RTA-TRACE

ユーザーズガイド

著作権について

Document TD00002-004

目次

1 本書について

RTA-TRACE は組込みシステム用のソフトウェアロジックアナライザです。アプリケーションと組み合わせて使用することにより、システムのデバッグやテストに役立つさまざまなサービスを利用できます。中でも特に優れた機能として、量産用にビルドされたアプリケーションソフトウェアについて、ランタイムにシステム内で起こっている事象を正確に把握することができます。

本書では、RTA-TRACE 用のターゲット API について説明します。

1.1 本書の対象ユーザー

本書『RTA-TRACE ユーザーズガイド』は、ターゲット上で実行される組込みソフトウェアの開発者を対象とした、RTA-TRACE 用ターゲット API のリファレンスガイドです。また本書には、参考のために RTA-TRACE アーキテクチャの概要も紹介されています。

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されています。

本書では、プログラムコード、ヘッダファイル名、C のデータ型名、C 関数および API 関数名はすべてクーリエ体（courier）で表記されています。オブジェクトの名前も、プログラマに公開され次第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は、`Task1` という名前のタスクハンドルとして表記されます。

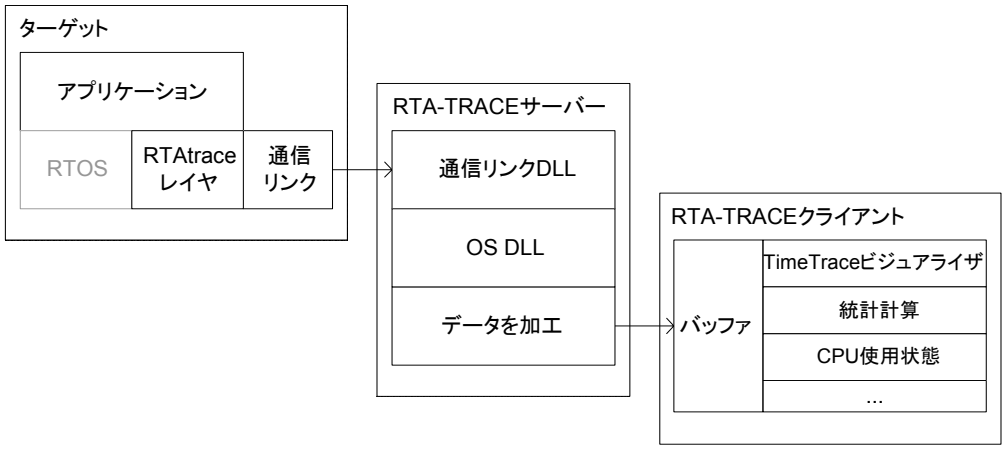
また PDF 文書において、索引、および他の部分を参照する個所（例 :「第 3 章を参照してください」の部分）については、その参照先へのリンクが設けられているので、必要な参照個所を素早く見つけることができます。

2 RTA-TRACE の概要

RTA-TRACE は組込みシステム用のソフトウェアロジックアナライザです。アプリケーションと組み合わせて使用することにより、システムのデバッグやテストに役立つさまざまなサービスを利用できます。中でも特に優れた機能として、量産用にビルドされたアプリケーションソフトウェアについて、ランタイムにシステム内で起こっている事象を正確に把握することができます。

RTA-TRACE 製品は、「ターゲット」、「RTA-TRACE サーバー」、「RTA-TRACE クライアント」の 3 つのレイヤで構成されています（各レイヤについては以下に詳述します）。

RTA-TRACE の全体の構成は下図のとおりです。



2.1 ターゲット

ターゲットレイヤは、アプリケーション、OSEK OS（「RTA-TRACE 計装キット（'Instrumenting Kit'）」を使用する場合）、および RTA-TRACE ターゲットソフトウェアで構成されます。ライブラリとヘッダファイルからなる RTA-TRACE ターゲットソフトウェアは、OS コールを「傍受」し、また、ユーザーが生成したトレース情報をロギングするための API を提供します。（ロギングには「トレースポイント」、「タスクトレースポイント」、および「インターバル」という 3 種類のトレースポイントを使用します。本書の当該部分を参照してください。）

システムビルド工程において、クラス単位でログデータ（OS トレース情報とユーザートレース情報）の有効／無効を切り替えることができます。このメカニズムについては『RTA-TRACE コンフィギュレーションガイド』を参照してください。

ターゲット上で生成されたトレースデータは、ECU リンク（『RTA-TRACE ECU リンクガイド』を参照してください）経由で RTA-TRACE サーバーに転送されます。

2.2 RTA-TRACE サーバー

RTA-TRACE サーバーはターゲット（所定のライセンスが取得されていれば、1 台のサーバーに複数のターゲットを接続することができます）により生成されたトレースデータを収集し、RTA-TRACE クライアントに適したフォーマットに加工します。加工されたデータは、ECU リンクと OS に対応した DLL により、表示用データとして変換／解凍されます。

サーバーアプリケーションはクライアントからは独立した存在ですが、多くの場合クライアントが起動されると自動的に実行されます。サーバーが稼働しているかどうかは、ホスト PC のシステムトレイにアイコンが表示されているかどうかによってのみわかります。ECU リンクのパラメータを修正する必要がある場合は、『RTA-TRACE ECU リンクガイド』を参照してください。

RTA-TRACE サーバーは、複数の RTA-TRACE クライアントおよびターゲットアプリケーションからのローカル接続（同じ PC 上）、またはリモート接続（ネットワーク経由）に対応できます。ただしクライアントにはそれぞれライセンスが必要です。

2.3 RTA-TRACE クライアント

RTA-TRACE クライアントの機能は記録されたデータを表示したり統計計算を行うもので、RTA-TRACE の中でユーザーの目に触れる部分です。クライアントにはさまざまな視覚化オプションがプラグインとして用意されていますが、これらの使用には個別のライセンスが必要です。

クライアントは、ターゲットにより生成された未加工のトレースデータを、コンフィギュレーション設定時にユーザー定義された規則に従い、読みやすいフォーマットに変換します。コンフィギュレーションの設定方法については、『RTA-TRACE コンフィギュレーションガイド』を参照してください。

プラグインの例としては、TimeTrace ビジュアライザ（アプリケーションのタスクアクティビティを調べるためのツール）や、CPU 使用状況のパイチャート／棒グラフ（システムオブジェクトによる CPU の使用状況を見ることができるツール）などがあります。

プラグイン機能については、オンラインヘルプファイルで詳しく説明されています。オンラインヘルプは、RTA-TRACE クライアントの **Help** メニューから見ることができます。

3 API 関数

3.1 トレースコントロール

以下の API 関数は、RTA-TRACE の動作モードをコントロールします。

3.1.1 StartFreeRunningTrace

構文： `void StartFreeRunningTrace(void)`

引数： なし

説明： フリーランニングモードでトレースを開始します。

注意事項： 「フリーランニング （'free running'）モード」では、イベント情報が継続的に捕捉され、収集されたデータは、すぐにホストにアップロードされます。
トレースデータのロギングは、トレースバッファが満杯になると保留され、バッファが空になると再開されます。このような状態は、トレースデータの量に対して ECU リンクの転送速度が低すぎる場合に発生する可能性があります。
トレース実行中にこの API がコールされると、トレースバッファの内容がクリアされてからトレースが再開されます。

使用できる OS： すべての OS

3.1.2 StartBurstingTrace

構文： `void StartBurstingTrace(void)`

引数： なし

説明： トレースをバーストモードで開始します。

注意事項： 「バースト（'bursting'）トレースモード」では、イベント情報がワンショット方式でトレースバッファに収集されます。バッファが満杯になるとトレースが停止し、データ転送が開始されます。つまり、サーバーへのデータアップロードはトレースバッファが満杯になるまでは行われません。
`SetTraceRepeat()`（3.1.4 項参照）により反復バーストトレースが有効に設定されている場合には、データ転送が完了した時点でトレースが再開されます。
トレース実行中にこのコールが発行されると、トレースバッファの内容がクリアされてからトレースが再開されます。

使用できる OS： すべての OS

3.1.3 StartTriggeringTrace

構文： `void StartTriggeringTrace(void)`

引数： なし

説明： トレースをトリガモードで開始します。

注意事項： 「トリガ（'triggering'）トレースモード」では、特定のイベントの発生を待ってからトレースデータがホストに転送されます。トリガイベントは `TriggerOn…()` という API 関数を使って設定します（3.7.4 項を参照してください）。
`SetTriggerWindow()`（3.7.1 項を参照してください）を使ってプリトリガおよびポストトリガのバッファサイズを指定し、トリガイベントの前後のイベントセットを表示することができます。
`SetTraceRepeat()`（3.1.4 項を参照してください）を使って、トリガトレースを繰り返して実行することができます。その場合、データ転送完了後にトリガイベントが発生するとトレースが再開されます。
トレース実行中にこのコールが発行されると、トレースバッファの内容がクリアされてからトレースが再開されます。

使用できる OS： すべての OS

3.1.4 SetTraceRepeat

構文：
```
void SetTraceRepeat(
     osTraceBool <mode>)
```

引数：

mode 0 以外の値は反復モードを有効にし、0 は反復モードを無効にします。

説明： バーストトレースとトリガ トレースの反復実行の有効／無効を切り替えます。

注意事項： 3.1.2 項と 3.1.3 項を参照してください。

使用できる OS： すべての OS

3.1.5 StopTrace

構文： `void StopTrace(void)`

引数： なし

説明： トレースバッファへのトレースデータの記録を停止します。

注意事項： これはデータリンクを切断するものではありません。この API をコールすると、トレースバッファに残っていたデータはすべてホスト PC にアップロードされます。

使用できる OS： すべての OS

3.2 全般

RTA-TRACE では、「トレースポイント」、「タスクトレースポイント」、「インターバル」という 3 種類のマーカを定義してプログラムのトレースを行うことができます。

- トレースポイント（3.3 項参照）は、TimeTrace ビジュアライザ上に表示される各トレースポイントの帯に表示されます。プログラムの任意のポイントでロギングを行うことにより、プログラム全体の状態を知ることができます。
- タスクトレースポイント（3.4 項参照）は、それをロギングするタスクの帯に表示されます。ある特定のタスクのアクティビティを把握するのに理想的です。

- インターバル （3.5 項参照）は、プログラムの任意の場所でロギングできます。インターバルには開始ポイントと終了ポイントとがあり、ランタイムの時間経過（例：スティミュラスからレスポンスまでの時間）を把握する目的で使用されます。

上記のどのマーカでロギングを行う場合も、16 ビットまたは 32 ビットの値（`Log…()` コールの `…Value()` バージョンを使用した場合）、またはそれより大きなデータブロック（`Log…()` コールの `…Data()` バージョンを使用した場合）をロギングし、TimeTrace ビジュアライザに表示することができます。この値やブロックデータは、コンフィギュレーション設定時に定義されたフォーマット文字列に従って表示されます（詳細は『RTA-TRACE コンフィギュレーションガイド』を参照してください）。

上記の各マーカは「識別子」を持ち、1 つまたは複数の「カテゴリ」に属することができます。これらの用語について、以下に詳しく説明します。

3.2.1 識別子

トレースポイント（'tracepoint'）、タスクトレースポイント（'tasktracepoint'）、インターバル（'interval'）に使用できる識別子（ID）の値の範囲は限られています。各タイプごとに、以下の範囲のコンパクト ID または拡張 ID（コンフィギュレーション設定時にどちらかが選択されます）を使用できます。

ID	コンパクト[1]	拡張（デフォルト）[1]
`Tracepoint`	8 ビット（1 ～ 255）	13 ビット（1 ～ 8191）
`TaskTracePoint`	4 ビット（1 ～ 15）	13 ビット（1 ～ 8191）
`Interval`	8 ビット（1 ～ 255）	13 ビット（1 ～ 8191）

識別子は、コンフィギュレーション設定時に任意に指定するか（ERCOS^{EK} と RTA-OSEK の場合にはこちらをお勧めします）、または名前の代わりにデフォルトの数値を使用することもできます。ERCOS^{EK} および RTA-OSEK の場合、生成された識別子は `rtatrace.h` というヘッダファイル内に `#define` 文で定義されます。コンフィギュレーション設定時に指定された識別子を使用する方が、視覚化されたトレースデータを容易に把握できます（たとえば、TimeTrace ビジュアライザ 上に、`"Tracepoint 5"` ではなく `"Tracepoint <name>"` というように表示されます）。

トレースポイント を名前と番号で設定しておけば、そのリテラル番号を識別子として使用することによりそれに対応する名前を RTA-TRACE クライアントに表示させることができるので、名前付き識別子を使用することをお勧めします。

OS 計装キットを使用している場合でもこのメカニズムを利用できますが、その場合にはディスクリプションファイル（拡張子は `.rta`）をマニュアル操作で生成する必要があるかもしれません。

注記

範囲外の識別子を指定して `Log…()` コールを行うと、そのコールでは何の処理も行われずにそのまま異常終了（'silently fail'）します。

3.2.2 カテゴリ

ユーザー定義された各トレースポイント（またはタスクトレースポイント）は、1 つまたは複数のユーザー定義カテゴリに属します。これにより、複数のトレースポイント（またはタスクトレースポイント）をランタイムに、またはコンフィギュレーション設定時にまとめて有効または無効にできます。ランタイムカテゴリは 31 個、コンフィギュレーション設定されたカテゴリは何個でも使用できます。

1. 0 という ID を生成することもできますが、実際には、ID が 0 のインターバル、トレースポイント、タスクトレースポイントにはトリガがかかりません。

カテゴリは、コンフィギュレーション設定時[2]にビットマスクの形で定義されるので、ビット論理和演算子（|）を使って複数のトレースカテゴリを組み合わせることができます。

注記

`Log…()` コールでは、指定されたカテゴリの**どれか 1 つでも**アクティブならトレースポイントがロギングされます（つまり、ある特定のトレースポイントについて、コンフィギュレーションで有効に指定されたカテゴリとランタイムカテゴリとを組み合わせた場合、ランタイムカテゴリだけを無効してもまったく意味はなく、ロギングは行われます）。

注記

ある特定の `Log…()` コールについて定義されているすべてのカテゴリをコンフィギュレーションで `FALSE` にすると、その `Log…()` コールはコード内に挿入されません。

注記

カテゴリを有効にしたり無効にしたりする処理（`EnableTraceCategories` と `DisableTraceCategories` を使用します）は、必ずトレースが停止している時に行ってください。そうしないと、RTA-TRACE ビジュアライザに不正確なデータが表示されてしまう可能性があります。

3.2.3 クラス

ロギングされる各イベントはいずれかの定義済みのクラスに属しているので、ユーザーはランタイムまたはコンフィギュレーション設定時にイベントの有効／無効の指定をクラス単位で行うことができます。定義されているクラスは次の表に示すとおりです。

クラスはビットマスクなので、ビット論理和演算子（|）により、クラス単位で有効か無効かを指定できます。

osTraceClassesType	使用できる OS
TRACE_ACTIVATIONS_CLASS	すべての OS
TRACE_OSEK_MESSAGES_CLASS	すべての OS
TRACE_RESOURCES_CLASS	すべての OS
TRACE_INTERRUPT_LOCKS_CLASS	すべての OS
TRACE_SWITCHING_OVERHEADS_CLASS	すべての OS
TRACE_TASKS_AND_ISRS_CLASS	すべての OS
TRACE_PROCESSES_CLASS	ERCOS^{EK}、カスタム OS
TRACE_EXPLICIT_STATE_MESSAGES_CLASS	ERCOS^{EK}
TRACE_ERRORS_CLASS	すべての OS
TRACE_TASK_TRACEPOINT_CLASS	すべての OS
TRACE_TRACEPOINT_CLASS	すべての OS
TRACE_INTERVALS_CLASS	すべての OS
TRACE_MESSAGE_DATA_CLASS	すべての OS
TRACE_STARTUP_AND_SHUTDOWN_CLASS	すべての OS
TRACE_ALARMS_CLASS	すべての OS
TRACE_TIMETABLES_CLASS	ERCOS^{EK}
TRACE_SCHEDULES_CLASS	RTA-OSEK

2. コンフィギュレーション設定の詳細については、ご使用のオペレーティングシステム用の『RTA-TRACE コンフィギュレーションリファレンス』を参照してください。

osTraceClassesType	使用できる OS
TRACE_OSEK_EVENTS_CLASS	RTA-OSEK、カスタム OS
TRACE_NO_CLASSES	すべての OS
TRACE_ALL_CLASSES	すべての OS

注記

クラスを有効にしたり無効にしたりする処理（EnableTraceClasses と DisableTraceClasses を使用します）は、必ずトレースが停止されている時に行ってください。そうしないと、RTA-TRACE ビジュアライザに不正確なデータがデータが表示されてしまう可能性があります。

3.2.4 EnableTraceClasses

構文：

```
void EnableTraceClasses(
     osTraceClassesType <mask>)
```

引数：

mask 有効にするクラス（1 つまたは複数）を表すマスク

説明： 1 つまたは複数のクラスのトレースイベントを実行時に有効にします。
3.2.5 項の DisableTraceClasses とペアで使用します。

注意事項： mask で指定されないクラスのステートは変わりません。

使用できる OS： すべての OS

3.2.5 DisableTraceClasses

構文：

```
void DisableTraceClasses(
     osTraceClassesType <mask>)
```

引数：

mask 無効にする 1 つまたは複数のトレースイベントファミリを表すマスク

説明： 1 つまたは複数のトレースイベントファミリを実行時に無効にします。
3.2.4 項の EnableTraceClasses とペアで使用します。

注意事項： mask で指定されないクラスのステートは変わりません。

使用できる OS： すべての OS

3.2.6 EnableTraceCategories

構文：
```
void EnableTraceCategories(
     osTraceCategoriesType <mask>)
```

引数：

mask イベントカテゴリを表すマスク

説明： ユーザー定義されたユーザートレースポイントのサブセットを有効にします。3.2.7 項の DisableTraceCategories とペアで使用します。

注意事項： 各ユーザートレースポイントは 1 つまたは複数のカテゴリに属しています。`mask` で指定されないカテゴリのステートは変わりません。

使用できる OS： すべての OS

3.2.7 DisableTraceCategories

構文：
```
void DisableTraceCategories(
     osTraceCategoriesType <mask>)
```

引数：

mask イベントカテゴリを表すマスク

説明： ユーザー定義されたユーザートレースポイントのサブセットを無効にします。3.2.6 項の EnableTraceCategories とペアで使用します。

注意事項： 各ユーザートレースポイントは 1 つまたは複数のカテゴリに属しています。`mask` で指定されないカテゴリのステートは変わりません。

使用できる OS： すべての OS

3.3 トレースポイント

トレースポイントは、TimeTrace ビジュアライザ 上の各トレースポイントの帯に表示されます。トレースポイントの識別子は、コンフィギュレーション設定時に任意に指定するか、または名前の代わりにデフォルトの数値を使用することもできます。

3.3.1 LogTracepoint

構文：
```
void LogTracepoint(
    <TTpointId>,
    osTraceCategoriesType <catmask>)
```

引数：

TpointId トレースポイント の識別子（サイズは設定に応じて異なります。3.2.1 項の「識別子」を参照してください。）

catmask このトレースポイントが属する 1 つまたは複数のカテゴリ

説明： 指定されたトレースポイント をロギングします。

注意事項： ロギングは、`TRACE_TRACEPOINT_CLASS` クラスと `<catmask>` で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.3.2 LogTracepointValue

構文：

```
void LogTracepointValue(
   <TpointId>,
   uint <val>,
   osTraceCategoriesType <catmask>)
```

引数：

TpointId トレースポイント の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

val トレースポイントにおいて送信される数値。`val` のサイズ（16 ビットか 32 ビット）は、コンフィギュレーション設定時に選択されたシステムタイムのサイズにより決まります。詳しくは『RTA-TRACE コンフィギュレーションガイド』を参照してください。

catmask このトレースポイントが属する 1 つまたは複数のカテゴリ

説明： 指定されたトレースポイント と、16 ビットまたは 32 ビットの符号なし整数値をロギングします。

注意事項： ロギングは `TRACE_TRACEPOINT_CLASS` クラスと `<catmask>` で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.3.3 LogTracepointData

構文：

```
void LogTracepointData(
   <TpointID>,
   osTraceDataPtr <dataPtr>,
   osTraceDataLength <length>,
   osTraceCategoriesType <catmask>)
```

引数：

TpointID トレースポイント の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

dataPtr トレースポイントにおいて送信されるデータブロックへのポインタ

length `<dataPtr>` により参照されるデータブロックの長さ（バイト数）

catmask このトレースポイントが属する 1 つまたは複数のカテゴリ

説明： 指定されたトレースポイント と、任意のバイナリデータをロギングします。

注意事項： ロギングは `TRACE_TRACEPOINT_CLASS` クラスと `<catmask>` で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.4 タスクトレースポイント

タスクトレースポイントは、TimeTrace ビジュアライザ の、`LogTaskTracepoint…()` コールを発行したタスクの帯に表示されます。表示されるタスクトレースポイントの識別子は、コンフィギュレーション設定時に任意に指定するか、または名前の代わりにデフォルトの数値を使用することもできます。

3.4.1 LogTaskTracepoint

構文：

```
void LogTaskTracepoint(
    <TTpointId>,
    osTraceCategoriesType <catmask>)
```

引数：

TTpointId タスクトレースポイントの識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

catmask このタスクトレースポイントが属する 1 つまたは複数のカテゴリ

説明： タスクに対するタスクトレースポイント <TTPointID> をロギングします。

注意事項： ロギングは TRACE_TASK_TRACEPOINT_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.4.2 LogTaskTracepointValue

構文：

```
void LogTaskTracepointValue(
    <TTpointId>,
    uint <val>,
    osTraceCategoriesType <catmask>)
```

引数：

TTpointId タスクトレースポイントの識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

val タスクトレースポイントにおいて送信される数値。val のサイズ（16 ビットか 32 ビット）は、コンフィギュレーション設定時に選択されたシステムタイムのサイズにより決まります。詳しくは『RTA-TRACE コンフィギュレーションガイド』を参照してください。

catmask このタスクトレースポイントが属する 1 つまたは複数のカテゴリ

説明： 指定されたタスクに対するタスクトレースポイント <TTPointID> と、16 ビットまたは 32 ビットの符号なし整数値をロギングします。

注意事項： ロギングは TRACE_TASK_TRACEPOINT_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.4.3 LogTaskTracepointData

構文：

```
void LogTaskTracepointData(
    <TTPointID>,
    osTraceDataPtr <dataPtr>,
    osTraceDataLength <length>,
    osTraceCategoriesType <catmask>)
```

引数：

TTPointID タスクトレースポイントの識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

dataPtr タスクトレースポイントと共に送信されるデータブロックへのポインタ

length <dataPtr> により参照されるデータブロックの長さ（バイト数）

catmask このタスクトレースポイントが属する 1 つまたは複数のカテゴリ

説明： 指定されたタスクに対するタスクトレースポイント <TTPointID> と、任意のバイナリデータをロギングします。

注意事項： ロギングは TRACE_TASK_TRACEPOINT_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.5 インターバル

インターバルを使うと、2 つのイベントの間の経過時間を測定することができます。インターバルは TimeTrace ビジュアライザ の専用の帯に表示されます。表示されるインターバルの識別子は、コンフィギュレーション設定時に任意に指定するか、または名前の代わりにデフォルトの数値を使用することもできます。

3.5.1 LogIntervalStart

構文：

```
void LogIntervalStart(
     <intervalId>,
     osTraceCategoriesType <catmask>)
```

引数：

intervalId インターバル の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

catmask このコールが属する 1 つまたは複数のカテゴリ

説明： <intervalId> により参照されるインターバル を開始します。

注意事項： ロギングは TRACE_INTERVALS_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.5.2 LogIntervalStartValue

構文：

```
void LogIntervalStartValue(
     <intervalId>,
     uint <val>,
     osTraceCategoriesType <catmask>)
```

引数：

intervalId インターバル の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

val インターバル 開始時に送信される数値。`val` のサイズ（16 ビットか 32 ビット）は、コンフィギュレーション設定時に選択されたシステムタイムのサイズにより決まります。詳しくは『RTA-TRACE コンフィギュレーションガイド』を参照してください。

catmask このコールが属する 1 つまたは複数のカテゴリ

説明： `<intervalId>` により参照されるインターバル を開始し、同時に 16 ビットまたは 32 ビットの符号なし整数値をロギングします。

注意事項： ロギングは `TRACE_INTERVALS_CLASS` クラスと `<catmask>` で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.5.3 LogIntervalStartData

構文：

```
void LogIntervalStartData(
     <intervalId>,
     osTraceDataPtr <dataPtr>,
     osTraceDataLength <length>,
     osTraceCategoriesType <catmask>)
```

引数：

intervalId インターバル の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

dataPtr インターバル 開始時に送信されるデータ ブロックへのポインタ

length `<dataPtr>` により参照されるデータ ブロックの長さ（バイト数）

catmask このコールが属する 1 つまたは複数のカテゴリ

説明： `<intervalId>` により参照されるインターバル を開始し、同時に任意のバイナリ データをロギングします。

注意事項： ロギングは `TRACE_INTERVALS_CLASS` クラスと `<catmask>` で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.5.4 LogIntervalEnd

構文：
```
void LogIntervalEnd(
    <intervalId>,
    osTraceCategoriesType <catmask>)
```

引数：

intervalId インターバル の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

catmask このコールが属する 1 つまたは複数のカテゴリ

説明： <intervalId> により参照されるインターバル を終了します。

注意事項： ロギングは TRACE_INTERVALS_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.5.5 LogIntervalEndValue

構文：
```
void LogIntervalEndValue(
    <intervalId>,
    uint <val>,
    osTraceCategoriesType <catmask>)
```

引数：

intervalId インターバル の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

val インターバル 終了時に送信される数値。val のサイズ（16 ビットか 32 ビット）は、コンフィギュレーション設定時に選択されたシステムタイムのサイズにより決まります。詳しくは『RTA-TRACE コンフィギュレーションガイド』を参照してください。

catmask このコールが属する 1 つまたは複数のカテゴリ

説明： <intervalId> により参照されるインターバル を終了し、同時に 16 ビットまたは 32 ビットの符号なし整数値をロギングします。

注意事項： ロギングは TRACE_INTERVALS_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.5.6 LogIntervalEndData

構文：
```
void LogIntervalEndData(
     <intervalId>,
     osTraceDataPtr <dataPtr>,
     osTraceDataLength <length>,
     osTraceCategoriesType <catmask>)
```

引数：

intervalId インターバル の識別子（サイズは設定に応じて異なります。3.2.1 項を参照してください。）

dataPtr インターバル 終了時に送信されるデータブロックのアドレス

length <dataPtr> により参照されるデータブロックの長さ（バイト数）

catmask このコールが属する 1 つまたは複数のカテゴリ

説明： <intervalId> により参照されるインターバル を終了し、同時に任意のバイナリデータをロギングします。

注意事項： ロギングは TRACE_INTERVALS_CLASS クラスと <catmask> で表されたカテゴリ（1 つまたは複数）について行われます。

使用できる OS： すべての OS

3.6 その他のロギング

3.6.1 LogProfileStart

構文：
```
void LogProfileStart(
     osTraceInfoType <profileId>)
```

引数：

profileId プロファイルの識別子

説明： 新しい実行時プロファイルへのスイッチオーバーをマークします。

注意事項： ロギングは TRACE_TASKS_AND_ISRS_CLASS クラスについて行われます。

使用できる OS： RTA-OSEK、Generic OS

3.6.2 LogCriticalExecutionEnd

構文：
```
void LogCriticalExecutionEnd(
     osTraceInfoType <critExecId>)
```

引数：

CritExecId クリティカル実行セクション（'Critical Execution Point'）の識別子

説明： クリティカルセクションの終了をマークします。スティミュラスに対するレスポンスを観察するために利用できます。

注意事項： ロギングは TRACE_TASKS_AND_ISRS_CLASS クラスについて行われます。

使用できる OS： RTA-OSEK、Generic OS

3.6.3 LogCat1ISRStart

構文： `void LogCat1ISRStart(`
`osTraceInfoType <IsrId>)`

引数：

IsrId カテゴリ 1 ISR の識別子

説明： カテゴリ 1 ISR の開始をマークします。このタイプの ISR はオペレーティングシステムにより制御されないので、マニュアル操作によるロギングが必要です。

注意事項： ロギングは `TRACE_TASKS_AND_ISRS_CLASS` クラスについて行われます。

使用できる OS： RTA-OSEK

3.6.4 LogCat1ISREnd

構文： `void LogCat1ISREnd(`
`osTraceInfoType <IsrId>)`

引数：

IsrId カテゴリ 1 ISR の識別子

説明： カテゴリ 1 ISR の終了をマークします。このタイプの ISR はオペレーティングシステムにより制御されないので、マニュアル操作によるロギングが必要です。

注意事項： ロギングは `TRACE_TASKS_AND_ISRS_CLASS` クラスについて行われます。

使用できる OS： RTA-OSEK

3.6.5 LogOverrunHook

構文： `void LogOverrunHook()`

引数： なし

説明： オーバーランフックがコールされたことをマークします。このフックは、タスクの実行時間が長すぎた場合に、そのタスクがターミネートした時点でコールされます。

注意事項： ロギングは `TRACE_ERRORS_CLASS` クラスについて行われます。

使用できる OS： RTA-OSEK

3.7 トリガ

3.7.1 ClearTrigger

構文： `void ClearTrigger(void)`

引数： なし

説明： トリガ条件をクリアすることにより、トリガが発生しないようにします。

注意事項： このコールより前にすでにトリガが発生していた場合は、イベントのロギングが引き続き行われます。

使用できる OS： すべての OS

3.7.2 TriggerNow

構文： `void TriggerNow(void)`

引数： なし

説明： トリガ条件とは無関係に、直ちにトリガを行います。

注意事項： トリガ条件が設定されていた場合、それらの条件はクリアされません。

使用できる OS： すべての OS

3.7.3 SetTriggerWindow

構文：

```
void SetTriggerWindow(
     UIntType <before>,
     UIntType <after>)
```

引数：

Before トリガイベントの前にアップロードするレコード数

After トリガイベントの後にアップロードするレコード数

説明： トリガの前後から何レコードをアップロードするかを設定します。

注意事項： アップロードするレコードの総数（`<before> + <after>`）は、使用できるバッファ全体のサイズより小さくなければなりません。
総レコード数がバッファサイズより大きい場合には、`<before>` が収まりきるように `<after>` が切り捨てられます。`<before>` だけでもバッファサイズより大きい場合には、`<before>` の値にはバッファサイズが設定され、`<after>` の値には ゼロが設定されます。

注記

1 つのトレースイベント が複数のトレースレコードに対して行われる場合があるので、トリガポイントの前と後のイベント数はこのコールに指定されるレコード数とは一致しません。また（たとえば `Log…Data()` コールで）トレースイベントが切り捨てられてしまい、データが不完全になったり、イベントが失われてしまう場合があります。
また、`after` の値が小さくなり過ぎないようにしてください。システムシャットダウンなど のターミナルイベントについてトリガする時には、`StopTrace()` を使用するようにしてください。

使用できる OS： すべての OS

3.7.4 TriggerOnActivation

構文：
```
void TriggerOnActivation(
        TaskType <taskId>)
```

引数：

taskId トリガを設定するタスクの識別子

説明： <taskId> を起動しようとする処理（つまり ActivateTask(<TaskId>) のコール）をトリガ条件として設定します。

注意事項： <taskId> に OSTRACE_TRIGGER_ANY を設定すると、すべての ActivateTask() コールによってトレースがトリガされるようになります。
このトリガは、明示的なタスク起動とタイムテーブルによるタスク起動の両方で発生します。また、アラームのロギングが有効になっていない場合には、アラームによるタスク起動においても発生します。
ただし ChainTask() により行われるタスク起動においては発生せず（3.7.5 項の TriggerOnChain を参照してください）、またタスク <taskId> が属するタスクセットの起動時においても発生しません（RTA-OSEK の場合のみ）。

使用できる OS： すべての OS

3.7.5 TriggerOnChain

構文：
```
void TriggerOnChain(TaskType <taskId>)
```

引数：

taskId トリガを設定するタスクの識別子

説明： タスク <taskId> をチェーニングしようとする処理（つまり ChainTask(<TaskId>) のコール）をトリガとして設定します。

注意事項： <taskId> に OSTRACE_TRIGGER_ANY を設定すると、すべての ChainTask() コールによってトレースがトリガされるようになります。

使用できる OS： RTA-OSEK、ERCOS^{EK}

3.7.6 TriggerOnTaskStart

構文：
```
void TriggerOnTaskStart(
        TaskType <taskId>)
```

引数：

taskId トリガを設定するタスクの識別子

説明： <taskId> の実行開始をトリガとして設定します。

注意事項： <taskId> に OSTRACE_TRIGGER_ANY を設定すると、すべてのタスクの実行開始によってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.7 TriggerOnTaskStop

構文： `void TriggerOnTaskStop(TaskType <taskId>)`

引数：

taskId トリガを設定するタスクの識別子

説明： <taskId> の終了をトリガとして設定します。

注意事項： <taskId> に OSTRACE_TRIGGER_ANY を設定すると、すべてのタスクの終了によってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.8 TriggerOnISRStart

構文： `void TriggerOnISRStart(ISRType <ISRId>)`

引数：

ISRId ISR の識別子

説明： <ISRId> の開始をトリガとして設定します。

注意事項： <ISRId> に OSTRACE_TRIGGER_ANY を設定すると、すべての ISR の開始によってトレースがトリガされるようになります。
<ISRId> はシステム生成時に使用された識別子でなければなりません。

使用できる OS： ERCOS^{EK}

3.7.9 TriggerOnISRStop

構文： `void TriggerOnISRStop(ISRType <ISRId>)`

引数：

ISRId ISR の識別子

説明： <ISRId> の終了をトリガとして設定します。

注意事項： <ISRId> に OSTRACE_TRIGGER_ANY を設定すると、すべての ISR の終了によってトレースがトリガされるようになります。
<ISRId> はシステム生成時に使用された識別子でなければなりません。

使用できる OS： ERCOS^{EK}

3.7.10 TriggerOnCat1ISRStart

構文： `void TriggerOnCat1ISRStart(ISRType <ISRId>)`

引数：

ISRId ISR の識別子

説明： <ISRId> の開始をトリガとして設定します。

注意事項： <ISRId> に OSTRACE_TRIGGER_ANY を設定すると、すべてのカテゴリ 1 ISR の開始によってトレースがトリガされるようになります。
<ISRId> はシステム生成時に使用された識別子でなければなりません。

使用できる OS： RTA-OSEK

3.7.11 TriggerOnCat1ISRStop

構文： `void TriggerOnCat1ISRStop(ISRType <ISRId>)`

引数：

ISRId ISR の識別子

説明： `<ISRId>` の終了をトリガとして設定します。

注意事項： `<ISRId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのカテゴリ 1 ISR の終了によってトレースがトリガされるようになります。
`<ISRId>` はシステム生成時に使用された識別子でなければなりません。

使用できる OS： RTA-OSEK

3.7.12 TriggerOnCat2ISRStart

構文： `void TriggerOnCat2ISRStart(ISRType <ISRId>)`

引数：

ISRId ISR の識別子

説明： `<ISRId>` の開始をトリガとして設定します。

注意事項： `<ISRId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのカテゴリ 2 ISR の開始によってトレースがトリガされるようになります。
`<ISRId>` はシステム生成時に使用された識別子でなければなりません。

使用できる OS： RTA-OSEK

3.7.13 TriggerOnCat2ISRStop

構文： `void TriggerOnCat2ISRStop(ISRType <ISRId>)`

引数：

ISRId ISR の識別子

説明： `<ISRId>` の終了をトリガとして設定します。

注意事項： `<ISRId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのカテゴリ 2 ISR の終了によってトレースがトリガされるようになります。
`<ISRId>` はシステム生成時に使用された識別子でなければなりません。

使用できる OS： RTA-OSEK

3.7.14 TriggerOnInitTaskStart

構文： `void TriggerOnInitTaskStart(AppModeType <mode>)`

引数：

mode アプリケーションモード

説明： アプリケーションモード `<mode>` の init タスクの開始をトリガとして設定します。

注意事項： この init タスクは、アプリケーションモード `<mode>` に入ると実行されます。

使用できる OS： ERCOS^{EK}

3.7.15 TriggerOnInitTaskStop

構文： `void TriggerOnInitTaskStop(AppModeType <mode>)`

引数：

`mode` アプリケーションモード

説明： アプリケーションモード `<mode>` の init タスクの終了をトリガとして設定します。

注意事項： この init タスクは、アプリケーションモード `<mode>` に入ると実行されます。

使用できる OS： ERCOS[EK]

3.7.16 TriggerOnGetResource

構文： `void TriggerOnGetResource(ResourceType <ResourceId>)`

引数：

`ResourceId` リソースの識別子

説明： リソース `<ResourceId>` を取得しようとする処理（`GetResource(<ResourceId>)` のコール）をトリガとして設定します。

注意事項： `<ResourceId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべての `GetResource()` コールによってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.17 TriggerOnReleaseResource

構文： `void TriggerOnReleaseResource(ResourceType <ResourceId>)`

引数：

`ResourceId` リソースの識別子

説明： リソース `<ResourceId>` を解放しようとする処理（`ReleaseResource(<ResourceId>)` のコール）をトリガとして設定します。

注意事項： `<ResourceId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべての `ReleaseResource()` コールによってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.18 TriggerOnSetEvent

構文： `void TriggerOnSetEvent (TaskType <TaskId>)`

引数：

TaskId タスクの識別子

説明： 指定されたタスクをターゲットとする `SetEvent()` コールをトリガとして設定します。

注意事項： `<TaskId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべての `SetEvent()` コールによってトレースがトリガされるようになります。

使用できる OS： RTA-OSEK、カスタム OS

> **注記**
>
> このコールの引数は、イベント ID ではなくタスク ID です。

3.7.19 TriggerOnTracepoint

構文： `void TriggerOnTracepoint(osTraceTracepointType <point>)`

引数：

point トレースポイント ID

説明： トレースポイント `<point>` のロギングをトリガとして設定します。

注意事項： `<point>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべての `LogTracepoint…()` コールによってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.20 TriggerOnTaskTracepoint

構文：

```
void TriggerOnTaskTracepoint(
     osTraceTaskTracepointType <point>,
     TaskType <task>)
```

引数：

point タスクトレースポイント ID

task タスク ID

説明： タスク `<task>` からのタスクトレースポイント `<point>` のロギングをトリガとして設定します。

注意事項： `<task>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのタスクからの `LogTaskTracepoint…()` コールによってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.21 TriggerOnIntervalStart

構文： `void TriggerOnIntervalStart(osTraceIntervalType <intervalId>)`

引数：

`intervalId` インターバル 識別子

説明： `<intervalId>` により参照されるインターバルの開始をトリガとして設定します。

注意事項： `<intervalId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべての `LogIntervalStart…()` コールによってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.22 TriggerOnIntervalEnd (TriggerOnIntervalStop)

構文：

```
void TriggerOnIntervalEnd(
   osTraceIntervalType <intervalId>)

void TriggerOnIntervalStop(
   osTraceIntervalType <intervalId>)
```

引数：

`intervalId` インターバル 識別子

説明： `<intervalId>` により参照されるインターバルの終了をトリガとして設定します。

注意事項： `<intervalId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべての `LogIntervalEnd…()` コールによってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.23 TriggerOnTimetableExpiry

構文： `void TriggerOnTimetableExpiry(TimeTableType <tt>)`

引数：

`tt` トリガを設定するタイムテーブルの識別子

説明： タイムテーブル `<tt>` の任意のポイントの満了をトリガとして設定します。

注意事項： `<tt>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのタイムテーブルのすべてのポイントの満了によってトレースがトリガされるようになります。

使用できる OS： ERCOS^{EK}

3.7.24 TriggerOnTickSchedule

構文： `void TriggerOnTickSchedule (ScheduleType <sched>)`

引数：

`sched` トリガを設定するスケジュールの識別子

説明： スケジュール `<sched>` の任意のポイントの満了をトリガとして設定します。

注意事項： `<sched>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのスケジュールのすべてのポイントの満了によってトレースがトリガされるようになります。

使用できる OS： RTA-OSEK

3.7.25 TriggerOnAdvanceSchedule

構文： `void TriggerOnAdvanceSchedule (ScheduleType <sched>)`

引数：

sched トリガを設定するスケジュールの識別子

説明： スケジュール `<sched>` の任意のポイントの満了をトリガとして設定します。

注意事項： `<sched>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのアドバンススケジュールのすべてのポイントの満了によってトレースがトリガされるようになります。

使用できる OS： RTA-OSEK

3.7.26 TriggerOnAlarmExpiry

構文： `void TriggerOnAlarmExpiry(AlarmType <alarm>)`

引数：

alarm トリガを設定するアラームの識別子

説明： `<alarm>` の満了をトリガとして設定します。

注意事項： `<alarm>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのアラームの満了によってトレースがトリガされるようになります。

使用できる OS： すべての OS

3.7.27 TriggerOnExplicitSendStateMessage

構文：
```
void TriggerOnExplicitSendStateMessage(
          STATEMESSAGE <messageId>)
```

引数：

messageId トリガを設定するステートメッセージの識別子

説明： ステートメッセージ `<messageId>` の送信をトリガとして設定します。

注意事項： `<messageId>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのステートメッセージの送信によってトレースがトリガされるようになります。
`<messageId>` は、システム生成時に使用された識別子でなければなりません。

使用できる OS： ERCOS^{EK}

3.7.28 TriggerOnExplicitReceiveStateMessage

構文： `void TriggerOnExplicitReceiveStateMessage(STATEMESSAGE <messageId>)`

引数：

messageId トリガを設定するステートメッセージの識別子

説明： ステートメッセージ <messageId> の受信をトリガとして設定します。

注意事項： <messageId> に OSTRACE_TRIGGER_ANY を設定すると、すべてのステートメッセージの受信によってトレースがトリガされるようになります。
<messageId> は、システム生成時に使用された識別子でなければなりません。

使用できる OS： ERCOS^{EK}

3.7.29 TriggerOnSendMessage

構文： `void TriggerOnSendMessage(SymbolicName <messName>)`

引数：

messName OSEK COM メッセージのシンボリック名

説明： OSEK COM メッセージ <messName> の送信をトリガとして設定します。

注意事項： <messName> に OSTRACE_TRIGGER_ANY を設定すると、すべての OSEK COM メッセージの送信によってトレースがトリガされるようになります。

使用できる OS： COM をサポートするすべての OS
これには、OS 計装キットによりロギングされるメッセージも含まれます。

3.7.30 TriggerOnReceiveMessage

構文： `void TriggerOnReceiveMessage(SymbolicName <messName>)`

引数：

messName OSEK COM メッセージのシンボリック名

説明： OSEK COM メッセージ <messName> の受信をトリガとして設定します。

注意事項： <messName> に OSTRACE_TRIGGER_ANY を設定すると、すべての OSEK COM メッセージの受信によってトレースがトリガされるようになります。

使用できる OS： COM をサポートするすべての OS
これには、OS 計装キットによりロギングされるメッセージも含まれます。

3.7.31 TriggerOnError

構文： `void TriggerOnError(StatusType <err>)`

引数：

err エラーコード

説明： エラー `<err>` をトリガとして設定します。

注意事項： `<err>` に `OSTRACE_TRIGGER_ANY` を設定すると、すべてのエラーの生成によってトレースがトリガされるようになります。
エラーコードの一覧は、各 OS のドキュメントを参照してください。

使用できる OS： すべての OS

3.7.32 TriggerOnShutdown

構文： `void TriggerOnShutdown(StatusType <stat>)`

引数：

stat 出口（'exit'）コード

説明： アプリケーションのシャットダウンをトリガとして設定します。

注意事項： —

使用できる OS： すべての OS

4 API の制限事項

4.1 はじめに

RTA-TRACE は、RTA-OSEK と ERCOS^{EK} の多くの API シンボルをマクロとして定義し直すことによって、OS のモニタリングを実現しています。ほとんどの場合、これについてユーザーは全く意識する必要はありませんが、まれに、本章に記述されているような問題点が発生する可能性があります。

4.2 問題の概要

`#define`文により引数を伴って定義されたマクロ内で1つの仮引数が2回以上使用されると、以下のような不具合が生じる可能性があります。

- 与えられた引数が不適当な結果を招くものであり、かつそれが 2 回以上評価される場合、その不適当な結果が重複して発生し、バグとなる可能性があります。
- 与えられた引数により、関数コールの実行時間が非常に長くなる場合、その引数が 2 回以上評価されると、オーバーヘッドが増大します。
- 引数自体がマクロである場合（たとえば、そのマクロを展開すると `if` 文や `switch` 文になる場合）、たとえ 2 回以上評価されなくても、コードサイズのオーバーヘッドが増大します。

以上の理由から、マクロの引数をマクロボディ内で 2 回以上使用することは望ましくありませんが、どうしてもそれを回避できない場合もあります。

4.3 該当するマクロ（ERCOS^{EK}）

以下のマクロには、2 回以上展開される引数があります。それらの引数は、ボールド体で表記されています。

```
ActivateTask(task)
GetResource(res)
ReleaseResource(res)
ChainTask(task)
TriggerOnActivation(task)
TriggerOnChain(task)
TriggerOnTaskStart(task)
TriggerOnTaskStop(task)
TriggerOnGetResource(res)
TriggerOnReleaseResource(res)
TriggerOnTaskTracepoint(point, task)
SendMessage(msg, pointer)
ReceiveMessage(msg, pointer)
```

4.4 該当するマクロ（RTA-OSEK）

以下のマクロには、2 回以上展開される引数があります。それらの引数は、ボールド体で表記されています。

```
ActivateTask(task)
ActivateTaskset(taskset)
ChainTask(task)
ChainTaskset(taskset)
TickSchedule(schedule)
AdvanceSchedule(schedule, status)
SendMessage(msg, pointer)
ReceiveMessage(msg, pointer)
GetResource(res)
ReleaseResource(res)
SetEvent(task, mask)
WaitEvent(mask)
ClearEvent(mask)
ShutdownOS(mode)
```

5 お問い合わせ先

製品サポートに関しては、各 ETAS 支社までお問い合わせください。

ドイツ（ETAS 本社）

ETAS GmbH

Borsigstr. 14
70469 Stuttgart
Germany

Phone: +49 (711) 8 96 61-0
Fax: +49 (711) 8 96 61-105
E-mail: sales@etas.de
WWW: www.etasgroup.com

日本

イータス株式会社

〒 220-6217
神奈川県横浜市西区
みなとみらい 2-3-5 クイーンズタワー C 17F

Phone: (045) 222-0900
Fax: (045) 222-0956
E-mail sales@etas.co.jp
WWW: www.etasgroup.com

韓国

ETAS Korea Co., Ltd.

3F Samseung Bldg
61-1, Yangjae-dong
Seocho-gu, Seoul
Republic of Korea

Phone: +82 (2) 5747 016
Fax: +82 (2) 5747 120
E-mail: sungik.hong@etas.co.kr

イギリス

ETAS Engineering Tools Application and Services Ltd.

Studio 3, Waterside Court
3rd Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ
UK

Phone: +44 (0) 1283 - 546512
Fax: +44 (0) 1283 - 548767
E-mail sales@etas-uk.net
WWW: www.etasgroup.com

フランス

ETAS S.A.S

1, place des Etats-Unis
SILIC 310
94588 Rungis Cedex
France

Phone: +33 (1) 56 70 00 50
Fax: +33 (1) 56 70 00 51
E-mail sales@etas.fr
WWW: www.etasgroup.com

北米

ETAS Inc.

3021 Miller Road
Ann Arbor, MI 48103
USA

Phone: +1 (888) ETAS INC
Fax: +1 (734) 997-9449
E-mail sales@etasinc.com
WWW: www.etasgroup.com

南米

UNIT

Rua Adolfo Maraccini, 399
c/o Sergio Augusto Alves
Campinas SP 13086 - 010
Brazil

Mobile: +55 19 9772 0793
Telefax: +55 (19) 3256 1939
E-mail: unit@mpc.com.br

索引

S

T