

説明

このコールバック関数は、カテゴリ 2 ISR が OS により強制的にターミネートされる際に、アプリケーションが適切な処理を実行できるようにするためのものです。

ISR は、以下のいずれかの状況でターミネートされる可能性があります。

- その ISR が実行状態のとき（その ISR が自分より優先度の高い割り込みにより中断されてしまっているときを含む）にユーザーが TerminateApplication()を呼び出す場合
- その ISR が実行状態のときにタイミングまたはメモリ保護違反があり、ユーザーが ProtectionHook()から PRO_TERMINATETASKISR を返す場合
- プリエンプトする ISR が実行状態のときにタイミングまたはメモリ保護違反があり、ユーザーが ProtectionHook()から PRO_TERMINATEAPPL または PRO_TERMINATEAPPL_RESTART を返す場合

ユーザーが割り込みソースの一部の「割り込みペンディング」ステータスをクリアしなければならないターゲットプロセッサについては、このコールバック関数を使用してそのステータスをクリアする必要があります。これを行わないと、後でプロセッサ優先度が下がったときに割り込みが発生してしまいます。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TERMINATED_<ISRNAME>_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_Terminated_App2Isr1(void) {
    clear_interrupt_source(_App2Isr1_);
}
```

RTA-OS コンフィギュレーション

割り込みの強制ターミネーションがサポートされている場合は、カテゴリ 2 ISR ごとに必須です。

参照

ProtectionHook
TerminateApplication

3.30 Os_Cbk_TimeOverrunHook

時間監視中に検出されたエラーをトラップするコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TimeOverrunHook(
    Os_StopwatchTickType Overrun
)
```

引数

引数	モード	説明
Overrun	in	Os_StopwatchTickType オーバーランの量を示すストップウォッチチック数

説明

このコールバック関数は、タスク／ISR の実行バジェットが定義されている場合、実行時間がこのバジェットを超えた際に呼び出されるのフックルーチンです。

バジェットオーバーランはプリエンプションポイントで、タスク／ISR がターミネートしたときに検出されます。このフックはオーバーランが初めて検出されたときにだけ呼び出されます。

バジェットオーバーランが発生しても、タスク／ISR が強制的にターミネートされることはありません。このフック内で TerminateTask を呼び出すことは許されません。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TIMEOVERRUNHOOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass}) Os_Cbk_TimeOverrunHook(Os_StopwatchTickType
    Overrun) {
}
```

RTA-OS コンフィギュレーション

時間監視が設定されていてバジェットが割り当てられている場合は、必須です。

参照

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime
GetISRID
GetTaskID

3.31 PostTaskHook

タスクからのコンテキスト切り替えのときに呼び出されるコールバック関数です。

構文

```
FUNC(void, {memclass}) PostTaskHook(void)
```

説明

このコールバック関数は、オペレーティングシステムが実行状態でなくなる直前にオペレーティングシステムから呼び出されるフックルーチンです。

つまり TaskID を評価しても問題は起きません。

ShutdownOS()の呼び出しが原因でタスクが実行状態でなくなる場合は、このフックルーチンは呼び出されません。

サンプルの PostTaskHook を rtaosgen で自動的に生成することができます。詳細は『RTA-OS ユーザーズガイド』を参照してください。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_POSTTASKHOOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) PostTaskHook(void){
  TaskType LeavingTask; GetTaskID(&LeavingTask);
  if (LeavingTask == TaskA) {
    /* Do action for leaving A */
  } else if (LeavingTask == TaskB) {
    /* Do action for leaving B */
  }
  ...
}
```

RTA-OS コンフィギュレーション

PostTaskHook が設定されている場合は必須です。

参照

PreTaskHook
Os_Cbk_TaskEnd
Os_Cbk_ISREnd

3.32 PreTaskHook

タスクへのコンテキスト切り替えのときに呼び出されるコールバック関数です。

構文

```
FUNC(void, {memclass}) PreTaskHook(void)
```

説明

このコールバック関数は、オペレーティングシステムが実行状態に入った直後、タスクが実行を開始する前に、オペレーティングシステムから呼び出されるフックルーチンですす。

つまり TaskID を評価しても問題は起きません。

サンプルの PreTaskHook を rtaosgen で自動的に生成することができます。詳細は『RTA-OS ユーザーズガイド』を参照してください。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_PRETASKHOOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) PreTaskHook(void){
  TaskType EnteringTask;
  GetTaskID(&EnteringTask);
  if (EnteringTask == TaskA) {
    /* Do action for entering A */
  } else if (EnteringTask == TaskB) {
    /* Do action for entering B */
  }
  ...
}
```

RTA-OS コンフィギュレーション

PreTaskHook が設定されている場合は必須です。

参照

PostTaskHook
Os_Cbk_TaskStart
Os_Cbk_TaskActivated
Os_Cbk_ISRStart

3.33 ProtectionHook

保護障害をトラップするためのコールバック関数です。

構文

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(
    StatusType FatalError
)
```

引数

引数	モード	説明
FatalError	in	StatusType 発生したエラーのタイプ

戻り値

ProtectionReturnType 型の値を返します。

説明

タイミングまたはメモリ保護障害が発生すると、障害のタイプを引数としてこのコールバック関数が呼び出されます。

OS は、このコールバック関数の戻り値に応じて以下のような処理を行います。

- PRO_IGNORE: 障害を無視して処理を続行します。これは E_OS_PROTECTION_ARRIVAL の場合だけに許されます。
- PRO_TERMINATETASKISR: この障害の原因となったタスク／ISR、または保護されている関数を強制的にターミネートします。メモリまたはタイミング保護が設定されている場合に限り有効です。
- PRO_TERMINATEAPPL: 障害を起こしたタスク／ISR が含まれる OS アプリケーションを強制的にターミネートします。メモリまたはタイミング保護が設定されている場合に限り有効です。
- PRO_TERMINATEAPPL_RESTART: 障害を起こしたタスク／ISR が含まれる OS アプリケーションを強制的にターミネートして再起動します。メモリまたはタイミング保護が設定されている場合に限り有効です。
- PRO_SHUTDOWN: ShutdownOS()を呼び出します。

カテゴリ 2 ISR がターミネートされた場合は、OS はコールバック関数 Os_Cbk_Terminated_<ISRName>()を使用して、割り込みソースが適切に処理されたことをユーザーが確認できるようにします。

ProtectionHook は OS レベルで実行され、タスクやカテゴリ 2 ISR によってプリエンプトされることはありません。

サンプルのProtectionHookをrtaosgenで自動的に生成することができます。詳細は『RTA-OSユーザーズガイド』を参照してください。

注記： 下記のコード内のmemclassは、AUTOSAR 3.xの場合はOS_APPL_CODE、AUTOSAR 4.0の場合はOS_CALLOUT_CODE、AUTOSAR 4.1の場合はOS_PROTECTIONHOOK_CODEです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(StatusType
    FatalError) {
  switch (FatalError) {
    case E_OS_PROTECTION_MEMORY:
          /* A memory protection error has been detected */
          break;
    case E_OS_PROTECTION_TIME:
          /* Task, Category 2 ISR or time-limited function
              exceeds its execution time */
          break;
    case E_OS_PROTECTION_ARRIVAL:
          /* Task/Category 2 arrives before its timeframe has
              expired */
          return PRO_IGNORE; /* This is the only case where
              PRO_IGNORE is allowed */
    case E_OS_PROTECTION_LOCKED:
          /* Task/Category 2 ISR blocks for too long */
          break;
    case E_OS_PROTECTION_EXCEPTION:
          /* Trap occurred */
          break;
  }
  return PRO_SHUTDOWN;
}
```

RTA-OS コンフィギュレーション

ProtectionHookが設定されている場合は必須です。タイミングやメモリの保護が必要な場合は適宜設定してください。

参照

Os_Cbk_Terminated_<ISRName>

3.34 ShutdownHook

OS のシャットダウン処理中に呼び出されるコールバック関数です。

構文

```
FUNC(void, {memclass}) ShutdownHook(
    StatusType Error
)
```

引数

引数	モード	説明
Error	in	StatusType シャットダウンの理由

説明

このコールバック関数は、OS API 関数 ShutdownOS()から呼び出されるフックルーチンです。

このフックルーチンはオペレーティングシステムのシャットダウン処理中に呼び出されます。OS を ShutdownHook()から再起動するには、Os_Restart()を使用します。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_SHUTDOWNHOOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) ShutdownHook(StatusType Error){
  if (Error == E_OS_STACKFAULT) {
    /* Attempt recovery by restart */
    Os_Restart();
    /* Never reach here... */
  } else if (Error == E_OK) {
    /* Normal shutdown procedure */
  }
  ...
}
```

RTA-OS コンフィギュレーション

ShutdownHook が設定されている場合は必須です。

参照

Os_Restart
StartupHook

3.35 StartupHook

OS の起動処理中に呼び出されるコールバック関数です。

構文

```
FUNC(void, {memclass}) StartupHook(void)
```

説明

このコールバック関数は、OS 初期化の終わりの部分で、スケジューラが実行される前に OS から呼び出されるフックルーチンです。

アプリケーションは、このフックルーチンの中でタスクの開始やデバイスドライバの初期化などを行うことができます。

このフックルーチンはカテゴリ 2 ISR がディセーブルになっている状態で実行されるので、このフックから割り込みソースをイネーブルにしても問題は起きません。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_STARTUPHOOK_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) StartupHook(void){
  /* Enable timer interrupt */
  CHANNEL0_CONTROL_REG |= ONE_MILLISECOND_TIMER; CHANNEL0_CONTROL_REG
  |= ENABLE;
}
```

RTA-OS コンフィギュレーション

StartupHook が設定されている場合は必須です。

参照

ShutdownHook

4 RTA-OS の型

4.1 AccessType

特定のメモリ領域に対して可能なアクセスの種類についての情報を保持する整数値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

定数

```
OS_ACCESS_READ
OS_ACCESS_WRITE
OS_ACCESS_EXECUTE
OS_ACCESS_STACK
```

例

```
FUNC(AccessType,{memclass}) Os_Cbk_CheckMemoryAccess(ApplicationType
    Application, TaskType TaskID, ISRType ISRID,
    MemoryStartAddressType Address, MemorySizeType Size) {
  AccessType Access = OS_ACCESS_EXECUTE;
  /* Address range is read/write if it is in RAM */
  if ((Address >= RAM_BASE) && (Address + Size < RAM_BASE +
      RAM_SIZE) ) {
      Access |= (OS_ACCESS_WRITE | OS_ACCESS_READ);
  }
  ...
  return Access;
}
```

4.2 AlarmBaseRefType

AlarmBaseType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
AlarmBaseType AlarmBase;
AlarmBaseRefType AlarmBaseRef = &AlarmBase;
```

4.3 AlarmBaseType

カウンタのコンフィギュレーションを定義します。この型はC構造体で、maxallowedvalue、ticksperbase、mincycle というフィールドで構成されています。

- maxallowedvalue は、最大許容カウント値（チック数）です。
- ticksperbase は、カウンタ固有の単位に到達するために必要なチック数です。
- mincycle は、SetRelAlarm と SetAbsAlarm の cycleParameters の最小許容値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

すべての値は TickType 型です。

例

```
TickType            max,min,ticks;
AlarmBaseType       SomeAlarmBase;
AlarmBaseRefType    PointerToSomeAlarmBase = &SomeAlarmBase;
max     = SomeAlarmBase.maxallowedvalue;
ticks   = SomeAlarmBase.ticksperbase;
min     = SomeAlarmBase.mincycle;
```

4.4 AlarmType

アラームの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
AlarmType SomeAlarm;
```

4.5 AppModeType

アプリケーションモードの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

コンフィギュレーション設定時に宣言されるアプリケーションモードのシンボリック名です。必ず OSDEFAULTAPPMODE というモードが宣言されている必要があります。

例

```
AppModeType SomeAppMode;
```

4.6 ApplicationStateRefType

ApplicationStateType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

例

```
ApplicationStateRefType SomeState = &state_variable;
GetApplicationState(MyApp, SomeState);
```

4.7 ApplicationStateType

OS アプリケーションの状態を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

値

APPLICATION_ACCESSIBLE
APPLICATION_RESTARTING
APPLICATION_TERMINATED

例

```
GetApplicationState(MyApp, &MyAppState);
if (MyAppState == APPLICATION_RESTARTING) {...}
```

4.8 ApplicationType

OS アプリケーションの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

コンフィギュレーション設定時に宣言される OS アプリケーションのシンボリック名です。

定数

```
INVALID_OSAPPLICATION
```

例

```
ApplicationType SomeOSApplication;
```

4.9 CoreIdType

プロセッサコアの ID を表すスカラ値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

例

```
CoreIdType core = GetCoreID();
```

4.10 CounterType

カウンタの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
CounterType SomeCounter;
```

4.11 EventMaskRefType

EventMaskType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
EventMaskRefType SomeEventRef;
```

4.12 EventMaskType

イベントの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

コンフィギュレーション設定時に宣言されるイベントマスクのシンボリック名です。

例

```
EventMaskType SomeEvent;
```

4.13 ISRRefType

ISRType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ISRType SomeISR;
ISRRefType PointerToSomeISR = &SomeISR;
```

4.14 ISRType

ISR の型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

コンフィギュレーション設定時に宣言される ISR のシンボリック名です。

定数

```
INVALID_ISR
```

例

```
ISRType SomeISR;
```

4.15 MemorySizeType

メモリ領域のサイズ（バイト数）を保持するデータ型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
MemorySizeType DatumSize = sizeof(datum);
CheckISRMemoryAccess(SomeISR, &datum, DatumSize);
```

4.16 MemoryStartAddressType

アドレス空間内の任意の位置を指し示すことのできるポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
MemoryStartAddressType StartAddress = &datum;
CheckISRMemoryAccess(SomeISR, StartAddress, sizeof(datum));
```

4.17 OSServiceIdType

OS API 関数の型です。ErrorHook()内でのみ使用されます。値は OSServiceId__APICallName_という形を取ります。ここで_APICallName_は API 関数の名前（先行する Os_は省略します）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  OSServiceIdType ServiceExecuting;
  ServiceExecuting = OSError_GetServiceID();
  switch ( ServiceExecuting ) {
    case OSServiceId_None: /* Used for errors detected when an
        ISR exits with resources or interrupts locked */
      ...
      break;
    case OSServiceId_ActivateTask:
      ...
      break;
    case OSServiceId_CancelAlarm:
      ...
      break;
    case OSServiceId_ChainTask:
      ...
      break;
    ...
    default:
      ...
  }
}
```

4.18 ObjectAccessType

OS アプリケーションがオブジェクトにアクセスできるかどうかを定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

ACCESS
NO_ACCESS

例

```
if (ACCESS == CheckObjectAccess(MyOSApp, OBJECT_TASK, MyTask)
   {...}
```

4.19 ObjectTypeType

OS オブジェクトのタイプを定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

OBJECT_TASK
OBJECT_ISR
OBJECT_ALARM
OBJECT_RESOURCE
OBJECT_COUNTER
OBJECT_SCHEDULETABLE

例

```
if (ACCESS == CheckObjectAccess(MyOSApp, OBJECT_TASK, MyTask)
   {...}
```

4.20 Os_AnyType

OS オブジェクトへの参照です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
CheckObjectOwnership(OBJECT_TASK, Task1);
```

4.21 Os_CounterStatusRefType

Os_CounterStatusType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_CounterStatusType MyHwCounterStatus;
do {
  Os_AdvanceCounter_MyHWCounter();
  Os_Cbk_State_MyHWCounter(&MyHwCounterStatus);
} while (MyHwCounterStatus.Running && MyHwCounterStatus.Pending);
```

4.22 Os_CounterStatusType

ハードウェアカウンタの状態を定義します。この型は Running、Pending、Delay というフィールドで構成される C の構造体（struct）です。

Running が TRUE になるのは、カウンタドライバが実行状態である場合だけです。

Pending が TRUE になるのは、関連付けられているアラームの満了またはスケジュールテーブルの満了ポイント、またはその両方がペンディング状態である場合だけです。

Delay は、前回の満了から次の満了までの相対チック数です。Os_CounterStatusType.Delay の値がゼロの場合は、カウンタの maxallowedvalue+1（モジュラス）を表します。

Delay フィールドは、Running と Pending がどちらも TRUE である場合に限り有効です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_CounterStatusType CounterStatus;
```

4.23 Os_SpinlockInfo

GetSpinlockInfo API 関数から返される可能性のあるスピンロック統計情報が格納されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

値

CoreIdType CurrentLockingCore
Os_StopwatchTickType CurrentLockTime
Os_LockerRefType CurrentLocker
uint32 LockAttempts[OS_NUM_CORES]
uint32 LockSucceeds[OS_NUM_CORES]
uint32 LockFails[OS_NUM_CORES]
Os_StopwatchTickType MaxLockTime[OS_NUM_CORES]
Os_LockerRefType MaxLockTimeLocker[OS_NUM_CORES]
Os_StopwatchTickType MaxSpinTime[OS_NUM_CORES]
Os_LockerRefType MaxSpinTimeLocker[OS_NUM_CORES]

例

```
Os_SpinlockInfo Info;
```

4.24 Os_SpinlockInfoRefType

Os_LockerRefType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.CurrentLocker == MyTask) {
    ...
  }
}
```

4.25 Os_StackOverrunType

スタックオーバーランの理由を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

値

OS_BUDGET
OS_ECC_START
OS_ECC_RESUME
OS_ECC_WAIT

例

```
FUNC(void, {memclass}) Os_Cbk_StackOverrunHook(Os_StackSizeType
    Overrun, Os_StackOverrunType Reason) {
  switch (Reason) {
    case OS_BUDGET:
      /* The currently running task or ISR has exceeded its stack budget
         */
      break;
    case OS_ECC_START:
      /* An ECC task has failed to start because there is
         insufficient room on the stack */
      break;
    case OS_ECC_RESUME:
      /* An ECC task has failed to resume from wait because there
         is insufficient room on the stack */
      break;
    case OS_ECC_WAIT:
      /* An ECC task has failed to enter the waiting state
         because it is exceeding its wait-stack budget */
      break;
  }
}
```

4.26 Os_StackSizeType

スタックの量（バイト数）を表す、符号なしの値です。

可搬性

RTA-OS　OSEK　R3.x　R4.x　MultiCore　RTA-TRACE

例

```
Os_StackSizeType stack_size;
stack_size = Os_GetStackSize(start_position, end_position);
```

4.27 Os_StackValueType

スタックポインタの位置を表す、符号なしの値（ESP）です。

可搬性

RTA-OS　OSEK　R3.x　R4.x　MultiCore　RTA-TRACE

例

```
Os_StackValueType start_position;
start_position = Os_GetStackValue();
```

4.28 Os_StatusRefType

StatusType へのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
StatusType status;
Os_StatusRefType status_ref = &status;
...
StartCore(OS_CORE_ID_0, status_ref);
```

4.29 Os_StopwatchTickRefType

Os_StopwatchTickType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickRefType SomeTick = &tickvalue;
Os_GetTaskActivationTime(MyTask, SomeTick);
```

4.30 Os_StopwatchTickType

ストップウォッチ（時間監視または保護）カウンタのチックを表すスカラ値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_StopwatchTickType Duration;
GetExecutionTime(&Duration);
```

4.31 Os_TasksetType

この型は、タスクの実行を遅延させることが設定されている場合に限り使用できます。同一コアに常駐する一連のタスクを格納するためのものです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

定数

OS_NO_TASKS

例

```
Os_TasksetType t1 = TASK_MASK(Task1);
```

4.32 Os_TimeLimitType

タイミング保護と一緒に使用される、実行のタイムリミットを表すスカラ値です。Os_TimeLimitType の長さは Os_StopwatchTickType と同じです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_TimeLimitType limit = 100;
CallAndProtectFunction(Func3, &data, limit);
```

4.33 Os_UntrustedContextRefType

Os_UntrustedContextType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(void, {memclass})
    Os_Cbk_SetMemoryAccess(Os_UntrustedContextRefType
    ApplicationContext) {}
```

4.34 Os_UntrustedContextType

実行されようとしているアントラステッドコードのコンテキストを定義します。メモリ保護機能が設定されている場合にコールバック関数 Os_Cbk_SetMemoryAccess()によってのみ使用されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

値

ApplicationType Application
TaskType TaskID
ISRType ISRID
MemoryStartAddressType Address
MemorySizeType Size

例

```
FUNC(void, {memclass})
    Os_Cbk_SetMemoryAccess(Os_UntrustedContextRefType
    ApplicationContext) {}
```

4.35 PhysicalTimeType

物理的（実測）時間の単位を表すスカラ値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
PhysicalTimeType Milliseconds = OS_TICKS2MS_MyCounter(42);
```

4.36 ProtectionReturnType

保護障害の後に実行される処理を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

PRO_IGNORE
PRO_TERMINATETASKISR
PRO_TERMINATEAPPL
PRO_TERMINATEAPPL_RESTART
PRO_SHUTDOWN

例

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(StatusType
    FatalError) {
  switch (FatalError) {
    case E_OS_PROTECTION_MEMORY:
          /* A memory protection error has been detected */
          break;
    case E_OS_PROTECTION_TIME:
          /* Task, Category 2 ISR or time-limited function
```

```
            exceeds its execution time */
        break;
    case E_OS_PROTECTION_ARRIVAL:
        /* Task/Category 2 arrives before its timeframe has
            expired */
        return PRO_IGNORE; /* This is the only case where
            PRO_IGNORE is allowed */
    case E_OS_PROTECTION_LOCKED:
        /* Task/Category 2 ISR blocks for too long */
        break;
    case E_OS_PROTECTION_EXCEPTION:
        /* Trap occurred */
        break;
  }
  return PRO_SHUTDOWN;
}
```

4.37 ResourceType

リソースの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

RES_SCHEDULER と、コンフィギュレーション設定時に宣言されたリソースのシンボリック名です。

定数

```
RES_SCHEDULER
```

例

```
ResourceType SomeResource;
```

4.38 RestartType

TerminateApplication()内で実行される処理を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

RESTART
NO_RESTART

4.39 ScheduleTableRefType

ScheduleTableType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ScheduleTableType SomeScheduleTable;
ScheduleTableRefType PointerToSomeScheduleTable =
    &SomeScheduleTable;
```

4.40 ScheduleTableStatusRefType

ScheduleTableStatusType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ScheduleTableStatusType SomeScheduleTableStatus;
GetScheduleTableStatus(&SomeScheduleTableStatus);
```

4.41 ScheduleTableStatusType

スケジュールテーブルのランタイムの状態を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

SCHEDULETABLE_STOPPED
SCHEDULETABLE_NEXT
SCHEDULETABLE_WAITING
SCHEDULETABLE_RUNNING
SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS

例

```
ScheduleTableStatusType SomeScheduleTableStatus;
```

4.42 ScheduleTableType

スケジュールテーブルの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
ScheduleTableType SomeScheduleTable;
```

4.43 SignedTickType

カウンタのチック数を表す符号付きスカラ値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
SignedTickType drift = -2;
Os_SyncScheduleTableRel(MyTable, drift);
```

4.44 SpinlockIdType

スピンロックの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

値

コンフィギュレーション設定時に宣言されるスピンロックのシンボリック名です。

定数

```
INVALID_SPINLOCK
```

例

```
TASK(MyTask){
  ...
  GetSpinlock(Spinlock1);
  ...
  ReleaseSpinlock(Spinlock1);
}
```

4.45 StatusType

API 関数のステータスを定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

E_OK
E_OS_ACCESS
E_OS_CALLEVEL
E_OS_ID
E_OS_LIMIT
E_OS_NOFUNC
E_OS_RESOURCE
E_OS_STATE
E_OS_VALUE
E_OS_SERVICEID
E_OS_ILLEGAL_ADDRESS
E_OS_MISSINGEND
E_OS_DISABLEDINT
E_OS_STACKFAULT
E_OS_PROTECTION_MEMORY
E_OS_PROTECTION_TIME
E_OS_PROTECTION_ARRIVAL
E_OS_PROTECTION_LOCKED
E_OS_PROTECTION_EXCEPTION
E_OS_CORE
E_OS_SPINLOCK
E_OS_INTERFERENCE_DEADLOCK
E_OS_NESTING_DEADLOCK
E_OS_SYS_NO_RESTART
E_OS_SYS_RESTART
E_OS_SYS_OVERRUN

例

```
StatusType ErrorCode;
ErrorCode = ActivateTask(MyTask);
```

4.46 Std_ReturnType

IOC API 関数のために AUTOSAR OS だけが使用する、AUTOSAR の標準 API 関数サービスの戻り型です。この型は 8 ビットの符号なし整数で、上位 6 ビットにモジュール固有のエラーコードをエンコードできます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

E_OK=0
E_NOT_OK
IOC_E_OK
IOC_E_NOK
IOC_E_LIMIT
IOC_E_LOST_DATA
IOC_E_NO_DATA

例

```
Std_ReturnType ErrorCode;
ErrorCode = IocSend_OverTheRainbow(Dorothy);
if (ErrorCode == IOC_E_OK){
  /* call succeeded */
} else {
  /* call failed */
}
```

4.47 Std_VersionInfoType

モジュールのAUTOSARバージョン情報が含まれているフィールドからなるCの構造体(struct)です。Std_Types.h 内に定義されます。

以下のフィールドで構成されています。

vendorID
moduleID
instanceID（AUTOSAR R3.x のみ）
sw_major_version
sw_minor_version
sw_patch_version

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

ETAS を表す vendorID フィールドの値は 11
AUTOSAR OS を表す moduleID フィールドの値は 1

例

```
Std_VersionInfoType Version;
GetVersionInfo(&Version);
if (Version.vendorID == 11) {
  /* Make ETAS-specific API call */
  AdvanceCounter(HardwareCounter);
}
```

4.48 TaskRefType

TaskType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TaskType SomeTask;
TaskRefType TaskRef = &SomeTask;
```

4.49 TaskStateRefType

TaskStateType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TaskStateType TaskState;
TaskStateRefType TaskStateRef = &TaskState;
```

4.50 TaskStateType

タスクの現在の状態を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

SUSPENDED
READY
WAITING
RUNNING

例

```
TaskStateType TaskState;
GetTaskState(&TaskState);
```

4.51 TaskType

タスクの型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

値

コンフィギュレーション設定時に宣言されるタスクのシンボリック名です。

定数

```
INVALID_TASK
```

例

```
TaskType SomeTask;
```

4.52 TickRefType

TickType 型のオブジェクトへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TickRefType SomeTick = &tickvalue;
GetCounterValue(MyCounter,SomeTick);
```

4.53 TickType

カウンタのチック数を表すスカラ値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TickType StartTime = 42;
TickType NoRepeat = 0;
SetAbsAlarm(MyAlarm,StartTime,NoRepeat);
```

4.54 TrustedFunctionIndexType

トラステッド関数のインデックスの値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

コンフィギュレーション設定時に宣言されるトラステッド関数のシンボリック名です。

定数

```
INVALID_FUNCTION
```

例

```
CallTrustedFunction(Func3, &data);
```

4.55 TrustedFunctionParameterRefType

トラステッド関数用の引数への参照です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
CallTrustedFunction(Func3, &data);
```

4.56 TryToGetSpinlockType

TryToGetSpinlock の呼び出しの結果を定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

値

TRYTOGETSPINLOCK_SUCCESS
TRYTOGETSPINLOCK_NOSUCCESS

例

```
TryToGetSpinlock(MyLock, &retval);
if (retval == TRYTOGETSPINLOCK_SUCCESS) {...}
```

4.57 boolean

TRUE か FALSE の値しか取らない 8 ビットの値です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

0=FALSE
1=TRUE

例

```
if (Condition == TRUE) {
  x = y;
}
```

4.58 float32

単精度浮動小数点数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
float32 x;
```

4.59 float64

倍精度浮動小数点数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
float64 x;
```

4.60 sint16

符号付き 16 ビット整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

-32768～32767

例

```
sint16 x;
```

4.61 sint16_least

16 ビット以上の符号付き整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

最小範囲： -32768～32767

例

```
sint16_least x;
```

4.62 sint32

符号付き 32 ビット整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

-2147483648～2147483647

例

```
sint32 x;
```

4.63 sint32_least

32 ビット以上の符号付き整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

最小範囲： -2147483648～2147483647

例

```
sint32_least x;
```

4.64 sint8

符号付き 8 ビット整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

-128～127

例

```
sint8 x;
```

4.65 sint8_least

8 ビット以上の符号付き整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

最小範囲： -128～127

例

```
sint8_least x;
```

4.66 uint16

符号なし 16 ビット整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

0～65535

例

```
uint16 x;
```

4.67 uint16_least

16 ビット以上の符号なし整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

最小範囲： 0～65535

例

```
uint16_least x;
```

4.68 uint32

符号なし 32 ビット整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

0～4294967295

例

```
uint32 x;
```

4.69 uint32_least

32 ビット以上の符号なし整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

最小範囲： 0～4294967295

例

```
uint32_least x;
```

4.70 uint8

符号なし 8 ビット整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

0～255

例

```
uint8 x;
```

4.71 uint8_least

8 ビット以上の符号なし整数です。Platform_Types.h 内に定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

値

最小範囲： 0～255

例

```
uint8_least x;
```

5 RTA-OS のマクロ

5.1 ALARMCALLBACK

アラームコールバック関数を宣言します。アラームコールバック内で実行できる OS API 関数は SuspendAllInterrupts()と ResumeAllInterrupts()だけです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
ALARMCALLBACK(MyCallback){...}
```

5.2 CAT1_ISR

カテゴリ 1 ISR のエントリ関数を作成するためのマクロです。ユーザーコードをターゲット間で移植できるようにするためのものです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE

例

```
CAT1_ISR(MyISR) {...}
```

5.3 DONOTCARE

マルチコアシステムでは、StartOS の呼び出しを除くすべての呼び出しが、DONOTCARE を AppModeType として渡すことができます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.4 DeclareAlarm

アラームを宣言するためのマクロで、C の変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内のアラームはすべて RTA-OS が自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
DeclareAlarm(MyAlarm);
```

5.5 DeclareCounter

カウンタを宣言するためのマクロで、C の変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内のカウンタはすべてRTA-OSが自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
DeclareCounter(MyCounter);
```

5.6 DeclareEvent

イベントを宣言するためのマクロで、C の変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内のイベントはすべてRTA-OSが自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
DeclareEvent(MyEvent);
```

5.7 DeclareISR

ISR を宣言するためのマクロで、C の変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内の ISR はすべて RTA-OS が自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
DeclareISR(MyISR);
```

5.8 DeclareResource

リソースを宣言するためのマクロで、C の変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内のリソースはすべてRTA-OSが自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
DeclareResource(MyResource);
```

5.9 （空き）

5.10 DeclareScheduleTable

スケジュールテーブルを宣言するためのマクロで、Cの変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内のスケジュールテーブルはすべて RTA-OS が自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
DeclareScheduleTable(MyScheduleTable);
```

5.11 DeclareTask

タスクを宣言するためのマクロで、C の変数の外部宣言と同様に機能します。通常は、ユーザーのコンフィギュレーション内のタスクはすべて RTA-OS が自動的に宣言するので、ユーザーがこのマクロを使用する必要はありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
DeclareTask(MyTask);
```

5.12 INVALID_SPINLOCK

'no spinlock'（該当するスピンロックがない）という意味を表します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.13 ISR

カテゴリ 2 ISR のエントリ関数を作成するために必ず使用されるマクロです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
ISR(MyISR) {...}
```

5.14 OSCYCLEDURATION

命令サイクルの長さ（ナノ秒単位）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
time_in_ns = CycleMeasurement * OSCYCLEDURATION;
```

5.15 OSCYCLESPERSECOND

1 秒当たりの命令サイクル数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
time_in_secs = CycleMeasurement / OSCYCLESPERSECOND;
```

5.16 OSErrorGetServiceId

エラーを生成したサービスの識別子を返します。

値はOSServiceIdType型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
OSServiceIdType WhatServiceFailed = OSErrorGetServiceId();
```

5.17 OSMAXALLOWEDVALUE

SystemCounter というカウンタの取り得る最大値（チック）を表す定数の定義です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
SetAbsAlarm(MyAlarm,OSMAXALLOWEDVALUE,0)
```

5.18 OSMAXALLOWEDVALUE_<CounterID>

CounterID というカウンタの取り得る最大値（チック）を表す定数の定義です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
SetAbsAlarm(MyAlarm,OSMAXALLOWEDVALUE_SomeCounter,0)
```

5.19 OSMEMORY_IS_EXECUTABLE

メモリが実行可能であることをアクセス権が示しているかどうかを調べます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_EXECUTABLE(rights)) {...}
```

5.20 OSMEMORY_IS_READABLE

メモリが読み取り可能であることをアクセス権が示しているかどうかを調べます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_READABLE(rights)) {...}
```

5.21 OSMEMORY_IS_STACKSPACE

メモリがスタック空間であることをアクセス権が示しているかどうかを調べます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_STACKSPACE(rights)) {...}
```

5.22 OSMEMORY_IS_WRITEABLE

メモリが書き込み可能であることをアクセス権が示しているかどうかを調べます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_WRITEABLE(rights)) {...}
```

5.23 OSMINCYCLE

SystemCounterというカウンタにアタッチされているサイクリックアラームの最小チック数を表す定数の定義です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
if (ComputedValue < OSMINCYCLE) { SetAbsAlarm(MyAlarm,42,OSMINCYCLE);
} else {
    SetAbsAlarm(MyAlarm,42,ComputedValue);
}
```

5.24 OSMINCYCLE_<CounterID>

CounterID というカウンタにアタッチされているサイクリックアラームの最小チック数を表す定数の定義です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
if (ComputedValue < OSMINCYCLE_SomeCounter) {
    SetAbsAlarm(MyAlarm,42,OSMINCYCLE_SomeCounter);
} else {
    SetAbsAlarm(MyAlarm,42,ComputedValue);
}
```

5.25 OSSWTICKDURATION

ストップウォッチの 1 チックの長さ（ナノ秒単位）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
time_in_ns = StopwatchMeasurement * OSSWTICKDURATION;
```

5.26 OSSWTICKSPERSECOND

ストップウォッチの 1 秒当たりのチック数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
time_in_secs = CycleMeasurement / OSSWTICKSPERSECOND;
```

5.27 OSTICKDURATION

SystemCounter というカウンタの 1 チックの長さ（ナノ秒単位）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
uint32   RealTimeDeadline = 50000000; /* 50 ms */
TickType Deadline = (TickType)RealTimeDeadline / OSTICKDURATION;
SetRelAlarm(Timeout,Deadline,0);
```

5.28 OSTICKDURATION_<CounterID>

CounterID というカウンタの 1 チックの長さ（ナノ秒単位）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32   RealTimeDeadline = 50000000; /* 50 ms */
TickType Deadline = (TickType)RealTimeDeadline /
    OSTICKDURATION_SomeCounter;
SetRelAlarm(Timeout,Deadline,0);
```

5.29 OSTICKSPERBASE

SystemCounter というカウンタの単位あたりのチック数を表す定数の定義です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

5.30 OSTICKSPERBASE_<CounterID>

CounterID というカウンタの単位あたりのチック数を表す定数の定義です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

5.31 OS_ACTIVATION_MONITORING

このマクロは、タスク起動監視が有効に設定されている場合に限り定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_ACTIVATION_MONITORING
Os_StopwatchTickType t;
Os_GetTaskActivationTime(MyTask,&t);
#endif
```

5.32 OS_ADD_TASK

このマクロは、Os_TasksetType にタスクを追加するためのものです。タスクの実行を遅延させるように設定されている場合に限り使用できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_TasksetType t1_and_t3 = TASK_MASK(Task1);
OS_ADD_TASK(t1_and_t3, Task3);
```

5.33 OS_CORE_CURRENT

Os_GetIdleElapsedTime および Os_ResetIdleElapsedTime という API 関数の呼び出し元コアの ID です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.34 OS_CORE_FOR_<TaskOrISR>

タスク／ISR が実行されているコアの ID を返します。マルチコアでのみ使用可能です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
CoreIdType core;
core = OS_CORE_FOR_MyTask;
core = OS_CORE_FOR_MyCat2ISR;
core = OS_CORE_FOR_MyCat1ISR;
```

5.35 OS_CORE_FOR_ISR

ISR が実行されているコアの ID を返します。マルチコアでのみ使用可能で、カテゴリ 2 ISR の名前だけを渡すことができます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
CoreIdType core;
```

```
core = OS_CORE_FOR_ISR(MyCat2ISR);
core = OS_CORE_FOR_ISR(MyCat1ISR);
```

5.36 OS_CORE_FOR_TASK

タスクが実行されているコアの ID を返します。マルチコアでのみ使用可能で、TaskType を渡すことができます。渡された TaskType の値に対するチェックは行われません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
CoreIdType core;
core = OS_CORE_FOR_TASK(MyTask);
```

5.37 OS_CORE_ID_0

コア 0 の論理 ID です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.38 OS_CORE_ID_1

コア 1 の論理 ID です。コア 2 以降も同様です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.39 OS_CORE_ID_MASTER

マスタコアの ID です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.40 OS_COUNT_USER_n

OS オプション'Collect OS usage metrics'が有効になっている場合は、10 個のユーザーカウンタ（0～9）が使用可能です。ユーザーはユーザーコード内に OS_COUNT_USER_0()～OS_COUNT_USER_9()を配置して、適切なカウンタをインクリメントさせることができます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
TASK(MyTask){
  ...
#ifdef OS_METRICS_ENABLED
 OS_COUNT_USER_5();
#endif /* OS_METRICS_ENABLED */
  ...
}
```

5.41 OS_COUNT_cat1isrname

OS はカテゴリ 1 ISR を装備できないので、OS オプション'Collect OS usage metrics'が有効になっている場合、OS は、発生した割り込みの数をキャプチャするためのマクロを作成するので、これをユーザーのハンドラ内に配置することができます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
CAT1_ISR(MyCat1ISR) {
  ...
#ifdef OS_METRICS_ENABLED
  OS_COUNT_MyCat1ISR();
#endif /* OS_METRICS_ENABLED */
  ...
}
```

5.42 OS_CURRENT_IDLEMODE

呼び出し元コア用のアイドルモードを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

5.43 OS_DURATION_<ScheduleTableID>

スケジュールテーブル ScheduleTableID の期間を表す定数（そのスケジュールテーブルのカウンタのチック数）を定義します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.44 OS_ELAPSED_TIME_RECORDING

このマクロは、経過時間記録が有効に設定されている場合に限り定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_ELAPSED_TIME_RECORDING
Os_StopwatchTickType total_idle_time;
Os_StopwatchTickType total_taskA_time;
total_idle_time = Os_GetIdleElapsedTime(OS_CORE_CURRENT);
total_taskA_time = Os_GetTaskElapsedTime(TaskA);
#endif
```

5.45 OS_EXTENDED_STATUS

拡張ステータスが設定される場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_EXTENDED_STATUS
CheckStatusType = ActivateTask(Task1);
if (CheckStatusType == E_OS_LIMIT) {
   /* Log an error */
}
#else
ActivateTask(Task1);
#endif
```

5.46 OS_FAST_TASK_TERMINATION

標準（STANDARD）ステータスに限り、タスクの高速ターミネーションが有効に設定されている場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.47 OS_IMASK_FOR_<TaskOrISR>

タスク／ISR の割り込み優先度／マスクを返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_imaskType imask;
imask = OS_IMASK_FOR_MyTask;
imask = OS_IMASK_FOR_MyCat2ISR;
imask = OS_IMASK_FOR_MyCat1ISR;
```

5.48 OS_IMASK_FOR_ISR

ISR の割り込み優先度マスクを返します。これには ISR の名前を渡す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_imaskType imask;
imask = OS_IMASK_FOR_ISR(MyCat2ISR);
imask = OS_IMASK_FOR_ISR(MyCat1ISR);
```

5.49 OS_IMASK_FOR_TASK

指定されたタスクの割り込み優先度マスクを返します。そのタスクが内部リソースを ISR と共有する場合は、通常、返される値は必ず「ゼロ以外（'nonzero'）」です。このマクロにはタスクの名前を渡す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_imaskType imask;
imask = OS_IMASK_FOR_TASK(MyTask);
```

5.50 OS_INDEX_TO_ISRTYPE

ISR 固有のインデックス（0～OS_NUM_ISRS-1 の範囲内の値）を有効な ISRType に変換するマクロです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
ISRType isr = OS_INDEX_TO_ISRTYPE(2);
```

5.51 OS_INDEX_TO_TASKTYPE

タスク固有のインデックス（0～OS_NUM_TASKS- 1 の範囲内の値）を有効な TaskType に変換するマクロです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
ActivateTask(OS_INDEX_TO_TASKTYPE(2));
```

5.52 OS_INVALID_TPL

実行状態のタスクがないときに Os_GetCurrentTPL()から返される値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
if (OS_INVALID_TPL == Os_GetCurrentTPL()) { /* In idle */ }
```

5.53 OS_ISRTYPE_TO_INDEX

有効な ISRType を ISR 固有のインデックス（0～OS_NUM_ISRS-1 の範囲内の値）に変換するマクロです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
extern Os_StopwatchTickType isr_time[OS_NUM_ISRS];
isr_time[OS_ISRTYPE_TO_INDEX(GetISRID())] = GetExecutionTime();
```

5.54 OS_MAIN

メインプログラムを宣言します。可搬性のあるコードにするには main()ではなく OS_MAIN()を使用することをお勧めします。main()の引数と戻り値についての要件はコンパイラごとに異なるからです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#include "Os.h" OS_MAIN() {
  /* Initialize target hardware */
  StartOS(OSDEFAULTAPPMODE);
}
```

5.55 OS_NOAPPMODE

OS が実行状態でないときに GetActiveApplicationMode()から返される値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.56 OS_NO_TASKS

空の Os_TasksetType を初期化するためのマクロです。タスクの実行を遅延させるように設定されている場合に限り使用できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_TasksetType t1 = OS_NO_TASKS;
```

5.57 OS_NUM_ALARMS

宣言されているアラームの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.58 OS_NUM_APPLICATIONS

宣言されている OS アプリケーションの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.59 OS_NUM_APPMODES

宣言されている AppMode（アプリケーションモード）の数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.60 OS_NUM_CORES

宣言されているコアの数（OsNumberOfCores）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.61 OS_NUM_COUNTERS

宣言されているカウンタの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.62 OS_NUM_EVENTS

宣言されているイベントの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.63 OS_NUM_ISRS

宣言されているカテゴリ 2 ISR の数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.64 OS_NUM_OS_CORES

OS 用として設定されているコアの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.65 OS_NUM_RESOURCES

宣言されているリソース（ただし内部リソース以外）の数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.66 OS_NUM_SCHEDULETABLES

宣言されているスケジュールテーブルの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.67 OS_NUM_SPINLOCKS

宣言されているスピンロックの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.68 OS_NUM_TASKS

宣言されているタスクの数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.69 OS_NUM_TRUSTED_FUNCTIONS

宣言されているトラステッド関数の数です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.70 OS_REGSET_<RegisterSetID>_SIZE

ランタイムにおいてレジスタセット<RegisterSetID>を保存するのに必要なバッファのサイズを定義するマクロです。バッファが必要ない場合、このマクロは宣言されません。そのような状況は、当該のレジスタセットを使用するタスク／ISR を同じレジスタセットを使用する別のタスク／ISR がプリエンプトできない場合に発生する可能性があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_REGSET_FP_SIZE
  fp_context_save_area fpsave[OS_REGSET_FP_SIZE];
#endif /* OS_REGSET_FP_SIZE */
```

5.71 OS_SCALABILITY_CLASS_1

AUTOSAR スケーラビリティクラス 1 が設定される場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_SCALABILITY_CLASS_1
ALARMCALLBACK(OnlyInSC1){
    ...
}
#endif
```

5.72 OS_SCALABILITY_CLASS_2

AUTOSAR スケーラビリティクラス 2 が設定される場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#if defined(OS_SCALABILITY_CLASS_2) ||
    defined(OS_SCALABILITY_CLASS_4)
StartScheduleTableSynchron(Table);
#endif
```

5.73 OS_SCALABILITY_CLASS_3

AUTOSAR スケーラビリティクラス 3 が設定される場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#if defined(OS_SCALABILITY_CLASS_3) ||
   defined(OS_SCALABILITY_CLASS_4)
FUNC(void, {memclass}) ErrorHook_MyApplication(StatusType Error){
 /* Handle OS-Application error */
}
#endif
```

5.74 OS_SCALABILITY_CLASS_4

AUTOSAR スケーラビリティクラス 4 が設定される場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#if defined(OS_SCALABILITY_CLASS_3) ||
   defined(OS_SCALABILITY_CLASS_4)
FUNC(void, {memclass}) ErrorHook_MyApplication(StatusType Error){
 /* Handle OS-Application error */
}
#endif
```

5.75 OS_STACK_MONITORING

このマクロは、スタック監視が設定される場合に限り定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
FUNC(boolean, {memclass}) Os_Cbk_Idle(void){
    #ifdef OS_STACK_MONITORING
        Os_StackSizeType Task1Stack, Task2Stack, Task3Stack;
        Task1Stack = Os_GetTaskMaxStackUsage(Task1);
        Task2Stack = Os_GetTaskMaxStackUsage(Task2);
        ...
        TaskNStack = Os_GetTaskMaxStackUsage(TaskN);
    #endif
    return TRUE;
}
```

5.76 OS_STANDARD_STATUS

標準ステータスが設定される場合に定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_STANDARD_STATUS
ActivateTask(Task1);
#else
CheckStatusType = ActivateTask(Task1);
if (CheckStatusType == E_OS_LIMIT) {
   /* Log an error */
}
#endif
```

5.77 OS_TASKTYPE_TO_INDEX

有効な TaskType をタスク固有のインデックス（0～OS_NUM_TASKS-1 の範囲内の値）に変換するマクロです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
extern Os_StopwatchTickType task_time[OS_NUM_TASKS];
TaskType this;
GetTaskID(&this);
task_time[OS_TASKTYPE_TO_INDEX(this)] = GetExecutionTime();
```

5.78 OS_TICKS2<Unit>_<CounterID>(ticks)

CounterID のチック数を、Unit を単位とする値に変換します。Unit は、NS（ナノ秒）、MS（ミリ秒）、US（マイクロ秒）、SEC（秒）のいずれかです。

RTA-OS は、可能であれば整数の乗算または除算を使用してこれらのマクロを生成しようとします。しかしチックレートによっては、精度を保つために計算に浮動小数点数を使用しなければならない場合があります。コンパイル時に判明する固定値がマクロに渡される時点において、通常はコンパイラが計算を実行して整数の結果を埋め込みます。渡される値が変数の場合は、コンパイラはランタイムに浮動小数点演算を使用するコードを生成する必要があります。ユーザーはファイル Os_Cfg.h をチェックしてマクロ用のコードを調べ、ユーザーのアプリケーション内でこのことが問題にならないかどうかを確認してください。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

例

```
time_in_ms = OS_TICKS2MS_SystemCounter(time);
```

5.79 OS_TIME_MONITORING

このマクロは、時間監視が設定される場合に限り定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
#ifdef OS_TIME_MONITORING
Os_StopwatchTickType start,end,function_duration;
start = Os_GetExecutionTime();
#endif
ThirdPartyFunction(x,y);
#ifdef OS_TIME_MONITORING
end = Os_GetExecutionTime();
function_duration = end - start;
#endif
```

5.80 OS_TPL_FOR_<Task>

Task について、内部的なタスク基本優先度を返します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 tpl;
tpl = OS_TPL_FOR_MyTask;
```

5.81 OS_TPL_FOR_TASK

指定されたタスクについて、内部的なタスク基本優先度を返します。これにはタスクの名前を渡す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
uint32 tpl;
tpl = OS_TPL_FOR_TASK(MyTask);
```

5.82 TASK

タスクのエントリ関数を作成するために使用されるマクロです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

例

```
TASK(MyTask) {...}
```

5.83 TASK_MASK

TaskType を Os_TasksetType に変換するためのマクロです。タスクの実行を遅延させるように設定されている場合に限り使用できます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

例

```
Os_TasksetType t1 = TASK_MASK(Task1);
```

6 RTA-TRACE の API 関数

6.1 説明の形式

各API関数の情報は、すべて以下の構成で記述されています。

構文

```
/* C function prototype for the API call （API 関数の C 関数プロトタイプ）*/
Return Value NameOfAPICall(Parameter Type, ...)
```

引数

API 関数の引数とそのモードが示されています。

in API関数に渡される引数

out API関数に引数への参照（ポインタ）を渡し、API関数がそこに値をセットします。

inout API 関数に引数を渡し、API 関数がそれを更新します。

戻り値

API 関数が StatusType 型の値を返す場合は、値の一覧と、それぞれの値が示すエラー／警告の内容が記載されています。「ビルド」列は、標準（standard）ステータスと拡張（extended）ステータスの両方において値が戻るか（「All」）、または拡張ステータスでのみ戻るか（「拡張」）を示しています。

説明

API関数の挙動の詳細説明です。

可搬性

RTA-TRACE の API 関数は以下の 6 つのクラスに分かれています。

OSEK: OSEK OS の仕様に準拠する関数です。他の OSEK OS にポーティング可能で、他の AUTOSAR OS にもポーティング可能です。

R3.x: AUTOSAR R3.x の仕様に準拠する関数です。他の AUTOSAR OS R3.x にポーティング可能です。OSEK OS にも準拠している場合に限り、他の OSEK OS にポーティング可能です。

R4.x: AUTOSAR R4.0 および R4.1 の仕様に準拠する関数です。他の AUTOSAR OS R4.x にポーティング可能です。OSEK OS にも準拠している場合に限り、他の OSEK OS にポーティング可能です。

MultiCore: AUTOSAR R4.0 multicore OS の仕様に準拠する関数です。他の AUTOSAR OS R4.x にポーティング可能です。複数のコアの構成が設定され、ターゲットバリアントが複数のコアをサポートしている場合に限り、使用可能です。

RTA-TRACE: RTA-OSがRTA-TRACEランタイム監視ツールに関連する関数です。RTA-TRACEのサポートが設定されている場合に限り、使用可能です。

RTA-OS: 他のすべてのクラスの関数とAUTOSAR OSの拡張機能を提供する関数です。RTA-OS独自のもので、他の OS にはポーティングできません。

コーティング例

```
A C code listing showing how to use the API calls
```
（API 関数呼び出しのコーディング例）

呼び出し元コンテキスト

API 関数を呼び出せる環境が示されています。✓ の付いた環境からの呼び出しが可能で、✗ の付いた環境からは呼び出せません。

参照

関連する API 関数の一覧（リンク）です。

6.2 マルチコアに関する注記

RTA-TRACE は制御やトレースのデータをマルチコアアプリケーション内のコア別のメモリ領域に格納します。 本章に記載されている RTA-TRACE のすべての主要 API 関数は、それぞれが呼び出されたコア上で個別に動作します。そのため、たとえばあるコアについてはバーストモードのトレースを行い、別のコアについてはトリガモードのトレースを行う、といったことが可能です。トレースされたデータのアップロードに関係する API 関数もコア別の挙動を取るので、マルチコアとシングルコアの両方のアプリケーションに同じ API 関数とアップロードストラテジを使用することができます。ただ 1 つ異なるのは、トレース通信ハードウェアの初期化はマルチコアアプリケーションの場合でも 1 つのコアだけで行えばよいという点です。RTA-OS は、必ず同時に 1 つのコアだけがトレースデータの送信を行うように管理を行います。

6.3 Os_CheckTraceOutput

トレースデータの有無を確認します。

構文

```
void Os_CheckTraceOutput(void)
```

説明

フリーランニングモードでトレースを行う場合、アプリケーションはこの API 関数を定期的に呼び出す必要があります。この API 関数は、RTA-TRACE にアップロードするデータがトレースバッファに格納されたことを検知します。

バーストモードやトリガモードの場合は、これを呼び出しても何も影響はありません。

送信するべきデータがある場合は Os_Cbk_TraceCommDataReady()が呼び出されます。

この API 関数は呼び出しが行われたコアについてのトレースデータの有無だけを確認します。2 つ以上のコアのデータをトレースしている場合は、各コアでこの API 関数 を個別に呼び出す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_CheckTraceOutput();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady
Os_Cbk_TraceCommTxStart

6.4 Os_ClearTrigger

すべてのトリガ発行条件をクリアします。

構文

```
void Os_ClearTrigger(void)
```

説明

この API 関数は、API 関数 Os_TriggerOnXXX()を使用してセットされたすべてのトリガ発行条件をクリアします。

トレース情報は引き続きトレースバッファに記録されますが、どのトレースレコードもホストへのトレースバッファのアップロードをトリガすることはありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_ClearTrigger();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_SetTraceRepeat
Os_SetTriggerWindow
Os_StartBurstingTrace
Os_StartFreeRunningTrace

6.5 Os_DisableTraceCategories

どのトレースポイントをトレースするかをコントロールします。

構文

```
void Os_DisableTraceCategories(
     Os_TraceCategoriesType CategoriesMask
)
```

引数

引数	モード	説明
CategoriesMask	in	Os_TraceCategoriesType ディセーブルにするトレースカテゴリのマスク

説明

トレースカテゴリは、トレースポイント、タスクトレースポイント、インターバルのうち、記録するかどうかを指定するものをフィルタリングするためのものです。一般的には、トレースされるデータ量を制限する目的で使用されます。

各カテゴリは、「常にアクティブ（ALWAYS）」、「常に非アクティブ（NEVER）」、「ランタイムにコントロール（RUNTIME）」のいずれかのモードでビルドすることができます。RUNTIME に指定されたカテゴリのトレースをイネーブルにするには Os_EnableTraceCategories を使用し、ディセーブルにするには Os_DisableTraceCategories を使用します。

この API 関数は指定されたランタイムカテゴリをディセーブルにするので、マスクで指定されたすべてのトレースポイント、タスクトレースポイント、インターバルの記録が抑止されます。

この API 関数の呼び出し時に指定されなかったカテゴリは現在の状態を維持します。マルチコアアプリケーションでは、このカテゴリフィルタはすべてのコアに適用されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_DisableTraceCategories(DebugTracePoints | DataLogTracePoints);
    /* Disable DebugTracePoints and DataLogTracePoints*/
Os_LogTracepoint(tpTest, DebugTracePoints); /* tpTest is not
    recorded: DebugTracePoints is disabled */
Os_LogTracepoint(tpTest, OS_TRACE_CATEGORY_ALWAYS); /* tpTest is
    recorded here */
Os_DisableTraceCategories(OS_TRACE_ALL_CATEGORIES); /* Disable
    all categories except for OS_TRACE_CATEGORY_ALWAYS */
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_EnableTraceCategories

6.6 Os_DisableTraceClasses

どの型のオブジェクトをトレースするかをコントロールします。

構文

```
void Os_DisableTraceClasses(
    Os_TraceClassesType ClassMask
)
```

引数

引数	モード	説明
ClassMask	in	Os_TraceClassesType ディセーブルにするトレースクラスのマスク

説明

トレースクラスは、記録するトレースイベントのクラスをフィルタリングするためのものです。一般的には、トレースされるデータ量を制限する目的で使用されます。

各トレースクラスは、「常にアクティブ（ALWAYS）」、「常に非アクティブ（NEVER）」、「ランタイムにコントロール（RUNTIME）」のいずれかのモードでビルドすることができます。RUNTIMEに指定されたクラスのトレースをイネーブルにするにはOs_EnableTraceClassesを使用し、ディセーブルにするにはOs_DisableTraceClassesを使用します。

このAPI関数は指定されたランタイムクラスをディセーブルにするので、それらのクラスによりフィルタリングされるイベントのトレースが抑止されます。指定されなかったクラスは現在の状態を維持します。

マルチコアアプリケーションでは、このクラスフィルタはすべてのコアに適用されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_DisableTraceClasses(OS_TRACE_TRACEPOINT_CLASS);
Os_LogTracepoint(tpTest, OS_TRACE_ALL_CATEGORIES); /* Will not
    get recorded */
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_EnableTraceClasses

6.7 Os_EnableTraceCategories

どのトレースポイントをトレースするかをコントロールします。

構文

```
void Os_EnableTraceCategories(
     Os_TraceCategoriesType CategoriesMask
)
```

引数

引数	モード	説明
CategoriesMask	in	Os_TraceCategoriesType イネーブルにするトレースカテゴリのマスク

説明

トレースカテゴリは、トレースポイント、タスクトレースポイント、インターバルのうち、記録するカテゴリをフィルタリングするためのものです。一般的には、トレースされるデータ量を制限する目的で使用されます。

各カテゴリは、「常にアクティブ（ALWAYS）」、「常に非アクティブ（NEVER）」、「ランタイムにコントロール（RUNTIME）」のいずれかのモードでビルドすることができます。RUNTIME に指定されたカテゴリのトレースをイネーブルにするには Os_EnableTraceCategories を使用し、ディセーブルにするには Os_DisableTraceCategories を使用します。

この API 関数は、指定されたランタイムカテゴリをイネーブルにし、指定されなかったカテゴリは現在の状態を維持します。マルチコアアプリケーションでは、このカテゴリフィルタはすべてのコアに適用されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_EnableTraceCategories(DebugTracePoints | DataLogTracePoints);
Os_LogTracepoint(tpTest, DebugTracePoints); /* tpTest is recorded
    */
Os_LogTracepoint(tpTest, FunctionProfileTracePoints); /* tpTest
    is not recorded - FunctionProfileTracePoints not enabled */
Os_LogTracepoint(tpTest, OS_TRACE_ALL_CATEGORIES); /* tpTest is
    recorded */
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_EnableTraceCategories

6.8 Os_EnableTraceClasses

どの型のオブジェクトがトレースされるかをコントロールします。

構文

```
void Os_EnableTraceClasses(
     Os_TraceClassesType ClassMask
)
```

引数

引数	モード	説明
ClassMask	in	Os_TraceClassesType イネーブルにするトレースクラスのマスク

説明

トレースクラスは、すべての種類のトレースイベントを記録するかどうかをフィルタリングするためのものです。一般的には、トレースされるデータ量を制限する目的で使用されます。

各トレースクラスは、「常にアクティブ（ALWAYS）」、「常に非アクティブ（NEVER）」、「ランタイムにコントロール（RUNTIME）」のいずれかのモードでビルドすることができます。RUNTIMEに指定されたクラスのトレースをイネーブルにするにはOs_EnableTraceClassesを使用し、ディセーブルにするにはOs_DisableTraceClassesを使用します。

このAPI関数は指定されたランタイムクラスをイネーブルにし、指定されなかったクラスは現在の状態を維持します。

マルチコアアプリケーションでは、このクラスフィルタはすべてのコアに適用されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_EnableTraceClasses(OS_TRACE_TRACEPOINT_CLASS);
Os_LogTracepoint(tpTest, OS_TRACE_ALL_CATEGORIES); /* Will get
       recorded */
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OSフック		RTA-OSフック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_DisableTraceClasses

6.9 Os_LogCat1ISREnd

カテゴリ 1 ISR の終了を記録します。

構文

```
void Os_LogCat1ISREnd(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType カテゴリ 1 ISR の ID

説明

この API 関数はカテゴリ 1 ISR を終了します。このタイプの ISR はオペレーティングシステムによって制御されるものではないので、これに対する自動トレースが発生することはありません。記録が必要な場合は、マニュアル操作でこの API 関数を呼び出す必要があります。

このイベントは、OS_TRACE_TASKS_AND_ISRS_CLASS トレースクラスがアクティブな場合に限り記録されます。

カテゴリ 1 ISR を正しくトレースするには、開始と終了の両方の時点で記録を行う必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
CAT1_ISR(Category1Handler) {
  Os_LogCat1ISRStart(Category1Handler);
  ...
  Os_LogCat1ISREnd(Category1Handler);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_LogCat1ISRStart

6.10 Os_LogCat1ISRStart

カテゴリ 1 ISR の開始を記録します。

構文

```
void Os_LogCat1ISRStart(
     ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType カテゴリ 1 ISR の ID

説明

この関数はカテゴリ 1 ISR の開始をマークします。このタイプの ISR はオペレーティングシステムによって制御されるものではないので、これに対する自動トレースが発生することはありません。カテゴリ 1 ISR の記録が必要な場合は、マニュアル操作でこの API 関数を呼び出す必要があります。

このイベントは、OS_TRACE_TASKS_AND_ISRS_CLASS トレースクラスがアクティブな場合に限り記録されます。

カテゴリ 1 ISR を正しくトレースするには、開始と終了の両方の時点で記録を行う必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
CAT1_ISR(Category1Handler) {
  Os_LogCat1ISRStart(Category1Handler);
  ...
  Os_LogCat1ISREnd(Category1Handler);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

参照

Os_LogCat1ISREnd

6.11 Os_LogCriticalExecutionEnd

クリティカルセクションの実行終了を記録します。

構文

```
void Os_LogCriticalExecutionEnd(
        Os_TraceInfoType CriticalExecutionID
)
```

引数

引数	モード	説明
CriticalExecutionID	in	Os_TraceInfoType クリティカルな実行プロファイルの識別子

説明

クリティカルポイントの実行終了をトレースバッファに記録します。一般的にこの API 関数は、タスク／ISR がタイムクリティカルなコードセクションの実行を完了したことを示すためのものです。タスク／ISR のデッドラインがそのタスク／ISR の終了より前に発生するような場合にこの API 関数が役立ちます。

CriticalExecutionID は、OS_TRACE_TASKS_AND_ISRS_CLASS クラスがアクティブな場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
TASK(MyTask){
  ...
  ReadSensor(X);
  Os_LogCriticalExecutionEnd(SensorRead);
  ...
  WriteActuator(Y);
  Os_LogCriticalExecutionEnd(SensorRead);
  ...
  TerminateTask();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.12 Os_LogIntervalEnd

測定インターバルの終了を記録します。

構文

```
void Os_LogIntervalEnd(
     Os_TraceIntervalIDType IntervalID,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

インターバルの終了をトレースバッファに記録します。

インターバルは、OS_TRACE_INTERVAL_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTime, SystemLoggingCategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.13 Os_LogIntervalEndData

測定インターバルの終了を、関連データと共に記録します。

構文

```
void Os_LogIntervalEndData(
     Os_TraceIntervalIDType IntervalID,
     Os_TraceDataPtrType DataPtr,
     Os_TraceDataLengthType Length,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID
DataPtr	in	Os_TraceDataPtrType 記録されるデータブロックの先頭アドレスへのポインタ
Length	in	Os_TraceDataLengthType データブロックの長さ（バイト数）
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

インターバルの終了を、関連するデータと共にトレースバッファに記録します。

インターバルは、OS_TRACE_INTERVAL_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEndData(EndToEndTime, &DataBlock,
    4,SystemLoggingCategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogIntervalEnd
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.14 Os_LogIntervalEndValue

測定インターバルの終了を、関連する値と共に記録します。

構文

```
void Os_LogIntervalEndValue(
     Os_TraceIntervalIDType IntervalID,
     Os_TraceValueType Value,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID
Value	in	Os_TraceValueType インターバルと共に記録される数値
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

インターバルの終了とそれに関連する数値をトレースバッファに記録します。

インターバルは、OS_TRACE_INTERVAL_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEndValue(EndToEndTime, 42, SystemLoggingCategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.15 Os_LogIntervalStart

測定インターバルの開始を記録します。

構文

```
void Os_LogIntervalStart(
     Os_TraceIntervalIDType IntervalID,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

インターバルの開始をトレースバッファに記録します。

インターバルは、OS_TRACE_INTERVAL_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTime, SystemLoggingCategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.16 Os_LogIntervalStartData

測定インターバルの開始を、関連データと共に記録します。

構文

```
void Os_LogIntervalStartData(
     Os_TraceIntervalIDType IntervalID,
     Os_TraceDataPtrType DataPtr,
     Os_TraceDataLengthType Length,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID
DataPtr	in	Os_TraceDataPtrType 記録されるデータブロックの先頭アドレスへのポインタ
Length	in	Os_TraceDataLengthType データブロックの長さ（バイト数）
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

インターバルの開始とそれに関連するデータをトレースバッファに記録します。

インターバルは、OS_TRACE_INTERVAL_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogIntervalStartData(EndToEndTime, &DataBlock, 4,
    SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTimeSystemLoggingCategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartValue

6.17 Os_LogIntervalStartValue

測定インターバルの開始を、関連する値と共に記録します。

構文

```
void Os_LogIntervalStartValue(
     Os_TraceIntervalIDType IntervalID,
     Os_TraceValueType Value,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID
Value	in	Os_TraceValueType インターバルと共に記録される数値
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

インターバルの開始とそれに関連する数値をトレースバッファに記録します。

インターバルは、OS_TRACE_INTERVAL_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogIntervalStartValue(EndToEndTime, 42, SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTime, SystemLoggingCategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartData

6.18 Os_LogProfileStart

新しい実行プロファイルの開始を記録します。

構文

```
void Os_LogProfileStart(
    Os_TraceInfoType ProfileID
)
```

引数

引数	モード	説明
ProfileID	in	Os_TraceInfoType プロファイルの ID

説明

どの実行プロファイルがアクティブであるかをトレースバッファに記録します。実行プロファイルは、あるタスク／ISR 内の実行ルートが外部条件によって異なる場合に、それを識別する際に役立ちます。

プロファイルは、OS_TRACE_TASKS_AND_ISRS_CLASS クラスがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
TASK(MyTask){
  if (some_condition()) {
    Os_LogProfileStart(TrueRoute);
    ...
  } else {
    Os_LogProfileStart(FalseRoute);
    ...
  }
  TerminateTask();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.19 Os_LogTaskTracepoint

指定されたカテゴリのトレースポイントを記録します。

構文

```
void Os_LogTaskTracepoint(
    Os_TraceTracepointIDType TaskTracepointID,
    Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モー	説明
TaskTracepointID	in	Os_TraceTracepointIDType タスクトレースポイントの ID
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

タスクトレースポイントイベントをトレースバッファに記録します。

TaskTracepointID は、OS_TRACE_TASK_TRACEPOINT_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogTaskTracepoint(MyTaskTracePoint, ACategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogTaskTracepointData
Os_LogTaskTracepointValue
Os_LogTaskTracepoint
Os_LogTaskTracepointData
Os_LogTaskTracepointValue

6.20 Os_LogTaskTracepointData

指定されたカテゴリのトレースポイントを、関連データと共に記録します。

構文

```
void Os_LogTaskTracepointData(
     Os_TraceTracepointIDType TracepointID,
     Os_TraceDataPtrType DataPtr,
     Os_TraceDataLengthType Length,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
TracepointID	in	Os_TraceTracepointIDType トレースポイントの ID
DataPtr	in	Os_TraceDataPtrType 記録されるデータブロックの先頭アドレスへのポインタ
Length	in	Os_TraceDataLengthType データブロックの長さ（バイト数）
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

タスクトレースポイントイベントとそれに関連するデータをトレースバッファに記録します。

TaskTracepointID は、OS_TRACE_TASK_TRACEPOINT_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogTaskTracepointData(MyTracePoint, &DataBlock, 4, ACategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogTaskTracepoint
Os_LogTaskTracepointValue
Os_LogTaskTracepoint
Os_LogTaskTracepointData
Os_LogTaskTracepointValue

6.21 Os_LogTaskTracepointValue

指定されたカテゴリのトレースポイントを、関連する値と共に記録します。

構文

```
void Os_LogTaskTracepointValue(
     Os_TraceTracepointIDType TracepointID,
     Os_TraceValueType Value,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
TracepointID	in	Os_TraceTracepointIDType トレースポイントの ID
Value	in	Os_TraceValueType トレースポイントと共に記録される数値
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

タスクトレースポイントイベントとそれに関連する数値をトレースバッファに記録します。

TaskTracepointID は、OS_TRACE_TASK_TRACEPOINT_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogTaskTracepointValue(MyTracePoint, 99, ACategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogTaskTracepoint
Os_LogTaskTracepointData
Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointValue

6.22 Os_LogTracepoint

指定されたカテゴリのトレースポイントを記録します。

構文

```
void Os_LogTracepoint(
     Os_TraceTracepointIDType TracepointID,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
TracepointID	in	Os_TraceTracepointIDType トレースポイントの ID
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

トレースポイントイベントをトレースバッファに記録します。

TracepointID は、OS_TRACE_TRACEPOINT_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogTracepoint(MyTracepoint, ACategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogTracepointData
Os_LogTracepointValue

6.23 Os_LogTracepointData

指定されたカテゴリのトレースポイントを、関連データと共に記録します。

構文

```
void Os_LogTracepointData(
     Os_TraceTracepointIDType TracepointID,
     Os_TraceDataPtrType DataPtr,
     Os_TraceDataLengthType Length,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
TracepointID	in	Os_TraceTracepointIDType トレースポイントの ID
DataPtr	in	Os_TraceDataPtrType 記録されるデータブロックの先頭アドレスへのポインタ
Length	in	Os_TraceDataLengthType データブロックの長さ（バイト数）
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

トレースポイントイベントとそれに関連するデータをトレースバッファに記録します。

TracepointID は、OS_TRACE_TRACEPOINT_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogTracepointData(MyTracePoint, &DataBlock, 4, ACategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogTracepoint
Os_LogTracepointValue

6.24 Os_LogTracepointValue

指定されたカテゴリのトレースポイントを、関連する値と共に記録します。

構文

```
void Os_LogTracepointValue(
     Os_TraceTracepointIDType TracepointID,
     Os_TraceValueType Value,
     Os_TraceCategoriesType CategoryMask
)
```

引数

引数	モード	説明
TracepointID	in	Os_TraceTracepointIDType トレースポイントの ID
値	in	Os_TraceValueType トレースポイントと共に記録される数値
CategoryMask	in	Os_TraceCategoriesType カテゴリマスク

説明

トレースポイントイベントとそれに関連する数値をトレースバッファに記録します。

TracepointID は、OS_TRACE_TRACEPOINT_CLASS クラスがアクティブで、かつ CategoryMask 内の 1 つ以上のカテゴリがアクティブである場合に限り記録されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_LogTracepointValue(MyTracePoint, 99, ACategory);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogTracepoint
Os_LogTracepointData

6.25 Os_SetTraceRepeat

トレースを繰り返すかどうかをコントロールします。

構文

```
void Os_SetTraceRepeat(
    boolean Repeat
)
```

引数

引数	モード	説明
Repeat	in	boolean バーストモードとトリガモードのトレースを繰り返すかどうかを示す論理値

説明

Repeat の値が TRUE の場合、バーストモードとトリガモードのトレースは最後のトレース内容がトレースバッファから RTA-TRACE クライアントに伝送された後に自動的に再開されます。

この API 関数はフリーランニングモードのトレースには影響がありません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_SetTraceRepeat(TRUE);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_StartBurstingTrace
Os_StartTriggeringTrace

6.26 Os_SetTriggerWindow

トリガモードのトレースでアップロードされるトレースバッファウィンドウのサイズを設定します。

構文

```
void Os_SetTriggerWindow(
    Os_TraceIndexType Before,
    Os_TraceIndexType After
)
```

引数

引数	モード	説明
Before	in	Os_TraceIndexType トリガイベントの前に記録するレコードの数
After	in	Os_TraceIndexType トリガイベントの後に記録するレコードの数

説明

この API 関数は、トリガイベントの前と後に記録するレコードの数を設定します。

トリガが発生すると、After 個のトレースレコードがトレースバッファに書き込まれるまでトレースイベントの記録が続行され、その後、記録されたデータがアップロードされます。

アップロードされるレコードの合計数（Before + After）はトレースバッファのサイズにより制限されます。

データ値が付随するトレースイベントを記録する場合は、1 つのイベントについて複数のレコードがトレースバッファに書き込まれる可能性があります。つまり、トリガポイントの前または後に検知されるイベントの数は、要求したレコードの数よりも少なくなる可能性があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_SetTriggerWindow(100,50);
  Os_StartTriggeringTrace();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_StartBurstingTrace
Os_StartFreeRunningTrace

6.27 Os_StartBurstingTrace

バーストモードのトレースを開始します。

構文

```
void Os_StartBurstingTrace(void)
```

説明

バーストモードのトレースでは、トレースバッファが満杯になるまでトレース情報がトレースバッファに記録されます。トレースバッファが満杯になると、トレースが停止してデータ転送が開始されます。トレースバッファが満杯にならないうちはデータのアップロードは行われません。

Os_SetTraceRepeat()によってバーストトレースの反復が有効になっている場合は、データ転送が完了してトレースバッファが空になると、トレースが再開（レジューム）されます。

トレース実行中にこの呼び出しが実行されると、トレースバッファがクリアされてからトレースが再開されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_StartBurstingTrace();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_SetTraceRepeat
Os_StartFreeRunningTrace
Os_StartTriggeringTrace

6.28 Os_StartFreeRunningTrace

フリーランニングモードのトレースを開始します。

構文

```
void Os_StartFreeRunningTrace(void)
```

説明

フリーランニングモードのトレースでは、トレースバッファ内に空きがある間はトレース情報が記録されます。トレースバッファからホストへのデータのアップロードは、データの記録と同時に並行して行われます。

トレースバッファが満杯になると、バッファ内に空きができるまでトレースデータの記録は停止（サスペンド）され、トレースバッファ内に空きができるとトレースが再開（レジューム）されます。転送したいトレースデータの量に対して通信速度が遅すぎると、トレースバッファが満杯になってしまう可能性があります。

トレース実行中にこの API 関数が呼び出されると、トレースバッファがクリアされてからトレースが再開されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_StartFreeRunningTrace();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_StartBurstingTrace
Os_StartTriggeringTrace

6.29 Os_StartTriggeringTrace

トリガモードのトレースを開始します。

構文

```
void Os_StartTriggeringTrace(void)
```

説明

トリガモードのトレースでは、トリガイベントが発生するまで、トレース情報がトレースバッファに連続的に記録されます。トレースバッファがオーバーフローすると、既存の情報の上に新しいトレース情報が上書きされます。

Os_SetTriggerWindow()を使用して「トリガウィンドウ」、つまり「トリガ前（プリトリガ）とトリガ後（ポストトリガ）の合計のバッファレコード数」を指定して、トリガイベントの前後の指定範囲のイベントだけが記録されるようにする必要があります。トリガウィンドウが指定されないと、予測不可能な挙動が発生する可能性があります。

トリガイベントは、Os_TriggerOnXXX() API 関数でセットされます。

トリガリングイベント（タスクの実行開始など）が発生すると、指定されたポストトリガレコード数だけ記録を行った後、ホストへのデータ転送が開始されます。

Os_SetTraceRepeat()によってトレースの反復が許可されている場合は、データ転送の完了後、トレースが再開（レジューム）されます。

トレース実行中にこの API 関数が呼び出されると、トレースバッファがクリアされてからトレースが再開されます。

マルチコアアプリケーションでは、コアごとに異なるトリガ設定が可能です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_SetTriggerWindow(100,50);
  Os_StartTriggeringTrace();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_SetTraceRepeat
Os_SetTriggerWindow
Os_StartBurstingTrace
Os_StartFreeRunningTrace

6.30 Os_StopTrace

トレースを停止します。

構文

```
void Os_StopTrace(void)
```

説明

トレースバッファへのデータの記録を停止します。トレースバッファ内に残っているデータがある場合、それらはすべてホストにアップロードされます。

この関数はデータリンクを停止しません。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_StopTrace();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_StartBurstingTrace
Os_StartFreeRunningTrace
Os_StartTriggeringTrace

6.31 Os_TraceCommInit

トレース用の外部通信機能を初期化します。

構文

```
Os_TraceStatusType Os_TraceCommInit(void)
```

戻り値

Os_TraceStatusType 型の値を返します。

説明

この API 関数はトレース通信リンクを初期化します。ユーザーがデバッガリンクを使用してトレースデータを抽出する際には、使用できません。

このAPI関数はコールバック関数Os_Cbk_TraceCommInitTarget()を呼び出して適切なターゲットハードウェアを初期化し、Os_Cbk_TraceCommInitTarget()の戻り値を返します。

自動トレースが設定されている場合、RTA-OS はこの API 関数を StartOS の処理の中で呼び出します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) StartupHook() {
  ...
  Os_TraceCommInit();
  Os_StartFreeRunningTrace();
  ...
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_Cbk_TraceCommInitTarget

6.32 Os_TraceDumpAsync

非同期通信を使用して、1 回のオペレーションでトレースデータをアップロードします。

構文

```
void Os_TraceDumpAsync(
     Os_AsyncPushCallbackType fn
)
```

説明

一般的に、この API 関数は Os_Cbk_TraceCommDataReady()に対応して呼び出されます。この API 関数には、1 文字だけを伝送する関数への参照を渡します。この API 関数は伝送すべき各文字ごとにその関数を呼び出した後、リターンします。

アップロードを行うには、初期化済みの適切な非同期シリアルデバイスが必要です。標準的なシリアルリンクの設定は 115200bps、8 データビット、パリティなし、1 ストップビットです。

この API 関数は、自身が呼び出されたコアのトレースデータだけをアップロードします。データをトレースしているコアが 2 つ以上ある場合は、それらの各コアについてこの API 関数を呼び出す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, OS_CODE) push_async_io(uint8 val) {
 while(!async_tx_ready) {/* wait for room */} async_transmit(val) ;
}
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void) {
 Os_TraceDumpAsync(push_async_io);
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_Cbk_TraceCommDataReady

6.33 Os_TriggerNow

トレースバッファのアップロードをトリガします。

構文

```
void Os_TriggerNow(void)
```

説明

この API 関数はトリガを強制的に発生させます。これにより、他のトリガ条件とは無関係にトレースバッファがアップロードされます。

この API 関数はトリガ条件の状態を変更しません。

この API 関数はアップロードのトリガリングのみを行います。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerNow();
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.34 Os_TriggerOnActivation

タスクの起動時にトリガを発行します。

構文

```
void Os_TriggerOnActivation(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

説明

指定されたタスクが起動された時点でトレーストリガ（起動トリガ）が発生するように設定します。

TaskID に OS_TRIGGER_ANY という値をセットすると、どのタスクが起動されたときでもトリガが発生します。

このトリガは、タスクが ActivateTask、StartOS、アラーム、スケジュールテーブルによって起動される際に発生します。

ChainTask(TaskID)が実行されても起動トリガは発生しません。Os_TriggerOnChain()を参照してください。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnActivation(InterestingTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnChain

6.35 Os_TriggerOnAdvanceCounter

カウンタに割り当てられたアラームやスケジュールテーブルの満了ポイントまで進んだ際にトリガを発行します。

構文

```
void Os_TriggerOnAdvanceCounter(
     CounterType CounterID
)
```

引数

引数	モード	説明
CounterID	in	CounterType ハードウェアカウンタの ID

説明

指定されたハードウェアカウンタが、アラームやスケジュールテーブルの満了ポイントまで進んだ時点でトレーストリガが発生するように設定します。

CounterID に OS_TRIGGER_ANY という値をセットすると、すべてのカウンタについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnAdvanceCounter(HWCounter);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnIncrementCounter

6.36 Os_TriggerOnAlarmExpiry

アラームの満了時にトリガを発行します。

構文

```
void Os_TriggerOnAlarmExpiry(
     AlarmType AlarmID
)
```

引数

引数	モード	説明
AlarmID	in	AlarmType アラームの ID

説明

指定されたアラームが満了した時点でトレーストリガが発生するように設定します。

AlarmID に OS_TRIGGER_ANY という値という値をセットすると、すべてのアラームの満了または満了ポイントについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnAlarmExpiry(Alarm_10ms);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.37 Os_TriggerOnCat1ISRStart

カテゴリ 1 ISR の開始時にトリガを発行します。

構文

```
void Os_TriggerOnCat1ISRStart(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType カテゴリ 1 ISR の ID

説明

指定されたカテゴリ 1 ISR が実行を開始した時点でトレーストリガが発生するように設定します。

ISRID に OS_TRIGGER_ANY という値をセットすると、すべてのカテゴリ 1 ISR についてトリガが発生します。

RTA-OS はカテゴリ 1 ISR をコントロールしないので、ユーザーの責任において、割り込みハンドラの冒頭で Os_LogCat1ISRStart()を呼び出す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnCat1ISRStart(InterestingCat1ISR);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogCat1ISREnd
Os_TriggerOnCat1ISRStop

6.38 Os_TriggerOnCat1ISRStop

カテゴリ 1 ISR の終了時にトリガを発行します。

構文

```
void Os_TriggerOnCat1ISRStop(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType カテゴリ 1 ISR の ID

説明

指定されたカテゴリ 1 ISR が実行を終了した時点でトレーストリガが発生するように設定します。

ISRID に OS_TRIGGER_ANY という値をセットすると、すべてのカテゴリ 1 ISR についてトリガが発生します。

RTA-OS はカテゴリ 1 ISR をコントロールしないので、ユーザーの責任において、割り込みハンドラの最後に Os_LogCat1ISREnd()を呼び出す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnCat1ISRStop(InterestingCat1ISR);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_LogCat1ISREnd
Os_LogCat1ISRStart
Os_TriggerOnCat1ISRStart

6.39 Os_TriggerOnCat2ISRStart

カテゴリ 2 ISR の開始時にトリガを発行します。

構文

```
void Os_TriggerOnCat2ISRStart(
     ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType カテゴリ 2 ISR の ID

説明

指定されたカテゴリ 2 ISR が実行を開始した時点でトレーストリガが発生するように設定します。

ISRID に OS_TRIGGER_ANY という値をセットすると、すべてのカテゴリ 2 ISR についてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnCat2ISRStart(InterestingCat2ISR);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnCat2ISRStop

6.40 Os_TriggerOnCat2ISRStop

カテゴリ 2 ISR の終了時にトリガを発行します。

構文

```
void Os_TriggerOnCat2ISRStop(
    ISRType ISRID
)
```

引数

引数	モード	説明
ISRID	in	ISRType カテゴリ 2 ISR の ID

説明

指定されたカテゴリ 2 ISR が実行を終了した時点でトレーストリガが発生するように設定します。

ISRID に OS_TRIGGER_ANY という値をセットすると、すべてのカテゴリ 2 ISR についてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnCat2ISRStop(InterestingCat2ISR);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnCat2ISRStart

6.41 Os_TriggerOnChain

タスクがチェーニングされた際にトリガを発行します。

構文

```
void Os_TriggerOnChain(
     TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

説明

指定されたタスクをチェーニングしようとした時点でトレーストリガが発生するように設定します。ただし実際には、チェーニングは失敗する可能性があります。

TaskID に OS_TRIGGER_ANY という値をセットすると、すべてのタスクのチェーニングについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnChain(InterestingTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnActivation

6.42 Os_TriggerOnError

エラーの発生時にトリガを発行します。

構文

```
void Os_TriggerOnError(
     StatusType Error
)
```

引数

引数	モード	説明
Error	in	StatusType エラーの ID

説明

指定されたエラーが発行されるとトレーストリガが発生するように設定します。

Error に OS_TRIGGER_ANY という値をセットすると、すべてのエラーについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnError(E_OS_LIMIT);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.43 Os_TriggerOnGetResource

リソースがロックされた際にトリガを発行します。

構文

```
void Os_TriggerOnGetResource(
        ResourceType ResourceID
)
```

引数

引数	モード	説明
ResourceID	in	ResourceType リソースの ID

説明

指定されたリソースがロックされた時点でトレーストリガが発生するように設定します。

ResourceID に OS_TRIGGER_ANY という値をセットすると、すべてのリソースのロックについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnGetResource(CriticalSection);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnReleaseResource

6.44 Os_TriggerOnIncrementCounter

カウンタがインクリメントされた際にトリガを発行します。

構文

```
void Os_TriggerOnIncrementCounter(
     CounterType CounterID
)
```

引数

引数	モード	説明
CounterID	in	CounterType ソフトウェアカウンタの ID

説明

指定されたカウンタがインクリメントされた時点でトレーストリガが発生するように設定します。

CounterID に OS_TRIGGER_ANY という値をセットすると、すべてのカウンタのインクリメントについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnIncrementCounter(SWCounter);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnAdvanceCounter

6.45 Os_TriggerOnIntervalEnd

トレースインターバルの終了時にトリガを発行します。

構文

```
void Os_TriggerOnIntervalEnd(
    Os_TraceIntervalIDType IntervalID
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID

説明

指定されたインターバルが終了した時点でトレーストリガが発生するように設定します。

IntervalID に OS_TRIGGER_ANY という値をセットすると、すべてのインターバルの終了についてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnIntervalEnd(EndToEndTimeMeasurement);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnIntervalStart
Os_TriggerOnIntervalStop

6.46 Os_TriggerOnIntervalStart

トレースインターバルの開始時にトリガを発行します。

構文

```
void Os_TriggerOnIntervalStart(
    Os_TraceIntervalIDType IntervalID
)
```

引数

引数	モード	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID

説明

指定されたインターバルが開始された時点でトレーストリガが発生するように設定します。

IntervalID に OS_TRIGGER_ANY という値をセットすると、すべてのインターバルの開始についてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnIntervalStart(EndToEndTimeMeasurement);
```

参照

Os_TriggerOnIntervalEnd
Os_TriggerOnIntervalStop

6.47 Os_TriggerOnIntervalStop

トレースインターバルの終了時にトリガを発行します。

構文

```
void Os_TriggerOnIntervalStop(
        Os_TraceIntervalIDType IntervalID
)
```

引数

引数	モー	説明
IntervalID	in	Os_TraceIntervalIDType インターバルの ID

説明

この API 関数は Os_TriggerOnIntervalEnd のシノニム（別名）です。

指定されたインターバルが終了した時点でトレーストリガが発生するように設定します。

IntervalID に OS_TRIGGER_ANY という値をセットすると、すべてのインターバルの終了についてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnIntervalStop(EndToEndTimeMeasurement);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnIntervalEnd

6.48 Os_TriggerOnReleaseResource

リソースがアンロックされた際にトリガを発行します。

構文

```
void Os_TriggerOnReleaseResource(
    ResourceType ResourceID
)
```

引数

引数	モード	説明
ResourceID	in	ResourceType リソースの ID

説明

指定されたリソースがアンロックされた時点でトレーストリガが発生するように設定します。

ResourceID に OS_TRIGGER_ANY という値をセットすると、すべてのリソースのアンロックについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnReleaseResource(CriticalSection);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnGetResource

6.49 Os_TriggerOnScheduleTableExpiry

指定された満了ポイントの満了時にトリガを発行します。

構文

```
void Os_TriggerOnScheduleTableExpiry(
    ExpiryID
)
```

引数

引数	モード	説明
ExpiryID	in	Os_TraceExpiryIDType 満了ポイントの ID（スケジュールテーブル名と満了ポイント名をアンダースコア文字でつないだもの）

説明

指定された満了ポイントに到達した時点でトレーストリガが発生するように設定します。

ExpiryID に OS_TRIGGER_ANY という値をセットすると、すべての満了ポイントまたはアラームについてトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
StartScheduleTableRel(SchedTable, 1);
Os_TriggerOnScheduleTableExpiry(SchedTable_ep1);
IncrementCounter(SystemCounter);
...
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.50 Os_TriggerOnSetEvent

タスクについてのイベントがセットされた際にトリガを発行します。

構文

```
void Os_TriggerOnSetEvent(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

説明

指定されたタスクについてイベントがセットされた時点でトレーストリガが発生するように設定します。

TaskID に OS_TRIGGER_ANY という値をセットすると、どのイベントがセットされてもトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnSetEvent(ExtendedTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

なし。

6.51 Os_TriggerOnShutdown

OS のシャットダウン時にトリガを発行します。

構文

```
void Os_TriggerOnShutdown(
    StatusType Status
)
```

引数

引数	モード	説明
Status	in	StatusType シャットダウン終了コードの ID

説明

指定された終了ステータスがShutdownOSに渡された時点でトレーストリガが発生するように設定します。

Status に OS_TRIGGER_ANY という値をセットすると、どのステータス値が ShutdownOS に渡されてもトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnShutdown(E_OK); /* Trigger on normal shutdown */
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

ShutdownOS

6.52 Os_TriggerOnTaskStart

タスクの開始時にトリガを発行します。

構文

```
void Os_TriggerOnTaskStart(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

説明

指定されたタスクが実行を開始した時点でトレーストリガが発生するように設定します。

TaskID に OS_TRIGGER_ANY という値をセットすると、すべてのタスクの開始についてトリガが発生します。

TaskID が開始されるのは、自身のエントリ関数が呼び出されたとき、または自身がウェイト状態からレジュームしたときです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnTaskStart(InterestingTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnTaskStop

6.53 Os_TriggerOnTaskStop

タスクの終了時にトリガを発行します。

構文

```
void Os_TriggerOnTaskStop(
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskID	in	TaskType タスクの ID

説明

指定されたタスクが実行を終了した時点でトレーストリガが発生するように設定します。

TaskID に OS_TRIGGER_ANY という値をセットすると、すべてのタスクの終了についてトリガが発生します。

TaskID が終了するのは、そのタスクがターミネートしたときとウェイト状態に入ったときです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnTaskStop(InterestingTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnTaskStart

6.54 Os_TriggerOnTaskTracepoint

タスクトレースポイントが記録された際にトリガを発行します。

構文

```
void Os_TriggerOnTaskTracepoint(
    Os_TraceTracepointIDType TaskTracepointID,
    TaskType TaskID
)
```

引数

引数	モード	説明
TaskTracepointID	in	Os_TraceTracepointIDType タスクトレースポイントの ID
TaskID	in	TaskType タスクの ID

説明

指定されたタスクの指定されたタスクトレースポイントが記録された時点でトレーストリガが発生するように設定します。

TaskID に OS_TRIGGER_ANY という値をセットすると、すべてのタスクについて、TaskTracepointID という値を持つトレースポイントによってトリガが発生します。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnTaskTracepoint(MyTaskTracepoint,InterestingTask);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnTracepoint

6.55 Os_TriggerOnTracepoint

トレースポイントが記録された際にトリガを発行します。

構文

```
void Os_TriggerOnTracepoint(
    Os_TraceTracepointIDType TracepointID
)
```

引数

引数	モード	説明
TracepointID	in	Os_TraceTracepointIDType トレースポイントの ID

説明

指定されたトレースポイントが記録された時点でトレーストリガが発生するように設定します。

TracepointID に OS_TRIGGER_ANY という値をセットすると、すべてのトレースポイントについてトリガが発生するようになります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TriggerOnTracepoint(MyTracepoint);
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_TriggerOnTaskTracepoint

6.56 Os_UploadTraceData

非同期通信を使用して、トレースデータを 1 バイトだけアップロードします。

構文

```
void Os_UploadTraceData(void)
```

説明

トレースデータの各バイトをシリアル通信リンク経由で送信します。アプリケーションコード内のコールバック関数を使用して、実際の通信リンクへのアクセスを管理します。

ポーリングモードでは、データをタイミングよく伝送できる頻度でこの関数を呼び出す必要があります。

特別なケースとして、割り込みモードでは、この関数は Os_Cbk_TraceCommDataReady() コールバックと伝送割り込みハンドラから呼び出される必要があります。

アップロードを行うには、初期化済みの適切な非同期シリアルデバイスが必要です。標準的なシリアルリンクの設定は 115200bps、8 データビット、パリティなし、1 ストップビットです。

注記： 効率上の理由から、この API 関数は必ずトラステッド OS アプリケーションのコードから呼び出す必要があります。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
/* This callback occurs when a new frame is ready for upload */
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void) {
  Os_UploadTraceData(); /* Causes call to
      Os_Cbk_TraceCommTxStart() */
}
ISR(asyncio) { Os_UploadTraceData();
}
FUNC(void, {memclass}) Os_Cbk_TraceCommTxStart(void) {
  /* Called from UploadTraceData when the first byte of a frame
      is ready to send.
   * It is immediately followed by a call to
      Os_Cbk_TraceCommTxByte().
   * In interrupt mode, this is used to enable the transmit
      interrupt.
   */
  enable_asyncio_interrupt();
```

```
}
FUNC(void, {memclass}) Os_Cbk_TraceCommTxByte(uint8 val) {
  /* Called from UploadTraceData when there is a byte ready to
     send */
  async_transmit(val);
}
FUNC(void, {memclass}) Os_Cbk_TraceCommTxEnd(void)    {
  /* Called from UploadTraceData when the last byte of data has
     been sent*/
  disable_asyncio_interrupt();
}
FUNC(boolean, {memclass}) Os_Cbk_TraceCommTxReady(void) {
  /* Called from UploadTraceData to determine whether there is
     room in the transmit buffer */
  /* This should always return true in interrupt mode, because
     the interrupt should only
   * fire when there is room to send the next byte. */
  return async_tx_ready();
}
```

呼び出し元コンテキスト

タスク／ISR		AUTOSAR OS フック		RTA-OS フック	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

参照

Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady
Os_Cbk_TraceCommTxStart
Os_CheckTraceOutput

7 RTA-TRACE のコールバック関数

7.1 説明の形式

コールバック関数は、ユーザーが提供する RTA-TRACE 用関数です。本章では RTA-TRACE が使用するすべてのコールバック関数について説明します。各コールバック関数の情報は、すべて以下の構成で記述されています。

構文

```
/* C function prototype for the callback （コールバック関数の C 関数プロトタイプ）*/
Return Value NameOfCallback(Parameter Type, ...)
```

引数

コールバック関数の引数とそのモードが示されています。

in RTA-TRACEがコールバック関数に渡す引数

out RTA-TRACEがコールバック関数に引数への参照（ポインタ）を渡し、コールバック関数がそこに値をセットして返します。

inout RTA-TRACE がコールバック関数に引数を渡し、コールバック関数がそれを更新して返します。

戻り値

コールバック関数からの戻り値の説明です。

説明

コールバック関数に要求される挙動の詳細説明です。

可搬性

他の環境（OSEK OS、AUTOSAR OS、RTA-OS、RTA- TRACE）への可搬性

コーディング例

```
A C code listing showing how to implement the callback.
（コールバック関数呼び出しのコーディング例）
```

RTA-OS コンフィギュレーション

当該コールバック関数が必要となる RTA-OS のコンフィギュレーション設定が示されています。

参照

関連するコールバック関数の一覧（リンク）です。

7.2 Os_Cbk_TraceCommDataReady

送信可能なトレースデータがあることを通知するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void)
```

説明

バーストモードまたはトリガモードでトレースを行っている場合は、RTA-TRACE にアップロードするべき新しいデータのフレームがあると、このコールバック関数が自動的に呼び出されます。

フリーランニングモードでトレースを行っている場合は、このコールバック関数はOs_CheckTraceOutput()から呼び出されます。Os_CheckTraceOutput()はアプリケーションから定期的に呼び出す必要があります。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TRACECOMMDATAREADY_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void) {
  Os_UploadTraceData(); /* Causes call to
      Os_Cbk_TraceCommTxStart() */
}
```

RTA-OS コンフィギュレーション

トレース機能で通信リンクを使用する場合は、このコールバック関数を実装することができます。カーネルライブラリにデフォルトバージョンが含まれています。

参照

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady

7.3 Os_Cbk_TraceCommInitTarget

アプリケーションで外部通信をトレース機能用に初期化するためのコールバック関数です。

構文

```
FUNC(Os_TraceStatusType, {memclass})
    Os_Cbk_TraceCommInitTarget(void)
```

戻り値

Os_TraceStatusType 型の値を返します。

説明

このコールバック関数には、RTA-TRACE への通信リンクの初期化（RS232 リンクのセットアップなど）を行うアプリケーション固有のコードを実装し、Os_TraceCommInit がこれを呼び出します。

初期化が成功した場合は、E_OK を返す必要があります。他の値を返すと、トレース通信がディセーブルになります。

注記： 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TRACECOMMINITTARGET_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(Os_TraceStatusType, {memclass})
    Os_Cbk_TraceCommInitTarget(void){
  initialize_uart();
  return E_OK;
}
```

RTA-OS コンフィギュレーション

外部通信リンクを使用するトレース機能を Os_TraceCommInit を使用して初期化するには、このコールバック関数を実装する必要があります。

参照

Os_TraceCommInit

7.4 Os_Cbk_TraceCommTxByte

1 バイトのトレースデータを送信用に提供するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxByte(
    uint8 val
)
```

説明

送信する 1 バイトのデータが存在する場合に、UploadTraceData から呼び出されます。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TRACECOMMTXBYTE_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxByte(uint8 val) {
  /* Called from UploadTraceData when there is a byte ready to
      send */
  async_transmit(val);
}
```

RTA-OS コンフィギュレーション

Os_UploadTraceData を使用する場合は、このコールバック関数を実装する必要があります。

参照

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady

7.5 Os_Cbk_TraceCommTxEnd

トレースデータの最後の 1 バイトが送信されたことを通知するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxEnd(void)
```

説明

フレームの最後の 1 バイトが送信された時点で UploadTraceData から呼び出されます。

割り込みモードにおいては、この API 関数で伝送割り込みをディセーブルにします。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TRACECOMMTXEND_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxEnd(void) {
  disable_asyncio_interrupt();
}
```

RTA-OS コンフィギュレーション

Os_UploadTraceData を使用する場合は、このコールバック関数を実装する必要があります。

参照

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxReady

7.6 Os_Cbk_TraceCommTxReady

トレースデータの次の 1 バイトを送信する余地があるかどうかを調べます。

構文

```
FUNC(boolean, {memclass}) Os_Cbk_TraceCommTxReady(void)
```

戻り値

boolean 型の値を返します。

説明

このコールバック関数は、伝送バッファ内に次の 1 バイトを送信するための空きがあるかどうかを調べる際に、UploadTraceData から呼び出されます。

割り込みモードにおいては、次のバイトを送信する余地があるときにのみ割り込みが発行されるので、このコールバック関数は常に TRUE を返すことになります。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TRACECOMMTXREADY_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(boolean, {memclass}) Os_Cbk_TraceCommTxReady(void) {
  return async_tx_ready();
}
```

RTA-OS コンフィギュレーション

Os_UploadTraceData を使用する場合は、このコールバック関数を実装する必要があります。

参照

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd

7.7 Os_Cbk_TraceCommTxStart

トレースデータの最初の 1 バイトが送信できる状態になっていることを通知するコールバック関数です。

構文

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxStart(void)
```

説明

フレームの最初の 1 バイトが送信できる状態になったときに UploadTraceData から呼び出されます。

これが呼び出された直後に Os_Cbk_TraceCommTxByte()が呼び出されます。

割り込みモードにおいては、このコールバック関数は伝送割り込みをイネーブルにします。

注記: 下記のコード内の memclass は、AUTOSAR 3.x の場合は OS_APPL_CODE、AUTOSAR 4.0 の場合は OS_CALLOUT_CODE、AUTOSAR 4.1 の場合は OS_OS_CBK_TRACECOMMTXSTART_CODE です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxStart(void) {
  enable_asyncio_interrupt();
}
```

RTA-OS コンフィギュレーション

Os_UploadTraceData を使用する場合は、このコールバック関数を実装する必要があります。

参照

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady

8 RTA-TRACE の型

8.1 Os_AsyncPushCallbackType

1 つの uint8 値を渡される void 関数へのポインタを表す型です。Os_TraceDumpAsync() により使用されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

8.2 Os_TraceCategoriesType

ユーザー定義のトレースフィルタカテゴリ用のマスク値を格納するための型です。「すべてのカテゴリ」および「該当カテゴリなし」を表す値がデフォルトで定義されています。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

値

OS_TRACE_NO_CATEGORIES
OS_TRACE_ALL_CATEGORIES

例

```
Os_TraceCategoriesType ExtraTracing = DebugTracePoints |
    DataLogTracePoints;
```

8.3 Os_TraceClassesType

トレースフィルタクラス用のマスク値を格納するための型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

値

OS_TRACE_ACTIVATIONS_CLASS
OS_TRACE_RESOURCES_CLASS
OS_TRACE_INTERRUPT_LOCKS_CLASS
OS_TRACE_SWITCHING_OVERHEADS_CLASS
OS_TRACE_TASKS_AND_ISRS_CLASS
OS_TRACE_ERRORS_CLASS
OS_TRACE_TASK_TRACEPOINT_CLASS
OS_TRACE_TRACEPOINT_CLASS

OS_TRACE_INTERVALS_CLASS
OS_TRACE_MESSAGE_DATA_CLASS
OS_TRACE_STARTUP_AND_SHUTDOWN_CLASS
OS_TRACE_ALARMS_CLASS
OS_TRACE_SCHEDULETABLES_CLASS
OS_TRACE_OSEK_EVENTS_CLASS
OS_TRACE_EXPIRY_POINTS_CLASS
OS_TRACE_NO_CLASSES
OS_TRACE_ALL_CLASSES

例

```
Os_TraceClassesType AllTracepoints = OS_TRACE_TRACEPOINT_CLASS |
    OS_TRACE_TASK_TRACEPOINT_CLASS;
```

8.4 Os_TraceDataLengthType

データブロックの長さ（バイト数）です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TraceDataLengthType BlockLength = 8;
```

8.5 Os_TraceDataPtrType

トレースポイントまたはインターバルで記録されるデータブロックへのポインタです。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TraceDataPtrType DataPtr;
uint8 DataValues[10];
...
 DataPtr = &DataValue;
```

8.6 Os_TraceExpiryIDType

満了ポイントを定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

値

満了ポイントの名前。<scheduletable_name>_<expiry_name>というパターンを使用します。

8.7 Os_TraceIndexType

トレースレコード数を表す 16 ビット以上の符号なし整数値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
Os_TraceIndexType PreTriggerRecords = 100;
```

8.8 Os_TraceInfoType

トレースされるオブジェクトを表す符号なし整数値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

8.9 Os_TraceIntervalIDType

RTA-TRACE のトレースインターバルを定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

値

ユーザー定義のトレースインターバルの名前。

8.10 Os_TraceStatusType

トレース API 関数のステータスが格納される型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

値

OS_TRACE_STATUS_OK
OS_TRACE_STATUS_COMM_INIT_FAILURE

8.11 Os_TraceTracepointIDType

RTA-TRACE のトレースポイントを定義する列挙型です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

値

ユーザー定義のトレースポイントの名前。

8.12 Os_TraceValueType

Compact Time のコンフィギュレーション設定に応じて 16 ビットか 32 ビットのいずれかを表す、符号なし整数値です。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9 RTA-TRACE のマクロ

9.1 OS_NUM_INTERVALS

宣言されているトレースインターバルの数。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.2 OS_NUM_TASKTRACEPOINTS

宣言されているタスクトレースポイントの数。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.3 OS_NUM_TRACECATEGORIES

宣言されているトレースカテゴリの数。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.4 OS_NUM_TRACEPOINTS

宣言されているトレースポイントの数。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.5 OS_TRACE

このマクロは、トレース機能が有効になっているに限り定義されます。

可搬性

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

例

```
#ifdef OS_TRACE
  ...
#endif
```

10 コーディング規約

10.1 ネームスペース

C 言語では、すべてのグローバル名は 1 つのネームスペース内に存在します。つまり、すべての関数、変数、定数、型の名前は、コンパイル単位が異なっていても、同じものは使用できません。AUTOSAR 規格は、名前の衝突に起因する問題を避けるため、すべての基本ソフトウェアモジュールについての命名規則を“AUTOSAR General Requirements on Basic Software Modules（AUTOSAR の基本ソフトウェアモジュールについての一般要件）”として定めています。RTA-OS はこの要件を満たしており、RTA-OS が使用するネームスペースでは、以下の文字列で始まる名前はすべて予約されています。

- OS*
- Os*

ただし実際には、AUTOSAR OS が提供している API インターフェースは AUTOSAR の命名規則に適合していません。AUTOSAR OS が使用している型、API 関数、マクロ、定数、コールバック関数などの名前は予約名なので、それらの名前をユーザーコード内で別の用途に使用することはできません。

統合のためのアドバイス 10.1: RTA-OS は、AUTOSAR の一般要件に従って OS の API 関数とマクロを内部的に定義し、それらの AUTOSAR OS 名を C のマクロを利用してユーザーに提供しています。ただし、ErrorHook()や ShutdownHook()などといった標準コールバック関数はこの限りではなく、標準名がそのまま使用されます。

つまり、以下の 2 つの構文の意味はまったく同じです。

```
Os_StatusType Os_ActivateTask(Os_TaskType, Os_TaskId)

StatusType ActivateTask(TaskType, TaskId)
```

ユーザーコード内では必要に応じてこれら 2 つの構文を区別なく使用できますが、標準の AUTOSAR OS の API 関数は後者だけです。

11 コンフィギュレーション言語

11.1 コンフィギュレーションファイル

RTA-OS の設定には、AUTOSAR の ECU パラメータディスクリプション言語を使用します。本章では、AUTOSAR XML を用いた AUTOSAR R4.x 基本ソフトウェアモジュールの設定と、ディスクリプション言語について ETAS が提供している拡張機能の概要をまとめます。

11.2 AUTOSAR XML コンフィギュレーション

AUTOSAR では eXtensible Markup Language（XML）をコンフィギュレーションファイルフォーマットとして使用し、XML スキーマによりタグとそのセマンティックスを定義します。

AUTOSAR XML ファイルは、AUTOSAR XML ファイル用の XML エレメントの構造体を定義した AUTOSAR スキーマインスタンスを参照します。参照情報は以下の例のように記述します。

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0"
    xsi:schemaLocation="http://autosar.org/schema/r4.0 autosar_4
    -0-1.xsd">
  ...
</AUTOSAR>
```

1 つの XML コンフィギュレーションファイルの中に<AUTOSAR>というエレメントを 1 つだけ定義し、その中に他のすべての AUTOSAR エレメントを定義します。<AUTOSAR>と</AUTOSAR>の間のエレメントは、すべて<**ELEMENT**-NAME>という形式で記述します。

11.2.1 パッケージ

<AUTOSAR>エレメントは、<AR-PACKAGES>エレメントを 1 つだけ包含できるコンテナです。<AR-PACKAGES>エレメントは XML オブジェクトツリーのルートを表し、ここからすべてのコンフィギュレーションファイル内のすべてのオブジェクトにアクセスできます。この<AR-PACKAGES>内に 1 つまたは複数のパッケージを<AR-PACKAGE>エレメントとして定義し、それぞれの<AR-PACKAGE>内に、AUTOSAR コンフィギュレーションの特定の部分に関連する 1 つのグループの AUTOSAR エレメントまたは 1 セットのサブパッケージを定義します。

各<AR-PACKAGE>には、<SHORT-NAME>エレメントで名前を付けます。各パッケージには一意の名前を付け、パッケージ内のエレメントを他のパッケージから参照できるようにする必要があります。同じ名前のパッケージがあると、それらは同じパッケージ内の異なる部分を表しているものとみなされます。

```
<AUTOSAR xmlns="http://autosar.org/schema/r4.0"
    xsi:schemaLocation="http://autosar.org/schema/r4.0 autosar_4
    -0-1.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>MyPackage</SHORT-NAME>
      <DESC>This is one of my packages</DESC>
    </AR-PACKAGE>
    ...
    <AR-PACKAGE>
      <SHORT-NAME>MyOtherPackage</SHORT-NAME>
      <DESC>This is another</DESC>
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

<AR-PACKAGE>エレメントは、パッケージ名の定義のほかに、他のエレメントのコンテナとしての役割も果たします。

非基本ソフトウェアコンフィギュレーションは<AR-PACKAGES>レベルでしか分割できません。そのため、複数の XML ファイルを使用する必要がある場合は、コンフィギュレーションを<AR-PACKAGES>レベルで分割します。上の例のファイルを 2 つのファイルに分割するには、第 1 のファイルの内容は以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0"
    xsi:schemaLocation="http://autosar.org/schema/r4.0 autosar_4
    -0-1.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>SWCs</SHORT-NAME>
      <DESC>This is one of my packages</DESC>
      ...
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

第 2 のファイルには、以下のように 2 番目の AR-PACKAGE を含めます。

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0"
    xsi:schemaLocation="http://autosar.org/schema/r4.0 autosar_4
    -0-1.xsd">
    <AR-PACKAGES>
```

```
      <AR-PACKAGE>
        <SHORT-NAME>Interfaces</SHORT-NAME>
        <DESC>This is another</DESC>
        ...
      </AR-PACKAGE>
    </AR-PACKAGES>
</AUTOSAR>
```

11.3 ECU コンフィギュレーションディスクリプション

AUTOSAR基本ソフトウェアには、AUTOSARの他の部分とは異なるコンセプトに基づく「ECU コンフィギュレーションディスクリプションファイル」を使用します。このファイルも XML ファイルですが、XML の使い方は前述の AUTOSAR コンフィギュレーションファイルとは大きく異なります。

ECU コンフィギュレーションディスクリプションは、各基本ソフトウェアモジュールのコンフィギュレーションだけのために XML タグのセットを定義するのではなく、<ECUC-CONTAINER-VALUE>内にコンフィギュレーションデータを保持する CONTAINERS（コンテナ）を包含するモジュールコンフィギュレーションを定義します。

各<ECUC-CONTAINER-VALUE>は<PARAMETER-VALUES>、<REFERENCE-VALUES>、<SUB-CONTAINERS>を含み、<SUB-CONTAINERS>は<ECUC-CONTAINER-VALUE>を含むので、コンフィギュレーションコンテナの階層構造を形成できます。

この階層構造はすべての AUTOSAR 基本ソフトウェアモジュールに共通で、OS についても、COM や NM などと同じフォーマットです。<DEFINITION-REF>を使用することにより、この構造を別の基本ソフトウェアモジュール用にカスタマイズすることができます。それぞれのモジュールコンフィギュレーションと<ECUC-CONTAINER-VALUE>には 1 つの<DEFINITION-REF>が含まれ、<DEFINITION-REF>は AUTOSAR ECU コンフィギュレーション定義を参照します。<DEFINITION-REF>は AUTOSAR ECU コンフィギュレーション定義内の 1 つのコンフィギュレーションアイテムの定義への絶対参照です。これも XML ファイルのひとつであり、コンテナのタイプとコンテナに包含できるコンフィギュレーションエレメントの種類を定義します。

デフォルトでは、参照のルートは/AUTOSAR です。たとえば、OS 関連への参照は以下のようになります。

- /AUTOSAR/EcuDefs/Os/OsTask
- /AUTOSAR/EcuDefs/Os/OsTask/OsTaskPriority
- /AUTOSAR/EcuDefs/Os/OsResource
- /AUTOSAR/EcuDefs/Os/OsIsr

定義ファイル内には、以下の事柄を定義します。

- モジュールコンフィギュレーション内に含めることができる<ECUC-CONTAINER-VALUE>のインスタンスの数

- コンテナ内に含めることができる<PARAMETER-VALUES>、<REFERENCE-VALUES>、<SUB-CONTAINERS>のそれぞれの数。これは「多重度」（Multiplicity）と呼ばれ、定義ファイルには<LOWER-MULTIPLICITY>と<UPPER- MULTIPLICITY >を定義します。
- <ECUC-CONTAINER-VALUE>に含めることができる<PARAMETER-VALUES>、<REFERENCE-VALUES>、<SUB-CONTAINERS>の定義

AUTOSAR OS の設定に使用されるディスクリプションファイルは、定義ファイル内に定義されているルールに従って記述されます。以下の例は、MyTask という名前のタスクが 1 つだけ含まれる OS 用のディスクリプションファイルを示しています。

```
<ELEMENTS>
<ECUC-MODULE-CONFIGURATION-VALUES>
  <SHORT-NAME>OsRTA</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-MODULE-DEF">/AUTOSAR/EcucDefs/Os</
     DEFINITION-REF>
  <CONTAINERS>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>MyTask</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/
         AUTOSAR/EcucDefs/Os/OsTask</DEFINITION-REF>
      <PARAMETER-VALUES>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">/
             AUTOSAR/EcucDefs/Os/OsTask/OsTaskPriority</
             DEFINITION-REF>
          <VALUE>27</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
        <ECUC-TEXTUAL-PARAM-VALUE>
          <DEFINITION-REF DEST="ECUC-ENUMERATION-PARAM-DEF">/
             AUTOSAR/EcucDefs/Os/OsTask/OsTaskSchedule</
             DEFINITION-REF>
          <VALUE>FULL</VALUE>
        </ECUC-TEXTUAL-PARAM-VALUE>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">/
             AUTOSAR/EcucDefs/Os/OsTask/OsTaskActivation</
             DEFINITION-REF>
          <VALUE>1</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
      </PARAMETER-VALUES>
    <REFERENCE-VALUES />
    <SUB-CONTAINERS />
    </ECUC-CONTAINER-VALUE>
  </CONTAINERS>
</ECUC-MODULE-CONFIGURATION-VALUES>
</ELEMENTS>
```

OS 用の標準の AUTOSAR コンフィギュレーションエレメントについては『AUTOSAR Specification of Operating System』に記載されています。

11.4 RTA-OS 用のコンフィギュレーション言語エクステンション

各 AUTOSAR OS ベンダーは、標準の AUTOSAR コンフィギュレーションエレメントに加えて、AUTOSAR で標準化されていない仕様（ベクタアドレスのアロケーションや割り込みの優先度など）を記述するための独自の ECU コンフィギュレーションを使用しています。

AUTOSAR コンフィギュレーションのベンダーエクステンションは、標準の AUTOSAR Standard Module Definition（StMD と呼ばれています）を基に Vendor Specific Module Definition（VSMD）を生成します。ここには AUTOSAR のすべてのエレメントと、ベンダー定義のエレメントが含まれます。この処理については『AUTOSAR Specification of ECU Configuration』に詳しく記載されています。

以降の項では、標準の AUTOSAR コンフィギュレーション属性に加えて RTA-OS がサポートしているエクステンションについて説明します。各項には<ECUC-CONTAINER-VALUE>と、その<ECUC-CONTAINER-VALUE>に含めることのできる<PARAMETER-VALUES>、<REFERENCE-VALUES>、<SUB-CONTAINERS>の定義（または拡張内容）が説明されています。

移植性に関する注意 11.1: ベンダー固有の AUTOSAR エクステンションを他社製の AUTOSAR コンフィギュレーションツール環境に移植することは可能ですが、移植できるのはエクステンションのシンタックスだけで、エクステンションの機能自体は移植できません。つまり、たとえばあるベンダーが OsEnableSpecialOptimization というコンフィギュレーションエレメントを定義している場合、他のベンダーのツールでは“special optimization”の意味が不明のため、何も行うことはできません。

11.4.1 コンテナ: OsAppMode

整数型パラメータ

名前	個数	説明
OsAppModeId	1..1	アプリケーションモードの内部 ID。他のモジュールからアプリケーションモードにアドレスできるようにするためのもので、3.1 でのみ必要です。 **値の範囲:** ..maxint

11.4.2 コンテナ: OsRTATarget

説明

特定のターゲットハードウェアを表すパラメータです。

多重度

0..1

文字列パラメータ

名前	個数	説明
OsRTATargetName	1..1	ターゲットシステムの名前
OsRTATargetVersion	0..1	ターゲットシステム上の OS のバージョン番号
OsRTATargetVariant	0..1	このターゲットシステム用の OS のバリアント

サブコンテナ: Param

説明

ターゲット固有のパラメータ表現です。

多重度

0..*

文字列パラメータ

名前	個数	説明
値	1..1	パラメータの値

11.4.3 コンテナ: OsCounter

文字列パラメータ

名前	個数	説明
OsFormat	0..1	各トレースポイントのフォーマットを定義する文字列

11.4.4 コンテナ: OsIsr

列挙型パラメータ

名前	個数	説明
OsTraceFilter	0..1	この ISR を RTA- TRACE でトレースするかどうかを指定します。許容される値は以下のとおりです。 **ALWAYS** この ISR を常にトレースします。 **NEVER** この ISR をトレースしません。 **RUNTIME** この ISR をトレースするかどうかをユーザーがランタイムにコントロールできます。

整数型パラメータ

名前	個数	説明
OsIsrPriority	1..1	割り込み優先度 **値の範囲:** 0..maxint

文字列パラメータ

名前	個数	説明
OsIsrBudget	0..1	実行バジェットの浮動小数点数表現、続いてタイムベース名、単位
OsIsrStackAllocation	0..1	ISR のマニュアルスタックアロケーション（バイト数）
OsIsrAddress	1..1	割り込みベクタ

参照パラメータ

名前	個数	参照先
OsRegSetRef	0..*	/AUTOSAR/Os/OsRegSet

11.4.5 コンテナ: OsOS

論理値パラメータ

名前	個数	説明
OsSuppressVectorGen	0..1	ベクタテーブルの生成を抑止するか否か

整数型パラメータ

名前	個数	説明
OsCyclesPerSecond	0..1	ターゲットのクロック速度を定義します。 **値の範囲:** 0..maxint
OsTicksPerSecond	0..1	ターゲットのストップウォッチ速度を定義します。 **値の範囲:** 0..maxint

文字列パラメータ

名前	個数	説明
OsDefTaskStack	0..1	デフォルトのスタック値
OsDefCat1Stack	0..1	デフォルトのカテゴリ 1 スタック値
OsDefCat2Stack	0..1	デフォルトのカテゴリ 2 スタック値

サブコンテナ: Param

説明

パラメータの値です。

多重度

0..*

文字列パラメータ

名前	個数	説明
値	1..1	パラメータの値

サブコンテナ: OsHooks

論理値パラメータ

名前	個数	説明
OsStackFaultHook	0..1	スタック障害フックを使用するか否か

11.4.6 コンテナ: OsRegSet

説明

タスク／ISR に関連付けることのできる、ターゲット固有のレジスタセットです。インテグレータは、当該レジスタセットに、そのレジスタセットを使用する特定のタスク／ISR を割り当てます。関連付けが定義されない場合は、最適化が可能になります。

多重度

0..*

11.4.7 コンテナ: OsSpinlock

列挙型パラメータ

名前	個数	説明
OsSpinlockLockMethod	0..1	スピンロックのロッキングメソッドを決定します。 許容される値は以下のとおりです。 **LOCK_ALL_INTERRUPTS** スピンロックが取得されると、すべての割り込みをロックします。 **LOCK_CAT2_INTERRUPTS** スピンロックが取得されると、OS 割り込みをロックします。 **LOCK_NOTHING** スピンロックが取得されても何もロックしません。 **LOCK_WITH_RES_SCHEDULER** スピンロックが取得されると、RES_SCHEDULER をロックします。

		NESTABLE_LOCK_ALL_INTERRUPTS スピンロックが取得されると、すべての割り込みをロックします。ネスティングにも対応します。 **NESTABLE_LOCK_CAT2_INTERRUPTS** スピンロックが取得されると、OS 割り込みをロックします。ネスティングにも対応します。 **NESTABLE_LOCK_NOTHING** スピンロックが取得されても何もロックしません。ネスティングにも対応します。 **NESTABLE_LOCK_WITH_RES_SCHEDULER** スピンロックが取得されると、RES_SCHEDULER をロックします。ネスティングにも対応します。 **COMMONABLE_LOCK_ALL_INTERRUPTS** スピンロックが取得されると、すべての割り込みをロックします。ロックは共有可能です。 **COMMONABLE_LOCK_CAT2_INTERRUPTS** スピンロックが取得されると、OS 割り込みをロックします。ロックは共有可能です。 **COMMONABLE_LOCK_NOTHING** スピンロックが取得されても何もロックしません。ロックは共有可能です **COMMONABLE_LOCK_WITH_RES_SCHEDULER** スピンロックが取得されると、RES_SCHEDULER をロックします。ロックは共有可能です。 **NESTABLE_COMMONABLE_LOCK_ALL_INTERRUPTS** スピンロックが取得されると、すべての割り込みをロックします。ロックは共有可能で、ネスティングにも対応します。

11.4.8 コンテナ: OsTask

列挙型パラメータ

名前	個数	説明
OsTraceFilter	0..1	このタスクを RTA- TRACE でトレースするかどうかを指定します。 許容される値は以下のとおりです。 **ALWAYS** このタスクを常にトレースします。 **NEVER** このタスクをトレースしません。 **RUNTIME** このタスクをトレースするかどうかをユーザーがランタイムにコントロールできます。

文字列パラメータ

名前	個数	説明
OsTaskStackAllocation	0..1	タスクの手動スタックアロケーション
OsTaskWaitStack	0..1	WaitEvent を呼び出すときのタスクのスタック使用量
OsTaskBudget	0..1	実行バジェットの浮動小数点数表現、続いてタイムベース名、単位

参照パラメータ

名前	個数	参照先
OsRegSetRef	0..*	/AUTOSAR/Os/OsRegSet

11.4.9 コンテナ: OsTrace

説明

RTA-TRACE のデータです。

多重度

0..1

論理値パラメータ

名前	個数	説明
OsTraceEnabled	0..1	トレースをイネーブル／ディセーブルにする
OsTraceCompactID	0..1	コンパクト識別子をトレースする
OsTraceCompactTime	1..1	コンパクトな時間フォーマット（16 ビット）を使用する
OsTraceTgtStack	1..1	スタック記録をイネーブルにする
OsTraceTgtTrigger	1..1	ランタイムターゲットトリガ
OsTraceAutoComms	1..1	起動時にトレース通信リンクを初期化する
OsTraceAutoRepeat	1..1	起動時にトレースの繰り返しを指定する

列挙型パラメータ

名前	個数	説明
OsTraceAuto	1..1	RTA-TRACE の自動起動モード。 許容される値は以下のとおりです。 **NONE** トレースを自動起動しません。 **BURSTING** トレースをバーストモード（バッファが満杯になるまで待ってからアップロードするモード）で起動します。 **TRIGGERING** トレースを起動し、トリガを待ちます。 **FREE_RUNNING** 連続的なトレースを起動します。

整数型パラメータ

名前	個数	説明
OsTraceBufferSize	1..1	トレースバッファのサイズ（トレースレコード数） **値の範囲:** 0..maxint

サブコンテナ: OsEnumeration

説明

トレース用の列挙型を定義します。

多重度

0..*

サブコンテナ: OsEnumeration/Param

説明

名前と値のペアです。

多重度

0..*

文字列パラメータ

名前	個数	説明
Value	1..1	パラメータの値

サブコンテナ: OsTraceTracepoint

説明

トレースポイントを定義します。

多重度

0..*

整数型パラメータ

名前	個数	説明
OsTraceTracepointID	1..1	トレースポイント ID（1～n、0 は「RTA-OS による自動割り当て」） **値の範囲:** 0..maxint

文字列パラメータ

名前	個数	説明
OsTraceTracepointFormat	0..1	各トレースポイント用のフォーマットを定義する文字列

サブコンテナ: OsTraceTaskTracepoint

説明

タスクトレースポイントを定義します。

多重度

0..*

整数型パラメータ

名前	個数	説明
OsTraceTaskTracepointID	1..1	タスクトレースポイント ID（1～n、0 は「RTA-OS による自動割り当て」） **値の範囲:** 0..maxint

文字列パラメータ

名前	個数	説明
OsTraceTaskTracepointFormat	0..1	各タスクトレースポイント用のフォーマットを定義する文字列

参照パラメータ

名前	個数	参照先
OsTaskRef	0..1	/AUTOSAR/Os/OsTask
OsIsrRef	0..1	/AUTOSAR/Os/OsIsr

サブコンテナ: OsInterval

説明

名前付きのインターバルを定義します。

多重度

0..*

整数型パラメータ

名前	個数	説明
OsIntervalID	1..1	インターバル識別子（1～n、0 は「RTA-OS による自動割り当て」） **値の範囲:** 0..maxint

文字列パラメータ

名前	個数	説明
OsIntervalFormat	0..1	各インターバル用のフォーマットを定義する文字列

サブコンテナ: Param

説明

名前と値のペアです。

多重度

0..*

文字列パラメータ

名前	個数	説明
値	1..1	パラメータの値

サブコンテナ: OsClass

説明

名前なしのトレースクラスを定義します。

多重度

0..*

論理値パラメータ

名前	個数	説明
OsClassAutostart	0..1	ランタイムのトレースクラスについて、ランタイムに自動で開始するかどうかを決定します。

列挙型パラメータ

名前	個数	説明
OsClassFilter	1..1	クラスのフィルタリングを定義します。 許容される値は以下のとおりです。 **ALWAYS** このクラスを常にトレースします。 **NEVER** このクラスを絶対にトレースしません。 **RUNTIME** このクラスをトレースするかどうかをユーザーがランタイムにコントロールできます。

サブコンテナ: OsCategory

説明

名前付きのトレースクラスを定義します。

多重度

0..*

論理値パラメータ

名前	個数	説明
OsCategoryAutostart	0..1	ランタイムのトレースカテゴリについて、ランタイムに自動で開始するかどうかを決定します。

列挙型パラメータ

名前	個数	説明
OsCategoryFilter	1..1	カテゴリのフィルタリングを定義します。 許容される値は以下のとおりです。 **ALWAYS** このカテゴリを常にトレースします。 **NEVER** このカテゴリをトレースしません。 **RUNTIME** このカテゴリをトレースするかどうかをユーザーがランタイムにコントロールできます。

整数型パラメータ

名前	個数	説明
OsCategoryMask	1..1	カテゴリマスクを定義します。0 は「RTA-OS による自動割り当て」を表します。 **値の範囲:** 0..maxint

11.5 プロジェクトディスクリプションファイル

1 つの論理 OS コンフィギュレーションを、複数の XML コンフィギュレーションファイルに分割して定義することができます。これらのファイルは、**rtaoscfg** コンフィギュレーションツールを使用して、個別または同時に編集することがきます。

RTA-OS では、大規模で複雑なコンフィギュレーションを管理しやすくするため、1 つの論理 OS コンフィギュレーションを構成する複数のファイルをまとめた「プロジェクト」という概念を採用しています。プロジェクトを構成するファイルはプロジェクトファイルから参照されます。

移植性に関する注意 11.2: プロジェクトファイルは RTA-OS ツールに固有のものなので、他社製の AUTOSAR ツール環境には移植できない可能性があります。

プロジェクトファイルは、以下の構造を持つ XML ファイルです。

```
     file  ::=   <?xml version="1.0"?>
             <RTAOS_Project version="1.0">
             [<Working name="filename"/>]
             {<File name="filename"/>}
             [options]
             </RTAOS_Project>
options ::=     <Options>
             {<Option name="filename">value</Option>}
             </Options>
value ::=  booleanvalue | stringvalue |integervalue
```

12 コマンドライン

RTA-OS に付属しているツールはコマンドラインから呼び出せるので、容易にビルドプロセスに組み込むことができます。 すべてのコマンドは任意の数の XML 入力ファイルをツール固有オプションと共に引数として受け取ります。一般的に、コマンドラインオプションは順不同で、オプションと XML ファイルを自由に混ぜて指定することができます。

一部のコマンドラインオプションは、ショート名とロング名（POSIX スタイル）のいずれかを使用でき、実現される機能はどちらでも同じです。

コマンドラインオプションが引数を取る場合は、引数はショート名オプションまたはロング名オプションの直後のコロンに続けて指定します。たとえば、引数 arg を伴うオプションは、以下のいずれかの形式で指定できます。

`command -oarg` または `command --option:arg`

これら 2 つの形式の意味は同じで、1 つのコマンドライン内に両方を混在させることもできます。

引数用のオプション設定値は引数自体の直前に角かっこで囲んで指定します。たとえば、引数 arg の設定値が s である場合は、以下のように指定します。

`command -o[s]arg` または `command --option:[s]arg`

12.1 rtaoscfg

コマンド **rtaoscfg** は、グラフィカルな RTA-OS コンフィギュレーションエディタを実行します。

```
rtaoscfg [オプション] <files>
```

12.1.1 オプション

Option	説明
@<FILE>	<FILE>からコマンドラインオプションを読み取ります。<FILE>内にはコマンドが 1 行に 1 つずつ記述されている必要があります。コマンドファイル内のファイル名に、ホワイトスペースをエスケープするための引用符は必要ありません。@<FILE>オプションは、<FILE>内に複数回指定できます。
--diagnostic	診断情報を標準出力に出力します。診断情報には以下の情報が含まれます。 • ツールの実行ファイルのバージョン • すべてのツールプラグインの名前とバージョン • すべてのターゲットプラグインの名前とバージョン • ライセンスファイルのロケーションと内容

--env:<VALUE>	このツールまたはツールが呼び出すプログラムから参照できる PATH 環境変数の前に<VALUE>を追加します。通常、これはユーザーのコンパイラがパス上にない場合に、そのコンパイラのパスを指定するために使用されます。
--env:[<NAME>]<VALUE>	環境変数<NAME>に、このツール、またはツールが呼び出すプログラムを表す値<VALUE>をセットします。これにより、未定義のコンパイラ関連情報（ライブラリパスなど）を定義することができます。
-h, -?, --help	使用法に関するヘルプ情報を標準出力に出力します。
--nomsgbox	エラーが原因となってコンフィギュレーションツールが終了する際に、ユーザーに処理を促すメッセージボックスを出力しないようにします。
--os_option:<NAME=VALUE>	OS オプション<NAME>を<VALUE>に書き換えます。--os_option:?を使用するとオプションの一覧が表示されます。
-o[<EXPS>]<DIR> --output:[<EXPS>]<DIR>	生成されるすべての出力ファイルをディレクトリ<DIR>に格納します。必須でない<EXPS>　節を指定すると、<EXPS>内にカンマで区切って指定された文字列のリストとマッチする名前を持つ、生成されるすべてのファイルをディレクトリ<DIR>に格納します。文字列には以下のワイルドカードを使用できます。 **?** 任意の 1 文字にマッチします。 ***** 1 文字以上の任意の文字列にマッチします。
--status:<STATUS>	指定された<STATUS>レベル用のカーネルライブラリを生成します。<STATUS>として有効なオプションは以下の 2 つです。 1．STANDARD 2．EXTENDED 入力されたコンフィギュレーションに OsStatus 値が設定されている場合、このオプションがその設定をオーバーライドします。
--target:[<VARIANT>]<TARGET>	指定された<TARGET>用のカーネルライブラリを生成します。複数バージョンの<TARGET>がインストールされている場合は、最も新しいバージョンの<TARGET>が選択されます。 <TARGET>_<VERSION>を使用すると、特定のバージョンを選択することができます。オプション<VARIANT>は<TARGET>のバリアントを選択します。<TARGET>および<VARIANT>の両方が、コンフィギュレーションファイル内の OsTarget および OsTargetVariant の設定値をそれぞれオーバーライドします。--target:?を使用すると、使用可能なターゲットとそれらに関連するバージョンおよびバリアントの一覧が表示されます。

--target_option:<NAME=VALUE>	ターゲットオプション<NAME>を<VALUE>に書き換えます。--target_option:?を使用すると、オプションの一覧を得ることができます。
--target_include:<PATH>	ターゲット DLL を検索するロケーションにディレクトリ<PATH>を追加します。 例: --target_include:..\MyTargets
--trace:<OPTION>	RTA-TRACE をイネーブルまたはディセーブルにします。 <OPTION>には以下のいずれかを指定できます。 **on** RTA-TRACE をイネーブルにします（OsTraceEnabled を TRUE にするのと同等）。 **off** RTA-TRACE をディセーブルにします（OsTraceEnabled を FALSE にするのと同等）。
--xml:<OPTION>	<files>を読み取る際の XML プロセッサの挙動をコントロールします。<OPTION>には以下のいずれかを指定できます。 **Novalidate** XML スキーマに照らしての入力内容の検証は行いません。
--xmlschema:<SCHEMA>	XML をスキーマに照らして検証する場合（--xml:novalidate が設定されていない場合）は、その検証に<SCHEMA>を使用します。

12.1.2 生成されるファイル

rtaoscfg はファイルを直接生成するのではなく、rtaoscfg 内でビルダが使用されると、**rtaosgen** を呼び出されます。**rtaosgen** によって生成されるファイルの詳細については、12.2.2 項を参照してください。

12.1.3 例

Config.arxml という名前のファイルを開いて編集する：

```
rtaoscfg Config.arxml
```

MyProject.rtaos という RTA-OS プロジェクトファイルを開いて編集する：

```
rtaoscfg MyProject.rtaos
```

12.2 rtaosgen

コマンド **rtaosgen** は RTA-OS カーネルライブラリジェネレータを実行します。

```
rtaosgen [オプション] <files>
```

12.2.1 オプション

Option	説明
@<FILE>	<FILE>からコマンドライン引数を読み取ります。<FILE>内ではコマンドが 1 行に 1 つずつ設定されていなければなりません。コマンドファイル内のファイル名に、ホワイトスペースをエスケープするための引用符は必要ありません。@<FILE>オプションは。<FILE>内に複数回指定できます。

--build:<OPTION>	<OPTION>をビルド環境に渡します。<OPTION>は以下のいずれかです。 **verbose** すべてのビルドメッセージを標準出力に出力します。 **quiet** ビルドメッセージを標準出力に出力しません。 **clean** ビルドディレクトリをクリーンアップしてからビルドを開始します。
--debug:<OPTION>	生成されたアセンブラまたはソースコード 1 を保持します。
--diagnostic	診断情報を標準出力に出力します。診断情報には以下の情報が含まれます。 • ツールの実行ファイルのバージョン • すべてのツールプラグインの名前とバージョン • すべてのターゲットプラグインの名前とバージョン • ライセンスファイルのロケーションと内容
--env:<VALUE>	このツールまたはこのツールが呼び出すプログラムから参照できるPATH環境変数の前に<VALUE>を追加します。通常、PATH 環境変数に定義されていないコンパイラのパスを指定する際に使用されます。
--env:[<NAME>]<VALUE>	環境変数<NAME>に、このツールまたはこのツールが呼び出すプログラムを表す値<VALUE>を設定します。これにより、未定義のコンパイラ関連情報（ライブラリパスなど）を定義することができます。
-h, -?, --help	使用法に関する情報を標準出力に出力します。
-I<PATH> --include:<PATH>	値<PATH>内のディレクトリをビルダ用のインクルードパスに追加します。
--nobuild	入力されたコンフィギュレーションについてのチェックは行いますが、RTA-OS ライブラリのビルドは行いません。ユーザーがソースコードライセンスを持っている場合、--debug:source を指定すれば、ターゲットコンパイラが使用可能かどうかを問わず、ソースファイルが作成されます。
--noinfo	すべての情報メッセージを抑止します。
--nowarnings	すべての警告メッセージを抑止します。
-o[<EXPS>]<DIR> --output:[<EXPS>]<DIR>	生成されるすべての出力ファイルをディレクトリ<DIR>に格納します。必須でない<EXPS>　節を指定すると、<EXPS>内にカンマで区切って指定された文字列のリストとマッチする名前を持つ、生成されるすべてのファイルをディレクトリ<DIR>に格納します。文字列には以下のワイルドカードを使用できます。 ? 任意の 1 文字にマッチします。 * 1 文字以上の任意の文字列にマッチします。

--os_option:<NAME=VALUE>	OS オプション<NAME>を<VALUE>に書き換えます。--os_option:?を使用すると、オプションの一覧が表示されます。
--report:<REPORT>	REPORT（レポート）を生成します。--report:?を使用すると、使用可能なレポートの一覧が標準出力に出力されます。
--samples:[<SAMPLE>]<OPTION>	<SAMPLE>用のサンプルコードを生成します。--samples:[<SAMPLE>]overwrite を使用すると、既存のサンプルの上に新しいサンプルが上書きされます。--samples:?を使用すると、使用可能なサンプルが表示されます。
--status:<STATUS>	<STATUS>レベル用のカーネルライブラリを生成します。<STATUS>として有効なオプションは以下の 2 つです。 1．STANDARD 2．EXTENDED 入力されたコンフィギュレーションに OsStatus 値が設定されている場合は、このオプションがその設定をオーバーライドします。
--target:[<VARIANT>]<TARGET>	指定された<TARGET>用のカーネルライブラリを生成します。複数バージョンの<TARGET>がインストールされている場合は、最も新しいバージョンの<TARGET>が選択されます。 <TARGET>_<VERSION>を使用すると、特定のバージョンを選択することができます。オプション<VARIANT>は<TARGET>のバリアントを選択します。<TARGET>と<VARIANT>は、コンフィギュレーションファイル内の OsTarget と OsTargetVariant の設定値をそれぞれオーバーライドします。--target:?を使用すると、使用可能なターゲットとそれらに関連するバージョンおよびバリアントの一覧を生成できます。
--target_include:<PATH>	ターゲット DLL を検索するロケーションにディレクトリ<PATH>を追加します。例: --target_include:..\MyTargets
--target_option:<NAME=VALUE>	ターゲットオプション<NAME>を<VALUE>に書き換えます。--target_option:?を使用すると、オプションの一覧を得ることができます。
--trace:<OPTION>	RTA-TRACE をイネーブルまたはディセーブルにします。 <OPTION>には以下のいずれかを指定できます。 **on**　RTA-TRACE をイネーブルにします（OsTraceEnabled を TRUE にするのと同等です）。 **off**　RTA-TRACE をディセーブルにします（OsTraceEnabled を FALSE にするのと同等です）。
--using:<FILES>	リスト<FILES>内にカンマで区切って指定されている各ファイルを、各ライブラリソースファイルの冒頭で **#include**（インクルード）します。
--verbose	実行時に追加の情報を生成します。

--version	バージョン情報をコンパクト形式で表示します。--diagnostic を使用すると、さらに詳しい情報を得ることができます。
--xml:<OPTION>	<files>を読み取るときの XML プロセッサの挙動をコントロールします。 <OPTION>には以下を指定できます。 **Novalidate** XML スキーマに照らしての入力内容の検証は行いません。
--xmlschema:<SCHEMA>	XML をスキーマに照らして検証する場合（--xml:novalidate が設定されていない場合）は、その検証に<SCHEMA>を使用します。

[1] ソースコードの保持は、有効なソースコードライセンスをお持ちの場合のみ可能です。

12.2.2 生成されるファイル

rtaosgen が実行され、エラーや重大メッセージを生成せずに終了した場合は、以下のファイルが生成されています。

ファイル名	内容
Os.h	OS 用のメインインクルードファイル
Os_Cfg.h	ユーザーが設定したオブジェクトの宣言。これは Os.h によりインクルードされます。
Os_MemMap.h	AUTOSAR メモリマッピングコンフィギュレーション。RTA-OS によって、システム全体の MemMap.h ファイルとマージされます。
RTAOS.<lib>	ユーザーアプリケーションのための RTA-OS ライブラリ。拡張子<lib>はターゲットにより異なります。
RTAOS.<lib>.sig	ユーザーアプリケーション用ライブラリのためのシグネチャファイル。これは、コンフィギュレーションが変更された場合に、カーネルライブラリのどの部分をリビルドする必要があるかを把握するために **rtaosgen** が使用するファイルです。拡張子<lib>はターゲットにより異なります。
<projectname>.log	ビルドプロセス実行中にツールとコンパイラが画面に送信したテキストのコピーが格納されたログファイルです。

これ以外に、各ポート固有のファイルも生成される場合があります。生成される可能性のある上記以外のファイルの一覧は、各ポート用の『Target/Compiler Port Guide』 に掲載されています。

12.2.3 例

以下に、コマンドラインの例を示します。

使用法についての情報を表示する：

```
rtaosgen --help
```

OSを生成する（他のAUTOSARソフトウェアと統合しない）：

Config.arxml によって定義されている OS を生成し、その OS と共に使用されるサンプルの AUTOSAR ヘッダファイルを生成します。さらに、生成されカレントディレクトリに格納されたこれらのファイルと OS 固有ファイルを含むライブラリを作成します。これは、RTA-OS を他社製の AUTOSAR ソフトウェアと統合しない場合に使用する標準的なコマンドラインです。

```
rtaosgen --samples:[Includes] --include:Samples\Includes
   Config.arxml
```

OSを生成する（他のAUTOSARソフトウェアと統合する）：

PathToAutosarHeaderFiles にある AUTOSAR ヘッダファイルとカレントディレクトリに生成される OS 固有ヘッダファイルを使用して、BigConfig.rtaos に定義された OS を生成します。これは、RTA-OS を他社製の AUTOSAR ソフトウェアと統合する場合に使用する標準的なコマンドラインです。

```
rtaosgen --include:PathToAutosarHeaderFiles BigConfig.rtaos
```

サンプルファイルの一覧を出力する（ターゲットManchesterMk1用）：

```
rtaosgen --target:ManchesterMk1 --samples:?
```

レポートの一覧を出力する（ターゲットManchesterMk1用）：

```
rtaosgen --target:ManchesterMk1 --report:?
```

OSを生成する：

最初の例のとおりに OS を生成しますが、既存のサンプルインクルードファイルに上書きし、ターゲットをオーバーライドして ManchesterMk1 にします。

```
rtaosgen --samples:[Includes]overwrite --include:Samples\Includes
   --target:ManchesterMk1 Config.arxml
```

CoreConfig.arxml と TargetConfig.arxml に分割されているディスクリプションから OS を生成します。

```
rtaosgen --include:PathToAutosarHeaderFiles CoreConfig.arxml
   TargetConfig.arxml
```

Config.arxml に定義されている OS を生成し、ヘッダファイルを C:\working\OS\inc に格納し、ライブラリ（および関連するシグネチャファイル）を C:\working\OS\lib に格納します。

```
rtaosgen --include:PathToAutosarHeaderFiles
   --output:[*.h]C:\working\OS\inc
   --output:[*.lib,*.sig]C:\working\OS\inc Config.arxml
```

12.3 ターゲットオプション

ターゲットオプションは、--target_option コマンドラインオプションを使用して設定することができ、AUTOSAR XML コンフィギュレーションに保存することもできます。

12.3.1 C スタートアップに使用されるスタック

XML名 SpPreStartOS

説明

StartOS()が呼び出される時点ですでに使用されている スタックの量です。この値は、ランタイムにおいて OS がすべてのタスクと割り込みをサポートするのに必要なスタックサイズに加算されます。一般的にこのオプションは、リンカがアロケートしなければならないスタック量を調べるためのものです。OS コンフィギュレーションの内容が変わっても、通常、この値は変わりません。

12.3.2 アイドル時に使用されるスタック

XML名 SpStartOS

説明

OSがアイドル状態のときに(一般的にはOs_Cbk_Idle()内で)使用されるスタックの量です。これは、Os_StartOS()が呼び出される時点で使用されるスタック量とタスクや割り込みが 1 つも実行されていないときに使用されるスタック量の差です。そのため、Os_Cbk_Idle()が読み出されていない場合はゼロになる可能性があります。これにはアイドル状態のときに呼び出されるすべての関数が使用するスタック量が含まれます。OS コンフィギュレーションの内容が変わっても、通常、この値は変わりません。

12.3.3 ISR 起動のためのスタックオーバーヘッド

XML名 SpIDisp

説明

タスクを ISR 内から起動する際に余分に必要となるスタックの量です。あるタスクがカテゴリ 2 ISR 内で起動され、そのタスクの優先度がそのときに実行中のタスクよりも高い場合、一部のターゲットでは、実行中のタスクよりも優先度の低いタスクを起動する場合に比べて、スタックの使用量が若干多くなります。 この値はその増加分を表すもので、ワーストケースのスタックサイズ計算に使用されますが、ほとんどのターゲットではゼロになります。この値は、OS コンフィギュレーションの内容（たとえば STANDARD/EXTENDED や SC1/2/3/4 など）が著しく変わった場合は、変わる可能性があります。

12.3.4 ECC タスクのためのスタックオーバーヘッド

XML名 SpECC

説明

ECC タスクを開始するために余分に必要となるスタックの量です。ECC タスクは、開始されるときに BCC タスクに比べて若干多くの状態をスタックに保存する必要があり、この値にはその差を表すものです。この値は、OS コンフィギュレーションの内容（たとえば STANDARD/EXTENDED や SC1/2/3/4 など）が著しく変わった場合は、変わる可能性があります。

12.3.5 ISR のためのスタックオーバーヘッド

XML名 SpPreemption

説明

カテゴリ 2 ISR を処理するために使用されるスタックの量です。カテゴリ 2 ISR は、タスクに割り込みをかける際に、通常は何らかのデータをスタックに格納します。その ISR がスタックサイズを読み取って、プリエンプトされたタスクがそのスタックバジェットを超えたかどうかを判断する場合は、読み取られたサイズからこの値を差し引かないと、スタック使用量が多めに推定されてしまいます。またこの値は、システムのワーストケースのスタック使用量を算出するときにも使用されます。この値は正確に設定するよう注意してください。この値が大きすぎると、減算で 32 ビットのアンダーフローが発生し、バジェットオーバーランが検出されたと OS が判断してしまう可能性があります。この値は、OS コンフィギュレーションの内容（たとえば STANDARD/EXTENDED や SC1/2/3/4 など）が著しく変わった場合は、変わる可能性があります。

12.3.6 ORTI22

XML名 Orti22

説明

Lauterbach デバッガ用の ORTI の生成を選択します。

Settings

値	説明
TRUE	ORTI を生成します。
FALSE	ORTI なし。

12.3.7 仮想コアの数

XML名 CoreCount

説明

作成するコアの数を定義します。デフォルトは1です。

12.3.8 マルチコア割り込み

XML名 MC_Interrupt

説明

マルチコアへ実装用の割り込みを予約します。

12.3.9 ランタイムライセンス名

XML名 LicName

説明

ランタイムライセンスの名前をオーバーライドします。デフォルトは LD_RTA-OS_VRTA です。

12.4 OS Options（OS オプション）

OS オプションは、 `--os_option` コマンドラインオプションを使用して設定することができ、AUTOSAR XML コンフィギュレーションに保存することもできます。

12.4.1 Optimize for core-local memory（コアのローカルメモリの最適化）

XML名 core_local

説明

TRUEにすると、マルチコアアプリケーションにおいてコア固有のデータを各プロセッサコアのローカル領域に配置できるように、各コア用データを別々のMemMapセクションに配置します。

12.4.2 Fast Terminate（高速ターミネート）

XML名 lightweight

説明

ChainTask()と TerminateTask()がタスクのエントリ関数内でのみ使用されること、およびそれらの戻り値がチェックされないことを宣言します。これにより、OSはタスクのターミネーションを最適化できます。

12.4.3 Disallow upwards activation（上位タスクの起動の不許可）

XML名 no_upward_activation

説明

ActivateTask または SetEvent を使用して自分より優先度の高いタスクを起動するタスクがないことを宣言します。これにより、OS はタスクの起動にかかる時間を最適化できます。

12.4.4 Disallow ChainTask()（ChainTask()の不許可）

XML名 no_chain

説明

ChainTask が使用されないことを宣言します。これにより、OS はタスク切り替え時間を最適化できます。

12.4.5 Disallow Schedule()（Schedule()の不許可）

XML名 no_schedule

説明

Schedule()が使用されないことを宣言します。これにより、システムのワーストケースのスタックサイズを小さくすることができます。

12.4.6 Optimize Schedule()（Schedule()の最適化）

XML名 optimize_schedule

説明

Schedule()呼び出しの関数呼び出しオーバーヘッドを極力小さくします。

12.4.7 Allow STANDARD Status in SC3/SC4（SC3/SC4 における標準ステータスの許可）

XML名 std_sc34_checks

説明

スケーラビリティクラス 3 または 4 で、エラーチェックによるオーバーヘッドを少なくするために標準（STANDARD）ステータスを選択できるようにします。通常、AUTOSAR ではこれは許可されていません。

12.4.8 Allow Alarm Callbacks in SC3/SC3/SC4（SC2/SC3/SC4 におけるアラームコールバックの許可）

XML名 sc234_alarmcallbacks

説明

どのスケーラビリティクラスでもアラームコールバックを行えるようにします。通常、AUTOSAR ではこれは SC1 でのみ許可されています。

12.4.9 Omit activation checks for WAITING state（待ち状態における起動チェックの省略）

XML名　no_ecc_activate_in_wait

説明

通常、ECC タスクは 1 回だけ起動され、ターミネートしません。そのため、RTA-OS はタスクが待ち（WAITING）状態になっているかどうかを調べるランタイムチェックを省略することにより、タスク起動コードを最適化することができます。デフォルトは TRUE です。

12.4.10 Timing Protection Interrupt（タイミング保護割り込み）

XML名　timing_interrupt

説明

これを TRUE にすると、カテゴリ 1 割り込みを使用してタイミング保護を実施することができます。これが TRUE になっていてタイミング保護が有効になっている場合は、ユーザーは Os_Cbk_SetTimeLimit コールバック関数と Os_Cbk_SuspendTimeLimit コールバック関数に対応し、割り込みがサスペンド状態にならずに発行された際には Os_TimingFaultDetected を呼び出す必要があります。各 OS コアごと専用のカテゴリ 1 タイミング保護割り込みが必要で、ユーザーはこの割り込みの作成と設定を行う必要があります。

12.4.11 Omit Timing Protection（タイミング保護の省略）

XML名　no_timing_protection

説明

Timing Protection（タイミング保護）の値が設定されているタスク／ISR が 1 つでもある場合は、OS にタイミング保護機能が含まれます。このオプションを使用すると、タイミング保護に関するすべてのチェックを省略できます。

12.4.12 Add Function Protection（関数保護の追加）

XML名　function_protection

説明

実行リミットを監視してメモリ保護例外から回復できるようにする機能を OS アプリケーション関数内に追加します。CallAndProtectFunction という API 関数 は CallTrustedFunction の機能を拡張したもので、実行のタイムリミットが追加され、関数をターミネートすることができます。

12.4.13 Omit Memory Protection（メモリ保護の省略）

XML名　no_memory_protection

説明

アントラステッド OS アプリケーションが 1 つでもある場合は、OS にメモリ保護機能が含まれています。このオプションを使用すると、すべてのメモリ保護機能を省略できます。

12.4.14 Untrusted code can read OS data（アントラステッドコードが OS データを読み取り可能）

XML名　protection_allows_os_reads

説明

メモリ保護が有効になっている場合は、アントラステッドコードは OS 変数を読み取ることができないものとされるため、すべての OS サービスコールはプロセッサを保護モードに切り替えます。ただし、ユーザーのメモリ保護ストラテジが OS データに対する読み取りアクセスを許容している場合は、 OS データの読み取りしか行わない API 関数を最適化し、このモード切り替えを行わないようにすることができます。

12.4.15 Single Memory Protection Zone（シングルメモリ保護ゾーン）

XML名　single_zone_protection

説明

すべてのアントラステッドコードを 1 セットの MPU 値と共に実行できる場合は、OS を起動する前にそれらの MPU 値をセットアップすることができるので、Os_Cbk_SetMemoryAccess を呼び出す必要がなくなります。このオプションを使用すると、Os_Cbk_SetMemoryAccess の呼び出しを省略できます。

12.4.16 Stack Only Memory Protection（スタックのみのメモリ保護）

XML名　stack_only_protection

説明

すべてのアントラステッドコードを同じ基本メモリ保護設定で実行し、スタックについてのみメモリ保護を適用したい場合は、OS が行うべきことはスタック関連のフィールド（アドレスとサイズ）とアプリケーションを Os_Cbk_SetMemoryAccess に渡すことだけです。このオプションはこのコールバック関数から Task、ISR、Function の値を削除します。

12.4.17 Omit Service Protection in SC3/SC4（SC3/SC4 におけるサービス保護の省略）

XML名　no_service_protection

説明

AUTOSAR は、SC3 と SC4 ではサービス保護チェックを追加適用するよう要求していますが、このオプションを使用すると、これらのチェックを省略できます。

12.4.18 Omit TerminateApplication（TerminateApplication の省略）

XML名　no_terminate_application

説明

通常、OS アプリケーションが 1 つでも存在する場合は TerminateApplication という API 関数がサポートされますが、このオプションを使用すると、このサポートを省略してシステムのオーバーヘッドを少なくすることができます。

12.4.19 Only Terminate Untrusted Applications（アントラステッドアプリケーションのみをターミネート）

XML名 only_terminate_untrusted

説明

タイミング保護違反がある場合、または TerminateApplication()が呼び出された場合は、デフォルトではトラステッド OS アプリケーションがターミネートされる可能性があります。このオプションは、トラステッドコードをターミネートしないように OS に指示するものです。これによりランタイムを節約できる可能性があります。

12.4.20 Enable Time Monitoring（時間監視の有効化）

XML名 meter_execution

説明

タスクとカテゴリ 2 ISR の実行時間を記録します。

12.4.21 Enable Elapsed Time Recording（実行時間の記録を有効化）

XML名 meter_elapsed_time

説明

タスク、カテゴリ 2 ISR、Idle の合計実行時間を積算します。これには Time Recording をイネーブルにしておく必要があります。

12.4.22 Enable Activation Monitoring（起動監視の有効化）

XML名 monitor_activations

説明

タスクの起動時間を記録します。

12.4.23 Support Delayed Task Execution（タスク起動遅延のサポート）

XML名 delayed_tasks

説明

Os_SetDelayedTasks、Os_AddDelayedTasks、Os_RemoveDelayedTasks という API 関数のサポートを追加します。

12.4.24 Collect OS usage metrics（OS 使用状態の計測）

XML名 metrics

説明

OS の諸機能が使用される頻度に関するランタイム情報を収集します。収集されるデータの種類については Os_Metrics.h を参照してください。

12.4.25 Additional Task Hooks（タスクフックの追加）

XML名　task_lifetime_hooks

説明

Os_Cbk_TaskStart フックと Os_Cbk_TaskEnd フックのサポートを追加します。これらのフックはプリタスクフックとポストタスクフックのように機能しますが、タスクがプリエンプトされる際には呼び出されません。

12.4.26 Task Activation Hook（タスク起動フック）

XML名　task_activation_hooks

説明

Os_Cbk_TaskActivated フックのサポートを追加します。このフックは、タスクの起動が成功するたびに呼び出されます。

12.4.27 Additional ISR Hooks（ISR 開始／終了時のフックサポート追加）

XML名　isr_lifetime_hooks

説明

Os_Cbk_ISRStart、Os_Cbk_ISREnd、Os_Cbk_CrosscoreISRStart、Os_Cbk_CrosscoreISREnd というフックサポートを追加します。これらのフックは、カテゴリ 2 ISR またはクロスコア ISR が開始／終了するときに呼び出されます。その ISR がプリエンプトされた際にはこれらのフックは呼び出されません。

12.4.28 Provide spinlock statistics（スピンロック統計情報の提供）

XML名　spinlock_info

説明

API 関数 GetSpinlockInfo のサポートを追加します。スピンロックに関する実行時統計情報が記録され、それらにアクセスするための GetSpinlockInfo がサポートされます。

12.4.29 Force spinlock error checks（スピンロックチェックの強制実行）

XML名　spinlock_std_check

説明

スピンロック API 関数は、標準（STANDARD）ステータスのビルドではエラーチェックを行いませんが、このオプションを使用すると、拡張（EXTENDED）ステータスと標準（STAN- DARD）の両方のビルドでスピンロックチェックを強制的に実行させることができます。

12.4.30 Add Spinlock APIs for CAT1 ISRs（カテゴリ 1 ISR 用スピンロック API 関数の追加）

XML名　unchecked_spinlocks

説明

AUTOSAR のスピンロック API 関数は、タスクまたはカテゴリ 2　ISR からしか呼び出されません。このオプションを使用すると、UncheckedGetSpinlock、UncheckedTryToGetSpinlock、UncheckedReleaseSpinlock という新しい 3 つの API 関数が生成され、カテゴリ 1 ISR から呼び出すことができます。ただしエラーチェックの範囲は大幅に制限されるので、注意が必要です。

12.4.31 Stack Sampling（スタックのサンプリング）

XML名　stack_sampling

説明

Os_Cbk_CheckStackDepth フックのサポートを追加します。このフックは、タスクまたはカテゴリ 2 ISR の処理開始時に呼び出され、API 関数 Os_GetStackUsage からも呼び出されます。

12.4.32 MemMap level（メモリマップレベル）

XML名　memmap_level

説明

AUTOSAR MemMap インクルードメカニズムを OS コードにどのように適用するかを選択します。

設定

値	説明
NONE **DECLARATION** **FULL**	MemMap インクルードを行いません。 MemMap インクルードを宣言の近くで使用します。 MemMap インクルードを宣言と定義/extern 宣言について使用します。（デフォルト）

13 出力ファイルのフォーマット

13.1 RTA-TRACE コンフィギュレーションファイル

RTA-TRACE が有効になっている場合、RTA-OS は RTA-TRACE コンフィギュレーションファイルを生成します。このファイルのフォーマットは ORTI フォーマットと似ています。このフォーマットについては『RTA-TRACE OS Instrumenting Kit Manual』に詳しく記載されています。

13.2 ORTI ファイル

本項では、RTA-OS が出力する ORTI オブジェクトについて説明します。

ポートが ORTI 出力をサポートしていて、ORTI の生成が設定されている場合は、 **rtaosgen** を使用してカーネルがビルドされる際に RTAOS.orti というファイルが生成されます。

ORTI オブジェクトは、RTA-OS 内の各 OS オブジェクト（タスク、ISR、アラームなど）に関する情報をカプセル化したものです。アプリケーションには各 ORTI オブジェクトのインスタンスがまったく含まれていないか、または 1 つ以上含まれていて、各オブジェクトの名前は一意です。各 ORTI オブジェクトには多くの属性があり、各属性には 1 つの値があります。

たとえば、OS には、現在実行中のタスクを示す RUNNINGTASK という属性があります。

以降の項では、生成される ORTI オブジェクトを紹介します。各項は以下の構成になっています。

オブジェクト

ORTI オブジェクトの名前

説明

ORTI オブジェクトについての説明

属性

ORTI オブジェクトの属性

属性	説明
Attribute Name	*ORTI ファイル内の属性記述* - 属性の説明

表の各行には、属性の名前とその属性についての簡単な説明が記載されています。各属性の名前は「属性」列に記載されています。vs_という接頭辞で始まる名前の属性は RTA-OS をサポートするために追加されたものであり、標準の ORTI 属性ではありません。ユーザーのデバッガが ORTI 規格にどの程度準拠しているかにより、これらの属性を表示できない場合があります。

デバッガによっては、属性名の代わりに、ORTI ファイル内に存在している属性記述を表示するものがあります。RTA-OS で使用される記述は、説明列の冒頭に引用符で囲んで記載されています。

13.2.1 OS

オブジェクト

OS

説明

1 個の AUTOSAR コアにつき 1 個 の OS オブジェクトがあります。このオブジェクトの名前は RTA-OS プロジェクト名から導出されます。

属性

属性	説明
RUNNINGTASK	*Running task* - 実行中のタスクの名前。ISR がタスクに割り込みをかけると、その ISR が実行されている間は、この属性には割り込みをかけられたタスクの名前が示されます。
RUNNINGTASKPRIORITY	*Running task priority* - 実行中のタスクの現在の優先度を、OIL ファイル内の記述と同じ用語を使用して示したもの。RUNNINGTASKPRIORITY は、タスクと ISR により共有されるリソースをロックする効果を示すものではありません。
RUNNINGISR2	*Running cat 2 ISR* - この属性の値は、実行中のカテゴリ 2 ISR が存在する場合は、その名前です。実行中のカテゴリ 2 ISR がない場合は NO_ISR が示されます。
SERVICETRACE	*OS Services Watch* - サービスルーチン（RTA- OS コンポーネント API 関数）の入口または出口とルーチン名を示します。デバッガによっては、この属性を特別なトレース属性として認識して診断を行えたり、どの API 関数が最後に実行開始または終了したかを表示できるものがあります。
LASTERROR	*Last OSEK error* - 発生した最後のエラーの名前を表示します。初期値は E_OK です。
CURRENTAPPMODE	*Current AppMode* - 現在のアプリケーションモードを、XML ファイルに記載されている名前を使用して示します。このアプリケーションモードが XML ファイル内の値に準拠していない場合は、*unknown AppMode* という値が示されます。

13.2.2 タスク

オブジェクト

TASK

説明

コンフィギュレーションファイル内のタスク宣言に応じて生成されます。

属性

属性	説明
STATE	*State* - タスクの状態（SUSPENDED、RUNNING、READY、WAITING のうちのいずれか）
vs_BASEPRIORITY	*Base priority* - タスクの基本優先度（当該タスクについて OIL ファイル内に定義されている優先度）
PRIORITY	*Dispatch priority* - タスクのディスパッチ優先度。ディスパッチ優先度は、タスクが実行開始するときの優先度です。内部リソースが使用される場合や、このタスクがノンプリエンプティブである場合は、ディスパッチ優先度を基本優先度より高くすることができます。
CURRENTACTIVATIONS	*Activations* - 最大許容起動数

13.2.3 カテゴリ 1 ISR

カテゴリ 1 ISR 用に生成される ORTI オブジェクトはありません。

13.2.4 カテゴリ 2 ISR

カテゴリ 2 ISR 用に生成される ORTI オブジェクトはありません。

13.2.5 リソース

オブジェクト

RESOURCE

説明

コンフィギュレーションファイル内のリソース宣言に応じて生成されます。

属性

属性	説明
PRIORITY	*Ceiling Priority* - 上限が定義されたタスク優先度に基づくリソースの上限優先度
LOCKER	*Resource locker* - リソースの現在のホルダ（保持者）
STATE	*Resource State* - リソースの状態（ロックされているかどうか）

13.2.6 イベント

イベント用に生成される ORTI オブジェクトはありません。

13.2.7 カウンタ

カウンタ用に生成される ORTI オブジェクトはありません。

13.2.8 アラーム

オブジェクト

ALARM

説明

コンフィギュレーションファイル内のアラーム宣言に応じてしか生成されません。

属性

属性	説明
ALARMTIME	*Alarm Time* - アラームが次に満了する時刻。現在のカウント値を確かめるには、COUNTER オブジェクト内の Count 値を参照してください。
CYCLETIME	*Cycle Time* - サイクリックアラーム用の周期。シングルショットアラームの場合はゼロになります。
ACTION	*Action* - アラーム満了時に実行される処理。以下のような処理が考えられます。 • タスクを起動する • イベントをセットする • コールバック関数を実行する
STATE	*Alarm state* - アラームが実行されているかどうかを示します。RUNNING か STOPPED の値を取ります。
COUNTER	*Counter* - このアラームがアタッチされているカウンタの名前

13.2.9 スケジュールテーブル

オブジェクト

SCHEDULETABLE

説明

コンフィギュレーションファイル内にスケジュールテーブルが宣言されている場合にのみ生成されます。

属性

属性	説明
COUNTER	*Counter* - スケジュールテーブルがアタッチされているカウンタの名前
STATE	*State* - スケジュールテーブルの状態
EXPIRYTIME	*Expiry Time* - 次の満了ポイントが処理される時点のチック値
NEXT	*Next table* - 次のスケジュールテーブル（セットされている場合のみ）

14 互換性と移植

本章では、RTA-OS への移植に役立つように、RTA-OS と他の ETAS ツール環境の適合性に関する情報を提供し、RTA-OS と以前の RTA-OSEK シリーズのオペレーティングシステムの主な相違点について概説します。

14.1 ETAS ツール

下の表は、RTA-OS と他の ETAS ソフトウェアツールの互換性の概要です。互換性は、コンフィギュレーション言語（Config）と API 関数の使用（API）という 2 つの要素に分かれています。以下の記号が使用されています。

✓ 完全互換
✓ 一部互換（詳細は下の注記を参照してください）
✗ 非互換

より詳しい情報が必要な場合は、ETAS のサポート窓口までお問い合わせください。

ETAS ツール	バージョン	互換性		注記
		Config	API	
ERCOSEK	4.x	✗	✓	1
RTA-OSEK	4.x	✗	✓	2
	5.x	✗	✓	2
RTA-TRACE	2.x	✓	✓	3
RTA-RTE2.x	1.x	✗	✓	4
ASCET	4.x	✗	✗	--
	5.x	✗	✓	--
	6.x	✗	✓	--

ETAS ツールの互換性に関する注記

1. OSEK API 関数は移植可能です。ただし、AUTOSAR OS では以下のように挙動が若干変わります。
   - StartOS()はリターンしません。
   - SetRelAlarm()のオフセットパラメータとしてゼロを使用することはできません。
2. 14.2 項を参照してください。
3. RTA-OS のすべての内部名については、基本ソフトウェアモジュール内のネームスペースに関する AUTOSAR ガイドラインが採用されています。一方、AUTOSAR OS API 関数、マクロ、型定義の外部名は、AUTOSAR 規格に対応するため、外部 API 関数内に提供されています。

   RTA-OS の機能のうち AUTOSAR OS に含まれていないものについては、内部名にも外部名にも以下のような AUTOSAR 命名規則が採用されています。
   - 変数、API 関数、定数の名前には Os_という接頭辞が付きます。
   - マクロ名には OS_という接頭辞が付きます。

   AUTOSAR の命名規則は RTA-TRACE 用コードにも適用されています。これは RTA-TRACE の挙動を変えるものではなく、ユーザーに対してトランスペアレントです

が、RTA-TRACE の付属資料には RTA-OS に使用されている API 関数や型の名前が正確に反映されてはいません。しかし、資料に記載されている名前を以下のように変えるだけで、RTA-OS で生成される名前になります。

- すべての API 関数、型、変数の名前には Os_という接頭辞が付きます。たとえば、

  LogTracepoint(MyTracepoint)

  という名前は以下のようになります。

  Os_LogTracepoint(MyTracepoint)

- すべてのマクロの名前には OS_という接頭辞が付きます。たとえば、

  TRACE_ERRORS_CLASS

  という名前は以下のようになります。

  OS_TRACE_ERRORS_CLASS

4. RTA-RTE2.x は、AUTOSAR OS R2.x には互換であっても AUTOSAR OS R3.x には非互換の OIL で OS コンフィギュレーションを生成します。RTA-RTE2.x によって生成されるほとんどの OS 呼び出しは、AUTOSAR OS に互換です。AUTOSAR OS R1.0 と統合する際には、RTE ライブラリが `StartScheduleTable()`という API 関数を使用しますが、AUTOSAR OS と共に使用するには、これを `StartScheduleTableRel()` に置き換える必要があります。しかし、その代わりに OSENV_UNSUPPORTED を定義することにより、RTE への完全適合が可能です。詳細については、『*RTA-RTE2x Toolchain Integration Guide* 』を参照してください。

14.2 API 関数の適合性

次の表は RTA-OS、AUTOSAR、RTA-OSEK、OSEK ファミリのオペレーティングシステムの互換性を示しています。AUTOSAR のマルチコアオペレーション用に追加された API 関数は、従来の規格には存在していなかったので、この表には含まれていません。

API 関数	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	参照先
ActivateTask	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ActivateTaskset		✓		✓						14.2.1
AdvanceSchedule		✓		✓						14.2.3
AssignTaskset		✓		✓						14.2.1
CallTrustedFunction							✓	✓	✓	
CancelAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ChainTask	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ChainTaskset		✓		✓						14.2.1
CheckISRMemoryAccess							✓	✓	✓	
CheckObjectAccess							✓	✓	✓	
CheckObjectOwnership							✓	✓	✓	
CheckTaskMemoryAccess							✓	✓	✓	
ClearEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	
CloseCOM	✓	✓	✓	✓						14.2.4
DisableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
EnableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetActiveApplicationMode	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetAlarmBase	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetApplicationID							✓	✓	✓	
GetArrivalpointDelay		✓		✓						14.2.3
GetArrivalpointNext		✓		✓						14.2.3
GetArrivalpointTasksetRef		✓		✓						14.2.3
GetCounterValue		✓		✓	✓	✓	✓	✓	✓	
GetElapsedCounterValue					✓	✓	✓	✓	✓	
GetEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetExecutionTime		✓		✓						14.2.2
GetISRID			✓	✓	✓	✓	✓	✓	✓	
GetLargestExecutionTime		✓		✓						14.2.2
GetMessageResource		✓	✓	✓						14.2.4
GetMessageStatus	✓	✓	✓	✓						14.2.4
GetResource	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetScheduleNext	✓	✓		✓						14.2.3
GetScheduleStatus		✓		✓						14.2.3
GetScheduleTableStatus		✓	✓	✓	✓	✓	✓	✓	✓	
GetScheduleValue		✓		✓						14.2.3
GetStackOffset		✓		✓						14.2.11
GetTaskID	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetTasksetRef		✓		✓						14.2.1
GetTaskState	✓	✓	✓	✓	✓	✓	✓	✓	✓	
IncrementCounter			✓	✓	✓	✓	✓	✓	✓	

API call	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	参照先
InitCOM	✓	✓	✓	✓						14.2.4
InitCounter				✓						
MergeTaskset		✓		✓						14.2.1
NextScheduleTable			✓	✓	✓	✓	✓	✓	✓	
Os_AdvanceCounter									✓	
Os_GetExecutionTime									✓	14.2.2
Os_GetISRMaxExecutionTime									✓	14.2.2
Os_GetISRMaxStackUsage									✓	14.2.11
Os_GetStackUsage									✓	14.2.11
Os_GetStackValue									✓	14.2.11
Os_GetTaskMaxExecutionTime									✓	14.2.2
Os_GetTaskMaxStackUsage									✓	14.2.11
Os_ResetISRMaxExecutionTime									✓	14.2.2
Os_ResetISRMaxStackUsage									✓	14.2.11
Os_ResetTaskMaxExecutionTime									✓	14.2.2
Os_ResetTaskMaxStackUsage									✓	14.2.11
Os_Restart									✓	
Os_SetRestartPoint									✓	
osAdvanceCounter				✓						0
osResetOS				✓						14.2.12
ReadFlag	✓	✓	✓	✓						14.2.4
ReceiveMessage	✓	✓	✓	✓						14.2.4
ReleaseMessageResource	✓	✓	✓	✓						14.2.4
ReleaseResource	✓	✓	✓	✓	✓	✓	✓	✓	✓	
RemoveTaskset		✓		✓						14.2.1
ResetFlag	✓	✓	✓	✓						14.2.4
ResetLargestExecutionTime		✓		✓						14.2.2
ResumeAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ResumeOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Schedule	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SendMessage	✓	✓	✓	✓						14.2.4
SetAbsAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SetArrivalpointDelay		✓		✓						14.2.3
SetArrivalpointNext		✓		✓						14.2.3
SetEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SetRelAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	14.2.8
SetScheduleNext		✓		✓						
SetScheduleTableAsync						✓		✓	✓	
ShutdownOS	✓	✓	✓	✓	✓	✓	✓	✓	✓	14.2.6
StartCOM	✓	✓	✓	✓						14.2.4
StartOS	✓	✓	✓	✓	✓	✓	✓	✓	✓	14.2.5
StartSchedule		✓		✓						14.2.3

API call	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	参照先
StartScheduleTable			✓	✓						14.2.9
StartScheduleTableAbs					✓	✓	✓	✓	✓	
StartScheduleTableRel					✓	✓	✓	✓	✓	
StartScheduleTableSynchron						✓		✓	✓	
StopCOM	✓	✓	✓	✓						14.2.4
StopSchedule		✓		✓						14.2.3
StopScheduleTable			✓	✓	✓	✓	✓	✓	✓	
SuspendAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SuspendOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SyncScheduleTable						✓		✓	✓	
TerminateApplication							✓	✓	✓	
TerminateTask	✓	✓	✓	✓	✓	✓	✓	✓	✓	
TestArrivalpointWriteable		✓		✓						14.2.3
TestEquivalentTaskset		✓		✓						14.2.1
TestSubTaskset		✓		✓						14.2.1
Tick_<CounterID>		✓		✓						14.2.10
TickSchedule		✓		✓						14.2.3
WaitEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	

14.2.1 Tasksets（タスクセット）

RTA-OSEK のタスクセット API 関数 Tasksets は、RTA-OS から削除されました。この機能は、複数の `ActivateTask()` 呼び出しを順に実行することにより実現できます。ただし、ランタイムの挙動が Tasksets と同じになるのは、起動される一連のタスクの優先度が呼び出し元のタスクの優先度と同じかそれより低い場合に限られます。

14.2.2 Time Monitoring（時間監視）

RTA-OSEK のタイミングビルドに代わって、RTA-OS ではコンフィギュレーションオプション 'Time Monitoring'（時間監視）が提供されています。つまり、時間監視用のコードをインクルードしなくても拡張ステータスを使用できるようになりました。

RTA-OS では、API 関数の名前に Os_という接頭辞が付きますが、挙動は以前の RTA-OSEK のものとまったく同じです。ただし、GetLargestExecutionTime と ResetLargestExecutionTime については、タスク専用と ISR 専用に分かれて新しい API 関数に変わりました。

- Os_GetLargestExecutionTime は Os_Get[Task|ISR]MaxExecutionTime に変わりました。
- Os_ResetLargestExecutionTime は Os_Reset[Task|ISR]MaxExecutionTime に変わりました。

14.2.3 スケジュール

RTA-OS では、RTA-OSEK のスケジュールメカニズムの代わりに AUTOSAR のスケジュールテーブルメカニズムを使用しているため、ランタイムにスケジュールを変更することはできません。この機能は AUTOSAR OS でサポートされていません。ランタイムに変更が必要な場合は、代わりにアラームを使用してください。

14.2.4 OSEK COM

OSEK COM が使用されない場合、OSEK OS では、OS が OSEK COM の諸機能を提供するケースがあります。AUTOSAR OS では、アプリケーション用の内部通信機能は AUTOSAR RTE により提供されるようになったので、この機能は削除されました。

14.2.5 StartOS()の挙動

StartOS()はリターンは OSEK OS では認められていて、RTA-OSEK はその挙動を提供しています。一方、AUTOSAR OS では StartOS()からのリターンは禁止されていて、RTA-OS はそちらの挙動を提供しています。そのため RTA-OS では、StartOS()の後にアイドルループを配置してアイドルメカニズムとして使用することができません。RTA-OS の場合、実行されるタスクや ISR がひとつもないときは、デフォルトではカーネルがビジーウェイト状態になります。その代わりにユーザー独自のアイドル（バックグラウンドタスク）機能が必要な場合は、Os_Cbk_Idle()という関数で実装してください。

14.2.6 ShutdownOS()の挙動

ShutdownOS()からのリターンは OSEK OS では認められていますが、AUTOSAR OS では禁止されています。RTA-OSEK では常にこの AUTOSAR OS の挙動を採用していましたが、別の OS から移植する場合は、アプリケーションの変更が必要になる場合があります。

14.2.7 ハードウェアカウンタドライバ

RTA-OSEK の osAdvanceCounter()というハードウェアカウンタドライバの API 関数は、RTA-OS では Os_AdvanceCounter という名前に変わり、挙動も変更されました。RTA-OSEK では、ユーザーが次の満了をセットアップできるようにカウンタのステータスを返しましたが、RTA-OS においては、最初のセットアップについては内部的に（ユーザー提供の Os_Cbk_Set_CounterID API コールバック関数により）行われるので、その後、アプリケーションコードが Os_Cbk_State_CounterID を呼び出して満了があるかどうかをチェックする必要があります。

14.2.8 SetRelAlarm()でのゼロの禁止

SetRelAlarm(_, 0, _)の使用は、OSEK OS では認められていますが AUTOSAR OS では禁止されています。

14.2.9 スケジュールテーブル API の改変

AUTOSAR OS 規格では、スケジュールテーブルを起動する API 関数が変更され、絶対起動と相対起動のコンセプトが OSEK OS アラームのものと同じになりました。
StartScheduleTable()という API 関数は規格から削除され、代わりに AUTOSAR R3.0 では StartScheduleTable[Rel|Abs]が提供されるようになりました。
StartScheduleTable(Tbl,At)の挙動を再現する必要がある場合は、
StartScheduleTableRel(Tbl,At)を使用してください。

14.2.10 ソフトウェアカウンタドライバ

RTA-OSEK の API 関数 Tick_CounterID()は、CounterID をパラメータとして取る AUTOSAR 規格の IncrementCounter()に置き換わりました。ただし、この API 関数の静的バージョンがもたらすパフォーマンス向上を再現するため、RTA-OS には AUTOSAR の API 関数の静的バージョンである IncrementCounter_CounterID()が含まれています。この API 関数の挙動は Tick_CounterID()と同じです。

14.2.11 スタック監視

スタック測定の挙動は、RTA-OSEK と RTA-OS とで異なります。RTA-OSEK のスタック測定はスタックのベースアドレスから GetStackOffset()を使用して行われます。通例では、スタックのベースアドレスは、リンク時に SP_INIT というラベルによって RTA-OSEK に提供されていました。

RTA-OS では GetStackOffset()の代わりに Os_GetStackValue()を使用します。

RTA-OSEK では、各タスク／ISR が使用するスタックの量をユーザーが計算する必要がありました。RTA-OS では、これと同じ方法も可能ですが、新しく追加された Os_GetStackUsage()という API 関数は、呼び出された時点において呼び出し元のタスク／ISR だけによって使用されているスタックの量を返すので、ユーザーがスタック計算を行う必要がなくなりました。

また RTA-OS は、コンテキスト切り替え（または Os_GetStackUsage()の呼び出し）が行われるときにも各タスク／ISR のワーストケースの実際のスタック使用量を記録します。さらに、各タスク／ISR について測定されたスタック使用量の最大値を取得する API 関数と、測定された最大値をリセットする API 関数も追加されました。

このモデルは RTA-OS が提供する時間監視機能とよく似ています。

14.2.12 OS の再起動

ランタイムに OS を再起動する手段は OSEK OS にも AUTOSAR OS にも用意されていませんが、これは一般的に必要とされる機能なので、RTA-OSEK は osResetOS()という API 関数を提供して再起動を行えるようにしています。

RTA-OSでは、これに代わる汎用の再起動メカニズムを提供しています。StartOS()を呼び出す前の任意の場所にOs_SetRestartPointというAPI関数の呼び出しを配置することにより、再起動ポイントをセットすることができ、StartOS()の前に任意のハードウェアの再初期化を行うことができます。そして、配置されたマーカーまでジャンプする Os_Restart を呼び出せば、再起動を実現できます。

15 お問い合わせ先

15.1 技術サポート

有効なサポート契約をお持ちのユーザーの方はETASの技術サポートをご利用いただけます。有効なサポート契約をお持ちでない場合は、お近くの営業窓口（15.2.2 項を参照）までお問い合わせください。

技術サポートに関するお問い合わせは、以下の RTA ホットライン窓口まで E-mail またはお電話でご連絡ください。

＜ETAS 株式会社 RTA ホットライン＞

E-Mail: RTA_Hotline.ETAS_MB_YH@etas.com
電話： 045-222-0953
受付時間： 平日 9:00～18:00

なお、ご連絡いただく際に以下のような情報やファイルをご提供いただきますと、より速やかな問題解決につながります。

- サポート契約番号
- ご使用の各種ファイル（.xml、.arxml、.rtaos、.stc など）
- エラーが発生したコマンドラインの内容
- ご使用の ETAS ツールのバージョン
- ご使用のコンパイラツールチェーンのバージョン
- エラーメッセージなどが出力された場合は、その内容
- 診断ファイル（Diagnostic.dmp）が生成された場合は、そのファイル

15.2 その他のお問い合わせ先

15.2.1 ETAS 本社

ETAS GmbH
Borsigstrasse 14 Phone: +49 711 3423-0
70469 Stuttgart Fax: +49 711 3423-2106
Germany WWW: http://www.etas.com

15.2.2 その他

上記以外のお問い合わせ先につきましては、ETAS ホームページをご覧ください。

各国支社	www.etas.com/ja/contact.php
技術サポート	www.etas.com/ja/hotlines.php

索引