RTA-OSEK

ユーザーズガイド

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。

本書に記述されているテクノロジーの一部は、以下の特許出願中です。

UK - 0209479 および USA - 10/146,654

UK - 0209800 および USA - 10/146,239

UK - 0219936 および USA - 10/242,482

目次

1 本書について

本書は RTA-OSEK の基本的なシステムコンセプトを紹介し、さらにそれを実際に使用する方法について説明するものです。

RTA-OSEK コンポーネントの技術的な詳細情報は、『RTA-OSEK リファレンスガイド』に記述されています。また、ご使用の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 GUI を用いてシステムアーキテクチャをモデリングする必要のあるシステム設計者の方、または RTA-OSEK コンポーネントをご自分のアプリケーションプログラムと統合する必要のある C プログラマの方を対象としています。

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。

本書では以下の用語が使用されています。

RTA-OSEK: PC 上で稼動するツール、ターゲットプロセッサ用ソフトウェアコンポーネント、およびドキュメントを含む、リアルタイムオペレーティングシステムの製品名です。

オフラインツール： オフラインツールには、ホスト PC 上で稼動する各種ツール（コンフィギュレーションツール、分析ツール、ビルドツール）、およびこれらのツールを統合する RTA-OSEK グラフィカルユーザーインターフェース（GUI）が含まれます。

RTA-OSEK GUI: 各オフラインツールを統合する RTA-OSEK グラフィカルユーザーインターフェース（GUI）を指します。

RTA-OSEK コンポーネント： ターゲットプロセッサ上で稼動する RTA-OSEK リアルタイムオペレーションシステムのカーネルを指します。本書内で「カーネル（'kernel'）」と記述されている部分は、RTA-OSEK コンポーネントを指します。

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。

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、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。

1.2.1 スクリーンショットについて

RTA-OSEK は、よりよい製品を目指して常に改良が加えられているため、本書内に用いられているスクリーンショットの内容は、実際にお使いの GUI ツールのものと多少異なる場合があります。また、お使いの Windows の設定によっても GUI の表示スタイルが変わる場合があります。

2 はじめに

RTA-OSEK のコア部分は、以下の 2 つの主要エレメントで構成されています。

- **RTA-OSEK オフラインツール（RTA-OSEK GUI）**
  RTA-OSEK オフラインツールには、コード生成ツールと、システムがタイミング要件を満たしているかを確認するための分析ツールが含まれます。これらのオフラインツールは GUI（グラフィカルユーザーインターフェース）により操作され、OIL（**O**SEK **I**mplementation **L**anguage）による OS コンフィギュレーション設定を行います。RTA-OSEK オフラインツールの詳細については 2.2 項、OSEK については 2.4 項をお読みください。
- **RTA-OSEK コンポーネント** - OSEK カーネル
  RTA-OSEK コンポーネントは効率的かつ高速で、さらにその挙動が予測可能なリアルタイムオペレーティングシステム（RTOS）です。OSEK/VDX OS 規格のバージョン 2.2.x に完全に準拠し、また RTA-OSEK V5.x のコンポーネントには AUTOSAR OS (SC1) V1.0 の機能も含まれています。各コンポーネントには複雑で効率的なリアルタイムシステムを構築するために必要な関数が含まれています。RTA-OSEK コンポーネントの詳細については 2.1 項を参照してください。

RTA-OSEK は信頼できるリアルタイムシステムの開発をサポートするものです。「信頼できるリアルタイムシステム」とは、システムが所定のタイミングデッドラインまでに応答できることを意味します。厳しいデッドラインに対応するには、各タスクおよび ISR（割込みサービスルーチン）についてワーストケースのレスポンスタイムを算出し、すべて毎回確実に時間どおりに実行されるようにする必要があります。

厳格な RTOS は、固定優先度のスケジューラビリティ分析[1] の条件に基づき上記の要件を満たしていなければなりません。RTA-OSEK コンポーネントはこれらの要件を満たしていて、RTA-OSEK オフラインツールにより自動的にデッドライン監視が行われます。

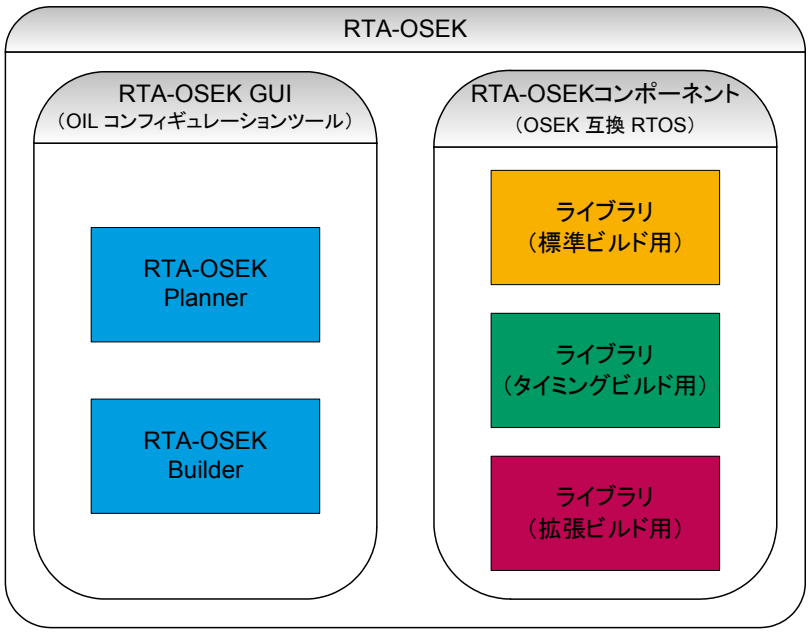


図 2-1 RTA-OSEK の構成

2.1 RTA-OSEK コンポーネント

RTA-OSEK コンポーネントのコンセプトは、リアルタイムシステムに関する長年の実績の上に築かれ、自動車業界などの量産用 OS としての厳しい要求により、さらに磨きをかけられてきました。

RTA-OSEK コンポーネントは、固定優先度に基づくプリエンプティブオペレーションシステムで、OSEK OS 規格 V2.2.x に適合しています。RTA-OSEK コンポーネントは OSEK の全コンフォーマンスクラス（BCC1、BCC2、ECC1、ECC2）をサポートし、さらに OSEK COM CCCA および CCCB に適合するプロセッサ内通信用メッセージハンドリングをサポートしています。

---

1. 参考文献：N.C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, 1995 "Fixed Priority Pre-emptive Scheduling: An Historical Perspective" *Real-Time Systems, 8,* 173-198

RTA-OSEK には、システムのユニット単価を削減するために有効な数々の**カーネル最適化**機能があります。たとえば**ライトウェイトタスクの最適化**により、タスクあたりの RAM 使用量を最大 256 バイトまで削減されます。これにより、32 タスクシステムにおいては極めて大きいリソース削減を実現できます。また**静的な API の最適化**により優先度の高い重要なタスクの起動時間が短縮されるため、プロセッサの空き時間が増加します。

RTA-OSEK コンポーネントのメモリ使用量は非常に少ないため、大規模システムの開発に最適であり、特に、ハードウェアコストが非常に厳しく制限された環境においてソフトウェアの正確な動作が厳格に求められるような場合に適しています。

RTA-OSEK は、さまざまなタイプのマイクロコントローラに対応し、メモリ使用量と CPU 負荷に関して同クラスの OS をリードしています。

RTA-OSEK コンポーネントはハードウェアに依存しません。つまりハードウェア（キャッシング、ウォッチドッグタイマ、I/O ポートなど）の制御を行わないため、アプリケーションがハードウェアを自由に使用することができ、「レガシーソフトウェア」の組み込みも柔軟に行えます。

OSEK をベースとする RTA-OSEK には、リアルタイムシステムの設計と分析のためのユニークな機能が盛り込まれています。特に、複数のタスクの起動を管理するためのメカニズムである**スケジュール**については、計画された周期的スケジュールの作成と調整が可能です。

ランタイムにおける RTA-OSEK のオーバーヘッド（タスク切り替え、割り込み処理、タスクのウェクアップなど）は**最小レベル**に押さえられていて、実行速度の変動もほとんどありません。通常、コンテキストの切り替えは**一定の実行時間**内で行われます。従来の RTOS の場合、オーバーヘッドはタスク数とシステム状態に依存するため、予測することは困難でした。

従来の RTOS にあった「無限ループ」タスク（終了要求が行われないタスク）とは異なり、RTA-OSEK 基本タスクの「シングルショット実行モデル」は、スケジューラビリティ分析に使用されるタスクモデルに適しています。

RTA-OSEK には、OSEK OS に定義されている標準（'Standard'）ビルドおよび拡張（'Extended'）ビルド以外に、**タイミング**（'Timing'）**ビルド**が追加されています。これにより、タスクのワーストケースの実行時間を測定し、**実行時間監視**（タスクの実行が所定の時間内に終了するかどうかをチェックする）を行うことが可能となります。

2.2 RTA-OSEK ツール

アプリケーションが正しく実行されるかどうかは、パフォーマンスに関する要件（たとえば、入力に対してどの程度早く応答する必要があるか、といった要件）に左右され、状況によってはそういった要件を満たすことが非常に難しい場合がありますが、現時点において、RTA-OSEK はこのような要件を保証することのできる唯一の RTOS 製品であるといえます。

コンフィギュレーション設定は **GUI（グラフィカルユーザーインターフェース）**を使用して行います。このインタフェースに**必須設定項目**が定められていて、これは、OS コンフィギュレーションで定義されたアーキテクチャによって開発中のソースコードを動作させるための「チェックリスト」の働きをします。また、この高度なインターフェースは**スタック使用量分析**の機能も備えているので、アプリケーションのワーストケースのスタック使用量を判定することができ、RAM の必要サイズを最小限に見積もることが可能となります。これらのコンフィギュレーションデータはすべて OSEK 標準の OIL ファイルに保存されます。

RTA-OSEK は、コンパクトで高速な OSEK OS であるだけでなく、タイミング挙動も保証されています。GUI に盛り込まれている **RTA-OSEK Planner** というツールには、モデリングと**スケジューラビリティ分析**の機能が盛り込まれています。スケジューラビリティ分析は、アプリケーションがすべてのデッドライン要件を満たしているかを数値的に分析する機能です。この機能により割込みに必要な最大のバッファサイズを決定することができ、このデータは、ハードウェアを選択したり、BCC2 タスクの最大同時起動数を決定したりする際に利用できます。

また RTA-OSEK には**センシティビティ分析**の機能も含まれています。これは、タスクや割り込みの実行時間を延長できるかどうかを決定する手助けとなるものです。この機能は、パフォーマンス要件を保ったままシステムを拡張する必要が生じた際に、非常に役に立ちます。

RTA-OSEK Planner を使用すると、アプリケーションを自動的に最適化することもできます。**優先度レベルの最適化**機能により、システムのプリエンプションパターンを調整してスタック使用量を削減できるかどうかが自動的に判定されます。実際の調査において、システムによって CPU が 99% 使用されているような状態においても、この手法によってプリエンプションパターンを調整でき、アプリケーションスタックの所要量を 8 分の 1 程度まで削減できることが確認されています。これは RAM サイズの大幅な削減につながり、ユニット単価の低減に大きく貢献します。

クロックスピードの最小化も RTA-OSEK の持つ分析機能のひとつです。これは、アプリケーションがデッドラインを守って実行される、という条件下においてクロックスピードをどこまで下げることができるかを分析するためものです。この機能を利用すれば、消費電力の削減や EMC 対策、さらにはより安価なチップを使用できるかどうかを判断することも可能です。

2.3 RTA-OSEK のデバッグサポート機能

RTA-OSEK は、**ORTI**（**O**SEK **R**un-**T**ime **I**nterface）規格をサポートしているため、ORTI 対応のデバッガで RTOS 変数にアクセスすることが可能です。

ORTI 対応のデバッガのリストは、各ターゲットのバインディングマニュアルに記載されています。

最新のデバッガを使用することにより、RTA-OSEK の高度な ORTI 機能を利用することができます。[1]

2.4 OSEK

OSEK は、欧州の自動車工業界における自動車制御用エレクトロニクスのオープンシステムインターフェースの策定を目的とするプロジェクトで、正式なプロジェクト名は、OSEK/VDX です。

OSEK は“Open Systems and Corresponding Interfaces for Automotive Electoronics”を意味するドイツ語のフレーズの略称です。VDX はフランス国内の規格（**V**ehicle **D**istributed e**X**ecutive）の略語で、この規格は現在では OSEK に統合されています。本書では、OSEK/VDX を「OSEK」と記します。

OSEK の最終目標は、複数のプロジェクトにおけるソフトウェアコンポーネントの移植と再利用をサポートすることにあります。これが実現すれば、各ベンダーは「自動車技術に関する知的資産」としての専門性を高め、純粋な「ソフトウェアソリューション」を開発することによってすべての OSEK 準拠の ECU で稼動するソフトウェアを作成することが可能となります。

そしてこの目標を達成するためには、アプリケーションに依存しないコンポーネントへのインターフェースを詳細に定義する必要があります。このため OSEK 規格は、ハードウェアや車両ネットワークの詳細な仕様に依存しない API（**A**pplication **P**rogramming **I**nterface）を装備しています。

詳しい情報は、http://www.osek-vdx.org を参照してください。

2.5 AUTOSAR

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。

AUTOSAR は「基本ソフトウェアモデル」（BSW：**B**asic **S**oft**w**are Modules）の仕様を定義するもので、この BSW にはオペレーティングシステムや通信ドライバ、メモリドライバ、マイクロコントローラの抽象レイヤなどが含まれます。また AUTOSAR ではコンポーネントベースの構造モデルも定義されています。このモデルは、通信アプリケーションソフトウェアコンポーネント（SWC）の抽象レイヤを定義する「仮想ファンクションバス」（VFB：**V**irtual **F**unction **B**us）を定義するものです。VFB により、SWC はハードウェアに依存しないものとなるため、異なる ECU への移植や複数の車両プロジェクトでの再利用が可能になります。VFB 抽象化レベルは、AUTOSAR RTE（**R**un-**T**ime **E**nvironment）内に隠蔽化されていて、RTE により SWC と BSW が接続されます。

詳しい情報は、http://www.autosar.org を参照してください。

2.6 RTA-OSEK 5.0 の新機能

2.6.1 ファイルのインポート

RTA-OSEK V5.0 は、実績ある RTA-OSEK の前バージョンに、以下の新機能を追加したものです。

- AUTOSAR OS V1.0 のスケーラビリティクラス 1 に準拠する以下の機能
  - スケジュールテーブル
  - 基本タスクおよび拡張タスクのランライムスタック監視

1. リストに記載されていない ORTI デバッガについては、ETAS のエンジニアリングサービスで対応可能です。詳しくはサポート窓口までお問い合わせください。

－ OSEK カウンタ用標準 API

- ライセンス管理機能の強化（複数のライセンスファイルを参照可能）
- OS 外のライブラリを統合するための新しいパッケージメカニズム
- RTA-OSEK Builder でのシステムビルド時に使用するカスタムマクロを定義するための、新しいマクロメカニズム
- RTA-OSEK のタスクセットメカニズムを利用して起動されるタスクについての、起動回数の超過チェック（E_OS_LIMIT）
- ERCOS^{EK}（ETAS の従来のオペレーティングシステム）からの移行を容易にするための、「協調スケジューリング」と「プロセス」のサポート
- タスク単位のタスクトレースを設定可能にするための、RTA-TRACE の機能強化

2.6.2 前バージョンとの互換性

RTA-OSEK V5.0 のツールでは、前バージョンの RTA-OSEK カーネルを扱うことができます。
以下の表は、RTA-OSEK V5.0 の各機能と、旧バージョンのカーネルの互換性を示したものです。

RTA-OSEK V5.x ツールの機能	**RTA-OSEK コンポーネント**	
	V5.x（RTA-OSEK V5.x）	**V3.x（RTA-OSEK V3.x/4.x）**
AUTOSAR スケジュールテーブル	○	○
アドバンスドカウンタインターフェース	○	○
タスクセットによるタスク起動についての起動回数の超過チェック	○	×
スタック監視	○	×
協調スケジューリング	○	○
RTA-TRACE によるタスクごとのトレース	○	×

RTA-OSEK は、使用されているカーネルのバージョンを自動的に認識するので、RTA-OSEK の GUI 上にはカーネルバージョンに応じた設定オプションが表示されます。

3 開発工程

本章では、RTA GUI を使用してアプリケーションを作成する際の工程を紹介します。初めて RTA-OSEK アプリケーションを作成する際には、ここに説明されているコンセプトに基づいて作業を進めてください。

本章では、ここまでの部分にまだ記述されていない内容も記載されていますが、そのような場合は、本書全体の中からその事項に関する項目を探して参照してください。

RTA-OSEK の GUI は 3 つのビューに分かれています。各ビューは、GUI の左下部分のタブで切り替えます（図 3-1 を参照してください）。

- *Planner*：3.2 項から 3.4 項にかけて説明されています。旧バージョンの RTA-OSEK をお使いの方はすでに使い慣れているタブです。このタブにおいてはアプリケーションの設計の検証が行われるため、通常はここでアプリケーションを記述することをお勧めします。
- *Builder*：OSEK のコンセプトをよく理解していて、*Planner* によるデザイン分析／検証を行わずにシステムを構築しようとするユーザー用のビューです。
- *RTA-Trace*：ETAS の製品、RTA-TRACE（組み込みシステム用ソフトウェアロジックアナライザ）に関連する RTA-OSEK パラメータのコンフィギュレーションを行うためのタブです。

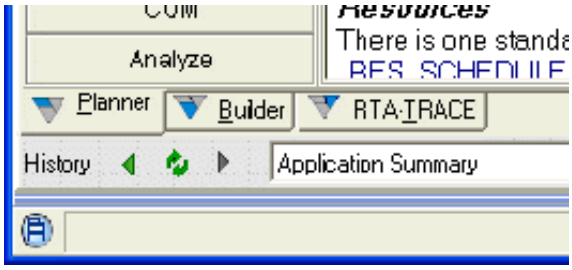


図 3-1 ビュー切り替え用タブ

3.1 概要

RTA-OSEK *Planner* で新しいアプリケーションを作成する工程は、いくつかのステップに分かれています。図 3-2 は、開発ライフサイクルにおける各ステップの流れを示しています。

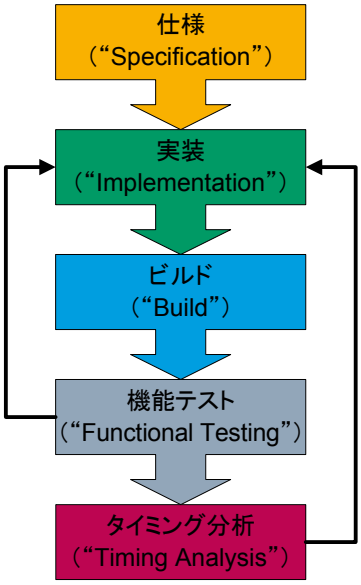


図 3-2 開発工程

図 3-2 に示されるように、新しいアプリケーションを作成する際には、まず**仕様**を決定する必要があります。その後アプリケーションを構築し、仕様に基づいた**実装**（つまりコーディング）を行います。コーディングが完了したらアプリケーションを**ビルド**し、続いて**機能テスト**と**タイミング分析**を行います。

機能テストと分析の段階で実装内容が変更される可能性があるので、実装が完了するまでアプリケーションのビルドとテストを繰り返し行う必要があります。

これらの各ステップについて、以降の項で詳しく説明します。

3.1.1 仕様

新しいターゲットアプリケーションを開発するには、まずその仕様を明らかにする必要があります。具体的な仕様の例は、3.2.1 項と 3.3.1 項に説明されています。

一般に、仕様には以下のような事柄が含まれています。

- ターゲットプラットフォームの詳細
  プロセッサタイプ、クロック速度、使用可能メモリ容量などが含まれます。
- 外部からシステムへの入力のリスト
  外部からの入力は**スティミュラス**と呼ばれ、具体的には、スイッチが閉じる、タイマのカウントダウンが終了する、ネットワークメッセージが受信される、エンジンが所定の回転角になる、といった事象を指します。また、時間を入力として考えることもできます。たとえば、ハードウェアをポーリングする必要がある場合、10ms ごとに発生するスティミュラスを作成する必要があります。
- システムからの出力のリスト
  出力は**レスポンス**と呼ばれます。レスポンスは、システムがスティミュラスに対してどのように応答するかを表すもので、具体的には、ライトを点灯する、内部カウントを更新する、モータを起動する、メッセージを別のコントローラに送る、といったアクションが考えられます。
- パフォーマンス要件
  各スティミュラスについて、通常、システムは必ず 1 つ以上のレスポンスを返します。このレスポンスは制限時間内に返されなければなりません。**デッドライン**は、スティミュラス後にレスポンスが発生するまでの最大許容時間です。システムデッドラインの仕様は、タイミング分析の際に重要となります。タイミング分析において、システム負荷が最大となった際にシステム要件が満たされるかどうかがチェックされます。

実際の例を見てみましょう。図 3-3 は、衝撃試験において車両が物体に衝突した状態を示しています。車が物体に衝突するとエアバッグが膨らみます。この場合、衝撃がスティミュラスであり、エアバッグが膨らむことが、衝撃に対するレスポンスです。このレスポンスは、デッドライン前に発生しなければなりません。そしてこのデッドラインは、乗員の負傷を最小限に留めるような値が設定されている必要があります。

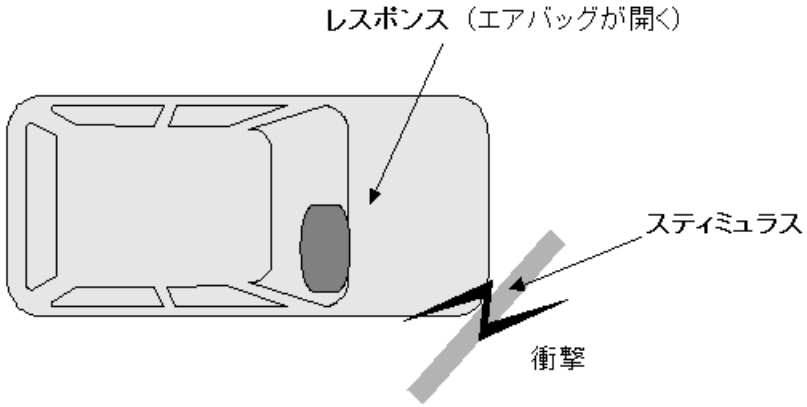


図 3-3 スティミュラスに対するレスポンス

3.1.2 実装

仕様が決定した後には、その仕様の実装について考えます。「実装」とは、仕様として定義された内容を実際のコードとして実現することを指します。この実装ステップでは、指定されたスティミュラスをターゲットがどのように検知し、さらにそれに対するレスポンスをどのように実装するかを明確にしていきます。

多くの場合、外部スティミュラスはハードウェア割込みによって検知されます。割込みに対してターゲットプロセッサが応答し、割込みサービスルーチン（ISR）が実行されます。

通常、ISR はレスポンスが実装されたタスク、つまりレスポンスを実現する処理を実行するタスクを起動しますが、必要に応じて ISR に直接レスポンスを実装することもできます。負荷の大きいシステムでは特に、短い ISR と、優先度が適切に設定されたタスクとを組み合わせることにより、最良の応答速度を実現できます。

プロセッサベクタテーブルに各割込みソース用のエントリポイントが設けられているターゲットの場合、発生する可能性のある割込みごとに ISR が必要になります。そうでないターゲットでは、複数の割込みソースが同じベクタを使用するので、ISR においてどの割込みソースがアクティブであるかを解読する必要があります。

単純なシステムでは、各レスポンスを複数のタスクに分けて実装することができます。デッドラインが最短であるレスポンス、つまり最も早く発行しなければならないレスポンスを最高の優先度のタスクに割り当ててください。また、「スケジューラビリティ分析」を行えば、そのタスクが常にデッドライン内に実行されているかどうかを確認することができます。

1 つのタスクに複数のレスポンスを実装することもできます。たとえば、図 3-4 に示すように、スティミュラス `S` について、時間 `T1` 内にレスポンス `R1`、時間 `T2` 内にレスポンス `R2`、さらに時間 `T3` 内にレスポンス `R3` を行うようにすることができます。

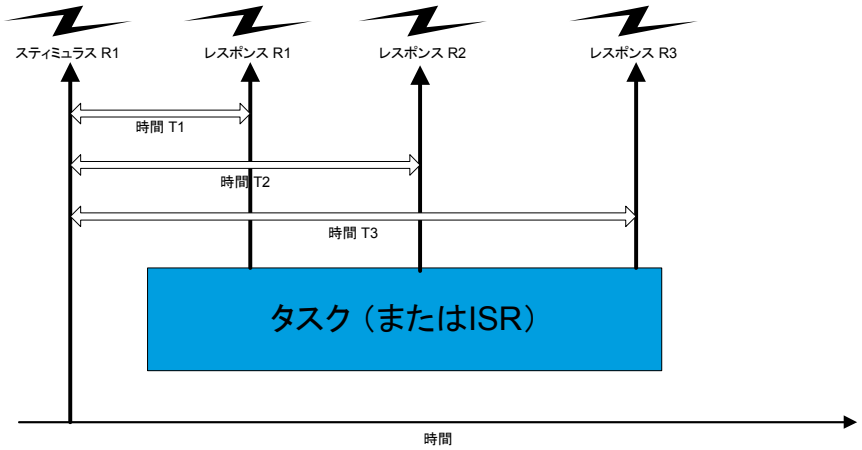


図 3-4 1 つのタスクで複数のレスポンスを行う

通常は、1 つのタスクに図 3-4 のすべてのレスポンスを実装することができ、RTA-OSEK は、そのタスクがそれぞれのデッドラインに対応できるかどうかをチェックすることができます。

タスクが現在どのスティミュラスに対して応答しているかをタスク自身で認識できれば、1 つのタスクから複数のスティミュラスに対するレスポンスを発生させることも可能です。タスクにどのスティミュラスが発生したかを知らせるには、以下のような方法があります。

- グローバル変数
- OSEK COM メッセージ
- OSEK OS イベント

アプリケーションの構造、つまりタスクと ISR の構造が確立できたら、リソース、キューイングメカニズム、イベント、メッセージなどの OS 機能を用いてアプリケーション構造をさらに精密に決定していきます。

RTA-OSEK のビルダが、実装すべきタスクと ISR のスケルトンソースコードを提供し、実際に実行されるコードはユーザーが任意に作成します。

RTA-OSEK *Planner* にはアプリケーション用の実装チェックリストが表示され、実装すべきコードが示されます。これに基づいてコーディングを行い、その後、すべてのファイルをコンパイル・リンクし、アプリケーションの実行ファイルを生成します。サンプルアプリケーションを実装する方法は、3.3.2 項を参照してください。

さまざまなコード群（各タスクと ISR のコード）は、それぞれ個別の C ソースファイルに書き込みますが、この際、以下のような点について注意が必要です。

- アプリケーションソースコードにおいて、各タスクスレッドは互いにアイソレートされるため、いくつかのバグを防ぐことができます。たとえば、ファイル内で変数を静的変数として宣言しておけば、その変数は、他のタスクや ISR によって不正に書き換えられてしまうことはありません。

- コンフィギュレーション管理でタスクに関する変更を行う際は、1 つのファイル内の 1 つのタスクに対してのみ行えるので、操作がシンプルです。
- テストの際、1 つのタスクについてのみテストを行うことができます。必要に応じて個々のタスクをダミーに置き換え、自動テストマネージメントツールに対応することも可能です。

RTA-OSEK を使用する際は、個々のタスクや ISR のコードは、それぞれ専用のファイルに書き込むようにしてください。これは、RTA-OSEK が各タスクおよび ISR 用のヘッダファイルを個別に自動生成するためです。

これらのヘッダファイルには、最適化バージョンの RTA-OSEK API（「静的インターフェース」と呼ばれます）へのアクセスが格納されます。ビルド時において RTA-OSEK は、RTA-OSEK が認識するアプリケーションに関する情報に基づき、各タスクと ISR に対して、最も適した API コールを自動実装します。静的インターフェースの場合、リンク時において、RTA-OSEK API 使用時に発生する可能性のあるランタイムエラーがチェックされるので、ランタイムデバッグによって API の使用に関する妥当性のチェックを行う手間が省けます。

静的インターフェースを使用しない場合は、RTA-OSEK コンポーネントを使用するすべてのソースファイルにおいて、タスクまたは ISR に固有ではない、共通のヘッダファイル `osek.h`（またはライブラリに含めるコードを作成している場合は `oseklib.h`）を `#include` 文でインクルードする必要があります。この場合もユーザーアプリケーションの機能は変わりませんが、処理速度は遅くなり、コードサイズも増大し、ランタイムデバッグに要する手間も大きくなります。

複数のタスクまたは ISR を 1 つのソースファイル内に書き込む必要がある場合でも、RTA-OSEK のヘビーウェイトターミネーションを用いるタスクとライトウェイトターミネーションを用いるタスクを同じファイル内に含めることはできません。`#define OS_HEAVYWEIGHT` または `#define OS_LIGHTWEIGHT` を適宜に使用してターミネーションタイプを定義し、`#include "osek.h"` で `osek.h` ファイルのみをインクルードしてください。

3.1.3 ビルド

アプリケーションの基本構造が完成した後、RTA-OSEK GUI でユーザーのソースコードファイルとリンクする必要のあるアセンブラファイル、C ソースファイル、ヘッダファイルをビルドし、それらを使用して実行形式のアプリケーションを作成します。アプリケーションのビルドの詳細については 3.6.2 項で説明します。

3.1.4 機能テスト

機能テストを行うには、アプリケーションをターゲットハードウェアにダウンロードする必要があります。アプリケーションを初めてテストする際には、一度に 1 つのスティミュラスだけをトリガすることをお勧めします。

テスト中に問題が発見された場合は、アプリケーションを修正し、ビルドとテストを再実行してください。アプリケーションの挙動が満足できる状態になるまでこれらのステップを繰り返します。機能テストの段階については、3.3.4 項で説明します。

3.1.5 タイミング分析

テストの最終段階において、アプリケーションがタイミングデッドラインに対応しているかを証明する必要があります。そのためには、コードの実行時間を測定してその情報を RTA-OSEK に入力する必要があります。

実行時間の測定は複雑なので、正確なタイミング分析を行うにはこの情報が必須となります。

RTA-OSEK は、システムが**スケジューラブル**であるかどうか判断します。アプリケーションがすべてのデッドラインに対応できている場合、そのアプリケーションは「スケジューラブル」であるとされます。システムがスケジューラブルではない場合、RTA-OSEK は、どのデッドラインが守られていないかを通知します。

システムをスケジューラブルなものにするには、以下のような方法があります。

- それぞれのタスクまたは ISR について、許容される最長の実行時間を与える
- タスクの優先度を調整する
- CPU クロックを調整する

分析の詳細については 3.3.5 項で説明します。

3.2 簡単なアプリケーションの例

ここでは実際に RTA-OSEK GUI を用いて OSEK OS オブジェクトを設定し、簡単なサンプルアプリケーションを作成してみます。

ここで作成するシステムの仕様は以下のとおりです。

- ターゲットプロセッサは Motorola HC12 です。（このターゲットがインストールされていない場合は、他のターゲットを選択してください。）
- ターゲットクロックは 8MHz です。
- 入力される CAN バスメッセージによって CPU に割込みがかかります。ISR は、各メッセージに関する最初の処理を行い、さらにその後の処理を行うメッセージ処理タスクを起動します。
- 3 つのタスクが、それぞれ 3ms、6ms、14ms の周期で実行される必要があります。
- 14ms 周期のタスクは、CAN メッセージ処理タスクとデータバッファを共有します。データの破損を防ぐため、バッファへのアクセスは相互排他的に行う必要があります。

3.2.1 RTA-OSEK GUI を使用して新しいアプリケーションを作成する

新しいアプリケーションを作成するには、RTA-OSEK GUI で、メニューコマンド **File → New** を選択します。**Select Target** ダイアログボックスが開くので、**Available Targets** と **Variant** のドロップダウンリストからターゲットプロセッサを選択してください。

図 3-5 では、*HC12/COSMIC 16 task* というターゲットが選択され、*HC12* というバリアントが選択されています。HC12 がインストールされていない場合は、別のターゲットを選択してください。

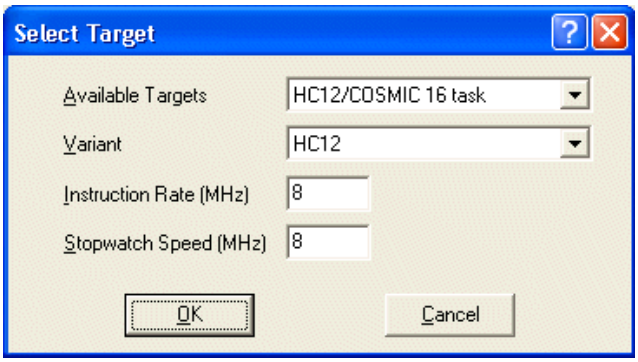


図 3-5 ターゲットプロセッサの選択

> **重要**
> Available Targets リストには、ユーザーのコンピュータにインストールされているターゲットだけが、ユーザーのライセンスファイルに基づいて表示されます。期待するターゲットが表示されない場合は、ETAS までお問い合わせください。各ターゲットには、共通のプロセッサコアをベースとする様々なチップバージョンに対応する、多数のバリアントがあります。

図 3-5 に示された **Select Target** ダイアログボックスでは、インストラクションレート（**Instruction Rate**）とストップウォッチ速度（**Stopwatch Speed**）の設定も行えます。

インストラクションレートには、CPU のクロックレートを設定してください。この情報に基づき、RTA-OSEK Planner が実時間（ミリ秒、秒など）を CPU サイクルに変換します。

ストップウォッチ速度は、タイミングビルドおよび拡張ビルドにおいて実行時間を測定するために用いられる API 関数 `GetStopwatch()` で使用されるタイマハードウェアの速度を指定するものです。またストップウォッチ速度は、RTA-OSEK Builder が OSEK の `OSTICKDURATION` のデフォルト値を生成する際（SystemTimer が明示的に定義されていない場合）にも使用されます。この値の単位はナノ秒単位なので、図 3-5 のように設定した場合、`OSTICKDURATION` は 125ns となります。

一般的には、ストップウォッチがインストラクションレートと同じ速度で動作することが理想的ですが、ターゲットによっては不可能な場合があります。タイマハードウェアにプリスケーラが内蔵されている場合などがそれに該当します。

ターゲット情報の設定後、**OK** ボタンをクリックすると、RTA-OSEK GUI は新しいアプリケーションについての情報を、図 3-6 のように表示します。この情報はいつでも参照できるので、作成中のシステム全体の概要を容易に把握することができます。

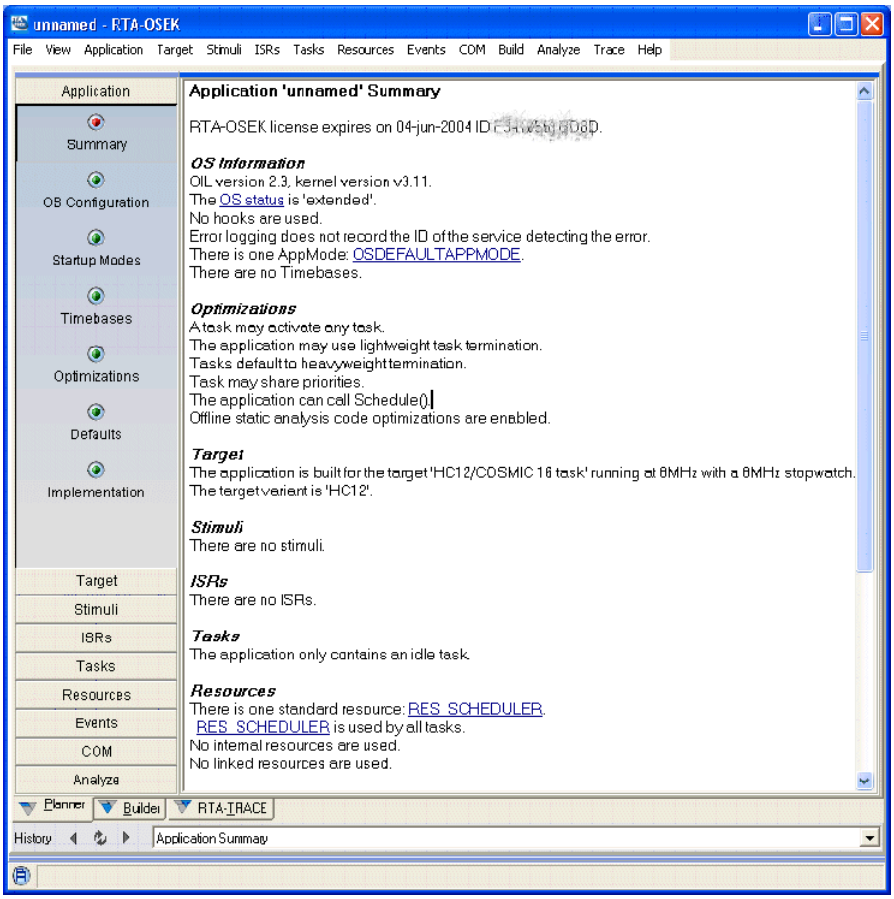


図 3-6 新しいアプリケーションについての情報の表示

3.2.2 アプリケーションを保存する

新しいアプリケーションを作成した後は、速やかに保存するようにしてください。アプリケーションを初めて保存するときは、**File → Save As...** を選択します。

Save As ダイアログボックスでは、**Save In** リストで、ファイルを保存したいロケーションまでナビゲートします。ここでは、**File Name** に ***UserApp*** という名前を入力し、**Save** ボタンをクリックして新しいアプリケーションを保存してください。

アプリケーションの保存は、**File → Save** を選択するか、キーボードの **<Ctrl>** キーと **<S>** キーを同時に押す（**<Ctrl> + <S>**）ことにより、随時行うことができます。

3.2.3 OIL ファイルを表示する

作成されたファイルは OIL のを用いて保存されるので、他の OSEK 互換ツールからもこのファイルを読むことができます。現在のアプリケーション用の OIL ファイルの内容を見るには、メニューから **View → OIL File** を選択してください。

図 3-7 のように、OIL ファイルの内容がワークスペースの下半分に表示されます。ウィンドウ上部のワークスペースに詳細情報が表示され、ウィンドウ下部に OIL ファイルが表示されています。

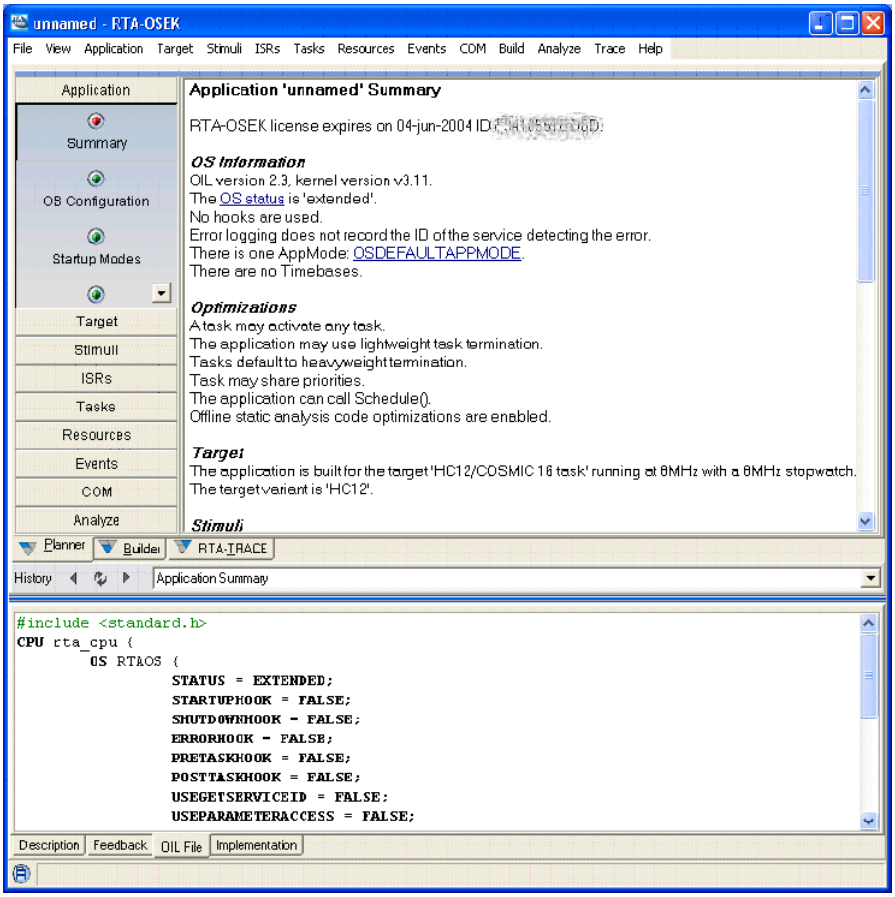


図 3-7 OIL ファイルを表示する

View → OIL File を再度選択してこのメニューコマンドに付いているチェックマークを消すと、OIL ビューが閉じます。

3.2.4 実装

次は実装段階に進みます。ここでは、アプリケーションに使用される RTA-OSEK の OS オブジェクト（タスク、ISR、カウンタ、アラームなど）の設定方法を説明します。

以下のステップにおいて行う操作のうちのいくつかは、OS オブジェクトのインスタンスを対象としていますが、これらの操作は、図 3-8 に示される共通アイコンセットを使用して行います。各アイコンの機能は、左から順に、**Add**、**Rename**、**Delete** です。

図 3-8 共通アイコン（**Add**、**Rename**、**Delete**）

ISR を作成する

ここでは、例として、2 つの割込みソースを使用します。1 つは CAN メッセージの到着を検知するためのもので、もう 1 つは 1ms ごとに割込みを発生させるハードウェアタイマにアタッチします。

新しい ISR を作成するには、ナビゲーションバーから **ISRs** グループを選択します。

ISRs グループを選択すると、図 3-9 のように、ISR についての情報がワークスペースに表示されます。

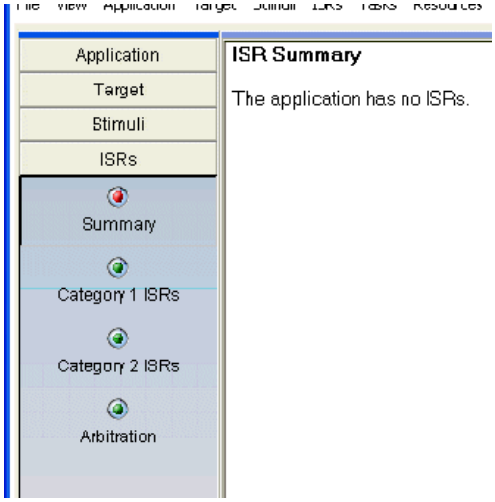


図 3-9 割込み

ISR は、「カテゴリ 2 の ISR」として追加されます。これは、ISR から OS コールが行われるためです（「カテゴリ 1 の ISR」が OS コールを行うことは禁止されています）。

最初に作成される ISR は、ハードウェアタイマによって生成される 1ms のチック（'tick'）を扱います。

- ナビゲーションバーから、**Category 2 ISRs** サブグループを選択します。
- ワークスペースの **Add** ボタンをクリックして、ISR を作成します。
- **Add Cat 2 ISR** ダイアログボックスに ***TimerISR*** という名前を入力し、**OK** をクリックします。
- ターゲットタイプによっては、**Vector**（割込みベクタ）と **Priority**（優先度）を入力する必要があります。この例を図 3-10 に示します。

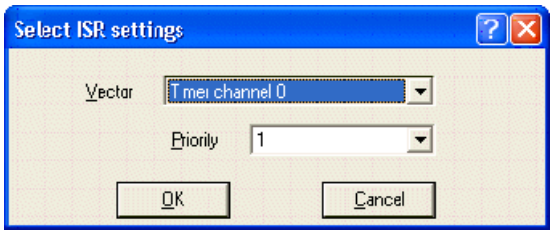


図 3-10 ベクタと優先度を選択する

図 3-11 のように、新しく作成された ISR についてのデフォルト設定がワークスペースに表示されます。

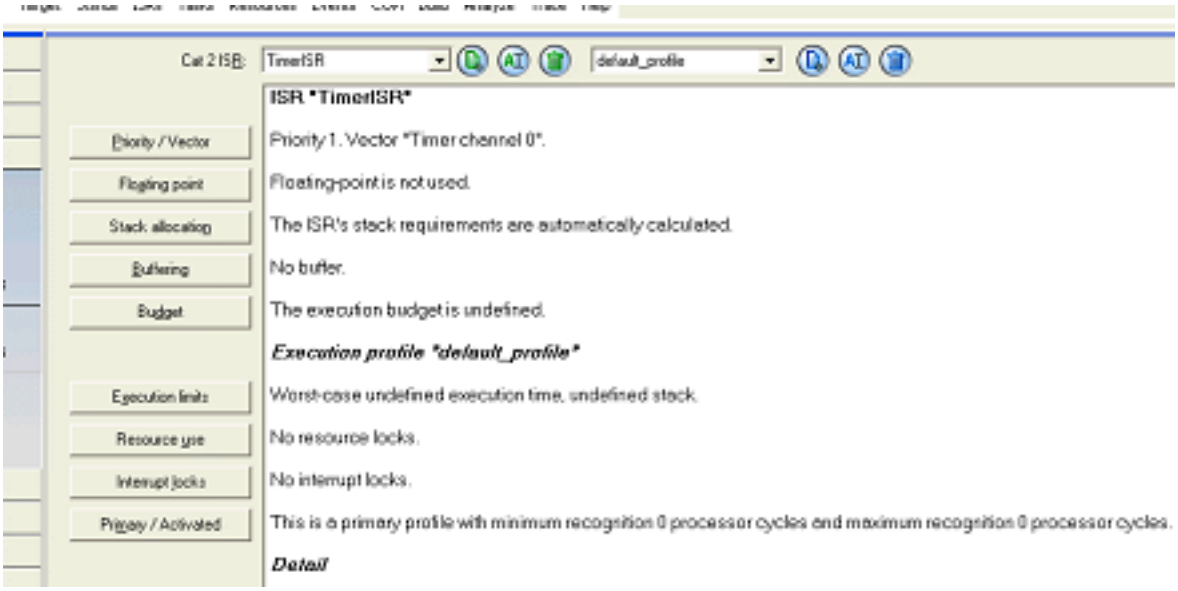


図 3-11 新しく作成された割込みについての情報

さらに上記の操作を繰り返して、*CanISR* というカテゴリ 2 の ISR を作成してください。この ISR は、CAN メッセージの受信割込みを扱います。

タスクを作成する

次に、アプリケーションの処理を行う 4 つのタスクを作成します。このうち、3 つのタスクはそれぞれ 3ms、6ms、14ms の周期で起動され、もう 1 つのタスクは *CanISR* によって起動されます。

新しいタスクを作成するには、ナビゲーションバーから **Tasks** グループを選択します。

Tasks グループを選択すると、図 3-12 のようにタスクについての情報がワークスペースに表示されます。

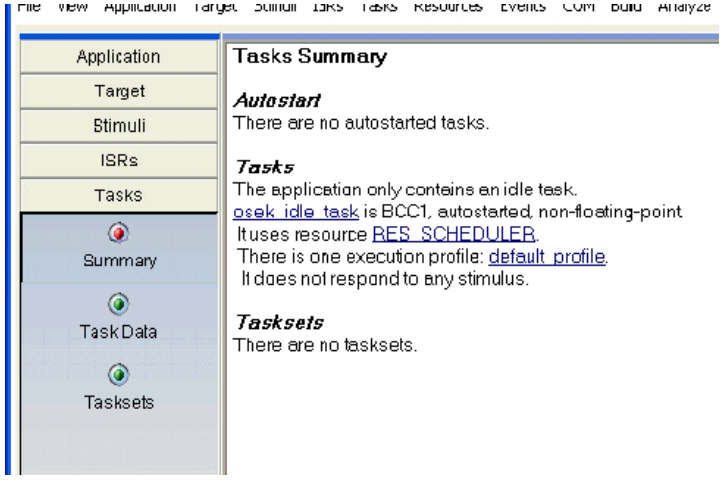


図 3-12 タスクについての情報（タスクがまだ作成されていない状態）

- ナビゲーションバーから、**Task Data** サブグループを選択します。
- ワークスペースの **Add** ボタンをクリックしてタスクを作成します。
- **Add Task** ダイアログボックスに ***Task1*** という名前を入力し、**OK** をクリックします。
- **Task "Task1" priority** ダイアログボックス（図 3-13 参照）に、優先度として **10** を入力します。

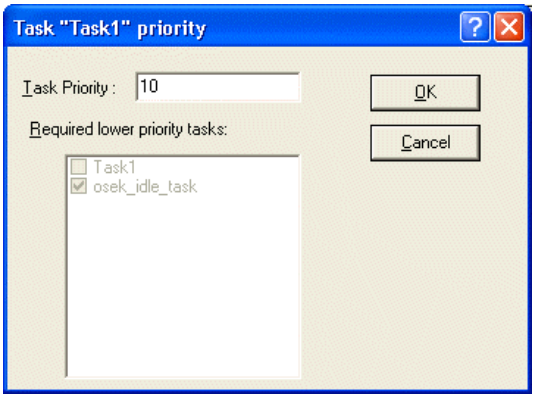


図 3-13 タスク優先度の設定

図 3-14 のように、新しく作成されたタスクについてのデフォルト設定がワークスペースに表示されます。

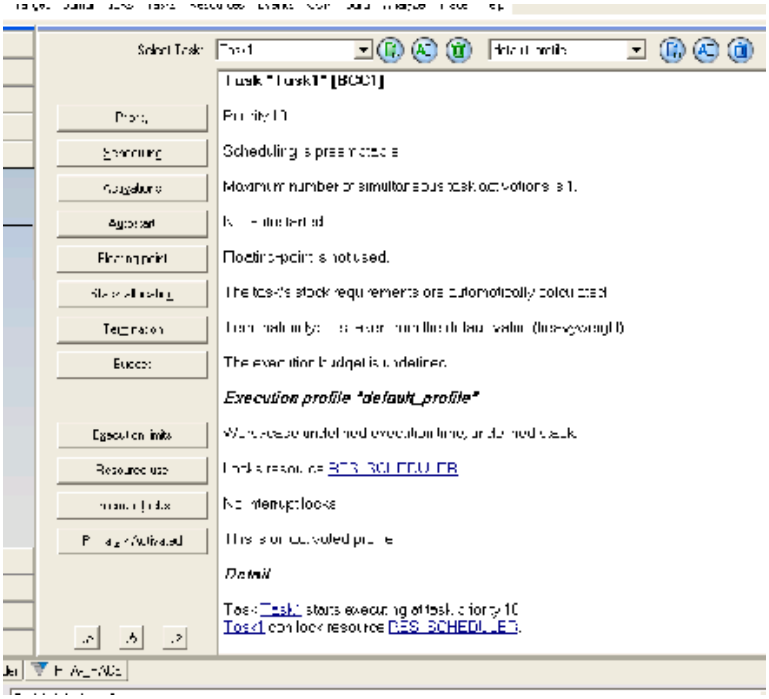


図 3-14 タスクについての情報

さらに上記の操作を繰り返して、以下の 3 つのタスクを作成してください。

- タスク名：*Task2*、優先度：9
- タスク名：*Task3*、優先度：8
- タスク名：*CanWorker*、優先度：3

カウンタとアラームを作成する

前のステップで作成した 3 つのタスクは、カウンタにアタッチされたアラームが満了（'expire'）することによって起動されます。この例においては、1ms ごとに呼び出される *TimerISR* でカウンタがカウントされるようにするので、カウンタのチック（'tick' - 刻み）は 1ms となります。カウンタにアタッチされたアラームは、所定のチックが経過することにより満了し、アラームの満了によってそのアラームに割り当てられたタスクが OS によって起動されます。

新しいカウンタを作成するには、ナビゲーションバーから **Stimuli** グループを選択します。

Stimuli グループを選択すると、図 3-12 のようにスティミュラスについての情報がワークスペースに表示されます。

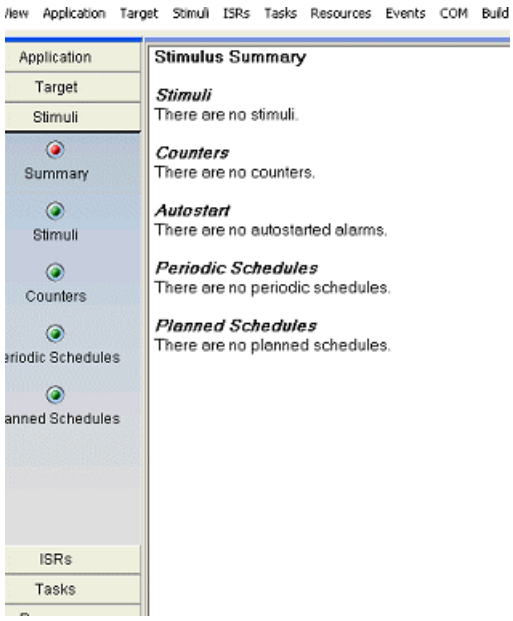


図 3-15 スティミュラスについての情報（アラームがまだ作成されていない状態）

- ナビゲーションバーから、**Counters** サブグループを選択します。
- ワークスペースの **Add** ボタンをクリックしてカウンタを作成します。
- **Add Counter** ダイアログボックスに ***TimerCounter*** という名前を入力し、**OK** をクリックします。
- **Tick rate for timebase "TimerCounter"** ダイアログボックス（図 3-16 参照）に、最速のチックレートとして 1ms（real time）を入力します。

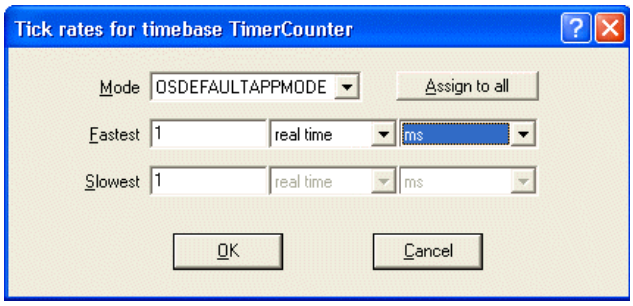


図 3-16 タイマのチックレートの設定

図 3-17 のように、新しく作成されたカウンタについてのデフォルト設定がワークスペースに表示されます。

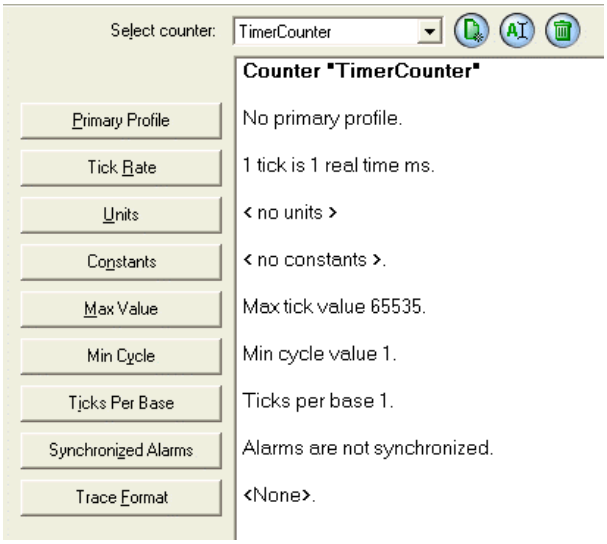


図 3-17 カウンタについての情報

続いて、カウンタが *TimerISR* によってカウントされるように設定します。

- **Primary Profile** ボタンをクリックし、プライマリプロファイルのドロップダウンリストから ***TimerISR*** を選択して **OK** をクリックします。

次に、3 つのタスクの起動を行うアラームを作成します。*Task1* は 3ms ごと、*Task2* は 6ms ごと、*Task3* は 14ms ごとに起動されるようにしてください。

- ナビゲーションバーから、**Stimuli** サブグループを選択します。
- ワークスペースの **Add** ボタンをクリックしてカウンタを作成します。
- **Add Stimulus** ダイアログボックスに ***Task1Alarm*** という名前を入力し、**OK** をクリックします。
- **Arrival Type** ボタンをクリックして、**periodic** を選択します。
- **Schedule/Counter** ボタンをクリックし、このアラーム用のカウンタとして ***TimerCounter*** を選択します。
- **Arrival Pattern** ダイアログボックスに、周期として 3 ***TimerCounter*** チック（3ms に相当します）を入力します。

図 3-18 のように、新しく作成されたアラームについてのデフォルト設定がワークスペースに表示されます。

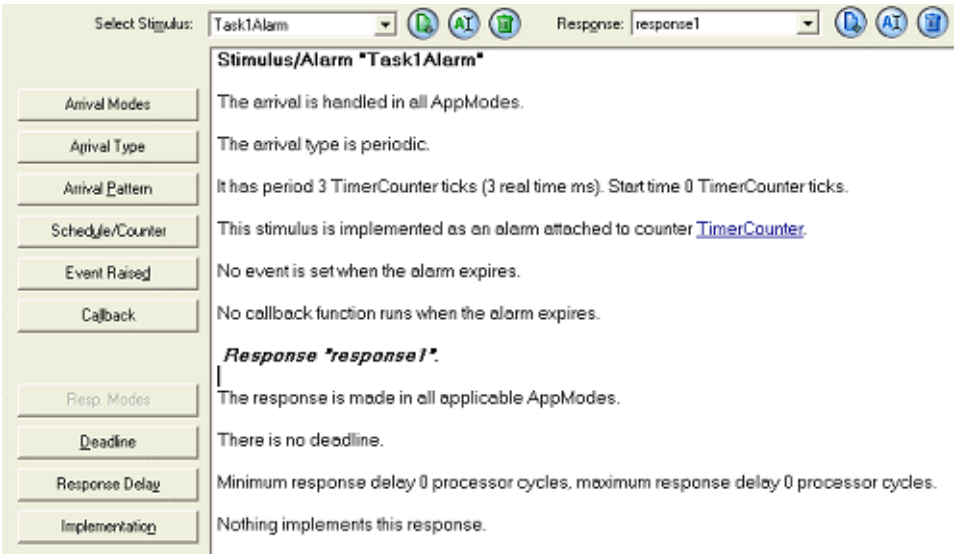


図 3-18 アラームについての情報

ここでさらに、アラームが *Task1* を起動するように設定します。この起動は、**スティミュラス** であるアラームに対する**レスポンス**として行われます。

- **Implementation** ボタンをクリックし、**Implementation** ドロップダウンリストから ***Task1*** を選択します。この時点においては、**Execution time** フィールドには何も入力する必要はありません。

さらに以下の 2 つのアラームを、同様の操作を行って作成してください。

- 6ms ごとに *Task2* を起動する、*Task2Alarm* という名前のアラーム
- 14ms ごとに *Task3* を起動する、*Task3Alarm* という名前のアラーム

リソースを作成する

リソースは、アプリケーションコード内のクリティカルセクションへのアクセスを相互排他的に行うために使用されるものです。これは通常、グローバル変数に格納されたデータの破損を防ぐために利用されます。このアプリケーションにおいては、2 つのタスク（*CanWorker* および *Task3* ）の間で共有されるリソースを作成します。このリソースのロックに成功したタスクのみが、データバッファ内のデータを、もう一方のタスクの介入なしに更新できます。

新しいリソースを作成するには、ナビゲーションバーから **Resources** グループを選択します。

Resources グループを選択すると、図 3-19 のようにリソースについての情報がワークスペースに表示されます。

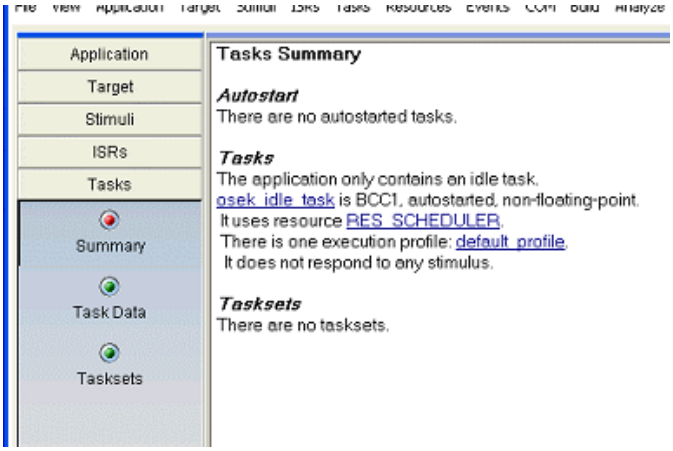


図 3-19 リソースについての情報（ユーザー定義のリソースがまだ作成されていない状態）

- ナビゲーションバーから、**Standard** サブグループを選択します。

- ワークスペースの **Add** ボタンをクリックしてリソースを作成します。
- **Add resource** ダイアログボックスに ***CanResource*** という名前を入力し、**OK** をクリックします。

リソースが作成され、そのリソースについてのデフォルト設定がワークスペースに表示されます。

次に、どのタスクがそのリソースを使用するかを定義します。

- **Change Users** ボタンをクリックし、**Select Users** ダイアログボックスで ***Task3*** と ***CanWorker*** を選択して **OK** をクリックします。

RTA-OSEK は、このリソースの効果的なタスク優先度を自動的に計算します。

ワークスペースに、図 3-20 のような情報が表示されます。

図 3-20 ユーザー定義されたリソースについての情報

タスクと ISR のコードを記述する

今度は、*TimerISR* および *CanISR* いう ISR と、*Task1*、*Task2*、*Task3*、*CanWorker* というタスク、そしてアプリケーションの `main()` 関数（アプリケーションの起動とアイドルメカニズムが含まれる関数）のコードを記述する必要があります。

これらの C ソースコードは、外部のエディタで記述することも可能ですが、その際、RTA-OSEK GUI 内で作成したテンプレートを使用することができます。これは、以下のようにして行います。

- ビルダ（*Builder* タブ）に切り替え、ナビゲーションバーから **Custom Build** グループを選択します。
- ワークスペースの **Create Templates** ボタンをクリックします。すると RTA-OSEK は 7 つの C ソースファイルを作成し、そこにスケルトンコードを挿入します。またさらに、ビルド工程で使用する `rtkbuild.bat` というバッチファイルも作成します。

TimerISR と CanISR のコードを記述する

ナビゲーションバーから **ISRs** グループを選択します。そして **Category 2 ISRs** サブグループを選択し、ワークスペースから ***TimerISR*** を選択します。

ISR 内で行うべき処理が不明な場合でも、RTA-OSEK GUI によって必要な事項が提示されるので、容易にコーディングを行えます。メニューから **View → Implementation** を選択すれば、ウィンドウの下の部分に、ISR の「実装ノート」が開き、実装に関する詳細情報が表示されます。このエリアのサイズは、青色のスプリットバーをマウスの左ボタンで上下にドラッグすることにより任意に調整できます。

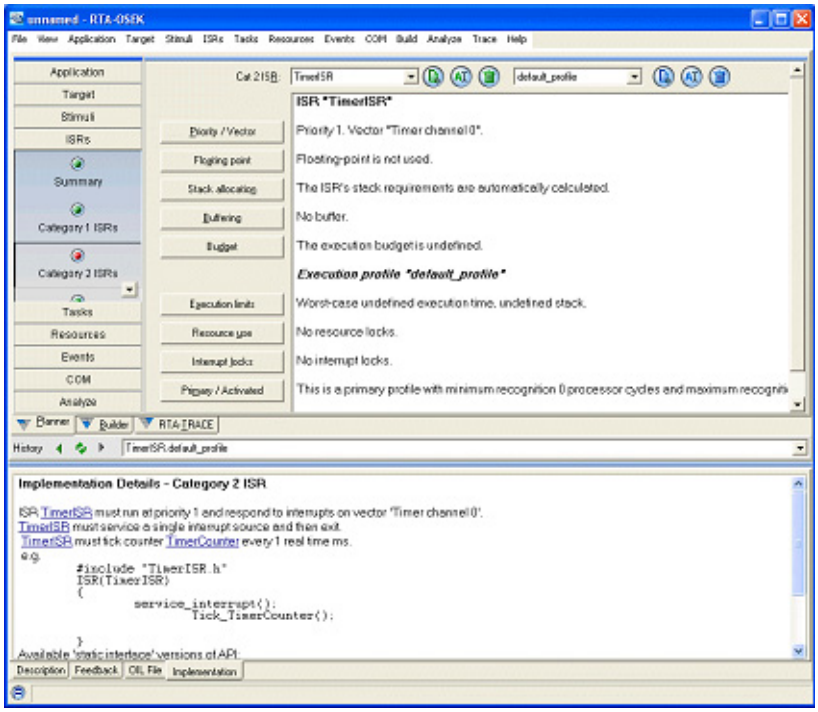


図 3-21 PrimaryISR の実装ノートを表示する

図 3-21 に示されたサンプルコードを見てみると、ISR 専用のヘッダファイル `TimerIsr.h` がインクルードされ、その後に ISR のボディが続いています。この ISR はカウンタ *TimerCounter* のプライマリプロファイルに設定されているので、実装ビューにはこの ISR が `Tick_TimerCounter()` を呼び出す必要があることが示されています。この関数を呼び出すことにより、*TimerCounter* が「チック」され、タイマに対して時間の経過が通知されます。このためには、タイマハードウェアが、定義された間隔（この例では 1ms）で「チック」を発生させるように設定されている必要があります。タイマハードウェアが正しく機能しないと、カウンタにアタッチされたアラームが正確なタイミングで満了せず、タスクの起動が正しく行われなくなってしまいます。

再度 **View → Implementation** を選択してこのメニューアイテムのチェックマークを消すと、実装ノートが閉じます。

Category 2 ISRs ワークスペースの .c ボタンをクリックすると、ISR のソースコードを直接編集することができます。

同様にして、*CanISR* のコードを編集します。ここでは、`ActivateTask_CanWorker()` を呼び出して ISR ボディに含まれる *CanWorker* タスクを起動します。

> **重要**
> RTA-OSEK GUI 内でファイルを編集する場合、Windows Notepad がデフォルトエディタとして設定されていますが、**Files → Options** を選択して、他のエディタを指定することができます。

> **移植性**
> ここで記述するコードの内容はターゲットに依存するものであるため、ペンディング状態の割込みソースを検知したり判定する部分のコードについては、ここでは特に説明しません。

Task1 のコードを記述する

ナビゲーションバーの **Tasks** グループから **Task Data** サブグループを選択し、さらにタスク ***Task1*** を選択します。

Implementation View（実装ビュー）で、このタスクに必要なコードを確認してください。

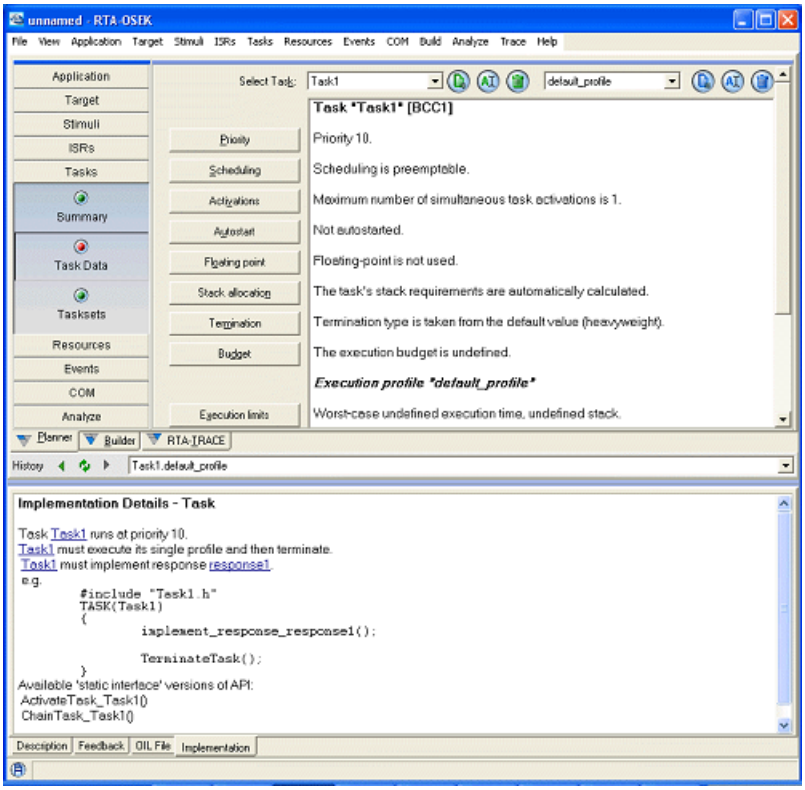


図 3-22 Task1 の実装ノートを表示する

ワークスペースの .c ボタンをクリックすると、タスクのソースコードを直接編集することができます。ここで記述するコードはターゲットに応じて異なりますが、実装ビューの内容を参考にして記述してください。

その他のタスクのコードを記述する

残りのタスク（*Task1*、*Task2*、*Task3*、*CanWorker*）のコードも、同じ要領で記述します。

RTA-OSEK の設定を変更する場合は、必ず、提示された実装内容どおりにコードが記述されているかを確認してください。

'main' のコードを記述する

C プログラムでは、`main()` がメインアプリケーションの開始点になります。これは、下位レベルの起動時初期設定の後に呼び出されます。通常、`main()` に入る前は割込みはディセーブルになっています。

コード例 3-1 は、RTA-OSEK が `main()` 用に生成するスケルトンコードです。

```
/* Template code for 'main' in project: UserApp */

#include "osekmain.h"

OS_MAIN()
{
   StartOS(OSDEFAULTAPPMODE);
   ShutdownOS(E_OK);
}
```

コード例 3-1 main( ) のテンプレートコード

コード例において、main() ではなく OS_MAIN() というマクロが使用されていることに注意してください。main( ) に使用できる引数と戻り値タイプはコンパイラにより異なるので、RTA-OSEK では、移植しやすいように OS_MAIN( ) を使用しています。

移植性

アプリケーションにおいて main( ) の代わりに OS_MAIN( ) が使用されていると、別のターゲットへの移植性が向上します。

オペレーティングシステムを起動するには、StartOS(OSDEFAULTAPPMODE) を呼び出します。この StartOS() の前には、オペレーティングシステム API コールを一切行わないでください。

ShutdownOS() は、アプリケーションの終了時に OS を終了させるために使用します。ShutdownOS() のデフォルトアクションは、無限ループを続け、コントロールを返さないようになっています。これによってアプリケーションは永久に、つまりプロセッサへの電力供給が停止されるか、またはプロセッサがリセットされるまで無限ループを続けてしまうことになるので、この関数は通常は使用されません。

サンプルアプリケーションにおいては、StartOS() を呼び出す前にターゲットハードウェアの初期設定を行う必要があります。ここでは、タイマを設定して 100ms ごとに割込みがかかるようにし、さらに各割込みソースをイネーブルにします。そして StartOS() を呼び出した後にアラームを有効にし、アイドルループに入るようにしてください。

StartOS() を呼び出した後は、SetAbsAlarm() を呼び出してアプリケーションが使用するアラームを設定します。SetAbsAlarm() には 3 つの引数があります。これは、設定するアラームの名前と開始時間、および周期です。「開始時間」は、アラームが最初に満了する時の時間です。この値は 0 にしないでください。0 にした場合、アラームカウンタの値が 0 に到達するまで、そのカウンタを 1 周分カウントしなければならなくなり、満了までの時間が非常に長くなってしまいます。「周期」は、開始時間の後、アラームが繰り返して満了する周期を定義します。このアプリケーションにおいては、各アラームの開始時間は 1ms です。また *Task1Alarm*、*Task2Alarm*、*Task3Alarm* の周期は、それぞれ 3ms、6ms、14ms です。

StartOS() の後に実行されるコードはアイドルタスクに属します。このアイドルタスクは osek_idle_task という名前です。このアイドルタスクは他のタスクと同様に、API コール、リソースの使用、メッセージ送受信、イベントの送信や待機など、さまざまな処理を行うことができます。ただし、このタスクは ShutdownOS() が呼び出されたときにしかターミネートしないので、直接起動することはできません。また、他のタスクの起動を妨げてしまうので、内部リソースを使用することもできません。

重要

アイドルタスク内にコードを記述することは、システム実装において非常に効率的な方法といえます。特に、OSEK イベントに応答するタスクが 1 つだけ必要な場合は、アイドルタスク osek_idle_task がイベントを待つようにすると、ユーザーのアプリケーションが大幅にサイズダウンし、応答速度が上がります。

RTA-OSEK Planner は典型的な OS_MAIN() の実装内容を提示します（Applicatin → Implementation）。ナビゲーションバーで、**Tasks** グループの **Task Data** サブグループ内の ***osek_idle_task*** タスクを選択し、その内容を確認してください。

ここではアイドルタスクは何も処理を行いませんが、「スリープ」ステートをサポートするターゲットでは、アイドルタスク内でプロセッサを「スリープ」ステートにすることができます。その場合、プロセッサは割込み発生によって「ウェイクアップ」します。

重要

アイドルタスクは絶対にターミネートせずに、無限ループを行わなければなりません。

タイマ／カウンタハードウェアをセットアップする

コード例 3-2 の関数 do_target_initialization() は、各種割込みソースを初期化します。このうちの 1 つのソースが、100ms ごとに割込みを発生させるハードウェアカウンタ／タイマです。

コード例 3-2 の内容を参考にしてください。

```
void do_target_initialization(void)
{

   unsigned int timer_divide;
   timer_divide = OSTICKDURATION_TimerCounter / OS_NS_PER_CYCLE;

   /* Initialize the timer hardware */
   SetupTimer(timer_divide);

   /* Other target initialization */
   ...
}
```

コード例 3-2 タイマハードウェアの初期化

コード例 3-2 は、RTA-OSEK が生成した `OSTICKDURATION_TimerCounter` と `OS_NS_PER_CYCLE` という 2 つの定数を用いる初期化処理を示しています。

`OSTICKDURATION_TimerCounter` という定数は、カウンタのチック（1 回のカウント）の間隔をナノ秒（ns）単位で定義するものです。この例では `OSTICKDURATION` の間隔は 1,000,000ns です。

`OS_NS_PER_CYCLE` 定数は、CPU 命令サイクルの長さを ns 単位で定義します。8MHz の CPU の場合は 125ns になります。

この例では、1ms の命令サイクルごとに割込みをかけるようにタイマを設定してください。これらの定数を使用して分周率を算出しておけば、クロックレートが変わった場合でもコードが自動的に調整を行います。

OS ステータス、エラーフック、コールバック

初期のテストにおいては、オペレーティングシステムの **Extended**（拡張）ビルドを使用してアプリケーションを実行してください。拡張ビルドは、OS が各 API コールについて厳密なチェックを行うことを意味しています。もちろん、これには時間とコードスペースがかかります。

初期テストにおいてアプリケーションが正しく機能していると判断されたら、通常は **Standard**（標準）ビルドに切り替えてください。標準ビルドではチェックはほとんど行われないので、OS は拡張ビルドの場合よりもはるかに効率的に実行されます。

拡張または標準ビルドを選択するには、以下のように操作します。

- *Planner* のナビゲーションバーから **Application** グループを選択してから、**OS Configuration** サブグループを選択します。
- OS Status をクリックし、ビルドステータスを選択します。

拡張ビルドを使用しているときには、各 API コールのリターンステータスコードを調べたり、エラーフックを使用するように要求することができます。エラーフックは、エラーが検知されたときに OS が呼び出す関数で、これはユーザーのアプリケーション内に実装します。通常、この関数は、デバッグを中断させ、エラーをユーザーに通知する処理を実行します。

エラーフックを使用するには、以下の手順を実行してください。

- ナビゲーションバーから **Application** グループを選択してから、**OS Configuration** サブグループを選択します。
- **Hooks** ボタンをクリックして、**Select Hooks** ダイアログボックスを開きます。
- **Error Hook** チェックボックスを選択してから **OK** ボタンをクリックします。

図 3-23 は、エラーフックを有効にした状態を示しています。

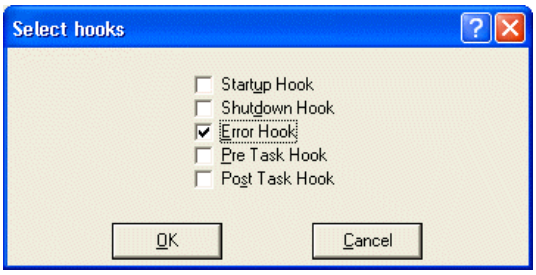


図 3-23 エラーフックを有効にする

`ErrorHook()` を実装するために必要なコードは、どのソースファイルに含めることもできますが、最初は `main.c` を使用することをお勧めします。

以下のコードを追加してください。

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType e)
{
   /* Put a debugger breakpoint here. */
   while (1) {
      /* Freeze. */
   }
}

#endif /* OSEK_ERRORHOOK */
```

コード例 3-3 ErrorHook( )

デバッグ時に `ErrorHook()` を使用する方法については、本書の第 16 章を参照してください。

`ErrorHook()` の他に、タイミングビルドまたは拡張ビルドを使用する場合に用意しなければならない関数が 3 つあります。オペレーティングシステムはこれらを使用してユーザーのコード実行のタイミングを管理します。

現時点では詳細について考える必要はないので、図 3-4 の内容をそのまま `ErrorHook()` の後に追加してください。

```
#ifdef OS_ET_MEASURE
   OS_HOOK(void) OverrunHook(void)
   {
      /* Put a debugger breakpoint here. */
      while (1) {
         /* Freeze. */
      }
   }
   OS_NONREENTRANT(StopwatchTickType) GetStopwatch(void)
   {
      /* Temporary implementation. A correct solution
       * returns the current stopwatch value. */
      return 0;
   }
```

```
   OS_NONREENTRANT(StopwatchTickType)
   GetStopwatchUncertainty(void)
   {
      /* Temporary implementation. A correct solution
       * returns the uncertainty in the stopwatch value. */
      return 0;
   }
#endif /* OS_ET_MEASURE */
```

コード例 3-4 タイミングコールバック

最終チェック

ナビゲーションバーの **Application** グループから **Implementation** サブグループを選択して、全体の実装情報を表示してください。

この情報は、アプリケーションが完全に実装されたことを確認するためのチェックリストとして使用できます。メニューから **File → Print Selection** を選択すれば、この情報を印刷することができます。

3.2.5 ビルド

このサンプルアプリケーション作成の全ステップを完了したら、ビルド工程を開始します。*Builder* に切り替え、3.6.2 項を参照してビルドを実行してください。

3.2.6 機能テスト

実行可能ファイルをターゲットハードウェアにダウンロードして、その挙動を確認します。

最初は、必ずエラーフックを含んだ拡張ビルドを行い、API コールの誤用があれば OS がそれを検知するようにしてください。アプリケーションが正しく実行できた後に、タイミングビルドまたは標準ビルドに切り替えてください。

3.3 タイミング分析を使用するアプリケーションの例

次の例として、ここでは「スティミュラス - レスポンス」モデルを使用する簡単なアプリケーションを作成し、そのタイミング分析を行う方法を説明します。

3.3.1 仕様

ここで作成するシステムの仕様は以下のとおりです。

- ターゲットプロセッサはモトローラ HC12 です。(このターゲットがインストールされていない場合は、他のターゲットを選択してください。)
- ターゲットクロックは 8MHz です。
- *Button1* というボタンは何度も押下できますが、その頻度は 1 秒間に 2 回までと限定されます。2 回連続して押す場合、最低で 0.1 秒の間隔をあける必要があります。

図 3-24 は、これらの要件を示したものです（図中の B は、ボタンが押下されるタイミングを示しています)。

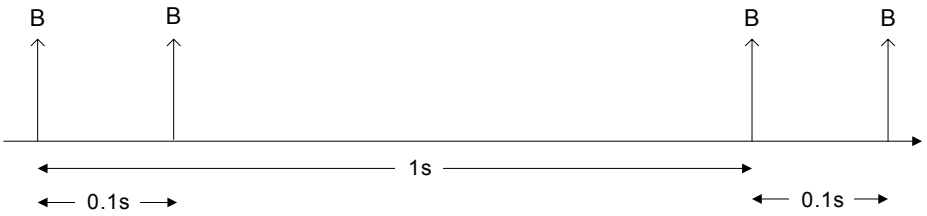


図 3-24 Button1 の要件

- *Lamp1* というライトは、*Button1* が押されてから 10ms 以内に点灯しなければなりません。
- *Lamp2* は、*Button1* が押されてから 11ms 以内に消灯しなければなりません。

- *Motor1* というモータは、*Button1* が押されてから 200ms 以内に始動しなければなりません。

図 3-25 は、これらの要件を示したものです。

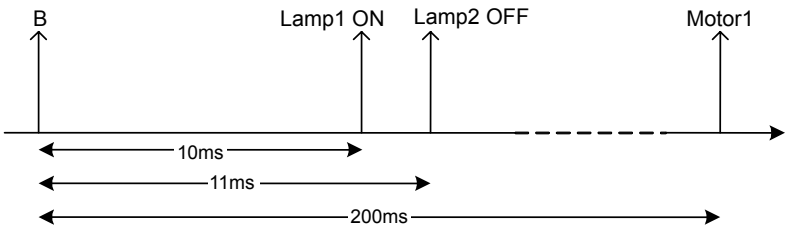


図 3-25 Button1 押下後の Lamp1、Motor1、Button1 の挙動要件

- *Motor1* が完全に作動、つまり所定の回転数に到達すると、割込みが発生します。
- *Lamp1* は、モータが完全に作動してから 11ms 以内に消灯しなければなりません。
- *Lamp2* は、モータが完全に作動してから 10ms 以内に点灯しなければなりません。

図 3-26 は、これらの要件を示したものです。

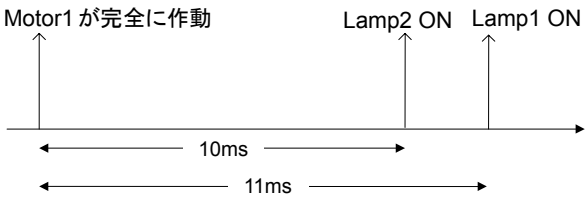


図 3-26 Motor1 作動後の Lamp1、Lamp2 の挙動要件

- *Lamp3* は、1 秒ごと（精度は± 2ms）に点灯と消灯に切り替わらなければなりません。

図 3-27 は、この要件を示したものです。

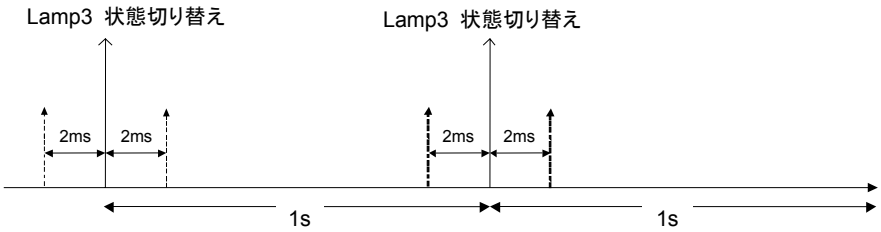


図 3-27 Lamp3 の挙動要件

- ボタン *Button1* の入力にはデバウンス回路が使用されているため、ボタン押下がプロセッサに伝達されるまでに 0.1 ～ 0.4ms かかります。
- プロセッサがライトへの電流供給を始めてからライトのフィラメントが実際に点灯するまでに 0.5ms かかります。
- プロセッサがライトへの電流供給を停止してからライトのフィラメントが消灯したとみなされるまでに 0.3ms かかります。
- プロセッサがモータに電力を供給してから *Motor1* が始動するまでに 50ms かかります。

RTA-OSEK GUI を使用して新しいアプリケーションを作成する

新しいアプリケーションを作成するには、RTA-OSEK GUI で、メニューコマンド **File → New** を選択します。**Select Target** ダイアログボックスが開くので、**Available Targets** と **Variant** のドロップダウンリストからターゲットプロセッサを選択してください。

図 3-28 では、*HC12/COSMIC 16 task* というターゲットが選択され、*HC12* というバリアントが選択されています。HC12 がインストールされていない場合は、別のターゲットを選択してください。

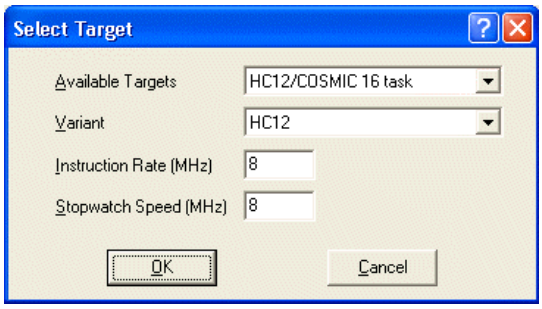


図 3-28 ターゲットプロセッサの選択

重要

Available Targets リストには、ユーザーのコンピュータにインストールされているターゲットだけが、ユーザーのライセンスファイルに基づいて表示されます。必要なターゲットが表示されない場合は、ETAS までお問い合わせください。各ターゲットには、共通のプロセッサコアをベースとする様々なチップバージョンに対応する、多数のバリアントがあります。

図 3-28 に示された **Select Target** ダイアログボックスでは、**Instruction Rate** と **Stopwatch Speed** の設定も行えます。インストラクションレートは、プロセッサ内で最小のインストラクションサイクルに基づいて設定してください。

ストップウォッチの速度は、タイミングビルドおよび拡張ビルドで用いられる `GetStopwatch()` にアタッチされているタイマハードウェアが使用するサンプルレートによって決まります。この関数は、OS とアプリケーションコードの実行時間を測定するために使用されます。ストップウォッチがインストラクションレートと同じ速度で作動することが理想的ですが、ターゲットによってはそれは不可能です。

ターゲット情報の設定後、**OK** ボタンをクリックすると、RTA-OSEK GUI は新しいアプリケーションについての情報を、図 3-6 のように表示します。この情報はいつでも参照できるので、作成中のシステム全体の概要を容易に把握することができます。

アプリケーションを保存する

新しいアプリケーションを作成した後は、速やかに保存するようにしてください。アプリケーションを初めて保存するときは、**File → Save As...** を選択します。

Save As ダイアログボックスでは、**Save In** リストで、ファイルを保存したいロケーションまでナビゲートします。**File Name** に新しい名前を入力し、**Save** ボタンをクリックして新しいアプリケーションを保存してください。

アプリケーションの保存は、**File → Save** を選択するか、キーボードの **<Ctrl>** キーと **<S>** キーを同時に押す（**<Ctrl> + <S>**）ことにより、随時行うことができます。

OIL ファイルを表示する

作成されたファイルは OIL V2.3 の構文で保存されるので、他の OSEK 互換ツールからもこのファイルを読むことができます。現在のアプリケーション用の OIL ファイルの内容を見るには、メニューから **View → OIL File** を選択してください。

図 3-7 のように、OIL ファイルの内容がワークスペースの下半分に表示されます。

ウィンドウ上部のワークスペースに詳細情報が表示され、ウィンドウ下部に OIL ファイルが表示されます。

その他の RTA-OSEK 固有の情報は、**//RTAOILCFG** から始まるコメントとして OIL ファイルに保存されます。これらのコメントは通常、OIL ビューには表示されませんが、これを表示されるようにすることもできます。

RTA-OSEK 固有のコメントを表示するには、メニューから **File → Option** を選択してください。**Options** ダイアログボックスが開くので、図 3-29 に示されるように、**Show RTA Extended OIL in View** オプションを有効にします。

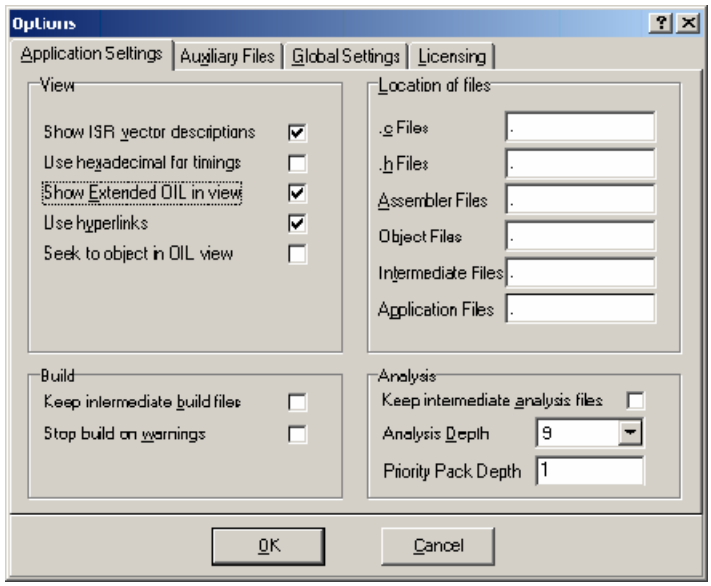


図 3-29 RTA-OSEK GUI の Options ウィンドウ

Options ダイアログボックスの **OK** ボタンをクリックすると、OIL ファイル内のコメントが表示されます。

レガシー OIL ファイルをインポートする場合、OIL の CPU オブジェクト内に記述されたコメント（ただし RTA-OSEK が生成したコメント以外）は保持されません。オブジェクト外のコメントと OIL ディスクリプションのみ保持されます。

View → OIL File を再度選択してこのメニューコマンドに付いているチェックマークを消すと、OIL ビューが閉じます。

重要

拡張された `//RTAOILCFG` 構文を使用する OIL ファイルは、ハンドコーディングによる編集は行わないでください。このような OIL ファイルには内部的な依存関係があるため、RTA-OSEK がそのファイルを読み込む際に重要な情報が失われてしまう可能性があります。

スティミュラスとレスポンスを入力する

仕様をコードに実装するには、スティミュラスとレスポンスを定義しておく必要があります。このためには、図 3-30 に示されているように、ナビゲーションバーから **Stimuli** グループを選択してください。ワークスペースには **Stimulus Summary**（スティミュラスについての情報）が表示されます。この時点では、まだこのアプリケーションにスティミュラスが存在しないことが示されています。

アプリケーションでは、**bursty**（バースト）、**alarm**（アラーム）、**periodic**（周期的）、**planned**（計画的）という 4 種類のスティミュラスを使用できますが（これらのスティミュリについては第 10 章と第 13 章に詳しく説明されています）、ここでは、*Button1* の押下と *Motor1* の完全な作動に対応するバーストスティミュラスを作成します。またさらに、*Lamp3* の 1 秒ごとの ON/OFF 切り替えに対応するアラームスティミュラスも作成します。

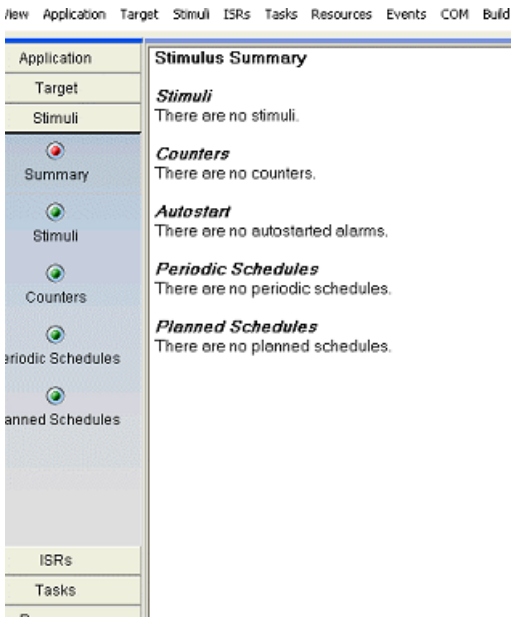


図 3-30 スティミュラスについての情報

新しいバーストスティミュラスを追加するには、ナビゲーションバーから **Stimuli** サブグループを選択してください。すると **Select Stimulus** ドロップダウンリストが表示され、3 つのボタンが有効になります。

アプリケーション内にスティミュラスが存在しない場合は、**Add Stimulus** ボタンのみが有効になります。

スティミュラスのワークスペースの **Add** ボタンをクリックすると、**Add Stimulus** ダイアログボックスが開くので、***Button1Press*** という名前を入力してください。

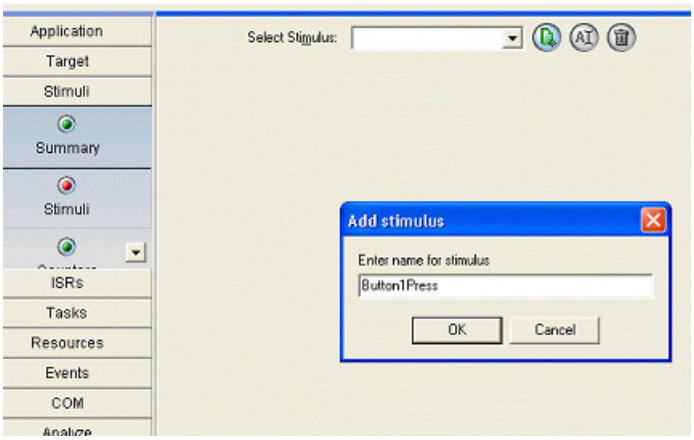


図 3-31 新しく作成するスティミュラスの名前を入力する

図 3-31 に示されている **OK** ボタンをクリックすると、新しいスティミュラスが作成されます。スティミュラスのデフォルトタイプはバースト（'bursty'）です。

図 3-32 に示されているように、ワークスペースの上部には **Response** ドロップダウンリストが表示され、すべてのボタンが有効になります。ワークスペースの左側にあるボタンは、スティミュラスのプロパティを変更する際に使用します。

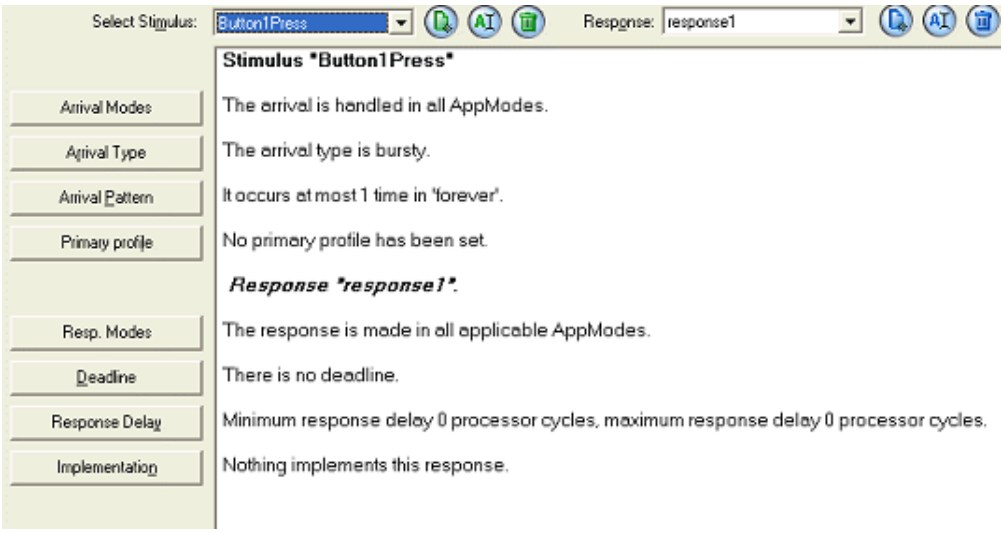


図 3-32 新しいバーストスティミュラスについての情報

これで、*Button1Press* というバーストスティミュラス（デフォルト設定）が作成されました。このスティミュラスは、1 回しか発生しません。ここではまだプライマリプロファイルが 1 つも設定されていないので、このスティミュラスはまだ検知できません。バーストスティミュラスは、発生時間が特定されず、かつ最大レートを定義できるようなイベントのモデリングに使用されます。

この時点において、*response1* という名前のデフォルトレスポンスが RTA-OSEK によって自動的に作成されています。

ワークスペースには、この新しく作成されたレスポンスについてのデフォルトの詳細情報が表示されていますが、この時点ではこのレスポンスにはデッドラインや実装についてなにも定義されていません。

仕様によれば、*Button1Press* は、毎秒 2 回を超えない頻度で、かつ 0.1 秒以上の間隔を置いて何度も発生する可能性があります。

この仕様をアプリケーションに追加するために、以下の手順を実行してください。

- Stimulus ワークスペースの **Arrival Pattern** ボタンをクリックして **Arrival Burst Pattern** ダイアログボックスを開きます。
- このダイアログボックスの 1 行目の **At Most...** 列に **1** を入力し、**In Any...** 列に **0.1** を入力してから、ドロップダウンリストから **real time** と **s** を選択します。

図 3-33 は、上記の手順で入力した情報を示しています。

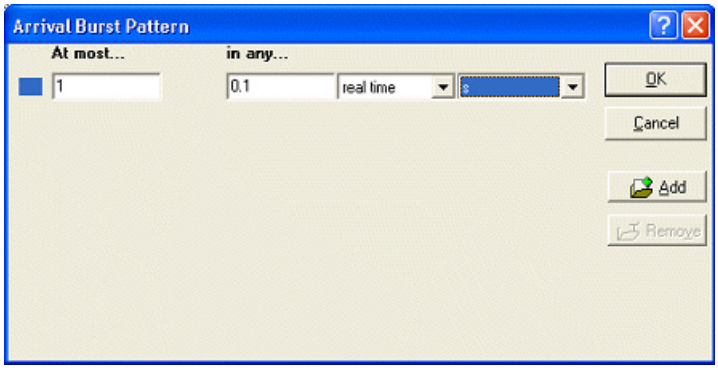


図 3-33 最初のバーストアライバルパターンを入力する

続いて、'at most 2 times in any 1 real time s' のバーストパターンをもう 1 つ追加する必要があります。以下の手順で追加してください。

Arrival Burst Pattern ダイアログボックスの 

ボタンをクリックすると、新しいエントリがダイアログボックスに追加されます。

- **At Most…** 列に **2** を入力します。**In Any…** 列に **1** を入力し、ドロップダウンリストから **real time** と **s** を選択します。
- **OK** ボタンをクリックして、変更内容を保存します。

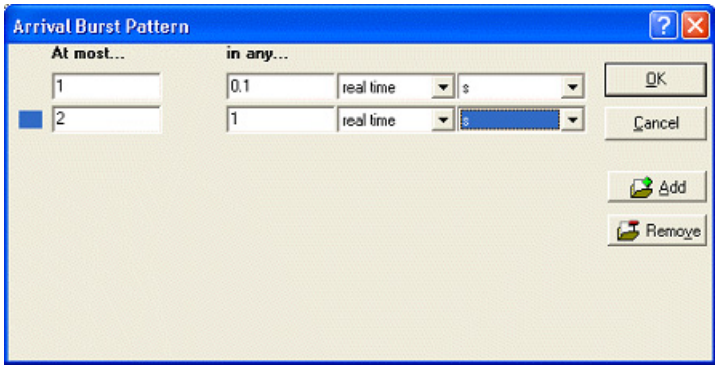


図 3-34 複雑なバーストパターンを持つスティミュラスの設定

仕様の次の部分では、*Button1Press* から 10ms 以内に *Lamp1* が点灯しなければなりません。これはつまり、*Lamp1* が点灯するまでのデッドラインです。

ここでは、電流がオンになってから実際にライトが点灯するまでに 0.5ms の遅延があることを考慮に入れます。

- Stimulus ワークスペースの **Rename Response** ボタンをクリックします。**Rename** ダイアログボックスが開くので、そこで *response1* という名前を ***Lamp1On*** に変更します。このように名前を変更することにより、*Button1Press* スティミュラスが検知されたときにどのレスポンスが生成されるかが明確になります。
- 図 3-35 に示される **OK** ボタンをクリックして新しい名前を保存します。

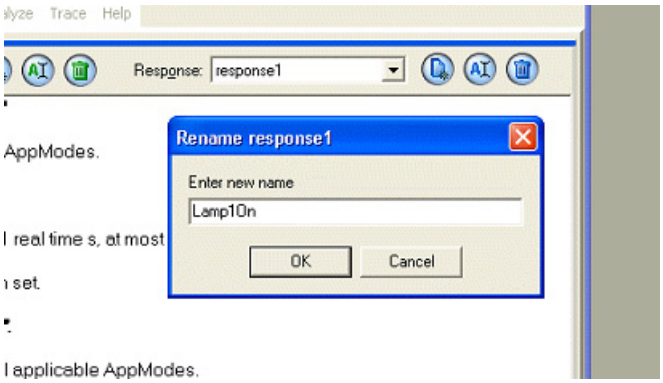


図 3-35 レスポンス名を変更する

- **Deadline** ボタンをクリックし、図 3-36 に示すように、デッドラインとして **10 real time ms** を設定します。

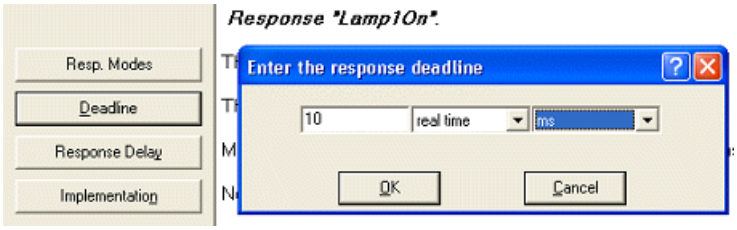


図 3-36 Lamp1On レスポンスのデッドラインを入力する

- **Response Delay** ボタンをクリックします。**Max** 値に **0.5 real time ms** を設定し、**Min** 値に **0** を設定します。

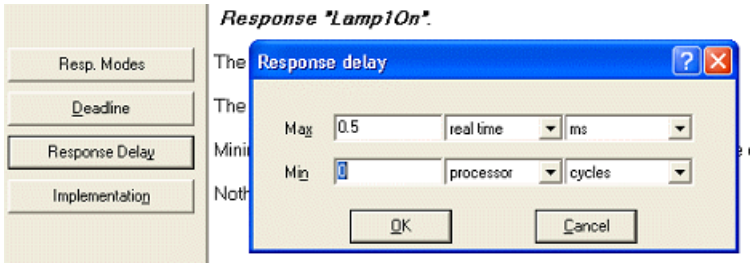


図 3-37 Lamp1On レスポンスのレスポンス遅延を設定する

ワークスペースには、図 3-38 のように、入力した詳細情報が表示されます。

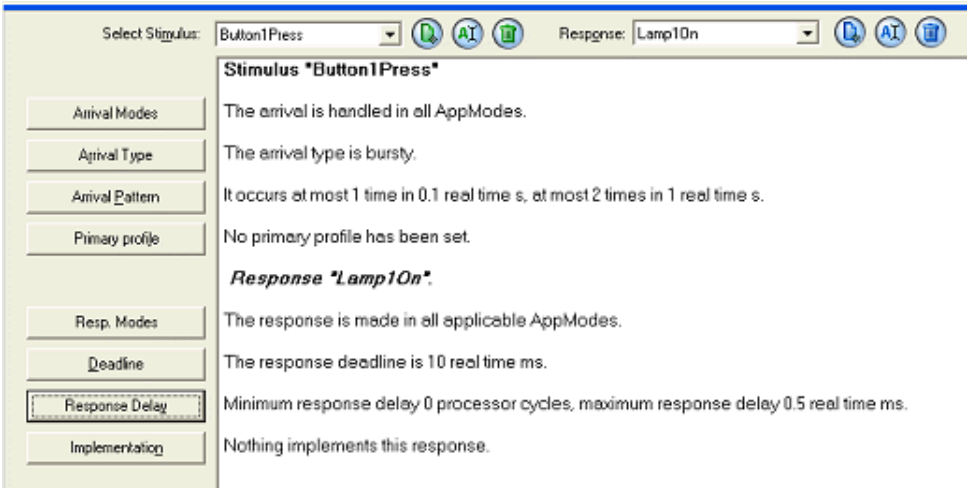


図 3-38 Button1Press スティミュラスと Lamp1On レスポンスについての情報

仕様のその次の部分では、*Button1Press* スティミュラスから 11ms 以内に *Lamp2* を消灯しなければなりません。また、電流がオフになってからライトが消灯するまでに、0.3ms の遅延があることも考慮する必要があります。以下の手順を実行してください。

- **Add** ボタン（**Response** ドロップダウンリストの右にあります）をクリックして、新しいレスポンスを追加します。
- ***Lamp2Off*** という名前を入力し、**OK** ボタンをクリックします。
- **Deadline** ボタンをクリックし、デッドラインとして **11 real time ms** を設定します。
- **Response Delay** ボタンをクリックし、**Max** 値として **0.3 real time ms** を設定し、**Min** 値として **0** を設定します。

この時点の詳細情報が、図 3-39 のようにワークスペースに表示されます。

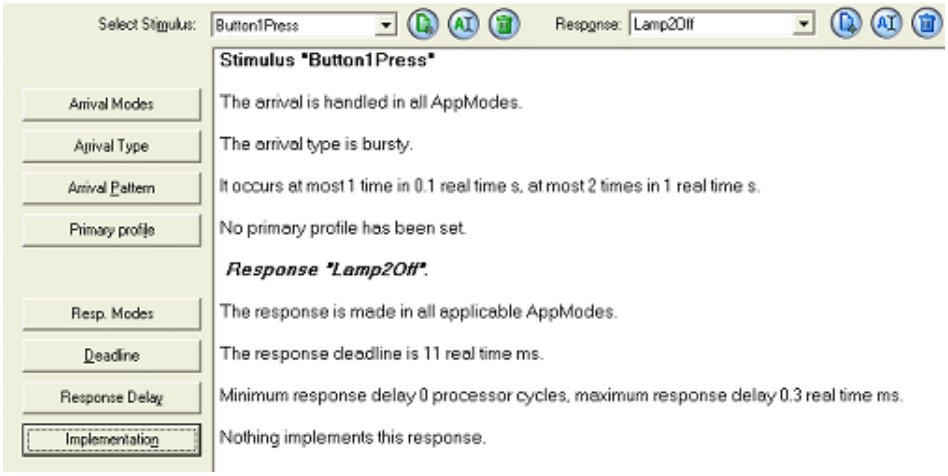


図 3-39 新しい Lamp2Off レスポンスについての情報

次に、以下の手順を実行して、*Motor1* を始動するレスポンスを追加する必要があります。

- ***Motor1On*** という新しいレスポンスを追加します。**Response** ドロップダウンリストの右にある **Add** ボタンをクリックしてください。
- *Motor1On* の **Deadline** に **200 real time ms** を設定します。
- **Response Delay Max** 値に **50 real time ms** を設定し、**Min** 値に **0** を設定します。

さらに、仕様によれば、1 秒ごと（誤差は± 2ms 以内）に *Lamp3* のオンとオフが切り替わる必要があります。これをアプリケーションに追加するには、新しいスティミュラスと新しいレスポンスを 1 つずつ追加する必要があります。

- **Select Stimulus** ドロップダウンリストの右にある **Add** ボタンをクリックして、新しいスティミュラスを追加します。
- ***Lamp3Toggle*** という名前を入力し、**OK** ボタンをクリックします。
- **Arrival Type** ボタンをクリックし、スティミュラスが周期的なものになるように ***Periodic*** オプションを選択します。
- **Arrival Pattern** ボタンをクリックし、周期として **1 real time s** を設定します。
- ± 2ms という切り替え時間の許容誤差を満たすために、**Deadline** ボタンをクリックし、**4 real time ms** というデッドラインを入力します。
- **Response Delay** ボタンをクリックし、**Max** 値として **0.5 real time ms** を設定します（ライト点灯の遅延）。

このスティミュラスとレスポンスについての情報は、ワークスペースに図 3-40 のように表示されます。

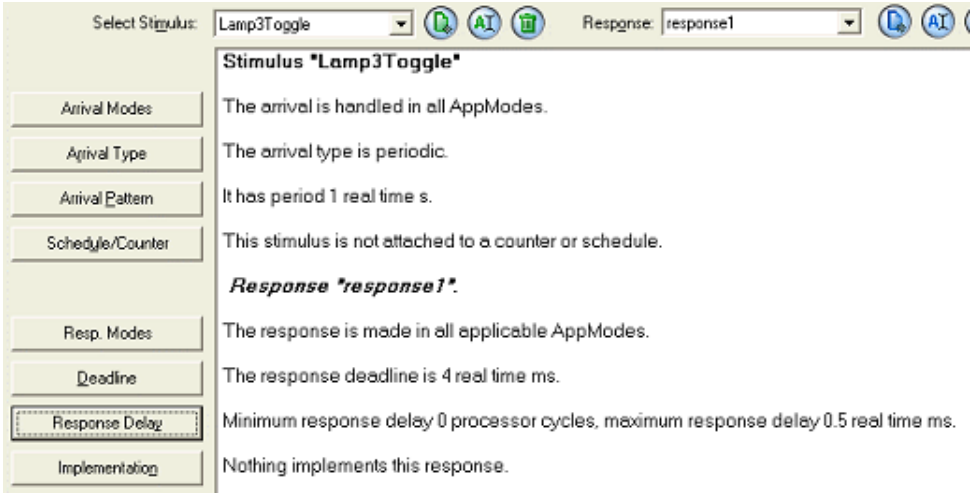


図 3-40 Lamp3Toggle というスティミュラスとそのレスポンスについての情報

最後に、モータが作動状態に到達したときの処理を追加します。

- ***Motor1Running*** という新しいバーストスティミュラスを追加します。
- デフォルトのレスポンス名を ***Lamp2On*** に変更します。
- **10 real time ms** という **Deadline** を入力します。
- **0.5 real time ms** という **Response Delay** を入力します。
- ***Lamp1Off*** という新しいレスポンスを作成します。
- **11 real time ms** という **Deadline** を指定します。
- **0.3 real time ms** という **Response Delay** を入力します。

Motor1Running スティミュラスについての情報は、図 3-41 のようになります。

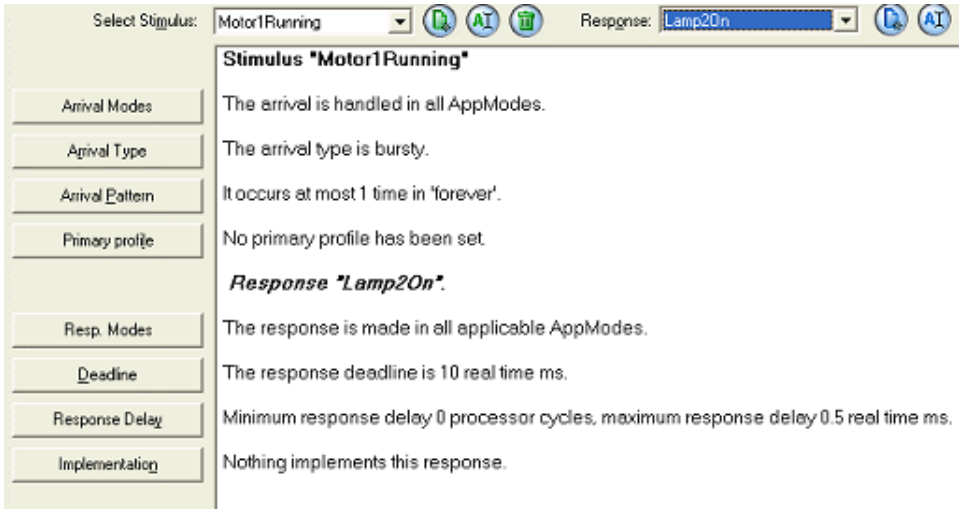


図 3-41 作成された Motor1Running スティミュラスについての情報

これで、仕様で定義された機能を入力できました。スティミュラスについての情報を表示して、入力した詳細情報を確認してください。ワークスペースには、図 3-42 のように表示されるはずです。

図 3-42 スティミュラスについての情報を確認する

メニューから **File → Save** を選択して、定期的にアプリケーションを保存するようにしてください。

3.3.2 実装

ここから実装段階に入ります。スティミュラスの検知方法とレスポンスの実現方法を決定し、RTA-OSEK GUI を使用して ISR とタスクを作成します。

ISR を作成する

このサンプルアプリケーションでは、3 つの割込みソースが必要です。1 つはボタン押下の検知に使用され、もう 1 つは 100ms ごとに割込みを発生させるハードウェアタイマ、3 つめはモータにアタッチされます。

これら 3 つの割込みソースはすべて 1 つの ISR により管理されるものとし、この ISR を *PrimaryISR* と呼びます。

新しい ISR を作成するには、ナビゲーションバーから **ISRs** グループを選択します。

ISRs グループを選択すると、図 3-43 に示されるように、ワークスペースに ISR についての情報が表示されます。

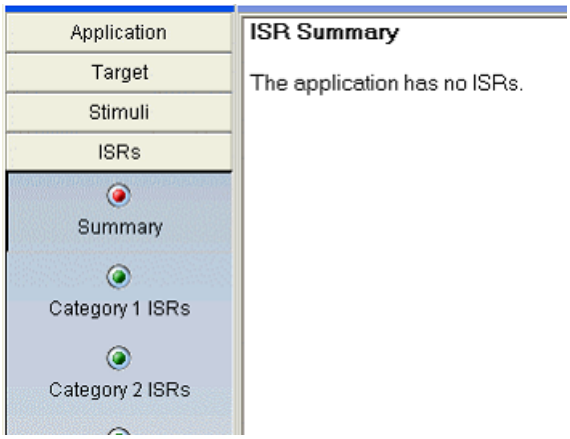


図 3-43 ナビゲーションバーから ISRs グループを選択する

ここで作成する ISR はタスクの起動を行う必要があるため、カテゴリ 2 の ISR を使用する必要があります。カテゴリ 1 の ISR からの OS API コールは許可されていません。

- ナビゲーションバーから、**Category 2 ISRs** サブグループを選択します。
- ワークスペースの **Add** ボタンをクリックして、ISR を作成します。
- **Add Cat 2 ISR** ダイアログボックスに、***PrimaryISR*** という名前を入力して **OK** をクリックします。
- ターゲットタイプによっては、**Vector**（割込みベクタ）と **Priority**（優先度）を入力する必要があります。この例を図 3-44 に示します。

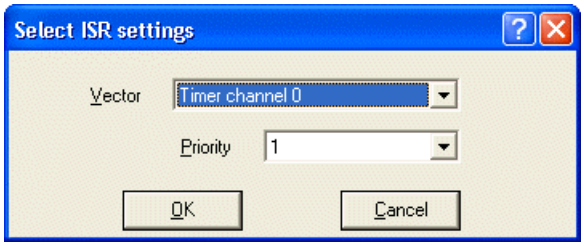


図 3-44 ISR のベクタと優先度を選択する

図 3-11 のように、新しく作成された ISR についてのデフォルト設定がワークスペースに表示されます。

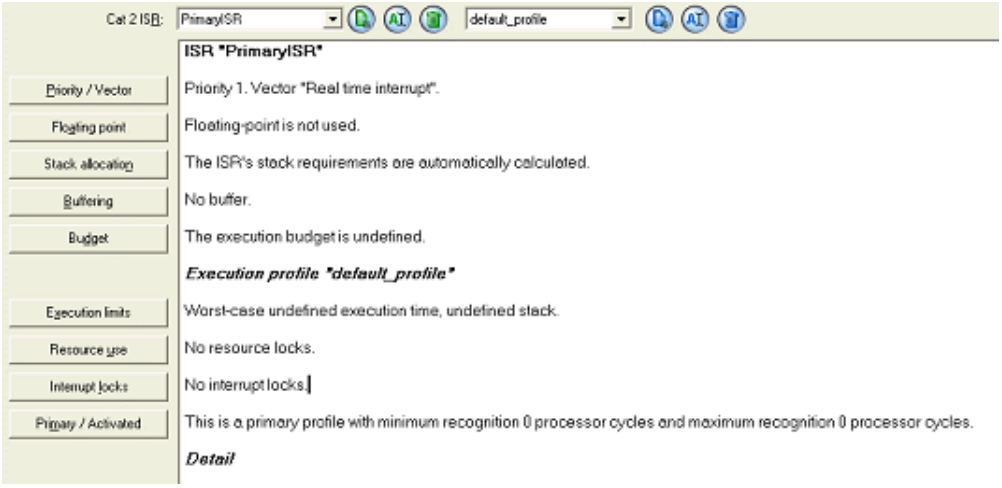


図 3-45 新しく作成された PrimaryISR についての情報

ワークスペースの上部に示されているように、RTA-OSEK は新しい ISR 用に *default_profile* という「実行プロファイル」を自動的に作成しています。

この「実行プロファイル」は、タイミング分析に用いられ、タスクや ISR 内の 1 つの実行パスを定義するものです。この例において ISR は、異なるスティミュラスがそれぞれ正しく処理されるように、どの割込みソースがペンディング（処理待ち）状態になっているかを判断する必要があります。

この ISR は、1 つの割込みソースを検知した後は、他のソースを調べずに終了します。そのため、この ISR が取りうる実行パスは 3 とおりになります。このコードはコード例 3-5 のようになります。

```
#include "PrimaryISR.h"
ISR(PrimaryISR)
{
   if (Button1PressInterruptPending()) {
      /* Button1Press detected. */
   } else if (Motor1RunningPending()) {
      /* Motor1Running detected. */
} else {
   /* Timer expiry detected. */
   }
}
```

コード例 3-5 ISR の実行パス

つまり、ここでは 3 つの実行プロファイルを作成する必要があります。

- 実行プロファイルドロップダウンリストの右にある **Rename** ボタンをクリックして、図 3-46 に示すように既存の実行プロファイルの名前を *default_profile* から ***pButtonPress*** に変更します。

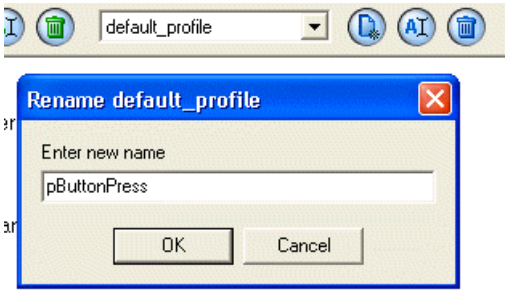


図 3-46 実行プロファイル名を変更する

次に、*Button1* のデバウンス回路に対応するための時間許容値を指定するため、1 番目のプロファイルに認識時間を設定します。これは実際のイベントの発生から、その結果としてプロセッサにおいて状態変化が発生するまでの最短／最長時間です。この「認識時間」の値は、特に、分散システムにおいてタイミング分析をする際に重要となります。

- **Primary/Activated** ボタンをクリックして、**Primary or Activated Profile** ダイアログボックスを開きます。
- 図 3-47 に示すように、この割込みの **Recognition Time** の **Max** 値として ***0.4 real time ms*** を設定し、***Min*** 値として ***0.1 real time ms*** を設定します。
- **OK** ボタンをクリックします。

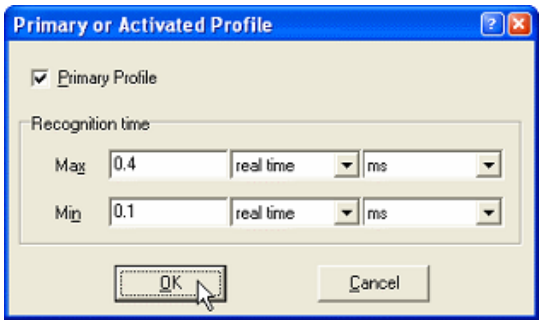


図 3-47 プライマリプロファイルを設定する

- 図 3-48 に示すように、（実行プロファイルドロップダウンリストの右にある）**Add** ボタンをクリックして新しいプロファイルを作成します。

図 3-48 新しい実行プロファイルを追加する

- **Add Execution Profile** ダイアログボックスに、***pMotor1Running*** という名前を入力します。
- 次に、***pTimer*** という別の新しいプロファイルを作成します。

ここで、この 3 つの新しいプロファイルが各割込みソースに対応していて、かつ一度に 1 つしか処理されない、ということを RTA-OSEK GUI に知らせる必要があります。

このためには、ISR が連続してトリガされることを RTA-OSEK GUI に知らせます。つまり、ISR は 1 つの割込みを処理すると終了し、その後、ペンディングされていた別の割込みによって再度トリガされます。

- ISR ワークスペースの **Buffering** ボタンをクリックし、**Specify ISR buffering behavior** ダイアログボックスを開きます（図 3-49 参照）。**Sample - no buffering** オプションを有効にし、さらに ISR のバッファリングオプションの **Retrigger after Leaving ISR** と **Buffer by Execution Profile** を有効にします。

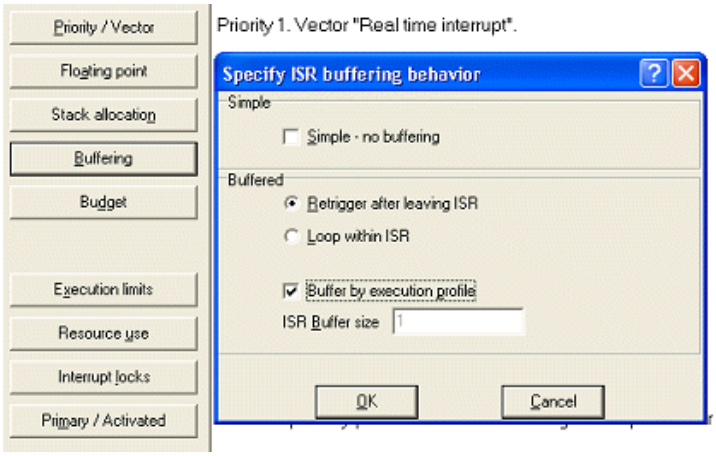


図 3-49 PrimaryISR の ISR バッファリング挙動を指定する

Summary サブグループを選択すると、図 3-50 に示すように、ISR についての情報が表示されます。

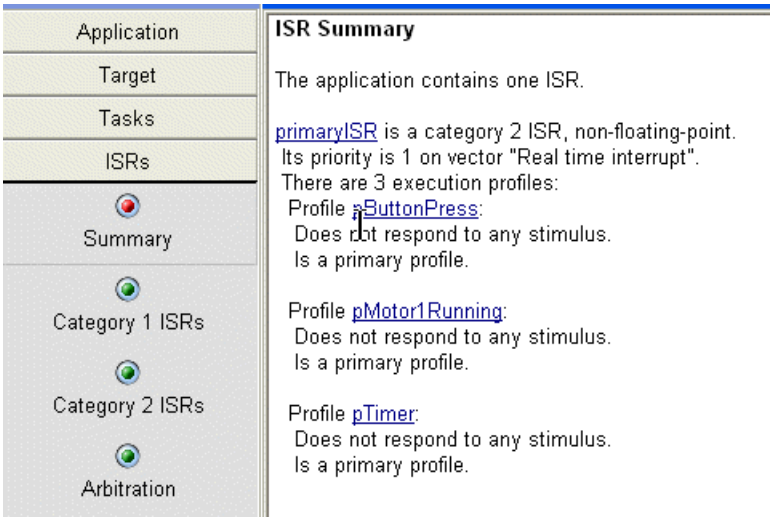


図 3-50 ISR についての情報を表示する

ISR をスティミュラスにアタッチする

ISR を作成したので、次にこれを作成済みのスティミュラスにアタッチする必要があります。この関連付けを行うことにより、そのスティミュラスが検知された時にその ISR が所定のレスポンスの生成を行う必要ある、ということを、RTA-OSEK GUI に知らせます。

- ナビゲーションバーから **Stimuli** グループを選択します。
- **Stimuli** サブグループを選択してから、**Select Stimulus** ドロップダウンリストから ***Button1Press*** を選択します。
- **Primary Profile** ボタンをクリックし、**Select Profile** ダイアログボックスを開きます。
- 図 3-51 に示すように、**Primary Profiles** ドロップダウンリストから ***PrimaryISR.pButtonPress*** を選択して **OK** ボタンをクリックします。

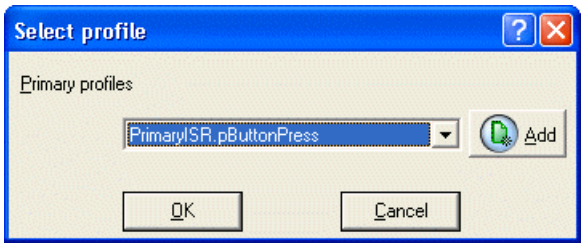


図 3-51 Button1Press のプライマリプロファイルを選択する

これで、*PrimaryISR* という ISR のプロファイル *pButtonPress* が、*Button1Press* というスティミュラスに対する処理を行う、ということが定義されました。

- **Select Stimulus** ドロップダウンリストから ***Motor1Running*** スティミュラスを選択します。
- **Primary Profile** ボタンをクリックします。**Select Profile** ダイアログボックスから ***PrimaryISR.pMotor1Running*** を選択し、**OK** ボタンをクリックします。

pTimer プロファイルを使える状態にするには、他にも設定しなければならないことがあります。このタイマは 100ms ごとに割込みをかけますが、*Lamp3* のオン／オフ切り替えは 1 秒ごとにしか発生しません。このため、各割込みのチックを数えて 10 チックごとに *Lamp3Toggle* スティミュラスを発生させるものが必要になりますが、これは OSEK カウンタオブジェクトを使用することによって実現できます。

- ナビゲーションバーの **Stimuli** グループから **Counters** サブグループを選択します。
- **Add** ボタンをクリックして、***TimerCounter*** という新しいカウンタを作成します。
- プロンプトが表示されたら、**Fastest Tick Rate** として ***100 real time ms*** を指定します。

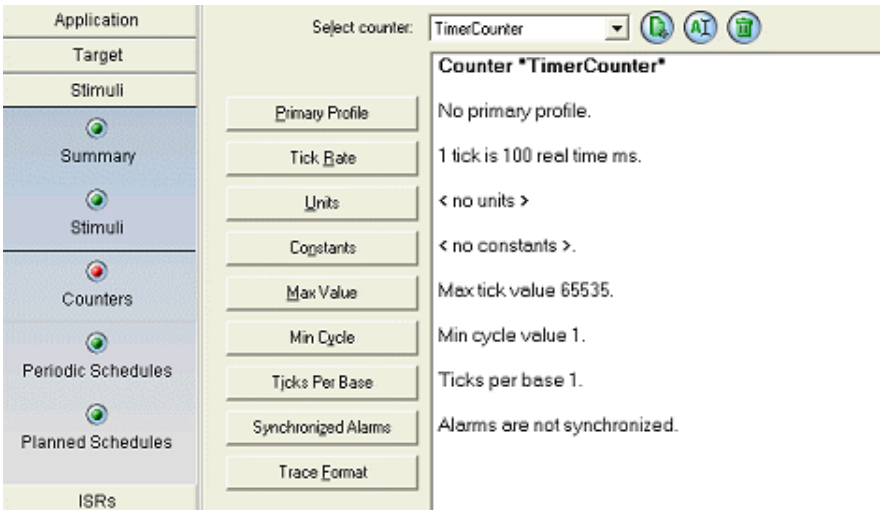


図 3-52 新しいカウンタについてのデフォルトの情報を表示する

- **Primary Profile** ボタンをクリックし、**Primary Profiles** ドロップダウンリストから ***PrimaryISR.pTimer*** を選択します。
- ナビゲーションバーから **Stimuli** サブグループに戻り、***Lamp3Toggle*** というスティミュラスを選択します。
- **Counter** タブの **Schedule/Counter** ボタンをクリックし、このスティミュラスが ***TimerCounter*** というカウンタにアタッチされるようにします（図 3-53 参照）。これにより、このスティミュラスは「OSEK アラーム」として実装されたことになります。

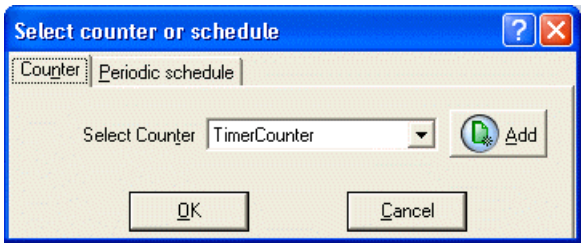


図 3-53 Lamp3Toggle 用のカウンタを選択する

ワークスペースに、スティミュラスについての情報が図 3-54 のように表示され、3 つのスティミュラスについてそれぞれ適切なプライマリプロファイルが定義されたことがわかります。

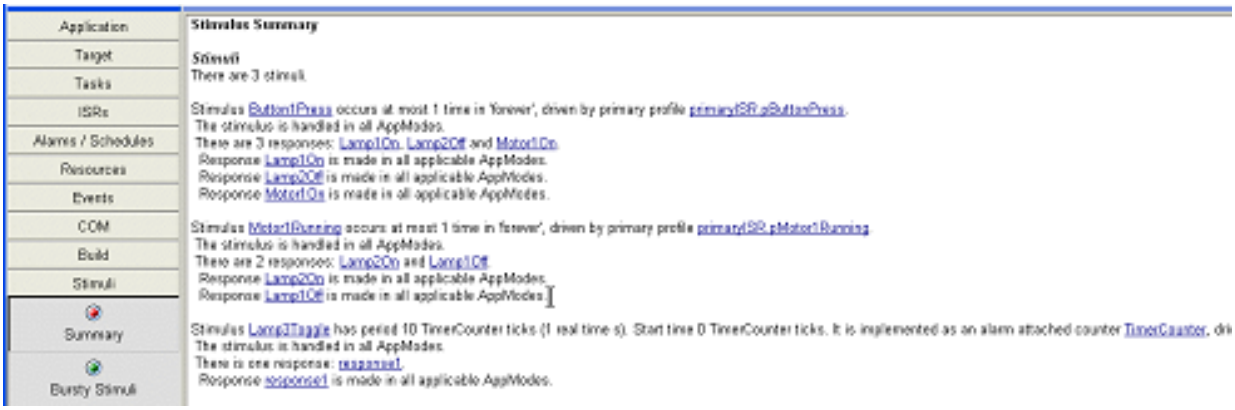


図 3-54 Stimulus Summary に表示されたプライマリプロファイルについての情報

レスポンスを作成する

レスポンスは通常、タスク内に実装される、ということはすでに説明しました。この例では、次の 4 つのタスクが必要です。

- *Button1* が押下されたときの *Lamp1On* と *Lamp2Off* というレスポンスを、*Button1Response* というタスクに実装します。
- *Button1* が押下されたときの *MotorOn* というレスポンスを、*MotorStart* というタスクに実装します。
- アラーム *LampToggle* が発生したときの *Lamp3Toggle* というレスポンスを、*Lamp3Toggle* というタスクに実装します。
- モータが作動したときの *Lamp2On* と *Lamp1Off* というレスポンスを、*MotorResponse* というタスクに実装します。

関係するデッドラインを見てみると、***LampToggle*** のデッドラインが最も短いので、このタスクの優先度を最も高くする必要があります。***MotorOn*** のデッドラインが最も長いので、このタスクの優先度を最も低くしても構いません。残りの 2 つのタスクの優先度は「中程度」にするとよいでしょう。

以下の手順に従って ***Button1Press*** スティミュラスのレスポンスを設定することから始めましょう。

- ***Button1Press*** というスティミュラスと ***Lamp1On*** というレスポンスを選択します。
- **Implementation** ボタンをクリックしてから、**Add** ボタンをクリックします。

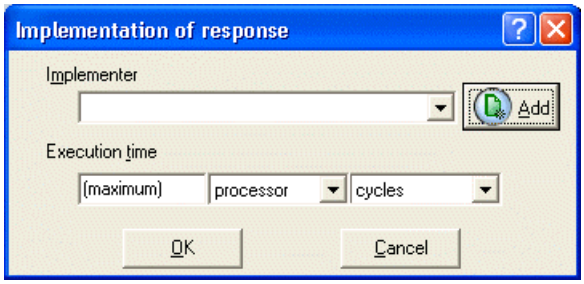


図 3-55 Implementation of response ダイアログボックスで、タスクまたは ISR を作成する

Create Task or ISR ダイアログボックスが開きます。

- **Task** オプションを選択し、**OK** ボタンをクリックします。

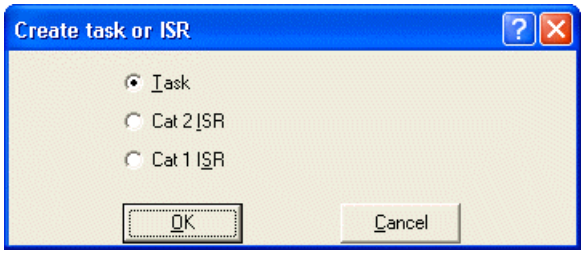


図 3-56 新しいタスクを作成する

Add Task ダイアログボックスが開きます。

- ***Button1Response*** という名前のタスクを作成します。これに、優先度 **10** を割り当て、**OK** をクリックします。
- 実行プロファイルは ***default_profile*** のままにしておきます。この段階では、実行時間を指定する必要はありません。

図 3-57 Lamp1On レスポンスの実装情報を表示する

- **Select Stimulus** ドロップダウンリストで *Button1Press* が選択されていることを確認してから、**Response** ドロップダウンリストから ***Lamp2Off*** を選択します。
- **Implementation** ボタンをクリックします。ここでは、直前に作成したタスク内にレスポンスを実装しようとしているので、単に実行プロファイルから ***Button1Response*** を選択するだけでかまいません。続いて **OK** をクリックします。
- ワークスペースの **Response** ドロップダウンリストから ***MotorOn*** レスポンスを選択します。
- **Implementation** ボタンをクリックし、優先度 **5** の ***MotorStart*** という新しいタスクを作成します。ここでも、実行プロファイルは ***default_profile*** のままにしておきます。この段階では、実行時間を指定する必要はありません。

次に、*Lamp3Toggle* というスティミュラスに対するレスポンスを追加します。

- スティミュラス ***Lamp3Toggle*** を選択します。
- **Implementation** ボタンをクリックし、優先度 **20** の ***LampToggle*** という新しいタスクを追加します。

最後に、*Motor1Running* というスティミュラスに対するレスポンスを追加します。

- ***Motor1Running*** というスティミュラスと ***Lamp2On*** というレスポンスを選択します。
- **Implementation** ボタンをクリックし、優先度 ***9*** の ***MotorResponse*** という新しいタスクを追加します。
- レスポンス ***Lamp1Off*** を選択します。
- **Implementation** ボタンをクリックし、タスク ***MotorResponse*** を選択します。

するとワークスペース上の Stimulus Summary には、3 つのスティミュラスとそれぞれのプライマリプロファイルが表示されます。これで、アプリケーションの設定は終わりです。

図 3-58 Stimulus Summary 内に表示されたプライマリプロファイルとレスポンスについての情報

タスクと ISR のコードを記述する

今度は、*PrimaryISR* という ISR と、*Button1Response*、*MotorStart*、*LampToggle*、*MotorResponse* というタスク、そしてアプリケーションの `main()` 関数（アプリケーションの起動とアイドルメカニズムが含まれる関数）のコードを記述する必要があります。

これらの C ソースコードは、外部のエディタで記述することも可能ですが、RTA-OSEK GUI 内で作成されたテンプレートを使用することもできます。これは、以下のようにして行います。

- ビルダ（*Builder* タブ）に切り替え、ナビゲーションバーから **Custom Build** グループを選択します。
- ワークスペースの **Create Templates** ボタンをクリックします。すると RTA-OSEK は 7 つの C ソースファイルを生成し、そこにスケルトンコードを挿入します。またさらに、ビルド工程で使用する `rtkbuild.bat` というバッチファイルも生成します。

PrimaryISR のコードを記述する

ナビゲーションバーから **ISRs** グループを選択します。そして **Category 2 ISRs** サブグループを選択し、ワークスペースから ***PrimaryISR*** を選択します。

ISR 内で行うべき処理が不明な場合でも、RTA-OSEK GUI によって必要な事項が提示されるので、容易にコーディングを行えます。メニューから **View → Implementation** を選択すれば、ウィンドウの下の部分に、ISR の「実装ノート」が開き、実装に関する詳細情報が表示されます。このエリアのサイズは、青色のスプリットバーをマウスの左ボタンで上下にドラッグすることにより任意に調整できます。

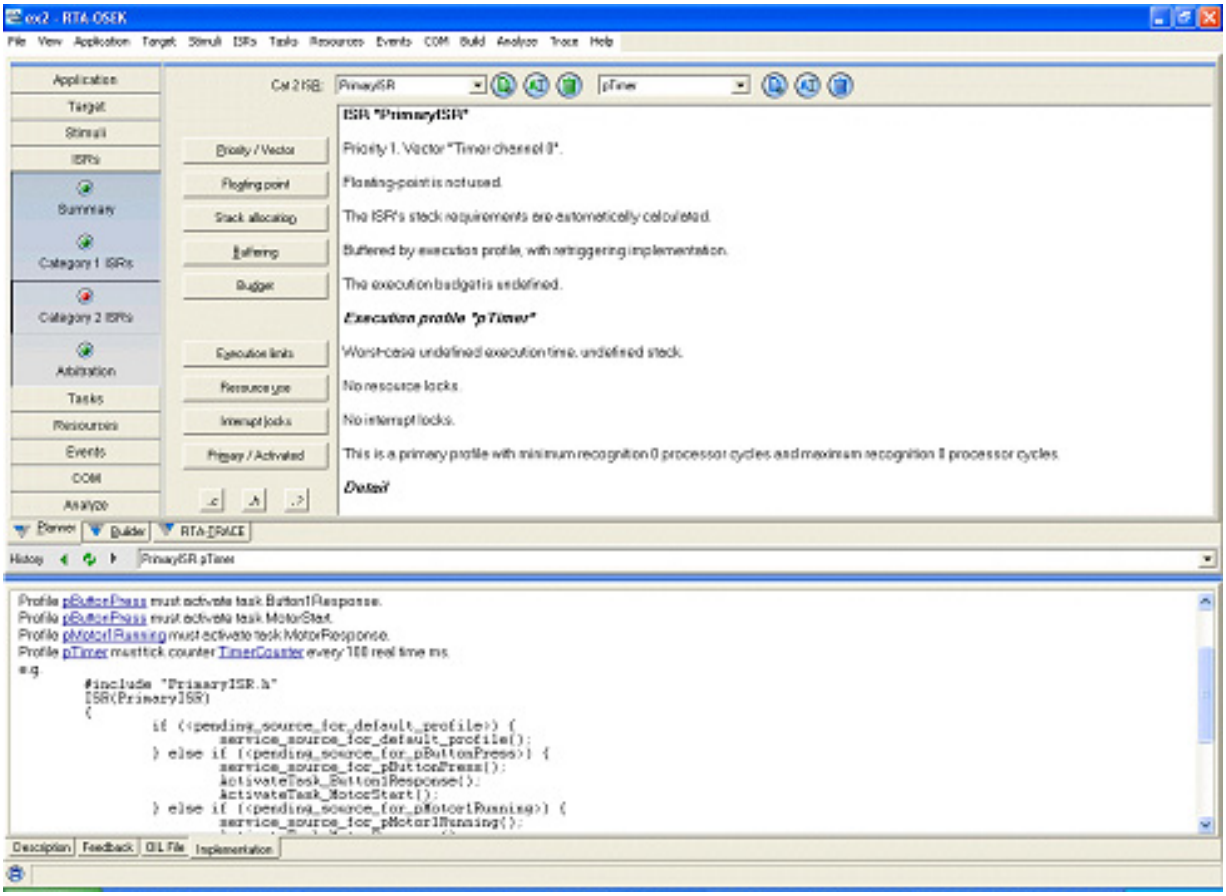


図 3-59 PrimaryISR の実装ノートを表示する

図 3-59 に示されたサンプルコードを見てみると、ISR 専用のヘッダファイル `PrimaryISR.h` がインクルードされ、その後に ISR のボディが続いています。そこでは 3 つの実行プロファイルが `if...else...else` 構文を使用した 3 つの経路でコード化されています。

レスポンスを発生させるタスクが各実行プロファイルで起動され、カウンタ *TimerCounter* は *pTimer* プロファイルでカウントされることに注目してください。

再度 **View → Implementation** を選択してこのメニューアイテムのチェックマークを消すと、実装ノートが閉じます。

重要

ユーザーシステムのコントロールのフローは、RTA-OSEK GUI が示すとおりに正確に実装する必要があります。正確な実装が行われない場合、ユーザーシステムは RTA-OSEK GUI が後でタイミング分析を行うために使用するシステムと異なるタイミング特性を持つようになってしまいます。特に、割込みソースのチェック順序を並べ替えないでください。また、ISR 内で、ペンディングされている割込みがないかどうかを調べるためにループバックを行うには、必ずその ISR が「ループピング」挙動を持つことを指定してください。

Category 2 ISRs ワークスペースの  ボタンを選択すると、ISR のソースコードを直接編集することができます。

重要

RTA-OSEK GUI 内でファイルを編集する場合、Windows Notepad がデフォルトエディタとして設定されていますが、**Files → Options** を選択して、他のエディタを指定することができます。

移植性

ここで記述するコードの内容はターゲットに依存するものであるため、ペンディング状態の割込みソースを検知したり判定する部分のコードについては、ここでは特に説明しません。

Button1Response のコードを記述する

ナビゲーションバーの **Tasks** グループから **Task Data** サブグループを選択し、さらにタスク ***Button1Response*** を選択します。

Implementation View（実装ビュー）で、このタスクに必要なコードを確認してください。

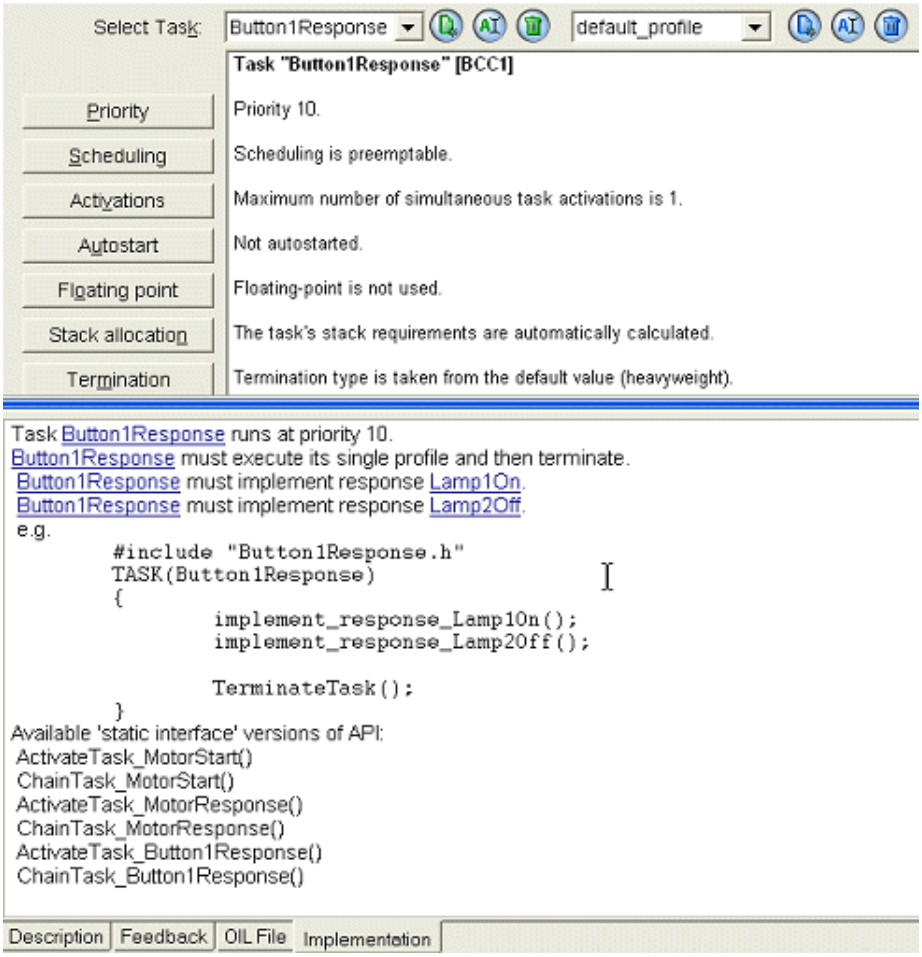


図 3-60 Button1Response の実装ノートを表示する

ワークスペースの .c ボタンをクリックすると、タスクのソースコードを直接編集することができます。ここで記述するコードはターゲット固有のものですが、実装ビューの内容を参考にして記述してください。

その他のタスクのコードを記述する

残りのタスク（*LampToggle*、*MotorResponse*、*Motor1On*）のコードも、同じ要領で記述します。

RTA-OSEK の設定を変更する場合は、必ず、提示された実装内容どおりにコードが記述されているかを確認してください。

'main' のコードを記述する

C プログラムでは、`main()` がメインアプリケーションの開始点になります。これは、低レベルの起動時初期設定の後に呼び出されます。通常、`main()` に入る前は割込みはディセーブルになっています。

コード例 3-6 は、RTA-OSEK が `main()` 用に生成するスケルトンコードです。

```
/* Template code for 'main' in project: UserApp */

#include "osekmain.h"

OS_MAIN()
{
   StartOS(OSDEFAULTAPPMODE);
   ShutdownOS(E_OK);
}
```

コード例 3-6 main( ) のテンプレートコード

ここで、main() ではなく OS_MAIN() というマクロが使用されていることに注意してください。main( ) に使用できる引数と戻り値タイプはコンパイラにより異なるので、RTA-OSEK では、移植しやすいように OS_MAIN( ) を使用しています。

移植性

アプリケーションで main( ) ではなく OS_MAIN( ) を使用すると、異なるターゲットへの移植性が高まります。

オペレーティングシステムを起動するには、StartOS(OSDEFAULTAPPMODE) を呼び出します。StartOS() が呼び出される前には、OS の API コールを一切行わないでください。

ShutdownOS() は、アプリケーションの終了時に OS を終了させるために使用されます。ShutdownOS() のデフォルトアクションは無限ループを続け、コントロールを返さないようになっています。これによってアプリケーションは永久に、つまりプロセッサへの電力供給が停止されるか、またはプロセッサがリセットされるまで無限ループを続けてしまうことになるので、この関数は通常は使用されません。

サンプルアプリケーションにおいては、StartOS() を呼び出す前にターゲットハードウェアの初期設定を行う必要があります。ここでは、タイマを設定して 100ms ごとに割込みがかかるようにし、さらに各割込みソースをイネーブルにします。そして StartOS() を呼び出した後にアラームを有効にし、アイドルループに入るようにしてください。

コード実行時において、StartOS() の後に実行されるコードはアイドルタスクに属します。このアイドルタスクは osek_idle_task という名前です。

このアイドルタスクは他のタスクと同様に、API コール、リソースの使用、メッセージの送受信、イベントの送信や待機など、処理を行うことができます。ただし、このタスクは ShutdownOS() が呼び出されたときにしかターミネートしないので、直接起動することはできません。また、他のタスクの起動を妨げてしまうので、内部リソースを使用することもできません。

重要

アイドルタスク内にコードを記述することは、システム実装において非常に効率的な方法といえます。特に、OSEK イベントに応答する必要があるタスクが 1 つしかない場合は、osek_idle_task を使用し、このアイドルタスクがイベントを待つようにすると、ユーザーのアプリケーションが大幅にサイズダウンし、応答速度が上がります。

RTA-OSEK GUI は典型的な OS_MAIN() の実装内容を提示します。ナビゲーションバーで、**Tasks** グループの **Task Data** サブグループ内の ***osek_idle_task*** タスクを選択し、実装内容を確認してください。

ここではアイドルタスクは何も処理を行いませんが、「スリープ」ステートをサポートするターゲットでは、アイドルタスク内でプロセッサを「スリープ」ステートにすることができます。その場合、プロセッサは割込み発生によって「ウェイクアップ」します。

重要

アイドルタスクは絶対にターミネートせずに、無限ループを行わなければなりません。

タイマ／カウンタハードウェアをセットアップする

コード例 3-7 の関数 do_target_initialization() は、割込みソースを初期化します。これらのソースの 1 つが、100ms ごとに割込みを発生させるハードウェアカウンタ／タイマです。

コード例 3-7 の内容を参考にしてください。

```
void do_target_initialization(void)
{

   unsigned int timer_divide;
   timer_divide = OSTICKDURATION_TimerCounter / OS_NS_PER_CYCLE;

   /* Target specific setup provided by user */
   SetupTimer(timer_divide);
   EnableTimerInterrupt();
   EnableKeyPressInterrupt();

   /* Set up Button1 and Motor1 interrupts. */
   ...
}
```

コード例 3-7 タイマハードウェアの初期化

コード例 3-7 は、RTA-OSEK が生成した `OSTICKDURATION_TimerCounter` と `OS_NS_PER_CYCLE` という 2 つの定数を用いる初期化処理を示しています。

`OSTICKDURATION_TimerCounter` 定数はカウンタのチック（1 回のカウント）の間隔をナノ秒（ns）単位で定義するものです。1 ナノ秒は 1 秒の 10 億分の 1 なので、この例では `OSTICKDURATION` は 1 億 ns（1 秒の 10 分の 1）です。

`OS_NS_PER_CYCLE` 定数は、CPU 命令サイクルの長さを ns 単位で定義します。10MHz の CPU の場合は 100ns になります。

この例では、1,000,000 命令サイクルごとに割込みをかけるようにタイマを設定してください。これらの定数を使用して分周率を算出しておけば、クロックレートが変わった場合でもコードが自動的に調整を行います。

OS ステータス、エラーフック、コールバック

初期のテストにおいては、オペレーティングシステムの **Extended**（拡張）ビルドを使用してアプリケーションを実行してください。拡張ビルドは、OS が各 API コールについて厳密なチェックを行うことを意味しています。もちろん、これには時間とコードスペースがかかります。

初期テストにおいてアプリケーションが正しく機能していると判断されたら、通常は **Standard**（標準）ビルドに切り替えてください。標準ビルドではチェックはほとんど行われないので、OS は拡張ビルドの場合よりもはるかに効率的に実行されます。

拡張ビルドを使用しているときには、各 API コールからのリターンステータスコードを調べたり、エラーフックを使用するように要求することができます。エラーフックは、エラーが検知されたときに OS が必ず呼び出す関数で、これはユーザーのアプリケーション内に実装します。通常、この関数は、デバッグを中断させ、エラーをユーザーに警告する処理を実行します。

エラーフックを使用するには、以下の手順を実行してください。

- ナビゲーションバーから **Application** グループを選択してから、**OS Configuration** サブグループを選択します。
- **OS Configuration Summary** ワークスペースの **Hooks** ボタンをクリックします。**Select Hooks** ダイアログボックスが開きます。
- **Error Hook** チェックボックスを選択してから **OK** ボタンをクリックします。

図 3-61 は、エラーフックを選択した状態を示しています。

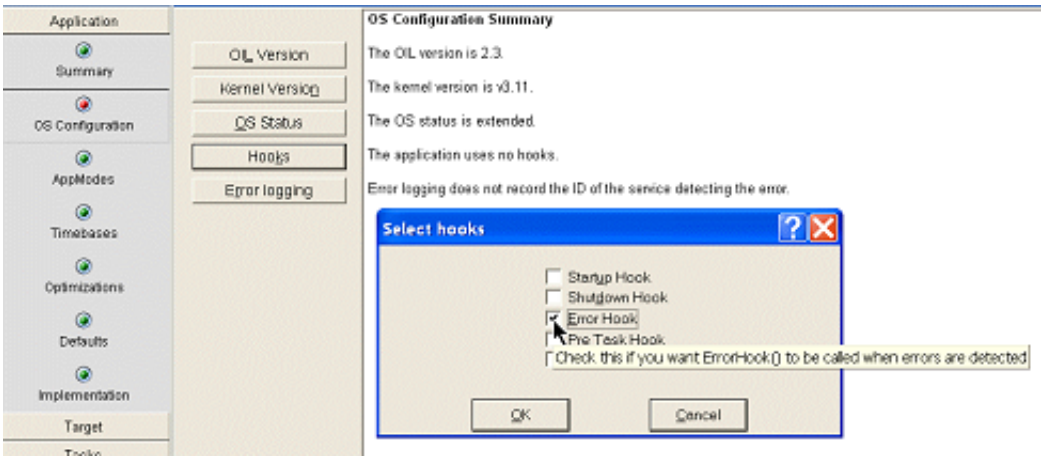


図 3-61 エラーフックを選択する

`ErrorHook()` を実装するために必要なコードは、どのソースファイルに含めることもできますが、最初は `main.c` を使用することをお勧めします。

以下のコードを追加してください。

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType e)
{
   /* Put a debugger breakpoint here. */
   while (1) {
      /* Freeze. */
   }
}

#endif /* OSEK_ERRORHOOK */
```

コード例 3-8 ErrorHook( )

デバッグ時の `ErrorHook()` の使用方法については、本書の第 16 章に詳しく説明されています。

他に、タイミングビルドまたは拡張ビルドを使用する場合に用意しなければならない関数が 3 つあります。オペレーティングシステムはこれらを使用してユーザーのコード実行のタイミングを管理します。

現時点では詳細について考える必要はないので、コード例 3-9 の内容をそのまま `ErrorHook()` の後に追加してください。

```
#ifdef OS_ET_MEASURE
   OS_HOOK(void) OverrunHook(void)
   {
      /* Put a debugger breakpoint here. */
      while (1) {
         /* Freeze. */
      }
   }
   OS_NONREENTRANT(StopwatchTickType) GetStopwatch(void)
   {
      /* Temporary implementation. A correct solution
       * returns the current stopwatch value. */
      return 0;
   }
```

```
OS_NONREENTRANT(StopwatchTickType)
GetStopwatchUncertainty(void)
{
   /* Temporary implementation. A correct solution
    * returns the uncertainty in the stopwatch value. */
   return 0;
}
#endif /* OS_ET_MEASURE */
```

コード例 3-9 タイミングコールバック

最終チェック

ナビゲーションバーの **Application** グループから **Implementation** サブグループを選択して、全体の実装情報を表示してください。

この情報は、アプリケーションが完全に実装されたことを確認するためのチェックリストとして使用できます。メニューから **File → Print Selection** を選択すれば、この情報を印刷することができます。

3.3.3 ビルド

このサンプルアプリケーション作成の全ステップを完了したら、ビルド工程を開始します。*Builder* に戻り切り替え、3.6.2 項を参照してビルドを実行してください。

3.3.4 機能テスト

実行可能ファイルをターゲットハードウェアにダウンロードして、その挙動を確認します。

最初は、必ずエラーフックを含んだ拡張ビルドを行い、API コールの誤用があれば OS がそれを検知するようにしてください。アプリケーションが正しく実行できた後に、タイミングビルドまたは標準ビルドに切り替えてください。

3.3.5 分析

通常のテストにおいてアプリケーションが一見は正しく機能しているように見えても、それだけでは、実際にアプリケーションが毎回すべてのデッドラインを守って動作しているかを判断することはできません。通常のテストをたとえ何千時間もかけて行ったとしても「百万回に 1 回」のミスタイミングを見つけ出すことは現実的に不可能です。しかし「タイミング分析」を行えば、デッドラインに間に合わなくなるような非常にまれな状況が発生しないかどうかを検証することができます。

本書の後の部分で、タスクと ISR の実行プロファイルの実行時間を測定する方法について説明されていますが、このタイミング分析がいかに簡単なものであるかをおわかりいただくために、ここでは「仮の」実行時間を使用してタイミング分析を行ってみます。

以下の「仮の」時間を使用します。

- *PrimaryISR.pButtonPress* という ISR の実行時間は、1,000 プロセッササイクルです。
- *PrimaryISR.pMotor1Running* という ISR の実行時間は、1,500 プロセッササイクルです。
- *PrimaryISR.pTimer* という ISR の実行時間は、2,000 プロセッササイクルです。
- *LampToggle* というタスクの実行時間は、8,000 プロセッササイクルです。このタスクは 7,000 プロセッササイクルが経過した時点で *Lamp3* をオンにします。
- *Button1Response* というタスクの実行時間は、20,000 プロセッササイクルです。このタスクは 12,000 プロセッササイクルが経過した時点で *Lamp1* をオンにし、16,000 プロセッササイクルが経過した時点で *Lamp2* をオフにします。
- *MotorResponse* というタスクの実行時間は、20,000 プロセッササイクルです。このタスクは 10,000 プロセッササイクルが経過した時点で *Lamp2* をオンにし、14,000 プロセッササイクルが経過した時点で *Lamp1* をオフにします。
- *MotorStart* というタスクの実行時間は、40,000 プロセッササイクルです。このタスクは 30,000 プロセッササイクルが経過した時点で電力を供給します。

スティミュラスとデッドラインを指定する際は、秒やミリ秒という実時間の単位を使用しましたが、ここではプロセッササイクルを単位として実行時間を表わしていることに注意してください。RTA-OSEK は、「プロセッサのクロック周波数が 2 倍になれば、実行時間は半分になっても、スティミュラスとデッドラインのタイミングは変わらない」ということを認識しています。「時間」を入力するときには、適切な単位を使用するように注意してください。

システムによっては RTA-OSEK ではタイミング分析を行えない場合がありますが、そのような問題が発生するのは極めて稀です。

分析可能なシステムとするためには、以下の 3 つの簡単なルールを守る必要があります。

- タスクが自分より優先度の高いタスクを起動しないこと

  一般的に、適切に設計されたリアルタイムシステムにおいては、タスクは ISR により起動されます。ISR はスティミュラスに対応し、レスポンスを発生させる処理を行う 1 つ以上のタスクを起動します。この「レスポンスタスク」は、その作業を自分より優先度の低いタスクに渡す場合はありますが、自分より優先度の高いタスクに渡さなければならないような状況はほとんどありません。一部の処理を「高い優先度」で行う必要がある場合は、OSEK リソースを用いる方が効率的だからです。

- 各タスクにそれぞれ異なる優先度が割り当てられていること

  OSEK システムにおいては、複数のタスクが相互排他的に実行されるようにするために、1 つのタスク優先度を複数のタスクで共有する場合がありますが、多くの場合は、タスクごとに異なるタスク優先度を設定し、内部リソースを使用してより確実な相互排他を行います。実際、「共有優先度」を使用しないシステムとして RTA-OSEK OS を実装すれば、各共有優先度ごとにタスク用の FIFO キューを管理する必要がなくなるため、より高い効率性を実現できます。

- `Schedule()` という OSEK API を使用しないこと

  `Schedule()` を呼び出すと、すべての内部タスクリソースが解放されますが、タイミング分析においてこれを検知することができません。なお、RTA-OSEK GUI を使ってビルドするアプリケーションにおいて `Schedule()` が必要になることはほとんどありません。

重要

拡張タスク、およびいずれかの拡張タスクよりも優先度の低い基本タスクのレスポンスタイムの分析は、行えません。

RTA-OSEK がシステムのタイミング分析を行うには、以下の手順を実行して、アプリケーションがこれらのルールに従っていることを RTA-OSEK に伝える必要があります。

- ナビゲーションバーから **Application** グループを選択し、**Optimizations** サブグループを選択します。
- **No Upward Activation**、**Unique Task Priorities** および **Disallow Schedule** というオプションを有効にします。また、RES_SCHEDULER という標準 OSEK リソースは不要なので、**No RES_SCHEDULER** も有効にしておいてください。これが使用されると、分析中に不必要なワーニングメッセージが表示される原因となります。

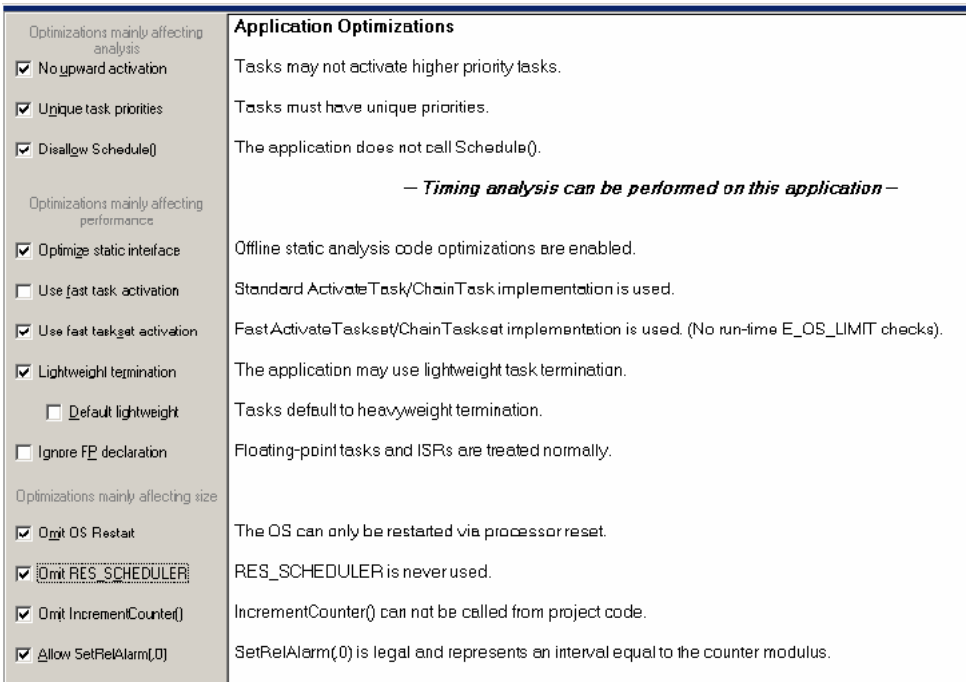


図 3-62 アプリケーションの最適化オプションを指定する

実行時間を入力する

PrimaryISR という ISR の実行時間を入力するには、以下の手順を実行してください。

- *Planner* のナビゲーションバーから **ISRs** グループを選択します。
- ナビゲーションバーから **Category 2 ISRs** サブグループを選択します。
- ***PrimaryISR*** という ISR を選択してから、***pButtonPress*** というプロファイルを選択します。**Execution Limits** ボタンをクリックし、ワーストケースの実行時間として、***1000 processor cycles*** を設定します（図 3-63 を参照してください）。必要であれば、使用するスタックの量もこの実行プロファイルに定義できます。

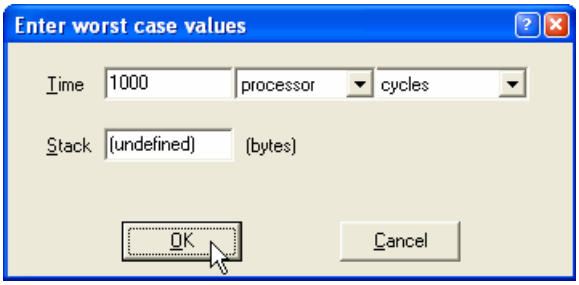


図 3-63 pButtonPress のワーストケース値を入力する

- ***pMotor1Running*** というプロファイルを選択し、ワーストケースの実行時間に ***1500 processor cycles*** を設定します。
- ***pTimer*** というプロファイルを選択し、そのワーストケースの実行時間に ***2000 processor cycles*** を設定します。

LampToggle というタスクについて、以下の手順を実行してください。

- ナビゲーションバーの **Tasks** グループから、**Task Data** サブグループを選択します。
- ***LampToggle*** というタスクを選択し、その実行限界として ***8000 processor cycles*** を設定します。
- ナビゲーションバーの **Stimuli** グループを選択し、**Stimuli** サブグループを選択します。ドロップダウンリストから ***Lamp3Toggle*** というスティミュラスを選択します。

- **Implementation** ボタンをクリックし、実行時間として ***7000 processor cycles*** を設定します。これは、このタスク内のコードがタスク終了の少し前にライトのオンとオフを切り替える処理を行うことを表わしています。

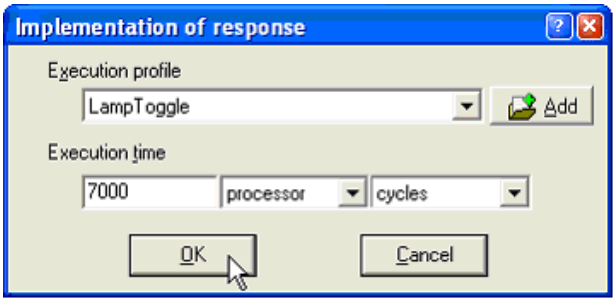


図 3-64 Lamp3Toggle の実行時間を指定する

Button1Response というタスクについて、以下の手順を実行してください。

- ナビゲーションバーの **Tasks** グループから、**Task Data** サブグループを選択します。
- ***Button1Response*** というタスクを選択し、そのワーストケースの実行時間に **20000 processor cycles** を設定します。
- ナビゲーションバーの **Stimuli** グループを選択し、***Button1Press*** というスティミュラスを選択します。
- ***Lamp1On*** というレスポンスを選択し、実装実行時間として ***12000 processor cycles*** を設定します。
- ***Lamp2Off*** というレスポンスを選択し、実装実行時間として ***16000 processor cycles*** を設定します。

MotorResponse というタスクについて、以下の手順を実行してください。

- ナビゲーションバーの **Tasks** グループから、**Task Data** を選択します。
- ***MotorResponse*** というタスクを選択し、そのワーストケースの実行時間として ***20000 processor cycles*** を設定します。
- ナビゲーションバーの **Stimuli** グループを選択し、***Motor1Running*** というスティミュラスを選択します。
- ***Lamp2On*** というレスポンスを選択し、実装実行時間として ***10000 processor cycles*** を設定します。
- ***Lamp1Off*** というレスポンスを選択し、実装実行時間として ***14000 processor cycles*** を設定します。

MotorStart というタスクについて、以下の手順を実行してください。

- ナビゲーションバーの **Tasks** グループから、**Task Data** を選択します。
- ***MotorStart*** というタスクを選択し、そのワーストケースの実行時間として ***40000 processor cycles*** を設定します。
- ナビゲーションバーの **Stimuli** グループを選択し、***Button1Press*** というスティミュラスを選択します。
- ***Motor1On*** というレスポンスを選択し、実装実行時間として ***30000 processor cycles*** を設定します。

スケジューラビリティ分析を行う

アプリケーションの準備ができたので、以下の手順でタイミング分析を行います。

- ナビゲーションバーの **Analyze** グループから、**Schedulability** サブグループを選択します。

分析結果がワークスペースに表示されます。

Schedulability Analysis

Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.

Creating files

Analysis

*** Schedulability Analysis results ***

task MotorStart is schedulable.
 Calculated response time on MotorStart.default_profile for response Button1Press.Motor1On is 485700 cycles (60.7125 ms).
 Calculated response time on MotorStart.default_profile is 95700 cycles (11.9625 ms), with blocking 0 cycles.
task MotorResponse is schedulable.
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp2On is 46500 cycles (5.8125 ms).
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp1Off is 48900 cycles (6.1125 ms).
 Calculated response time on MotorResponse.default_profile is 52500 cycles (6.5625 ms), with blocking 0 cycles.
task Button1Response is schedulable.
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp1On is 27700 cycles (3.4625 ms).
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp2Off is 34100 cycles (4.2625 ms).
 Calculated response time on Button1Response.default_profile is 35700 cycles (4.4625 ms), with blocking 0 cycles.
task LampToggle is schedulable.
 Calculated response time on LampToggle.default_profile is 12500 cycles (1.5625 ms), with blocking 0 cycles.
interrupt primaryISR is schedulable.
 Calculated response time on primaryISR.pButtonPress is 6200 cycles (775 us), with blocking 2000 cycles (250 us), caused by IST primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pButtonPress is 1.
 Calculated response time on primaryISR.pMotor1Running is 4500 cycles (562.5 us), with blocking 2000 cycles (250 us), caused by IST primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pMotor1Running is 1.
 Calculated response time on primaryISR.pTimer is 4500 cycles (562.5 us), with blocking 0 cycles.
 Maximum buffer required on primaryISR.pTimer is 1.
 Maximum retriggers is 3.

The system is schedulable.

図 3-65 タイミング分析結果（テキスト表示）

画面の一番下にある **Text/Graphic** というタブをクリックすると、結果のテキスト表示とグラフィック表示を切り替えることができます。（グラフィックを右クリックすると、拡大／縮小オプションを利用できます。）

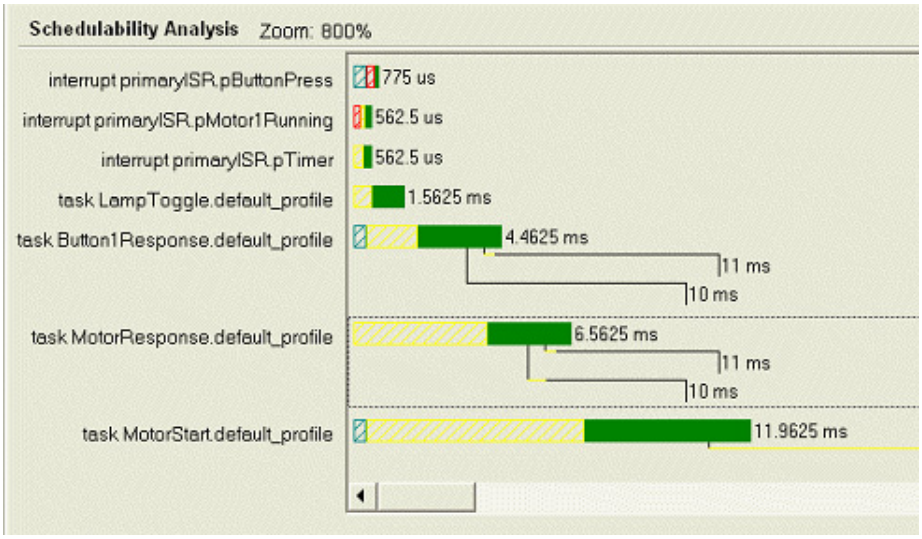


図 3-66 タイミング分析結果（グラフィック表示）

図 3-66 は、このサンプルシステムがスケジューラブルであることを示しています。
分析結果から、以下のことがわかります。

- *PrimaryISR* というISRのプロファイル*pButtonPress*は必ず、Button1押下後775µs以内に実行されて終了します。この時間には、ボタンのデバウンス遅延として最大 400µs、*pMotor1Running* または *pTimer* の実行によりプロファイル開始が妨げられる可能性がある時間として最大 250µs、さらに *pButtonPress* 自体の実行時間として 125µs が含まれています。
- *PrimaryISR* という ISR のプロファイル *pMotor1Running* は必ず、モータが完全作動状態になった後 562.5µs 以内に実行されて終了します。この時間には、*pTimer* の実行によりプロファイル開始が妨げられる可能性がある時間として最大 250µs、*pButtonPress* がプロファイルの実行を妨げる（これは両方の割込みソースが同時にレディ状態になり、*pButtonPress* が先に処理された場合に起こります）可能性がある時間として 125µs、さらに *pMotor1Running* 自体の実行時間として 187.5µs が含まれています。
- *PrimaryISR* という ISR のプロファイル *pTimer* は必ず、タイマ割込み発生後 562.5µs 以内に実行されて終了します。この時間には、*pButtonPress* または *pMotor1Running* の実行によりプロファイル開始が妨げられる可能性がある時間として最大 312.5µs、*pTimer* 自体の実行時間として 250µs が含まれています。
- *LampToggle* というタスクは、タイマ割込み発生後必ず 1.5625ms 以内に終了します。この時間には、ISR に実行を妨げられる時間として最大 562.5µs と、自分自身の実行時間 1ms が含まれています。分析結果を見ると、4ms のデッドラインに対して 1.9375ms を残して対応できていることがわかります。なぜなら、このタスクは 1.5625ms までにオンオフ切り替え命令を発行し、その後 0.5ms のレスポンス遅延があるので、レスポンスタイムの合計が 2.0625 となるためです。
- *Button1Response* というタスクは 4.4625ms 以内に完了するので、デッドラインに間に合っています。
- *MotorResponse* というタスクは 6.5625ms 以内に完了するので、デッドラインに間に合っています。
- *MotorStart* というタスクは 11.9625ms 以内に完了するので、デッドラインに間に合っています。

実行時間とデッドラインを調整し、その影響を分析してみてください。

割込みの認識とシステムのタイミング

実際のシステムの分析においては、何セクションもある OS コードの実行時間を考慮する必要があります。より正確な分析を行うためには、割込みの認識やさまざまなシステムタイミングも考慮に入れてください。

これらの詳細と計算方法については後で説明します。

センシティビティ分析を行う

センシティビティ分析は、システムのスケジューラビリティの限界を決定するためのものです。センシティビティ分析では、スケジューラブルシステム、つまりシステムの正しいスケジューリングが行われる範囲において、あるパラメータが取りうる最大値が求められます。

センシティビティ分析を行うには、以下の手順を実行してください。

- ナビゲーションバーの **Analyze** グループから **Sensitivity** サブグループを選択します。

分析結果はワークスペースに表示されます。

Sensitivity Analysis

Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.

Creating files

Analysis

*** Sensitivity Analysis results ***

--- Deadline sensitivity
In task MotorStart.default_profile, the deadline for response Button1Press.Motor1On can be met for execution time up to 40000 cycles (5 ms
In task MotorResponse.default_profile, the deadline for response Motor1Running.Lamp2On can be met for execution time up to 20000 cycles
(2.5 ms).
In task MotorResponse.default_profile, the deadline for response Motor1Running.Lamp1Off can be met for execution time up to 20000 cycles
(2.5 ms).
In task Button1Response.default_profile, the deadline for response Button1Press.Lamp1On can be met for execution time up to 20000 cycles
(2.5 ms).
In task Button1Response.default_profile, the deadline for response Button1Press.Lamp2Off can be met for execution time up to 20000 cycles
(2.5 ms).

--- System sensitivity to execution and lock times
In task MotorStart.default_profile, the system can be schedulable for execution time up to 745100 cycles (93.1375 ms).
In task MotorResponse.default_profile, the system can be schedulable for execution time up to 725100 cycles (90.6375 ms).
In task Button1Response.default_profile, the system can be schedulable for execution time up to 53500 cycles (6.6875 ms).
In task LampToggle.default_profile, the system can be schedulable for execution time up to 41500 cycles (5.1875 ms).
In interrupt primaryISR.pButtonPress, the system can be schedulable for execution time up to 34500 cycles (4.3125 ms).
In interrupt primaryISR.pMotor1Running, the system can be schedulable for execution time up to 35000 cycles (4.375 ms).
In interrupt primaryISR.pTimer, the system can be schedulable for execution time up to 35500 cycles (4.4375 ms).

--- System sensitivity to clock speed
The system remains schedulable if processor clock speed is reduced to 58.13% of its current value.

図 3-67 センシティビティ分析結果（テキスト表示）

分析結果はグラフィックで表示することもできます。

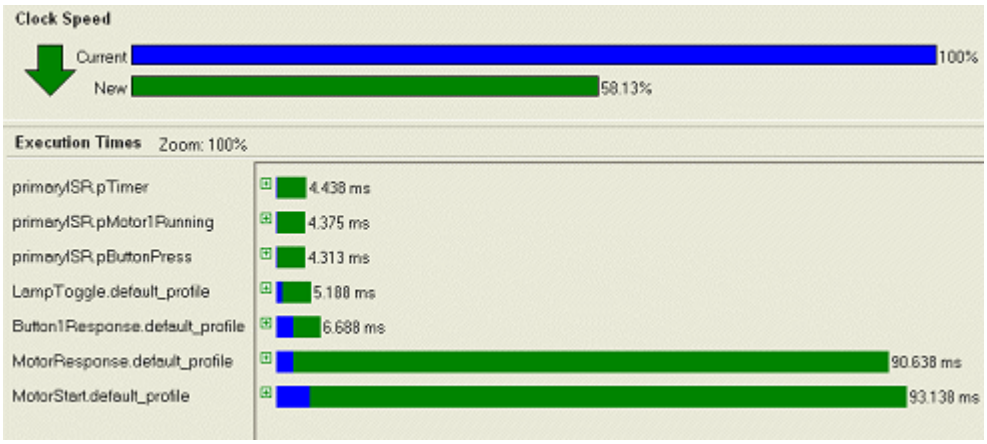


図 3-68 センシティビティ分析結果（グラフィック表示）

分析結果から、以下のことがわかります。

- タスク *Button1Response* 内の *Lamp1* をオンにし *Lamp2* をオフにするコードの実行時間は、最長で 20000 プロセッササイクルまで許容され、この範囲でシステムはスケジューラブルな状態を保ちます。つまり、このオン／オフ切り替え処理はタスク内の最後の命令で実行することが可能です。実際には、このクロック速度においては、システム内のすべての「クリティカルな処理」の実行を、当該タスクの最後の命令まで延ばすことができます。
- タスク *Button1Response* の実行には、最長 6.688ms まで許容されます。
- *MotorStart* の実行には最長 93.138ms まで許容されます。
- *MotorResponse* の実行には最長 90.638ms まで許容されます。
- *LampToggle* の実行には最長 5.188ms まで許容されます。

- *PrimaryISR* 内の実行プロファイルの実行時間はそれぞれ最長 2.188ms、2.25ms、2.313ms まで許容されます。
- CPU クロックは、宣言されている値の 58.13% まで落とすことができます。それによってプロセッサに必要な電力はおよそ半分になります。

これらの値は、それぞれの項目を単独で考慮した場合の値です。つまり、これらの結果すべてを適用してしまうと、システムがスケジューラブルな状態を保つことは保証されません。

最良のタスク優先度を算出する

「最良タスク優先度」の算出は、システムをスケジューラブルに保ちながらタスクのプリエンプションの回数を減らして、スタック使用量を削減するためのもので、各タスクの理想的な優先度と使用すべき内部リソースを提示します。以下の手順を実行してください。

- ナビゲーションバーの **Analyze** グループから **Best Task Priorities** を選択します。

結果がワークスペースに表示されます。

```
*** Priority Allocation results ***

Task MotorStart is schedulable at priority level 2.
Task MotorResponse is schedulable at priority level 3.
Task Button1Response is schedulable at priority level 4.
Task LampToggle is schedulable at priority level 1.
Tasks LampToggle, MotorStart, MotorResponse, Button1Response must not preempt each other.

*** Schedulability Analysis results ***

task MotorStart is schedulable.
 Calculated response time on MotorStart.default_profile for response Button1Press.Motor1On is 485700 cycles (60.7125 ms).
 Calculated response time on MotorStart.default_profile is 95700 cycles (11.9625 ms), with blocking 8000 cycles (1 ms), caused by task
LampToggle.default_profile executing at its dispatch priority.
task MotorResponse is schedulable.
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp2On is 78500 cycles (9.8125 ms).
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp1Off is 80900 cycles (10.1125 ms).
 Calculated response time on MotorResponse.default_profile is 84500 cycles (10.5625 ms), with blocking 40000 cycles (5 ms), caused by
task MotorStart.default_profile executing at its dispatch priority.
task Button1Response is schedulable.
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp1On is 39700 cycles (4.9625 ms).
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp2Off is 46100 cycles (5.7625 ms).
 Calculated response time on Button1Response.default_profile is 47700 cycles (5.9625 ms), with blocking 20000 cycles (2.5 ms), caused
task MotorResponse.default_profile executing at its dispatch priority.
task LampToggle is schedulable.
 Calculated response time on LampToggle.default_profile is 92500 cycles (11.5625 ms), with blocking 0 cycles.
interrupt primaryISR is schedulable.
 Calculated response time on primaryISR.pButtonPress is 6200 cycles (775 us), with blocking 2000 cycles (250 us), caused by IST
primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pButtonPress is 1.
 Calculated response time on primaryISR.pMotor1Running is 4500 cycles (562.5 us), with blocking 2000 cycles (250 us), caused by IST
primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pMotor1Running is 1.
 Calculated response time on primaryISR.pTimer is 4500 cycles (562.5 us), with blocking 0 cycles.
 Maximum buffer required on primaryISR.pTimer is 1.
 Maximum retriggers is 3.

The system is schedulable.
```

図 3-69 優先度分析結果（テキスト表示）

結果はグラフィック表示することもできます。

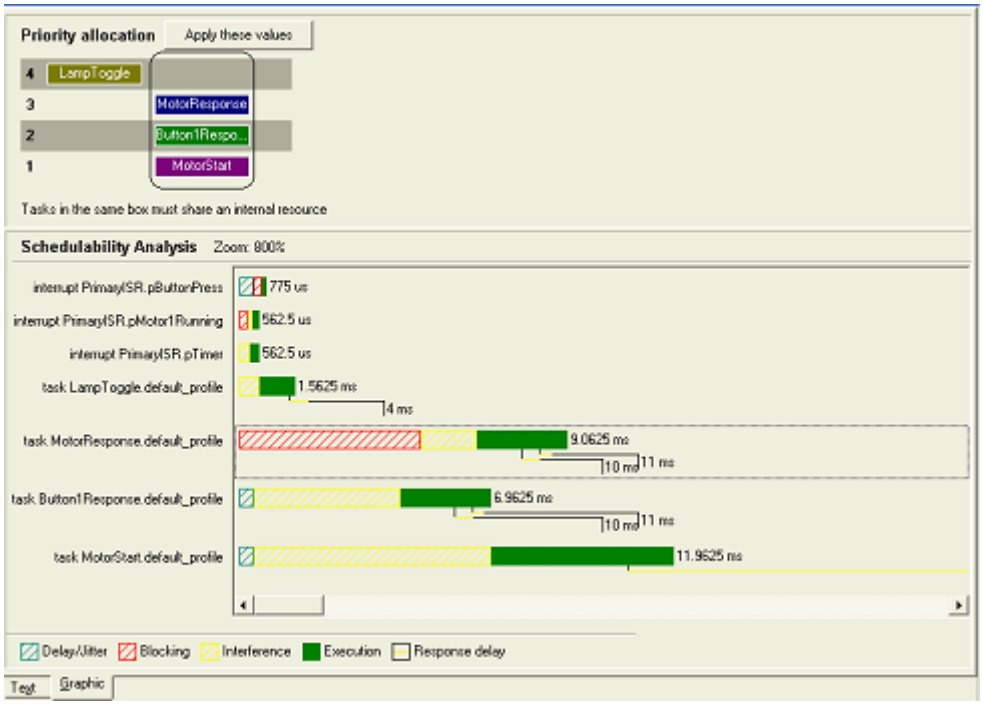


図 3-70 優先度分析結果（グラフィック表示）

この場合、タスク優先度を変更してタスク *MotorStart*、*Button1Response*、*MotorResponse* が互いにプリエンプトしないようにすることにより、プリエンプションの回数を減らすことができます。これは、当該タスクを内部リソースに割り当てることにより実現できます。

これらの条件が実際に適用されると個々のレスポンスタイムが変わりますが、システムはスケジューラブルな状態を保ちます。

CPU クロックレートを計算する

CPU クロックレート分析では、システムをスケジューラブルのまま保ちながら CPU クロックレートを低減しようとし、このクロックレートを実現するための各タスクの理想的な優先度を提案します。以下の手順を実行してください。

- ナビゲーションバーの **Analyze** グループから **CPU Clock Rate** を選択します。

結果がワークスペースに表示されます。

```
Clock Rate Analysis

Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.

Creating files

Analysis

*** Clock Optimization results ***

The system is schedulable if processor clock speed is reduced to 59% of its current value based on the following task priorities.

1 schedulable solution found. Current minimum 4 preemption levels.
Task MotorStart is schedulable at priority level 1.
Task MotorResponse is schedulable at priority level 2.
Task Button1Response is schedulable at priority level 3.
Task LampToggle is schedulable at priority level 4.
```

図 3-71 CPU クロックレート分析結果（テキスト表示）

結果はグラフィック表示することもできます。

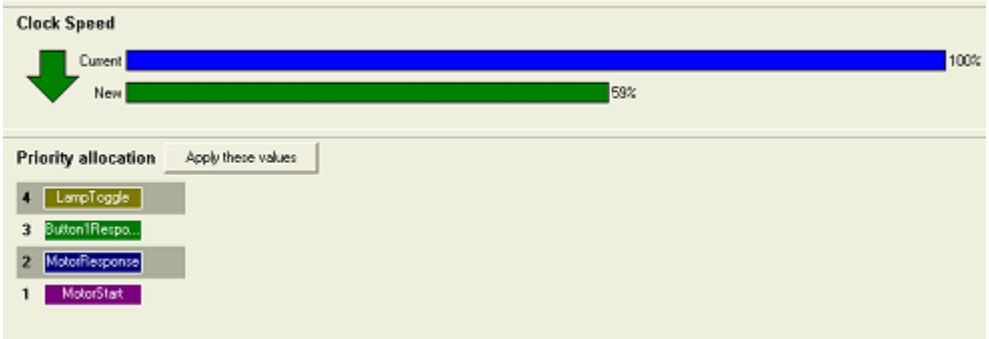


図 3-72 CPU クロックレート分析結果（グラフィック表示）

3.4 サンプルアプリケーションの終了

ここまでで、サンプルアプリケーションに関する一連の作業工程を終了しました。

いくつかのサンプルアプリケーションの仕様（'specification'）を確認し、それを RTA-OSEK GUI を用いて実装（'implementation'）しました。

新しいアプリケーションの作成や、スティミュラス、レスポンス、タスク、ISR を定義して設定するための基本的な操作方法を学び、さらにアプリケーションコードの記述も行いました。

また、アプリケーションのビルドと機能テストも行い、いくつかの分析オプションについても説明しました。

以降の項では、RTA-OSEK の各機能について、順に詳しく説明します。

3.5 複数の OIL ファイルの使用

プロジェクトによっては複数の OIL ファイルを使用する必要が生じる場合があります。これにはたとえば、OIL ファイルを生成する別のツールを使用したり、完全な OS アプリケーションの形式になっているサードパーティ製のソウトウェアを使用したりする場合が考えられます。

OIL の規格では、複数のファイルをマージするインクルードメカニズムが定義されています。これは C 言語の #include 文と同様に機能しますが、OIL の構文では CPU 節は 1 つしか使用できません。このため、構文的に完結している 1 つのマスタ OIL ファイルと、完結していない複数の OIL ファイル群が必要となります。

RTA-OSEK には、それ以外に以下の 2 とおりの方法で複数の OIL ファイルを扱うことができます。

- インポート
- 補助（“Auxiliary”）OIL ファイル

3.5.1 ファイルのインポート

RTA-OSEK では、外部ファイルをインポートすることにより、構文的に完結している複数の外部 OIL ファイルの内容をマージしてプロジェクトにインポートすることができます（メニューコマンド **File → Import**）。

この際、以下の点についての注意が必要です。

- インポートされるすべての OIL ファイルは、構文的に完結していること、つまりサブシステムの宣言文がすべて 1 つの CPU 節内に含まれている必要があります。
- インポートされる OIL ファイル内の設定によって、プロジェクトファイル内の既存の設定がオーバーライドされます。

重要

RTA-OSEK がプロジェクトファイルを保存する際、インポートされた値もすべて保存されます。このため、コンフィギュレーションに応じてオブジェクト（タスクなど）の追加や削除を行うサブシステムがある場合は、プロジェクトファイルを保存しないようにするか、または外部 OIL ファイルを使用するようにしてください。

3.5.2 外部 OIL ファイル

RTA-OSEK のプロジェクトファイルを保存すると、コンフィギュレーションデータはすべて 1 つの OIL ファイルに書き込まれます。このため、もしも RTA-OSEK に読み込まれていた OIL ファイルが、#include メカニズムによって複数の OIL ファイルをバインドするものであった場合、この構造は失われてしまいます。

これは多くの場合において問題にはなりませんが、アプリケーションに含まれるすべてのパーツ（TCP/IP スタックなど）を管理する外部ツールが使用されている場合、このツールがサブシステムに関連する OIL 宣言文を含むファイルを生成する可能性があります。

そのような場合、サブシステムの内容が変更されると、RTA-OSEK は、関連するすべての OIL ファイルを読み込み直してプロジェクトを更新する必要があります。

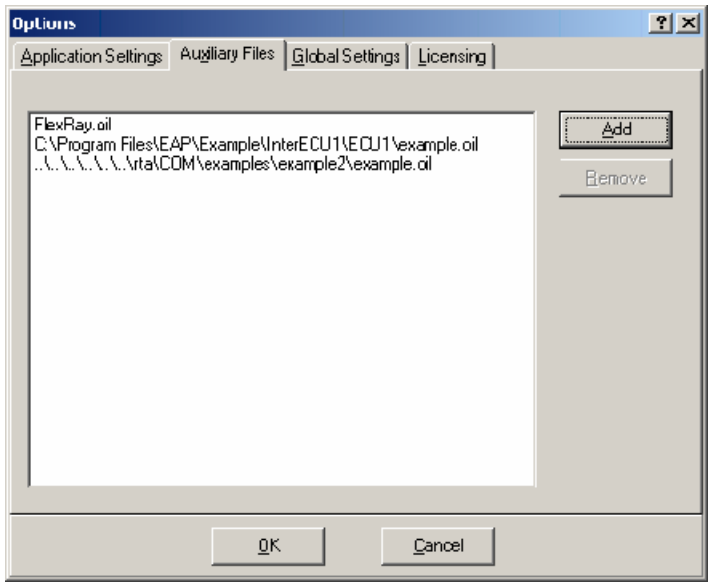
これを行うには、ファイルをマニュアル操作でインポートする（メニューコマンド **File → Import**）か、またはファイルを**外部 OIL ファイル**としてプロジェクトに登録します。

RTA-OSEK は、メインプロジェクトファイルを読み込んだ後に外部 OIL ファイルを読み込みます。これは、プロジェクトファイルの最後部において #include 文でファイルを読み込む場合と同じ結果となります。

外部 OIL ファイルについては、以下の点についての注意が必要です。

- 外部 OIL ファイル内で構文が完結していることが必要です。つまり、サブシステムの宣言文がすべて 1 つの CPU 節内に含まれている必要があります。
- 外部 OIL ファイル内の設定によって、プロジェクトファイル内の既存の設定、および、すでに読み込まれている他の外部 OIL ファイルの設定がオーバーライドされます。
- RTA-OSEK のプロジェクトファイルを保存する際、外部ファイルやインポートファイルからの値もすべて保存されます。このため、サブシステムのコンフィギュレーションに応じてオブジェクト（タスクなど）の追加や削除が行われた場合、プロジェクトファイルを保存してしまうと、GUI を使用して不要なオブジェクトを削除しなければならなくなる場合があります。
- 外部 OIL ファイル内のオブジェクトのコンフィギュレーションを変更した場合、その変更内容はプロジェクト OIL ファイル内にのみ保存され、外部 OIL ファイルには保存されません。その後、プロジェクト OIL ファイルを開くと、外部 OIL ファイルから読み込まれていたオブジェクトの値が再度読み込まれ、プロジェクト OIL ファイル内の変更内容が上書きされてしまいます。

メニューコマンド **File → Options → Auxiliary** で、使用する外部 OIL ファイルのファイル名とパスを指定できます。パスは、絶対パス、またはプロジェクト OIL ファイルのロケーションへの相対パスを使用できます。



3.6 RTA-OSEK *Builder*

RTA-OSEK Builder には以下の 2 つの機能が含まれています。

- アプリケーションを構築するための基本的な入力手段を提供します。これはすでに OSEK コンセプトを理解しているユーザー向けの方法です。これは 3.6.1 項に説明されています。
- アプリケーションのビルドを行います。オプション設定とビルドを実行する方法は、3.6.2 項に説明されています。

以降に、Builder に含まれるオプションについて順に説明します。

3.6.1 基本データエントリ

RTA-OSEK の *Planner* には、OSEK コンセプトに十分に精通していないユーザーのためのさまざまな便利な機能が含まれていますが、これに対して *Builder* では、標準の OSEK 機能に限定したテーブル形式のインターフェースを使用して、アプリケーションを作成したり変更したりすることができます。

Builder においては、アプリケーションのデータは“Basic Data Entry”ビューを使用して入力します。さまざまな OSEK オブジェクトが、クラスごとに個別のタブに表示され、**Add** ボタンや **Remove** ボタンで各オブジェクトの作成と削除を行えます。

各タブを選択すると、個々の OSEK オブジェクトについての詳細な情報がすべて表示されます。

図 3-73 は **Tasks** タブを示しています。

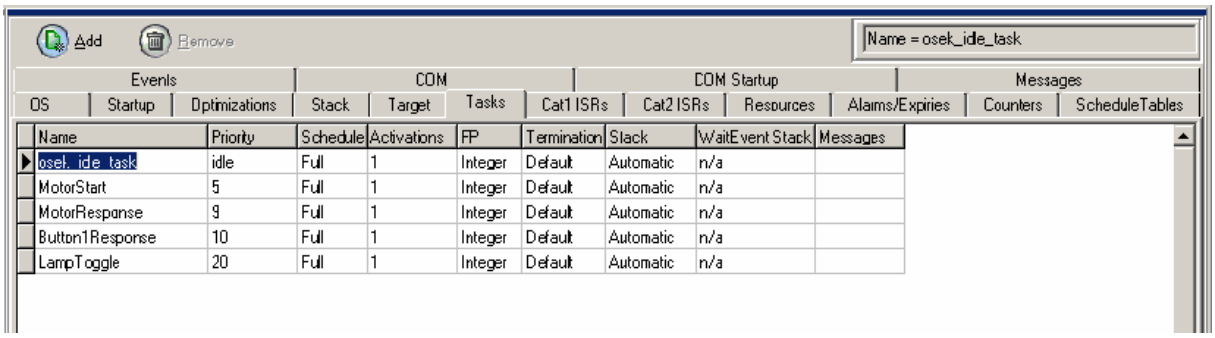


図 3-73 “Basic Data Entry“ビューを使用してタスクを作成する

3.6.2 アプリケーションのビルド

アプリケーションを作成するすべてのステップが完了したら、ビルド工程に移ります。ビルド工程には以下の処理が含まれます。

- タスクと ISR の C ファイルのコンパイル
- RTA-OSEK が生成した `osekdefs.c` という C ファイルのコンパイル
  このファイルには、アプリケーションで使用する RTA-OSEK オブジェクトを定義するデータが含まれています。
- RTA-OSEK が生成した `osgen`（ファイル拡張子はターゲットにより異なります）というアセンブラファイルのアセンブル
  このファイルには、RTA-OSEK の低レベルの OS データを定義するデータが含まれています。
- その他の C ファイルのコンパイル
  レスポンスの実装やハードウェアの初期化に用いられるターゲット固有ファイルなどが含まれます。
- ファイルのリンク
  上記の処理によって生成されたファイルを、RTA-OSEK の OS API ライブラリ、コンパイラのランタイムライブラリ、任意のランタイム起動コードなどとリンクします。

アプリケーションのビルドには、マニュアル操作で行う方法と、カスタムビルドスクリプトを用いる方法の 2 通りがあります。

- 「マニュアルビルド」は、RTA-OSEK GUI の外部でアプリケーションをビルドする方法で、これは RTA-OSEK を大規模なビルド処理に統合するような場合に使用されます。この方法の概略は、3.6.5 項に説明されています。

- 「カスタムビルド」は、RTA-OSEK GUI 内でビルドを行う方法で、小さなサンプルファイルをビルドするような場合に使用されます。カスタムビルドを行うには、RTA-OSEK に対してコンパイラツールチェーンや OS 以外のソースコードファイル、リンカ設定などを指定する必要があります。カスタムビルド工程については、3.6.6 項に説明されています。

3.6.3 RTA-OSEK コンフィギュレーションの整合性チェック

Build Checks ボタンをクリックすると、システムが完全に定義されてビルドを行える状態になっているかがチェックされます。このステップにおいてはコンパイルやコードチェックは一切行われず、必要なオブジェクトが *Planner* または基本データエントリビューにおいて定義されているかどうか、という点だけがチェックされます。

必要なオブジェクトがすべて定義されていると、各チェック項目ごとに“OK”と表示されます。

必要なオブジェクトについては、各ターゲットのバインディングマニュアルを参照してください。

3.6.4 マニュアルビルド

マニュアルビルドを行うには、RTA-OSEK Builder のコード生成ツールを起動し、OS コンフィギュレーションを処理してソースファイル（C ソースファイルとヘッダファイル、およびアセンブラソースファイル）を生成する必要があります。Make ファイルなどによる外部ビルド処理を行う際、これらのファイルを組み込みます。

この機能により、ORTI デバッガファイル（ターゲットコンフィギュレーションにおいてデバッガが選択されている場合のみ）、および RTA-TRACE 用ディスクリプションファイル（トレースが有効になっている場合のみ）も生成されます。

これらのファイルを生成するには、RTA-OSEK Builder の **Create Files** ボタンをクリックします。

ビルドエラーがなければ、RTA-OSEK GUI はこれらのファイルを生成し、ファイル一覧を表示します。この例を図 3-74 に示します。

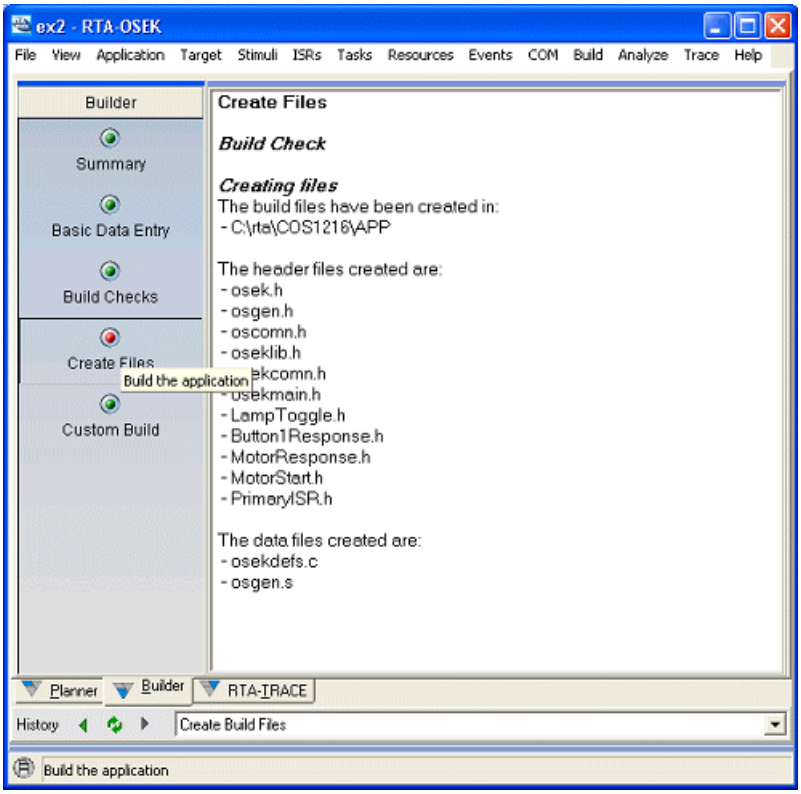


図 3-74 RTA-OSEK ファイルの生成

デフォルト状態において、生成されたファイルはアプリケーション OIL ファイルと同じディレクトリに保存されます。このロケーションはプロジェクトごとに任意に変更でき、新しいデフォルトロケーションを設定することもできます。

これらの設定を行うには、**File → Options....** を選択して **Options** ダイアログボックスを開きます。

アプリケーションの設定は **Application Settings** タブで行います。生成されるファイルのタイプごとに異なるロケーションを指定することができます。パス内で使用されるピリオド（.）は現在のディレクトリを表し、2 つのピリオド（..）は親ディレクトリを表します。

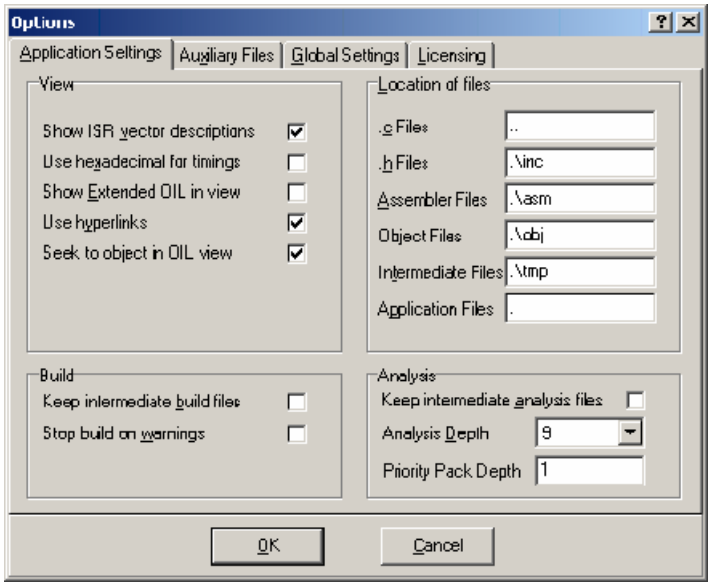


図 3-75 アプリケーションファイルのロケーション設定

同様に、グローバルデフォルト設定は **Global Settings** タブで行います。

重要
実際には、アプリケーション設定がグローバル設定をオーバーライドします。

大規模なシステムビルド処理の一部として OS コードを生成する場合、RTA-OSEK GUI を使用せずに OIL ファイルの処理を行えると便利です。このような場合は、コマンドラインから RTA-OSEK Builder のコード生成を実行することができます。

```
$ rtabuild application.oil
RTABuild version 5.x.x
Copyright (c) LiveDevices Ltd 2001-2007
$
```

コマンドラインオプションの詳細は、`rtabuild -h` を実行するか、または『RTA-OSEK リファレンスガイド』を参照してください。

アプリケーション OIL ファイルから RTA-OSEK ファイルを生成した後は、以下のことを行ってください。

- ヘッダファイルをインクルードパスに追加します。
- `osekdefs.c` をコンパイルします。
- `osgen.<asm>` をアセンブルします。
- 適切な RTA-OSEK コンポーネントライブラリとリンクします。この際、ビルドステータス（標準、タイミング、拡張）によって、使用するライブラリが異なる点に注意してください。ライブラリの名前と格納場所は、アプリケーション実装ノートに記載されています。

重要
アプリケーションのコンパイルとアセンブルを行う際は、**`rtkbuild.bat`** ファイルを参考にしてください。RTA-OSEK GUI の **'Custom Build'** ビューで **Create 'rtkbuild.bat'** ボタンを押すと、このファイルが生成されます。`osekdefs.c` および `osgen.s` 用に設定されているコンパイラ／アセンブラオプション以外のオプションを使用する際には、特に注意が必要です。

3.6.5 カスタムビルド

カスタムビルドには、rtkbuild.bat というビルドスクリプトが使用されます。これは、各タスク、ISR、osekdef、osgen ファイルのコンパイルとアセンブルを行うための MS-DOS バッチファイルで、**Build Now** ボタンをクリックすると自動的に生成されます。

カスタムビルドにおいては、各タスクと ISR ごとにそれぞれ個別のソースファイルが存在していることが前提であり、またどの RTA-OSEK ソースファイルを生成してそれらをどのようにコンパイルするか、という情報も明らかになっています。

そのため、ここで行うことは、必要なその他のファイルのコンパイルとアセンブル処理、およびオブジェクトモジュールのリンクとロケートをビルとスクリプトに追加するだけです。これらの設定を行うには、RTA-OSEK Builder のワークスペースの **Configure** ボタンをクリックし、**Custom Build Options** ダイアログボックスを開きます。詳しくは 3.6.6 項を参照してください。

ビルドスクリプトが完成したら、以下の操作を行ってください。

- アプリケーションを保存します。
- **Build Now** ボタンをクリックします。

RTA-OSEK GUI はシステム内容をチェックし、エラーが見つかると、エラーメッセージを発行して処理を中止します。大きな影響のない問題が見つかった場合は、ワーニングを発行して処理を続行します。

エラーがなければ RTA-OSEK GUI はスクリプトを実行します。RTA-OSEK GUI ウィンドウには実行中のスクリプトが表示されます。スクリプトがすべて実行されると、新しい実行ファイルが作成され、テストを行えるようになります。

3.6.6 カスタムビルドのオプション

Configure ボタンをクリックすると **Custom Build Options** ダイアログボックスが開きます。このダイアログボックスで、ビルドスクリプトの編集、環境変数の設定、カスタムボタンの定義などを行うことができます。カスタムビルドスクリプトは、生成されたファイル（_rtkbuild.bat）に含まれています。このスクリプトは、rtkbuild.bat を呼び出す前のカスタムオプションを定義するものです。

環境

Custom Build Options ダイアログボックスの **Environment** タブで、カスタムビルド処理で使用したい環境変数を定義します。環境変数を定義する際は、RTA-OSEK GUI マクロを使用できます。ビルトインマクロの一覧を見るには、**Application → Macros → Built-In Macros** を選択するか、または『RTA-OSEK リファレンスガイド』を参照してください。

マクロを使用すると、RTA-OSEK の設定（リンクする RTA-OSEK ライブラリ名、RTA-OSEK インクルードファイルのパスなど）にアクセスすることができます。

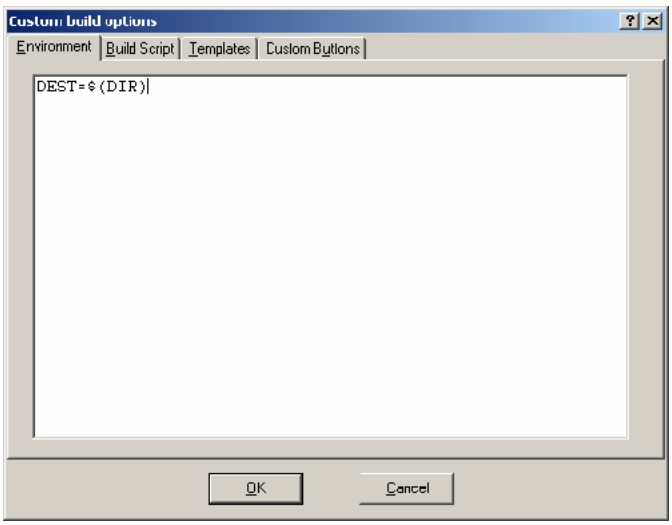


図 3-76 カスタムビルドの環境変数

ビルドスクリプト

デフォルト状態においては、ビルドスクリプトに含まれる文は 'call rtkbuild.bat' のみです。

call rtkbuild.bat では、最初に Toolinit.bat（RTA-OSEK の <install dir>¥rta¥<target> ディレクトリに格納されています）が実行されてコンパイラツールチェーン用の環境変数が設定されます。

アプリケーションを完全にビルドするには rtkbuild.bat だけでは十分ではありません。通常、その他のコードをコンパイル／アセンブルしたりその他のライブラリをリンクするための情報を、スクリプトに追加する必要があります。

ここで、target.c というファイル内にターゲット固有の初期化とレスポンスを実装したコードを記述した、と仮定します。その場合、以下のような行を追加する必要があります。

```
'%cc% %COPTS% target.c'
```

ここで注意すべき点は、cc および copts は、rtkbuild.bat 内ですでに設定されている環境変数でなければならない、ということです。コンパイラ名とオプションは、さらに明示的に記述することもできます。また、コンパイラコマンドラインに -debug オプションを追加することもでき、この場合、環境変数 APP_COPT を追加する必要があります。

リンク／ロケートの工程は、さらにターゲットによって異なる可能性があります。以下に、RTA-OSEK マクロを使用して 1 行のコードで記述した例を示します。

```
%lnk% -v -l%RTA_LIB% -l%CBASE%¥lib -m$(NAME).map -otemp.out link.lkf
$(RTKOBJECTS) target.$(OBJEXT) $(RTKLIB) crtsi.$(LIBEXT) libm.$(LIBEXT)
libi.$(LIBEXT)
```

図 3-77 に **Custom Build Options** ダイアログボックスの表示例を示します。

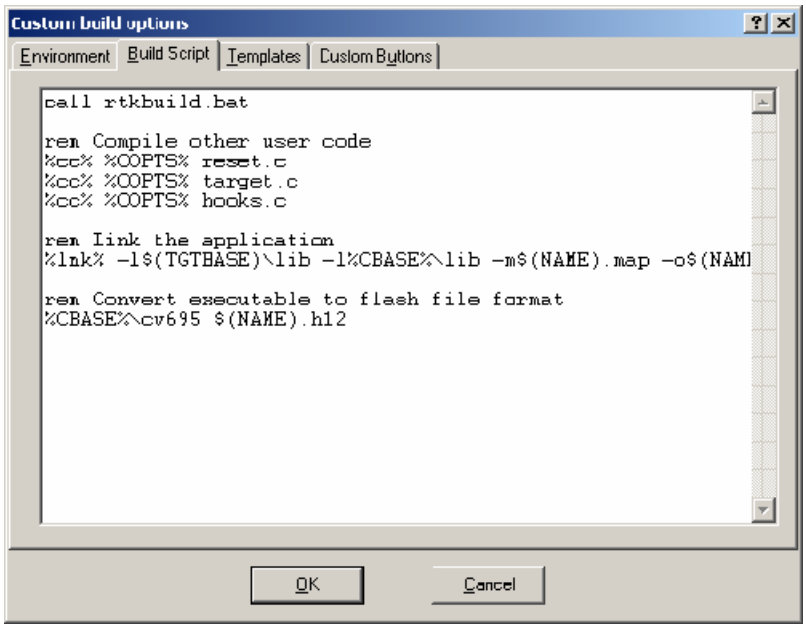


図 3-77 カスタムビルドスクリプトを作成する

'%' で始まる語は、Toolinit.bat および rtkbuild.bat 内で設定される環境変数です。

また、$(NAME) のように '&()' で囲まれた語は、RTA-OSEK GUI のマクロ変数です。これらの変数は、カスタムビルド用の任意のエントリで使用でき、ビルド実行時に適切な値に展開されます。

使用できる環境変数とマクロの一覧は、『RTA-OSEK リファレンスガイド』を参照してください。

テンプレート

RTA-OSEK のカスタムビルダは、メインプログラム以外に、ユーザーが宣言したタスクと ISR 用の、テンプレートコードを生成します。

Templates タブでは、このテンプレートコードの設定を行うことができます。

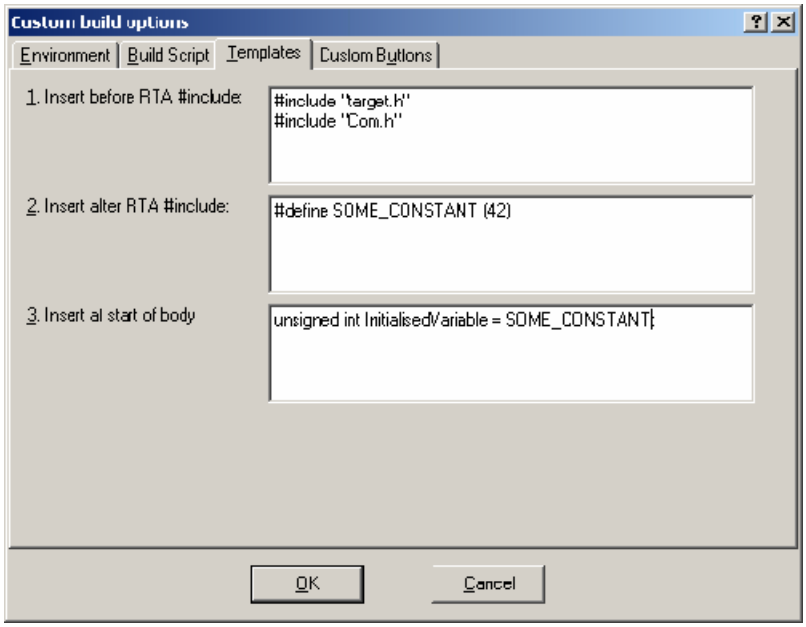


図 3-78 テンプレートコードのカスタマイズ

テンプレートコードが生成される際、このタブに定義された文がコード内にそのまま挿入されます。

```
/* Template code for 'T1' in project: MyProject */
#include "target.h"
#include "Com.h"
#include "T1.h"
#define SOME_CONSTANT (42)
TASK(T1)
{
unsigned int InitialisedVariable = SOME_CONSTANT;
        TerminateTask();
}
```

カスタムボタン

Custom Buttons タブでは、カスタムビルドで使用できるユーザー定義ボタンを最大 5 個まで作成することができます。デフォルトでは、第 1 番目のボタンは“Quick Edit”というボタンに設定されています。

これらのボタンを使用して、図 3-79 項のように外部プログラム（ソースコード管理システム、デバッガなど）を起動することができます。

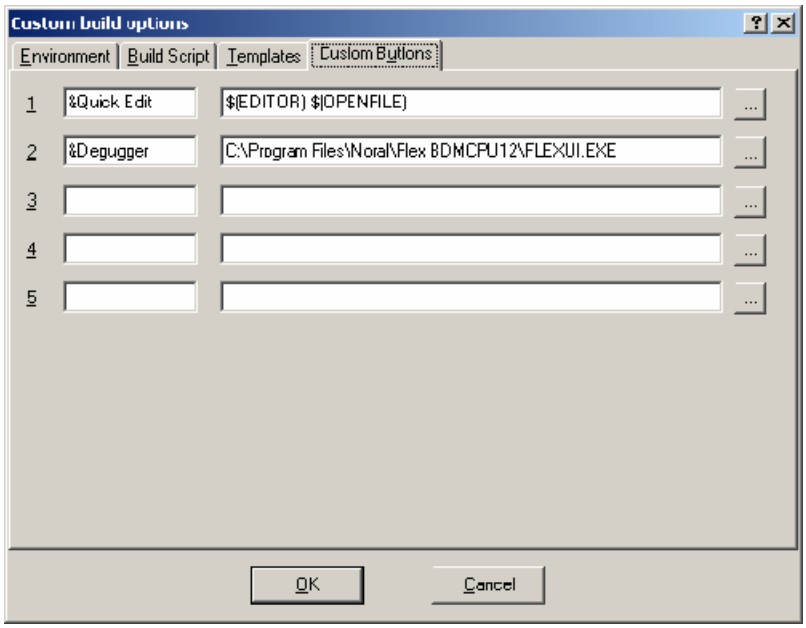


図 3-79 カスタムボタンの設定

3.6.7 パッケージの使用

RTA-OSEK では、「パッケージ」という概念を利用して、サードパーティ製ソフトウェアのビルドを共通の方法で統合することができます。パッケージはライブラリ関数のセットを定義するもので、これによってライブラリ関数を呼び出す際のワーストケースの実行時間や、スタックやビルドに関する情報を設定することができます。

パッケージ定義は `<install dir>¥rta¥packages` に保存され、`.pdef` という拡張子が付きます。RTA-OSEK V5.0 には、RTA-COM 用のパッケージ定義が含まれています。

図 3-80 に示されるように、アプリケーション用に使用するパッケージをアクティブにします。

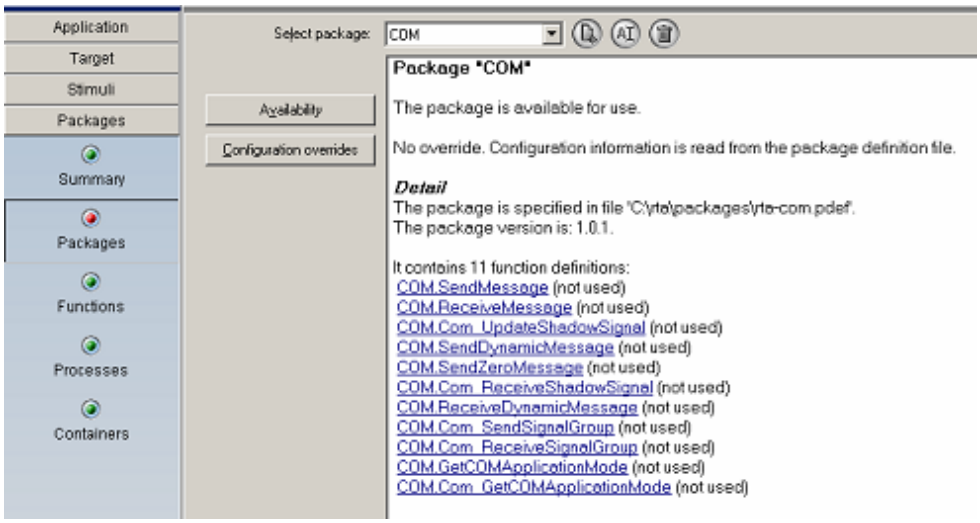


図 3-80 パッケージ

RTA-OSEK が使用できるのは、アクティブなパッケージに含まれる関数のみです。また各タスクや ISR ごとにどのライブラリ関数を呼び出すかを指定することができ、さらにそれを呼び出す際のスタックサイズも指定できるので、RTA-OSEK がスタック使用量を計算する際にパッケージ関数のスタック使用量も合算されます。

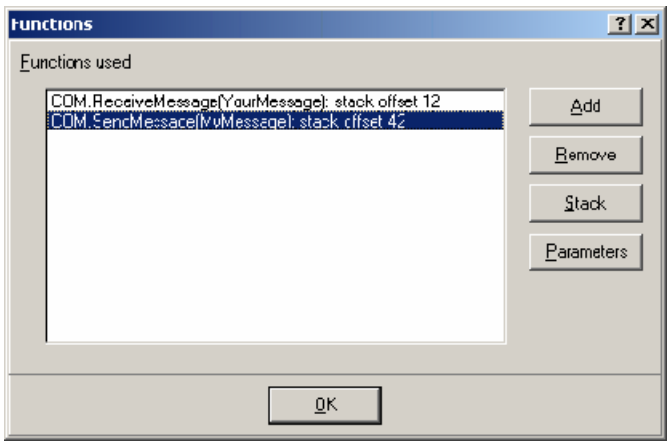


図 3-81 パッケージ

パッケージの使用に関する詳細な情報は、サポート窓口までお問い合わせください。

3.7 実装に関するその他の情報

この項では、ユーザーアプリケーションを作成する際に注意すべき点を説明します。

3.7.1 ネームスペース

RTA-OSEK コンポーネントには**ネームスペース**が定義されています。既存の名前と競合する名前や内部的に使用される名前を使用することはできません。一般に、RTA-OSEK コンポーネントが使用する内部名には 'os'、'OS'、'_os'、'_OS' というプレフィックスが付きます。その他の内部名としては、拡張 OIL 文法で使用されるトークンなどがあります。以下のルールに従ってください。

- 'os'、'OS'、'_os'、'_OS' のいずれかで始まる名前を使用しないこと。これらはすべて RTA-OSEK コンポーネント用に確保されています。
- 'os' で始まるオブジェクトモジュールまたはファイル名を使用しないこと。

3.7.2 リエントランシー

すべての RTA-OSEK コンポーネントの呼び出しはリエントラントに行うことができ、リエントリ（再入）を防ぐ特別なプロテクションは必要ありません。しかしコンパイラと共に提供されている C ライブラリは、通常、リエントラントではありません。

浮動小数点が使用される場合、C ライブラリにはリエントランシーに関する共通の問題が生じますが、コンパイラが自動的に浮動小数点ライブラリコールを挿入することにより、この問題が暗黙的に発生する可能性があります。RTA-OSEK コンポーネントの場合は、浮動小数点を使用するオブジェクトを指定することにより、タスクと ISR 内で安全に浮動小数点を使用できます。

移植性

一部のターゲットでは、関数が構造体を返す場合にリエントランシーの問題が発生する可能性があります。そのような場合は、構造体を返す関数を呼び出す前後で必ずリエントランシーに対するプロテクションを行わなければなりません。

重要

リエントラントでない関数に対するリエントリ（再入）を防ぐのは、ユーザーの責任で行ってください。通常、これは割込みをディセーブルにするか、または RTA-OSEK コンポーネントの「リソース」を用いることにより実現できます。実際、非リエントラント関数の C ライブラリには、再入対策のために RTA-OSEK コンポーネントリソースの get 関数と release 関数を挿入できる「フック」が含まれている場合があります。

RTA-OSEK コンポーネントを実行しているシステムのリエントラント関数は、どれも**完全にリエントラント**である必要はなく、「シリアルにリエントラント」であれば十分です。**シリアルにリエントラント**な関数とは、現在その関数を実行しているスレッドから、その関数を次に実行するスレッドに切り替えることができるものです。

一部のコンパイラにおいては、リエントラント関数と非リエントラント関数として別々のコードを生成することができます。たとえば、非リエントラント関数ではパラメータとローカル変数について静的データオーバーレイを使用しますが、リエントラントバージョンではスタックを使用します。

RTA-OSEK コンポーネントには、適切なコードが確実に生成されるようにするための `OS_REENTRANT` および `OS_NONREENTRANT` というマクロがあります。コンパイラによってはこれらのマクロによって何も影響を受けないものもありますが、移植性を考慮して、これらのマクロを使用しておくことをお勧めします。これらのマクロについては、『RTA-OSEK リファレンスガイド』に詳しく説明されています。

RTA-OSEK コンポーネントは、通常は標準 C ライブラリの関数を使用しませんが、C コンパイラライブラリから一部のターゲット固有コードを使用しなければならない場合があります。RTA-OSEK コンポーネントは浮動小数点演算を自分では行いません。各ターゲットにおける RTA-OSEK コンポーネントの要件については、『RTA-OSEK バインディングマニュアル』を参照してください。

3.8 まとめ

- RTA-OSEK には厳密なリアルタイムシステムを構築するための仕様作成、設計、実装、ビルド、および分析のための機能が含まれています。
- アプリケーションは「スティミュラスとレスポンス」の形でモデリングされます。パフォーマンス上の制約条件は、レスポンスについてのデッドラインとして定義されます。
- 設計と実装の工程において、スティミュラスの検知方法とレスポンスの生成方法を、OSEK OS オブジェクトによって定義します。
- 使いやすい開発環境インターフェースを利用してカスタムビルドやマニュアルビルドを行い、アプリケーション全体をビルドすることができます。
- アプリケーション内の各タスクや ISR の実行時間に関する要件が明らかになっていれば、スティミュラス - レスポンスモデルについてタイミング分析を行い、ランタイムにおいてパフォーマンスに関するすべての制約条件が満たされていることを調べることができます。

4 タスク

一度に多くの処理を行う必要のあるシステムは、**コンカレント**なシステムと呼ばれています。これらの処理は何らかのソフトウェアパーツとしての形をとり、それらを実行するプログラムはコンカレントに実行される必要があります。このようなプログラムは、データを共有しなければならないときなどには、互いに協調し合う必要があります。

リアルタイムシステムでは、コンカレントな処理が**タスク**として表現され、アプリケーションコードの大部分はタスク内に存在します。

同時に実行しなければならないタスクが多数ある場合には、コンカレント処理を実現するための手段を講じる必要があります。これを行うには、タスクごとに別のプロセッサを設けるのも 1 つの方法です。また並列コンピュータを使用する方法もありますが、これは費用がかかりすぎてしまいます。

これよりもはるかに費用効率的にコンカレント処理を実現できる方法は、1 つのプロセッサ上でタスクを一度に 1 つずつ実行するというものです。タスクは順次切り替わるので、同時に複数のタスクが実行されているかのように見えます。

4.1 タスク切り替え

タスクの切り替えには、**スケジューラ**が用いられます。スケジューラは実装された**スケジューリングポリシー**（'scheduling policy'）に基づいてタスク切り替えを行います。このポリシーは、1 つのタスクがいつ実行を終了（または一時停止）して次のタスクの実行が開始されるか、ということを指示するものです。

OSEK オペレーティングシステムには、**固定優先度**（'fixed priority'）スケジューリングポリシーを使用するスケジューラが定義されています。

このポリシーでは、各タスクに固定優先度が割り当てられます。スケジューラは常に、実行準備ができているタスクのうち優先度が最も高いものを実行します。あるタスクの実行中にそれよりも優先度の高いタスク の実行準備ができると、そのタスクが実行中のタスクを**プリエンプト**（強制排除）します。そして優先度の高い方のタスクの処理が終了すると、プリエンプトされていた優先度の低い方のタスクは、プリエンプトされた時点からの処理を**レジューム**（実行再開）します。

図 4-1 にその仕組みを図解します。

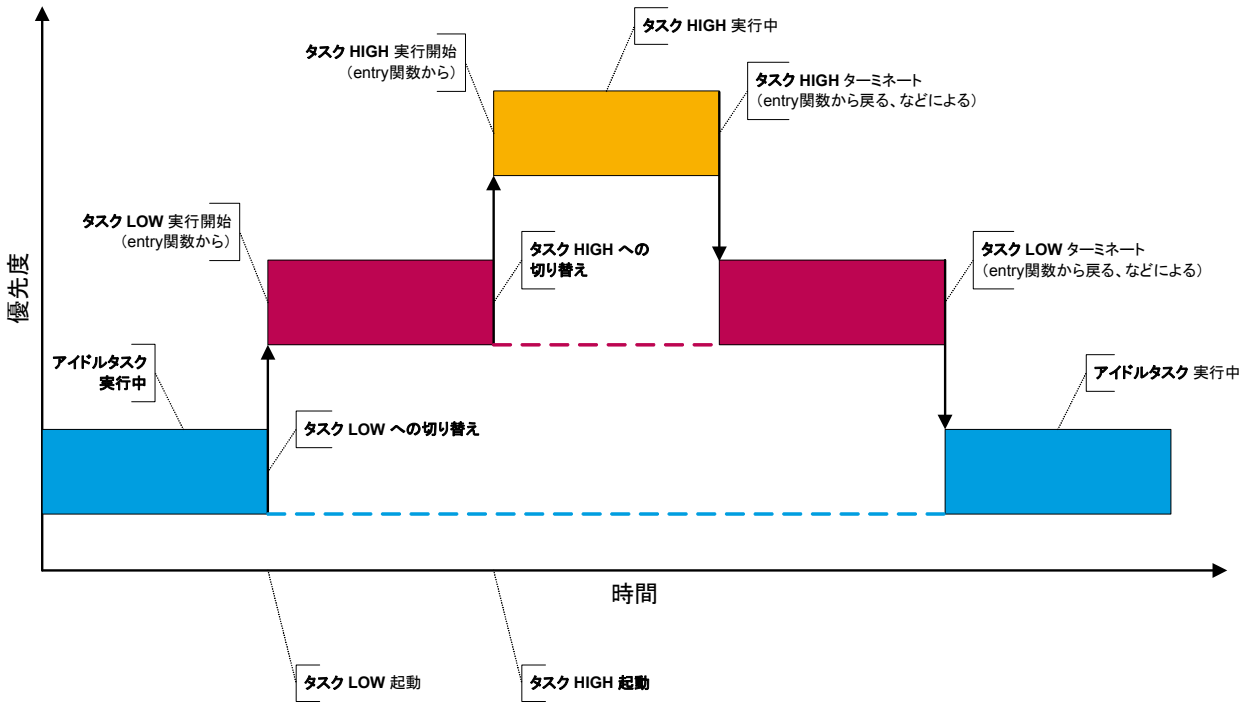


図 4-1 タスクの実行例

図 4-1 の例では、まず、アイドルタスクが実行されていて（アイドルタスクについては、4.8 項で説明します）、ある時点で低優先度のタスク `LOW` が起動されています。それによってタスク切り替えが行われて `LOW` が実行され、最初にそのエントリ関数が実行されます。

その後、LOW より優先度の高い HIGH というタスクが起動され、再びタスク切り替えが行われます。LOW はプリエンプトされ、HIGH がそのエントリ関数の最初から実行を開始します。

H が終了すると、LOW が、プリエンプトされた時点から実行を再開し、最終的に終了します。そして最後に、アイドルタスクがプリエンプトされた時点から実行を再開します。

4.2 シングルスタックアーキテクチャ

RTA-OSEK では、すべてのタスクと割込み処理が 1 つのスタック[1] 上で稼動する「シングルスタックモデル」を使用しています。「シングルスタック」とは、アプリケーションが使用する C 言語のスタックのことを指します。

一般的には、タスクが実行されると、そのスタック使用量は増え、再度減少します。あるタスクが優先度の高いタスクにプリエンプトされると、そのスタック使用量はそのまま変化しません（通常の関数呼び出しと同じです）。タスクがターミネートすると、そのタスクが使用していたスタックスペースは解放され、次に実行されるタスクによって再利用されます（これも通常の関数呼び出しと同じです）。

シングルスタックモデルにおいては、スタックサイズは、タスクと ISR の数ではなく、優先度レベルの数に比例します。複数のタスクによる 1 つの優先度の共有は、直接、または内部リソースを共有することにより、またはノンプリエンプティブに設定されることによって実現されますが、これらのタスクは同時にスタックを使用することはないため、スタック領域を安全に共有することができます。これは同じハードウェア優先度を共有する複数の ISR についても同様です。

また、シングルスタックモデルにより、リンク時のスタック領域割り当て処理が簡略化されます。システム全体のスタック用のメモリセクションを 1 つだけ指定すればよいため、OS を使用しない場合と同じように処理できます。

4.3 基本タスク（'basic task'）と拡張タスク（'extended task'）

OSEK オペレーティングシステムでは、タスクは**基本タスク**と**拡張タスク**という 2 つのタイプに分類されます。各タスクタイプごとに、個々のタスクの状態が、オペレーティングシステムのステートモデルとして定義されています。

各タイプには、タイプ 1 とタイプ 2 という 2 つのレベルがあります。これらについては、本章で後述します。

4.3.1 基本タスク（'basic task'）

基本タスクは**シングルショット**タスクです。つまり、タスクはレディ状態にされてから、そのエントリポイント（開始点）から実行を開始します。実行中、自分より優先度の高いタスクがレディ状態になると、そのタスクによりプリエンプト（強制排除）されますが、それ以外の場合は、ターミネートするまで実行を続けます。ターミネーションの後、このタスクを再びレディ状態にして実行することも可能です。

基本タスクの状態

基本タスクには以下の状態があります。

- レディ（'Ready'）
- 実行中（'Running'）
- サスペンド（'Suspended'）

タスクのデフォルト状態はサスペンドです。タスクは、タスク起動の API コール、または起動を引き起こす他の何らかの方法により、レディ状態に移行します。基本タスクの状態遷移図は図 4-2 のとおりです。

1. 一部のマイクロコントローラにおいては、ハードウェアで複数のスタックをサポートしているものがあり（例：割込み用スタックなど）、RTA-OSEK はそれらのスタックを使用する場合があります。

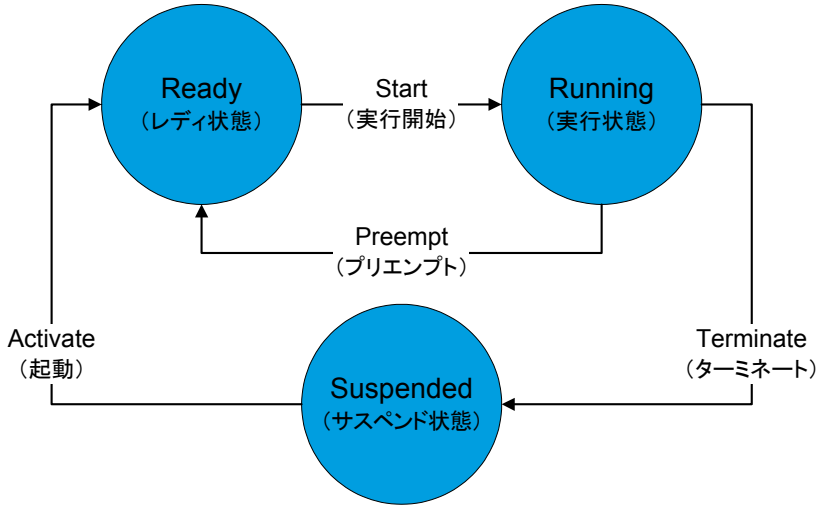


図 4-2 基本タスクの状態遷移

図 4-2 に示されるように、RTA-OSEK コンポーネントがあるタスクの実行を指示すると、そのタスクがレディ状態から実行状態に移行します。タスクの実行はタスクのエントリポイントから開始されます。

実行中のタスクよりも優先度の高いタスクがレディ状態になると、実行中のタスクはプリエンプト（強制排除）されて実行状態からレディ状態に移行します。一度に 1 つのタスクだけが実行状態になることができます。

ターミネートしたタスクはサスペンド状態に戻ります。

重要

基本タスクでは、ある特定のイベントを待ったり、（ビジーウエイト時以外は）任意のディレイを設けたりすることはできません。図 4-2 に示すように、タスクをサスペンド状態にするにはそのタスクをターミネートする以外に方法はありません。

タイプ 1 とタイプ 2 の基本タスク（BCC1 と BCC2）

前述のように、各コンフォーマンスクラス内には 2 つのレベルがあります。基本コンフォーマンスクラス（BCC：**B**asic **C**onformance **C**lass）にはタイプ 1 とタイプ 2 のタスクがあります。

- タイプ 1 の基本タスク（**BCC1**）
  シングルショットタスクです。一意のタスク優先度を持ち、サスペンド状態の時以外は起動できません。
- タイプ 2 の基本タスク（**BCC2**）
  シングルショットタスクです。他のタスクと優先度を共有でき、多重起動も行えます。**多重起動**（'multiple activation'）は、タスクがレディ状態または実行状態にある時に、指定された回数までは重複して起動できることを意味します。これについては、4.4.2 項で詳しく説明します。

2 つ以上のタスクの優先度が同じ場合、優先度を共有するタスク同士は**相互排除的**（'mutual exclusive'）に実行されます。つまり、あるタスクが実行中の場合、同じ優先度を共有する他のタスクはそれをプリエンプトすることはできず、それらのタスクの起動は FIFO 方式でキューイングされます。その結果、システムのタイミング分析は不可能となります。

重要

RTA-OSEK コンポーネントをより効率的に稼働させるには、各タスクに一意のタスク優先度を割り当て、内部リソースを使用することで相互排除を実現してください。そうすることによって、タイミング分析も可能になります。

RTA-OSEK GUI で、キューイングできるタスク起動の最大数をユーザー設定できます。RTA-OSEK コンポーネントでは、タスクはユーザー設定された最大起動数までは、各起動につき 1 回実行されます。複数のタスクが同じ優先度を共有する場合には、タスクは起動された順序に厳密に従って実行されます。

4.3.2 拡張タスク（'extended task'）

拡張タスクは、通常は無限ループの形で存在しています。一度実行状態になると、通常はターミネートしません。イベントの結果を待つ間、待機（'Waiting'）状態になって「スリープ」することができます。

拡張タスクの状態

拡張タスクには、基本タスクと同じく以下の 3 つの状態があります。

- レディ（'Ready'）
- 実行中（'Running'）
- サスペンド（'Suspended'）

以上に加えて、もう 1 つの状態もあります。

- 待ち（'Waiting'）

図 4-3 の状態遷移図は、OSEK オペレーティングシステムの拡張タスクの 4 つの状態を示しています。レディ、実行中、およびサスペンド状態の拡張タスクの挙動は、基本タスクの挙動（図 4-2）と同じであることがわかります。

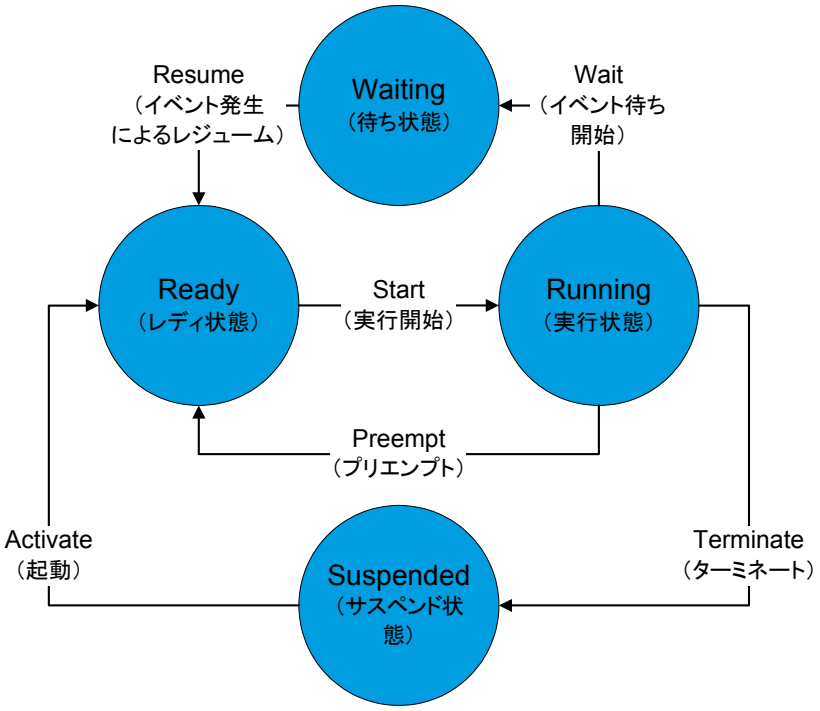


図 4-3 拡張タスクの状態遷移

拡張タスクは、イベントを待つことにより自発的に実行を保留すると、実行状態から待ち状態に移行します。

イベントは、システムイベントの発生を知らせるための単純なシステムオブジェクトです。イベントの例としては、「データが使用可能となったこと」や「センサ値が読み取られたこと」などがあります。イベントが発生すると、タスクは待ち状態からレディ状態に移行します。

拡張タスクがイベントを待っている間は、このタスクよりも優先度の低いタスクを実行することができます。

タイプ 1 とタイプ 2 の拡張タスク（ECC1 と ECC2）

拡張コンフォーマンスクラス（ECC：Extended Conformance Class）にもタイプ 1 とタイプ 2 のタスクがあります。

- タイプ 1 の拡張タスク（**ECC1**）
  イベントを待つことができ、一意のタスク優先度を持っているタスクです。BCC1 と似ていますが、ECC1 の場合はイベントを待つことができます。
- タイプ 2 の拡張タスク（**ECC2**）
  イベントを待つことができ、他のタスクと同じ優先度を持つことができるタスクです。複数のタスクで同じ優先度を共有する場合、タスクは起動された順序に厳密に従って実行されます。タイプ 2 の拡張タスクは、タイプ 2 の基本タスクとは異なり、多重起動（Multiple Activation）を行うことはできません。

重要

拡張タスクをタイミング分析の対象にすることはできません。RTA-OSEK GUI が分析を行える対象は、どの拡張タスクよりも優先度の高い基本タスクと ISR だけに限定されます。したがって、システムの中でリアルタイム性が厳密に守られるべきタスクには、拡張タスクの中で優先度が最高のタスクよりも高い優先度を割り当てる必要があります。

4.4 基本的なタスク設定

他のリアルタイムオペレーティングシステムとは異なり、OSEK、つまり RTA-OSEK ではタスクは静的に定義されます。これによって、RAM の使用量と実行時間を節減できます。

タスクを動的に作成したり削除したりすることはできません。タスクに関する情報はほとんどすべてオフラインで計算し、それを ROM に格納しておくことができます。

タスク優先度を設定するときには、RTA-OSEK GUI を使用します。図 4-4 の例を見て、タスクの構成を確認してください。

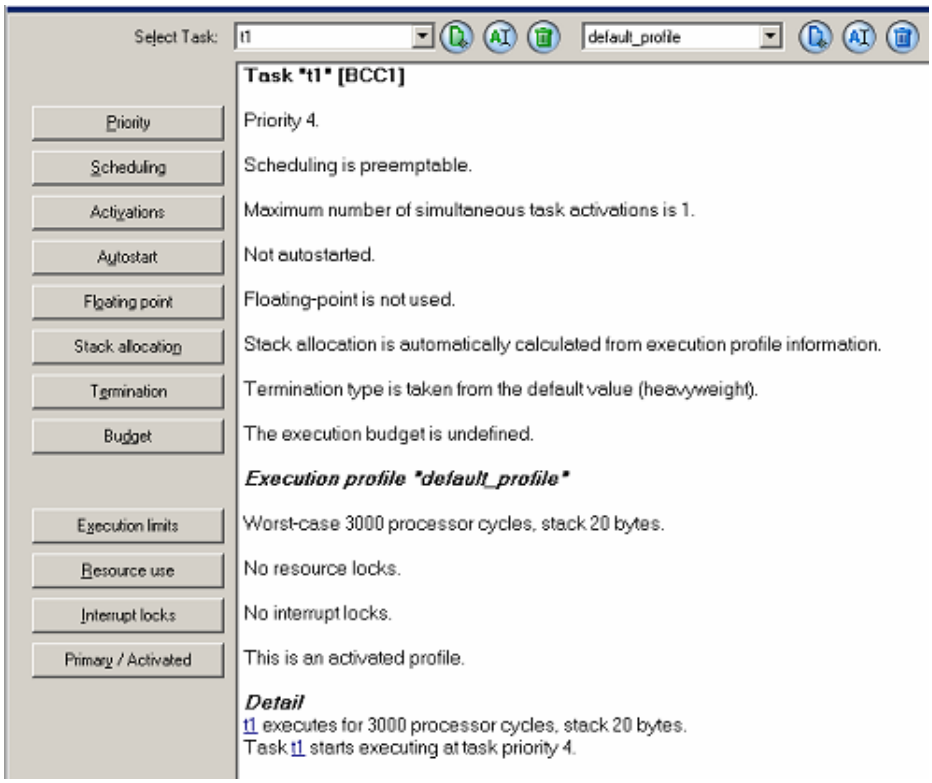


図 4-4 RTA-OSEK GUI でタスクを設定する

各 OSEK タスクには、以下の 5 つの属性が割り当てられます。

- タスク名（Name）
  タスクの機能を実装するためにユーザーが作成する C コードを「参照する」、つまり「**ハンドル**を提供」するために使用されます。

- 優先度（Priority）

  優先度は、スケジューラがそのタスクをいつ実行させるかを判断するために使用されます。優先度は動的に変更することはできません。RTA-OSEK では、0 が最下位のタスク優先度を表し、この値（整数値）が大きくなるほど優先度が上がります。複数のタスクが同じ優先度を共有することができますが、リアルタイムシステムを構築する場合はタスク分析が行えなくなってしまうため、1 つの優先度に 1 つのタスクを割り当てることをお勧めします。

- スケジューリング（Scheduling）

  OSEK タスクはプリエンプティブまたはノンプリエンプティブのいずれかのモードで実行されます。一般的に、アプリケーションのパフォーマンスを上げるにはノンプリエンプティブよりもプリエンプティブを選択してください。

- 起動（Activation）

  OSEK では、起動できるタスクは、現在サスペンド状態にあるタスクのみです。アプリケーションが一時的にオーバーロードとなったような場合、複数のタスク起動をキューイングする必要があります。

- 自動起動（Autostart）

  OS の起動時に、タスクが自動的に起動されるようにするかどうかを指定します。

移植性

定義できるタスクの数は、ターゲットごとに決まっています（通常は 16 か 32 です）。詳細については各ターゲットの『RTA-OSEK バインディングマニュアル』に記述されています。

4.4.1 ノンプリエンプティブタスク

完全なプリエンプティブタスクは、自分より優先度の高いタスクによりプリエンプト（強制排除）される可能性がありますが、タスクを「ノンプリエンプティブ」なタスクとして RTA-OSEK GUI で宣言することにより、そのタスクが他のタスクにプリエンプトされないようにすることができます。[1]

タスクをノンプリエンプティブとして宣言した場合、そのタスクが他のタスクにプリエンプトされることはありません。ノンプリエンプティブタスクが実行状態に遷移するとと、4.10 項で説明する `Schedule()` をそのタスク内で呼び出さない限り、処理終了まで実行を続けてからターミネートします。しかし ISR からそのノンプリエンプティブタスクに割込みをかけることはできます。

ただしノンプリエンプティブタスクについては、他の方法を利用して同じ効果を得ることができ、また他の方法を用いた方がレスポンスのよいシステムになります。ノンプリエンプティブタスクに代わるこれらの方法には以下のようなものがあげられますが、詳細については後述します。

- 標準リソースを使用して、データやデバイスへのアクセスをシリアル化する
- 内部リソースを使用して、他のどのタスクがプリエンプションを行えないかを正確に指定する

4.4.2 多重起動（’multiple activation’）

一般的に、タスクを起動する際はサスペンド状態のタスクを起動しますが、場合によっては、同じタスクを何回も重複して起動する必要があったり、連続して起動する最短の間隔がタスクの実行に必要な時間よりも短いようなシステムを実現しなければならない場合もあります。

これは、タスクがレディ状態や実行状態になっている時にそのタスクを起動することになり、通常、その「起動」は無効となってしまいます。

このような状態を回避するため、タスクに適切な多重起動の最大数を設定する必要があります。

重要

OSEK OS 規格に従い、この機能は基本タスクにのみ利用できます。拡張タスク用に多重起動を指定することはできません。

---

1. ただし、ノンプリエンプティブタスク自身が API 関数 `Schedule( )` を呼び出すことにより、自分より優先度の高いタスクがレディ状態になっている場合に強制的にタスク切り替えを発生させるということは可能です。

RTA-OSEK GUI を使用して、一度に行うことのできるタスク起動の最大数を指定します。図 4-5 の例では、起動の最大数を 10 に設定しています。

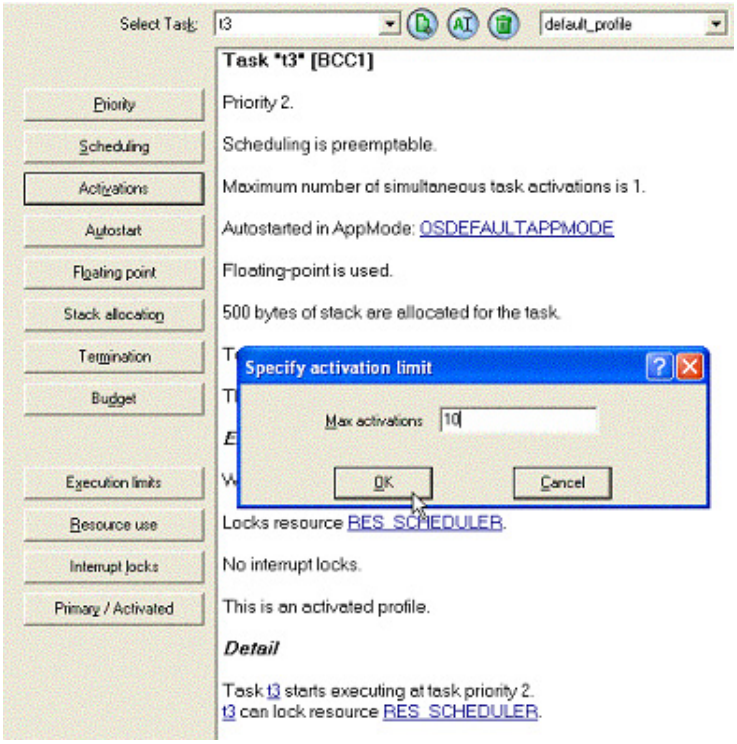


図 4-5 起動の最大数を指定する

多重起動が設定されたタスクは、RTA-OSEK によって「BCC2」タスクであると認識されます。アプリケーションの分析を行うと、RTA-OSEK は各 BCC2 タスクに必要な多重起動キューの最大サイズを計算します。

複数の BCC2 タスクが同じ優先度を共有する場合、RTA-OSEK は保留された「起動」を FIFO キュー内に保持します。OSEK アプリケーション内で 1 つの優先度を占有する BCC2 タスクの場合は、RTA-OSEK はキューイング機構を自動的に最適化し、「カウントされる起動」（'counted activation'）を行います。「カウントされる起動」は、FIFO による起動よりもはるかに効率的であるため、可能な限りそちらを使用することが推奨されます。

4.4.3 タスクの自動起動

タスクが**自動起動**（'autostarted'）されるように設定すると、オペレーティングシステム起動時、タスクは `StartOS()` の処理中に自動的に起動されます。

起動後、実行されてターミネートする基本タスクの場合、自動起動が行われると、1 回だけ実行された後にサスペンド状態に戻り、再度起動できる状態になります。

一般的に自動起動は、イベントを待つタスク用に使用されます。自動起動によって、タスクを起動するコードを作成する必要がなくなるためです。

RTA-OSEK GUI 上で、あるタスクが特定のアプリケーションモードでのみ自動起動されるように指定することができます。それには、アプリケーションモードを選択して自動起動したいタスクを指定します。

図 4-6 のように指定すると、デフォルトアプリケーションモードにおいて、`t2` と `t3` が自動起動されます。

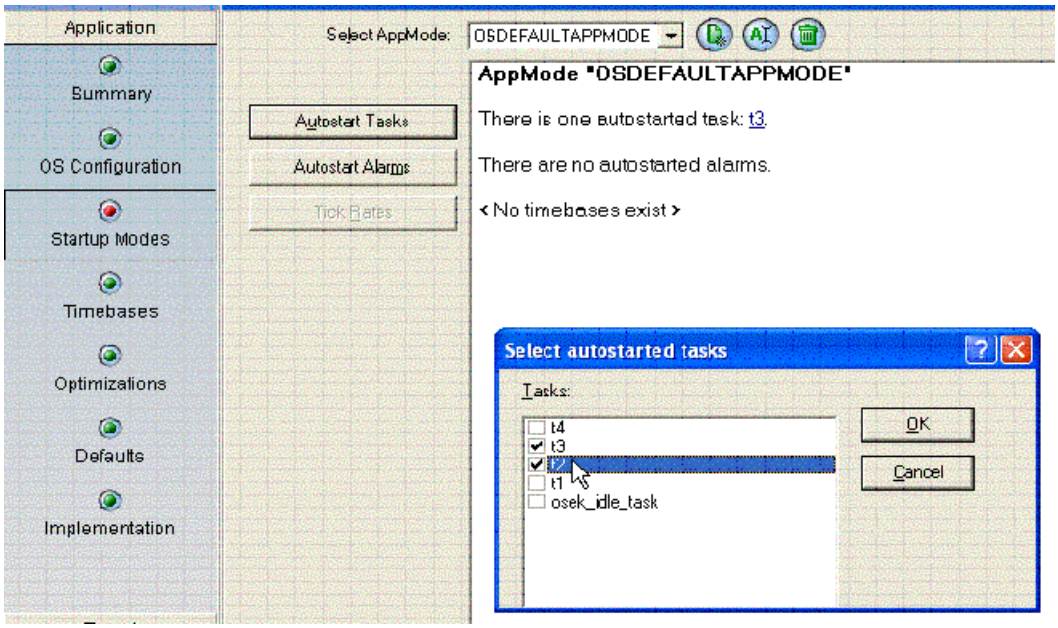


図 4-6 所定のアプリケーションモードでタスクを自動起動する

4.5 タスクの実装

タスクは、RTA-OSEK コンポーネントに呼び出されることによって何らかのシステム機能を実現する C 関数のようなものです。他の C 関数とまったく同様に、別のタスクから呼び出すことができます。

重要

タスクのエントリ関数用の C 関数プロトタイプは RTA-OSEK によって生成されるヘッダファイル内に含まれるため、ユーザーが宣言する必要はありません。各タスク用のファイルにはそのタスク固有の宣言が含まれているので、必ず各タスクに対応するヘッダファイルをインクルードしてください。

タスクが実行を開始すると、そのタスクの処理はタスクの**エントリ関数**から始まります。タスクのエントリ関数は、コード例 4-1 に示される構文で作成されます。

```
TASK(task_identifier)
{
   /* Your code */
}
```

コード例 4-1 タスクのエントリ関数

基本タスクはシングルショットタスクなので、固定的なエントリポイントから処理が開始され、処理が完了するとターミネートします。

コード例 4:2 は、`Task1` という基本タスクのコードを示しています。

```
/* Include header file generated RTA-OSEK. */
#include “BCC_Task.h”

TASK(BCC_Task) {
```

```
  do_something();
  /* Task must finish with TerminateTask()
     or equivalent. */
  TerminateTask();
}
```

コード例 4-2 基本タスク

次に、コード例 4-2 とコード例 4-3 を比べてみます。コード例 4-3 に示されるように、拡張タスクは必ずしもターミネートする必要はなく、イベントを待ちながらループし続けることができます。

```
/* Header file generated by RTA-OSEK */
#include “ECC_Task.h”
TASK(ECC_Task) {

  InitialiseTheTask();

  while (WaitEvent(SomeEvent)==E_OK) {
    do_something();
    ClearEvent(SomeEvent);
  }

  /* Task never terminates. */
}
```

コード例 4-3 イベントを待つ拡張タスク

4.6 タスクの起動

タスクが実行されるためには、まずそのタスクが**起動**される必要があります。起動により、タスクがサスペンド状態からレディ状態に移行するか、またはタスクの多重起動が設定されている場合はレディタスクキューにエントリが 1 つ追加されます。タスクは、1 回の起動に対して 1 回実行されます。最大起動数を超えるとエラーが発生し、ユーザーアプリケーションが（標準ビルドの場合でも）`E_OS_LIMIT` エラーを生成します。

レディ状態になったタスクは、レディタスクの中で優先度が最高のタスクになると実行状態に移行し、RTA-OSEK がそのタスクのエントリ関数を呼び出します。

重要
タスクを起動しても、その時点でそのタスクがすぐに実行されるとは限りません。

タスクの起動は、タスクまたは ISR から行えます。タスクを ISR から起動した場合、起動されたタスクは、レディ状態に移行します。RTA-OSEK は、ISR の処理が完了し、そのタスクよりも優先度の高い、レディ状態または実行状態のタスクがすべてターミネートした時点で、そのタスクを実行状態に移行させる必要があるかどうかをチェックします。

タスクを別のタスクから起動した場合、その挙動はタスク優先度の相対関係により異なります。一般に、起動されたタスクの方が起動を行ったタスクよりも優先度が高い場合は、新たに起動されたタスクが現在のタスクをプリエンプト（強制排除）します。そうでない場合は、新たに起動されたタスクは、現在のタスクがターミネートして自分がレディタスクの中で優先度が最高のタスクになるまで待ちます。

図 4-7 は、このプリエンプションの例を示しています。この例では、実行中であった Task1 が、それよりも優先度が高い Task2 というタスクによりプリエンプトされています。そして Task2 が最後まで処理を実行した後、Task1 がプリエンプトされた時点から処理を再開して最後まで実行します。

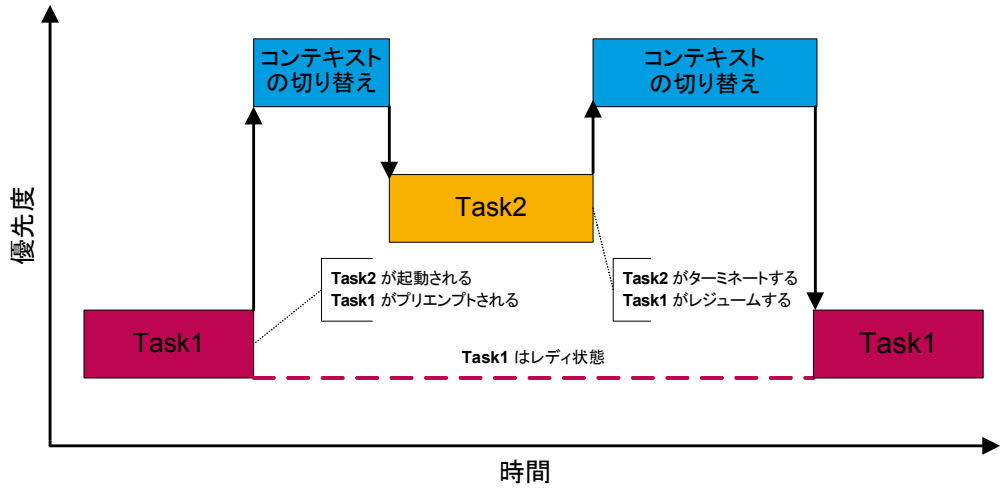


図 4-7 実行中のタスクを優先度の高いタスクがプリエンプトする

適切に設計されたリアルタイムシステムにおいては、タスクが自分より優先度の高いタスクを起動するというのは稀なケースです。通常、外部からのスティミュラスに対応するのは ISR で、ISR が実際にレスポンスを発行するタスクを起動し、次にそのタスクが優先度の低いタスクを起動して、デッドラインの長いレスポンス処理を実行します。

この条件に従うことが、RTA-OSEK の最適化の 1 つにつながります。「ユーザータスクが自分より優先度の高いタスクを起動することが絶対にない」ということを明確に指定しておけば、プリエンプションの発生の有無を API 関数内で調べる必要がなくなるので、RTA-OSEK コンポーネントは大量の内部コードを省略できます。

4.6.1 直接起動

タスクの起動にはいくつかの方法があります。タスク起動の基本的なメカニズムとして、API 関数 `ActivateTask(TaskID)` を使用してタスクを直接起動する方法があります。

`ActivateTask()` は、指定されたタスクを起動してレディ状態にします。

また `ChainTask(TaskID)` は、呼び出し元のタスクをターミネートし（4.7 項を参照してください）、指定されたタスクを起動してレディ状態にします。

API 関数名	説明	静的バージョン
`ActivateTask()`	タスクまたは ISR がこの関数を呼び出して、タスクを直接起動することができます。	`ActivateTask_TaskID()` 静的バージョンでは、RTA-OSEK は呼び出しタスクと起動されたタスクの優先度の相対関係に基づいて、生成されたコードを最適化することができます。
`ChainTask()`	タスクがこの関数を呼び出して、実行中のタスク（自分自身）をターミネートし、指定したタスクを起動することができます。	`ChainTask_TaskID()`

4.6.2 間接起動

上述の「直接起動」とは異なる以下のような OSEK および RTA-OSEK メソッドを用いて、タスクを間接的に起動することができます。これらのメソッドに関する詳しい情報は、本書内で後述します。

- **イベントによる起動**
  システム内のイベントにタスクを割り当て、イベント発生ごとにタスクが起動されるようにすることができます。
- **メッセージによる起動**
  システム内のメッセージにタスクを割り当て、メッセージが送信されるたびにタスクが起動されるようにすることができます。
- **アラームによる起動**
  システム内のアラームにタスクを割り当て、アラームが満了するたびにタスクが起動されるようにすることができます。
- **スケジュールテーブルによる起動**
  システム内のスケジュールテーブルにタスクを割り当て、スケジュールテーブルの各ポイントにおいてタスクが起動されるようにすることができます。
- **周期的スケジュールによる起動**
  周期的スケジュールを作成すると、1 つまたは複数のタスクに対して周期的な起動パターンが作成されます。このパターンに基づき、RTA-OSEK コンポーネントによって各タスクが確実に起動されます。
- **計画的スケジュールによる起動**
  計画的スケジュールを作成する際は、1 つまたは複数のタスクに対して周期的な起動パターンを定義します。このパターンに基づき、RTA-OSEK コンポーネントによって各タスクが確実に起動されます。

4.6.3 タスクの高速起動

OSEK の仕様に基づき、RTA-OSEK コンポーネントは、タスク起動を行う API コールが行われるたびにタスクの起動数をチェックし、起動限界を超えると `E_OS_LIMIT` エラーを発行しますが、このチェックには、API コール `ActivateTask()` の倍の実行時間を要します。

RTA-OSEK Planner を使用してアプリケーションがスケジューラブルであることが確認されれば、それは、ランタイムにおいて `E_OS_LIMIT` が絶対に発生しない、ということがオフラインで判明されたことを意味します。この状態においては、RTA-OSEK は、`E_OS_LIMIT` チェックを行う必要のない「高速タスク起動」を行うことできます。

「高速タスク起動」は、図 4-8 のように RTA-OSEK GUI のアプリケーション最適化オプションとして設定します。

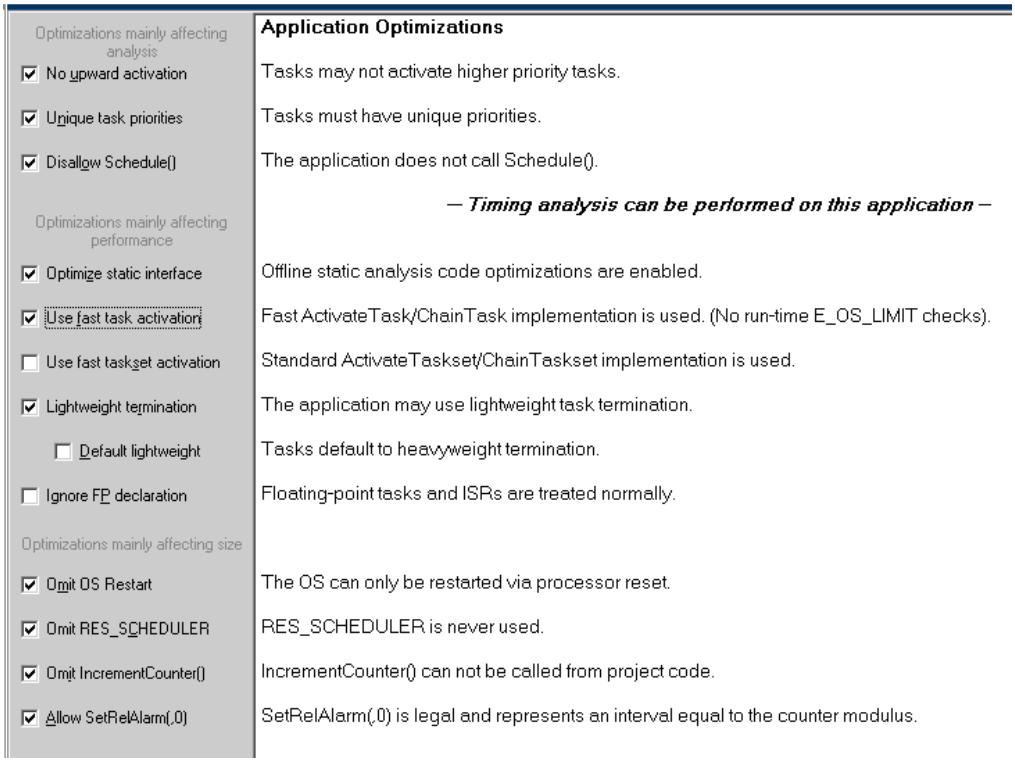


図 4-8 高速タスク起動を使用する

4.7 タスクのターミネーション

OSEK において、ターミネートするタスクは必ず API コールを行って OS にターミネーションを通知する必要があります。

OSEK 規格には、タスクターミネーション用に以下の 2 つの API 関数が定義されています。どのタスクをターミネートする場合でも、このいずれかを使用して行います。

- `TerminateTask()`
- `ChainTask(TaskID)`

タスクの終了時に上記のいずれかを呼び出すことにより、RTA-OSEK コンポーネントは、レディ状態のタスクの中から次に実行するタスクを正しく選択することができます。

タスクが `TerminateTask()` を呼び出すと、そのタスクは強制的にサスペンド状態になり、その後、レディ状態にあるタスクの中から優先順位の最も高いものが実行されます。

`ChainTask(TaskID)` も呼び出し元タスクがターミネートしますが、この場合は、API コールで指定されたタスクが起動されます。これは、`TerminateTask()` と `ActivateTask(TaskID)` を続けて実行したのと同じことになります。これにより、指定されたタスクがレディ状態になります。

重要

`ChainTask()` は、タスクのエントリ関数の最終行でしか実行できません。

4.7.1 ヘビーウェイトターミネーションとライトウェイトターミネーション

OSEK オペレーティングシステム規格では、深くネストしている関数呼び出しの中の任意の時点で、ターミネーション用 API をタスクから呼び出すことができることになっています。

コード例 4-4 では、タスクのエントリ関数が他の関数を呼び出し、その関数がさらに別の関数を呼び出しています。

```
/* Include Header file generated by RTA-OSEK */
#include “TaskA.h”
```

```
void Function1(void) {
  ...
  Function2();
  ...
}

void Function2(void) {
  if (SomeCondition) {
    TerminateTask();
  }

TASK(TaskA) {

  /* Make a nested function call. */
  Function1();

  /* Terminate the task in the entry function*/
  TerminateTask();
}
```

コード例 4-4 タスクをターミネートする

コード例 4-4 では、`Task1` が実行されると `Function1()` が呼び出され、`Function1()` はさらに `Function2()` を呼び出します。`Function2()` には、呼び出し元タスク（この例では `Task1`）をターミネートするコードが含まれています。

この例は OSEK 規格には準拠していますが、これは `goto` 文のようなものであるため、プログラミングの方法としては好ましいものではありません。そのため RTA-OSEK では、スタックサイズを大幅に削減してパフォーマンスを向上するために**シングルスタック**という機構を採用しています。

RTA-OSEK ではタスクのターミネーションタイプを以下の 2 つのタイプに分類しています。

- **ライトウェイトターミネーション**（'Lightweight termination'）
  タスクのエントリ関数からしかターミネーション API を呼び出さない方法です。
- **ヘビーウェイトターミネーション**（'Heavyweight termination'）
  ネストしている関数の中からターミネーション API を呼び出します。

シングルスタック機構においては、ライトウェイトターミネーションによってターミネートするタスクは、単純にエントリ関数から戻り、`TerminateTask()` 内では何も処理を行う必要はありません。それに対してヘビーウェイトタスクの場合、RTA-OSEK は、タスクがエントリ関数以外の場所でターミネートした時にスタックをクリアできるようにするための情報を保存しておく必要があります。これは、通常は `setjmp` と `longjmp` のペアを使用して行われます。

タスクがエントリ関数でしかターミネートしない場合には、この情報を保存する必要はありません。タスクがライトウェイトであることを定義しておくと、RTA-OSEK GUI はこの情報を保存するコードを生成しない設定となるため、スタックスペースを節減でき、さらにタスクのターミネーションが一般的な C 関数から戻る場合と同じ速さで実行されます。

RTA-OSEK GUI で、デフォルトのターミネーションタイプをライトウェイトかヘビーウェイトのいずれかに設定することができ、個々のタスクの設定において、このアプリケーションデフォルトを使用するかどうかを設定できます。

通常は、デフォルトのターミネーションタイプを「ライトウェイト」に設定し、各タスクのターミネーションを可能な限り「デフォルト」に設定しておくことをお勧めします。図 4-9 はデフォルトのターミネーションタイプの設定方法を示しています。

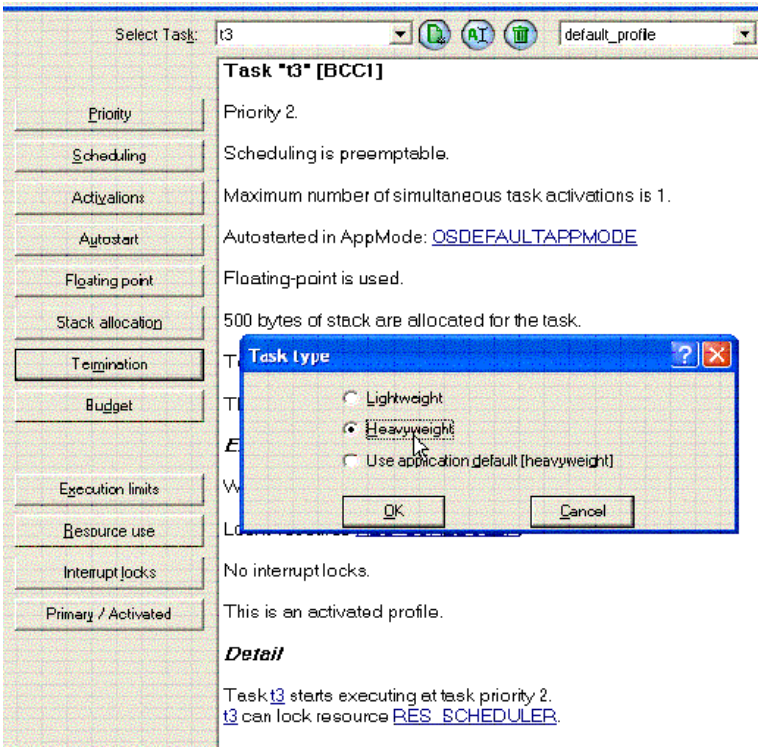


図 4-9 デフォルトのターミネーションタイプを設定する

重要

ライトウェイトタスクの最適化を行うには、タスクをコンパイルするときにタスク専用のヘッダファイル（ファイル名は `<TaskID>.h`）をインクルードする必要があります。汎用的な `osek.h` や `oseklib.h` では最適な挙動が得られません。汎用ヘッダファイルをインクルードする際は、そのタスクをライトウェイトターミネーションに設定しないでください。

4.8 アイドルタスク

プリエンプティブなオペレーティングシステムにおいては、タスクや ISR がまったく実行されていないときにも何らかの処理が行われます。OSEK では、このような処理は**アイドルメカニズム**によって行われます。RTA-OSEK の場合、アイドルメカニズムは `osek_idle_task` という**アイドルタスク**を用いて実現されています。

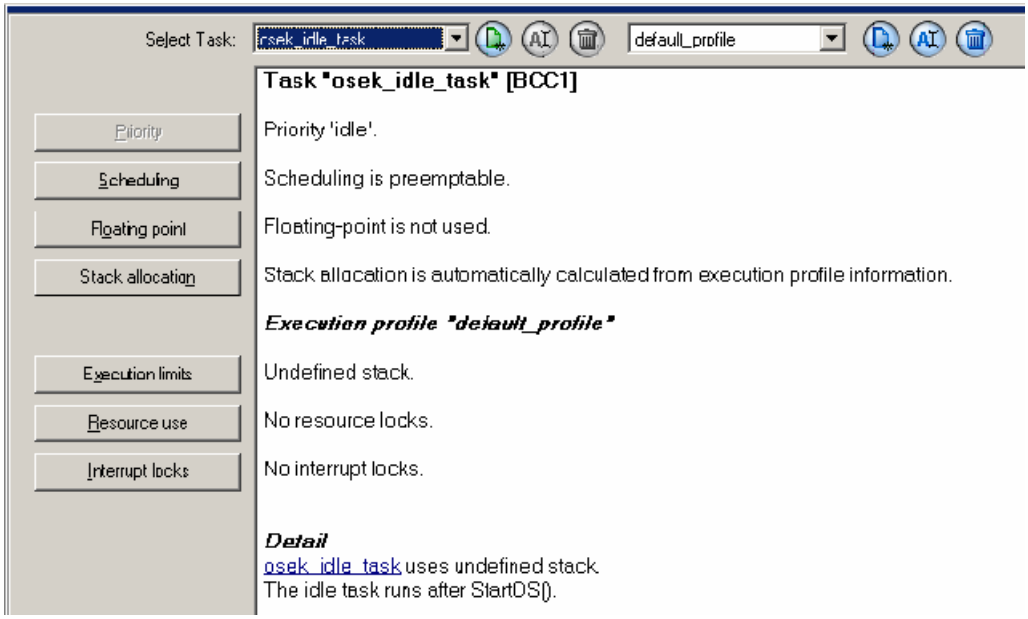


図 4-10 osek_idle_task

`osek_idle_task` は、一般のタスクと以下の点が異なっています。

- 起動できません。
- ターミネートできません。
- チェーンできません。
- OSEK の内部リソースを使用できません。

`osek_idle_task` の優先度はシステム内のどのタスクよりも低くなっています。標準リソース、リンクリソース（'linked resource'）、メッセージリソースを使用でき、ターゲットハードウェアで使用可能なユーザータスクの最大数（通常は 16 か 32）のカウントには含まれません。

RTA-OSEK はアイドルタスク用に `osekmain.h` という専用のヘッダファイルを生成します。通常は、`osek_idle_task` のコードが含まれているファイルにこのヘッダファイルをインクルードしてください。

アイドルタスクのコードは、`StartOS()` からリターンした後に実行されます。通常、これらはアプリケーションの起動関数内のコードとなりますす。起動関数の例をコード例 4-5 に紹介します。

```
#include "osekmain.h"
OS_MAIN(main)
{
   /* System hardware initialization. */
   StartOS(OSDEFAULTAPPMODE);
   for (;;) {
   /* This loop body is the osek_idle_task. */
   }
}
```

コード例 4-5 アプリケーションの起動関数の例

アイドルタスクは拡張タスクなので、イベントを待つことができます。通常の拡張タスクではなくアイドルタスクを使用してイベントを待つようにすると、OS のオーバーヘッドがはるかに少なくなります。

4.9 拡張タスクの使用

RTA-OSEK では、独自の方法でシングルスタックモデルを拡張し[1]、基本タスクのパフォーマンスに影響与えずに OSEK 拡張タスクを使用できるようになっています。

- サスペンド状態 → レディ状態
  タスクがレディキューに追加されます。
- レディ状態 → 実行状態
  タスクがディスパッチされますが、基本タスクの場合のようにコンテキストはスタックの現在の最上部に置かれず、自分よりも優先順位の低いすべてのタスクを含めたワーストケースのプリエンプション深度が計算され、その位置にコンテキストが置かれます。
- 実行状態 → レディ状態
  拡張タスクはプリエンプトされます。基本タスクにプリエンプトされた場合、そのタスクは通常どおりスタックの最上部からディスパッチされます。しかし別の拡張タスクにプリエンプトされた場合は、そのタスクは、自分よりも優先順位の低いすべてのタスクを含めたワーストケースのプリエンプション深度からディスパッチされます。
- 実行状態 → ウェイト状態
  タスクの「ウェイトイベントスタック」コンテキスト（OS コンテキスト、ローカルデータ、関数呼び出し用スタックフレームなどを含みます）が内部 OS バッファに保存されます。
- ウェイト状態 → レディ状態
  タスクがレディキューに追加されます。
- 実行状態 → サスペンド状態
  タスクの「ウェイトイベントスタック」コンテキストが、内部 OS バッファからスタック（自分よりも優先順位の低いすべてのタスクを含めたワーストケースのプリエンプション深度）に書き戻されます。

このような処理により、拡張タスクの管理のための負荷は、拡張タスクが実行状態に移行する際にのみ適用されるようになっています。これによって、基本タスクは純粋な基本タスクとして動作できるため、拡張タスクの存在による基本タスクのパフォーマンス低下を防ぐことができます。

拡張タスクのライフサイクルのキーとなる部分は、ワーストケースのプリエンプション深度と、スタックコピーのオン／オフです。

ワーストケースのプリエンプションポイントからのディスパッチを行うことにより、ウェイトしていた拡張タスクがレジュームする際、常に、メモリ上の同じ位置でローカル変数を使用することができます。どのようなパターンで低優先度のタスクがプリエントされた場合でも、拡張タスクのディスパッチポイントを超えることはありません。図 4-11 に拡張タスク D のライフサイクル（ディスパッチ→ウェイト→レジューム）の例を示します。

[1] UK patent: 0219936, US patent: 10/242,482

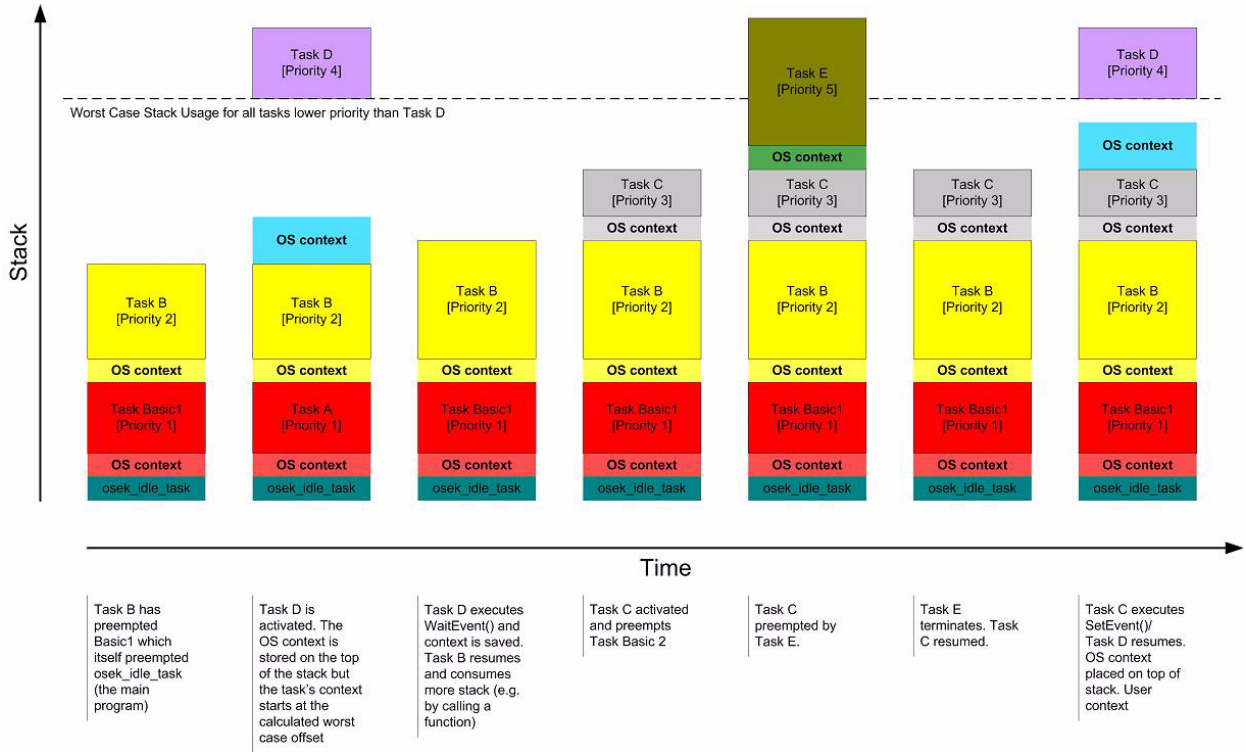


図 4-11 拡張タスクのサイクル - ディスパッチ／ウェイト／レジューム

コピーのオン／オフ機能は、拡張タスクのスタックのコンテキストの復旧を行うためのものです。拡張タスクがウェイト状態にある時に優先度の高いタスクや ISR が実行状態になると、これらのタスクまたは ISR は、ワーストケースのプリエンプションポイントを超えてスタックを消費する可能性があり、その場合、拡張タスクのコンテキストを上書きしてしまいます。これは、「ワーストケースのプリエンプションポイント」は自分自身よりも低い優先度のタスクのみを考慮したものであるためです。しかし RTA-OSEK は「固定優先度に基づくプリエンプティブスケジューリング」を行うため、拡張タスクがレジュームする時点において、それより高い優先度のタスクがレディ状態になっていることはありません。つまり、自分より高い優先度のタスクがレディ状態になっている時にそのタスクがレジュームすることはありません。

4.9.1 スタックアロケーションの指定

基本タスクしか含まないシステムの場合、RTA-OSEK に対してスタックアロケーションを指定する必要はありません。リンカやロケータにおいて、アプリケーション全体で使用できる十分なサイズのスタックを割り当てるだけですみます。これはシングルスタック機構のメリットであるといえます。

それに対し、拡張タスクを使用するアプリケーションの場合は、リンカでの指定以外に、RTA-OSEK に対しても、最上位の拡張タスクよりも優先度の低いすべてのタスク（基本タスクを含む）について、それぞれスタックアロケーションを指定する必要があります。RTA-OSEK はこの情報に基づいて各拡張タスクの「ワーストケース プリエンプションポイント」をオフライン計算します。

> **重要**
> ここで指定したスタック情報は、RTA-OSEK がワーストケース プリエンプションポイントを算出する際にのみ使用されます。これ以外に、アプリケーションスタックサイズも通常どおり指定する必要があります。

スタックアロケーションを設定するには、**Task → Task Data → Stack Allocation** を選択して以下のダイアログボックスを開きます。

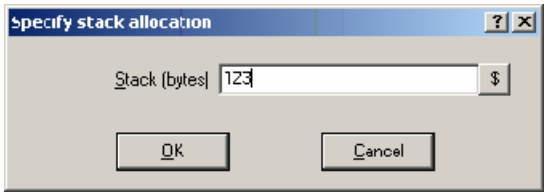


図 4-12 スタックアロケーション設定

拡張タスクの場合、続いて第 2 のダイアログボックスが開くので、そこで `WaitEvent()` スタックを指定します。この値は、`WaitEvent()` が呼ばれた時に使用されるスタックサイズ（単位：バイト）です。デフォルトでは“Automatic”という値が設定されていて、この場合、ワーストケーススタックアロケーションと同じサイズの RAM バッファが割り当てられます。

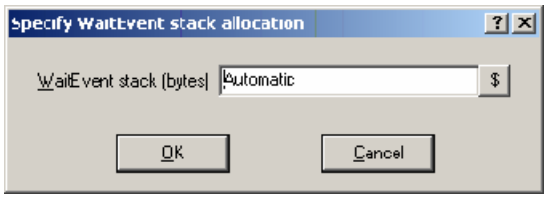


図 4-13 WaitEvent( ) スタックの設定

しかし実際には、ほとんどの拡張タスクはエントリ関数内で `WaitEvent()` を実行するので、エントリ関数内のローカルデータ用のスペースのみで十分です。そのため、ここで `WaitEvent()` を呼び出すポイントにおけるワーストケースのスタック深度を正確に指定することにより、スタックサイズを節約することがきます。

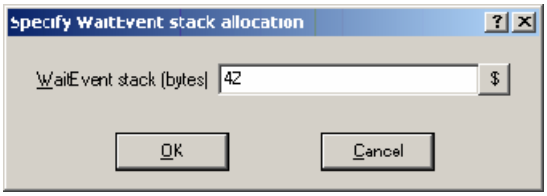
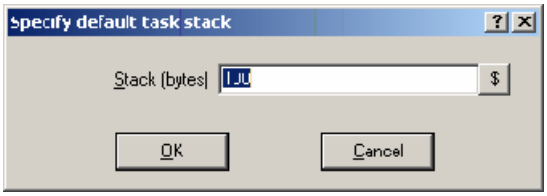


図 4-14 WaitEvent( ) スタックの設定

デフォルト値の使用

使用メモリの最適化のためにはタスクごとにスタックサイズを指定する必要がありますが、すべてのタスクで使用されるグローバルデフォルト値を設定することもできます。それには **Application → Defaults → Default Task Stack** を選択します。



スタックアロケーションが指定されていないタスクについては、RTA-OSEK はこのデフォルト値を使用して以下のことを行います。

1. ワーストケース スタックオフセットの算出
2. `WaitEvent()` 呼び出し時に保存／復帰を行う領域の設定
3. スタック監視（設定時のみ）

RTA-OSEK Builder を実行する際、どのタスクについてデフォルト値を使用するかが尋ねられます。

4.9.2 スタックのベースアドレスの指定

算出されたワーストケース ディスパッチポイントは、スタックのベースアドレス、つまりメインプログラム開始時のスタックポイントからの相対位置です。これらのオフセットは、ROM 上の拡張タスクのコントロールブロック内に保存され、ランタイムにおいて基本スタックポインタに加算されます。

このため、RTA-OSEK に対してスタックポインタのベースアドレスを指定しておく必要があり、その方法はターゲットに応じて異なります。ターゲットによっては、コンパイラツールチェーンによって定義されている初期スタックポインタの名前を RTA-OSEK が自動的に使用するものや、あるいは、リンク時にスタックのベースアドレスを指定する必要があるものもあります。ご使用のターゲットに関する詳細な情報は、サンプルアプリケーションや、『RTA-OSEK バインディングマニュアル』を参照してください。

4.9.3 スタックに関するエラーの扱い

RTA-OSEK に対して設定したスタックアロケーションが適切でない場合（たとえばサイズが小さすぎる場合）、これによってランタイムエラーが発生する可能性があります。このようなエラーの発生を防ぐため、RTA-OSEK コンポーネントは、拡張タスクのスタック管理において問題が検知されるたびに `StackFaultHook()` を呼び出します。

`StackFaultHook()` はユーザー定義のコールバック関数で、拡張タスクの存在するシステムにおいては必ず用意しておく必要があります。内容は、以下のような構造にしていください。

```
if defined(OSEK_ECC1)
 || defined(OSEK_ECC2C)
 || defined(OSEK_ECC2F)
OS_HOOK(void)
StackFaultHook(SmallType StackID,
               SmallType StackError,
               UIntType  Overflow)
{
  /* Identify problem */
   for(;;) {
     /* Do not return! */
   }
}
#endif /*OSEK_ECC1||OSEK_ECC2C||OSEK_ECC2F*/
```

このフックには、RTA-OSEK によって以下の 3 つのパラメータが渡されます。

1. StackID

   エラーが発生したスタックを識別するための ID です。一般のターゲットはシングルスタックのため、この値は常に 0（ゼロ）ですが、一部のターゲットは強制的に複数のスタックを使用するため、そのようなターゲットにおいては、RTA-OSEK がシングルスタック機構を使用していても、スタックの数は複数となります。

2. StackError

   RTA-OSEK が検知したエラーのタイプです。3 つのエラータイプに応じて以下の値が渡されます。

   - OS_EXTENDED_TASK_STARTING: RTA-OSEK が拡張タスクをディスパッチした際に、スタックポインタが、算出されたワーストケースディスパッチポイントよりも高くなりました。
   - OS_EXTENDED_TASK_RESUMING : RTA-OSEK が拡張タスクをレジュームした際（ウェイト状態のタスクが待っているイベントが、`SetEvent()` によって発行された時など）に、スタックポインタが、算出されたワーストケースディスパッチポイントよりも高くなりました。
   - OS_EXTENDED_TASK_WAITING : 保存されたコンテキストのサイズが、`WaitEvent()` バッファのサイズよりも大きくなりました。

3. Overflow

   現在のスタックポインタがワーストケースディスパッチポイントをどれだけ超過したか、または保存するコンテキストのサイズが `WaitEvent()` 用に設定されたスタックサイズをどれだけ超過したかを、バイト数で通知します。

4.10 OSEK における協調スケジューリング

あるタスクがノンプリエンプティブに実行されている場合、他のタスク（そのタスクより優先度の高いタスクを含みます）は実行できません。しかしその場合でも、ノンプリエンプティブタスク内にリスケジューリングを行うことのできるポイントを明示的に定義しておくことにより、そのタスクより優先度の高いタスクのレスポンスタイムを短縮することができます。

すべてのタスクがノンプリエンプティブに稼働し、かつ各タスク内にリスケジューリング用のポイントが定義されているシステムは、**協調スケジューリング**システムと呼ばれています。

`Schedule()` を使用することにより、ノンプリエンプティブタスクと内部リソースを使用するタスクによるプリエンプションの制約を、一時的に解消することができます。

`Schedule()` が呼び出されると、呼び出し元のタスクより優先度の高いすべてのレディタスクが実行できるようになります。また、`Schedule()` は、呼び出し元タスクより優先度の高いタスクがすべてターミネートするまで、呼び出し元タスクに戻りません。

以下のコード例では、ノンプリエンプティブタスクである `Cooperative` に複数の関数呼び出しが含まれています。このタスクの実行が開始されると、各関数はプリエンプトされずに必ず最後まで実行されますが、タスク全体では、各関数呼び出し間でプリエンプトされる可能性があります。

```
#include "Cooperative.h"
TASK(Cooperative){
  Function1();
  /* Allow preemption here */
  Schedule();
  Function2();
  /* Allow preemption here */
  Schedule();
  Function3();
  /* Allow preemption here */
  Schedule();
  Function4();
  TerminateTask();
}
```

図 4-15 は、2 つのタスク（Task1 と Task2）が協調的に実行されるようすを示しています。

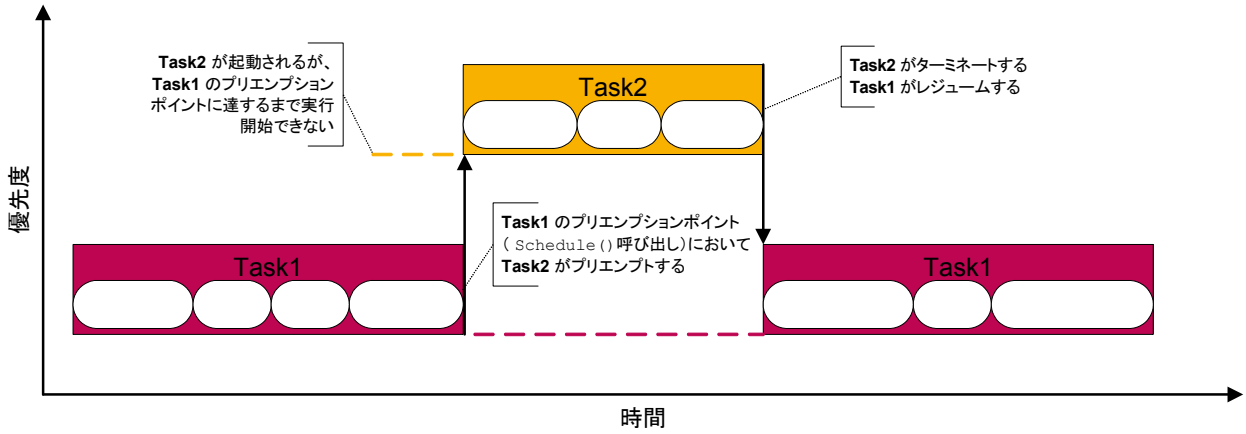


図 4-15 協調タスク

4.10.1 RTA-OSEK における協調スケジューリング

協調スケジューリングは一般的なデザインメカニズムであり、ETAS の従来のオペレーティングシステム - ERCOS^{EK} はこれをサポートしていました。RTA-OSEK V5.x では、この ERCOS^{EK} から RTA-OSEK に移行しようとするユーザーを対象に、「プロセス」と呼ばれる関数のリストからタスクボディを自動生成する機能が追加されました。

移植性

RTA-OSEK における協調タスクの自動生成機能は、OSEK 規格の範囲外です。

最小プリエンプション優先度の設定

一般に、協調タスクには他のプリエンプティブタスクよりも低い優先度が割り当てられます。**Application → OS Configuration** で **最小プリエンプション優先度**（'Minimum Preemption Priority'）を定義することにより、どのタスクを協調的に実行しどのタスクをプリエンプティブに実行するかを設定することができます。プロセスを使用するかしないかに関わらず、最小プリエンプティブ優先度よりも低い優先度のタスクは、すべて協調的にスケジュールされます。

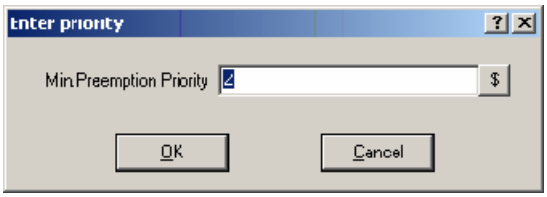


図 4-16 最小プリエンプション優先度の設定

プロセスの作成

プロセスは、図 4-19 に示されるように、RTA-OSEK の「パッケージ」ワークスペース内で作成します。必要に応じて、スケジューラビリティ分析のための実行時間を各プロセスに設定することができます。プロセスが OSEK リソースを使用する場合、そのリソースを指定し、RTA-OSEK が各リソース用に正しく上限優先度を計算できるようにする必要があります。

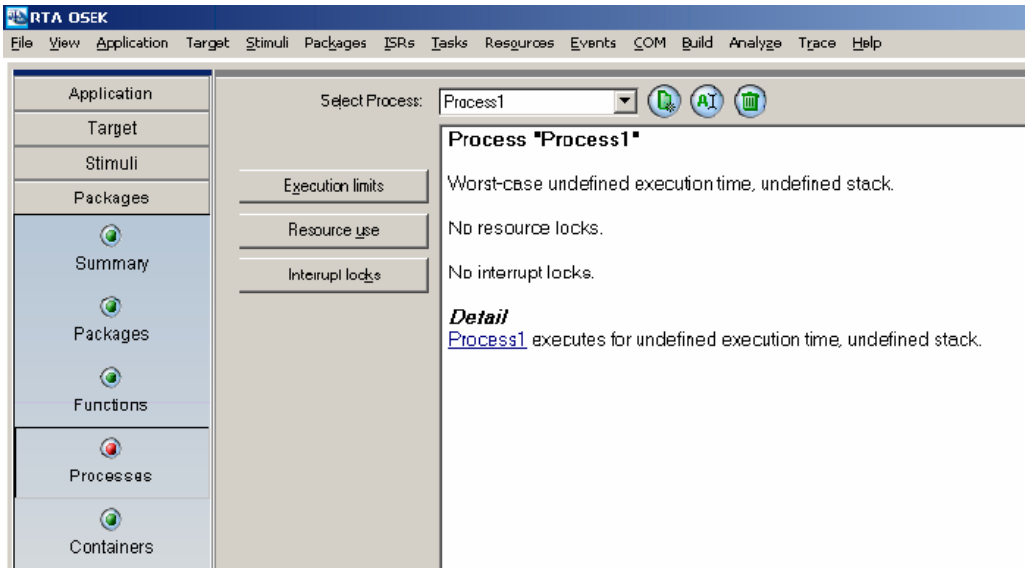


図 4-17 プロセスの作成

プロセスをコンテナに割り当てる

プロセスは直接タスクに割り当てるのではなく、各プロセスを 1 つまたは複数のコンテナに割り当てます。コンテナには同じプロセスの複数のインスタンスを含めることができます。各コンテナに含まれるプロセスは、リストに表示される順番で実行されます。コンテナにプロセスを追加するには **[+]** ボタン、削除するには **[-]** ボタンを使用し、プロセスの順番の変更は **[▲ / ▼]** ボタンを使用します。

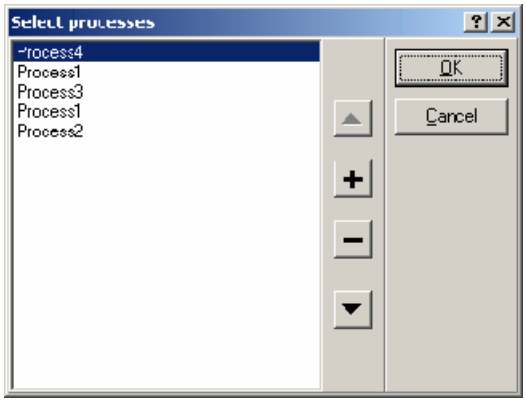


図 4-18 プロセスをコンテナに割り当てる

コンテナをタスクに割り当てる

コンテナは、タスクまたはカテゴリ 2 の ISR に割り当てることができます。通常は、1 つのコンテナをタスク／ISR に割り当て、各コンテナは 1 回のみ割り当てるようにしてください。

> **重要**
> 最小プリエンプション優先度よりも低いタスクに割り当てられたコンテナ内のプロセスのみが、強調的に実行されます。

コンテナを含むタスクまたは ISR のエントリ関数は、RTA-OSEK によって自動的に生成されます。協調スケジューリングが行われるタスクは、`Schedule()` の最適化バージョン（タスクボディにインライン展開されます）を使用します。

ただし、各プロセスの実際のコードはユーザーが記述します。プロセスは、以下のように `void-void` 関数の形式で記述してください。

```
void Process1(void) {
  /* Code implementing the process */
}
```

4.10.2 Schedule( ) API の最適化

`Schedule()` は、完全な協調システムにおいては必要ありません。この API 関数を使用しない場合は、RTA-OSEK GUI の **Application → Optimazations** で、`Schedule()` を使用禁止に設定することができます。図 4-19 を参照してください。

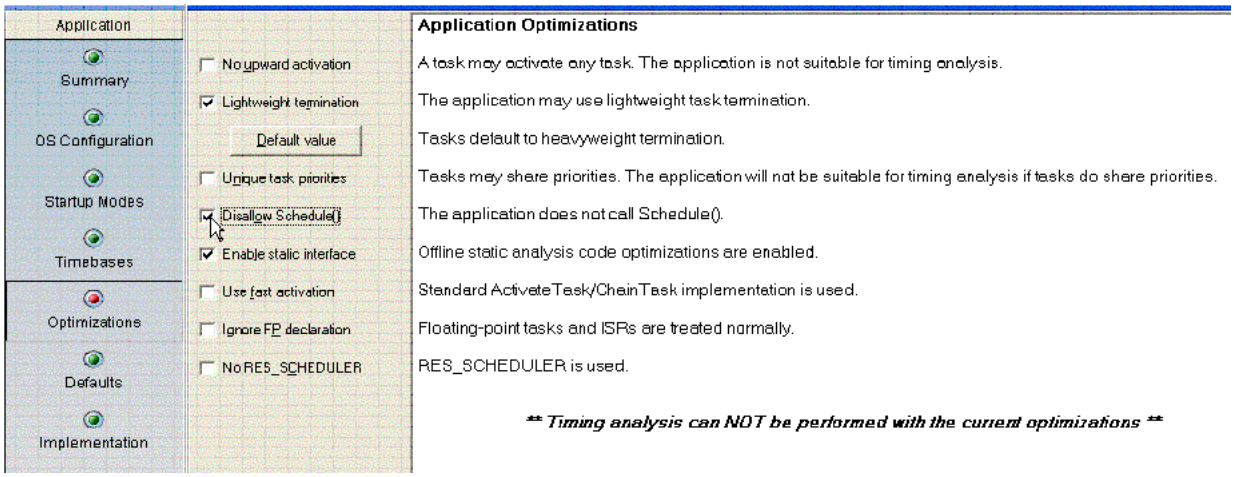


図 4-19 Schedule( ) の呼び出しを禁止する

RTA-OSEK GUI で `Schedule()` の呼び出しを禁止すると、以下のようなメリットが得られます。

- `Schedule()` が呼び出されなければ、ワーストケースのスタック使用量が少なくなります。
- タイミング分析が利用可能になります。（現時点では、`Schedule()` を呼び出すシステムのタイミング分析を行うことはできません。）

4.11 浮動小数点の使用

浮動小数点演算は、ソフトウェアのみで行うと時間がかかり、また専用のハードウェアを使用すればコストが高くなってしまいます。

このため、浮動小数点演算をフルに使用する組み込みシステムは非常にまれです。RTA-OSEK では、デフォルト状態において、アプリケーション内で浮動小数点が使用されないことが想定されています。

ユーザータスクまたは ISR で浮動小数点を使用する場合は、RTA-OSEK に対してその旨を通知することにより、プリエンプションの時点において浮動小数点演算のコンテキストが保存され、再度レストアされるようにする必要があります。

RTA-OSEK は、浮動小数点を使用するタスクや ISR のコンテキストを保存するために確保すべきメモリサイズを正確に計算することができます。これは、RTA-OSEK が、これらのタスクや ISR のワーストケースのプリエンプション深度を算出でき、保存するコンテキストの数を最適化できるためです。

この計算結果は、RTA-OSEK GUI 上で確認することができます。

図 4-20 にスタック分析の例を示します。この例では、`t1`、`t2`、`t3` が浮動小数点を使用しています。

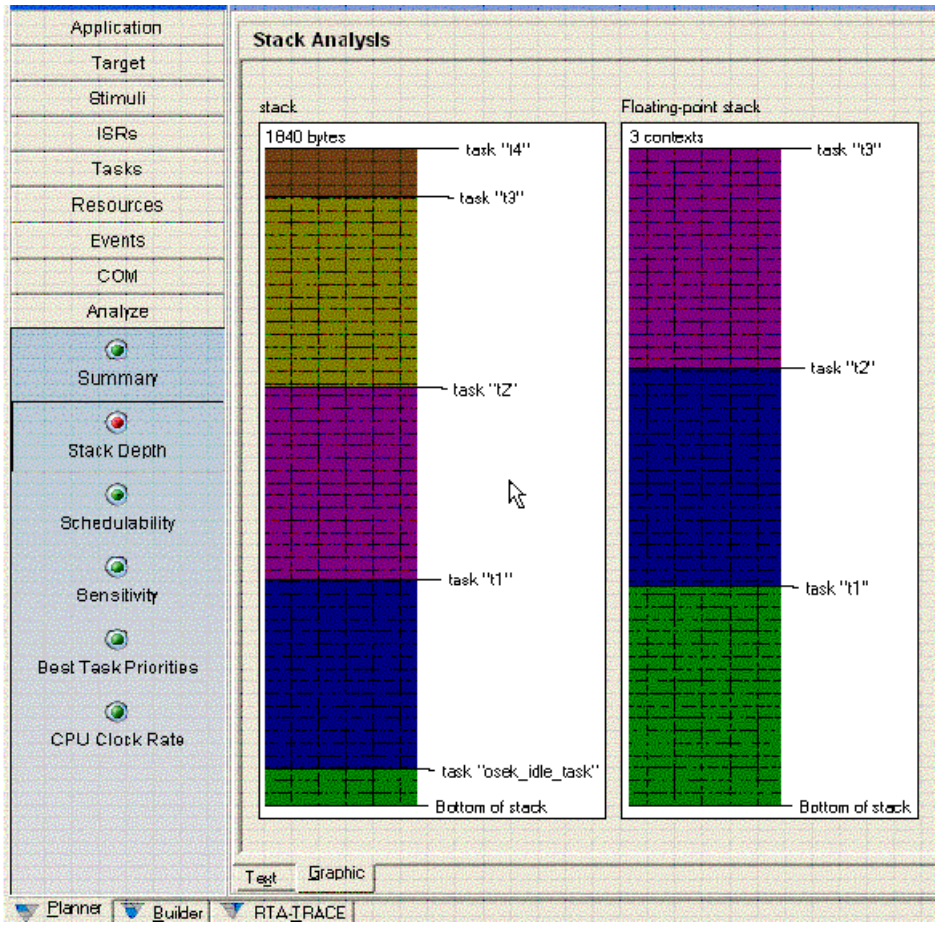


図 4-20 スタック深度の分析例

優先度が最低のタスクではレジスタの保存を行う必要がないので、浮動小数点レジスタ保存用領域は 2 つだけ確保されています。

浮動小数点を用いるタスクまたは ISR が 1 つしかない場合には、浮動小数点の保存やリストアはまったく必要なく、浮動小数点が原因で RTA-OSEK コンポーネントがアプリケーションにオーバーヘッドをかけることはまったくありません。

4.11.1 浮動小数点演算のカスタマイズ

RTA-OSEK がサポートしている各ターゲットプロセッサは、浮動小数点を扱うタスクと ISR をサポートしています。これらのターゲットには、浮動小数点演算をソフトウェアでサポートしているものと、ハードウェアでサポートしているものとがあります。

ターゲットの中には、RTA-OSEK がサポートしていない浮動小数点機構を持つものもあります。たとえば、独立した浮動小数点コプロセッサを使用しているターゲットがなどがあります。このため、RTA-OSEK には 2 つのソースファイルが用意されていて、それらを任意に変更してアプリケーションにリンクすることができます。

これらのソースファイルは `osfptgt.c` と `osfptgt.h` という名前で、`<install dir>¥<target>¥inc` フォルダ内にあります。これらのファイルを使用すると、浮動小数点の保存とリストアの実行方法を変更することができます。

4.12 タスクセット

移植性

タスクセットは RTA-OSEK に組み込まれた拡張機能で、OSEK 規格には含まれていません。

RTA-OSEK では、一度に複数のタスクを効率的に起動するために、**タスクセット** という拡張機能が採用されています。これは OSEK 規格には含まれてません。

タスクセットは、単なるタスクの集合に名前を付けたものですが、1 回の API コールでそれらのタスクを同時に起動することができます。アイドルタスク以外ならどのタスクでもタスクセットに含めることができます。また、1 つのタスクを複数のタスクセットに含めることもできます。

タスクセットは RTA-OSEK GUI で宣言します。図 4-21 では、`ts1` というタスクセットが宣言され、そこにタスク `t2` と `t4` が割り当てられています。

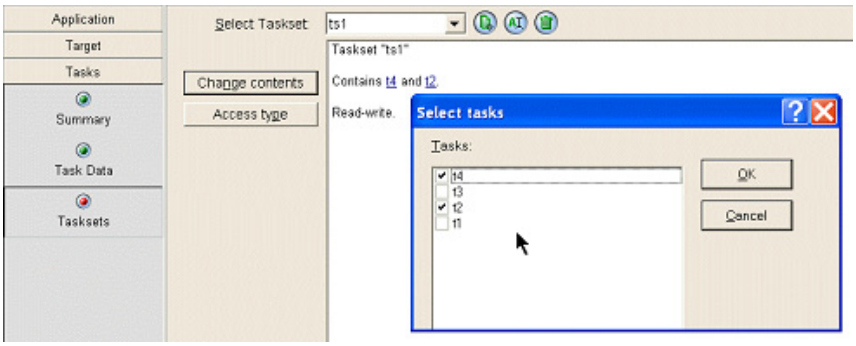


図 4-21 タスクセットを設定する

タスクセットは、アクセスタイプを**読み取り専用**（'read-only'）または**読み取り／書き込み可能**（'read-write'）にすることができます。アクセスタイプを読み取り／書き込み可能に設定しておくと、ランタイムにタスクセットの内容を変更することができます。

読み取り／書き込み可能なタスクセットの場合、ランタイムにおいてタスクの追加や削除を行えます。RTA-OSEK には、タスクセットの設定に関する API 関数（タスクの追加、削除、タスクセットのマージなど）が用意されています。タスクセットの操作に使用される RTA-OSEK コンポーネントの API 関数については、『RTA-OSEK リファレンスガイド』を参照してください。

4.12.1 タスクセットの起動

RTA-OSEK には、タスクセット内のタスクを同時に起動するための API 関数 `ActivateTaskset()` が用意されています。

複数のタスクを同時に起動する必要がある場合、タスクセットを使用することによりパフォーマンスが大きく向上します。これは、ランタイムにおいて 1 回の API コールを行えばよいので、1 つのタスクを起動する場合とオーバーヘッドが変わらないためです。

タスクセットの起動には `ChainTaskset()` も使用できます。タスクセットのチェーニングを行うと、1 回の API コールで複数のタスクにチェーニングすることができます。

API 関数名	説明	静的バージョン
`ActivateTaskset()`	タスクまたは ISR がこの関数を呼び出すことにより、タスクセット内のタスクをすべて起動することができます。	`ActivateTaskset_TasksetID()` 静的バージョンの場合、呼び出し元タスクと起動されるタスクの優先度の相対関係に基づいて、最適化されたコードが生成されます。
`ChainTaskset()`	タスクがこの関数を呼び出すことにより、実行中のタスクをターミネートして、指定されたタスクセット内のタスクを起動することができます。	`ChainTaskset_TasksetID()`

4.12.2 タスクセットの高速起動

タスクセット用 API を使用すると、RTA-OSEK の最適化機構によるタスクの起動チェックを省略できる、というメリットがあります。これは「高速タスクセット起動」と呼ばれ、E_OS_LIMIT のチェックを省略、またはチェックを行ってもエラーを発行しない「高速タスク起動」と似ています。

4.12.3 定義済みタスクセット

RTA-OSEK は、常に、以下の読み取り専用のタスクセットを生成します。

タスクセット	説明
os_all_tasks	システム内のすべてのタスクが含まれます。
osek_cc2_tasks	すべての BCC2 タスクと ECC2 タスクが含まれます。
osek_ecc_tasks	すべての ECC1 タスクと ECC2 タスクが含まれます。
os_no_tasks	メンバが含まれないからのタスクセットです。
os_ready_tasks	レディ状態のタスクと実行状態のタスクがすべて含まれます。

これらのタスクセットは、読み取り／書き込み可能なタスクセットを操作する際に役立ちます。たとえば、タスクセットのクリアなどを行う際に必要な「メンバーシップテスト」を、これらのタスクセットを使用して行うことができます。

4.13 タスク実行順序の管理

多くの場合、タスクの実行順序について制約を設けることが必要とされます。これは特に、1 つのタスクが何らかの計算を行ってからその計算で算出された値を別のタスクが使用するというような、データフローに基づいたソフトウェア設計において重要です。

実行順序の制約を設けないと、**競合状態**（'race condition'）に陥る可能性があり、アプリケーションの挙動を予測できなくなってしまいます。このため、以下のような方法でタスクの実行順序を管理します。

- 直接起動によるチェーン（4.13.1 項を参照してください。）
- 優先度レベル（4.13.2 項を参照してください。）
- ノンプリエンプティブタスク（4.4.1 項を参照してください。）

4.13.1 直接起動チェーン

直接起動チェーンを使用して実行順序を管理する場合、タスクは、ActivateTask() を呼び出すことにより、自分の後に実行されるべき別のタスク（1 つまたは複数）を起動します。

ここでたとえば、Task1、Task2、Task3 というタスクがあり、これらを Task1、Task2、Task3 の順に実行する必要があるとします。

この場合には、コード例 4-6 のようなタスクボディを作成してください。

```
#include "Task1.h"
TASK(Task1) {

  /* Task1 functionality. */
  ActivateTask(Task2);
  TerminateTask();

}
```

```
#include “Task2.h”
TASK(Task2) {

   /* Task2 functionality. */
   ActivateTask(Task3);
   TerminateTask();

}

#include “Task3.h”
TASK(Task3) {

   /* Task3 functionality. */
   TerminateTask();

}
```

コード例 4-6 直接起動チェーンを使用する

このアプリケーションをタイミング分析に適したものにするには、すべてのタスク起動の方向を下向きにしておく必要があります。つまり、どのタスクも必ず自分より優先度の低いタスクだけを起動するようにしなければなりません。

4.13.2 優先度レベルの使用

優先度レベルを使用してタスク実行順序に制約を設ければ、プリエンプティブスケジューリングポリシーの性質を利用して起動順序を管理できます。

固定優先度プリエンプティブスケジューリング下ではスケジューラは優先度が最高のタスクを必ず実行するということを 4.1 項で説明しました。複数のタスクが起動されてレディキューに入れられた場合、それらのタスクは優先度順に実行されます。これはつまり、タスク優先度を利用した実行順序の管理、ということになります。

前のコード例 4-6 の続きとして、3 つのタスクのうち、優先度が最高のタスクは `Task1` で最低のタスクは `Task3` であると仮定し、優先度レベルによる起動管理を利用するようにタスクボディを書き直してみます。

これはコード例 4-7 のようになります。

```
#include “Task1.h”
TASK(Task1) {
   /* Task1 functionality. */
   ActivateTask(Task2);  /* Runs when Task1 terminates. */
   ActivateTask(Task3);  /* Runs when Task2 terminates. */
   TerminateTask();

}

#include “Task2.h”
TASK(Task2) {
   /* Task2 functionality. */
   TerminateTask();

}
```

```
#include "Task3.h"
TASK(Task3) {
  /* Task3 functionality. */
  TerminateTask();

}
```

コード例 4-7 優先度レベルによる起動管理を使用する

この管理方法は、1 回の呼び出しでタスクセットに含まれる `Task1`、`Task2`、`Task3` を同時に起動する必要がある場合、特に役立ちます。

優先度の自動割り当てを行う場合、ユーザーが指定した優先度を変更することができます。これにより、優先度レベルによる起動管理を調整することができます。

優先度の相対順序を確実に維持するために、あるタスクについて、そのタスクよりも優先度が必ず低くなければならないタスク群（'required lower priority tasks'）があるということを指定することができます。それらのタスクは選択されたタスクよりも必ず後に実行されなければなりません。

`t1` と `t3` の優先度が `t4` よりも低くなければならない場合は、図 4-22 のように設定してください。

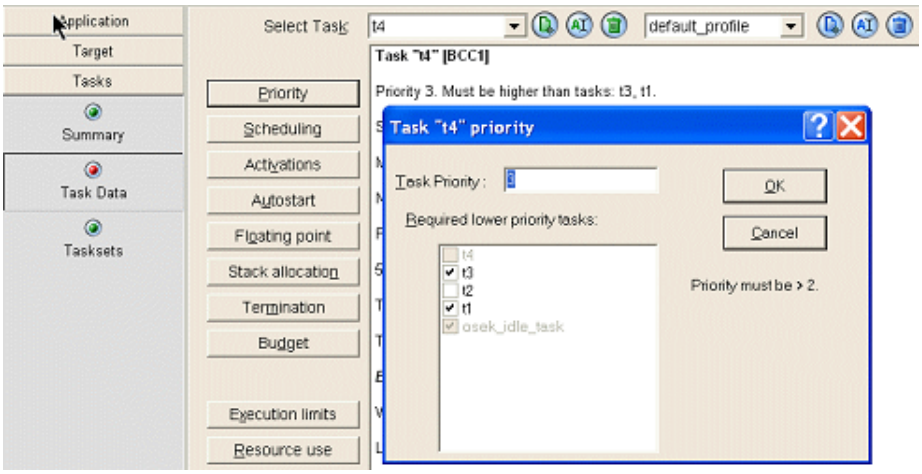


図 4-22 自分よりも優先順位を低くすべきタスクを設定する

4.14 基本タスクとの同期化

基本タスクを同期できるタイミングは、そのタスクの実行の最初と最後だけです。他の同期ポイントが必要な場合には、ユーザーが自分でそれを実装する必要があります。

たとえば、あるタスクが switch 文を使用したステートマシンとして構築されている場合、ステート変数を設定して `TerminateTask()` を呼び出し、再起動を待つことができます。コード例 4-8 にこのコード例を紹介します。

```
#include “Task1.h”

int State;

TASK(Task1) {

   switch (State) {

      case 0:
          /* Synchronization point 0. */
         State = 1;
         break;

      case 1:
          /* Synchronization point 1. */
         State = 2;
         break;

      case 2:
          /* Synchronization point 2. */
         State = 0;
         break;
   }

   TerminateTask();
}
```

コード例 4-8 基本タスクに複数の同期ポイントを設ける

4.14.1 基本タスクを擬似的にウェイト状態にする

基本タスクしか含まれないシステムにおいて、いずれかのタスクが何らかのイベントを待たなければならないような状況が発生する場合も考えられますが、4.3.2 項で説明されているとおり、イベントを待つことができるのは拡張タスクだけです。

このような場合、アプリケーション内でタスクセットを使用して、イベント待ち状態をシミュレートすることができます。このシミュレーションにおいては、タスクセットが擬似イベントになります。

この例をコード例 4-9 に紹介します。

```
#include “Task1.h”
TASK(Task1) {

   TasksetType Tmp;
   /* Create a singleton set holding this task. */
   GetTasksetRef(Task1, &Tmp);
   /* Subscribe. */
   MergeTaskset(DataReady, Tmp);
   TerminateTask();

}
```

```
#include "Task2.h"
TASK(Task2) {
   /* Process Data. */
   /* Notify any waiting tasks. */
   ActivateTaskset(DataReady);
   AssignTaskset(DataReady, os_no_tasks);
   TerminateTask();

}
```

コード例 4-9 擬似イベントを使用する

この例は、Task2 がデータ処理を完了したときに、そのことが Task1 に通知されなければならない、という条件を実装したものです。Task1 はタスクセット DataReady を受け取ります。Task2 はデータを処理し終えると、タスクセット DataReady を起動することにより、そのデータを待っているすべてのタスクに処理完了を知らせます。

4.15 パフォーマンスの最大化とメモリ使用量の最小化

RTA-OSEK は、ターゲットアプリケーションのコードとデータの容量を積極的に最小化するように設計されているため、アプリケーションの特性を分析し、必要な機能だけを含むシステムを生成します。

ここにおいて、ユーザーが選択するタスク特性は、アプリケーションの最終的なサイズと処理速度に大きく影響します。たとえば、BCC2 タスクをタスクセットにすると効率が非常に悪くなります。

最も効率的なアプリケーションを作成するには、システムには BCC1 タスクだけを定義してライトウェイトターミネーションを使用するようにし、浮動小数点は使用しないようにしてください。

ただし、ユーザーのアプリケーションに機能を追加するにつれてシステムは大きくなり、処理も遅くなっていくことは避けられません。

BCC2 タスクが含まれるシステムのオーバーヘッドは、BCC1 タスクしかないシステムより大きくなります。優先度共有のないシステムは、たとえ多重起動が許可されている場合でも、優先度共有が行われているシステムよりは効率的です。

ECC1 タスクを持つシステムのオーバーヘッドはさらに大きく、また ECC2 タスクが含まれるシステムは、オーバーヘッドが最も大きくなります。

4.16 まとめ

- タスクには、並列的に実行されるべき処理が含まれます。
- タスクには基本（'basic'）クラスと拡張（'extended'）クラスという 2 つのクラスがあり、各クラスにはレベル 1 とレベル 2 という 2 つのレベルがあります。
- タスクは、優先度に基づいてスケジューリングされます。優先度の高いタスクがレディ状態になると、自分より優先度が低いタスクをプリエンプト（強制排除）します。
- タスクはレディ状態、実行状態、サスペンド状態、または待ち状態のいずれかの状態で存在します（ただし、待ち状態になることができるのは拡張タスクだけです）。
- タスクのターミネートは、そのタスクが TerminateTask()、ChainTask(TaskID)、または ChainTaskset(TasksetID) を呼び出すことによって行われます。これらの関数呼び出しは、必ず、タスクのエントリ関数の最終行で行います。
- アプリケーションコードにはタスク固有のヘッダ（<TaskID>.h）をインクルードする必要があります。したがって、各タスクのソースコードは別々のファイルとして作成することをお勧めします。
- 多重起動（'multiple activation'）を有効にしていない場合、起動できるのはサスペンド状態のタスクのみです。
- タスクセットは、RTA-OSEK の特殊機能です。これを利用して、一度に複数のタスクを起動できます。この場合、タスクのターミネートは ChainTaskset(TaskID) を使用します。

5 割込み

割込みは、ユーザーのアプリケーションと、その外側で実際に発生する事象とのインターフェースとなります。たとえば、ボタン押下をとらえる割込みや、時間の経過を知らせる割込み、またはその他のさまざまなスティミュラスをとらえる割込みを使用することができます。

通常、割込みが発生すると、プロセッサは、メモリ内にあらかじめ定義されている**ベクタ**と呼ばれるロケーションを見ます。ベクタには普通、各割込みに関連付けられている割込みハンドラのアドレスが設定されています。これについては、プロセッサのマニュアルとターゲット用の『RTA-OSEK バインディングマニュアル』に詳しく記載されています。アプリケーション内のすべてのベクタが設定されているメモリブロックは、一般的に**ベクタテーブル**と呼ばれます。

5.1 シングルレベルプラットフォームとマルチレベルプラットフォーム

ターゲットプロセッサは、サポートしている割込み優先度レベルの数に応じて 2 種類に分類されます [1]。ご使用のターゲットハードウェアの割込みメカニズムについて、十分に理解しておいてください。

ターゲットには以下の 2 つのタイプがあります。

- シングルレベル
  シングルレベルプラットフォームでは、割込み優先度は 1 つだけです。1 つの割込みが処理されている場合、その処理が終わるまで、他の未処理の割込みはすべて保留されます。
- マルチレベル
  マルチレベルプラットフォームでは、割込みレベルが 2 つ以上あります。1 つの割込みが処理されている場合でも、その処理はその割込みよりも優先度の高い割込みによりプリエンプト（強制排除）される可能性があります。

5.2 割込みサービスルーチン

OSEK オペレーティングシステムは、**割込みサービスルーチン**（ISR）を使用して割込みを捕捉します。ISR はタスクとよく似ていますが、以下の点が異なります。

- RTA-OSEK コンポーネントの API で起動することはできません。
- ターミネート用 API（`TerminateTask()` や `ChainTask()`）を呼び出せません。
- タスクセットに割り当てることはできません。
- 関連付けられている割込み優先度レベルのエントリポイントから実行を開始します。
- RTA-OSEKコンポーネントの 一部のAPIしか呼び出せません。(ISR内からRTA-OSEKコンポーネントの API を呼び出す場合は、『RTA-OSEK リファレンスガイド』に記載されている関数呼び出し環境を参照してください。)

5.2.1 カテゴリ 1 の割込みとカテゴリ 2 の割込み

OSEK オペレーティングシステムでは、割込みは、**カテゴリ 1** と**カテゴリ 2**、という 2 つのカテゴリに分類されています。カテゴリは、割込みの処理に OS が関与するかどうかを表すものです。

カテゴリ 1 の割込み（'Category 1 Interrupt'）

カテゴリ 1 の割込みは RTA-OSEK との相互関係を持ちません。カテゴリ 1 の割込みは、ユーザーアプリケーション内で最高の優先度にしてください。ユーザー の責任においてハードウェアを正しく設定し、ハンドラを作成して、割込み処理終了後にコントロールを戻す処理を行ってください。

カテゴリ 1 の割込みハンドラについては、5.5.1 項で説明します。

ハンドラは、RTA-OSEK コンポーネントの優先度レベル、またはそれより上の優先度レベルで実行されます。しかし、ユーザーアプリケーション内において、割込みのイネーブル／ディセーブル、レジューム／サスペンドを行うための RTA-OSEK コンポーネント API を呼び出すことができます。

---

1. 「ターゲットプロセッサ上の割込み優先度レベル」と「タスクの優先度レベル」とを混同しないように注意してください。

カテゴリ 2 の割込み（'Category 2 Interrupt'）

カテゴリ 2 の割込みの場合、割込みベクタは RTA-OSEK コンポーネントの内部コード（「コードラッパー」）をポイントします。割込みが発行されると、RTA-OSEK コンポーネントはそのコードを実行し、さらにユーザーが用意したハンドラを呼び出します。

このハンドラは、割込み（これは非常に優先度の高いタスクと考えることができます）に結びつけられた ISR として用意されるものです。実行は ISR の所定のエントリポイントから始まり、エントリ関数の処理が終了するまで続行されます。エントリ関数の処理が終わると、RTA-OSEK コンポーネントは内部コードの残りの部分を実行した後、割込み処理から戻ります。

図 5-1 に、カテゴリ 2 の割込みハンドラの状態遷移を示します。

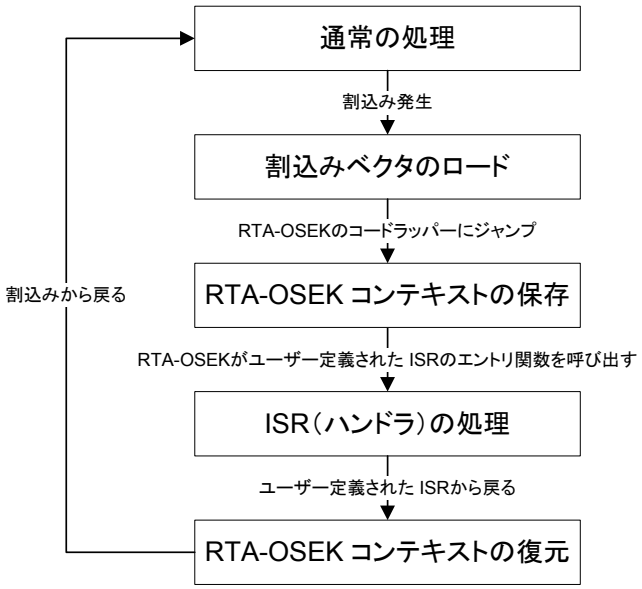


図 5-1 カテゴリ 2 の割込み処理状態図

図 5-2 に、RTA-OSEK コンポーネントのコードラッパーのしくみを示します。

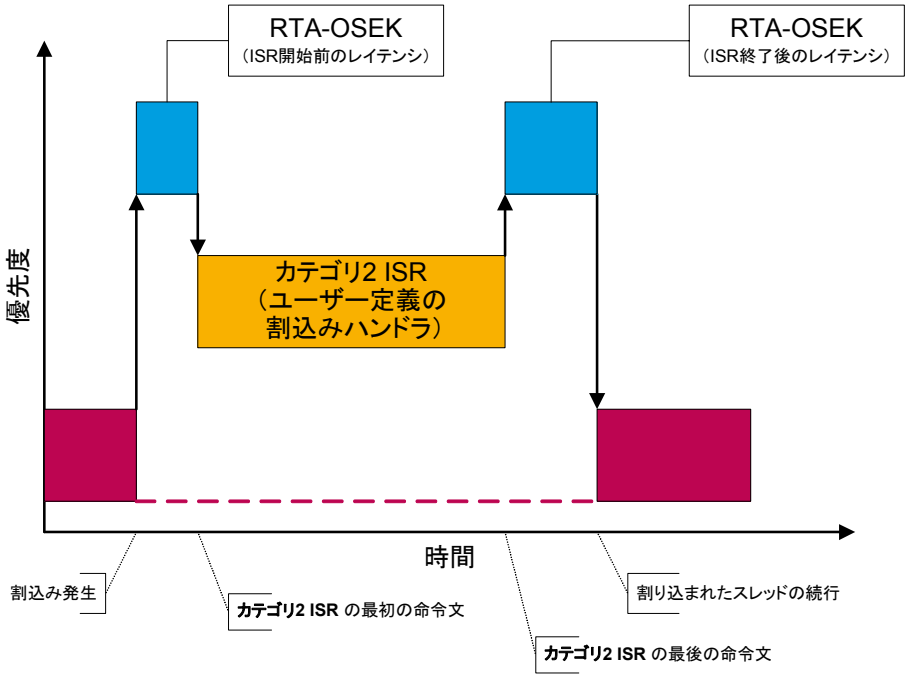


図 5-2 RTA-OSEK コンポーネントのカテゴリ 2 のラッパー

5.3 割込み優先度

割込みは、**割込み優先度レベル**（IPL：**I**nterrupt **P**riority **L**evel）という優先度レベルで実行されます。RTA-OSEK では、すべてのターゲットマイクロコントローラに対応する標準的な IPL が使用されていて、IPL 0 がタスクレベル、IPL 1 以上が、発生した割込みを表します。IPL とタスク優先度レベルは別のものなので、注意が必要です。IPL 1 は、ユーザーアプリケーション内の最上位のタスク優先度よりも高い優先度となります。

IPL は、ターゲットハードウェア上の割込み優先度を、プロセッサに依存しない形で定義するものです。IPL がターゲットハードウェアの割込み優先度にどのようにマッピングされているかについては、ターゲット用の『RTA-OSEK バインディングマニュアル』に記載されています。

プロセッサが割込みのネスティングをサポートしている場合、ISR をネストさせることができるので、たとえば優先度の高い ISR が、優先度の低い ISR の実行に割込みをかけることができます。ただし、タスクが ISR をプリエンプトすることは絶対にできません。

カテゴリ 2 の割込みがカテゴリ 1 の割込みハンドラの処理に割り込むこともできません。つまり、カテゴリ 2 の割込みの優先度をカテゴリ 1 の割込みの優先度よりも高くすることはできません。RTA-OSEK GUI は、ユーザーが割込みのコンフィギュレーションを設定する際、自動的にこの点についてチェックします。

割込み優先度の上下関係を図 5-3 に示します。

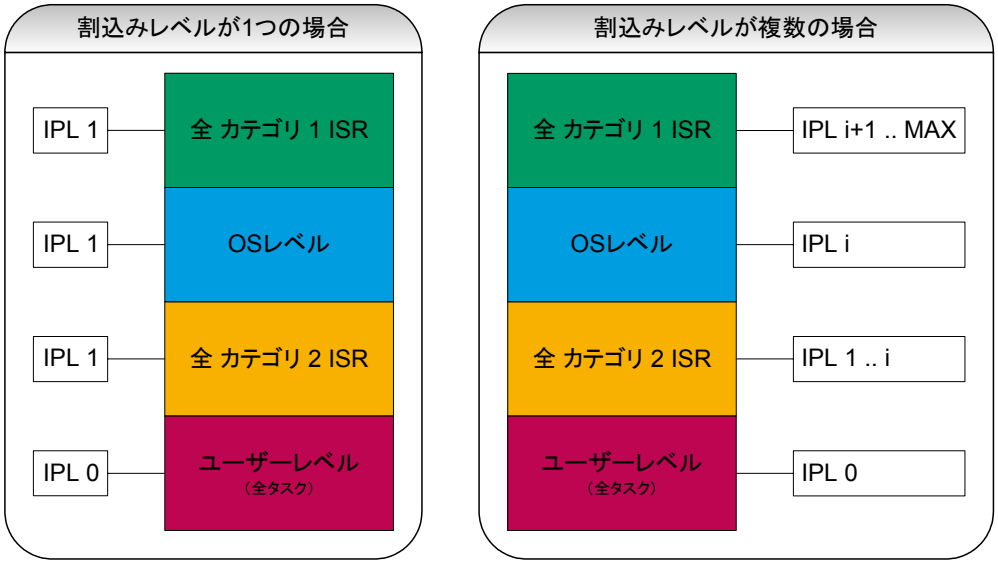


図 5-3 割込み優先度の上下関係

5.3.1 ユーザーレベル

ユーザーレベルは最下位の割込み優先度レベルで、すべての割込みを処理できます。すべてのタスクは、そのエントリポイントからユーザーレベルで処理を開始します。

場合によっては、あるタスクをユーザーレベルより高いレベルで稼働させ、割込みハンドラと共有するデータにアクセスする必要がありますが、そのような処理が実行されている間は、割込みが処理されないようにしなければなりません。

ISR は、タスクがユーザーレベルよりも高い割込み優先度レベルで実行されているときでも、そのタスクをプリエンプトする可能性がありますが、これは、その ISR の割込み優先度レベルが、実行中のタスクのその時点の割込み優先度レベルよりも高い場合だけです。

5.3.2 OS レベル

最高優先度のカテゴリ 2 割込みは、**OS レベル**として定義されます。この割込み処理が OS レベルまたはそれより高いレベルで実行されると、他のカテゴリ 2 割込みは一切発生できなくなります。

RTA-OSEK コンポーネントは、OS の内部データ構造体への同時アクセスを、OS レベルを使って防いでいます。あるタスクが OS レベルで実行されると、そのタスクに呼び出されるもの以外の RTA-OSEK コンポーネントの処理は一切行われなくなります。

5.4 割込みのコンフィギュレーション設定

RTA-OSEK コンポーネントでは、割込みは RTA-OSEK GUI を使用して静的に設定されます。図 5-4 は、割込みのコンフィギュレーション設定の方法を示しています。

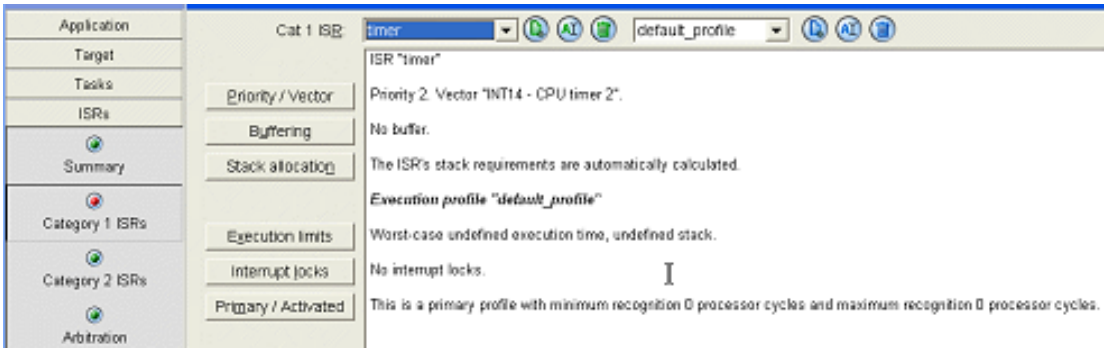


図 5-4 RTA-OSEK GUI を使用して割込みのコンフィギュレーションを設定する

割込みの最も基本的な属性には、以下のものがあります。

- 割込み名
  この名前は、ユーザーが作成したハンドラの C コードを参照するために使用されます（この使い方については、5.5 項で説明します）。
- 割込みカテゴリ
  ハンドラが RTA-OSEK API コールを実行する必要がある場合はカテゴリ 1、それ以外はカテゴリ 2 です。
- 割込み優先度
  優先度は、スケジューラが（タスクにタスク優先度を使用するのと同じように）その割込みをいつ実行するかを判断するために使用されます。ただし、一部のターゲットは 1 つの割込み優先度しかサポートしていません。

重要

実際に割込みをかけるデバイスにプログラムされた優先度レベルが、RTA-OSEK GUI で設定されたレベルと一致していなければなりません。

- 割込みベクタ
  RTA-OSEK は、指定されたベクタを使用して、割込み用のベクタテーブルエントリを生成します。

デフォルトでは、RTA-OSEK が各割込みベクタにシンボリックな名前を付けます。これは、新しい OIL ファイルを作成する際に選択するターゲットによって決まります。

マイクロコントローラファミリ用の割込みベクタアドレスを使用したい場合は、**File → Options...** を選択して **Options** ダイアログボックスを開き、図 5-5 のように、**Application Settings** タブの“Show ISR vector desciptions”オプションをオフにしてください。

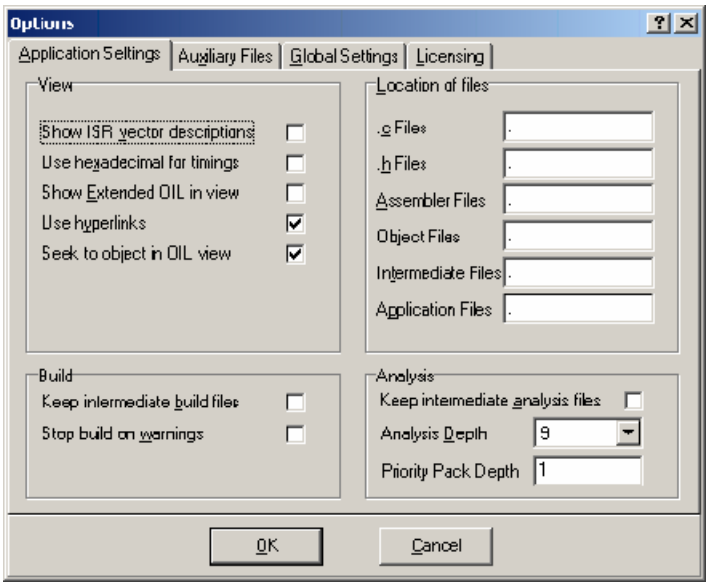


図 5-5 ISR ベクタディスクリプションを無効にする

5.4.1 ベクタテーブルの生成

通常、RTA-OSEK はベクタテーブルを自動生成することができます。内部ラッパーコードをポイントするベクタを使用してベクタテーブルを作成します。RTA-OSEK が生成したベクタテーブルは `osgen` ファイルに出力されます。

ユーザーが独自のベクタテーブルを作成したい場合は、RTA-OSEK にベクタテーブルを生成させないようにする必要があります。ベクタテーブルの生成の禁止は、図 5-6 に示すように **Target Vectors** 設定で指定してください。

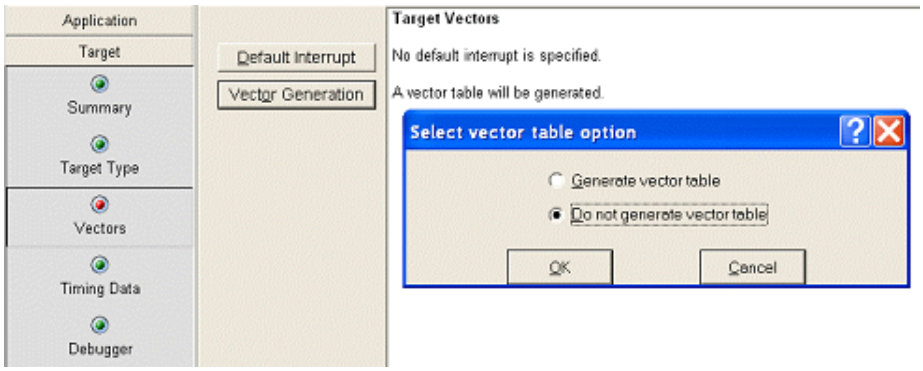


図 5-6 RTA-OSEK によるベクタテーブルの自動生成を禁止する

ユーザー独自のベクタテーブルを作成する方法については、ご使用のターゲットの『RTA-OSEK バインディングマニュアル』を参照してください。

5.5 割込みハンドラの実装

ここでは、カテゴリ 1 割込み用とカテゴリ 2 割込み用の割込みハンドラについて説明します。

5.5.1 カテゴリ 1 の割込みハンドラ

一般に、ユーザー定義されたカテゴリ 1 の割込みハンドラは移植不可能です。通常、これらのハンドラは、コンパイラ固有の ANSI C 拡張機能を用いて作成されますが、一部のコンパイラではこれを行うことができないため、そのような場合は、アセンブリ言語でハンドラを作成する必要があります。

カテゴリ 1 ISR のエントリ関数の名前は、設定時にユーザーが ISR の名前として定義した名前と同じでなければなりません。

カテゴリ 1 ISR の場合、一般的には、エントリ関数を定義するときに用いなければならないコンパイラ固有のキーワードがあります。

カテゴリ 1 ISR 用のエントリ関数の例をコード例 5-1 に紹介します。

```
interrupt void Interrupt1(void) {
/* Handler body. */
/* Return from interrupt. */
}
```

コード例 5-1 カテゴリ 1 ISR 用のエントリ関数

ターゲット固有の情報は、すべてターゲット用の『RTA-OSEK バインディングマニュアル』に記載されています。

5.5.2 カテゴリ 2 の割込みハンドラ

カテゴリ 2 の割込みが RTA-OSEK コンポーネントの管理下で処理されることは前述のとおりです。カテゴリ 2 の割込みハンドラはタスクとよく似ています。このハンドラにはエントリ関数があり、このハンドラを実行する必要があるときには RTA-OSEK コンポーネントがこのエントリ関数を呼び出します。カテゴリ 2 の割込みハンドラは、コード例 5-2 に示す C 言語の構文で作成されます。

```
ISR (isr identifier){ … }
```

コード例 5-2 カテゴリ 2 割込みハンドラ

コード例 5-3 は、`Interrupt1` という単純な割込みハンドラのコードを示しています。

```
#include "Interrupt1.h"/* Header file generated
                        * by RTA-OSEK GUI. */

ISR (Interrupt1) {

DismissInterrupt();/* User supplied function to
                    * cancel interrupt. */
ActivateTask_Task1();/* Let Task1 do the work. */

}
```

コード例 5-3 カテゴリ 2 ISR 用のエントリ関数

重要

カテゴリ 2 の ISR のエントリ関数用の C 関数プロトタイプは、RTA-OSEK により生成されるヘッダファイル内に用意されるので、ユーザーがこれを用意する必要はありません。各 ISR ごとにその ISR が使用するハンドラ固有の宣言が含まれる適切なファイルをインクルードする必要があるので、各 ISR をそれぞれ別のソースファイルとして作成してください。

重要

ユーザー定義のカテゴリ 2 のハンドラ内で、割込みから戻るコマンドを使用することはできません。割込みから戻る処理は RTA-OSEK によって行われます。

5.5.3 効率的な割込みハンドラの作成

ハンドラのコードはできる限り短くなるように記述し、割込み優先度が 1 つしかないターゲットの場合は特にこれに注意してください。ハンドラの実行時間が長くなると、それより低い優先度の割込みを処理するまでの遅れが長くなってしまいます。

カテゴリ 2 のハンドラの場合は、必要な処理をタスクに移動することができます。この場合、割込みハンドラはタスクの起動のみを行っ後、すぐにターミネートします。

コード例 5-4 とコード例 5-6 に 2 種類の方法を示します。

```
#include “Interrupt1.h”
ISR(Interrupt1) {
  /* Long handler code. */
}
```

コード例 5-4 処理の長い割込みハンドラ（ブロック時間が長い）

```
#include “Interrupt1.h”

ISR(Interrupt1) {
  ActivateTask(Task1);
}


#include “Task1.h”

TASK(Task1) {
  /* Long handler code. */
  TerminateTask();
}
```

コード例 5-5 処理の短い割込みハンドラ（ブロック時間が短い）

5.6 割込みのイネーブルとディセーブル

割込みは、その割り込みがイネーブルされている（= 有効になっている）場合にのみ発生します。デフォルトでは、RTA-OSEK コンポーネントは `StartOS()` の処理終了時にすべての割込みをイネーブルにします。

重要

OSEK において、「ディセーブル」という語は割込みをマスクすることを意味し、「イネーブル」とは割込みのマスクを解除することを意味します。そのため、イネーブルやディセーブルを行う API コールは、割込みソースをイネーブル／ディセーブルするのではなく、プロセッサの割込みマスクを変更することにより、プロセッサの割込み検知を禁止するかどうかを切り替えます。

割込みを短時間の間だけディセーブルして、タスクや ISR の**クリティカルセクション**において割込みが発生しないようにする必要が生じる場合があります。「クリティカルセクション」とは、共有データにアクセスする一連の文を指します。

割込みのイネーブルとディセーブルは、様々な API 関数を用いて行うことができます。

- `DisableAllInterrupts()` と `EnableAllInterrupts()`
  ハードウェア上でディセーブルにできるすべての割込み（通常はマスクできるすべての割込み）をディセーブルにします。これらの呼び出しをネストすることはできません。
- `SuspendAllInterrupts()` と `ResumeAllInterrupts()`
  ハードウェア上でディセーブルにできるすべての割込み（通常はマスクできるすべての割込み）をサスペンド／レジュームします。これらの呼び出しはネストすることができます。
- `SuspendOSInterrupts()` と `ResumeOSInterrupts()`
  ハードウェア上のすべてのカテゴリ 2 の割込み（通常はマスクできるすべての割込み）をサスペンド／レジュームします。これらの呼び出しはネストすることができます。

重要

`'Suspend'` 呼び出しよりも `'Resume'` 呼び出しの方が回数が少なくなるようにしてください。そうしないと、深刻なエラーが発生して挙動が不確定になってしまう可能性があります。不正な `'Resume'` 呼び出しが行われると、以降の `'Suspend'` 呼び出しは機能しなくなり、その結果、クリティカルセクションが保護されなくなってしまいます。

コード例 5-6 では、割込み制御の API 関数が適切にネストされています。

```
#include "Task1.h"
TASK(Task1) {

   DisableAllInterrupts();

   /* First critical section, nesting not allowed. */

   EnableAllInterrupts();
   SuspendOSInterrupts();

   /* Second critical section, nesting allowed. */

   SuspendAllInterrupts();

   /* Third critical section. */

   ResumeAllInterrupts();
   ResumeOSInterrupts();
   TerminateTask();

}
```

コード例 5-6 割込み制御 API コールのネスティング

カテゴリ 1 の割込みがディセーブルになっている間は、RTA-OSEK コンポーネントの API コール（他の `Suspend/Resume` 呼び出し以外）が絶対に行われないようにしなければなりません。

カテゴリ 2 ISR の割込みレベルが OS レベルより高い場合は、その割込み優先度を元に戻すための呼び出し以外の RTA-OSEK コンポーネントの API コールは一切行うことはできません。ISR を実行するときには、割込み優先度レベルを元のレベルより低くすることはできません。

5.7 浮動小数点の使用

「タスク」の章で説明されているように、RTA-OSEK では、アプリケーション内では一般的に浮動小数点が使用されないものと想定されています。浮動小数点を使用する ISR がある場合は、その旨を RTA-OSEK GUI で宣言する必要があります。その情報に基づき、RTA-OSEK コンポーネントは、その ISR の入口と出口において、浮動小数点演算に使用されるすべてのハードウェアレジスタの保存と復元が確実に行われるようにします。

RTA-OSEK は、浮動小数点を使用するタスクや ISR のコンテキストを保存するために確保すべきメモリサイズを正確に計算することができます。これは、RTA-OSEK が、これらのタスクや ISR のワーストケースのプリエンプション深度を算出でき、保存するコンテキストの数を最適化できるためです。

5.8 デフォルト割込み

RTA-OSEK GUI を使用してベクタテーブルを生成する場合、未使用のベクタロケーションに**デフォルト割込み**を設定する方法があります。

図 5-7 は、デフォルト割込みを定義する方法を示しています。

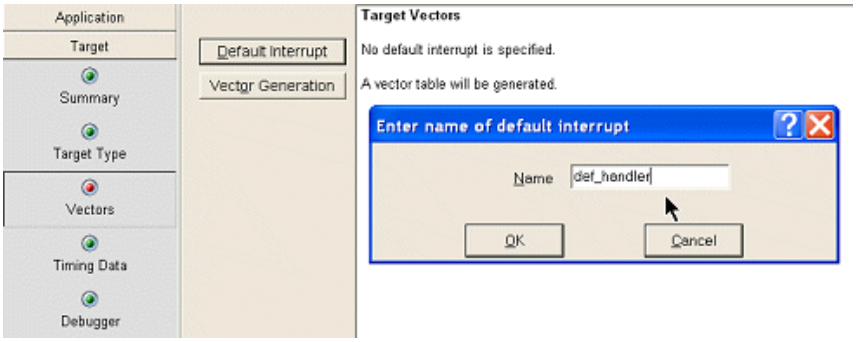


図 5-7 ベクタテーブルにデフォルト割込みを設定する

移植性

ターゲットによってはデフォルト割込みをサポートしていないものもあります。

デフォルト割込みは、他の通常の割込みとは若干異なり、ベクタテーブル内で、まだ具体的な割込みが定義されていない部分を埋めておくために使用されるものです。この機能は、デバッグ用に用いられるほか、量産システムにおいては不正な割込みによってシステムが暴走することを防ぐ「フェールストップ」の目的で使用されます。通常の処理を行う割込みハンドラをベクタにアタッチするには、明示的に ISR を作成する必要があります。

デフォルト割込みの使用には制約条件があります。OS コールを行うことはできず、また、ハンドラからリターンしてしまうと、その後のシステム挙動は未定義なものとなります。

重要

デフォルト割込みから RTA-OSEK コンポーネントの API コールを実行することはできません。また、ハンドラからコントロールを返すことはできません。

デフォルト割込みは、OSEK のカテゴリ 1 割込みに似ているので、コンパイラで定義されている構文で、割込みであることを明示する必要があります。デフォルト割込みハンドラ内の最後の文は、コード例 5-7 のように、無限ループ文にしてください。

```
__interrupt void default_handler(void)
{
  /* invoke target-specific code to lock interrupts */
  asm ("di"); /* or whatever on your platform */
  for (;;) {
  }
  /* Do NOT return from default handler. */
}
```

コード例 5-7 デフォルト割込みハンドラ

5.9 割込みのアービトレーション

複数の割込みが同じ割込み優先度レベルを共有している場合、RTA-OSEK GUI で**アービトレーションオーダー**（'arbitration order'）を定義する必要があります。これは分析に使用されるものです。

アービトレーションオーダーは、同じ優先度の複数個の割込みが同時に処理待ち状態になっている場合にそれらを処理する順序を定義するものです。

割込みアービトレーションオーダーは、図 5-8 のようにして設定します。

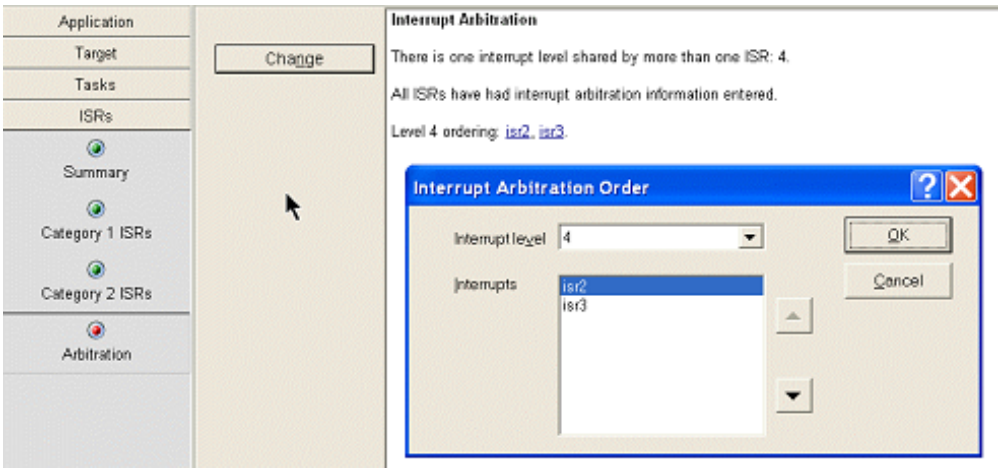


図 5-8 アービトレーションオーダーを定義する

図 5-8 の例では、両方の割込みが同時に処理待ち状態になった場合、`ISR_1` が先に処理されることがアービトレーションオーダーに規定されています。

多くのプロセッサでは割込みアービトレーションオーダーは固定されています（詳細についてはプロセッサのリファレンスガイドに記載されています）が、一部のプロセッサではユーザーがアービトレーションオーダーを定義することができます。

重要

定義されたアービトレーションオーダーは、アプリケーション内の割込みのアービトレーションオーダーと一致している必要があります。コンフィギュレーションファイルに定義されている情報がランタイムにおける割込みの挙動を正しく定義していることを確認してください。

5.10 まとめ

- 割込みは、システムの外部で発生したスティミュラスをとらえるためのメカニズムです。
- OSEK はカテゴリ 1 とカテゴリ 2 という 2 種類の割込みをサポートしています。
- カテゴリ 1 の割込みは、RTA-OSEK コンポーネントによって処理されません。
- カテゴリ 2 の割込みは、RTA-OSEK コンポーネントによって処理されます。

6 リソース

複数のタスクや ISR で共有する必要があるハードウェアやデータへのアクセスは、一般的に信頼性が低く、安全ではありません。これは、優先度の低いタスクや ISR が共有データを更新している途中でタスクや ISR によるプリエンプション（強制排除）が発生する可能性があるためです。この状況は**競合状態**（'race condition'）と呼ばれ、これについてテストを行うのは極端に困難です。

前述のように、共有データにアクセスする一連の文は**クリティカルセクション**と呼ばれます。

クリティカルセクション内でコードやデータに安全にアクセスできるようにするためには、**相互排除**（'mutual exclusion'）を徹底する必要があります。つまり、あるタスクがクリティカルセクションを実行している時に、システム内の他のタスクやカテゴリ 2 ISR がそのタスクをプリエンプトできないようにしなければなりません。

RTA-OSEK などの OSEK オペレーティングシステムでは、相互排除は**リソース**により実現されています。OSEK において「リソース」とは、単純な「バイナリセマフォ」を指します。

あるタスクまたはカテゴリ 2 の ISR がリソースを**取得**すると、他のタスクまたは ISR はそのリソースを取得することができません。クリティカルセクションが終了すると、リソースを使用していたタスクまたは ISR はそのリソースを**解放**します。

OSEK において、リソースは **ロッキングプロトコル** に従ってロックされます。このロッキングプロトコルは一般に **優先度シーリングプロトコル**（'priority ceiling protocol'）と呼ばれるもので、その中でも特に「即時継承・優先度シーリングプロトコル」（'immediate inheritance priotiry ceiling protocol'）、または「スタックリソースプロトコル」（'stack resource protocol'）とも呼ばれるものが使用されています。

OSEK の優先度シーリングプロトコルでは、リソースの「上限（'ceiling'）優先度」（そのリソースを取得するすべてのタスクまたは ISR の優先度の中で最高の優先度）という概念が用いられます。タスクまたは ISR があるリソースを取得すると、そのタスクまたは ISR の優先度は即時にそのリソースの上限優先度まで引き上げられます（ただしその上限優先度が現在の優先度よりも高い場合に限ります）。その後リソースが解放されると、そのタスクまたは ISR の優先度は、コールを実行する前の優先度に戻ります。

即時継承・優先度シーリングプロトコルを利用すると、以下の 2 つのメリットが得られます。

- デッドロックが発生しないことが保証されます。

  あるタスクまたは ISR がそのリソースを使用していた場合、そのタスクまたは ISR は上限優先度で実行されているため、別のタスクまたは ISR がリソースのロックを行おうとしても、それらの優先度は、現在リソースをロックしているものよりも低いため、ロックを行う命令を実行することができません。

- 優先度の逆転を最小限にすることができます。

  タスクまたは ISR は、実行中にブロックされる場合があり、実行開始時には必ずブロックが発生しますが、あるタスクまたは ISR がレディ状態になった際、その実行を妨げる可能性があるのは、リソースを使用している 1 つの低優先度のタスクまたは ISR のみとなります。多重のブロックは発生しないので、ワーストケースのブロック時間が最低限に抑えられます。

6.1 リソースの設定

RTA-OSEK は、どのタスクや ISR がどのリソースを使用するかを知っておく必要があります。その情報を元にして優先度シーリングプロトコルで用いられる上限優先度を算出します。

各タスクまたは ISR についてのその他のリソース使用情報は、タスクまたは ISR の設定時に設定できます。この情報は、分析の目的でのみ必要です。ユーザーのアプリケーション内では最大 255 個のリソースを宣言できます。

最も基本的な設定は、リソースを指定することです。図 6-1 のように設定してください。

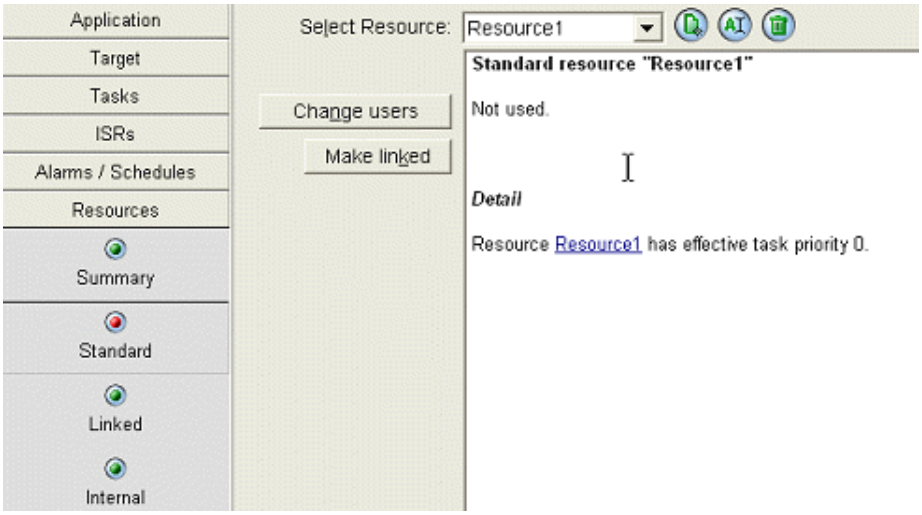


図 6-1 RTA-OSEK GUI を使用してリソースを設定する

図 6-1 は、`Resource1` というリソースが宣言されたことを示しています。プログラム内でこのリソースを参照するときには、これと同じ名前を使用してください。

6.1.1 割込みレベルのリソース

OSEK ではオプションとして、タスクと割込みとの間で共有されるリソースがサポートされています。

RTA-OSEK はこのような「結合リソース」を自動的に検知するので、特別の設定を行う必要はありません。

あるタスクが、ある ISR との間で共有するリソースを取得した際、RTA-OSEK は、そのリソースを共有する割込みの中で最も高いものと同じ優先度を持つ割込み、またはそれより低い割込みを、すべてマスクします。

これは、優先度シーリングプロトコルを拡張したものです。

つまり「タスクと ISR とがリソースを共有する」、ということは、ある優先度レベルの割込みをマスクできることになり、`Enable` ／ `Disable` や `Suspend` ／ `Resume` を用いるよりも優れた割込みマスク制御が実現できます。

この「割込みレベルのリソース」は、多重割込みをサポートするターゲット用の RTA-OSEK を使用する際に有効です。ある範囲の優先度の割込みをディセーブルしたい場合、その中の最上位の割込みと任意のタスクとでリソースを共有させることにより、それらの割込みの発生をマスクすることができます。

6.2 リソースの使用

リソースの取得には API 関数 `GetResource()` を使用し、解放には `ReleaseResource()` を使用します。タスクまたは ISR は、確保しているすべてのリソースを解放するまでターミネートしてはいけません。

タスクまたは ISR が使用できるリソースは、RTA-OSEK 設定時にユーザー定義されたものだけです。コード例 6-1 は、`Task1` 内でリソースが使用されています。

```
#include "Task1.h"

TASK(Task1) {

  /* Task functionality. */
  GetResource(Resource1);
  /* Critical section. */
  ReleaseResource(Resource1);
```

```
    /* Remainder of task functionality. */
    TerminateTask();
}
```

コード例 6-1 リソースを使用する

重要

GetResource() と ReleaseResource() は、必ず 1 組で続けて使用してください。つまり、すでに取得しているリソースを続けて取得したり、取得していないリソースを解放したりすることはできません。

GetResource() が呼び出されると、呼び出し元のタスクまたは ISR の優先度が、リソースの上限優先度まで引き上げられます。「リソースの上限優先度」は、そのリソースを共有するタスクや ISR の優先度中で最高の優先度で、これは RTA-OSEK により自動的に算出されます。

図 6-2 では、Task1 の優先度は 3 になっています。このタスクは Resource1 というリソースを Task2 と共有しています。Task2 の優先度は 7 なので、リソース優先度は 7（リソース Resource1 を共有するすべてのタスクの中で最高の優先度）になります。リソースが確保されると、Task1 は優先度レベル 7 で実行され、リソースが解放されると優先度レベル 3 に戻ります。このため、Task1 がリソースを保持している間に Task2 が起動された場合、Task2 はリソースが解放されるまでブロックされます。

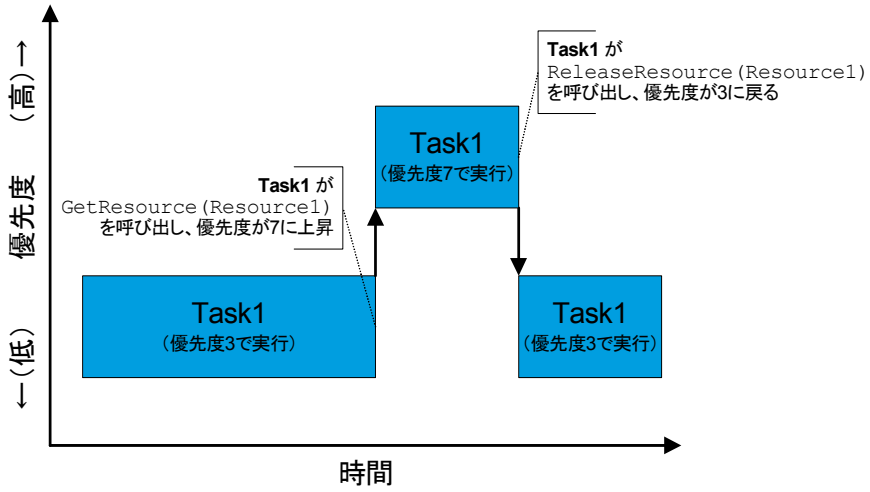


図 6-2 リソースを使用するタスク

6.2.1 リソースコールのネスティング

複数のリソースを同時に取得することは可能ですが、その場合も API コールを厳密にネストさせる必要があります。以下の 2 つのコード例は、1 つめが API コールのネストが間違っている例、もう 1 つが正しくネストしている例です。コード例 6-2 では、Resource1 と Resource2 を解放する順序が間違っています。

```
GetResource(Resource1);
GetResource(Resource2);
ReleaseResource(Resource1); /* Illegal! */

/* You must release Resource2 before Resource1 */

ReleaseResource(Resource2);
```

コード例 6-2 リソースコールのネストが間違っている例

コード例 6-3 では、すべてのリソースが確保された後、正しい順序で解放されています。

```
GetResource(Resource1);
  GetResource(Resource2);
    GetResource(Resource3);
    ReleaseResource(Resource3);
  ReleaseResource(Resource2);
ReleaseResource(Resource1);
```

コード例 6-3 リソースコールが正しくネストされている例

6.2.2 静的インターフェースの使用

タスクあるリソースを使用するには、そのリソースをあらかじめ宣言しておく必要があります。OSEK においては、リソースの上限優先度より低い優先度を持つタスクが、そのリソースをロックすることが許されていますが、もしもこの上限優先度より高い優先度を持つタスクまたは ISR が `GetResource()` を呼び出すと、`E_OS_ACCESS` というエラーが返ります。

このエラーの発生を防ぐには、RTA-OSEK の **静的インターフェース** の使用が有効です。

静的インターフェースとは、RTA-OSEK が、ユーザーアプリケーションに合わせて最適化されたシステムコールを生成するためのメカニズムで、ここでは `GetResource()` と `ReleaseResource()` の「静的バージョン」が使用されます。

以下の 2 つの例では、どちらも `Resource1` が取得され、その後、解放されていますが、コード例 6-4 では「動的コール」、コード例 6-5 では「静的コール」が用いられています。

```
GetResource(Resource1);/* Dynamic call. */
/* Critical section. */
ReleaseResource(Resource1);
```

コード例 6-4 動的リソースコール

```
GetResource_Resource1();/* Static call. */
/* Critical section. */
ReleaseResource_Resource1();
```

コード例 6-5 静的リソースコール

パフォーマンスの最適化のためには、できる限りこの「静的バージョン」を使用してください。動的コールが必要なのは、コンパイル時にリソースが未定義である場合、つまりリソースがパラメータとしてライブラリ内の関数に渡される場合などに限られます。

静的バージョンを使用すると、RTA-OSEK は実際に必要な処理を予測することができます。たとえば、あるリソースをロックする最高優先度のタスクや ISR については、優先度がすでにリソースレベルと一致しているので、実際には何の処理も必要ありません。そのような場合、`GetResource()` と `ReleaseResource()` には空の文が割り当てられます。

6.3 リンクリソース（'linked resource'）

OSEK では、同じリソースに対する `GetResource()` の呼び出しをネストさせることはできませんが、場合によってはそのような呼び出しが必要になる場合があります。

たとえば、ある関数が多くのタスクに共有されているとします。その関数内で、その関数を共有するタスクのうち 1 つのタスクだけが使用するリソースを取得しなければならない場合、どのようなことが起こるでしょうか。コード例 6-6 を見てみましょう。

```
#include “Task1.h”
TASK(Task1) {
   …
   GetResource(Resource1);
   /* Critical section. */
   SomeFunction();
   ReleaseResource(Resource1);
   …
}
#include “Task2.h”
TASK(Task2) {
   …
   SomeFunction();
   …
}
#include “osek.h” /* Generic header file. */
void SomeFunction(void) {
   …
   GetResource(Resource1);      /* Not allowed! */
   /* Critical section. */
   ReleaseResource(Resource1);  /* Not allowed! */
   …
}
```

コード例 6-6 不正にネストしているリソース API コール

このように、潜在的なリソースのネスティングが行われる可能性がある場合は、**リンクリソース**（'linked resource'）を使用する必要があります。リンクリソースは既存リソースのエイリアスで、同じ共有オブジェクトを保護するものです。

リンクリソースは、RTA-OSEK GUI で以下のように定義します。

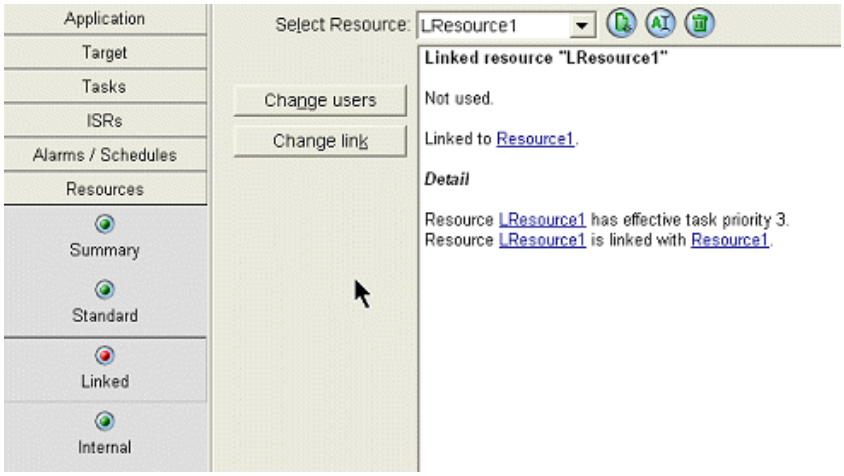


図 6-3 リンクリソースを RTA-OSEK GUI で設定する

リンクリソースの確保と解放は、標準リソース用の同じ API 関数を使用して行います（これらの API 関数については、6.2 項で説明されています）。また、既存のリンクリソースにリンクされたリソースを作成することもできます。

6.4 内部リソース（'internal resource'）

一連のタスクが非常に緊密にデータを共有する場合、リソースの各データ項目に対するアクセスの保護は容易ではなく、場合によっては、リソースを取得する必要があるポイントを見極めることも不可能になる場合もあります。

共有データへの同時アクセスは、**内部リソース**を使って防ぐことができます。内部リソースは、タスクのライフサイクルの期間だけアロケートされるリソースです。

このリソースは、RTA-OSEK GUI を使用してオフライン設定します。通常のリソースとは異なり、内部リソースを取得したり解放したりすることはできず、また ISR は内部リソースを使用できません。

ビルドプロセス中に RTA-OSEK が計算を行うため、RTA-OSEK コンポーネントの内部リソースはランタイムにおいてプロセッサリソースをまったく消費しません。

内部リソースを共有するタスクは、RTA-OSEK GUI のコンフィギュレーション設定において定義します。このメンバーシップは「静的」です。

図 6-4 は、`IntResource1` という内部リソースの宣言を示しています。この内部リソースは `t1` と `t3` という 2 つのタスクに共有されます。

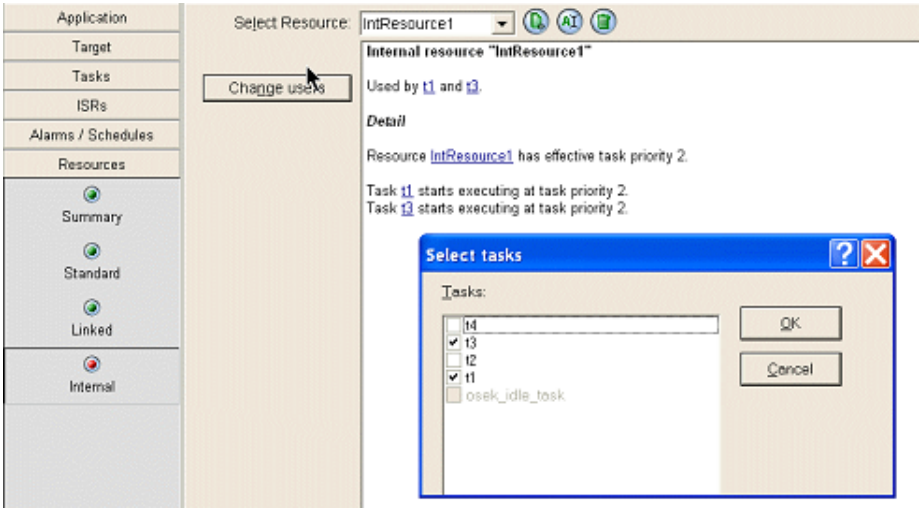


図 6-4 RTA-OSEK GUI を使用して内部リソースを宣言する

あるタスクが内部リソースを使用する場合、RTA-OSEK コンポーネントは、そのタスクのエントリ関数を呼び出す前に、自動的にそのリソースを取得します。その後そのリソースは、そのタスクがターミネートするか、`Schedule()` または `WaitEvent()` を呼び出すと、自動的に解放されます。

タスクの実行中、その内部リソースを共有する他のすべてのタスクは、その内部リソースが解放されるまでは実行できなくなります。ただし、実行中のタスクより優先度が高く、その内部リソースを共有しないタスクは実行可能です。図 6-5 にその例を示します。この例では、`Task1` が優先度 3 の内部リソースを共有しています。

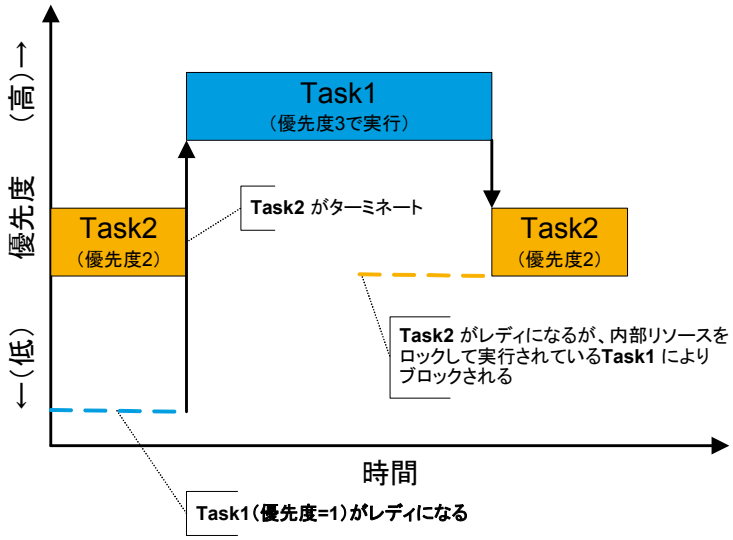


図 6-5 内部リソースを共有しないタスクのプリエンプション

図 6-5 では、最初に Task2 が実行されていて、それよりも優先度の低い Task1 はレディ状態になっています。Task2 がターミネートすると、Task1 が実行されます。これは、Task1 は優先度レベル 3 の内部リソースを Task3 と共有しているためです。

Task1 実行中に Task2 がレディ状態になりますが、Task1 が優先度レベル 3 のリソースを確保したままなので、Task2 は Task1 をプリエンプトすることができません。Task1 がターミネートすると、Task2 が実行を再開します。

内部リソースを共有するタスク同士は、それらのタスク同士でのみ、互いにノンプリエンプティブに実行されます。通常のノンプリエンプティブタスクはアプリケーション全体にわたって常にノンプリエンプティブに実行されるので、これとは異なります。

内部リソースを使用すると、アプリケーションのタイミング挙動を一層適切に管理できるようになります。またプリエンプションの総数を制限することができるため、システムの使用メモリの削減にも役立ちます。

内部リソースを共有するタスクはシーケンシャルに実行されるので、どの時点においても、スタックに保持されているのはそれらのタスクのうち 1 つだけです。そのため、スタック全体の使用スペースが少なくなります。

6.5 スケジューラをリソースとして使用する

システム内の他のどのタスクからもプリエンプトされずに実行される必要のある「クリティカルセクション」を持つタスクは、スケジューラを「リソース」として取得することができます。このためにあらかじめ定義されている RES_SCHEDULER というリソースは、すべてのタスクで使用することができます。

あるタスクが RES_SCHEDULER を取得すると、そのタスクが RES_SCHEDULER を解放するまで、他のタスクはプリエンプションを行うことができません。つまり、タスクが RES_SCHEDULER を保持している間、そのタスクはノンプリエンプティブ状態になります。これは、タスク自体をノンプリエンプティブタスクとして定義するよりも好ましい方法であるといえ、特に、タスクの全ステップのうちの限られた短い部分だけをプリエンプション禁止にしたいような場合に有効です。

RES_SCHEDULER を使用すると、タスクが他のタスクにより何回もプリエンプトされてしまうことを防ぐことができ、タスクのレスポンスタイムを短縮することができます。

6.5.1 RES_SCHEDULER の無効化

RTA-OSEK における RES_SCHEDULER は、内部的に生成される、標準の OSEK リソースです。アプリケーション内で RES_SCHEDULER を使用する必要がない場合は、これを無効化することによって ROM と RAM のスペースを節約することができます。**Application → Optimazation** を選択し、図 6-6 のように設定してください。

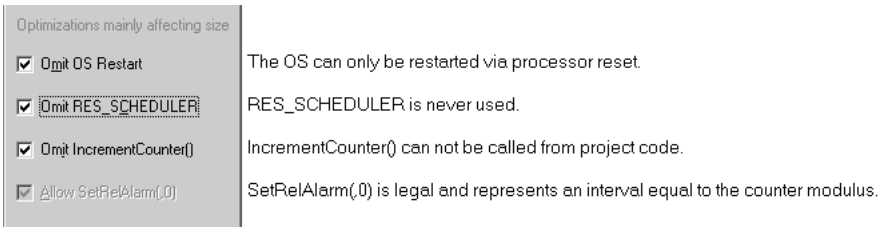


図 6-6 RES_SCHEDULE を無効化する

6.6 プリエンプション管理メカニズムの選択

ロックを必要としないコードが GetResource() と ReleaseResource() の間で実行されると、システムのレスポンスが低下してしまう可能性があります。

これを念頭に置いて、アプリケーションでリソースを使用するときには、リソースで保護したいコード部分の前後のできるだけ近い位置に GetResource() と ReleaseResource() を配置してください。

ただし、このルールには 1 つの例外があります。それは、短時間で処理が終わってしまうタスクまたは ISR の中で、同じリソースに対して何度も GetResource() と ReleaseResource() を呼び出す場合です。そのような場合は、そのタスク全体の実行時間の中で API コールによる部分がかなりの割合を占めるようになってしまいます。

そのようなタスクまたは ISR は、ボディ全体を GetResource() と ReleaseResource() の間に配置すると、ワーストケースのレスポンスタイムが実際に短縮されます。

また、ノンプリエンプティブタスクの使用はできる限り避け、RES_SCHEDULER を取得するようにしてください。さらに、リソースが確保される時間を最小限にし、リソース確保の影響を受けるタスクの数を最小限にすると、システムの反応性とスケジューラビリティが向上します。

6.7 競合状態（'race condition'）の回避

OSEK 規格において、リソースは、TerminateTask() が呼び出される前に解放されなければならないことになっています。状況によっては、このことが原因でアプリケーション内に競合状態が生じてしまう可能性があり、さらにはタスクの起動が失われてしまう可能性があります（競合状態については、本章の冒頭で説明されています）。

コード例 6-7 は、競合状態が問題になる可能性のある種類のシステムを示しています。ここでは仮に、2 つの BCC1 タスクが、共通のバッファを用いてデータを交換する、と想定します。

```
#include "Write.h"

TASK(Write) { /* Highest priority .*/

   WriteBuffer();
   GetResource(Guard);
   BufferNotEmpty = True;
   ReleaseResource(Guard);
   ChainTask(Read);

}
```

```
#include “Read.h”

TASK(Read) { /* Lowest priority. */

   ReadBuffer();
   GetResource(Guard);

   if (BufferNotEmpty) {
      ReleaseResource(Guard);
      /* Race condition occurs here. */
      ChainTask(Read);
   } else {
      ReleaseResource(Guard);
      /* Race condition occurs here. */
      TerminateTask();
   }
}
```

コード例 6-7 競合状態が発生する可能性のあるシステム

コード例 6-7 では、リソースが解放されてからタスクがターミネートするまでの間に、`Read` タスクが `Write` タスクによりプリエンプトされる可能性があります。すると、`Write` が `Read` をチェーンした時、`Read` はまだ実行中であるため、`Read` の起動は失われてしまいます。

この問題を解決するためには、`Read` タスクを BCC2 にして、起動をキューイングできるようにしてください。

6.8 まとめ

- リソースは、共有データやハードウェアリソースへのアクセスについて相互排除を実現するために用いられます。
- タスクと ISR は任意の数のリソースを共有できます。
- すべての `GetResource()` と `ReleaseResource()` が正しくネストしていなければなりません。
- すべてのリソースは、それを確保したタスクまたは ISR がターミネートする前に解放されなければなりません。
- スケジューラをリソースとして使用することもできますが、できれば内部リソースを優先して使用するようにしてください。
- 内部リソースを使うと、プロセッサリソースを消費することなく、複数のタスク間のプリエンプションを管理するメカニズムを実現できます。

7 イベント

OSEK システムでは、シグナル情報をタスクに送るためにイベントが使用されます。イベントの設定方法については、7.1 項で説明します。

イベントを使用して、拡張タスク用の**同期ポイント**を設けることができます。同期化の例を図 7-1 に示します。

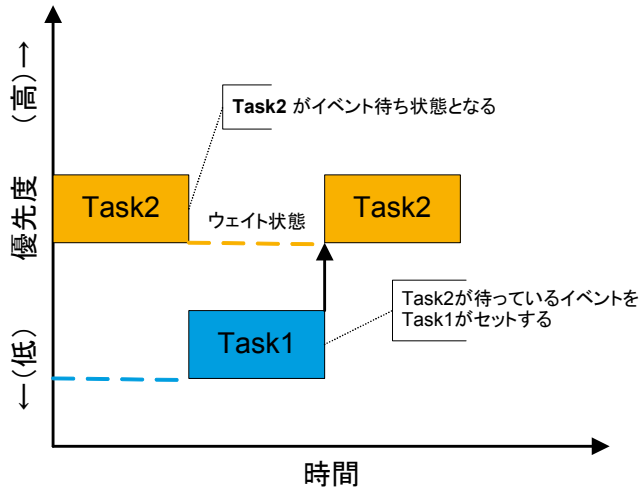


図 7-1 同期化

拡張タスクは、イベントを**待つ**ことができ、これによって待ち状態になります。これについては、7.1.1 項で詳しく説明します。

システム内のあるタスクまたは ISR によりイベントがセットされると、待ち状態のタスクはレディ状態に移行します。そのタスクがレディ状態のタスクの中で優先度が最高のタスクになると、RTA-OSEK コンポーネントがそのタスクを選択して稼働させます。

イベントは、それに関連付けられている拡張タスクにより所有されます。通常、拡張タスクは無限ループになっていて、その中に、そのタスクが所有するイベントを待つための、保護された一連の「待ち」呼び出しが含まれます。このイベントメカニズムにより、イベント駆動による「ステートマシン」を OSEK で実現することが可能となります。

システムにおいてタイミング挙動が重要である場合、すべての拡張タスク（つまりイベントを宣言しているすべてのタスク）の優先度を、基本タスクよりも低くする必要があります。

7.1 イベントを設定する

イベントの宣言は RTA-OSEK GUI を使用して行います。ユーザーアプリケーションに定義できるイベントの最大数は、ターゲットハードウェアにより決まっています。RTA-OSEK GUI の Target Summary を見ると、この最大数がわかります。

図 7-2 の例では、ターゲットは最大 16 個のイベントを待つことができます。

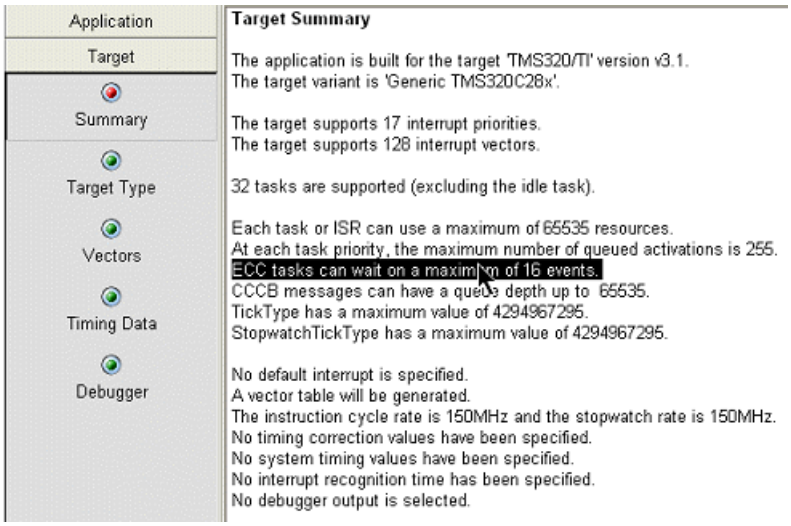


図 7-2 ターゲットのイベント最大数を表示する

イベントの宣言には、以下の項目が含まれていなければなりません。

- 名前
  名前は、設定時にイベントの目的を示すためだけに使用されます。
- イベントを利用する 1 つ以上のタスク
- イベントマスク
  RTA-OSEK GUI で定義されるイベント名は、ランタイムにはイベントマスク用のシンボリック名として使用されます。マスクは 1 つのビットだけがセットされた `N` ビットのベクタです（`N` はタスクが待つことのできるイベントの最大数）。セットされたビットにより、イベントが識別されます。

イベント名は、ランタイムにはマスク用のシンボリック名として使用されます。マスクは明示的に宣言することもでき、またはユーザーのアプリケーション用のマスクを RTA-OSEK で自動生成することもできます。複数のタスクが多くのイベントを待つようなアプリケーションについては、この自動生成を行うことをお勧めします。

図 7-3 では、`Event1` というイベントが宣言されています。この例では、RTA-OSEK がイベントマスクを自動生成し、また、宣言されたイベントが `t3` というタスクに使用されるように設定されています。

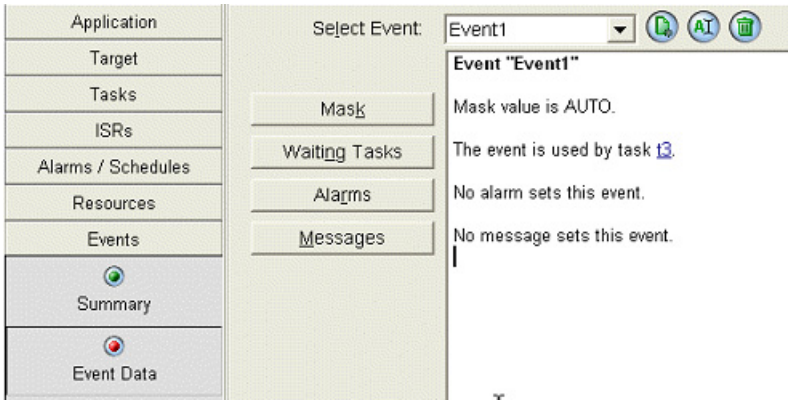


図 7-3 RTA-OSEK GUI でイベントを宣言する

1 つのイベントが複数のタスクに使用される場合、各タスクにそのイベントのコピーが必要です。イベントがセットされるときには、タスクも同時に指定されなければなりません。たとえば `Event2` というイベントを `t3` というタスク用にセットしても、タスク `t4` 用の `Event2` には影響しません。あるタスクがターミネートする時、そのタスクが所有するすべてのイベントはクリアされます。

7.1.1 イベント待ちタスクの定義

イベント待ちを行うタスクは、RTA-OSEK GUI で指定します。あるイベントを待つタスクを宣言すると、そのタスクは自動的に拡張タスクとして扱われるようになります。

図 7-4 では、`Event1` というイベントが宣言され、`t1` および `t3` というタスクがそのイベントを待つように設定されています。

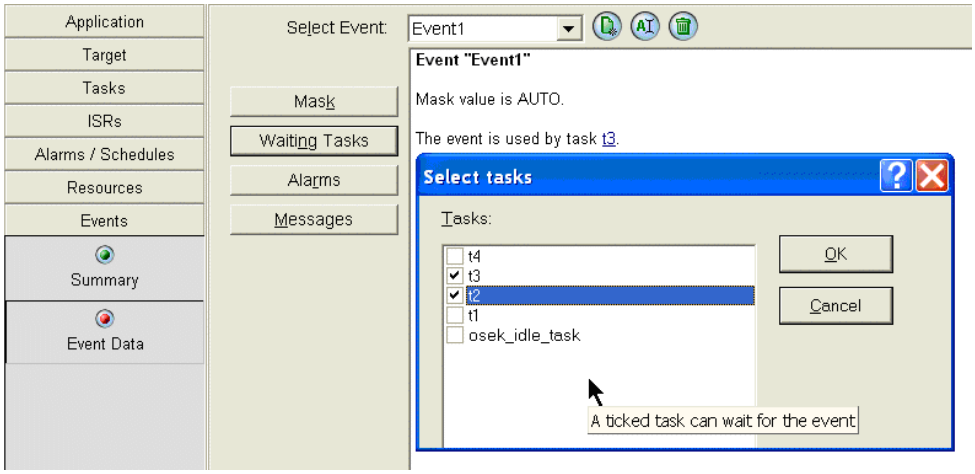


図 7-4 イベントを待つタスクを選択する

イベントを待つ拡張タスクは通常、自動起動され、ターミネートすることはありません。タスクの実行が開始されるときに、そのタスクが所有するすべてのイベントは RTA-OSEK コンポーネントによりクリアされます。

7.2 イベントを待つ

タスクがイベント待ちを行う際は、そのタスクが `WaitEvent(EventMask)` を呼び出します。この `EventMask` は RTA-OSEK GUI で宣言されたイベントマスクと一致していなければなりません。

`WaitEvent()` は、1 つのイベントを引数として受け取ります。その時点におけるイベントの状態に応じて、以下の 2 とおりの処理が行われます。

1. イベントがまだ発生していない
   呼び出し元のタスクはウェイト状態になり、RTA-OSEK は、レディ状態のタスクの中から優先度の最も高いタスクを実行します。
2. イベントがすでに発生している
   呼び出し元のタスクはそのまま実行状態を保ち、`WaitEvent()` の直後の命令文から続行されます。

7.2.1 シングルイベント

1 つのイベントのみを待つ場合の処理は、API コールにイベントマスクを渡すだけです。コード例 7-1 にその例を示します。

```
TASK(Task1) {
  ...
  WaitEvent(Event1);
  /* Task enters waiting state
     if event has not happened */

  /* Otherwise task continues execution
     at next statement */
  ...
}
```

コード例 7-1 1 つのイベントを待つ

一般に、イベント待ちを行うタスクの構造は、以下のようなイベント待ち命令による無限ループとなります。

```
TASK(Task1){
  /* Entry state */
  while(true){
    WaitEvent(Event1);
    /* State 1 */
    WaitEvent(Event2);
    /* State 2 */
    WaitEvent(Event3);
    /* State 3 */
  }
}
```

コード例 7-2 イベントを使用した単純なステートマシン

7.2.2 マルチイベント

OSEK において、イベントは 1 バイトのビットマスクで表現されるため、複数のビットを 'OR' 条件で組み合わせることにより、同時に複数のイベントを待つことができます。

あるタスクが複数のイベントを待っている場合、待っているイベントのうちのいずれか 1 つが発生した時点で待ち状態が解除され、レジュームされます。そのため、複数のイベントの待ち状態からレジュームした際は、どのイベントが発生したのかをチェックする必要があります。

そのため、OSEK には `GetEvent()` という API が用意されていて、現在そのタスク用にセットされているイベントの状態を取得することができます。

コード例 7-3 は、1 つのタスクで一度に複数のイベントを待ち、レジューム後にどのイベントがセットされているかをテストしている例です。

```
TASK(Task1){
  EventMaskType WhatHappened;

  while(true){
    WaitEvent(Event1|Event2|Event3);
    GetEvent(Task1, &WhatHappened);
    if( WhatHappened & Event1 ) {
      /* Take action on Event1 */
      ...
    } else if( WhatHappened & Event2 ) {
      /* Take action on Event2 */
      ...
    } else if( WhatHappened & Event3 ) {
      /* Take action on Event3 */
      ...
    }
  }
}
```

コード例 7-3 複数のイベントを待つ

7.3 イベントをセットする

イベントのセットは、SetEvent() を使用して行います。

SetEvent() には 2 つの引数（タスクとイベントマスク）があります。この関数は、指定されたタスクについて、イベントマスクに定義されているイベントをセットします。同じイベントを共有する他のタスクに対してはイベントをセットしません。

1 つのタスクに対して同時に複数のイベントをセットするには、複数のビットを 'OR' 条件で組み合わせた 1 バイトのイベントマスクを SetEvent() に渡します。

サスペンド状態のタスクについてイベントをセットすることはできない [1] ので、イベントをセットする前に、対象タスクがサスペンド状態になっていないことを確認する必要があります。

イベント待ち状態にある拡張タスクは、自分が待っているイベントのうちのどれか 1 つがセットされると、ウェイト状態からレディ状態に移行します。

コード例 7-4 に、タスクがイベントをセットする例を示します。

```
TASK(Task1) {

  /* Set a single event */
  SetEvent(Task2, Event1);

  /* Set multiple events */
  SetEvent(Task3, Event1 | Event2 | Event3);
  ...
  TerminateTask();
}
```

コード例 7-4 イベントをセットする

多数のタスクが 1 つの共通なイベントを待つことができますが、コード例 7-4 を見るとわかるように、イベントをブロードキャストするメカニズムは存在しません。つまり、ある 1 つのイベントを待っている複数のタスクに対して 1 回の API コールだけでそのイベントの発生を知らせる、ということはできません。これを行うには、RTA-OSEK のタスクセットを使用する必要があります。

イベントは、OSEK アラームとメッセージによってセットすることもできます。

7.3.1 静的インターフェース

RTA-OSEK には、SetEvent() の静的バージョンが用意されています。この場合、タスクとイベントマスクが、API の呼び出し文に埋め込まれます。ただしこれはシングルイベントの場合にのみ使用でき、マルチイベントには対応していません。

```
TASK(Task1) {

  /* Set a single event */
  SetEvent_Task2_Event1();

  TerminateTask();
}
```

コード例 7-5 SetEvent( ) の静的インターフェース

7.3.2 アラームを用いてイベントをセットする

アラームを使用して、ターミネートしない拡張タスクを周期的に起動することができます。アラームが満了するたびにイベントがセットされ、そのイベントを待っていたタスクがウェイト状態からレディ状態に移行します。

---

1. このことからも、拡張タスクのボディはイベントを待つ無限ループでなければならないことがわかります。

7.3.3 メッセージを用いてイベントをセットする

COM メッセージを設定し、メッセージによってイベントがセットされるようにすることもできます。設定されたメッセージが送信されるたびにイベントがセットされ、そのイベントを待っていたタスクがウェイト状態からレディ状態に移行します。

7.4 イベントをクリアする

イベントのセットはどのタスクや ISR でもできますが、イベントをクリアできるのはそのイベントの所有者だけです。

あるイベント待ちのタスクに対してイベントが発生し、タスクがレジュームした後、そのタスクが同じイベントに対する `WaitEvent()` を再度呼び出すと、そのイベントはまだセットされた状態になっているため、`WaitEvent()` はすぐに終了し、タスクはウェイト状態にならずにそのまま実行されてしまいます。

そのため、同じイベントを続けて待つ場合は、最後に発生したイベントをクリアする必要があります。

イベントのクリアは、`ClearEvent(EventMask)` を使用して行います。この `EventMask` は、RTA-OSEK GUI で宣言されたものと一致していなければなりません。

コード例 7-6 に、`ClearEvent()` の一般的な使用例を示します。

```
TASK(Task1){
  EventMaskType WhatHappened;
  ...
  while( WaitEvent(Event1|Event2|Event3)==E_OK ) {
    GetEvent(Task1, & WhatHappened);
    if(WhatHappened & Event1 ) {
      ClearEvent(Event1);
      /* Take action on Event1 */
      ...
    } else if( WhatHappened & (Event2 | Event3 ) {
      ClearEvent(Event2 | Event3);
      /* Take action on Event2 or Event3*/
      ...
    }
  }
}
```

コード例 7-6 イベントのクリア

7.5 アイドルタスク内でのイベント待ち

前述のように、アイドルタスクはシステム内で優先度が最低のタスクですが、RTA-OSEK コンポーネントではアイドルタスクはイベントを待つことができます。つまり、アイドルタスクを拡張タスクにすることができます。

この「拡張アイドルタスク」は一般の拡張タスクとは異なり、`WaitEvent()` を呼び出したときでもスタックの割り当てをはずされないので、RTA-OSEK は現在のコンテキストをセーブするためにメモリをアロケートする必要はありません。

これらのことは、システムの最適化に役立ちます。ユーザーのアプリケーションに必要な拡張タスクが 1 つだけの場合は、「拡張アイドルタスク」を使用することにより、アプリケーションの他の部分の実行時間やメモリスペースの負担をなくすことができます。

アイドルタスクを唯一の拡張タスクとして持つシステムは、完全な基本コンフォーマンスクラスのシステムとまったく同じパフォーマンスを発揮できます。

このようなアイドルタスクを使用することにより、システム全体を「BCC」の状態にしてタイミング分析できる状態を保ったまま、拡張タスクを使うことが可能となります。この方法においては、拡張タスク以外の部分のタイミング挙動が犠牲になりません。

7.6 まとめ

- イベントは、拡張タスクによって使用される同期化オブジェクトです。
- イベントは 1 つのタスクだけに所有されます。
- タスク、ISR、アラーム、およびメッセージを使用してイベントをセットできます。

8 メッセージ

タスクと割込みは、通信を行う必要がある場合があります。たとえば、通信バス割込みが情報をタスクに渡すことにより、共有バッファから何バイト読み取るべきかをタスクに指示するようにすることもできます。

オブジェクト間の通信は、**メッセージの受渡し**により実現できます。RTA-OSEK では、メッセージ受渡しは**非同期**に行われます。つまり、センダはメッセージを送信してからも実行を続けます。レシーバは実行開始時に、送信されたメッセージを受け取ります。

メッセージは 1 つの CPU 上のオブジェクト間で送信されるだけなので、データ転送はすべてメモリ間転送です。「転送エラー」という概念はありません。

8.1 OSEK 内での通信

OSEK オペレーティングシステム内でのメッセージ受渡しは OSEK COM（**Com**munication）規格の一部に定義されています。

RTA-OSEK 内のメッセージ受渡しは、内部タスクと割込みの通信に関する OSEK COM コンフォーマンスクラス CCCA（'non-queued messages'：非キューイングメッセージ）および CCCB（'queued message'：キューイングメッセージ）に適合しています。CCCB は、タスクと ISR の内部通信のための機能で、キューイングメッセージ転送と非キューイングメッセージ転送の両方をサポートしています。

8.1.1 OSEK COM のバージョン

RTA-OSEK には 3 つの COM 機能が含まれ、それぞれ以下の COM バージョンをサポートしています。

- **COM2**：OSEK COM V2.2.2
- **COM3**：OSEK COM V3.0.3
- **RTA-COM**：AUTOSAR COM V1.0

> **重要**
> COM3 と RTA-COM（AUTOSAR COM）は RTA-COM 製品の一部で、RTA-OSEK には含まれません。

本章では、COM2 について詳しく説明します。RTA-COM のコンフィギュレーションと使用方法については 『RTA-COM ユーザーズガイド』を参照してください。

8.2 メッセージの設定

RTA-OSEK の通信機能を使用する場合は、RTA-OSEK GUI で COM メッセージを宣言する必要があります。

COM メッセージの設定は、以下のようなステップで順に行います。

- メッセージを宣言する
- センダとレシーバを宣言する
- アクセサを定義する
- 転送メカニズムを定義する

これらの各段階について、これから詳しく説明します。

8.2.1 メッセージを宣言する

メッセージの宣言は、RTA-OSEK GUI を用いて行います。新しいメッセージは、図 8-1 の例のように、アプリケーションに追加されます。

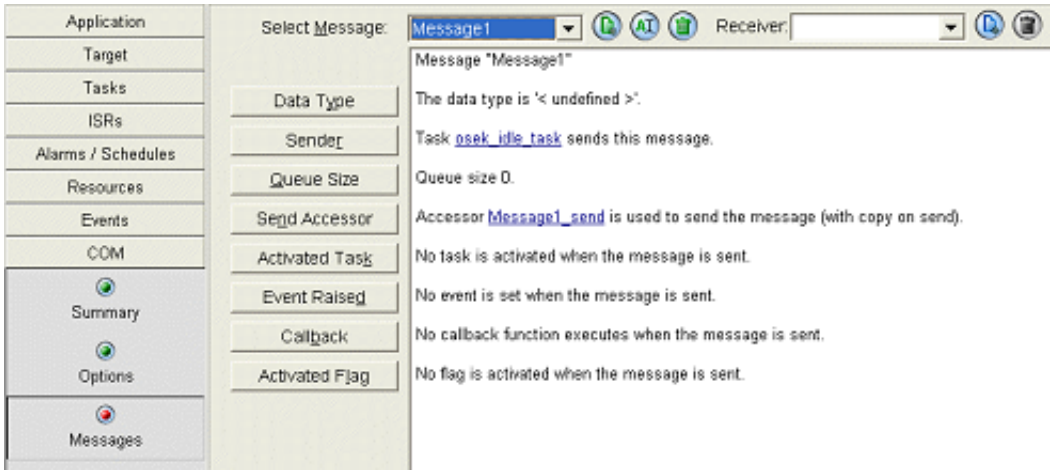


図 8-1 新しいメッセージを宣言する

メッセージの基本的な属性には以下のものがあります。

- 名前
  ランタイムにメッセージを参照する際に使用されます。
- データタイプ
  メッセージの内容を定義します。これはメッセージデータの C データ型です。`unsigned char` などの単純な型と `struct myMessage` のような複雑な型があります。

図 8-2 は、整数データ型をメッセージタイプとして使用する、`Message1` という名前のメッセージを示しています。

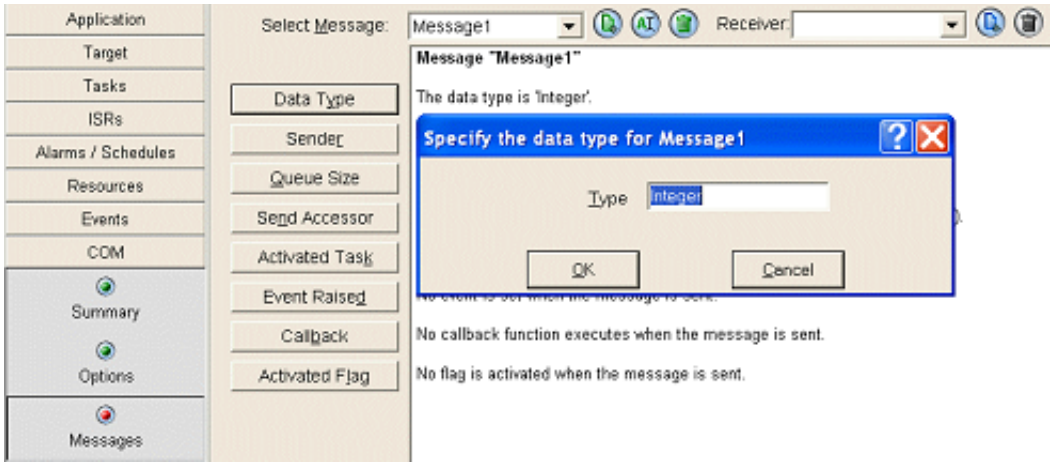


図 8-2 メッセージのデータ型を定義する

COM はメッセージのタイプや内容を知る必要はありませんが、これらの情報は、ビルド時において、RTA-OSEK がメッセージ用に適切な量のメモリをアロケートできるようにするために必要です。

整数、配列、文字列、およびリンクされたリストなど任意のタイプのデータをメッセージとして渡すことができます。データタイプが標準の C または RTA-OSEK のデータ型でない場合は、それをファイル内で宣言する必要があります。

デフォルトでは、この情報は **`comstrct.h`** というファイル内に定義されますが、別のファイルを使用することもできます。

このファイルの名前の定義は、RTA-OSEK GUI の COM Options を使用して行います。図 8-3 の例では、`comstrct.h` という既存のファイルの名前が変更されています。#include ファイルが必要ない場合は、その名前を削除してブランクのままにしておいてください。

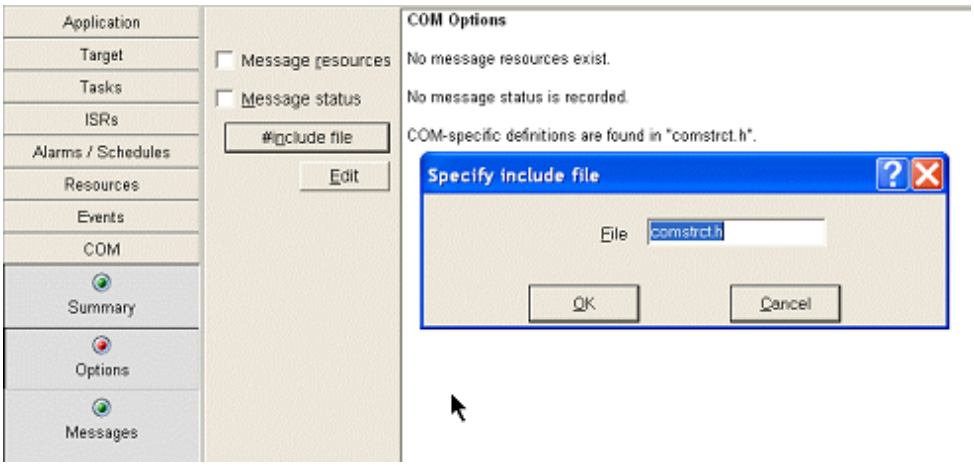


図 8-3 インクルードファイル名を定義する

COM メッセージとして定義されたタイプは、完全な C 言語のデータ型、つまり変数を宣言する際に使用できるものでなければなりません。（さらに RTA-OSEK はこの型をメッセージアクセサの生成に使用します。）たとえば、**comstrct.h** というファイルの内容は、以下のようになります。

```
struct myStruct
{
   char a;
   char b;
};

typedef struct
{
   char x;
   char y;
}
myRecord;

typedef char myrray[10];
```

メッセージタイプには以下の型を使用できます。

```
int
struct myStruct
myRecord
myArray
```

8.2.2 センダとレシーバを宣言する

1 つのメッセージは 1 つのセンダからしか送信されませんが、複数のレシーバがそれを受信することができます。このメカニズムにより、システム全体に情報をブロードキャストすることができます。

メッセージのセンダとレシーバは、RTA-OSEK GUI を使用して定義する必要があります。タスクとカテゴリ 2 の ISR のどちらも、センダにもレシーバにもなることができます。

図 8-4 では、`Message1` のセンダとして `Task1` が定義されています。各メッセージについてセンダが 1 つしか定義できないようになっていることに注意してください。

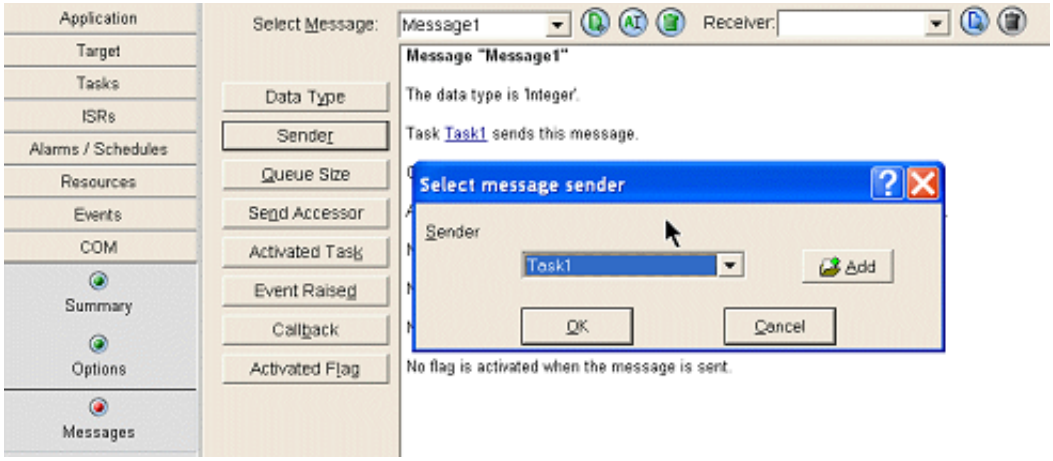


図 8-4 メッセージのセンダを宣言する

図 8-5 では、`Message1` のレシーバとして `Task2` が追加されています。このように、各メッセージに任意の数のレシーバを定義できます。

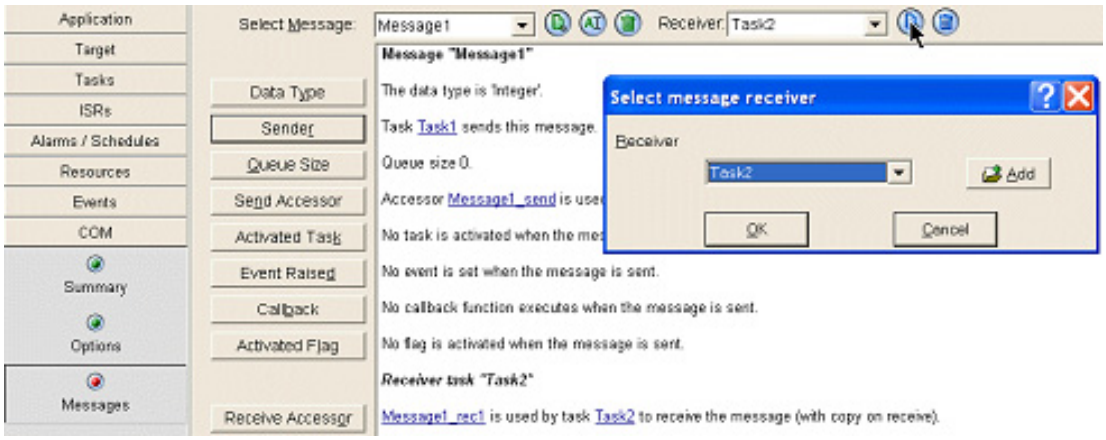


図 8-5 メッセージにレシーバを追加する

8.2.3 アクセサを定義する

センダとレシーバは**アクセサ**を使用してメッセージデータを操作します。アクセサは、アプリケーションが API コールを用いてメッセージの送受信を行うためのものです。

アクセサは、センダとレシーバの両方について宣言する必要があります。また、タスクまたは ISR の各メッセージペアに固有なものです。

アクセサはデータオブジェクトへの参照で、タイプはメッセージと同じです。メッセージの特性により、アクセサはメッセージ内の実データかメッセージデータのコピーのどちらかを参照することができます。

図 8-6 の例では、RTA-OSEK GUI を使用して `Message1_send` という名前の送信アクセサを作成しています。

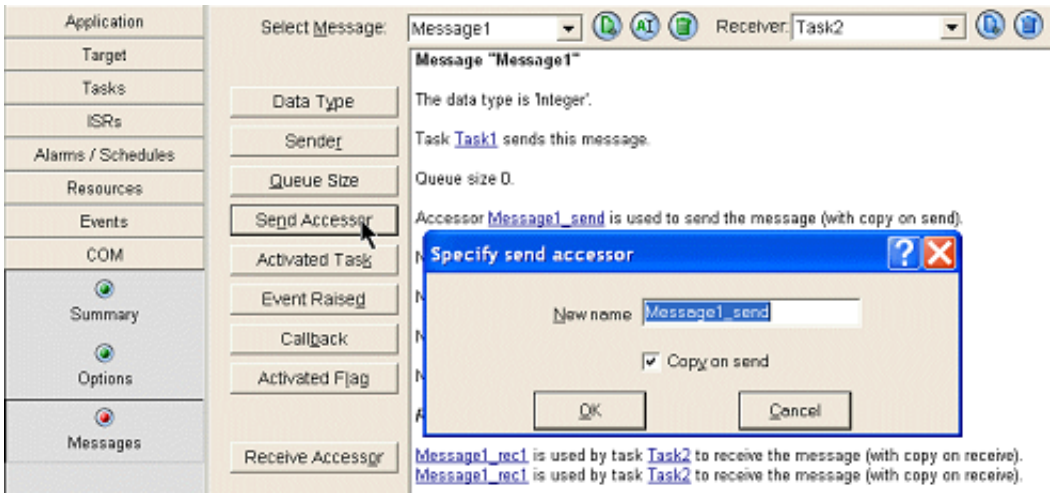


図 8-6 送信アクセサを定義する

図 8-7 は、タスクや ISR の間でのメッセージ受渡しにアクセサが使用されるようすを示しています。

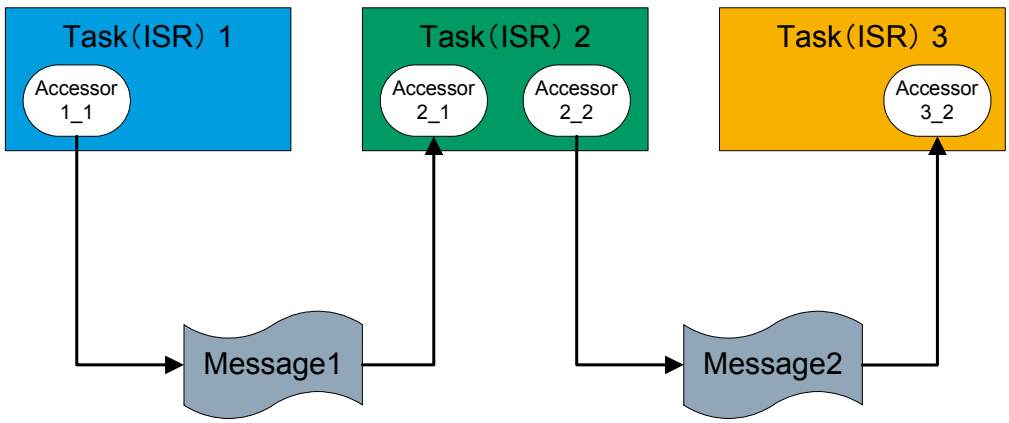


図 8-7 アクセサを使用してメッセージデータの送受信を行う

複数のタスクまたは ISR が同じメッセージの送受信を行う場合は、各タスクまたは ISR ごとに異なるアクセサを使用する必要があります。RTA-OSEK はセンダ用には `<MessageID>_send` というアクセサを作成し、レシーバ用には `<MessageID>_recN`（`N` はレシーバ番号）というアクセサを作成します。アクセサ名は各タスクまたは各 ISR に固有です。

これらのアクセサのシンボル名は変更可能です。図 8-8 の例では、`Message1` について定義されている `Message1_rec1` というアクセサの名前を `Rec1Message1` に変更しています。

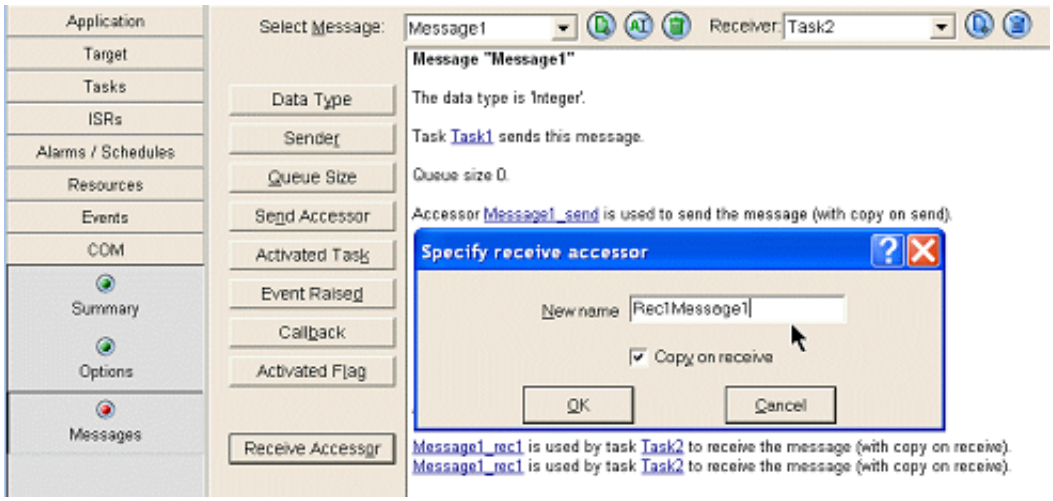


図 8-8 受信アクセサを定義する

アクセサは、メッセージと同じ型の変数です。そのため、8.2.1 項のコード例の続きとして、もし `myArray` という型の `message1` というメッセージがあり、そのメッセージに `message1_send` というアクセサが定義されていて、さらに `myStruct` という型の `message2` というメッセージに `message2_send` というアクセサが定義されている場合、以下のようにしてアクセサを使用します。

```
for (i = 0; i < 10; i++)
{
   cmessage1_send[i] = (char) i;
};

message2_send.a = 1;
message2_send.b = 2;
```

重要

アクセサを API 関数に渡すときには、必ずアクセサのアドレスを渡してください（すべての API 関数で `&AccessorName` を使用してください）。

8.2.4 転送メカニズムを定義する

アクセサはメッセージ領域へのアクセスを可能にしますが、メッセージの送信方法は規定しません。OSEK COM には次の 2 種類のメッセージ転送メカニズムが定義されています。

- `WithCopy`
- `WithoutCopy`

WithCopy モード

WithCopy 転送メカニズムでは、アクセサはメッセージのローカルコピーを参照します。メッセージが送信されると、RTA-OSEK コンポーネントはローカルコピーの内容をメッセージバッファにコピーします。

メッセージが受信されると、メッセージロケーションの内容がレシーバ用のローカルコピー領域にコピーされます。

`WithCopy` モードは、タスクと ISR のどちらでも使用できます。

重要

`WithCopy` は、ISR で使用できる唯一の転送モードです。

図 8-9 の例では、どちらのアクセサも `WithCopy` として宣言されています。（この例では、大きい矢印がメッセージデータのコピーを表し、小さい矢印が参照を表しています。）

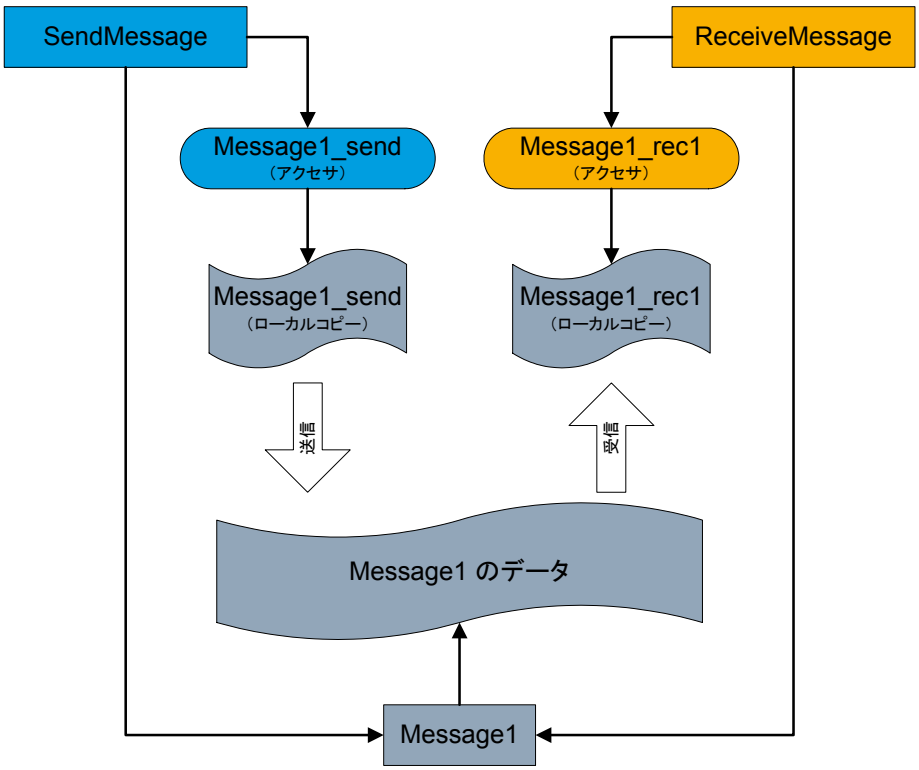


図 8-9 WithCopy モード時のメッセージの送受信

このメカニズムは、メッセージタイプがサイズの大きい型であったり複雑な構造体である場合は、システムの負担が大きくなってしまう可能性があります。しかし、`WithCopy` モードではセンダとレシーバの両方がメッセージそのものに影響を与えずにメッセージのコピーを操作することができます。

WithoutCopy モード

タスクは、**WithoutCopy** モードのメッセージ転送を使用できますが、ISR ではこのモードを使用できません。

`WithoutCopy` モードでは、アクセサがメッセージのデータバッファを直接参照するので、前の例で見たような余分なコピー処理は必要ありません。

図 8-9 を次の例（図 8-10）と比べてみると、図 8-10 のメッセージ転送では両方のアクセサが `WithoutCopy` として宣言されています（この図でも、小さい矢印は参照を表しています）。

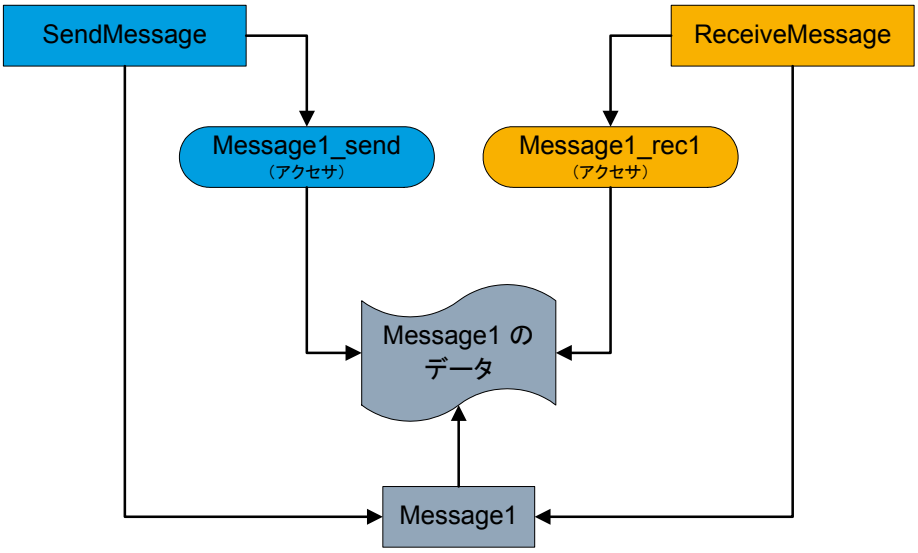


図 8-10 WithoutCopy モード時のメッセージの送受信

`WithoutCopy` モードでは、一度に多くのタスクが同じデータ領域にアクセスできます。この際、アクセスの衝突が起こらないようにするのは、ユーザーの責任となります（メッセージの読み書きについて平行処理を行うことをお勧めします）。

RTA-OSEK GUI を使用して各メッセージ用の**メッセージリソース**を作成することができます。メッセージリソースは標準の OSEK リソースと似ていますが、各メッセージに固有のものです。

メッセージリソースの名前はメッセージ名と同じです。また、そのメッセージにアクセスできる各タスクと ISR で使用できます。メッセージリソースを取得すると、システムのアクティブ優先度が、そのメッセージにアクセスできるタスクや ISR の中で最高の優先度まで引き上げられます。

このリソースへのアクセスは `GetMessageResource(MessageID)` と `ReleaseMessageResource(MessageID)` という API コールにより行われます。これらのコールの詳細については『RTA-OSEK リファレンスガイド』に記載されています。

8.3 メッセージの送受信

メッセージの送信は、`SendMessage(MessageID, &AccessorID)` を使用して行います。

メッセージを送信したいタスクまたは ISR には、そのメッセージ用の送信アクセサが宣言されていなければなりません。この宣言は RTA-OSEK GUI で行う必要があります。無効なメッセージまたは無効なアクセサを使用してこの API を呼び出すと、エラーになります。

8.3.1 メッセージを送信する

メッセージを送信するには、以下の手順を実行してください。

- メッセージとして送信するデータを、`Accessor` が指し示しているデータバッファにコピーします。使用するアクセサはメッセージを送信するタスク用または ISR 用として有効なものでなければなりません。
- `SendMessage(Message, &Accessor)` を呼び出します。

  `Message` は宣言されているメッセージの識別子です。`Accessor` はアクセサへの参照で、メッセージを送信するタスクまたは ISR のみが使用できます。

コード例 8-1 では、`DataArrived` というメッセージが ISR により送信されています。このメッセージのタイプは `struct MyMessage` として定義されています。

```
struct MyMessage {
      chartext[6];
      boolaFlag;
};
ISR (MessageArrived) {
```

```
    /* Prepare data for sending. */
    DataArrived_send.aFlag = true;
    memcpy(DataArrived_send.text, "HELLO", 6);

    /* Send the message. */
    SendMessage(DataArrived, &DataArrived_send);

}
```

コード例 8-1 ISR からメッセージを送信する

8.3.2 メッセージを受信する

メッセージの受信は、`ReceiveMessage(MessageID, &AccessorID)` という API コールを使用して行います。メッセージを受信したいタスクまたは ISR には、そのメッセージ用の受信アクセサが RTA-OSEK GUI で宣言されていなければなりません。

メッセージを受信するには、以下の手順を実行する必要があります。

- `ReceiveMessage(Message, &Accessor)` を呼び出します。
  `Message` は宣言されているメッセージの識別子、`Accessor` はメッセージを受信するタスクまたは ISR が使用を許されているアクセサの参照です。
- この API 関数の処理が終わったら、`Accessor` が指し示しているデータバッファのデータにアクセスします。

コード例 8-2 は、メッセージを受信する `ProcessData` というタスクのコードです。

```
TASK(ProcessData) {

    char buffer [6];

    /* Receive the message. */
    ReceiveMessage(DataArrived, &DataArrived_Rec1);

    /* Retrieve data from accessor. */
    memcpy(buffer, DataArrived_Rec1.text, 6);

}
```

コード例 8-2 メッセージを受信するタスク

8.4 COM の開始と終了

`StartCOM()` は、メッセージの送信または受信の前に呼び出されなければなりません。この API 関数は、実装されたコードに固有な内部状態と変数を初期化します。

`StopCOM(COM_SHUTDOWN_IMMEDIATE)` を使用すると、いつでも COM を終了することができます。拡張（'extended'）エラーチェックモードにおいては、この関数呼び出しにより、後続の送信処理と受信処理が `E_COM_SYS_STOPPED` というステータスを返すようになります。

8.5 COM の初期化とシャットダウン

COM の初期化や終了を行うための API 関数が用意されています。これらの関数は、外部ハードウェア（CAN ドライバなど）を用いて他のプロセッサにメッセージを渡す場合に使用するためのものです。これは RTA-OSEK では直接はサポートしていませんが、他のアドオンライブラリを使って実現することができます。

`InitCOM()` は、ネットワークハードウェアを初期化するための API 関数です。この関数は `StartCOM()` の前に実行される必要があり、通常は起動フックから呼び出されます。

CloseCOM() は、ネットワークハードウェアを非起動状態にするための API 関数です。この関数は必ず StopCOM() の後に実行される必要があり、通常はシャットダウンフックから呼び出されます。

コード例 8-3 では、StartupHook() と ShutdownHook() により COM の初期化とシャットダウンを行っています。

```
OS_HOOK(void) StartupHook(void) {
   InitCOM;
}

OS_HOOK(void) ShutdownHook(StatusType status) {
   CloseCOM();
}
```

コード例 8-3 COM の起動とシャットダウンを行うためのフックルーチン

コールバック関数 MessageInit() を使用して、ユーザーメッセージオブジェクトを初期化することができます。これは、StartOS() から自動的に呼び出される、ユーザー定義関数です。デフォルト状態においては RTA-OSEK コンポーネントがこの関数を提供するので、これを任意に書き換えることができます。コード例 8-4 に例を示します。

```
StatusType MessageInit(void)
```

コード例 8-4 MessageInit( ) コールバック関数

MessageInit() が返すステータスタイプは、StartCOM() のステータスコードとして返されます。

8.6 キューイングされるメッセージ（'queued message'）

前項までに説明されているメッセージはキューイングされないメッセージですが、RTA-OSEK コンポーネントは、キューイングされるメッセージ転送機能を備えた COM コンフォーマンスクラス B（CCCB）をサポートしています。

キューイングされるメッセージについては、RTA-OSEK コンポーネントは内部 FIFO（先入れ先出し）キューの管理を行います。ユーザーは、RTA-OSEK GUI でメッセージを設定するときにキューのサイズを定義する必要があります。これを行うと、RTA-OSEK コンポーネントはそのキューのためにどのくらいのスペースを確保すればよいかを知ることができます。図 8-11 には、キューサイズの定義方法が示されています。

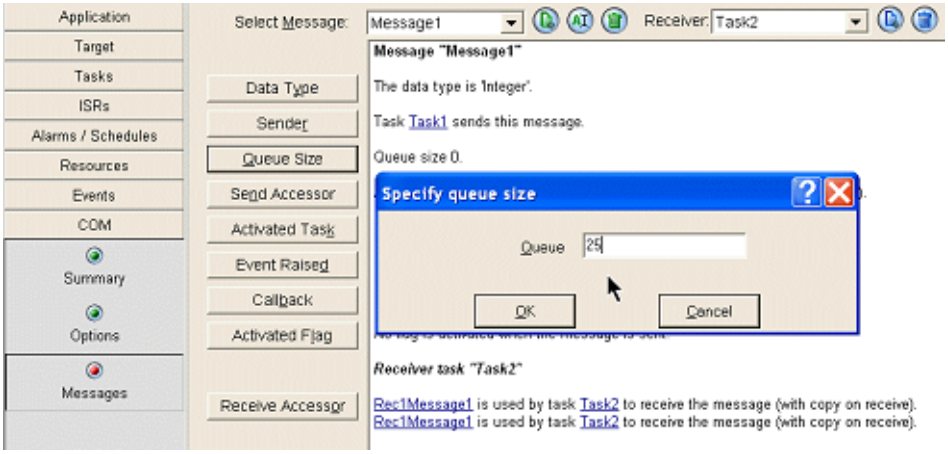


図 8-11 キューサイズを定義する

キューイングされるメッセージは、キューイングされないメッセージと同じメッセージ名とアクセサを持ちますが、両者には重要な相違があります。

キューイングされるメッセージの場合、以下の特徴があります。

- センダとレシーバの転送モードは必ず WithCopy でなければなりません。

- キューイングされるメッセージについては**削除読み取り**（'destructive read'）が行われるため、それぞれのメッセージにつき宣言できるレシーバは 1 つだけです。レシーバがキューの先頭にあるメッセージを読むと、そのメッセージは削除されます。
- ISR からはキューイングされるメッセージを送信することはできません。

8.7 ミックスモード転送（'mix mode transmission'）

8.2.4 項で説明されているように、OSEK COM には `WithCopy` と `WithoutCopy` という 2 種類のメッセージ転送メカニズムが定義されています。

必要に応じて、センダとレシーバで互いに異なるモードを使用してメッセージを転送することもできます。つまり、たとえばメッセージを `WithoutCopy` モードで送信して `WithCopy` モードで受信することも可能です。これは**ミックスモード転送**と呼ばれます。

8.8 メッセージ送信時にタスクを起動する

メッセージ送信時にはタスクを起動することができます。各メッセージにつき 1 つのタスクだけを起動できます。

通常はメッセージのレシーバとなるタスクを起動しますが、別のタスクも起動できます。

ただし、アプリケーションのタイミングの的確性を分析する場合は、メッセージセンダよりも低い優先度のタスクだけを起動するようにしてください。つまり、上方向のタスク起動は許されません。

図 8-12 では、`Message1` が送信された時に起動されるタスクとして `Task1` が選択されています。

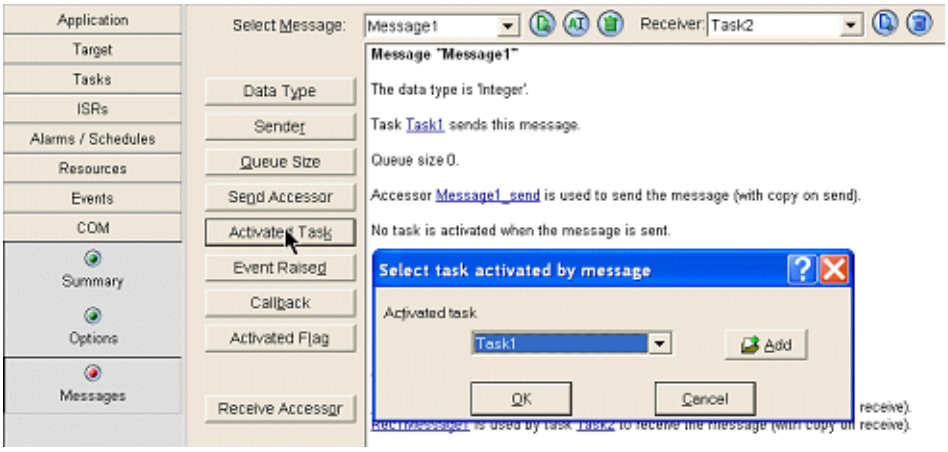


図 8-12 メッセージ送信時にタスクを起動する

8.9 メッセージ送信時にイベントをセットする

拡張タスクがあるメッセージを受信する必要がある場合、メッセージの送信時にイベントをセットしてその受信タスクに知らせることができます。これは、図 8-13 のように設定します。

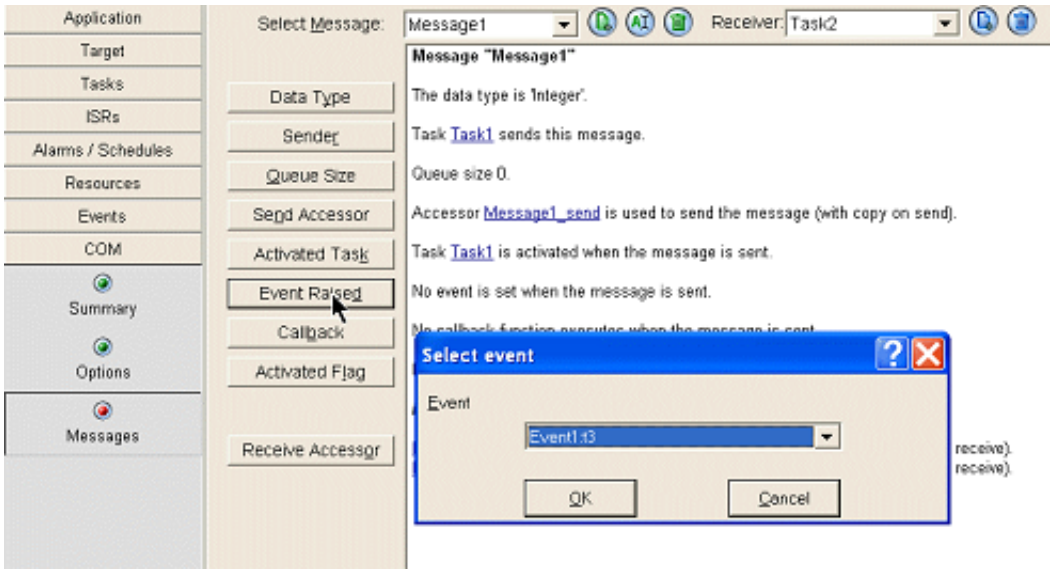


図 8-13 メッセージ送信時にイベントをセットする

8.10 コールバック関数

メッセージに**コールバック関数**を定義することができます。コールバックは、引数を取らない C 関数で、メッセージが送信されたときに RTA-OSEK コンポーネントにより呼び出されます。この C 関数は、ユーザーが責任をもって作成してください。

コールバック関数の例をコード例 8-5 に示します。

```
void MyCallback (void) {
    /* Callback code. */
}
```

コード例 8-5 コールバック関数を作成する

たとえば、デバッグ時にメッセージ送信回数を記録したい場合、呼び出し毎にカウンタをインクリメントするコールバック関数を作成することができます。

コールバック関数は図 8-14 の例のようにして定義します。

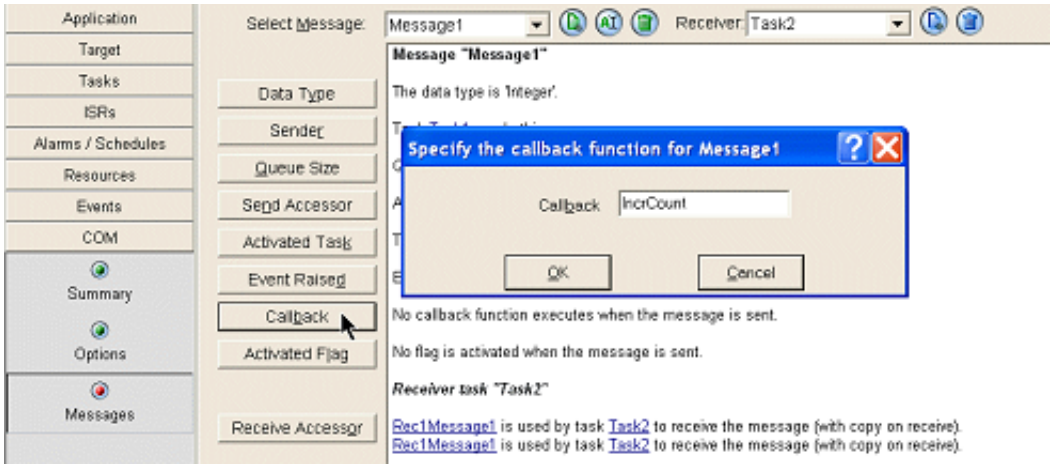


図 8-14 メッセージにコールバック関数を割り当てる

コールバック関数内で使用できる API 関数は、`SuspendAllInterrupts()` と `ResumeAllInterrupts()` のみです。

8.11 フラグの使用

フラグは論理値を取るので、セットまたはアンセットのいずれかの状態になります。メッセージが送信された時にフラグをセットすることにより、メッセージとの同期を行うことができます。このフラグは、`ReceiveMessage()` を呼び出す前に新しいメッセージがあるかどうかを調べる必要がある場合、任意の箇所で使用できます。

図 8-15 では、`Message1` について `Flag1` というフラグ名が定義されています。

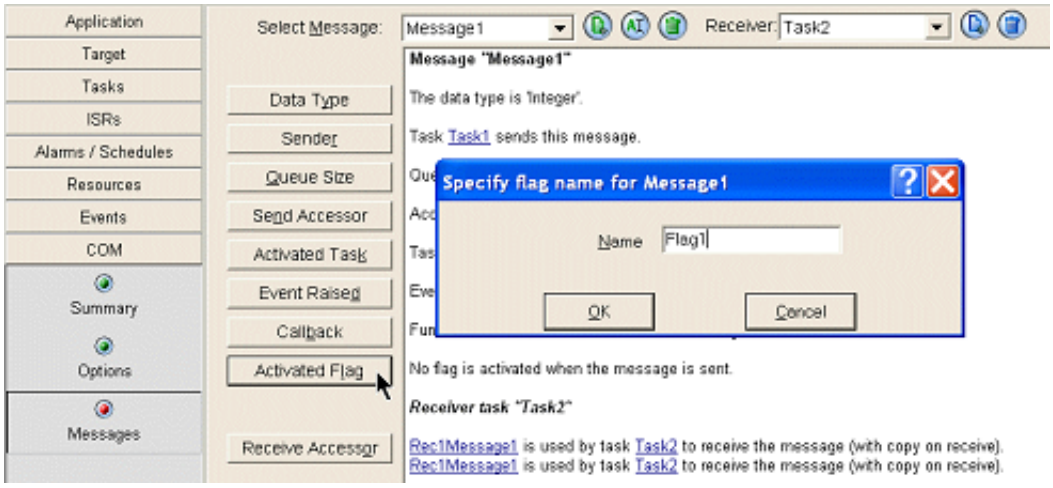


図 8-15 メッセージ送信時にフラグをセットする

メッセージフラグにアクセスできる API 関数は 2 つあります。`ReadFlag()` は指定されたフラグのその時点の状態を返し、`ResetFlag()` は指定されたフラグをクリアします。

> **重要**
> 使用されているフラグ名の妥当性チェックは行われないので、ユーザーの責任において正しいフラグを使用してください。

メッセージにアタッチされているフラグを用いてタスクがメッセージを受信する方法を、コード例 8-6 に示します。

```
TASK(ProcessData)
{
   char buffer [8];

   /* Only receive message if the flag is set. */

   if(ReadFlag (DataHasArrived)) {

      /* Receive the message. */
      ReceiveMessage(DataArrived, &ReceiveAccessor);

      /* Retrieve data from accessor. */
      memcpy(buffer, ReceiveAccessor, 8);

      /* Reset flag. */
      ResetFlag (DataHasArrived);
   }
}
```

コード例 8-6 アタッチされているフラグを使用してメッセージを受信するタスク

8.12 まとめ

- COM には、タスクや ISR の間のメッセージ受渡しを実現する機能があります。
- キューイングされないメッセージについては､1 つのセンダと 1 つまたは複数のレシーバが定義されます。このようなメッセージは `WithCopy` モードでも `WithoutCopy` モードでも送信できます。
- キューイングされるメッセージについては、1 つのセンダと 1 つのレシーバが定義されます。このようなメッセージは `WithCopy` モードでのみ送信できます。
- メッセージの送受信はアクセサを用いて行われます。
- `WithoutCopy` モード使用時とキューイングされるメッセージ使用時には、平行処理を確実に正しく行う必要があります。

9 スティミュラス／レスポンスのモデリングを行うためのヒント

ここまでの各章で、開発工程における RTA-OSEK GUI の使用方法、また RTA-OSEK の各種システムオブジェクトを定義して使用する方法について説明しました。

本章では、タイミング要件をモデリングしてアプリケーションに組み込む方法について説明します。

リアルタイムシステムは入力を受け取って出力を生成しますが、RTA-OSEK においては、入力は**スティミュラス**（'stimulus'）と呼ばれ、出力は**レスポンス**（'response'）と呼ばれます。

正しく機能するリアルタイムシステムを構築するには、以下の点を明らかにする必要があります。

- どの出力がどの入力に対応するのか
- 入力はどのような頻度で発生するのか

RTA-OSEK GUI は、この単純な情報をスティミュラス - レスポンスモデルの形で把握します。しかし、システムを分析するには、さらに他の情報も必要です。これについては、詳しく後述します。

RTA-OSEK GUI を使用すると、システムオブジェクト（タスク、割込み、アラーム、スケジュールなど）の概念を用いて、ユーザーの要件を設計に反映させることができ、定義されたタイミングの的確性を分析することができます。そして RTA-OSEK はコードを生成し、このコードが、ユーザーが作成したコードと共に設計を実現します。

本章では、カウンタ、アラーム、およびスケジュールを使用するシステムを設計する際の仕様決定工程について説明します。

開発工程が完了すると、各スティミュラスがプライマリプロファイルに関連付けられ、各レスポンスは**プライマリプロファイル**（'primary profile'）か**アクティベーテッドプロファイル**（'activated profile'）のいずれかに関連付けられます。

スティミュラスはシステムの内部と外部のどちらで発生するものでもかまいません。外部スティミュラスの例としてはボタン押下などがあり、内部スティミュラスの例としてはターゲットハードウェアからのタイマ割込みなどが考えられます。

通常、スティミュラスは、アプリケーションにおいて「割込み」として発生します。割込み自体をスティミュラスにすることができ、また、RTA-OSEK がアラーム生成などの内部スティミュラスを生成するために割込みを使用することも可能です。スティミュラスがどのような形を取るかは、ユーザーが設計工程において決定します。

スティミュラスが発生したときには、それに対応して、1 つまたは複数のレスポンスがシステム内で生成されなければなりません。スティミュラスと同じく、レスポンスもシステムの内部と外部のどちらで発生するものでもかまいません。外部レスポンスの例としてはハードウェアを作動させることなどがあり、内部レスポンスの例としては何らかの計算が終了して結果が有効になることなどがあります。設計においては、ユーザーはレスポンスを生成するためにプログラムの処理内容を決定していきます。

9.1 スティミュラスとレスポンスの宣言

仕様決定の最初の部分では、システム内のスティミュラスとレスポンスを宣言する必要があります。

スティミュラスとレスポンスのそれぞれの組み合わせには、固有の名前がつきます。スティミュラスを宣言すると、RTA-OSEK GUI はデフォルトのレスポンスをスティミュラスと同じ名前で生成します。ユーザーは必要に応じてこの名前を変更できます。

レスポンス名の変更は図 9-1 のように行います。

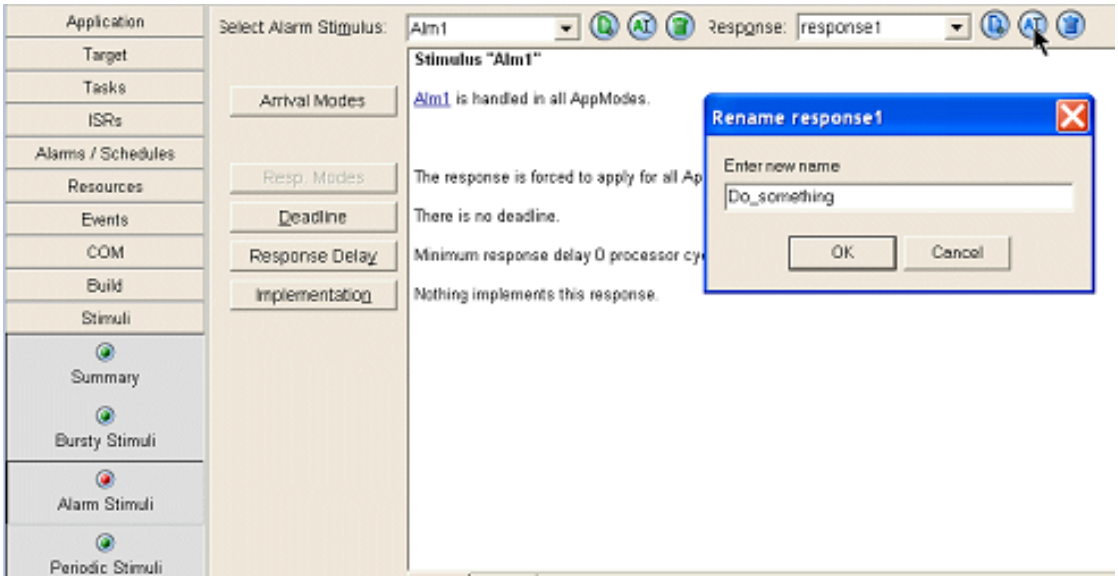


図 9-1 デフォルトレスポンス名を変更する

1 つのスティミュラスに複数のレスポンスを割り当てることができます。その場合、各レスポンスに固有の名前がつきます。たとえば、`10ms_stimulus` というスティミュラスには `checked_vessel_pressure` というレスポンスを割り当て、また `brake_pressed` というスティミュラスには `hydraulics_primed`、`pads_applied`、`brake_lights_on` といったレスポンスを割り当てることができます。

9.2 アライバルタイプ（'arrival type'）とアライバルレート（'arrival rate'）

システムのスティミュラスとレスポンスを宣言することにより、どの入力がどの出力に対応しているかが定義された後は、そのスティミュラスがどのくらいの頻度で発生するかを定義します。

各スティミュラスは、そのパターンによって以下の 3 とおりの**アライバルタイプ**に分けられます。

- バースト（'bursting'）
  バーストアライバルは、一般的に、割込みがスティミュラスの役割を持つケース、つまり、レスポンスを生成するタスクを割込みから直接起動するケースをモデリングする際に使用します。
- 周期的（'periodic'）
  周期的アライバルは、周期的なレスポンス（例： 20ms ごとにレスポンスを生成する）をモデリングする際に使用します。
- 計画的（'planned'）
  計画的アライバルは、非周期的、かつ連続的に発生するスティミュラス（例：10ms、15ms、50ms の各時点でレスポンスを生成する）をモデリングする際に使用します。

周期的アライバルと計画的アライバルについては、その内容を詳細に定義します。設計を行う際は、実際のランタイムにおいて所望の挙動が得られるように十分に考慮する必要があります。

各タイプのアライバルパターンの例を図 9-2 に示します。

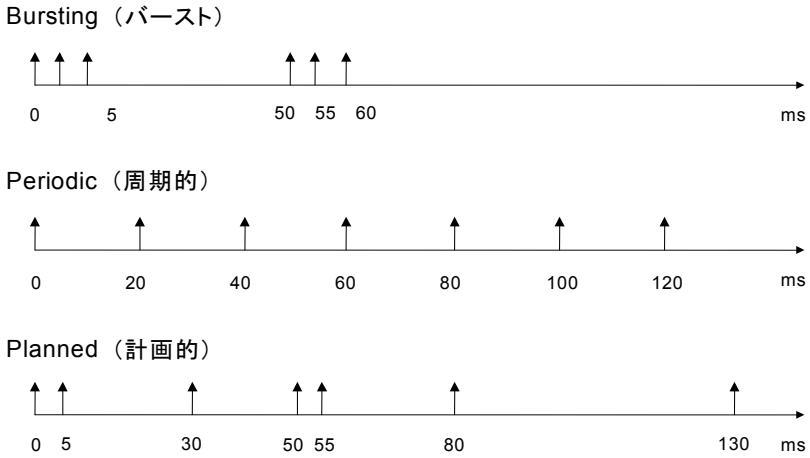


図 9-2 スティミュラスのアライバルパターンの例

各アライバルタイプには、そのパターンに必要なタイミング挙動を表わす**アライバルレート**が定義されます。

バーストアライバルは、タイミング分析にのみ用いられます。周期的アライバルは、分析にも、アプリケーション用のランタイムデータ生成にも用いられます。計画的アライバル場合、実際のタイミングは設計段階まで持ち越されることになります。設計の一部としてスティミュラスが発生するタイミングを定義してください。

9.3 スティミュラスの実装

通常、バーストスティミュラスは、プライマリプロファイルに定義されている ISR により生成されます。バーストスティミュラスを作成するときにユーザーが定義しなければならないのはこれだけです。

周期的スティミュラスと計画的スティミュラスについては、そのスティミュラスが RTA-OSEK コンポーネントでどのように生成されるかをユーザーが決める必要があります。

周期的スティミュラスは、以下の形で実装します。

- OSEK：アラーム
- AUTOSAR：スケジュールテーブル
- RTA-OSEK：周期的スケジュール

計画的スティミュラスは、以下の形で実装します。

- RTA-OSEK：計画的スケジュール

図 9-3 に、スティミュラスの設計方法を示します。

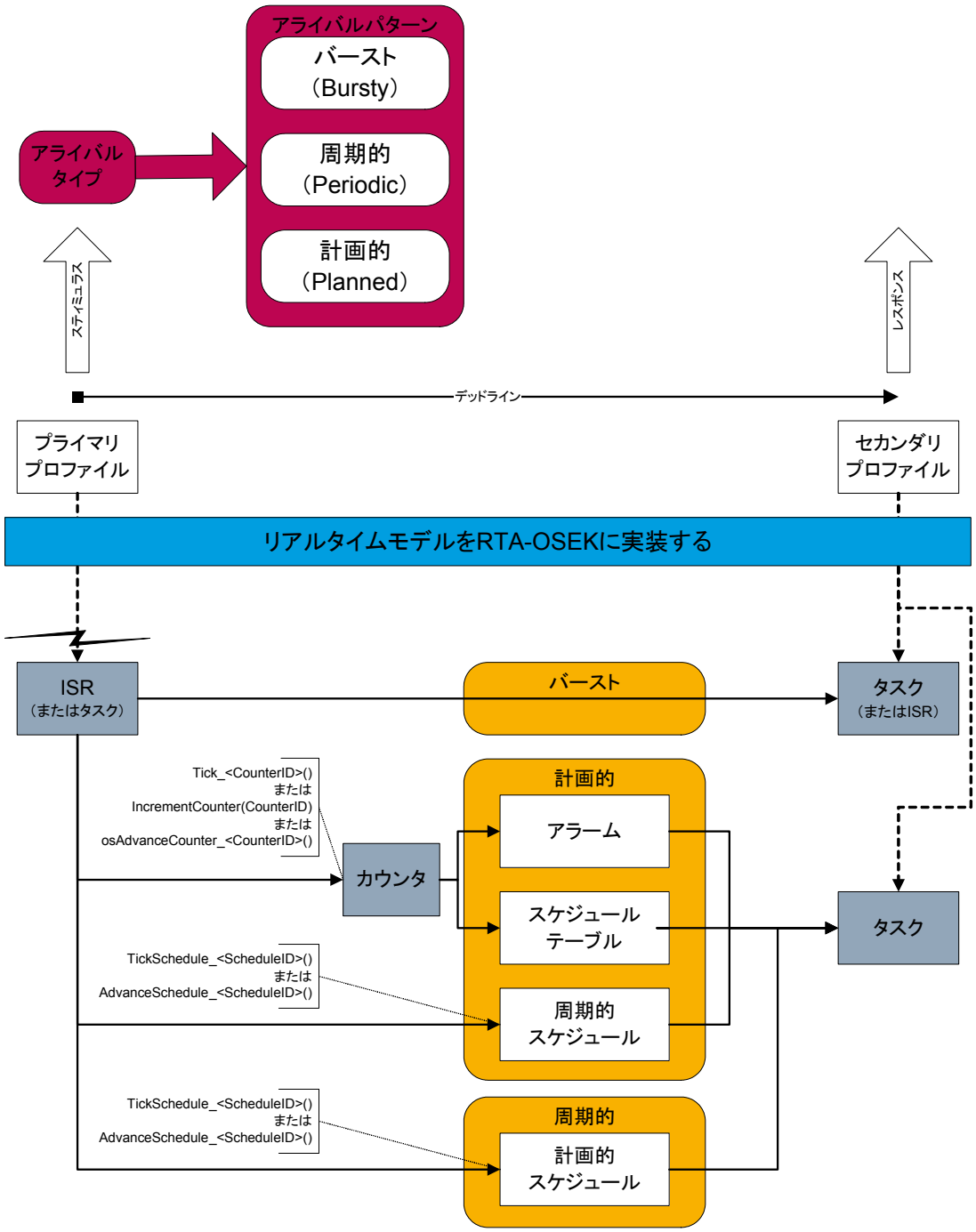


図 9-3 スティミュラスを設計する

9.4 レスポンスの実装

仕様決定時にユーザーが宣言したレスポンスの実際の処理は、ユーザーコード内に実装します。

レスポンスは、アプリケーション内において任意のコードで作成できますが、ユーザーはどのタスクまたは ISR がそのレスポンスを実行するかということを宣言する必要があります。図 9-4 に示す例のように、1 つのタスクまたは ISR に複数のレスポンスを実装することもできます。

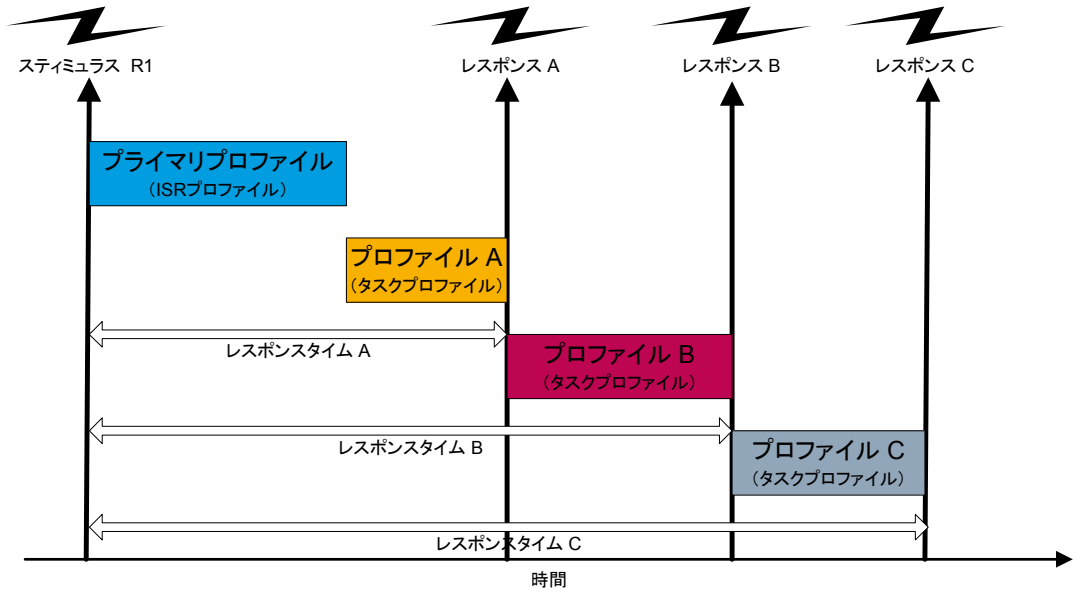


図 9-4 スティミュラスとレスポンスの実装

1 つの ISR 内で、スティミュラスの受け取りと、レスポンスの生成の両方を行うことができます。たとえば、1 つの割込みハンドラの中で割込みソースを受け取り、さらに何らかの処理を行ってから結果を外部に返すことができます。

9.5 まとめ

- 1 つのシステムには必ず入力と出力があります。RTA-OSEK では、入力はスティミュラスと呼ばれ、出力はレスポンスと呼ばれます。
- スティミュラスは、1 つ以上のレスポンスに関連付けられます。
- レスポンスは、ユーザーのアプリケーションコード内でタスクまたは ISR に実装されます。
- スティミュラスにはアライバルタイプとアライバルパターンが定義されます。これらのアライバル情報は分析に使用され、さらに周期的アライバルと計画的アライバルの場合はランタイム情報の生成にも使用されます。
- 周期的スティミュラスと計画的スティミュラスについては、RTA-OSEK 下でアプリケーションが稼働する場合にそのスティミュラスをどのように到達させるかをユーザーが設計する必要があります。

10 カウンタ

カウンタは、OS においていくつの「事象」が発生したかを「チック」という単位でカウントするものです。「1 チック」は抽象的な単位であるため、その実際の意味はユーザーが任意に定義できます。つまり任意のものを「事象」としてカウントすることができます。

以下にその例を示します。

- 時間 ― 「ミリ秒」、「マイクロ秒」、「分」などを単位として、経過時間を測定します。
- 回転 ― 角度や時間などを使用することにより、カウンタによって対象物がどれだけ回転したかがわかります。
- ボタンの押下 ― ボタンを押した回数をカウントします。
- エラー ― エラーの発生回数をカウントします。

カウンタの駆動を行う「ドライバ」としては、ISR（または場合によってはタスク）を利用できます。ドライバは、RTA-OSEK コンポーネントの API 関数を正しく呼び出してカウンタを「チックする」処理を担当します。

10.1 カウンタのコンフィギュレーション設定

カウンタの宣言は、RTA-OSEK GUI を使用して行います。カウンタを宣言するには、以下の項目を定義する必要があります。

- カウンタ名
  RTA-OSEK GUI は、各カウンタ用のハンドルを作成し、カウンタと同じ名前を識別子として用います。
- カウンタのチックレート
- プライマリプロファイル
  これは通常、カウンタをチックするためにユーザーが使用する割込みを定義するものです。

図 10-1 では、`Counter1` というカウンタが宣言されています。

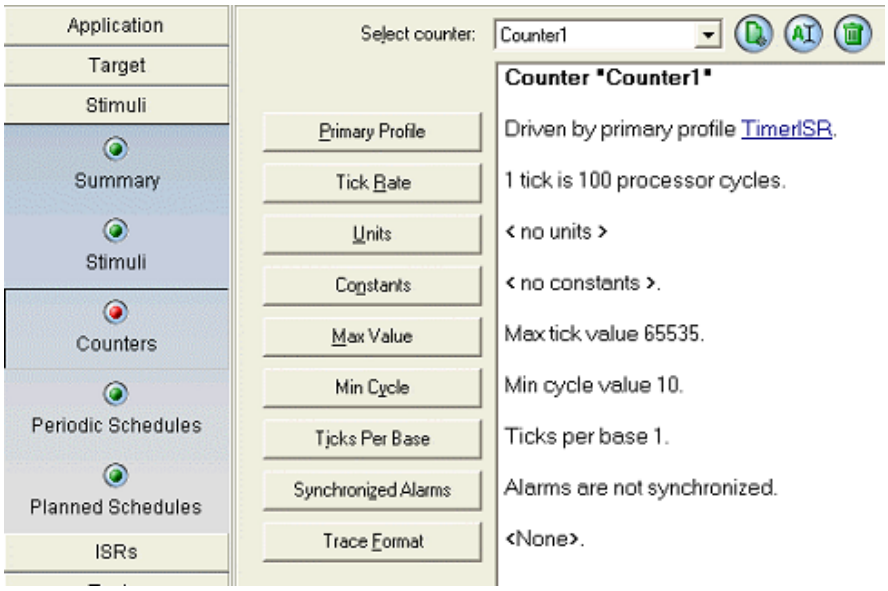


図 10-1 カウンタを宣言する

10.1.1 チックレートの設定

RTA-OSEK GUI でカウンタのチックレートを指定する際、1 チックのスケールを CPU クロックレートまたは実時間（ナノ秒、マイクロ秒、ミリ秒など）の単位で指定できます（図 10-2 参照）。

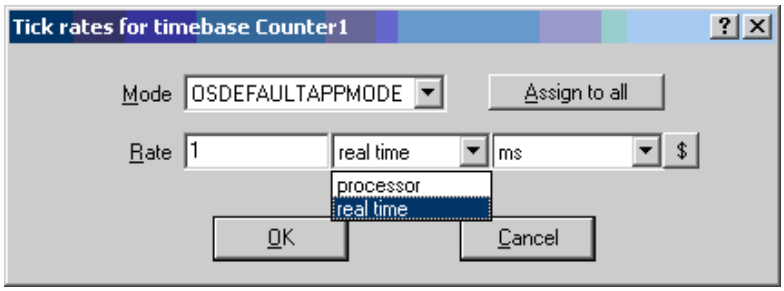


図 10-2 カウンタのチックレートを指定する

一般的なアプリケーションの場合、イベントの相対タイミングは、システム要件として定義された実時間値から求められます。つまりアラーム値とカウンタ値は、実時間単位で指定するのが一般的です。

これらの単位を使用することには大きな利点があります。ターゲットの仕様変更等により CPU のクロックレートが変更された場合、実時間単位が使用されていれば、新しい CPU クロックレートから新しいカウンタタイミング値が自動的にスケーリングされます。

10.1.2 起動タイプ

RTA-OSEK では、カウンタドライバとしてハードウェアを制御することはありません。このため、RTA-OSEK においてさまざまなチックソース（タイマチック、イベントカウント、ボタン押下、TPU 周辺機器など）を非常に容易に使用することが可能となっています。

これは一方で、RTA-OSEK 内で宣言されたすべてのカウンタについて、ユーザーがドライバを作成して OS に組み込む必要があることを意味します。

ドライバの組み込み方法には以下の 2 通りがあります。

1. 「チックドカウンタ」（'ticked counter'）を使用する場合

   カウンタ値は RTA-OSEK 内部に保持されます。ユーザーアプリケーション内で、RTA-OSEK にカウンタを 1 チック分だけインクリメントさせるための API コールを作成します。カウンタは常にゼロからカウントアップされ、MAXALLOWEDVALUE+1 に達した時点でゼロに戻ります。AUTOSAR OS の場合、これは「ソフトウェアカウンタ」と呼ばれます。詳細は 10.2 項を参照してください。

2. 「アドバンスカウンタ」（'advanced counter'）を使用する場合

   カウンタ値は外部のハードウェア周辺機器内に保持されます。この場合、ユーザーアプリケーション内に複雑なドライバを用意する必要があります。このドライバは、必要な数のチックが経過したことを RTA-OSEK に伝えます。カウンタには特別なコールバック関数を使用し、この関数を RTA-OSEK が使用して、要求された数のチックのセット、要求のキャンセル、現在のカウンタ値の取得、カウンタステータスの取得を行います。AUTOSAR OS の場合、これは「ハードウェアカウンタ」と呼ばれます。詳細は 10.1.3 項を参照してください。

要求される分解能が比較的低い場合（例： 1 ミリ秒以上）は、チックドカウンタを使用してください。アドバンスドカウンタは、非常に高い分解能が要求される場合（例：マイクロ秒単位）、または RTA-OSEK を周辺機器（例：TPU、ネットワーク経由のグローバルな時間ソース）と同期させる必要がある場合に使用します。

カウント範囲と分解能に応じて、いずれかのタイプのドライバを選択することができます。

10.1.3 カウンタ属性

各カウンタには以下の属性があります。

- 最大値

  カウンタの最大カウント値を定義します。デフォルトで設定される値はターゲットにより異なります。これは `MAXALLOWEDVALUE` という OSEK 属性に相当します。詳細についてはターゲット用の『RTA-OSEK バインディングマニュアル』を参照してください。

- 最小サイクル値

  サイクル値を設定するときに使用できる最も短い時間の単位を定義します。デフォルトでは、これは 1 チックです。これは `MINCYCLE` いう OSEK 属性に相当します。

- ベースあたりのチック数
  RTA-OSEK はこの属性を使用しないので、この属性には任意の値を設定できます。これは TICKSPERBASE という OSEK 属性に相当します。

以上の値はすべて任意に変更できます。たとえば、16 ビットカウンタの代わりに 8 ビットカウンタを使用したり、デバッグ時に使用する最小サイクル値を定義することができます。これにより、セット呼び出しが実行されたときに、すでに到達している値がカウンタに設定される、という状況を防ぐことができます。

重要
アドバンスドカウンタを使用する場合、MAXALLOWEDVALUE+1 を常に周辺機器内の最大値と一致させる必要があります。

10.1.4 カウンタの単位

カウンタは、単にプライマリプロファイルによって提供される「チック」を蓄えるだけのものです。カウンタはどのようなチックソースからもチックできます。そしてカウンタにアタッチされたすべてのアラームはそのチックソースに関連付けられます。前述のように、RTA-OSEK のようなイベントベースのオペレーティングシステムでは、システム内で捕捉できるものであれば何でもチックとして使用できます。

RTA-OSEK GUI では、カウンタの**単位**（'unit'）を宣言すると、時間とは関係のないチックソースを任意の単位で定義することができます。この単位と時間との変換には、ワーストケースに対応できるレートを定義しなければなりません。これはたとえば、ボタンが押下される最短の間隔の場合もあれば、タイミングホイールの最速の回転数、といった値です。

重要
ワーストケースの変換レートが正しく定義されていないと、アプリケーション内で行われるすべての分析処理が正確に行われなくなってしまいます。

この機能を利用して、たとえば、歯付きタイミングホイールの歯数をカウントし、所定の回転角になったらタスクを起動するようなカウンタを定義することができます。RTA-OSEK GUI で「度（'degree'）」という角度単位を宣言してから 1 回転が 360 度であることを定義します。設定内容を図 10-3 に示します。

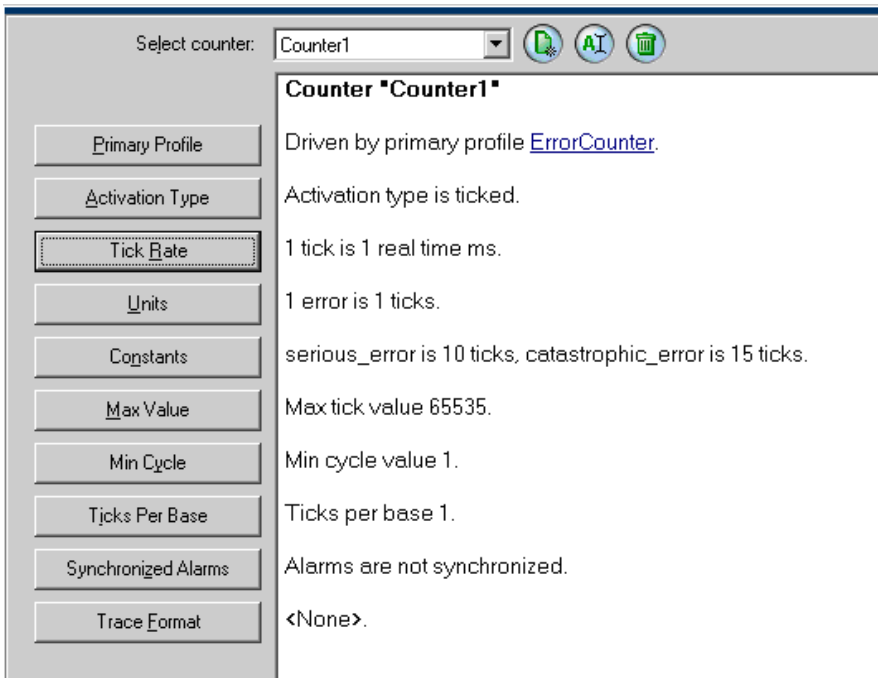


図 10-3 スレッシュホールド値を用いたエラーカウンタを宣言する

10.1.5 カウンタ定数

RTA-OSEK GUI では、一般的なカウンタ値用のシンボリック定数を宣言することができます。これはユーザーアプリケーション内でシンボリックな名前（開始時間、インクリメント、アラーム用の周期時間など）を使用する場合に便利です。

エラーカウンタ用のスレッシュホールド値は図 10-4 の例のように定義されています。ここではエラーが 10 個発生すると「深刻なエラー」（"serious_error"）として扱い、15 個発生すると「壊滅的なエラー」（"catastrophic_error"）として扱います。

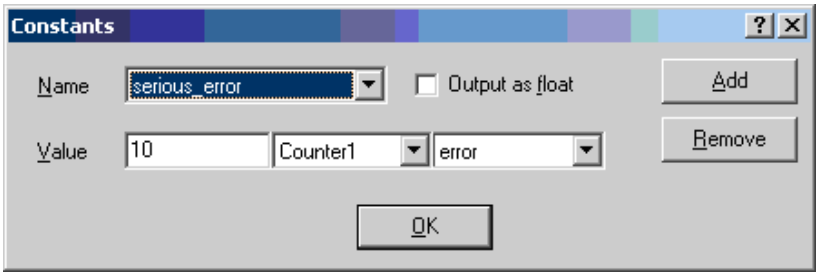


図 10-4 カウンタ定数を宣言する

宣言されたカウンタ定数は、アプリケーションコード用に生成されるヘッダファイル内に生成されます。

10.2 チックドカウンタ

「チックドカウンタ」（'ticked counter'）、つまりチックによってカウントされるカウンタの場合、「チック」を供給するドライバをユーザーが作成する必要があります。RTA-OSEK には、チックソースを容易に OS に接続できるインターフェースが用意されています。

カウンタをインクリメントする場所とタイミングについては制約はありませんが、一般的にはカテゴリ 2 の ISR ハンドラ内で行います。また、タスクから API 関数を呼び出してインクリメントを行うこともできます。

10.2.1 OSEK OS

RTA-OSEK は、設定ファイルに宣言されている各カウンタ用に `Tick_<CounterID>()` という API 関数を生成します（`CounterID` はカウンタ名です）。

移植性

OSEK OS 規格にはカウンタドライバインターフェースは定義されていないため、他の OSEK OS への移植性は考慮されていません。

`Tick_<CounterID>()` という API 関数は、カウンタをインクリメントする必要がある箇所において必ず呼び出される必要があります。たとえばアプリケーションに 2 つのカウンタ（`TimeCounter` および `AngularCounter`）がある場合、RTA-OSEK はコード例 10-1 のように 2 つの API コールを生成します。

```
Tick_TimeCounter();
Tick_AngularCounter();
```

コード例 10-1 RTA-OSEK が生成するカウンタ API コールの例

タイマ割込みと角度割込みを処理するためにユーザーが供給する割込みハンドラでは、これらの API 関数を呼び出す必要があります。

コード例 10-2 に割込みハンドラの例を示します。

```
#include HandleTimerInterrupt.h

ISR(HandleTimerInterrupt) {

   ServiceTimerInterrupt();
   Tick_TimeCounter();

}

#include HandleAngularInterrupt.h

ISR(HandleAngularInterrupt) {

   ServiceAngularInterrupt();
   Tick_AngularCounter();

}
```

コード例 10-2 コード例 10-1 用の割込みハンドラ

同じチックレートで複数のチックドカウンタを使用する場合、ハンドラ内において `Tick_<CounterID>()` を必要な回数だけ呼び出すことができます。

```
# include "MillisecondInterrupt.h"

ISR(MillisecondInterrupt) {

   ServiceTimerInterrupt();
   Tick_TimeCounter1();
   Tick_TimeCounter2();
   ...
   Tick_TimeCounterN();
```

10.2.2 AUTOSAR OS

AUTOSAR OS の場合、OSEK OS とは異なり、カウンタをチックするための `IncrementCounter()` という標準 API が定義されています。この API コールを行う際は、カウンタの名前をパラメータとして渡します。つまり、この API コールは、RTA-OSEK の `Tick_<CounterID>()` よりも処理時間が長く、スタック消費量も多くなります。

デフォルト状態において RTA-OSEK では、OSEK OS バージョンの API コールが使用されることを前提としています。つまりカウンタは ISR 内でチックされ、ハンドラは可能な限り高速に処理されるべきである、ということが考慮されます。

AUTOSAR OS の API コールはコードサイズが大きく低速であるため、この機能が必要な場合にのみ有効にするようにしてください。この API を有効にするには、**Application → Optimizations** を選択し、図 10-5 のようにオプションを設定します。

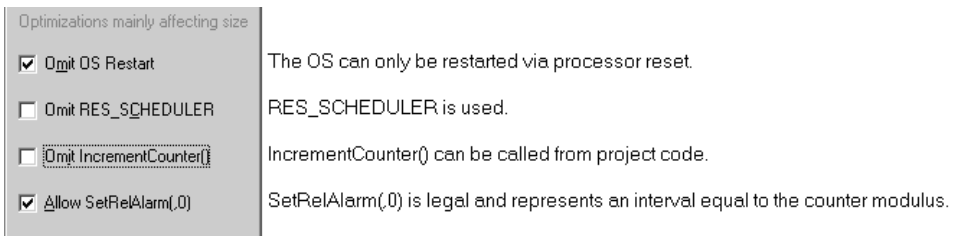


図 10-5 `IncrementCounter()` を有効にする

10.3 アドバンスドカウンタ

アドバンスドカウンタ（'advanced counter'）の場合、ユーザーが OS に対してカウンタドライバとインターフェースを提供する必要があります。チックドカウンタと同様、RTA-OSEK にはアドバンスドカウンタ用ドライバを OS に接続するためのインターフェースが用意されています。

移植性

OSEK OS および AUTOSAR OS において、アドバンスドカウンタ用の標準 API は定義されていません。他の OS 用アプリケーションを RTA-OSEK に移植する場合、アドバンスドカウンタ用ドライバの API コールを変更する必要となる場合があります。

RTA-OSEK は、次回のスケジューリングアクションが発生する時点、つまりアラーム満了時またはスケジューリングテーブルの満了ポイント（'expiry point'）における目標値（'match value'）を内部的に認識しています。

チックドカウンタを使用する場合、1 チック分の時間が経過するたびにユーザーが RTA-OSEK に通知を行い、RTA-OSEK はそのチック数をカウントします。そしてカウンタ値が目標値に達するとアクションを実行し、さらに次の目標値をセットして同じ処理を繰り返します。

一方、アドバンスドカウンタを使用する場合は、RTA-OSEK が目標値（次のアクションを実行すべきタイミング）をユーザーに知らせ、ユーザーは、カウンタがその目標値に達した時点で RTA-OSEK に対して通知を行います。すると RTA-OSEK はアクションを実行し、同じ処理を繰り返します。

チックドカウンタとアドバンスドカウンタを駆動するには、一般的には割り込みを使用します。ただしチックドカウンタの場合は各カウンタチックごとに割り込みを受けますが、アドバンスドカウンタの場合は、アクションを実行するタイミングにおいてのみ割り込みを受けます。つまり、アドバンスドカウンタを使用することにより、割り込みによる影響を少なくすることができます。

10.3.1 アドバンスドカウンタの扱い

目標値に達したことを RTA-OSEK に通知するには `osAdvanceCounter_<CounterID>()` を使用します。**Application → Implementation** を選択すると、ドライバの基本的な構成が表示されます。

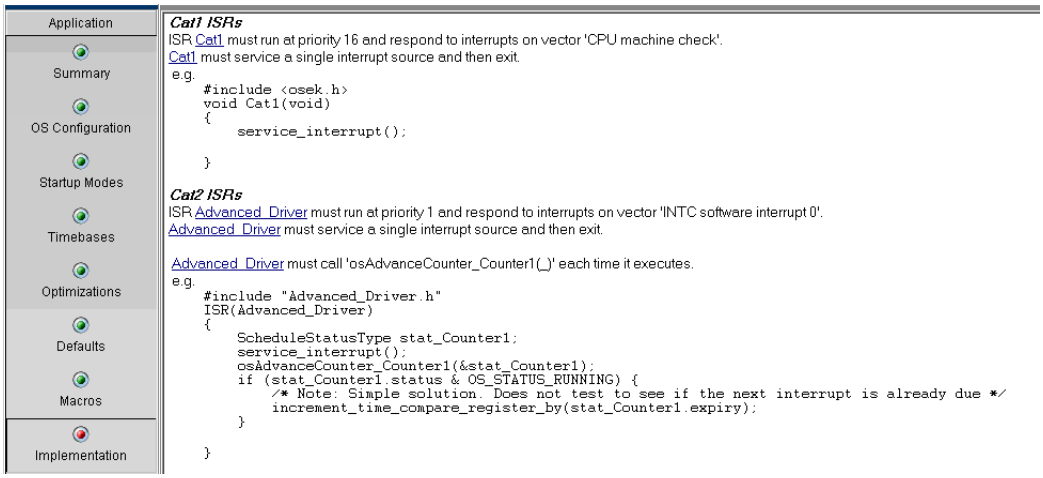


図 10-6 アドバンスドカウンタ用ドライバの実装ノート

`osAdvanceCounter_<CounterID>()` という API コールは構造体を返します。この構造体には、カウンタのステータスと、ハードウェアカウンタの次の目標値（前回の目標値からの相対値で、RTA-OSEK がカウンタについて次のアクションを行うべきタイミング）がセットされます。

重要

`osAdvanceCounter_<CounterID>()` を呼び出すドライバは、ユーザーの責任において作成し、次のアクションが正しいタイミングで実行されるようにしてください。そのためには、アドバンスドカウンタのためのチックソースのチックレートがコンフィギュレーションファイルの内容と一致している必要があります。

アドバンスドカウンタ用のドライバを作成するための情報は、第 14 章に詳しく説明されています。

10.3.2 コールバック関数

RTA-OSEK は、ランタイムにおいてカウンタの制御も行いますが、これは「コールバックインターフェース」を用いて行われます。コールバック関数の詳しい要件は『RTA-OSEK リファレンスガイド』に記載されています。またコールバック関数の作成方法は、第 14 章に詳しく説明されています。

Set_<CounterID>

割込みのステートをセットして、次のアクションを実行すべきタイミングに割込みが発生するようにします。このコールバックには、アクションを実行すべきタイミングにおけるカウンタの絶対値を渡します。このコールバックをカウンタ用に使用する場合、以下の 2 つの用途に使用できます。

1. 起動
   スケジュールテーブルまたはアラームの起動時に、初期割り込みソースをセットします。
2. リセット時
   次のアクションまでの時間を短くします。

2 番めの用途としては、たとえば 100 チック以上経過しないと次の割り込みの発生しないような時点において `SetRelAlarm(alrm, 100)` を発行したい場合などに用います。

State_<CounterID>

このコールバックは、カウンタの次のアクションが発生しているかどうか、また、アクションが発生していな場合は次の目標値までの残りのチック数を返します。

Now_<CounterID>

このコールバックは `GetCouterValue()` によって使用されるもので、外部カウンタの現在の値を返す必要があります。詳しくは 10.4 項を参照してください。

State_<CounterID>

このコールバックは、カウンタの次のアクションが発生しているかどうか、また、アクションが発生していない場合は次の目標値までの残りのチック数を返します。

Cancel_<CounterID>

このコールバックは、カウンタ用の割込みをすべてクリアし、`Set_<CounterID>` が呼び出されるまで割込みが発生しないようにするためのものです。このコールは、カウンタ用のアラームをすべてキャンセルする場合や、カウンタによって駆動されるスケジュールテーブルを停止する際に使用します。詳細は 10.4 項を参照してください。

10.4 カウンタの初期値の設定

チックドカウンタは、起動時において RTA-OSEK により自動的にゼロに初期化されます。またデフォルト状態において RTA-OSEK は、すべてのアドバンスドカウンタがゼロから開始されるものと仮定します。

カウンタにゼロ以外の初期値を適用するには、RTA-OSEK の `InitCounter()` を使用します。

```
InitCounter(Counter1, (Ticktype)42);
```

移植性

`InitCounter()` は RTA-OSEK 固有のものであるため、他の OSEK OS に移植することはできません。

重要

`InitCounter()` はカウント値を直接変更するものであるため、アラームやスケジュールテーブルの実行中にこの関数を呼び出すと、そのタイミング挙動が不正になる可能性があります。これを使用する際は十分に注意してください。

10.5 カウンタの現在の値の取得

RTA-OSEK には、カウンタの現在の値を取得するための `GetCounterValue()` という API 関数が用意されています。

```
TickType Now;
GetCounterValue(Counter1, &Now);
```

移植性

`GetCounterValue()` は RTA-OSEK 固有のものであるため、他の OSEK OS に移植することはできません。

重要

`GetCounterValue()` でアドバンスドカウンタの現在の値を取得する場合、この API からリターンする際にもカウント用ハードウェアは値のインクリメントを継続していることに注意してください。つまり、取得した値は現在の値よりも古い値になります。

10.6 カウンタ属性へのアクセス

RTA-OSEK コンポーネントの API コール `GetAlarmBase()` は、常にカウンタに関するコンフィギュレーション値を返します。属性は以下のような構造体で受け取ります。

```
AlarmBaseType Info;
GetAlarmBase( Alarm2, &Info );

MaxValue   = Info.maxallowedvalue;
BaseTicks  = Info.ticksperbase;
MinCycle   = Info.mincycle;
```

コード例 10-3 GetAlarmBase( ) が返す構造体の内容

カウンタのコンフィギュレーション値へのアクセスには、以下に示すシンボリック定数を用いることができます。OSEK 規格の 3 つの定数に加え、RTA-OSEK では OSTICKDURATION_<CounterID> という 4 番目の定数が定義されています。この定数はカウンタのチック幅をナノ秒単位で表します。

```
OSMAXALLOWEDVALUE_<CounterID>
OSTICKSPERBASE_<CounterID>
OSMINCYCLE_<CounterID>
OSTICKDURATION_<CounterID>
```

コード例 10-4 GetAlarmBase( ) が返す構造体の内容

10.7 まとめ

- OSEK においてカウンタは、何らかのチックソースのチック回数を記録するために使用されます。
- カウンタは任意のチック値をカウントでき、RTA-OSEK では時間以外の単位を使用できます。
- チックするカウンタの場合、RTA-OSEK がカウントを行います。
- アドバンスドカウンタの場合、周辺ハードウェアがカウントを行います。

11 アラーム

ISR を用いてタスクを様々なレートで起動するシステムを構築することは可能ですが、複雑なシステムになると、この方法では効率が悪くなってしまい実用的ではありません。

OSEK のアラームメカニズムは、以下の 2 つの要素で構成されています。

- カウンタ
  前章を参照してください。
- カウンタにアタッチされたアラーム
  アラームは、ある特定のカウンタ値に到達したときに実行するアクション（1 つまたは複数）を定義するものです。システム内の各カウンタに、任意の数のアラームをアタッチすることができます。

カウンタ値がそのカウンタにアタッチされているアラームの値と等しくなると、アラームが「**満了**した（'expired'）」と表現されます。アラームが満了すると RTA-OSEK コンポーネントはそのアラームに関連付けられているアクション（タスクの起動、アラームコールバックルーチンの実行、またはイベントのセット）を実行します。さらに AUTOSAR OS の場合は、4 つ目のアクション、つまりチックドカウンタ（'ticked counter'）のチックを行います。

アラーム満了値は、実際のカウンタ値を基準とする相対値で定義することも、絶対値で定義することもできます。満了値が実際のカウンタ値に対する相対値として定義されるアラームは**相対アラーム**と呼ばれ、絶対値として定義されるアラームは**絶対アラーム**と呼ばれます。

1 回だけ満了するようにアラームを設定することができます。1 回だけ満了するアラームは、**シングルショットアラーム**（'single-shot alarm'）と呼ばれます。

また、アラームは周期的に満了するようにも定義できます。このタイプのアラームは**サイクリックアラーム**（'cyclic alarm'）と呼ばれます。サイクリックアラームの詳細については 11.2 項で詳しく説明します。

11.1 アラームのコンフィギュレーション設定

RTA-OSEK では、アラームは直接宣言するのではなく、以下のようにして作成します。

- スティミュラスを宣言する
- スティミュラスをカウンタにアタッチする

カウンタにアタッチされたスティミュラスは、そのカウンタによるアラームとなります。

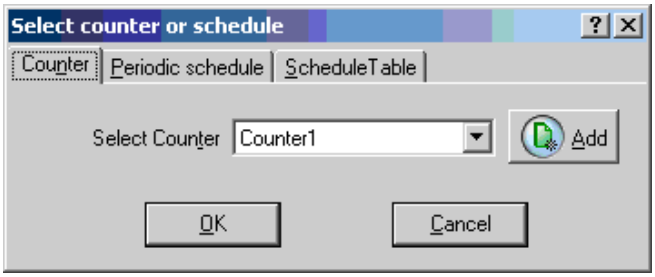


図 11-1 スティミュラスを Counter1 にアタッチする

アラームが満了すると、それに関連付けられたスティミュラスに対するレスポンスが、アラーム満了時のアクションとして実行されます。このようにして、1 つのアラームに最大 4 つのアクションを割り当てることができます。

- タスクの起動
- イベントの発行
- コールバック関数の実行
- カウンタのインクリメント（AUTOSAR の場合のみ）

移植性

OSEK（および AUTOSAR）においては、1 つのアラームで、タスクの起動、イベントのセット、コールバック関数、またはカウンタのインクリメントのいずれかの処理を実行できますが、RTA-OSEK においてはさらに柔軟な設定が可能です。1 つのアラームでタスクを起動し、イベントをセットし、さらにコールバック関数を実行することもできます。

アラーム満了時に複数のイベントをセットしたり、複数のコールバックを行ったり、複数のタスクを起動する必要がある場合は、同じ満了値を持つアラームが複数個必要になります。（AUTOSAR のスケジュールテーブルおよび RTA-OSEK のスケジュールには、複数のアラームオブジェクトを使わずに複数のタスクを起動できる代替メカニズムがあります。詳細は本書で後述します。）

重要

カウンタにアタッチできるのは周期的スティミュラスだけです。ただし、実装方法によっては柔軟に使用でき、ランタイムに任意の値をアラーム周期として設定することができます*。

*. 非サイクリックアラームについてタイミング分析を行う必要がある場合は、アラーム宣言で最短の周期を指定しておく必要があります。

11.1.1 タスクの起動

スティミュラスをカウンタにアタッチする場合、そのレスポンスは、アラームアクションとして実装します。一般的に、レスポンスはタスクとして実装します。図 11-2 の例では、`Stimulus1` が `Counter1` にアタッチされています。レスポンスである `Response1` は `Task1` に実装されているため、`Task1` がアラームアクションとなります。

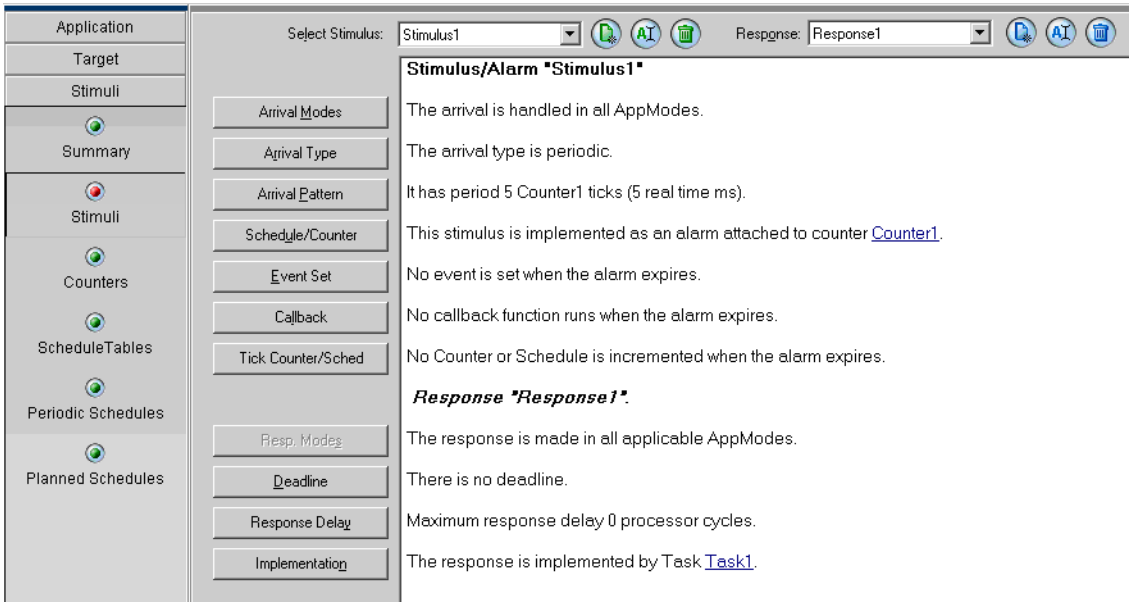


図 11-2 アラームを作成する

RTA-OSEK GUI を使用して、各カウンタに関連付けられたアラームをグラフィックで表示することができます。Counters ワークスペースの **Graphic** タブを選択すると、図 11-3 の例のように、各カウンタにアタッチされているアラームが表示されます。

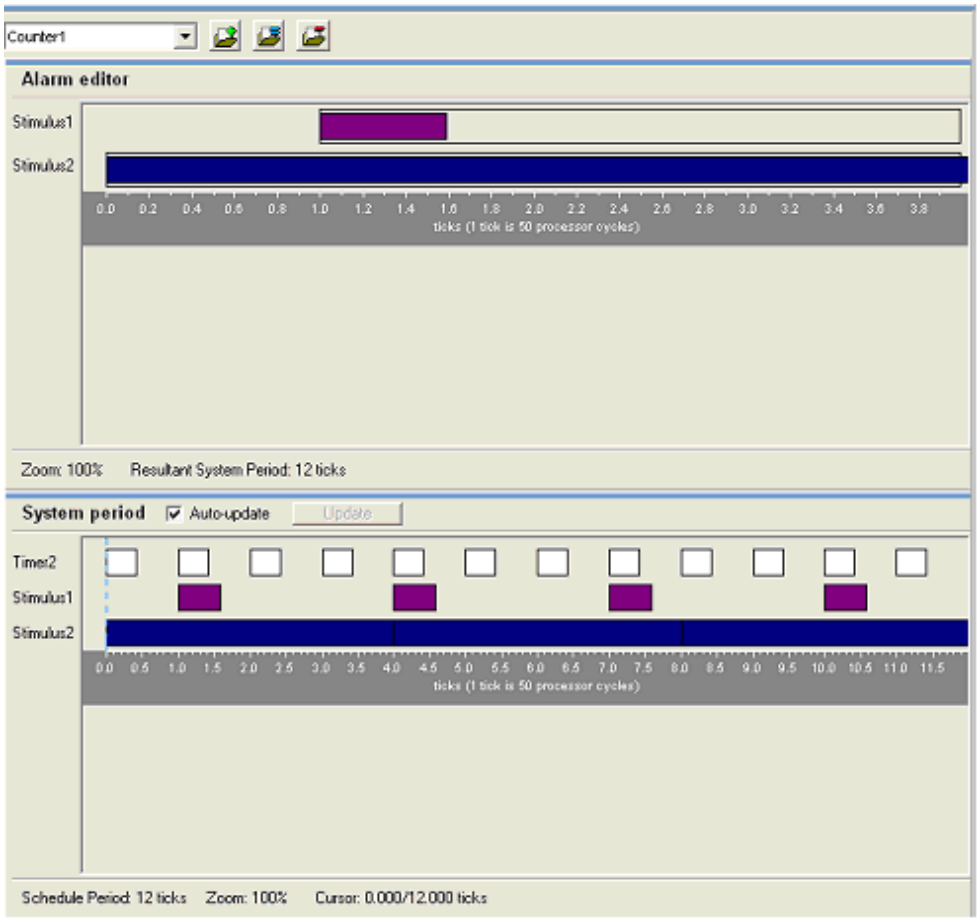


図 11-3 カウンタをグラフィック表示する

グラフィック表示においては、アラームを新しい場所にドラッグして、アラーム設定を変更することができます。

アラームによる複数のタスクの解放

OSEK では、各アラームについてタスクを 1 つだけ起動できます。1 つのスティミュラスに対して複数のレスポンスが定義されていて、そのスティミュラスがアラームを用いて実装されている場合、アラーム満了時には最高優先度のレスポンス実装プロファイルだけが適用されます。

> **重要**
> チェーニングを用いたタスク起動を利用して他のレスポンスを生成する場合、ユーザーの責任においてその処理が確実に行われるようにしてください。RTA-OSEK GUI は、この起動スキームを実装するには直接起動チェーンを使用する必要があることをユーザーに知らせます。

11.1.2 イベントのセット

各アラームは、指定されたタスクについてイベントをセットすることができます。アラームによってイベントがセットされる場合、`SetEvent()` を使用してセットされる場合と同じプロパティが使用されます。図 11-4 は、アラームのイベントアクションの設定方法を示しています。

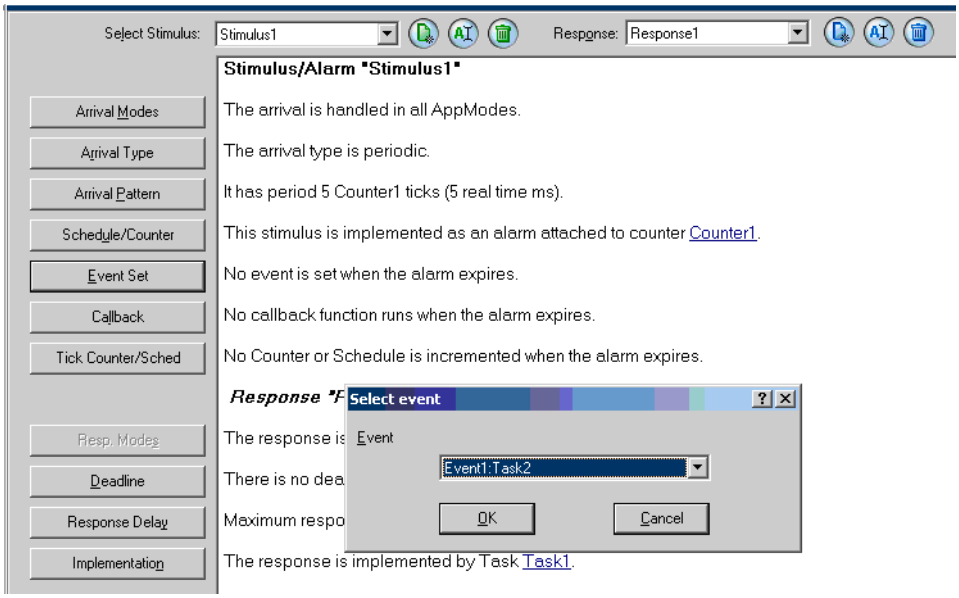


図 11-4 アラーム用のイベントアクションをセットする

11.1.3 アラームコールバック

各アラームには、コールバック関数を割り当てることができます。コールバックは、アラーム満了時に呼び出される一般的な C 関数です。

図 11-5 にアラーム用コールバックルーチンの設定方法を示します。

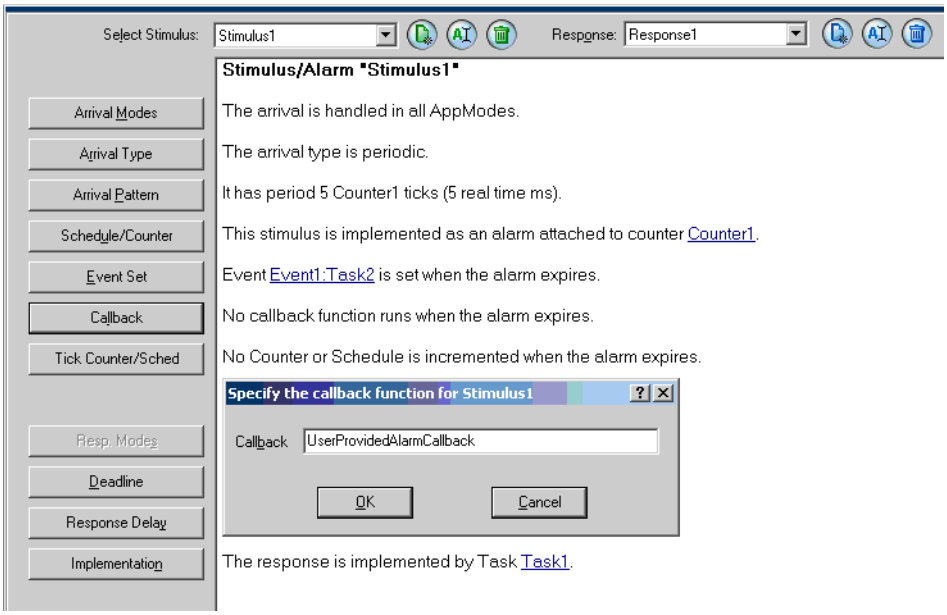


図 11-5 アラーム用のコールバックルーチンを設定する

コード例 11-1 に示すように、各コールバックルーチンは ALARMCALLBACK() マクロを使用して作成します。

```
ALARMCALLBACK(UserProvidedAlarmCallback) {
/* Callback code. */
}
```

コード例 11-1 コールバックルーチンのコード

重要

コールバックルーチンは OS レベルで実行されるため、実行中はカテゴリ 2 割込みがディセーブルになります。そのためコールバックルーチンはできる限り短く作成し、タスクや ISR がブロックされる時間が最小限となるようにしてください。

コールバックルーチン内で使用できる RTA-OSEK コンポーネント API は SuspendAllInterrupts() と ResumeAllInterrupts() だけです。

11.1.4 カウンタのインクリメント

標準の OSEK OS では、タスク起動、イベントセット、コールバック実行をアラームに割り当てることができますが、AUTOSAR OS ではさらに第 4 のアクションとしてカウンタのチックを行うことができます。

アラームからカウンタのチックを行うことにより、1 つの ISR に複数のカウンタをカスケード式に連結することができます。アラームによってチックされるカウンタはアラーム周期を継承するため、たとえば 5 マイクロ秒ごとに満了するアラームが定義されている場合、このアラームを使用して、5 マイクロ秒ごとにチックされる 2 番目のチックドカウンタ（'ticked counter'）を駆動することができます。図 11-6 にその設定方法を示します。

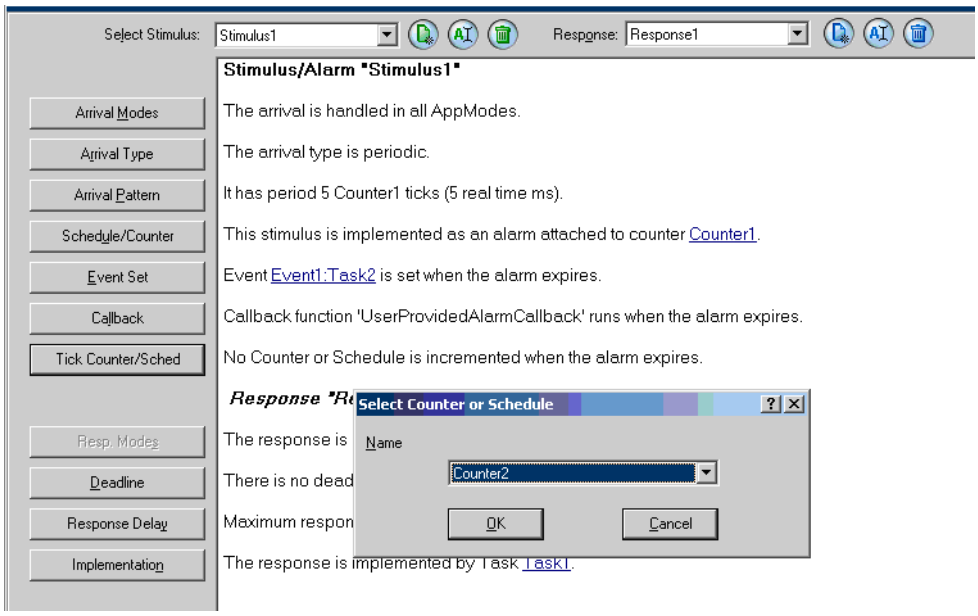


図 11-6 1 つのアラームにチックドカウンタをカスケードする

以下の例では、ISR が 1 ミリ秒ごとに発生し、その ISR が Counter1 をチックします。

```
#include "MillisecondInterrupt.h"
ISR(MillisecondInterrupt){
   CLEAR_PENDING_INTERRUPT();
   Tick_Counter1();
   /* Every 5th call internally performs
      Tick_Counter2() */
}
```

カスケードされたカウンタのチックレートは、カウンタを駆動するアラームの整数倍とする必要があります。複数のカスケードレベルを使用できますが、再帰的なカスケードは行えません。

重要

カスケードされたカウンタのタイミングは、スティミュラスがアタッチされたカウンタのチックの単位で定義れます。そのため、カスケード内の最初のカウンタによって他のすべてのカウンタの基本チックが定義されます。最初のカウンタのチックレートを変更すると、それに応じてアプリケーション全体のタイミング挙動がスケーリングされます。

11.2 アラームのセット

アラームをセットする API 関数は 2 つあります。

- `SetAbsAlarm(AlarmID, Start, Cycle);`

  カウンタ値が次に `Start` という値になった時点で満了するように、アラームをセットします。ただしこの場合、もしもカウンタがちょうどこの値にチックされた時点でこのセットが行われると、カウンタがもう 1 周分カウントされないとアラームは満了しないため、注意が必要です。つまりこの場合、カウンタの値が最大値に到達した後に 0 に戻り、満了時までカウントが行われます。

- `SetRelAlarm(AlarmID, Increment, Cycle);`

  このコールを行った時点から `Increment` チックだけ経過した時点で満了するように、アラームをセットします。つまり、`Increment` は現在のカウンタチック値からのチック数のオフセットです。

2 つの API コールにおいて `Cycle` 値を 0 にすると、アラームはシングルショットアラームになり、一度だけ満了してキャンセルされます。`Cycle` 値が 0 より大きいとサイクリックアラームとなり、初回の満了が発生した後も、指定されたレートにおいて、満了するまでのカウントアップを繰り返します。

RTA-OSEK GUI は、アラームのサイクルレートを定義するための実装ガイドラインを提示します。図 11-7 に実装ガイドラインの一例を紹介します。

図 11-7 実装のガイドライン

11.2.1 絶対アラーム（'absolute alarm'）

絶対アラームは、アラームが満了する時点におけるカウンタの絶対値を指定するものです。コード例 11-2 に絶対シングルショットアラームをセットする方法を示します。

```
/* Expire when counter value reaches 42. */
SetAbsAlarm (Alarm3, 42, 0);
```

コード例 11-2 絶対シングルショットアラームの例

コード例 11-2 の内容は図 11-8 のように図示できます。

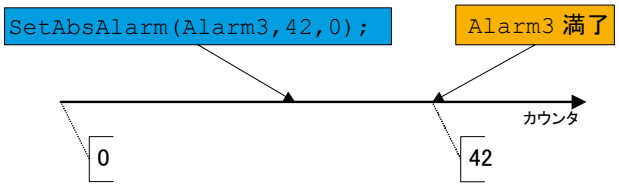


図 11-8 絶対シングルショットアラームの概念図

シングルショットアラームは、たとえば所定時間のタイムアウト監視を行い、タイムアウト発生時に何らかのアクションを行うような処理を作成する場合に有用です。

一方、絶対アラームの `Cycle` にゼロ以外の値を設定すると、指定された `Start` チックにおいてアラームが最初に満了し、以降は `Cycle` チックごとに満了します。

```
/* Expire when counter value reaches 10 and then
every 30 ticks thereafter */
SetAbsAlarm (Alarm1, 10, 30);
```

コード例 11-3 絶対サイクリックアラームの例

コード例 11-3 の内容は図 11-9 のように図示できます。

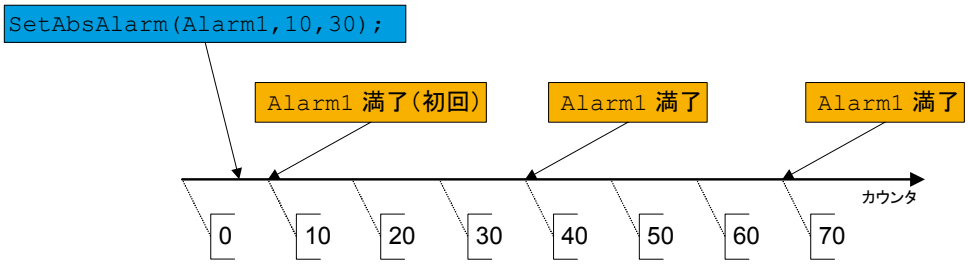


図 11-9 絶対サイクリックアラームの概念図

絶対アラームの場合、初回値を表す 0 という値は、他の値と同じように扱われます。そのため、もしも現在のアラームカウンタ値が 0 であった場合、カウンタ値が MAXALLOWEDVALUE+1 にならないとアラームは満了しません。一方、アラームカウンタ値がすでに MAXALLOWEDVALUE であった場合は、次のチックで満了となります。

重要

非常に小さい相対インクリメント値、または現在のカウンタ値に非常に近い絶対開始値を指定すると、予想外の結果を招く場合があります。つまり、タスクがまだ実行している間にアラームが消失してしまう可能性があります。
起動されるタスクが BCC1 か ECC1 である場合、起動はキューイングされないので、タスク実行が「失われてしまう」可能性があります。アラームが最初に満了するタイミングには、タスクまたは ISR ハンドラが完了するために十分な時間を含める必要があります。

絶対周期アラームをカウンタのラップアラウンドごとに発生させる

アラームが特定の同期ポイントで周期的に発生するようにセットすることは、リアルタイムシステムにおいては非常に重要なことです。しかし OSEK においては、カウンタがラップアラウンドする（最大値までカウントした後に 0 に戻る）たびに絶対アラームが周期的に発生するようにすることはできません。

たとえばここで、1 度の分解能で角度をカウントするカウンタを使用し、上死点において、つまりクランクシャフトの 1 回転ごとにあるタスクを起動する、というケースを考えてみます。このカウンタの最大値は 360 チックとします。

ここでは `SetAbsAlarm(Alarm1, 0, 360)` を実行すべきところですが、OSEK の規格ではこれは禁止されています。これは、引数 `Cycle` の値は `OSMAXALLOWEDVALUE` より大きくすることはできないためです。`OSMAXALLOWEDVALUE` の値は常に最大値 -1（この例では 359）となります。

そのため、この例のようなカウント機能が必要な場合は、絶対シングルショットアラームを使用し、それが満了するたびにリセットする必要があります。

たとえば Task1 が Alarm1 にアタッチされている場合、コード例 11-4 に示されるように、Task1 のボディ内でタスク起動時にアラームをリセットします。

```
TASK(Task1) {

   /* Single-shot alarm reset. */
   SetAbsAlarm(Alarm1, 10, 0);

   /* User code. */
   TerminateTask();
}
```

コード例 11-4 タスク起動時に絶対シングルショットアラームをリセットする

11.2.2 相対アラーム（'relative alarm'）

コード例 11-5 は、最初に 10 チック後にで満了し、その後は 20 チックごとに満了する相対アラームを示しています。

```
/* Expire after 10 ticks, then every 20ticks. */
SetRelAlarm (Alarm1, 10, 20);
```

コード例 11-5 相対サイクリックアラーム（20 チック周期）

図 11-10 は、このアラームを図解したものです。

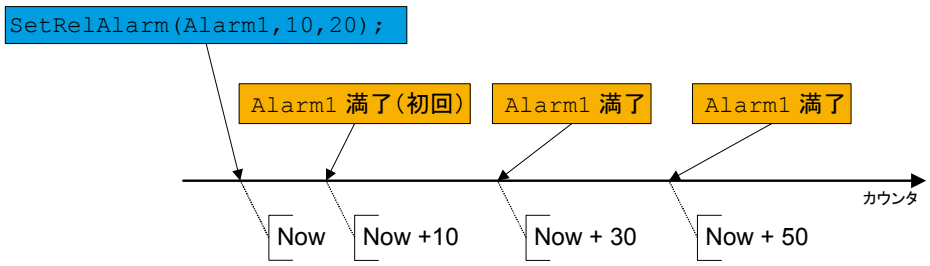


図 11-10 コード例 11-5 のアラームのタイミング

ゼロの意味

OSEK では、SetRelAlarm() の引数 Increment にゼロを設定した場合の意味は定義されていませんが、RTA-OSEK の場合、デフォルト状態において、ゼロは「現時点（'now'）+ カウンタの最大値」、つまり now + MAXALLOWEDVALUE + 1 と解釈されるため、相対アラームを、カウンタの最大値をカウントした時点で満了するようにセットアップすることが可能です。これは、ハードウェアカウンタを基準として正確なインターバルタイミングを生成する必要がある場合に非常に有用です。

コード例 11-6 に、相対シングルショットアラームをセットする方法を示します。

```
/* Expire after Max Value ticks. */
SetRelAlarm (Alarm1, 0, 0);
```

コード例 11-6 相対シングルショットアラームをセットする

図 11-11 は、このアラームを図解したものです。

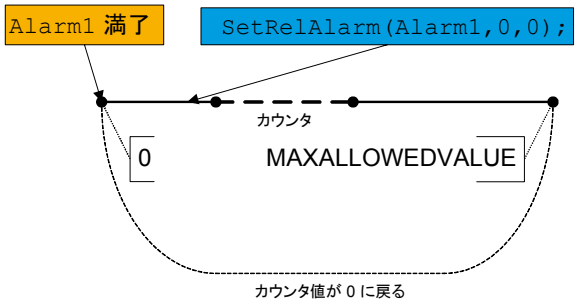


図 11-11 コード例 11-6 のアラームのタイミング

AUTOSAR OS の場合は、`SetRelAlarm()` にゼロを渡すことは禁止されており、`Increment` にゼロをセットすると E_OS_VALUE エラーが発生します。

つまり AUTOSAR OS では上記のような便利な機能が利用できないため、RTA-OSEK では、ゼロの意味を任意に選択しておくことができます。これは **Applications → Optimizations** で、図 11-12 のように設定します。

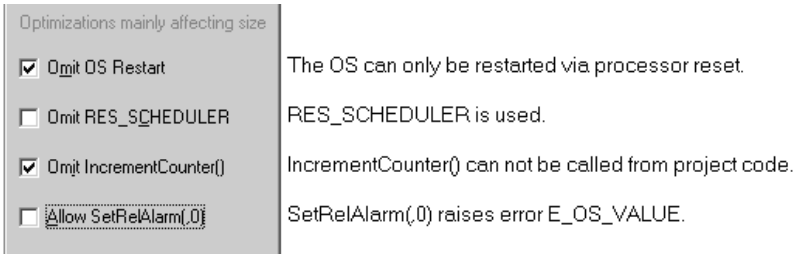


図 11-12 SetRelAlarm(AlarmID, 0, x) の引数の意味を選択する

11.2.3 アラームの自動起動

アラームの起動は、メインプログラム内で `SetRelAlarm()` または `SetAbsAlarm()` を呼び出すことによって行えますが、サイクリックアラームを使用する最も簡単な方法は、自動起動機能を使用することです。自動起動されるアラームは、`StartOS()` 内で起動されます。

自動起動されるアラームは、アプリケーションモードごとにセットできます。アラームを作成する際、開始時間は 0 チックにセットされます。開始時間とは、カウンタ起動時から最初の満了までの時間です。開始時間は、ランタイムにおいて自動起動されるアラームについてのみ使用されます。

アラームが自動起動されるようにするには、モードペインで図 11-13 のように設定します。

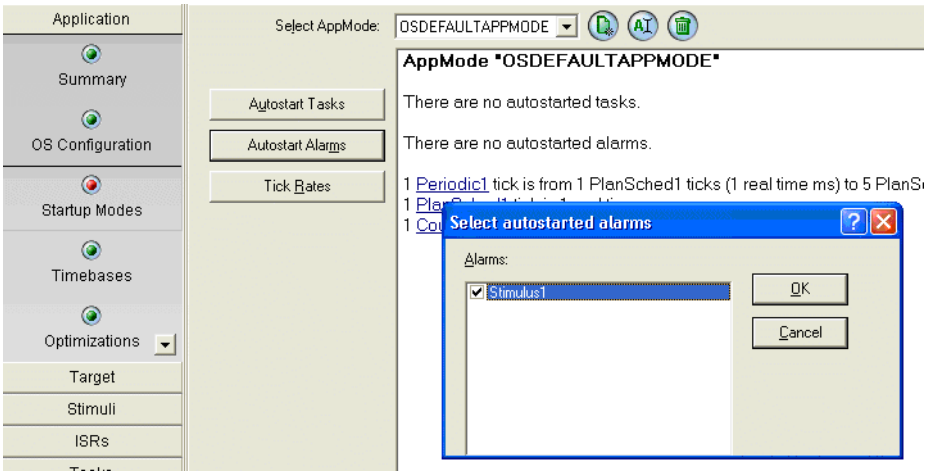


図 11-13 アラームを自動起動する

自動起動されるアラームは、RTA-OSEK 内部において絶対値を用いて設定されます。つまり、アラームに関連付けられたカウンタの値が `StartOS()` 内で 0 チックに初期設定されます。このため 0 チックというデフォルトの開始時間を使用する場合は注意が必要です。アラームが起動された時点でカウンタ値が 0 チックになっているので、アラームの初回の満了は、そのカウンタがラップアラウンドするまで発生しません。

アラームの自動起動を用いて、複数のアラームを同期して起動させることができます。`StartOS()` 内において、RTA-OSEK コンポーネントにより、1 つのカウンタに関連付けられたすべての自動起動アラームが同期して起動されます。

重要

ユーザーによってアラームの同期化が定義されていて、かつユーザーがアプリケーションについてタイミング分析を行う場合は、開始時間をアラーム周期より小さい値に設定する必要があります。

11.3 アラームのキャンセル

`CancelAlarm()` を使用して、アラームをキャンセルすることができます。

アラームのキャンセルは、たとえば所定のタスクが起動されるのを停止したいような場合に必要となります。キャンセルしたアラームの再起動は、`SetAbsAlarm()` または `SetRelAlarm()` を使用して行います。

11.4 次回のアラーム満了タイミングの取得

RTA-OSEK では、アラームが満了するまでの残り時間を取得することができます。この機能を利用することにより、たとえば、設定された絶対値にすでに到達している時点で絶対アラームをセットしてしまうようなことを防ぐことができます。

`GetAlarm()` を使用することにより、次にアラームが満了するまでのチック数を取得できます。

11.5 アラームを利用する同期化

アラームを同期化する方法として最も安全でしかも簡単なのは、複数の絶対アラームを同じカウンタにリンクさせてセットする方法です。複数の絶対アラームを使用しても、起動時間への影響はありません。このようにすることにより、割込みによる潜在的な問題を回避することができます。

相対アラームの同期化は、割込みやプリエンプションによる中間遅延が発生し、起動時に異なるアラームオフセットがセットされてしまう可能性があるため、複雑になります。

アラームの同期化については、図 11-14 に示すように、RTA-OSEK GUI でカウンタ設定時に選択できます。

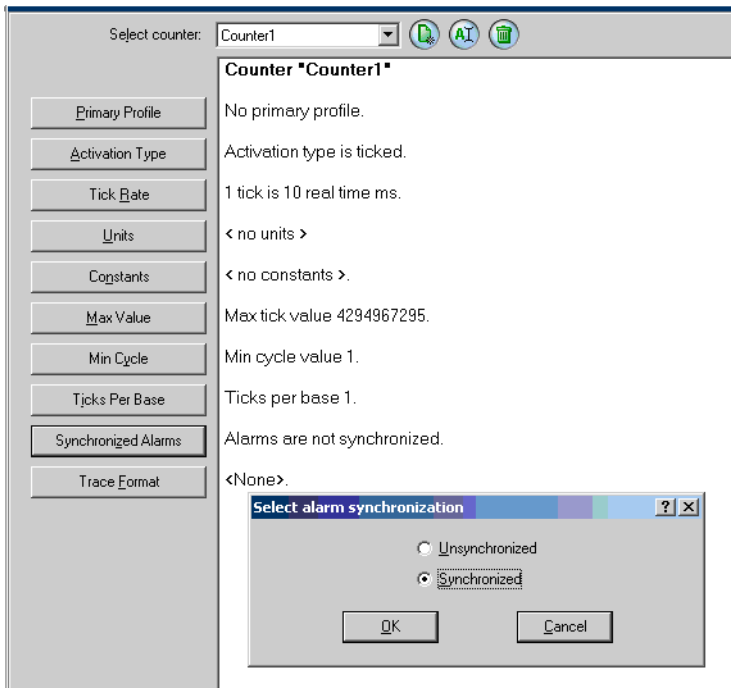


図 11-14 アラームの同期化

ランタイムにおいて、1 つのカウンタに関連付けられた複数のアラームが互いに同期することが保証されるには、それらのアラームが確実に自動起動されるようにする必要があります。また、アラームがキャンセルされた時には `SetAbsAlarm()` でそのアラームをリセットすることも必要です。

重要

RTA-OSEK GUI でアラームの同期化が有効に設定されていても、それらのアラームが実際に同期化されることが保証されるわけではなく、RTA-OSEK Planner にユーザーが同期化を保証する、ということを知らせるだけです。

タイミング分析用のシステムを構築する場合は同期化が保証されるようにすることが望ましいので、そのような場合は、AUTOSAR のスケジュールテーブルまたは RTA-OSEK のスケジュールを使用するようにしてください。これらのメカニズムを利用すれば、タスク間の同期化が保証され、イベントベースの厳密なリアルタイムシステムを柔軟に設計することができます。スケジュールの詳細については本書で後述します。

11.6 非周期的アラーム

RTA-OSEK GUI は、一連のアラームを作成する際の実装方法をユーザーに提示します。この情報を見れば、所定のタイミング挙動を実現するために何をすればよいかがわかります。

非周期的な挙動を実現するには、起動されたタスクが次の満了値をセットする、といったシングルショットアラームを使用してください。

コード例 11-7 に、アラームが 10 チック後に満了し、さらに 12 チック後に満了する、という例を示します。このアラームは `Task1` というタスクを起動します。最初のアラームは RTA-OSEK コンポーネントにより自動起動され、`Task1` 内でアラームを次の満了に向けてリセットします。

```
TASK(Task1) {

   SetRelAlarm(Alarm1, 12, 0);
   /* Rest of task. */

}
```

コード例 11-7 非周期アラームの例

この方法を使用する場合は、スティミュラス - レスポンスモデルで定義されている時間が、後続のアラーム満了間の最短時間の値となるようにする必要があります。

11.7 まとめ

- アラームは、それに関連付けられたカウンタによってセットされます。
- 1 つのカウンタに複数のアラームをセットすることができます。
- 各アラームで以下のいずれかのアクションを実行できます。
  - タスクの起動
  - イベントのセット
  - コールバックの実行
  - チックドカウンタのチック（インクリメント）
- アラームは、絶対または相対カウンタ値（現時点からの相対値）において満了するようにセットできます。
- アラームは自動起動することができます。
- スケジューラビリティ分析の目的で、複数のアラームを同期化することができます。

12 スケジュールテーブル

ここまで OSEK で定義されているアラームとカウンタについて説明しましたが、これらは周期的なタスク起動が必要なシステムを構成する際に使用されます。

しかしアラームは、各スティミュラス間の時間が異なるシステムにおいてはあまり適していません。そのようなシステムを OSEK のアラームで作成することは可能ですが、ランタイムにアプリケーションのタイミングプロパティが不正に変更されてしまうことを防ぐ手段は、コードレビューを行うくらいしかありません。さらに、複数のタスクの起動を同時に行いたい場合、1 つのアラームで複数のタスクを起動する代わりに、複数のアラームを作成しなくてはなりません。

AUTOSAR OS は、このようなアラームの制限事項を**スケジュールテーブル**によって解決しています。

移植性

スケジュールテーブルは AUTOSAR OS の機能であるため、OSEK OS に移植することはできません。

スケジュールテーブルは、1 つの OSEK カウンタに関連付けられ、複数のディレイ（'delay'）で定義された一連の**満了ポイント**（**'expiry points'**）で構成されます。ディレイは、対応するアラームのチック数で定義されます。つまりスケジュールテーブルは、連続的な満了ポイント間の長さをチック数で定義したものです。

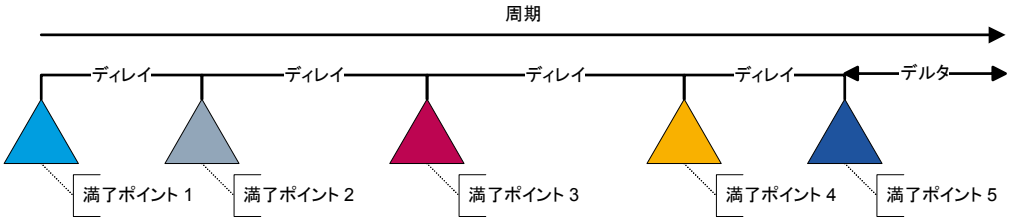


図 12-1 スケジュールテーブルの概念図

各満了ポイント間のディレイの合計時間と**周期**（**'period'**）の差は**デルタ**（**'delta'**）と呼ばれます。

満了ポイントは、RTA-OSEK が何らかのアクションを実行するタイミングを示すものである、という点はアラームと似ていますが、以下の表のように、満了ポイントとアラームでは実行できるアクションが異なります。

アクション	Alarm	満了ポイント
ActivateTask	可（1 タスクのみ）	可（複数タスク可）
SetEvent	可（1 イベントのみ）	可（複数イベント可）
Callback	可	不可
IncrementCpi	可	不可

移植性

RTA-OSEK では 1 つの満了ポイントで複数のコールバックを実行して複数のカウンタをインクリメントすることができます。これは AUTOSAR OS の仕様には含まれないため、移植性は考慮されていません。

12.1 スケジュールテーブルのコンフィギュレーション設定

各スケジュールテーブルは、1 つの OSEK カウンタによって駆動されます。つまり、カウンタがスケジュールテーブルに対してチックソースを供給します。同じカウンタを使用して複数のスケジュールテーブルを駆動することができますが、ランタイムにおいては、実行ステートのある時間ポイントにおいては、1 つのスケジュールテーブルしか使用できません。

1 つのカウンタは、複数のスケジュールテーブルや任意の数のアラームで共有することができます。

各スケジュールテーブルには 1 つの周期が定義されている必要があり、周期の値がゼロより大きい場合、スケジュールテーブルはその周期で反復実行されます。周期がゼロの場合は「シングルショット」とみなされ、最後の満了ポイントが処理された後、スケジューリングは終了します。シングルショットスケジュールテーブルは、たとえば閉ループの制御システムなどにおいて一連のアクションを順に起動したいような場合に、有用です。

図 12-2 は、`Table` という名前のスケジュールテーブルのコンフィギュレーション設定例です。

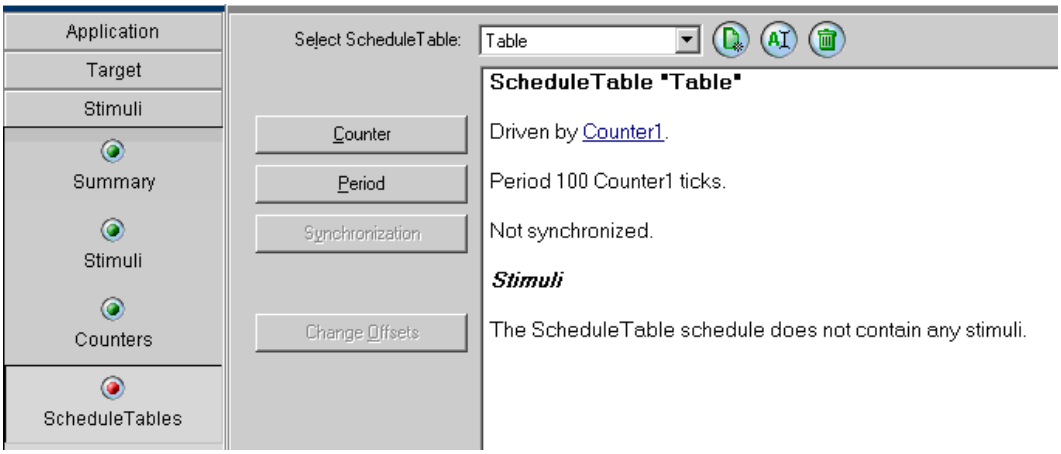


図 12-2 スケジュールテーブルのコンフィギュレーション

12.2 満了ポイント（'expiry point'）のコンフィギュレーション設定

RTA-OSEK における「満了ポイント」('expity point') は、アラームと同様、直接的に宣言するものではありません。最初にスティミュラスを宣言し、それにレスポンス（複数可）を関連付ける必要があります。

スケジュールテーブルを用いてスティミュラスを実装する場合、そのスティミュラスは周期的なアライバルパターンを持っている必要があります。このアライバルレートは、スティミュラスをアタッチするスケジュールテーブルの周期で決まります。

図 12-3 にスティミュラスをスケジュールテーブルにアタッチする方法を示します。

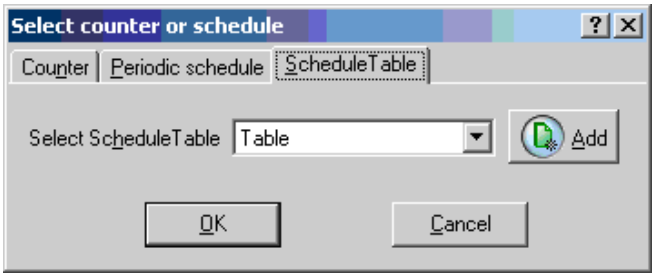


図 12-3 スティミュラスをスケジュールテーブルにアタッチする

スティミュラスに対するレスポンスは、満了ポイントにおいて実行される「アクション」となります。また、スティミュラスでタスクの起動、イベントのセット、コールバックの実行、カウンタのチックを行うこともできます。これらのアクションはすべて満了ポイントにアタッチされます。

重要

1 つの満了ポイントアクションを、同じスケジュールテーブル内で複数回実行させるには、同じレスポンスを持つ複数のスティミュラスを定義する必要があります。この結果、一般的には、スケジュールテーブルを含むシステムについてスケジューラビリティ分析を行うことはできなくなります。

12.2.1 オフセットの設定

スケジュールテーブルにアタッチしたスティミュラスは、それぞれ 1 回ずつ発生します。デフォルト状態においてスティミュラスは、スケジュールテーブルの論理的な開始時点からオフセットゼロの時点で発生します。このオフセットを用いて、スケジュールテーブル内のスティミュラスの発生位置を設定できます。

「オフセット」は、スケジュールテーブル内のどのカウント時点でスティミュラスが発生すべきかを表すものであるため、これによってスケジュールテーブル内でいつアクションが実行されるかが決まります。オフセットは、スケジュールテーブル内の各スティミュラスごとに 0 から Perid-1 の範囲で設定できます。

図 12-4 にスティミュラスをスケジュールテーブルにアタッチする方法を示します。

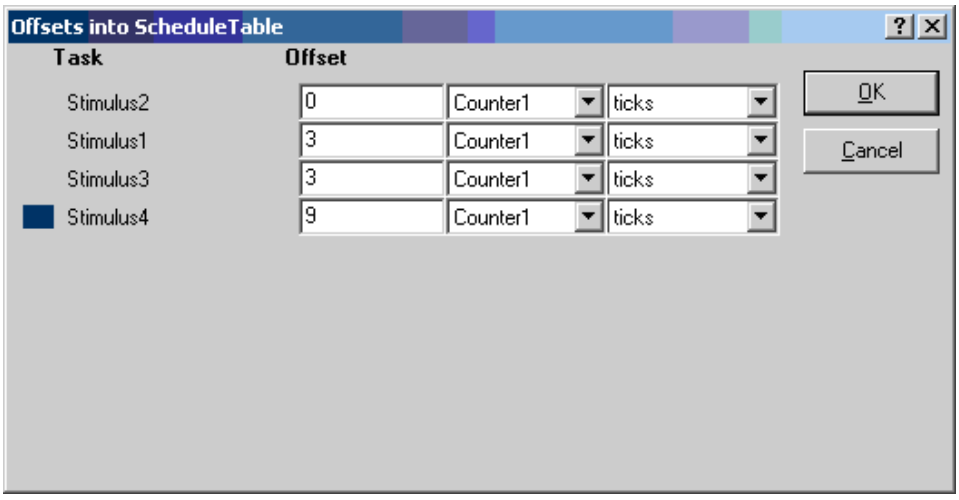


図 12-4 スティミュラスのオフセットを指定する

テーブル内の各満了ポイントは、同じオフセットで発生するスティミュラスごとに、「スティミュラスセット」として定義します。図 12-4 の例では 3 つの満了ポイントが定義されています。

1. 満了ポイント 1：スティミュラス 2
2. 満了ポイント 2：スティミュラス 1 および 3
3. 満了ポイント 3：スティミュラス 4

これをわかりやすくまとめると、図 12-5 のようになります。

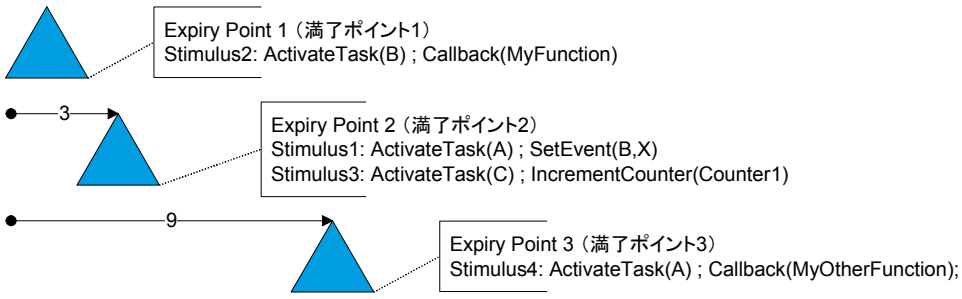


図 12-5 オフセットの使用

満了ポイントは、概念的なゼロの時点に配置することができますが、オフセットはすべてゼロ以上に設定する必要があります。

重要

同じアクションを実行するレスポンスが 2 つのスティミュラスに対して割り当てられていると、満了ポイントにおいて、同じアクションが 2 回発生します。これは、タスクを複数回起動したいような場合に有用です。この際、イベントは複数回セットされますが、OSEK のイベントメカニズムにおいては 1 回分の `SetEvent()` のみが登録されます。同じ満了ポイントにおいて 1 つのコールバックが複数回実行される場合は注意が必要です。これは、コールバックは OS レベルで実行され、システム全体をブロックしてしまうためです。また、1 つの満了ポイントにおいて 1 つのカウンタを複数回インクリメントするような処理も避けてください。

12.3 スケジュールテーブルの起動

スケジュールテーブルの起動には `StartScheduleTable(ScheduleTableID, Offset)` という API を使用します。

`StartScheduleTable()` の引数 `Offset` には、RTA-OSEK が最初に満了ポイントを処理すべきタイミングを、現時点（'now'）からの相対的なチック数で指定します。ゼロを指定することも可能です。

```
/* Start Table1 20 ticks from now */
StartScheduleTable(Table1, 20);
```

コード例 12-1 コールバックルーチンを作成する

移植性

RTA-OSEK の場合、`Offset` の値をゼロにすると次のチックが満了ポイントとなります。つまり、`Offset` の値を N にすると、N+1 番目のチックが満了ポイントとなりますが。AUTOSAR OS V1.0 ではこれについて明確な定義されていないので、OS ごとに挙動が異なる場合があります。

重要

スケジュールテーブル内のすべてのオフセットをゼロ以外の値に設定すると、最初のオフセットは `StartScheduleTable()` によってオーバーライドされ、無効になります。

アラームや RTA-OSEK のスケジュールとは異なり、OS の起動時に自動的にスケジュールテーブルを起動することはできません。自動起動のような機能が必要な場合は、`StartupHook()` 内でスケジュールテーブルを起動してください。

重要

`StartScheduleTable()` に渡す `Offset` の値が小さすぎると、コールから戻る前にオフセット時間が経過してしまう場合があるため、十分に長い値をセットするようにしてください。チックドカウンタによって駆動されるスケジュールテーブルの場合、デフォルト状態においてカウンタの値は起動時にゼロになるため、`StartScheduleTable(Table, N)` というコールを実行すると、カウンタの N チック後に次の満了ポイントの処理が実行されます。

12.4 スケジュールの終了

周期がゼロのスケジュールテーブル、つまりシングルショットスケジュールテーブルは、RTA-OSEK か最後の満了ポイントを処理した後、直ちに停止します。

周期的スケジュールテーブルの場合は、テーブルが切り替えられる（12.5 項参照）か、またはユーザーが `StopScheduleTable(ScheduleID)` を呼び出すまで実行状態を保ちます。

12.4.1 スケジュールテーブルの再起動

停止したスケジュールテーブルは、`StartScheduleTable()` を呼び出すことによって再起動できます。この際スケジュールテーブルは、常に最初の満了ポイントから再起動されます。

12.5 スケジュールテーブルの切り替え

ランタイムに NextSchedulTable() を呼び出すと、現在実行中のスケジュールテーブルが別のテーブルに切り替わります。スケジュールテーブル間の切り替えは、常に「テーブルの最終ポイント」において行われます。

シングルショットスケジュールテーブルの場合、最後の満了ポイントが処理された直後が「テーブルの最終ポイント」となります。

周期的スケジュールテーブルの場合は、周期によって最終ポイントが決定されます。

以下にコード例を示します。

```
/* Start New after Current has fiished */
NextScheduleTable(Current, Next);
```

Current の最後の満了ポイントから Next の最初の満了ポイントまでの時間（ディレイ）は以下のようになります。

Delay = Delta(Current) + OffsetToFirstExpiryPoint(Next)

スケジュールテーブル Current がシングルショットで、かつ Next のオフセットゼロの時点に満了ポイントが定義されている場合、Next の最初の満了ポイントは、カウンタの次のチックにおいて処理されます。

Current の実行中に NextScheduleTable() を複数回実行すると、最後に実行されたコールで指定された Next が起動されます。

12.6 スケジュールテーブルのステータス

スケジュールテーブルのステータスは、GetScheduleTableStatus() で取得できます。ステータスは出力引数として戻ります。

```
ScheduleTableStatusType State;


GetScheduleTableStatus(Table, &State);
```

取得されるステータスの値は、以下のいずれかです。

- SCHEDULETABLE_NOT_STARTED

  指定のテーブルが起動しておらず、またそのテーブルは最後に発行された NextScheduleTable() の引数 Next で指定されたものでありません。
- SCHEDULE_TABLE_ASYNCHRONOUS

  テーブルが起動しています。

12.7 まとめ

- スケジュールテーブルを使用すると、計画された一連のアクションを所定の時間に実行することができます。
- スケジュールテーブルには 1 つの OSEK カウンタを関連付け、周期を定義し、1 つまたは複数の満了ポイントを定義します。
- RTA-OSEK における満了ポイントは、スケジュールテーブルによって発行されるスティミュラスのオフセットを指定することにより、暗黙的に生成されます。
- StarScheduleTable() によって起動されたスケジュールテーブルは、常に最初の満了ポイントから処理を開始します。
- スケジュールテーブルを切り替えることができますが、実際の切り替えはテーブルの最終ポイントにおいて行われます。

13 スケジュール

前述のように、OSEK 規格にはアラームとカウンタが定義されていて、これらを使用して繰り返しタスク起動を行うシステムを構築することができます。また AUTOSAR OS にはスケジュールテーブルがあり、アクションのセットを 1 つの複合オブジェクトとして制御することが可能です。

それらに加え、RTA-OSEK には「スケジュール」というメカニズムがあります。これを使用すると、カウンタ／アラームや AUTOSAR スケジュールテーブルメカニズムより柔軟なシステムを実現できます。

移植性

スケジュールは、複雑なシステムを構築して管理するために設けられている、RTA-OSEK の独自の機能です。OSEK OS 規格には含まれていません。

13.1 スケジュールの使用

スケジュールを使用すると、複数のアラームを定義しなくても、あるタイミングで同時に複数のタスクが起動されるシステムを構築することができます。また、スケジュール全体の同期を維持したまま、タスク起動を行う間隔を相対的に変更することができます。

13.1.1 スケジュールのタイプ

スケジュールは、その使用方法によって以下の 2 つのタイプに分けられますます。

- 周期的スケジュール（'periodic schedule'）
  周期的スティミュラスを実装できます。
- 計画的スケジュール（'planned schedule'）
  非周期的スティミュラスを実装できます。

計画的スケジュールは、周期的スケジュールよりもはるかに柔軟性があります。ここではユーザーがランタイムにスケジュールをセクション単位で組み込んだり除外したり、スケジュール全体をシングルショットとして指定することができます。これにより、たとえば複数のタスクシーケンスを段階的にリリースする、というようなことが可能になります。

13.1.2 アライバルポイント

周期的スケジュールも計画的スケジュールも、一連の**アライバルポイント**（'arrivalpoint'）といくつかの状態変数とで定義されます。スケジュールがアライバルポイントに達すると、スケジュールが「**到達した**」と表現されます。

周期的スケジュールのアライバルポイントは暗黙的なものです。それらは RTA-OSEK により自動生成され、RTA-OSEK コンポーネントにより内部的に使用されます。一方、計画的スケジュールについては、ユーザーがアライバルポイントを設定する必要があります。

アライバルポイントはアラームに似ています。アラームはシステム内でスティミュラスを実現するために用いられますが、アライバルポイントは以下のような点でアラームとは異なっています。

- 1 つのアライバルポイントで複数のスティミュラスを実現できます。
- 1 つのスティミュラスに複数のレスポンスが定義されている場合、RTA-OSEK コンポーネントは複数のタスクの起動を管理してそれらの各レスポンスを生成します。つまり、スケジュールを使用すれば、カウンタ／アラームメカニズムを使用する際に必要なタスク起動のチェーニングを行う必要がありません。
- アライバルポイントでイベントをセットしたりコールバックを行ったりすることはできません。

アライバルポイントには、以下のプロパティがあります。

- 到達時にトリガするスティミュラス群
- 次のアライバルポイント発生までのディレイ
- 次のアライバルポイント
- 一連の分析属性

1 つのアライバルポイントに関連付けられている各スティミュラスについて、到達時に、そのスティミュラスによりトリガされるレスポンスが発行されます。スティミュラスに対するレスポンスはタスクで生成されなければならないため、ランタイムにおいて RTA-OSEK コンポーネントは、アライバルポイントのすべてのスティミュラスに関連付けられているすべてのタスクを同時に起動します。

スティミュラスをアライバルポイントにアタッチすると、ランタイムにはそのアライバルポイントが実際のスティミュラスとして機能し、到達時に同時に起動されるタスクが、そのスティミュラスに関連付けられるレスポンス（'response'）となります。

スケジュールには、アライバルポイントだけでなく、以下の 4 つの状態変数も記録されます。

- `State`
  スケジュールが実行中か停止中かを記録します。
- `Next`
  どのアライバルポイントが次に処理されるかを記録します。
- `Now`
  スケジュールのチック（'tick'：刻み）の現在値を保持します。
- `Match`
  次のアライバルポイントが処理される予定のチック値を保持します。

図 13-1 は、スケジュールオブジェクトを図解したものです。

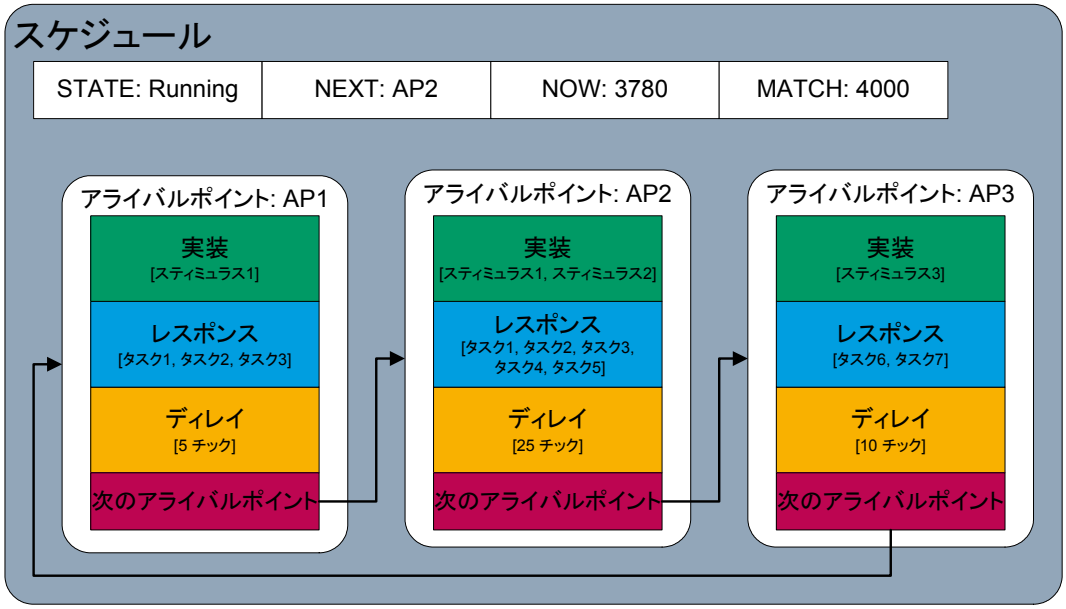


図 13-1 スケジュールの構造

13.1.3 チックド（'ticked'）スケジュールとアドバンスド（'advanced'）スケジュール

スケジュールは、その処理方法により以下の 2 種類に分けられます。

- チックドスケジュール（'ticked schedule'）
  スケジュールをチック（'tick'）することは、カウンタをチックすることに似ています。スケジュールは、経過チック数の内部カウントを保持し、カウンタ値が `match` の値になるとアライバルポイントの処理を行います。
- アドバンスドスケジュール（'advanced schedule'）
  アドバンスドスケジュールでは、ターゲットハードウェア上のカウンタ比較ハードウェアを利用します。このハードウェアは、次のアライバルポイントを直ちに処理する必要があることをスケジュールに知らせるための割込みを生成します。これにより、処理しなければならないチック割込みの数を最小限に抑えることができます。つまり、アドバンスドスケジュールを使用すれば、割込みはアライバルポイントの処理が必要なときにのみ発生します。

チックドスケジュールまたはアドバンスドスケジュールを使用する場合、ユーザーの責任において**ドライバ**を用意してください。チックドスケジュールの場合、ドライバは周期的割込みのみを行います。アドバンスドスケジュールを使用する場合は、一連のコールバックルーチンを使用して、RTA-OSEK がカウンタ比較ハードウェアを管理できるようにする必要があります。コールバック関数については、13.5.1 項で詳しく説明します。

13.2 周期的スケジュールの設定

周期的スケジュールの宣言は、RTA-OSEK GUI を使用して行います。各スケジュールには固有の名前と所定の**チックレート**（'tick rate'）が定義されなければなりません。図 13-2 は、周期的スケジュールがどのように宣言されているかを示しています。

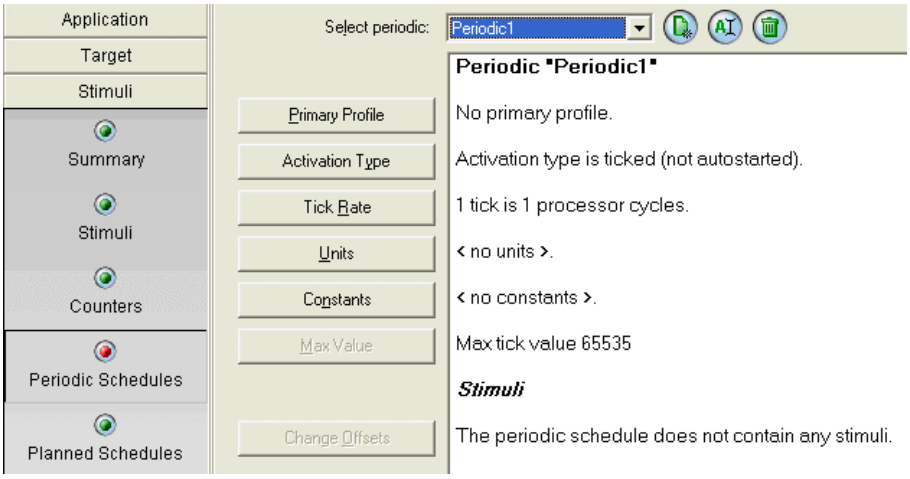


図 13-2 周期的スケジュールを設定する

スケジュールはプライマリプロファイルにより処理されます。ユーザーのアプリケーションでは、このプライマリプロファイルは通常、ISR です。デフォルトでは、周期的スケジュールはすべてチックドスケジュールとして設定されます。これをアドバンスドスケジュールに変更したい場合は、13.5 項を参照してください。

重要

スケジュールが定義されたとおりに動作するためには、プライマリプロファイルのアライバルパターンが、スケジュールの 1 チックの解像度で実現できる、という条件が必要です。定義されたスケジュールの解像度をより高い精度のアライバルパターンが指定されていると、アライバルレートは、指定されたチックレートの解像度で実現できる次のアライバルパターンに切り上げられます。

13.2.1 アライバルポイントの作成

周期的スケジュールは、周期的スティミュラスを周期的スケジュールにアタッチすることによって組み立てられます。図 13-3 にその例を示します。

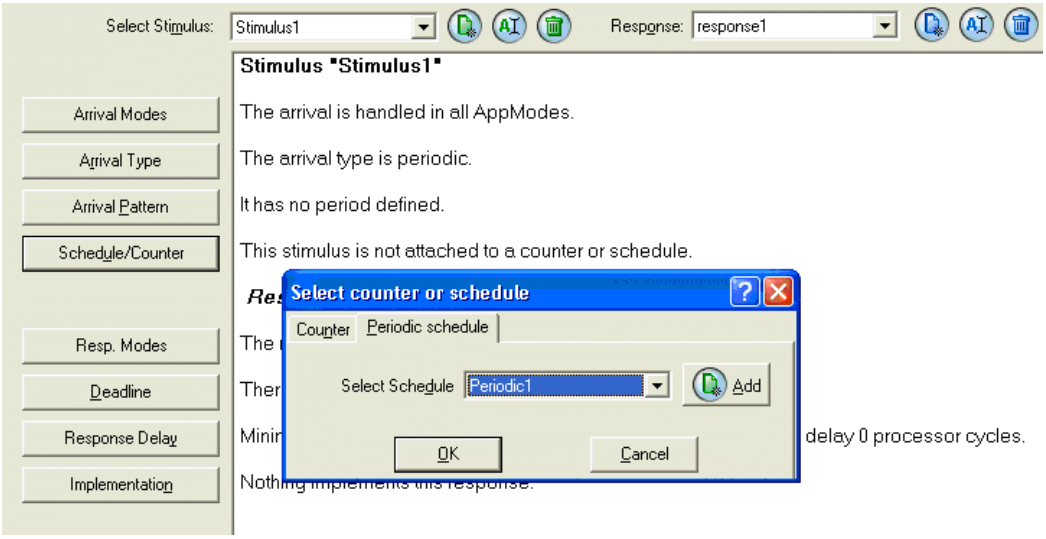


図 13-3 周期的スティミュラスを周期的スケジュールにアタッチする

スティミュラスをスケジュールにアタッチすると、RTA-OSEK がランタイムに使用できる暗黙的なアライバルポイントが作成されます。アライバルポイントに到達すると、スティミュラスに対してレスポンスを生成するために必要なすべてのタスクが起動されます。

アライバルポイントの周期は、スティミュラスのアライバルパターンとして定義されている周期が適用されますが、それはスケジュールのチックとして使用されます。

たとえば、20ms のスティミュラスが定義されていてチックレートが「5ms ごとに 1 チック」と定義されている場合、このスティミュラスのスケジュール周期は「4 スケジュールチック」になります。これは図 13-4 のように設定します。

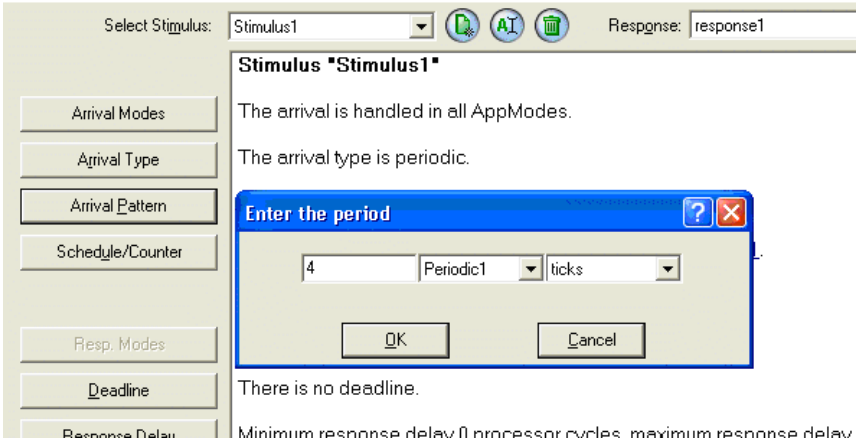


図 13-4 周期的アライバルパターンを定義する

重要

スケジュールが定義されたとおりに動作するためには、プライマリプロファイルのアライバルパターンが、スケジュールの 1 チックの解像度で実現できる、という条件が必要です。定義されたスケジュールの解像度をより高い精度のアライバルパターンが指定されていると、アライバルレートは、指定されたチックレートの解像度で実現できる次のアライバルパターンに切り上げられます。

13.2.2 周期的スケジュールの可視化

RTA-OSEK GUI では、Periodic Schedule ワークスペースの **Graphic** タブを選択することにより、スケジュールをグラフィックとして可視化することができます。このグラフィック表示は、タスクと ISR の実行時間が定義されている場合にしか表示されません。

ここにはスケジュール内のスティミュラスのアライバルとプライマリプロファイルが表示されます。しかしスティミュラスレスポンスの実行時間は、ユーザーが実行プロファイルを定義していない場合は表示されません。

図 13-5 の例では、5ms、10ms、20ms という周期を持つ 3 つのスティミュラス、`Stimulus1`、`Stimulus2`、`Stimulus3` があります。これらは、5ms につき 1 チックというチックレートを持つ周期的スケジュールにアタッチされています。

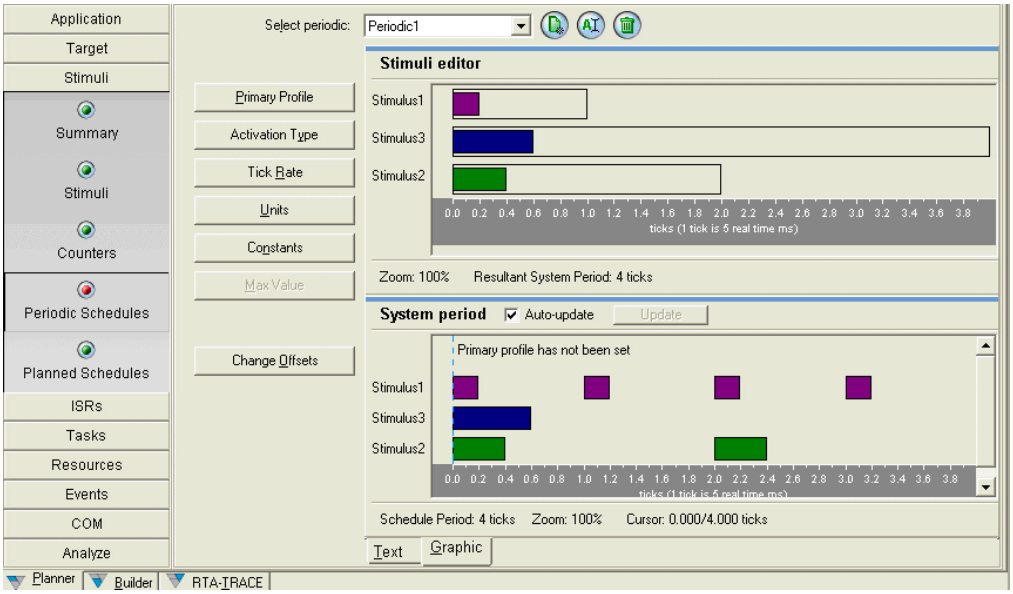


図 13-5 周期的スケジュールのグラフィック表示

13.2.3 周期の編集

周期的スケジュール上のスティミュラスの周期を変更するために、Stimulus ダイアログでスティミュラスのアライバルパターンを変更することができます。また、この変更は Periodic Schedule ワークスペースの **Graphic** タブで Stimuli Editor（スティミュラスエディタ）を使用して行うこともできます。周期は、スケジュールチックレートの整数の倍数でなければなりません。

Stimuli Editor には、各スティミュラスについて、その周期がボックスで表わされます。このボックスの右端を左右にドラッグして、スティミュラス周期を短くしたり長くしたりすることができます。周期が変更されると、System Period フィールドに新しい実行パターンが表示されます。

13.2.4 オフセットの編集

スケジュール内の挙動は制約的です。つまり、スケジュールの構成に従い、スケジュール自身と相対的にどのタスクがどのタイミングで実行されるかが正確に予測できます。

これに加え、周期的スケジュールにおいては **オフセット** を使用できます。オフセットによって任意のアライバルポイントの発行タイミングをずらすことにより、他のアライバルポイントで起動されるタスクによる干渉やブロックを最小限に留めることが可能となります。

また各スティミュラスにオフセットを与えることにより（ただし最低 1 つのスティミュラスに関してはオフセットがゼロでなければなりません）、プロセッサの負荷を軽減することができます。図 13-5 の例では、3 つのスティミュラスはすべて時間ゼロのタイミングで同時にトリガされていますが、ここで `Stimulus3` に 1 チックのオフセットを与えれば、同時に発生するスティミュラスは 2 つのみとなります。たとえばシステムがスケジューラブルでなくなった場合、この方法で対処できる場合もあります。

オフセットは、スティミュラスの周期よりも短くする必要があり、またスケジュール内で最低 1 つのスティミュラスにゼロのオフセットを与える必要があります。

13.2.2 項と 13.2.3 項で説明した機能を使用してスケジュールをグラフィック表示すると、図 13-6 のようになります。`Stimulus3` が `Stimulus1` と `Stimulus2` と同時に発行されていることがわかります。

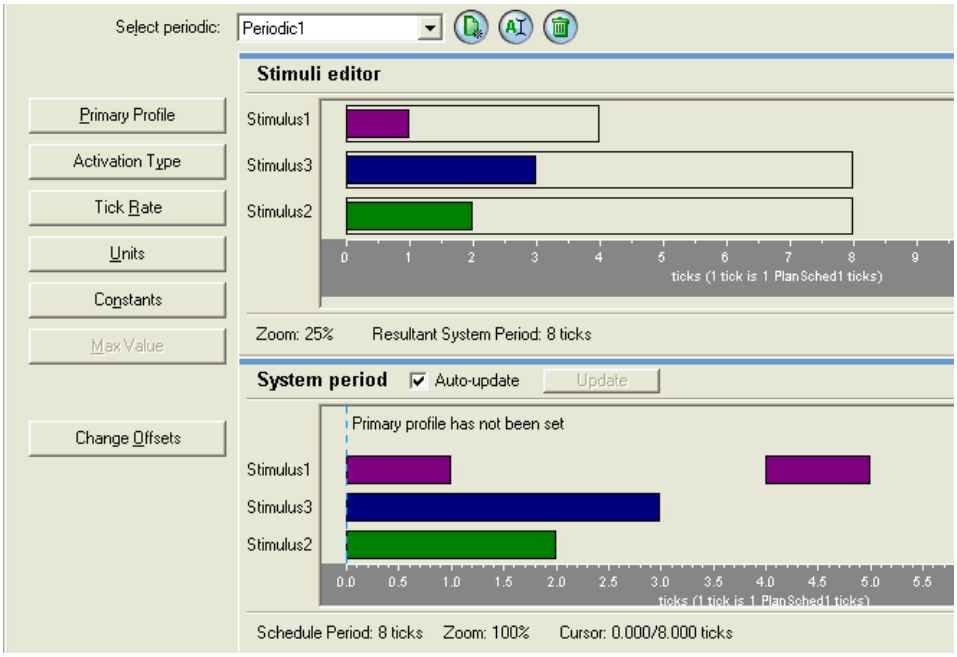


図 13-6 周期オフセットの使用

`Stimulus3` の発行によって起動されるタスクは、起動後最大 3ms までは CPU にアクセスせず、稼働後は、後続の `Stimulus1` の到達によりプリエンプトされます。図 13-6 から、このスケジュール上のタスクがまったく稼働していないタイムフレームがあり、このタイムフレームが `Stimulus3` の実行時間よりも長いことがわかります。

そこで `Stimulus3` の開始を 5ms だけオフセットで遅らせることにより、スケジュール上のプリエンプションをなくし、タスクのレスポンスタイムを短縮することができます。

このオフセットの効果を図 13-7 に示します。

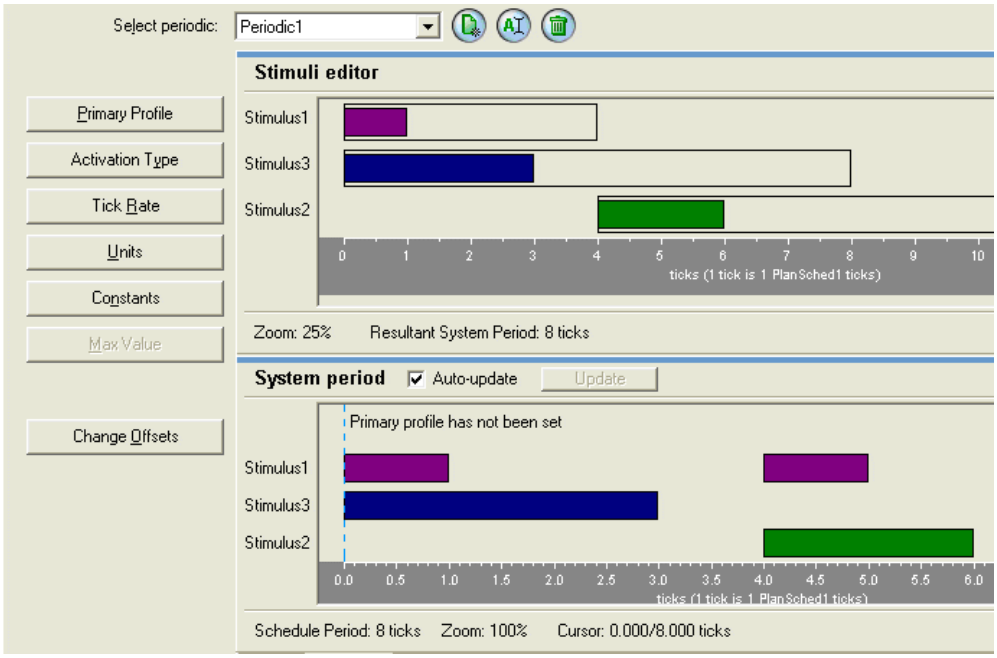


図 13-7 オフセットを変更する

ただし、アプリケーション内の他のタスク起動について見た場合、このオフセットを設けることによって全体的なパフォーマンスが低下してしまう場合もあります。タイミング分析を使用して、そのような状態になっていないかどうかを確認してください。

13.2.5 スケジュール／アライバルポイントのトレードオフ

周期的スケジュールにスティミュラスが追加されると、RTA-OSEK は暗黙的なアライバルポイントを作成します。これらのアライバルポイントは RTA-OSEK コンポーネントがランタイムに所定のタイミング挙動を実現するために使用されます。

1 つの周期的スケジュールには任意の数の周期レートをアタッチすることができ、各レートは互いに倍数である必要はありません。しかし RTA-OSEK は所望の挙動を実現するのに必要なアライバルポイントを自動的に作成するため、その数は非常に多くなってしまいます。

たとえば、周期的スケジュールを作成し、それに周期レートがそれぞれ 8ms、13ms、16ms、32ms、1024ms のスティミュラスをアタッチした場合、アライバルポイントの数は以下のように算出されます。

最小公倍数　= 13 × 1024
　　= 13312

最小の「倍数でない」レート用のアライバルポイント数　= 13312/8 + 13312/13
　　= 2688

「倍数でない」レートの共通のアライバルポイント数　= 13312 / (8 × 13)
　　= 128

アライバルポイントの総数　= 2688 - 128
　　= 2560

つまり、RTA-OSEK は 2560 点のアライバルポイントを作成することになりますが、各アライバルポイントにはメモリが消費されるため、無駄なメモリ消費が発生してしまいます。そこで代案として 2 つのスケジュールを宣言し、1 つを 1024ms スティミュラス用、もう 1 つを残りのスティミュラス用にすると、アライバルポイントは 80 + 1 = 81 点ですみます。

これよりもさらによいのは、13ms 周期のスティミュラス用に第 3 のスケジュールを宣言する方法です。するとアライバルポイントは全部で 7 点あればよいことになります。

> **重要**
> 定義された数のスティミュラス用に生成されるアライバルポイントの正確な数は、各スティミュラス間のオフセットにも依存します。

13.3 計画的スケジュールの設定

スティミュラスが非周期的に発生するシステムの構築は、計画的スケジュールを使用して行うことができます。

各スケジュールには固有の名前とチックレートを定義する必要があります。図 13-8 には、計画的スケジュールの設定内容が表示されています。

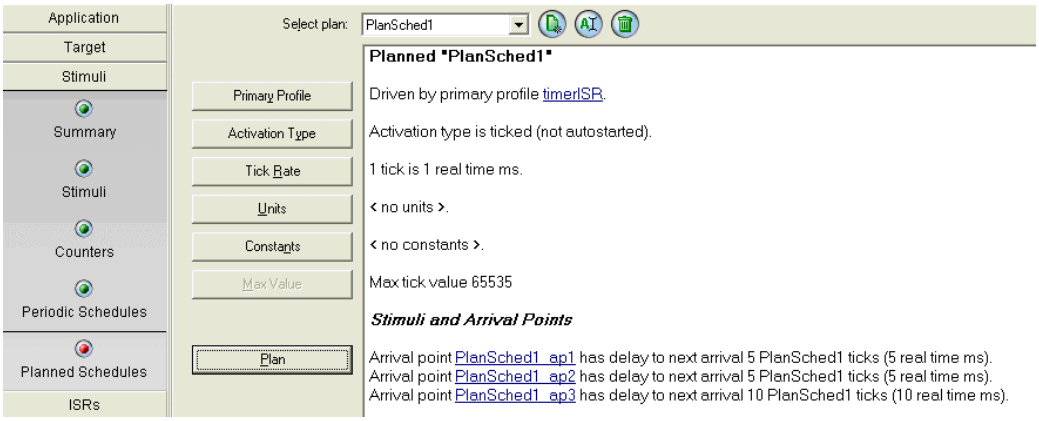


図 13-8 計画的スケジュールの設定

スケジュールはプライマリプロファイルにより駆動されます。ユーザーのアプリケーションでは、このプライマリプロファイルは普通は ISR です。デフォルトでは、計画的スケジュールはすべてチックドスケジュールとして設定されます。このスケジュールをアドバンスドスケジュールに変更したい場合は、13.5 項を参照してください。

13.3.1 計画的スケジュールの構築

周期的スケジュールを構築する際、スティミュラスの周期はスティミュラスの発生を定義するために使用されます。たとえば、20ms という周期を指定すると、スティミュラスは 0ms、20ms、40ms.... に発生します。

計画的スケジュールでは、計画的スティミュラスのアライバルについて完全に定義されていなければなりません（このため、計画的スティミュラスのアライバルパターンは、周期的スティミュラスの場合とは異なり、Stimulus ワークスペースには定義されません）。タイミング情報は計画的スティミュラスのみと関連付けられます。

計画的スケジュールを作成する場合、ユーザーは以下のことを行う必要があります。

- 計画的スティミュラスを計画的スケジュールにアタッチする
  つまり、スケジュールで発生させるスティミュラスを RTA-OSEK に通知します。
- どのスティミュラスをどのアライバルポイントにアタッチするかを指定する
  つまり、スティミュラスが発生すべきタイミングを RTA-OSEK に通知します。

図 13-9 は、計画的スティミュラスを計画的スケジュールにアタッチする方法を示しています。

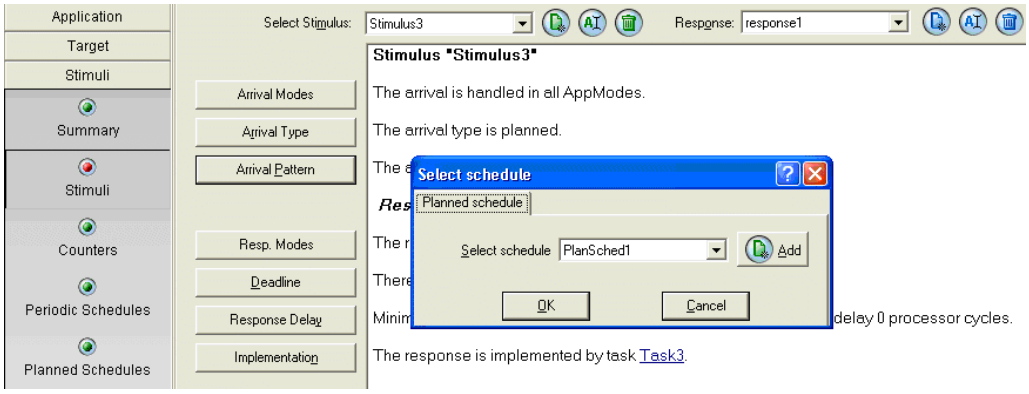


図 13-9 計画的スティミュラスを計画的スケジュールにアタッチする

13.3.2 アライバルポイントの作成

それぞれの計画的スケジュールには 1 つの **プラン** があり、そのプランの中に一連のアライバルポイントが設定されます。このプランを使用して、スティミュラスがいつ発生するかを定義します。各アライバルポイントに複数のスティミュラスを含めることも、1 つのスティミュラスを複数のアライバルポイントにアタッチすることもできます。

図 13-10 は、アライバルポイントの設定方法を示しています。

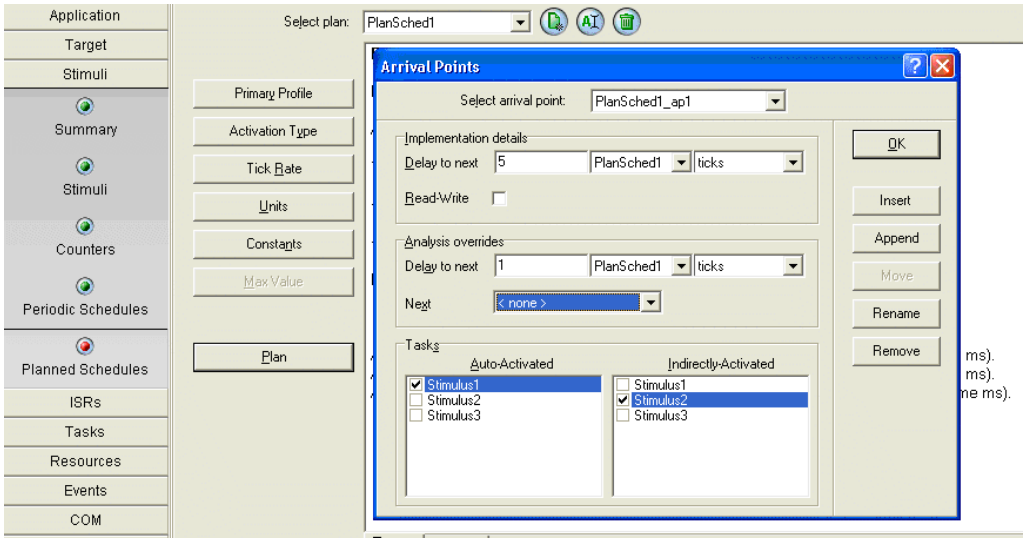


図 13-10 アライバルポイントの設定

各アライバルポイントについて、以下の項目を定義します。

- 固有の名前
- 次のアライバルポイントまでのディレイ
- アライバルポイント到達時にトリガするスティミュラス

各アライバルポイントについて、分析オーバライドを設定することができます。これは、タイミング分析でしか使用されません。

プランは、ランタイムに定義する必要のあるアライバルポイントのシーケンスを入力することにより作成されます。RTA-OSEK GUI でのスケジュールの設定は、リンクされたリストの作成と比較することができます。アライバルポイントはスケジュールに挿入することもスケジュールに付加することもできます。

RTA-OSEK コンポーネントは、アライバルポイントを、それらがリストに定義されている順序で処理します。この順序はワークスペース上で確認することができます。

図 13-10 のダイアログボックスを使用して、選択されているポイントの前にアライバルポイントを挿入したり、リストの終わりにアライバルポイントを付加したりできます。

反復アライバルポイントが選択されていない場合は、スケジュールは**シングルショット**です。つまり、スケジュールは開始されると最後まで実行されて停止します。このことは、一部の散発的な外部スティミュラス（実世界の割込みなど）に対する応答として発行される可能性のある内部スティミュラスのシーケンスを作成するのに好都合です。

計画的スケジュールは、リストの最後のアライバルポイントの Next 属性を、それより前の任意のアライバルポイントを「指し示す」ように設定することにより、ループ構造を作成することができます。これを行うには、**反復アライバルポイント**を定義する必要があります。

アライバルポイント間の最小ディレイは、デフォルトでは 1 スケジュールチックですが、ほとんどのアプリケーションにおいて、このデフォルトを変更する必要があります。シングルショットの周期的スケジュールの場合、リストの最後のアライバルポイントの遅延は問題ありません。

13.3.3 スティミュラスをアライバルポイントにアタッチする

アライバルポイント到達時にトリガする必要のあるスティミュラスは、**自動起動**（'auto-activated'）と呼ばれます。これらのスティミュラスは、ユーザーのスケジュールに関連付けられているスティミュラスの中から選択されます。1 つのアライバルポイントに任意の数のスティミュラスをアタッチすることができ、また 1 つのスティミュラスをスケジュール内の複数のアライバルポイントにアタッチすることができます。スケジュールにアタッチされているすべてのスティミュラスは、必ず 1 つ以上のアライバルポイントにアタッチされていなければなりません。

ここで例として、`Stimulus1` と `Stimulus2` という 2 つのスティミュラスがあり、以下のような到達条件が要求されるものとします。

- `Stimulus1` は 0、5、20、25、40、45ms... のタイミングで実行される必要があります。

- `Stimulus2` は 10ms ごとに周期的に実行される必要があります。

このシステムは、以下の 3 つのアライバルポイントを持つ計画的スケジュールを用いて実現できます。

- `ap1` は `Stimulus1` と `Stimulus2` を自動的に起動します。`ap2` までのディレイは 5ms です。
- `ap2` は `Stimulus1` を自動的に起動します。`ap3` までのディレイは 5ms です。
- `ap3` は `Stimulus2` を自動的に起動します。`ap1` までのディレイは 10ms です。

このプランを入力すると、周期的スケジュールのワークスペースの表示は図 13-11 のようになります。

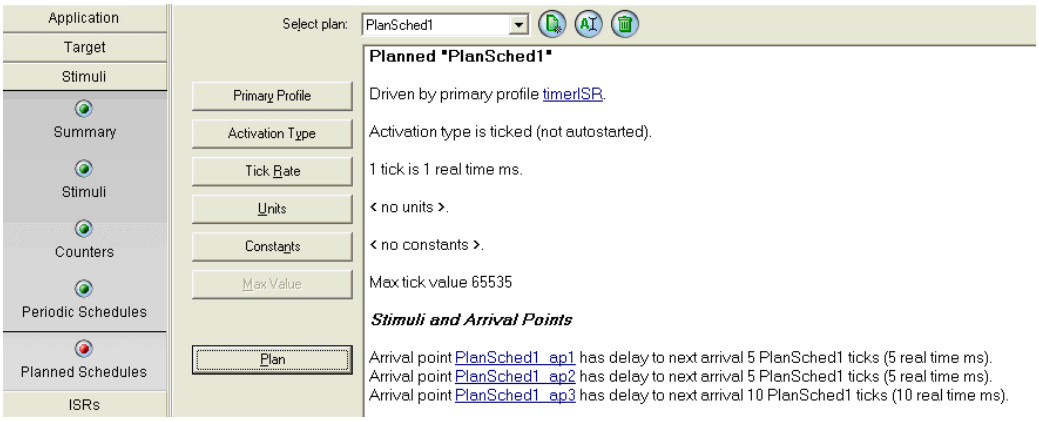


図 13-11 スティミュラスをアライバルポイントにアタッチする

13.3.4 計画的スケジュールの可視化

計画的スケジュール用のプランを作成したら、それをグラフィックとして可視化することができます。ここには、スティミュラスとアライバルポイントが表示され、タスクと ISR の実行時間が定義されている場合は、それらの情報も表示されます。

スケジュールのこのグラフィック表示は、図 13-12 のように Planned Schedule ワークスペースの **Graphic** タブで見ることができます。ここにはスケジュールとプライマリプロファイルのスティミュラスのアライバルが表示されます。

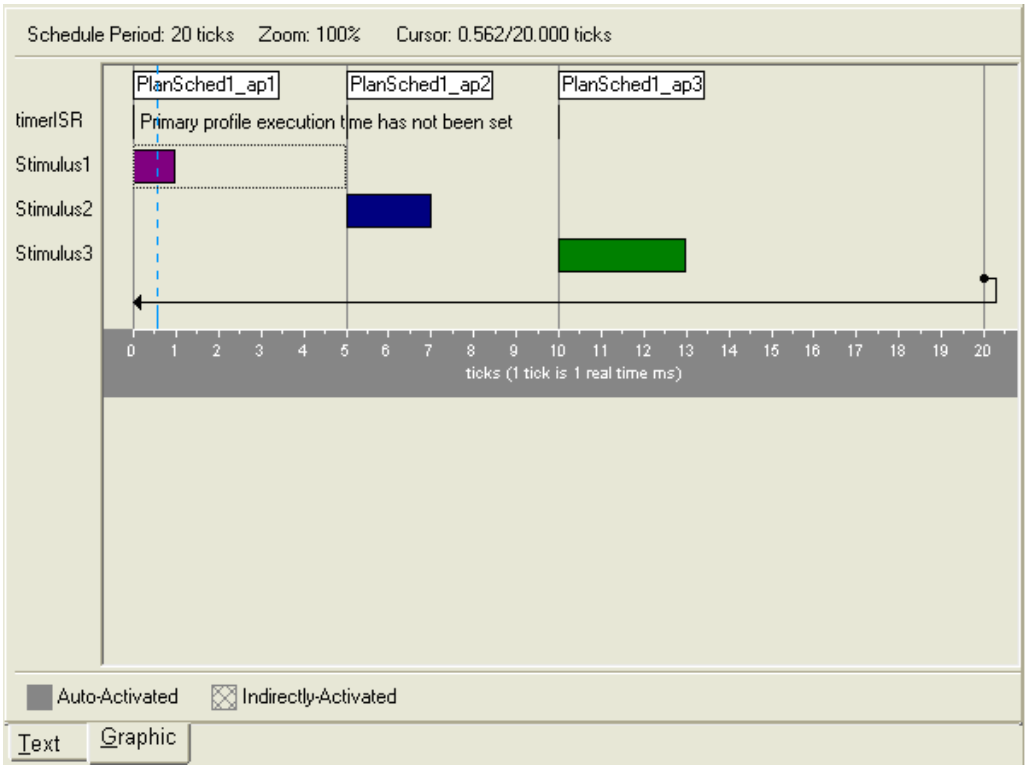


図 13-12 計画的スケジュールのグラフィック表示

13.3.5 プランの編集

計画的スケジュールのプランは、Planned Schedule ワークスペースの **Graphic** タブでグラフィカルに編集できます。アライバルポイント上にマウスポインタを合わせるとツールチップが開き、アライバルポイントに設定されているディレイの値が表示されます。

ディレイを変更するには、アライバルポイントの時間インジケータ（垂直バー）を左右にドラッグします。また、計画的スケジュールの反復プロパティを変更するには、反復シーケンスの先頭にしたいアライバルポイントを右クリックし、'Repeat Arrivalpoint' コマンドを選択します。

13.4 スケジュールをチックする

アプリケーションでスケジュールが使用される場合、ユーザーはチックソースを設けることによりスケジュールをチックする必要があります。チックの方法には制約はありませんが、一般的にはカテゴリ 2 の ISR を使用します。

RTA-OSEK を用いてアプリケーションをビルドする際、定義されているチックドスケジュールごとに `TickSchedule_<ScheduleID>` という API コールが自動的に作成されます。このコールは、スケジュールをチックする必要があるポイントで必ず実行されなければなりません。

たとえば、アプリケーション内で `Schedule1` と `Schedule2` が共にチックドスケジュールとして定義されている場合、RTA-OSEK は以下の API コールを作成します。

```
TickSchedule_Schedule1()
TickSchedule_Schedule2()
```

コード例 13-1 RTA-OSEK により生成される TickSchedule( ) API コール

スケジュールを進めるためにユーザーが作成する割込みハンドラでは、必ずこの API を呼び出してください。コード例 13-2 に、スケジュールのチックドライバの使用例を示します。

```
ISR(ISR1) {
   ServiceInterrupt();
   TickSchedule_Schedule1();
}
```

コード例 13-2 チックドスケジュールドライバを作成する

TickSchedule_<ScheduleID> コール処理の状態モデルは図 13-13 のとおりです。

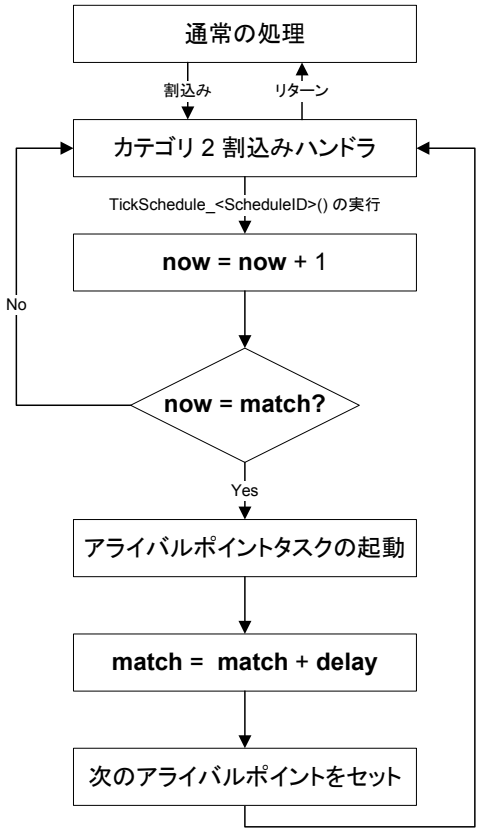


図 13-13 チックドスケジュールによるタスク起動処理

13.4.1 チックドスケジュールの自動開始

チックドスケジュールは、StartOS() の処理が終わってから所定のチック数だけ経過した後で自動的に開始するように設定することができます。

スケジュールを直ちに開始させる場合は、1 チック後に自動的に開始するように設定してください。最初のアライバルポイントが次の TickSchedule_<ScheduleID> コールで処理されます。スケジュールは必ず最初のアライバルポイントから開始されます。

図 13-14 の例では、スケジュールが自動的に開始するように設定されています。

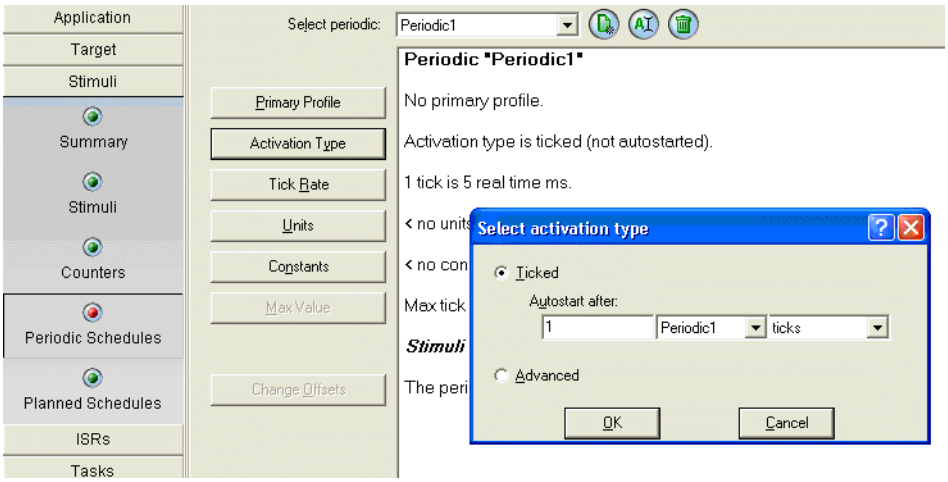


図 13-14 スケジュールを自動的に開始させる

13.5 アドバンスドスケジュール

前項までに説明されているチックドスケジュールは、アライバルポイントのディレイの解像度が粗めの場合に好都合です。RTA-OSEK コンポーネントが現在のカウント値を記録するために使用する内部カウンタの解像度は、`TickType` のサイズによる制約を受けています。`TickType` のサイズは、RTA-OSEK GUI の Target Details で見ることができます。

チックが 1ms である場合、16 ビットの `TickType` で定義できる最長のディレイは 65.535 秒ですが、チックを 1µs にすると、定義できる最長のディレイは 65.535 ミリ秒になります。

このように、チックドスケジュールでは、ディレイの範囲と解像度のどちらか一方を犠牲にすることになってしまいますが、この問題は、アドバンスドスケジュールを使用することにより解決できる可能性があります。アドバンスドスケジュールにおいてはターゲット上のカウンタ比較ハードウェアが使用されるので、広範囲の値を高解像度で扱うことができます[1]。

図は、アドバンスドスケジュールの処理を示しています。

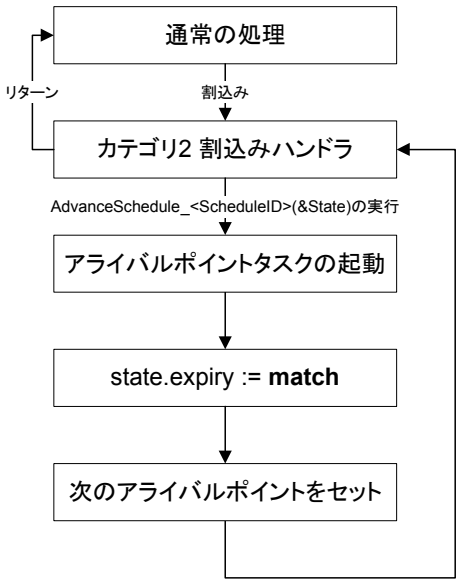


図 13-15 アドバンスドスケジュールによるタスク起動処理

1. 実際の範囲は、ターゲットハードウェアの設定により異なります。

アドバンスドスケジュールでは、割込みはアライバルポイントを処理する必要があるときにだけ生成されます。たとえば、スケジュールの 0、3、および 7ms にアライバルポイントがあり、スケジュールチックレートが 1ms である場合、そのスケジュールがチックドスケジュールであれば、システムには 8 回の割込みが発生します。

一方、アドバンスドスケジュールであれば、3 つの各アライバルポイントについてそれぞれ 1 回ずつ割込みを受けるだけなので、アプリケーションがスケジュールドライバ割込みにより受ける影響を減らすことができます。

図 13-16 と図 13-17 は、この効果を両方のスケジュールについて比較したものです。図 13-16 は、チックドスケジュールのグラフィック表示を示しています。

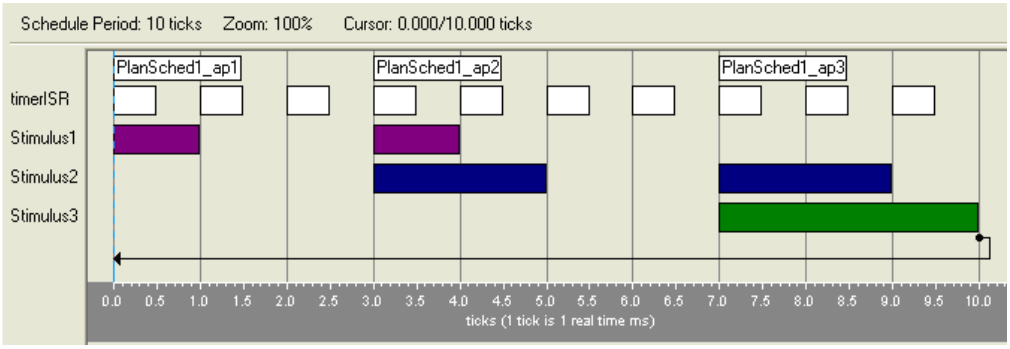


図 13-16 チックドスケジュールのグラフィック表示

図 13-17 は、このスケジュールのアドバンスバージョンを示しています。これを見ると、割り込み処理の時間をどのくらい削減できたかがわかります。

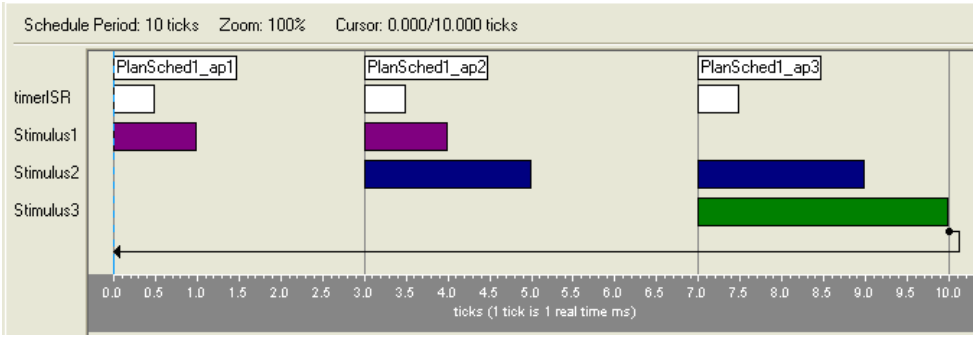


図 13-17 アドバンスドスケジュールのグラフィック表示

13.5.1 アドバンスドスケジュールのドライバコールバック

アドバンスドスケジュールの場合、RTA-OSEK コンポーネントはカウンタ比較ハードウェアにアクセスし、次の満了時を処理できるようにする必要があり、ユーザーはこのための関数を用意する必要があります。

1 つのアドバンスドスケジュールにつき以下の 4 つのコールバック関数が必要です。

- `Set_<ScheduleID>`

  カウンタ比較ハードウェアをセットアップします。この関数のプロトタイプは以下のとおりです。
  ```
  OS_CALLBACK(void) Set_<ScheduleID>(TickType Match);
  ```

- `State_<ScheduleID>`

  スケジュールのステータスと、スケジュールの次回満了時を返します。この関数のプロトタイプは以下のとおりです。
  ```
  OS_CALLBACK(void) State_<ScheduleID>(ScheduleStatusRefType State);
  ```

- `Now_<ScheduleID>`

  カウンタの現在値を返します。この関数のプロトタイプは以下のとおりです。
  ```
  OS_CALLBACK(TickType) Now_<ScheduleID>(void);
  ```

- `Cancel_<ScheduleID>`

  未処理のカウンタ満了をキャンセルします。この関数のプロトタイプは以下のとおりです。
  `OS_CALLBACK(void) Cancel_<ScheduleID>(void);`

上記の関数のうち、最初の 3 つはそれぞれ 3 つのスケジュール状態変数に対応しています。4 番目のキャンセル関数には RTA-OSEK コンポーネントがカウンタを停止するためのハンドルを提供します。アドバンスドスケジュールでは、この情報は RTA-OSEK コンポーネントによってではなくカウンタ比較ハードウェアによって扱われます。アドバンスドスケジュールドライバインターフェースの詳細については『RTA-OSEK リファレンスガイド』を参照してください。

13.6 スケジュールの起動

スケジュールの起動は、`StartSchedule(ScheduleID, TickType)` という API を使用して行います。スケジュールは通常 `OS_MAIN` 内で起動されますが、アプリケーション内の任意の箇所で起動することもできます。

`StartSchedule()` の引数 `TickType` は、関数が呼び出されてからスケジュールが最初のアライバルポイントを処理するまでの相対時間をスケジュールチック単位で指定するものです。つまり、この引数でスケジュール上の最初のアライバルポイント用の `match` 値を設定します。コード例 13-3 は、2 つのスケジュールを起動する方法を示しています。

```
StartSchedule(PeriodicSchedule1, 20);
StartSchedule(PlannedSchedule1, 200);
```

コード例 13-3 スケジュールを開始する

上の例の 1 つめのコールを実行すると、`PeriodicSchedule1` が現在のスケジュールの `now` 値から 20 スケジュールチック経過後に起動されます。また、2 つめのコールを実行すると、`PlannedSchedule1` が現在の `now` 値から 200 チック後に起動されます。

重要：`StartSchedule()` に渡される `match` 値は、コールが終了する前に満了することのないように、十分な長さにしてください。

13.6.1 シングルショットスケジュールの再起動

終了したシングルショットスケジュールを再起動する際は、特別な注意が必要です。この時点では、スケジュールの `next` 値は最後のアライバルポイントを指し示しています。スケジュール全体をもう一度実行するには、`SetScheduleNext(ScheduleID, ArrivalpointID)` という API を呼び出すことにより、next ポインタをリセットしてスケジュールの最初のアライバルポイントを指し示すようにしてください。

13.7 スケジュールの停止

スケジュールの停止は、`StopSchedule(ScheduleID)` という API を使用して任意のタイミングで行うことができます。

この関数は、スケジュールをその時点のカウント値で止めます。ただしシングルショットスケジュールが使用されている場合は、スケジュールは最後のアライバルポイントが処理されてから自動的に停止します。

13.8 時間ベースでないスケジュール単位の使用

ここまでの項ではすべて時間をチックとして使用するスケジュールについて説明されていますが、RTA-OSEK GUI にはスケジュール**単位**（'schedule **unit**'）を宣言する機能があり、この「スケジュール単位」を使えば、実際の物理単位を使用してスケジュールチックを定義することができます。

たとえば、歯付きタイミングホイールの歯をカウントし、所定の回転角においてタスクを起動する、というスケジュールを想定します。これを抽象化するには、まず「度」('degrees') というスケジュール単位を宣言し、さらに 1 回転が 360 度であることを定義します。この宣言は図 13-18 のように行います。

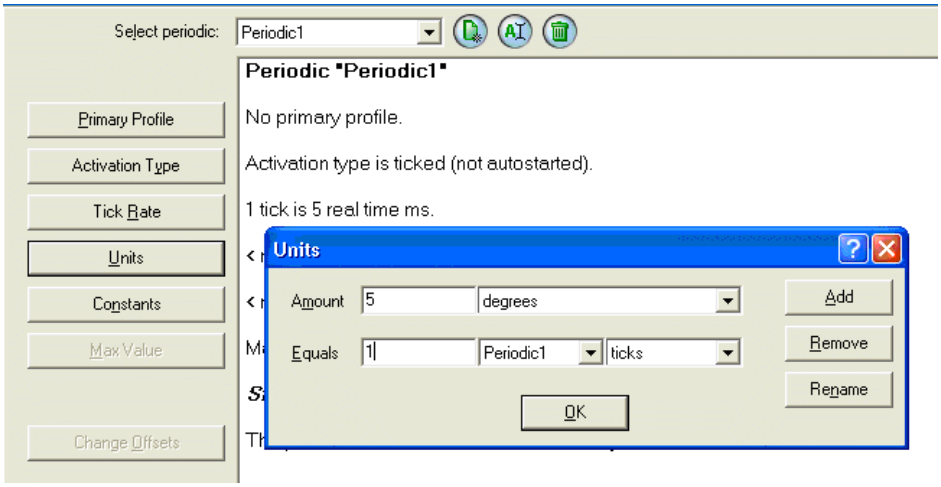


図 13-18 角度をスケジュール単位として宣言する

ここで、たとえば角度スケジュールが 1 度につき 5 チックを必要とする場合、ユーザーは 1 度回転するたびに 5 チックになるような割込みソースを用意する必要があります。

13.9 スケジュール定数の定義

ランタイムにスケジュールを変更する必要がある場合は、ディレイ値（'delay value'）を定義することができます。ディレイ値は、次のアライバルポイントが処理されるまでに経過しなければならない時間を表します。共通して使用されるディレイ値をシンボリック**定数**（'symbolic **constant**'）として宣言します。

ハードコーディングされた数値をユーザーのアプリケーションで使用する場合は、それらの数値がアプリケーションコードで適切にスケーリングされるようにする必要がありますが、このシンボリック定数を使用すれば、様々な値（チック解像度など）を変更してもアプリケーション内では正しい定数の値が使用されます。

スケジュール API 関数に渡すスケジュールチック値の定義には、できるだけスケジュール定数を使用してください。これらの定数は、生成されるヘッダファイルを通じてアプリケーションコード内で使用できます。

図 13-19 では、360 度という値を持つ `Revolution` という定数が設定されています。

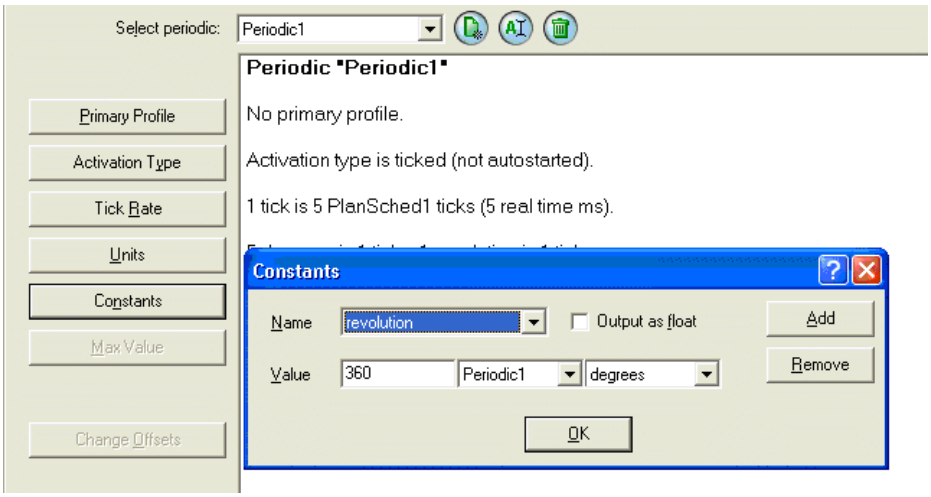


図 13-19 スケジュール定数を宣言する

13.10 ランタイムにおいて計画的スケジュールを変更する

計画的スケジュールは、スケジュールとそれに関連付けられているアライバルポイントをランタイムにおいて変更できるので、柔軟性があります。

たとえば、エンジン制御アプリケーションにおいてスケジュールを使用して、エンジン回転数の上昇や下降に伴ってタスク起動間の時間間隔を変動させる、というようなタスクの段階的起動を実現することができます。

また、これを利用してランタイムのフォルトトレランス性を実現することもできます。たとえばセンサハードウェアで障害が検知された際に、実際のセンサ値を読み取るタスクの代わりに適切なセンサ値を合成するタスクを使用する、といったことが可能となります。

RTA-OSEK コンポーネントには、スケジュールとそれに関連付けられているアライバルポイントの現在の状態を取得する API 関数があります。また、プロパティに新しい値を設定する API 関数もあります。

RTA-OSEK コンポーネントはスケジュールのステータス値をランタイムに更新する必要があるので、それらの値は常に RAM 内に置かれます。アライバルポイントはデフォルトでは ROM に置かれます。

アライバルポイントを読み書き可能として定義すると、ランタイムにおいて、そのアライバルポイントに関連付けられているスティミュラスに対応して起動されるタスクセットを変更することができます（これについては 13.10.3 項で説明します）。また、ディレイ値や Next 値もランタイムに変更できます（13.10.1 項と 13.10.2 項で説明します）。

読み書き可能なアライバルポイントは、RAM に配置されます。図 13-20 に示すように、RTA-OSEK GUI を使用してアライバルポイントを読み書き可能にする設定を行います。

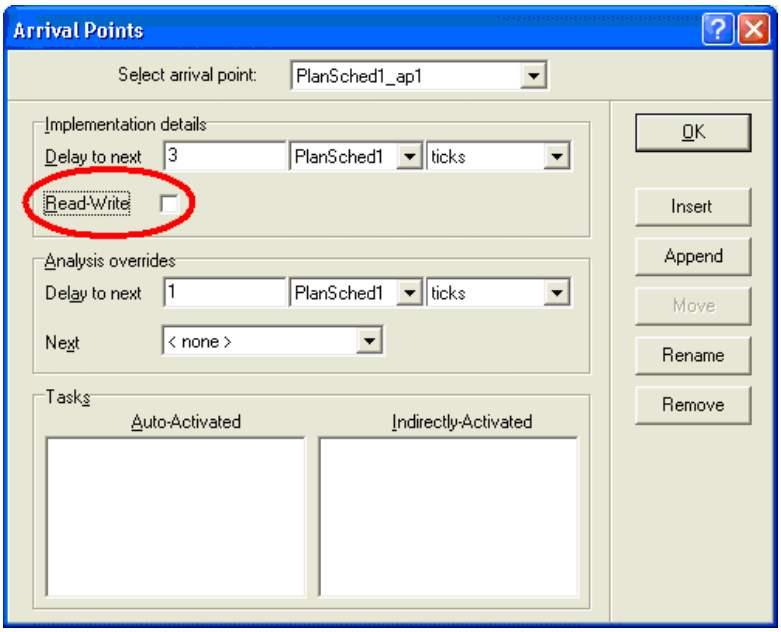


図 13-20 読み書き可能なアライバルポイントを定義する

重要

アプリケーションについてタイミング分析を行う場合は、ユーザーはスケジュールに対するワーストケースの変更についての情報を、詳細に指定する必要があります。

13.10.1 ディレイの変更

ディレイ値の変更に使用できる API 関数は、以下の 2 つです。

- `GetArrivalpointDelay(ArrivalpointID, TickType)`
  アライバルポイントの現在のディレイ値にアクセスします。

- `SetArrivalpointDelay(ArrivalpointID, TickType)`
  読み書き可能なアライバルポイントにディレイ値を設定します。

重要

`GetArrivalpointDelay()` と `SetArrivalpointDelay()` に「ディレイ値ゼロ」を指定すると、ディレイの実際の値はゼロにはならず、スケジュールの最大値となります。

13.10.2 Next 値の変更

スケジュールとアライバルポイントの Next 値は、どちらも変更することができます。スケジュールの Next 値を変更すれば、次に処理されるアライバルポイントを変更することができます。

通常、スケジュールの Next 値を編集するのは、スケジュールを一時的に変更したい場合です。

スケジュールを永久的に変更したい場合は、アライバルポイントリストを編集して、スケジュールが処理されるたびにその変更が有効となるようにしてください。

Next 値を変更すると、任意指定のアライバルポイントを用いるスケジュールを作成することができます。このようなアライバルポイントは、ランタイムに組み込んだり除外したりすることができます。ただし、組み込む必要のあるアライバルポイントはすべて設定時に宣言しておく必要があります。アライバルポイントをランタイムに動的に作成することはできません。

Next 値の変更に使用できる API 関数は次の 2 つです。

- `SetScheduleNext()`
  これを使用してスケジュールの Next 値を変更します。
- `SetArrivalpointNext()`
  これを使用してアライバルポイントの Next 属性をランタイムに変更します。

次の例では、スケジュールに `Arrivalpoint1`、`Arrivalpoint2`、`Arrivalpoint3` という 3 つのアライバルポイントがあります。メインプログラムでは、`Arrivalpoint1` の次のアライバルポイントとして `Arrivalpoint3` が設定されています。

```
OSMAIN {
   ...
   StartOS(OSDEFAULTAPPMODE);
   SetArrivalpointNext(Arrivalpoint1, Arrivalpoint3);
   ...
}
TASK(Task1) {
   ...
   /* Switch in the pre-declared Arrivalpoint2.
    * Note that the next for Arrivalpoint2 is already set
    * to Arrivalpoint3 during configuration. */

   SetArrivalpointNext(Arrivalpoint1, Arrivalpoint2);
   ...
}
```

コード例 13-4 アライバルポイントの Next 値を変更する

スケジュールの反復挙動を変更する必要がある場合は、反復の次のアライバルポイントとして、アプリケーション内の別のアライバルポイントを設定することができます。

13.10.3 自動的に起動されるタスクの変更

各アライバルポイントにはタスクセットが割り当てられています。このタスクセットは、到達時に自動的に起動され、関連付けられているスティミュラスに対するレスポンスを生成するタスクが含まれます。このタスクセットには、`GetArrivalpointTasksetRef()` を使用してランタイムにアクセスすることができます。アライバルポイントが読み書き可能であれば、この API 関数は読み書き可能タスクセットへのポインタを返します。

アライバルポイントタスクセットの挙動はアプリケーション内の他のタスクセットと同様なので、その内容をランタイムに変更することができます。

この機能を利用することにより、起動されるタスクの動的な変更が可能になります。一例をコード例 13-5 に紹介します。

```
GetArrivalpointTasksetRef(Arrivalpoint1, &TmpTaskset);
MergeTaskset(TmpTaskset, NewTasks);
```

コード例 13-5 アライバルポイントで起動されるタスクを変更する

13.11 スケジュールの RAM 使用量を最小限にする

周期的スケジュールの場合、状態情報は RAM に保持され、暗黙的なアライバルポイントは ROM に保持されます。

計画的スケジュールの場合、アライバルポイントを ROM か RAM のどちらに配置するかをユーザーが選択することができます。たとえば、書き込みを行う必要があるアライバルポイントが 1 つだけの場合は、スケジュールの残りの部分を ROM に配置してください。

13.12 スケジュールのまとめ

- スケジュールは、複雑なイベントベースのシステムを構築する際に OSEK のカウンタ／アラームメカニズムの代わりに使用できる、一層柔軟な方法です。
- スケジュールは OSEK 規格には定義されていません。
- スケジュールは 4 つの状態変数と 1 つのアライバルポイントリストから成ります。
- アライバルポイントは、ランタイムにスティミュラスを実装するために使用されます。
- スケジュール上のアライバルポイントは、いつでも確実に同期化されます。
- アライバルポイントに到達すると、RTA-OSEK コンポーネントは、そのアライバルポイントで実装されるスティミュラスに対するレスポンスを生成するために必要なタスク群を起動します。
- 周期的スケジュールを使うと、周期的挙動を手早く定義することができます。アライバルポイントはすべて暗黙的に定義され、RTA-OSEK コンポーネントだけにより内部的に使用されます。
- 計画的スケジュールを定義するには、スケジュールのタイミング特性について明示的なプランを作成する必要があります。
- 計画的スケジュールをランタイムに変更して、特別なシステム挙動に対応することができます。

14 アドバンスドドライバの作成

前述のように RTA-OSEK には、カウンタや RTA-OSEK スケジュールを駆動するための簡単でスマート、かつパワフルなインターフェースが用意されています。この「アドバンスドドライバ」('advanced driver') メカニズムは、ソフトウェアとハードウェアとのやりとりをユーザー定義のコード内に配置することにより、非常に高い柔軟性を実現しています。そのため、新しいハードウェアやアプリケーション要件にも容易に対応でき、また複数の機能で使用されるハードウェアの「ピギーバック」ドライバ操作も可能となります。

ハードウェアをアプリケーションでどのように使用するかは、ユーザー自身が一番よくわかっているためこのアドバンスドドライバ関数はユーザーの責任において作成していただく必要があります。

本章は、ユーザーがアドバンスドドライバ関数を作成する際のガイドラインをまとめたものです。ほとんどは各種ペリフェラルタイマ用ドライバの作成時に適用されるものですが、さらに外部イベント（例：歯車の回転で発生する割込み）に応じて他の周辺機器をインクリメントする場合にも適用されます。

サンプルコードは理解しやすいように簡単な構造になっていますが、これに手を加えることにより（例：`if .. break` を使用する `while(1)` ループを、適切に条件定義された `do .. while` 文に置き換える）、より質の高いターゲット用コードを生成することができます。ただしこのようなコードの最適化を行う際は、各命令文のセマンティックスと順番が保持されるように十分注意してください（例：C 言語の論理演算子 `&&` では、実行順序と遅延評価という点を考慮する必要があります）。

14.1 アドバンスドドライバモデル

「アドバンスドドライバ」コンセプトは、フリーランニングを行うペリフェラルカウンタを使用することが前提となります。カウンタの初期値はユーザーが設定し、ゼロからカウントアップを行い、最大値においてゼロに戻り、カウントを続行します。

> **重要**
> 本章ではドライバの「モデル」を示すものです。この「モデル」としての条件に満たないハードウェアを使用してモデルを実現する方法についても、本章内で後述します。

「アドバンスドカウンタドライバ」は、アラームや、カウンタに関連付けられた満了ポイントに定義されたタイミングになると、`osAdvanceCounter_<CounterID>` という API 関数を使用し、RTA-OSEK に対して可能な限り速やかにその処理を行うように通知します。同様に「アドバンスドスケジュールドライバ」は、アライバルポイントに定義されたタイミングになると、`AdvanceSchedule_<ScheduleID>` という API 関数を使用し、RTA-OSEK に対して可能な限り速やかにその処理を行うように通知します。この時点でカウンタやスケジュールがまだ稼働中であれば、ユーザー定義のハンドラにおいて所定のアクションを行い、次のアラーム、満了ポイント、アライバルポイントが適切なタイミングで確実に処理されるようにする必要があります。

これら 2 つの API 関数は、以下のような C 構造体である ScheduleStatusType を返します。

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

コード例 14-1 RTA-OSEK により生成される TickSchedule( ) API コール

`status` は、以下のいずれかのドライバステータスが格納されます。

- OS_STATUS_RUNNING ー 実行中で、まだ満了ポイントに達していない
- OS_STATUS_PENDING ー 実行中で、満了ポイントが処理可能状態

`expiry` が定義されている場合、これは現時点（'now'）から次のポイントが処理されるまでのチック数を表します。つまり `expiry` は 'now' からの相対時間となります。

一般に、RTA-OSEK のアドバンスドドライバーインターフェースの呼び出しは、ユーザー定義のカテゴリ 2 割込みサービスルーチンから行います。

2 つの関数は、そのコンセプトにおいて非常に似通っています。以降の説明においては、以下のような記述方法を採用しています。

- 2 つの RTA-OSEK API 関数（`osAdvanceCounter_<CounterID>` と `AdvanceSchedule_<ScheduleID>`）のいずれかを示す場合、`Advance()` というダミーの関数を使用します。両者において差異がある場合はテキスト文で注釈を付記します。
- アラーム、満了ポイント、アライバルポイントは、すべて「ポイント」と記述します。を周期的スティミュラスを実装できます。これらのアイテム間で差異がある場合はテキスト文で注釈を付記します。
- 関数内で「アドバンスドカウンタ」または「アドバンスドスケジュール」と記述する部分では、「アドバンスド」と記述します。

14.1.1 アライバルポイント

割込みサービスタスク（ISR）は、対応するポイントが「満了する」（つまりそのポイントに定義されたタイミングになる）ことによってトリガされます。ISR は `Advance()` をコールすることにより、アドバンスドドライバに対して現在のポイントが満了したことを通知し、さらに次のポイントになるまでの時間（ディレイ）を取得します。また ISR はハードウェアを設定し、ディレイ経過後に再度割込みが発生するようにする必要があります。一般に、ISR の挙動は以下に示す 3 つのタイプに分かれます。各タイプとごにシステム挙動やスケジューラビリティ分析との関連性についても説明されているので、正しい動作を行う起動ドライバを実現するには、ドライバの割込みハンドラコンポーネントを作成する際に、この情報をもとに最適なタイプのハンドラを選択してください。

シンプルハンドラ（'**simple** handler'）： 1 つのポイントを処理するハンドラです。このタイプのハンドラは、次の割込みが発生する前に処理が完了している必要があります。このことが保証される場合は、このシンプルハンドラを使用することをお勧めします。一般的にこのハンドラは、3 つのタイプの中で、ワーストケースにおける処理時間が最短です。

再トリガハンドラ（'**retriggering** handler'）： 1 つまたは複数のポイントを処理できるハンドラで、先に発生した割込みによってトリガされた処理が完了する前に次のポイントを処理できます。このタイプのハンドラは、呼び出されるたびに 1 つのポイントを処理します。もしも処理中に別のポイントに到達すると、保持されている割込みを呼び出してから終了します。

ループピングハンドラ（'**looping** handler'）： 1 つまたは複数のポイントを処理できるハンドラで、先に発生した割込みによってトリガされた処理が完了する前に次のポイントを処理できます。このタイプのハンドラは複数のポイントの処理を順に実行し、処理するべきポイントがない場合、または 1 つの割込みが保留されている場合にのみ終了します。

シンプルハンドラが使用できない場合、再トリガまたはループピングトリガのいずれかを選択します。なおループピング機能がある割込みハンドラは、必ずループピングハンドラにします。

この選択には以下の 3 つ条件を考慮してください。

1. ハードウェアが再トリガ処理をサポートしていない場合はループピングハンドラを使用します。
2. ハンドラを呼び出す割込みと同じレベルの割込みがシステム内に存在し、また、その他方の割込みがより高いアービトレーションオーダー（つまり 2 つの割込みが同時に発生した際に先に処理される）を持っている場合、再トリガハンドラの方が適しています。それによってその他方の割込みのレイテンシが低減されるためで、これは特に割込み優先度レベルが 1 つしかないシステムに適用されます。
3. 1 つのポイントについて処理する際の実行時間は、一般的に再トリガハンドラの方がループピングハンドラよりも短くなります。一般的にループピングハンドラは効率的ですが、これは 1 回の呼び出しで複数のポイントが処理されるような場合にはあてはまりません。各ポイントが 1 回ごとの呼び出しで処理される際にワーストケース挙動が生じます。

ハンドラのワーストケース応答が各割込み間の最小間隔より短い場合は、シンプルハンドラの使用をお勧めします。ハードウェアの制約によりシンプルハンドラが使用できない場合にのみ、再トリガハンドラを使用するようにしてください。

14.1.2 コールバック関数

起動ドライバの一部として、以下の 4 つのコールバック関数も必要です。

1. Now_Advanced ー ペリフェラルカウンタの現在値を返します。

2. Cancel_Advanced ー 発生しているカウンタ用割込みをすべてクリアし、`Set_Advanced()` がコールされるまで割込みが発生しないようにします。この機能は、カウンタによって駆動されるアラーム、スケジュールテーブル、スケジュールが、アプリケーションにより、またはスケジュールテーブルやスケジュールが最後部に到達たことにより停止される場合に必要です。そのような状況が発生しない場合は、ダミー関数を用意しておいてください。
3. State_Advanced ー 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るときのみコールされます。次のポイントに到達するまでのチック数、または次のポイントにすでに到達して処理待ち状態になっていることを返します。この機能を利用してアプリケーションがステータスを取得できます。
4. Set_Advanced ー ステータスをセットし、次にカウンタ値が所定の値になったときに割込みが発生するようにします。このコールバック関数には、次のポイントを処理すべきタイミングを示す match 値（目標値）を絶対値で渡します。スケジュールの場合はスケジュールを起動する際にのみ使用され、カウンタの場合は、スケジュールの起動時に加え、次のポイントまでの時間を短くしたい場合にも使用できます。具体的には、定義されている match 値よりも早いタイミングにアラームをセット（またはスケジュールテーブルを起動）したい場合です。

これらのコールバック関数は、必ず OS レベルの優先度で作成してください。つまり、カテゴリ 2 割込みによってプリエンプトされることがなく、リエントラントな構造にする必要がないようにしてください。

14.2 「出力比較」ハードウェアの使用

本項では、出力比較（「コンペアマッチ」とも呼ばれます）を行うカウンタハードウェア用ドライバの構成について説明します。これらのハードウェアは、カウンタ値（クロック周波数や、センサからのイベント通知などの外部要因によってカウントされます）がソウトウェアで設定された比較値と一致したときに割込みを発行します。カウンタ値と現在の比較値はソフトウェアで取得できます。本項では、カウンタハードウェアのレジスタが OUTPUT_COMPARE および COUNTER という変数にマッピングされることを前提としています。以降に、正しいコールバック関数の内容を概説し、それに続いて、必要な動作やハードウェア機能に応じて異なる割込みハンドラの例を紹介します。

最初に、`TickType` と同じデータ幅（'modulus'）を持つカウンタについて説明します。`TickType` の最大値は通常、16 ビットターゲットでは 2^{16}、32 ビットでは 2^{32} です。

最大値までフルに使用して演算が行われる場合、1 ディレイ内のチック数は、ディレイの終了値から開始値を引くことによって求まります。現在のカウンタ値（COUNTER）を次の比較値（OUTPUT_COMPARE）から差し引くと、その結果は比較ポイントに達するまでのチック数となります。もしも、比較値をセットした後にこの値を読み込み、それが所定のディレイより大きかった場合、カウンタ値はすでに比較値を超えていて、比較が発生する前に 1 回の最大値ラップ（'modulus wrap'）、つまり `TickType` チックが必要となることを意味しています。このような状態は、次のポイントまでのディレイが非常に短いために（例：1 チック）、カウンタが所定の比較ポイントを通過するのと比較ポイントをセットする処理との間で競合状態（'race condition'）が生じた場合に発生します。

14.2.1 コールバック関数

Set

`Set_Advanced()` をコールすることにより、カウンタ値が引数の値にマッチしたときに割込みが発生するようにします。これを実現するには、コンペアマッチを無効にし、割込みをクリアして比較値をセットし、割込みをイネーブルにします。しかしコンペアマッチを無効にする機能を持たないハードウェアを使用する場合は、代わりに現在のカウンタ値より小さい値を比較値にセットし、次に比較値がセットされるまで値の一致が起こらないようにする必要があります。

ただし、コンペアマッチの無効化は必要ない場合もあります。システム起動時から起動ドライバが起動するまでの間にマッチが発生しないことが保証されていれば、コンペアマッチ機能の無効化は必要ありません。そのためには、たとえば以下の例のように比較レジスタを現在のカウンタ値の 1 つ前の値に設定することにより、チック数がカウンタの最大値に到達するまで「マッチ」割込みが発生しないようにします。それだけの時間があれば、`Set_Advanced()` 関数の残りの処理をすべて実行できます。ただしこれが行えるのは、比較レジスタを占有的に使用している場合に限ります。

```
OS_CALLBACK(void)  Set_Advanded(TickType Match)
{
   /* prevent match interrupts for "modulus" ticks */
   OUTPUT_COMPARE = COUNTER - 1u;
   dismiss_interrupt();
   OUTPUT_COMPARE = Match;
   enable_interrupt();
}
```

上記のコード、および以降のコードにおいて、以下のような関数が使用されています。

- `dismiss_interrupt( )`
- `enable_interrupt( )`
- `disable_interrupt( )`

これらの関数は、ペリフェラルカウンタのステータスレジスタやコントロールレジスタを使用してそれぞれの処理を実行することにより、適正な起動ドライバの機能を提供するものです。アドバンスドドライバ内のこれらの関数（またはそれに該当するコード）は、ユーザーの責任において作成してください

重要

上記のコードは、2 つの潜在的な競合状態（'race condition'）を避けるように工夫されています。競合状態とは、正しい方法で割込みをクリアにしなかったために、予期しない割込みが発生したり、処理すべき割込みが喪失してしまったりする状態を指します。具体的には以下のような状態です。

1. 前回の比較値とカウンタ値によって、比較レジスタがセットされる前に割込みが発生してしまう場合があります。これにより、前回の比較値とカウンタ値の比較の結果ではなく、`Set_Advanded()` の実行によって割込みが発生する、という状態になってしまします。
2. 比較レジスタがセットされた後に `dismiss_interrupt()` をコールすることにより、マッチ割込みをディセーブルにすることなく 1 番目の競合状態を避けることができます。しかしこれによって非常に短いディレイ（つまり、`Set_Advanced()` をコールした時にカウンタレジスタの値の次の 1 チック ）は無視されてしまい、カウンタのカウントが 1 周するまで比較による割込みが発生しない可能性もあります。またハードウェアによっては割込みが（ラップが行われた後も）まったく発生しなくなってしまうことも考えられます。

いずれのケースについても、非常に短いディレイを使用する場合、つまり比較ポイントがセットされる前にカウンタが比較ポイントに達してしまう可能性がある場合は、十分な注意が必要です。さらに、クロックの現在値を読み込んでセットポイントを計算し、そのポイントをセットするユーザーコードの実行パスが長い場合は、特に注意してください。このような状態においては、カウンタが 1 周しないと次のポイントに到達しません。

上記の例では、マッチ割込みは出力比較レジスタを変更することによって避けることができます。ただし以降の例には、その方法は示されていません。つまり、関数 `disable_compare()` によってハードウェアからのマッチ割込みが禁止されることが前提になっています。

重要

カウンタが起動ドライバ以外の目的で使用される場合、`disable_compare()` 関数がカウンタを HALT することは禁止されています。これは、時間経過がずれてしまうのを避けるためです。
コンペアマッチを再度有効にするには、比較レジスタのセットをアトミックに行う（つまり中断なく実行される）必要があります。これが守られないと、短いディレイ値が出力比較レジスタにセットされた場合、別の競合条件が発生する可能性があります。

上記のコールバック関数は、スケジュール、およびアラーム／スケジュールテーブルに共通して使用されるので、起動後の調整は必要はありません。ただしユーザーが `Set[Abs|Rel]Alarm()` または `NextScheduleTable()` をコールする可能性がある場合、別の `Set_Advanced()` コールバック関数を作成する必要があります。コールバック関数には、現時点（'now'）に近いポイントに現在設定されているマッチ値をリセットできる機能が必要です。

以下の説明文では、次の用語を使用します。

- **now** ー カウンタの現在値（連続的に増加します）

- **old** ― 前回設定された比較値
- **match** ― 新しい比較値（絶対値）
- **"-"** ― 2 進減算

また、優先度の高い割込みによる遅延は、カウンタの全範囲内においては小さなものであると仮定します。

単純な（よく練られていない）コードの場合、アトミックな処理によって比較値に **match** をセットしますが、これは正しくありません。より優先度の高い割込み（例：カテゴリ 1 割込み）によってハードウェアへの書き込み処理に遅延が生じるため、比較レジスタに **match** の書き込みが終了する前に、**now** がすでに **match** よりも大きくなってしまう可能性があるためです。これにより、スケジュール全体のすべての処理が 2^{16}（または 2^{32} など）チックの間、行われなくなってしまうことになります。実際には、比較レジスタに **match** を書き込む準備ができる前に **now** が **match** と **old** の両方の値よりも大きくなってしまう可能性もあります。

`Set_<CounterID>()` のコードにおいては、起動時（つまり割込みが停止している状態）のケースと、リセット時（スケジュールが稼動していて、ディレイを現在の比較値 **old** まで短くするために使用されている状態）のケースとで、異なる処理を実装する必要があります。

2 番目のケースにおいては、`Set_<CounterID>()` は、新しい **match** 値が書き込まれた比較レジスタと、以下のいずれかを戻す必要があります。

- **now** が match を超えていない
- コンペアマッチ割込みのフラグがすでに保留されている（割込みフラグが保留されている場合、**now** がすでに match や **old** を経過しているかどうかは関係ありません。これは、`osAdvanceCounter_<CounterID>` を処理するユーザー定義のアドバンスドカウンタドライバのコードは、現在の時間を認識できるためです。

関数内では、最初に **match** 値を比較レジスタに書き込みます。

もし **now** が **match** と **old** の間である場合、つまり **old** - **match** > **now** - **match** である場合、**now** はすでに **match** を経過しています。この場合、戻る前に割込みフラグをセットして保留状態にしておく必要があります。

もし **now** が **match** と **old** の間でない場合、**match** と **old** がすでに経過していれば、戻る前に割込みフラグをセットします。これは **now** - **old** < **old** - **now** [1] で判断できます。

```
Set_Counter1(TickType Match)
{
   TickType Old = (TickType)COMPARE;
   TickType Now = (TickType)COUNT;

   /* Updata COMPARE with new Match */
   COMPARE = Match;

   if ((Old-Match > Now-Match)
      || (Now-Old < Old-Now))
   {
      SET_INTERRUPT_PENDING();
   }
}
```

State

`State_Advanced()` は、カウンタまたはスケジュールが稼働している間のみコールされます。この関数では、最初に、次のマッチがすでに発生しているかどうか（つまり、割込みが保留されているかどうか、これはすべてのコールバック関数が OS レベルで実行されていて、ISR が現在実行中のタスクをプリエンプトすることがない、という条件に基づきます）をチェックします。この条件に該当しない場合、次の満了時までの残り時間も必要です。

---

1. カウンタ最大値（modulus）が m の場合、この評価方法は **now** - **old** < m/2 となります。

```
OS_CALLBACK(void) State_Advanced {
                      ScheduleStatusRefType State)
{
   State.expiry = OUTPUT_COMPARE - COUNTER;
   if (interrupt_pending())   {
      State.status =
         OS_STATUS_PENDING | OS_STATUS_RUNNING;
   }  else {
      State.status = OS_STATUS_RUNNING;
   }
}
```

重要

満了値（'expiry value'：残り時間）は、割込みが保留されているかどうかのチェックの前に計算します。これは、このチェックの後、満了値がまだ計算されていない状態で割込みが保留状態となると、不正な値が算出されていまうためです。

Now

`Now_Advanced()` は、フリーランニングカウンタを読み取って現在の時間に基づくカウンタ値を提供するものです。

```
OS_CALLBACK(TickType) Now_Advanced(void)
{
   return (TickType)COUNTER;
}
```

重要

8 ビットデバイスのカウンタを読み取る場合は注意が必要です。つまり上位バイトと下位バイトを正しい順番で読み込んでラッチしてからカウンタをリリースしてください。比較値を書き込む際にも同様の注意が必要です。

Cancel

`Cancel()` は、割込みが発生しないようにする関数です。ハードウェアに応じて処理は異なりますが、一般的にはカウンタデバイスによる割込み発生をディセーブルにします。

```
OS_CALLBACK(TickType) Cancel_Advanced(void)
{
   disable_interrupt();
}
```

14.2.2 割込みハンドラ

Simple（シンプルハンドラ）

最も単純なケースにおいては、実行すべき処理は、割込みをクリアして所定の `Advance()` をコールし、もしカウンタまたはスケジュールがまだ稼働中の場合、比較ポイントを次の満了ポイントに進めるだけです。ただしここでは、ハンドラのレイテンシが、駆動されるカウンタ／スケジュール内で最も短いディレイよりも小さいことが前提となるため、新しい比較ポイントは必ずカウンタの前方（つまり未来のタイミング）の値となります。

```
#include "Advanced_Driver.h"
ISR(Advanced_Driver)
{
   ScheduleTableStatusType State;
   dismiss_interrupt();
   Advance(&CurrentState);
   if (State.status & OS_STATUS_RUNNING) {
      OUTPUT_COMPARE += State.expirery;
}
```

ここでは、出力比較ポイントが常にタイマの前方にあることが必要です。もしも満了時間がハンドラレスポンスよりも短いとその条件が満たされなくなり、次のポイントが処理されるまでにカウンタの 1 ラップ分の余計なカウントが必要になってしまいます。シンプルハンドラを安全に使用できるかを検証するには、RTA-OSEK Planner を使用してスケジューラビリティ分析を行ってください。ユーザーアプリケーションがスケジューラブルになるには、シンプルハンドラが次回の呼び出し前に処理が完了している必要があります。

Retriggering（再トリガハンドラ）

各ポイントが非常に接近していて、次のポイントに達するまでにハンドラが比較値を進める（'Advance' する）ことができない場合、ハンドラ内で、すでに次のポイントに達しているケースを考慮する必要があります。

下記のコード例では、クリアシーケンス実行中に発行された割込みが不用意にクリアされてしまうの防ぐため、出力比較タイマにハードウェアインターロックを使用しています。ここでは、ステータスレジスタを読み込んだ後、割込みビットをクリアするためのビットパターンをステータスレジスタに書き込むことによって、割込みをクリアするインターロックが適用されることを前提としています。この例においては以下の 2 つの関数がインターロック機能を構成しています。

- `prepare_interrupt_clear()`
- `commit_interrupt_clear()`

ドライバが実行されている間、比較ポイントが進められ（ラップ時、つまり現時点が最大値である場合は、0 になります）、クリアシーケンスの最初の部分（ステータスレジスタの読み込み）が実行されます。その後、新しい比較ポイントが、現在の時間よりも先（= 未来）の時間になっているかがチェックされ、もしカウンタが比較値まで進んだときに割込みが発生しない（つまり、まだ次のポイントに達していない）ことが判断された場合、割込みクリアシーケンスの残りの部分（ステータスレジスタにクリアビットを書き込む）が最後まで実行されます。もしこのチェックで上記のような結果とならない場合（つまり、現時点ですでに次のポイントに達している場合）、割込みがすでに保留されているため、ハンドラは次のポイントの処理を「再トリガ」されることになります。この場合は、2 段階の割込みクリアシーケンスを実行することが必要です。これは、テストを実行してから割込みをクリアするまでの間にカウンタが次のポイントに達してしまう、という競合状態（'race condition'）を避けるためです。ただしこれによって次のポイントの割込みがクリアされてしまいます。そのためハードウェアに要求されることは、1 回目のシーケンスの後に割込みが発生した場合、2 番目のシーケンスでは実際には割込みをクリアしない、ということです。

割込みがソフトウェアによって再発行されるようなデバイスについてもこれと同様のアプローチを行えますが、その場合は、割込みはハンドラにエントリする際にクリアでき、次のポイントにすでに達していても割込みが再発行されるため、競合状態は発生しません。ただしこれは、ハードウェアがすでに発行した割込みをソフトウェアが再発行することによって問題が発生しない場合に限ります。

```
ISR(Advanced_Driver)
{
   ScheduleTableStatusType State;
   TickType remaining_ticks;
   osUnit16Type clear_tmp;
   Advance(&State);
   if (State.status & OS_STATUS_RUNNING) {
      OUTPUT_COMPARE += State.expirery;
      clear_tmp = prepare_interrupt_clear();
      remaining_ticks = OUTPUT_COMPARE - COUNTER;
      if ((State.expiry == 0u) ||
            ((remaining_ticks != 0u) &&
             (remaining_ticks <= State.expiry))) {
         commit_interrupt_clear(clear_tmp);
      }
   }
}
```

重要

ハードウェアによっては、出力比較ハードウェアに各割込みを設定するために比較レジスタに書き込みを行う必要のあるものがあります。そのような場合は、上記の例のようにコードを階層化し、比較値が 0 の場合は比較レジスタに前回の比較値を書き込みます。

Looping（ループハンドラ）

ここでは、プログラム可能な出力比較機能を持つカウンタ（最大値は `TickType`）用の汎用的なループ構造を持つ ISR について説明します。

割込みハンドラは、発生した割り込みをクリアした後にループに入ります。ループ内では、ポイントを処理した後に他に処理すべきポイントがあるかをチェックします。このチェックには 4 つの exit 条件があり、以下の順で評価されます。

1. Exit 1 ー カウンタ／スケジュールが停止していて、これ以上アクションを行う必要がない場合に実行されます。カウンタ／スケジュールが停止していない場合は、要求された数のチックだけ比較ポイントを先に進めます（現時点が最大値の場合、ゼロになります）。そして、次のポイントに達した時点で割込みが発生するかどうかをチェックします。
2. Exit2 ー 満了値（'expiry value'、満了までの残り時間）をチェックして、次のポイントを処理する前にラップ（カウントがゼロに戻る）が発生するかどうかを調べます。この場合はコンペアマッチ値を変更する必要はありません。満了値がゼロであるということは、新しい比較ポイントがタイマの現在の値よりも大きくことが確かである、ということで、そのポイントに達した時点で割込みが発生することが保証されているためです。ここで exit しておけば、もしもラップが必要なタイミングであり、かつカウンタがまだ進んでいない時点でイベントが発生した場合、以降のチェックによってカウンタと比較ポイントとのマッチが不正に判定されることを防ぐことができます。なおこれは、カウンタと比較ポイントが連続してマッチする際、割込みは再発行されず、最初のマッチにおいてのみ発行される、という前提条件に基づきます。もしこれに該当しないシステムの場合、ハンドラがこの状態において exit するのを防ぐため、満了値が 0 にならないようにするなどの処理が必要となります。
3. Exit3 ー 現在のタイマ値がまだ新しい比較ポイントに達していない場合に実行されます。このことは、次の割込みまでの時間（OUTPUT_COMPARE - COUNTER）が次のポイントまでのディレイより小さいかどうかで判定されます。ここでは、`TickType` へのキャストを行い、カウンタ最大値（'modulus'）における挙動が正しく行われるようにする必要があり、そのためにはカウンタの最大値が同じ `TickType` である必要があります。任意の最大値を扱う方法についての詳細は、カウンタジュールについての注釈を参照してください。カウンタが満了値よりも小さい値でカウントアップされる場合、正しいタイミングで割込みが発生し、ハンドラは exit することができますが、そうでない場合は新しいポイントが捕捉できない可能性があります。
4. Exit4 ー 新しい比較ポイントのセット処理と、そのポイントがカウンタの前方（未来）の値であるかのチェックを行う処理との間で発生する可能性がある競合状態に対応するためのものです。これはつまり、Exit3 のチェックが行われる前にカウンタのカウントが進んでしまう可能性を考慮する

ものです。もし Exit3 が実行されない場合、次のポイントはすでに満了していますが、その時点でもしも割込みが保留されていれば、ポイントの満了はハードウェアによって認識されていることになるため、ハンドラは exit でき、保留されている割込みの処理のために再度呼び出されることができます（まだ割込みが保留されていない状態では exit することはできません）。なお、このしくみにおいては、Exit3 が実行されなかったときに割込みが保留されているかどうかは問題になりません。それは、カウンタは確実に満了値まで進んでいて、保留された割込み、またはルーピングの結果のいずれかによって、次のポイントが処理されるためです。

もしもいずれの exit も実行されなかった場合、それは次のポイントが満了した（またはすでに満了期限が切れている）ことを意味するのいで、ループは要求された満了コールを実行し、次のポイントについての exit チェックを繰り返します。

なお、次のポイントは未来の時点にあるため、このハンドラは一般的に `Advance()` コールを 1 回だけ行うことが要求されます。これは、各ポイントの処理がそれぞれ別々のトリガによって処理されることによってワーストケースビヘイビアとなるため、ハンドラはできる限り速やかに対応することが必要となるためです。

```
#include "Advanced_Driver.h"
ISR(Advanced_Driver)
{
   ScheduleTableStatusType State;
   TickType remaining_ticks;
   dismiss_interrupt();
   while(1)   {
      Advance(&State);
      if (!(State.status & OS_STATUS_RUNNING)) {
         return; /* exit 1: activator stopped */
      }
      OUTPUT_COMPARE += State.expiry;
      if (State.expiry == 0u) {
         return; /* exit 2: full wrap */
      remaining_ticks = OUTPUT_COMPARE - COUNTER;
      if ((remaining_ticks != 0u) &&
            (remaining_ticks <= State.expiry)) {
         return; /* exit 3: compare point is in the future */
      }
      if (interrupt_pending()) {
         return; /* exit 4: interrupt pending */
      }
   }
}
```

重要

ここでは、使用するコンペアマッチハードウェアの割込み挙動を十分に理解しておく必要があります。現在のカウンタ値と同じ値が比較値として設定された際の挙動は、3 通りが考えられます。つまり、値が設定された時点で割込みを発行するか、カウンタ値が比較値を超えたときに割込みを発行するか、またはカウントがラップ（0 に戻って 1 周する）して比較値と等しくなった時点で割込みを発行するかのいずれかです。

上記のコード例の Exit 3 のテストは、コンペアマッチハードウェアハードウェアが 1 番目または 3 番目の動作を行うことを前提としています。2 番目の動作を行うハードウェアであれば、もし `remain_ticks` がゼロであった場合に exit する必要があります。これは、カウンタ値と比較値が等しくなった後に割込みが発行されるためです。

14.2.3 TickType よりデータ幅の小さいカウンタハードウェア

前項で概説されたドライバは、カウンタと比較レジスタが TickType と同じデータ幅を持ち、演算は符号なしの TickType のデータ幅で行われることが前提となっていますが、ハードウェアの中にはこの条件に該当しないものもあります。

そのようなハードウェアを使用する場合は、カウンタがある値（*m* - 1）になった後にゼロにラップ（戻る）すること、つまり TickType よりも小さい *m* というデータ幅を持っていることを前提とする必要があります。これによってドライバの構造は複雑になりますが、ハードウェアの挙動よってはそれが避けられず、また異なるシステム要件をサポートする際に必要となる可能性もあります。たとえば、タイマの最大値（'modulus'）が 50000 で 1 チックが 1ms の場合、チックドカウンタまたはスケジュールで使用される 50ms の割込みをオーバフローによって提供でき、またアドバンスドドライバに使用される出力比較割込みを提供できます。

このようなケースにおいては、新しい比較値を算出する計算と、比較値とカウンタ値との関係をチェックする計算を変更する必要があります。ここでは TickType の最大値を 2^{16} と仮定します。

もしも *m* が 2x（ここで x < 16）という数であれば、演算の最大値は、2x-1 との AND を行うことによって簡単に調整できます。データ幅が 8 ビットの場合は、以下のようにして新しい比較値をセットできます。

```
new_cmp = (old_cmp + ret.expiry) & 0xFF;
```

比較ポイントまでの残りチック数の計算も、同様に変更します。

しかし最大値が 2 のべき乗でない場合は、より複雑な計算が必要です。

新しい比較値を計算するには、最大値 2^{16} の TickType を用いて現在の比較値と新しい満了値（次の満了ポイントまでの残り時間）の合計が計算される際に、4 とおりの結果が得られることを考慮する必要があります。

1. 満了値がゼロになる ― ラップが発生しても比較値は変わりません。
2. 合計が古い比較値より大きく、*m* よりも小さい ― 加算によって正しい結果が得られます。
3. 合計が *m* より大きい ― 加算の結果について、*m* におけるラップの発生を考慮する必要があります。この場合は、*m* を差し引きますが、剰余演算子の使用は避けてください。
4. 合計が古い比較値より小さい ― 2^{16} においてラップが発生したことを表しているため、合計に（2^{16} ― *m*）を加えて *m* においてラップが発生した値となるように調整します。

もしも *m* が演算幅の 2 分の 1 より小さい（≦ 2^{16} × 1/2）場合、4 番目のケースは起こりえません。

新しい出力比較値がカウンタの前方（未来）の値にセットされたかどうかをチェックする際の状況としては、以下の 3 通りが考えられます。引き算によって演算幅 2^{16} のアンダーフローは発生しません。

1. 満了値（残り時間）がゼロ ― この場合、新しい比較ポイントが前方（未来）の値であると判断できます。ハンドラはカウンタの最大値未満で終了することが要求されます。
2. 新しい比較値がカウンタ値以上である ― 比較値からカウンタ値を引いて次のマッチが発生するまでの時間を求め、その値が所定の満了時間以下であるかどうかをチェックします。もしそれより大きい場合、次のポイントはすでに満了しています。
3. 新しい比較値がカウンタ値より小さい ― カウンタ値から比較値を差し引いた値を最大値から差し引くことによって、次のマッチが発生するまでの時間を求めます。つまり *m* ― (COUNTER ― OUTPUT_COMPARE) を行います。

コールバック関数 State_Advanced() 内で満了までの残り時間を計算する際も、同じアプローチが適用できます。

汎用的な「出力比較」ドライバに上記のメカニズムを追加すると、以下のようになります。

```
#include “Advanced_Driver.h”
/* Next line should result in a constant being
   Substituted. We assume that the expression will
   be evaluated at compile time, avoiding modulus
   overflow at run time */

#define CMP_ADJUST ((TickType)65536u - m)
```

```
/* Where m is the timebase modulus */

ISR(Advanced_Driver){
  ScheduleTableStatusType State;
  TickType counter_cache;
  TickType remaining_ticks;
  TickType new_cmp;
  dismiss_interrupt();
  while(1) {
    AdvanceSchedule(&State);
    if (!(State.status & OS_STATUS_RUNNING)) {
      return; /* exit 1: activator stopped */
    }
    if (State.expiry == 0u) {
      /* OUTPUT_COMPARE = OUTPUT_COMPARE if
       * needed to arm next interrupt */
      return; /* exit 2: full wrap */
    }
    new_cmp = OUTPUT_COMPARE + State.expiry;
    if (new_cmp > OUTPUT_COMPARE) {
      if (new_cmp >= m) {
        new_cmp -= m;
      }
    } else {
      new_cmp += (CMP_ADJUST);
    }
    OUTPUT_COMPARE = new_cmp;
    counter_cache = COUNTER;
    if (new_cmp >= counter_cache) {
      remaining_ticks = new_cmp - counter_cache;
    } else {
      remaining_ticks =
        m - (counter_cache?new_cmp);
    }
    if ((remaining_ticks != 0u) &&
        (remaining_ticks <= State.expiry)) {
      return; /* exit 3: compare in the future */
    }
    if (interrupt_pending()) {
      return; /* exit 4: interrupt pending */
    }
  }
}
```

14.2.4 TickType よりデータ幅の大きなカウンタハードウェア

次に、ハードウェアのカウンタの最大幅が TickType より大きい場合を考えてみます。このようなカウンタを使用して、最大位置が 2^{16} である TickType に対応する動作を容易に実現できます。

ここでは最大値が 2 のべき乗（例：32 ビットカウンタ）である場合のみを想定します。32 ビットカウンタの場合、下位の 16 ビットは正しい動作となりますが、オーバーフローの影響を考慮する必要があります。

割込みハンドラ内で比較値を進めた際、コンペアレジスタの下位の 16 ビットから発生したオーバーフローは、レジスタの残りの部分に反映されなければなりません。さらに、満了値が 0 の場合、比較値に 2^{16} を加える必要があります。これより大きな値が比較値に加えられることはないので、カウンタと比較レジスタの下位の 16 ビットを比較するだけで、タイマが比較ポイントを超えているかどうかをチェックすることができます。

コールバック関数 `Set_Advanced()` においては、比較ポイントを適切にセットして、カウンタの下位 16 ビットの値が `Set_Advanced()` に渡された引数と（今回ではなく）次に等しくなるときに比較ポインタとカウンタのマッチが起こるようにする必要があります。これは以下のようにして行います。なお以下の例では、カウンタ値と比較値は 32 ビット符号なしの値になっています。

```
OS_CALLBACK(void) Set_Advanced(TickType Match)
{
  osUint32Tye to_compare;
  disable_interrupt();
  disable_compare();
  dismiss_interrupt();
  OUTPUT_COMPARE =
    (COUNTER & 0xFFFF0000ul) | Match;
  to_compare = OUTPUT_COMPARE - COUNTER;
  if ((to_compare == 0ul) ||
      (to_compare >= 0x10000ul) {
    if(!(interrupt_pending())) {
      OUTPUT_COMPARE += 0x10000ul;
      to_compare = OUTPUT_COMPARE - COUNTER;
      if ((to_compare == 0ul) ||
          (to_compare >= 0x10000ul)){
        if(!(interrupt_pending())) {
          OUTPUT_COMPARE += 0x10000ul;
        }
      }
    }
  }
  enable_interrupt();
}
```

この処理は、アトミックに、つまり中断されることなく実行される必要があるため、ハードウェアデバイスからの割込みをディセーブルにした状態で行います。最初に保留されている割込みをクリアしますが、これは、比較を無効にした後に行う必要があります。つまり、比較ポイントを適切にセットして、保留されている割込みが新しい比較値とのマッチによってのみ処理されるようにする必要があります。続いて、比較レジスタにカウンタ値をセットし、その下位 16 ビットを、渡された引数に置き換えます。

もしも比較ポイントが現在よりも 2^{16} チック未満だけ先（未来）の時点にセットされていれば、それは正しくセットされていることになります。もしも割込みが保留されていれば、比較ポイントにすでに到達しているはずなので、その割込みを処理することができます。そうでない場合、比較ポイントを 2^{16} だけ進めます。そしてその後、再度チェックを行って、新しい比較ポイントがセットされる前にカウンタがその値に達してしまう、という競合条件を回避します。チェックを 2 回行うことは、`Set_Advanced()` の処理が 2^{16} タイマチック以内に完了する、という前提条件において有効です。

このコードは、比較値がカウンタと同じ値にセットされたときに割込みが保留されるかどうかが不明である、ということを前提にしています。2 つの値が同じになったとき、またはその後に割込みが保留される、ということがわかっている場合は、`to_compare` のゼロチェックは削除する必要があります。

なおこの関数は、アプリケーションの挙動をより理解することによって大幅な簡素化が可能です。たとえば、起動時にカウンタがゼロになり、起動後、Match チック以内に起動ドライバが 1 回のみ起動されるのであれば、比較値に Match という値をセットすることができます。

重要

最大値が 2^{16} の挙動は、2 のべき乗ではない最大値を持つカウンタの下位 16 ビットでは表せません。タイマがラップする前の最後のインターバルは「カウンタ最大値 MOD 2^{16}」チックとなります。

14.3 フリーランニングカウンタとインターバルタイマ

上述のカウンタのコンペアマッチハンドラは、時間のずれのない正確な起動ドライバを実現するものですが、ターゲットプラットフォームによっては、そのようなカウンタハードウェアが搭載されていない場合があります。

しかしそのような場合でも、独立したフリーランニングカウンタがあれば、ダウンカウンタを使用することによって時間のずれを回避できます。フリーランニングカウンタを使用すれば、ずれのない時間参照を行え、次のポイントに達した時点で割込みを発行させることができます。ダウンカウンタのセット時の遅れにより、各満了ポイントにおいていくらかのジッタ（遅れ）が生じる可能性はありますが、これは積算されません。このようなジッタは割込みハンドリングと同じ方法で処理できます。本項では、変数として扱える 2 つのレジスタ（COUNTER および DOWN_COUNTER）を持つダウンカウンタについて説明します。前の例と同様、両レジスタは TickType と同じ幅のレジスタで、扱える値は TickType サイズの符号なし整数です。

14.3.1 コールバック関数

次のマッチ値はソフトウェア内で保持され、次の割込みまでのダウンカウント値を計算する際に使用されます。

```
TickType   next_match;
```

Set

```
OS_CALLBACK(void) Set_Advanced(TickType Match)
{
  /* Record value at which expire is due */
  next_match = Match;
  disable_compare();
  dismiss_interrupt();
  /* set up interrupt when counter reaches match
     value */
  DOWN_COUNTER = next_match - COUNTER;
  enable_interrupt();
}
```

State

以下に示す State_Advanced() コールは、Status.expiry に DOWN_COUNTER の値をセットしてリターンします。ダウンカウンタのセット時にいくらかのジッタが発生した場合、次の満了が処理されるタイミングよりも、満了が通知されるタイミングに影響します。しかし最大値が TickType と異なり、かつ追加の計算が必要のないカウンタの場合は、以下のようにすることができます。

```
OS_CALLBACK(void) State_Advanced(
                     ScheduleStatusRefType State)
{
  State.expiry = next_match - COUNTER;
  if (interrupt_pending()) {
    State.status =
```

```
        OS_STATUS_RUNNING | OS_STATUS_PENDING;
    } else {
      State.status = OS_STATUS_RUNNING;
    }
    return;
  }
```

Cancel

キャンセル関数は以下のように作成してください。

```
OS_CALLBACK(void) Cancel_Advanced(void)
{
  disable_interrupt();
}
```

14.3.2 ISR

```
ISR(Advanced_Driver)
{
  ScheduleStatusType State;
  TickType remaining_ticks;
  dismiss_interrupt();
  while(1) {
    AdvanceSchedule(&State);
    if (!(State.status & OS_STATUS_RUNNING)) {
      return; /* exit 1: activator stopped */
    }
    next_match += State.expiry;
    /* also subtract adjustment for */
    /* delay before COUNTER is set? */
    remaining_ticks = next_match - COUNTER;
    if (State.expiry == 0u) {
      DOWN_COUNTER = remaining_ticks;
      return; /* exit 2: full wrap */
    }
    if ((remaining_ticks!= 0u) &&
        (remaining_ticks <= State.expiry)) {
      DOWN_COUNTER = remaining_ticks;
      return; /* exit 3:
                 counter set for next expire */
    }
    /* assume we only get an interrupt due to
       setting the counter and we only set the
       counter when we are going to exit so no
       need to test for pending interrupt */
  }
}
```

これはループ構造の ISR です。1 回の関数呼び出しで 1 つのポイントを処理するのではなく、再トリガ形式のように、処理すべき満了ポイントがなくなるまでループを続けます。

Exit 2 の部分は、カウンタをゼロにセットすると、チックの 1 回のラップの後に割込みが発生することを前提としています。

14.4 「ゼロカウントでマッチする」ダウンカウンタの使用

ハードウェアによってはフリーランニングカウンタを搭載していないものがあり、また搭載していても、それをアドバンスドドライバ用に使用できない場合もあります。

このような場合は、単純なインターバルカウンタを使用する必要があります。以下の例では 16 ビットのデクリメントカウンタを使用していて、0 に到達した時点で割込みが発行され、デクリメントが続行されます。そのため、新しいカウントダウンの開始ポイントは、カウンタ値に満了時間を加える（最大値が 2^{16} の演算を想定しています）ことによって決まります。カウンタを更新する際の時間のずれは最小に留める必要がありますが、更新中に割込みが発生するのを防ぎ、さらに更新にかかる時間を調整値としてカウンタと `next_match` に加えることにより、カウンタの更新ごとの遅れを 1 チックまで低減できる可能性があります。なおこのカウンタは、更新処理と非同期に動作し、不確実性があることが前提となります。そこで `counter_adjust` を用いて 'now' 値を算出できるようにします。この値は、`next_match` からカウンタ値を差し引くことによって求めます。ただし、カウンタの更新と `counter_adjust` の更新を行う間は、'now' 値を取得するコールが実行されないようにして、不正な値が取得されるのを防ぐ必要があります。

ドライバが稼動していないときは、ダウンカウンタはフリーランを行っているものと仮定されます。起動時から、ゼロからのカウントダウンを行い、（0 - カウンタ値）が 'now' 値となります。また `counter_adjust` は常に、次にダウンカウンタの値が 0 になるときのフリーランニングカウンタの実際のチック値を保持しています。

14.4.1 コールバック関数

Set

```
TickType counter_adjust = 0;
OS_CALLBACK(void) Set_Advanced(TickType Match)
{
  TickType AdjustedMatch;
  AdjustedMatch =
    Match - (counter_adjust - DOWN_COUNTER);
  /* dismiss interrupt in a way that avoids race
     conditions */
  disable_compare();
  dismiss_interrupt();
  DOWN_COUNTER = AdjustedMatch;
  counter_adjust += AdjustedMatch;
  enable_interrupt();
}
```

前出の競合条件はここでも存在します。割込みがクリアされてからダウンカウンタがセットされると、その間に割込みが発生する恐れがあります。もしダウンカウンタがセットされた後に割込みがセットされれば、必要な割込みが破棄されることによって若干の遅れが生じる可能性があります。ハードウェアの保護がない場合、これを回避するには、上記の例のように `disable_compare()` 関数でカウンタを最大値 -1 にセットし、さらに、`AdjustedMatch` の値を決定してからカウンタをセットするまでの間に割り込みをクリアします。

State

`State_Advanced()` は前述と同様ですが、満了時間はダウンカウンタから直接読み込めます。

```
OS_CALLBACK(void) State_Advanced(
                     ScheduleStatusRefType State)
{
  State.expiry = DOWN_COUNTER;
  if (interrupt_pending()) {
    State.status =
```

```
        OS_STATUS_PENDING | OS_STATUS_RUNNING;
    } else {
      State.status = OS_STATUS_RUNNING;
    }
  }
```

Now

以下のようにして正しい 'now' の値を求めます。

```
OS_CALLBACK(TickType) Now_Advanced(void)
{
  return (counter_adjust ? DOWN_COUNTER);
  /* counter_adjust is still correct adjustment
   * as counter runs to and through 0 */
}
```

Cancel

ドライバのキャンセルは以下のように行います。

```
OS_CALLBACK(void) Cancel_Advanced(void)
{
  disable_interrupt();
}
```

14.4.2 割込みハンドラ

```
#include “Advanced_Driver.h”
ISR(Advanced_Driver)
{
  ScheduleStatusType State;
  TickType counter_cache;
  dismiss_interrupt();
  while(1) {
    Advance(&State);
    if (!(State.status & OS_STATUS_RUNNING)) {
      return; /* exit 1: activator stopped */
    }
    if (State.expiry == 0u) {
      return; /* exit 2: full wrap */
    }
    counter_cache = COUNTER + State.expiry;
    COUNTER = counter_cache;
    counter_adjust += State.expiry;
    if ((counter_cache != 0u) &&
        (counter_cache <= State.expiry)) {
      return; /* exit 3:
               * next time point has not yet
               * been reached */
    }
    if (interrupt_pending()) {
      return; /* exit 4: interrupt pending */
    }
  }
```

```
}
```

Exit 3 では、カウンタがゼロに到達した後ではなく、到達と同時に割込みが発行されることを前提としていますが、ゼロにセットされた場合、カウンタがゼロになるとポイントが満了してルーピングハンドラまたは再トリガハンドラによって処理されます)。2 つの比較演算の間でカウンタ値が変化した場合、競合状態状態（'race condition'）が生じてしまう恐れがあるため、`counter_cache` を使用して比較を行っています。

カウンタがゼロにセットされたときの割込み挙動がわかっている場合、Exit 4 とそのチェックを省略してコードを簡素化しすることができます。これは、`counter_cache` がゼロのときの割込みステータスがわかっているからです。カウンタがゼロにセットされたときに割込みが発行されない、ということがわかっている場合は、コードを削除するだけですが、もしカウンタがゼロにセットされたときに割込みが発行されるのであれば、Exit 3 で `counter_cache` が満了値未満であるかどうかというチェックのみを行います。ゼロであれば割込みは保留され、次のイベントが処理されます。

システムクロックが非常に高速な場合（クロック速度がプロセッサ速度と同じ、またはそれより速い場合)、カウンタの読み込みと新しい値のセットとの間に発生するチック数を、オフセットとしてカウンタ値に加えてる必要があります。またどのような場合も、ダウンカンタをセットする際の最大 1 チックのずれは避けられません。複数の割込み優先レベルを持つプラットフォームの場合、`counter` の読み込みと書き込みを行う際は、すべての割込みをディセーブルにすることにより、2 つの処理の間に割込みが発生してカウントの大きなずれが発生するのを避けるようにしてください。

14.5 インターバルタイマで駆動されるソフトウェアカウンタ

周期的なインターバルタイマ（またはイベント割込みソース）を使用すると、カウンタとコンペア値をソフトウェアで合成することができます。この場合、カウンタと比較値がハンドラでのみ変更されるため、競合状態を考慮する必要がなくなりますが、このようなタイプのハンドラは 1 チックごとに 1 回の割込みとなるため、あまり実用的ではありません。これよりもチックドカウンタの使用をお勧めします。

14.6 まとめ

- 各アドバンスドカウンタおよびアドバンスドスケジュールごとに、ユーザーがアドバンストドライバを提供する必要があります。
- ドライバインターフェースは以下のものからなります。
  - RTA-OSEK に処理実行を通知するためのカテゴリ 2 割込み
  - RTA-OSEK がカウンタ／スケジュールを制御するために使用するコールバック関数
- 可能であれば、コンペアマッチハードウェアと簡単な割込みハンドラで構成されるフリーランニングカウンタを使用します。

15 起動とシャットダウン

他のオペレーティングシステムの中にはハードウェアを制御するものもありますが、RTA-OSEK コンポーネントはそれらのものとは異なります。

ユーザーが API 関数を使用して明示的にオペレーティングシステムを起動するまでは、オペレーティングシステムは稼働せず、この間は、ユーザーは一切の制約なしにハードウェアを自由に使用することができます。

RTA-OSEK コンポーネントは様々な**アプリケーションモード**で起動することができます。「モード」はアプリケーション機能の全体またはその一部となるもので、アプリケーションのある特定の機能を実行するものです。アプリケーションモードの詳細については、15.2.1 項で説明します。

15.1 システムリセットから `StartOS()` コールまでの処理

この項ではまず、組み込みプロセッサに電源が投入された後、StarOS() によって RTA-OSEK コンポーネントとユーザーアプリケーションが起動するまでの間に実行されるべき処理について説明します。ここで行われる処理はプロセッサによって多少異なりますが、基本的な内容は同じです。各ターゲット用のリファレンスガイドをご参照の上、本項に書かれている内容を反映させた処理を記述してください。

15.1.1 パワーオンリセットから `main()` までの処理

組み込みプロセッサに対して電源投入またはリセットが行われた際のプロセッサの起動手順は、プロセッサに応じて 2 通りの方法があります。

1 つはメモリの固定位置からコードを実行するもので、もう 1 つはメモリの固定位置からアドレスを読み込み、このアドレスからコードを実行するものです。後者の場合、実行する最初の命令のアドレスを格納したメモリ上の固定位置は、「リセットベクタ」と呼ばれ、プロセッサによってはそれが割込みベクタテーブル内に含まれる場合もあります。

一般的に、量産用の実行環境においては、このリセットベクタや最初に実行される命令は、さまざまなタイプの不揮発性メモリ内に配置されますが、開発時の環境においては、通常、プログラミングが容易な RAM に配置されます。また、評価用ボード（EVB）の場合は、リセットベクタや最初の命令を EEPROM と RAM のどちらに配置するかを選択できるスイッチやジャンパが装備されています。

電源投入またはリセットが行われて最初の命令が実行されるまでのことは、「リセットからの抜け出し（'coming out of reset'）と呼ばれます。プロセッサは、リセットから抜け出すと、通常、以下のことを行います。

- 割込みをディセーブルにします。
- スーパーバイザーモードに切り替わります（プロセッサがこのモードをサポートしている場合）。つまり、すべての命令を実行でき、どのアドレスにアクセスしても例外が発生することはなく、すべてのメモリと I/O に対する保護が解除されます。
- シングルチップモードに切り替わります（プロセッサがこのモードをサポートしている場合）。つまり、外部メモリは使用できず、外部バスも無効となります。

プロセッサがリセットから抜け出した後は、任意のユーザーコードを実行することができますが、C 言語のような高級言語が使用される場合は、通常、この「ブートストラップコード」（'bootstrap code'）は、PC とともに供給されます。

ブートストラップコードは、コンパイラ製造元から供給されるオブジェクトモジュールまたはライブラリ内に含まれています。ブートストラップコードには以下の 2 つのキーアイテムが含まれます。

1. 基本的なプロセッサ設定（例：バス設定を行って内部 RAM へのアクセスを有効にします）
2. C 言語で記述されたスタートアップコード（構造体の初期化、メモリクリア、スタックポインタのセットなどが行われます）の呼び出し

オブジェクトモジュールやライブラリ、またはリンカ設定ファイルに書かれた命令文により、ブートストラップコードは（必要に応じてリセットベクタに書かれる値も）、メモリの所定の位置に正しく配置されます。

C 言語のスタートアップコード

通常、C 言語で記述されたスタートアップコードは、コンパイラの提供元、またはプラットフォームによってはそれを少し変更したものが ETAS から提供されます。スタートアップコードは通常、"crt0" または "startup" といった名前のオブジェクトモジュールとして提供され、マップファイルを見ると、"_start" や "_main" といった名前のシンボルとして見つけることができます。またこのモジュールのソースコードも通常、ユーザーに提供されます。

LiveDevices が提供するいくつかのプラットフォームについては、RTA-OSEK アプリケーションで使用するための標準スタートアップコードが何種類か提供されています。この使用方法は、『RTA-OSEK バインディングマニュアル』と RTA-OSEK に添付されているサンプルに詳しく示されています。

スタートアップコードは、C 言語環境の初期化、つまり、スタックポインタのセットや malloc() 用のヒープメモリの確保、またグローバル変数の初期化（値を ROM から RAM にコピーする）、といった処理を行い、最後にアプリケーションのスタートアップコードを呼び出します。

15.1.2 アプリケーションのスタートアップコード

アプリケーションのスタートアップコードは、"main()" 関数、または RTA-OSEK アプリケーションの場合は "OS_MAIN()" マクロで宣言される関数です。RTA-OSEK アプリケーションにおいて、アプリケーションのスタートアップ関数は以下の 3 つの処理を行う必要があります。

- ターゲットハードウェアを初期化し、RTA-OSEK とアプリケーションを実行できる状態にする。
- StartOS() を呼び出し、RTA-OSEK コンポーネントの実行を開始する。
- アイドルタスクの処理を実行する。

以下に、RTA-OSEK アプリケーションのスタートアップコードの例を示します。

```
OS_MAIN()
/* note that we use this macro for portability
   rather than main() as some compilers expect strange
   declarations of main() */
{
   init_target();

   StartOS(OSDEFAULTAPPMODE);

   /* Code that makes up the idle task */
   /* functionality. */

   /* The idle task must never terminate so if */
   /* there is no idle task functionality then */
   /* use something like: */

   for (;;) { /* Do nothing. */ }
}
```

コード例 15-1 メイン処理とアイドルタスクの例

StartOS() によって、RTA-OSEK コンポーネントが実行状態となります。カーネルが実行状態となると、発生した割り込みに応じて ISR が呼び出され、タスクがスケジュールされます。StartOS() が終了してリターンしてくると、アプリケーションのスタートアップコードはアイドルタスクとしての処理を実行します。アイドルタスクはターミネートすることはできないので、特に処理が必要ない場合は、無限ループを記述しておきます。

上記の例に含まれる init_target() 関数は、ターゲットハードウェアの初期化を行うもので、ユーザーが用意するものです。それ以降の部分では、アプリケーションや RTA-OSEK コンポーネントを実行できる状態するために必要なターゲットハードウェアの初期化処理などを行います。処理内容はターゲットごとに異なるので、ターゲット固有の『RTA-OSEK バインディングマニュアル』とプロセッサのリファレンスガイドをよくお読みください。

スタートアップフック（'Startup Hook'）についての注意事項

RTA-OSEK GUI の“Application / OS configuration”で、`StartOS()` 関数内において、RTA-OSEK コンポーネントを初期化した後にスタートアップフックを呼び出すように設定することができますが、この呼び出しは、RTA-OSEK コンポーネントが割込みレベルをユーザーレベルに下げてタスクのスケジュールを行う前に行わなければなりません。この機能は、ターゲット初期化の最終段階の処理を行うために使用されます。詳しくは、以下の「割込みの設定」の項を参照してください。スタートアップフックは、`StartupHook()` という関数として提供されるアプリケーションです。

メモリの設定

一般的に、メモリの設定は、アプリケーションのスタートアップコードが実行される前に実行されるブートストラップコードによって行われます。複雑な組み込みプロセッサの場合、ブートストラップのコードによって行われるメモリ設定は、アプリケーションに必要な設定とは異なる場合があります。たとえば、プロセッサが内部 RAM と外部メモリバスを持っている場合、ブートストラップは通常、プロセッサが内部 RAM を使用するように設定します。しかし実際のアプリケーションが外部メモリバス上の RAM を使用する必要がある場合、プロセッサが外部 RAM を使用するように設定しなければなりません。アクセス先を変更するには、プロセッサに応じて、バンク選択を行ったりマスクレジスタを使用します。

周辺機器の設定

一般的な組み込みアプリケーションは、組み込みプロセッサの一部である周辺デバイスや、I/O やメモリバス経由でアタッチされた周辺デバイスを利用します。これには、CAN コントローラ、イーサネットコントローラ、UART などがあります。これらのデバイスの設定は、RTA-OSEK コンポーネントが起動する前、つまりアプリケーションコードがプリエンプトされることなく、割込みを完全に制御できる状態において行うようにしてください。

タイマの設定

一般的な組み込みアプリケーションは、ハードウェアタイマを利用します。通常、タイマは、チック（'tick'）を行って所定の周波数で割込みを生成するため用いられます。タイマ割込みが発生すると、その割込みに割り当てられた ISR がタスクを直接起動するか、または OSEK カウンタのチック（`Tick_xxxx()` の呼び出し、`xxxx` はカウンタ名）を行います。

ハードウェアタイマの設定方法はタイマの仕様に応じて異なりますが、ここでは 2 通りの方式があります。1 つ目は、カウントレジスタを 0 にセットし、限界値（'bound'）レジスタにカウンタレジスタの最大カウント値をセットする方法です。この場合、カウントレジスタがプロセッサによって所定の周期でカウントアップされ、その値が限界値レジスタの値と等しくなった時に割込みが発生し、カウントレジスタが 0 にリセットされます。2 番目の方法は、カウントレジスタに割込みが生成されるまでのチック数がセットされます。プロセッサが所定の周波数でこのカウントレジスタをカウントダウンし、その値が 0 になった時点で割込みが発生します。カウンタレジスタの再設定は、通常、各割込みに割り当てられた ISR が行います。

タイマのカウントが行われる周波数は、アプリケーションで任意に設定できます。OSEK カウンタは、定義されたとおりの周波数でチックされる必要があります。

RTA-OSEK アプリケーションの拡張ビルドおよびタイミングビルドにおいては、`GetStopwatch()` というコールバック関数が必要です。この関数は、RTA-OSEK GUI の“Target / Timing Data”で設定され、周波数でインクリメントされるフリーランニングタイマの値を返します。詳しくは、『RTA-OSEK リファレンスガイド』の「実行時間」の項を参照してください。

また、アドバンスドスケジュールを実行するためのタイマのセットアップも必要です。ここで行うべき処理は、『RTA-OSEK リファレンスガイド』の「アドバンスドスケジュール」の項に詳しく説明されています。

割込みの設定

カテゴリ 1 および 2 の割込みソースは、`StartOS()` が呼び出される前に設定されている必要があります。またカテゴリ 1 の割込みの扱いは完全に RTA-OSEK コンポーネントの範囲外であるため、この割込みをイネーブルにしてすぐに割込みが生成されるようにする必要があります。カテゴリ 2 の割込みソースは、`StartOS()` が初期化を終了するまで、割込みを生成することはできません。

`StartOS()` を呼び出す前にカテゴリ 2 の割込みソースを設定し、その後、`StartOS()` から呼び出される `StartupHook()` 関数内で実際の割込み生成をイネーブルにします。呼び出された `StartOS()` は、直ちに割込みレベル（IPL）を OS レベルに引き上げ、リターンする前にユーザーレベルに引き下げます。このため、`StartupHook()` 内で割り込み生成をイネーブルにする処理が行われても、`StartOS()` がリターン直前に IPL を引き下げるまで実際の割込みは発生しません。

IPL が OS レベルに引き上げられ、その後、割込みソースの設定と割込みのイネーブル処理が行われるようにしてください。実際の割込みは、`StartOS()` がリターン直前に IPL を引き下げるまで発生しません。

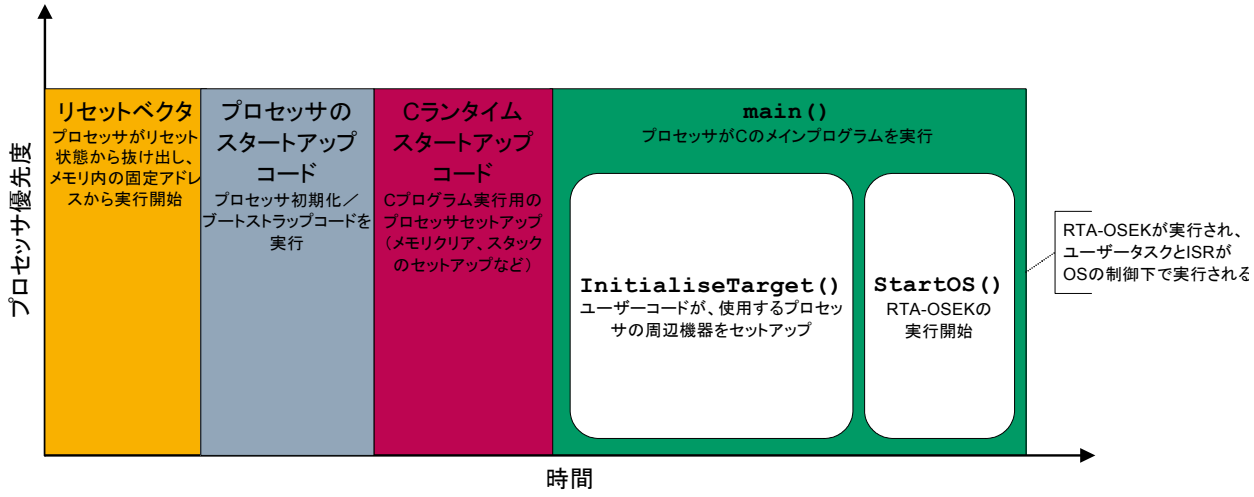


図 15-1 システムのスタートアップ

15.1.3 メモリイメージとリンカファイル

アプリケーションをビルドする際には、さまざまなコードやデータ、および ROM や RAM をメモリ上の適切な位置に配置する必要がありますが、これは通常、リンカによって行われます。リンカは、ユーザーコードから RTA-OSEK コンポーネントライブラリへの参照を解決し、必要なオブジェクトモジュールをバインドして出来上がったコードとデータをメモリ内のアドレスに配置し、最終的にターゲットにロードされるイメージを生成します。

ここで、リンカがどのようにしてメモリ配置を決定するのかを、例をあげて説明します。

セクション

コンパイラとアセンブラから出力されるコードとデータは、一般的に、「セクション」という単位でまとめられます。アセンブラの場合は以下のようになります。

```
.section CODE
.public MYPROC
mov   r1, FRED
add   r1, r1
ret
.end CODE
.section DATA
.public FRED
.word 100, 200, 300, 400
.end DATA
.section BSS
.public WORKSPACE
.space 200
.end BSS
```

コード例 15-2 セクションが含まれるアセンブラ出力の例

上記の例の場合、`MYPROC` は、アセンブルされたオブジェクトコードは“`CODE`”というメモリセクションに配置されます。`CODE` セクションの位置は、リンカで定義されます。同様に、“`FRED`”というラベルの付いたデータは、“`DATA`”というセクションに配置され、“`WORKSPACE`”というラベルの付いた 200 バイトの領域は“`BSS`”セクション内に配置されます。

通常、C コンパイラは、コードを“CODE”または“text”といった名前のセクションに割り当て、ROM に配置されるべき定数は“const”、変数は“data”、といった名前の付いたセクションに割り当てます。詳しくは、お使いのツールのマニュアルを参照してください。

RTA-OSEK 用には、以下のようなセクションが正しく定義されている必要があります。

セクション	ROM/RAM	説明（用途）
os_intvec	ROM	割込みベクタテーブル（RTA-OSEK が生成する場合） セクション名はターゲットに応じて異なる場合がありますので、バインディングマニュアルをご参照ください。
os_pur	RAM	RTA-OSEK の初期化を行わないデータ
os_pid	ROM	RTA-OSEK の読み取り専用データ
os_pir	RAM	RTA-OSEK が使用する RAM データのうち、ランタイムに初期化が必要なもの。これらの変数の初期値は os_pird 内に格納され、StartOS() によってコピーされます。
os_pird	ROM	os_pir の初期値
以下の 2 つのセクションは、near および far アドレス空間（以下の「near アドレス空間と far アドレス空間」の項を参照してください）をサポートするターゲットの場合のみ使用できます。		
os_pnir	RAM	RTA-OSEK が使用する RAM データのうち、ランタイムに初期化が必要なもの。これらの変数の初期値は os_pnird 内に格納され、StartOS() によってコピーされます。
os_pnird	ROM	os_pnir の初期値

次に、これらのセクションを実際のメモリ上に配置する方法を説明します。

near アドレス空間と far アドレス空間

プロセッサによっては、メモリ空間の一部に、サイズの小さい命令を使用でき、簡単なアドレス計算によりアクセスできる領域を持っています。この領域は、オンチップメモリや、短いサイクルでアクセスできる特殊なメモリに配置されます。RTA-OSEK ではこのメモリを ROM または RAM の near 空間と呼び、この領域にいくつかの主要なデータを配置しています。こういったプラットフォームにおいて、near 空間を ROM 内のどこに配置するべきであるか、といった情報は、バインディングマニュアルに記述されています。far 空間は、メモリ全体を指します。

Harvard アーキテクチャのプログラム空間とデータ空間

本書のメモリについての説明は、主に「フォンノイマン」アーキテクチャ、つまりデータとコードが ROM と RAM 内の 1 つのアドレス空間を異なるオフセットによって分割して使用する、という概念に基づいています。これに対し、一部のプロセッサ（PIC のような超小型のマイクロプロセッサや、高性能な DSP など）は“Harvard”アーキテクチャを採用し、ここではコードとデータ用にアドレス空間が明確に分離されています。Harvard アーキテクチャのプロセッサにおいて RTA は、データ空間（通常は RAM）を、通常のフォンノイマンアーキテクチャであれば ROM 定数として使用されるデータを格納するために使用でき、スタートアップコードには通常、定数データのコピーをデータ空間にフェッチするためのコードが含まれます。Harvard アーキテクチャのプロセッサを使用する場合は、『RTA-OSEK バインディングマニュアル』に定数のコピーを RAM に保存する方法が記述されていますので、参照してください。

リンカコントロールファイル

リンカコントロールファイルは、コード、データ、およびターゲットマイクロコントローラにダウンロードされるイメージ内の予約された空間を管理します。リンカファイルの内容はプラットフォームやターゲットによって異なりますが、以下の内容は常に含まれます。

- オンチップの ROM と RAM の配置（同じ CPU ファミリでも機種に応じて異なります。）
- 各メモリ空間に配置するセクションのリスト
- スタックポインタ、リセットアドレス、割込みベクタなどの初期化

以下にその一例を示します。

```
ONCHIPRAM start 0x0000 {
   Section .stack size 0x200 align 16 # system stack
   Section .sdata align 16 # small data
   Section os_pnir align 16 # RTA near data
}

def __SP = start stack     # initialize stack ptr

RAM start 0x4000 {
   Section .data align 16  # compiler data
   Section .bss align 16   # compiler BSS
   Section os_pur align 16 # RTA zeroed RAM
   Section os_pir align 16 # RTA initialized RAM
}

ROM start 0x8000 {
   Section .text              # compiler code
   Section .const             # compiler constants
   Section os_pid align 16    # RTA data
   Section os_pird align 16   # RTA initializer
   Section os_pnird align 16 # RTA initializer
}

VECTBL start 0xFF00 {
   Section os_vectbl          # RTA vector table
}

def __RESET = __main          # reset to __main
```

コード例 15-3 リンカコントロールファイル

上記のファイルには、ONCHIPRAM、RAM、ROM、VECTBL というメモリセクションが定義されています。各セクションに配置されるコードやデータは、コメント内に説明されています。

RTA-OSEK と共に提供されているサンプルアプリケーションのリンカコントロールファイルには、すべての項目にコメントが付けられています。各セクションを正しく配置する方法は、このコメントや、RTA-OSEK バインディングマニュアルを参照してください。

15.1.4 ターゲットへのダウンロード

リンカによって出力されるファイルは、通常、一般的なフォーマット（a.out、coff、elf、または IEEE695）のバイナリファイルです。一般的にこれらのファイルはデバッガやインサーキットエミュレータ、インサーキットプログラマなどに読み込むことができますが、場合によっては、バイナリファイルからテキストファイル（S レコード、インテル HEX など）に変換し、ターゲットに装着されたブートモニタなどにシリアルケーブルで転送するようなケースもあります。この変換を行うツールは、通常、リンカなどと共に提供されます。詳しくは、ターゲットプラットフォームや開発ツールチェーンの説明書を参照してください。

15.1.5 ROM での実行

RTA-OSEK はすべて ROM に書き込みでき、デバッガや開発環境に接続することなく、ターゲット CPU 上で実行できます。

15.2 RTA-OSEK コンポーネントの起動

RTA-OSEK コンポーネントは `StartOS()` が実行されて初めて起動されます。通常、これは `main()` から呼び出されます[1]。アプリケーションに必要なハードウェアの初期設定は、ユーザーの責任において行ってください。RTA-OSEK コンポーネントの初期状態については、『RTA-OSEK リファレンスガイド』に記載されています。

`StartOS(Appmode)` は、アプリケーションモードを示す引数を 1 つだけ取ります。この引数は、デフォルトモードである `OSDEFAULTAPPMODE` か、RTA-OSEK GUI で設定されているその他のモードのどちらかです。

コード例 15-4 のメイン関数の例では、オペレーティングシステムがデフォルトアプリケーションモードで起動されています。

```
#include "osekmain.h"
OS_MAIN(main) {

   InitializeTarget();

   StartOS(OSDEFAULTAPPMODE);

   for (;;) {
   /* Idle task. */
   }
}
```

コード例 15-4 メイン関数の例

`StartOS()` から戻ってくると、RTA-OSEK コンポーネントは稼働状態になりすべての割込みが有効になります。呼び出し元関数内の `StartOS()` 後のコードは、アイドルタスクとして扱われます。

アイドルタスクは他のタスクとまったく同様ですが、ターミネートはできません。RTA-OSEK コンポーネントをターミネートさせたくない場合、このアイドルタスクを無限ループにしてください。

ほとんどの RTA-OSEK コンポーネント API 関数はこのアイドルタスクから呼び出すことができます。しかし、このアイドルタスクがターミネートすることを要求する API 関数を使用することはできません。これらの API 関数の詳細については、『RTA-OSEK リファレンスガイド』を参照してください。

重要

`StartOS(Appmode)` からコントロールが戻ってくる前には、RTA-OSEK コンポーネント API コールを行うことはできず、カテゴリ 2 の割込みは処理されません。

すべてのカテゴリ 2 割込みをディセーブルにし、出力コンペアマッチなど、どのイベントによってもカテゴリ 2 の割込みが発生しないようにすることにより、RTA-OSEK コンポーネントをサスペンド状態にすることができます。

このため、RTA-OSEK コンポーネントは、カテゴリ 2 の割込みがまったく発生していない状態でアイドルタスクが稼働中である場合は、サスペンド状態になっています。RTA-OSEK コンポーネントを再び稼働させるには、カテゴリ 2 の割込みを再び有効にしてから RTA-OSEK コンポーネント関数の実行を再開します。

15.2.1 アプリケーションモード

OSEK には**アプリケーションモード**があります。これらを使うと、オペレーティングシステムを起動したときにどのタスクとアラームが自動起動されるかをユーザーが管理することができます。

アプリケーションは様々なモードで起動できます。各モードは全体の機能の一部をなすものであり、アプリケーションの特定の機能に対応しています。たとえば、エンドオブラインプログラミングモード、トランスポートモード、ノーマルモードなどを定義することができます。

1. RTA-OSEK コンポーネントアプリケーションには `main()` ではなく `OS_MAIN()` が使用される傾向があります。その方が移植性に優れたアプリケーションになるためです。

OSDEFAULTAPPMODE はデフォルトのアプリケーションモードです。ユーザーは RTA-OSEK GUI を使用して、必要な数だけアプリケーションモードを定義できます。図 15-2 のようにして、アプリケーションモードをアプリケーションに追加します。

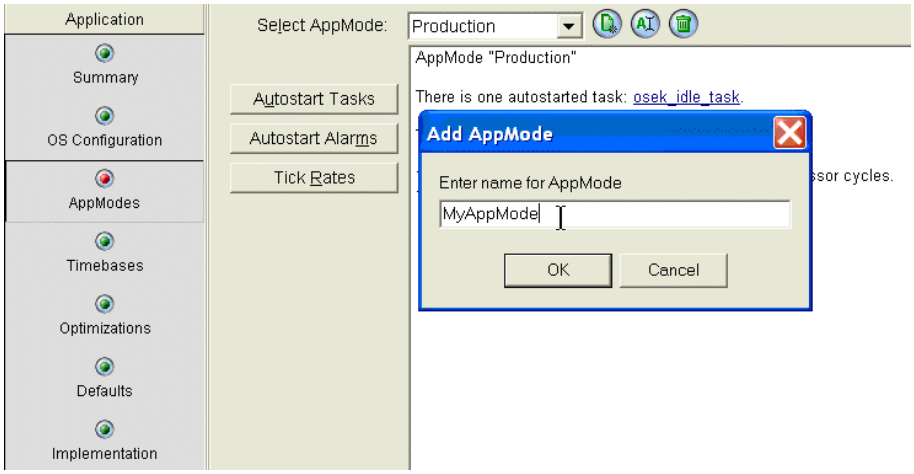


図 15-2 アプリケーションモードを設定する

StartOS(Appmode) は、自動起動されるようにユーザーが指定したすべてのタスクを起動し、さらに、自動起動されるように指定したすべてのアラームをセットします。

15.2.2 タスクの自動起動

RTA-OSEK GUI で、StartOS() の中で自動起動されるタスクを図 15-3 のようにして設定できます。

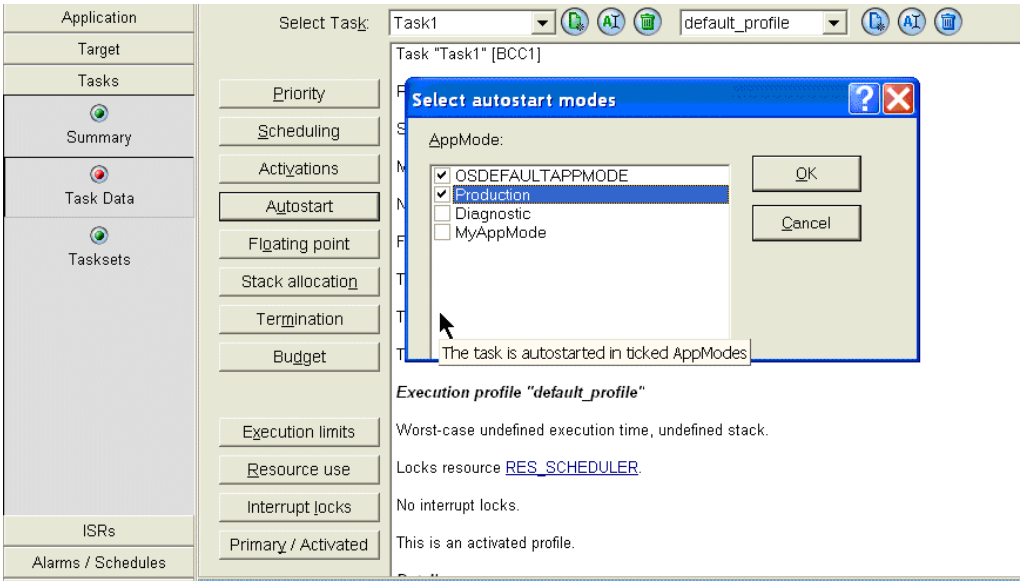


図 15-3 自動起動されるタスクを宣言する

また、自動起動がどのアプリケーションモードで行われるかを設定することができます。自動起動されるすべてのタスクは、StartOS() から戻った時点において、すでに起動されています。

上記の例では、Task1 が OSEKDEFAULTAPPMODE および Production アプリケーションモードで自動起動されるように設定されています。

15.2.3 アラームの自動起動

RTA-OSEK GUI ではアラームの自動起動を設定することができます。自動起動されるすべてのアラームは、`StartOS()` から戻ってきた時点ですべて有効になっています。図 15-4 にアラームの自動起動を設定する方法を示します。

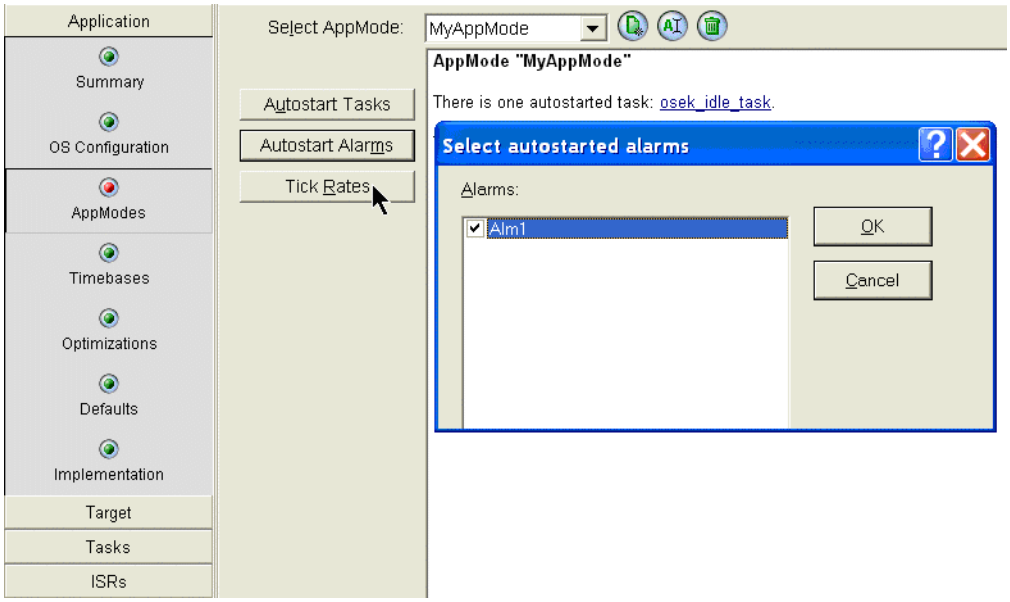


図 15-4 アラームを自動起動させる

アラームはアプリケーションモードペインで設定されるので、アプリケーションモードごとに自動起動するアラームを選択することになります。図 15-4 の例では、`MyAppMode` というアプリケーションモードについてのアラームに自動起動が設定されています。

多数のアラームをランタイムに同期させたい場合は、それらのアラームが必ず自動起動されるように設定してください。アラームの同期化を保証する方法は他にありません。

15.3 RTA-OSEK コンポーネントのシャットダウン

オペレーティングシステムは、任意の時点で `ShutdownOS()` を呼び出すことによりシャットダウンすることができます。この呼び出しが行われると、RTA-OSEK コンポーネントは直ちに割込みをディセーブルにしてから無限ループに入ります。ユーザーが `ShutdownHook()` を設定している場合は、無限ループに入る前にこのフックが呼び出されます。

15.4 RTA-OSEK コンポーネントの再起動

RTA-OSEK では、`osRestartOS()` を呼び出すことによってカーネルを初期状態にリセットすることができます。その後、再度 `StartOS()` を呼び出せば、異なるアプリケーションモードでアプリケーションを実行することができます。

移植性

`osRestartOS()` は RTA-OSEK 固有のもので、OSEK や AUTOSAR の仕様には含まれません。

アプリケーション内で `osRestartOS()` を使用するには、図 15-5 に示す方法でこれを有効にしておく必要があります。

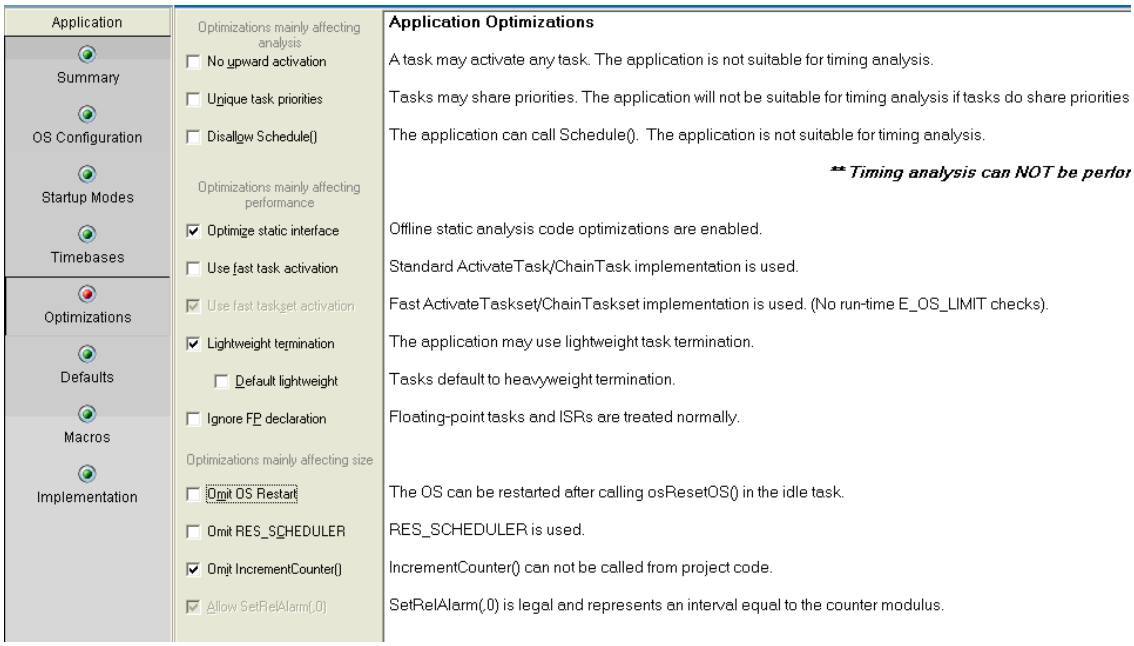


図 15-5 アラームを自動起動させる

アプリケーション内で `osRestartOS()` を使用する際は、2 つの点に留意してください。

まず、`osRestartOS()` は必ずアプリケーションのアイドルタスクから、他のカーネルサービス（アラーム、スケジュールテーブル、スケジュール）が起動しておらず、かつ他のアプリケーションタスクが実行中、ウェイト状態、レディ状態でないときにのみ呼び出すようにしてください。またタスクの起動を行う割込みソースはディセーブル状態にしておいてください。

2 番目に、アイドルタスクの構造が、カーネルを起動できるようになっている必要があります。図 15-5 にその例を示します。示す方法でこれを有効にしておく必要があります。

```
   AppModeType CurrentAppMode;

     InitialiseTarget();

     /* Set up normal application mode */
     CurrentAppMode = Default;

     while(1){
       StartOS(CurrentAppMode);
       /* Idle task */
       while(1) {
         /* Test for mode switch */
         if( ModeSwitchNecessary )
           break;
       }
       /* Reset OS */
       osResetOS()
     }
}
```

図 15-6 アイドルルルタスク内で osResetOS() を使用する

15.5 まとめ

- RTA-OSEK コンポーネントは、すべてのコードやデータが正しいメモリ空間に配置されないと、機能しません。
- RTA-OSEK コンポーネントが実行される前には、いくつかの処理が必要です。
- RTA-OSEK コンポーネントは `StartOS()` が呼び出されるまでは実行されません。
- `ShutdownOS()` を使用することにより、RTA-OSEK コンポーネントをいつでも停止することができます。
- `osResetOS()` を使用することにより、RTA-OSEK コンポーネントをリセットすることができます。その後、異なるアプリケーションモードで再起動できます。
- アプリケーションモードを使用して、タスクとアラームを自動起動することができます。

16 エラー処理と実行監視

開発の初期段階においては、デバッグを行いながら、アプリケーションの実行状況を監視する必要があります。「実行監視」（'execution monitoring'）は、タスク実行のトレースを生成するのと同じくらい簡単に行うことができますが、場合によっては、タスクの実際の実行時間やスタック使用量をモニタしてタイミング分析とスタック分析のためのワーストケース値を取得することも必要となるかもしれません。

RTA-OSEK コンポーネントには OSEK **フック**が用意されています。フックは、所定の API を含むユーザー C 関数です。フックは RTA-OSEK コンポーネントの処理中の特定のポイントで RTA-OSEK コンポーネントにより呼び出されます。

フック関数内で実行されるコードは、限られた API 関数を呼び出すことができます。この制限については『RTA-OSEK リファレンスガイド』に記載されています。

OSEK には以下のフックが定義されています。

- 起動フック（'Startup Hook'）
- シャットダウンフック（'Shutdown Hook'）
- エラーフック（'Error Hook'）
- プリタスクフック（'PreTask Hook'）
- ポストタスクフック（'PostTask Hook'）

OSEK フックルーチンは任意に使用でき、RTA-OSEK コンポーネントのどのビルドモードでも使用できます。上記の OSEK フックルーチンの他に、RTA-OSEK コンポーネントにより以下の 2 つのフックが定義されています。

- スタック障害フック（'Stack Fault Hook'）
- オーバーランフック（'Overrun Hook'）

これらの RTA-OSEK コンポーネントフックルーチンは必ず使用しなければなりません。またスタック障害フックは拡張タスク内でのみ使用でき、オーバーランフックはタイミングビルドと拡張ビルドにおいてのみ使用できまる。

これらの各フックの詳細については、本章で後述します。

16.1 フックルーチンを有効にする

RTA-OSEK GUI で、アプリケーションで使用したいフックを任意に選択することができます。図 16-1 にその例を示します。

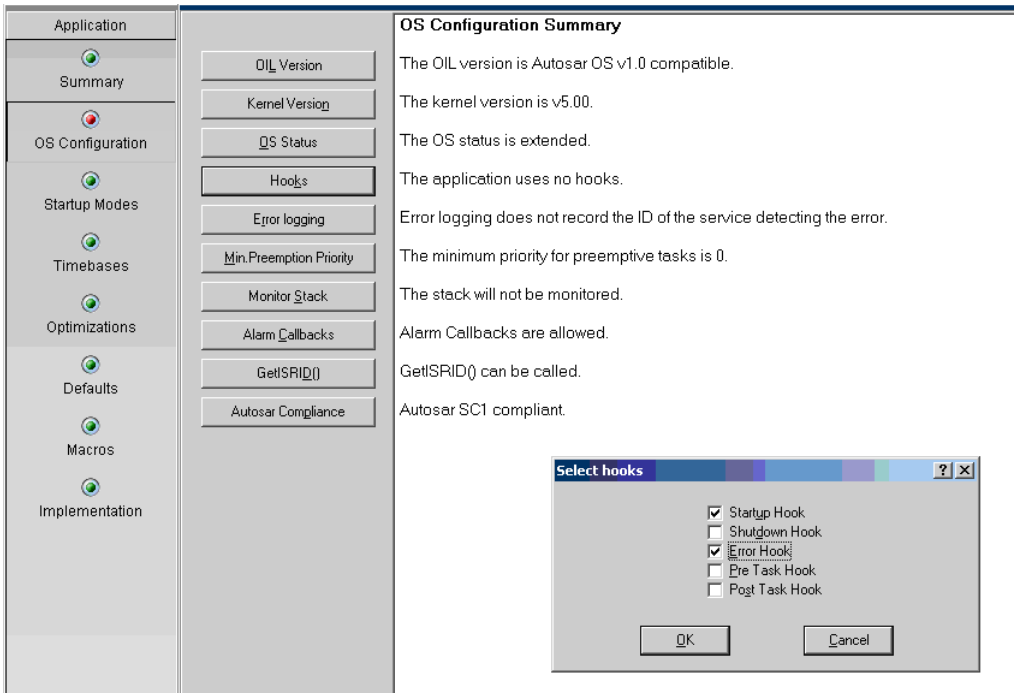


図 16-1 アプリケーション用の OSEK フックを設定する

図 16-1 では、スタートアップフックとエラーフックが有効になっています。

> **重要**
> フックが有効になっている場合、そのフック用の有効なコードがユーザーによって用意されていないと、プログラムは正しくリンクされません。

RTA-OSEK では以下の一連のマクロがあり、これらは、対応するフックを有効にした場合にのみ定義されます。

```
OSEK_STARTUPHOOK
OSEK_SHUTDOWNHOOK
OSEK_PRETASKHOOK
OSEK_POSTTASKHOOK
OSEK_ERRORHOOK
```

これらのマクロを使用して、ユーザーコード内でフックルーチンを条件コンパイルすることがます。

```
#ifdef OSEK_STARTUPHOOK
OS_HOOK(void) StatupHook (void)
{
  /* Your code */
}
#endif /* OSEK_STARTUPHOOK */
```

16.2 スタートアップフック（'Startup Hook'）

スタートアップフックは、`StartOS(OSDEFAULTAPPMODE)` 内でカーネルが初期化された後、スケジュールが実行状態となる前に、RTA-OSEK コンポーネントにより呼び出されます。図 16-2 は、RTA-OSEK コンポーネントの初期化の中でのスタートアップフック実行の相対的位置を示しています。

図 16-2 スタートアップフックの実行

スタートアップフックは、コード例 16-1 のように作成してください。

```
#ifdef OSEK_STARTUPHOOK

  OS_HOOK(void) StartupHook(void) {
    /* Startup hook code. */
  }

#endif
```

コード例 16-1 スタートアップフックを使用する

多くの場合、スタートアップフックは、OSEK COM の初期化やターゲットハードウェアの初期化（割込みソースの設定と初期化など）のために使用されます。

16.3 シャットダウンフック（'Shutdown Hook'）

シャットダウンフックは、ShutdownOS() の実行中に呼び出されます。図 16-3 は、ShutdownOS() の中でのシャットダウンフックの実行の相対的位置を示しています。

図 16-3 シャットダウンフックの実行

シャットダウンフックは、コード例 16-2 のように作成してください。

```
#ifdef OSEK_SHUTDOWNHOOK

  OS_HOOK(void) ShutdownHook(StatusType s) {
    /* Shutdown hook code. */
  }

#endif
```

コード例 16-2 シャットダウンフックを使用する

一般的に、シャットダウンフックは、COM をシャットダウンするために使用されます。

通常、シャットダウンフックはコントロールを呼び出し元に戻しませんが、戻すようにすると、RTA-OSEK コンポーネントは OS レベルで稼働する無限ループに入ります。

16.4 エラーフック（'Error Hook'）

RTA-OSEK コンポーネントのすべての API 関数はステータスコードを返します。ステータスコードの詳細については、『RTA-OSEK リファレンスガイド』に記載されています。

API 関数が返すステータスコードは、ランタイムにチェックすることができるので、ステータスに応じたフォルトトレランス性をアプリケーションに組み込むことが可能です。

これにより、標準ビルドにおいて発生する可能性のあるエラー（例：ActuvateTask() が E_OS_LIMIT を返す）をチェックすることができます。コード例 16-4 にその使用方法を示します。

```
if (ActivateTask(Task1) != E_OK) {

   /* Handle error during task activation. */
}
```

コード例 16-3 推奨されるエラーフックの構造

また、OSEK の "catch all" エラーハンドラを設定することもできます。これは**エラーフック**と呼ばれます。エラーフックが有効になっていると、このフックは、API 関数が `E_OK` 以外のステータスコードを返そうとするときに RTA-OSEK によって呼び出されます。このステータスコードはエラーフックルーチンに渡され、エラーのタイプの判定に使用されます。

エラーの重大さに応じて、`ShutdownOS()` を呼び出してターミネートするか、エラー処理やロギングを行ってから `ErrorHook()` からコントロールを戻してタスクを再開するかを選択することができます。

コード例 16-4 は、エラーフックの一般的な構造を示しています。

```
#ifdef OSEK_ERRORHOOK

   OS_HOOK(void) ErrorHook(StatusType status) {
      switch (status) {

      case E_OS_ACCESS:
         /* Handle error then return. */
         break;

      case E_OS_LIMIT:
         /* Terminate. */
         ShutdownOS(status);

      default:
         break;
      }
     }

#endif
```

コード例 16-4 推奨されるエラーフックの構造

エラーフックは精密なデバッグの用途に適していますが、エラーについてより詳細な情報（どの API コールでエラーが発生したのか、また API にどのような引数が渡されたのか、など）が必要になる場合もあります。このような情報は、RTA-OSEK GUI を使用して高度なエラーロギングを設定することによってランタイムに取得することができます。

16.4.1 高度なエラーロギングの設定

RTA-OSEK では、以下の 2 通りのレベルで情報を取得できます。

- サービスの詳細情報を記録しない（デフォルト設定）
- API 名だけを記録する
- API 名とその API に渡された引数を記録する

図 16-4 は、RTA-OSEK GUI で詳細情報のレベルを定義する方法を示しています。

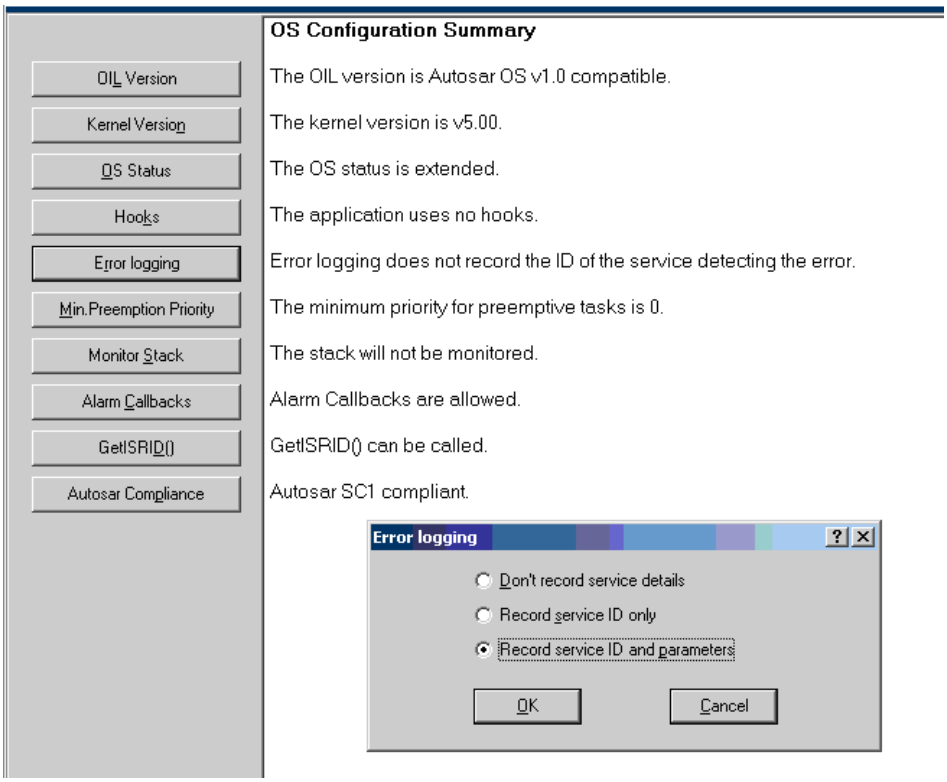


図 16-4 高度なエラーロギングを設定する

サービスの詳細情報を記録しないようにすれば、この情報の収集のためのオーバーヘッドがなくなります。

16.4.2 高度なエラーロギングの使用

エラーロギングを有効にすると、RTA-OSEK はエラーの原因となった API 関数の名前とその関数に渡された引数にアクセスするためのマクロ群を生成します。

エラーの原因となった API 関数は、`OSErrorGetServiceId()` マクロを用いて調べることができます。このマクロは、`OSServiceIdType` を `OSServiceId_<API name>` という形で返します。たとえば、`ActivateTask()` がエラーになった場合、`OSErrorGetServiceId()` は `OSServiceId_ActivateTask` という値を返します。

API 関数の引数は、コード例 16-5 に示す形式のマクロを使用して取得することができます。1 つの API 関数の各引数ごとに 1 つのマクロが定義されます。

```
OSError_<API Name>_<API Parameter Name>
```

コード例 16-5 高度なエラーロギング

`ActivateTask()` の例を使用すると、`OSError_ActivateTask_TaskId` は、`ActivateTask()` に渡された引数 `TaskId` を返します。このエラーロギング情報は、コード例 16-6 のように `ErrorHook()` コードに組み込んで利用することができます。

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType status) {

   OSServiceIdType callee;
```

```
      switch (status) {
         case E_OS_ID:
            /* API call called with invalid handle. */
            callee = OSErrorGetServiceId();

            Switch (callee) {
               case OSServiceId_ActivateTask:
                  /* Handle error. */
                  break;
               case OSServiceId_ChainTask:
                  /* Handle error. */
                  break;
               case OSServiceId_SetRelAlarm:
                  /* Handle error. */
                  break;
               default:
                  break;
            }

            break;

         case E_OS_LIMIT:
            /* Terminate. */
            ShutdownOS();

         default:
            break;

      }
   }

   #endif
```

コード例 16-6 付加的なエラーロギング情報

API の名前と引数を取得するマクロは、必ずエラーフック内から使用してください。これらのマクロが表す値は、フックのスコープ外では保持されません。

> **重要**
> 拡張エラーロギングを使用する場合、`OSErrorGetServiceId()` が、期待されるものではない値を返す場合があります。一般に、このような事態は API 関数に二次的機能がある場合に起こります。たとえば、COM を用いてメッセージを送信する場合、二次的機能として考えられるのはタスクの起動です。このタスク起動がエラーになると、ユーザーが行った API コールは `SendMessage()` であるのに関わらず、`OSErrorGetServiceId()` は `OSServiceId_ActivateTask` を返します。

16.4.3 どのタスク／ISR が実行中であるかをチェックする

RTA-OSEK アプリケーションをデバッグする際、どのタスクまたはカテゴリ 2 タスクがエラーを発行したかを調べるには、OSEK OS の `GetTaskID()` を使用できます。

コード例 16-7 にその使用例を示します。

```
   TaskType CurrentTaskID;

     /* Passes a TaskRefType for the return
      * value of GetTaskID) */
```

```
    GetTaskID (&CurrentTaskID);

    if (CurrentTaskID == Task1) {
      /* Code for task 1 */
    } else {
      if (CurrentTaskID == Task2) {
        /* Code for task 2 */
      }
      ...
    }
  }
```

コード例 16-7 `GetTaskID()` の使用方法

AUTOSAR OS ではこの機能が拡張され、`GetISRID()` によってカテゴリ 2 の ISR もチェックできるようになっています。

RTA-OSEK の場合、コンフィギュレーション設定でこの `GetISRID()` を使用するかどうか、つまり、高度なデバッグ機能を利用するか、それとも API の OSEK OS への互換性を保持するかどうかを選択できます。これは図 16-5 のように設定します。

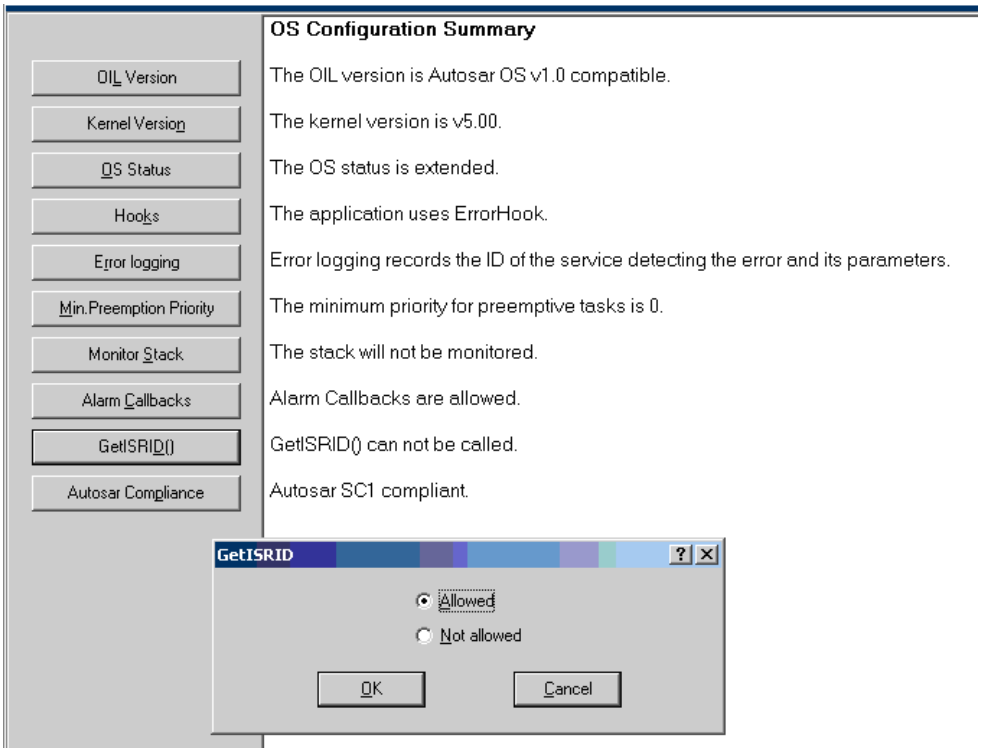


図 16-5 GetTaskID() の使用方法

`GetISRID()` は、`GetTaskID()` とは異なり、関数の出力引数ではなく関数の戻り値として ISR の ID を返します。タスクの実行中に `GetISRID()` を呼び出すと、そこから `INVALID_ISR` が戻ります。

以下に `GetISRID()` と `GetTaskID()` の両方を使用する例を示します。

```
ISRType  CurrentISRID
TaskType CurrentTaskID;

  /* Is an ISR running? */
  CurrentISRID = GetISRIS();
  if ( CurrentISRID /= INVALID_ISR )
  {
```

```
    if (CurrentISRID == ISR1) {
      /* Work out which ISR */
    }

  } else {
    GetTaskID(&CurrentTaskID);
    if ( CurrentTaskID == Task1 ) {
      /* Work out which task */
    }
  }
}
```

16.5 プリタスクフック（'PreTask Hook'）とポストタスクフック（'PostTask Hook'）

プリタスクフックは、タスクが実行状態になる時に必ず RTA-OSEK コンポーネントにより呼び出されます。つまり、プリタスクフックは、タスクがプリエンプション後に再開される際に必ず呼び出されます。

ポストタスクフックは、タスクが実行状態から抜ける時に必ず RTA-OSEK コンポーネントにより呼び出されます。つまり、ポストタスクフックは、タスクがターミネートする時と、タスクがプリエンプトされる際に必ず呼び出されます。

図 16-6 は、タスクのプリエンプション処理の中でのプリタスクフックとポストタスクフックの相対的位置を示しています。

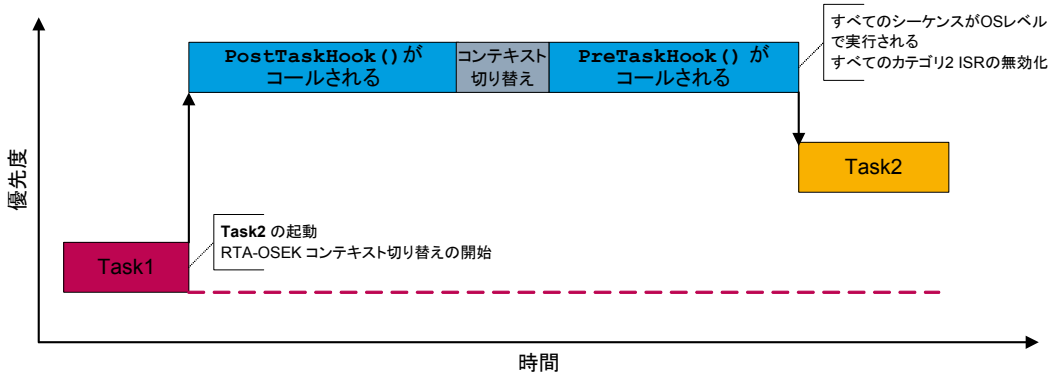


図 16-6 タスクのプリエンプト時の `PreTaskHook( )` と `PostTaskHook( )` の相対的位置

コード例 16-8 に、これらのフックの使用例を示します。

```
#ifdef OSEK_PRETASKHOOK

OS_HOOK(void) PreTaskHook(void) {
   /* PreTask hook code. */
}

#endif

#ifdef OSEK_POSTTASKHOOK

OS_HOOK(void) PostTaskHook(void) {
/* PostTask hook code. */
}

#endif
```

コード例 16-8 OSEK のプリタスクフックとポストタスクフック

プリタスクフックとポストタスクフックはタスクの入口と出口において呼び出され、さらにプリエンプトされたときと実行再開時にも呼び出されます。これらのフックを使用して、アプリケーションの実行トレースを記録することができます。なお、アプリケーション内の全タスクにおいて、同じプリタスクフックとポストタスクフックが使用される必要があるため、フックルーチンに入ったときに `GetTaskID()` を使用して、どのタスクが実行されているか、またはどのタスクが実行されるかを調べてください。

16.6 スタック障害フック（'Stack Fault Hook'）

スタック障害フックは、RTA-OSEK コンポーネントが拡張タスクのスタック管理上の問題を検知した時に必ず呼び出されます。

重要

拡張タスクを使用する際は、必ずアプリケーションコード内に `StackFaultHook()` を扱う関数を用意してください。

移植性

スタック障害フックは RTA-OSEK コンポーネントだけで使用されるもので、OSEK OS 規格には含まれていません。

`StackFaultHook()` は RTA-OSEK コンポーネントから、次の 3 つの引数を用いて呼び出されます。

- `StackID`
  この値は、スタックを 1 つしか使用しないターゲットの場合は常に 0 です。それ以外の場合は、障害がどのスタックのものかを示す整数になります。各ターゲットのスタックにどのように番号が付けられているかは、『RTA-OSEK バインディングマニュアル』に記載されています。
- `StackError`
  エラーの原因を示す整数です。
  `OS_EXTENDED_TASK_STARTING`:
  アプリケーションのスタックポインタがすでに高すぎたか低すぎたため、タスクは開始スタックポインタを設定することができませんでした。

  `OS_EXTENDED_TASK_RESUMING`:
  アプリケーションのスタックポインタがすでに高すぎたか低すぎたため、タスクは再開スタックポインタを設定することができませんでした。

  `OS_EXTENDED_TASK_WAITING`:
  タスクが RTA-OSEK GUI での設定時に宣言されたよりも多くのスタックを使用したため、タスクはスタックを空にすることができず、待ち状態に移行できませんでした。
- `Overflow`
  アプリケーションで想定されていたバイト数よりもスタックの実際の使用量の方が何バイト多いかを示します。

スタック障害フックの例をコード例 16-9 に示します。

```
OS_HOOK(void) StackFaultHook(
   SmallType StackID,
   SmallType StackError,
   UIntType  Overflow) {
   for (;;) {

     /* Loop forever. */
   }
}
```

コード例 16-9 スタック障害フック

StackFaultHook() は、間違ったスタック使用量情報が RTA-OSEK GUI に入力された時にしか発生しません。現在稼働中のタスクよりも優先度の低いタスクのスタック宣言を確認してください。

重要

StackFaultHook() からはコントロールを戻さないでください。普通、このフックに入るということは、スタックが壊れていることを意味しているため、このフックからコントロールを戻しても、その後のアプリケーションの挙動は不確定なものになってしまいます。

16.7 実行時間の測定と監視

移植性

RTA-OSEK のタイミングモニタと測定の機能は、OSEK および AUTOSAR の規格には含まれていないため、他の OSEK OS への移植性は考慮されていません。

RTA-OSEK コンポーネントには、ユーザーコードの実行時間をカーネルレベルで測定する機能があります。アプリケーションの実行時間を測定する場合はタイミングビルド（'Timing build'）を使用するのが普通ですが、実際にはタイミング測定機能は RTA-OSEK コンポーネントのタイミングビルドと拡張ビルドの両方で使用できます。拡張ビルドでタイミング測定機能を使用すると、ユーザーが取得する時間情報には、詳細なエラーチェックを行うためのオーバーヘッド分の時間も含まれてしまいます。

タイミングビルドと標準ビルドとで、カーネルとアプリケーションのコードは、タイミング測定をサポートするために必要なコード以外の部分はまったく同じです。

16.7.1 タイミング測定の有効化

タイミング測定を行うには、「ストップウォッチ」ソースが必要です。通常、これはターゲットハードウェア上のフリーランニングカウンタを使用します。RTA-OSEK コンポーネントは GetStopwatch() コールバック関数を用いてストップウォッチにアクセスします。この関数の例をコード例 16-10 に示します。

```
OS_NONREENTRANT(StopwatchTickType)
GetStopwatch(void) {
   return CurrentValueOfFreeRunningCounter;
}
```

コード例 16-10 ストップウォッチにアクセスする

ストップウォッチの動きがプロセッサクロックよりも遅い場合は、実行時間を求めるための引き算に不確実性が含まれてしまい、それを補償するための関数も作成しなければならなくなります。コード例 16-11 は、この GetStopwatchUncertainty() の使用法を示しています。

```
OS_NONREENTRANT(StopwatchTickType)
GetStopwatchUncertainty(void){
   return Uncertainty;
}
```

コード例 16-11 不確実性を補償する

ストップウォッチのチックの長さが CPU 命令サイクルの長さと同じ場合は、この関数から返される不確実性（'uncertainty'）の値は通常 0 で、そうでない場合は 1 です。

不確実性値が 1 より大きくなるシステムも考えられます。これはまれですが、たとえばストップウォッチが 40MHz で稼働し、カウンタハードウェアが 10MHz で稼働しているような場合です。このような場合は、GetStopwatch() 内でカウンタ値を 4 倍にし、不確実性値として 4 を返してください。

重要

タイミングビルドまたは拡張ビルドを使用する場合は、GetStopwatchUncertainty() と GetStopwatch() のコードをユーザーが用意する必要があります。これらの関数がないとプログラムが正しくリンクされません。

16.7.2 実行時間の測定

アプリケーションがタイミングビルドまたは拡張ビルドを使用する場合、RTA-OSEK コンポーネントはアプリケーション内のそれぞれのタスクとカテゴリ 2 の ISR の実行時間を測定します。

RTA-OSEK コンポーネントはそれぞれのタスクまたはカテゴリ 2 の ISR について測定された最長の実行時間のログを維持しているので、ユーザーは `GetLargestExecutionTime()` を使用して、それぞれのタスクまたは ISR について測定された最長の実行時間を取得することができます。

16.7.3 タイミングバジェット（'timing budget'）の設定

それぞれのタスクとカテゴリ 2 の ISR についての「実行時間バジェット」('execution time budget') をアプリケーションに設定することができます。これらの値は必要に応じて任意に設定します。実行バジェットを設定する方法を図 16-7 に示します。

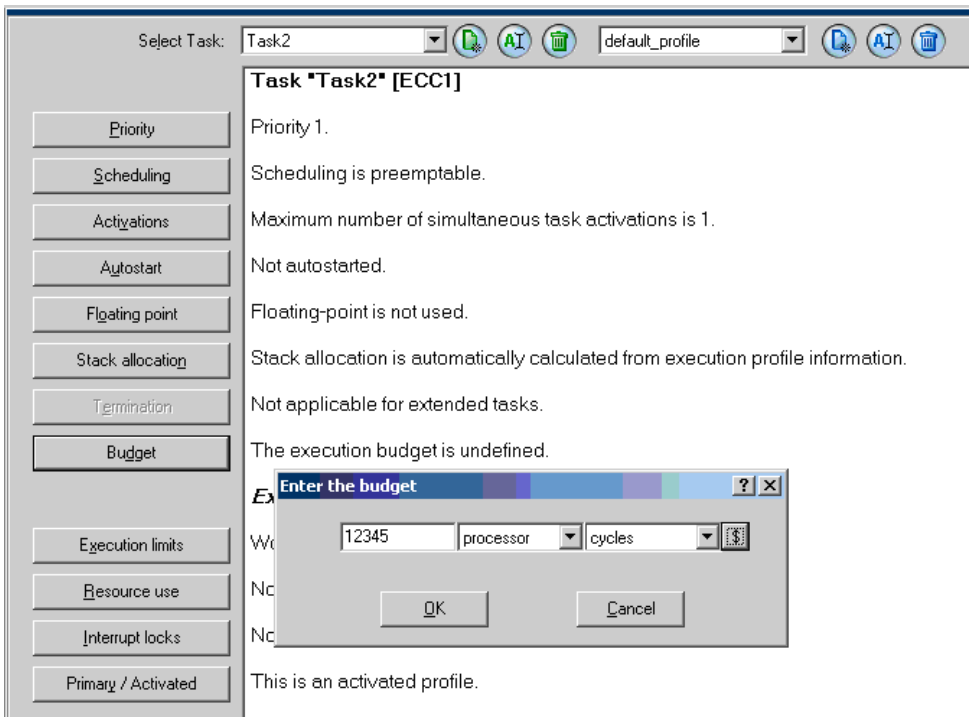


図 16-7 実行時間バジェットを定義する

タイミングビルドまたは拡張ビルドを使用している場合、RTA-OSEK コンポーネントはタスクまたはカテゴリ 2 の ISR がバジェットに定義されているよりも多くの時間を消費しないかどうかをチェックします。バジェットを超えると、RTA-OSEK コンポーネントはそのタスクがターミネートする時（または、そのタスクが拡張タスクの場合は `WaitEvent()` を呼び出す時）に `OverrunHook()` を呼び出します。これにより、ユーザーはバジェット超過をロギングすることができます。

> **重要**
> RTA-OSEK コンポーネントのタイミングビルドまたは拡張ビルドを使用する場合、`OverrunHook()` は必須のものとなります。

`OverrunHook()` のプロトタイプをコード例 16-12 に紹介します。

```
#ifdef OS_ET_MEASURE

   OS_HOOK(void) OverrunHook(void) {
      /* Log budget overruns. */
   }

#endif
```

コード例 16-12 オーバーランフックのプロトタイプ

拡張タスクの場合、実行時間はタスク開始時と `WaitEvent()` からの再開時に 0 にリセットされます。通常、バジェットは連続する `WaitEvent()` の呼び出し間の実行時間をチェックするために使用されます。

また、タスクが別のタスクまたは ISR によりプリエンプトされている間は、実行時間は RTA-OSEK コンポーネントによってのみサンプリングされます。

ごく稀に、バジェット超過が見過ごされてしまう場合があります。このような状態は、連続するプリエンプション間の間隔が `StopwatchTickType` で測定できる最大時間に近くなると発生する可能性があります。`StopwatchTickType` の範囲はターゲットにより異なりますが、通常は 2^{16} または 2^{32} です。

16.7.4 ブロッキング時間の取得

タイミング分析が「悲観的」すぎる結果とならないようにするためには、ユーザーは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内で費やされる時間と、割込みがディセーブルになっている時間の両方について正確なタイミングを計る必要があります。

`GetStopwatch()` または `GetExecutionTime()` というタイミング API 関数を使用して、これらのコードセクションの直前と直後にストップウォッチの値を取得します。また、中間値を保持して「高水位線（'high watermark'）」の時間を維持するためのコードを追加する必要があります。

ユーザーのアプリケーションが実行時間を取得するために使用するコードは、すべて条件コンパイルが行われるようにしておいてください。RTA-OSEK コンポーネントには、このための `OS_ET_MEASURE` というマクロが用意されています。コード例 16-13 は、リソースが保持されている時間を取得する条件コンパイルの例を示しています。

```
TASK(Task1) {
   ...
#if defined(OS_ET_MEASURE)

   /* Get time for GetExecutionTime() call itself. */
   start = GetExecutionTime();
   finish = GetExecutionTime();
   correction = finish - start - GetStopwatchUncertainty();

   /* Measure resource lock time. */
   start = GetExecutionTime();

#endif

   GetResource(Resource1);
   /* Critical section. */
   ReleaseResource(Resource1);

#if defined(OS_ET_MEASURE)

   finish = GetExecutionTime();
   /* Calculate amount of time used. */
   used = finish - start - correction +
             GetStopwatchUncertainty();
```

```
#endif
}
```

コード例 16-13 条件コンパイルの定義方法

16.7.5 不確定な計算

タイミングモニタの機能によるオーバヘッドは小さいため、量産コードをタイミングビルドで生成することが可能です。これにより、「不確定な計算」（'imprecise computation'）を行うことができます。

不確定な計算は、反復して 1 つの結果が確定されるようなアプリケーションにおいて有効で、たとえば Newton-Raphson 法を使用して値を確定することができます。

タスクがワーストケースパスを通っていない場合、ワーストケースの実行時間より短い時間で実行されることになるため、その際に余った CPU サイクルを利用して結果の精度を向上させることができます。この技法をコード例 16-14 に紹介します。

```
TASK(Task1) {
   ...
   while ((Budget - GetExecutionTime()) > LoopTime)
   {
      /* Perform iterative refinement of output. */
   }
   ...
}
```

コード例 16-14 不確定な計算

16.8 スタック使用量の測定と監視

16.8.1 測定

タスクプロファイルには、RTA-OSEK がアプリケーション全体のワーストケースのスタック使用量を計算するために使用する、任意指定のスタックスペース数値を含めることができます。

ユーザーが定義するこの数値は、各タスクが使用するワーストケースのスタック使用量を表すもので、タスク自身に必要な量と、そのタスクが行う関数呼び出し階層のワーストケースパスにおいて消費される量を合計したものでなければなりません。

通常、ユーザーはこの情報を、リンカ、あるいはデバッグ／エミュレーション環境から取得します。この方法は最も好ましい方法ですが、もしもユーザーのツールチェーンでこの情報を得ることができない場合は、RTA-OSEK コンポーネントに備わっている内部機能を使用してこれらの数値を測定することもできます。

API 関数 `GetStackOffset()` は、RTA-OSEK コンポーネントにおけるスタック測定に使用されます。スタックを 1 つしか使用しないターゲットでは、`GetStackOffset()` は使用されたスタックスペースのバイト数を示すスカラ値を返します。一方、複数のスタックを持つターゲットの場合は、`GetStackOffset()` は各スタックで使用されているバイト数を含む構造体を返します。この構造体からスタックスペース情報を取り出す方法については、各ターゲットの『RTA-OSEK バインディングマニュアル』に記載されています。

移植性

RTA-OSEK のスタック測定機能は、OSEK および AUTOSAR の規格には含まれていないため、他の OSEK OS への移植性は考慮されていません。

返される値はスタックポインタの初期位置からの値です。したがって、タスクまたは ISR 内でこの呼び出しを行うと、返される値には C スタートアップコード、メインプログラム（アイドルタスク）、およびプリエンプトされているすべてのタスクまたは ISR により使用されているスタック（OS コンテキストのアイドルタスクとメインプログラムにより使用されているスペースを含む）が含まれます。
ただし、`GetStackOffset()` が返す数値にはこの関数呼び出し自体に使用されるスタックスペースは含まれません。図 16-8 は、`GetStackOffset()` がタスク `TaskHIGH` から呼び出された時に返すサイズを示しています。

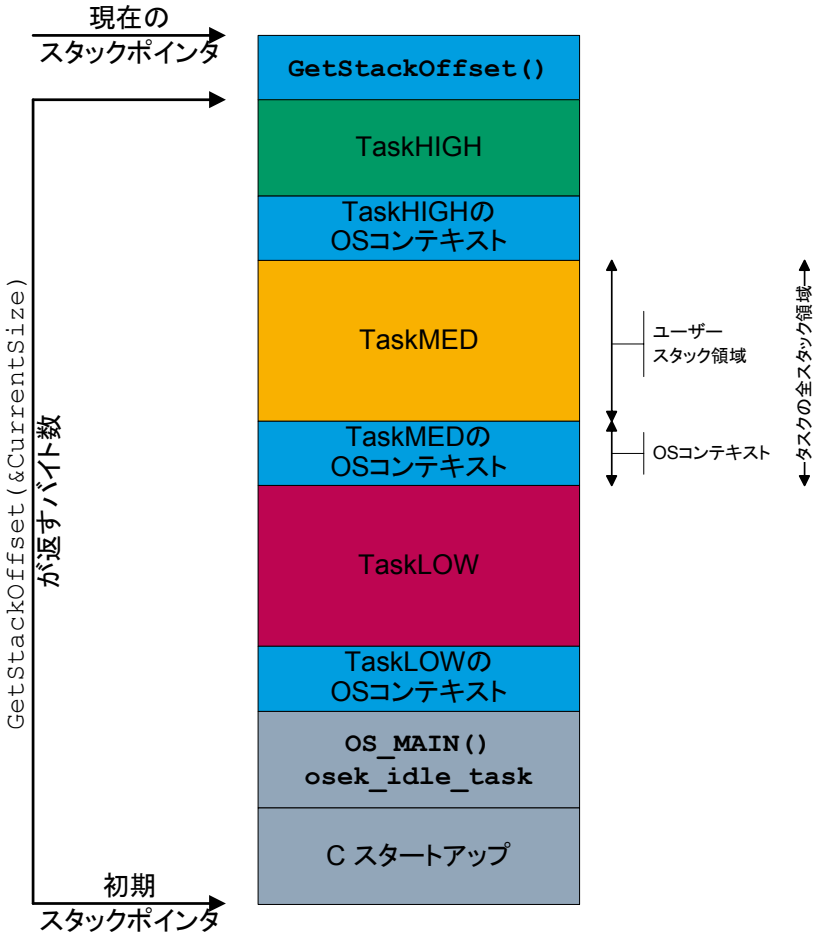


図 16-8 スタック構成図

それぞれのタスクまたは ISR についてワーストケースのスタック使用量を計算するには、関数呼び出し階層の各リーフ（末端部）において `GetStackOffset()` を呼び出す必要があります。また、これらの関数から返される最大値も算出する必要があります。

ライブラリ関数がリーフとなる場合は、その親関数で `GetStackOffset()` を呼び出して、そのライブラリ呼び出しのワーストケースのスタックスペースを明らかにする必要があります。RTA-OSEK コンポーネント API のワーストケースの所要スタックスペースは、ご使用のターゲットの『RTA-OSEK バインディングマニュアル』に記載されています。

呼び出し階層のリーフから他のライブラリを呼び出している場合は、そのライブラリのベンダーに問い合わせてライブラリ呼び出しのワーストケースの所要スタックに関する情報を入手してください。

コード例 16-15 に、複数の関数呼び出しを行うタスクの例を紹介します。この例を見ると、スタック使用量を測定するために必要な `GetStackOffset()` の呼び出しをどこにどのように配置すればよいかがわかります。

```
StackOffsetRefType Measurement1;
StackOffsetRefType Measurement2;
StackOffsetRefType Measurement3;

void f1(void) {
   ...
   GetStackOffset(&Measurement1);
   ActivateTask(TaskB);
   ...
}
```

```
void f2(void) {
   ...
   f3();
   GetStackOffset(&Measurement2);
   memcpy(x,y);
   ...
}
void f3(void) {
   ...
   GetStackOffset(&Measurement3);
   ...
}
TASK(Task3) {
   f1();
   f2();
   TerminateTask();
}
```

コード例 16-15 スタック使用量を測定する

コード例 16-15 の場合、ワーストケースのスタック使用量は以下にあげる 3 つの値のうちの最大値になります。

1. `Measurement1` + （RTA-OSEK コンポーネント `ActivateTask()` の所要スタックスペース）
   – （プリエンプトされるタスク用のスタックオフセット）
   – （OS コンテキスト用のスタックスペース）
2. `Measurement2` + （C ライブラリ `memcpy` の所要スタックスペース）
   – （プリエンプトされるタスク用のスタックオフセット）
   – （OS コンテキスト用のスタックスペース）
3. `Measurement3` – （プリエンプトされるタスク用のスタックオフセット）
   – （OS コンテキスト用のスタックスペース）

タスクに必要なスタックスペースをプリエンプション時のスタックサイズを除外して測定する最も簡単な方法は、割込みをディセーブルにして各タスクを分離して実行するというものです。

通常は、`StartOS()` の直後に `GetStackOffset()` を呼び出すことによりスタックポインタをベースラインに戻してから、測定を行ってください。

しかし、この方法は、`StartOS()` からコントロールが戻される場合に限り機能します。コントロールを戻さない、自動的に起動されるタスクがある場合は、`StartOS()` からはコントロールが戻ってこないので、ベースライン値はリセットされません。

このような場合は、他の何らかの方法でスタックをベースラインに戻す必要があります。これにはたとえば、`StartOS()` を呼び出す前にスタックポインタの値を記録しておく、といった方法があります。

16.8.2 監視

AUTOSAR OS に含まれるスタックオーバランの監視機能は、RTA-OSEK のすべてのビルドレベルで利用できます。

この「スタックモニタリング」が有効になっていると、RTA-OSEK はコンテキストの切り替えが発生するたびに、スタックが定義された範囲を超えていないかを調べます。

> **重要**
> スタックモニタリングを有効にすると、スケジューラビリティ分析を行えなくなります。これは、スタックオーバーランが発生するシステムにおいては正しい時間管理が行えないためです。このため、デバッグ目的でスタックモニタリングを行う必要がある場合は、先にモニタリングを行ってアプリケーションが正しく動作することを確認した後にスケジューラビリティ分析を行うようにしてください。

スタックモニタリングにおいては、以下のいずれかの処理から選択できます。

- `ShutdownOS()` を呼び出す

  AUTOSAR OS で定義された処理です。スタック障害が発生した際、RTA-OSEK は自動的に `ShutdownOS()` を呼び出します。`ShutdownHook()` を設定していれば、通常どおり、これが呼ばれます。

- `StackFaultHook()` を呼び出す

  `StackFaultHook()` は RTA-OSEK 固有のものです。`StackFaultHook()` に渡された引数から、アプリケーションで発生したエラーの程度がわかります。`StackFaultHook()` についての詳細は 16.2 項を参照してください。

スタックモニタリングは、RTA-OSEK のメモリ使用量とランタイムパフォーマンスに影響するため、デフォルトでは無効になっています。有効にするには、**Application → OS Configuration** を選択してください。図 16-9 にその方法を示します。

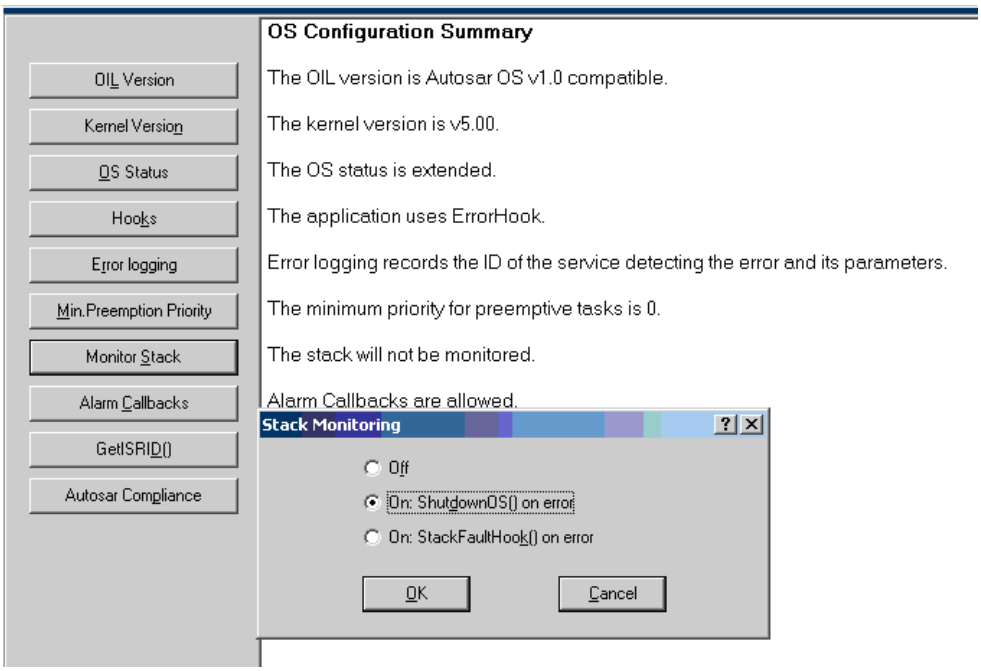


図 16-9 スタックモニタリングを有効にする

RTA-OSEK のコンテキスト用のスタックサイズはすでにツールに組み込まれているため、スタックモニタリングを設定する際にそのサイズを定義する必要はありません。スタックアロケーションとして定義が必要な情報は、タスクとカテゴリ 2 ISR のワーストケースのスタック使用量のみです。

RTA-OSEK では、以下の 3 つのレベルでスタックアロケーションを定義できます。

1. タスク／ ISR のデフォルト設定
2. タスク／ ISR ごとのコンフィギュレーション
3. 実行プロファイル情報

拡張タスクのスタック使用量の設定も同じ方法で行え、各レベルの設定内容の優先度は図 16-9 のようになります。

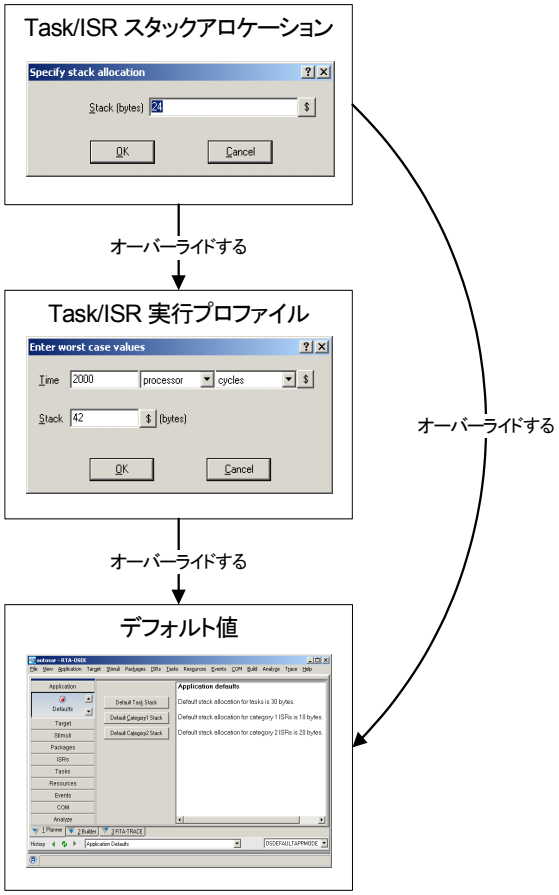


図 16-10 スタックアロケーションのオーバライド優先度（一番上が最も優先的に使用されます）

デフォルトの設定

スタックアロケーションのデフォルト設定は、全タスクとカテゴリ 2 および 1 の ISR に適用されます。設定方法は図 16-11 のとおりです。ここ以外にスタックアロケーションが設定されていない場合、RTA-OSEK はこのデフォルト設定を使用します。

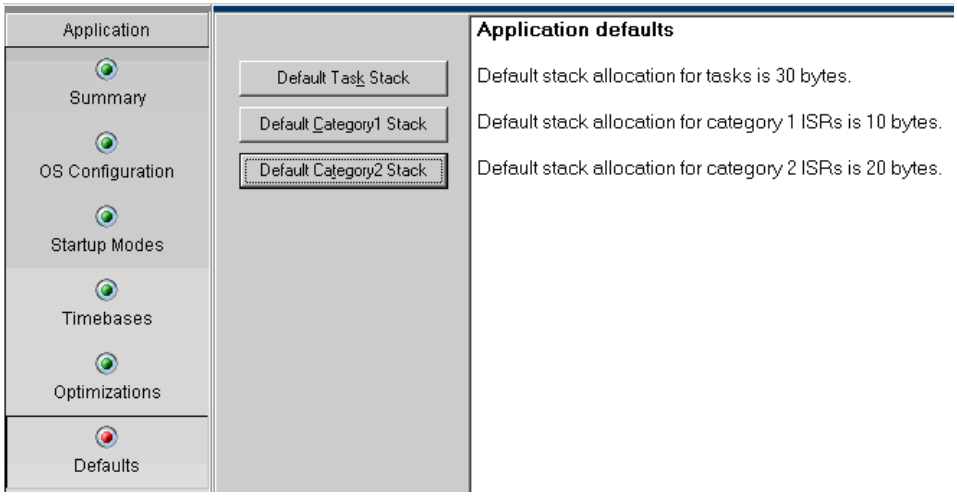


図 16-11 デフォルトのスタックアロケーションを設定する

各タスク／ISR のスタックアロケーションの設定

個々のタスクと ISR のコンフィギュレーションの一部として、スタックアロケーションを設定することができます。図 16-12 と図 16-13 を参照してください。ここに設定された値は、前項のデフォルト設定をオーバーライドします。

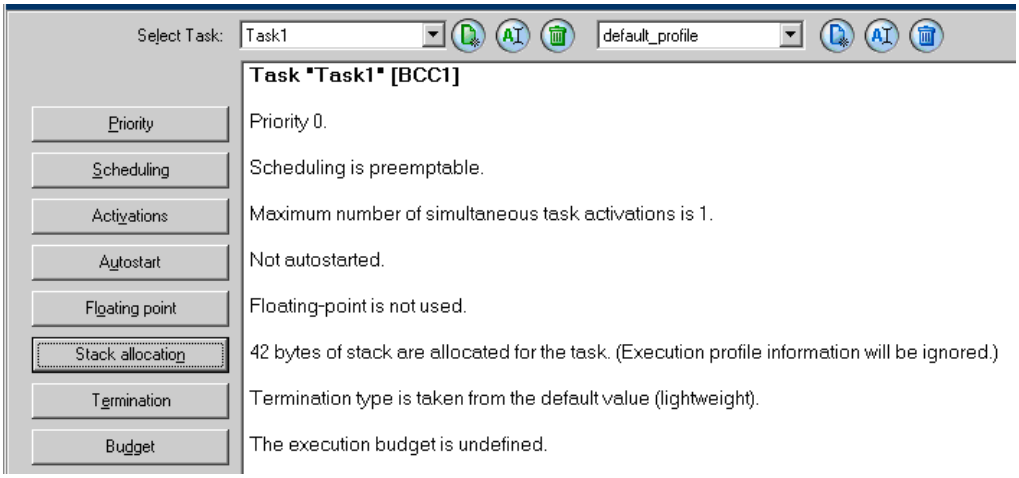


図 16-12 タスクのスタックアロケーションを設定する

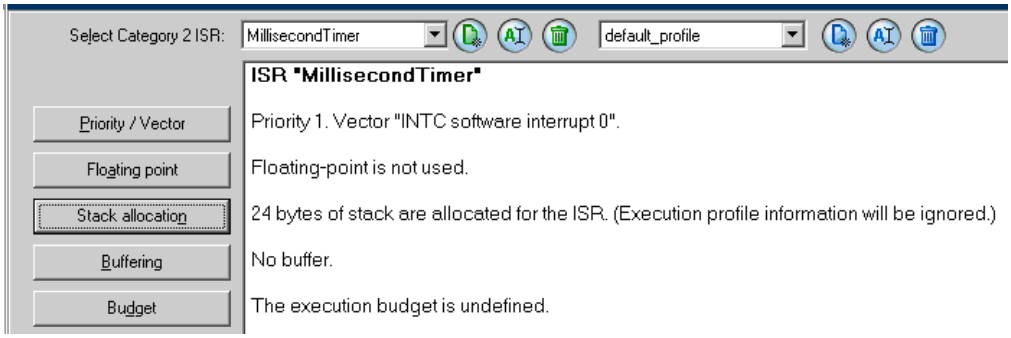


図 16-13 カテゴリ 2 ISR のスタックアロケーションを設定する

ここでスタックアロケーションが指定されている場合、その値は、実行プロファイルから自動的に算出される値をオーバーライドします。

実行プロファイルから自動的に算出されるスタックアロケーション

スタックアロケーションを指定するための第 3 の方法として、実行プロファイル情報を使用する方法があります。実行プロファイル情報は、デフォルトのスタックアロケーション設定をオーバーライドします。

タイミングモデルをビルドして RTA-OSEK Planner のスケジューラビリティ分析機能を利用する際は、分析に適したスマートなスタックアロケーションを使用することができます。

各タスクの実行プロファイルにはワーストケースの実行時間とスタック使用量が含まれています。タイミングモデルにはタスクや ISR 用に複数のプロファイルを含めることができ、タスクや ISR が呼び出されるタイミングに応じて異なる実行時間やスタック容量を適用することができます。

RTA-OSEK は、実行プロファイル情報を使用して必要なスタックアロケーションを算出します。

16.9 コンパイル時のエラーの捕捉

前項まではランタイムのエラーチェックについて説明されていますが、コンパイルの時点でエラーをできるだけ除去しておくことも重要です。RTA-OSEK の静的インターフェースが使用されていると、コンパイル時のエラー発見に役立ちます。

静的インターフェースを使用する場合、個々のタスクまたは ISR は、それらに対応する特定の API 関数とオブジェクトしか使用できません。そのため、API コールの妥当性をあらかじめ RTA-OSEK がビルド工程の中で確認できることになります。

また無効な引数を使用しようとするとコンパイラがそれを検知するので、デバッグの時間を大幅に短縮できる可能性があります。

16.10 まとめ

- OSEK にはフックメカニズムを通じてデバッグを行う機能があります。
- スタートアップフック、シャットダウンフック、プリタスクフック、ポストタスクフックを使って、ユーザーのアプリケーションの実行状態をランタイムに追跡することができます。
- エラーフックには、ランタイムに例外条件をトラップするメカニズムがあり、例外処理の再開モデルを用意することもできます。
- エラーの原因についての詳細情報は、エラーフック内からアクセスできるマクロを通じて得ることができます。
- RTA-OSEK は、AUTSAR OS のスタックモニタリングに加え、さらに高度なスタック監視機能をサポートしています。
- RTA-OSEK コンポーネントは、さらに実行時間とスタック使用量の監視もサポートしています。RTA-OSEK コンポーネントのタイミングビルドと拡張ビルドを使うと、ユーザーコードの実行時間を測定することができます。

17 タイミングモデルのビルド

前章までに、システムがどのようにスティミュラスを受け取りレスポンスを生成するか、また、システムの挙動を定義したスティミュラス - レスポンスモデルが RTA-OSEK にどのように用いられるか、またさらにユーザーが設計工程でそれらのモデルをどのように定義すればよいかが説明されています。

リアルタイムシステムを構築する場合、このモデルにはさらに別の条件が必要です。それは、リアルタイムパフォーマンスに関する要求仕様を定義する、ということです。

「リアルタイム（'real-time'）」という言葉の意味は「リアルファースト（'real-fast'）」、つまり非常に速く処理されることであると誤解されることがありますが、実際の「リアルタイムシステム」は、スティミュラスが到達した時に常にそれに対応するレスポンスが所定の時間内に生成されるシステムを指します。

レスポンスの生成を、スティミュラス到達後何分、何時間、あるいは何年以内で行わなければならないかは、状況に応じて異なりますが、時間の長さに関わらず、レスポンスが必ず所定の時間内に生成されるのであれば、そのシステムは「リアルタイム」であるといえます。

この「遅くともその時間までにはレスポンスが生成されていなければならない時間」を、**デッドライン**と呼びます。図 17-1 は、スティミュラス、レスポンス、デッドライン、および周期の関係を図解したものです。

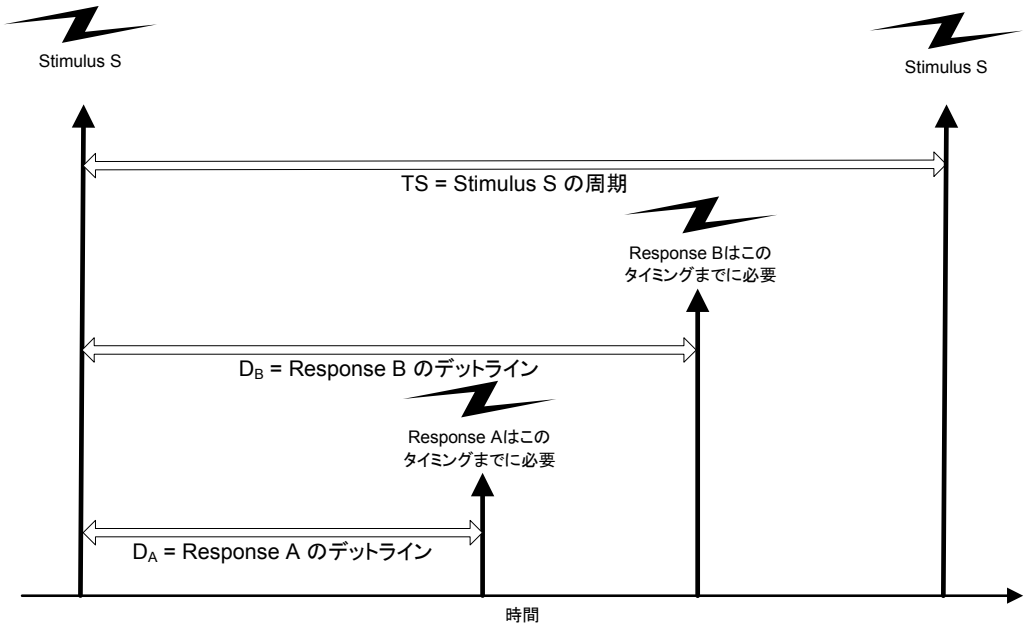


図 17-1 スティミュラス - レスポンスモデル

レスポンスを生成するためのコードがデッドラインまでにレスポンスを生成し終えると、このレスポンスはデッドラインに間に合ったことになります。レスポンス生成までにかかる時間は「レスポンスタイム」と呼ばれます。レスポンスがデッドラインより前に生成されることにより、デッドラインは守られます。

図 17-2 はレスポンスを生成するタスク、つまりレスポンスを実装したタスクを示しています。この例では、タスク t1 は最後にレスポンス r1 を生成しています。

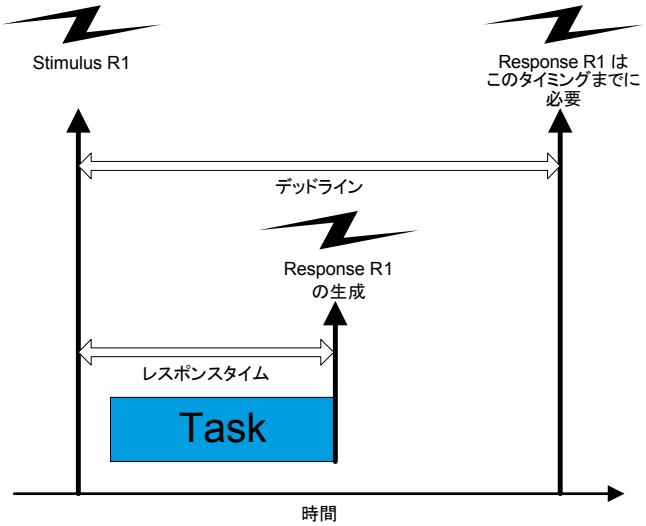


図 17-2 タスク内でレスポンスを生成する

レスポンスは必ずしもタスクの最後において生成される必要はなく、また、1 つのタスク内に複数のレスポンスを実装することができます。図 17-3 はタスクの前半部分でタスクレスポンスが生成される例を示しています。

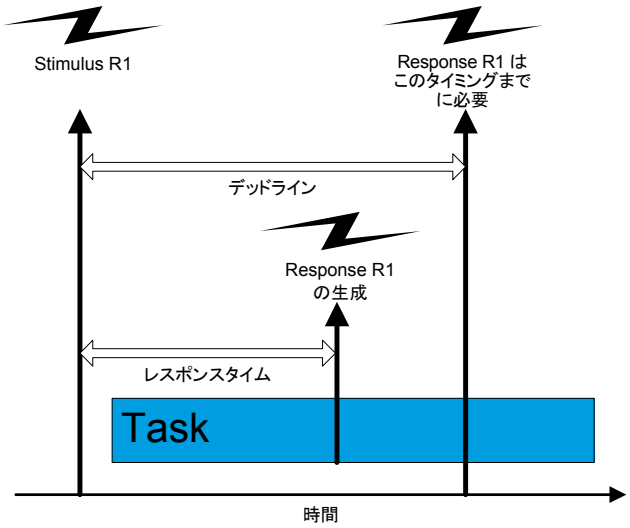


図 17-3 デッドライン満了前にレスポンスを生成する

リアルタイムシステムにおいては、すべてのレスポンスがそれぞれのデッドラインより前に発生する必要があります。RTA-OSEK GUI では、RTA-OSEK Planner を使用して、アプリケーション内の各レスポンスについてワーストケースのレスポンスタイムを算出し、すべてのレスポンスがデッドラインに間に合うかどうかを調べます。

各タスク内の各レスポンス実装について、ワーストケースレスポンスタイムは以下の項目で定義されます。

- ワーストケースの実行時間
  プリエンプションがないと仮定した上での、タスクの実行開始からターミネートまでの最長時間です。
- 「妨害」（'interference'）
  タスクがシステム内の自分より優先度の高いタスクや ISR によりプリエンプトされる最長時間です。
- ブロッキング
  タスクの実行が自分より優先度の低いタスクにより妨げられる最長時間です。

RTA-OSEK Planner はそれぞれのタスクまたは ISR が受けるプリエンプションの時間を計算します。この計算には、レスポンス実装のワーストケースの実行時間と、さらに、リソースが使用されている時間と割込みがディセーブルになっている時間（ブロッキング時間）が使用されます。

つまり、RTA-OSEK Planner はワーストケースのレスポンスタイムを、レスポンスのワーストケースの実行時間とブロッキング時間を使用して算出できます。

リアルタイムシステムを分析するために、ユーザーは以下の情報を入力する必要があります。

- ソフトウェアアーキテクチャ（タスク、割込み、タスクセット、リソース、カウンタ、アラーム、スケジュール）の定義。
- 実行可能オブジェクト間のタイミング関係を定義するスティミュラス - レスポンスモデル。これはアプリケーションの周期とデッドラインを定義するものです。
- それぞれのタスクと ISR の実行時間。
- ターゲット固有のタイミング情報。

ソフトウェアアーキテクチャの定義方法については前章までに説明されています。ここでは、スティミュラス - レスポンスモデル、実行時間、およびターゲット固有のタイミング情報の定義方法について説明します。

システムを分析用にモデリングするためにデータを入力する場合、守らなければならない重要な原則が 2 つあります。

- 正確に
  できるだけ正確なデータを入力することが重要です。
- 「悲観的に」
  正確なデータを入力できない場合は、「悲観的な（'pessimistic'）」データ、つまり考えられる値よりも厳しい値を入力する必要があります。たとえば、実際の実行時間よりも長い実行時間を入力したり、スティミュラス間の遅延を実際の遅延よりも短めに宣言するようにしてください。

重要

分析用のデータを入力する時には、実行時間を短めに見積もったり最短周期を長めに見積もったりしないように注意してください。そのようにしてしまうと、RTA-OSEK はユーザーのシステムが実際にはアンスケジューラブルであるのにかかわらず、スケジューラブルであると結論付けてしまう可能性があります。

17.1 アプリケーション分析用の設定

アプリケーションをタイミング分析用にビルドしたい場合は、以下のルールに従う必要があります。

- 上方向へのタスク起動は受け付けられません（タスクは自分より優先度の低いタスクのみ起動できます。）
- 各タスクに固有の優先度を割り当てる必要があります。
- リスケジューリングを強制的に実行するための API 関数 `Schedule()` を使用することはできません。
- AUTOSAR スケジュールテーブルは使用できません。

RTA-OSEK GUI の Application Optimizations を用いてこれらのルールを徹底させることができます。図 17-4 を参照してください。

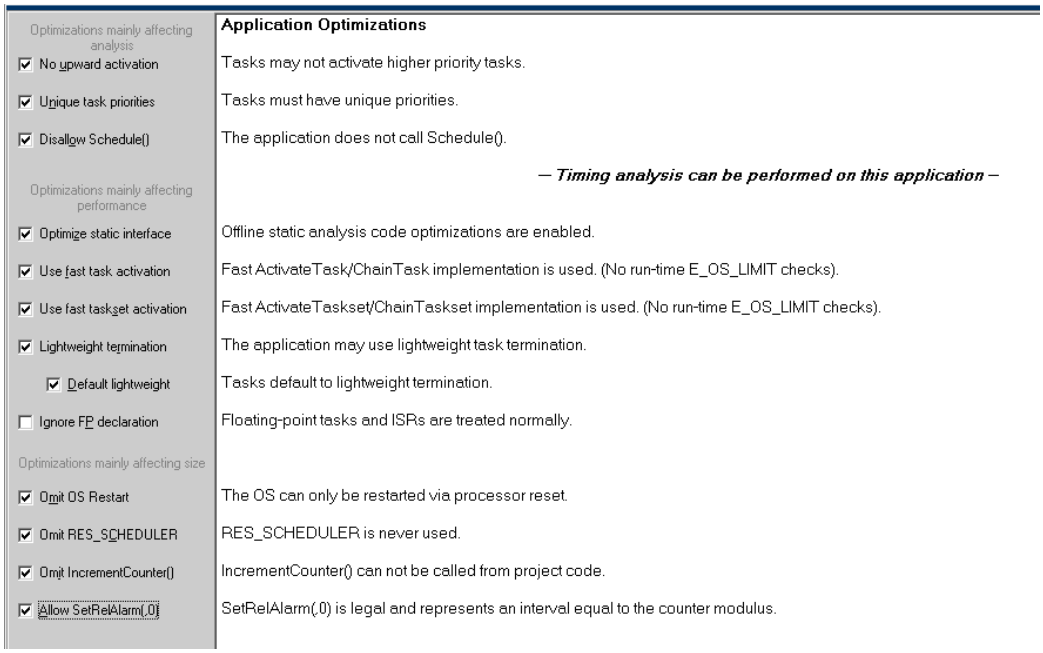


図 17-4 Application Optimization の設定

システムを分析しようとした時に、もしもアプリケーションが分析に適していないと、RTA-OSEK Planner はユーザーにその旨を通知します。

17.2 スティミュラス - レスポンスのタイミング関係の定義

RTA-OSEK GUI を使用して、スティミュラス - レスポンスモデルを作成してアプリケーションをビルドする方法については、前章までに説明されていますが、ここでは特に、タイミング分析のための予備情報を追加する方法についても説明します。

17.2.1 スティミュラスのアライバルタイプとアライバルパターンについて（復習）

前述されているように、各スティミュラスにはいずれかのアライバルタイプが割り当てられています。アライバルタイプはスティミュラスのクラスを定義するものです。

アライバルタイプには以下の 3 種類があります。

- バースト（'bursty'）
- 周期的（'periodic'）
- 計画的（'planned'）

バーストスティミュラスは、スティミュラスが割込みにより直接捕捉されるような単純なケースをモデリングするために使用されます。周期的スティミュラスと計画的スティミュラスは、スティミュラスがカウンタにアタッチされているアラームによりモデリングされる場合やスケジュールのアライバルポイントによりモデリングされる場合など、複雑なアライバルをモデリングするために使用されます。

各タイプのアライバルは、明確なアライバルパターンを持っています。これらのアライバルパターンについて以下に詳しく説明します。

17.2.2 バーストアライバルパターン

バーストアライバルパターンについて、ユーザーは「バーストクローズ」('bursting clause'：バースト節）と呼ばれる一連のルールを定義することができます。「クローズ」は、スティミュラスのアライバルパターンについて定義するものです。

シンプルなバーストアライバルパターンでは、周期的タイマ割込みのアライバルを定義できます。図 17-5 では、10ms の周期的割込みについてのバーストアライバルが定義されています。この場合、バーストクローズが 1 つだけ使用されています。

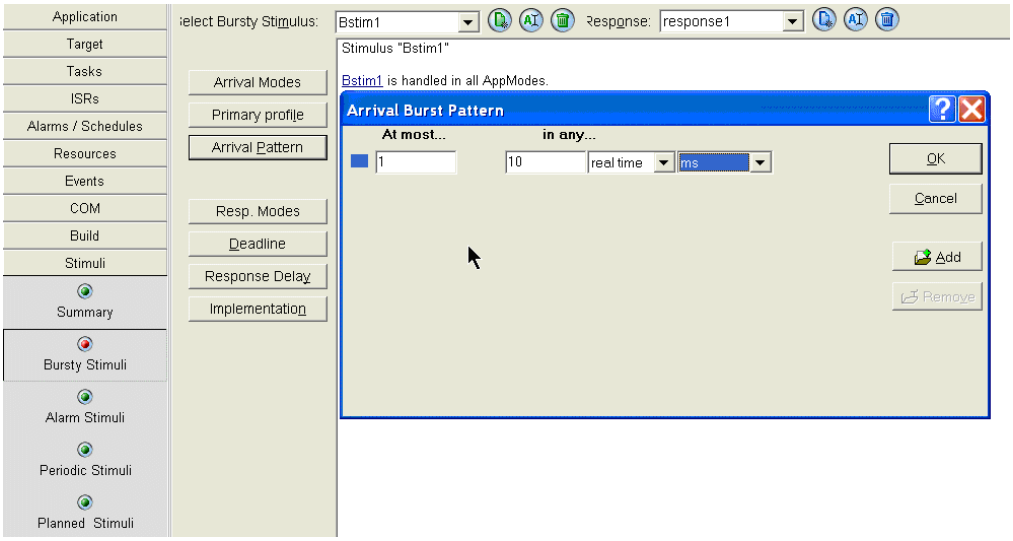


図 17-5 1 つのバーストクローズで定義できる場合

図 17-6 は、複数のアライバルルールを使用する複雑なバーストアライバルの例を示しています。

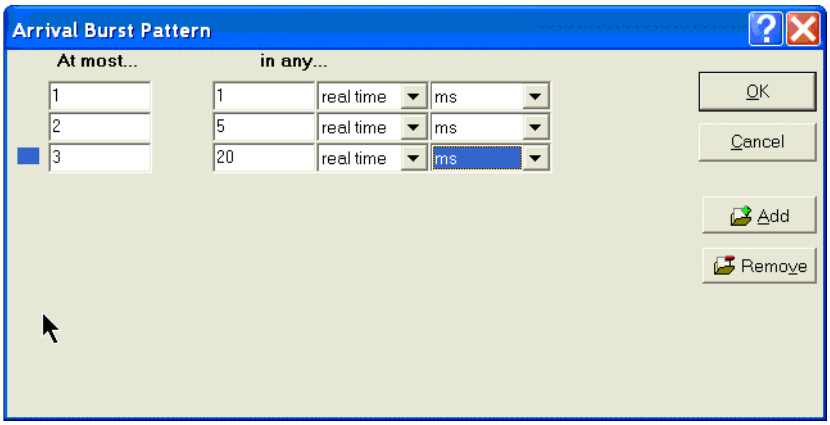


図 17-6 複数のバーストクローズが必要な場合

図 17-6 では、トランザクションのバーストクローズについて、スティミュラスが以下のルールで発生するように定義されています。

- **ルール** 1：1ms の間に多くても 1 回しか発生しない
- **ルール** 2：5ms の間に多くても 2 回しか発生しない
- **ルール** 3：20ms の間に多くても 3 回しか発生しない

これらのルールを組み合わせると、以下のようなワーストケースのアライバルパターンができます。

- 0ms、1ms、2ms、3ms...（ルール 1 で許されている最短のアライバル間隔は 1ms です。）

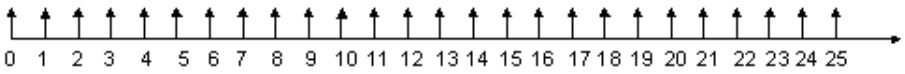


図 17-7 ルール 1 のアライバルパターン

- 0ms、1ms、5ms、6ms、10ms、11ms...（ルール 2 では、5ms の間に 2 回より多い回数のアライバルは許されないので、2 つのバーストが 5ms 周期ごとに分けられています。）

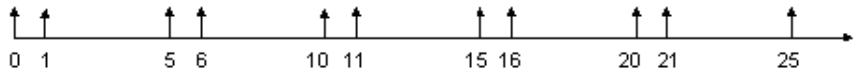


図 17-8 ルール 2 のアライバルパターン

- 0ms、1ms、5ms、20ms、21ms、25ms...（ルール 3 では、20ms の間に 3 回より多い回数のアライバルは許されません。）

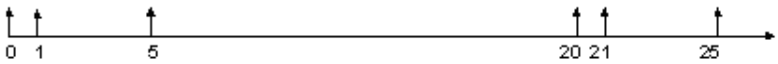


図 17-9 ルール 3 のアライバルパターン

2 つ以上のアライバルルールが定義されている場合、あるルールで許容された値を対象にして次のルールを適用していきます。値が昇順に並んでいる場合は、連続する値のペア（アライバルと 間隔）は前のペアよりも大きくなければなりません。アライバルのレート（つまりアライバル÷間隔）は厳密に減少していなければなりません。

上の例についてみてみると、以下のようになっていることがわかります。

- 1 回＜ 2 回＜ 3 回
- 1ms ＜ 5ms ＜ 20ms
- 1/ms（1ms に 1 回）＞ 0.4/ms（5ms に 2 回）＞ 0.15/ms（20ms に 3 回）

一般に、定義されるバーストクローズが多ければ多いほど、分析の「悲観性」は低くなります。しかし、バースト間隔がシステムの最長ビジー期間よりも長い場合は、そのアライバルルールを定義しても何のメリットもありません。したがって、この例では、システムがアイドルタスクの稼働前に長くても 20ms しか稼働しないことがわかっている場合は、ルール 3 を定義しても分析の精度は上がりません。

アイドルタスクはシステム内で優先度が最低のタスクであり、他のすべてのタスクと ISR がサスペンド状態か待ち状態になっている時にしか実行状態になりません。

システムの稼働サイクル中に許容されるアライバル数は、有限個に限定されます。稼働サイクルとは、システムが起動してからリセットされるまでの時間です。この場合、'forever'（無限）という期間を使用してアライバルの数を制限することができます。

'1 times in forever' というバーストクローズは、そのイベントのアライバルがシステムの稼働サイクル中に 1 回しか発生できないことを意味しています。これを使用して、自動車のエアバッグのような 1 回限りの安全装置のトリガを表すことができます。このクローズは、図 17-10 のようにして定義します。

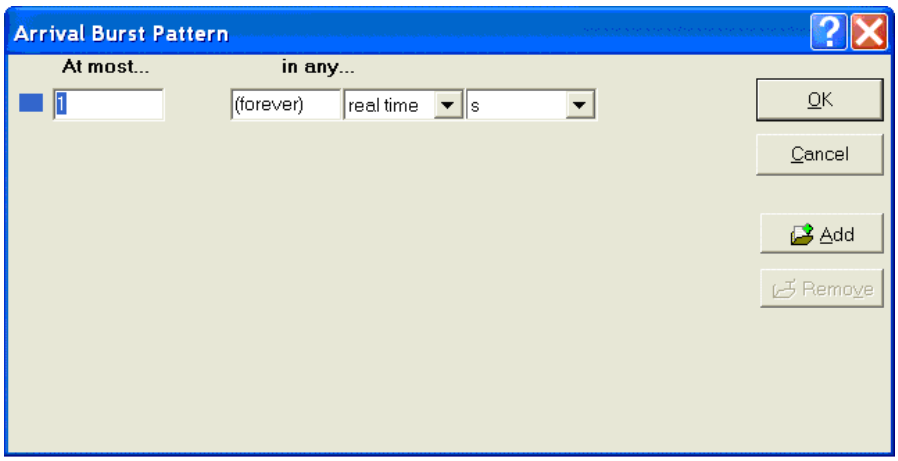


図 17-10 'One Time in Forever' というバーストクローズを定義する

17.2.3 周期的アライバルパターン

周期的アライバルパターンは、スティミュラスがどのくらいの頻度でアライバルポイントに到達するかを定義します。この情報は、アラームやスケジュール周期などのランタイム情報の生成に必要で、分析用に暗黙的なデッドラインを設けるためにも使用されます。

17.2.4 計画的アライバルパターン

計画的アライバルパターンは、スティミュラス - レスポンスモデリングの段階では定義されません。このアライバルは、設計でプランを作る時に計画されます。RTA-OSEK Planner はプラン上のタイミングを使用し、スティミュラスの相対周期とそれらのスティミュラスに関連するレスポンスの暗黙的デッドラインを算出します。

17.2.5 レスポンスのデッドラインの設定

レスポンスには暗黙的にデッドラインが想定される可能性があります。たとえば、20ms の周期的スティミュラスは、20ms に 1 回レスポンスを生成する必要があると考えられます。

またレスポンスには、明示的なデッドラインを定義することもできます。たとえば、20ms 周期のスティミュラスについて、アライバル後 10ms 以内にレスポンスが生成されなければならない、といった内容です。

図 17-11 に、明示的なレスポンスデッドラインの例を紹介します。

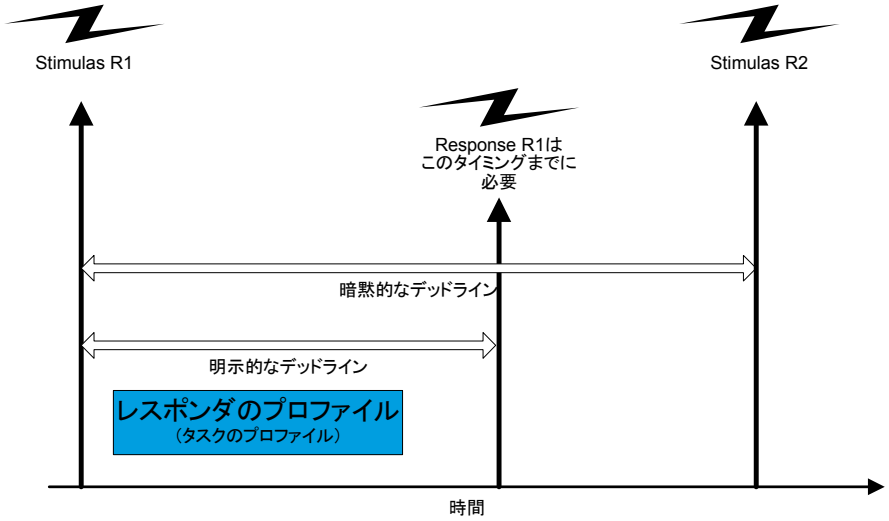


図 17-11 明示的なレスポンスデッドライン

すべてのレスポンスには、そのレスポンスがタスクで生成されたものであるか、ISR で生成されたのであるかに応じて、またさらに実行可能オブジェクトのプロパティに応じて、暗黙的なデッドラインがあります。

タスクは次の起動より前に、あるいは起動がキューイングされるタスクの場合はキューが満杯になる前に、完了しなければなりません。ISR も、その割込みをバッファリングするようにユーザーが定義していない場合は、次にトリガされる前に完了しなければなりません。

レスポンスデッドラインを定義することにより、タイミングパフォーマンス上の制約をさらに設けることができます。デッドラインは、レスポンスがいつまでに生成されなければならないかを、スティミュラスが発生してからの経過時間で表したものです。図 17-12 のようにして、RTA-OSEK GUI で明示的デッドラインを定義できます。

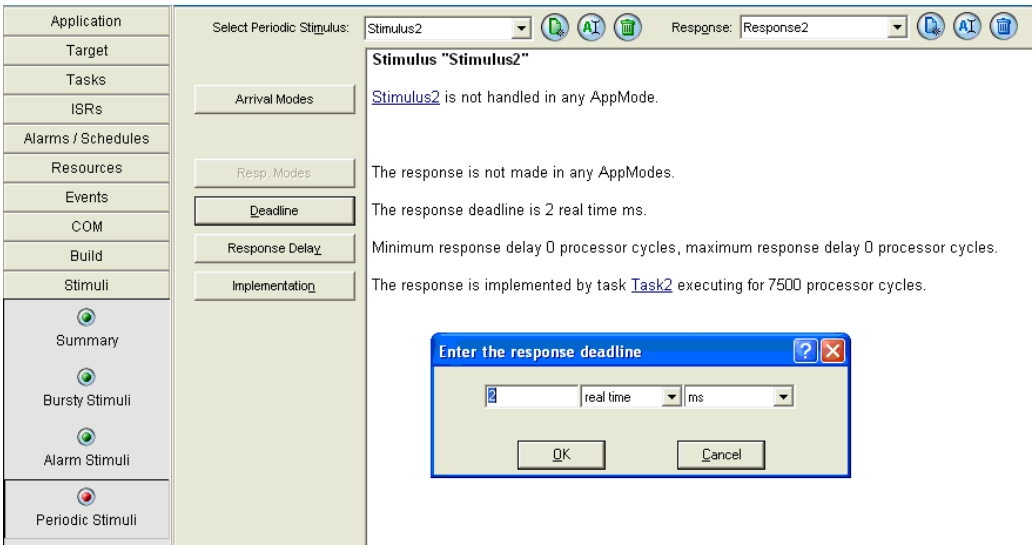


図 17-12 レスポンスデッドラインを定義する

17.2.6 レスポンス生成時間を定義する

レスポンスの実装は、タスクまたは ISR 内で行われます。デフォルト状態において RTA-OSEK GUI は、タスクまたは ISR がターミネートする時にレスポンスが生成されると想定します。

しかし、これはスケジューラビリティ分析の結果を悪くすることにつながります。一例として、スティミュラスが 10ms ごとに発生し、それに関連付けられているレスポンスがその 1ms 後に生成されなければならない場合について考えてみましょう。

レスポンスを生成するタスクが 2ms の間実行されるものである場合は、このシステムはスケジューラブルではありません。つまりそのタスクがレスポンスデッドラインより前に完了することは不可能です。しかし、そのタスクが 0.5ms 実行されたところでレスポンスが生成されるようになっていれば、レスポンス生成時間としてタスク起動後 0.5ms という値を設定することができます。そうすると、デッドラインに間に合わせることができます。

したがって、この例では、このタスクの実行について暗黙的なデッドラインと明示的なデッドラインがあることがわかります。

つまり、「スティミュラスのアライバルから 1ms」という明示的デッドラインと、「タスクがスティミュラスに対応できるように、タスク周期は 10ms 未満ででなければならない」という暗黙的なデッドラインです。

各レスポンスの実装を行う際、レスポンスが生成されるまでの実行時間を定義することができます。図 17-13 のように行います。

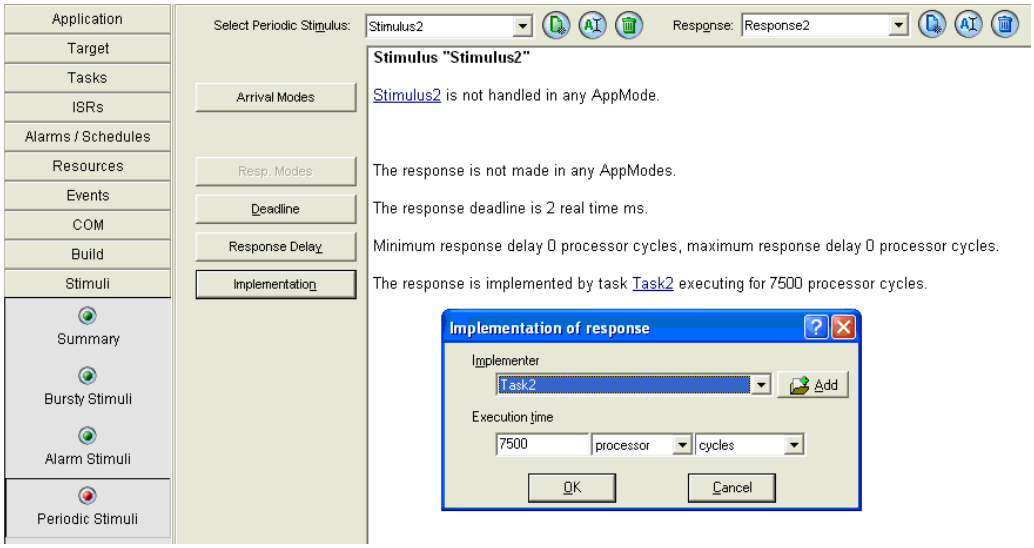


図 17-13 レスポンス生成までに要する実行時間を定義する

17.2.7 ジッタのモデリング

組み込みリアルタイムシステムを分析する場合、タイミング数値が、「組み込みシステム」のスティミュラスやレスポンスを正しく表わしていることが重要です。RTA-OSEK Planner では、システムがスティミュラスを受信してからレスポンスを生成するまでの時間と、システムがスティミュラスを処理してからレスポンスを生成するまでの時間との差をモデリングすることができます。

場合によっては、実際のスティミュラスが発生してから、そのスティミュラスに対するプライマリプロファイルがシステム内で発行されるまでに遅延が生じることがあります。この遅延の原因としては、スティミュラスの検知に使用されているハードウェア（A/D コンバータなど）の処理速度が遅いなどのさまざまなものが考えられます。

スティミュラスの「認識時間」（'recognition time'）は、スティミュラスの発生を認識できる最も早い時間と最も遅い時間の間の時間です。これを図 17-14 に図解します。

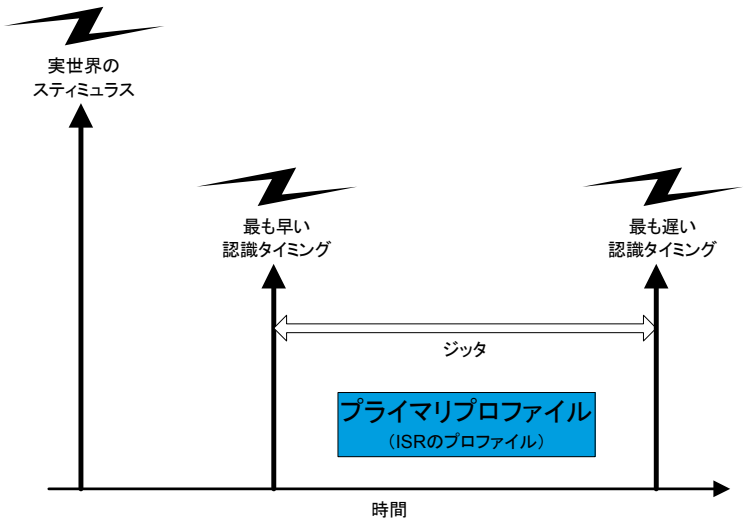


図 17-14 認識時間

最大認識時間と最小認識時間の差は、「入力ジッタ」（'input jitter'）と呼ばれます。入力ジッタはアプリケーション内の各プライマリプロファイルごとに定義できます。

たとえば、あるプライマリプロファイルにより処理される実世界のすべてのスティミュラスの最小レスポンス遅延が 170ns で最大レスポンス遅延が 220ns である場合、このスティミュラスには 50ns のジッタ（220ns − 170ns ＝ 50ns）を定義します。

同様に、レスポンスの「出力ジッタ」（'output jitter'）も定義できます。これはたとえば、アクチュエータを駆動する際に物理デバイスのヒステリシスを考慮する必要がある状況をモデリングするためのものです。

各レスポンスについて最小／最大の「レスポンス遅延」（'response delay'）を定義することができます。これは、レスポンスが生成されるべきデッドラインまでに許される経過時間を指定するものです。

図 17-15 にレスポンス遅延を図解します。

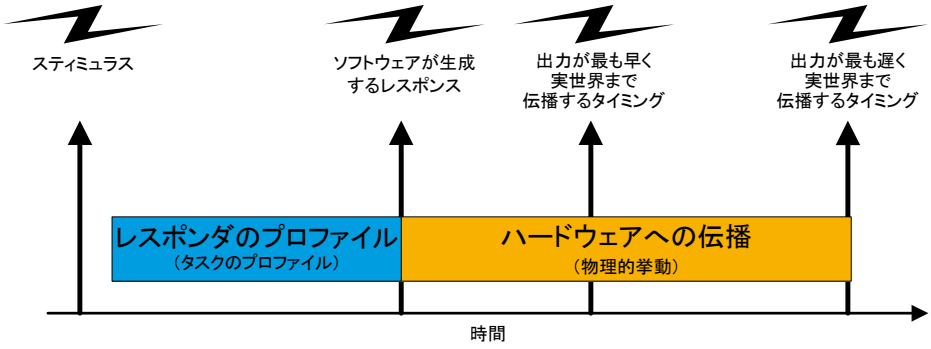


図 17-15 レスポンス遅延

図 17-16 では、`EnergizeCoil` というレスポンスとそのデッドライン（70ms）、最小レスポンス遅延（20ms）、最大レスポンス遅延（50ms）が定義されています。

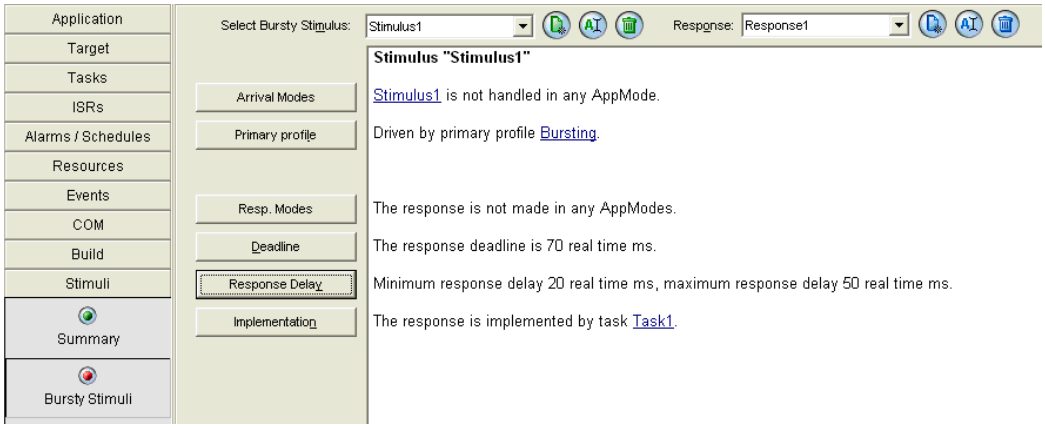


図 17-16 デッドラインとレスポンス遅延を定義する

重要

レスポンス遅延や最小／最大認識時間が定義されていないと、デフォルト値の 0 が使用されます。

17.3 実行情報の捕捉

アプリケーションについてタイミング分析を実行したい場合は、システム内の各タスクと ISR の実行時間を知る必要があります。

この値は、CPU サイクルをカウントすることにより、または静的なタイミング分析ツールを使用することにより、静的に算出することができます。またそれとは別に、RTA-OSEK コンポーネントのタイミングビルド（'Timing build'）を使用して実行時間を測定するという方法もあります。タイミングビルドの詳細については本書でも後述します。

各タスクと ISR についてワーストケースのスタック使用量を入力すると、RTA-OSEK GUI でワーストケースのスタック使用量を算出することができます。それぞれのタスクと ISR のワーストケースのスタック使用量についての情報は、多くの場合。コンパイラから得ることができます。

タスクと ISR の実行特性は、**実行プロファイル**（'execution profile'）に宣言します。分析の際は、それぞれのタスクと ISR ごとに少なくとも 1 つの実行プロファイルを定義する必要があります。複数のプロファイルを使用することもできますが、その場合の詳細については 14.6 項で説明します。

実行プロファイルは、それぞれのタスクや ISR のワーストケースの実行時間およびワーストケースのスタック使用量を宣言するものです。通常、ワーストケースの実行時間は実行されるコードの量により決まるので、プロセッササイクル単位で測定されます。つまり、CPU クロックレートを変更すると、タスクや ISR の実行時間は自動的にスケーリングされます。

ただし不確定な処理を行うタスクは、このルールの例外となります。このタイプのタスクは、ある一定の時間、所定の値が得られるまで実行を続けます。このため、実行時間は実際の時間単位で表現してください。ワーストケースのスタック使用量はバイト数で定義されます。

17.3.1 プライマリプロファイルとアクティベーテッドプロファイル

アプリケーション内のそれぞれのタスクと ISR は、1 つ以上の「プロファイル」が割り当てられます。プロファイルは、以下の目的で使用されます。

- タスクまたは ISR についての実行情報（タイミング情報とスタック使用量の情報）を捕捉するため。
- タスクまたは ISR をスティミュラスの捕捉またはレスポンスの生成に使用できるかどうかを示します。スティミュラスの直接的な捕捉、カウンタ駆動、スケジュール駆動を行うタスクまたは ISR は「プライマリプロファイル」で、そうでないものは「アクティベーテッドプロファイル」です。

RTA-OSEK は、デフォルトでは、すべての ISR がプライマリプロファイルで、すべてのタスクはアクティベーテッドプロファイルであると仮定します。

まれに、この設定の変更が必要な場合があります。たとえば、ISR ではなくタスクを使用してカウンタやスケジュールを駆動する必要がある場合は、タスクをプライマリプロファイルにする必要があります。

プロファイルを 1 つしか持たないタスクや ISR の場合、タスクや ISR の名前を用いてプロファイルにアクセスします。複数のプロファイルがある場合（これについては 14.6 項で詳しく説明します）には、ドットでタスク名とプロファイル名を結合した名前を使用して各プロファイルにアクセスします。たとえば、`Task1` に `Profile1` と `Profile2` がある場合、プロファイルにアクセスするには `Task1.Profile1` と `Task1.Profile2` という名前を使用します。

アプリケーションでプロファイルを使用するにあたり、以下の制約があります。

- プライマリプロファイルが 1 つだけの場合

  プライマリプロファイルには 1 つのバーストスティミュラス、1 つのアドバンスドスケジュール、あるいは 1 つまたは複数のカウンタとチックスケジュールを関連付けることができます。このうち最初の 2 つに関しては、2 つのバーストまたはアドバンス起動が時間内に発生ことは期待できません。一方、3 つめの場合には、プロファイルは一定速度でチックされるので、各カウンタ／スケジュールのチックレートが互いに異なっていても、複数のカウンタ／スケジュールをチックすることが可能です。

- プライマリプロファイルが複数ある場合

  スティミュラスが実行プロファイルによりバッファリングされる限り、プライマリプロファイルにはさまざまなスティミュラスを関連付けることができます。

- アクティベーテッドプロファイルが 1 つだけの場合

  アクティベーテッドプロファイルには、1 つのプライマリプロファイルを関連付けることができます。

- アクティベーテッドプロファイルが複数ある場合

  すべてのアクティベーテッドプロファイルを 1 つのカウンタ／スケジュールに関連付けるか、または各プロファイルをそれぞれ異なるプライマリプロファイルで駆動する必要があります。

17.3.2 タスクと ISR

タスクと ISR のワーストケースの実行時間とは、タスクのエントリ関数における最初のマシンコード命令の開始から、時間的に最長のパスを通ってリターン命令の終了までの時間を測定したものです。これには、プリエンプションや割込みの影響は含まれません。

カテゴリ 1 の ISR のワーストケースの実行時間としては、キャッシュまたは命令パイプラインの影響が最も「悲観的な（'pessimistic'）」値になっている必要があります。

タスクとカテゴリ 2 の ISR のワーストケースのスタック使用量は、エントリ関数において所得されます。これにはネストしている関数呼び出しシーケンスについてのワーストケースも含める必要がありますが、タスクまたはカテゴリ 2 ISR にエントリする際に必要なスタック容量を含める必要はありません。この分は、分析時に自動的に加算されます。

カテゴリ 1 の ISR 割込みハンドラのワーストケースのスタック使用量には、プロセッサ割込みのスタックフレームと、ハンドラが消費するスタックを含める必要があります。

図 17-17 には、RTA-OSEK GUI にワーストケースの実行時間とスタック使用量を入力する方法が示されています。

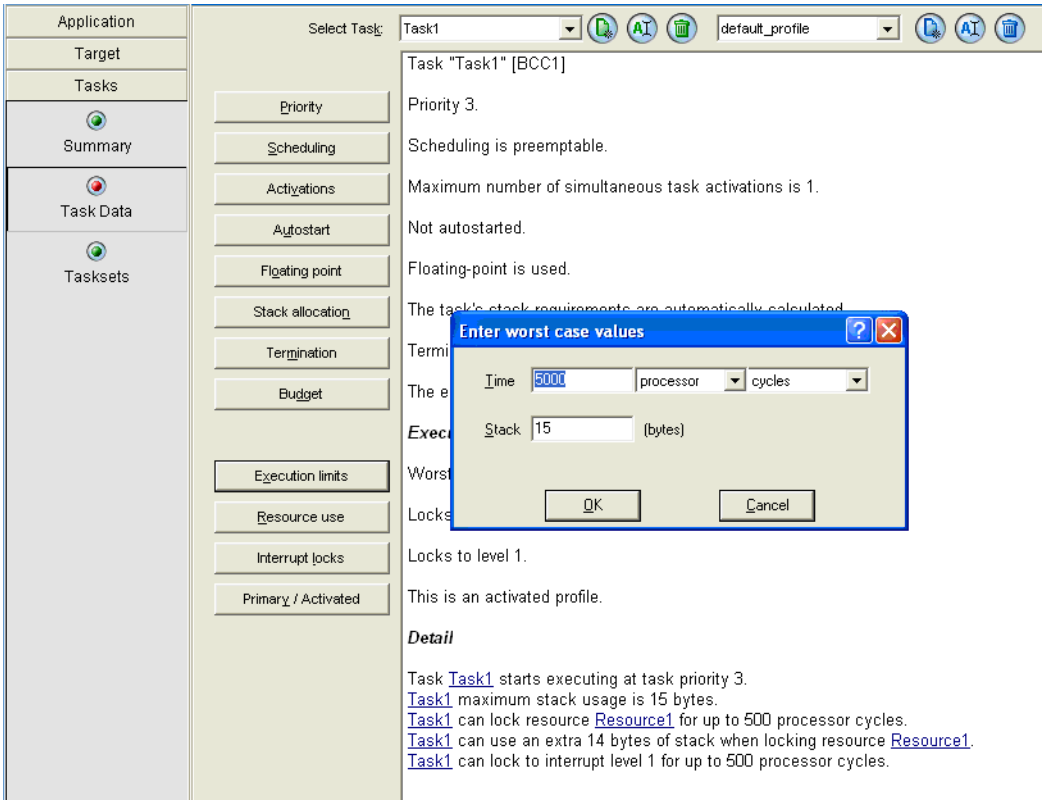


図 17-17 ワーストケースの値を定義する

この例では、タスクはワーストケースにおいて 5000 プロセッササイクルを使用し、15 バイトのスタックスペースを消費します。

17.3.3 アイドルタスクのモデリング

アイドルタスクが最初の `StartOS(OSDEFAULTAPPMODE)` 以外の RTA-OSEK コンポーネント API 関数を呼び出す場合は、ブロッキングを導入することができます。分析ではこのことについて考慮する必要があります。

アイドルタスクのプロファイルも、他のタスクの場合と同じ方法で定義されます。ただし、対応しなければならないデッドラインがアイドルタスクにない場合は、実行時間の正確な値を定義しても意味がありません。

アイドルタスクのワーストケースのスタック使用量は、通常は C スタートアップコードで設定される初期スタックポインタ値から測定されることに注意してください。

17.3.4 リソースと割込みのロック

リソースを取得したり割込みをディセーブルにするタスクや ISR は、自分より優先度の高いタスクや ISR の実行をブロックすることができます。これを 2 つのタスクが含まれているシステムの例で説明します。`Task1` と `Task2` という 2 つのタスクがリソースを共有していて、`Task2` の方が `Task1` よりも優先度が高いとします。

`Task1` がリソースを所有している時に `Task2` が起動されてレディ状態になると、`Task1` がリソースを解放するまで `Task2` は「ブロック」されます。これを図 17-18 で図解します。

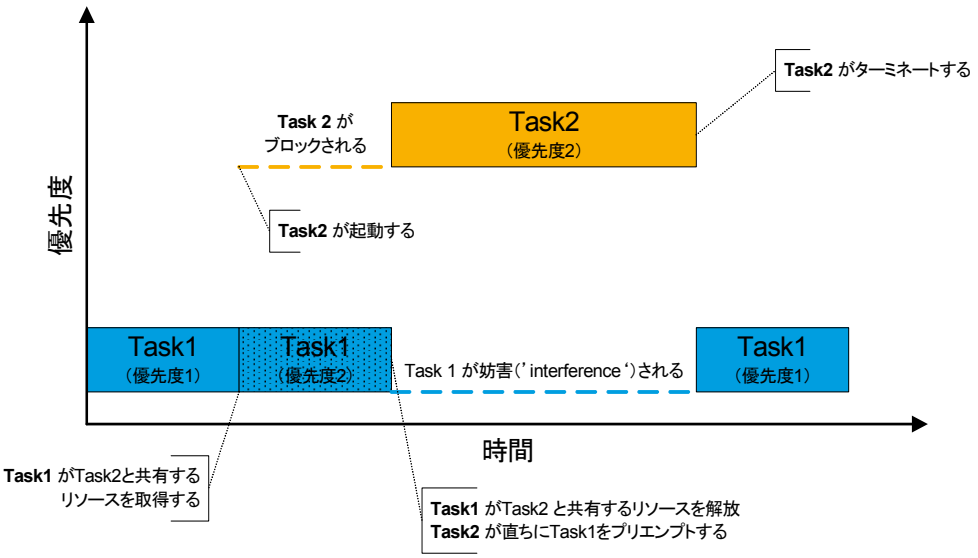


図 17-18 タスクのブロッキングと妨害

アプリケーションがスケジューラブルであるかどうかを判断するために、RTA-OSEK Planner はリソースが保持されている時間と、割込みがディセーブルになっている時間を知る必要があります。

分析時においては、リソースのレスポンスタイムがワーストケースとなる状態が想定されます。つまり、RTA-OSEK Planner は、リソースを取得するタスクまたは ISR の開始を基準としてリソースがどのタイミングで保持されるかということを知る必要はありません。

ロック時間、つまりリソースの使用時間は、RTA-OSEK GUI において、実行プロファイルの「resource use」セクションと「interrupt lock」セクションで定義します。ロック時間は、一般的に、実行時間と同様にプロセッササイクル単位で定義します。

ここで正確な値を入力することにより、分析における「悲観性」の度合いを少なくすることができます。もしもリソースと割込みのロック時間が定義されていないと、RTA-OSEK Planner はそのタスクまたは ISR の実行時間全体にわたって、リソースが保持されるかまたは割込みがディセーブルになると想定します。

図 17-19 は、RTA-OSEK GUI でリソース使用時間を定義する方法を示しています。最長の時間を 1 つだけ定義してください。

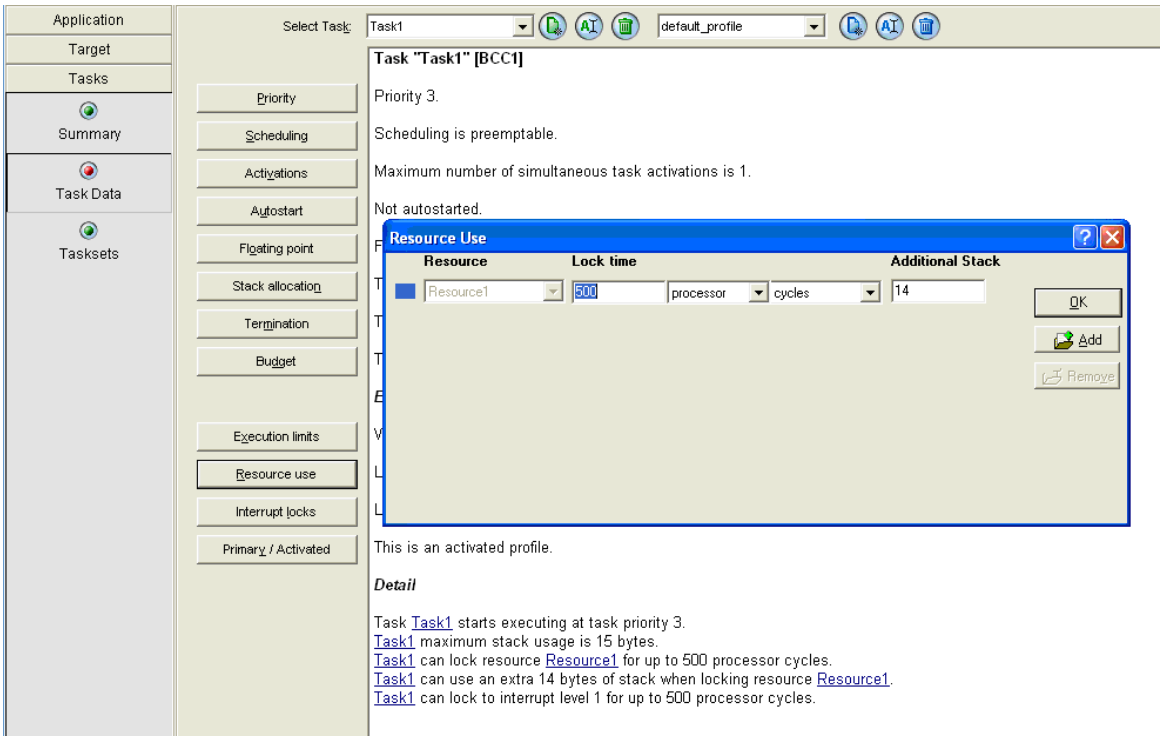


図 17-19 リソース使用時間を定義する

図 17-20 は、RTA-OSEK GUI で割込みロック時間を定義する方法を示しています。この場合も、割込みロック時間については、最長の実行時間を 1 つだけ定義してください。

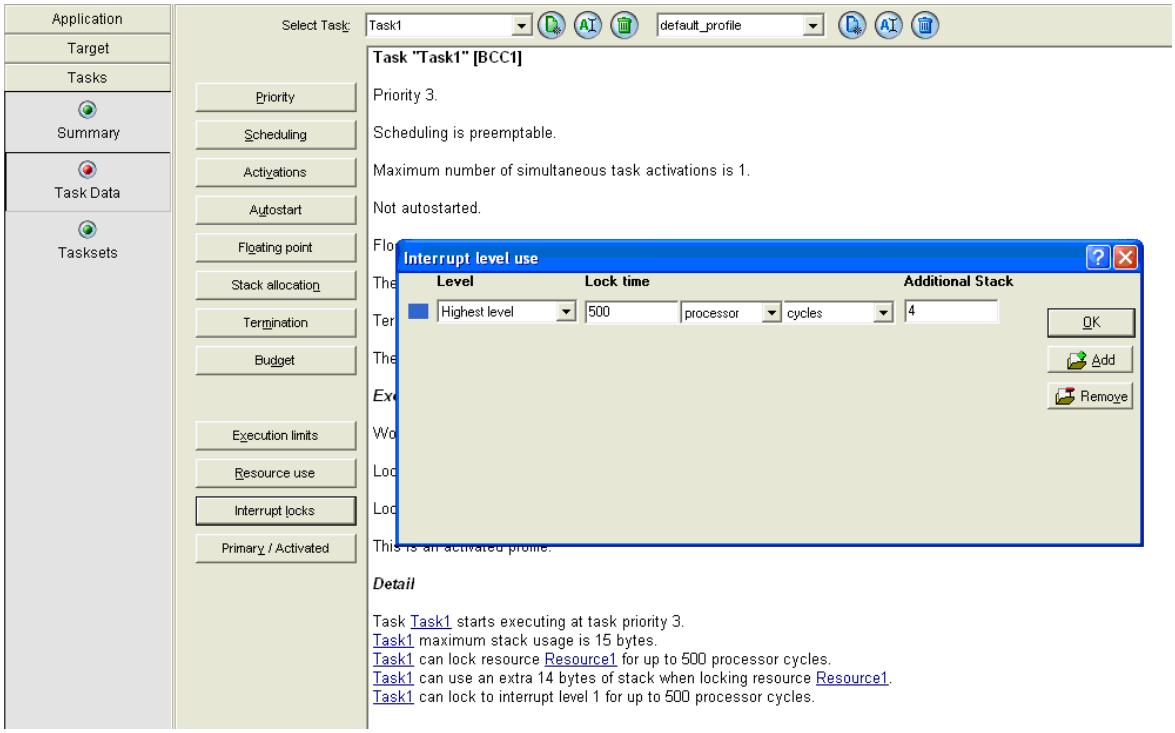


図 17-20 割込みロック時間を定義する

コード例 17-1 は、`Resource1` がタスク内の 2ヶ所においてそれぞれ 100 サイクルと 300 サイクルだけ保持されることを示しています。この場合、最長の時間（300 サイクル）だけを定義すればよいのですが、両方とも定義しておくことができます。

```
TASK(Task1) {
   ...
   GetResource(Resource1);
   /* Held for 100 processor cycles. */
   ReleaseResource(Resource1);
   ...
   GetResource(Resource1);
   /* Held for 300 processor cycles. */
   ReleaseResource(Resource1);
   TerminateTask();

}
```

コード例 17-1 リソースを占有する

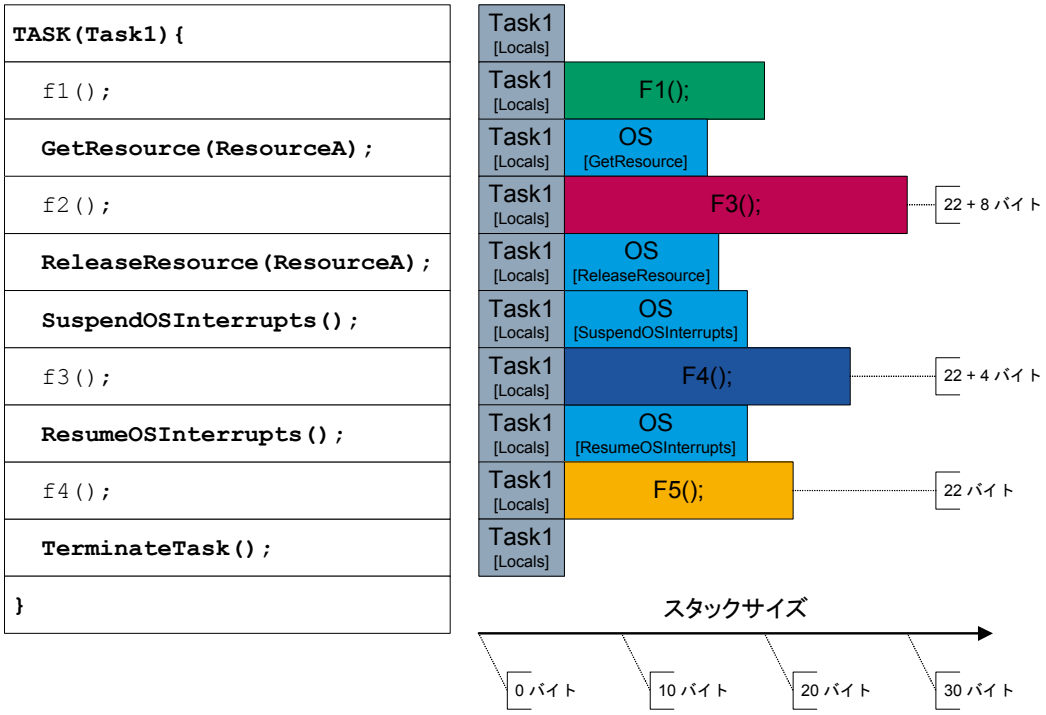
複数のロックををそれぞれ個別に定義しておくと、設定ファイルの内容がわかりやすくなり、保守性が向上します。ただしこの場合、どのロック時間もタスクの実行時間を超えないようにしてください。

時間を設定する際、リソースリクエストがネストしているかどうかを区別する必要はありません。これについては RTA-OSEK Planner が分析時に自動的に考慮します。

保持される各リソースや割込みについてスタック使用量の情報を定義することができるので、各リソースおよび割込みごとにこの数値を入力しておくことをお勧めします。スタック使用量の数値が 1 つしか定義されていないと、RTA-OSEK Planner はスタックオーバーレイについて一切考慮に入れることができません。

スタックオーバーレイについての情報を RTA-OSEK Planner に入力しておけば、リソースが保持されている時や割込みがディセーブルになっている時のスタック使用量を減らすことができます。

図 17-21 は、`Task1` の実行に必要なスタックスペースを示しています。



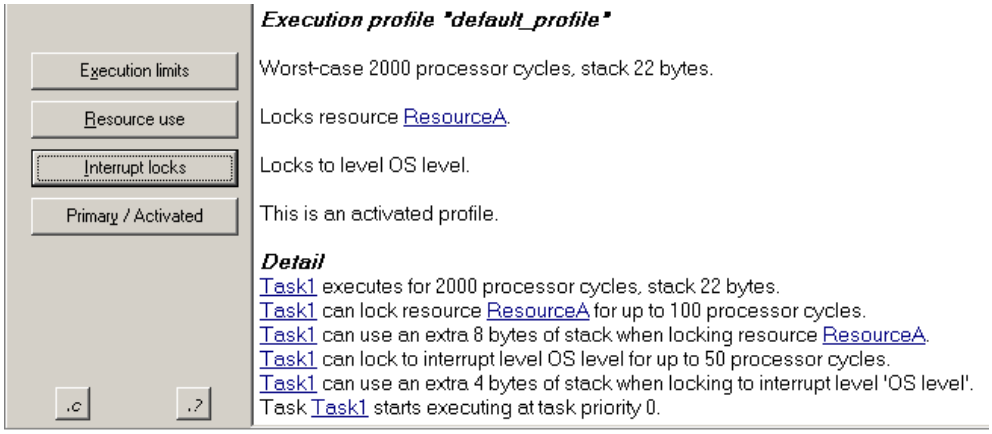


図 17-21 Task1 に必要なスタックスペース

さらに正確な数値を得るには、ユーザーがさまざまな測定を行ってワーストケースのスタック使用量を明らかにする必要があります。タスクのどの時点で最も多くのスタックが消費されるかを明らかにするためには、アプリケーションの関数についての知識が必要となります。

ここでは以下のような測定を行う必要がります。

- タスクまたは ISR のエントリポイントの最初のマシンコード命令から任意の時点までの、ワーストケースのスタック使用量。ただし、リソースを保持した状態でコードが実行される箇所や、割込みがディセーブルになっているかまたはサスペンドされている箇所は除きます。

  この数値が、タスクまたは ISR のスタック使用量となります。
- リソースが保持されている箇所のワーストケースのスタック使用量（保持されるリソースごとの値）。

  この数値が、リソースのスタック使用量となります。
- 割込みがサスペンドしているかディセーブルになっている箇所のワーストケースのスタック使用量（割込みレベルごとの値）。

  この数値から、各割込みロックのスタック使用量が得られます。

17.3.5 複数の実行プロファイルの定義

タスクや ISR は、さまざまな状況で実行される可能性があります。このような場合は、複数の実行プロファイルを宣言することにより分析の「悲観性」を低くすることができます。

複数のプロファイルを定義しておくと、呼び出された時の状況によってタスクまたは ISR の実行時間が非常に短くなったり非常に長くなったりするような場合に対応できます。

コード例 17-2 は、複数の実行プロファイルの定義方法を示しています。

```
if (Condition) {

   /* Short computation. */

} else {

   /* Long computation. */

}
```

コード例 17-2 複数の実行プロファイルを定義する

以下のような場合は、複数の実行プロファイルを使用してください。

- 1 つの ISR が複数の異なる割込みソースを処理し、その実行挙動がソースごとに異なる場合。
- 1 つのタスク内にラウンドロビン式のスケジューリングが実装されている場合。たとえば、初めて起動された時に A という処理を実行し、2 度目に起動された時に B という処理を実行する、というようなタスクの場合。
- タスクまたは ISR の処理内容が、そのときのアプリケーションモードにより異なる場合。

リソースを取得したり割込みをディセーブルにしたりする可能性のあるタスクまたは ISR について複数のプロファイルを定義する場合は、各プロファイルが各リソースを取得するかどうかを考慮する必要があります。

コード例 17-3 の場合、`Task1` は 1 つのプロファイルでは `Resource1` というリソースを取得し、別のプロファイルでは割込みをディセーブルにします。

```
TASK(Task1) {
   if (Condition) {
      ...
      GetResource_Resource1();
      ...
      ReleaseResource_Resource1();
      ...
   } else {
      ...
      DisableAllInterrupts();
      ...
      EnableAllInterrupts();
   }
   TerminateTask();
}
```

コード例 17-3 複数のプロファイルを使用してリソース取得や割込みディセーブルを行う

リソースを使用したり割込みをディセーブルにするプロファイルについては、実行時間とスタック使用量を定義し、それらを行わないプロファイルについては実行時間として 0 を入力してください。

足りない情報があると、アプリケーションの分析結果が正確なものでなくなってしまします。

重要

複数のプロファイルが定義される場合は必ず、それらをモデル内で使用するか、または、（直接的、または自動起動されるアラームにより）自動的に起動されるようにしてください。これを忘れると、そのプロファイルは分析に含まれなくなります。

17.3.6 割込みのルーピング（'looping'）と再トリガ（'retriggering'）

ここまでに説明されているシステムでは、スティミュラスに対するレスポンスのデッドラインはそのスティミュラスのアライバル周期以内でした。このようなシステムにおいては、各タスクと ISR は次に呼び出される前に完了していなければなりません。しかし、場合によっては、スティミュラスが、そのプライマリプロファイルとして関連付けられているタスクや ISR によって処理できる状態になる前に到達してしまうことがありますが、そのような場合、そのレスポンスは次回（または n 回め）のアライバル後に生成することができます。このような状況は、たとえば、ネットワーク経由でバーストメッセージのアライバルを処理しなければならない場合に起こります。このような場合は、スティミュラスに対するレスポンスを実現するためのデッドラインはそのスティミュラスの周期よりも長くなります。

RTA-OSEK Planner ではキューイングされるタスク（BCC2 タスク）の挙動を自動的に扱うことができます。これらのタスクが「再トリガ」を行います。つまり、次のインスタンスが起動する前に最初のインスタンスがターミネートします。

このような場合、ISR については、割込み処理を行えるようになるまで割込みをバッファリングする手段が必要となります。これは、外部の割込み制御ロジックを利用するか、あるいは、ターゲットマイクロプロセッサがこれをサポートしている場合もあります。たとえば、一般的な CAN コントローラには、ネットワーク経由で到達するメッセージのハードウェアバッファリング機能が搭載されています。

バッファリングされた割込みを ISR が処理する方法には、以下の 2 通りがあります。

- ルーピング

  ISR の一番外側のレベルにループを持たせ、そのループにおいて未処理の割込みが残っているかどうかを調べ、残っていればその割込みについての処理を続けて行います。

- 再トリガ

  ISR の最後の部分で未処理のイベントが残っているかどうかを調べ、残っている場合は割込みハンドラをもう一度トリガするための割込みを発生させます。

移植性

再トリガの実装方法は、ターゲットプラットフォームの割込みメカニズムに依存します。通常は、ユーザーが割込みをアサートし直す必要があります。

RTA-OSEK Planner での分析においてバッファリング挙動を考慮に入れたい場合は、以下の事柄を定義する必要があります。

- バッファリングが使用されること
- 再トリガとループのどちらの方法でバッファを処理するか
- バッファのサイズ

図 17-22 は、RTA-OSEK GUI に ISR バッファリング挙動を入力する方法を示しています。

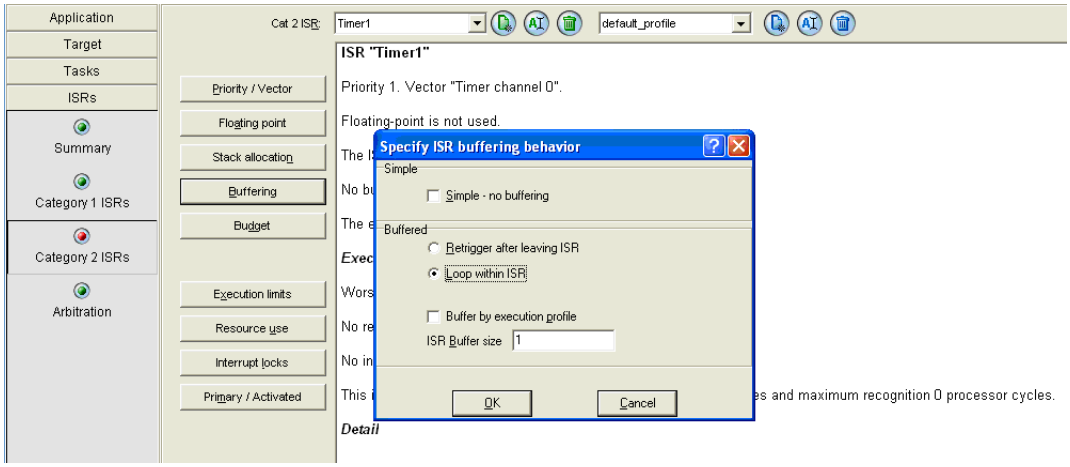


図 17-22 バッファリング挙動を定義する

上記の説明のように、バッファリングされた割込みが ISR により処理される場合、再トリガを行うかルーピングを行うかを選択することができますが、通常は再トリガを行うことをお勧めします。いずれかを選択する際には、以下の要素を考慮してください。

第 1 に、一部のハードウェアは割込みの再トリガ処理をサポートしていません。この場合、ルーピング処理を行う必要があります。

第 2 に、ハンドラを呼び出す割込みがシステム内の別の割込みと同じレベルになっていて、後者の割込みの方がアービトレーションオーダーが上位である場合は、再トリガを行うことをお勧めします。アービトレーションオーダーが上位であるということは、両方が処理待ちになった場合に先に処理されるということです。これにより、上位の割込みが受けるブロッキングが少なくなる可能性があり、これは、ターゲットが割込みレベルを 1 つしかサポートしていない場合には特に重要になります。

第 3 に、1 つの割込みだけを処理する場合は、再トリガが実装された実行オブジェクトの方がルーピングのものよりも実行時間が短くなる可能性があります。一般的には、ルーピングハンドラの方が、1 回の呼び出しで複数の割込みを処理できる、という点で効率的であるといえますが、分析においては、ワーストケースの挙動を想定する必要があります。ワーストケースの挙動が発生するのは、各割込みがそれぞれ独立した ISR コールにより処理される、という実行パターンにおいて割込みが処理される場合です。

コード例 17-4 に、複数のプロファイルを使用する別の例を紹介します。この ISR は `Source1()`、`Source2()`、`Source3()` という関数でそれぞれ検知される 3 つの割込みソースを処理します。

```
ISR(isr1) {

   if (Source1()) {

   /* Handle Source1. */

   } else if (Source2()) {

   /* Handle Source2. */

   } else if (Source3()) {

   /* Handle Source3. */

   }
}
```

コード例 17-4 複数のプロファイルを使用する

コード例 17-4 に示される ISR には、3 つの実行プロファイルが定義されています。それらの特性は、以下のように判定されます。

- `Source1()` が `TRUE` を返す
  このプロファイルの実行時間には、`Source1` のチェック結果が `TRUE` となった場合のワーストケースの実行時間が含まれます。
- `Source1()` が `FALSE` を返し、`Source2()` が `TRUE` を返す
  このプロファイルの実行時間には、`Source1` のチェック結果が `FALSE` になり、かつ `Source2` のチェック結果が `TRUE` となった場合のワーストケースの実行時間が含まれます。
- `Source1()` と `Source2()` が `FALSE` を返し、`Source3()` が `TRUE` を返す
  このプロファイルの実行時間も、上記 2 つのプロファイルの場合と同様の方法で算出されます。

これらのプロファイルは、ISR 全体（最初の命令から最後の return 命令まで）のすべてのパスを表しています。すべてのチェック結果が `FALSE` になった場合のプロファイルは定義されていません。これは、上記の条件のいずれか 1 つが真（`TRUE`）にならないと、割込みが入ることはあり得ないからです。

状況によっては、複数の割込みソースを処理するハンドラを 1 つだけ作成し、その中に含まれる各プロファイルがそれぞれ異なるスティミュラスに対応するようにしなければならない場合もあります。そのような場合は、図 17-23 に示すように、プロファイルが実行プロファイルによってバッファリングされることを定義してください。

図 17-23 実行プロファイルの割込みバッファリング

その場合、コード例 17-4 は、コード例 17-5 のように変更してください。

```
ISR(LoopingHandler) {
   do {
      if (Source1()) {
         /* Handle Source1. */
      } else if (Source2()){
         /* Handle Source2. */
      } else if (Source3()){
         /* Handle Source3. */
      }
   } while (interrupt_pending());
}
```

コード例 17-5 実行プロファイルによるバッファリング

タスクをバッファリングすることもできますが、これには、タスクの起動をキューイングできるようにする必要があります。この場合、RTA-OSEK Planner はすでに、RTA-OSEK コンポーネントの設定によりバッファのサイズがわかっているため、このためのモデリング情報を追加入力する必要はありません。このタスクは、キューが空になるまで繰り返しトリガされます。

17.4 ターゲット固有のタイミング情報

ここまでに説明されている情報はユーザーのアプリケーションに関するものですが、正確な分析を行うには、RTA-OSEK Planner はさらに、ターゲットハードウェアのタイミングと稼働についての情報を知る必要があります。

- システムタイミング
  RTA-OSEK コンポーネントのさまざまな実行時間（タスクの開始と終了など）です。
- 割込み認識時間（'interrupt recognition time'）
  割込みの最初の命令が実行されるまでに要する、ハードウェアに起因する最大遅延です。
- アービトレーションオーダー（'arbitration ordering'）
  同じ優先度の複数の割込みが処理される順序です。

割込み認識時間とアービトレーションオーダーはターゲットにより決まります。システムタイミングはアプリケーションがターゲット固有の機能をどのように使用するかにより決まります。

通常、ユーザーはアプリケーションがターゲットにどのように実装されるかについてある程度知っておく必要があります。しかしこれらの情報は設計の初期段階で得られるとは限りません。そのような場合は、合理的な仮定に基づく情報を入力しておき、実際のデータが入手可能になった時点でそれを修正する必要があります。

17.4.1 システムタイミング

正確なタイミング分析を行うには、オペレーティングシステムのオーバーヘッドを算出するための正確な情報を RTA-OSEK に与える必要があります。システムタイミングデータは、オペレーシングシステムがどれだけのプロセッササイクルを要するかを表わすデータです。

システムタイミングの最適な測定方法は、ターゲットプラットフォームやその実装方法によって異なります。実際にご使用の環境に適した測定方法について詳しい情報が必要な場合は、ETAS のエンジニアリングサービスをご利用ください。

重要

システムタイミング情報は、ハードウェア構成によって異なります。ハードウェアの変更やアプリケーションのメモリ配置の変更（オンチップからオフチップ ROM への移動など）を行った場合は、システムタイミングを測定し直す必要があります。
また、拡張タスクやアラームを追加するなど、アプリケーションの特性を変更した場合にも、システムタイミング値は変わってしまいます。

分析を行う際にはシステムタイミング値が必要です。これらの値が取得できない場合は、推測値を入力する必要があります。これは、たとえば開発工程の早期の段階においてシステムのタイミング挙動を調べるような場合です。システムタイミング値がまったく入力されていないと、RTA-OSEK Planner はそれらの値が 0 あると想定して処理します。

17.4.2 割込み認識時間（'interrupt recognition time'）

割込み認識時間は、割込みがターゲットハードウェアに認識されるまでの最大遅延時間を表します。これは CPU サイクルを単位とする値で表されます。

通常、割込み認識時間の長さは、最短でも、最も長い命令の実行時間以上となります（ただし、割込みによって長い命令が分割される場合を除きます）。『RTA-OSEK バインディングマニュアル』とメーカーのターゲット用のデータブックで、この情報の取得方法を調べてください。

図 17-24 は、割込み認識時間の入力方法を示しています。

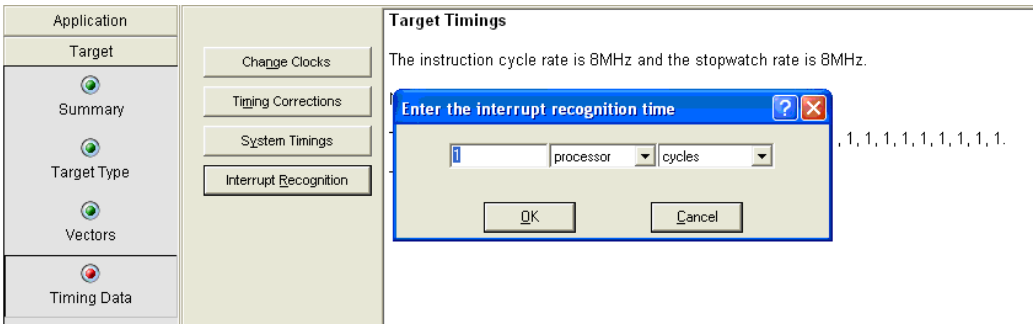


図 17-24 割込み認識時間を定義する

割込み認識時間は、分析時にはブロッキング時間として扱われます。つまり、割込み認識時間の間は、プロセッサは割込みが起こっていないかのようにタスクの命令を実行し続けます。割込み処理のオーバーヘッドを割込み認識時間として分類しないようにしてください。

17.4.3 割込みアービトレーション（'interrupt arbitration'）

複数の ISR が 1 つの割込み優先度レベルを共有している場合は、ユーザーは「割込みアービトレーションオーダー」（'interrupt arbitration order'）を入力する必要があります。アービトレーションオーダーは、複数個の同じ優先度の割込みが同時に処理待ち状態になった場合にそれらの割込みが処理される順序です。通常、この情報はターゲットプロセッサ用のデータブックに記載されています。

アービトレーションオーダーを定義すると、RTA-OSEK Planner は定義された割込みについての割込みブロッキングを適切に判断できます。図 17-25 では、`Bursting`、`Timer1`、`Timer2` という 3 つの割込みが、割込み優先度レベル 1 を共有しています。ここでは、これらすべての割込みが同時に処理待ち状態になった場合、`Bursting`、`Timer1`、`Timer2` の順で処理されるように設定されています。

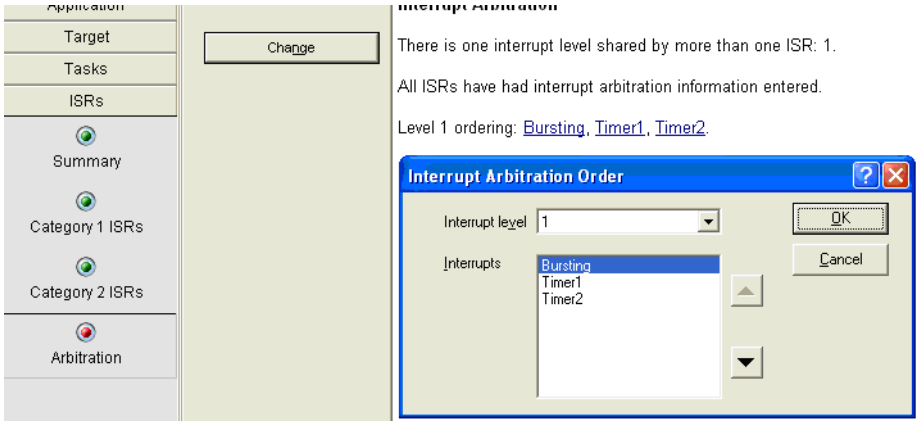


図 17-25 割込みアービトレーションオーダーの設定

17.5 アラームのモデリング

アプリケーションに、1 つのカウンタと一連のアラームによるタスク起動シーケンスを実装する場合、RTA-OSEK Planner は各アラームの停止と再起動が個別のタイミングで行われるものであると仮定します。これにより、そのカウンタに対応するアラームのワーストケースのタイミング挙動が確実に計上されます。

しかしこの方法では、アラームが自動起動されてランタイムに変更されない、という条件がある場合は、非常に「悲観的」な分析結果となってしまいます。このような場合は、カウンタに対応するアラームが同期化されていることを定義することにより、そのような結果を回避することができます。

図 17-26 は、RTA-OSEK GUI でアラーム同期化設定を選択する方法を示しています。この場合、ユーザーはアラーム同期化が確実に維持されるようにする必要があります。

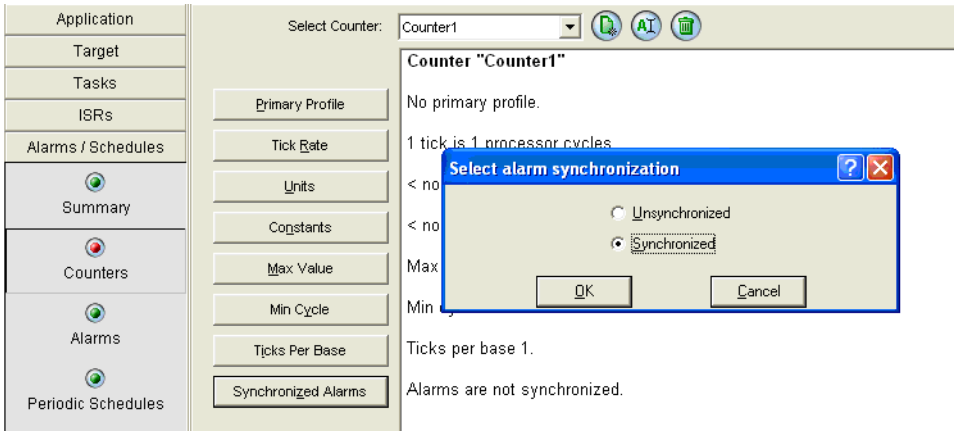


図 17-26 アラーム同期化設定を選択する

17.6 スケジュールテーブルのモデリング

現時点では RTA-OSEK Planner で AUTOSAR スケジュールテーブルを使用するシステムの分析は行えません。

17.7 計画的スケジュールのモデリング

計画的スケジュールを使用して複雑なスティミュラスシーケンスを実装する方法についてはすでに説明されていますが、アプリケーションのタイミングの確度を分析しようとする場合、付加的なタイミング情報を入力する必要があります。

アプリケーションの挙動を変更するために、ランタイムに計画的スケジュールを変更することができます。遅延を変更したり、アライバルポイントにレスポンスを追加したり、Next クローズ（'next clause'）を変更してレスポンスをスケジュールに組み込んだりはずしたりすることができます。

ランタイムにこの柔軟性を活用するには、スケジュールのワーストケースの挙動についての付加的な情報を入力しておく必要があります。ワーストケース挙動は以下の情報により表現されます。

- アライバルポイント間の最短遅延
- 各アライバルポイントで間接的にトリガされるスティミュラスの最大数

計画的スケジュールの構造に対する変更は、**分析オーバライド**（'analysis override'）として定義されます。あるアライバルポイントからトリガされるスティミュラスについての付加的情報（潜在的変化など）は、間接的に起動されるスティミュラスとして定義されます。

> **重要**
> ここでは、スケジュールについてのワーストケースの情報を入力する必要があります。そうしないと、ユーザーがランタイムに変更を行うとアプリケーションがアンスケジューラブルになってしまう、といった場合にも、RTA-OSEK Planner はそのアプリケーションがスケジューラブルであると示してしまいます。

17.7.1 分析オーバライドの定義

分析オーバライドを使用して、スケジュールの構造がランタイムにどのように変更される可能性があるかを知らせることができます。タイミング分析については、分析オーバライドと実装詳細情報の両方が存在する場合、`delay` と `next` という分析属性はアプリケーションの属性をオーバライドします。

ランタイムにおいて遅延が実装遅延より短くなるように変更される場合は、短い方の遅延を分析オーバライドとして定義します。図 17-27 を参照してください。

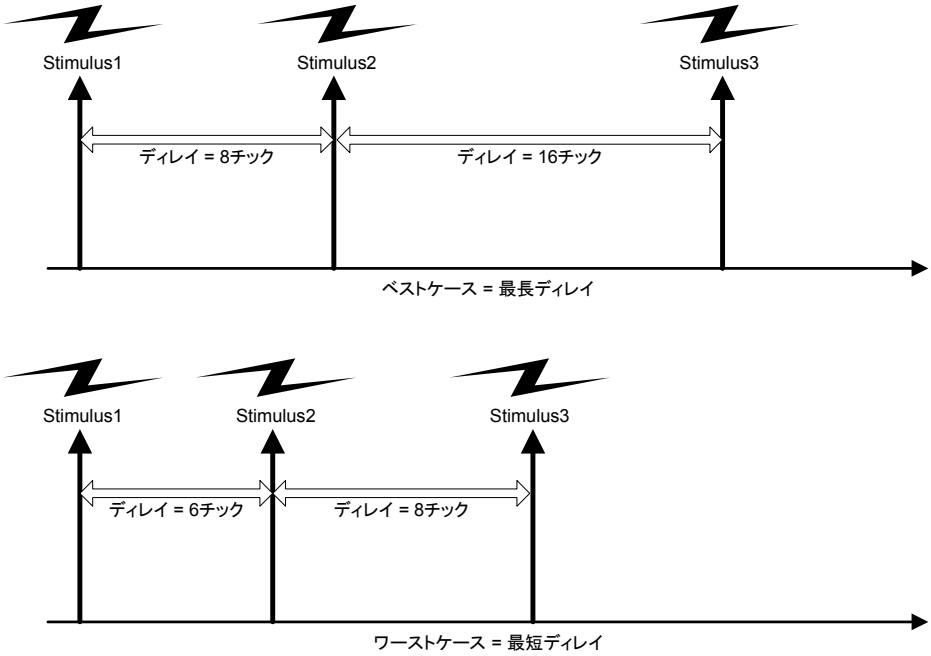


図 17-27 ベストケースとワーストケース

17.7.2 間接的に起動されるスティミュラス

計画的スケジュールについての説明においては、到達時に自動的に起動されるスティミュラスだけを定義しましたが、分析の際は、RTA-OSEK Planner は間接的に起動されるスティミュラスがあるかどうかを知っておく必要があります。

「間接的な起動」とは、たとえば、自動的に起動されるスティミュラスがレスポンスをトリガし、そのレスポンスが別のタスクを起動するような場合を指します。あるタスクが別のタスクを起動する場合や、ランタイムにアライバルポイントのタスクセットに追加する場合は、この「間接的な起動」を定義する必要があります。

各アライバルポイントに、間接的に起動されるスティミュラスを複数個定義することができます。同じスティミュラスを自動的に起動することも、間接的に起動することもできます。これは、タスクが自分自身をチェーニングするための機能です。

分析においては、直接起動されるスティミュラスと間接的に起動されるスティミュラスには何の違いもありません。分析されるシステム内には上向きの起動は許されていないため、これら 2 つの状況について観察される挙動はまったく同じです。

タイミング分析の場合、RTA-OSEK Planner は自動的に起動されるスティミュラスと間接的に起動されるスティミュラスはすべて一度にトリガされると想定します。たとえば、あるアライバルポイントが `Stimulus1` を自動的に起動し、`Stimulus2` を間接的に起動する場合、RTA-OSEK Planner は、レスポンスを生成するすべてのタスクがアライバルポイントで同時に生成されると想定します。これが、そのアライバルポイントのワーストケースとなります。

17.8 シングルショットスケジュールのモデリング

RTA-OSEK Planner は、シングルショットスケジュールはアプリケーションのランタイム全体の中で 1 回しか実行されないと想定しています。

しかし、シングルショットスケジュールをシステムにより反復的に処理することができます。このような場合は、スケジュールの実装がシングルショットタイプであること、およびそのスケジュールが分析の目的で繰り返し実行されることを示しておく必要があります。

分析のための反復は、計画的スケジュールにおける最後のアライバルポイント用の分析オーバライドとして定義します。次のオーバライドがスケジュール上の最初のアライバルポイントを示し、次のアライバルポイントまでの遅延は連続するスケジュール起動間の最小時間となるように設定してください。

この設定は、RTA-OSEK GUI で図 17-28 のように行います。

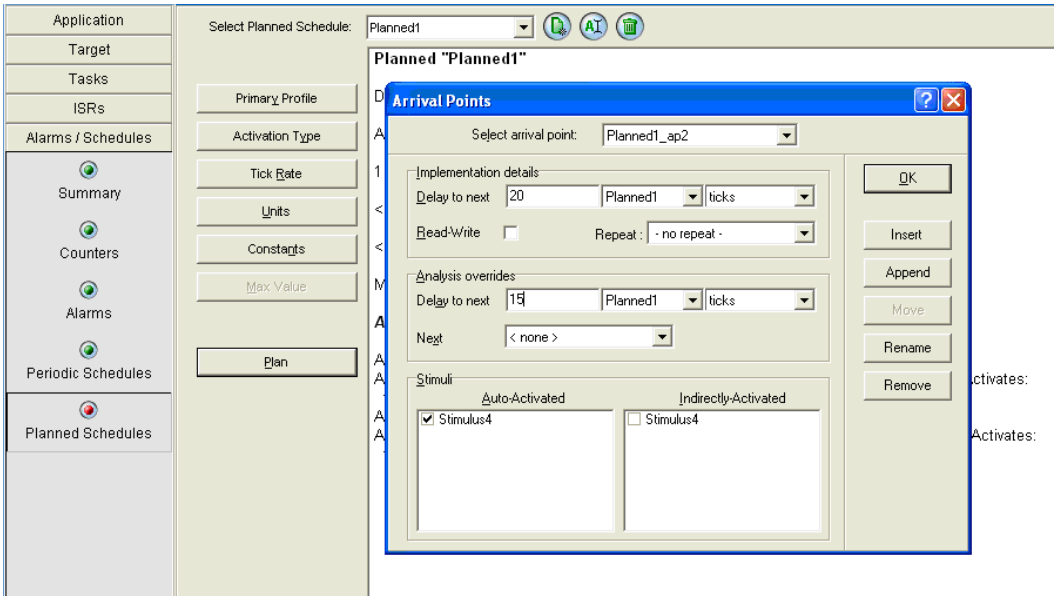


図 17-28 シングルショットスケジュールをモデリングする

重要

シングルショットスケジュールの反復時の遅延として指定する値は、アプリケーションの詳細な情報に基づいて決定されたものである必要があります。最小遅延より大きい遅延を選択すると、「楽観的」な分析になってしまいます。そのような分析ではシステムがスケジューラブルであると誤って知らせてしまう可能性があります。逆に、値が小さすぎる場合は不必要なまでに「悲観的」になってしまう可能性があります。

17.9 拡張タスクを含むモデリング

ユーザーアプリケーション内で拡張タスクを使用する場合、すべての拡張タスクよりも優先度の高い基本タスクのセットしか分析できません。拡張タスクはイベント待ちを行うため、RTA-OSEK がこれを分析することは不可能です。

しかしそれでもリソースと割込みによるロック時間を指定する必要はあります。これは、優先度の高い基本タスクによるブロック量に影響するためです。

重要

拡張タスクの挙動を用いる必要があり、かつアプリケーション全体の分析も行う必要がある場合は、「タスク」の章に概説されているスキームを用いて拡張タスクをシミュレートする方法を検討してみてください。

17.10 まとめ

- アプリケーションを分析する必要がある場合は、スティミュラス - レスポンスモデルについての実行パフォーマンス上の制約、それぞれのタスクまたは ISR のワーストケースの実行時間、およびターゲットのタイミングを設定する必要があります。
- パフォーマンス上の制約は、スティミュラス - レスポンスモデルの一環として設定します。
- バーストスティミュラスは、プライマリプロファイルがスティミュラスを直接捕捉するという単純なケースのモデリングに使用されます。
- 計画的スティミュラスと周期的スティミュラスは、プライマリプロファイルがカウンタまたはスケジュールを駆動してスティミュラスを生成するような、複雑なケースのモデリングに使用されます。

- アプリケーション内のそれぞれのタスクと ISR にはプロファイルが 1 つ以上必要で、そのプロファイルには実行情報、およびそのプロファイルがスティミュラスの捕捉または生成に使用できるかどうかを設定する必要があります。
- できるだけアラームが同期化されるように設定しておくことにより、システムの「悲観性」を低くすることができます。
- また、それぞれのタスクと ISR について複数のプロファイルを使用することによっても、「悲観性」を低くすることができます。
- 割込みのバッファリングをモデリングすることができます。
- 組み込むシステムに関してタイミングが適切でなければならない場合は、各プライマリプロファイルの入力ジッタと各レスポンスの出力ジッタを定義する必要があります。
- 計画的スケジュールには、他のスティミュラスをトリガするスティミュラスを把握し、さらにランタイムにおいて行われるスケジュール挙動の変更について把握するための分析情報を含める必要があります。

18 タイミングモデルの分析

RTA-OSEK GUI では、RTA-OSEK Planner を使用してアプリケーションを分析することができます。ここでは以下の 5 つの分析オプションを利用できます。

- スタック深度（'Stack Depth'）分析
- スケジューラビリティ ('Schedulability'）分析
- センシティビティ（'Sensitivity'）分析
- ベストタスクプライオリティ（'Best Task Priorities'）分析
- CPU クロックレート（'CPU Clock Rate'）分析

スタックの深さ、スケジューラビリティ、およびセンシティビティ分析を行うと、アプリケーションのメモリ使用量とタイミング挙動がわかります。ベストタスクプライオリティとクロックレート分析では、アプリケーションのメモリスペースや実行時間を最適化するための方法が提示されます。

分析を行うには、ユーザーは実行時間とスタックスペースの情報を定義する必要があります。これは、アプリケーション内の各タスクと ISR 用の実行プロファイル内に定義します。この情報の一部が不足していると、どの部分が不足しているかが分析結果として通知されます。

また分析の際は、アプリケーションが RTA-OSEK Planner モデルに正確に反映されるようにすることが重要です。スティミュラス - レスポンスモデルにアタッチされていないタスクや ISR を作成した場合、それらのものは分析には含まれません。

18.1 スタック深度分析

アプリケーション内の各タスクと ISR についてワーストケースのスタック使用量の数値を定義すると、RTA-OSEK Planner はアプリケーションのワーストケースのスタック使用量を算出することができます。図 18-1 は、タスクについてのこれらの値を設定する方法を示しています。

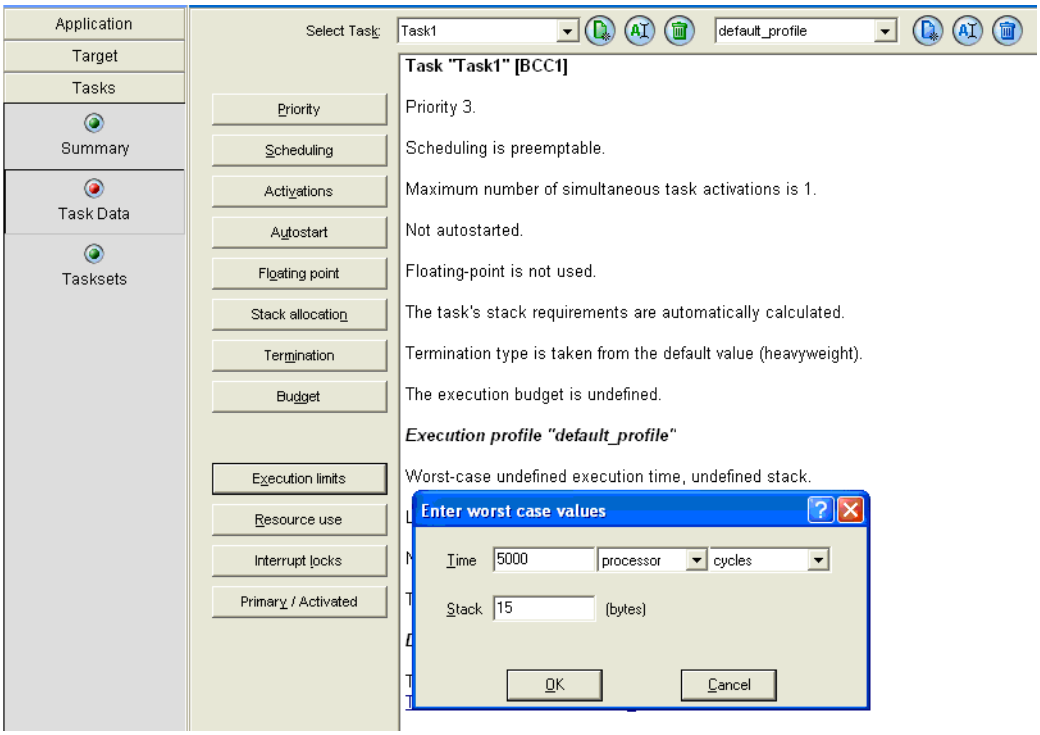


図 18-1 ワーストケースのスタック使用量

スタック分析を実行すると、スタック分析レポートがワークスペースに表示されます。そこでは 2 つのタブを使用して 2 通りの形式で分析結果を見ることができます。テキスト形式の場合、図 18-2 のように分析結果が表示されます。

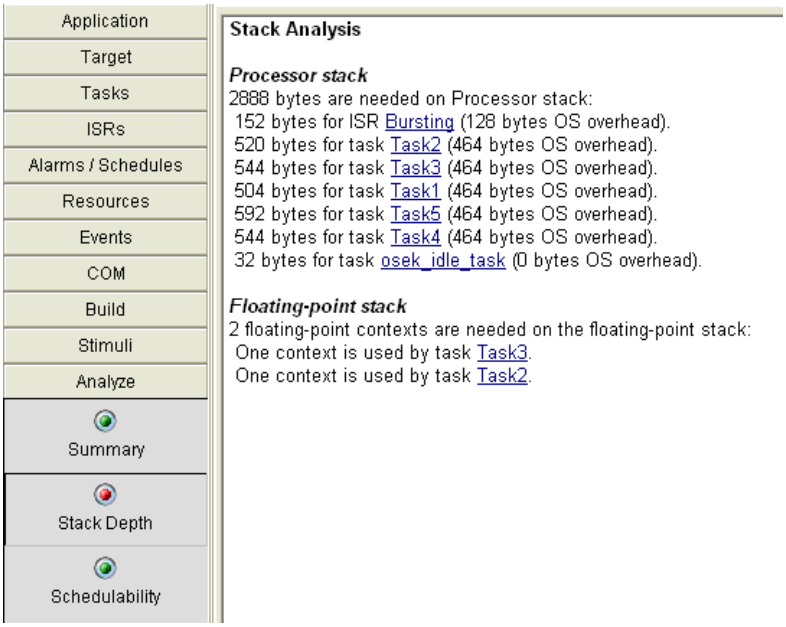


図 18-2 スタック分析結果（テキスト表示）

同じ結果を **Graphic** タブでも見ることができます。ここでは、スタックの最大サイズがワークスペースに表示されます。図 18-3 にその例を紹介します。

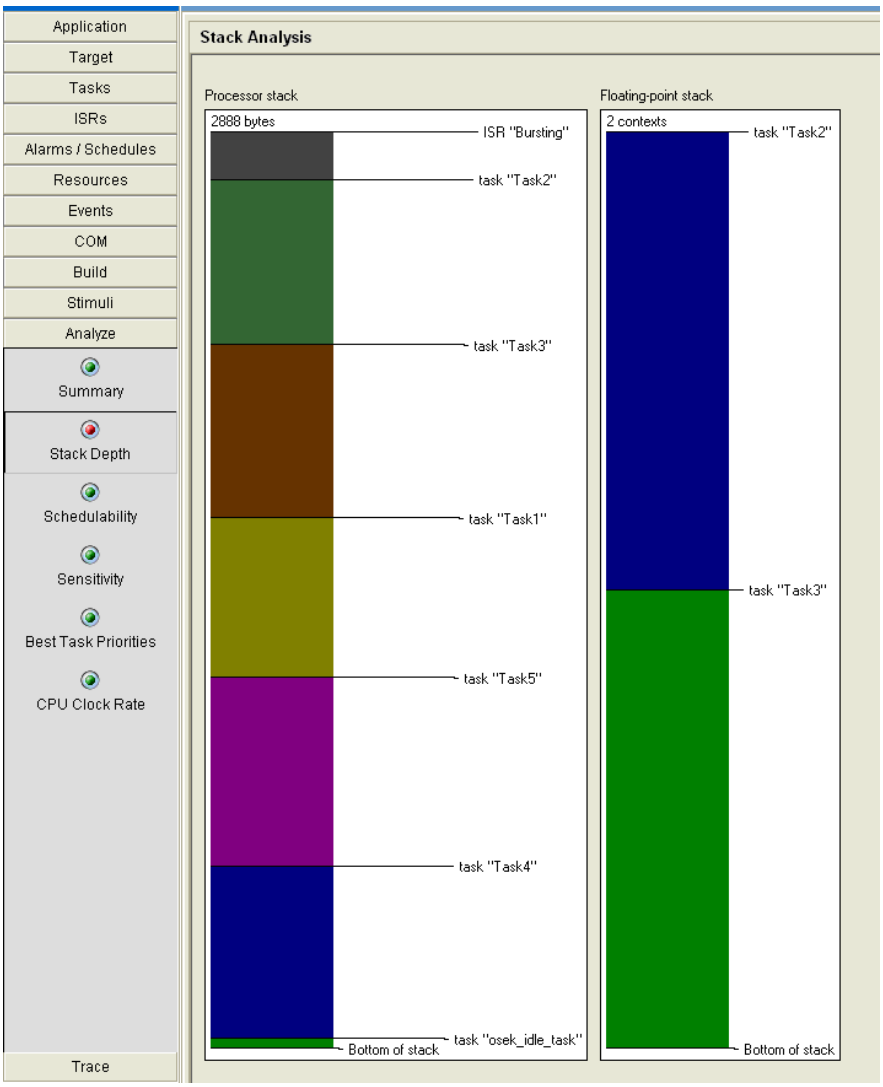


図 18-3 スタック分析結果（グラフィック表示）

分析時には、以下の情報を使用してワーストケースのスタック使用量が計算されます。

- 各タスクと ISR プロファイルのワーストケースのスタック使用量
- 各タスクと ISR に関する RTA-OSEK コンポーネントのオーバーヘッド
- 内部リソースに基づくノンプリエンプション情報
- リソース保持に基づくノンプリエンプション情報
- 割込みのディセーブル化に基づくノンプリエンプション情報

フック関数、コールバック関数、または `GetStopwatch()` で使用されるスタックは、アプリケーションのスタック所要量の計算には含まれません。

通常、これらの関数を使用してもアプリケーションのスタック所要量の合計は変わりませんが、フックやコールバックを使用するアプリケーションのワーストケースのスタック使用量や `GetStopwatch()` のスタック使用量を正確に算出する必要がある場合は、ETAS のサポート窓口までお問い合わせください。

通常、`GetStopwatch()` はタイマレジスタの内容を返すだけで、スタックは使用しません。

18.1.1 浮動小数点コンテキストのセーブ

タスクまたは ISR が浮動小数点を使用する場合、RTA-OSEK コンポーネントは必要に応じて浮動小数点コンテキストをセーブします。RTA-OSEK コンポーネントはアプリケーションの構成に従って、ランタイムにセーブする必要がある浮動小数点コンテキストの最大数を明らかにします。

たとえば、2 つのタスクが浮動小数点を使用し、1 つの内部リソースを共有する場合、それらのタスクは互いにプリエンプトしあうことはないので、浮動小数点コンテキストをセーブする必要はありません。RTA-OSEK GUI は、アプリケーションに必要な浮動小数点コンテキストの最大数を提示します。

18.1.2 スタック使用量の最小化

アプリケーションに必要なスタックスペースが、ターゲットハードウェア上で使用可能なスペースよりも大きい場合があります。このような場合は、アプリケーションのスタック使用量を最小限に抑えるための方法がいくつかあります。

- 内部リソースをタスク間で共有する。

  この場合、タスク同士は互いにプリエンプトせず、複数のタスクがスタックスペースを同時に使用することはありません。この方法で、内部リソースを共有するすべてのタスクのスタックを効果的に重複させることができます。これは、ベストタスクプライオリティ分析を使用して自動的に行うことができます（18.4 項を参照してください）。

- 多くのスタックスペースを使用する関数をタスクが呼び出す場合、その関数呼び出しの近くでリソースを取得するようにして、優先度が高いタスクとリソースを共有できるようにする。

  この際、優先度の高い方のタスクはそのリソースを使用する必要はありません。この方法により、関数を呼び出すタスクが多くのスタックスペースを使用している時に、そのタスクを優先度の高いタスクがプリエンプトすることがなくなり、全体的なスタック使用量が少なくてすむようになります。

- 割込みをディセーブルにするか、上の方法で結合リソースを使用する。

  この方法により、関数が呼び出されている間に割込みが発生するのを防ぐことができます。ただし、この場合は割込みのレスポンスタイムが悪くなってしまいます。

18.2 スケジューラビリティ分析

スケジューラビリティ分析は、各レスポンスをデッドラインまでに生成できるかどうかを明らかにするために使用されます。デッドラインは明示的なものでも暗黙的なものでもかまいません。

明示的なデッドラインの例としては、「タスクが起動されてから 5ms 以内にレスポンスを生成しなければならない」ということをユーザーが定義した場合などがあります。暗黙的なデッドラインの例としては、「タスクが、次回レディ状態になる前にターミネートしなければならない」という条件などがあります。たとえば、キューイングされない基本タスクが 20ms ごとに起動される場合、このタスクは次に起動される前に完了しなければならないので、20ms という暗黙的なデッドラインが設定されることになります。

スケジューラビリティ分析は、アプリケーション内の各実行プロファイルのワーストケースレスポンスタイムを計算してから、これらのレスポンスタイムが、それに関連付けられているデッドラインよりも短いかどうかを明らかにします。RTA-OSEK GUI にはスケジューラビリティ分析の結果が表示されます。図 18-4 にその一例を紹介します。

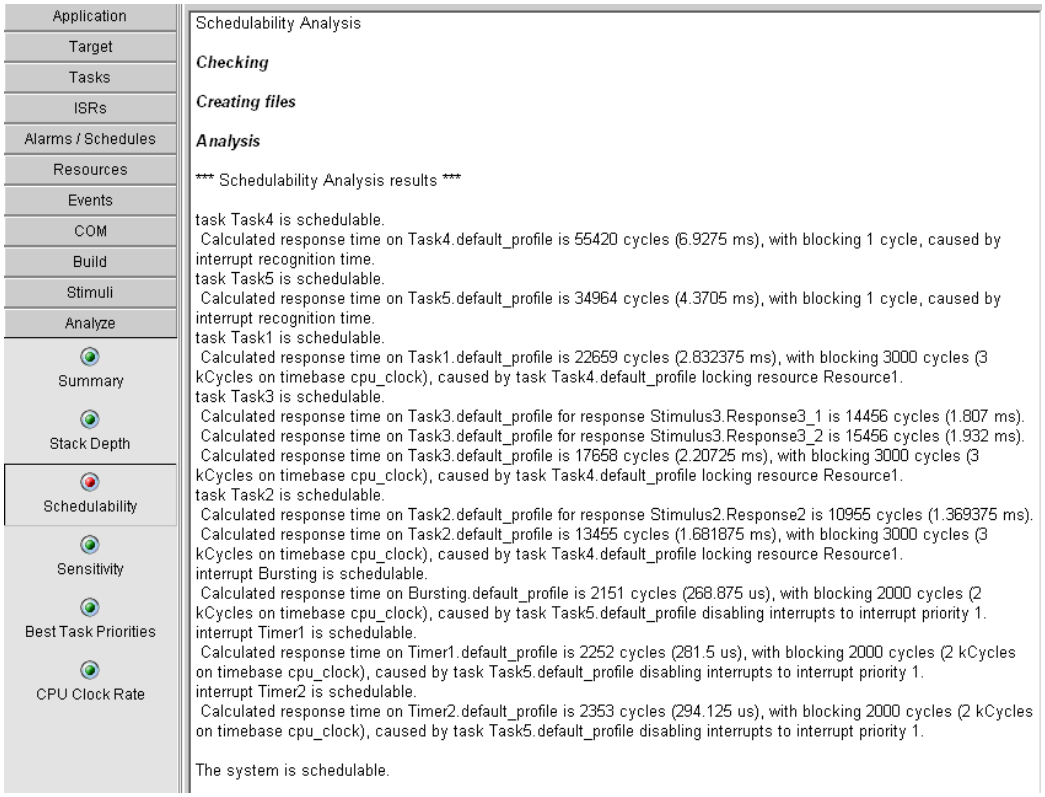


図 18-4 スケジューラビリティ分析の結果（テキスト表示）

図 18-5 のように、分析の結果をグラフ表示することもできます。

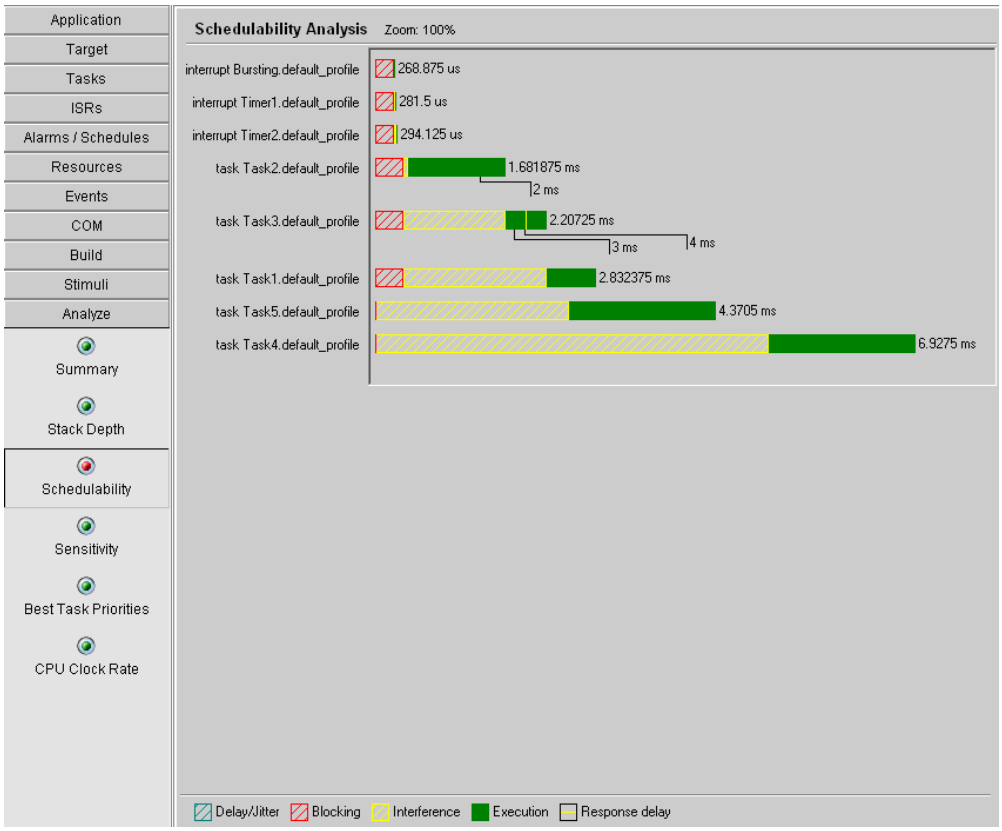


図 18-5 スケジューラビリティ分析の結果（グラフィック表示）

分析レポートのグラフの各バーが、実行プロファイルのレスポンスタイムを示しています。1 つのバーは、最大 5 つの部分に分かれます。

- 遅延／ジッタ（'delay/jitter'）
  スティミュラスに応答するプライマリプロファイルがスティミュラスを認識するためにかかる最大時間です。これは各プライマリプロファイルの **Primary or Activated Profile** ダイアログボックスで定義されます。
- ブロッキング時間（'blocking time'）
  実行プロファイルが自分より優先度の低い、共有リソースを保持したり割込みをディセーブルにしているプロファイルにより実行を妨げられる時間です。
- 妨害（'interference'）
  実行プロファイルが自分より優先度の高いタスクまたは ISR により実行を妨げられる時間です。これは、プロファイルが実行中にプリエンプトされる合計時間です。
- 実行時間（'execution time'）
  実行プロファイルについてユーザーが定義したワーストケースの実行時間です。
- レスポンス遅延（'response delay'）
  レスポンスがソフトウェアにより生成されてから外部環境で実際に認識されるまでの時間です。通常これは、レスポンスによって外部ハードウェアが駆動される場合にのみ定義されます。

グラフ表示のそれぞれのタスクまたは ISR の実行時間を表わすバーの各部分にマウスポインタを移動すると「ツールチップ」が表示され（図 18-6 を参照してください）、ここに各部分の実際の時間が表示されます。

図 18-6 実際の時間を表示するツールチップ

レスポンスの明示的なデッドラインが定義されている場合、それらは定義されたデッドラインを示す小さいタグとしてバー上に表示されます。暗黙的なデッドラインは表示されません。

この情報に加えて、キューイングされている起動の数、バッファリングされている割込み、およびブロッキング時間に関与しているシステムの部分についてのテキストが表示されます。

分析可能なシステムについて、スケジューラビリティ分析レポートにはシステムが「**スケジューラブル**」と「**アンスケジューラブル**」のどちらであるかが出力されます。システムがスケジューラブルである場合、そのシステム内のすべてのタスクまたは ISR がすべてのデッドラインに対応できることを意味します。

システムがスケジューラブルではないとレポートされた場合、以下のいずれかが原因です。

- システム内の一部のレスポンスをデッドラインより前に生成することができない。この場合、システムは「**アンスケジューラブル**」です。
- システムがスケジューラブルかどうかを判断できない。この場合、システムには「**不確定的なスケジューラビリティ**」があります。

アプリケーションのスケジューラビリティが不確定的な場合やアプリケーションがアンスケジューラブルである場合に行うべき対策は、後述します。

また、センシティビティ分析を使用して、非常に有用な情報を取得することができます。これについては 18.3 項で説明します。

重要
アプリケーションの設定を変更する場合は、必ず最初にその変更内容を詳細に分析し、妥当性を検証してください。RTA-OSEK Planner はユーザーが入力する情報を使用してタイミングの的確性を評価します。

18.2.1 アンスケジューラブルシステム

システムは、様々な理由でアンスケジューラブルになる可能性があります。

- タスクまたは ISR が次の起動より前に完了できない。

- タスクまたは ISR が、定義されているデッドラインより前にレスポンスを生成しない。
- ループまたは再トリガの挙動を伴う ISR がバッファ限界を超えてしまうか、起動がキューイングされる基本タスクのキューイング限界を超えてしまう。
- システムの CPU 使用率が 100% を超えてしまう。

上記の各状況について、以下に詳しく説明します。また、システムをスケジューラブルにする修正方法についても説明します。

タスクまたは ISR が次の起動より前に完了できない

タスクまたは ISR が次の起動より前に完了できない場合、RTA-OSEK Planner は図 18-7 に示すようなメッセージを出力します。この例では、`Task2` はスケジューラブルではありません。

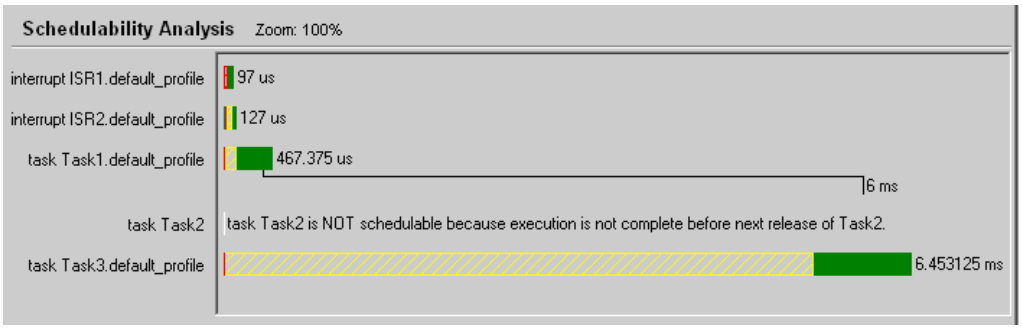


図 18-7 タスクの実行が次の起動より前に完了しない

このタイプのシステムをスケジューラブルにするために実行できるアプローチがいくつかあります。

- タスク、あるいはそれより優先度の高い他のタスクまたは割込みの実行時間を少なくします。実行時間を少なくすることにより、優先度の低いタスクと割込みが受ける妨害を少なくすることができます。
- タスク、あるいはそれより優先度の高いタスクまたは割込みが周期的タスクである場合は、それらの周期を長くすることができます。調整されるタスクに複数のオフセットが定義されている場合、それらを変更することができます。
- タスクまたはバッファ割込みについて起動のキューイングを導入することにより、タスクまたは ISR の実行中に行われる起動が失われないようにします。
- システム内の他のタスクが特定のタスクをアンスケジューラブルにしている可能性があります。ベストタスクプライオリティ分析（詳細については 18.4 項を参照してください）を行って、優先度の順序を変えればシステムをスケジューラブルにできるかどうかを調べることができます。
- アンスケジューラブルなタスクまたは ISR が自分より優先度の低いタスクまたは ISR とリソースを共有している場合、これらのタスクや ISR によりリソースが保持されている時間を短くすることができます。そうすることにより、ブロッキング時間が少なくなり、タスクがスケジューラブルになる可能性があります。

他の理由でシステムがスケジューラブルでないことがわかった場合にも、上記の対策は有効です。

タスクまたは ISR がデッドラインに対応できていない

タスクまたは ISR がデッドラインに対応できていないと、アンスケジューラブルなシステムになってしまいます。検知については以下の 2 通りのシナリオがあります。

- キューイングされないタスク起動とバッファリングされない ISR

  ある特定のプロファイルについて、デッドラインを超えてしまったためそれがスケジューラブルでないと判断されました。この際、同じタスクまたは ISR の他のプロファイルがスケジューラブル（またはアンスケジューラブル）であることもわかります。
- キューイングされるタスク起動（BCC2 タスク）とバッファリングされる ISR

  タスクまたは ISR が、そのいずれかのプロファイルのデッドラインを超えてしまったため、スケジューラブルでないと判定されました。この場合、そのタスクまたは ISR の他のプロファイルはどれもスケジューラブルではないことがわかります。

タスクまたは ISR がそのデッドラインに対応できていないためにスケジューラブルでない場合、以下の対策を試してみてください。

- デッドラインの値を大きくする。

- レスポンス生成コードをプログラム内の早い位置に移動する。
  これにより、タスクまたは ISR がレスポンスを生成するまでに必要な時間が短くなります。
- 上述のアンスケジューラブルシステム向けの提案を採用する。

タスク起動のキューイングと割込みのバッファリングが限界を超えてしまう

タスク起動のキューイング用のキューが、タスクが稼働している間に起こりうる最大数の起動を保持できるだけの長さがないことが原因で、システムがスケジューラブルにならない場合があります。同様に、割込みのバッファリングが行われる時、バッファリングする必要のある割込み数がバッファサイズを超過してしまう場合もあります。

図 18-8 は、バッファリングされる割込み用の起動の数が少なすぎる場合の RTA-OSEK Planner の出力を示しています。

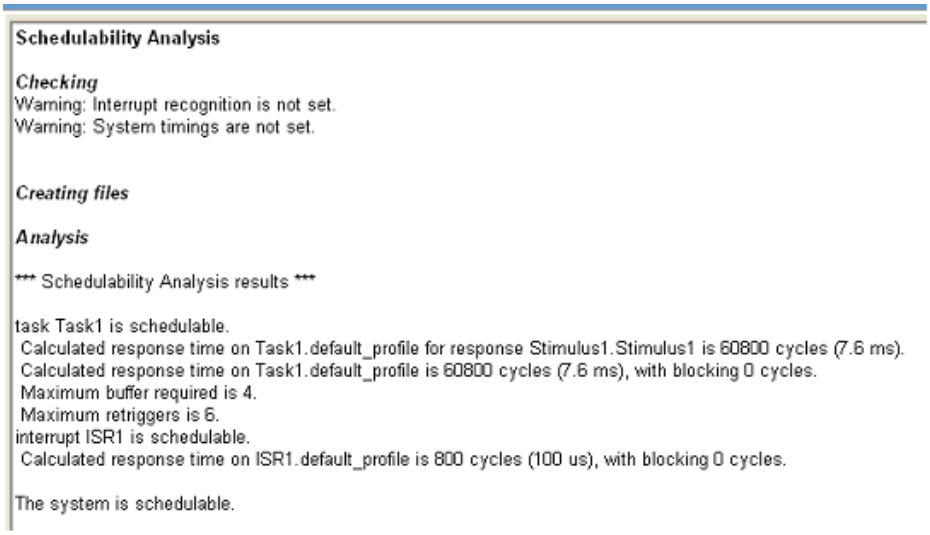


図 18-8 バッファリングされる起動を示すスケジューラビリティ分析の結果

この問題の原因としては、以下の 2 つの事柄が考えられます。

- タスクまたは割込みが、処理できる回数よりも頻繁に起動されている。
- バッファサイズが小さすぎる。

これらの理由でアンスケジューラブルになっているシステムは、以下のような方法でスケジューラブルにすることができる可能性があります。

- 要求されるレートでタスクが入力を確実に処理できるように、優先度を変更する。
  これを行う場合は、ベストタスクプライオリティ分析を行ってみてください（18.4 項を参照してください）。
- タスクまたは ISR の周期を短くする。
- バッファサイズを大きくする。
- タスクの実行時間を短くする。

キューのサイズまたはバッファのサイズを大きめにして分析を行うと、RTA-OSEK Planner はシステムをスケジューラブルにするために必要なバッファサイズをレポートします。

プロセッサ使用率が 100% を超えている

RTA-OSEK Planner は、プロセッサ使用率が 100% を超えていることをレポートする場合があります。これは、現在のアプリケーションを実行するためには、ターゲットハードウェア上で使用可能なプロセッサの実行時間よりも多い時間が必要であることを示しています。図 18-9 に、そのレポートの例を示します。

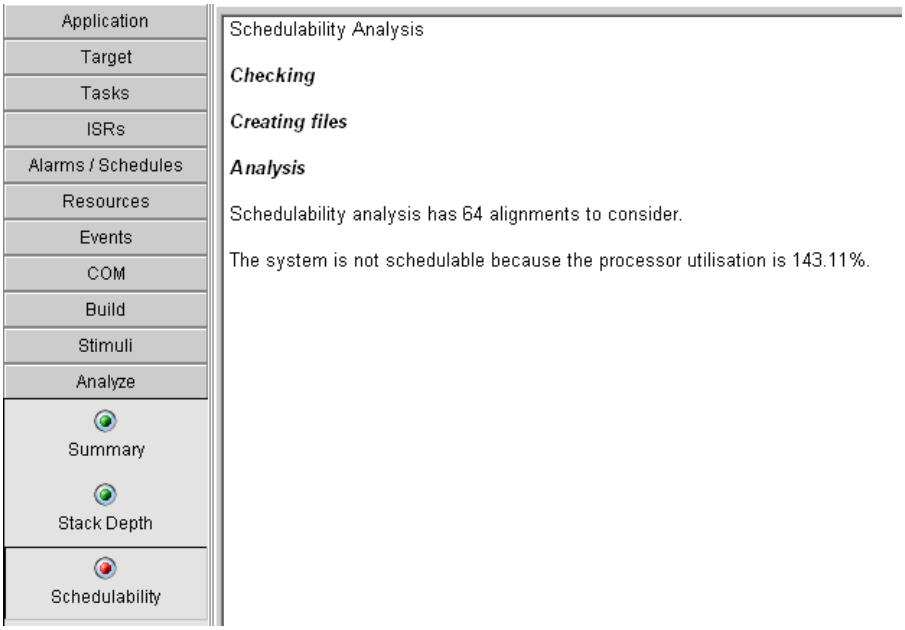


図 18-9 プロセッサ使用率が 100% を超えている

この問題を修正する方法はいくつかあります。

- CPU の処理速度を上げる。
  ハードウェア設計において、処理速度の速いパーツを使用する、というような方法が考えられます。
- タスクや ISR の実行時間を短くする。
- システムスティミュラスの周期を長くする。

18.2.2 不確定的なスケジューラビリティ（'indeterminate schedulability'）

「不確定的なスケジューラビリティ」は、以下のような場合に発生します。

- ビジー期間が長すぎて分析できない。
- 優先度の低いタスクによる不確定的なブロッキングがある。
- 手軽な分析ではスケジューラビリティを判断できない。

不確定的なスケジューラビリティを解決するために、18.2.1 項で説明した方法を使用することができます。

ビジー期間が長すぎて分析できない

ビジー期間とは、タスクがレディ状態か実行状態にある時間と最大認識時間の合計です。

RTA-OSEK Planner は、2^{32} 命令サイクルを超えるビジー期間については、長すぎて分析できないと考えます。有効な実環境システムにおいては、ビジー期間がこの値に到達することはほとんどありません。

優先度の低いタスクまたは ISR による不確定的なブロッキング

優先度の低いタスクによる不確定的なブロッキングがある場合、RTA-OSEK Planner は、優先度の低いタスクとリソースを共有するどのタスクのレスポンスタイムも計算することができません。このような場合、RTA-OSEK Planner からの出力は図 18-10 のようになります。

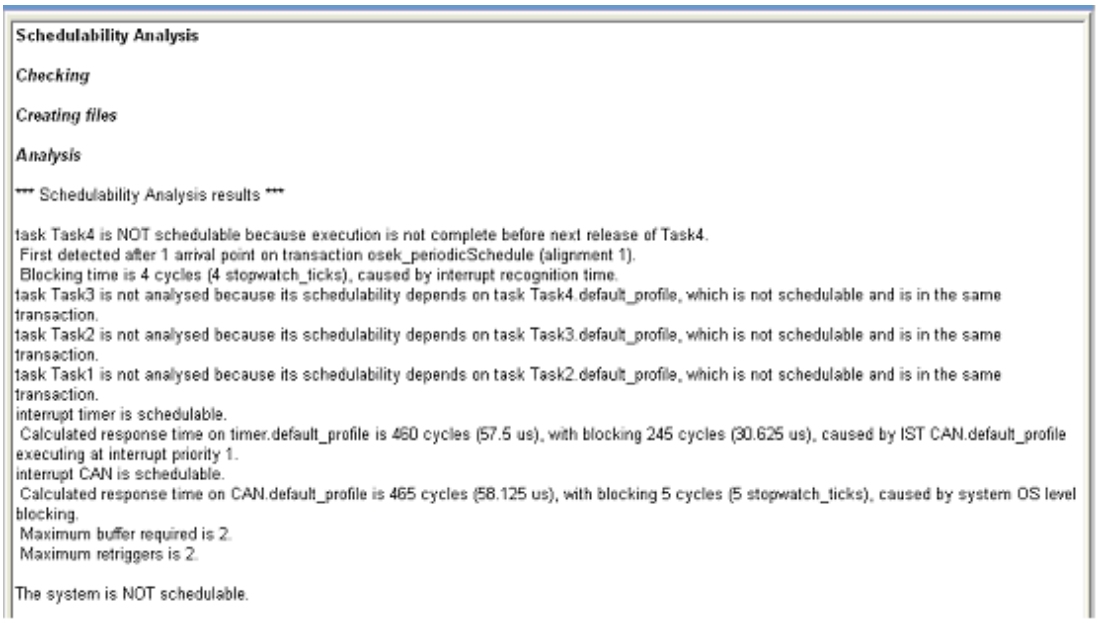


図 18-10 不確定的なブロッキングを示すスケジューラビリティ分析結果

この状況を解決するには、漸進的なアプローチが必要になります。システムがスケジューラブルになるまで、まず優先度の低いタスクをスケジューラブルにしてからスケジューラビリティ分析を繰り返し適用していきます。

手軽な分析ではスケジューラビリティを判断できない

RTA-OSEK Planner は、RTA-OSEK GUI により駆動される際に、デフォルトでは正確な分析（分析の深さ：9）を行うように設定されています。粗い分析（分析の深さ：1）を行った場合は、スケジューラビリティを判断することができない可能性があります。

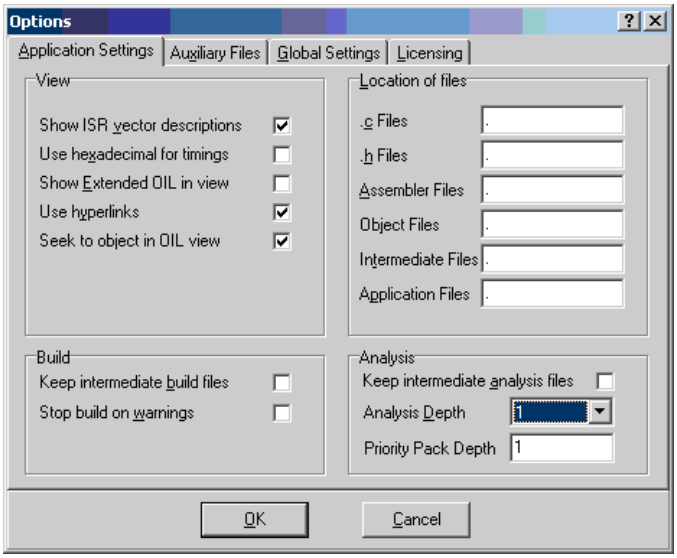


図 18-11 分析の深さ（'analysis depth'）を設定する

粗い分析では 2 つの概算（スケジューラビリティテストとアンスケジューラビリティテストを 1 つずつ）を行います。これらの概算の結果として、システムは以下のようなカテゴリに分類されます。

- 確実にスケジューラブル
- 確実にアンスケジューラブル
- 不確実にスケジューラブル

3 番目の結果となった場合は、分析の深さを 9 にしてください。すると、不確実にスケジューラブルであると分類されたシステムが実際にはスケジューラブルであるのかどうかを確かめるための分析が、再度行われます。

18.3 センシティビティ分析

センシティビティ分析は、システム限界を探るために使用されます。この分析を行うと、以下の情報を得ることができます。

- どのようなクロック速度が適当か。
- タスクまたは ISR の実行時間として、最大でどのくらいまで許容されるか。
- ある特定のリソースをどのくらいの時間保持できるか。
- 割込みをどのくらいの時間ディセーブルにできるか。
- レスポンスの実行時間を変えてもデッドラインに対応できるか。

センシティビティ分析を行うと、どのような変化を与えるとアンスケジューラブルシステムをスケジューラブルにできるかを明らかにすることができます。タスクと ISR の実行時間を最適化できる場合、それをわずかに短くするだけでシステムをスケジューラブルにできるかもしれません。

あるいは、既存のアプリケーションに付加的な機能を追加したい場合、センシティビティ分析を使用してどのタスクまたは ISR 上にどのくらいの余裕があるかを調べることができます。

タスクと ISR のセンシティビティは、以下の要素に対して判断されます。

- プロセッサのクロック速度に対するセンシティビティ
- 実行時間に対するセンシティビティ
- リソースと割込みの保持時間に対するセンシティビティ
- レスポンスデッドラインに対するセンシティビティ

図 18-12 は、サンプルシステムについて行ったセンシティビティ分析の結果を示しています。

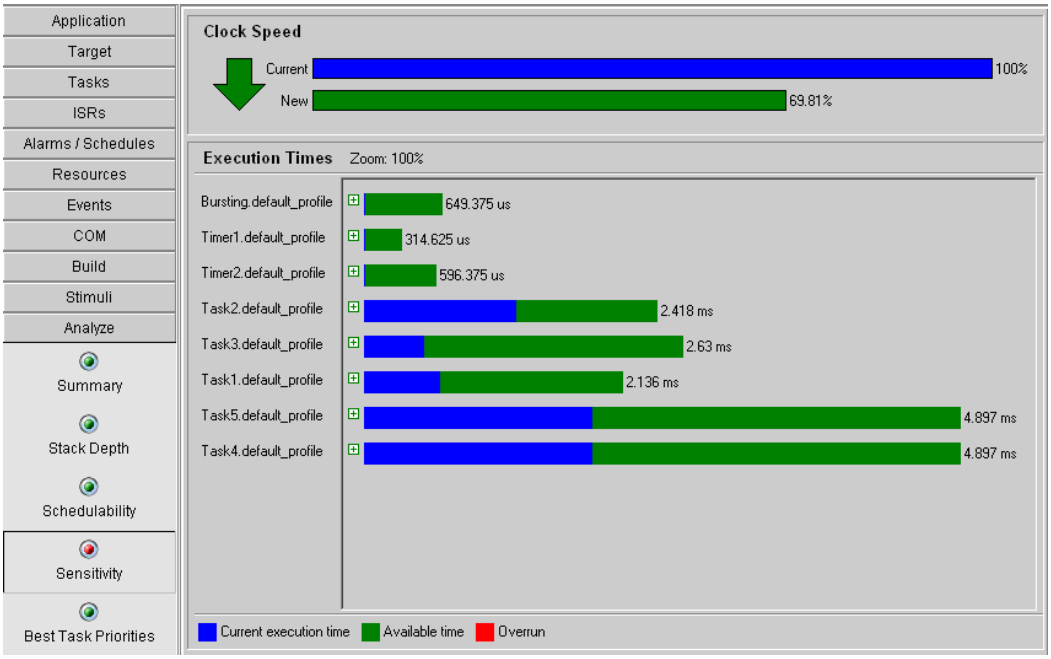


図 18-12 センシティビティ分析の結果（グラフィック表示）

センシティビティワークスペースの上部には、システムがスケジューラブルであるために必要な最小クロック速度が表示されています。また、実行時間、ロッキング時間、およびレスポンス生成時間に対するセンシティビティは、ワークスペースの下部に表示されています。

結果はカラーで表示されます。青色は現在の実行時間を示し、緑色は許容される最大時間を示し、また赤色はオーバーランを示します。

各バーの端に表示されている数値は最大の実行時間、リソース保持時間、または割込みがディセーブルになる時間です。これらの値になるまで、スケジューラブルなシステムを維持することができます。

表示される値はそれぞれ相互排他的な値です。つまり、それらのうちどれか 1 つを実現することにより、システムはちょうどスケジューラブルな状態となります。

センシティビティ分析レポートは、テキスト形式でも表示できます。この一例を図 18-13 に示します。

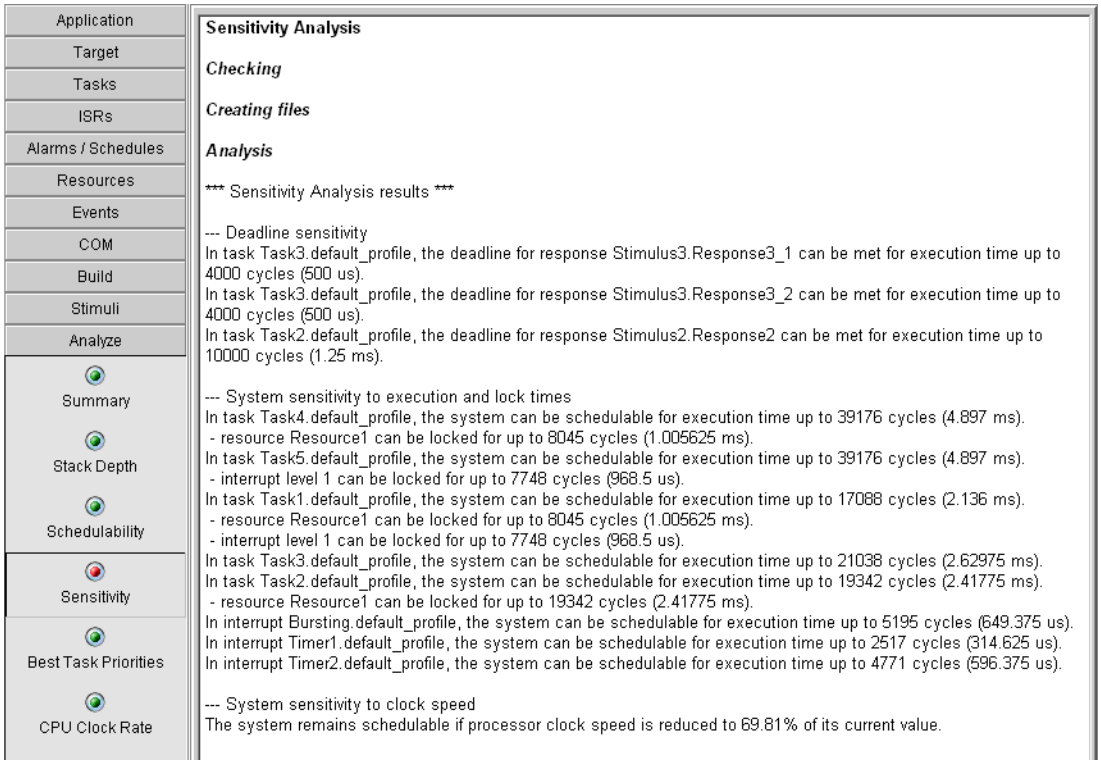


図 18-13 センシティビティ分析の結果（テキスト表示）

センシティビティ分析の結果に基づき、アプリケーションを修正することができます。以降の項で、分析結果の解釈について説明します。

18.3.1 クロック速度に対するセンシティビティ

「現在の速度」として表示される値は、常に CPU クロック周波数の 100% の値です。「新しい速度」は、システムをスケジューラブルにするために必要なクロック速度のパーセンテージになります。

新しい数値が 100% より小さい場合は、ターゲットハードウェアのクロック速度を下げる余地があります。これは、電力消費を最小限に抑える必要があるデバイスの場合などに役立ちます。

この新しい数値が 100% より大きい場合は、システムをスケジューラブルにするために CPU のクロック速度を高める必要があります。この分析により、ユーザーはアプリケーションをスケジューラブルにするために CPU のクロック速度を最低でもどのくらい高めればよいかを知ることができます。

18.3.2 実行時間に対するセンシティビティ

センシティビティ分析が終わると、デフォルトでは実行時間に対するセンシティビティがワークスペースに表示されます。各バーには、各タスクまたは ISR の現在の実行時間（青色）と最大許容実行時間（緑色）が示されます。

システムがスケジューラブルでない場合は、バーは赤色で表示され、オーバーランを示します。これは、タスクまたは ISR の現在の実行時間が、許容される最大実行時間をどのくらい超えているかを示すもので、図 18-14 のように表示されます。

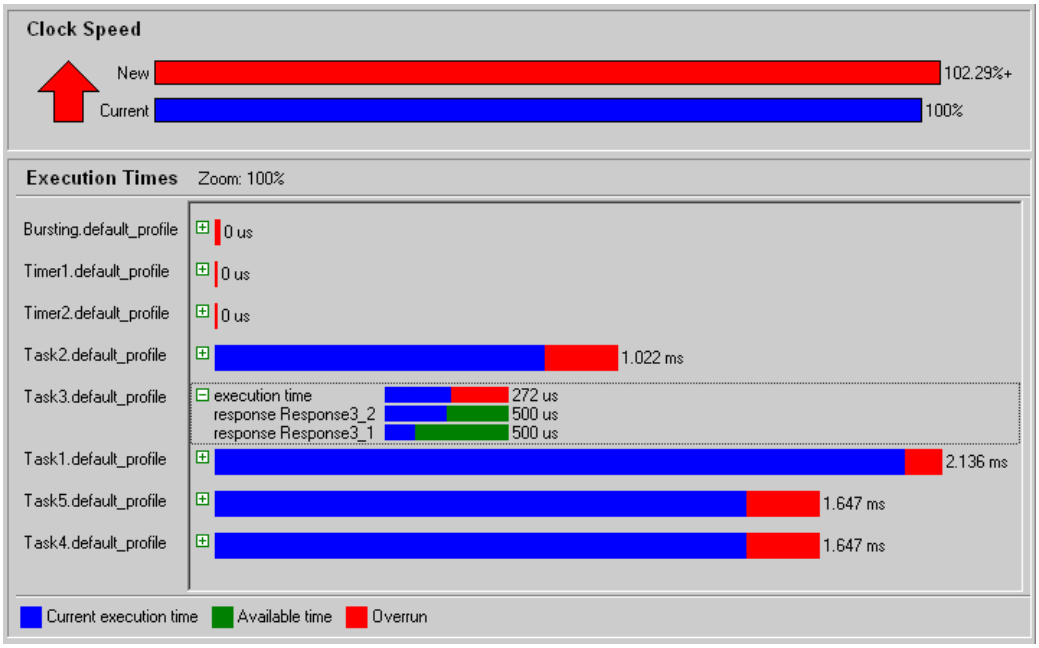


図 18-14 オーバーランを示すセンシティビティ分析

レポートされる実行限界は相互排他的です。つまり、システムがちょうどスケジューラブルな状態を保つには、任意の 1 つのタスクまたは ISR についてのみ、実行時間をセンシティビティ分析で示された最大時間に変更することができます。

複数のタスクまたは ISR の実行時間を変更した場合は、センシティビティ分析をもう一度実行してそれらの変更の妥当性を検証する必要があります。

この分析により、ユーザーのタスクまたは ISR に対する変更が何の効果もないことがわかったとしても、その情報はグラフには表示されません。一方、テキスト形式のレポートには、それらのタスクや ISR の実行時間を変更しても何の効果もなさそうである、ということが示されます。

この機能によって、ユーザーは自分の作業の的を絞ることができ、実行時間を短縮してアンスケジューラブルシステムをスケジューラブルにするための情報を得ることができます。

センシティビティ分析が終わると、デフォルトでは実行時間に対するセンシティビティがワークスペースに表示されます。図 18-15 のように、各タスクまたは ISR についての情報を展開し、リソースと割込みロッキング時間に対するセンシティビティを見ることができます。

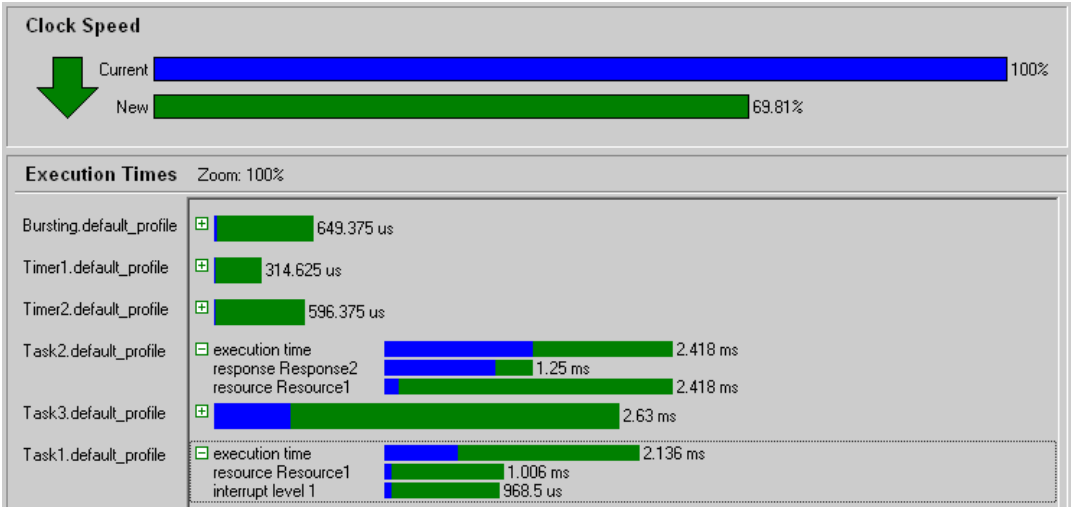


図 18-15 センシティビティ分析で割込みとロッキング時間を表示する

センシティビティ分析は、タスクまたは ISR によりロックされるリソースやディセーブルになる割込みについて、現在のロック時間と、実際にロックまたはディセーブルにできる最大時間を出力します。

各タスクまたは ISR についてのリソースと割込みロック時間は相互排他的なものです。リソースと割込みのロック時間についてさらにオーバーヘッドがある場合、センシティビティ分析により算出された最大実行時間を利用できるのは、いずれか 1 つのリソースロックまたは割込みロックのみに限られます。もちろん、複数のロックの実行時間を提示された範囲内で少しずつ値を調整しながが分析を再実行し、その妥当性を検証していくことも可能です。

18.3.3 デッドラインに対するセンシティビティ

センシティビティ分析が終わると、デフォルトでは実行時間に対するセンシティビティがワークスペースに表示されます。各タスクまたは ISR についての情報を展開することにより、明示的なレスポンスデッドラインに対するセンシティビティを確認することができます。

デッドラインが定義されている各レスポンスについて、センシティビティ分析は、レスポンス生成処理の現在の実行時間（これはユーザーがタイミングモデルをビルドする時に定義します）、およびシステムをスケジューラブルのままに保つために守らなければならないレスポンス生成処理の最大時間を出力します。オーバーランがある場合は赤色で表示されます。

各タスクまたは ISR のデッドラインに対するセンシティビティ分析の結果は「相補的（'complementary'）」です。つまり各レスポンスの実行時間を分析で算出された最大値にしても、システムはスケジューラブルな状態を維持します。

18.4 ベストタスクプライオリティ分析

ベストタスクプライオリティ分析は、可能であれば、タスク優先度を割り当ててシステムをスケジューラブルにする目的で行われます。また、アプリケーションに必要なスタックスペースを最小限に抑えるために、内部リソースを共有するタスクを明らかにします。ユーザーのアプリケーションに内部リソースがすでに含まれている場合は、それらは分析に含まれます。

RTA-OSEK GUI はこの分析の結果をグラフ形式で表示します。一例を図 18-16 に示します。

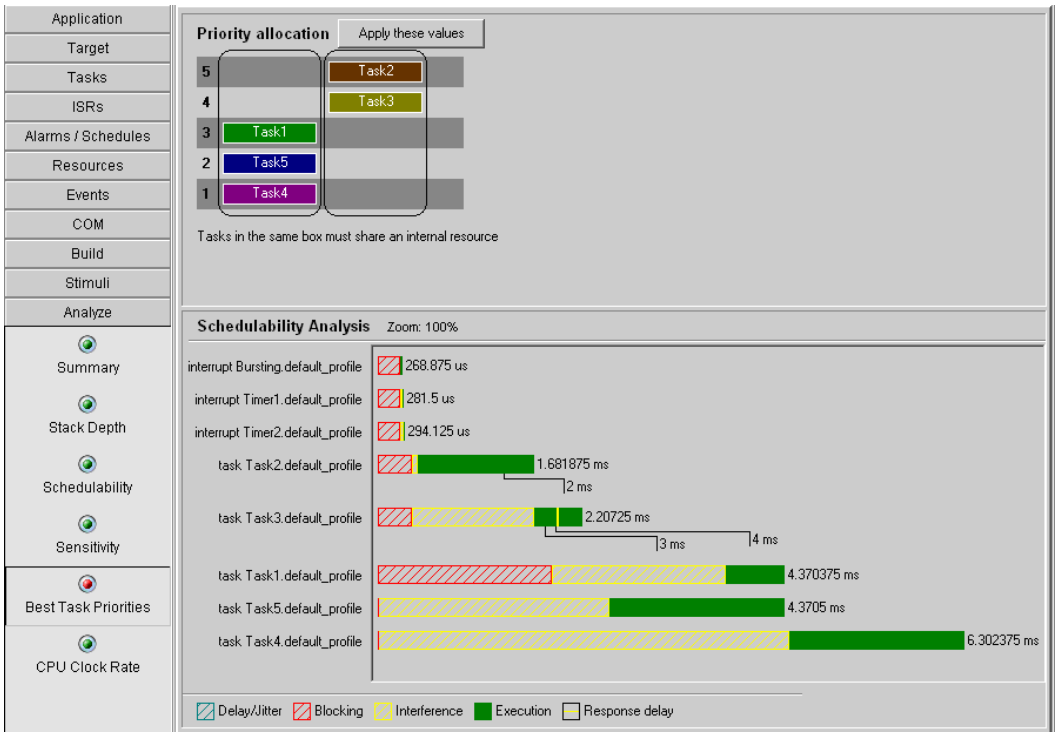


図 18-16 ベストタスクプライオリティ分析の結果（グラフィック表示）

ベストタスクプライオリティ分析の結果は、図 18-17 に示すようにテキスト形式で表示することもできます。

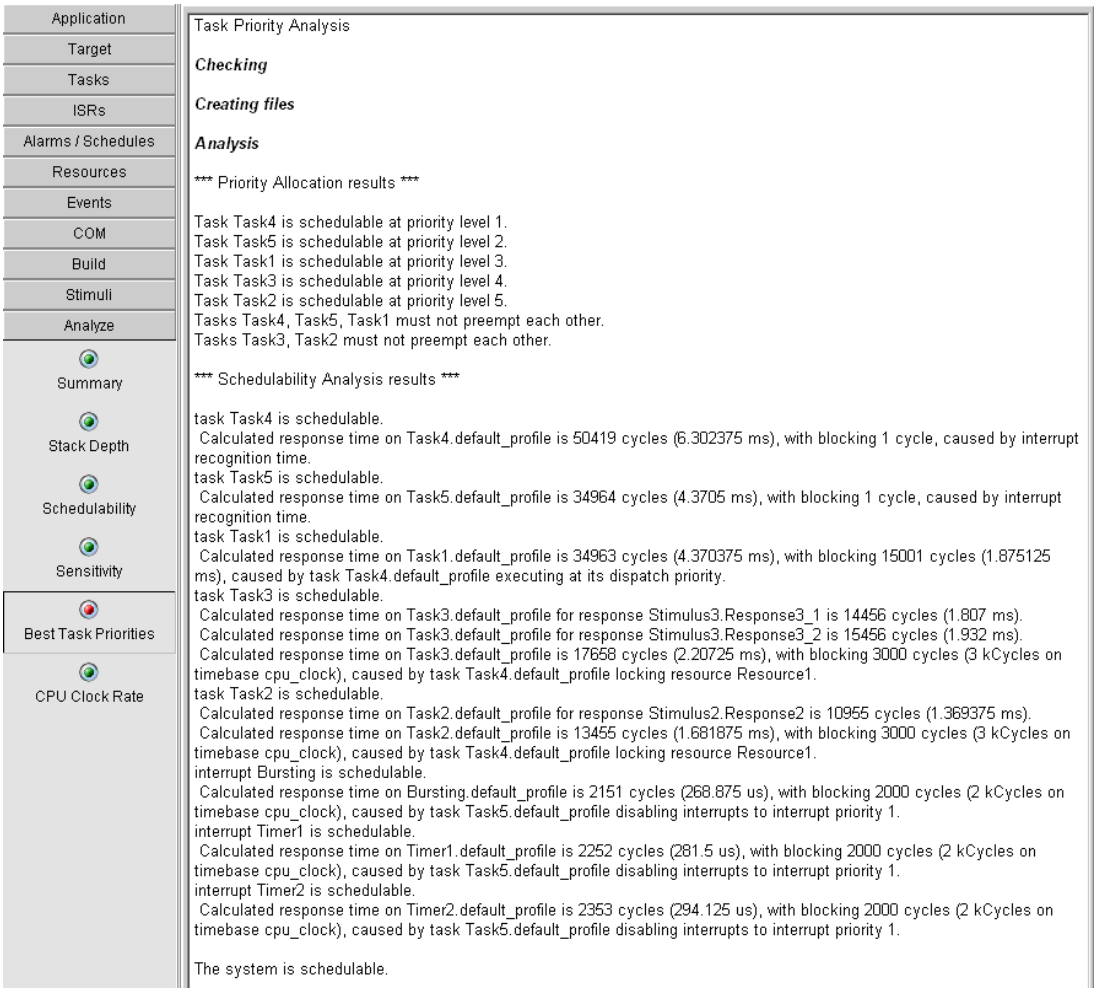


図 18-17 ベストタスクプライオリティ分析の結果（テキスト表示）

優先度の割り当てについては、各タスクが算出された最良の優先度とともに表示されます。また、タスクは内部リソースを共有できるかどうか（そうすることによりアプリケーションのスタックスペースを最小化できるかどうか）に従って分類されます。ワークスペースの下の部分には、ユーザーがこれらの変更を適用すると仮定した場合のスケジューラビリティ分析の結果が表示されます。

内部リソースを共有するタスク数の最大化には、システムに対する 2 つの重要な効果があります。

- システムの合計所要スタックが最小化されます。

  通常、任意のタスクセットについてのワーストケースのスタック使用量は、各タスクのワーストケースのスタック使用量の合計です。これらのタスクが内部リソースを共有する場合、ワーストケースのスタック使用量はそのリソースを共有するタスクのうち、最大量のスタックを使用するタスクのスタック使用量になります。

- 複数のタスクで内部リソースを共有するようにすると、スタック使用量が少なくなるのと引き替えに自由度が低下し、システムのスケジューラビリティが低下することが予想されます。しかし、タスク切り替えのために生じる付加的なオーバーヘッドがタスクの実行時間の大部分を占める場合は、タスク間で内部リソースを共有することによりスケジューラビリティが向上する可能性があります。

18.4.1 優先度が低くなければならないタスク群

ある特定のタスク群の優先度を他よりも高くしておかなければならないことがわかっている場合は、その制約条件を RTA-OSEK GUI に入力してください。

これを入力すると、たとえば、所定の実行順序を保証することができます。したがって、たとえば、`Task3` と `Task4` は一緒に起動できるけれども、`Task4` で使用するデータは `Task3` で準備されるので、`Task3` を先に実行する必要がある、といった具合です。したがって、`Task4` は `Task3` にとって「優先度が低くなければならないタスク（'required lower priority task'）」です。

各タスクについて、優先度が低くなければならないタスクがあれば、図 18-18 のように定義してください。

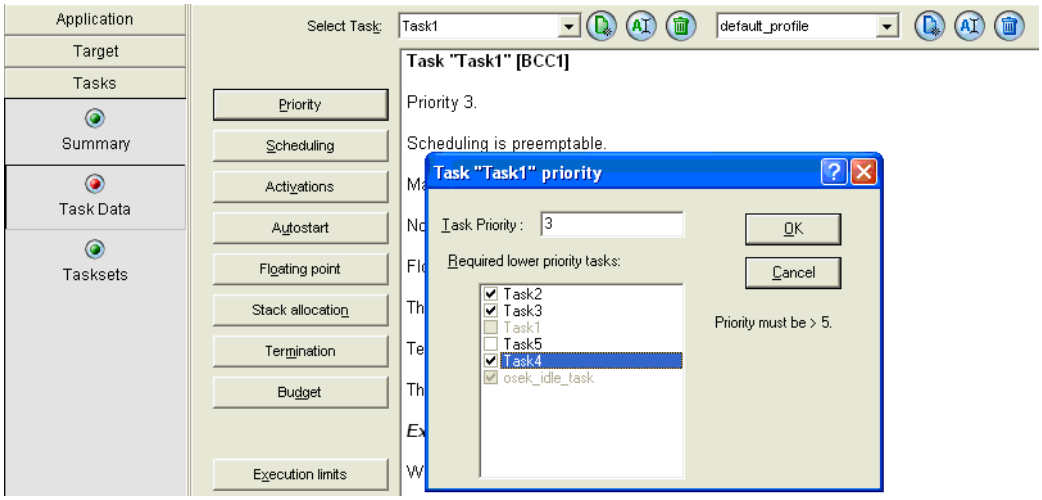


図 18-18 優先度が低くなければならないタスクを選択する

ベストタスクプライオリティ分析によって優先度の割り当てが行われる際には、優先度が低くなければならないタスクに関する制約条件に基づき、システムがスケジューラビリティ要件に最も的確に対応できるような優先度順序が求められます。

一般に、この「自動優先度割り当て」機能を使用する場合、優先度に関しての制約が少なければ少ないほど、優れた優先度順序を定義できます。つまり、優先度条件の宣言において、絶対に必要な制約条件だけを定義してください。

18.5 クロックレートの最適化

CPU クロックレートの最適化の考え方は、ベストタスクプライオリティとよく似ていますが、スペースではなく時間について最適化を行う点が異なります。スケジューラブルシステムを実現できる最低限のクロックレートを探します。

CPU クロックレートの最適化では、タスク優先度を割り当て直すことにより現在よりも低いクロック周波数でもシステムをスケジューラブルにできる場合は、タスク優先度を割り当て直します。

図 18-19 の例には、CPU クロックレート最適化の結果がテキスト形式で表示されています。

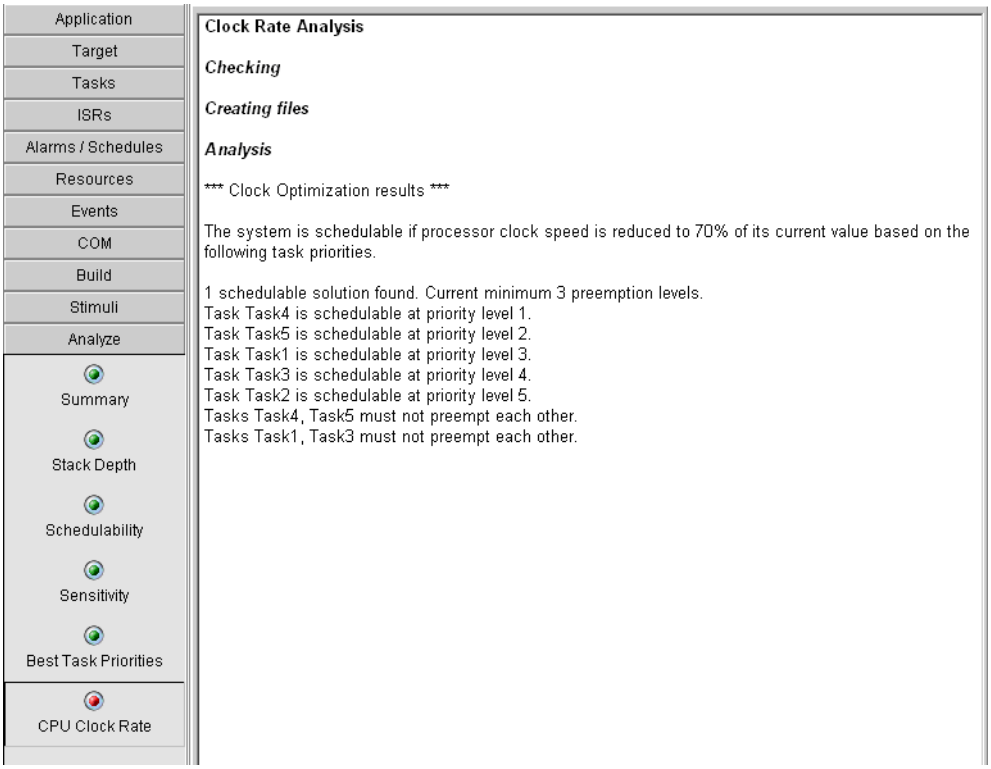


図 18-19 CPU クロックレート最適化分析の結果（テキスト表示）

CPU クロックレート最適化の結果は図 18-20 のようにグラフィックでも表示できます。

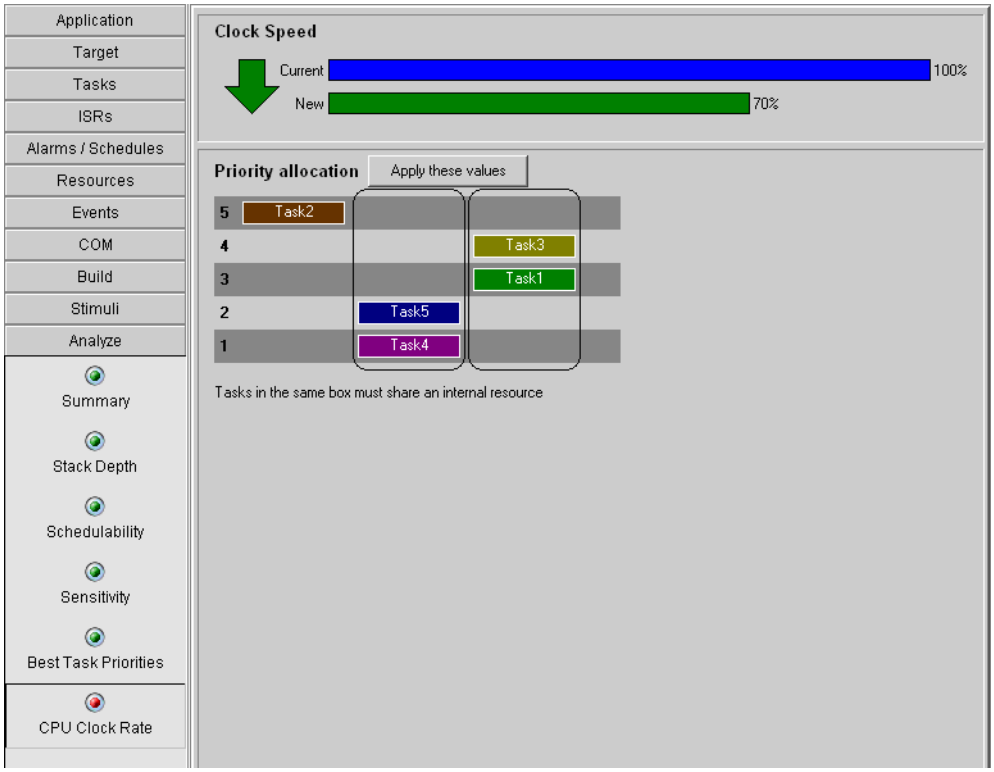


図 18-20 CPU クロックレート最適化分析の結果（グラフィック表示）

クロック最適化は、ベストタスクプライオリティ分析に基づく優先度割り当てとセンシティビティ分析を組み合わせたものと考えることができます。

- クロックのセンシティビティ（18.3 項を参照してください）の場合と同様に、クロック周波数の変更による効果は実行時間（クリティカル実行時間、およびリソースと割込みのロック時間を含みます）にしか及びません。クロック周波数を変更しても、アラーム間やスケジュール上の到達ポイント間のデッドラインとディレイは変わりません。
- 優先度割り当ての場合と同様に、タスク優先度も割り当て直すことができます。優先度の再割り当てについて制約条件を設ける必要がある場合は、優先度が低くなければならないタスクを指定してください。

電力消費や放熱についてシステムに重大な要件がある場合は、最終的なアプリケーションにおいて、このクロック最適化を行うことをお勧めします。

18.6 まとめ

- RTA-OSEK GUI には、RTA-OSEK Planner を使用してユーザーのアプリケーションのタイミングモデルを分析する機能があります。
- スタック分析では、アプリケーションのワーストケースのスタック使用量を明らかにし、算出されたアプリケーションの実行時の挙動に基づいてスタックスペースを効果的に重ね合わせることのできるような状況を把握することができます。
- スケジューラビリティ分析では、考えられるすべてのスティミュラス到達について、ランタイムにアプリケーション内のすべてのレスポンスデッドラインに対応できるかどうかを知ることができます。アプリケーションがアンスケジューラブルであることがわかった場合は、それをスケジューラブルにするための方法がいくつかあります。
- センシティビティ分析では、アプリケーションの限界値を探り、システムをアンスケジューラブルにしている箇所を検知したり、将来拡張できる余地を調べたりすることができます。
- ベストタスクプライオリティ分析は、システムをスケジューラブルにするために、タスクにとって最良の優先度割り当てを明らかにします。実行順序が重要なタスクについては、優先度がそのタスクよりも低くなければならないタスクを定義することができます。ベストタスクプライオリティ分析では、スタックスペースを最小化するために、どのタスクが内部リソースを共有できるかを明らかにすることもできます。
- CPU クロックレートの最適化はベストタスクプライオリティと似ていますが、スタックスペースではなく CPU クロックレートを最低のレベルに抑えるための最適化を行うものです。

19 RTA-OSEK をコマンドラインから使用する

19.1 操作の概要

19.1.1 機能

rtabuild というツールは、RTA-OSEK の各種ツールを呼び出して以下の処理を行うコマンドラインプログラムです。

- **カーネルとアプリケーションサポートデータのビルド**
  アプリケーションをコンパイルしリンクするために必要な RTA-OSEK ヘッダ、C ファイル、およびアセンブラファイルを作成します。
- **スケジューラビリティ分析**
  すべてのタスクと ISR が、ワーストケースにおいてデッドラインやその他の制約条件を満たしているかどうかを判断します。
- **センシティビティ分析**
  システムをスケジューラブルな状態にしておくためには各タスクと ISR のタイミングパラメータ値がどの程度変動可能であるかを算出します。
- **ベストタスクプライオリティ**
  スケジューラブルシステムを実現できるための最小限のプリエンプションレベル数が設けられるように、タスク優先度の割り当てを行います。
- **クロック最適化**
  スケジューラブルシステムを実現できる最低限のプロセッサクロックレートと、そのレートで稼働するために必要なタスク優先度を明らかにします。

rtabuild は OIL 構文で書かれた設定ファイルを読み取ります。

19.1.2 メッセージ

rtabuild は処理の過程でさまざまなメッセージを出力します。各メッセージは以下のように分類されます。

- **情報**（'Information'）メッセージ
  使用されたメモリ量や構造体のサイズなどの有用な情報を出力します。
- **ワーニング**（'Warning'）メッセージ
  一般的でない条件や冗長な条件、正確に表現できないために丸め誤差が生じるような条件などが入力ファイルに定義されている場合に出力されます。ワーニングが出力されても、出力ファイルは作成され、アプリケーションはビルド可能です。
- **エラー**（'Error'）メッセージ
  入力ファイル内に、分析を正しく行えなくしたり正しい情報を出力できなくするような矛盾がある場合に発行されます。ツールは、見つけられるすべてのエラーをレポートしてから終了します。すべてのエラーを除去して処理を再実行してください。
- **フェイタル**（'Fatal'）メッセージ
  入力ファイル内にリカバリが不可能な条件が定義されている場合に出力されます。rtabuild はこのような条件を検知すると、フェイタルエラーメッセージを出力して直ちに処理を中止します。

19.1.3 戻り値

実行の終わりに、rtabuild は下の表に示すようなコードを返します。

値	説明
0	リクエストされた処理に成功しました。
1	エラーまたはフェイタルメッセージが出力され、処理が終了しました。
2	RTA-OSEK Planner の実行中にユーザーが処理をキャンセル（ESC）しました。

値	説明
3	ユーザーが処理をアボート（^C/^Break）しました。
4	システムはスケジューラブルではありません。 （-u コマンドラインオプションが指定されている場合は、0 が返されます。）
5	優先度割り当ての結果、スケジューラブルシステムを生成できませんでした。

19.1.4 コマンドラインオプション

rtabuild は、以下の構文を用いてコマンドラインから呼び出されます。

```
rtabuild [[*options*]] [*input_file*]
```

ハイフンに続く文字がコマンドラインオプションとして解釈されます。各コマンドラインオプションは任意の順序で使用できます。

コマンドラインには、1 つまたは複数の入力ファイルを指定できます。すべてのファイルは OIL 構文を用いて作成されていなければなりません。複数の入力ファイルを指定した場合は、それらの入力された順に処理されます。各ファイルのテキストがその直前のファイルのテキストの後に追加され、処理対象となる一時ファイルが作成されます。

指定できるオプションは『RTA-OSEK リファレンスガイド』に記載されています。

19.1.5 出力ファイル

処理中、rtabuild は中間ファイル、一時ファイル、およびリストファイルを生成する可能性があります。一時ファイルは処理完了時に必ず削除されます。中間ファイルは通常は削除されますが、-k オプションを使用することで残しておくことができます。リストファイルは、-o オプションを選択した場合のみ生成されます。

20 RTA-OSEK と RTA-TRACE の併用

RTA-TRACE は、組み込みシステム用の「ソフトウェアロジックアナライザ」で、組み込みアプリケーションの開発においてシステムのデバッグやテストに役立つ、強力な機能を豊富に備えています。特に、製品用ビルドによって作成されたアプリケーションシステムについて、ランタイムにおけるシステム内部の状態を、正確に把握することができます。

RTA-OSEK には RTA-TRACE 用オプションパラメータがあり、これらを RTA-OSEK GUI で設定することができます。GUI の操作方法は非常にわかりやすくできているので、ここでは、各パラメータの内容と簡単な設定方法のみ紹介します。

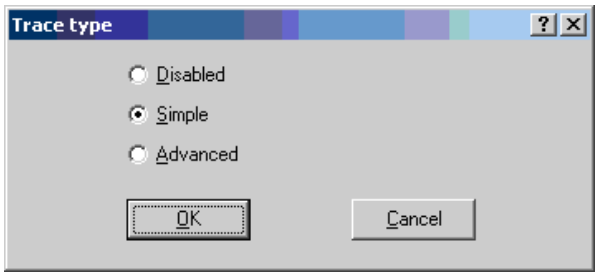
なお本章の内容は、RTA-OSEK GUI の操作方法をすでに熟知しているユーザーを対象としています。

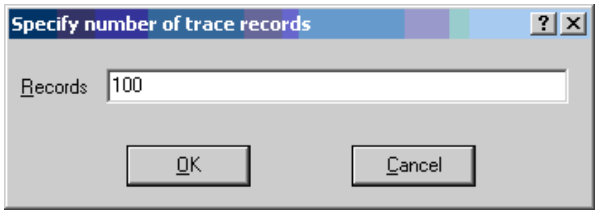
RTA-TRACE 用オプションは、RTA-OSEK GUI ウィンドウの左下にある **RTA-TRACE** タブからアクセスできます。

20.1 コンフィギュレーション（'Configuration' ペイン）

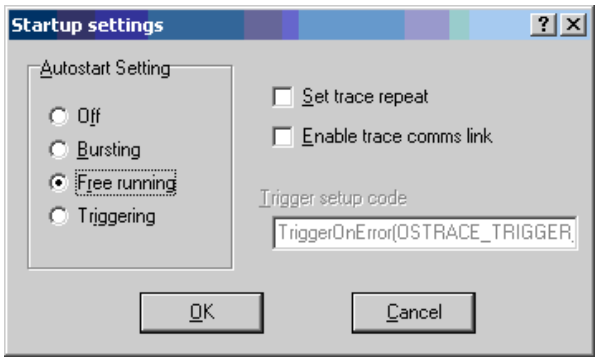
このペインでは以下のオプションを設定できます。

オプション	説明
Trace Type	トレース処理を無効／有効（シンプルまたはアドバンスド）にします。 アドバンスドトレースはシンプルトレースよりも詳細なトレースを行うため、トレースレコードの数が増加します。
Compact IDs	コンパクトトレースフォーマットを有効にします。バッファスペースの削減のため、タスクトレースポイント ID は 4 ビット、トレースポイントおよびインターバル ID は 8 ビットになります。その他の ID（タスク、リソースなど）は 8 ビットです。 コンパクト ID が指定されない場合は、トレースポイント、タスクトレースポイント、インターバルの各 ID は 12 ビット、その他の ID は 16 ビットとなります。
Compact Time	*compact*（16 ビット）、*extended*（32 ビット）の時間フォーマットを選択します。 このオプションは一部のターゲットには存在しません。
Trace Stack	スタック使用量を記録するかどうかを指定します。
Target Triggering	ランタイムターゲットトリガを使用するかどうかを指定します。
Buffer Size	ターゲット上に予約されるトレース情報用バッファのサイズを指定します。ここには、バイト数ではなくレコード数を入力します。実際のバッファサイズは、時間と ID のサイズに応じて変わります。

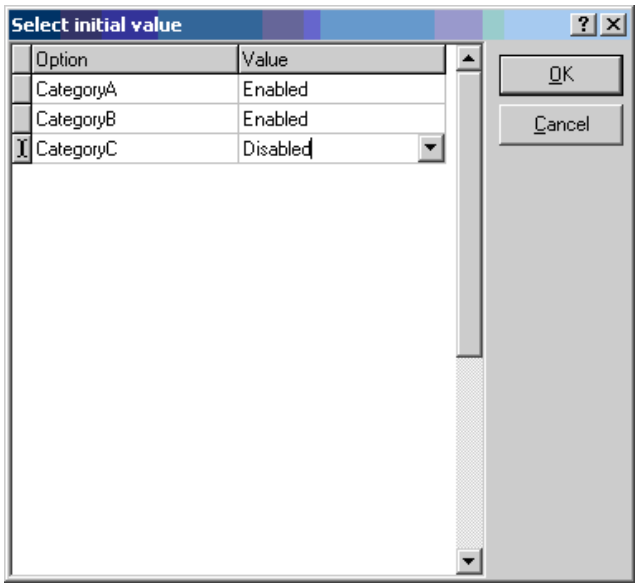




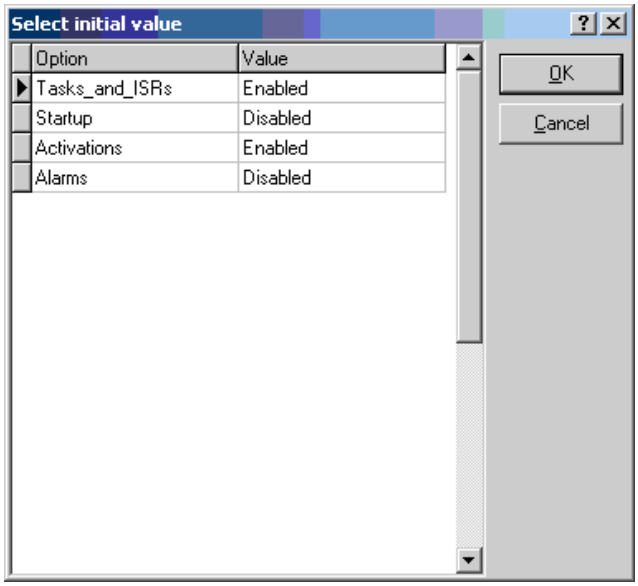
オプション	説明
Autostart	トレースを自動的に開始するかどうか、および開始時のトレースモードを指定します。 トリガを選択した場合は、トリガセットアップコード（`TriggerOn...`）を入力します。トリガ API についての詳細な情報は、『RTA-TRACE ユーザーズガイド』を参照してください。



Initial Categories	ランタイムカテゴリ（20.5 項、および『RTA-TRACE ユーザーズガイド』を参照してください）が定義されている場合、このダイアログボックスで、トレース開始時にどのユーザー定義のランタイムカテゴリが有効になるようにするかを指定できます。以下の図では、3 つのランタイムカテゴリがユーザー定義されていて、そのうちの 1 つが初期状態においては無効になるように設定されています。

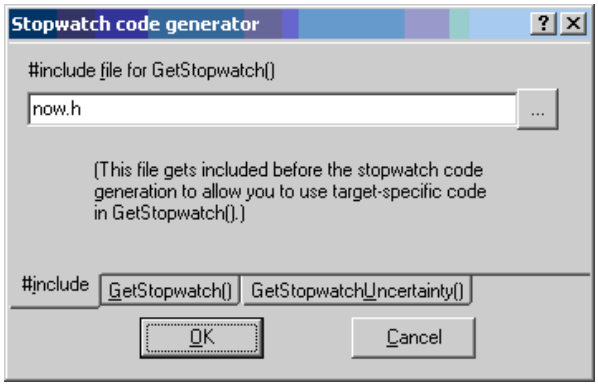


オプション	説明
Initial Classes	トレース開始時にどのレコードクラスが有効になるようにするかを指定します。下の図では、タスクと ISR、起動、イベントトレースの 3 つをランタイムに有効／無効にすることができ、ここでは初期状態においてイベントトレースが無効となるように指定されています。



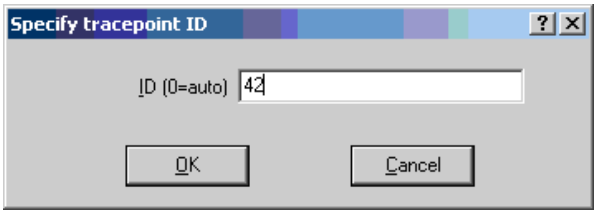
Stopwatch

このダイアログボックスで、`GetStopwatch()` を実装する関数を指定します。以下の例では、ユーザー定義の `now()` という関数が指定されていて、この関数に対応するヘッダファイルは `now.h` となっています。

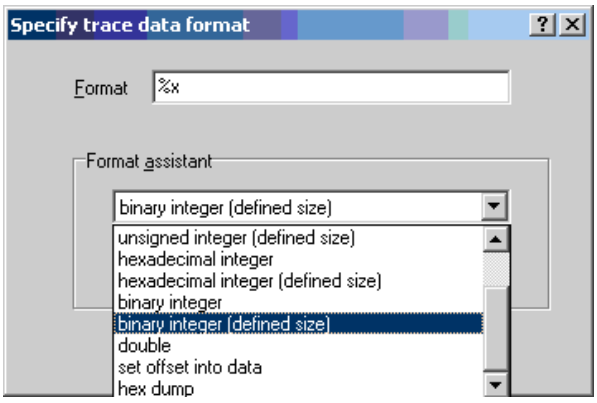


20.2 トレースポイント（'Tracepoints' ペイン）

このペインでトレースポイントを設定します。作成された新しいトレースポイントには自動的に ID が割り当てられますが、**ID** ボタンをクリックするとこれを任意に変更できます。



トレースポイントにデータが定義されている場合、フォーマット文字列（20.8 項を参照してください）を使用してデータの出力形式を設定することができます。



20.3 タスクトレースポイント（'Task Tracepoints' ペイン）

このペインでは、タスクトレースポイントを設定できます。作成された新しいタスクトレースポイントには自動的に ID が割り当てられますが、**ID** ボタンをクリックするとこれを任意に変更できます。

トレースポイントと同様にフォーマット文字列を使用できます。

20.4 インターバル（'Intervals' ペイン）

このペインでインターバルを定義できます。作成された新しいインターバルには自動的に ID が割り当てられますが、**ID** ボタンをクリックするとこれを任意に変更できます。

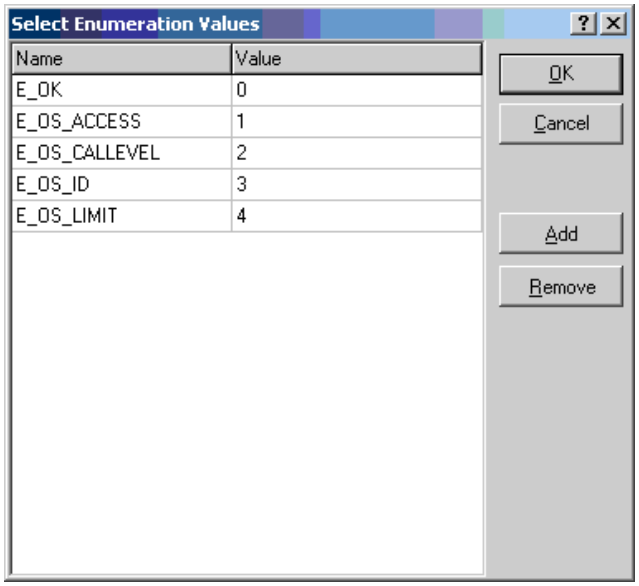
トレースポイントと同様にフォーマット文字列を使用できます。

20.5 カテゴリ（'Categories' ペイン）

このペインでトレースカテゴリとそのマスク値を定義できます。カテゴリについての詳細は、『RTA-TRACE ユーザーズガイド』に説明されています。カテゴリは、**filter** ペイン（20.7 項参照）で、ランタイムにおいて「常に有効」、「常に無効」、または「有効／無効」のいずれかに設定できます。

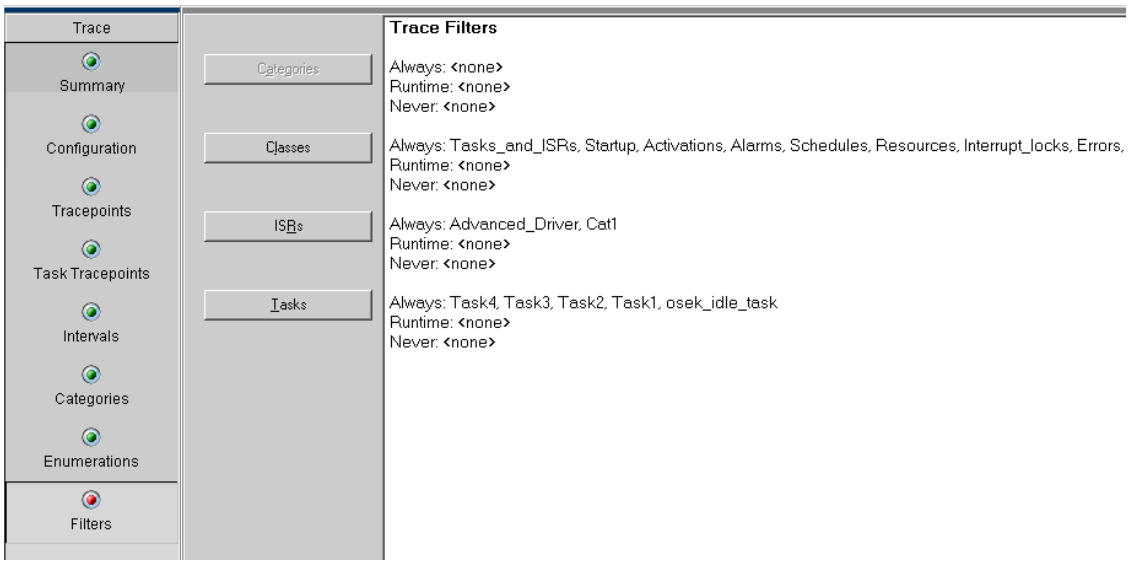
20.6 列挙（'Enumerations' ペイン）

このペインで列挙型 ID とその列挙子を定義できます。以下の例には、OSEK のエラーコードを表わす列挙子が表示されています。



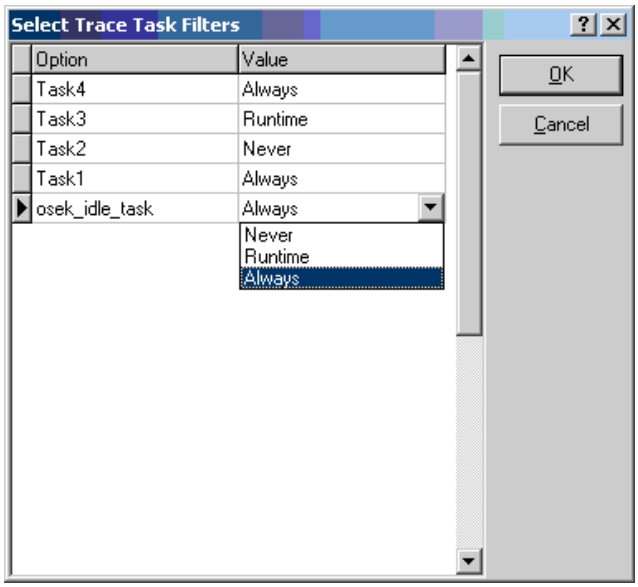
20.7 フィルタ（'Filter' ペイン）

このペインでイベントクラスとカテゴリのフィルタ設定とタスク／ ISR のトレースフィルタを設定できます。デフォルトではすべてのものがトレースされますが、ランタイムにおけるオブジェクトクラスのトレースはできる限り無効にしておいたほうが効率的です。同様に、一部のタスクや ISR のトレースを無効にしておくこともお勧めします。



各フィルタのステートはいかのいずれかです。

1. Always ー トレースは常に有効（デフォルト）
2. Never ー トレースは常に無効
3. Runtime ー ランタイムに有効／無効を切り替え可能す。



ランタイムフィルタの初期値はデフォルトでは無効になっていますが、Configuration ペイン（20.1 項を参照してください）の **Initial Classes** オプションで有効／無効を切替えることができます。

20.8 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。

重要

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。

20.8.1 フォーマット規則

フォーマット文字列には、C 関数の `printf()` の 1 番目の引数とほぼ同じ規則が適用されます。

- 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。

重要

`printf()` とは異なり、フィールドを出力する際、フィールドのポインタは現在の位置から自動的に移動しません。これは、1 つのフィールドを複数のフォーマットで出力する場合を考慮しているためです。

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。
`%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。

20.8.2 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`（虹の色が順に定義されています）内の値を出力する	`"%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。

21 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。

ヨーロッパ（フランス、ベルギー、ルクセンブルグ、イギリスを除く）

ETAS GmbH

Borsigstrasse 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

フランス、ベルギー、ルクセンブルグ

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

イギリス

ETAS Ltd.

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

Phone: +44 1283 - 54 65 12
Fax: +44 1283 - 54 87 67
E-mail sales@etas-uk.net
WWW: www.etasgroup.com

アメリカ

ETAS Inc.

3021 Miller Road
Ann Arbor, MI 48103
USA

Phone: +1 (888) ETAS INC
Fax: +1 (734) 997-9449
E-mail sales@etas.us
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.

4F, 705 Bldg. 70-5
Yangjae-dong, Seocho-gu
Seoul 137-889
Korea

Phone: +82 2 57 47-016
Fax: +82 2 57 47-120
E-mail sales@etas.co.kr
WWW: www.etasgroup.com

中国

ETAS (Shanghai) Co., Ltd.

2404, Bank of China Tower	Phone:	+86 21 5037 2220
200 Yincheng Road Central	Fax:	+86 21 5037 2221
Shanghai 200120	E-mail	sales.cn@etasgroup.com
P.R. China	WWW:	www.etasgroup.com

インド

ETAS Automotive India Pvt. Ltd.

No. 690, Gold Hill Square, 12F	Phone:	+91 80 4191 2585
Hosur Road, Bommanahalli	Fax:	+91 80 4191 2586
Bangalore, 560 068	E-mail	sales.in@etasgroup.com
India	WWW:	www.etasgroup.com