RTA-OSEK

リファレンスガイド

著作権について

Version：RTA-OSEK V5.0.2

免責事項

本書の内容は、予告なく変更されることがあり、LiveDevices のいかなる部門からのお約束を表明するものでもありません。本書の情報は正確であると想定されておりますが、LiveDevices はいかなる誤りや記載漏れに対する責任も負わないものとします。

いかなる場合も、LiveDevices とその社員、受託業者あるいは本書の著者は、あらゆる性質または種類の損失利益、手数料、または出費に対する、特別、直接的、あるいは間接的な損害、損失、犠牲、負債、料金、要求、請求についての責任も負わないものとします。

商標

RTA-OSEK および LiveDevices は LiveDevices Ltd. の商標です。

Windows および MS-DOS は Microsoft Corp. の商標です。

OSEK/VDX は Siemens AG の商標です。

AUTOSAR は AUTOSAR GdR 商標です。

他のすべての製品名も、各社の商標または登録商標です。

本書に記載されている技術的内容の一部は、以下の特許出願の対象になっています。
UK - 0209479.5 および USA - 10/146,654
UK - 0209800.2 および USA - 10/146,239
UK - 0219936.2 および USA - 10/242,482

目次

1 本書について

本書には RTA-OSEK の技術的な詳細情報のうち、すべてのターゲットハードウェアに共通する部分について記載されています。

各ターゲットに固有な情報は、各ターゲットの『RTA-OSEK バインディングマニュアル』を参照してください。

1.1 本書の対象ユーザー

本書は、RTA-OSEK コンポーネントを使用して、予測された挙動を持つリアルタイムシステムを構築しようとする開発者を対象としています。本書の記述内容は、『RTA ユーザーズガイド』の内容や OSEK と OIL の仕様についてすでに理解していることが前提となります。

1.2 表記上の規約

重要：このように表記されている注記には、ユーザーが知っておく必要のある重要な情報が記載されています。内容をよく読み、記載されているすべての指示に必ず従ってください。

移植性：このように表記されている注記では、RTA-OSEK コンポーネントが実行されるプロセッサ上で実行できるコードを作成する場合に知っておく必要がある事柄について説明しています。

本書では、プログラムコード、ヘッダファイル名、データ型名、C 関数、および RTA-OSEK コンポーネントの API 関数名はすべてクーリエ体（`courier`）で表記されています。またプログラマに公開されているオブジェクトの名前も、やはりクーリエ体で表記されます。たとえば、Task1 という名前のタスクのタスクハンドルは、`Task1` と表記されます。

RTA-OSEK API リファレンスの章では、呼び出し元の環境を示した表の中に  という記号が用いられていますが、この記号は、RTA-OSEK 以外の他の OSEK には移植できない可能性があることを示します。

PDF 文書においては、目次、および他の部分を参照する個所（例：「第 3 章を参照してください」の部分）については、その参照先へのリンクが設けられているので、必要な参照個所を素早く見つけることができます。

1.3 参考情報

OSEK は、車両用エレクトロニクスのオープンなシステムとインターフェースの構築を目標とする、ヨーロッパの自動車工業メーカーによる共同プロジェクトです。OSEK 規格の詳細については、http://www.osek-vdx.org をご覧ください。

AUTOSAR（**AUTO**motive **O**pen **S**ystem **AR**chitecture）は、自動車用ソフトウェアアーキテクチャのオープン規格で、世界の車両メーカー、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。

2 RTA-OSEK の型定義

RTA-OSEK は、以下の 3 つのタイプのオペレーティングシステムの機能をサポートしています。

1. OSEK OS
2. AUTOSAR (SC1) OS
3. RTA-OSEK 固有の機能

下の表に、各 OS で定義されている C のデータ型を示します。「移植性」の列には異所期できる OS タイプが示されています。ここに「OSEK」と示されているものは、AUTOSAR (SC1) OS にも移植可能です。

シンボル	説明	移植性
AccessNameRef	メッセージデータフィールドのアドレス	OSEK
AlarmBaseRefType	AlarmBaseType へのポインタ	OSEK
AlarmBaseType	カウンタのプロパティを格納する構造体（2.1 項を参照してください）	OSEK
AlarmType	アラームオブジェクト	OSEK
AppModeType	アプリケーションモード	OSEK
ArrivalpointConstType	読み取り専用のアライバルポイントオブジェクト	RTA-OSEK
ArrivalpointRefType	ArrivalpointType へのポインタ	RTA-OSEK
ArrivalpointType	アライバルポイントオブジェクト	RTA-OSEK
BooleanRefType	BooleanType へのポインタ	RTA-OSEK
BooleanType	0 は FALSE を表し、0 以外は TRUE を表します。	RTA-OSEK
ByteType	符号なしバイト型	RTA-OSEK
CounterType	カウンタオブジェクト	AUTOSAR
CycleType	プロセッササイクルの単位で測定される実行時間 16 ビット以上の符号なし整数型	RTA-OSEK
EventMaskRefType	EventMaskType へのポインタ	OSEK
EventMaskType	0 個以上のイベントのマスク	OSEK
FlagType	FlagValue へのポインタ	OSEK
FlagValue	フラグオブジェクトには TRUE と FALSE というメンバがあります。	OSEK
ISRType	ISR オブジェクト	AUTOSAR
OSServiceIdType	システムサービスの識別表示を表します。	OSEK
ResourceType	リソースオブジェクト	OSEK
ScheduleStatusRefType	ScheduleStatusType へのポインタ	RTA-OSEK
ScheduleStatusType	RTA-OSEK スケジュールおよび OSEK のアドバンスドカウンタのステータス情報を格納する構造体（2.1 項を参照してください）	RTA-OSEK
ScheduleType	RTA-OSEK のスケジュールオブジェクト	RTA-OSEK
ScheduleTableType	AUTOSAR のスケジュールテーブルオブジェクト	AUTOSAR
ScheduleTableStatusType	AUTOSAR のスケジュールテーブルオブジェクトのステータス	AUTOSAR
ScheduleTableStatusRefType	ScheduleTableStatusType へのポインタ	AUTOSAR
SmallType	メモリ効率のよいデータ型（8 ビット以上の符号なし整数型）	RTA-OSEK
StackOffsetRefType	StackOffsetType へのポインタ	RTA-OSEK
StackOffsetType	すべてのスタックオフセット値（通常は 1 つのみ）を含むデータ型	RTA-OSEK
StatusType	API 関数が出力するステータス情報	OSEK
StopwatchTickRefType	StopwatchTickType へのポインタ	RTA-OSEK

シンボル	説明	移植性
StopwatchTickType	ストップウォッチカウンタのチック単位で測定される実行時間（16 ビット以上の符号なし整数型）	RTA-OSEK
SymbolicName	メッセージオブジェクト識別子	OSEK
TaskRefType	TaskType へのポインタ	OSEK
TasksetConstType	RTA-OSEK 読み取り専用タスクセット	RTA-OSEK
TasksetRefType	TasksetType へのポインタ	RTA-OSEK
TasksetType	RTA-OSEK タスクのセット	RTA-OSEK
TaskStateRefType	TaskStateType へのポインタ	OSEK
TaskStateType	タスクの状態（READY、RUNNING、SUSPENDED、WAITING のいずれか）	OSEK
TaskType	タスクオブジェクト	OSEK
TickRefType	TickType へのポインタ	OSEK
TickType	チック単位のカウンタ値（16 ビット以上の符号なし整数型）	OSEK
UInt16Type	数量（16 ビットの符号な整数型）	RTA-OSEK
UInt32Type	数量（32 ビットの符号なし整数型） ※この型は、ターゲットによってはサポートされていない場合もあります。	RTA-OSEK
UIntType	数量（16 ビット以上の符号なし整数型）	RTA-OSEK

2.1 型定義に関する注記

AlarmBaseType: OSEK アラームと共に使用される、以下のような構造体です。

```
struct {
   TickType maxallowedvalue;
   TickType ticksperbase;
   TickType mincycle;
}
```

ここで、maxallowedvalue は許容される最大カウント値（単位はチック）、ticksperbase はカウンタ固有の単位に至るために必要なチック数（RTA-OSEK では使用されません）、mincycle は SetRelAlarm() または SetAbsAlarm() の引数 cycle として許容される最小値です。

ScheduleStatusType: RTA-OSEK のスケジュール、および OSEK のアドバンスドカウンタと共に使用される、以下のような C 構造体です。

```
struct {
   SmallType status;
   TickType expiry;
}
```

status には、図 2-1 に示すような 2 つのビットフィールドが含まれています。

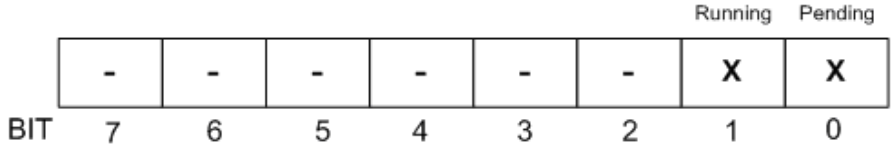


図 2-1 ScheduleStatusType() のビットフィールド

これらのビットフィールドの値により、expiry の値が定義されているかどうかがわかります。その内容は、以下の表のとおりです。

OS_STATUS_RUNNING	OS_STATUS_PENDING	status の値	expiry の定義	説明
Not Set	Not Set	0	なし	停止していて、処理待ち状態ではありません。
Not Set	Set	1	なし	停止していて、処理待ち状態です（実際にはあり得ない状態です）。
Set	Not Set	2	あり	稼働中であり、処理待ち状態ではありません。
Set	Set	3	なし	稼働中で、処理待ち状態です。

expiry が定義されている場合、expiry の値は以下のタイミングまでのチック数を示します。

- RTA-OSEK スケジュールが次のアライバルポイントを処理する
- OSEK のアドバンスドアラームまたは AUTOSAR スケジュールテーブルの次の満了ポイントに到達する

重要

expiry の値は次のポイントまでのチック数です。この値が 0 の場合、スケジュールが次のアライバルポイントを処理するタイミングになるまでのチック数が、RTA-OSEK スケジュールの最大値、または OSEK のアドバンスドカウンタの maxallowedvalue に等しいことを意味しています。

3 RTA-OSEK API リファレンス

本項では、RTA-OSEK API 関数をアルファベット順に紹介し、各関数について詳しく説明しています。これらの API 関数はすべてのコンフォーマンスクラスで使用できます。

3.1 OSEK コンフォーマンス

RTA-OSEK は、OSEK OS コンフォーマンスクラス BCC1、BCC2、ECC1、ECC2 の要件、および OSEK COM 用の CCCA と CCCB を満たすものとして認定されています。

3.2 AUTOSAR と RTA-OSEK の機能

RTA-OSEK には、AUTOSAR OS スケーラビリティクラス 1（リリース 1.0）の機能や、標準の OSEK OS 規格に含まれていない付加的機能もあり、これらの機能についても以降の項に明確に記述されています。

3.3 静的インターフェースと動的インターフェース

移植性

静的インターフェースは、RTA-OSEK に固有のものです。

静的インターフェース（RTA-OSEK の API 関数の「静的バージョン」と呼ばれます）は、動的インターフェースの当該機能と同じ処理を実現する一連の API 関数で構成されていますが、実行時間を少ししか、またはまったく犠牲にしません。API 関数の静的バージョンの方が、より効率的で、静的なエラーチェックが行われる可能性があります。API 関数の静的バージョンは、以下のように動的呼び出しから派生した形式で呼び出されます。

```
APIName(param1) → APIName_param1()
APIName(param1, param2) → APIName_param1_param2()
APIName(param1, param2) → APIName_param1(param2)
```

次の章で説明する API 関数のうち、静的バージョンも用意されているものは、その呼び出し方法も紹介されています。また 11.2 項には API の静的バージョンの一覧と概要が示されています。

3.4 API 関数の説明文のテンプレート

各 API 関数についての説明は、以下のような形式で記述されています。

各項のタイトルは、API 関数の名前です。

API 関数についての簡単な説明です。

関数宣言：

C 言語の構文でのインターフェース

静的バージョン：API 関数の静的バージョンがある場合、それを記述します。

引数：

引数	入力／出力	説明
Parameter Name	入力／出力	説明

説明：

API 関数の機能についての説明です。

エラーコード：

ビルド	コード	説明
RTA-OSEK のビルドステータス	Return values	戻り値の説明

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	
カテゴリ 1 割込み	
カテゴリ 2 割込み	

環境（フック）	有効／無効
スタートアップ	
シャットダウン	
プリタスク	
ポストタスク	
エラー	
オーバーラン	

有効性： ✓ = 有効、× = 無効、

 = RTA-OSEK でサポートされていますが、他の OSEK には移植できない場合があります。

移植性：

同じバージョンの各 OS 規格間の移植性が明記されています。

OSEK ― OSEK OS との間で移植性があります。

AUTOSAR ― AUTOSAR OS との間で移植性があります。

RTA-OSEK ― RTA-OSEK 間でのみ移植性があります。

注記：

API 関数の使用上の制約条件と注記

参照：

関連する API 関数のリスト

3.5 ActivateTask()

タスクを起動します。

関数宣言：

```
StatusTypeActivateTask(TaskType TaskID)
```

静的バージョン：`ActivateTask_TaskID(TaskID)`

引数：

引数	入力／出力	説明
`TaskID`	入力	起動されるタスク

説明：

タスク `TaskID` がサスペンド状態（'Suspended'）になっている場合は、レディ状態（'Ready'）に移行します。BCC2 タスクの場合、すでにそのタスクがレディ状態または実行状態（'Running'）であっても、起動数の合計が起動最大数より少なければ、起動処理が行われ、起動キューに入ります。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし
All	`E_OS_LIMIT`	複数回起動されているタスク `TaskID` の起動数が多すぎます。そのため、このタスクの起動要求は無視されます。
Extended	`E_OS_ID`	`TaskID` のタスクタイプが無効です。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR

OSEK

注記：

タイミング分析を正しく行うには、優先度の低いタスクから優先度の高いタスクを起動しないようにする必要があります。

`ActivateTask()` を呼び出した後のリスケジューリングは、呼び出し元に応じ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：このタスクがターミネートするか `Schedule()` を呼び出すまでリスケジューリングは行われません。
- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。
- カテゴリ 2 の ISR：この ISR がターミネートするまでリスケジューリングは行われません。

- フック：このフックが終了するまでリスケジューリングは行われません。

参照：

```
ChainTask()
DeclareTask()
GetTaskID()
GetTaskState()
TerminateTask()
```

3.6 ActivateTaskset()

タスクセット（1 組のタスク）を起動します。

関数宣言：

```
StatusTypeActivateTaskset(TasksetType TasksetID)
```

静的バージョン：`ActivateTaskset_TasksetID()`

引数：

引数	入力／出力	説明
`TasksetID`	入力	タスクセットの名前

説明：

タスクセット `TasksetID` 内のタスクがサスペンド状態（'Suspended'）からレディ状態（'Ready'）に移行します。タスクセットのうちのいずれかのタスクがすでにレディ状態または実行状態（'Running'）であるとき、そのタスクが起動のキューイングをサポートしていない場合（つまりそのタスクが BCC2 タスクではない）、またはキューイングをサポートしていてもキューが満杯の場合、起動要求は無効となります。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし
Extended	`E_OS_ID`	`TasksetID` のタスクセットタイプが無効です。
Extended	`E_OS_LIMIT`	タスクの起動数が多すぎます。そのため、この呼び出しによるタスク起動要求はすべて無視されます。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

タイミング分析を正しく行うには、優先度の高いタスクが含まれているタスクセットを優先度の低いタスクから起動してはいけません。

`ActivateTaskset()` を呼び出した後のリスケジューリングは、呼び出し元の環境により以下のように異なります。

- ノンプリエンプティブタスク：このタスクがターミネートするか `Schedule()` を呼び出すまでリスケジューリングは行われません。
- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。
- カテゴリ 2 の ISR：この ISR がターミネートするまでリスケジューリングは行われません。
- フック：このフックが終了するまでリスケジューリングは行われません。

参照：

```
AssignTaskset()
ChainTaskset()
GetTasksetRef()
MergeTaskset()
RemoveTaskset()
TestEquivalentTaskset()
TestSubTaskset()
```

3.7 AdvanceSchedule()

アドバンスドスケジュール上の次のアライバルポイントを処理します。

関数宣言：

```
StatusType AdvanceSchedule(
              ScheduleType            ScheduleID,
              ScheduleStatusRefType   ScheduleStatus)
```

静的バージョン：

```
AdvanceSchedule_ScheduleID (
              ScheduleStatusRefType ScheduleStatus)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	スケジュールの名前
ScheduleStatus	出力	スケジュールステータスへのポインタ

説明：

アドバンスドスケジュール ScheduleID に対して、次のアライバルポイントを処理しなければならないことを知らせます。そのアライバルポイントに関連付けられているタスクが起動されます。

最後のアライバルポイントがこの API 関数により処理されると、そのアドバンスドスケジュールは停止し、アドバンスドスケジュールのドライバコールバック関数 Cancel_<ScheduleID>() が呼び出されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ID	ScheduleID が無効です
Extended	E_OS_LIMIT	タスクの起動数が多すぎます。そのため、この呼び出しによるタスク起動要求はすべて無視されます。
Extended	E_OS_STATE	ScheduleID は実行されていません。
Extended	E_OS_SYS_S_MISMATCH	スケジュールにアドバンスドカウンタが含まれています（この API 関数はチックカウンタだけに使用できます）。
Extended	E_OS_SYS_CONFIG_ERROR	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

AdvanceSchedule() を呼び出した後のリスケジューリングは、呼び出し元の環境により以下のように異なります。

- カテゴリ 2 の ISR：この ISR がターミネートするまでリスケジューリングは行われません。
- ノンプリエンプティブタスク：このタスクがターミネートするか `Schedule()` を呼び出すまでリスケジューリングは行われません。
- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。
- フック：このフックが終了するまでリスケジューリングは行われません。

戻されたステータスが Running（実行中）の場合、‘expire’ チック経過後に再びコールを行う必要があります。アプリケーションプログラマの責任において、このような処理が行われるようにしてください。

タイミング分析を正しく行うには、優先度の高いタスクの起動を発生させるようなアドバンスドスケジュールを優先度の低いタスクが行うことがないようにしてください。

詳細については、2.1 項に記述されています。

参照：

`GetArrivalpointDelay()`
`GetArrivalpointNext()`
`GetArrivalpointTasksetRef()`
`GetScheduleNext()`
`GetScheduleStatus()`
`GetScheduleValue()`
`SetArrivalpointDelay()`
`SetScheduleNext()`
`StartSchedule()`
`StopSchedule()`
`TestArrivalpointWritable()`
`TickSchedule()`

3.8 AssignTaskset()

あるタスクセットのメンバを別のタスクセットに割り当てます。

関数宣言：

```
StatusType AssignTaskset(TasksetType DestTaskset,
                         TasksetType SourceTaskset)
```

引数：

引数	入力／出力	説明
DestTaskset	出力	書き込み可能タスクセットの名前
SourceTaskset	入力	割り当てられるタスクが属しているタスクセット

説明：

DestTaskset というタスクセットが、SourceTaskset タスクセット内のタスクをすべて含むように更新されます。

SourceTaskset タスクセットと DestTaskset タスクセットのメンバが同じである場合、このコールを行ってもタスクセットのメンバはまったく変わりません。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_TS_INVALID	いずれかのタスクセットが無効です。
Extended	E_OS_SYS_TS_READONLY	DestTaskset は書き込み禁止になっています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

ActivateTaskset()
ChainTaskset()
GetTasksetRef()
MergeTaskset()
RemoveTaskset()
TestEquivalentTaskset()
TestSubTaskset()

3.9 CancelAlarm()

アラームをキャンセルします。

関数宣言：

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

引数：

引数	入力／出力	説明
AlarmID	入力	アラーム名

説明：

アラーム AlarmID をキャンセルします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_NOFUNC	アラーム AlarmID は現在アクティブになっていません。
Extended	E_OS_ID	アラーム AlarmID は無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

なし

参照：

```
DeclareAlarm()
GetAlarm()
GetAlarmBase()
SetAbsAlarm()
SetRelAlarm()
```

3.10 ChainTask()

呼び出し元のタスクをターミネートし、指定されたタスクを起動します。

関数宣言：

```
StatusTypeChainTask(TaskType TaskID)
```

静的バージョン： `ChainTask_TaskID()`

引数：

引数	入力／出力	説明
TaskID	入力	現在のタスクがターミネートした後に起動されるタスク

説明：

この API 関数は、呼び出し元のタスクをターミネートします。ターミネート後は、TaskID で指定されたタスクが起動します。このコールにより、後続のタスクの実行は、呼び出し元のタスクがターミネートした後の最も早いポイントで開始されます。

内部リソースは自動的に解放されます。

タスクは、起動数に影響を与えることなく自分自身にチェーンすることができます。

エラーコード：

ビルド	コード	説明
All	E_OS_LIMIT	タスク TaskID の起動数が多すぎます。そのため、このタスクの起動要求は無視されます。
Extended	E_OS_ID	TaskID のタスクタイプが無効です。
Extended	E_OS_CALLEVEL	割込みレベルから呼び出されました。
Extended	E_OS_RESOURCE	呼び出し元タスクがまだリソースを占有しています。
Extended	E_OS_SYS_IDLE	アイドルタスクから呼び出されました。
Extended	E_OS_SYS_CONFIG_ERROR	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	×
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

`ChainTask()` はタスクのエントリ関数の最後の命令文として呼び出される必要があります。

この API 関数の処理が成功すると、`ChainTask()` は呼び出し元のレベルにコントロールを戻さないので、呼び出し元でコールステータスを評価することはできません。

拡張（'Extended'）ビルドのヘビーウェイトタスクが原因でこの API 関数の呼び出しが失敗すると、エラーが返されるので、アプリケーション内でそれを評価することができます。

拡張ビルドの場合、たとえエラーがあってもライトウェイトタスクは必ずターミネートします。エラーフックが設定されている場合はそれが呼び出されます。

現在のタスクがまだ解放されていないリソースを保持している場合は、コールは失敗します。

ライトウェイトとして宣言されたタスクは、`TASK()` マクロで宣言されている関数からしか `ChainTask()` を呼び出すことはできません。また、この API 関数の呼び出しは、他の式に埋め込まれた形ではなく、必ず最上位レベルの文になっていなければなりません。

タイミング分析を正しく行うには、優先度の低いタスクから優先度の高いタスクにチェーンすることがないようにしてください。

参照：

`ActivateTask()`
`DeclareTask()`
`GetTaskState()`
`TerminateTask()`

3.11 ChainTaskset()

呼び出し元のタスクをターミネートし、指定されたタスクセット内のすべてのタスクを起動します。

関数宣言：

```
StatusTypeChainTaskset(TasksetConstType TasksetID)
```

静的バージョン： `ChainTaskset_TasksetID()`

引数：

引数	入力／出力	説明
`TasksetID`	入力	現在のタスクがターミネートしてから起動されるタスクが属するタスクセットの名前

説明：

この API 関数は、呼び出し元のタスクをターミネートします。ターミネート後は、`TasksetID` のタスクセットが起動されます。このコールを行うと、後続のタスクの実行は、呼び出し元のタスクがターミネートした後の最も早いポイントで開始されます。

内部リソースは自動的に解放されます。

タスクは、起動数に影響を与えることなく自分自身にチェーンすることができます。

エラーコード：

ビルド	コード	説明
Extended	`E_OS_ID`	`TasksetID` のタスクセットタイプが無効です。
Extended	`E_OS_LIMIT`	タスクの起動数が多すぎます。そのため、この呼び出しによるタスクの起動要求はすべて無視されます。
Extended	`E_OS_CALLEVEL`	割込みレベルから呼び出されました。
Extended	`E_OS_SYS_IDLE`	アイドルタスクから呼び出されました。
Extended	`E_OS_RESOURCE`	呼び出し元タスクがまだリソースを占有しています。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	×
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

`ChainTaskset()` はタスクのエントリ関数の最後の命令文として呼び出される必要があります。

この API 関数の処理が成功すると、`ChainTaskset()` は呼び出し元のレベルにコントロールを戻さないので、呼び出し側でコールステータスを評価することはできません。

拡張（'Extended'）ビルドのヘビーウェイトタスクが原因で呼び出し処理が失敗すると、エラーが返されるので、アプリケーション内でそれを評価することができます。

拡張ビルドの場合、たとえエラーがあってもライトウェイトタスクは必ずターミネートします。エラーフックが設定されている場合はそれが呼び出されます。

現在のタスクがまだ解放されていないリソースを保持している場合は、コールは失敗します。

ライトウェイトとして宣言されたタスクは、`TASK()` マクロで宣言されている関数からしか `ChainTaskset()` を呼び出すことはできません。また、この API 関数の呼び出しは、他の式に埋め込まれた形ではなく、必ず最上位レベルの文になっていなければなりません。

タイミング分析を正しく行うには、優先度の低いタスクから優先度の高いタスクにチェーンすることがないようにしてください。

参照：

`ActivateTaskset()`
`AssignTaskset()`
`GetTasksetRef()`
`MergeTaskset()`
`RemoveTaskset()`
`TestEquivalentTaskset()`
`TestSubTaskset()`

3.12 ClearEvent()

呼び出し元のタスクのイベントをクリアします。

関数宣言：

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

引数：

引数	入力／出力	説明
Mask	入力	クリアされるイベントのマスク

説明：

呼び出し元タスクのイベントのうち、Mask で指定されたものをクリアします。

エラーコード：

ビルド	コード	説明
Standard	E_OK	エラーなし
Extended	E_OS_ACCESS	拡張タスク以外からの呼び出しです。
Extended	E_OS_CALLEVEL	割込みレベルでの呼び出しです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

この API 関数は拡張タスクからしか呼び出せません。
イベントマスクにセットされていないイベントについては、何も変わりません。

参照：

```
DeclareEvent()
GetEvent()
SetEvent()
WaitEvent()
```

3.13 CloseCOM()

COM 通信が使用する低レベルのハードウェアリソースを解放します。

関数宣言：

```
StatusType CloseCOM(void)
```

引数：

なし

説明：

`CloseCOM()` は、COM 通信に使用される低レベルリソースを解放するために使用されます。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	✓
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

COM 処理を終了するには、`StopCOM()` を使用してから `CloseCOM()` を使用してください。

移植性を保つために、この呼び出しがタスクレベルから行われる場合は、割込みをマスクしておく必要があります。

RTA-OSEK ライブラリには、`E_OK` を返すだけの `CloseCOM()` のデフォルトコールが含まれています。必要に応じて独自の `CloseCOM()` を作成し、それをデフォルトコールの代わりにアプリケーションにリンクすることができます。

参照：

`CloseCOM()`
`DeclareMessage()`
`InitCOM()`
`MessageInit()`
`ReadFlag()`
`ReceiveMessage()`
`ResetFlag()`
`SendMessage()`
`StartCOM()`
`StopCOM()`

3.14 DisableAllInterrupts()

すべての割込みをディセーブルにします。

関数宣言：

```
void DisableAllInterrupts(void)
```

引数：

なし

説明：

この API 関数を使うと、ハードウェアでディセーブルにすることができる割込みをすべてディセーブルにすることができます。

後で `EnableAllInterrupts()` を呼び出されたときのために、この呼び出しが行われる前の状態が保存されます。

この API 関数は、クリティカルセクション（割込み禁止で実行されるコード部分）を実行する際に呼び出し、このセクションの終わりには必ず `EnableAllInterrupts()` を呼び出す必要があります。このクリティカルセクション内では他の API 関数を使用することはできません。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	✓
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	
シャットダウン	
プリタスク	
ポストタスク	
エラー	
オーバーラン	

移植性

AUTOSAR
OSEK

注記：

この API 関数はネスティングをサポートしていません。クリティカルセクション内でネスティングを行う場合は、`SuspendAllInterrupts()` と `ResumeAllInterrupts()` を使用してください。

参照：

```
DeclareISR()
EnableAllInterrupts()
ResumeAllInterrupts()
ResumeOSInterrupts()
SuspendAllInterrupts()
SuspendOSInterrupts()
```

3.15 EnableAllInterrupts()

`DisableAllInterrupts()` により開始されたクリティカルセクションを終了します。

関数宣言：

```
void EnableAllInterrupts(void)
```

引数：

なし

説明：

この API 関数は、`DisableAllInterrupts()` により保存された状態を復元します。

`DisableAllInterrupts()` により開始されたコードのクリティカルセクションは、`EnableAllInterrupts()` を呼び出すことにより終了します。このクリティカルセクション内では他の API 関数を使用することはできません。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	✓
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	
シャットダウン	
プリタスク	
ポストタスク	
エラー	
オーバーラン	

移植性：

AUTOSAR
OSEK

注記：

なし

参照：

```
DeclareISR()
DisableAllInterrupts()
ResumeAllInterrupts()
ResumeOSInterrupts()
SuspendAllInterrupts()
SuspendOSInterrupts()
```

3.16 GetActiveApplicationMode()

現在のアプリケーションモードを取得します。

関数宣言：

```
AppModeType GetActiveApplicationMode(void)
```

引数：

なし

説明：

現在のアプリケーションモードを返します。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	✓
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	✓

移植性：

AUTOSAR
OSEK

注記：

なし

参照：

なし

3.17 GetAlarm()

アラームの次の満了時までのチック数を取得します。

関数宣言：

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

引数：

引数	入力／出力	説明
AlarmID	入力	アラームの名前
Tick	出力	アラーム満了までの相対チック数

説明：

アラーム AlarmID が満了するまでの相対チック数を返します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_NOFUNC	アラーム AlarmID は現在アクティブになっていません。
Extended	E_OS_ID	アラーム AlarmID は無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◐
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	◐

移植性：

AUTOSAR
OSEK

注記：

なし

参照：

CancelAlarm()
DeclareAlarm()
GetAlarmBase()
SetAbsAlarm()
SetRelAlarm()

3.18 GetAlarmBase()

アラームの基本特性を取得します。

関数宣言：

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

引数：

引数	入力／出力	説明
AlarmID	入力	アラーム名
Info	出力	アラームベースの定数を持つ構造体へのポインタ

説明：

GetAlarmBase() はアラームの基本特性を読み取ります。戻り値 Info は構造体で、そこには AlarmBaseType というデータ型の情報が格納されています。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ID	アラーム AlarmID は無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◔
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	◔

移植性：

AUTOSAR
OSEK

注記：

AlarmBaseType の詳細については 2.1 項を参照してください。

参照：

CancelAlarm()
DeclareAlarm()
GetAlarm()
SetAbsAlarm()
SetRelAlarm()

3.19 GetArrivalpointDelay()

1 つのアライバルポイントから次のアライバルポイントまでの遅延時間を取得します。

関数宣言：

```
StatusType GetArrivalpointDelay(
              ArrivalpointType   ArrivalpointID,
              TickRefType        Delay)
```

引数：

引数	入力／出力	説明
ArrivalpointID	入力	アライバルポイント名
Delay	出力	チック値へのポインタ

説明：

アライバルポイントの Delay プロパティに基づいて Delay を更新します。
Delay がゼロの場合、遅延時間は、現在使用されているスケジュールの最大値と等しくなります。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.20 GetArrivalpointNext()

あるアライバルポイントの次のアライバルポイントを取得します。

関数宣言：

```
StatusType GetArrivalpointNext(
             ArrivalpointType     ArrivalpointID,
             ArrivalpointRefType  ArrivalpointNextID)
```

引数：

引数	入力／出力	説明
ArrivalpointID	入力	アライバルポイント名
ArrivalpointNextID	出力	アライバルポイントへのポインタ

説明：

戻り値 ArrivalpointNextID は、入力されたアライバルポイントの Next（次のアライバルポイント）プロパティのハンドルを返します。

アライバルポイント ArrivalpointNextID はヌルの場合もあります。その場合、この引数の値はゼロです。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.21 GetArrivalpointTasksetRef()

アライバルポイントに割り当てられたタスクセットへのポインタを取得します。

関数宣言：

```
StatusType GetArrivalpointTasksetRef(
             ArrivalpointType   ArrivalpointID,
             TasksetRefType     TasksetID)
```

引数：

引数	入力／出力	説明
ArrivalpointID	入力	アライバルポイント名
TasksetID	出力	タスクセットの名前

説明：

TasksetID は、TasksetType へのポインタです。この API 関数はアライバルポイントに割り当てられているタスクセットへのポインタを返します。このポインタはタスクセット API 関数で使用することができ、そのアライバルポイントが書き込み可能な場合は、アライバルポイントのタスクセットを直接更新することができます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	●
タスク	●
カテゴリ 1 割込み	×
カテゴリ 2 割込み	●

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	●
ポストタスク	●
エラー	●
オーバーラン	●

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.22 GetCounterValue()

カウンタ値を取得します。

関数宣言：

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

引数：

引数	入力／出力	説明
CounterID	入力	カウンタ名
CountRef	出力	チック値へのポインタ

説明：

指定されたカウンタ CounterID の現在の値を CountRef が参照するメモリに代入して返します。
アドバンスドカウンタの場合、ユーザーが作成したコールバック関数 Get_<CounterID> が呼ばれます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_COUNTER_INVALID	無効なカウンタ

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

アプリケーションで有効な値を取得するには、GetCounterValue() に続けて
Tick_<CounterID>() ／ osAdvanceCounter_<CounterID>() を呼び出す必要があります。

参照：

DeclareCounter()
IncrementCounter()
InitCounter()
osAdvanceCounter_<CounterID>()
Tick_<CounterID>()

3.23 GetEvent()

指定されたタスク用のイベントを取得します。

関数宣言：

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

引数：

引数	入力／出力	説明
TaskID	入力	タスクの名前
Event	出力	リターンデータのメモリへのポインタ

説明：

この API 関数は、タスク TaskID のすべてのイベントビットの現在の状態を返します（現在タスク TaskID が待っているイベントを返すわけではありません）。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ACCESS	指定されたタスクは拡張タスクではありません。
Extended	E_OS_ID	タスク TaskID が無効です。
Extended	E_OS_STATE	指定されたタスクがサスペンド状態（'Suspended'）になっているので、イベントをセットすることができません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	

移植性：

AUTOSAR
OSEK

注記：

TaskID は拡張タスクの ID である必要があります。

参照：

ClearEvent()
DeclareEvent()
SetEvent()
WaitEvent()

3.24 GetISRID()

現在実行されている ISR の ID を取得します。

関数宣言：

```
ISRType GetISRID(void)
```

引数：

なし

説明：

この API 関数は、現在実行されている ISR の ID を返します。

エラーコード：

ビルド	コード	説明
All	INVALID_ISR	呼び出し元がアクティブな ISR ではありません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	×
タスク	×
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	
ポストタスク	
エラー	
オーバーラン	

移植性：

AUTOSAR

注記：

なし

参照：

GetTaskID()

3.25 GetMessageResource()

メッセージリソースを取得します。

関数宣言：

```
StatusType GetMessageResource(SymbolicName Message)
```

引数：

引数	入力／出力	説明
Message	入力	メッセージのシンボル名

説明：

メッセージリソースを使用するように設定されているシステムの場合、この API 関数はメッセージオブジェクト Message 用のリソースを占有してから、このメッセージオブジェクトのステータスをビジーにします。この API 関数を ReleaseMessageResource() と共に用いることにより、メッセージオブジェクトの同時アクセスを防ぐことができます。

メッセージリソースの共有は、タスク間、またはカテゴリ 2 の ISR 間、あるいはタスクとカテゴリ 2 の ISR との間で行うことができます。

クリティカルセクションの最後では必ず ReleaseMessageResource() を使用してください。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_COM_ID	引数 Message が無効です。
Extended	E_OS_ACCESS	取得しようとしたリソースが、いずれかのタスクかカテゴリ 2 の ISR、または呼び出し元タスクに静的に割り当てられている優先度によって、すでに占有されています。 または、割込みルーチンの優先度の方が、算出されたシーリング優先度よりも高くなっています。
Extended	E_OS_SYS_R_PERMISSION	メッセージリソースを使用するように宣言されていないタスクが、この API 関数を呼び出しました。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

リソース取得のネスティングは厳密に行われる必要があり、また同じリソースの占有をネストさせることは禁止されています。
リソースを占有しているタスクをターミネートしたり、ISR がリソースを占有したままにしておくことはできません。

メッセージリソースは、WithoutCopy モードのメッセージについてだけ使用してください。

参照：

```
GetMessageStatus()
ReleaseMessageResource()
```

3.26 GetMessageStatus()

メッセージステータスを取得します。

関数宣言：

```
StatusType GetMessageStatus(SymbolicName Message)
```

引数：

引数	入力／出力	説明
Message	入力	メッセージオブジェクトのシンボル名

説明：

この API 関数はメッセージオブジェクト Message のカレントステータスを返します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_COM_BUSY	Message により特定されたメッセージがすでにビジーになっています。
All	E_COM_LIMIT	Message により特定された待ち行列メッセージの FIFO でオーバーフローが発生しました。
All	E_COM_NOMSG	使用できるメッセージ（つまり、FIFO が空になっている待ち行列メッセージ）がありません。
Extended	E_COM_ID	引数 Message が無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◎
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	◎
ポストタスク	◎
エラー	◎
オーバーラン	◎

移植性：

OSEK

注記：

なし

参照：

```
GetMessageResource()
ReleaseMessageResource()
```

3.27 GetResource()

リソースを取得します。

関数宣言：

```
StatusTypeGetResource(ResourceType ResID)
```

静的バージョン： `GetResource_ResID()`

引数：

引数	入力／出力	説明
`ResID`	入力	取得しようとするリソース

説明：

コードのクリティカルセクションを、他のタスクまたは ISR による同時アクセスから守ります。

リソースの共有は、タスク間、カテゴリ 2 の ISR 間、またはタスクとカテゴリ 2 の ISR との間で行えます。

クリティカルセクションの最後では必ず `ReleaseResource()` を使用してください。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし
Extended	`E_OS_ACCESS`	取得しようとしたリソースが、いずれかのタスク、カテゴリ 2 ISR、または呼び出し元タスクに静的に割り当てられている優先度によって、すでに占有されています。 または、割込みルーチンの優先度の方が、算出されたシーリング優先度よりも高くなっています。
Extended	`E_OS_ID`	`ResID` のリソースタイプが無効です。
Extended	`E_OS_SYS_R_PERMISSION`	このメッセージリソースを使用するように宣言されていないタスクが、この API 関数を呼び出しました。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

リソース占有のネスティングは、内側のクリティカルセクションがその外側のクリティカルセクション内で完全に実行される場合、つまり厳密にネストされている場合に限り可能です。同じリソースを多重に占有することもできません。

同じリソースを多重に占有する必要がある場合は、RTA-OSEK の GUI で、リンクリソース（'linked resource'）を適宜に設定しておく必要があります。

実行状態（'Running'）のタスクを他の状態にする API 関数（`ChainTask()`、`Schedule()`、`TerminateTask()`、`WaitEvent()` など）をクリティカルセクション内で使用することができません。

参照：

`DeclareResource()`
`ReleaseResource()`

3.28 GetScheduleNext()

スケジュールにより処理される次のアライバルポイントを取得します。

関数宣言：

```
StatusType GetScheduleNext(
                ScheduleType          ScheduleID,
                ArrivalpointRefType   ArrivalpointID)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	スケジュール名
ArrivalpointID	出力	次のアライバルポイントへのポインタ

説明：

スケジュールの Next プロパティに基づいて ArrivalpointID を更新します。
Next がヌルの場合は、ゼロという値が定義されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_S_INVALID	スケジュールハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.29 GetScheduleStatus()

スケジュールのカレントステータスを取得します。

関数宣言：

```
StatusType GetScheduleStatus(
              ScheduleType            ScheduleID,
              ScheduleStatusRefType   ScheduleStatus)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	スケジュール名
ScheduleStatus	出力	スケジュールステータスへのポインタ

説明：

スケジュールの現在の状態、および、現時点からスケジュールが次のアライバルポイントを処理しなければならない時点までのスケジュールのカウンタのチック数をセットして、ScheduleStatus を更新します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_S_INVALID	スケジュールハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

ScheduleStatusType の詳細については、2.1 項を参照してください。

参照：

```
AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()
```

3.30 GetScheduleTableStatus()

スケジュールテーブルのカレントステータスを取得します。

関数宣言：

```
StatusType GetSchedulTableeStatus(
              ScheduleTableType            ScheduleID,
              ScheduleTableStatusRefType   ScheduleStatus)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	スケジュールテーブル名
ScheduleStatus	出力	スケジュールステータスへのポインタ

説明：

スケジュールテーブルの現在の状態を ScheduleStatus にセットします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_ID	スケジュールテーブルハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	

移植性：

AUTOSAR

注記：

なし

参照：

NextSetScheduleTable()
StartScheduleTable()
StopScheduleTable()

3.31 GetScheduleValue()

スケジュールの Now プロパティを取得します。

関数宣言：

```
StatusType GetScheduleValue( ScheduleType ScheduleID,
                             TickRefType  Tick)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	スケジュール名
Tick	出力	チック値へのポインタ

説明：

スケジュールに含まれているカウンタの Now プロパティへのポインタを、Tick にセットします。Now プロパティの値は、0 から（スケジュール最大値－ 1）までの値を取ります。

アドバンスドスケジュールの場合、ユーザーが作成したデバイスドライバ Now_<ScheduleID>() 関数が呼び出されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_S_INVALID	無効なスケジュールです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

有効な値を確実に取得するには、GetScheduleValue() に続けて TickSchedule() ／ AdvanceSchedule() を呼び出す必要があります。

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.32 GetStackOffset()

現在のスタックポインタを取得します。

関数宣言：

```
GetStackOffset(StackOffsetRefType StackVal)
```

引数：

引数	入力／出力	説明
StackVal	出力	リターンデータのメモリへのポインタ

説明：

スタック上のバイト数を示す値を返します。StackOffsetType の型は、ターゲットの種類に応じて、スカラ、またはスカラの構造体です。ターゲットに固有な情報は、各ターゲットの『RTA-OSEK バインディングマニュアル』に記述されています。

この API は、ユーザーコードのスタック使用量の測定と妥当性検証に利用できます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	●
タスク	●
カテゴリ 1 割込み	●
カテゴリ 2 割込み	●

環境（フック）	有効／無効
スタートアップ	●
シャットダウン	●
プリタスク	●
ポストタスク	●
エラー	●
オーバーラン	●

移植性：

RTA-OSEK

注記：

なし

参照：

なし

3.33 GetTaskID()

指定されたタスクのステータスを取得します。

関数宣言：

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

引数：

引数	入力／出力	説明
TaskID	出力	実行状態（'Running'）のタスクの名前

説明：

現在実行状態（'Running'）であるタスクの TaskID を返します。

実行状態（'Running'）のタスクがない場合や、現在アイドルタスクが実行状態（'Running'）である場合は、INVALID_TASK を返します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◔
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	◔

移植性：

AUTOSAR
OSEK

注記：

この API 関数は、ライブラリ関数やフックルーチンが使用するためのものです。

参照：

```
ActivateTask()
ChainTask()
DeclareTask()
GetISRID()
GetTaskState()
TerminateTask()
```

3.34 GetTasksetRef()

指定されたタスクだけが属するタスクセットを 1 つ生成します。

関数宣言：

```
StatusType GetTasksetRef( TaskType        TaskID,
                          TasksetRefType  TasksetRef)
```

引数：

引数	入力／出力	説明
TaskID	入力	タスク名
TasksetRef	出力	書き込み可能な TasksetType ハンドルへのポインタ

説明：

TasksetRef によりポイントされる TasksetType が更新され、TaskID によって表される 1 つのタスクだけが属するタスクセットを指し示すようになります。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ID	TaskID のタスクタイプが無効です。
Extended	E_OS_SYS_IDLE	TaskID はアイドルタスクです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	●
タスク	●
カテゴリ 1 割込み	×
カテゴリ 2 割込み	●

環境（フック）	有効／無効
スタートアップ	●
シャットダウン	×
プリタスク	●
ポストタスク	●
エラー	●
オーバーラン	●

移植性：

RTA-OSEK

注記：

なし

参照：

ActivateTaskset()
AssignTaskset()
ChainTaskset()
MergeTaskset()
RemoveTaskset()
TestEquivalentTaskset()
TestSubTaskset()

3.35 GetTaskState()

タスクステートを取得します。

関数宣言：

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

引数：

引数	入力／出力	説明
TaskID	入力	タスク名
State	出力	タスク TaskID の状態へのポインタ

説明：

GetTaskState() を呼び出した時点でのタスクの状態を返します。ステートは以下のいずれかです。

- RUNNING
- READY
- SUSPENDED
- WAITING

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ID	TaskID のタスクタイプが無効です。
Extended	E_OS_SYS_IDLE	TaskID はアイドルタスクです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	

移植性：

AUTOSAR
OSEK

注記：

完全にプリエンプティブなシステムにおいてこの API 関数を使用して有効な結果を得るためには、割込みがディセーブルになっている状態でタスクが稼働している、という状態においてこの API 関数を呼び出す必要があります。

参照：

ActivateTask()
ChainTask()
DeclareTask()

3.36 IncrementCounter()

カウンタを 1 チックだけインクリメントします。

関数宣言：

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

静的バージョン： `Tick_<CounterID>()` を参照してください。

引数：

引数	入力／出力	説明
CounterID	入力	カウンタ名

説明：

この API 関数は、カウンタ `CounterID` に対して 1 チックが経過したことを知らせます。

カウンタにアタッチされたアラームに定義されているチック数が経過した場合、そのアラームに定義されたアクションが実行されます。

カウンタにアタッチされたスケジュールテーブルが稼動していて、その満了ポイントに定義されたチック数が経過した場合、その満了ポイントに定義されたアクションが実行されます。

アラームが 0 より大きいサイクルタイムで始動されている場合は、アラームの次回の満了のための match 値がセットされます。

また、カウンタに関連付けられているスケジュールテーブルが稼動している場合は、次の満了ポイントの match 値がセットされます。

エラーコード：

`IncrementCounter()` は直接エラーとして `E_OS_ID` を発行する可能性があり、さらに、アラームの満了、または満了ポイントの処理における、タスク起動やイベントセットの処理結果によっても 2 次的エラーが発生する可能性があります。エラー発生時には `ErrorHook()` が呼び出されます。

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ID	CounterID が無効です。
Extended	E_OS_LIMIT	タスクの起動数が多すぎます。そのため、この呼び出しによるタスクの起動要求はすべて無視されました。
Extended	E_OS_STATE	参照されるタスクがサスペンド状態のため、イベントをセットできません。
Extended	E_OS_ACCESS	参照されるタスクが拡張タスクではありません。
Extended	E_OS_SYS_CONFIG_ERROR	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	●
タスク	●
カテゴリ 1 割込み	×
カテゴリ 2 割込み	●

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR

注記：

`IncrementCounter()` を呼び出した後のリスケジューリングは、呼び出し元の環境により以下のように異なります。

- カテゴリ2のISR: カテゴリ2のISRがターミネートするまではリスケジューリングは行われません。
- ノンプリエンプティブタスク： ノンプリエンプティブタスクがターミネートするか `Schedule()` を呼び出すまではリスケジューリングは行われません。
- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。
- フック： フックが終了するまではリスケジューリングは行われません。

タイミング分析を正しく行うには、アラームの満了により優先度の高いタスクが起動されるカウンタを、優先度の低いタスクがチックしないようにしてください。

参照：

`DeclareCounter()`
`GetCounterValue()`
`InitCounter()`
`osAdvanceCounter_<CounterID>()`
`Tick_<CounterID>`

3.37 InitCOM()

COM が使用する低レベルリソースを初期化します。

関数宣言：

```
StatusType InitCOM(void)
```

引数：

なし

説明：

`InitCom()` は、COM により使用されるすべての低レベルリソースを初期化します。この API 関数は、RTA-OSEK を起動する前に、または任意のタスク内の任意のカーネルスタートアップフックルーチン内から呼び出すことができます。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

`StartCOM()` を使用する前に、この `InitCOM()` を使用するようにしてください。

移植性を保つために、この API 関数がタスクレベルから呼び出される場合は、割込みをマスクしておく必要があります。

RTA-OSEK ライブラリには、`E_OK` を返すだけの `InitCOM()` のデフォルトコールが含まれています。必要に応じて独自の `InitCOM()` を作成し、それをデフォルトコールの代わりにアプリケーションにリンクすることができます。

現バージョンの RTA-OSEK においては、プロセッサ内のメッセージングしかサポートされていないため、`InitCOM()` は必要ありませんが、移植性を確保するために、この API 関数を使用しておくことをお勧めします。

参照：

```
DeclareMessage()
ClearEvent()
CloseCOM()
MessageInit()
ReadFlag()
ReceiveMessage()
ResetFlag()
SendMessage()
StartCOM()
StopCOM()
```

3.38 InitCounter()

カウンタに所定のチック値を初期設定します。

関数宣言：

```
StatusType InitCounter( CounterType  Counter,
                        TickType     Start)
```

引数：

引数	入力／出力	説明
Counter	入力	カウンタ名
Start	入力	チック値

説明：

指定されたカウンタに、Start で指定された絶対値を設定します。

アドバンスドカウンタの場合、ユーザーが作成したコールバック関数 Set_<CounterID> が呼ばれます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_STATE	カウンタはすでに稼動しています。
Extended	E_OS_SYS_COUNTER_INVALID	カウンタが無効です。
Extended	E_OS_VALUE	Start が maxallowedvalue を超えています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

RTA-OSEK は起動時において、自動的にチックドカウンタをゼロに初期設定します。アプリケーションでは、InitCounter() に続けて Tick_<CounterID>() ／ osAdvanceCounter_<CounterID>() を呼び出す必要があります。

参照：

DeclareCounter()
GetCounterValue()
IncrementCounter()
osAdvanceCounter_<CounterID>()
Tick_<CounterID>()

3.39 MergeTaskset()

2 つのタスクセットをマージします。

関数宣言：

```
StatusType MergeTaskset(   TasksetType DestTaskset,
                           TasksetType SourceTaskset)
```

引数：

引数	入力／出力	説明
DestTaskset	出力	書き込み可能なタスクセットの名前
SourceTaskset	入力	マージするタスクが属するタスクセットの名前

説明：

DestTaskset が SourceTaskset とマージされ、その結果が DestTaskset に返されます。この機能は、共用体（union）をセットする操作と同じです。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_TS_INVALID	いずれかのタスクセットが無効です。
Extended	E_OS_SYS_TS_READONLY	DestTaskset が書き込み禁止になっています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

ActivateTaskset()
AssignTaskset()
ChainTaskset()
GetTasksetRef()
RemoveTaskset()
TestEquivalentTaskset()
TestSubTaskset()

3.40 NextScheduleTable()

指定されたスケジュールテーブルを停止し、別のスケジュールテーブルを起動します。

関数宣言：

```
StatusType NextScheduleTable(
              ScheduleTableType  ScheduleTableID_current,
              ScheduleTableType  ScheduleTableID_next)
```

引数：

引数	入力／出力	説明
ScheduleTableID_current	入力	現在稼働中のスケジュールテーブルのハンドル
ScheduleTableID_next	入力	次に起動したいスケジュールテーブルのハンドル

説明：

スケジュールテーブル ScheduleTableID_current 内の現在の満了ポイントに達した後、ScheduleTableID_next が起動します。

ScheduleTableID_current がシングルショットスケジュールテーブルの場合、ScheduleTableID_current の最後の満了ポイントの処理中に ScheduleTableID_next が起動するように設定されます。

ScheduleTableID_current が周期的スケジュールテーブルの場合、現在の周期の最後に ScheduleTableID_next が起動するように設定されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_ID	スケジュールテーブルのハンドルが無効です。
All	E_OS_NOFUNC	ScheduleTableID_current は稼動していません。
All	E_OS_STATE	ScheduleTableID_next はすでに稼働中であるか、または次のステートになっています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR

注記：

ScheduleTableID_next が起動する前に ScheduleTableID_current が停止した場合、ScheduleTableID_next は起動されません。

参照：

GetScheduleTableStatus()
StartScheduleTable()
StopScheduleTable()

3.41 osAdvanceCounter_<CounterID>()

アドバンスドカウンタに関連付けられた次のアラーム、またはスケジュールテーブルの次の満了ポイントを処理します。

関数宣言：

```
StatusType osAdvanceCounter_<CounterID>(
              ScheduleStatusRefType osStat)
```

引数：

引数	入力／出力	説明
osStat	入力	スケジュールステータスへの参照

説明：

この API 関数は、アドバンスドカウンタ CounterID を使用するオブジェクトに対して、次のアラームまたは満了ポイントが処理されるタイミングになったことを通知します。それによってアラームや満了ポイントに定義されたアクションが実行されます。

処理されるべきアラームや満了ポイントがセットされていない場合、アドバンスドカウンタは停止し、ユーザー定義のデバイスドライバ用コールバック関数 Cancel_<CounterID>() が呼び出されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_STATE	CounterID は稼動していません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

osAdvanceCounter_<CounterID>() を呼び出した後のリスケジューリングは、呼び出し元の環境により以下のように異なります。

- カテゴリ 2 の ISR：この ISR がターミネートするまでリスケジューリングは行われません。
- ノンプリエンプティブタスク：このタスクがターミネートするか Schedule() を呼び出すまでリスケジューリングは行われません。
- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。

- フック：このフックが終了するまでリスケジューリングは行われません。

戻されたステータスが Running（実行中）の場合、‘expire’チック経過後に再びコールを行う必要があります。アプリケーションプログラマの責任において、このような処理が行われるようにしてください。

詳細は 2.1 項を参照してください。

タイミング分析を正しく行うには、優先度の高いタスクの起動を発生させるようなアドバンスドスケジュールを優先度の低いタスクが行うことがないようにしてください。

参照：

```
DaclareCounter()
GetCounterValue()
IncrementCounter()
InitCounter()
Tick_<CounterID>()
```

3.42 osResetOS()

オペレーティングシステムのデータ構造体をリセットします。

関数宣言：

```
void osResetOS(void)
```

引数：

なし

説明：

キーとなる OS データ構造体をリセットし、OS が安全に再起動できるようにします。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	×
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

この API を呼び出すには、アプリケーションがアイドルタスク実行中の状態であり、すべての OS オブジェクト（アラーム、スケジュールテーブル、スケジュール）が稼動していない状態である必要があります。

参照：

```
StartOS()
ShutdownOS()
```

3.43 ReadFlag()

OSEK COM メッセージに関連付けられているメッセージフラグのステータスを読み取ります。

関数宣言：

```
FlagValue ReadFlag(FlagType FlagName)
```

引数：

引数	入力／出力	説明
FlagValue	戻り値	フラグ FlagName の状態
FlagName	入力	メッセージフラグ名

説明：

ReadFlag() は指定された通知フラグ FlagName の値を返します。メッセージがすでに到着している場合には TRUE を返し、そうでない場合には FALSE を返します。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◔
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	◔
ポストタスク	◔
エラー	◔
オーバーラン	◔

移植性：

OSEK

注記：

なし

参照：

DeclareMessage()
CloseCOM()
InitCOM()
MessageInit()
ReceiveMessage()
ResetFlag()
SendMessage()
StartCOM()
StopCOM()

3.44 ReceiveMessage()

指定された OSEK COM メッセージを受信します。

関数宣言：

```
StatusType ReceiveMessage(SymbolicName    Message,
                          AccessNameRef   Data)
```

引数：

引数	入力／出力	説明
Message	入力	メッセージのシンボル名
DatTa	出力	受信データを格納するメッセージデータフィールドへのポインタ

説明：

ReceiveMessage() を実行すると、メッセージコピー設定に従って、指定されたメッセージが提供されます。WithCopy の場合は、Message により特定されるメッセージオブジェクトが、Data により参照されるメッセージデータフィールドにコピーされます。WithoutCopy の場合は、アプリケーションはメッセージオブジェクトに直接アクセスします（つまりこの場合は返されるステータスだけが有効です）。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_COM_LIMIT	待ち行列メッセージ Message 用の FIFO が満杯になっているため、1 つ以上のメッセージが失われました。
All	E_COM_NOMSG	待ち行列メッセージ Message 用の FIFO が空です。
Extended	E_COM_ID	引数 Message が無効です。
Extended	E_COM_SYS_STOPPED	COM が起動されていません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

なし

参照：

DeclareMessage()
CloseCOM()
InitCOM()
MessageInit()
ReceiveMessage()
ResetFlag()
SendMessage()
StartCOM()
StopCOM()

3.45 ReleaseMessageResource()

現在保持されているメッセージリソースを解放します。

関数宣言：

```
StatusType ReleaseMessageResource(SymbolicName Message)
```

引数：

引数	入力／出力	説明
Message	入力	メッセージのシンボル名

説明：

この API 関数は、Message オブジェクトのステータスとして not busy をセットし、その後、そのメッセージリソースを解放します。これは、指定されたメッセージリソースにより保護されているクリティカルセクションの終わりを示すものです。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_COM_ID	引数 Message が無効です。
Extended	E_OS_ACCESS	シーリング優先度が呼び出し元タスクの優先度よりも低いリソースを解放しようとしました。
Extended	E_OS_NOFUNC	このタスクまたはカテゴリ 2 ISR が現在保持していないリソースを解放しようとしたか、あるいはリソースロックのネスティングが厳密に行われていません。
Extended	E_OS_SYS_R_PERMISSION	メッセージリソースを使用することが宣言されていないタスクにより呼び出されました。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◔
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	◔

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

Schedule()、TerminateTask()、および WaitEvent() など、また上向きの起動の場合には通常 ActivateTask() も、タスクがリソースを保持している間にそのタスクのステータスを変更する API 関数を実行することはできません。

参照：

GetMessageResource()
GetMessageStatus()

3.46 ReleaseResource()

現在保持されているリソースを解放します。

関数宣言：

```
StatusTypeReleaseResource(ResourceType ResID)
```

静的バージョン： `ReleaseResource_ResID()`

引数：

引数	入力／出力	説明
ResID	入力	リソース名

説明：

指定されたリソースを解放し、保護されているクリティカルセクションの終わりを示します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_ID	ResID のリソースタイプが無効です。
Extended	E_OS_ACCESS	シーリング優先度が呼び出し元タスクの優先度よりも低いリソースを解放しようとしました。
Extended	E_OS_NOFUNC	このタスクまたはカテゴリ 2 ISR が現在保持していないリソースを解放しようとしたか、あるいはリソースロックのネスティングが厳密に行われていません。
Extended	E_OS_SYS_R_PERMISSION	メッセージリソースを使用することが宣言されていないタスクにより呼び出されました。
Extended	E_OS_SYS_CONFIG_ERROR	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

リソースの取得のネスティングは、厳密に行われる必要があります。同じリソースの占有をネストさせることはできません。

Schedule()、TerminateTask()、および WaitEvent() など、また上向きの起動の場合には通常 ActivateTask() も、タスクがリソースを保持している間にそのタスクのステータスを変更する API 関数を呼び出すことはできません。

参照：

DeclareResource()
GetMessageResource()

3.47 RemoveTaskset()

タスクセットからタスクを除外します。

関数宣言：

```
StatusType RemoveTaskset( TasksetType DestTaskset,
                          TasksetType SourceTaskset)
```

引数：

引数	入力／出力	説明
DestTaskset	出力	書き込み可能なタスクセットの名前
SourceTaskset	入力	除外するタスクのタスクセットの名前

説明：

DestTaskset のタスクセットから、SourceTaskset に含まれるタスクを除外します。

ソースとデスティネーションが同じタスクセットである場合、デスティネーションのタスクセットは空になります。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_TS_INVALID	いずれかのタスクセットが無効です。
Extended	E_OS_SYS_TS_READONLY	DestTaskset は書き込み禁止になっています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

ActivateTaskset()
AssignTaskset()
ChainTaskset()
GetTasksetRef()
MergeTaskset()
TestEquivalentTaskset()
TestSubTaskset()

3.48 ResetFlag()

OSEK COM メッセージに関連付けられているフラグをリセットします。

関数宣言：

```
StatusType ResetFlag(FlagType FlagName)
```

引数：

引数	入力／出力	説明
FlagName	入力	メッセージフラグ名

説明：

`ResetFlag()` は指定された通知フラグ `FlagName` の値を `FALSE` にします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

なし

参照：

```
DeclareMessage()
CloseCOM()
InitCOM()
MessageInit()
ReadFlag()
ReceiveMessage()
SendMessage()
StartCOM()
StopCOM()
```

3.49 ResumeAllInterrupts()

割込みをイネーブルに戻します。

関数宣言：

```
void ResumeAllInterrupts(void)
```

引数：

なし

説明：

この API 関数によって、割込みから保護されているコードセクションが終わります。この「クリティカルセクション」と呼ばれるセクションは、`SuspendAllInterrupts()` を呼び出すことによって開始された部分です。

割込みの受け付け状態は、`SuspendAllInterrupts()` を呼び出す前の状態に復元されます。

エラーコード：

ビルド	コード	説明
Extended	`E_OS_STATE`	割込みはサスペンドされていません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	✓
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	✓
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	✓

移植性：

AUTOSAR
OSEK

注記：

この API 関数と、これに対応して使用される `SuspendAllInterrupts()` は、プロセッサの割込みの処理レベルを調整します。

`SuspendAllInterrupts()` と `ResumeAllInterrupts()` の間のクリティカルセクションでは、他の API 関数は使用できません。ただし `SuspendAllInterrupts()` と `ResumeAllInterrupts()` の組合せはネスティングが可能です。

RTA-OSEK の拡張（'Extended'）ビルドにおいては、このコールに対応するサスペンドコールが存在しない場合に `E_OS_STATE` でエラーフックが呼び出されるようにすることができます。

クリティカルセクション内で API 関数を使用する場合は、結合リソースを使用して、ISR との相互排除関係が保たれるようにしてください。

参照：

```
DeclareISR()
DisableAllInterrupts()
EnableAllInterrupts()
ResumeOSInterrupts()
SuspendAllInterrupts()
SuspendOSInterrupts()
```

3.50 ResumeOSInterrupts()

カテゴリ 2 割込みをイネーブルに戻します。

関数宣言：

```
void ResumeOSInterrupts(void)
```

引数：

なし

説明：

この API 関数は `SuspendOSInterrupts()` の後に実行され、`SuspendOSInterrupts()` が実行される前にイネーブルであったカテゴリ 2 割込みも再びイネーブルに戻します。

エラーコード：

ビルド	コード	説明
Extended	`E_OS_STATE`	割込みはサスペンドされていません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	✓
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

この API 関数と、これに対応して使用される `SuspendOSInterrupts()` は、プロセッサの割込みの処理レベルを調整します。

`SuspendOSInterrupts()` と `ResumeOSInterrupts()` の間のクリティカルセクションでは、他の API 関数は使用できません。ただし、`SuspendOSInterrupts()` と `ResumeOSInterrupts()` の組合せはネスティングが可能です。

RTA-OSEK の拡張（'Extended'）ビルドにおいては、このコールに対応する `SuspendOSInterrupts()` が存在しない場合、`E_OS_STATE` でエラーフックが呼び出されるようにすることができます。

クリティカルセクション内で API 関数を呼び出す場合は、結合リソースを使用して、ISR との相互排除関係が保たれるようにしてください。

参照：

```
DeclareISR()
DisableAllInterrupts()
EnableAllInterrupts()
ResumeAllInterrupts()
SuspendAllInterrupts()
SuspendOSInterrupts()
```

3.51 Schedule()

ノンプリエンプティブタスク用のリスケジューリングポイントを提供します。

関数宣言：

```
StatusType Schedule(void)
```

引数：

なし

説明：

この API 関数は、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が、そのタスクより優先度の高いタスクによってプリエンプトされることを許可する際に使用します。

この API 関数が呼び出されると、呼び出し元タスクよりも優先度の高いタスクが 1 つでもレディ状態（'Ready'）になっていた場合、その中で最高の優先度を持つタスクが実行されます。該当するタスクが 1 つもない場合は、呼び出し元タスクがそのまま続行されます。

`Schedule()` により優先度の高いタスクが実行されることになった場合、内部リソースは解放されますが、`Schedule()` から呼び出し元タスクの処理に戻った後は、そのリソースは再びそのタスクによって保持されます。

エラーコード：

ビルド	コード	説明
Standard	`E_OK`	エラーなし
Extended	`E_OS_CALLEVEL`	タスクが割込みをディセーブルにしている時に呼び出されました。
Extended	`E_OS_RESOURCE`	リソースが保持されている間に呼び出されました。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	×
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

この API 関数は、割込みリソースを持たない完全なプリエンプティブタスクでは、効力を持ちません。

この API 関数は、タイミング分析やスタック最適化が必要な場合には使用しないでください。

参照：

なし

3.52 SendMessage()

指定のデータを OSEK COM メッセージとして送信します。

関数宣言：

```
StatusTypeSendMessage(  SymbolicName Message,
                        AccessNameRef Data)
```

静的バージョン： `SendMessage_Message_Data()`

引数：

引数	入力／出力	説明
`Message`	入力	メッセージのシンボル名
`Data`	入力	送信するメッセージデータフィールドへのポインタ

説明：

`SendMessage()` は、メッセージコピー設定に従い、必要に応じて `Message` でポイントされるメッセージオブジェクトを更新し、さらに必要に応じてメッセージオブジェクトの送信要求も行います。転送モードが `WithCopy` の場合は、`Data` により参照されるメッセージコピーが `Message` により参照されるメッセージオブジェクトにコピーされます。`WithoutCopy` の場合は、アプリケーションがメッセージオブジェクトに直接アクセスします。

受信タスクが定義されている場合、そのタスクはサスペンド状態（'Suspended'）からレディ状態（'Ready'）に移行し、オペレーティングシステムはそのタスクのコードの最初の文から実行されるようにします。要求に応じて、コールバックが行われます。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし
Extended	`E_COM_ID`	`Message` が無効です。
Extended	`E_COM_LIMIT`	`Message` により参照される待ち行列メッセージの FIFO でオーバーフローが発生しました。
Extended	`E_COM_SYS_STOPPED`	COM が起動されていません。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

タイミング分析を正しく行うには、優先度の高いタスクを起動したり優先度の高いタスク用のイベントをセットしたりする結果になるようなメッセージを、優先度の低いタスクが送信してはいけません。

参照：

DeclareMessage()
CloseCOM()
InitCOM()
MessageInit()
ReadFlag()
ReceiveMessage()
ResetFlag()
StartCOM()
StopCOM()

3.53 SetAbsAlarm()

アラームを満了させるためのカウンタ値をセットします。

関数宣言：

```
StatusType SetAbsAlarm( AlarmTypeAlarmID,
                        TickType start,
                        TickType cycle)
```

引数：

引数	入力／出力	説明
AlarmID	入力	セットされるアラーム
start	入力	アラームをトリガするカウンタ値
cycle	入力	アラームが再びトリガされるまでのカウンタチック数

説明：

この API 関数は、アラームを始動させ、アラームをトリガするカウンタに値（「match 値」）を設定し、カウンタの値がこの値になったときにアラームがトリガされるようにします。アラームが 1 回しかトリガされないようにするには、0 を cycle にセットし、また繰り返しトリガされるようにするには、次回のアラームトリガまでのカウンタチック数を cycle にセットします。

アラームがトリガされると、そのアラームに関連付けられているタスクが起動されます。アラームは、設定に応じて、タスクの起動やイベントのセット、またアラームコールバックの呼び出しなどを行います。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_STATE	アラーム AlarmID は現在アクティブになっています。
Extended	E_OS_ID	アラーム AlarmID は無効です。
Extended	E_OS_VALUE	start または cycle の値が、カウンタに認められている範囲外の値になっています（start は 0 から maxallowedvalue まで、cycle は 0、または mincycle から maxallowedvalue までの値でなければなりません）。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

`SetAbsAlarm()` でアラームの値をセットするには、その前に必ず `CancelAlarm()` を使用してそのアラームを停止させておく必要があります。

`start` の値が現在のカウンタ値に近い場合は、タスク起動について注意が必要です。`SetAbsAlarm()` の処理が完了する前にカウンタ値が `start` と同じ値になった場合、タスクはアラームによって直ちに起動され、そうでない場合は、カウンタが一度オーバーランした後に `start` 値になった時点で起動されます。

参照：

`CancelAlarm()`
`DeclareAlarm()`
`GetAlarm()`
`GetAlarmBase()`
`SetRelAlarm()`

3.54 SetArrivalpointDelay()

あるアライバルポイントから次のアライバルポイントまでの遅延時間を設定します。

関数宣言：

```
StatusType SetArrivalpointDelay(
                ArrivalpointType   ArrivalpointID,
                TickType           Delay)
```

引数：

引数	入力／出力	説明
ArrivalpointID	入力	遅延時間を変更するアライバルポイント
Delay	入力	次のアライバルポイントまでの新しい遅延時間

説明：

アライバルポイントの Delay プロパティに、指定された値を設定します。

Delay プロパティがゼロの場合、遅延時間は、使用されているスケジュールの最大値と等しくなります。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。
Extended	E_OS_SYS_AP_READONLY	アライバルポイントが読み取り専用です。
Extended	E_OS_SYS_S_MODULO	Delay が、使用されているスケジュールの最大値を超えています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	●
タスク	●
カテゴリ 1 割込み	×
カテゴリ 2 割込み	●

環境（フック）	有効／無効
スタートアップ	●
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

この API 関数は、書き込み可能なアライバルポイントについてしか使用できません。

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.55 SetArrivalpointNext()

次のアライバルポイントを変更することにより、スケジュールを再設定します。

関数宣言：

```
StatusType SetArrivalpointNext(
             ArrivalpointType ArrivalpointID,
             ArrivalpointType ArrivalpointNextID)
```

引数：

引数	入力／出力	説明
ArrivalpointID	入力	変更されるアライバルポイント
ArrivalpointNextID	入力	次のアライバルポイント（変更後）

説明：

アライバルポイント ArrivalpointID の Next プロパティをセットします。

アライバルポイント ArrivalpointNextID はヌル（=0）でもかまいません。これにより、スケジュールをアライバルポイント ArrivalpointID で終了させることができます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイント ArrivalpointID または ArrivalpointNextID が無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイント ArrivalpointID がヌルです。
Extended	E_OS_SYS_AP_READONLY	アライバルポイント ArrivalpointID が読み取り専用です。
Extended	E_OS_SYS_S_MISMATCH	各アライバルポイントのスケジュールが一致しません（「ヌル」のアライバルポイントはすべてのスケジュールに使用できます）。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

この API 関数は、書き込み可能なアライバルポイントについてしか使用できません。

参照：

```
AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
```

```
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()
```

3.56 SetEvent()

指定されたタスク用のイベントをセットします。

関数宣言：

```
StatusTypeSetEvent(  TaskType        TaskID,
                     EventMaskType   Mask)
```

静的バージョン： `SetEvent_TaskID_Mask()`

引数：

引数	入力／出力	説明
`TaskID`	入力	イベントがセットされるタスクの名前
`Mask`	入力	セットされるイベントのマスク

説明：

この `API` 関数は、`Mask` に従ってタスク `TaskID` 用のイベントをセットします。タスクがシステム内で現在最も優先度の高いタスクであり、かつそのタスクが指定のイベントについて待ち状態（'Waiting'）であった場合、そのタスクは直ちにレディ状態（'Ready'）または実行状態（'Running'）になります。複数のイベントをセットするには、各イベントをビット単位で OR 処理した値をマスクとして使用します。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし
Extended	`E_OS_ACCESS`	当該タスクは拡張タスクではありません。
Extended	`E_OS_ID`	タスク `TaskID` が無効です。
Extended	`E_OS_STATE`	当該タスクがサスペンド状態（'Suspended'）であるため、イベントをセットできません。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

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まったく変わりません。

静的コールは 1 つのイベントだけをセットします。静的コール名は、タスク名とイベント名で構成されています。

タイミング分析を正しく行うには、優先度の低いタスクから優先度の高いタスク用のイベントをセットしないようにしてください。

標準（'Standard'）またはタイミング（'Timing'）ビルドの場合、サスペンド状態（'Suspended'）のタスクについてイベントをセットしても何の効果もなく、返されるステータスは `E_OK` です。これは、RTA-OSEK 固有の挙動です。

参照：

`ClearEvent()`
`DeclareEvent()`
`GetEvent()`
`WaitEvent()`

3.57 SetRelAlarm()

アラーム満了までのカウンタチック数を設定します。

関数宣言：

```
StatusType SetRelAlarm (   AlarmTypeAlarmID,
                           TickType increment,
                           TickType cycle)
```

引数：

引数	入力／出力	説明
AlarmID	入力	セットされるアラーム
increment	入力	アラームがトリガされるまでのカウンタチック数
cycle	入力	アラームが再びトリガされるまでのカウンタチック数

説明：

この API 関数は、アラームを始動させ、アラームがトリガされるまでのカウンタチック数を設定します。アラームが 1 回しかトリガされないようにするには、0 を cycle にセットし、また何度もトリガされるようにするには、次回のアラームトリガまでのカウンタチック数を cycle にセットします。

アラームがトリガされると、そのアラームに関連付けられているタスクが起動されます。アラームは、設定に応じて、タスクの起動やイベントのセット、またアラームコールバックの呼び出しなどを行います。

increment がゼロの場合の挙動は、設定に応じて以下のように異なります。

- OSEK (default) ー maxallowedvalue+1 だけ進んだ時点でアラームがトリガされます。
- OSEK (SetRelAlarm(0) disallowed) ー アラームはセットされず、API は E_OS_VALUE を返します。
- AUTOSAR ー アラームはセットされず、API は E_OS_VALUE を返します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_STATE	アラーム AlarmID は現在アクティブになっています。
Extended	E_OS_ID	アラーム AlarmID は無効です。
Extended	E_OS_VALUE	increment または cycle の値が、カウンタに認められている値の範囲を外れています（increment は 0 から maxallowedvalue まで、cycle は 0、または mincycle から maxallowedvalue までの値でなければなりません）。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

`SetRelAlarm()` によるアラームの値の変更は、アラームを始動させる前にしか行えません。

`increment` の値が現在のカウンタ値に近い場合は、アラーム満了のタイミングについて注意が必要です。
`SetRelAlarm()` の処理が完了する前にカウンタチック値が `increment` と同じ値になった場合、タスクはアラームによって直ちに起動され、そうでない場合は、カウンタが一度オーバーランした後に `increment` 値になった時点で起動されます。

参照：

`CancelAlarm()`
`DeclareAlarm()`
`GetAlarm()`
`GetAlarmBase()`
`SetAbsAlarm()`

3.58 SetScheduleNext()

スケジュールに次のアライバルポイントを設定します。

関数宣言：

```
StatusType SetScheduleNext(
                ScheduleType ScheduleID,
                ArrivalpointType ArrivalpointID)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	スケジュール名
ArrivalpointID	入力	スケジュールにより処理される、次のアライバルポイント

説明：

スケジュールに、ArrivalpointID で指定されるアライバルポイントが次のアライバルポイントであることを知らせます。

この API 関数は、スケジュールの状態とは無関係に使用できます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。
Extended	E_OS_SYS_S_INVALID	スケジュールハンドルが無効です。
Extended	E_OS_SYS_S_MISMATCH	アライバルポイント ArrivalpointID がスケジュール ScheduleID に属していません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.59 ShutdownOS()

オペレーティングシステムをシャットダウンします。

関数宣言：

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

引数：

引数	入力／出力	説明
Error	入力	シャットダウンフックに渡されるエラーメッセージ

説明：

オペレーティングシステムのすべてのアクティビティ（タスク、割込み、アラームの処理）を中止し、シャットダウンフックが定義されている場合にはそれを呼び出します。シャットダウンフックの処理が終わると、`ShutdownOS()` は無限ループに入ります。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	◐
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	✓
オーバーラン	◐

移植性：

AUTOSAR
OSEK

注記：

なし

参照：

`StartOS()`

3.60 StartCOM()

COM サービスを開始します。

関数宣言：

```
StatusType StartCOM(void)
```

引数：

なし

説明：

この API 関数は内部データ構造体を初期化して COM を起動します。またアプリケーション側で用意された `MessageInit()` コールバック関数がある場合はそれを呼び出して、アプリケーション固有のメッセージオブジェクトを初期化します。

`StartCOM()` はタスク内から呼び出されなければなりません。

エラーコード：

ビルド	コード	説明
Standard	`E_OK`	エラーなし
	（その他の値）	`MessageInit()` から返された、アプリケーション固有のエラーコード、

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

`StartCOM()` から返される `StatusType` の値は、`MessageInit()` により返された値です。ユーザーが独自の `MessageInit()` を定義していない場合、RTA-OSEK では、`E_OK` を返すだけの `MessageInit()` のデフォルトコールが組み込まれます。

参照：

```
CloseCOM()
DeclareMessage()
InitCOM()
MessageInit()
ReadFlag()
ReceiveMessage()
ResetFlag()
SendMessage()
StopCOM()
```

3.61 StartOS()

OS を起動します。

関数宣言：

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

引数：

引数	入力／出力	説明
Mode	入力	アプリケーションモード

説明：

StartOS() は、Mode により指定されたアプリケーションモードで RTA-OSEK を起動します。起動時にはすべてのデータ構造体が初期化され、すべてのカウンタ値が 0 になります。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	×
タスク	×
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

StartOS() コールに続くコードはアイドルタスクになるので、これは絶対にターミネートしないようにしてください。

参照：

ShutdownOS()

3.62 StartSchedule()

停止しているスケジュールを起動します。

関数宣言：

```
StatusType StartSchedule( ScheduleType ScheduleID,
                          TickType When)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	起動するスケジュール
When	入力	最初のアライバルポイントが処理される時のカウンタ値

説明：

この API 関数は、スケジュールによるアライバルポイントの処理を開始し、スケジュールを実行状態（'Running'）にします。

スケジュールが現在 Stopped 状態になっている場合、絶対カウンタ値 When のカウンタイベントがセットされます。When の値として有効なのは、0 から（スケジュールの最大値－ 1）までです。この最大値は、指定されたスケジュールの最大値です。

When の値が 0 の場合、スケジュールは、関連付けられているカウンタが一周した後に起動されます。アプリケーションプログラマは、スケジュールがアドバンスドスケジュールである場合には、この点について注意が必要です。

スケジュールの状態がすでに Running になっている場合は、何も処理は行われません。

アドバンスドスケジュールの場合で、すでに Stopped 状態になっていた場合は、デバイスドライバである Set_<ScheduleID>() 関数が呼び出されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。
Extended	E_OS_SYS_S_MODULO	When の値が、使用されているスケジュールの最大値を超えています。
Extended	E_OS_SYS_S_INVALID	スケジュールハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StopSchedule()
TestArrivalpointWritable()
TickSchedule()

3.63 StartScheduleTable()

スケジュールテーブルを起動します。

関数宣言：

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

引数：

引数	入力／出力	説明
ScheduleTableID	入力	起動するスケジュールテーブル
Offset	入力	現時点（'now'）から最初のアラーム満了までの相対チック数

説明：

Offset で示されるカウンタチック数のディレイが経過した後、スケジュールテーブル ScheduleTableID を最初の満了ポイントから起動します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_ID	スケジュールテーブルのハンドルが無効です。
All	E_OS_VALUE	オフセット値が MAXALLOWEDVALUE より大きくなっています。（チックドカウンタの場合）
All	E_OS_STATE	指定されたスケジュールテーブルはすでに稼動しているか、またはスケジュールテーブルが関連付けられたカウンタが別のスケジュールテーブルを駆動しています。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	✓
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR

注記：

なし

参照：

GetScheduleTableStatus()
NextScheduleTable()
StopScheduleTable()

3.64 StopCOM()

COM サービスを停止します。

関数宣言：

```
StatusType StopCOM(UIntType ShutdownMode)
```

引数：

引数	入力／出力	説明
ShutdownMode	入力	シャットダウンモード

説明：

StopCOM() は、すべての COM アクティビティを中止し、COM が使用していたすべてのリソースを解放します。すべての処理は直ちに中止され、データは失われます。処理待ち状態のすべての COM 処理が完了し、それらのリソースを解放できるようになるまで、StopCOM() はコントロールを戻しません。

StopCOM() が正常に終了すると、システムは、StartCOM() の呼び出せば COM を再起動できる状態となります。

ShutdownMode に使用できる値は COM_SHUTDOWN_IMMEDIATE のみです。シャットダウンは、処理待ち状態の処理が完了するのを待たずに直ちに行われます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_COM_BUSY	あるアプリケーションがメッセージリソースを保持しているため、COM をシャットダウンできませんでした。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	●
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	●

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	●
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

OSEK

注記：

なし

参照：

InitCOM()
MessageInit()
ReadFlag()
ReceiveMessage()
ResetFlag()
SendMessage()
StartCOM()

3.65 StopSchedule()

実行状態（'Running'）のスケジュールを停止させます。

関数宣言：

```
StatusType StopSchedule(ScheduleType ScheduleID)
```

引数：

引数	入力／出力	説明
ScheduleID	入力	停止したいスケジュール

説明：

この API 関数は、スケジュールによるアライバルポイントの処理を停止させます。

スケジュールは Stopped 状態になります。

ScheduleID がアドバンスドスケジュールである場合は、ユーザーが用意したアドバンスドスケジュールドライバ Cancel_<ScheduleID>() が呼び出されます。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_S_INVALID	スケジュールハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
TestArrivalpointWritable()
TickSchedule()

3.66 StopScheduleTable()

スケジュールテーブルを停止させます。

関数宣言：

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

引数：

引数	入力／出力	説明
ScheduleTableID	入力	停止したいスケジュールテーブル

説明：

この API 関数は、スケジュールテーブル ScheduleTableID の以後の満了ポイントの処理が行われないようにします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
All	E_OS_ID	スケジュールテーブルのハンドルが無効です。
All	E_OS_NOFUNC	スケジュールテーブルは稼動していません。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	✓
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR

注記：

なし

参照：

GetScheduleTableStatus()
NextScheduleTable()
StartScheduleTable()

3.67 SuspendAllInterrupts()

すべての割込み処理をディセーブルにします。

関数宣言：

```
void SuspendAllInterrupts(void)
```

引数：

なし

説明：

この API 関数は、すべての割込み処理をディセーブルにします。現在の状態は保存されるので、後で復元できます。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	✓
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	
シャットダウン	
プリタスク	✓
ポストタスク	✓
エラー	✓
オーバーラン	

移植性：

AUTOSAR
OSEK

注記：

この API 関数と、これに対応して使用される `ResumeAllInterrupts()` は、プロセッサの割込み処理レベルを調整します。

`SuspendAllInterrupts()` と `ResumeAllInterrupts()` の間のクリティカルセクションにおいては、他の API 関数は使用できません。ただし `SuspendAllInterrupts()` と `ResumeAllInterrupts()` の組合せはネスティングが可能です。

クリティカルセクション内で API 関数を呼び出す場合は、結合リソースを使用して、ISR との相互排除関係が保たれるようにしてください。

参照：

```
DeclareISR()
DisableAllInterrupts()
EnableAllInterrupts()
ResumeAllInterrupts()
ResumeOSInterrupts()
SuspendOSInterrupts()
```

3.68 SuspendOSInterrupts()

カテゴリ 2 割込みの割込み処理をディセーブルにします。

関数宣言：

```
void SuspendOSInterrupts(void)
```

引数：

なし

説明：

この API 関数は、現在の割込み処理レベルをセーブしてから、すべてのカテゴリ 2 割込みの割込み処理をディセーブルにします。保存された割込み処理レベルは、`ResumeOSInterrupts()` により復元されます。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	✓
カテゴリ 2 割込み	✓

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

この API 関数と、これに対応して使用される `ResumeOSInterrupts()` は、プロセッサの割込み処理レベルを調整します。

`SuspendOSInterrupts()` と `ResumeOSInterrupts()` の間のクリティカルセクションにおいては、他の API 関数は使用できません。ただし `SuspendOSInterrupts()` と `ResumeOSInterrupts()` の組合せはネスティングが可能です。

クリティカルセクション内で API 関数を呼び出す場合は、結合リソースを使用して、ISR との相互排除関係が保たれるようにしてください。

参照：

```
DeclareISR()
DisableAllInterrupts()
EnableAllInterrupts()
ResumeAllInterrupts()
ResumeOSInterrupts()
SuspendAllInterrupts()
```

3.69 TerminateTask()

呼び出し元タスクをターミネートします。

関数宣言：

```
StatusType TerminateTask(void)
```

引数：

なし

説明：

`TerminateTask()` は、呼び出し元タスクをターミネートさせるために使用されます。呼び出し元タスクは実行状態（'Running'）からサスペンド状態（'Suspended'）になり、後続の起動がキューイングされる BCC2 タスクの場合は実行状態（'Running'）からレディ状態（'Ready'）になります。

内部リソースは自動的に解放されます。

エラーコード：

ビルド	コード	説明
Extended	`E_OS_CALLEVEL`	割込みレベルから呼び出されました。
Extended	`E_OS_RESOURCE`	タスクがまだリソースを占有しています。
Extended	`E_OS_SYS_IDLE`	アイドルタスクから呼び出されました。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	×
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

内部リソース以外のすべてのリソースは、`TerminateTask()` を呼び出す前に解放されている必要があります。

すべてのタスクは、`TerminateTask()`、`ChainTask()`、または `ChainTaskset()` によってターミネートします。

ライトウェイトタスクとして宣言されているタスクの場合は、必ず `Task()` マクロで宣言されている関数から `TerminateTask()` を呼び出します。また、この呼び出し文は、式に埋め込まれたものではなく、最上位レベルの文でなければなりません。

拡張（'Extended'）ビルドにおいて、ヘビーウェイトタスクからの呼び出しが失敗した場合はエラーが返されるので、アプリケーション内で結果を評価することができます。

それに対してライトウェイトタスクの場合は、エラーがあってもタスクは必ずターミネートし、エラーフックが定義されている場合はそれが呼び出されます。

参照：

```
ActivateTask()
ChainTask()
DeclareTask()
GetTaskState()
```

3.70 TestArrivalpointWritable()

アライバルポイントが書き込み可能かどうかをテストします。

関数宣言：

```
StatusType TestArrivalpointWritable(
             ArrivalpointType   ArrivalpointID,
             BooleanRefType     WritableFlag)
```

引数：

引数	入力／出力	説明
ArrivalpointID	入力	テストされるアライバルポイント
WritableFlag	出力	論理フラグへのポインタ

説明：

アライバルポイントが書き込み可能である場合は、WritableFlag の値を TRUE にします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
Extended	E_OS_SYS_AP_NULL	アライバルポイントがヌルです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

なし

参照：

AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TickSchedule()

3.71 TestEquivalentTaskset()

2 つのタスクセットに含まれているタスクがまったく同じかどうかをテストします。

関数宣言：

```
StatusType TestEquivalentTaskset(
              TasksetType     Taskset1,
              TasksetType     Taskset2,
              BooleanRefType  EqFlag)
```

引数：

引数	入力／出力	説明
Taskset1	入力	第 1 のタスクセット
Taskset2	入力	第 2 のタスクセット
EqFlag	出力	論理フラグへのポインタ

説明：

EqFlag によりポイントされる論理値に、タスクセット Taskset1 内のタスクセットがタスクセット Taskset2 内のタスクとまったく同じである場合には TRUE になり、そうでない場合には FALSE をセットします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_TS_INVALID	いずれかのタスクセットが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

ActivateTaskset()
AssignTaskset()
ChainTaskset()
GetTasksetRef()
MergeTaskset()
RemoveTaskset()
TestSubTaskset()

3.72 TestSubTaskset()

タスクセット内に指定のタスクセットが存在するかどうかをテストします。

関数宣言：

```
StatusType TestSubTaskset(
                TasksetType     SubTaskset,
                TasksetType     RefTaskset,
                BooleanRefType  SubsetFlag)
```

引数：

引数	入力／出力	説明
SubTaskset	入力	テストされるタスクセット
RefTaskset	入力	参照されるタスクセット
SubsetFlag	出力	論理フラグへのポインタ

説明：

この API 関数は、あるタスクセットが、別のタスクセット内にサブセットとして含まれているかどうかを調べるためのものです。

SubsetFlag によりポイントされる論理値に、タスクセット SubsetTaskset 内のタスクがすべてタスクセット RefTaskset 内に存在する場合には TRUE、そうでない場合には FALSE をセットします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_TS_INVALID	いずれかのタスクセットが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	○
シャットダウン	×
プリタスク	○
ポストタスク	○
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

なし

参照：

ActivateTaskset()
AssignTaskset()
ChainTaskset()
GetTasksetRef()
MergeTaskset()
RemoveTaskset()
TestEquivalentTaskset()

3.73 Tick_<CounterID>()

カウンタを 1 チックだけインクリメントします。

関数宣言：

```
void Tick_<CounterID>()
```

引数：

なし

説明：

この API 関数は、カウンタ `CounterID` に、1 チックが経過したことを知らせます。このカウンタに関連付けられている各アラームをトリガするためのチック数がカウントされると、そのアラームに割り当てられいるアクションが実行されます。

また、カウンタにスケジュールテーブルが関連付けられていて、それが稼動している場合、所定のチック数がカウントされると、満了ポイントに割り当てられいるアクションが実行されます。

アラームが 0 より大きいサイクルタイムで始動されている場合は、アラームの次回の満了のための match 値がセットされます。

また、カウンタに関連付けられているスケジュールテーブルが稼動している場合は、次の満了ポイントの match 値がセットされます。

エラーコード：

`Tick_<CounterID>()` は直接エラーを発行することはできませんが、アラーム満了時のタスクの起動処理の結果により、以下のエラーが発生する可能性があります。その場合は、`ErrorHook()` が呼び出されます。

ビルド	コード	説明
All	`E_OK`	エラーなし
Extended	`E_OS_LIMIT`	タスクの起動数が多すぎます。そのため、この呼び出しによるタスクの起動要求はすべて無視されました。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

Tick_<CounterID>() を呼び出した後のリスケジューリングは、呼び出し元の環境により以下のように異なります。

- カテゴリ2のISR： カテゴリ2のISRがターミネートするまではリスケジューリングは行われません。
- ノンプリエンプティブタスク： ノンプリエンプティブタスクがターミネートするか `Schedule()` を呼び出すまではリスケジューリングは行われません。
- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。

- フック： フックが終了するまではリスケジューリングは行われません。

タイミング分析を正しく行うには、アラームの満了により優先度の高いタスクが起動されるカウンタを、優先度の低いタスクがチックしないようにしてください。

参照：

DeclareCounter()
GetCounterValue()
InitCounter()
IncrementCounter()
osAdvanceCounter_<CounterID>

3.74 TickSchedule()

チックスケジュールの Now プロパティの値を進め、その結果、場合によってはタスクを起動します。

関数宣言：

```
StatusTypeTickSchedule(ScheduleType ScheduleID)
```

静的バージョン： `TickSchedule_ScheduleID()`

引数：

引数	入力／出力	説明
`ScheduleID`	入力	スケジュール名

説明：

この API 関数は、チックスケジュール `ScheduleID` に、1 チック経過したことを知らせます。所定のチック数が経過すると、次のアライバルポイントが処理され、そのアライバルポイントに定義されているタスクが起動します。

最後のアライバルポイントがこの API 関数より処理されると、スケジュールは停止するので、通常、アプリケーションは `TickSchedule()` を周期的に呼び出し続けます。

エラーコード：

ビルド	コード	説明
All	`E_OK`	エラーなし
Extended	`E_OS_LIMIT`	タスクの起動数が多すぎます。そのため、この呼び出しによるタスクの起動要求はすべて無視されます。
Extended	`E_OS_ID`	`ScheduleType ScheduleID` が有効ではありません。
Extended	`E_OS_SYS_S_MISMATCH`	`ScheduleID` はアドバンスドスケジュールです。
Extended	`E_OS_SYS_CONFIG_ERROR`	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

`TickSchedule()` を呼び出した後のリスケジューリングは、呼び出し元の環境により以下のように異なります。

- カテゴリ2のISR： カテゴリ2のISRがターミネートするまではリスケジューリングは行われません。
- ノンプリエンプティブタスク： ノンプリエンプティブタスクがターミネートするか `Schedule()` を呼び出すまではリスケジューリングは行われません。

- プリエンプティブタスク： 起動されたタスクの方が優先度が高い場合は、リスケジューリングが直ちに行われます。
- フック： このフックが終了するまではリスケジューリングは行われません。

タイミング分析を正しく行うには、優先度の高いタスクが起動されるスケジュールが、優先度の低いタスクによってチックされることのないようにしてください。

参照：

```
AdvanceSchedule()
GetArrivalpointDelay()
GetArrivalpointNext()
GetArrivalpointTasksetRef()
GetScheduleNext()
GetScheduleStatus()
GetScheduleValue()
SetArrivalpointDelay()
SetScheduleNext()
StartSchedule()
StopSchedule()
TestArrivalpointWritable()
```

3.75 WaitEvent()

1 つまたは複数のイベントを待ちます。

関数宣言：

```
StatusType WaitEvent(EventMaskType Mask)
```

引数：

引数	入力／出力	説明
Mask	入力	待ち対象のイベントのマスク

説明：

この API 関数は、指定されたイベントのいずれか 1 つがセットされるまで、呼び出し元タスクを待ち状態（'Waiting'）にします。1 つまたは複数のイベントがすでにセットされている場合は、タスクは実行状態（'Running'）のまま、変化しません。

エラーコード：

ビルド	コード	説明
Standard	E_OK	エラーなし
Extended	E_OS_ACCESS	呼び出し元タスクは拡張タスクではありません。
Extended	E_OS_CALLEVEL	割込みレベルから呼び出されました。
Extended	E_OS_RESOURCE	呼び出し元タスクがリソースを占有しています。
Extended	E_OS_SYS_CONFIG_ERROR	コンフィギュレーションエラーです。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	
タスク	✓
カテゴリ 1 割込み	×
カテゴリ 2 割込み	×

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

AUTOSAR
OSEK

注記：

なし

参照：

```
ClearEvent()
DeclareEvent()
GetEvent()
SetEvent()
```

4 コンストラクトエレメント

4.1 DeclareAccessor()

アクセサを宣言します。

関数宣言：

```
DeclareAccessor(AccessorID)
```

引数：

引数	入力／出力	説明
AccessorID	入力	宣言されるアクセサ

説明：

`DeclareAccessor()` は、アクセサの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

`DeclareAccessor()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

なし

4.2 DeclareAlarm()

アラームを宣言します。

関数宣言：

```
DeclareAlarm(AlarmID)
```

引数：

引数	入力／出力	説明
AlarmID	入力	宣言されるアラーム

説明：

`DeclareAlarm()` は、アラームの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

AUROSAR
OSEK

注記：

`DeclareAlarm()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

`CancelAlarm()`
`GetAlarm()`
`GetAlarmBase()`
`SetAbsAlarm()`
`SetRelAlarm()`

4.3 DeclareCounter()

カウンタを宣言します。

関数宣言：

```
DeclareCounter(CounterID)
```

引数：

引数	入力／出力	説明
CounterID	入力	宣言されるカウンタ

説明：

`DeclareCounter()` は、カウンタの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

`DeclareCounter()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

```
GetCounterValue()
IncrementCounter()
InitCounter()
osAdvanceCounter_<CounterID>()
Tick_<CounterID>()
```

4.4 DeclareEvent()

イベントを宣言します。

関数宣言：

```
DeclareEvent(EventID)
```

引数：

引数	入力／出力	説明
EventID	入力	宣言されるイベント

説明：

`DeclareEvent()` は、イベントの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

AUTOSAR
OSEK

注記：

`DeclareEvent()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

`ClearEvent()`
`GetEvent()`
`SetEvent()`
`WaitEvent()`

4.5 DeclareFlag()

フラグを宣言します。

関数宣言：

```
DeclareFlag(FlagID)
```

引数：

引数	入力／出力	説明
FlagID	入力	宣言されるフラグ

説明：

`DeclareFlag()` は、フラグの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

`DeclareFlag()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

`ReadFlag()`
`ResetFlag()`

4.6 DeclareISR()

ISR を宣言します。

関数宣言：

```
DeclareISR(ISRID)
```

引数：

引数	入力／出力	説明
ISRID	入力	宣言される ISR

説明：

`DeclareISR()` は、ISR の外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

`DeclareISR()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

```
DisableAllInterrupts()
EnableAllInterrupts()
ResumeAllInterrupts()
ResumeOSInterrupts()
SuspendAllInterrupts()
SuspendOSInterrupts()
```

4.7 DeclareMessage()

メッセージを宣言します。

関数宣言：

```
DeclareMessage(MessageID)
```

引数：

引数	入力／出力	説明
MessageID	入力	宣言されるメッセージ

説明：

DeclareMessage() は、メッセージの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

DeclareMessage() は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

InitCOM()
MessageInit()
ReadFlag()
ReceiveMessage()
ResetFlag()
SendMessage()
StartCOM()
StopCOM()

4.8 DeclareResource()

リソースを宣言します。

関数宣言：

```
DeclareResource(ResourceID)
```

引数：

引数	入力／出力	説明
ResourceID	入力	宣言されるリソース

説明：

`DeclareResource()` は、リソースの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

AUTOSAR
OSEK

注記：

`DeclareResource()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

`GetResource()`
`ReleaseResource()`

4.9 DeclareTask()

タスクを宣言します。

関数宣言：

```
DeclareTask(TaskID)
```

引数：

引数	入力／出力	説明
TaskID	入力	宣言されるタスク

説明：

`DeclareTask()` は、タスクの外部定数宣言を行います。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

AUTOSAR
OSEK

注記：

`DeclareTask()` は、RTA-OSEK によって適切なファイルに挿入されます。マニュアル操作で挿入する必要はありません。

コンストラクトエレメントをランタイムに呼び出すことはできません。

参照：

`ActivateTask()`
`ChainTask()`
`GetTaskID()`
`GetTaskState()`
`TerminateTask()`

5 アドバンスドカウンタとアドバンスドスケジュール用ドライバインターフェース

アプリケーションプログラマは、アドバンスドスケジュールやアドバンスドカウンタを駆動するチックソースと RTA-OSEK との間のインターフェースとして、所定のデバイスドライバ関数を作成する必要があります。

重要

OIL 設定ファイルに定義された各アドバンスドスケジュールまたはアドバンスドカウンタごとに、それぞれ個別のハードウェアチックソース用デバイスドライバ関数を使用する必要があります。同じチックソースドライバを複数のアドバンスドスケジュールやアドバンスドカウンタで共有することはできません。

アドバンスドスケジュールには以下のドライバ関数が必要です。

- `Set_<ScheduleID>()`
- `State_<ScheduleID>()`
- `Now_<ScheduleID>()`
- `Cancel_<ScheduleID>()`

上記のドライバ関数名の `<ScheduleID>` の部分にはスケジュールの名前が入ります。

同様に、アドバンスドカウンタにも以下のドライバ関数が必要です。

- `Set_<CounterID>()`
- `State_<CounterID>()`
- `Now_<CounterID>()`
- `Cancel_<CounterID>()`

上記のドライバ関数名の `<CounterID>` の部分にはアドバンスドカウンタの名前が入ります。

移植性

OSEK OS と AUTOSAR OS ではチックソースアクセス用の標準 API は定義されていません。本章に記述されているインターフェースは RTA-OSEK 固有のものです。

RTA-OSEK では、これらのドライバ関数をリエントラントな関数としては使用しません。これらの関数は、必ず OS レベルから排他的に呼び出されます。これらの関数をアプリケーションからも直接呼び出す場合は、絶対に OS レベルからしか呼び出されないように注意してください。

これらのドライバ関数から RTA-OSEK の API 関数を呼び出すことはできません。

5.1 チックソースのセマンティックス

アドバンスドスケジュールとアドバンスドカウンタは、以下のような条件を満たすハードウェアチックソースによって稼働します。

- 1 チックごとにインクリメントされる「now 値」を持ち、所定の最大値でラップアラウンドする、フリーランニングカウンタです。
- さらに「match 値」を持っています。
- now 値が match 値と等しくなると、「チックソースの満了」が発生します。満了時に割込みを発生させるように設定することができます。
- チックソースが割込みを発生させないように設定することもできます。（このことを、「カウンタ満了をキャンセルする」と言います。）

一般的に、ハードウェア（ハードウェアカウンタとタイマ出力コンペアチャネル）を使用して、アドバンスドスケジュールやアドバンスドカウンタ用のチックソースを実装することができます。

5.2 初期化

各アドバンスドスケジュール／カウンタは、最初の OS コールが行われるより前にアプリケーションにより以下のように初期化されている必要があります。

- スケジュール／カウンタドライバのチックソースの最大値は、スケジュール／カウンタに割り当てられている最大値（OIL 設定ファイルに定義されています）に相当するものであること。
- チックソースは、チックソースの 1 回のチックが、スケジュール／カウンタのチックレートの 1 チックの長さに相当するように設定されていること。
- チックソースは、インクリメントされるフリーランニングカウンタであること。
- チックソースは、割込みを発生させないように設定されていること。

5.3 Cancel_<ScheduleID>() （ユーザー定義）

未処理のチックソース満了をすべてキャンセルするためのドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(void) Cancel_<ScheduleID>(void)
```

引数：

なし

説明：

関数 `Cancel_<ScheduleID>()` は、未処理のチックソース満了をすべてキャンセルする必要があります。割込みを発生させないようにハードウェアを設定してから、未処理の割込みをすべてクリアするようにしてください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

この関数では、ハードウェアカウンタのインクリメントを停止させる必要はありません。

参照：

```
Now_<ScheduleID>()
Set_<ScheduleID>()
State_<ScheduleID>()
```

5.4 Now_<ScheduleID>() （ユーザー定義）

現在のチックソース値を返すドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(TickType) Now_<ScheduleID>(void)
```

引数：

この関数は、`TickType` 値を返します。

説明：

対応するチックソースの now 値を返してください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

なし

参照：

```
Cancel_<ScheduleID>()
Set_<ScheduleID>()
State_<ScheduleID>()
```

5.5 Set_<ScheduleID>() （ユーザー定義）

アドバンスドスケジュール用の次の match 値をセットするためのドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(void) Set_<ScheduleID>(TickType Match)
```

引数：

引数	入力／出力	説明
Match	入力	次にマッチする絶対値

説明：

関数 Set_<ScheduleID>() は、Match 値を引数として受け取ります。未処理の割込みをすべてクリアし、now 値が新しい Match 値になった時に割込みが発生するようにハードウェアを設定します。従って、Match はスケジュールが次のアライバルポイントを処理する時の絶対値になります。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

Set_<ScheduleID>() は、ハードウェアの初期化を行う必要はありません。ハードウェアの初期化は、RTA-OSEK が起動される前に初期化コード内で行ってください。

参照：

```
Cancel_<ScheduleID>()
Now_<ScheduleID>()
State_<ScheduleID>()
```

5.6 State_<ScheduleID>() （ユーザー定義）

対応するチックソースハードウェアの状態を取得するドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(void)
State_<ScheduleID>(ScheduleStatusRefType State)
```

引数：

引数	入力／出力	説明
State	出力	対応するチックソースのステータス

説明：

この関数は、スケジュールのステータス構造体を更新する必要があります。この関数は、スケジュールが稼働中であるときにのみ呼び出されます。

ScheduleStatusType の詳細については 2.1 項を参照してください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

match が now と等しい場合は、expiry フィールドの値がゼロになります。「ゼロ」がセットされると、その時点からスケジュールの最大チック数が経過した時点で次のカウンタイベントが発生します。

参照：

```
Cancel_<ScheduleID>()
Now_<ScheduleID>()
Set_<ScheduleID>()
```

5.7 Cancel_<CounterID>() （ユーザー定義）

未処理のチックソース満了をすべてキャンセルするためのドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(void) Cancel_<CounterID>(void)
```

引数：

なし

説明：

関数 `Cancel_<CounterID>()` は、未処理のチックソース満了をすべてキャンセルする必要があります。割込みを発生させないようにハードウェアを設定してから、未処理の割込みをすべてクリアするようにしてください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

この関数では、ハードウェアカウンタのインクリメントを停止させる必要はありません。

参照：

```
Now_<CounterID>()
Set_<CounterID>()
State_<CounterID>()
```

5.8 Now_<CounterID>() （ユーザー定義）

現在のチックソース値を返すドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(TickType) Now_<CounterID>(void)
```

引数：

この関数は、`TickType` 値を返します。

説明：

対応するチックソースの now 値を返してください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

なし

参照：

```
Cancel_<CounterID>()
Set_<CounterID>()
State_<CounterID>()
```

5.9 Set_<CounterID>() （ユーザー定義）

アドバンスドカウンタ用の次の match 値をセットするためのドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(void) Set_<CounterID>(TickType Match)
```

引数：

引数	入力／出力	説明
Match	入力	次にマッチする絶対値

説明：

関数 Set_<CounterID>() は、Match 値を引数として受け取ります。未処理の割込みをすべてクリアし、now 値が新しい Match 値になった時に割込みが発生するようにハードウェアを設定します。従って、Match は RTA-OSEK が次のアラームまたはスケジュールテーブルの満了ポイントを処理するタイミングを表す絶対値になります。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

Set_<CounterID>() は、ハードウェアの初期化を行う必要はありません。ハードウェアの初期化は、RTA-OSEK が起動される前に初期化コード内で行ってください。

参照：

```
Cancel_<CounterID>()
Now_<CounterID>()
State_<CounterID>()
```

5.10 State_<CounterID>() （ユーザー定義）

対応するチックソースハードウェアの状態を取得するドライバコールバックルーチンです。

関数宣言：

```
OS_CALLBACK(void)
State_<CounterID>(ScheduleStatusRefType State)
```

引数：

引数	入力／出力	説明
State	出力	対応するチックソースのステータス

説明：

この関数は、スケジュールのステータス構造体を更新する必要があります。この関数は、スケジュールが稼働中であるときにのみ呼び出されます。

`ScheduleStatusType` の詳細については 2.1 項を参照してください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

match が now と等しい場合は、`expiry` フィールドの値がゼロになります。「ゼロ」がセットされると、その時点からスケジュールの最大チック数が経過した時点で次のカウンタイベントが発生します。

参照：

```
Cancel_<CounterID>()
Now_<CounterID>()
Set_<CounterID>()
```

6 実行時間監視

移植性

実行時間監視用の API 関数は、すべて RTA-OSEK に固有のものです。

6.1 GetExecutionTime()

呼び出し元のタスクまたはカテゴリ 2 ISR が、今回の実行おいて消費した実行時間を取得します。

関数宣言：

```
StopwatchTickType GetExecutionTime(void)
```

引数：

なし

説明：

基本タスクまたはカテゴリ 2 ISR によって呼び出され、実行開始以降に経過した時間を返します。

拡張タスクの場合は、呼び出し元タスクからの現在の呼び出しにより消費された実行時間、または実行開始以降に消費された実行時間、あるいは前回の `WaitEvent()` の呼び出し以降に消費された実行時間を返します。

値がオーバーフローした場合は、ラップアラウンドした値が返されます。

エラーコード：

なし

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	×
オーバーラン	×

移植性：

RTA-OSEK

注記：

この API 関数を使用するには、ユーザーが `GetStopwatch()` コールバックルーチンによるフリーランニングカウンタへのアクセスを用意しておく必要があります。

またこの API 関数は、OIL 設定ファイル内で OS ステータスが Standard に設定されている場合は常にゼロを返す必要があります。

参照：

```
GetLargestExecutionTime()
GetStopwatch()
GetStopwatchUncertainty()
ResetLargestExecutionTime()
```

6.2 GetLargestExecutionTime()

指定されたタスクについて計測された、最長の実行時間を取得します。

関数宣言：

```
StatusType GetLargestExecutionTime(TaskType TaskID, StopwatchTickRefType
WCET)
```

引数：

引数	入力／出力	説明
TaskID	入力	タスク名
WCET	出力	ワーストケースの実行時間

説明：

TaskID により特定されるタスクまたはカテゴリ 2 の ISR の実行がターミネートしたときに計測された実行時間のうちで、最も長い時間を返します。この最大値は、そのタスクまたはカテゴリ 2 の ISR について前回 ResetLargestExecutionTime() が呼び出されたとき、または StartOS() が呼び出されたとき以降に完了した、そのタスクまたはカテゴリ 2 の ISR の実行時間の最大値です。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_T_INVALID	タスクまたはカテゴリ 2 の ISR ハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

指定されたタスクの実行が 1 回も完了していない場合は、この API 関数は 0 を返します。

またこの API 関数は、OIL 設定ファイル内で OS ステータスが Standard に設定されている場合は常にゼロを返す必要があります。

参照：

GetExecutionTime()
GetStopwatch()
GetStopwatchUncertainty()
ResetLargestExecutionTime()

6.3 GetStopwatch() （ユーザー定義）

フリーランニングカウンタの現在値を返す RTA-OSEK コールバックルーチンです。

関数宣言：

```
OS_NONREENTRANT(StopwatchTickType) GetStopwatch(void)
```

引数：

戻り値は `StopwatchTickType` です。

説明：

`GetStopwatch()` はフリーランニングタイマ（インクリメントし、上限値に達するとオーバーフローするタイマ）の現在値を返します。このタイマの値は、実行時間測定のための基準時間として使用されるので、この基準時間の単位で時間値を返すようにしてください。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

OIL 設定ファイル内で OS ステータスが Timing または Extended に設定されている場合は、ユーザーが `GetStopwatch()` を定義しておく必要があります。

参照：

```
GetExecutionTime()
GetLargestExecutionTime()
GetStopwatchUncertainty()
ResetLargestExecutionTime()
```

6.4 GetStopwatchUncertainty() （ユーザー定義）

実行時間監視用の `StopwatchUncertainty()` を含む RTA-OSEK コールバックルーチンです。

関数宣言：

```
OS_NONREENTRANT(StopwatchTickType) GetStopwatchUncertainty(void)
```

引数：

不確実性を `StopwatchTickType` で表して返します。

説明：

一部のターゲットでは、CPU サイクルと同じレートで稼働するストップウォッチを使用することができません。その場合、タイミング測定は不確実性を帯びます。

すべてのタイミング測定は、コード内の 2 つのポイントの間に経過した時間を基準としていますが、`GetStopwatch()` クロックが CPU クロックよりも遅い場合、2 つのポイントの時間の引き算を行うだけだと、経過時間は少なく算出されてしまう場合があります。これが、経過時間測定における不確実性です。

`GetStopwatchUncertainty()` は、`GetStopwatch()` に適用される測定の不確実性を返します。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

OIL 設定ファイル内で OS ステータスが Timing または Extended に設定されている場合は、ユーザーが `GetStopwatch()` を定義しておく必要があります。

参照：

```
GetExecutionTime()
GetLargestExecutionTime()
GetStopwatch()
ResetLargestExecutionTime()
```

6.5 ResetLargestExecutionTime()

現在の最長実行時間をリセットします。

関数宣言：

```
StatusType ResetLargestExecutionTime(TaskType TaskID)
```

引数：

引数	入力／出力	説明
TaskID	入力	タスク名

説明：

指定されたタスクまたはカテゴリ 2 の ISR について、現在までに記録された最大実行時間の値を 0 にリセットします。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
Extended	E_OS_SYS_T_INVALID	タスクまたはカテゴリ 2 ISR ハンドルが無効です。

呼び出し元の環境：

環境	有効／無効
アイドルタスク	○
タスク	○
カテゴリ 1 割込み	×
カテゴリ 2 割込み	○

環境（フック）	有効／無効
スタートアップ	×
シャットダウン	×
プリタスク	×
ポストタスク	×
エラー	○
オーバーラン	○

移植性：

RTA-OSEK

注記：

OIL 設定ファイル内で OS ステータスが Standard に設定されている場合、この API 関数は機能しません。

参照：

```
GetExecutionTime()
GetLargestExecutionTime()
GetStopwatch()
GetStopwatchUncertainty()
```

7 フックルーチン

システムがフックルーチンを使用する設定になっている場合、RTA-OSEK が呼び出すコールバックルーチンをユーザー定義しておく必要があります。

7.1 ErrorHook() （ユーザー定義）

RTA-OSEK API の使用法が適切でなかった場合に発生するエラーを捕捉するために使用されるフックルーチンです。

関数宣言：

```
OS_HOOK(void) ErrorHook(StatusType Error)
```

引数：

引数	入力／出力	説明
Error	入力	発生したエラー

説明：

このフックは、API 関数が E_OK 以外の StatusType を返した時、ユーザーレベルに戻る前に RTA-OSEK により呼び出されます。RTA-OSEK は StatusType を ErrorHook() に渡します。

このフックは、ErrorHook() から呼び出される API 関数がステータス値として E_OK を返さない場合には呼び出されません。

ErrorHook() 内では、情報を取得するためのマクロを使用できます。これらのマクロを使用することにより、ErrorHook() を呼び出した API 関数と、その API 関数に渡された引数についての情報を調べることができます。

- マクロ OSErrorGetServiceID は、値が OSServiceId_xxx の OSServiceIDType のデータを返します。値の xxx は API 関数名です。これは、どの API 関数でエラーが発生したかを明らかにするために使用されます。
- 各コールごとに OSError_APICallName_Parameter() マクロを使用できます。これらは、各 API 関数に渡された引数を明らかにするために使用されます。

これらのマクロは ErrorHook() 内だけで使用でき、これらを使用するためには RTA-OSEK を適切に設定しておく必要があります。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

なし

参照：

OverrunHook()

7.2 OverrunHook() （ユーザー定義）

割り当てられたバジェットタイムのオーバーランを捕捉するためのフックルーチンです。

関数宣言：

```
OS_HOOK(void) OverrunHook(void)
```

引数：

なし

説明：

このフックルーチンは、タイミング（'Timing'）ビルドまたは拡張（'Extended'）ビルドにおいて、タスク実行時間が、そのタスクに割り当てられたバジェットタイム（実行許容時間）を超過した時に呼び出されます。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

OIL 設定ファイル内で OS ステータスが Timing または Extended に設定されている場合、実行時間監視を行うかどうかには関わらず、ユーザーは必ず `OverrunHook()` を定義しておく必要があります。

参照：

```
GetExecutionTime()
GetLargestExecutionTime()
GetStopwatch()
GetStopwatchUncertainty()
```

7.3 MessageInit() （ユーザー定義）

ユーザーメッセージデータ構造体を初期化するフックルーチンです。

関数宣言：

```
StatusType MessageInit(void)
```

引数：

なし

説明：

このルーチンは、アプリケーション固有のすべてのメッセージオブジェクトを初期化します。

エラーコード：

ビルド	コード	説明
All	E_OK	エラーなし
	（その他の値）	初期化が正常に終了しなかった場合は、アプリケーションに固有のエラーコードを返します。

呼び出し元の環境：

なし

移植性：

OSEK

注記：

MessageInit() 関数は、アプリケーションプログラマにより用意されている必要があり、また StartCOM() ルーチンからしか呼び出すことはできません。

MessageInit() により返された StatusType は、その後 StartCOM() から返されます。RTA-OSEK では、ユーザーが独自の MessageInit() を定義していない場合には、MessageInit() のデフォルトコールが実装されます。このデフォルトコールの処理は、E_OK を返すだけです。

参照：

DeclareMessage()
ClearEvent()

7.4 PostTaskHook() （ユーザー定義）

タスクがレディ状態（'Ready'）またはサスペンド状態に入る時に呼び出されるフックルーチンです。

関数宣言：

```
OS_HOOK(void) PostTaskHook(void)
```

引数：

なし

説明：

このフックルーチンは、現在のタスクの実行が終了した後、オペレーティングシステムによってそのタスクが実行状態（'Running'）でなくなる前に `TaskID` を評価する必要があるときに使用します。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

このルーチンは、タスクのプリエンプションが発生するたびに呼び出されます。

参照：

`PreTaskHook()`

7.5 PreTaskHook() （ユーザー定義）

タスクが実行状態（'Running'）からレディ状態（'Ready'）またはサスペンド状態（'Suspended'）に入る時に呼び出されるフックルーチンです。

関数宣言：

```
OS_HOOK(void) PreTaskHook(void)
```

引数：

なし

説明：

このフックルーチンは、オペレーティングシステムが新しいタスクを実行する際、そのタスクが実行状態に移行した後に `TaskID` を評価するために呼び出されます。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

このルーチンは、タスクのプリエンプションが発生するたびに呼び出されます。

参照：

`PostTaskHook()`

7.6 ShutdownHook() （ユーザー定義）

オペレーティングシステムのシャットダウン時に呼び出されるフックルーチンです。

関数宣言：

```
OS_HOOK(void) ShutdownHook(StatusType Error)
```

引数：

引数	入力／出力	説明
Error	入力	発生したエラー

説明：

ShutdownHook() が定義されている場合、このフックルーチンは、ShutdownOS() という OS API 関数が呼び出された時、オペレーティングシステムのシャットダウン処理中にオペレーティングシステムにより呼び出されます。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

ShutdownOS() の内部呼び出しで発生する可能性のあるエラー引数値は、以下のとおりです。

- E_OS_SYS_STACK_FAULT：アプリケーションの処理が StackFaultHook から戻りました。
- E_OS_SYS_CALLEVEL：アプリケーションが、OS レベルより高い不正な OS サービスを使用しました。

参照：

ShutdownOS()

7.7 StackFaultHook()（ユーザー定義）

いずれかのタスクについて、スタックサイズが設定されているサイズを超えたときに呼び出されます。

関数宣言：

OS_HOOK(void) StackFaultHook(SmallType StackID,
SmallType StackError,
UIntType Overflow)

引数：

引数	入力／出力	説明
StackID	入力	データが出し入れされるスタックを示します。
StackError	入力	（*下の説明を参照してください。*）
Overflow	入力	システムに設定されている限界よりも多いスタックバイト数

説明：

StackFaultHook() は、拡張タスクが存在する場合には必ず定義しておく必要があります。このルーチンは、いずれかのタスクについて、スタックサイズが設定されたサイズを超えた時に呼び出されます。

StackFaultHook() は通常、タスクのスタックサイズが不正に設定されていたときに呼び出されますが、アプリケーションの欠陥でスタックオーバフローが発生したような時にも呼び出されます。

StackError は以下のいずれかの値です。

- OS_EXTENDED_TASK_STARTING：拡張タスクがディスパッチ（実行開始）されています。
- OS_EXTENDED_TASK_RESUMING：WaitEvent() を呼び出した後、拡張タスクの処理が再開されています。
- OS_EXTENDED_TASK_WAITING：拡張タスクが WaitEvent() を呼び出しています。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

RTA-OSEK

注記：

StackID は、各ターゲットの『RTA-OSEK バインディングマニュアル』に特に定義されていない場合には、常に 0 です。

このフックルーチン内で呼び出す必要のある API 関数は、GetTaskID() と GetActiveApplicationMode() だけです。

参照：

なし

7.8 StartupHook() （ユーザー定義）

OS の初期化の後、スケジューラが稼働する前に呼び出されるフックルーチンです。

関数宣言：

```
OS_HOOK(void) StartupHook(void)
```

引数：

なし

説明：

`StartupHook()` が設定されている場合、このフックルーチンはオペレーティングシステムの初期化が終わってスケジューラが稼働する前に、オペレーティングシステムにより呼び出されます。このルーチン内で、アプリケーションはタスクの起動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。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

なし

参照：

`StartOS()`

8 その他のコールバックルーチン

8.1 ALARMCALLBACK() （ユーザー定義）

アラーム満了時に、そのアラームにコールバックが設定されている場合に呼び出されます。

関数宣言：

```
void ALARMCALLBACK(AlarmcallbackRoutineName)
```

引数：

なし

説明：

これは、アラーム満了時に、そのアラームにコールバックが設定されている場合に RTA-OSEK により呼び出されます。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

呼び出せる RTA-OSEK API 関数は、`ResumeAllInterrupts()` と `SuspendAllInterrupts()` だけです。

参照：

`osAdvanceCounter_<CounterID>()`

`Tick_<CounterID>()`

8.2 COMCALLBACK() （ユーザー定義）

メッセージ受信時に、そのメッセージにコールバックが設定されている場合に呼び出されます。

関数宣言：

```
COMCALLBACK(COMcallbackRoutineName)
```

引数：

なし

説明：

これは、メッセージを受信した時に、そのメッセージにコールバックが設定されている場合に RTA-OSEK により呼び出されます。

エラーコード：

なし

呼び出し元の環境：

なし

移植性：

OSEK

注記：

呼び出せる API 関数は、`SendMessage()`、`ReceiveMessage()`、`GetMessageStatus()` だけです。

参照：

なし

9 定義済みオブジェクト

RTA-OSEK は、設定ファイルに明示的に宣言されているオブジェクト以外に、以下のシンボル、オブジェクト、ハンドルも生成します。

9.1 OSEK カウンタ属性

オブジェクト	説明	OSEK
OS_CYCLES_PER_x	カウンタ x の 1 チックあたりの CPU サイクル数	
OS_CYCLES_PER_SWTICK	システムカウンタの 1 チックあたりの CPU サイクル数	
OSMAXALLOWEDVALUE	システムカウンタの最大値（単位はチック）	✓
OSMAXALLOWEDVALUE_x	カウンタ x の最大値（単位はチック）	✓
OSMINCYCLE	システムカウンタの最小値（単位はチック）	✓
OSMINCYCLE_x	カウンタ x の最小値（単位はチック）	✓
OS_NS_PER_CYCLE	CPU サイクルの間隔（単位はナノ秒）	
OSTICKDURATION	システムカウンタの 1 チックの長さ（単位はナノ秒）	✓
OSTICKDURATION_x	カウンタ x の 1 チックの長さ（単位はナノ秒）	
OSTICKSPERBASE	システムカウンタの指定の単位が経過するのに必要なチック数	✓
OSTICKSPERBASE_x	カウンタ x の指定の単位が経過するのに必要なチック数	✓

9.2 OSEK タスクステート

オブジェクト	説明
READY	Ready というタスク状態を示す TaskStateType データ型の定数
RUNNING	Running というタスク状態を示す TaskStateType 型の定数
SUSPENDED	Suspended というタスク状態を示す TaskStateType 型の定数
WAITING	Waiting というタスク状態を示す TaskStateType 型の定数。BCC タスクについては使用されません。

9.3 OSEK リソース

オブジェクト	説明
RES_SCHEDULER	ResourceType 型の定数。すべてのタスクにより共有されます。

9.4 OSEK アプリケーションモード

オブジェクト	説明
OSDEFAULTAPPMODE	デフォルトのアプリケーションモード。StartOS() の引数として使用できます。

9.5 RTA-OSEK ビルドレベル

オブジェクト	説明
OS_EXTENDED_BUILD	RTA-OSEK 設定ツールにより拡張（'Extended'）ビルドにおいて生成されるマクロ
OS_STANDARD_BUILD	RTA-OSEK 設定ツールにより標準（'Standard'）ビルドにおいて生成されるマクロ
OS_TIMING_BUILD	RTA-OSEK 設定ツールによりタイミング（'Timing'）ビルドにおいて生成されるマクロ
OS_ET_MEASURE	RTA-OSEK 設定ツールによりタイミング（'Timing'）および拡張（'Extended'）ビルドにおいて生成されるマクロ

9.6 RTA-OSEK タスクセット

パラメータ	説明
os_all_tasks	定義されているすべてのタスク（アイドルタスクを含む）が含まれる読み取り専用のタスクセット
os_no_tasks	タスクが 1 つも含まれない、読み取り専用タスクセット
os_ready_tasks	os_ready_tasks は読み取り専用タスクセット。その内容は RTA-OSEK により変更され、レディ状態（'Ready'）と実行状態（'Running'）のすべてのタスクを示します。
osek_cc2_tasks	すべての BCC2 タスクと ECC2 タスクが含まれるタスクセット。BCC2 タスクまたは ECC2 タスクが存在する場合、または拡張（'Extended'）ビルドの場合に限り定義されます。
osek_ecc_tasks	すべての ECC1 タスクと ECC2 タスクが含まれるタスクセット。ECC1 タスクまたは ECC2 タスクが存在する場合、または拡張（'Extended'）ビルドの場合に限り定義されます。

9.7 RTA-OSEK アプリケーション特性値

アプリケーションの特性を表す以下のシンボルが osekcomn.h に定義されます。

シンボル	説明
OSEK_BCC1	BCC1 タスクのみが含まれます。
OSEK_BCC2	BCC2 タスクが含まれます。
OSEK_BCC2C	専用の優先度を持つ BCC2 タスクが含まれます。
OSEK_BCC2F	共有の優先度を持つ BCC2 タスクが含まれます。
OSEK_ECC1	ECC1 タスクが含まれます。
OSEK_ECC2	ECC2 タスクが含まれます。
OSEK_ECC2C	専用優先度を持つ BCC2 タスクが含まれ、さらに ECC1 タスクが含まれます。
OSEK_ECC2F	「共有優先度を持つ ECC2 タスク」と「共有優先度を持つ BCC2 タスクと ECC タスク」のいずれか一方、または両方が含まれます。
OS_STC_COMPATIBLE	STC 準拠、つまり、RTA-OSEK Planner がタイミング挙動を分析できる形式でシステムが構築されていることを示します。詳しくは『RTA-OSEK ユーザーズガイド』を参照してください。
OS_NO_SCHEDULECALL	アプリケーションが Schedule() をコールしないことを示します。

シンボル内の "F" と "C" の意味

アプリケーションに起動をキューイングでき、他のタスクと優先度を共有するタスクが含まれている場合、タスク起動は FIFO キューに格納されます。この場合、`OSEK_BCC2F` または `OSEK_ECC2F` が定義されます。

アプリケーションが共有優先度を使用しない場合は、RTA-OSEK は「カウントされる起動」（'counted activation'）と呼ばれる最適化された方式を使用します。この場合は `OSEK_BCC2C` または `OSEK_ECC2C` が定義されます。

10 マクロ定義

マクロ	説明
ALARMCALLBACK()	アラーム満了時に呼び出されるコールバックルーチンのフォーマットです。

ISR()	アプリケーションのカテゴリ 2 の ISR は必ず以下のフォーマットで使用してください。
	`ISR(IsrID)`
	`{`
	`...`
	`}`

OSError_APICallName_Parameter()	各 API コールに対して使用できます。API 関数の呼び出し時に渡された引数を取得できます。
OSErrorGetServiceID	どの API 関数でエラーが発生したかを明らかにします。値が OSServiceId_xxx（xxx は API 関数名）の OSServiceIDType を返します。
OS_ATOMIC(expr)	すべての割込みを一時的にディセーブルにし、C の式 expr を評価します。
OS_CALLBACK()	RTA-OSEK から直接呼び出されるユーザー関数であることを示します。 関数宣言：OS_CALLBACK(void) Cancel_<ScheduleID>(void);
OS_HOOK()	OS によりサポートされているフックルーチンであることを示します。 関数宣言： OS_HOOK(void) ErrorHook(StatusType Error)
OS_MAIN()	移植可能な形式のメイン関数であることを示します。 関数宣言：OS_MAIN()
OS_NONREENTRANT()	リエントラントでないユーザー関数であることを示します。 関数宣言：OS_NONREENTRANT(void) GetStopwatch(void);
OS_STATUS_RUNNING	スケジュールが稼動中であることを示します。 #define OS_STATUS_RUNNING (SmallType (2))
OS_STATUS_PENDING	アライバルポイントの処理をするべき時が来ているのにかかわらず、スケジュールが処理待ち状態であることを示します。 #define OS_STATUS_PENDING (SmallType (1))

TASK()	アプリケーションのタスクは以下のフォーマットで使用してください。
	`TASK(TaskID)`
	`{`
	`...`
	`}`

11 クイックリファレンスガイド

11.1 RTA-OSEK API 関数（動的インターフェース）

名前	説明
ActivateTask()	タスクを起動します。
ActivateTaskset()	タスクセットを起動します。
AdvanceSchedule()	アドバンスドスケジュールの次のアライバルポイントを処理します。
AssignTaskset()	指定のタスクセットのメンバを別のタスクセットに割り当てます。
CancelAlarm()	アラームをキャンセルします。
ChainTask()	呼び出し元タスクをターミネートし、指定のタスクを起動します。
ChainTaskset()	呼び出し元タスクをターミネートし、指定のタスクセットを起動します。
ClearEvent()	呼び出し元タスクのイベントをクリアします。
CloseCOM()	COM 通信が使用する低レベルのハードウェアリソースを解放します。
DisableAllInterrupts()	すべての割込みをディセーブルにします。
EnableAllInterrupts()	DisableAllInterrupts() で開始されたクリティカルセクションを終わらせます。
GetActiveApplicationMode()	現在のアプリケーションモードを取得します。
GetAlarm()	アラームの次回の満了までのチック数を返します。
GetAlarmBase()	アラームの基本特性を取得します。
GetArrivalpointDelay()	指定のアライバルポイントから次のアライバルポイントまでの遅延時間を取得します。
GetArrivalpointNext()	指定のアライバルポイントの次のアライバルポイントを取得します。
GetArrivalpointTasksetRef()	アライバルポイントのタスクセットへのポインタを取得します。
GetCounterValue()	カウンタ値を取得します。
GetEvent()	指定されたタスク用のイベントを取得します。
GetISRID()	現在の ISR のハンドルを取得します。
GetMessageResource()	メッセージリソースを取得します。
GetMessageStatus()	メッセージステータスを返します。
GetResource()	リソースを取得します。
GetScheduleNext()	スケジュールにより処理される次のアライバルポイントを取得します。
GetScheduleStatus()	スケジュールの現在の状態を取得します。
GetScheduleTableStatus()	スケジュールテーブルの現在の状態を取得します。
GetScheduleValue()	スケジュールの Now プロパティを取得します。
GetStackOffset()	現在のスタックポインタを取得します。
GetTaskID()	指定されたタスクのステータスを取得します。
GetTasksetRef()	指定されたタスクだけを含むタスクセットを 1 つ作成します。
GetTaskState()	タスクの状態を取得します。
IncrementCounter()	カウンタを 1 チックだけインクリメントします。
InitCOM()	COM が使用する低レベルリソースを初期化します。

名前	説明
InitCounter()	カウンタに所定のチック値を初期設定します。
MergeTaskset()	2 つのタスクセットをマージします。
NextScheduleTable()	現在稼働中のスケジュールテーブルを停止して別のスケジュールテーブルを起動します。
osAdvanceCounter_<CounterID>()	カウンタに関連付けられた次のアラーム、またはスケジュールテーブルの満了ポイントを処理します。
osResetOS()	システムを再起動する際、この関数で OS 変数をリセットします。
ReadFlag()	OSEK COM メッセージに関連付けられているメッセージフラグのステータスを読み取ります。
ReceiveMessage()	指定された OSEK COM メッセージを受信します。
ReleaseMessageResource()	すでに保持されているメッセージリソースを解放します。
ReleaseResource()	すでに保持されているリソースを解放します。
RemoveTaskset()	タスクセットからタスクを除外します。
ResetFlag()	OSEK COM メッセージに関連付けられているフラグをリセットします。
ResumeAllInterrupts()	割込みの処理を再開します。
ResumeOSInterrupts()	カテゴリ 2 割込みの割込み処理を再開します。
Schedule()	ノンプリエンプティブタスク用のリスケジューリングを行います。
SendMessage()	指定されたデータを OSEK COM メッセージとして送信します。
SetAbsAlarm()	アラーム満了時のカウンタ値を設定します。
SetArrivalpointDelay()	指定のアライバルポイントから次のアライバルポイント間の遅延時間を設定します。
SetArrivalpointNext()	アライバルポイントの次のアライバルポイントを変更し、スケジュールを再設定します。
SetEvent()	指定されたタスク用のイベントをセットします。
SetRelAlarm()	アラーム満了までに発生するカウンタチック数を設定します。
SetScheduleNext()	スケジュールに次のアライバルポイントを知らせます。
ShutdownOS()	オペレーティングシステムをシャットダウンします。
StartCOM()	COM サービスを開始します。
StartOS()	OS を起動します。
StartSchedule()	停止していたスケジュールを開始します。
StartScheduleTable()	停止していたスケジュールテーブルを開始します。
StopCOM()	COM サービスを停止します。
StopSchedule()	現在実行状態であるスケジュールを停止します。
StopScheduleTable()	現在実行状態であるスケジュールテーブルを停止します。
SuspendAllInterrupts()	すべての割込みの割込み処理をディセーブルにします。
SuspendOSInterrupts()	カテゴリ 2 割込みの割込み処理をディセーブルにします。
TerminateTask()	呼び出し元タスクをターミネートします。
TestArrivalpointWritable()	アライバルポイントが書き込み可能かどうかをテストします。
TestEquivalentTaskset()	2 つのタスクセットに含まれているタスクがまったく同じかどうかをテストします。

名前	説明
TestSubTaskset()	1 つまたは複数の所定のタスクがタスクセット内にあるかどうかをテストします。
Tick_<CounterID>()	カウンタを 1 チックだけインクリメントします。
TickSchedule()	チックスケジュールの now 値を進め、必要に応じてタスクを起動します。
WaitEvent()	1 つまたは複数のイベントを待ちます。

11.2 RTA-OSEK API 関数（静的インターフェース）

名前	説明
ActivateTask_<TaskID>()	タスクを起動します。
ActivateTaskset_<TasksetID>()	タスクセットを起動します。
AdvanceSchedule_<ScheduleID>()	アドバンスドスケジュールの次のアライバルポイントを処理します。
ChainTask_<TaskID>()	呼び出し元タスクをターミネートし、指定ののタスクを起動します。
ChainTaskset_<TasksetID>()	呼び出し元タスクをターミネートし、指定のタスクセットを起動します。
GetResource_<ResID>()	リソースを取得します。
ReleaseResource_<ResID>()	すでに保持されているリソースを解放します。
SendMessage_<Message>_<Data>()	指定されたデータを OSEK COM メッセージとして送信します。
SetEvent_<TaskID>_<Mask>()	指定されたタスク用のイベントをセットします。
TickSchedule_<ScheduleID>()	チックドスケジュールの now 値を進め、必要に応じてタスクを起動します。

11.3 コンストラクトエレメント

名前	説明
DeclareAccessor()	アクセサを宣言します。
DeclareAlarm()	アラームを宣言します。
DeclareCounter()	カウンタを宣言します。
DeclareEvent()	イベントを宣言します。
DeclareFlag()	フラグを宣言します。
DeclareISR()	ISR を宣言します。
DeclareMessage()	メッセージを宣言します。
DeclareProcess()	プロセスを宣言します。
DeclareResource()	リソースを宣言します。
DeclareTask()	タスクを宣言します。

11.4 アドバンスドスケジュールドライバインターフェース

名前	説明
Cancel_<ScheduleID>()	未処理のカウンタ満了をすべてキャンセルするためのドライバコールバックルーチンです。
Now_<ScheduleID>()	現在のカウンタ値を返すドライバコールバックルーチンです。
Set_<ScheduleID>()	アドバンスドスケジュールドライバ用の次の match 値を設定するドライバコールバックルーチンです。
State_<ScheduleID>()	使用されるカウンタ／比較ハードウェアの状態を取得するドライバコールバックルーチンです。

11.5 アドバンスドカウンタドライバインターフェース

名前	説明
Cancel_<CounterID>()	未処理のカウンタ満了をすべてキャンセルするためのドライバコールバックルーチンです。
Now_<CounterID>()	現在のカウンタ値を返すドライバコールバックルーチンです。
Set_<CounterID>()	アドバンスドカウンタドライバ用の次の match 値を設定するドライバコールバックルーチンです。
State_<CounterID>()	使用されるカウンタ／比較ハードウェアの状態を取得するドライバコールバックルーチンです。

11.6 実行時間監視インターフェース

名前	説明
GetExecutionTime()	呼び出し元のタスクまたはカテゴリ 2 の ISR のために消費された実行時間を取得します。
GetLargestExecutionTime()	指定されたタスクについて現在までに測定された最大実行時間を取得します。
GetStopwatch()	フリーランニングカウンタの現在値を返す RTA-OSEK コールバックルーチンです。
GetStopwatchUncertainty()	実行時間モニタリング用の StopwatchUncertainty() を含むコールバックルーチンです。
ResetLargestExecutionTime()	現在までの最大実行時間をリセットします。

11.7 フック

名前	説明
ErrorHook()	RTA-OSEK API の不適切な使用により発生したエラーを捕捉するために使用されるフックルーチンです。
OverrunHook()	実行バジェットオーバーランを捕捉するためのフックルーチンです。
MessageInit()	ユーザーメッセージデータ構造体を初期化するフックルーチンです。
PostTaskHook()	タスクがレディ状態（'Ready'）またはサスペンド状態（'Suspended'）に入る時に使用されるフックルーチンです。

名前	説明
PreTaskHook()	タスクが実行状態（'Running'）からレディ状態（'*Ready*'）またはサスペンド状態（'Suspended'）に入る時に使用されるフックルーチンです。
ShutdownHook()	オペレーティングシステムのシャットダウン時に呼び出されるフックルーチンです。
StackFaultHook()	いずれかのタスクのスタックサイズが、設定されているサイズよりも大きくなった場合に呼び出されます。
StartupHook()	OS 初期化の後、スケジューラが稼働する前に呼び出されるフックルーチンです。

11.8 その他のコールバック

名前	説明
ALARMCALLBACK (AlarmcallbackRoutineName)	アラーム満了時に、そのアラームにコールバックが設定されている場合に呼び出されます。
COMCALLBACK(COMcallbackRoutinName)	メッセージ受信時に、そのメッセージにコールバックが設定されている場合に呼び出されます。

11.9 定義済みオブジェクト

RTA-OSEK は、設定ファイルに明示的に宣されているオブジェクト以外に、以下のシンボル、オブジェクトおよびハンドルも生成します。

名前	説明
os_all_tasks	アイドルタスクを含む、定義されているすべてのタスクが含まれる読み取り専用のタスクセットです。
OS_EXTENDED_BUILD	RTA-OSEK 設定ツールにより拡張（'Extended'）ビルドで生成されるマクロです。
os_no_tasks	タスクが 1 つも含まれない、読み取り専用タスクセットです。アイドルタスクも含まれません。
os_ready_tasks	os_ready_tasks は読み取り専用タスクセットですが、その内容は RTA-OSEK により変更され、レディ状態（'Ready'）のすべてのタスクと実行状態（'Running'）のすべてのタスクを示します。
OS_STANDARD_BUILD	RTA-OSEK 設定ツールにより標準（'Standard'）ビルドで生成されるマクロです。
OS_TIMING_BUILD	RTA-OSEK 設定ツールによりタイミング（'Timing'）ビルドで生成されるマクロです。
OSDEFAULTAPPMODE	デフォルトのアプリケーションモードです。StartOS() の引数として、常に使用できます。
osek_cc2_tasks	すべての BCC2 タスクと ECC2 タスクが含まれるタスクセットです。BCC2 タスクまたは ECC2 タスクが存在する場合、または拡張（'Extended'）ビルドの場合に限り定義されます。
osek_ecc_tasks	すべての ECC1 タスクと ECC2 タスクを示します。ECC1 タスクまたは ECC2 タスクが存在する場合、または拡張（'Extended'）ビルドの場合に限り定義されます。
OSMAXALLOWEDVALUE	システムカウンタに許容される最大値（単位はチック）です。
OSMAXALLOWEDVALUE_x	カウンタ x に許容される最大値（単位はチック）です。
OSMINCYCLE	システムカウンタのサイクリックアラームに許容される最小チック数です。
OSMINCYCLE_x	カウンタ x のサイクリックアラームに許容される最小チック数です。

名前	説明
OSTICKDURATION	システムカウンタの 1 チックの長さ（単位はナノ秒）です。
OSTICKSPERBASE	システムカウンタが指定のユニットになるために必要なチック数です。
OSTICKSPERBASE_x	カウンタ x が指定のユニットになるために必要なチック数です。
READY	Ready というタスク状態を示す TaskStateType データ型の定数です。
RES_SCHEDULER	ResourceType 型の定数です。すべてのタスクにより共有されます。
RUNNING	Running というタスク状態を示す TaskStateType 型の定数です。
SUSPENDED	Suspended というタスク状態を示す TaskStateType 型の定数です。
WAITING	Waiting というタスク状態を示す TaskStateType 型の定数です。BCC タスクについては使用されません。

11.10 マクロ定義

名前	説明
ALARMCALLBACK()	アラーム満了時に呼び出されるコールバックルーチンのフォーマットです。

ISR()	アプリケーションのカテゴリ 2 の ISR は必ず以下のフォーマットで使用してください。
	`ISR(IsrID)`
	`{`
	`  ...`
	`}`

OSError_APICallName_Parameter()	各 API コールに対して使用できます。API 関数の呼び出し時に渡された引数を取得できます。
OSErrorGetServiceID	どの API 関数でエラーが発生したかを明らかにします。値が OSServiceId_xxx（xxx は API 関数名）の OSServiceIDType を返します。
OS_ATOMIC(expr)	すべての割込みを一時的にディセーブルにして、C の式 expr を評価します。
OS_CALLBACK()	RTA-OSEK から直接呼び出されるユーザー関数であることを示します。 関数宣言： `OS_CALLBACK(void) Cancel_<ScheduleID>(void);`
OS_HOOK()	OS によりサポートされているフックルーチンであることを示します。 `OS_HOOK(void) ErrorHook(StatusType Error)`
OS_MAIN()	移植可能な形式のメイン関数であることを示します。 関数宣言：OS_MAIN()

名前	説明
OS_NONREENTRANT()	リエントラントでないユーザー関数であることを示します。 関数宣言： OS_NONREENTRANT(void) GetStopwatch(void);
OS_STATUS_RUNNING	スケジュールが稼動中であることを示します。 #define OS_STATUS_RUNNING (SmallType (2))
OS_STATUS_PENDING	アライバルポイントの処理をするべき時が来ているのにかかわらず、スケジュールが処理待ち状態であることを示します。 #define OS_STATUS_PENDING (SmallType (1))

TASK()	アプリケーションのタスクは以下のフォーマットで使用してください。
	TASK(TaskID)
	{
	...
	}

11.11 エラーコード

名前	説明
E_COM_BUSY	メッセージがアプリケーションのタスク／関数により使用されています。
E_COM_ID	無効なメッセージ名が引数として渡されました。
E_COM_LIMIT	待ち行列メッセージに関連付けられている FIFO がオーバーフローしました。
E_COM_LOCKED	サービスコールが拒否されました。メッセージオブジェクトは保持されています。
E_COM_NOMSG	使用できるメッセージがありません。
E_COM_SYS_STOPPED	COM が起動していません。
E_OK	エラーなし
E_OS_ACCESS	リソースがすでに保持されています。あるいは、呼び出し元のタスクまたはカテゴリ 2 ISR に割り当てられている優先度の方が算出されたシーリング優先度よりも高くなっています。
E_OS_CALLEVEL	この API をカテゴリ 2 ISR から呼び出すことはできません。
E_OS_ID	指定されたオブジェクト識別子は無効です。
E_OS_LIMIT	定義されているタスクの起動数の限界を超えました。
E_OS_NOFUNC	API の機能を完了できませんでした。
E_OS_RESOURCE	リソースがまだ保持されています。このサービスを呼び出す前にリソースを解放する必要があります。
E_OS_STATE	不適切な状態のオブジェクトに対してアクションを実行しようとしました。
E_OS_SYS_AP_INVALID	アライバルポイントが無効です。
E_OS_SYS_AP_NULL	アライバルポイントがヌルです。
E_OS_SYS_AP_READONLY	読み取り専用のアライバルポイントを変更しようとしました。
E_OS_SYS_CALLEVEL	OS レベルより高い不正な OS サービスをアプリケーションが使用しました。

名前	説明
E_OS_SYS_CONFIG_ERROR	不適切な API コールです。 これは、以下のような理由で発生する可能性があります。 ² この API 関数は呼び出し元より優先度の高いタスクを起動しましたが、アプリケーションはどのタスクも自分より優先度の高いタスクを起動しないように最適化されています。 ² タスクまたはカテゴリ 2 ISR が、使用するように設定されていないリソースを占有しようとしました。 ² アプリケーションの C ファイルが、現在の設定にとっては内容が古い RTA-OSEK ヘッダファイルをインクルードしたため、不正な API コールが行われました。 ² アプリケーションの C ファイルが、アプリケーションコードにとって不適切な RTA-OSEK ヘッダファイルをインクルードしたため、不正な API コールが行われました。
E_OS_SYS_COUNTER_INVALID	カウンタが無効です。
E_OS_SYS_IDLE	アイドルタスクからの呼び出しは許されていません。
E_OS_SYS_R_PERMISSION	メッセージリソースを使用することが宣言されていないタスクにより呼び出されました。
E_OS_SYS_S_INVALID	スケジュールハンドルが無効です。
E_OS_SYS_S_MISMATCH	スケジュールに、チック／アドバンスドカウンタが含まれています（呼び出しはアドバンス／チックについてのみ認められます）。
E_OS_SYS_S_MODULO	遅延時間がスケジュールの最大値より大きくなっています。
E_OS_SYS_STACK_FAULT	アプリケーションに、StackFaultHook() からコントロールが戻ってきました。
E_OS_SYS_T_INVALID	タスクまたはカテゴリ 2 ISR ハンドルが無効です。
E_OS_SYS_TS_INVALID	タスクセットハンドルが無効です。
E_OS_SYS_TS_READONLY	タスクセットが読み取り専用になっています。
E_OS_VALUE	指定されたアラーム値は、定義されているカウンタ限界値を超えています。

12 アプリケーションビルドリファレンス

12.1 コマンドラインオプション

以下に、コマンドラインから rtabuild に渡すことができるオプションを示します。

-a、-p、-s オプションのうちのどれも指定されない場合、rtabuild は自動的にカーネルとアプリケーションサポートファイルをビルドします。

12.1.1 一般的なオプション

オプション	説明
-e	ワーニングをエラーとして扱います。ワーニングが出力された場合は、rtabuild はエラー E0402 を出力してから終了します。
-i<path>	インクルードファイルの検索パスを追加します。
-k	中間ファイルを保持します。rtabuild は実行中に、他のコマンドラインツールに渡される情報が設定されているファイルを作成する場合がありますが、通常は、それらのファイルは使用後に削除されます。
-o[name]	リストファイルを生成します。name が指定されない場合、リストファイル名は最初の入力ファイル名のルート部分に .lst というファイル拡張子を付けたものになります。このリストファイルの内容には、呼び出されたコマンドラインなどのコメントを含む、起動されたプログラムからのすべての出力が含まれます。
-v	処理中の詳細な情報表示をオンにします。
-w[vvvv]	すべてのワーニングを無視する（-w）か、指定された番号のワーニング（-wvvvv）を無視します。すべてのワーニングが無視される場合は、ワーニングは画面にもリストファイルにもまったく表示されません。指定されたワーニング vvvv が無視される場合は、vvvv の文字列がワーニングメッセージコードとのマッチングに用いられます。一致するメッセージが発生した場合、そのワーニングは画面とリストファイル出力から除外されます。ワーニングは、-e オプションが効果を生じるようになるまでの間無視されます。

12.1.2 ビルドオプション

オプション	説明
-d<status>	入力 OIL ファイルに定義されているビルドステータスをオーバライドします。 <status> = s： 標準（'STANDARD'） <status> = t： タイミング（'TIMING'） <status> = e： 拡張（'EXTENDED'） <status> = ts： 標準 + 簡単な RTA-TRACE <status> = tt： タイミング + 簡単な RTA-TRACE <status> = te： 拡張 + 簡単な RTA-TRACE <status> = att： タイミング + 高度な RTA-TRACE <status> = ate： 拡張 + 高度な RTA-TRACE
-g	ベクタテーブルの生成を抑止します。これはターゲットにより異なります。出力ファイルを生成する場合にのみ該当します。

オプション	説明
-l	非常に長いスケジュールを設定できるようにします。64K バイトというターゲットスケジュールのサイズ制限はオフになります。出力ファイルを生成する場合にのみ該当します。
-xname	name という名前の新しい出力 OIL ファイルに入力データを保存します。これには、RTA-OSEK GUI でファイルを保存するのと同じ効果があります。-d オプションが使用されている場合は、生成されるファイルには指定された値が反映されます。
-yname	RTA-OSEK GUI のカスタムビルドスクリプトを name という名前のファイルに展開します。通常、これはコマンドラインから実行することにより RTA-OSEK GUI のカスタムビルドと同じ効果を実現できるバッチファイルです。また、rtkbuild.bat も生成されます。

12.1.3 分析オプション

オプション	説明
-a[n]	スケジューラビリティ分析を選択し、使用されるスケジューラビリティアルゴリズムを設定します。n は分析の深さを決める値で、1 ～ 9 の値を取ることができます（1 は粗い分析、9 は正確な分析で、それ以外の値は将来使用するために確保されています）。-a は -a1 と同等です。このオプションが -s と一緒に指定されるとセンシティビティ分析用のスケジューラビリティアルゴリズムが設定され、-p と一緒に指定されると優先度割り当て用のスケジューラビリティアルゴリズムが設定されます。
-c[n]	クロック最適化を行うことにより、スケジューラブルシステムを実現できる最小クロックレートを明らかにします。タスク優先度が調整される場合があります。優先度 n は優先度割り当ての「パッキングアグレッシブネス」を決める値です。n が指定されない場合は、デフォルトの 1 になります。 このオプションを -p または -s と一緒に選択することはできません。
-nname	タイミング分析中に、指定されたタスク／割込み／プロファイルを無視します。
-p[n]	タスク優先度の自動割り当てを選択します。n は「パッキングアグレッシブネス」を決める値です。n が指定されない場合は、デフォルトの 1 になります。 このオプションを -c または -s と一緒に選択することはできません。
-s[name]	センシティビティ分析を行います。name が指定された場合は、その名前のタスクまたは割込みのセンシティビティ分析を行います。name が指定されない場合は、各実行可能オブジェクトを順に分析します。 このオプションを -c または -p と一緒に選択することはできません。
-u	スケジューラビリティがないことをエラーとして扱いません。分析時にのみ意味があります。

12.2 生成されるファイル

RTA-OSEK は OIL 設定ファイルから以下の 3 種類のファイルを生成します。

1. 複数の C ヘッダファイル: RTA-OSEK への標準および静的インターフェースへのアクセスを提供します。
2. C ソースコードファイル(osekdefs.c): RTA-OSEK および生成された OSEK API 関数用の C データ構造体が含まれます。
3. アセンブリコードファイル（osgen.<asm>）: RTA-OSEK 用、さらに設定に応じて割込みベクタ用のアセンブリデータ構造体が含まれます。

12.2.1 ヘッダファイル

RTA-OSEK を使用するアプリケーションプログラムモジュールには、生成される以下の RTA-OSEK ヘッダファイルのうちのいずれか 1 つをインクルードする必要があります。

ファイル名	説明
`Taskname.h` `ISRname.h`	タスクまたは ISR 固有のヘッダファイルで、設定ファイルに定義されている名前が付きます。 このヘッダファイルは、各タスクまたは ISR に、静的およびダイナミックな OS API 関数へのアクセスを提供します。 各ヘッダファイルは、複数のタスク／ ISR からのインクルードに対して保護されています。 1 つのタスクまたは ISR が実行するアプリケーションコードにのみインクルードしてください。
`osekmain.h`	`osek_idle_task` 用のヘッダファイルです。メインプログラム（通常は `main.c`）にのみインクルードしてください。
`osek.h`	汎用的なヘッダファイルです。タスク間で共有されるコードが含まれているモジュール（ユーザー定義のフックルーチンやコールバック関数など）にはこれをインクルードしてください。
`oseklib.h`	RTA-OSEK API 関数を使用するコードをオブジェクトライブラリとしてサードパーティに渡す際は、このヘッダファイルをインクルードしてください。 このファイルはタスクや ISR にはインクルードしないでください。

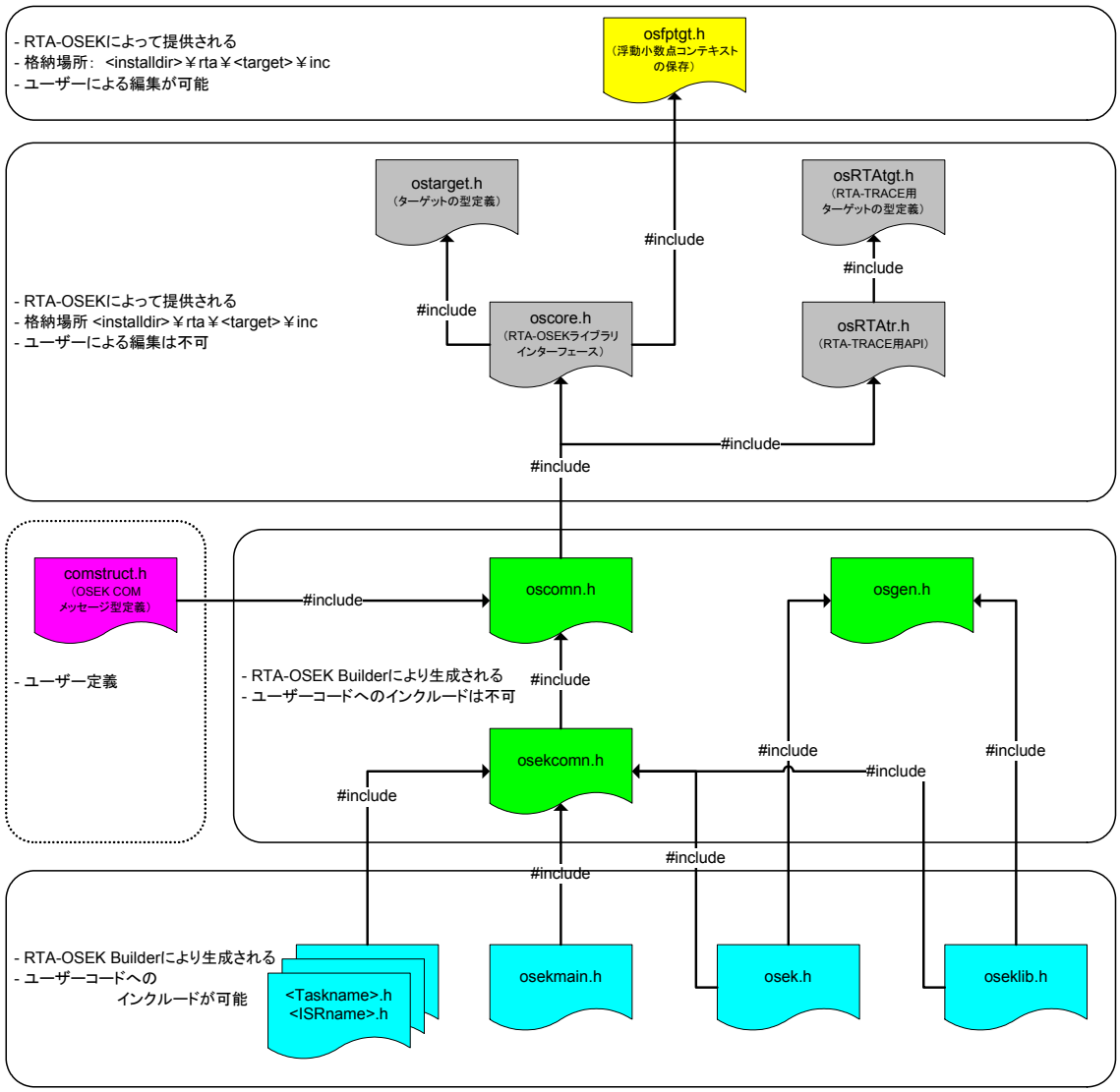
RTA-OSEK コンポーネントのライブラリは、`oscore.h` というプライマリヘッダファイルと共に供給されます。このファイルは、設定ツールが生成するすべてのヘッダファイルにインクルードされるので、ユーザーが明示的にこのファイルをインクルードする必要はありません。

RTA-OSEK ビルダは、RTA-OSEK コンポーネントのビルドステータスを表す複数のシンボルが定義された 3 つのヘッダファイルを生成します。

ファイル名	説明
`osekcomn.h`	選択されている RTA-OSEK ライブラリの最適化設定や、OSEK オブジェクトの定義
`oscomn.h`	システムのビルドレベルの定義、RTA-OSEK オブジェクト名と内部データ構造体とのマッピング情報
`osgen.h`	RTA-OSEK オブジェクト名と内部データ構造体とのマッピング情報（リソース用）

12.2.2 ヘッダファイルのインクルード構造

ヘッダファイルは、以下の図のように階層的にインクルードされます。



12.3 RTA-OSEK ライブラリ

RTA-OSEK は、3 つの定義済みライブラリファイルとして提供されます。各ファイルはそれぞれのビルドレベルで使用されます。

1. `rtk_s.<lib>`
   standard（標準）ビルドライブラリ
2. `rtk_t.<lib>`
   timing（タイミング）ビルドライブラリ
3. `rtk_e.<lib>`
   extended（拡張）ビルドライブラリ

さらに、RTA-TRACE をサポートする RTA-OS バージョンに応じて、以下のようなプリコンパイルされたライブラリファイルがあります。

4. `rtk_ts.<lib>`
   standard（標準）ビルドライブラリと **simple** トレース
5. `rtk_tt.<lib>`
   timing（タイミング）ビルドライブラリと **simple** トレース

6. rtk_te.<lib>
   extended（拡張）ビルドライブラリと **simple** トレース
7. rtk_at.<lib>
   timing（タイミング）ビルドライブラリと **advanced** トレース
8. rtk_ate.<lib>
   extended（拡張）ビルドライブラリと **advanced** トレース

ユーザーアプリケーションは、上記のうちの正しいライブラリにリンクされる必要があります。これは RTA-OSEK GUI の **Application → Implementation notes** で確認できます。

12.4 RTA-OSEK ビルダ

RTA-OSEK アプリケーションのビルドは、RTA-OSEK Builder で行います。Builder は、RTA-OSEK のビルドスクリプトで使用される環境変数とマクロ定義を提供します。

12.4.1 環境変数

RTA-OSEK のビルダを使用する際は以下の環境変数が使用できます。

<installdir>¥rta¥<target>¥Toolinit.bat に定義されている環境変数が、使用するツールチェーンに適合していることを確認してください。

rtkbuild.bat に定義されている環境変数は、RTA-OSEK がビルドスクリプトヘルパーファイル rtk-build.bat を生成する際にセットされます。各ターゲットに応じた変数の値は、『RTA-OSEK Binding Manual』に記述されています。

変数は、**Custome Build → Build Script** で使用します。その際は以下の DOS 形式でネーミングが適用されます。

名前	意味	定義
%CBASE%	コンパイラと各ツールのベースロケーション 例：C:¥compiler	Toolinit.bat
%CC%	コンパイラへのパス 例：%CBASE%¥bin¥cc.exe	Toolinit.bat
%AS%	アセンブラへのパス 例：%CBASE%¥bin¥as.exe	Toolinit.bat
%LNK%	リンカへのパス 例：%CBASE%¥bin¥lnk.exe	Toolinit.bat
%AR%	ライブラリアンへのパス 例：%CBASE%¥bin¥lnk.exe	Toolinit.bat
%CBASE_INC%	インクルードファイルへのパス 例：%CBASE%¥inc	Toolinit.bat
%COPTS%	rtkbuild.bat 内で C ソースコードファイルをコンパイルする際のデフォルトオプション。カスタムビルドセットアップの **Environment settings** セクションに環境変数 APP_COPT を宣言することにより、オプションを追加できます。 例：APP_COPT=-debug	rtkbuild.bat
%AOPTS%	rtkbuild.bat 内でファイルのアセンブルを行う際のデフォルトオプション。カスタムビルドセットアップの **Environment settings** セクションに環境変数 APP_COPT を宣言することにより、オプションを追加できます。	rtkbuild.bat

12.4.2 マクロ

名前	意味
$(ASMEXT)	コンパイラツールチェーンで使用されるアセンブラファイルに使用される拡張子（例：asm）

名前	意味
$(CONFORMANCE)	ライブラリのビルド時などに使用されるシステムコンフォーマンス情報 '1' ー システム内で優先度の共有は行われません。 '2C' ー 多重起動が行われるタスクが 1 つ以上存在しますが、優先度の共有は行われません。 '2F' ー 複数のタスクが同じ優先度を共有しています。 'e' ー 上記 3 つの指示子の修飾子として使用され、イベントが使用されているかどうか（つまり ECC システムであるかどうか）を示すものです。 これによって以下の 6 とおりの組み合わせが使用されます。 '1', '1e', '2C', '2Ce', '2F', '2Fe'
$(DIR)	OIL ファイルが保存されるディレクトリへのパス。このパスが、ビルドファイルが生成されるデフォルトロケーションとなります。
$(EDITOR)	RTA-OSEK GUI で使用されるデフォルトエディタ。このエディタはデフォルトでは Windows の Notepad（メモ帳）ですが、RTA-OSEK GUI のシステムオプション（**File → Options**）で任意のエディタに変更できます。
$(LIBEXT)	コンパイラツールチェーンで使用されるライブラリファイルに使用される拡張子（例：lib）
$(LIBTYPE)	'S', 'T', 'E' のいずれか（'Standard'、'Timing'、'Extended" に対応）
$(NAME)	アプリケーション名（OIL ファイル名から拡張子 .oil を取り除いたもの）
$(OBJEXT)	コンパイラツールチェーンで使用されるオブジェクトファイルに使用される拡張子（例：obj）
$(OPENFILE)	'Open File'（「ファイルを開く」）ダイアログボックスを開き、選択されたファイル名を戻します。 $(EDITOR) $(OPENFILE) の使用により、RTA-OSEK GUI 内に 'Quick Edit' ボタンが追加されます。
$(OS_STATUS)	'Standard', 'Timing', 'Extended' のいずれか
$(RTABASE)	RTAINIT.BAT が格納されたディレクトリへのパス。これは RTA ツールとターゲットのインストールディレクトリ（例：C:¥rta）です。
$(RTKOBJECTS)	rkbuild.bat で生成されるタスクと ISR のオブジェクトファイル（スペースで区切られます）
$(RTKOBJECTS_C)	rkbuild.bat で生成されるタスクと ISR のオブジェクトファイル（カンマで区切られます）
$(RTKLIB)	アプリケーションにリンクする RTA-OSEK コンポーネントライブラリの名前（例：標準ビルドと拡張ビルドとでは異なります）
$(TARGET)	ターゲット名（例：'HC12/COSMIC 16 task'）
$(TGTBASE)	TOOLINIT.BAT が格納されたディレクトリへのパス。これは現在のターゲットのインストールディレクトリです。
$(VARIANT)	ターゲットバリアント（例：'Star12'）。通常、ターゲット固有のコードをコンパイルする際にコマンドラインの 'define' として渡され、チップバリアントに応じた適切な設定が行われます。

13 target.ini ファイル

13.1 target.ini ファイルとは？

target.ini ファイルは、RTA-OSEK の挙動をターゲットハードウェアとデバッガに合わせて調整するメカニズムの一環です。target.ini ファイルの役割について理解するためには、まずターゲット DLL について考える必要があります。

RTA-OSEK インスタレーションの *¥instdir¥*`bin` フォルダには、`tgt`*<target>*`.dll` という形式の名前の DLL があります。これが「ターゲット DLL」で、RTA-OSEK が以下の事柄を行うために使用します。

- ターゲット割込みモデルについての情報を提供します。これには、使用可能なベクタと優先度、およびそれらを使用する際に適用されるルールなどが含まれています。
- ターゲットのタイプと制限事項についての情報を提供します。
- アプリケーションのビルド方法についての情報を提供します。
- ORTI デバッガについての情報を提供します。

ターゲット DLL は、RTA-OSEK とともに使用されるターゲットハードウェアについての詳細情報を提供します。

target.ini ファイルは、ターゲット DLL と関連付けられていて、*<target>*`.ini`、または必要に応じて *<target>*_*<variant>*`.ini` という名前で、同じフォルダに格納されます。target.ini ファイルはターゲット DLL 内の情報をオーバーライドするものです。これにより、RTA-OSEK の挙動を個々のターゲットハードウェアに合わせて調整した「バリアント（'variant'）」を作成することができます。

なお RTA-OSEK コンポーネント（つまりオペレーティングシステムカーネル）と低水準コードの生成は、ユーザーが target.ini ファイルやターゲット DLL により指定するバリエーションとは一切無関係です。従って、ある .ini ファイルにより作成されたターゲットバリアントが、コアソフトウェアに定義されていない新しいベクタやメモリモデルなどをサポートできるようにすることはできません。

13.2 target.ini ファイルの命名規則

前出のように、target.ini ファイルを使って RTA-OSEK インスタレーションのバリアントを作成することができます。ターゲット DLL が「デフォルトバリアント」を提供し、target.ini ファイルにより作成されるバリアントでこのデフォルトバリアントをオーバーライドしたり、また target.ini ファイルとデフォルトバリアントとを合わせて新しいバリアントを作成することもできます。target.ini ファイルの命名のしかたにより、このどちらの方法でバリアントが作成されるのかが決まります。

すでに説明したように、target.ini ファイル名の形式は *<target>*`.ini` か *<target>*_*<variant>*`.ini` です。前者の形式を用いると、その target.ini ファイルはデフォルトバリアントの値を変更します。後者の形式を用いると、*<variant>* という新しいバリアントが作成されます。

RTA-OSEK GUI を用いて新しいアプリケーションを作成する時には、使用可能なすべてのバリアントが **Select Target** ダイアログに表示されます。ユーザーはその中から、新しいアプリケーションに使用したいバリアントを選択できます。

13.3 target.ini ファイルのフォーマット

target.ini ファイルのフォーマットは、一般的な .ini ファイルと同じです。

- ファイルの内容はいくつかのセクションに区分され、各セクションはセクション名を角かっこで囲んだ行で始まります。
- 各セクションに含まれる属性は、*<attribute>*=*<value>* という形式の行で表わされます。属性値にはスペースを含めることもできます。
- セミコロンで始まっている行はコメントとして処理されます。

以下に例を紹介します。

```
;
; A comment at the start of the file
;
[globals]
def_cpuspeed = 100
osgen_name = osgen.s
```

ファイルの内容については、他にも以下のようなルールが適用されます。

- 同じセクション名を 2 回以上使用することはできません。
- 1 つのセクション内で同じ属性名を二度以上使用することはできません。
- 属性名やセクション名では大文字と小文字は区別されません。
- “=” の前後のスペース文字は無視されます。
- セクション、およびセクション内の属性の順序は意味を持ちません。

属性名として使用できるのは、あらかじめ定義されている名前だけです。これらの名前については本章で後述します。また、以降の部分では、属性値の例を普通は二重引用符 " " で囲んで表記しています。実際にそれらの値を使用するには、二重引用符をはずしてください。

13.4 [globals] セクション

target.ini ファイルのこのセクションの属性は、いくつかのグループに分類できます。各グループごとに、以降の各項で説明します。

13.4.1 CPU データ

このグループの属性は、ターゲット上の CPU のプロパティを定義するために使用されます。

属性	説明
def_cpuspeed	ターゲット CPU の命令サイクルレート（MHz）です。これは、新しいアプリケーションを作成する際にデフォルトとして使用されます。命令サイクルレートの定義については『RTA-OSEK ユーザーズガイド』を参照してください。
def_swspeed	CPU が上記のレートで稼働する場合に実現されるストップウォッチレート（MHz）です。これは、新しいアプリケーションを作成する際にデフォルトとして使用されます。

13.4.2 カスタムビルドデータ

これらの属性は、RTA-OSEK がユーザーのアプリケーションのカスタムビルドを実行する方法を定義するもので、特に rtkbuild.bat というバッチファイルを生成する際に影響します。

rtkbuild.bat には、ユーザーのアプリケーションの各部分をビルドするためのコンパイラやアセンブラなどの呼び出しが含まれています。一般に、rtkbuild.bat からこれらのツールにコマンドラインオプションを渡す必要がある場合には、ツールが認識できる環境変数を指定してください。

- COPTS は、C コンパイラにコマンドラインオプションを渡すために使用されます。
- COPTS2 は、osekdefs.c をコンパイルするときに C コンパイラにコマンドラインオプションを渡すために使用されます。
- AOPTS は、アセンブラにコマンドラインオプションを渡すために使用されます。

ただし、コマンドラインオプションに “=” という文字が含まれている場合には、rtkbuild.bat は環境変数を使用せずに、代わりにツールのコマンドラインにコマンドラインオプションをそのまま配置します。

rtkbuild.bat により設定された環境変数は、rtkbuild.bat を呼び出した後で行われるコンパイラやアセンブラコールにおいて使用されます。

属性	説明
osgen_name	RTA-OSEK により生成されるメインアセンブラファイルの名前（例：“osgen.s”）
c_include	インクルード探索パスを拡張するために使用される、コンパイラのコマンドラインオプション（例：“-i”）
c_include_sep	デフォルトでは、c_include 内のコマンドラインオプション（-ia -ib -ic など）で定義されたパスが、コンパイラのインクルードパスに追加されます。c_include_sep が空でない場合には、これらが c_include_sep の値で区切られて、まとめて 1 つのオプションになります。たとえば、c_include_sep に “,” という値を指定すると、-ia,b,c というコマンドラインオプションができます。
c_include_trail	コンパイラのインクルードパスを拡張するコマンドラインオプションの後ろに付くテキストです。たとえば、コマンドラインオプションを -include(*folder*) のようにしたい場合には、c_include に “-include(” を設定し、c_include_trail に “)” を設定します。
c_define	プリプロセッサシンボルを定義するために使用される、コンパイラ用コマンドラインオプションです。例：“-d”
c_defopt	すべてのタスクと ISR の C ファイル用としてコンパイラに供給されるデフォルトのコマンドラインオプションです。例：“-nowiden”
osekdefsc_defopt	これが空でない場合、osekdefs.c をコンパイルする時の c_defopt をオーバーライドします。
osekdefsc_ortiopt	これが空でない場合、ORTI データが生成される時、つまり、デバッガの使用が指定されている時には、osekdefs.c をコンパイルするための c_defopt をオーバーライドします。
c_insertopt	これは c_defopt とほぼ同じですが、環境変数 COPTS にはオプションが設定されない点だけが異なります。その代わり、それらのオプションは C コンパイラのコマンドライン内のコンパイル対象ファイル名の直前に直接追加されます。
c_prefixopt	これは c_insertopt とほぼ同じですが、オプションがコマンドライン内のコンパイル対象ファイル名の直後に追加される点だけが異なります。
a_include	これらは、上記の c_... 属性と同じ目的で使用されますが、C コンパイラではなくアセンブラに適用されます。
a_include_trail	
a_define	
a_defopt	
a_insertopt	
a_prefixopt	

属性	説明
osgen_defopt	これが空でない場合には、osgen.<*ext*>（<*ext*> はターゲット上のアセンブラソースファイル用の拡張子）をアセンブルする際に a_defopt をオーバーライドします。
extobj	ターゲットのオブジェクトファイル用の拡張子（例：“obj”）
extasm	ターゲットのアセンブラソースファイル用の拡張子（例：“asm”）
extlib	ターゲットのライブラリファイル用の拡張子（例：“lib”）

13.5 [vectors] セクション

このセクションの属性により、ターゲット上のベクタテーブル内のベクタについて RTA-OSEK が使用する名前に対して、任意の名前をオーバーライドすることができます。これらの名前は、ユーザーが RTA-OSEK GUI を使用して ISR に関連付けるベクタを選択するときに表示されます。

[Vectors] セクションの各属性は以下の形式で定義されます。

```
0x<vector number> = <vector name>
```

<*vector number*> は、ユーザーが ISR に関連付けるベクタを選択するときに表示されるベクタリスト内のベクタの、ゼロをベースとするインデックスです。<*vector number*> は必ず小文字の 16 進数形式で表現し、上記のように前に 0x を付けます。<*vector name*> は、ベクタに付けたい新しい名前です。

以下に、ターゲットのベクタテーブルの一部のベクタ名をオーバーライドする例を紹介します。

```
[vectors]
0x0 = Foo vector
0x1 = Bar vector
0xa = Flibble vector
```

13.6 ORTI デバッガのセクション

13.6.1 ORTI デバッガについて記述する

『ORTI ユーザーズガイド』には、RTA-OSEK が、ユーザーアプリケーションについての情報をデバッガに渡すためのファイル（拡張子が .ort のファイル）をどのようにして生成するかについて説明されています。まったく同じ ORTI デバッガは 2 つとしてなく、また ORTI 自体にも複数のバージョンがあるので、RTA-OSEK が .ort ファイルを生成する方法はデバッガごとに異なります。上述したように、ターゲット DLL には、ユーザーの RTA-OSEK インスタレーションによりサポートされている特定のデバッガの挙動についての詳細を RTA-OSEK に知らせる、という機能があります。

ORTI デバッガについて定義するためには数多くの属性が使用されます。これらの属性用の値は 4 通りにプリセットされていて、それぞれに以下の番号が付けられています。

- Preset 0　汎用 ORTI デバッガの定義です。
- Preset 1　Noral Flex 4.2 の定義です。
- Preset 2　CrossView OSEK/ORTI v2.0 の定義です。
- Preset 3　Cosmic Zap 3.5 および 3.6 の定義です。

各属性と、それらの属性にプリセットされている値は、下の表に示すとおりです。

属性	説明
header_comments	.ort ファイルの冒頭にコメントを設定できるかどうかを示します。
inline_comments	.ort ファイル内の行に // で始まるコメント（インラインコメント）を設定できるかどうかを示します。
enum_addresses	.ort ファイル内に列挙型データを ENUM <*address*> […] という形で宣言できるかどうかを示します。

属性	説明
totrace_used	SERVICETRACE という ORTI 属性の最下位ビット（LSB）の意味をデバッガが理解しているかどうかを示します。
orti_task	true の場合、osek_running_task の内容をデバッガが直接解読できないことを示します。osek_running_task の内容に間接アクセスできるように、定数 orti_task が osekdefs.c に挿入されます。
orti_active	true の場合、アクティブなタスクまたは ISR を示す所定の内部変数の内容をデバッガが直接解読できないことを示します。これらの変数の内容に間接アクセスできるように、定数が osekdefs.c に挿入されます。
orti_lasterror	true の場合、osek_last_error の内容をデバッガが直接解読できないことを示します。osek_last_error の内容に間接アクセスできるように、定数 orti_err が osekdefs.c に挿入されます。
orti_appmode	true の場合、osek_cur_mode の内容をデバッガが直接解読できないことを示します。osek_cur_mode の内容に間接アクセスできるように、定数 orti_mode が osekdefs.c に挿入されます。
orti_trace	true の場合、osek_orti_call の内容をデバッガが直接解読できないことを示します。osek_orti_call の内容に間接アクセスできるように、定数 orti_trace が osekdefs.c に挿入されます。
max_string_size	ゼロでない場合、.ort ファイル内の文字列の長さが、ここで指定された文字数以内に制限されます。
use_currentservice_name	CURRENTSERVICE を.ortファイル内の SERVICETRACE の代わりに使用するかどうかを示します。
var_prefix	.ort ファイルにより参照される名前付き変数にプレフィックスを付けるために使用されます。この表の他のプレフィックスが付いている名前付き変数は .ort ファイルにより参照されることはありません。たとえば CrossView デバッガが MyVar という変数にアクセスする場合には、その変数名を _MyVar にする必要があります。
ienum_address	enum_address が設定されている場合、.ort ファイル内の列挙型データは ENUM *<ienum_address>* […] という形で宣言されます。ORTI v2.0c 準拠のデバッガ では *<ienum_address>* の値が ADDRESS になりますが、それより後のバージョンでは UINT16 などのタイプを使用します。
itask_prefix	.ort ファイルの implementation セクションでタスクおよびカテゴリ 2 ISR の名前に使用されるプレフィックスを定義します。
dtask_prefix	.ort ファイルの information セクションでタスクおよびカテゴリ 2 ISR の名前に使用されるプレフィックスを定義します。
dpri_prefix	.ort ファイルの information セクションで変数 orti_pri を参照する際に使用されるプレフィックスを定義します。
dtrace_prefix	.ort ファイルの information セクションで変数 orti_trace / osek_orti_call を参照する際に使用されるプレフィックスを定義します。*
derr_prefix	.ort ファイルの information セクションで変数 orti_err / osek_last_error を参照する際に使用されるプレフィックスを定義します。*
dmode_prefix	.ort ファイルの information セクションで変数 orti_mode / osek_cur_mode を参照する際に使用されるプレフィックスを定義します。*
drdy_prefix	.ort ファイルの information セクションで変数 orti_rdy / OS_L0001 を参照する際に使用されるプレフィックスを定義します。*
dwait_prefix	.ort ファイルの information セクションで変数 orti_wai / OS_L0004 を参照する際に使用されるプレフィックスを定義します。*
dtime_prefix	.ort ファイルの information セクションで TickType タイプの変数を参照する際に使用されるプレフィックスを定義します。
dboolptr_prefix	.ort ファイルの information セクションで boolean 型を指すポインタ変数を参照する際に使用されるプレフィックスを定義します。

*. これらの変数の一方をデバッガが直接解読できない場合は、他方の変数が使用されている場合があります。

上の表で紹介している属性の一部は boolean 型で、その属性が true（真）であるか false（偽）であるかを示します。このような属性を “true” にするには “1” を設定し、“false” にするには “0” を設定します。プリセットされている値は以下のとおりです。

属性	Preset 0	Preset 1	Preset 2	Preset 3
header_comments	0	1	0	0
inline_comments	0	1	0	0
enum_addresses	0	1	1	0
totrace_used	0	0	0	0
orti_task	0	1	0	0
orti_active	0	1	0	0
orti_lasterror	0	0	0	0
orti_appmode	0	0	0	0
orti_trace	0	0	0	0
max_string_size	0	0	55	0
use_currentservice_name	0	1	0	0
var_prefix	*<empty>*	*<empty>*	_	*<empty>*
ienum_address	ADDRESS	ADDRESS	UINT16	ADDRESS
itask_prefix	*<empty>*	*<empty>*	*_	*<empty>*
dtask_prefix	*<empty>*	*<empty>*	*_	*<empty>*
dpri_prefix	*<empty>*	*<empty>*	*_	*<empty>*
dtrace_prefix	*<empty>*	*<empty>*	*(unsigned char *)_	*<empty>*
derr_prefix	*<empty>*	*<empty>*	*(unsigned char *)_	*<empty>*
dmode_prefix	*<empty>*	*<empty>*	*(unsigned char *)_	*<empty>*
drdy_prefix	*<empty>*	*<empty>*	*_	*<empty>*
dwait_prefix	*<empty>*	*<empty>*	*_	*<empty>*
dtime_prefix	*<empty>*	*<empty>*	*_	*<empty>*
dboolptr_prefix	*<empty>*	*<empty>*	(unsigned char *)*_	*<empty>*

これらのプリセットの 1 つを選択することにより、デバッガの設定を容易に行えます。また、必要に応じて一部の属性値を変更して新しいデバッガディスクリプションを新しい名前で作成することもできます。ユーザー用ターゲット DLL は、現在インストールされている RTA-OSEK によりサポートされているデバッガ用に、すでに変更されています。

実際に使用するデバッガが現在インストールされているものと異なる場合には、target.ini ファイルを使用して、ターゲット DLL の情報をオーバーライドまたは追加することにより、デバッガの挙動を RTA-OSEK に伝えることができます。以下に、オーバーライドする方法と追加する方法を説明します。

13.6.2 既存のデバッガディスクリプションをオーバーライドする方法

target.ini ファイルを使用してユーザーの RTA-OSEK インスタレーションの既存のデバッガディスクリプションをオーバーライドしたい場合には、まず、修正したいデバッガディスクリプションの名前を見つけ出す必要があります。このためには、RTA-OSEK GUI のナビゲーションバーから **Target** グループの **Debugger** サブグループを選択してください。すると、ワークスペースに、現在選択されているデバッガディスクリプションの名前が表示されます。このデバッガディスクリプションをオーバーライドするためにはまず、ディスクリプション名（つまり、二重引用符に囲まれたテキスト）をクリップボードにコピーします。

ユーザーの target.ini ファイルを開き、そこにデバッガディスクリプションを貼り付けて角かっこで囲み、新しいセクション名にします。これで前出の表で紹介した属性値を新しいセクションに追加して、ターゲット DLL により定義される値をオーバーライドすることができます。

たとえば、現在の RTA-OSEK インスタレーションが Debug-o-matic デバッガ対応のものである場合、RTA-OSEK GUI のナビゲーションバーから **Target** グループの **Debugger** サブグループを選択すると、ワークスペースには以下のテキストが表示されます。

```
Debug output type "Debug-o-matic", version 2.0.
```

Debug-o-matic デバッガ用の所定の属性をオーバーライドするためには、target.ini ファイル内に以下のようなセクションを作成してください。

```
[Debug-o-matic]
orti_task = 1     ; Override orti_task
orti_trace = 1    ; Override orti_trace
```

13.6.3 新しいデバッガディスクリプションを追加する方法

target.ini ファイルを使用して現在の RTA-OSEK インスタレーションに新しいデバッガディスクリプションを追加したい場合には、まず、そのインスタレーションがすでにサポートしているディスクリプションの数を確認する必要があります。このためには、RTA-OSEK GUI のナビゲーションバーから **Target** グループの **Debugger** サブグループを選択します。ワークスペースの **Debugger** ボタンをクリックし、表示される **Select debugger** ダイアログのドロップダウンリストに含まれているデバッグディスクリプションの数を数えます（"<none>" というディスクリプションは数に入れません）。ここでは、デバッガディスクリプションの数を *n* として説明します。

target.ini ファイルを編集し、以下のように `[ORTI]` というセクションを作成してください。

```
[ORTI]
ndebuggers = <n+1>
debugger_name_<n+1> = <Name of new debugger description>
```

それから、target.ini ファイル内にセクションをもう 1 つ作成し、それに *[<Name of new debugger description>]* という見出しを付けます。この時点で、新しいデバッガディスクリプションのベースとして、4 つのプリセット（前述の Preset 0 ～ 3）のうちのどれを使用するかを決める必要があります。どのプリセットを使用するかが決まったら、新しいセクションに `vendor` という名前の属性を追加し、その属性値として、選択したプリセットの番号を設定してください。この設定が終わると、以下のようになります。

```
[ORTI]
ndebuggers = <n+1>
debugger_name_<n+1> = <Name of new debugger description>

[<Name of new debugger description>]
vendor = <number from 0-3>
<attribute> = <value>
<attribute> = <value>
...
```

これについて、例を用いてわかりやすく説明します。すでに 2 つのデバッガをサポートしている RTA-OSEK インスタレーションに Debug-o-matic という第 3 のデバッガのサポートを追加したい場合には、target.ini ファイル内に以下の行を作成します。

```
[ORTI]
ndebuggers = 3
debugger_name_3 = Debug-o-matic

[Debug-o-matic]
vendor = 1        ; Base it on Noral Flex 4.2
```

```
orti_task = 0      ; Make a few further changes
orti_trace = 1     ;
```

これが終わると、“Debug-o-matic” というディスクリプションが、前述の **Select debugger** ダイアログのデバッガディスクリプションのドロップダウンリストに表示されるようになります。このディスクリプションを **Select debugger** ダイアログから選択することにより、RTA-OSEK が .ort ファイルを作成するときにそのディスクリプションが使用されるようにすることができます。

13.6.4 新しい属性値を選択する

これまで、target.ini ファイルに属性を追加することにより既存のデバッガディスクリプションをオーバーライドしたり新しいデバッガディスクリプションを追加したりする方法について説明してきました。多くの場合、これらの属性に代入する値を選択する操作は、以下のような反復的プロセスになる可能性があります。

- target.ini ファイル内の値を変更します。
- RTA-OSEK を使用して .ort ファイルを生成します。
- .ort ファイルをデバッガにロードし、それが期待どおりに機能するかどうかを確認します。
- 必要に応じて、以上の手順を繰り返します。

状況によっては、デバッガに式を直接入力し、デバッガがそれを正しく解読できるようになるまで変更を加えた上で、target.ini ファイルを調整して正しい式を生成する方が容易な場合があります。