RTA-TRACE

OS 計装キットマニュアル（OS Instrumenting Kit Manual）

著作権について

Document TD00007-004

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、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ための機能は、特に優れています。

本書では、プリエンプティブシステムやサイクリック実行プログラム、あるいはスケジューラをまったく使用しないシステムなど、RTA-TRACE をさまざまな組込みシステムと共に使用する方法について説明します。

1.1 本書の対象ユーザー

本書は、組込みシステムの開発者を対象としています。RTA-TRACE をカスタマイズすることにより、現在 RTA-TRACE でサポートされていない組込みシステムで RTA-TRACE を使用できるようにする必要がある際に、お読みください。

本書の読者は、組込みシステム用の C 言語のプログラミング概念、および RTA-TRACE の操作に習熟している必要があります。

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。

コマンドプロンプトのセッションについては、出力される部分がクーリエ体（`courier`）で表記され、ユーザーが入力する部分がクーリエボールド体（**`courier bold`**）で表記されます。

GUI エレメントとのインタラクションについての記述では、エレメントのキャプションは**ボールド体（bold）**で表記されています。また、メニューなどの階層的なナビゲーションは矢印でレベルを区切り、たとえば、「メニューコマンド **Edit → Select All** を選択します。」、または「メニューから **Edit → Select All** を選択します。」のように表記されています。

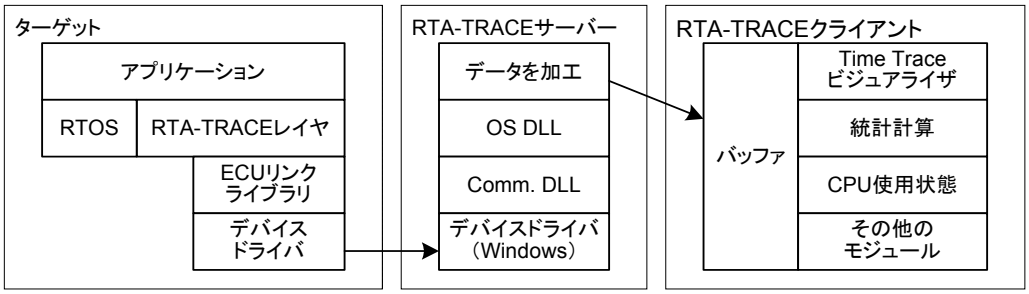
また PDF 文書において、目次や索引、および他の部分を参照する個所（例：「第 3 章を参照してください」の部分）については、その参照先へのリンクが設けられているので、必要な参照個所を素早く見つけることができます。

2 はじめに

RTA-TRACE は、実行中のアプリケーションの挙動を記録し、即時情報と要約情報を画面に表示します。これにより、実際のシステム挙動を正確に把握でき、リアルタイムに表示される情報や作成されたレポート、またさらに測定ツールを用いて、異常時の状態を詳しく分析することができます。

RTA-TRACE では、OS の動作（タスク起動、リソースのロック、アラームなど）とユーザー定義の事象を記録できますが、これらのことを行うためには、OS とアプリケーションの「計装」（'instrument'）、つまり計測可能な状態にするための装備が必要となります。

下図は、関連するコンポーネントを示しています。



RTA-TRACE は以下の OS に実装できます。

- RTA-OSEK
- ERCOS^{EK}
- その他の OS

本書では、上記の OS のうち、「その他の OS」をベースとするシステムを計装する際に使用するメカニズムについて説明します。これは、OS をまったく使用しないアプリケーションにも応用できます。

RTA-OSEK や ERCOS^{EK} とともに使用する場合については、別のドキュメントで説明しています。

2.1 メカニズム

ターゲットの RTA-TRACE レイヤは、トレースデータをトレースバッファに格納します。トレースデータは、OS またはアプリケーションコードから得られます。ECU リンクライブラリは、適切な通信メカニズムを通じてトレースバッファの内容を RTA-TRACE サーバーに送ります。

『OS 計装キット（OS Instrumenting Kit）』には、ターゲット上で使用されるコードと、PC 上で稼動する RTA-TRACE サーバーアプリケーション用のプラグインソフトウェアが含まれています。

ターゲットコードは計装（'instrumenting'）部とサポート（'support'）部に分かれています。「計装 API」は、ターゲットコード内に計装ポイントを配置するために使用される一連の C 関数（またはマクロ）を指し、「サポート API」は、計装 API をサポートするために必要な一連の C 関数とデータを指します。

API についての詳しい情報は、第 4 章と第 5 章を参照してください。

2.2 使い方

OS 計装キットには、コンパイルしてアプリケーションや OS と一緒にリンクできる C ソースコードが含まれています。正確なトレースデータを抽出するためには、ユーザーの責任において、アプリケーションまたは OS のコード内の適切な位置で計装 API を呼び出すようにしてください（これについては第 6 章を参照してください）。さらに、作成されたアプリケーションを、適切な ECU リンクテクノロジとリンクする必要があります。これについては、『RTA-TRACE ECU リンクガイド』を参照してください。

2.3 キットの内容

OS 計装キットは、`OS Instrumenting Kit¥RTLib¥` ディレクトリにある複数のソースファイルで構成されています。これらはマクロ定義（`RTapi.h` に含まれています）とサポートコード（`.c` ファイル）に大きく分けられます。

ECU リンクのソースコードは `OS Instrumenting Kit¥RTComm¥` ディレクトリに格納されています。

また、`OS Instrumenting Kit¥Example¥` ディレクトリに格納されているサンプルアプリケーションを参考にすれば、計装コールの組み込み方が具体的にわかります。

このサンプルアプリケーションはシングルスレッドで、OS サービスをまったく使用しません。そのため、Microsoft Windows PC など任意のプラットフォームで実行できます（組込みアプリケーションを作成しなくてもインスタレーションをテストできるように、Windows 用の実行ファイルがあらかじめ用意されています）。このアプリケーションにおいては、実例として組み込まれている OS API コールはすべてコメントアウトされています。そしてそれらのコールによる挙動を擬似的に行うことにより、マルチスレッドアプリケーションの挙動をシミュレートします。これらの OS API コールは、どれも `OSxxx()` という形になっています。

このアプリケーションはシングルスレッドなので、トレース情報をアップロードするための明示的コールが全体にわたって挿入されていますが、これはマルチスレッド環境では必要ありません。トレース情報がアップロードされるためのコードを実際のアプリケーションに挿入する方法は、『RTA-TRACE ECU リンクガイド』を参照してください。

3 トレースオブジェクトの記述

RTA-TRACE は、トレースの対象となる各種オブジェクト（タスク、メッセージ、リソース、トレースポイントなど）と、それら各オブジェクトについて発生するトレースイベントを取り扱います。各オブジェクトには、トレースデータストリーム内で識別できるように、トレース ID が割り当てられます。

RTA-TRACE がターゲットトレースデータを正しくデコードできるようにするために、トレースオブジェクト、およびその特性とトレース ID を RTA-TRACE に対して宣言する必要があります。この宣言は、ランタイムインターフェースファイル（拡張子が `.rta` のファイル）により行われます。

`.rta` ファイルのフォーマットは、OSEK 対応デバッガをサポートするために使用される ORTI 言語[1]をベースにしています。これは、既存の OSEK コード生成ツールで容易に RTA-TRACE をサポートできるようにするためです。ただし、非 OSEK システムにおいても、同等のファイルを生成するのはそれほど難しくありません。

3.1 ターゲットシステムについて記述する

`.rta` ファイルは、ターゲット上のトレースの対象となるオブジェクトについて、RTA-TRACE に宣言するためのものです。

ターゲットアプリケーションはトレースオブジェクトの集合として構成されています。オブジェクトには以下の種類があります。

- タスク
- ISR
- リソース
- カウンタ
- アラーム
- メッセージ
- プロセス
- プロファイル
- トレースポイント（= 任意のトレースイベント）
- タスクトレースポイント（= ある特定のタスクに関連付けられている任意のトレースイベント）
- インターバル（= ある特定の動作の所要時間を測定するために使用されるメカニズム）

ターゲット内に存在する上記のようなオブジェクトについて、`.rta` ファイルに記述します。

本章の残りの部分では、RTA-TRACE が認識できるオブジェクトを紹介します。RTA-TRACE は、主に OSEK/VDX のようなオペレーティングシステムとともに稼働するように設計されいて、このことは、RTA-TRACE がターゲットオブジェクトをモデル化する方法に反映されています。OSEK について詳しいユーザーの方には、以下の記述の多くの部分はなじみ深いものであるはずです。RTA-TRACE は OSEK に焦点を当てていますが、他の型のターゲット（サイクリック実行プログラムなど）にも使用できます。そのためには、`.rta` ファイル内の記述を、ターゲットの計装方法と確実に整合させるようにしてください。非 OSEK システムの計装上のヒントについては、第 6 章を参照してください。

---

1. OSEK/VDX OSEK Run Time Interface (ORTI) Part A: Language Specification”、バージョン 2.1.1、2002 年 3 月 4 日、http://www.osek-vdx.org/orti_documents.htm

3.2 .rta ファイルフォーマットの概要

以下に、.rta ファイルのフォーマットについて説明します。ここではすべての内容が説明されているわけではありませんが、.rta ファイルを作成するために必要な情報はすべて含まれています。

.rta ファイルは下の 2 つの主要セクションで構成されています。

宣言セクション（'Declaration section'） システムに存在する可能性のあるオブジェクトの型を宣言します。

情報セクション（'Information section'） システムに存在するすべてのオブジェクトについての情報が含まれます。各オブジェクトには、宣言セクションで宣言されている型が定義されます。

3.2.1 宣言セクション

宣言セクションでは、システムに存在する可能性のあるオブジェクトの型を、以下の形式で宣言します。

```
IMPLEMENTATION implementation_name {
  object type declarations
};
```

implementation_name は、ファイルのオーサーにより選択されたシステムの名前であり、有効な任意の識別子を定義できます。この識別子は C 言語の識別子と同じルールに従います。識別子およびキーワード（例：IMPLEMENTATION）については、大文字と小文字が区別されます。

オブジェクト型は以下の形式で宣言します。

```
object_type {
  attribute declarations
};
```

object_type はオブジェクト型の名前（例：TASK、ISR1、RESOURCE）です。必ず有効な識別子でなければなりません。

属性の宣言形式は何種類かあります。.rta ファイルに関連するものについて、下の表に示します。どの場合も、attribute_name が属性の名前で、これは有効な識別子でなければなりません。Label も属性の名前ですが、簡潔でわかりやすい名前を任意に指定できます。

STRING *attribute_name, "label";*	文字列の値を持つ属性の宣言です。
UINT16 *attribute_name, "label";*	16 ビットの符号なし整数値を持つ属性の宣言です。

属性の名前は、その属性を含むオブジェクト型の中でのみ有効です。つまり、複数のオブジェクト型に同じ名前の属性を定義することができます。

以下は、宣言セクションの簡単な例です。

```
IMPLEMENTATION rta_trace {
  OBJECT_A {
    STRING attr_a, "String attribute of OBJECT_A";
    UINT16 attr_b, "uint16 attribute of OBJECT_A";
  };
  OBJECT_B {
    STRING attr_a, "String attribute of OBJECT_B";
    UINT16 attr_c, "uint16 attribute of OBJECT_B";
    };
  OBJECT_C {
    STRING attr_a, "String attribute of OBJECT_C";
    UINT16 attr_d, "uint16 attribute of OBJECT_C";
  };
};
```

この宣言セクションでは、OBJECT_A、OBJECT_B、OBJECT_C という 3 つのオブジェクト型が宣言されています。OBJECT_Aにはattr_aという文字列属性とattr_bという16ビット符号なし整数属性、OBJECT_B には attr_a という文字列属性と attr_c という 16 ビット符号なし整数属性、さらに OBJECT_C には attr_a という文字列属性と attr_d という 16 ビット符号なし整数属性があります。

3.2.2 .rta の実装例

.rta の宣言セクション（Implementation セクション）に必要な内容は、トレースの列挙型データ（後述）以外は、アプリケーションに合わせて変更しなくてもよい場合が多いので、下の例をテンプレートとして利用することができます。

```
IMPLEMENTATION rta_trace  {
  OS {
    ENUM UINT8 [
      "E_OK" = 0,
      "E_OS_ACCESS" = 1,
      "E_OS_CALLEVEL" = 2,
      "E_OS_ID" = 3,
      "E_OS_LIMIT" = 4,
      "E_OS_NOFUNC" = 5,
      "E_OS_RESOURCE" = 6,
      "E_OS_STATE" = 7,
      "E_OS_VALUE" = 8,
      "E_OS_SYS_IDLE" = 16,
      "E_OS_SYS_AP_INVALID" = 17,
      "E_OS_SYS_AP_NULL" = 18,
      "E_OS_SYS_AP_READONLY" = 19,
      "E_OS_SYS_TS_INVALID" = 20,
      "E_OS_SYS_TS_READONLY" = 21,
      "E_OS_SYS_S_MODULO" = 22,
      "E_OS_SYS_S_INVALID" = 23,
      "E_OS_SYS_S_MISMATCH" = 24,
      "E_OS_SYS_STACK_FAULT" = 25,
      "E_OS_SYS_T_INVALID" = 26,
      "E_OS_SYS_R_PERMISSION" = 28,
      "E_OS_SYS_COUNTER_INVALID" = 29,
      "E_OS_SYS_CONFIG_ERROR" = 30,
      "E_OS_SYS_CALLEVEL" = 31,
      "E_COM_ID" = 32,
      "E_COM_BUSY" = 33,
      "E_COM_NOMSG" = 34,
      "E_COM_LIMIT" = 35,
      "E_COM_LOCKED" = 36,
      "E_COM_SYS_STOPPED" = 48,
      "Budget Overrun" = 255
    ] LASTERROR, "Last OSEK error";
    ENUM UINT8 [
      "NO_APPMODE" = 0,
      "OSDEFAULTAPPMODE" = 1
    ] CURRENTAPPMODE, "Current AppMode";
    ENUM UINT32 [
      "a" = 1,
      "b" = 2,
```

```
      "c" = 3,
      "d" = 4
   ] a, "OS_1";
   ENUM UINT32 [
      "t" = 44,
      "s" = 55,
      "r" = 66,
      "q" = 77,
      "p" = 88
   ] b, "OS_2";
   STRING vs_p_Fmt1, "StartOS Data format";
   STRING vs_p_Fmt2, "ShutdownOS Data format";
};
TASK {
   STRING vs_ID, "Trace ID";
   UINT16 vs_ACTIVATIONS, "Max activations";
   STRING vs_TYPE, "Conformance type";
   STRING vs_p_Pri, "Base priority";
   STRING vs_p_Disp, "Dispatch priority";
   STRING vs_InternalRes, "Internal resource ID";
   STRING vs_p_StackCeiling, "Stack limit";
   STRING vs_p_StackRange, "Stack range";
   STRING vs_p_Budget, "Budget";
   STRING vs_p_Excl, "Excluded?";
   STRING vs_p_OSEvents, "Events";
};
ISR2 {
   STRING vs_ID, "Trace ID";
   STRING vs_RESOURCES, "Resources";
   STRING vs_BUFFERING, "Buffering";
   STRING vs_p_Pri, "Base priority";
   STRING vs_p_Disp, "Dispatch priority";
   STRING vs_p_StackCeiling, "Stack limit";
   STRING vs_p_StackRange, "Stack range";
   STRING vs_p_Budget, "Budget";
   STRING vs_p_Excl, "Excluded?";
   STRING vs_p_Arb, "Arbitration";
};
ISR1 {
   STRING vs_ID, "Trace ID";
   STRING vs_BUFFERING, "Buffering";
   STRING vs_p_Pri, "Base priority";
   STRING vs_p_Disp, "Dispatch priority";
   STRING vs_p_StackCeiling, "Stack limit";
   STRING vs_p_StackRange, "Stack range";
   STRING vs_p_Budget, "Budget";
   STRING vs_p_Excl, "Excluded?";
   STRING vs_p_Arb, "Arbitration";
};
ISR0 {
   STRING vs_ID, "Trace ID";
```

```
   STRING vs_BUFFERING, "Buffering";
   STRING vs_p_Pri, "Base priority";
   STRING vs_p_Disp, "Dispatch priority";
   STRING vs_p_StackCeiling, "Stack limit";
   STRING vs_p_StackRange, "Stack range";
   STRING vs_p_Budget, "Budget";
   STRING vs_p_Excl, "Excluded?";
   STRING vs_p_Arb, "Arbitration";
};
ALARM {
   STRING vs_ID, "Trace ID";
   STRING ACTION, "Action";
   STRING vs_Owner, "Owning counter ID";
   STRING vs_Activates, "Activates";
   STRING vs_SetEvent, "Sets Event";
};
COUNTER {
   STRING vs_ID, "Trace ID";
   STRING vs_p_Fmt, "Data format";
};
MESSAGECONTAINER {
   STRING vs_ID, "Trace ID";
   STRING MSGNAME, "Message Name";
   STRING vs_CDATATYPE, "C type";
   STRING vs_p_Fmt, "Data format";
   STRING vs_Activates, "Activates";
   STRING vs_SetEvent, "Sets Event";
};
Trace {
   STRING vs_VERSION, "Trace version";
   STRING vs_p_TickDuration, "Stopwatch tick duration";
   STRING vs_p_MaxAbsTime, "Max Stopwatch value";
   STRING vs_p_BigEndian, "BigEndian";
   STRING vs_p_IntSize, "IntSize";
   STRING vs_ErrorFmt, "ErrorFmt";
   STRING vs_KindSize, "KindSize";
   STRING vs_InfoSize, "InfoSize";
   STRING vs_TimeSize, "TimeSize";
   STRING vs_TASKS_AND_ISRS, "Filter TASKS_AND_ISRS";
   STRING vs_STARTUP_SHUTDOWN, "Filter STARTUP_SHUTDOWN";
   STRING vs_ACTIVATIONS, "Filter ACTIVATIONS";
   STRING vs_COUNTERS_ALARMS, "Filter COUNTERS_ALARMS";
   STRING vs_SCHEDULES, "Filter SCHEDULES";
   STRING vs_RESOURCES, "Filter RESOURCES";
   STRING vs_INTERRUPT_LOCKS, "Filter INTERRUPT_LOCKS";
   STRING vs_ERRORS, "Filter ERRORS";
   STRING vs_OSEK_MESSAGES, "Filter OSEK_MESSAGES";
   STRING vs_MESSAGE_DATA, "Filter MESSAGE_DATA";
   STRING vs_SWITCHING_OVERHEADS, "Filter SWITCHING_OVERHEADS";
   STRING vs_OSEK_EVENTS, "Filter OSEK_EVENTS";
   STRING vs_TRACEPOINTS, "Filter TRACEPOINTS";
```

```
        STRING vs_TASK_TRACEPOINTS, "Filter TASK_TRACEPOINTS";
        STRING vs_INTERVALS, "Filter INTERVALS";
        STRING vs_STACK, "Filter STACK";
    };
    Profile {
        STRING vs_ID, "Trace ID";
        STRING vs_Owner, "Owning Task/ISR";
    };
    TaskTracepoint {
        STRING vs_ID, "Trace ID";
        STRING vs_Owner, "Owning Task/ISR";
        STRING vs_p_Fmt, "Data Format";
    };
    Tracepoint {
        STRING vs_ID, "Trace ID";
        STRING vs_p_Fmt, "Data Format";
    };
    CritExec {
        STRING vs_ID, "Trace ID";
        STRING vs_Owner, "Owning Task/ISR/Profile";
        STRING vs_p_Budget, "Budget";
    };
    Interval {
        STRING vs_ID, "Trace ID";
        STRING vs_p_Fmt, "Data Format";
    };
    Resource {
        STRING vs_ID, "Trace ID";
        STRING vs_p_Pri, "Task priority";
        STRING vs_p_Isr, "ISR priority";
        STRING vs_Owner, "Owning resource";
        STRING vs_Internal, "Internal?";
    };
    Schedule {
        STRING vs_ID, "Trace ID";
    };
};
```

3.2.3 情報セクション

情報セクションには、システム内に存在する各オブジェクトを定義します。各オブジェクトは、宣言セクション（Implementation セクション）の後に、以下の形式で順に定義します。

```
object_type object_name {
  attribute_name0 = “value0”;
  …
  attribute_nameN = “valueN”;
};
```

object_type は宣言セクションで宣言されたオブジェクト型の名前です。object_name はオブジェクトの名前で、これは必ず有効な識別子でなければなりません。attribute_name0 ～ attribute_nameN は object_type として宣言されている属性の名前で、value0 ～ valueN は属性に割り当てられる値です。ある型のオブジェクトを定義する際、そのオブジェクト型宣言で宣言されているすべての属性に値を割り

当てなければならないというわけではありません。次項の各オブジェクト型の属性の一覧表には、その属性の値設定が必須であるか、そうでない（任意設定）かが示されています（3.3 項を参照してください）。また、情報セクションにおいて、宣言セクションで宣言されたすべてのオブジェクト型のオブジェクトを定義しなければならないというわけではありません。

以下に、前出の宣言セクションの例に対応する情報セクションの例を示します。

```
TASK myTask {
   vs_ID = "7";
   vs_ACTIVATIONS = "1";
   vs_TYPE = "ECC2";
   vs_p_Pri = "2";
   vs_p_Disp = "2";
   vs_p_OSEvents = "ea2.1 ";
   vs_p_StackCeiling = "169";
   vs_p_StackRange = "37";
};
Resource RES_SCHEDULER {
   vs_ID = "1";
   vs_p_Pri = "5";
};
Trace Trace {
   vs_VERSION = "2.0.0";
   vs_p_TickDuration = "250";
   vs_p_MaxAbsTime = "65535";
   vs_p_BigEndian = "1";
   vs_p_IntSize = "32";
   vs_ErrorFmt = "%99E";
   vs_KindSize = "1";
   vs_InfoSize = "1";
   vs_TimeSize = "2";
   vs_TASKS_AND_ISRS = "false";
   vs_STARTUP_SHUTDOWN = "runtime";
   vs_ACTIVATIONS = "true";
   vs_COUNTERS_ALARMS = "true";
   vs_SCHEDULES = "true";
   vs_RESOURCES = "true";
   vs_INTERRUPT_LOCKS = "true";
   vs_ERRORS = "true";
   vs_OSEK_MESSAGES = "true";
   vs_MESSAGE_DATA = "true";
   vs_SWITCHING_OVERHEADS = "true";
   vs_OSEK_EVENTS = "true";
   vs_TRACEPOINTS = "true";
   vs_TASK_TRACEPOINTS = "true";
   vs_INTERVALS = "true";
   vs_STACK = "true";
};
```

3.3 トレースオブジェクト

ここでは、.rta ファイル内に定義するトレースオブジェクトの情報について記述します。以下の記述は、RTA-TRACE により理解されるオブジェクト型と、各オブジェクト型の属性一覧とで構成されています。オブジェクト型の中には RTA-TRACE に固有のものもありますが、多くは OSEK ORTI 言語によっても使用されます。OSEK OS ツールにより生成された ORTI ファイルを元に作業を行う場合は、そのファイルの内容を見てみると、TASK 型のオブジェクトの例がすでに記述されています。

RTA-TRACE は、使用するオブジェクトと属性についての情報を .rta ファイルからのみ取得するので、.rta ファイル内に、RTA-TRACE が使用しないオブジェクトや属性も定義しておくことができます。そのため、自動生成されたファイルを編集して使用する場合、RTA-TRACE が使用しないオブジェクトや属性をわざわざ削除する必要はありません。

注記

数値は必ず 10 進数で定義してください。

以降の説明で引用されている「フォーマット文字列」については、第 8 章を参照してください。

3.3.1 オブジェクト型 - OS

オペレーティングシステムの全般的な特性と列挙型の値（後述）を定義するために、OS 型のオブジェクトを 1 つだけ使用します。

属性	型	説明
vs_p_Fmt1	STRING（任意指定）	osTraceOSStart() というトレース API に関連付けられている値の表示形式を指定するフォーマット文字列です。典型的な値は “%98E” で、これは OS オブジェクト内の CURRENTAPPMODE という列挙型データに開始値を割り当てます。
vs_p_Fmt2	STRING（任意指定）	osTraceOSExit() というトレース API に関連付けられている値の表示形式を指定するフォーマット文字列です。典型的な値は “%99E” で、これは OS オブジェクト内の LASTERROR という列挙型データに初期値を割り当てます。

列挙型データ を使うと、ターゲットアプリケーション内の数値を RTA-TRACE で使用できるテキスト記述に対応付けることにより、ターゲットの挙動をよりわかりやすく視覚化できるようになり、エラーコードを解読する際などに便利です。

列挙型データは、OS 実装節（’implementation clause’）内に以下の形で定義します。

```
ENUM <type> [
    "<name>" = <value>,
    …
] <enum_name>, "<enum_reference>";
```

例：虹の色を表す RAINBOW は、以下のように定義できます。

```
ENUM UINT8 [
    "RED" = 0,
    "ORANGE" = 1,
    "YELLOW" = 2,
    "GREEN" = 3,
    "BLUE" = 4,
    "INDIGO" = 5,
    "VIOLET" = 6
] RAINBOW, "OS_1";
```

トレースデータのフォーマット定義により、列挙型データは “%*n*E” という形で参照され、この中の *n* は enum_reference フィールドで指定されます（つまり “OS_*n*”）。

列挙型データには、2 つの特殊なものがあります。1 つは “OS_98” と等価と見なされる `CURRENTAPPMODE` で、もう 1 つは “OS_99” と等価と見なされる `LASTERROR` です。

3.3.2 オブジェクト型 - Trace

`Trace` オブジェクト型は、RTA-TRACE にトレース対象システム全般についての情報を提供するために使用されます。この型のオブジェクトは必ず 1 つだけ存在していなければなりません。

属性	型	説明
`vs_VERSION`	STRING（必須）	`.rta` ファイルのバージョン。一般的には “2.0.0” を設定します。
`vs_p_TickDuration`	STRING（必須）	トレース対象システムのタイムスタンプクロックの 1 チックに相当するナノ秒数（小数点を含むこともできます）。
`vs_p_MaxAbsTime`	STRING（必須）	トレース対象システムのクロックの最大絶対値。例えば、トレース対象システムのクロックが 16 ビットなら、この値は “65535” です。
`vs_p_BigEndian`	UINT8（必須）	トレース対象システムがリトルエンディアンなら “0”、ビッグエンディアンなら “1” です。
`vs_p_IntSize`	UINT8（必須）	トレース対象システムの C 言語の `int` 型のビット数。
`vs_ErrorFmt`	STRING（必須）	エラー情報の表示形式を定義するフォーマット文字列。一般的には `%99E` です。
`vs_KindSize`	UINT8（必須）	トレース対象システムの “kind” フィールドのバイト数で、1 か 2 のいずれかです。“kind” フィールドにはイベントの種類（タスク開始、リソース取得など）を示す記述子を設定します。
`vs_InfoSize`	UINT8（必須）	トレース対象システムの “info” フィールドのバイト数で、1 か 2 のいずれかです。“info” フィールドにはイベントが関係するオブジェクト（‘Task1’、‘Resource7’ などの識別子）を設定します。
`vs_TimeSize`	UINT8（必須）	トレースレコードに時刻を記録するために使用されるバイト数。2 か 4（つまり、16 ビットか 32 ビット）のいずれかです。

`vs_TASKS_AND_ISRS`
`vs_ERRORS`
`vs_ACTIVATIONS`
`vs_SCHEDULES`
`vs_RESOURCES`
`vs_OSEK_EVENTS`
`vs_TRACEPOINTS`
`vs_INTERVALS`
`vs_MESSAGE_DATA`
`vs_COUNTERS_ALARMS`
`vs_STARTUP_SHUTDOWN`
`vs_OSEK_MESSAGES`
`vs_INTERRUPT_LOCKS`
`vs_SWITCHING_OVERHEADS`
`vs_TASK_TRACEPOINTS`
	STRING（任意指定）	ビルド時オプションによりトレースクラス（『RTA-TRACE ユーザーズガイド』を参照してください）ごとに、トレースを行うかどうかを指定できます。ビルド時において、そのクラスを除外したり（文字列値 "false"）または必ずトレースされるようにしたり（文字列値 "true"）、あるいは実行時にフィルタリングするようにしたり（文字列値 "`runtime`"）することができます。

注記

現在のバージョンにおいては、`vs_KindSize` と `vs_InfoSize` の値は同じでなければなりません。これらのフィールドのサイズは標準 ID とコンパクト ID（標準：16 ビット、コンパクト：8 ビット）の使用と相互に関連しています。詳しくは 5.1 項を参照してください。

3.3.3 タスク、ISR、プロセス、プロファイルについての概要

タスクは、プログラム実行エレメントの 1 つであり、「スレッド」と呼ばれる場合もあります。タスクはプログラムコードを実行して特定の機能を実行します。RTA-TRACE においては、タスクには数々なステート（unknown、activated、running、preempted、waiting）があり、各タスクには優先度が割り当てられています。

プロセッサは 1 つしかないので、一度に 1 つのタスクしかコードを実行できません。つまり、running（実行中）ステートになることができるタスクは一度に 1 つだけで、どのタスクを実行中にするかは OS が管理します。通常、OS は最高の優先度を持つ実行可能タスクが必ず他のタスクより優先的に実行されるようにします。現在実行中のタスクよりも優先度の高いタスクが実行できる状態になると、OS は現在実行中のタスクをサスペンドさせ、優先度の高い方のタスクの実行を開始します。この場合、「優先度の高いタスクが低いタスクを *プリエンプトした*」と言います。

`.rta` ファイル内には、優先度について 2 種類の記述を行います。タスクの「ベース優先度」は、どのタスクを起動するかを決める際に、そのタスクを他のタスクに比べてどの程度優先するかを定義します。またタスクの「ディスパッチ優先度」は、タスクが実行を開始する際に取得する優先度を示すために使用されます。これは、一度に 1 つのグループのタスクしか絶対に実行されないようにするための手段として使用できます。たとえば、ベース優先度が 1 でディスパッチ優先度が 2 のタスク A とベース優先度が 2 でディスパッチ優先度が 2 のタスク B があるとします。OS は、A と B のどちらを先に実行するか選ばなければならなくなると、ベース優先度の高い B を選びます。しかし、B が起動されたときに A が実行中になっていた場合には、両タスクのディスパッチ優先度は同じなので B は A をプリエンプトしません。また、1 つのタスクのベース優先度とディスパッチ優先度は同じでも構いません。

ISR は、割込みを処理するターゲットコードエレメントです。一般に、ISR の方がタスクよりも優先度が高くなっています。割込み信号が発生すると、OS またはハードウェアは実行中のタスクまたは ISR の優先度を調べ、新しい割込みの優先度の方が高い場合には、実行中のタスク／ ISR はその割込みによりプリエンプトされることになります。新しい割込みの優先度の方が低い場合には、その ISR は実行できる最高優先度の ISR になるまで実行されません。ISR の優先度はターゲットハードウェアにより決定される場合もあります（ISR 優先度の定義は、ターゲットに応じて異なります）。

RTA-TRACE は OSEK のようなカテゴリ 1 およびカテゴリ 2 の割込みをサポートしています。カテゴリ 1 の割込みは、すぐに実行されることを目的としているので、OS API 関数やサービスへのアクセスを必要としません。カテゴリ 2 の割込みは OS により管理され、その ISR では OS API 関数のサブセットを使用できます。またカテゴリ 0 の割込みも用意されています。これは OSEK ではありませんが、他の割込みと同様に使用できます。

RTA-TRACE においては、ISR はタスクとまったく同じように扱われます。

プロセスは、一般に、タスクまたは ISR 内で実行される小規模のコードで、特定のサブファンクションを実行します。プロセスは最初から最後まで 1 つのタスク／ ISR により実行されます。1 つのタスクまたは ISR の中に複数のプロセスを設け、次々に実行させることができます。RTA-TRACE では、プロセスの開始ポイントと終了ポイントを容易に見ることができます。1 つのプロセスは 1 つのタスク／ ISR にしか所有できません。

（この場合のプロセスの概念は、UNIX や Windows などのような OS におけるプロセスの概念とは異なります。）

プロファイルは、1 つのプログラム全体の実行パスについて記述する手段です。たとえば、ある実行時条件に基づいて多くの選択的分岐の中から 1 つを実行する ISR を計装することにより、呼び出しのたびにどの実行プロファイル（‘CAN_Rx_Interrupt’、‘Timer3_Expiry’、‘ADC_complete’ など）を実行しようとしているのかをレポートさせることができます。

注記

タスク、プロセス、および ISR の識別子は同じ数値空間を共有するため、これらのオブジェクトの `vs_ID` 値は絶対に重複しないようにしてください。

3.3.4 オブジェクト型 - Task

TASK オブジェクトは OS タスクについて記述するために使用されます。TASK オブジェクトは、システム内の各タスクにつき 1 つ存在しなければなりません。

属性	型	説明
vs_ID	STRING（必須）	タスクオブジェクトのトレース識別子。
vs_p_Pri	STRING（必須）	タスクのベース優先度。値が小さいほど優先度も低くなります。
vs_p_Disp	STRING（必須）	タスクのディスパッチ優先度。ベース優先度と同じかそれより高く設定されます。3.2 項のタスクについての説明を参照してください。
vs_ACTIVATIONS	UINT16（任意指定）	認識できる未処理のタスク起動の最大数。単純な OSEK ‘BCC1’ タスクの場合、起動要求はキューイングされないので、この値は ‘1’ になります。他のタスクについては、OS は最大 4 回の起動要求をキューイングできるので、それに応じた値を設定できます。
vs_TYPE	STRING（必須）	タスク型についての短い記述。OSEK システムの場合、これは “BCC1”、“BCC2”、“ECC1”、および “ECC2” ですが、他の値も設定できます。
vs_InternalRes	STRING（任意指定）	タスクが実行されている時に暗黙的にロックされる内部リソースの vs_ID。内部リソースを複数のタスクにより共有させることにより、一度にそれらのタスクのうちの 1 つしか実行できないようにすることができます。
vs_p_StackCeiling	STRING（任意指定）	このタスクの実行中にロギングされるどのイベントについても予想される最大スタック値。この値はバイト数です。
vs_p_StackRange	STRING（任意指定）	このタスクの実行中に予想されるスタックの最大使用量。
vs_p_Budget	STRING（任意指定）	このタスクについて宣言されている実行バジェット、つまり、このタスクが実行可能な最大ストップウォッチチック数。
vs_p_Excl	STRING（任意指定）	このタスクをトレースしない場合には “1” を設定します。このタスクをトレースする場合にはこの属性を指定しないでください。
vs_p_OSEvents	STRING（任意指定）	タスクが待機する OSEK イベントの名前。ECC タスクについてのみ指定します。<eventname>:mask のペアをスペースで区切った形式で指定します。たとえば、"ea1.1 ea2:8" のように指定すると、マスク値 1 の ea1 というイベントとマスク値 8 の ea2 というイベントを表します。

3.3.5 オブジェクト型 - ISR2 ／ ISR1 ／ ISR0

ISR オブジェクトは、割込みサービスルーチンについて記述するためのものです。

注記

タスク、プロセス、および ISR の識別子は同じ数値空間を共有するため、これらのオブジェクトの vs_ID 値は絶対に重複しないようにしてください。

属性	型	説明
vs_ID	STRING（必須）	ISR オブジェクトのトレース識別子。
vs_p_Pri	STRING（必須）	ISR のベース優先度。値が小さいほど優先度も低くなります。
vs_p_Disp	STRING（必須）	ISR のディスパッチ優先度。ベース優先度と同じかそれより高い値です。
vs_p_Arb	STRING（必須）	ISR のアービトレーション順序。同じ優先度の 2 つの ISR が同時に実行可能状態になった場合、ターゲットハードウェアはアービトレーション値が最も高い ISR に最初に対応します。この順序は、しばしばターゲットハードウェアにより決められています。
vs_p_StackCeiling	STRING（任意指定）	トレース対象システムのスタックの最大サイズ。
vs_p_StackRange	STRING（任意指定）	このタスクの実行中にタスクが使用すると予想されるスタックの最大量。
vs_p_Budget	STRING（任意指定）	この ISR について宣言された実行バジェット。つまり、この ISR が実行できる最大ストップウォッチチック数。
vs_p_Excl	STRING（任意指定）	この ISR をトレースしない場合には “1” を設定します。この ISR をトレースする場合にはこの属性を指定しないでください。

3.3.6 オブジェクト型 - Process

Process オブジェクトは、一般的には、タスクの一部として実行される小規模のコードについて記述するために使用されます。これは機能ブロックを区分するための手段の 1 つです。

属性	型	説明
vs_ID	UINT16（必須）	プロセスオブジェクトのトレース識別子。 各 Process オブジェクトに 1 以上のユニークな番号を付ける必要があります。
vs_Owner	UINT16（必須）	このプロセスを所有するタスクまたは ISR の vs_ID。

3.3.7 オブジェクト型 - Profile

Profileオブジェクトは、タスク／ISRが実行時にさまざまな機能を実行する可能性があるような場合に、実際にどのコード部分が実行されているのかを RTA-TRACE に知らせる目的でのみ使用されます。

注記

タスク、プロセス、および ISR の識別子は同じ数値空間を共有するため、これらのオブジェクトの vs_ID 値は絶対に重複しないようにしてください。

属性	型	説明
vs_ID	STRING（必須）	プロファイルオブジェクトのトレース識別子。
vs_Owner	STRING（必須）	このプロファイルを所有するタスクまたは ISR の vs_ID。

3.3.8 オブジェクト型 - Resource

リソースは、アプリケーションの実行順序を制御するために用いられるオブジェクトです。X というタスクまたは ISR があるリソースをロック（取得）してしまうと、他のタスクや ISR は、そのリソースを X がアンロック（解放）するまではロックすることができません（リソースは “mutex” とも呼ばれます）。OSEK

オペレーティングシステムは「優先度シーリングプロトコル」を使用してリソースを実装しています。つまり、各リソースには優先度が付けられていて、その値はそのリソースをロックすることのできるすべてのタスク／ ISR の最大ディスパッチ優先度と等しくなっています。タスク／ ISR がリソースをロックすると、その実効ディスパッチ優先度がそのリソースの優先度の値まで引き上げられます。これにより、そのリソースをロックできる他のタスク／ ISR が、OS により実行されないようになります。リソースをロックしていたタスクがそのリソースをアンロックすると、そのタスクの実効ディスパッチ優先度は元の値に戻ります。OSEK では、ターゲットによっては、タスクと ISR でリソースを共有することができます（他の OS の場合、リソースはセマフォ（バイナリセマフォまたは計数セマフォ）とも呼ばれます）。

Resource オブジェクトは、すべてのリソース型（標準リソース、内部リソース、リンクリソース）について存在していなければなりません。

`Resource` オブジェクトはリソースについて記述するために使用されます。

属性	型	説明
`vs_ID`	STRING（必須）	リソースオブジェクトのトレース識別子。 各リソースオブジェクトには、1 以上のユニークな番号を付ける必要があります。
`vs_p_Pri`	STRING（右の説明を参照してください）	リソースをタスクでしかロックできない場合の、このリソースのシーリング優先度。ISR でもこのリソースをロックできる場合には、この設定を省略してください。
`vs_p_Isr`	STRING（右の説明を参照してください）	リソースを ISR でロックできる場合の、このリソースのシーリング優先度。このリソースを ISR でロックできない場合には、この設定を省略してください。
`vs_Owner`	STRING（任意指定）	このリソースの所有者のトレース ID。リンクリソースの場合のみ存在します。
`vs_Internal`	STRING（任意指定）	この値を “1” にすると、このリソースの型は ‘internal’ または ‘automatic’ と見なされます。このリソースのトレース ID と一致する `vs_InternalRes` 値を持つタスクは、開始時／終了時にこのリソースを自動的にロック／アンロックするものと判断されます。

3.3.9 COUNTER オブジェクト型

カウンタは、アラーム（次の項を参照してください）と緊密に連結されます。OSEK アラームは、将来のある時点でアクティビティを発生させるための OS リソースです（アラームは、さらに周期的挙動の実装にも使用できます）。アラームは満了時に所定のアクション（タスクを起動する、コールバック関数を呼び出す、ECC タスク用のイベントをセットするなど）を実行します。満了時刻はコンフィギュレーション設定時または実行時に設定されます。アラームはカウンタに関連付けられ、そのカウンタは所定のソース（周期的タイマ、外部スティミュラス、タスクなど）によってチックされます。

`COUNTER` オブジェクトはカウンタについて記述するために使用されます。カウンタは、アラームを駆動するために使用されるメカニズムです。アラーム満了時刻とカウンタのカレント値が一致するとアラームが実行されます。

属性	型	説明
`vs_ID`	STRING（必須）	カウンタオブジェクトのトレース識別子。 各 COUNTER オブジェクトには、1 以上のユニークな番号を付ける必要があります。
`vs_p_Fmt`	STRING（任意指定）	任意のカウント値の表示形式を定義するフォーマット文字列。

3.3.10 オブジェクト型 - ALARM

ALARM オブジェクトは、アラームについて記述するためのものです。

属性	型	説明
vs_ID	STRING（必須）	アラームオブジェクトのトレース識別子。 各 ALARM オブジェクトには、1 以上のユニークな番号を付ける必要があります。
vs_Owner	STRING（必須）	このアラームを所有するカウンタのトレース ID。
vs_p_Action	STRING（任意指定）	アラーム満了時に発生する事象についての記述。これは RTA-TRACE フローティングヒントのテキストとして表示されます。
vs_Activates	STRING（任意指定）	このアラームによって起動されると見なされているタスクのトレース ID。これが定義されると、RTA-TRACE により ‘activate’ という情報が自動的に挿入されます。
vs_SetEvent	STRING（任意指定）	“<num1>:<num2>” という形式の文字列は、タスク ID が <num1> でイベントマスクが <num2> のタスクに属するイベントをこのアラームがセットすることを意味します。これが定義されると、RTA-TRACE により ‘set event’ という情報が自動的に挿入されます。

3.3.11 オブジェクト型 - MESSAGECONTAINER

OSEK OS に関連する「OSEK COM」という通信規格では、タスク間のメッセージ送信手段が定義されています。この「メッセージ」はメッセージコンテナを使って送信されます。メッセージコンテナには、メッセージの C 言語データ型、キューイング可能なメッセージの数、および必要に応じて、メッセージ送信時または受信時の通知（タスクの起動、イベントのセット、コールバック関数の呼び出し）の情報を定義します。

MESSAGECONTAINER オブジェクトは、OSEK 方式のメッセージについて記述するためのものです。コンフィギュレーションを適宜設定することにより、メッセージの内容を RTA-TRACE に表示させることができます。

属性	型	説明
vs_ID	STRING（必須）	メッセージオブジェクトのトレース識別子。 各 MESSAGECONTAINER オブジェクトには、1 以上のユニークな番号を付ける必要があります。
vs_p_CType	STRING（任意指定）	メッセージに使用される C 言語のデータ型。
vs_p_Fmt	STRING（任意指定）	メッセージデータの表示形式を定義するフォーマット文字列。
vs_Activates	STRING（任意指定）	このメッセージが起動することになっているタスクのトレース ID。これを設定すると、RTA-TRACE により ‘activate’ 指示が自動的に挿入されるので、ターゲット上で記録する必要はありません。
vs_SetEvent	STRING（任意指定）	“<num1>:<num2>” という形の文字列は、タスク ID が <num1> でイベントマスクが <num2> のタスクに属するイベントをこのメッセージがセットすることを意味します。これが定義されると、RTA-TRACE により ‘set event’ という情報が自動的に挿入されるので、ターゲット上で記録する必要はありません。

3.3.12 オブジェクト型 - Tracepoint

トレースポイントは、任意のイベントの発生をロギングするために使用されます。ターゲットでは、プログラムの任意のポイントにトレースポイントを設定することができます。

`Tracepoint` オブジェクトはトレースポイントについて記述するためのオブジェクトです。`Tracepoint` オブジェクトは、特定の名前またはフォーマット文字列を割り当てる必要がなければ、前もって宣言しなくても使用することができます。

属性	型	説明
`vs_ID`	STRING（必須）	トレースポイントオブジェクトのトレース識別子。各トレースポイントオブジェクトには、1 以上のユニークな番号を付ける必要があります。
`vs_p_Fmt`	STRING（任意指定）	トレースポイントに関連付けられているデータの表示に使用されるフォーマット文字列。

3.3.13 タスクトレースポイント

タスクトレースポイントは特殊な形態のトレースポイントで、そのポイントをロギングするタスク／ ISR に関連付けられます。RTA-TRACE では、タスクトレースポイントは、それがロギングされた時に実行中だったタスク／ ISR とともに取得されます。

`TaskTracepoint` オブジェクトは、タスクトレースポイントについて記述するためのものです。`TaskTracepoint` オブジェクトは、特定の名前またはフォーマット文字列を割り当てる必要がなければ、前もって宣言しなくても使用することができます。

属性	型	説明
`vs_ID`	STRING（必須）	タスクトレースポイントオブジェクトのトレース識別子。 各 TaskTracepoint オブジェクトには、1 以上のユニークな番号を付ける必要があります。
`vs_Owner`	STRING（任意指定）	このタスクトレースポイントを所有するタスク／ ISR のトレース ID。この指定を省略すると、トレース対象システム内の、このオブジェクトのトレース ID を指定するすべてのタスクトレースポイントがこの記述を使用します。
`vs_p_Fmt`	STRING（任意指定）	トレースポイントに関連付けられているデータの表示に使用されるフォーマット文字列。

3.3.14 オブジェクト型 - Interval

インターバルは、ある動作に要する時間の長さを測定するために使用されます。ターゲットプログラムコード内のその動作の前後に、インターバルの開始と終了についての計装を行います。

`Interval` オブジェクトは、インターバルについて記述するためのものです。`Interval` オブジェクトは、特定の名前またはフォーマット文字列を割り当てる必要がなければ、前もって宣言しなくても使用することができます。

属性	型	説明
`vs_ID`	STRING（必須）	インターバルオブジェクトのトレース識別子。 各インターバルオブジェクトには、1 以上のユニークな番号を付ける必要があります。
`vs_p_Fmt`	STRING（任意指定）	インターバルに関連付けられているデータの表示に使用されるフォーマット文字列。

3.3.15 オブジェクト型 - CritExec

`CritExec` オブジェクトは、タスク／ ISR ／プロファイル内のクリティカルな実行ポイントを表すために使用されます。タスクトレースポイントと似ていて、一般的にはコードの特定セクションの完了をマークするために使用されます。RTA-TRACE は、タスク／ ISR の開始から各クリティカル実行ポイントまでの最短／最長実行時間をモニタできます。

属性	型	説明
`vs_ID`	STRING（必須）	CritExec オブジェクトのトレース識別子。 各 CritExec オブジェクトには、1 以上のユニークな番号を付ける必要があります。
`vs_Owner`	STRING（任意指定）	このクリティカル実行ポイントを所有するタスク／ ISR のトレース ID。
`vs_p_Budget`	STRING（任意指定）	この CritExec オブジェクトについて宣言される実行時間。つまり、クリティカル実行ポイントが発生するまでに経過すると予想される最大ストップウォッチチック数 。

4 マクロ API の計装

ここで説明するマクロは、RTapi.h というヘッダファイル内で定義されます。

トレースイベントは、供給されるヘッダファイル内で定義されているマクロを使用することにより、トレースバッファに格納されます。各トレースイベントについて、OS オブジェクトの各挙動への対応付けはそれぞれ異なるため、Time-Trace ビジュアライザにもそれぞれ独自の形式で表示されます。ここでは、使用可能なマクロとその典型的な使い方、さらにビジュアライザへの表示形式について説明します。

詳細については、製品 CD に含まれているサンプルアプリケーションを参考にしてください。

注記

ビジュアライザに意味のあるトレースデータを表示させるために、使用する各オブジェクト参照を .rta ファイルに定義しておく必要があります（第 3 章を参照してください）。

4.1 OS

本項に記載されている API マクロは、OS そのものの機能を扱います。そのため、OS ベースでないシステムを計装する場合には、一部のマクロは使用できない場合があります。

4.1.1 osTraceOSStart

用法： `osTraceOSStart(<value>)`

説明： OS ／スケジューラ／システムが起動されたことを示します。OSEK システムの場合、これはたとえばスタートアップフック内に配置します。
<value> は OS により異なります。システムがどの動作モードで稼働しているかを示すこともできます。OS オブジェクトの属性 vs_p_Fmt1 を適切に定義すると、この値を RTA-TRACE で表示できます。

4.1.2 osTraceOSExit

用法： `osTraceOSExit(<value>)`

説明： OS ／スケジューラ／システムがシャットダウンしたことを示します。OSEK システムの場合、これはたとえばシャットダウンフック内に配置します。
<value> は OS により異なります。システムが停止した原因を示すこともできます。OS オブジェクトの属性 vs_p_Fmt2 を適切に定義すると、この値を RTA-TRACE で表示できます。

4.1.3 osTraceSchedulerEntry

用法： `osTraceSchedulerEntry()`

説明： 計装されたシステムにおいて、オペレーティングシステムのスケジューラにちょうど入ろうとしている所であることを示します。具体的には、優先度の高いタスクが新たにレディ状態になったため、または割込みが発生したためにプリエンプションが行われる直前のタイミングで使用します。
割込みハンドラの場合、このイベントは割込ハンドラ内のできるだけ早い位置に配置します。

4.1.4 osTraceSchedulerExit

用法： `osTraceSchedulerExit()`

説明： 計装されているシステムにおいて、オペレーティングのシステムスケジューラがちょうど終了しようとしていることを示します。

4.2 タスク／ISR

この計装キット（'Instrumenting Kit'）では、タスクと割込みハンドラ （ISR）は多くの特性を共有しているので、これらには *Tasks*（タスク）という汎用グループ名が与えられています。そのため、タスクと ISR の識別子が重複しないようにすることが重要です（たとえば、1 つのシステムに ID が 3 のタスクと ID が 3 の ISR の両方を定義することはできません）。

プロファイルのロギングは、標準の RTA-TRACE 計装関数を用いて行われます。これについては、『RTA-TRACE ユーザーズガイド』で説明されています。

4.2.1 osTraceTaskActivate

用法： `osTraceTaskActivate(<task_id>)`

説明： タスク起動が要求されたことを示します。OSEK システムでは、この API は `ActivateTask()` 関数の中（または直前）のできるだけ早い位置に配置します。

注記 `<task_id>` はタスクを参照するものでなければなりません。

4.2.2 osTraceTaskStart

用法： `osTraceTaskStart(<task_id>)`

説明： 指示されたタスクが開始されたことを示します。
OS ベースでないシステムの場合は、たとえば重要なコードブロックの開始を示します。

4.2.3 osTraceTaskEnd

用法： `osTraceTaskEnd(<task_id>)`

説明： 指示されたタスクが終了したことを示します。

4.2.4 osTraceCat1Start

用法： `osTraceCat1Start(<isr1_id>)`

説明： 指示されたカテゴリ 1 ISR が開始されたことを示します。
このコールは、割込みハンドラ内のできるだけ早い位置に挿入してください。

4.2.5 osTraceCat1End

用法： `osTraceCat1End(<isr1_id>)`

説明： 指示されたカテゴリ 1 ISR が終了したことを示します。
このコールは、割込みハンドラ内のできるだけ遅い位置に挿入してください。

4.2.6 osTraceCat2Start

用法： `osTraceCat2Start(<isr2_id>)`

説明： 指示されたカテゴリ 2 ISR が開始されたことを示します。
このコールは、割込みハンドラ内のできるだけ早い位置に挿入してください。

4.2.7 osTraceCat2End

用法： `osTraceCat2End(<isr2_id>)`

説明： 指示されたカテゴリ 2 ISR が終了したことを示します。
このコールは、割込みハンドラ内のできるだけ遅い位置に挿入してください。

4.2.8 osTraceProcessStart

用法：
`osTraceProcessStart(<process_id>)`
`osTraceProcessStart(<task_id>)`

説明： プロセスが開始されたことを示します。プロセスは、タスクに含まれているサブファンクションです。
`<process_id>` がわからない場合には、このプロセスを所有しているタスクの ID を代わりに指定すると、プロセスが順序どおりに実行されると想定してどのプロセスを指示するべきかをビジュアライザが推察します。
非 OS システムの場合は、下位関数の開始をマークするために使用できます。

4.2.9 osTraceProcessEnd

用法：
`osTraceProcessEnd(<process_id>)`
`osTraceProcessEnd(<task_id>)`

説明： プロセスが終了したことを示します。
プロセスの終了は省略しても差し支えありません。RTA-TRACE は、次のプロセスが開始するか、このプロセスの属するタスクが終了すると、プロセスが終了したものと推測します。

4.3 タスク／ISR スイッチング API

4.3.1 osTraceTaskSchedulerEntry

用法： `osTraceTaskSchedulerEntry()`

説明： タスクがリスケジューリングポイントを提供したこと（OSEK システムの場合は `Schedule()` コール）を示します。
協調スケジューリングシステムでは、これはタスクが制御を返すポイントになります。

4.3.2 osTraceTaskSchedulerExit

用法： `osTraceTaskSchedulerExit()`

説明： リスケジューリングポイントが呼び出し側タスクに戻されたことを示します。

4.3.3 osTraceInterruptHandlerEntry

用法： `osTraceInterruptHandlerEntry()`

説明： 割込みが認識されたことを示します。このポイントから割込みハンドラ本体の開始（`osTraceCat`*`n`*`Start()` 関数によりロギングされます）までには少し時間がある場合があります。
このコールは割込み認識プロセス内のできるだけ早い位置に配置して、オペレーティングシステムのオーバーヘッドを測定できるようにしてください。

4.3.4 osTraceInterruptHandlerExit

用法： `osTraceInterruptHandlerExit()`

説明： 割込みが認識されたことを示します。割込みハンドラ本体の終了（`osTraceCat`*`n`*`End()` 関数によりロギングされます）からこのポイントまでには少し時間がある場合があります。
このコールは、割込みハンドラ内のできるだけ遅い位置に配置してください。

4.3.5 osTraceTaskSleep

用法： `osTraceTaskSleep(<task_id>)`

説明： タイムスライスの終了時などに、タスクがスリープ状態に入ることを示します。タスクがタイムスライスの処理を意識することはないはずなので、このイベントはオペレーティングシステム内から生成されます。
一般的にタイムスライス式の OS では、タスクは OS により起動されてタスク開始イベントがロギングされ、各タイムスライスごとに sleep と wake がペアで使用されます。

4.3.6 osTraceTaskWake

用法： `osTraceTaskWake(<task_id>)`

説明： タスクがスリープ後にウエイクすること、つまり、タスクに新しいタイムスライスが割り当てられたことを示します。

4.4 イベント

以下の API 関数は OSEK 方式のイベント挙動についてのものです。イベントは必ずマスクの形で定義します。

4.4.1 osTraceEventWaitEntry

用法：
```
osTraceEventWaitEntry(<task_id>,
      EventMaskType <event_mask>)
```

説明： タスクが OSEK イベント（`<event_mask>`）がセットされるのを待っていることを示します。そのイベントがセットされるとタスクの実行が再開されます。

4.4.2 osTraceEventWaitExit

用法： `osTraceEventWaitExit(<task_id>)`

説明： OSEK イベントを待っていたタスクが再開されたことを示します。

4.4.3 osTraceEventSet

用法：
```
osTraceEventSet(<task_id>,
       EventMaskType<event_mask>)
```

説明： `<task_id>` により参照されるタスク用に、`<event_mask>` で指定されたイベントがセットされることを示します。

注記 イベントは任意のタスクからセットできますが、`<task_id>` は、イベントが設定されるタスクへの参照でなければなりません。

4.4.4 osTraceEventClear

用法：
```
osTraceEventClear(<task_id>,
             EventMaskType<event_mask>)
```

説明： `<task_id>` により参照されるタスクについて、`<event_mask>` で指定されたイベントがクリアされることを示します。

4.5 リソース

以下の API 関数はリソースについてのものです。

4.5.1 osTraceResourceGet

用法： `osTraceResourceGet(<resource_id>)`

説明： リソース `<resource_id>` がロックされることを示します。

4.5.2 osTraceResourceRelease

用法： `osTraceResourceRelease(<resource_id>)`

説明： 指示されたリソースがアンロックされることを示します。

4.6 アラームとカウンタ

以下の関数により、カウンタとアラームを計装できます。これらのメカニズムは、スケジュールやプログラムされたタスク起動のタイムテーブルを計装する場合にも使用できます。

4.6.1 osTraceCounterTick

用法：

```
osTraceCounterTick(<counter_id>,
          TickType <counter_value>)
```

説明： 指示されたカウンタがチックされていることを示すために使用され、これによって新しいカウンタ値も提供されます。

4.6.2 osTraceAlarmExpire

用法： `osTraceAlarmExpire(<alarm_id>)`

説明： 指示されたアラームが満了したことを示します。

4.7 メッセージ

これらの API 関数では、メッセージをトラックできます。

4.7.1 osTraceMessageSend

用法：

```
osTraceMessageSend(<msg_id>)
osTraceMessageSendData(<msg_id>,
                       <data_ptr>,
                       <data_len>)
```

説明： 指示されたメッセージが送信されたことを示します。
上記の用法の 2 番目の形式でコールすれば、メッセージ識別子だけでなく実際のメッセージの内容もロギングできます。

4.7.2 osTraceMessageReceive

用法：

```
osTraceMessageReceive(<msg_id>)
osTraceMessageReceiveData(<msg_id>,
                          <data_ptr>,
                          <data_len>)
```

説明： 指示されたメッセージについて、メッセージ受信関数がコールされたことを示します。
上記の用法の 2 番目の形式でコールすれば、メッセージ識別子だけでなく実際のメッセージの内容もロギングできます。

4.8 割込みの処理

4.8.1 osTraceInterruptAllDisable

用法：

```
osTraceInterruptAllDisable()
```

説明： すべての割込みがディセーブルになることを示します。

4.8.2 osTraceInterruptAllEnable

用法：

```
osTraceInterruptAllEnable()
```

説明： すべての割込みがイネーブルになることを示します。

4.8.3 osTraceInterruptAllSuspend

用法：

```
osTraceInterruptAllSuspend(
                    <nesting_count>)
```

説明： 割込みがサスペンド状態になることを示します。割込みレベルが上げられた場合には `<nesting_count>` がゼロになり、そうでない場合には正数になります。`<nesting_count>` は ‘SuspendAll’ コールのたびに 1 ずつ大きくなります。

4.8.4 osTraceInterruptAllResume

用法：

```
osTraceInterruptAllResume(
                    <nesting_count>)
```

説明： 対応する ‘Suspend All’ コール時の割込みレベルに戻ることを示します。割込みレベルが下げられた場合には `<nesting_count>` がゼロになり、そうでない場合には正数になります。`<nesting_count>` は ‘Resume All’ コールのたびに 1 ずつ小さくなります。

4.8.5 osTraceInterruptOSSuspend

用法： `osTraceInterruptOSSuspend(<nest_count>)`

説明： OS レベル以下の割込みがサスペンド状態になっていることを示します。割込みレベルが上げられた場合には `<nest_count>` はゼロになり、そうでない場合には正数になります。`<nest_count>` は 'Suspend OS' コールのたびに 1 ずつ大きくなります。

4.8.6 osTraceInterruptOSResume

用法： `osTraceInterruptOSResume(<nest_count>)`

説明： 対応する 'Suspend OS' コール時の割込みレベルに戻ることを示します。割込みレベルが下げられた場合には `<nest_count>` がゼロになり、そうでない場合には正数になります。`<nest_count>` は 'Resume OS' コールのたびに 1 ずつ小さくなります。

4.9 エラーレポーティング

この API はエラーを計装するために使用されるものです。エラーコードに列挙型変数を使用することにより、コードの代わりに任意のテキストをビジュアライザに表示させることをお勧めします。

4.9.1 osTraceError

用法： `osTraceError(<error_code>)`

説明： エラーがレポートされたことを示します。エラーコードの表示形式は、.rta ファイル内の OS オブジェクトの `vs_ErrorFmt` 属性により規定されます。

5 サポート API の計装

第 4 章で説明されている API マクロ以外に、C コードの形でいくつかの関数が供給されています。これらの関数はトレースレコードをトレースバッファに挿入したり、トレースバッファを管理したりする処理を行います。RTA-TRACE ターゲットコードの通信エレメントについては、『RTA-TRACE ECU リンクガイド』で別に説明されています。

このコードは 'vanilla' C で書かれていて、通常は特に修正する必要はありませんが、ターゲットに固有なコード拡張（データエレメントの near 領域への配置など）が必要な場合もあります。そのような修正については本書では説明していませんので、詳しい情報が必要な場合は ETAS までお問い合わせください。

これらの「サポート API」が正しく機能するようにするためには、ターゲット固有のマクロや関数をユーザーが作成する必要があります。ここでは、それらについて説明します。

5.1 使用される型

以下のような基本型を、`RTLib¥RTtarget.h` というファイルに定義してください。

名前	説明
`Int8Type`	符号付き 8 ビット整数
`UInt8Type`	符号なし 8 ビット整数
`Int16Type`	符号付き 16 ビット整数
`UInt16Type`	符号なし 16 ビット整数
`Int32Type`	符号付き 32 ビット整数
`UInt32Type`	符号なし 32 ビット整数
`BooleanType`	論理値
`IntType`	プラットフォームにとって「自然な」整数
`UIntType`	プラットフォームにとって「自然な」符号なし整数
`osTraceEventMaskType`	イベントビットマスクに使用される型。ECC の OSEK OS を計装する場合、この型は `EventMaskType` として定義する必要があります。
`osTraceTickType`	カウンタ値の取得に使用される型。OSEK OS を計装する場合、この型は `TickType` として定義する必要があります。

上記のもの以外に、トレーシングライブラリにより使用される `osTraceTimeType`、`osTraceInfoType`、`osTraceKindType` という 3 つの基本型があります。これらの型のサイズは使用する識別子が標準（16 ビット）かコンパクト（8 ビット）か、あるいは時刻が標準（32 ビット）かコンパクト（16 ビット）かにより決まります。デフォルトの処理では、標準の識別子（16 ビット）と時刻（32 ビット）が使用されます。

コンパクトの時刻は `COMPACT_TIME` というプリプロセッサシンボルが定義されている場合に使用され、コンパクト識別子は `COMPACT_ID` というプリプロセッサシンボルが定義されている場合に（'kind' および 'info' 型について）使用されます。これらの識別子は `RTconfig.h` というファイルに設定されています（5.2 項を参照してください）。

これは下の表に示すとおりです。

型名	対応する型	
	標準	コンパクト
`osTraceTimeType`	`UInt32Type`	`UInt16Type`
`osTraceKindType`	`UInt16Type`	`UInt8Type`
`osTraceInfoType`	`UInt16Type`	`UInt8Type`
`osTraceCategoriesType`	`UInt32Type`	
`osTraceClassesType`	`UInt16Type`	

注記

標準／コンパクトの設定は、.rta ファイル内の Trace オブジェクトに定義されている info ／ kind のサイズと一致していなければなりません。

5.2 ターゲットコンフィギュレーション

ターゲットコンフィギュレーションの詳細は、RTconfig.h というファイル内に定義します。このファイルはユーザーのアプリケーションに固有のものなので、アプリケーションディレクトリに格納してください。このファイルには、.rta ファイル内に定義されているコンフィギュレーションパラメータと同じものを設定します。

注記

このファイルの内容を変更した場合には、ライブラリコードをビルドし直す必要があります。

注記

ユーザーの責任において、RTconfig.h とユーザーアプリケーションの .rta ファイルの内容（バッファサイズ、コンパクト時刻、コンパクト識別子）は常に一致させてください。

OSTRACE_ENABLED	このシンボルを定義すると、ターゲット上のトレースが有効になります。トレースが必要でない場合には、このシンボルを定義しないでください。
OSTRACEBUFFERSIZE	トレースバッファのサイズ（レコード数）。一般的には、システムを正確に分析するために十分な連続データを取得するためには 200 ～ 600 レコードのバッファサイズが必要です。
COMPACT_TIME	このシンボルを定義するとコンパクト表現の時刻が使用され（5.1 項を参照してください）、定義しないと標準表現が使用されます。
COMPACT_ID	このシンボルを定義するとコンパクト表現の ‘kind’ および ‘info’ フィールドが使用され（5.1 項を参照してください）、定義しないと標準のサイズが使用されます。
OSTRACE_TRIGGERING_ENABLED	ターゲット上でランタイムトリガが必要な場合には、このシンボルを定義してください。このシンボルを定義しないでおくと、ランタイムトリガチェックが行われないので、トレーシングコードのサイズが小さくなります。
OSTRACE_USING_COMMLINK	ターゲットからのトレースデータ転送に ECU リンク が使用される場合には、このシンボルを定義してください。このシンボルを定義した場合、…¥RTcomm¥ ディレクトリ下のファイルもユーザーのアプリケーションに組み込む必要があります。このシンボルを定義しない場合は、デバッガインターフェースが使用されます。

OSTRACE_ACTIVATIONS_FLTR

OSTRACE_TASKS_AND_ISRS_FLTR

OSTRACE_RESOURCES_FLTR

OSTRACE_PROCESSES_FLTR

OSTRACE_OSEKEVENTS_FLTR

OSTRACE_ERRORS_FLTR

OSTRACE_ALARMS_FLTR

OSTRACE_OSEK_MESSAGES_FLTR

OSTRACE_INTERRUPT_LOCKS_FLTR

OSTRACE_SWITCHING_OVERHEADS_FLTR

OSTRACE_STARTUP_AND_SHUTDOWN_FLTR

OSTRACE_TRACEPOINTS_FLTR

OSTRACE_TASK_TRACEPOINTS_FLTR

OSTRACE_INTERVALS_FLTR

各種クラスのイベントについて、トレーシングサポートを定義します。これらすべてのシンボルを、以下のいずれかの値に定義する必要があります。

OSTRACE_ALWAYS、
OSTRACE_NEVER、
OSTRACE_RUNTIME

ランタイムにおけるクラスの有効化／無効化については、サンプルアプリケーションの demo_preemption() ルーチン （main.c 内）を参考にしてください。

注記

コードサイズを小さく保つため、トレースのオン／オフを実行時に切り替える必要がない場合には、OSTRACE_RUNTIME ではなく OSTRACE_ALWAYS か OSTRACE_NEVER を使用してください。

5.3 osTraceGetSystemTime

プロトタイプ： `osTraceTimeType osTraceGetSystemTime(void);`

説明： イベントを挿入するすべてのコードに対してシステムタイム（時刻値）を供給します。
osTraceTimeType のサイズは COMPACT_TIME シンボルの設定に応じて 16 ビットか 32 ビットになります。

5.4 osTraceRunningTaskID

プロトタイプ： `osTraceInfoType osTraceRunningTaskID (void)`

説明： 現在実行中のタスク、つまり現在 CPU を占有しているタスクのトレース識別子を供給します。
この関数は、LogTaskTracepoint(…) という API 関数群によって、ランタイムトリガを実装するために使用します。ランタイムトリガが有効になっていない場合には、この関数は 0 を返します。タスクトレースポイントがまったく使用されない場合には、この関数は必要ありません。

5.5 割込み処理マクロ

OS 計装キットは、割込みレベルを処理するために以下のようなマクロを使用します。

マクロ名： `RESERVE_PREV()`
`SAVE_PREV()`
`DISABLE_INTRPTS()`
`RESTORE_PREV()`

説明： `RESERVE_PREV()` は割込みレベルを後でセーブするための変数を宣言します。
`SAVE_PREV()` はカレントの割込みレベルを `RESERVE_PREV()` マクロで宣言した変数にセーブします。
`DISABLE_INTRPTS()` は割込みをディセーブルにします。
`RESTORE_PREV()` は `SAVE_PREV()` にセーブされている割込みレベルをリストアします。

5.6 その他の API 関数

その他の API 関数（トレースの開始／終了、トレースポイントのロギング、トリガの使用など）の機能については、『RTA-TRACE ユーザーズガイド』に説明されています。

6 計装上のヒント

この「OS 計装キット」を使えば、オペレーティングシステムを使用するしないに関わらず、さまざまな組込みアプリケーションのシナリオを構築できます。

計装に関する大まかなガイドラインを以下に示しますので、さらに詳しい情報が必要な場合は、ETAS のサポート窓口まで問い合わせください。

6.1 全般的なヒント

対になっている API 関数（例： `osTraceTaskStart()` と `osTraceTaskEnd()`）は、常に正しくネストさせることが重要です。たとえば、セクションを閉じる API 関数（`osTraceTaskEnd()`）が省略されると、RTA-TRACE はバックグラウンドタスクの実行状態ではなく、プリエンプションのレベルの上昇を表示します。

このことは、すべての入口／出口、および開始／終了の関数に適用されます。

以下の例の場合、2 つのタスク終了パスの両方に `osTraceTaskEnd()` 関数が存在していなければなりません。

```
void task_a(void)
{
   osTraceTaskStart(TASKA)
   if (on_button_pressed()) {
      switch_system_on();
   } else {
      osTraceTaskEnd(TASKA);
      return;
   }
   check_system_status();

   osTraceTaskEnd(TASKA);
   return;
}
```

6.2 機能ブロック

重要な機能ブロック（つまり OSEK(time) タスク）を構成するコードは、タスクオブジェクトの形にモデル化し、ブロック内のできるだけ早い位置に `osTaskStart` コールを配置し、できるだけ遅い位置に `osTaskEnd` コールを配置してください。またこのブロックを実行させる特定のアクティビティが存在する場合には、それを `osTaskActivate` コールでマークするのがよいでしょう。`.rta` ファイル内で適切なタスクタイプ（`vs_TYPE`）を選択してください。多くの場合、BCC1 が最適です。

機能ブロックの表現は、プロセスを使用し、`osProcessStart` コールと `osProcessEnd` コールを適切に配置することにより実現できます。

6.3 周期的イベント

周期的イベントのモデル化は、カウンタとアラームを使って行うことができます。一般的には、一定のコードを周期的に実行させる「チック」イベントがあります。この「チック」イベントをカウンタとしてモデル化し、アラームを使用してコード部分（あるいはタスク）が起動された時刻を示すようにできます。

タイムスライス式システムの場合、‘Sleep’ トレースコールと ‘Wake’ トレースコールを適宜に挿入してください。

7 OS 計装キットに含まれるファイル

ここでは、計装キットライブラリディレクトリ（¥RTLib¥）に格納されているファイルと、各ファイルに含まれている関数について説明します。

7.1 RTapevnt.c

osTraceAppendEventMask	トレースレコードとイベントマスクをバッファに追加します（osTraceAppendData をコールします）。

7.2 RTapnd.c

osTraceAppend	トレースレコードをバッファに追加します（osTraceWriteTraceRecord をコールします）。

7.3 RTapnddt.c

osTraceAppendData	トレースレコードと任意のデータバイトをバッファに追加します（osTraceWriteTraceRecord をコールします）。

7.4 RTapndvl.c

osTraceAppendVal	トレースレコードと整数データをバッファに追加します（osTraceWriteTraceRecord をコールします）。

7.5 RTbef.c

osTraceBufferEmptyFunction	バッファが空になるとコールされます。

7.6 RTbreak.c

osTraceBreakLabel	デバッガ組込みラベル：ECU リンクではなくデバッガリンクを使用している場合、バッファが満杯になったことを検知するために、この関数でブレイクポイントを設定します。

7.7 RTbwcf.c

osTraceBufferWrCheckFunction	一般的にはバッファ満杯時にバッファへの書き込みをチェックするためにコールされます。

7.8 RTctick.c

`osTraceCounterTick1`	トレースレコードとカウンタ値をバッファに追加します（`osTraceAppendData` をコールします）。

7.9 RTcto.c

`CheckTraceOutput`	『RTA-TRACE ECU リンクガイド』を参照してください。

7.10 RTdata.c

このファイルには、OS 計装キット実装変数が格納されています。これらの変数は、キットを正しく機能させるために必要なものです。

7.11 RTfin.c

`osTraceFinished`	実行したデータトレースを正しく終了させます。

7.12 RTsetrep.c

`SetTraceRepeat`	『RTA-TRACE ユーザーズガイド』を参照してください。

7.13 RTsettrg.c

`SetTriggerConditions`	トリガサポート関数

7.14 RTsetwin.c

`SetTriggerWindow`	トリガサポート関数

7.15 RTstbt.c

`StartBurstingTrace`	『RTA-TRACE ユーザーズガイド』を参照してください。

7.16 RTstfr.c

`StartFreeRunningTrace`	『RTA-TRACE ユーザーズガイド』を参照してください。

7.17 RTstop.c

`StopTrace`	『RTA-TRACE ユーザーズガイド』を参照してください。

7.18 RTsttt.c

`StartTriggeringTrace`	『RTA-TRACE ユーザーズガイド』を参照してください。

7.19 RTtrgsup.c

`osTraceCheckForTrigger` `TriggerNow`	トリガサポート関数

7.20 RTwrrec.c

`osTraceWriteTraceRecord`	トレースレコードをバッファに書き込みます（リエントラントではありません）。

8 フォーマット文字列（'Format Strings'）

フォーマット文字列で、各トレースアイテムのデータの出力形式を指定することができます。単純な数値データは 1 つのフォーマット指定子（'format specifier'）で出力し、複雑なデータ（C 言語の構造体など）の場合は、データポインタをデータブロックの前後に移動させながら、複雑なフォーマット指定子を使用してデータを出力します。

フォーマット文字列が指定されていないと、データは以下のように出力されます。

- データサイズがターゲットの `int` 型を超えない場合は、データは "%d" と指定されたものとみなされて出力されます。
- 上記以外の場合は、以下のように HEX コードでダンプされます。

```
0000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

- 最大 256 バイトまで出力されます。

注記：フォーマット記述子が定義されていると、ターゲットのエンディアンが考慮されますが、HEX コードのダンプ出力の場合は、ターゲットのメモリが 1 バイトずつ出力されます。このため、`%x` というフォーマット記述子を使用した場合、HEX ダンプの内容とは違う出力内容になる場合があります。

8.1 フォーマット規則

- フォーマット文字列は、二重引用符（"）で囲みます。
- フォーマット文字列には 2 種類のタイプのオブジェクトを含めることができます。1 つは出力ストリームにそのままコピーされる通常の文字で、もう 1 つは、イベントとともに供給されるデータを変換して出力するためのフォーマットエレメントです。
- フォーマットエレメントは、`%` 文字と、桁数を表わす数字、そして 1 つの文字で構成されます。ただし `%E` のみは例外です。以下の項を参照してください。
- フォーマットエレメントは、以下の表の規則に従って変換され、その結果が出力文字列に加えられます。
- 特殊なフォーマットエレメント `%%` は、`%` と出力されます。
- 通常の文字や変換方法に加え、「バックスラッシュ（= 円記号）- エスケープシーケンス」を用いて特殊な文字を出力することができます。たとえば、文字の二重引用符（"）を出力するには `\"`（または `¥"`）と表わし、`\`（または `¥`）文字を出力する場合は `\\`（または `¥¥`）と表わします。
- 整数フォーマット指定子用のオプションのサイズパラメータは、フィールドの幅をバイト数で表わすものです。有効な値は、1、2、4、8 です。

フォーマットエレメント	説明
`%offset@`	データポインタを *offset* バイト分だけ移動します。構造体の中の任意のフィールドの値を出力する際に使用します。
`%[size]d`	現在のアイテムを符号付整数として解釈し、符号付き 10 進数で出力します。
`%[size]u`	現在のアイテムを符号なし整数として解釈し、符号なし 10 進数で出力します。
`%[size]x`	現在のアイテムを符号なし整数として解釈し、符号なし 16 進数で出力します。
`%[size]b`	現在のアイテムを符号なし整数として解釈し、符号なし 2 進数で出力します。
`%enum[:size]E`	現在のアイテムを、*enum* という ID を持つ列挙型クラス用のインデックスとして解釈し、列挙型クラス内のテキストのうち、そのインデックスの値に対応するものを出力します。 列挙型クラスは、`ENUM` 命令で定義されている必要があります。ただし ERCOS^{EK} の一覧のエラーを表わす列挙型クラス 99 に限り、暗黙的に定義されています。

フォーマットエレメント	説明
%F	現在の値を IEEE の倍精度の浮動小数点として解釈し、double 型（必要に応じて指数形式）として出力します。
%?	HEX ダンプ形式で出力します。
%%	% という文字を出力します。

8.2 フォーマットの例

出力内容	フォーマット文字列	表記例	注意事項
1 つの整数値を 10 進数と 16 進数で出力	"%d 0x%x"	10 0xA	%x というフォーマット指定子のみでは "0x" という文字は出力されないので、直接それらの文字を文字列として表記する必要があります。
1 つの符号なしのバイト値を % という文字とともに出力	"%1u%%"	73%	1 バイトのサイズ指定子を使用し、% という文字は %% と表記します。

32 ビットプロセッサで以下の内容を出力
struct {
int x;
int y;
};
	"(%d, %4@%d)"	(20, -15)	%offset@ を使用して構造体内部のバイトオフセットを指定します。

enum 型の e_Rainbow（3.3.1 項を参照してください）内の値を出力	"%1E"	Yellow	1 という数字は、ENUM 命令で定義された列挙型クラスの ID を示し、フィールドの幅を示すものではありません。

9 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

U