embOS

レファレンスドキュメント

ポジティブワン株式会社

目　　次

警告

この仕様書に書かれている明細事項は信頼の置けるものですが、完全に欠陥が存在しないことを保証するものではありません。この仕様書に記載されている情報は機能あるいは性能の改善によって予告なく変わることがあります。この仕様書が最新版であることを確認してください。この仕様書に記載されている事項が正しいと判断される場合、SEGGER（製造業者）はいかなる誤りや欠陥に対しても責任を負いません。製造業者いかなる機能保証および法的責任も負いません。製造業者は、特定の目的に対する商品性または適合性の黙示的な保証を一切否認します。

登録

メールでソフトウェアを登録します。アップデートの更新や通知をすぐに受け取るすことがで

きます。登録のために以下の情報を入力してください。

・氏名

・職業

・Ｅメールアドレスおよび電話番号

・商品名およびバージョンの番号

これらの情報を poc_sales@positive-one.com　に送信してください。

連絡先

ポジティブワン株式会社

〒150-0043 東京都渋谷区道玄坂 1-12-1 渋谷マークシティウエスト 22 階

poc_sales@positive-one.com

www.positive-one.com

ソフトウェア及び仕様書のバージョン

本仕様書では、現在のソフトウェアバージョンについて説明します。何か問題が発生した場合はご連絡ください。早急に対応します。詳述されていないトピックや手順の詳細についてはお問い合わせください。

Print date: September 13, 2011

前提事項

この文書はあなたが以下の知識を持つことを前提としています。

・アプリケーションを構築するために必要なソフトウェアツール（アセンブラ、リンカ、Ｃコンパイラ）

・Ｃプログラミング言語

・ターゲットプロセッサ

・ＤＯＳのコマンドライン

Ｃ言語の知識が十分でないと感じた場合、標準的なＣプログラミングを記述したカーニハンとリッチー（ISBN0-13-1103628）のプログラミング言語Ｃ及び標準的なＡＮＳＩ Ｃまでカバーした新版を推奨します。

本仕様書の利用方法

このマニュアルの目的は、CPU およびコンパイラ独立の embOS の紹介およびすべての embOS の API 関数の参考としていただくことです。embOS を迅速かつ簡単にスタートアップさせる場合には、ステップバイステップ方式で概説を含む embOS ドキュメントの CPU とコンパイラの細目マニュアルの章を参照してください。

Chapter 1　embOS の手引き

1.1　　embOS とは

embOS は、マイクロコントローラーの様々なリアルタイム･アプリケーションを開発するための組み込みオペレーティングシステムとして使用されるように設計された優先制御のマルチタスクシステムです。

embOS は RAM と ROM の両方で最小のメモリ消費量を実現し、高速で汎用性のある高性能のツールです。

1.2　　特徴

マイクロコントローラーの資源が限られていることは、embOS の開発プロセス全体を通じて常に気にかけられてきました。リアルタイムオペレーティングシステム（RTOS）の内部構造は、産業界のニーズに合わせて、さまざまな顧客との様々なアプリケーションに最適化されています。完全にソース互換の RTOS は各種マイクロコントローラーのために用意されており、リアルタイム･オペレーティング･システムとのリアルタイムプログラムを構築する方法を学ぶ価値のあるものとなっています。

embOS は、高度にモジュール化されています。これは、必要な機能のみがリンクされ、ROM サイズが非常に小さく抑えられることを意味します。最小のメモリ消費量は、ROM で 1 バイト以下、RAM で約 30 バイト（及びスタックのメモリ）です。embOS を使用することによっ

て柔軟性を失うことがないよう、またニーズに合わせてシステムをカスタマイズすることができるように 2 つのファイルがソースコードで提供されます。
作成したタスクは、お互いがそのようなセマフォ、メールボックス、イベントなどの通信メカニズムのパレットを使用して簡単かつ安全に通信することができます。

embOS の特徴:
·割り込み式スケジューリング：
優先順位の逆転が適用される場合を除いて、READY 状態にあるすべてのタスクの優先度が高いことを保証する。
·同一の優先度を持つタスクのラウンドロビンスケジューリング。
·プリエンプションは、全体のタスクのため、あるいはプログラムのセクションのために無効にすることができる。
·優先タスクは最大 225 個。
·各タスクは、個々の優先度を持つことができます=>タスクの応答はアプリケーションの要件に応じて正確に定義することができます。
·タスク数は無制限（メモリ使用可能量によってのみ制限される）
·セマフォ数は無制限（メモリ使用可能量によってのみ制限される）
·セマフォのタイプは 2 種類:リソースとカウンティング
·メールボックス数は無制限（メモリ使用可能量によってのみ制限される）
·メールボックスを初期化する際にメッセージのサイズと数を自由に決定することができる。
·ソフトウェアタイマ数は無制限（メモリ使用可能量によってのみ制限される）
·各 8 ビットのイベント
·時間分解能は自由に選択できる（デフォルトは 1ms）
·時間変数は簡単にアクセス可能。
·電源管理
·未使用の計算時間は、自動的に休止モードとなる。消費電力は最小限に抑えられる。
·割り込みサポート:
割り込みは、データを待っているものを除く任意の機能を呼び出すことができ、またタスクの優先度を作成、削除及び変更することができる。
割り込みはタスクを起動あるいは一時停止させることができ、また使用可能なすべての通信インスタンス（メールボックス、セマフォ、イベント）を使用してタスクと通信することができる。

·割り込み無効時間が非常に短い=>割り込み待ち時間が短い

·ネストされた割り込みが許可されている

·embOS 独自の割り込みスタックがある (オプション使用)

·簡単に開始するためのフレームアプリケーション

·デバッグバージョンは、実行時間チェックにより開発を簡素化する。

·指定されたライブラリを選択することによってプロファイリングとスタックチェックを実行することができる。

·UART を介して実行時間を監視することが可能（embOSView)。

·非常に迅速かつ効率的だが小さなコード。

·最小限の RAM 使用。

·コアはアセンブリ言語で書かれている。

·API は、アセンブリ言語、C または C + +コードから呼び出すことができる。

·ソース（BSP）としてマイクロコントローラ·ハードウェアの初期化。

Chapter 2　基本的概念

この章では、embOS の基本的な概念について説明します。比較的簡単な説明であり、他の章に進む前に読むことを推奨します。

2.1　タスク

タスクとは、マイクロコントローラーの CPU コア上で動作するプログラムです。マルチタスクカーネル（RTOS）がない場合、CPU 上で一度に一つのタスクしか実行することができません。これは、シングルタスクシステムと呼ばれています。リアルタイム·オペレーティング·システムは、一つの CPU 上で複数のタスクを実行することができます。タスクはすべて CPU 全体を"所有"しているかのように実行されます。タスクはスケジュールされており、RTOS はすべてのタスクをアクティブ化または非アクティブ化することができます。

2.1.1　スレッド

スレッドは同一のメモリレイアウトを共有するタスクです。二つのスレッドが同じメモリ位置にアクセスすることができます。仮想メモリが使用されている場合、同じ物理アドレスへの変

換が行われ同じアクセス権が使用されます。embOS のタスクはスレッドであり、それらはすべて同じメモリアクセス権および変換です（仮想メモリのシステム上）。

2.1.2 プロセス

プロセスは、独自のメモリレイアウトタスクです。通常、同じメモリ位置に二つのプロセスがアクセスすることはできません。異なるプロセスは、異なるアクセス権と（MMU の場合）異なる変換テーブルを持ちます。プロセスは、embOS の現在のバージョンではサポートされません。

2.2 シングルタスクシステム（スーパーループ）

組込みシステム設計の代表的な方法は RTOS なしであり、"スーパーループ･デザイン"とも呼ばれています。リアルタイムカーネルは使用されていないため、割り込みサービス･ルーチン（ISR）はソフトウェアや重要な操作（割り込みレベル）のリアルタイム部品として使用される必要があります。このタイプのシステムは、一般的に小型で単純なシステムやリアルタイムの動作が重要でない場合に使用されます。

リアルタイムカーネルは使用されておらずスタックが一つだけ使用されているため、小型のアプリケーションではプログラム（ROM）及び RAM のサイズは小さくなります。当然、スーパーループアプリケーションとのタスク間の同期の問題はありませんが、プログラムが大きすぎる場合スーパーループをメンテナンスすることが困難になることがあります。あるソフトウェアコンポーネントが別のコンポーネント（ISR のみ）に割り込みされることはないので、一つのコンポーネントの応答時間は、システム内の他のすべてのコンポーネントの実行時間に依存します。従って、リアルタイム動作はうまくいきません。

2.2.1 長所･短所

長所

･単純な構造（小型アプリケーションの場合）

･スタック使用量が少ない（一つしか必要としない）

短所

･「遅延」機能がない

·スリープモードがない（消費電力大）
·プログラムが大きくなるにつれてメンテナンスが困難になる
·ソフトウェアコンポーネントのタイミングは他のすべてのコンポーネントに依存
一カ所での小さな変化がほかの場所で重大な影響を及ぼす可能性がある
·モジュラープログラミングに勝る
·割り込みのみのリアルタイム動作

2.2.2 スーパーループアプリケーションにおける embOS の使用

スーパーループ·アプリケーションでは本来タスクは使用されないため、マルチタスクを使用するためにアプリケーションが変換されない限り、RTOS の最大の長所が生かされません。しかし、一つでもタスクがある場合、embOS を使用すると、以下のような利点があります。
·ソフトウェアタイマが使用可能
·省電力：アイドルモードを使用できる
·後の拡張は別のタスクに入れることができる

2.2.3 スーパーループからマルチタスクへの移行

通常、アプリケーションが一定時間存在しており、単一のタスク、スーパーループアプリケーションとして設計されています。ある時点で、この方法の欠点により RTOS を使用するということが決心されます。その問題とは「これをどのように行うのですか？」というものです。最も簡単な方法は、embOS に付属する開始アプリケーションを利用し、"スーパーループコード"をタスクに入れることです。この時点で、このタスクのスタックサイズが十分であることを確認する必要があります。その後、ソフトウェアに追加される機能は、1 つまたは複数のタスクに入れることができ、スーパーループの機能も複数のタスクに配布することができます。

2.3 マルチタスクシステム

マルチタスクシステムにおいて、異なるタスク間で CPU 時間を配布する方法は様々です。このプロセスはスケジューリングと呼ばれます。

2.3.1 タスクスイッチ

基本的なタスクスイッチは 2 種類あり（協調タスクスイッチとプリエンプティブタスクスイッチ)、コンテキストスイッチとも呼ばれます。

2.3.2 協調タスクスイッチ

協調タスクスイッチは、タスク自身によって実行されます。タスクの協力、ゆえに名前を必要とします。OS_Delay()または OS_WaitEvent()などブロッキング RTOS 機能を呼び出すことにより、タスクが自身をブロックします。

2.3.3 プリエンプティブタスクスイッチ

プリエンプティブタスクスイッチは、割り込みによるタスクスイッチです。通常、他の優先度の高いタスクが実行可能な状態になり、その結果現在のタスクが中断されます。

2.3.4 プリエンプティブマルチタスク

embOS のようなリアルタイムシステムはプリエンプティブマルチタスクのみで動作します。リアルタイム･オペレーティング･システムは、定義された時間に作業を中断したり必要に応じてタスクスイッチを実行したりするためにタイマ割り込みを必要とします。割り込みタスクであるかどうかに関わらず READY 状態にある最も優先順位の高いタスクが常に実行されます。ISR が優先順位の高いタスクを実行可能にした場合にタスクスイッチが作動し、中断されたタスクが返される前にタスクが実行されます。

2.3.5 協調マルチタスク

協調マルチタスクは、すべてのタスクの協調を期待しています。タスクの切り替えは実行中のタスクスイッチが OS_Delay()または OS_Wait()などのブロッキング関数を呼び出して自身をブロックする場合にのみ起こります。そうでない場合、システムは"ハング"し、他のタスクは最初のタスクが行われている間、CPU によって実行されることができません。これは下記の図に示されています。 ISR が優先順位の高いタスクを実行可能にした場合でも、タスクの切り替えが起こる前に、割り込みされたタスクは返され終了されます。

純粋な協調マルチタスクシステムには、優先度の高いタスクを実行可能な状態にするまでの応答時間が長いという欠点があります。そのため、組み込みシステムにおいては滅多に使用されません。

2.4 スケジューリング

実行するタスクを決定するスケジューラとよばれる様々なアルゴリズムがあります。すべての

スケジューラに共通する点は、実行可能状態のタスク（READY）と何らかの理由（遅延、メールボックス待ち、セマフォ待ち、イベント待ちなど）で中断されているタスクとを区別することです。スケジューラは、READY 状態のタスクのうち一つを選択し、それをアクティブ化（タスクのプログラムを実行）します。現在実行しているタスクは、running task と呼ばれます。スケジューラ間の主な違いは、READY 状態のタスクに計算時間を分配する方法です。

2.4.1 ラウンドロビン･スケジューリング

ラウンドロビン･スケジューリングを使用すると、スケジューラはタスクのリストを持ち、実行中のタスクを非アクティブにする場合は READY 状態にある次のタスクをアクティブにします。ラウンドロビンは、プリエンプティブマルチタスクまたは協調マルチタスクのいずれかで使用することができ、応答時間を保証する必要がない場合にうまく機能します。ラウンドロビンスケジューリングは以下の図で説明できます:

すべてのタスクが同じレベルにあり、CPU の所持は事前に定義された実行時間の後に定期的に変更されます。この実行時間はタイムスライスと呼ばれ、各タスクに個別に定義することができます。

2.4.2 優先制御スケジューリング

現実のアプリケーションでは、タスクが異なれば必要な応答時間も異なります。例えば、モータ、キーボード、ディスプレイを制御するアプリケーションでは、モータは通常キーボード やディスプレイよりも速い応答時間を必要とします。ディスプレイが更新されている間、モータを制御する必要があります。よってプリエンプティブマルチタスクが必要となります。ラウンドロビンがうまく機能する可能性はありますが特定の応答時間を保証することはできないため、改良されたアルゴリズムを使用する必要があります。優先制御スケジューリングでは、各タスクに優先度が割り当てられており、実行順序はこの優先順位によって決まります。ルールは非常に簡単です。

メモ：スケジューラは、READY 状態のすべてのタスクのうち優先順位が最も高いタスクをアクティブにする。

よって、実行中のタスクよりも高い優先度を持つタスクを常に準備することができ、すぐに実行タスクになります。ただし、スケジューラは、危険領域と呼ばれるタスクの切り替えが禁止されているプログラムのセクションにおいてはオフに切り替わることがあります。

embOS は、同一優先順位のタスク間のラウンドロビンに優先制御スケジューリングのアルゴリズムを使用しています。あるタスクが他のタスクよりも重要であるかどうかを考える必要が

ないため、ラウンドロビンスケジューリングは便利な機能です。同一の優先度を持つタスクがタイムスライスよりも長くお互いをブロックすることはありませんが、2 つ以上の同一優先順位のタスクが準備できていてそれらよりも高い優先度のタスクの準備ができていない場合、常に同一な優先度のタスクが切り替わるため、ラウンドロビン方式のスケジューリングでも時間がかかります。各タスクに異なる優先順位を割り当てれば、不要なタスク切り替えが回避されるためより効率的になります。

2.4.3 優先順位の逆転

スケジューラの決まるルールは次のとおりです。
READY 状態のすべてのタスクのうち優先順位が最も高いタスクをアクティブにします。
しかし、最も優先順位の高いタスクが優先順位の低いタスクの所有しているリソースを待機していてブロックされている場合はどうなるのでしょうか。上記のルールによれば、低優先度のタスクが再度実行されリソースを解放するまで待つことになります。他にもルールがあります（例外のない規則はありません）。このような状況を避けるために、最も優先順位の高いタスクをブロックしている低優先度のタスクは、リソースを解放するまで最も高い優先順位を割り当てられます。これは、優先順位の逆転として知られています。

優先順位の低いタスクが OS_Use()でセマフォを要求しています。割り込みは優先順位の高いタスクを起動し、さらに OS_Use()を呼び出します。 OS_Delay()で中間優先度のタスクが中断され、優先順位の低いタスクが OS_Unuse()を呼び出します。低優先度のタスクがセマフォを解放した後、高優先度のタスクはセマフォを要求することができます。

優先順位の高いタスクがセマフォを要求しようとする場合、優先順位の逆転が起こると、優先度の低いタスクは中間優先度のタスクの代わりにアクティブ化されます。

2.5 タスク間の情報伝達

マルチタスク（マルチスレッド）プログラムでは、複数のタスクおよび ISR は、全く別に動作します。すべて同じアプリケーションの中で動作するため、時々お互いに情報を交換する必要があります。

2.5.1 定期的なポーリング

最も簡単な方法は、グローバル変数を使用することです。特定の状況では、タスク同士がグローバル変数を介して通信することも有効ですが、ほとんどの場合この方法には様々な欠点があ

ります。
たとえば、グローバル変数の値が変更されるときに開始するタスクを同期したい場合、貴重な計算時間や電力を費やしてその変数をポーリングする必要があり、しかも応答時間はポーリングの頻度に依存します。

2.5.2 通信メカニズム

複数のタスクが互いを操作する場合、以下のことをする必要があります。
·データの交換
·タスクの同期、または
·リソースが一度に一つのタスクで使用されていることの確認
以上の目的のために、embOS はメールボックス、キュー、セマフォ、イベントを提供しています。

2.5.3 メールボックスおよびキュー

メールボックスは、RTOS によって管理されるデータ·バッファであり、タスクにメッセージを送信するために使用されます。複数のタスクや割り込みが同時にアクセスしようとしている場合でも、衝突することなく動作します。 embOS は、メールボックスが新しいデータを受信するとで新しいメッセージを待っている任意のタスクを自動で活性化し、必要な場合は自動的にそのタスクに切り替えます。
キューは、同様の方法で動作しますが、メールボックスよりサイズの大きなメッセージを処理し、メッセージはそれぞれ個々のサイズを有します。
詳細については、147 ページのキューの章、および 127 ページのメールボックスの章を参照してください。

2.5.4 セマフォ

セマフォは、タスクの同期とリソースの管理のために 2 種類が使用されます。カウンティングセマフォも使用されてはいますが、最も一般的なのはリソースセマフォです。詳細およびサンプルについては、99 ページのリソースセマフォの章と 113 ページのセマフォカウントの章を参照してください。サンプルは当社のウェブサイト（www.segger.com）当社のウェブサイトでも閲覧できます。

2.5.5 イベント

タスクは、計算時間なしで特定のイベントを待つこともできます。この考え方は説得力があり

かつ簡単です。イベントを待っているタスクを単純に起動するのならば、ポーリングする意味はありません。計算パワーを大量に節約し、タスクが遅滞なくイベントに応答できるようになります。イベントの標準的なアプリケーションは、データ、キー、コマンドや文字の受信、または外部のリアルタイムクロックのパルスを待っています。詳細については、161 ページのタスクの章および 171 ページのタスク･イベントの章を参照してください。

2.6 タスク切り替えの動作

リアルタイムマルチタスクシステムでは、単一の CPU 上で複数のシングルタスクのプログラムのようなタスクを半同時に動かすことができます。タスクは、マルチタスキングの世界で次の 3 つの部分から構成されます

･通常 ROM に常駐する･プログラムコード（必要条件ではない）

･スタック･ポインタでアクセスできる RAM 領域にあるスタック

･RAM にあるタスク制御ブロック

スタックは、シングルタスクシステムと同じ機能を持っています。（関数呼び出し・パラメータ・ローカル変数の戻りアドレスの保存、および中間の計算結果およびレジスタ値の一時的な格納）各タスクは別のスタック･サイズを持つことができます。より詳しい情報は 205 ページのスタックの章をご参照ください。

タスク制御ブロック（TCB）は、タスクが作成されたときにそれに割り当てられたデータ構造です。これは、スタックポインタやタスクの優先度、現在のタスクのステータス（レディ、待機中、中断の理由）およびその他の管理データを含むタスクのステータス情報を持っています。スタックポインタがわかるとほかのレジスタにアクセスすることができ、他のレジスタはタスクが作られた時や中断された時にスタックに押し付けられます。この情報によって割り込まれたタスクがちょうど中断した場所から実行を継続することができます。TCB は RTOS のみからアクセスされます。

2.6.1 スイッチングスタック

次の図は、あるスタックから別のスタックに切り替えるプロセスを示しています。

スケジューラは、プロセッサレジスタをタスクに保存することにより、中断（タスク 0）するタスクを非アクティブにします。その後、タスク n のスタックに格納された値からスタックポ

インタ（SP）及びプロセッサレジスタをロードすることによって優先度の高いタスク（タスク n）をアクティブにします。

タスクの非アクティブ化

スケジューラは次のようにして中断（タスク 0）するタスクを非アクティブにします:

1. プロセッサレジスタをタスクのスタックに保存（プッシュ）します
2. スタックポインタをタスクコントロールブロック（TCB）に保存します。

タスクのアクティブ化

そして、逆の順序で実行することにより、優先度の高いタスク（タスク n）をアクティブにします。:

1. TCB からスタックポインタ（SP）をロード（ポップ）します。
2. タスク n のスタックに格納された値からプロセッサのレジスタをロードします。

2.7 タスクスタックの変更

タスクは、任意の時点で、いくつかの状態のうちいずれかをとっています。タスクが作成されると、自動的に READY 状態（TS_READY）に置かれます。

READY 状態のタスクは、それよりも高い優先順位を持つ他のタスクが存在しないため、可能な限り早くアクティブになります。一度に実行されるタスクはひとつだと思われます。優先度の高いタスクが READY 状態になった場合、このタスクが起動され、プリエンプトされたタスクは READY 状態のままになります。

実行中のタスクは、指定された期間または指定された時間まで遅れることがあり、この場合は DELAY 状態（TS_DELAY）に入れられ、READY 状態にある次に優先度の高いタスクがアクティブ化されます。

実行中のタスクはイベント（またはセマフォ、メールボックス、キュー）を待たなければならない場合があります。イベントがまだ発生していない場合、タスクは待ち状態に置かれ、READY 状態で次に高い優先度のタスクがアクティブ化されます。

存在しないタスクは embOS では使用できず、まだ作成されていないか終了しているかのどちらかです。

次の図は、起こり得るすべてのタスクの状態とそれらの間で起こりうる遷移を示しています。

2.8 How the OS gains control

CPU がリセットされると、特殊機能レジスタはそれぞれの値に設定されます。リセット後、

プログラムの実行が開始されます。 PC レジスタは、スタートベクトルまたは（CPU に依存する）開始アドレスによって定義された開始アドレスに設定されます。この開始アドレスは、通常 C コンパイラに付属するスタートアップモジュールにあり、標準ライブラリの一部ですであることもあります。

スタートアップコードは次のように動作します:

· ほとんどの CPU の定義されたスタックセグメントの終了となるデフォルトの値のスタックポインタをロードする

· すべてのデータセグメントをそれぞれの値に初期化する

· main()ルーチンを呼び出す

Main()のルーチンは、C のスタートアップ直後に制御するプログラムの一部です。embOS は通常、設定変更なしに標準の C スタートアップモジュールと連携して動作します。必要な変更がある場合、それらは embOS の CPU およびコンパイラ細目書（CPU & Compiler Specifics manual）に記載されています。

embOS に加え、main()ルーチンは、アプリケーションプログラムの一部です。Main()は基本的には一つ以上のタスクを作成し、次に OS_Start()を呼び出すことによりマルチタスクを開始します。スケジューラはその時点から実行されるタスクを制御します。

Startup code
main()
OS_IncDI()
OS_InitKern()
OS_InitHW()
Additional initialization code;
creating at least one task.
OS_Start()

これらのタスクのみ OS_Start()への呼び出しの後に実行されるため、main（）のルーチンは、作成されたいずれのタスクにも割り込まれることはありません。したがって、通常、メールボックスやセマフォなどの制御構造だけでなくすべてあるいはほとんどのタスクをここに作成することが推奨されます。（ある程度まで）再利用可能なモジュールの形でソフトウェアを書くことがよい習慣です。これらのモジュールには通常、必要なタスク及び/または制御構造を作成する初期化ルーチンがあります。

典型的な main()は次のような例に似ています：

例

```
/*********************************************************************
*
*          main
*
*********************************************************************
*/
void main(void) {
OS_IncDI();
OS_InitKern();      /* Initialize OS (should be first !)      */
OS_InitHW();        /* Initialize Hardware for OS (in RtosInit.c) */
/* Call Init routines of all program modules which in turn will create
the tasks they need ...        (Order of creation may be important) */
MODULE1_Init();
MODULE2_Init();
MODULE3_Init();
MODULE4_Init();
MODULE5_Init();
OS_Start();         /* Start multitasking */
}
```

OS_Start()の呼び出しに加え，スケジューラは main()が作成した最も優先度の高いタスクをを開始します。 OS_Start（）が起動プロセス中に一度だけ呼び出されると戻りませんので注意してください。

2.9 embOS の様々なビルド

embOS は様々なビルドあるいはライブラリのバージョンに入ります。ビルドのが様々である理由は、開発期間中に要求が変化するためです。ソフトウェアを開発する間のパフォーマンス（およびリソース使用量）は通常、製品のリリースバージョンとなる最終版におけるほど重要ではありません。しかし開発中は、たとえ小規模なプログラミングエラーであってもアサーションの使用によってキャッチされる必要があります。これらのアサーションは embOS ライブラリのデバッグバージョンにコンパイルされ、リリースバージョンあるいは最終的な製品に使用されるスタックチェックバージョンよりもコードが少し大きく（約 50%）かつ若干遅くなり

ます。
この概念には 2 つの最高の利点があります：最終製品（ライブラリのリリースバージョンまたはスタックチェックバージョン）のコンパクトで非常に効率的なビルド、および最終的な製品の構築、および開発中ほとんどの一般的なアプリケーションプログラミングエラーをキャッチする安全（ただし大きく遅くなるが）なバージョンです。もちろん、開発中に embOS のリリースバージョンを使用することもできるが、このようなエラーをキャッチすることはありません。

2.9.1 プロファイリング

embOS は、プロファイリングビルドにおいてプロファイリングをサポートします。プロファイリングによって個々のタスクの実行時間に関する正確な情報が利用できます。プロファイリング· ライブラリを使用すると、大きなタスク制御ブロックや追加の ROM（約 200 バイト）のようなオーバーヘッドや、追加の実行時のオーバーヘッドをもたらします。このオーバーヘッドは通常許容できますが、この機能を使用しない場合は最適なパフォーマンスを維持するために embOS の非プロファイリングビルドを使用することも可能です。

2.9.2 ライブラリのリスト

アプリケーション· プログラムでは、どのエンボスのビルドを使用しているかをコンパイラに知らせる必要があります。これは、RTOS.h.を取り込む前に単一の識別子を定義することによって行われます。

Table 2.1: List of libraries

2.9.3 embOS 関数コンテキスト

すべての embOS 関数がアプリケーション内のどの場所からでも呼び出せるわけではありません。Main（OS_Start()の呼び出しの前に）、タスク、ISR およびソフトウェアタイマ間で異なる必要があります。
例えば、embOS 関数が ISR から呼び出されることを許可されているかを確認するために、embOS API のテーブルを確認してください。 embOS のデバッグビルドは、あなたがこれらのルールを破っていないことを自動的に確認するのに役立ちます。

Chapter 3　embOS の動作

この章では、アプリケーションで embOS を使用する方法についての推奨事項を提供します。我々がアプリケーションの設計及び構築の際に純粋に役立つと感じていることが推奨されています。

3.1　　一般的なアドバイス

-可能な限り RR を避ける
-動的なタスクの作成/終了を避ける
-可能な限り割り込みのネスティングを避ける

3.1.1　タイマあるいはタスク

定期ジョブの場合は、タスクあるいはソフトウェアタイマのいずれかを使用できます。embOS ソフトウェアタイマは、C スタック上で実行されるため自身のタスクのスタックを必要としないという利点があります。

Chapter 4　タスク

この章では、タスクおよび embOS のタスク API 関数に関する基本事項を説明しています。比較的読みやすいはずなので、他の章に移る前にお読みになることを推奨します。

4.1 導入

embOS の下で実行するタスクは、タスク制御ブロック（TCB)、スタック、および C 言語で書かれた通常のルーチンです。タスクルーチンには以下の規則が適用されます:

· タスクのルーチンは、パラメータ（void パラメータリスト）を取らず OS_CreateTask（）が使われる場合と、パラメータとしてひとつの void ポインタを取り OS_CreateTaskEx（）が使われる場合があります。
· タスクのルーチンが返すことはできません。
· タスクのルーチンが無限ループとして実装されるか、自身で終了しなければなりません（下記

の例を参照)。

4.1.1 無限ループとしてのタスクルーチンの例

```
/*        Example of a task routine as an endless loop */
void Task1(void) {
while(1) {
DoSomething()      /* Do something    */
OS_Delay(1);       /* Give other tasks a chance */
}
}
```

4.1.2 自身で終了するタスクルーチンの例

```
/*        Example of a task routine that terminates      */
void Task2(void) {
char DoSomeMore;
do {
DoSomeMore = DoSomethingElse() /* Do something  */
OS_Delay(1);       /* Give other tasks a chance */
} while(DoSomeMore);
OS_TerminateTask(0); /* Terminate yourself      */
}
```

タスクを作成する方法はいろいろあります。embOS はこれを簡単にする単純なマクロを提供しており、ほとんどの場合これで十分です。しかし、タスク動的に作成し削除する場合は、ルーチンは、あらゆるパラメータを"微調整"することを許可するのに有効です。ほとんどのアプリケーションでは、少なくとも最初は、サンプルのスタートプロジェクトのように、マクロを使用すれば正常に動作します。

4.2 強調タスクスイッチ対プリエンプティブタスクスイッチ

プリエンプティブタスクスイッチは、一般的に RTOS の重要な機能です。プリエンプティブタスクスイッチは、優先順位が高くタイムクリティカルなタスクの応答性を確実にすることが求められます。しかし、特定の状況では特定のタスクのプリエンプティブタスクスイッチを無効

にすることが望ましいです。 embOS のデフォルトの動作は、プリエンプティブタスクスイッチを常に可能にすることです。

4.2.1 Disabling preemptive task switches for tasks at samepriorities

同一の優先順位で実行されているタスク間のプリエンプティブタスクスイッチは、望ましくない場合もあります。これを解決するために、同一の優先順位で実行しているタスクのタイムスライスは、下記の例のように 0 に設定する必要があります：

```
#include "RTOS.h"
#define PRIO_COOP
#define TIME_SLICE_NULL 0
OS_STACKPTR int StackHP[128], StackLP[128]; /* Task stacks */
OS_TASK TCBHP, TCBLP;  /* Task-control-blocks */
/*********************************************************************/
static void TaskEx(void * pData) {
while (1) {
OS_Delay ((OS_TIME) pData);
}
}
/*********************************************************************
*
*          main
*
*********************************************************************/
int main(void) {
OS_IncDI();          /* Initially disable interrupts */
OS_InitKern();       /* initialize OS      */
OS_InitHW();         /* initialize Hardware for OS          */
/* You need to create at least one task before calling OS_Start() */
OS_CreateTaskEx(&TCBHP, "HP Task", PRIO_COOP, TaskEx, StackHP,
sizeof(StackHP), TIME_SLICE_NULL, (void*) 50);
OS_CreateTaskEx(&TCBLP, "LP Task", PRIO_COOP, TaskEx, StackLP,
sizeof(StackLP), TIME_SLICE_NULL, (void*) 200);
```

```
OS_Start();          /* Start multitasking          */
return 0;
}
```

4.2.2 Completely disabling preemptions for a task

これは簡単です。次の例に示すように、コードの最初の行は OS_EnterRegion()でなければなりません。

```
void MyTask(void *pContext) {
OS_EnterRegion();  // Disable preemptive context switches
while (1) {
// Do something. In the code, make sure that you call a blocking funtion
// periodically to give other tasks a chance to run
}
}
```

メモ:これにより、特定のタスクからのプリエンプティブコンテキストスイッチを完全に禁止し、それゆえ高優先度のタスクのタイミングに影響を与えることになります。あなたが何をやっているか理解している場合にのみ行ってください。

4.3 API 関数

Table 4.1: Task routine API list

- 4.3.1 OS_CREATETASK()

説明

タスクを作成します。

プロトタイプ

```
void      OS_CREATETASK ( OS_TASK *     pTask,
char *    pName,
void *    pRoutine,
unsigned  char Priority,
void *    pStack);
```

Table 4.2: OS_CREATETASK() parameter list

補足情報

OS_CREATETASK（）は OS のライブラリ関数を呼び出すマクロです。タスクを作成し、それを READY 状態に置くことによって実行可能な状態にします。新しく作成されたタスクは、他により優先度の高いタスクが存在しなければ、すぐにスケジューラによって起動されます。同じ優先順位を持つ別のタスクがある場合は、新しいタスクが先に配置されます。このマクロは使用するパラメータがより少なく使用が簡単であるため、タスク作成のために OS_CreateTask（）の関数呼び出しの代わりに使用されます。

OS_CREATETASK（）は初期化中の main()または他のタスクのいずれかから、いつでも呼び出すことができます。タスクの構造を簡単に理解できるようにしておくため、初期化中にすべてのタスクを main()に作成することが推奨されます。

優先度（Priority ）の絶対値は、他のタスクとの優先順位と比較値にすぎないので重要ではありません。

OS_CREATETASK（）は、sizeof（）を使用して自動的にスタックのサイズを決定します。これはメモリ領域がコンパイル時に定義されている場合にのみ可能です。

重要

ユーザーが定義したスタックは、CPU が実際にスタックとして使用できる領域に存在する必要があります。ほとんどの CPU は、メモリ領域全体をスタックとして使用することはできません。ほとんどの CPU は、バイトの倍数でスタックのアライメントを必要とします。タスクスタックが整数の配列として定義されている場合、これは自動的に行われます。

タスクのスタックは 1 つのタスクのみに割り当てられている必要があります。タスクスタックとして使用されるメモリは、タスクが存在する限り他の目的に使用することはできません。スタックは、他のタスクと共有できません。

使用例

```
OS_STACKPTR int UserStack[150]; /* Stack-space */
OS_TASK UserTCB;              /* Task-control-blocks */
void UserTask(void) {
while (1) {
```

```
Delay（100);
}
}
void InitTask(void) {
OS_CREATETASK(&UserTCB, "UserTask", UserTask, 100, UserStack);
}
```

- 4.3.2 OS_CreateTask()

説明

タスクを作成します。

プロトタイプ

```
void OS_CreateTask （ OS_TASK * pTask,
char * pName,
unsigned char Priority,
voidRoutine * pRoutine,
void * pStack,
unsigned StackSize,
unsigned char TimeSlice );
```

Table 4.3: OS_CreateTask() parameter list

補足情報

この関数は、タスクのすべてのパラメータを指定できることを除いて OS_CREATETASK()と同様の働きをします。

マクロと違ってスタックサイズが自動的に計算されないため、タスクは動的に作成できます。

embOS のデバッグビルドを使用する場合は、違法タイムスライス値が設定されるとエラーコード OE_ERR_TIMSLICE とともにエラーハンドラ OS_Error()が呼び出されます。

重要

ユーザーが定義したスタックは、CPU が実際にスタックとして使用できる領域に存在する必要

があります。ほとんどの CPU は、メモリ領域全体をスタックとして使用することはできません。ほとんどの CPU は、バイトの倍数でスタックのアライメントを必要とします。タスクスタックは整数の配列として定義されている場合、これは自動的に、行われます。タスクのスタックは一つのタスクのみに割り当てられる必要があります。タスクスタックとして使用されるメモリは、タスクが存在する限り、他の目的に使用することはできません。スタックは、他のタスクと共有できません。

使用例

```
/* Demo-program to illustrate the use of OS_CreateTask */
OS_STACKPTR int StackMain[100], StackClock[50];
OS_TASK TaskMain,TaskClock;
OS_SEMA SemaLCD;
void Clock(void) {
while(1) {
/* Code to update the clock */
}
}
void Main(void) {
while (1) {
/* Your code          */
}
}
void InitTask(void) {
OS_CreateTask(&TaskMain,  NULL, 50, Main,  StackMain, sizeof(StackMain), 2);
OS_CreateTask(&TaskClock, NULL, 100, Clock,StackClock,sizeof(StackClock),2);
}
```

- 4.3.3 OS_CREATETASK_EX()

説明

タスクを作成しそれにパラメータを渡します。

プロトタイプ

```
void        OS_CREATETASK_EX  (  OS_TASK  * pTask,
char  *     pName,
void  *     pRoutine,
unsigned  char       Priority,
void  *     pStack,
void  *     pContext  );
```

Table 4.4: OS_CREATETASK_EX() parameter list

補足情報

OS_CREATETASK_EX() は embOS ライブラリ関数を呼び出すマクロです。OS_CREATETASK()と同様に動作しますが、タスクにパラメータを渡すことができます。追加パラメータとして void ポインタを使用すると、タスク関数にどのような種類のデータでも渡すことができます。

使用例

下記の例は、embOS の Samples フォルダに配信されます。

```
/*---------------------------------------------------------------
File        : Main_TaskEx.c
Purpose : Sample program for embOS using OC_CREATETASK_EX
--------- END-OF-HEADER -------------------------------------------*/
#include "RTOS.h"
OS_STACKPTR int StackHP[128], StackLP[128]; /* Task stacks */
OS_TASK TCBHP, TCBLP;  /* Task-control-blocks */
/*********************************************************************/
static void TaskEx(void* pData) {
while (1) {
OS_Delay ((OS_TIME) pData);
}
}
/*********************************************************************
```

```
*
*          main
*
*********************************************************************/
int main(void) {
OS_IncDI();          /* Initially disable interrupts */
OS_InitKern();       /* initialize OS      */
OS_InitHW();         /* initialize Hardware for OS          */
/* You need to create at least one task before calling OS_Start() */
OS_CREATETASK_EX(&TCBHP, "HP Task", TaskEx, 100, StackHP, (void*) 50);
OS_CREATETASK_EX(&TCBLP, "LP Task", TaskEx,        50, StackLP, (void*) 200);
OS_SendString("Start project will start multitasking !\n");
OS_Start();          /* Start multitasking          */
return 0;
}
```

- 4.3.4 OS_CreateTaskEx()

説明

タスクを作成し、タスクにパラメータを渡します。

プロトタイプ

```
void  OS_CreateTaskEx  (  OS_TASK  *            pTask,
char  *    pName,
unsigned  char       Priority,
voidRoutine  *       pRoutine,
void  *    pStack,
unsigned  StackSize,
unsigned  char       TimeSlice,
void  *    pContext  );
```

Table 4.5: OS_Create_TaskEx() parameter list

補足情報

この関数は、パラメータがタスク関数に渡されていることを除いて、OS_CreateTask()と同様の働きをします。

タスクに渡すパラメータの例は OS_CREATETASK_EX()の下に表示されます。embOS のデバッグビルドを使用する場合、違法タイムスライス値の設定により、エラーコード OE_ERR_TIMSLICE とともにエラーハンドラ OS_Error()が呼び出されます。

重要

ユーザーが定義したスタックは、CPU が実際にスタックとして使用できる領域に存在する必要があります。ほとんどの CPU は、メモリ領域全体スタックとしてを使用することはできません。ほとんどの CPU は、バイトの倍数でスタックのアライメントを必要とします。タスクスタックが整数の配列として定義されている場合、これは自動的に行われます。タスクスタックは一つのタスクのみに割り当てられている必要があります。タスクスタックとして使用されるメモリは、タスクが存在する限り他の目的に使用することはできません。スタックは、他のタスクと共有できません。

- 4.3.5 OS_Delay()

説明

指定された期間、呼び出し元のタスクを中断します。

プロトタイプ

```
void  OS_Delay  (OS_TIME  ms);
```

Table 4.6: OS_Delay() parameter list

補足情報

呼び出し元のタスクは、指定された期間 TS_DELAY 状態に置かれます。指定した時間が経過するまでタスクは遅延状態になります。パラメータ ms は、正確な間隔を指定し、その間にタスクが決められた基本的な時間間隔（通常 1/1000 秒）で中断する必要があります。実際の遅

延（基本的な時間間隔）は、スケジューラへの割り込みが発生するタイミングに応じて ms - 1 <= delay <= ms の範囲になります。
遅延の終了後、タスクはスケジューラの規則に従って再び待ち状態にされ活性化されます。別のタスクあるいは割り込みハンドラ呼び出し OS_WakeTask()によって遅延が早まって終了することもあります。

使用例

```
void Hello() {
printf("Hello");
printf("The next output will occur in 5 seconds");
OS_Delay (5000);
printf("Delay is over");
}
```

- 4.3.6 OS_DelayUntil()

説明
指定された期間、呼び出し元のタスクを中断します。

プロトタイプ

```
void OS_DelayUntil (OS_TIME t);
```

Table 4.7: OS_DelayUntil() parameter list

補足情報
呼び出し元のタスクは、指定された時間まで TS_DELAY 状態に置かれます。
OS_DelayUntil（）関数は、時間変数 OS_Time が一定の値に達するまで遅延します。あなたは遅れを蓄積しないようにする場合には非常に有効です。

使用例

```
int sec,min;
void TaskShowTime() {
int t0;
```

```
t0 = OS_GetTime();
while (1) {
ShowTime();          /* Routine to display time */
t0 += 1000;
OS_DelayUntil (t0);
if (sec < 59) {
sec++;
} else {
sec=0;
min++;
}
}
}
```

上記の例では、OS_Delay()の使用が遅延の蓄積につながる可能性があり、単なる"クロック"を遅くすることがあります。

- 4.3.7 OS_Delayus()

説明

マイクロ秒単位で指定された時間待機します。

プロトタイプ

```
void OS_Delayus (OS_U16 us);
```

Table 4.8: OS_Delay() parameter list

補足情報

この関数は、短い遅延に使用することができます。

使用例

```
void Hello() {
printf("Hello");
```

printf("The next output will occur in 500 microseconds");
OS_Delayus（500);
printf("Delay is over");
}

- 4.3.8 OS_ExtendTaskContext()

説明

この関数は、様々な目的で使用することができます。代表的なアプリケーションには含まれるが、これらに限定されるわけではありません:

· C-lib の関数をスレッドセーフにする、C ライブラリにおける"errno"などのグローバル変数。

· デフォルトごとにタスクコンテキストに保存されていない場合、MAC / EMAC（積和演算単位）などの追加・オプションの CPU/レジスタ

· VFP のレジスタ（浮動小数点コプロセッサ）などのコプロセッサ

· また、CRC 計算ユニットなどの追加ハードウェアユニットのデータレジスタ

これにより、システムの要求に応じてユーザーがタスクのコンテキストを拡張することができます。主な利点は、タスクの拡張子はタスク固有であることです。これは、追加情報（例えば浮動小数点レジスタなど）が実際にそのレジスタを使用するタスクに保存される必要があることを意味します。他のタスクの切り替え時間が影響されないという利点があります。必要なスタック領域についても同様のことが言えます。追加スタック領域は、実際に追加レジスタを保存するタスクにのみ必要とされます。

プロトタイプ

void OS_ExtendTaskContext(const OS_EXTEND_TASK_CONTEXT * pExtendContext);

Table 4.9: OS_ExtendTaskContext() parameter list

補足情報

OS_EXTEND_TASK_CONTEXT 構造は次のように定義されます:

typedef struct OS_EXTEND_TASK_CONTEXT {
void (*pfSave) (void * pStack);
void (*pfRestore)(const void * pStack);

} OS_EXTEND_TASK_CONTEXT;

保存及び復元関数は構造の中で使用されている関数の型に従って宣言される必要があります。以下のサンプルでは、タスクスタックが拡張されたタスクコンテキストを保存および復元する方法を示しています。

OS_ExtendTaskContext()は XR ライブラリでは利用できません。

使用例

下記の例は embOS のサンプルフォルダに配置されています。

```
/*---------------------------------------------------------------------
File : ExtendTaskContext.c
Purpose : Sample program for embOS demonstrating how to dynamically
extend the task context.
This example adds a global variable to the task context of
certain tasks.
-------- END-OF-HEADER ---------------------------------------------
*/
#include "RTOS.h"
OS_STACKPTR int StackHP[128], StackLP[128]; /* Task stacks */
OS_TASK TCBHP, TCBLP;  /* Task-control-blocks */
int GlobalVar;
/*********************************************************************
*
*          _Restore
*          _Save
*
*          Function description
*          This function pair saves and restores an extended task context.
*          In this case, the extended task context consists of just a single
*          member, which is a global variable.
*/
typedef struct {
```

```
int GlobalVar;
} CONTEXT_EXTENSION;
static void _Save(void * pStack) {
CONTEXT_EXTENSION * p;
p = ((CONTEXT_EXTENSION*)pStack) - (1 - OS_STACK_AT_BOTTOM); // Create pointer
//
// Save all members of the structure
//
p->GlobalVar = GlobalVar;
}
static void _Restore(const void * pStack) {
CONTEXT_EXTENSION * p;
p = ((CONTEXT_EXTENSION*)pStack) - (1 - OS_STACK_AT_BOTTOM); // Create pointer
//
// Restore all members of the structure
//
GlobalVar = p->GlobalVar;
}
/*********************************************************************
*
*       Global variable which holds the function pointers
*       to save and restore the task context.
*/
const OS_EXTEND_TASK_CONTEXT _SaveRestore = {
_Save,
_Restore
};
/*********************************************************************/
/*********************************************************************
*
*       HPTask
```

```
*
*          Function description
*          During the execution of this function, the thread-specific
*          global variable has always the same value of 1.
*/
static void HPTask(void) {
OS_ExtendTaskContext(&_SaveRestore);
GlobalVar = 1;
while (1) {
OS_Delay (10);
}
}
/*********************************************************************
*
*          LPTask
*
*          Function description
*          During the execution of this function, the thread-specific
*          global variable has always the same value of 2.
*/
static void LPTask(void) {
OS_ExtendTaskContext(&_SaveRestore);
GlobalVar = 2;
while (1) {
OS_Delay (50);
}
}
/*********************************************************************
*
*          main
*/
int main(void) {
OS_IncDI();          /* Initially disable interrupts */
```

```
OS_InitKern();       /* initialize OS      */
OS_InitHW();         /* initialize Hardware for OS          */
/* You need to create at least one task here !  */
OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
OS_CREATETASK(&TCBLP, "LP Task", LPTask, 50, StackLP);
OS_Start();          /* Start multitasking          */
return 0;
}
```

- 4.3.9 OS_GetpCurrentTask()

説明

現在実行中のタスクのタスク制御ブロック構造にポインタを返します。

プロトタイプ

```
OS_TASK*  OS_GetpCurrentTask (void);
```

戻り値

OS_TASK*: タスク制御ブロック構造へのポインタ。

補足情報

この関数は、実行されているタスクを決定するために使用することができます。任意の関数の応答が現在実行中のタスクに依存する場合に有効です。

- 4.3.10 OS_GetPriority()

説明

指定されタスクの優先度を返します。

プロトタイプ

```
unsigned char  OS_GetPriority (OS_TASK*  pTask);
```

Table 4.10: OS_GetPriority() parameter list

戻り値

"符号なし文字"（1～255 の範囲）として指定されたタスクの優先順位。

補足情報

pTask が NULL ポインターである場合、この関数は、現在実行中のタスクの優先度を返します。 pTask が有効なタスクを指定しない場合、embOS のデバッグバージョンは OS_Error() を呼び出します。 embOS のリリースバージョンは pTask の妥当性をチェックすることはできないため、pTask が有効なタスクを指定していない場合は無効な値を返すことがあります。

重要

この関数を割り込みハンドラの中から呼び出すことはできません。

- 4.3.11 OS_GetSuspendCnt()

説明

この関数は、サスペンションカウントとそれにより指定されたタスクの中断状態を返します。この関数は、タスクが以前の OS_Suspend() の呼び出しによって中断されているかどうかを調べるために使用することができます。

プロトタイプ

unsigned char OS_GetSuspendCnt (OS_TASK* pTask);

Table 4.11: OS_GetSuspendCnt() parameter list

戻り値

符号なし文字の値として指定されたタスクのサスペンドカウント。

0：タスクが中断されていません。

> 0：タスクが OS_Suspend() の少なくとも 1 つの呼び出しによって中断されています。

補足情報

pTask が有効なタスクを指定しない場合は、embOS のデバッグバージョンは OS_Error() を呼び出します。

embOS のリリースバージョンは pTask の妥当性をチェックすることはできないため、pTask が有効なタスクを指定していない場合は無効な値を返すことがあります。タスクが動的に作成され終了している場合、タスクが有効かどうか調べるために OS_IsTask() が OS_GetSuspendCnt()より先に呼ばれることがあります。返される値は、OS_Resume()を呼び出すことによって中断されたタスクを再開するために使用することができます。

使用例

```
/* Demo-function to illustrate the use of OS_GetSuspendCnt() */
void ResumeTask(OS_TASK* pTask) {
unsigned char SuspendCnt;
SuspendCnt = OS_GetSuspendCnt(pTask);
while(SuspendCnt > 0) {
OS_Resume(pTask); /* May cause a task switch */
SuspendCnt--;
}
}
```

- 4.3.12 OS_GetTaskID()

説明

現在実行中のタスクのタスク制御ブロック構造体にポインタを返します。このポインタは、タスク固有のものであり、タスク ID として使用されます。

プロトタイプ

```
OS_TASK * OS_GetTaskID ( void );
```

戻り値

タスク制御ブロックへのポインタ。 0（NULL）の値は、タスクが実行されていないことを示します。

補足情報

この関数は、どのタスクが実行されているかを判断するために使用します。いずれかの関数の応答反応が現在実行中のタスクに依存する場合に役に立つでしょう。

▪ 4.3.13 OS_IsRunning()

説明

embOS スケジューラが OS_Start()の呼び出しによって開始されたかどうかを判断します。

プロトタイプ

unsigned char OS_IsRunning (void);

戻り値

文字値：

0：スケジューラが開始されませんでした。

！= 0：スケジューラが実行され、OS_Start()が呼び出されました。

補足情報

この関数は、main()あるいは実行中のタスクによって呼び出されうる関数に役立つかもしれません。

スケジューラが開始されておらず、関数が main()から呼びだされていれば、ブロッキングタスクスイッチは許可されません。

タスクまたは main()から呼び出されうる関数は、ブロッキングタスクスイッチが許可されているかどうかを判断するために OS_IsRunning()を使用することができます。

▪ 4.3.14 OS_IsTask()

説明

タスク制御ブロックが実際に妥当なタスクに属しているか判断します。

プロトタイプ

char OS_IsTask (OS_TASK* pTask);

Table 4.12: OS_IsTask() parameter list

戻り値

文字値：

0：TCB は任意のタスクで使用されません。
1：TCB はタスクによって使用されます。

補足情報
この関数は、指定されたタスクが内部タスクリストに残っているかどうかを確認します。タスクが終了した場合は内部タスクリストから削除されます。この関数は、タスクのタスクコントロールブロックとタスクのスタックが動的にタスクを作成したり終了したりするアプリケーション内の別のタスクに再利用できるかどうかを判断するのに役に立つかもしれません。

- 4.3.15 OS_Resume()

説明
サスペンドカウントがゼロになった場合、指定されたタスクのサスペンドカウントを減少させ再開させます。

プロトタイプ
void OS_Resume (OS_TASK* pTask);

Table 4.13: OS_Resume() parameter list

補足情報
指定されたタスクのサスペンドカウントが減少します。結果の値が 0 の場合、指定したタスクの実行が再開されます。
タスクが他のタスクブロッキングメカニズムによってブロックされていない場合、タスクはスケジューラの規則により、READY 状態に戻って設定し実行が継続されます。
embOS のデバッグバージョンでは、OS_Resume()関数は、指定されたタスクのサスペンドカウントを確認します。OS_Resume()が呼び出されたときにサスペンドカウントが 0 である場合、指定されたタスクは現在中断されず OS_Error() によってエラー OS_ERR_RESUME_BEFORE_SUSPEND が呼び出されます。

- 4.3.16 OS_ResumeAllSuspendedTasks()

説明
サスペンドカウントがゼロに達すと、設定されている場合には、全てのタスクのサスペンドカウントを減少させます。

プロトタイプ

void　OS_ResumeAllSuspendedTasks　(void);

補足情報

この関数は、一度に複数のタスクを同期させたり開始したりするのに便利かもしれません。関数は、すべてのタスクを再開し、特定のタスクを指定する必要はありません。この関数は、関数 OS_SuspendAllTasks() と OS_SetInitialSuspendCnt() とともに使用することができます。

この関数は、呼び出し元のタスクよりも高い優先度を持つタスクが再開されたときには、タスク切り替えを起こす可能性があります。

中断したすべてのタスクが再開された後にタスク切り替えが実行されます。

この関数は非ブロッキング関数なので、すべてのコンテキスト、メイン、ISR またはタイマから呼び出される可能性があります。

この関数はどのタスクが中断されたかに関わらず呼び出される可能性があります。タスクが中断されていないにはエラーは発生しません。

- 4.3.17　OS_SetInitialSuspendCnt()

説明

新しく作成されたタスクの初期サスペンドカウントを設定します。最初に中断されたタスクを作成するために用いることができます。

プロトタイプ

void　OS_SetInitialSuspendCnt　(unsigned　char　SuspendCnt);

Table 4.14: OS_SetInitialSuspendCnt() parameter list

補足情報

OS_SetInitialSuspendCnt()は　main()、任意のタスク、ISR またはソフトウェアタイマからいつでも呼び出すことができます。

この関数をサスペンドカウント 0 で呼び出した後は、新たに作成されたすべてのタスクは自動的に中断されます。よって、この関数はタスク切り換えを禁止するために使用できます。これ

は、システムを初期化するときに役立つかもしれません。

重要

この関数が中断状態のすべてのタスクを初期化する為に main()関数から呼び出された場合、S_Start()の呼び出しによるシステムの起動前に少なくとも一つのタスクが再開されなければなりません。

初期サスペンドカウントはシステムが起動する前に一般のタスクの作成ができるようにリセットする必要があります。

使用例

```
/* Sample to demonstrate the use of OS_SetInitialSuspendCnt */
void InitTask(void) {
//
// High priority task started first after OS_Start()
//
OS_SuspendAllTasks();          // Ensure, no other existing task can run.
OS_SetInitialSuspendCnt(1);  // Ensure, no newly created task will run.
//
// Perform application initialization
//
...            // New tasks may be created, but can not start.
...            // Even when InitTask() blocks itself by a delay, no other task will run.
OS_SetInitialSuspendCnt(0);  // Reset the initial suspend count for tasks.
//
// Resume all tasks taht were blocked before or were created in suspended state.
//
OS_ResumeAllSuspendedTasks();
while (1) {
...            // Do the normal work
}
}
```

▪ 4.3.18 OS_SetPriority()

説明

特定のタスクに特定の優先順位を割り当てます。

プロトタイプ

```
void OS_SetPriority (OS_TASK* pTask,
unsigned char Priority);
```

Table 4.15: OS_SetPriority() parameter list

補足情報

この関数はあらゆるタスクまたはソフトウェアタイマからいつでも呼び出すことができます。

この関数を呼び出すと、すぐにタスク切り換えが起こるかもしれません。

重要

この関数は割り込みハンドラ内から呼び出すことはできません。

▪ 4.3.19 OS_SetTaskName()

説明

実行中にタスク名前の変更を許可します。

プロトタイプ

```
void OS_SetTaskNamePriority (OS_TASK* pTask,
const char* s);
```

Table 4.16: OS_SetTaskName() parameter list

補足情報

この関数はあらゆるタスクまたはソフトウェアタイマからいつでも呼び出すことができます。

pTask が NULL ポインタである場合、現在実行中のタスクの名前が変更されます。

▪ 4.3.20 OS_SetTimeSlice()

説明

指定されたタスクにタイムスライスの値を割り当てます。

プロトタイプ

unsigned char OS_SetTimeSlice (OS_TASK* pTask,
unsigned char TimeSlice);

Table 4.17: OS_SetTimeSlice() parameter list

戻り値

unsigned char としてタスクの前のタイムスライス値。

補足情報

この関数はどのタスクまたはソフトウェアタイマからでもいつでも呼び出すことができます。
タイムスライス値の設定は、ラウンドロビンモードで実行中のタスクにのみ影響します。これは同一の優先度を持つ別のタスクが存在する必要があることを意味します。
新しいタイムスライス値はリロード値と認識されます。これは、タスクの次回アクティブ化後に使用されます。実行中のタスクの残りのタイムスライスには影響を与えません。
違法タイムスライス値の設定により embOS のデバッグビルドを使用する場合は、エラーコード OE_ERR_TIMSLICE とともにエラーハンドラ OS_Error()を呼び出します。

▪ 4.3.21 OS_Start()

説明

embOS スケジューラを起動します。

プロトタイプ

void OS_Start (void);

補足情報

この関数は embOS スケジューラを起動する関数であり、main()から呼び出される最後の関数である必要があります。OS_Start()は embOS が起動していることを記録します。実行状態は、

関数 OS_IsRunning()の呼び出しによって調べることができます。
OS_Start()は、最も優先順位の高いタスクをアクティブにし開始します。
OS_Start()は自動的に割り込みを可能にします。
OS_Start()は戻せません。
OS_Start()は割り込み、embOS タイマ、またはタスクから呼び出すことはできず、main()から一度だけ呼び出されることができます。

- 4.3.22 OS_Suspend()

説明
特定のタスクを中断します。

プロトタイプ
void OS_Suspend (OS_TASK* pTask);

Table 4.18: OS_Suspend() parameter list

補足情報
pTask が NULL ポインタの場合、現在のタスクが中断されます。関数がうまく機能した場合、指定したタスクの実行は中断され、タスクのサスペンドカウントが増加します。指定されたタスクはすぐに中断されます。OS_Resume()の呼び出しによってのみ再開することができます。
各タスクは、OS_MAX_SUSPEND_CNT の最大値を持つ中断カウントを持っています。サスペンドカウントがゼロより大きい場合は、そのタスクは中断されています。
OS_Resume()を呼び出さずに、OS_MAX_SUSPEND_CNT より高頻度で OS_Suspend()を呼び出す embOS のデバッグバージョンでは、タスクの内部サスペンドカウントが増加されず OS_ERR_SUSPEND_TOO_OFTEN と OS_Error()が呼び出されます。
この関数は、タスク切り替えをすぐに引き起こすので、割り込みハンドラやタイマーから呼び出すことはできません。
embOS のデバッグバージョンは関数 OS_Suspend()が割り込みハンドラから呼び出されると OS_Error()関数を呼び出します。

- 4.3.23 OS_SuspendAllTasks()

説明

実行しているタスクを除く全てのタスクを中断します。

プロトタイプ

```
void  OS_SuspendAllTasks  (void);
```

補足情報

この関数は、タスク切り換えを抑制するために使用することができます。アプリケーションを初期化または監視する時に便利かもしれません。呼び出し元のタスクは中断されません。

OS_SuspendAllTasks()の呼び出し後、呼び出し元のタスクは自身をブロックしたり、停止させたりすることができます。すべてのタスクが再開されるまで、他のタスクはアクティブ化されません。

中断したタスクはすべて OS_ResumeAllSuspendedtasks()の呼び出しによって再開することができます。

使用例

```
/* Sample to demonstrate the use of OS_SuspendAllTasks */
void InitTask(void) {
//
// High priority task started first after OS_Start()
//
OS_SuspendAllTasks();          // Ensure, no other existing task can run.
OS_SetInitialSuspendCnt(1);  // Ensure, no newly created task will run.
//
// Perform application initialization
//
...          // New tasks may be created, but can not start.
...          // Even when InitTask() blocks itself by a delay, no other task will run.
OS_SetInitialSuspendCnt(0);  // Reset the initial suspend count for tasks.
//
// Resume all tasks taht were blocked before or were created in suspended state.
//
OS_ResumeAllSuspendedTasks();
while (1) {
```

```
...          // Do the normal work
}
}
```

- 4.3.24 OS_TerminateTask()

説明

タスクを終了します。

プロトタイプ

```
void OS_TerminateTask (OS_TASK* pTask);
```

Table 4.19: OS_Terminate() parameter list

補足情報

pTask が NULL ポインタの場合は、現在のタスクは終了します。指定されたタスクは直ちに終了します。スタックとタスク制御ブロックに使用されるメモリは再度割り当てることができます。

embOS のバージョン 3.26 以降では、終了したタスクによって保持されているすべてのリソースは解放されます。あらゆるタスクは、その状態にかかわらず終了することがあります。この機能は、16 ビットまたは 32 ビット CPU ののどれでもデフォルトであり、embOS リソースの再コンパイルによって変更できます。 8 ビット CPU では、セマフォのような、他のタスクをブロックする可能性のあるリソースを一つでも保持しているタスクを終了させることは禁止されています。

安全に終了させるためには、embOS ソースは、コンパイルタイムスイッチ OS_SUPPORT_CLEANUP_ON_TERMINATE をアクティブ化して再コンパイルする必要があります。

embOS バージョン 3.82u 以降、OS_TerminateTask()が OS_Terminate()から置き換わっていますが OS_Terminate()まだ使用される可能性もあります。

重要

この関数は割り込みハンドラ内から呼び出せません。

- 4.3.25 OS_WakeTask()

説明

タスクの遅延を直ちに終了します。

プロトタイプ

void　OS_WakeTask　(OS_TASK*　pTask);

Table 4.20: OS_WakeTask() parameter list

補足情報

OS_Delay()または OS_DelayUntil()によって一定時間中断されていた特定のタスクを TS_READY（実行準備完了）に戻します。

指定されたタスクは前に最高の優先順位であったタスクの優先度よりも高い優先順位を持っている場合、すぐに開始されます。指定したタスクが、状態 TS_DELAY でない場合（すでにアクティブ化されていたり、既に遅延が期限切れていたりする、など何らかの理由で）このコマンドは無視されます。

- 4.3.26　OS_Yield()

説明

タスクスイッチを動かすためにスケジューラを呼び出します。

プロトタイプ

void　OS_Yield　(void);

補足情報

タスクはラウンドロビン方式で実行され手いる場合、同じ優先順位を持つ他のタスクがあり実行の準備ができていると中断されます。

- Chapter 5　　ソフトウェアタイマ

- 5.1　導入

ソフトウェアタイマは、ユーザーが指定したルーチンを指定された遅延の後に呼び出すオブジェクトです。基本的にマクロ OS_CREATETIMER()を使用してソフトウェアタイマの数を無制限に定義できます。

タイマは停止、開始、再開をハードウェアタイマと同じよう行うことができます。タイマを定

義するときは、遅延の期限切れ後に呼び出されるいずれかのルーチンを指定します。タイマルーチンは、割り込みルーチンと同じく、他のタスクよりも高い優先度を持ちます。そのため、タイマルーチンは割り込みルーチンと同様短くする必要があります。
ソフトウェアタイマは embOS によって割り込みを有効にして呼び出されるので、それらはいかなるハードウェアの割り込みによっても中断される可能性があります。一般的に、タイマはシングルショットモードで実行され、つまり一度だけ期限が切れコールバックルーチンを一度だけ呼び出します。コールバックルーチンの中から OS_RetriggerTimer()を呼び出すことによって、タイマはその初期遅延時間で再起動されるので、フリーランニングタイマと同じように動作します。タイマの状態は関数 OS_GetTimerStatus()、OS_GetTimerValue()、および OS_GetTimerPeriod()で確認することができます。

タイムアウトの最大値及び周期

タイムアウト値は整数値、すなわち 8/16 ビット CPU では 16 ビット値、32 ビット CPU では 32 ビット値として格納されます。比較は、(期限切れのタイムアウトが許可されているため)符号付き比較として行われます。これは 8/16 ビット CPU では 15 ビットのみ、32 ビット CPU では 31 ビットのみが使用できることを意味します。考慮すべきもう一つの要素はクリティカル領域で費やされる最大時間です。クリティカル領域でタイマが期限切れになるかもしれませんが、クリティカル領域からタイマルーチンを呼び出すことはできないので(タイマは「保留に」される)、タイマが一度に費やす最大時間は差し引かれる必要があります。(臨界領域から呼び出すことができないためです。ほとんどのシステムでは、これは単一の目盛りに過ぎません。しかし、安全のために、システムがクリティカル領域内の行の 255 ティック以上使っていないことを想定し、最大タイムアウト値を定義するマクロを設定しました。これは、通常は 8/16 ビットシステムには 0x7F00 あるいは 32 ビットシステムには 0x7FFFFF00 であり、OS_TIMER_MAX_TIME として RTOS.h で定義されています。システムがクリティカルセクションで break なしに 225 ビット以上使っている場合(このタイムにスケジューラを無効にする。推奨しません!)、お使いのアプリケーションが短いタイムアウト値を使用していることを確認する必要があります。

拡張ソフトウェアタイマ

タイマーコールバック関数にパラメータを渡すときに便利かもしれません。これにより、異なるソフトウェアタイマに対して 1 つのコールバック関数が使用できるようになります。
embOS のバージョン 3.32m 以降、拡張タイマ構造と関連する拡張タイマ関数は、パラメータをコールバック関数に渡すことを許可するように実装されました。

パラメーターの受け渡しではないコールバック関数を除き、拡張タイマは通常 embOS ソフトウェアタイマとまったく同様に動作し通常ソフトウェアタイマと同じように使用できます。

5.2 API 関数

Table 5.1: Software timers API

5.2.1 OS_CREATETIMER()

説明

ソフトウェアタイマを作成・開始するマクロです。

プロトタイプ

void OS_CREATETIMER (OS_TIMER* pTimer,
OS_TIMERROUTINE* Callback,
OS_TIME Timeout);)

Table 5.2: OS_CREATETIMER() parameter list

補足情報

embOS は、リンクされたリストを使用してタイマを追跡します。タイムアウトの有効期限が切れると、コールバックルーチンがすぐに呼び出されます（現在のタスクがクリティカル領域にあるか、または割り込み禁止された割り込みを持っていない限り）。

このマクロは関数 OS_CreateTimer()と OS_StartTimer()を使用します。これは、下位互換性（古いバージョンに対応した）のために供給され、新しいアプリケーションでは直接呼び出す必要があります。

OS_TIMERROUTINE は次のように RTOS.h で定義されています。

typedef void OS_TIMERROUTINE(void);

マクロのソース（RTOS.h 内):

```
#define OS_CREATETIMER(pTimer,c,d)\
OS_CreateTimer(pTimer,c,d); \
OS_StartTimer(pTimer);
```

使用例

```
OS_TIMER TIMER100;
void Timer100(void) {
LED = LED ? 0 : 1;          /* Toggle LED       */
OS_RetriggerTimer(&TIMER100); /* Make timer periodical */
}
void InitTask(void) {
/* Create and start Timer100          */
OS_CREATETIMER(&TIMER100, Timer100, 100);
}
```

- 5.2.2 OS_CreateTimer()

説明

ソフトウェアタイマを作成します（開始はしません）。

プロトタイプ

```
void OS_CreateTimer (OS_TIMER*    pTimer,
OS_TIMERROUTINE*  Callback,
OS_TIME   Timeout);)
```

Table 5.3: OS_CreateTimer() parameter list

補足情報

embOS は、リンクされたリストを使用してタイマを追跡します。タイムアウトの期限が切れると、コールバックルーチンがすぐに呼び出されます（現在のタスクがクリティカル領域にあるか、割り込み禁止された割り込みを持っていない限り）。タイマは自動的に開始されません。OS_StartTimer()または OS_RetriggerTimer()の呼び出しによって明示的に行われる必要があります。

RTOS.h で OS_TIMERROUTINE は次のように定義されています。

```
typedef  void  OS_TIMERROUTINE(void);
```

使用例

```
OS_TIMER  TIMER100;
void  Timer100(void)  {
LED  =  LED  ?  0  :  1;          /*  Toggle  LED       */
OS_RetriggerTimer(&TIMER100);  /*  Make  timer  periodical  */
}
void  InitTask(void)  {
/*  Create  Timer100,  start  it  elsewhere  */
OS_CreateTimer(&TIMER100,  Timer100,  100);
OS_StartTimer(&TIMER100);
}
```

- 5.2.3 OS_StartTimer()

説明

ソフトウェアタイマを開始します。

プロトタイプ

```
void  OS_StartTimer  (OS_TIMER*  pTimer);
```

Table 5.4: OS_StartTimer() parameter list

補足情報

OS_StartTimer() は以下のような目的で使用されます。:

·OS_CreateTimer()によって作成されたタイマを起動する。タイマはその初期タイマ値で起動する。

·OS_StopTimer()の呼び出しによって停止したタイマを再開する。この場合タイマは、タイマを停止することによって保たれていた残りの時間値を継続します。

重要

この関数は実行中のタイマに影響を与えません。実行していないタイマではなく、既に終了しているタイマに影響を与えます。これらのタイマを再開するには OS_RetriggerTimer()を使用してください。

- 5.2.4 OS_StopTimer()

説明

ソフトウェアタイマを停止します。

プロトタイプ

void　　opTimer　(OS_TIMER*　pTimer);

Table 5.5: OS_StopTimer() parameter list

補足情報

実際のタイマの値（終了までの時間）は OS_StartTimer()によって続行を許可されるまで保持されます。

- 5.2.5 OS_RetriggerTimer()

説明

初期タイムの値でソフトウェアタイマを再開します。

プロトタイプ

void　OS_RetriggerTimer　(OS_TIMER*　pTimer);

Table 5.6: OS_RetriggerTimer() parameter list

補足情報

OS_RetriggerTimer（）は、タイマの作成時にプログラムされた初期タイム値あるいは関数 OS_SetTimerPeriod（）を使用してタイマを再開します。

OS_RetriggerTimer（）は、タイマの状態に関係なく呼び出すことができます。実行中のタイマは完全な初期タイムを使って続行します。事前に停止した、あるいは期限が切れていたタイマは再起動されます。

使用例

OS_TIMER　TIMERCursor;

```
BOOL CursorOn;
void TimerCursor(void) {
if (CursorOn) ToggleCursor(); /* Invert character at cursor-position */
OS_RetriggerTimer(&TIMERCursor); /* Make timer periodical */
}
void InitTask(void) {
/* Create and start TimerCursor */
OS_CREATETIMER(&TIMERCursor, TimerCursor, 500);
}
```

- 5.2.6 OS_SetTimerPeriod()

説明

ソフトウェアタイマの新しいタイマリロード値を設定します。

プロトタイプ

```
void OS_SetTimerPeriod (OS_TIMER* pTimer,
OS_TIME Period);
```

Table 5.7: OS_SetTimerPeriod() parameter list

補足情報

OS_SetTimerPeriod() は特定のタイマの初期タイム値を設定します。 Period は、タイマが OS_RetriggerTimer()によって再始動する時に初期値として使用されるリロード値です。

使用例

```
OS_TIMER TIMERPulse;
BOOL CursorOn;
void TimerPulse(void) {
if TogglePulseOutput(); /* Toggle output */
OS_RetriggerTimer(&TIMERCursor); /* Make timer periodical */
}
void InitTask(void) {
/* Create and start Pulse Timer with first pulse = 500ms */
```

```
OS_CREATETIMER(&TIMERPulse, TimerPulse, 500);
/* Set timer period to 200 ms for further pulses */
OS_SetTimerPeriod(&TIMERPulse, 200);
}
```

- 5.2.7 OS_DeleteTimer()

説明

ソフトウェアタイマを停止し削除します。

プロトタイプ

```
void OS_DeleteTimer (OS_TIMER* pTimer);
```

Table 5.8: OS_DeleteTimer() parameter list

補足情報

タイマは停止されそれにより実行中のタイマの連結リストから除かれます。embOS のデバッグビルドではこのタイマは無効と記されます。

- 5.2.8 OS_GetTimerPeriod()

説明

ソフトウェアタイマの現在のリロード値を返します。

プロトタイプ

```
OS_TIME OS_GetTimerPeriod (OS_TIMER* pTimer);
```

Table 5.9: OS_GetTimerPeriod() parameter list

戻り値

タイマ値許容範囲が 8/16 ビット CPU 用に 1 から 215-1=0x7FFF = 32767 の間の整数として、また 32 ビット CPU 用に 1 から<=231-1=0x7FFFFFFF の間の整数として定義されているタイプ OS_TIME。

補足情報

タイマが OS_RetriggerTimer()によって再開された場合、返される時間はタイマの初期値として設定されたリロード値です。

- 5.2.9 OS_GetTimerValue()

説明

ソフトウェアタイマの残りタイマ値を返します。

プロトタイプ

OS_TIME OS_GetTimerValue (OS_TIMER* pTimer);

Table 5.10: OS_GetTimerValue() parameter list

戻り値

タイマ値許容範囲が 8/16 ビット CPU 用に 1 から 215-1=0x7FFF = 32767 の間の整数として、また 32 ビット CPU 用に 1 から<=231-1=0x7FFFFFFF の間の整数として定義されているタイプ OS_TIME。

返されるタイム値は、タイマが期限切れになるまでの embOS ティック単位での残りのタイマです。

- 5.2.10 OS_GetTimerStatus()

説明

ソフトウェアタイマの現在実行中のタイマステータスを返します

プロトタイプ

unsigned char OS_GetTimerStatus (OS_TIMER* pTimer);

Table 5.11: OS_GetTimerStatus parameter list

戻り値

指定されたタイマーが動作しているかどうかを示す符号なしの文字：

0:タイマーが停止しました。

！ = 0:タイマーが実行中です。

- 5.2.11 OS_GetpCurrentTimer()

説明

終了したタイマーのデータ構造にポインタを返します。

プロトタイプ

OS_TIMER* OS_GetpCurrentTimer (void);

戻り値

OS_TIMER*: タイマ制御構造へのポインタ

補足情報

OS_GetpCurrentTimer()の戻り値は、タイマーコールバック関数の実行時に有効で、それ以外の場合には定義されません。一つのコールバック関数が複数のタイマを使用する必要がある場合に限り、この機能は期限が切れたタイマを調べるために使用することができます。

以下の例では OS_GetpCurrentTimer()の使用法を示しています。 embOS のバージョン 3.32m 以降では embOS に付属する拡張タイマの構造と関数は、コールバック関数用の個々のパラメータを持つソフトウェアタイマを作成し、使用するために使うことができます。OS_TIMER は、構造内の最初の要素でなければならないことに注意してください。

使用例

```
#include "RTOS.H"
/*********************************************************
*
*          Types
*/
typedef struct {     /* Timer object with its own user data */
OS_TIMER Timer; /* OS_TIMER have to be first element */
void*       pUser;
} TIMER_EX;
/*********************************************************
*
*          Variables
```

```
*/
TIMER_EX Timer_User;
int a;
/*********************************************************
*
*          Local Functions
*/
void CreateTimer(TIMER_EX* timer, OS_TIMERROUTINE* Callback, OS_UINT
Timeout,
void* pUser) {
timer->pUser = pUser;
OS_CreateTimer((OS_TIMER*) timer, Callback, Timeout);
}
void cb(void) { /* Timer callback function for multiple timers */
TIMER_EX* p = (TIMER_EX*)OS_GetpCurrentTimer();
void* pUser = p->pUser;      /* Examine user data */
OS_RetriggerTimer(&p->Timer);          /* Retrigger timer */
}
/*********************************************************
*
*          main
*/
int main(void) {
OS_InitKern();       /* Initialize OS */
OS_InitHW();         /* Initialize Hardware for OS */
CreateTimer(&Timer_User, cb, 100, &a);
OS_Start();          /* Start multitasking */
return 0;
}
```

- 5.2.12 OS_CREATETIMER_EX()

説明

拡張ソフトウェアタイマを作成・開始するマクロです。

プロトタイプ

```
void  OS_CREATETIMER_EX  (OS_TIMER_EX*  pTimerEx,
OS_TIMER_EX_ROUTINE*  Callback,
OS_TIME Timeout
void*       pData)
```

Table 5.12: OS_CREATETIMER_EX() parameter list

補足情報

embOS は、リンクされたリストを使用してタイマを追跡します。タイムアウトの有効期限が切れていると、コールバックルーチンはすぐに呼び出されます（現在のタスクがクリティカル領域にあるか、または割り込み禁止された割り込みを持っていない限り）。

このマクロは、関数 OS_CreateTimerEx()と OS_StartTimerEx()を使用します。

RTOS.h で OS_TIMER_EX_ROUTINE は次のように定義されています。

```
typedef  void  OS_TIMER_EX_ROUTINE(void  *);
```

マクロのソース (RTOS.h 内):

```
#define  OS_CREATETIMER_EX(pTimerEx,cb,Timeout,pData)  \
OS_CreateTimerEx(pTimerEx,cb,Timeout,pData);  \
OS_StartTimerEx(pTimerEx)
```

使用例

```
OS_TIMER  TIMER100;
OS_TASK TCB_HP;
void  Timer100(void*  pTask)  {
LED  =  LED  ?  0  :  1;          /*  Toggle  LED       */
if  (pTask  !=  NULL)  {
OS_SignalEvent(0x01,  (OS_TASK*)pTask);
}
OS_RetriggerTimerEx(&TIMER100);  /*  Make  timer  periodical  */
}
```

```
void InitTask(void) {
/* Create and start Timer100 */
OS_CREATETIMER_EX(&TIMER100, Timer100, 100, (void*) &TCB_HP);
}
```

- 5.2.13 OS_CreateTimerEx()

説明

拡張ソフトウェアタイマを作成します（開始はしません）。

プロトタイプ

```
void OS_CreateTimerEx (OS_TIMER_EX* pTimerEx,
OS_TIMER_EX_ROUTINE* Callback,
OS_TIME Timeout,
void* pData)
```

Table 5.13: OS_CreateTimerEx() parameter list

補足情報

embOS は、リンクされたリストを使用してタイマを追跡します。タイムアウトが期限切れになると、（現在のタスクがクリティカル領域にあるか、または割り込み禁止された割り込みを持っていない限り）コールバックルーチンがすぐに呼び出されます。

拡張ソフトウェアタイマは自動的には開始されません。OS_StartTimerEx() または OS_RetriggerTimerEx() の呼び出しによって明示的に行う必要があります。

RTOS.h で OS_TIMER_EX_ROUTINE は次のように定義されています。

```
typedef void OS_TIMER_EX_ROUTINE(void*);
```

使用例

```
OS_TIMER TIMER100;
OS_TASK TCB_HP;
void Timer100(void* pTask) {
LED = LED ? 0 : 1; /* Toggle LED */
if (pTask != NULL) {
```

```
OS_SignalEvent(0x01, (OS_TASK*) pTask);
}
OS_RetriggerTimerEx(&TIMER100); /* Make timer periodical */
}
void InitTask(void) {
/* Create Timer100, start it elsewhere later on*/
OS_CreateTimerEx(&TIMER100, Timer100, 100, (void*) & TCB_HP);
}
```

- 5.2.14 OS_StartTimerEx()

説明

拡張ソフトウェアタイマを開始します。

プロトタイプ

```
void OS_StartTimerEx (OS_TIMER_EX* pTimerEx);
```

Table 5.14: OS_StartTimereEx() parameter list

補足情報

OS_StartTimerEx() は以下のような目的で使用されます:

· OS_CreateTimerEx()によって作成された拡張ソフトウェアタイマを起動します。タイマは初期タイマ値で起動します。

· OS_StopTimerEx()の呼び出しによって停止したタイマを再開します。この場合、タイマは自身を停止することによって保存されていた残りのタイマ値から継続されます。

重要

この関数は、実行中のタイマには影響を与えません。実行されていないが期限が切れているタイマにも影響しません。これらのタイマを再起動するには OS_RetriggerTimerEx()を使用します。

- 5.2.15 OS_StopTimerEx()

説明

拡張ソフトウェアタイマを停止します。

プロトタイプ

void　OS_StopTimerEx　(OS_TIMER_EX*　pTimerEx);

Table 5.15: OS_StopTimerEx() parameter list

補足情報

拡張ソフトウェアタイマの実際のタイム値（期限切れまでのタイム）は imerEx() によりタイマの続行が許可されるまで保持されます。

- 5.2.16　OS_RetriggerTimerEx()

説明

初期タイム値で拡張ソフトウェアタイマを再開します。

プロトタイプ

void　OS_RetriggerTimerEx　(OS_TIMER_EX*　pTimerEx);

Table 5.16: OS_RetriggerTimerEx() parameter list

戻り値

OS_RetriggerTimerEx()は、タイマの作成時にプログラムされたか、関数OS_SetTimerPeriodEx()を使用して設定された初期タイム値を使用して、拡張ソフトウェアタイマを再起動します。

OS_RetriggerTimerEx()はタイマの状態に関係なく呼び出すことができます。実行中のタイマは完全な初期タイムを使って継続されます。事前に停止した、または期限が切れていたタイマは再起動されます。

使用例

```
OS_TIMER  TIMERCursor;
OS_TASK  TCB_HP;
BOOL  CursorOn;
```

```
void TimerCursor(void* pTask) {
if (CursorOn != 0) ToggleCursor(); /* Invert character at cursor-position */
OS_SignalEvent(0x01, (OS_TASK*) pTask);
OS_RetriggerTimerEx(&TIMERCursor); /* Make timer periodical */
}
void InitTask(void) {
/* Create and start TimerCursor */
OS_CREATETIMER_EX(&TIMERCursor, TimerCursor, 500, (void*)&TCB_HP);
}
```

- 5.2.17 OS_SetTimerPeriodEx()

説明

拡張ソフトウェアタイマの新しいタイマリロード値を設定します。

プロトタイプ

```
void OS_SetTimerPeriodEx (OS_TIMER_EX* pTimerEx,
OS_TIME Period);
```

Table 5.17: OS_SetTimerPeriodEx() parameter list

補足情報

OS_SetTimerPeriodEx()は指定された拡張ソフトウェアタイマの初期タイム値を設定します。期間は、タイマが OS_RetriggerTimerEx()によって次回再開されたときに初期値として使用されるリロード値です。

OS_SetTimerPeriodEx()を呼び出しても、拡張ソフトウェアタイマの残り時間には影響を与えません。

使用例

```
OS_TIMER_EX TIMERPulse;
OS_TASK TCB_HP;
void TimerPulse(void* pTask) {
OS_SignalEvent(0x01, (OS_TASK*) pTask);
```

```
OS_RetriggerTimerEx(&TIMERPulse); /* Make timer periodical */
}
void InitTask(void) {
/* Create and start Pulse Timer with first pulse == 500ms */
OS_CREATETIMER_EX(&TIMERPulse, TimerPulse, 500, (void*)&TCB_HP);
/* Set timer period to 200 ms for further pulses */
OS_SetTimerPeriodEx(&TIMERPulse, 200);
}
```

- 5.2.18 OS_DeleteTimerEx()

説明

拡張ソフトウェアタイマを停止し消去します。

プロトタイプ

```
void OS_DeleteTimerEx(OS_TIMER_EX* pTimerEx);
```

Table 5.18: OS_DeleteTimerEx() parameter list

補足情報

拡張ソフトウェア･タイマは停止し、したがって、実行中のタイマのリンクされたリストのうちに削除されます。 embOS のデバッグビルドでは、タイマも無効として記されます。

- 5.2.19 OS_GetTimerPeriodEx()

説明

拡張ソフトウェアタイマの現在のリロード値を返します。

プロトタイプ

```
OS_TIME OS_GetTimerPeriodEx (OS_TIMER_EX* pTimerEx);
```

Table 5.19: OS_GetTimerPeriodEx() parameter list

戻り値

タイマ値の許容範囲が 88/16 ビット CPU 用に 1 から 215-1=0x7FFF=32767 の間の整数として、また 32 ビット CPU 用に 1 から 231-1=0x7FFFFFFF の間の整数として定義されているタイプ OS_TIME。

補足情報

返される期間は、タイマが作成された場合、または OS_SetTimerPeriodEx()の呼び出しによって変更された初期値として設定されたタイマのリロード値です。このリロード値は、タイマが OS_RetriggerTimerEx()によって再起動される時点として使用されます。

- 5.2.20 OS_GetTimerValueEx()

説明

拡張ソフトウェアタイマの残りのタイマ値を返します。

プロトタイプ

```
OS_TIME  OS_GetTimerValueEx(OS_TIMER_EX*  pTimerEx);
```

Table 5.20: OS_GetTimerValueEx() parameter list

戻り値

タイマ値の許容範囲が 88/16 ビット CPU 用に 1 から 215-1=0x7FFF=32767 の間の整数として、また 32 ビット CPU 用に 1 から 231-1=0x7FFFFFFF の間の整数として定義されているタイプ OS_TIME。返されたタイム値は embOS の tick 単位で拡張ソフトウェアタイマが期限切れになるまでの残り時間です。

- 5.2.21 OS_GetTimerStatusEx()

説明

現在実行中の拡張ソフトウェアタイマのタイマステータスを返します。

プロトタイプ

```
unsigned  char  OS_GetTimerStatusEx  (OS_TIMER_EX*  pTimerEx);
```

Table 5.21: OS_GetTimerStatusEx parameter list

戻り値

指定されたタイマが動作しているかどうかを示す符号なしの文字：

0:タイマが停止しました。

! =0:タイマが実行されています。

- 5.2.22 OS_GetpCurrentTimerEx()

説明

ポインタに終了した拡張タイマのデータ構造を返します。

プロトタイプ

OS_TIMER_EX* OS_GetpCurrentTimerEx (void);

戻り値

OS_TIMER_EX*: 拡張ソフトウェアタイマの制御構造へのポインタ

補足情報

OS_GetpCurrentTimerEx()の戻り値は、タイマーコールバック関数の実行時に有効で、それ以外の場合は定義されません。 一つのコールバック関数を複数の拡張タイマのために使用する必要がある場合は、この関数は期限切れのタイマを調べるために使用することができます。

使用例

```
#include "RTOS.H"
OS_TIMER_EX MyTimerEx;
/*********************************************************
*
*           Local Functions
*/
void cbTimerEx(void* pData) { /* Timer callback function for multiple timers */
```

```
OS_TIMER_EX*  pTimerEx;
pTimerEx  =  OS_GetpCurrentTimerEx();
OS_SignalEvent(0x01,  (OS_TASK*)  pData);
OS_RetriggerTimer(pTimerEx);  /*  Retrigger  timer                */
}
```

- Chapter 6　　リソースセマフォ

6.1　　導入

リソースセマフォは、リソースの同時使用によって引き起こされる衝突を回避することによってリソースを管理するために使用されます。管理されるリソースはどの種類でも可能です。リエントラントではないプログラム、ディスプレイのようなハードウェアの一部、一度に一つのタスクによってのみ書かれるフラッシュプロム、一度に一つのタスクによってのみ管理されるCNC 制御の中のモーターなどです。

基本的な手順は次のとおりです。

最初のリソースを使用するすべてのタスクが OS_Use()または embOS の OS_Request()ルーチンを呼び出してリソースを要求します。リソースが利用可能である場合、タスクのプログラムの実行が継続されますが、リソースは他のタスクのためにブロックされます。 2 番目のタスクが最初のタスクで使用中のリソースを使用しようとする場合、最初のタスクがリソースを解放するまでこの 2 回目のタスクは中断されます。しかし、リソースを使用する最初のタスクがそのリソースのために再度 OS_Use()を呼び出した場合、リソースが他のタスクのためだけにブロックされているため、そのタスクは中断されません。

次の図は、リソースを使用するプロセスを示しています。

(略)

リソースセマフォは、リソースが特定のタスクで OS_Request()または OS_Use()を呼び出すことによって要求された回数を追跡するカウンタを含んでいます。リソースセマフォは、そのカウンタが 0 になったときに解放されますが、これは OS_Unuse()ルーチンが OS_Use()または OS_Request()とまったく同じ回数呼びだされなければならいことを意味します。そうでない場

合は、リソースが他のタスクのためにブロックされたままです。
一方、タスクは OS_Unuse()を呼び出して所有していないリソースを解放することはできません。 embOS のデバッグバージョンでは、このタスクによって所有されていないセマフォのために OS_Unuse()の呼び出すと、エラーハンドラ OS_Error()が呼び出されます。
リソースセマフォの使用例
ここでは、2 つのタスクが互いから完全に独立して LC ディスプレイにアクセスします。 LCD はリソースセマフォで保護する必要があるリソースです。タスクは LCD への書き込みが行われている別のタスクを中断しないことがあり、そうしないと次のことが起こります。
·タスク A がカーソルを置く
·タスク B がタスク A に割り込みカーソルを置き直す
·タスク A は、LCD のメモリ内に間違った場所に書き込まれる
このような状況を回避するために、LCD ごとにタスクがアクセスする必要があり、それは最初に OS_Use()を呼び出す（リソースがブロックされている場合には自動的に待つ）ことによって要求されます。LCD が書き込まれた後、OS_Unuse()の呼び出しによって解放される。

```
/*
*          demo program to illustrate the use of resource semaphores
*/
OS_STACKPTR int StackMain[100], StackClock[50];
OS_TASK TaskMain,TaskClock;
OS_RSEMA SemaLCD;
void TaskClock(void) {
char t=-1;
char s[] = "00:00";
while(1) {
while (TimeSec==t) Delay(10);
t= TimeSec;
s[4] = TimeSec%10+'0';
s[3] = TimeSec/10+'0';
s[1] = TimeMin%10+'0';
s[0] = TimeMin/10+'0';
OS_Use(&SemaLCD);          /* Make sure nobody else uses LCD */
```

```
LCD_Write(10,0,s);
OS_Unuse(&SemaLCD);        /* Release LCD      */
}
}
void TaskMain(void) {
signed char pos ;
LCD_Write(0,0,"Software tools by Segger !          ") ;
OS_Delay(2000);
while (1) {
for ( pos=14 ; pos >=0 ; pos-- ) {
OS_Use(&SemaLCD);          /* Make sure nobody else uses LCD  */
LCD_Write(pos,1,"train ");     /* Draw train        */
OS_Unuse(&SemaLCD);        /* Release LCD      */
OS_Delay(500);
}
OS_Use(&SemaLCD);          /* Make sure nobody else uses LCD  */
LCD_Write(0,1,"      ") ;
OS_Unuse(&SemaLCD);        /* Release LCD  */
}
}
void InitTask(void) {
OS_CREATERSEMA(&SemaLCD);       /* Creates resource semaphore  */
OS_CREATETASK(&TaskMain,          0, Main,  50, StackMain);
OS_CREATETASK(&TaskClock, 0, Clock, 100, StackClock);
}
```

ほとんどのアプリケーションでは、リソースにアクセスするルーチンは、リソースを使用するときにユーザーが心配せずシングルタスクシステムを使用するのと同様に使用できるようにOS_Use()と OS_Unuse()を自動的に呼び出す必要があります。以下は、実際にディスプレイにアクセスするルーチンにリソースを提供する方法の例を示しています。

```
/*
*          Simple example when accessing single line dot matrix LCD
```

```
*/
OS_RSEMA RDisp; /* Define resource semaphore */
void UseDisp() {    /* Simple routine to be called before using display */
OS_Use(&RDisp);
}
void UnuseDisp() { /* Simple routine to be called after using display */
OS_Unuse(&RDisp);
}
void DispCharAt(char c, char x, char y) {
UseDisp();
LCDGoto(x, y);
LCDWrite1(ASCII2LCD(c));
UnuseDisp();
}
void DISPInit(void) {
OS_CREATERSEMA(&RDisp);
}
```

- 6.2 API 関数

Table 6.1: Resource semaphore API functions

- 6.2.1 OS_CREATERSEMA()

説明

リソースセマフォを作成するマクロです。

プロトタイプ

```
void OS_CREATERSEMA (OS_RSEMA* pRSema);
```

Table 6.2: OS_CREATESEMA() parameter list

補足情報

リソースセマフォの作成後、リソースはブロックされません。カウンターの値は 0 になります。

- 6.2.2 OS_Use()

説明

リソースを要求し他のタスクのためにブロックします。

プロトタイプ

int　OS_Use　(OS_RSEMA*　pRSema);

Table 6.3: OS_Use() parameter list

戻り値

セマフォのカウンタ値。

1 より大きい値は、リソースがすでに呼び出し元のタスクによってロックされたことを意味します。

補足情報

以下のような状況になる可能性があります:

· ケース A:リソースが使用されていない。

リソースがタスクで使用されていない場合、そのセマフォのカウンタは 0 であり、そのリソースはカウンタを増加させかつセマフォにそれを使用するタスクの一意のコードを書くことによって、他のタスクのためにブロックされます。

· ケース B:リソースは、このタスクによって使用されています。

セマフォのカウンタは増加します。プログラムは途切れることなく続行されます

· ケース C:リソースが別のタスクで使用されていますリソースセマフォが解放されるまで、このタスクの実行は中断されます。

一方、リソースセマフォでブロックされているタスクがセマフォをブロックしているタスクよりも優先順位が高い場合、ブロックしているタスクがリソースセマフォを要求しているタスクの優先度を割り当てられます。これは優先順位の逆転と呼ばれます。優先順位の逆転は一時的にのみ、タスクの優先順位を上げるものであり、減らすことはできません。

リソースセマフォを待つことができるタスクの数には制限がありません。スケジューラの規則に従って、リソースを待機しているすべてのタスクのうち最も優先順位の高いタスクがリソー

スにアクセスでき、プログラムの実行を継続することができます。

重要

この関数は割り込みハンドラ内から呼び出せません。

下記のダイアグラムは OS_Use()のルーチンワークを示しています:

- 6.2.3 OS_UseTimed()

説明

リソースが特定の時間利用可能である場合にリソースを要求し、他のタスクのためにブロックしようとします。

プロトタイプ

int OS_UseTimed(OS_RSEMA* pRSema, OS_TIME TimeOut)

Table 6.4: OS_UseTimed() parameter list

戻り値

整数値:

0:失敗。セマフォはタイムアウト前に使用不可能。

>0:成功。リソースセマフォが使用可能。セマフォのカウンタ値。

1 より大きい値は、リソースがすでに呼び出しタスクによってロックされたことを意味します。

補足情報

以下のような状況になる可能性があります。:

· ケース A:リソースがが使用されていない。

リソースがタスクに使用されていない場合、そのセマフォのカウンタが 0 であることを意味し、そのリソースはカウンタを増加させかつセマフォにそれを使用するタスクの一意のコードを書くことによって、他のタスクのためにブロックされます。

· ケース B: リソースがこのタスクに使用されている。

セマフォのカウンタは単純に増加されます。プログラムは途切れることなく続行されます。

· ケース C: リソースが他のタスクに使用されている。

リソースセマフォが解放されるかタイムアウトタイムが期限切れになるまで、このタスクの実

行は中断されます。その間にリソースセマフォにブロックされているタスクがセマフォをブロックしているタスクよりも優先順位が高い場合、ブロックしているタスクはリソースセマフォを要求しているタスクの優先度を割り当てられます。これは、優先順位の逆転と呼ばれています。優先順位の逆転は一時的タスクの優先順位を上げるだけのものであり、減らすことはできません。リソースセマフォがタイムアウトタイム内に利用可能になった場合、呼び出し元のタスクはリソースを要求し、関数は 0 より大きな値を返し、そうでない場合、つまりリソースが利用可能でなければ、関数は 0 を返します。リソースセマフォを待つができるタスクの数に制限はありません。スケジューラの規則により、リソースを待つタスクのうち最も優先順位の高いタスクは、リソースへのアクセスを取得して、プログラムの実行を継続することができます。

重要
この関数は割り込みハンドラ内から呼び出せません。

▪ 6.2.4 OS_Unuse()

説明
現在タスクで使用されているセマフォを解放します。

プロトタイプ
void OS_Unuse (OS_RSEMA* pRSema)

Table 6.5: OS_Unuse() parameter list

補足情報
OS_Unuse()はそのセマフォが OS_Use()または OS_Request()の呼び出しによって使用された後にのみ、リソースセマフォ上で使用することができます。 OS_Unuse()はセマフォの使用カウンタを減少させますが、その値は負になることはありません。このカウンタの値が負になった場合、デバッグバージョンでは、エラーコード OS_ERR_UNUSE_BEFORE_USE で embOS エラーハンドラ OS_Error()を呼び出します。デバッグバージョンでは、OS_Unuse()がリソースを持たないタスクから呼び出された場合にも OS_Error()が呼び出されます。
この場合のエラーコードは OS_ERR_RESOURCE_OWNER です。

重要

この関数は割り込みハンドラ内から呼び出せません。

- 6.2.5 OS_Request()

説明

特定のセマフォが利用可能である場合に要求し、他のタスクのためにブロックします。どのような場合でも実行は継続されます。

プロトタイプ

```
char  OS_Request  (OS_RSEMA*  pRSema);
```

Table 6.6: OS-Request() parameter list

戻り値

1: リソースは現在呼び出しタスクに使用されており、利用可能です

0: リソースは利用不可能でした

補足情報

下記のダイアグラムは、OS_Request()の動作を示しています:

使用例

```
if (!OS_Request(&RSEMA_LCD) ) {
LED_LCDBUSY = 1;          /* Indicate that task is waiting for     */
/* resource              */
OS_Use(&RSEMA_LCD);       /* Wait for resource*/
LED_LCDBUSY = 0;          /* Indicate task is no longer waiting   */
}
DispTime();           /* Access the resource LCD   */
OS_Unuse(&RSEMA_LCD);   /* Resource LCD is no longer needed   */
```

- 6.2.6 OS_GetSemaValue()

説明

指定されたリソースセマフォの使用カウンタの値を返します。

プロトタイプ

int　OS_GetSemaValue　(OS_SEMA*　pSema);

Table 6.7: OS_GetSemaValue() parameter list

戻り値

セマフォのカウンタ。

0 の値はリソースが利用可能であることを示しています。

- 6.2.7 OS_GetResourceOwner()

説明

現在リソースを利用中（ブロック中）のタスクにポインタを返します。

プロトタイプ

OS_TASK*　OS_GetResourceOwner　(OS_RSEMA*　pSema);

Table 6.8: OS_GetResourceOwner() parameter list

戻り値

リソースをブロックしているタスクへのポインタ。

0 の値はリソースが利用可能であることを示します。

- 6.2.8 OS_DeleteRSema()

説明

特定のリソースセマフォを消去します。そのセマフォのメモリはほかの目的で使用したり同じメモリを使用する他のリソースセマフォを作成したりするために使うことができます。

プロトタイプ

void　OS_DeleteRSema　(OS_RSEMA*　pRSema);

Table 6.9: OS_DeleteRSema parameter list

補足情報

リソースセマフォを消去する前に、リソースセマフォを要求しているタスクがないことを確認してください。embOS のデバッグバージョンでは、既に使用されているリソースセマフォが消去された場合 OS_Error() を呼び出します。リソースセマフォを動的に作成するシステムでは、再度作成する前にリソースセマフォを削除する必要があります。そうでない限りセマフォ処理が正しく動作しません。

Chapter 7　　計数セマフォ

7.1　導入

計数セマフォは embOS に管理されるカウンタです。リソースセマフォやイベント及びメールボックスに比べ広く利用されてはいませんが、場合によっては非常に便利です。計数セマフォは、タスクが信号により一定期間待機する必要がある場合に使用されます。セマフォはあらゆる位置、タスクおよび割り込みからどんな方法でもアクセス可能です。

計数セマフォの使用例

```
OS_STACKPTR int Stack0[96], Stack1[64];        /* Task stacks */
OS_TASK TCB0, TCB1;       /* Data-area for tasks (task-control-blocks)       */
OS_CSEMA SEMALCD;
void Task0(void) {
while(1) {
Disp("Task0 will wait for task 1 to signal");
OS_WaitCSema(&SEMALCD);
Disp("Task1 has signaled !!");
OS_Delay(100);
}
}
void Task1(void) {
while(1) {
```

```
OS_Delay(5000);
OS_SignalCSema(&SEMALCD);
}
}
void InitTask(void) {
OS_CREATECSEMA(&SEMALCD);       /* Create Semaphore */
OS_CREATETASK(&TCB0, NULL, Task0, 100, Stack0); /* Create Task0       */
OS_CREATETASK(&TCB1, NULL, Task1,          50, Stack1); /* Create Task1*/
}
```

- 7.2 API 関数

Table 7.1: Counting semaphores API functions

- 7.2.1 OS_CREATECSEMA()

説明

初期カウント値 0 で計数セマフォを作成するマクロです。

プロトタイプ

```
void OS_CREATECSEMA (OS_CSEMA* pCSema);
```

Table 7.2: OS_CREATECSEMA() parameter list

補足情報

計数セマフォを作成するために、OS_SCEMA タイプのデータ構造ははメモリ内で定義され OS_CREATECSEMA()で開始される必要があります。このマクロで作成されるセマフォの値は 0 になります。どのような目的であれ 0 より大きいカウント値でセマフォを作成しなければならない場合には、関数 OS_CreateCSema()を使用してください。

- 7.2.2 OS_CreateCSema()

説明

初期カウント値で計数セマフォを作成します。

プロトタイプ

void　OS_CreateCSema　(OS_CSEMA*　pCSema,

OS_UINT InitValue);

Table 7.3: OS_CreateCSema() parameter list

補足情報

計数セマフォを作成するために、OS_CSEMA タイプのデータ構造はメモリで定義され OS_CreateCSema()によって初期化される必要があります。作成されたセマフォをゼロにする必要がある場合には、マクロ OS_CREATECSEMA()を使用する必要があります。

- 7.2.3 OS_SignalCSema()

説明

セマフォのカウンタを増加させます。

プロトタイプ

void　OS_SignalCSema　(OS_CSEMA　*　pCSema);

Table 7.4: OS_SignalCSema() parameter list

補足情報

OS_SignalCSema()はカウンタを増加させることによってセマフォにイベントを通知します。1 つ以上のタスクがセマフォに通知されるイベントを待機している場合、最も優先順位の高いタスクが実行中のタスクになります。カウンタは、8/16 ビット CPU では 0xFFFF の値、また 32 ビット CPU では 0xFFFFFFFF の値の最大値を持てます。この上限を超いないことを確認するのはアプリケーションの責任です。embOS のデバッグバージョンではオーバーフローが発生した場合、カウンタのオーバーフローを検出しエラーコード OS_ERR_CSEMA_OVERFLOW で OS_Error()を呼び出します。

- 7.2.4 OS_SignalCSemaMax()

説明

セマフォのカウンタを決められた最大値まで増加させます。

プロトタイプ

void　OS_SignalCSemaMax　(OS_CSEMA*　　pCSema,
OS_UINT MaxValue　);

Table 7.5: OS_SignalCSemaMax() parameter list

補足情報

セマフォカウンタの現在の値が指定された最大値以下である場合に限り、OS_SignalCSemaMax()はカウンタを増加することによってセマフォにイベントを通知します。1 つ以上のタスクがこのセマフォに信号が送られるイベントを待機している場合は、タスクが READY 状態に置かれ、最も優先順位の高いタスクが実行タスクになります。 MaxValue1 で OS_SignalCSemaMax()を呼び出すとバイナリセマフォとしてカウンティングセマフォを処理します。

- 7.2.5 OS_WaitCSema()

説明

セマフォのカウンタを減少させます。

プロトタイプ

void　OS_WaitCSema　(OS_CSEMA*　pCSema);

Table 7.6: OS_WaitCSema() parameter list

補足情報

セマフォのカウンタが 0 でない場合、カウンタは減少し、プログラムの実行は継続されます。カウンタが 0 の場合、WaitCSema()はカウンタが別のタスク、タイマーまたは OS_SignalCSema()の呼び出しを介した割り込みハンドラによって増加されるまで、待機します。そしてカウンタは減少し、プログラムの実行は継続されます。

セマフォを待つタスクの数には制限がありません。スケジューラの規則によって、セマフォを待っているすべてのタスクのうち最も優先順位の高いタスクがプログラムの実行を継続します。

重要

この関数は割り込みカウンタから呼び出せません。

- 7.2.6 OS_WaitCSemaTimed()

説明

セマフォが決められた時間内に利用可能である場合にセマフォカウンタを減少させます。

プロトタイプ

int OS_WaitCSemaTimed (OS_CSEMA* pCSema,
OS_TIME TimeOut);

Table 7.7: OS_WaitCSemaTimed parameter list

戻り値

整数値:

0:失敗。セマフォはタイムアウト前に使用できません。

1:OK。セマフォは使用可能でカウンタは減少します。

補足情報

セマフォのカウンタが 0 でない場合、カウンタは減少し、プログラムの実行は継続されます。カウンタが 0 の場合、WaitCSemaTimed()はセマフォが OS_SignalCSema()の呼び出しを介して別のタスク、タイマ、割り込みハンドラにより信号が送信されるまで、待機します。そしてカウンタは減少し、プログラムの実行は継続されます。セマフォが指定した時間内に信号を送られない場合は、プログラムの実行は続行されますが 0 の値を返します。セマフォを待つタスクの数には制限がありません。スケジューラの規則によって、セマフォを待っているタスクのうち最も高い優先順位を持つタスクがプログラムの実行を継続します。

重要

この関数は割り込みハンドラ内から呼び出せません。

- 7.2.7 OS_CSemaRequest()

説明

信号があった場合、セマフォカウンタを減少させます。

プロトタイプ

char OS_CSemaRequest (OS_CSEMA* pCSema);

Table 7.8: OS_CSemaRequest() parameter list

戻り値

整数値:

0：失敗。セマフォに信号が送信されませんでした。

1：OK。セマフォが使用可能であり、カウンタが一回減少されました。

補足情報

セマフォのカウンタが 0 でない場合、カウンタは減少しプログラムの実行が継続されます。

カウンタが 0 の場合、OS_CSemaRequest()は待機せずセマフォカウンタを修正しません。関数はエラーを返します。

この関数は呼び出し元のタスクをブロックしないため、割り込みハンドラから呼び出される可能性があります。

- 7.2.8 OS_GetCSemaValue()

特定のセマフォのカウンタ値を返します。

プロトタイプ

int OS_GetCSemaValue (OS_SEMA* pCSema);

Table 7.9: OS_GetCSemaValue() parameter list

戻り値

セマフォのカウンタ値

- 7.2.9 OS_SetCSemaValue()

説明

特定のセマフォのカウンタ値を設定します。

プロトタイプ

OS_U8 OS_SetCSemaValue (OS_SEMA* pCSema,
OS_UINT Value);

Table 7.10: OS_SetCSemaValue() parameter list

戻り値

0: 値が設定されている場合

!= 0: エラーの場合

- 7.2.10 OS_DeleteCSema()

説明

特定のセマフォを消去します。

プロトタイプ

void OS_DeleteCSema (OS_CSEMA* pCSema);

Table 7.11: OS_DeleteCSema() parameter list

補足情報

セマフォを消去する前に、そのセマフォを待っているタスク及び後でそのセマフォに信号を出すタスクが存在しないことを確認してください。embOS のデバッグバージョンは消去されたセマフォに信号が送られた場合エラーを返します。

- Chapter 8 メールボックス

- 8.1 導入

前章では、セマフォの使用によるタスクの同期について説明しました。しかし残念なことに、

セマフォは一つのタスクから別のタスクにデータを転送することはできません。例えばバッファを介してタスク間でデータを転送する必要がある場合、バッファにアクセスするたびにリソースセマフォを使用することができます。しかしそうすることで、プログラムの効率は低下します。もう一つの大きな欠点は、割り込みハンドラがリソースセマフォを待つことを許可されていないため、割り込みハンドラからバッファにアクセスすることができないということです。一つの解決法は、グローバル変数の使用です。この場合、これら変数にアクセスするたびにあらゆる場所で割り込みを無効にしなければならないでしょう。これは可能ですが、落とし穴があります。また、タスクがバッファのなかの文字の数を含むグローバル変数をポーリングせずに文字がバッファに配置されるのを待つことも容易ではありません。もう一つの解決法は、文字がバッファに配置されるたびにイベントによってタスクに通知することです。これがリアルタイム OS でこれを行う簡単な方法、即ちメールボックスの使用です。

▪ 8.2 基本事項

メールボックスは、リアルタイムオペレーティングシステムによって管理されているバッファです。このバッファは通常のバッファと同じように動作し、メッセージと呼ばれるものを入れたり、後で取り出したりすることができます。メールボックスは、通常 FIFO(first in, first out)として動作します。したがって、通常最初に入れられたメッセージは最初に取得されます。「メッセージ」というと抽象的に聞こえるかもしれませんが、「データのアイテム」と言えば非常に単純でしょう。次のセクションで説明する一般的なアプリケーションではより明確になります。

メールボックスの数は、使用可能なメモリの量によって制限されます。

メッセージサイズ：1 <= x<=127 バイト

メッセージの数：1 <= x <= 32767

これらのメールボックスの制限は、効率的な符号化を保証するし効率的な経営を確保するためにあり、通常は問題ありません。127 バイトを超えるメッセージを処理するためには、キューを使用することができます。詳細は、147 ページのキューの章を参照してください。

▪ 8.3 代表的なアプリケーション

キーボードバッファ

ほとんどのプログラムでは、キーボードをチェックするためにタスク、ソフトウェアタイマ、割り込みハンドラのいずれかを使用しています。キーが押されたことを検出した場合、そのキ

ーはキーボードバッファとして使用されているメールボックスに格納されます。そしてそのメッセージはキーボード入力を処理するタスクによって取得されます。この場合、メッセージは通常、キーコードを保持するシングルバイトであり、したがってメッセージのサイズは 1 バイトです。

キーボードバッファの利点は、管理が非常に効率的であり、信頼性のある実証済みのコードであるため心配する必要がなく、また追加費用なしで先行入力バッファを持てることです。さらに、タスクは、バッファをポーリングしなくてもキーが押されるのを簡単に待つことができます。これは、特定のメールボックスの OS_GetMail()ルーチンを呼び出すだけで実現されます。先行入力をバッファに格納できるキーの数は、メールボックスを作成するときに定義するメールボックスバッファのサイズにのみに依存します。

シリアル I/O 用のバッファ

ほとんどの場合、シリアル I/O は割り込みハンドラの助けをかりて行われています。これらの割り込みハンドラへの通信は、メールボックスを使えば非常に簡単です。タスクプログラムと割り込みハンドラの両方がデータを同じメールボックスに格納あるいは逆に取得します。キーボードバッファと同様に、メッセージのサイズは 1 文字です。割り込みによる送信のため、タスクは OS_PutMail()または OS_PutMailCond()を使用してメールボックスに文字を配置します。新たに文字を送信できたときに起動される割り込みハンドラは OS_GetMailCond()を使用してこの文字を取得します。

割り込み駆動受信するために、新たに文字を受信したときに起動される割り込みハンドラは OS_PutMailCond()を使用してその文字をメールボックスに入れ、タスクは OS_GetMail()または OS_GetMailCond()を使って取得します。

マシンを制御するアプリケーションを持っているのと同じように、モータを制御する 1 つのタスクを持っているとお考えください。このタスクにコマンドを付けるための簡単な方法は、コマンドの構造を定義することです。メッセージのサイズはこの構造のサイズとなります。

▪ 8.4 シングルバイトメールボックス関数

多くの（ほとんど、ではないとしても）場合、メールボックスはシングルバイトのメッセージを保持して転送するためだけに使用されます。これは例えば、シリアルインターフェースを介して送受信される文字を運ぶメールボックスや、キーボードバッファとして使われるメールボックスに当てはまります。これらの場合のうちでも、特に短時間で多くのデータが転送される場合、非常に重要です。 embOS のメールボックス管理によるオーバーヘッドを最小限にする

ために、いくつかのメールボックス関数のバリエーションのうち、シングルバイトのメールボックスが利用できるものもあります。一般的な関数 OS_PutMail()、OS_PutMailCond()、OS_GetMail()、および OS_GetMailCond()はそれぞれ、1～127 バイトのサイズのメッセージを転送することができます。これらと等価のシングルバイト関数 OS_PutMail1()、OS_PutMailCond1()、OS_GetMail1()、および OS_GetMailCond1()は、管理が簡単であるためはるかに速く実行される以外は同様に機能します。メールボックスを介して、多くのシングルバイトデータを転送する場合、シングルバイトバージョンを使用することをお勧めします。ルーチン OS_PutMail1()、OS_PutMailCond1()、OS_GetMail1()、および OS_GetMailCond1()はそれぞれ等価な関数と全く同じように動作するので、別々に記載されてはいません。唯一の違いは、シングルバイトのメールボックスのみ使用できるということです。

8.5 API 関数

Table 8.1: Mailboxes API functions

8.5.1 OS_CREATEMB()

説明

新しくメールボックスを作成するマクロです。

プロトタイプ

```
void   OS_CREATEMB   (OS_MAILBOX* pMB,
unsigned   char  sizeofMsg,
unsigned   int          maxnofMsg,
void*       pMsg);)
```

Table 8.2: OS_CREATEMB() parameter list

使用例

キーボードバッファとして使用されるメールボックス:

```
OS_MAILBOX  MBKey;
char  MBKeyBuffer[6];
```

```
void InitKeyMan(void) {
/* Create mailbox, functioning as type ahead buffer */
OS_CREATEMB(&MBKey, 1, sizeof(MBKeyBuffer), &MBKeyBuffer);
}
```

Mailbox used for transferring complex commands from one task to another:

```
/*
* Example of mailbox used for transfering commands to a task
* that controls 2 motors
*/
typedef struct {
char Cmd;
int Speed[2];
int Position[2];
} MOTORCMD ;
OS_MAILBOX MBMotor;
#define MOTORCMD_SIZE 4
char BufferMotor[sizeof(MOTORCMD)*MOTORCMD_SIZE];
void MOTOR_Init(void) {
/* Create mailbox that holds commands messages */
OS_CREATEMB(&MBMotor, sizeof(MOTORCMD), MOTORCMD_SIZE, &BufferMotor);
}
```

- 8.5.2 OS_PutMail() / OS_PutMail1()

説明

事前に定義されたサイズの新しいメッセージをメールボックスに保存します。

プロトタイプ

```
void OS_PutMail (OS_MAILBOX* pMB,
const void* pMail);
void OS_PutMail1 (OS_MAILBOX* pMB,
const char* pMail);
```

Table 8.3: OS_PutMail() / OS_PutMail1() parameter list

補足情報

メールボックスが満杯である場合は、呼び出し元のタスクが中断されます。このルーチンは中断を要求する場合があるため、割り込みルーチンから呼び出してはいけません。ISR 内からメールボックスにデータを保存する必要がある場合には、
OS_PutMailCond()/OS_PutMailCond1()を使用してください。

重要

この関数は割り込みハンドラ内から呼び出せません。

使用例

Single-byte mailbox as keyboard buffer:

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];
void KEYMAN_StoreKey(char k) {
OS_PutMail1(&MBKey, &k); /* Store key, wait if no space in buffer */
}
void KEYMAN_Init(void) {
/* Create mailbox functioning as type ahead buffer */
OS_CREATEMB(&MBKey, 1, sizeof(MBKeyBuffer), &MBKeyBuffer);
}
```

- 8.5.3 OS_PutMailCond() / OS_PutMailCond1()

説明

メールボックスが新たに一つメッセージを収容可能である場合、事前にメールボックスで定義された新しいメッセージを保存します。

プロトタイプ

```
char OS_PutMailCond (OS_MAILBOX* pMB,
const void* pMail);
```

```
char   OS_PutMailCond1   (OS_MAILBOX*  pMB,
const   char*  pMail);)
```

Table 8.4: OS_PutMailCond() / OS_PutMailCond1() overview

戻り値

0: 成功。メッセージが保存されました。

1: メッセージを保存できませんでした（メールボックスが一杯です)。

補足情報

メールボックスが満杯の場合、メッセージは保存されません。

この関数は呼び出しもとのタスクを中断しないため、割り込みタスクに呼び出される可能性があります。

使用例

```
OS_MAILBOX  MBKey;
char  MBKeyBuffer[6];
char  KEYMAN_StoreCond(char  k)  {
return  OS_PutMailCond1(&MBKey,  &k);  /*  Store  key  if  space  in  buffer  */
}
```

この例は先に示したメールボックスをキーボードバッファとして扱っているサンプルプログラムとともに使用することができます。

- 8.5.4 OS_PutMailFront() / OS_PutMailFront1()

説明

事前に指定されたサイズの新しいメッセージを、メールボックスのメッセージのうち一番前に保存します。この新しいメッセージは最初に取り出されます。

プロトタイプ

```
void   OS_PutMailFront          (OS_MAILBOX*  pMB,
const   void*  pMail);
```

```
void  OS_PutMailFront1  (OS_MAILBOX*  pMB,
const  char*  pMail);
```

Table 8.5: OS_PutMailFront() / OS_PutMailFront1() parameter list

補足情報

メールボックスが満杯である場合は、呼び出し元のタスクが中断されます。このルーチンは中断を要求する場合があるため、割り込みルーチンから呼び出してはいけません。ISR 内からメールボックスにデータを保存する必要がある場合には、OS_PutMailFrontCond()/OS_PutMailFrontCond1()を使用してください。

この関数はメールボックスに送球に扱うべき“緊急の”メッセージを保存する場合に便利です。また、一般的に、メールボックスの FIFO 構造を LIFO 構造に変更する際に OS_PutMail()の代わりに使用することができます。

重要

この関数は割り込みハンドラ内から呼び出せません。

使用例

LIFO パターンのキーボードバッファとしてのシングルバイトメールボックス:

```
OS_MAILBOX  MBCmd;
char  MBCmdBuffer[6];
void  KEYMAN_StoreCommand(char  k)  {
OS_PutMailFront1(&MBCmd,  &k);  /*  Store  command,  wait  if  no  space  in  buffer*/
}
void  KEYMAN_Init(void)  {
/*  Create  mailbox  for  command  buffer  */
OS_CREATEMB(&MBCmd,  1,  sizeof(MBCmdBuffer),  &MBCmdBuffer);
}
```

- 8.5.5 OS_PutMailFrontCond() / OS_PutMailFrontCond1()

説明

メールボックスにもう一つメッセージを収容する余裕がある場合、事前に指定されたサイズの

新しいメッセージをメールボックスの一番前に保存します。新しいメッセージは最初に取り出されます。

プロトタイプ

```
char  OS_PutMailFrontCond  (OS_MAILBOX*  pMB,
const  void*  pMail);
char  OS_PutMailFrontCond1  (OS_MAILBOX*  pMB,
const  char*  pMail);)
```

Table 8.6: OS_PutMailFrontCond() / OS_PutMailFrontCond1() parameter list

戻り値

0:成功。メッセージは保存されました。

1:メッセージを保存できませんでした（メールボックスが一杯です)。

補足情報

メールボックスが満杯である場合は、メッセージは保存されません。この関数は呼び出し元のタスクを中断しないので、割り込みルーチンから呼び出され可能性があります。この関数はメールボックスに送球に扱うべき“緊急の”メッセージを保存する場合に便利です。

また、一般的に、メールボックスの FIFO 構造を LIFO 構造に変更する際に OS_PutMailCond()の代わりに使用することができます。

- 8.5.6 OS_GetMail() / OS_GetMail1()

説明

事前に指定されたサイズのメッセージをメールボックスから新たに取り出します。

プロトタイプ

```
void  OS_GetMail  (OS_MAILBOX*  pMB,
void*       pDest);
void  OS_GetMail1  (OS_MAILBOX*  pMB,
char*  pDest);
```

Table 8.7: OS_GetMail() / OS_GetMail1() parameter list

補足情報

メールボックスが空である場合、メールボックスが新たにメッセージを受け取るまでタスクは中断されます。このルーチンは中断を要求することがあるので、割り込みタスクからは呼び出せません。ISR でメールボックスからデータを取り出す必要がある場合は OS_GetMailCond/OS_GetMailCond1 を使用してください。

重要

この関数は割り込みハンドラ内から呼び出せません。

使用例

```
OS_MAILBOX  MBKey;
char  MBKeyBuffer[6];
char  WaitKey(void)  {
char  c;
OS_GetMail1(&MBKey,  &c);
return  c;
}
```

▪ 8.5.7 OS_GetMailCond() / OS_GetMailCond1()

説明

メッセージが利用可能である場合、事前に指定されたサイズのメッセージを新たにメールボックスから取り出します。

プロトタイプ

```
char   OS_GetMailCond        (OS_MAILBOX   *   pMB,
void*   pDest);
char   OS_GetMailCond1   (OS_MAILBOX   *   pMB,
char*   pDest);
```

Table 8.8: OS_GetMailCond() / OS_GetMailCond1() parameter list

戻り値

0: 成功。メッセージが取り出されました。

1: メッセージを取り出せませんでした（メールボックスが空です)。送付先は変わりません。

補足情報

メールボックスが空の場合メッセージは取り出されませんが、プログラムは続行します。この関数は呼び出しもとのタスクを中断しないので、割り込みルーチンから呼び出すこともできます。

使用例

```
OS_MAILBOX  MBKey;
/*
* If a key has been pressed, it is taken out of the mailbox and returned to caller.
* Otherwise, 0 is returned.
*/
char  GetKey(void) {
char  c  =  0;
OS_GetMailCond1(&MBKey,  &c)
return  c;
}
```

- 8.5.8 OS_GetMailTimed()

説明

メッセージが与えられた時間に使用可能である場合、事前に指定されたメッセージを新たにメールボックスから取り出します。

プロトタイプ

```
char   OS_GetMailTimed   (OS_MAILBOX*   pMB,
void*   pDest,
OS_TIME   Timeout);
```

Table 8.9: OS_GetMailTimed() parameter list

戻り値

0: 成功。メッセージが取り出されました。

1: メッセージを取り出せませんでした（メールボックスが空です)。送信先は変わりません。

補足情報

メールボックスが空の場合、メッセージは取り出せず、タスクは指定されたタイムアウトの間中断されます。タスクは、スケジュールの規則に従って、メールが指定されたタイムアウト内で利用可能になるとすぐに、またはタイムアウト値の期限が切れたあとに、実行を継続します。

重要

この関数は割り込みハンドラ内から呼び出せません。

使用例

```
OS_MAILBOX  MBKey;
/*
* If a key has been pressed, it is taken out of the mailbox and returned to caller.
* Otherwise, 0 is returned.
*/
char GetKey(void) {
char c =0;
OS_GetMailTimed(&MBKey, &c, 10) /* Wait for 10 timer ticks */
return c;
}
```

- 8.5.9 OS_WaitMail()

説明

メールが利用可能になるまで待機しますが、メールボックスからメッセージを取り出しはしません。

プロトタイプ

```
void OS_WaitMail (OS_MAILBOX* pMB);
```

Table 8.10: OS_WaitMail() parameter list

補足情報

メールボックスが空の場合、タスクはメールが利用可能になるまで中断され、でない場合には続行されます。メールが利用可能である場合、タスクはスケジューラの規則に沿って実行されますが、メールはメールボックスから取り出されません。

重要

この関数は割り込みハンドラ内からは呼び出せません。

- 8.5.10 OS_WaitMailTimed()

説明

メールが利用可能になるかタイムアウトの期限が切れるまで待ちますが、メールボックスからメッセージを取り出すことはしません。

プロトタイプ

char OS_WaitMailTimed (OS_MAILBOX* pMB, OS_TIME Timeout)

Table 8.11: OS_WaitMail() parameter list

戻り値

0: 成功。メッセージは利用可能です。

1: タイムアウト。指定されたタイムアウト時間内に使用可能なメッセージがありません。

補足情報

メールボックスが空の場合、タスクは指定されたタイムアウトの間中断されます。タスクは、スケジューラの規則に従って、メールが指定されたタイムアウト内で利用されるとすぐに、またはタイムアウト値の期限が切れた後、実行を継続します。

重要

この関数は割り込みハンドラ内から呼び出せません。

- 8.5.11 OS_ClearMB()

説明

指定されたメールボックス内の全てのメッセージを消去します。

プロトタイプ

void OS_ClearMB (OS_MAILBOX* pMB);

Table 8.12: OS_ClearMB() parameter list

使用例

```
OS_MAILBOX MBKey;
/*
* Clear keyboard type ahead buffer
*/
void ClearKeyBuffer(void) {
OS_ClearMB(&MBKey);
}
```

- 8.5.12 OS_GetMessageCnt()

説明

指定されたメールボックスの中で現在使用可能なメッセージの数を返します。

プロトタイプ

unsigned int OS_GetMessageCnt (OS_MAILBOX* pMB);

Table 8.13: OS_GetMessageCnt() parameter list

戻り値

メールボックスの中のメッセージの数。

8.5.13 OS_DeleteMB()

説明

指定されたメールボックスを消去します。

プロトタイプ

void OS_DeleteMB (OS_MAILBOX* pMB);

Table 8.14: OS_DeleteMB() parameter list

補足情報

システムを完全に動的に保つために、メールボックスを動的に作成できることが不可欠です。これは、メールボックスが不要になったときにそれを削除する方法がなければならないことも意味します。制御構造のためのバッファで使用されているメモリ及びバッファーは再利用または再割り当てすることができます。

以下のものはプログラマの責任です：

·プログラムが、削除するメールボックスを使用していないことを確認する。

·削除されるメールボックスが実際に存在することを確認する。

使用例

```
OS_MAILBOX  MBSerIn;
void  Cleanup(void)  {
OS_DeleteMB(MBSerIn);
return  0;
}
```

Chapter 9 キュー

9.1 導入

前章では、メールボックスを使用してのタスク間の通信について説明しました。メールボックスは、一定のデータサイズに限って小さなメッセージを処理することができます。

キューでは大きいメッセージや様々なサイズのメッセージを持つタスク間の通信が可能になります。

9.2 基本事項

キューは、リアルタイムオペレーティングシステムによって管理されているデータバッファと制御構造で構成されています。キューは通常のバッファと同じように動作し、メッセージと呼ばれるものを入れて、後でそれを取り出すことができます。キューは FIFO として動作します（first in, first out）。したがって、最初に入れられたメッセージは、最初に取得されます。

キューおよびメールボックスの主な違いは 3 つあります：

1.キューは、様々な大きさのメッセージを受け取ります。メッセージをキューに入れるときは、メッセージの大きさはパラメータとして渡されます。

2.キューからメッセージを取得してもメッセージはコピーされませんが、メッセージへのポインタとそのサイズが返されます。メッセージがキューに書き込まれたときデータは一度だけコピーされるため、性能は向上します。

3.取得する関数は、処理した後のすべてのメッセージを削除する必要があります。

キューの数と大きさは使用可能なメモリーの量に限られます。どんなデータ構造でもキューに書き込むことができます。メッセージサイズは固定されていません。

9.3 API 関数

9.3.1 OS_Q_Create()

説明

メッセージキューを作成・初期化します。

プロトタイプ

```
void OS_Q_Create (OS_Q* pQ,
void*pData,
OS_UINT Size);
```

Table 9.2: OS_Q_Create() parameter list

使用例

```
#define MEMORY_QSIZE 10000;
static OS_Q _MemoryQ;
```

```
static char             _acMemQBuffer[MEMORY_QSIZE];
void MEMORY_Init(void) {
OS_Q_Create(&_MemoryQ, &_acMemQBuffer, sizeof(_acMemQBuffer));
}
```

- 9.3.2 OS_Q_Put()

説明

与えられたサイズのメッセージをキューに保存します。

プロトタイプ

```
int  OS_Q_Put  (OS_Q*  pQ,
const  void*  pSrc,
OS_UINT  Size);
```

Table 9.3: OS_Q_Put() parameter list

戻り値

0: 成功。メッセージが保存されました

1: メッセージを保存できませんでした（キューが一杯です)。

補足情報

キューが満杯の場合、関数は 0 でない値を返します。このルーチンは呼び出し元のタスクを中断しないため、割り込みルーチンから呼び出すこともできます。

使用例

```
char MEMORY_Write(char* pData, int Len) {
return OS_Q_Put(&_MemoryQ, pData, Len));
}
```

- 9.3.3 OS_Q_GetPtr()

説明

キューからメッセージを取り出します。

プロトタイプ

```
int   OS_Q_GetPtr   (OS_Q*     pQ,
void**   ppData);
```

Table 9.4: OS_Q_GetPtr() parameter list

戻り値

取り出されたメッセージのサイズ。

ポインタ ppData を取り出すべきメッセージに設定します。

補足情報

キューが空の場合、呼び出しもとのタスクは、キューが新たにメッセージを受け取るまで中断されます。このルーチンは中断を要求する場合があるので、割り込みルーチンから呼び出すことはできません。変わりに OS_GetPtrCond()を使用してください。取り出されたメッセージはキューには戻されません。戻すには、メッセージが処理されたあと、OS_Q_Purge()を呼び出してください。

使用例

```
static  void  MemoryTask(void)  {
char  MemoryEvent;
int  Len;
char*  pData;
while  (1)  {
Len  =  OS_Q_GetPtr(&_MemoryQ,  &pData);        /*  Get  message       */
Memory_WritePacket(*(U32*)pData,  pData+4,  Len);   /*  Process  message  */
OS_Q_Purge(&_MemoryQ);     /*  Delete  message   */
}
}
```

- 9.3.4 OS_Q_GetPtrCond()

説明

メッセージが利用可能である場合、キューからメッセージを取り出します。

プロトタイプ

```
int   OS_Q_GetPtrCond  (OS_Q*          pQ,
void**   ppData);
```

Table 9.5: OS_Q_GetPtrCond() parameter list

戻り値

0：キューの中にの使用可能なメッセージはありません。

> 0：キューから取得されたメッセージのサイズ。取得するメッセージへのポインタ ppData を設定します。

補足情報

キューが空の場合、関数は 0 を返します。 ppData の値は定義されていません。この関数は、呼び出し元のタスクを中断することはありません。したがって、割り込みルーチンから呼び出すこともできます。メッセージが取得されても、それがキューから削除されることはありません。メッセージが処理された後に OS_Q_Purge()の呼び出しによって行ってください。

使用例

```
static void MemoryTask(void) {
char MemoryEvent;
int Len;
char* pData;
while (1) {
Len = OS_Q_GetPtrCond(&_MemoryQ, &pData); /* Check message */
if (Len > 0) {
Memory_WritePacket(*(U32*)pData, pData+4, Len); /* Process message */
OS_Q_Purge(&_MemoryQ);    /* Delete message */
} else {
DoSomethingElse();
}
}
}
```

- 9.3.5 OS_Q_GetPtrTimed()

説明

メッセージが利用可能である場合、指定された時間内にキューからメッセージを取り出します。

プロトタイプ

```
int   OS_Q_GetPtrTimed   (OS_Q*          pQ,
void**   ppData,
OS_TIME   Timeout);
```

Table 9.6: OS_Q_GetPtrTimed() parameter list

戻り値

0:キュー内に使用可能なメッセージがありません。

>0:キューの中から取り出されたメッセージのサイズ。

取り出されるべきメッセージにポインタ ppData を設定します。

補足情報

キューが空である場合、メッセージは取り出されず、タスクは指定されたタイムアウトの間中断されます。ppData の値は定義されていません。タスクはスケジューラの規則にしたがって、メッセージが指定されたタイムアウト内に使用可能になるとすぐに、またはタイムアウト値が期限切れになった後に、実行を継続します。

使用例

```
static void MemoryTask(void) {
char MemoryEvent;
int Len;
char* pData;
while (1) {
Len = OS_Q_GetPtrTimed(&_MemoryQ, &pData, 10);       /* Check message   */
if (Len > 0) {
Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
OS_Q_Purge(&_MemoryQ);     /* Delete message   */
```

```
} else {    /* Timeout              */
DoSomethingElse();
}
}
}
```

- 9.3.6 OS_Q_Purge()

説明

キューの中で最後に取り出されたメッセージを消去します。

プロトタイプ

```
void  OS_Q_Purge  (OS_Q*  pQ);
```

Table 9.7: OS_Q_Purge() parameter list

補足情報

このルーチンは、メッセージが処理された後、キューから最後のメッセージを取り出したタスクによって呼び出されなければなりません。

使用例

```
static void MemoryTask(void) {
char MemoryEvent;
int Len;
char* pData;
while (1) {
Len = OS_Q_GetPtr(&_MemoryQ, &pData);        /* Get message      */
Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
OS_Q_Purge(&_MemoryQ);    /* Delete message  */
}
}
```

- 9.3.7 OS_Q_Clear()

説明

キュー内の全てのメッセージを消去します。

プロトタイプ

void　OS_Q_Clear　(OS_Q*　pQ);

Table 9.8: OS_Q_Clear() parameter list

- 9.3.8 OS_Q_GetMessageCnt()

説明

現在キューの中にあるメッセージの数を返します。

プロトタイプ

int　OS_Q_GetMessageCnt　(OS_Q*　pQ);

Table 9.9: OS_Q_GetMessageCnt() parameter list

戻り値

キューの中にあるメッセージの数。

- 9.3.9 OS_Q_Delete()

説明

特定のキューを消去します。

プロトタイプ

void　OS_Q_Delete　(OS_Q*　pQ);

Table 9.10: OS_Q_GetMessageCnt() parameter list

補足情報

システムを動的に保つためには、キューが動的に作成されることが不可欠です。これは、キューが必要なくなったときには消去する方法がなければならないことを意味します。制御構造のためにキューに使用されたメモリおよびバッファは再利用あるいは再度割り当てできます。

以下はプログラマーの責任です:

・プログラムが消去されるキューを使用していないことを確認する。

・消去されるキューが実際に存在すること（先に作成されていること）を確認する。

使用例

```
OS_Q  QSerIn;

void  Cleanup(void) {
OS_Q_Delete(QSerIn);
}.
```

- 9.3.10 OS_Q_IsInUse()

説明

キューが実際に使用されているかどうかという情報を伝えます。

プロトタイプ

```
OS_BOOL   OS_Q_IsInUse(OS_Q*   pQ)
```

Table 9.11: OS_Q_GetMessageCnt() parameter list

戻り値

0: キューは使用されていません。

!=0: キューは使用されており消去したりクリアしたりできません。

補足情報

キューはタスクや関数で使用されている場合はクリアしたり消去したりできません。使用されている、というのはタスクや関数が一つでも実際にキューにアクセスしキューの中でデータのポインタを持っているということです。

OS_Q_IsInUse()はキューが使用されているときには動作できないため、キューをクリアしたり消去したりできるようになる前にキューの状態を確認するのに使用できます。

使用例

```
void  DeleteQ(OS_Q*  pQ) {
OS_incDI();            // Avoid  state  changes  of  the  queue  by  task  or  interrupt
//
```

```
// Wait until queue is not used
//
while (OS_Q_IsInUse(pQ) != 0) {
OS_Delay(1);
}
OS_Q_Delete(pQ);
OS_DecRI();
}
```

Chapter 10 タスクイベント

10.1 導入

タスクイベントは、もう一つのタスク間の通信手段です。セマフォやメールボックスに対して一つの、指定された受信者です。言い換えれば、タスクイベントは指定されたタスクに送信されます。タスクイベントの目的はタスクが特定のイベント（またはいくつかのイベントのうち一つ）が起こるのを待機することができるようにすることです。このタスクは、イベントが別のタスク、S/W タイマ、または割り込みハンドラによって送信されるまで非アクティブ状態にしておくことができます。イベントは、どんな方法にせよソフトウェアが知らされたものを構成することができます。例えば、入力信号の変化、タイマの期限切れ、キーを押す、文字の受信、コマンドの満了などです。

タスクはそれぞれ 1 バイト（8 ビット）のマスクを持っており、これは 8 つの異なるイベントがそれぞれのタスクに送信され、タスクはそれを区別することができることを意味します。OS_WaitEvent()を呼び出すことにより、タスクは、ビットマスクとして指定されたイベントのうち一つを待機します。いずれかのイベントが起こるとすぐに、このタスクは OS_SignalEvent()を呼び出して通知される必要があります。待機中のタスクは、その後すぐに READY 状態になります。そのタスクは、スケジューラの規則にしたがって、READY 状態にあるタスクうち優先順位の最も高いタスクすぐに活性化されます。

10.2 API 関数

Table 10.1: Events API functions

▪ 10.2.1 OS_WaitEvent()

説明

ビットマスクで指定されたイベントのうち一つを待機し、イベントが起きた後にそのイベントをクリアします。

プロトタイプ

```
char OS_WaitEvent (char EventMask);
```

Table 10.2: OS_WaitEvent() parameter list

戻り値

実際に起こった全てのイベント。

補足情報

指定されたイベントのうちのどれも送信されなかった場合、タスクは中断されます。指定されたイベントのうち最初のものがタスクを起動します。これらのイベントは別のタスク、S/W タイマ、または割り込みハンドラによって送信されます。8 ビットのイベントマスクのいずれのビットでも、対応するイベントを有効にすることができます。

使用例

```
OS_WaitEvent(3);  /* Wait for event 1 or 2 to be signaled */
```

For a further example, see OS_SignalEvent().

▪ 10.2.2 OS_WaitSingleEvent()

説明

ビットマスクに指定されたイベントのうち一つを待機し、イベントが起こった後そのイベントのみをクリアします。

プロトタイプ

```
char OS_WaitSingleEvent (char EventMask);
```

Table 10.3: OS_WaitSingleEvent() parameter list

戻り値

実際に起こったマスクされたイベント全て。

補足情報

指定されたイベントのいずれも送信されていない場合、タスクは中断されます。指定されたイベントのうち最初のものがタスクを起動します。これらのイベントは別のタスク、S/W タイマ、または割り込みハンドラによって送信されます。8 ビットのイベントマスクのいずれのビットでも、対応するイベントを有効にすることができます。マスクされていないイベントは全て変更されません。

使用例

```
OS_WaitSingleEvent(3);          /* Wait for event 1 or 2 to be signaled */
```

▪ 10.2.3 OS_WaitEvent_Timed()

説明

指定された時間指定されたイベントを待機し、イベントが実行された後はイベントメモリをクリアします。

プロトタイプ

```
char  OS_WaitEventTimed  (char  EventMask,
OS_TIME  TimeOut);
```

Table 10.4: OS_WaitEventTimed() parameter list

戻り値

指定された時間内に実際に起こったイベント。時間内にイベントを受信しなかった場合は 0 になる。

補足情報

指定されたイベントのうちのどれも送信されなかった場合、タスクは指定された時間中断されます。指定されたイベントのうち最初のものがタスクを起動します。これらのイベントは指定

されたタイムアウト時間内に別のタスク、S/W タイマ、または割り込みハンドラによって送信されます。
イベントが送信されなかった場合、タスクは指定されたタイムアウトの後、かつ全ての実際のイベントが戻されクリアされた後、アクティブ化されます。8 ビットのイベントマスクのいずれのビットでも、対応するイベントを有効にすることができます。

使用例

```
OS_WaitEvent_Timed(3, 10); /* Wait for event 1 or 2 to be signaled within 10 ms */
```

▪ 10.2.4 OS_WaitSingleEventTimed()

説明

指定された時間指定されたイベントを待機します。イベントが起こった後は、そのイベントのみがクリアされます。

プロトタイプ

```
char OS_WaitSingleEventTimed (char EventMask,
OS_TIME TimeOut);
```

Table 10.5: OS_WaitSingleEventTimed() parameter list

戻り値

指定された時間内に実際に起こった、マスクされたイベント。時間内にマスクされたイベントが送信されなかった場合は 0 を返す。

補足情報

指定されたイベントのうちのどれも使用可能でなかった場合、タスクは指定された時間中断されます。指定されたタイムアウト時間内に別のタスク、S/W タイマ、または割り込みハンドラによってイベントが送信された場合、指定されたイベントのうち最初のものがタスクを起動します。
イベントが送信されなかった場合、タスクは指定されたタイムアウトの後、かつ関数が 0 を返した後、アクティブ化されます。8 ビットのイベントマスクのいずれのビットでも、対応するイベントを有効にすることができます。マスクされていないイベントは変更されません。

使用例

```
OS_WaitSingleEventTimed(3, 10); /* Wait for event 1 or 2 to be
signaled within 10 ms */
```

- 10.2.5 OS_SignalEvent()

説明

指定さイベントを送信します。

プロトタイプ

```
void OS_SignalEvent (char Event,
OS_TASK* pTask);
```

Table 10.6: OS_SignalEvent() parameter list

補足情報

指定されたタスクがイベントの中の一つを待機している場合、そのタスクは READY 状態になりスケジューラの規則にしたがってアクティブ化されます。

使用例

シリアル入力とキーボードを扱い、キーボード（EVENT_KEYPRESSED）またはシリアルインタフェース（EVENT_SERIN）を介して受信される文字を待機するタスク:

```
/*
* Just a small demo for events
*/
#define EVENT_KEYPRESSED (1)
#define EVENT_SERIN (2)
OS_STACKPTR int Stack0[96];          // Task stacks
OS_TASK TCB0;   // Data area for tasks (task control blocks)
void Task0(void) {
OS_U8 MyEvent;
while(1)
```

```
MyEvent = OS_WaitEvent(EVENT_KEYPRESSED | EVENT_SERIN)
if (MyEvent & EVENT_KEYPRESSED) {
/* handle key press          */
}
if (MyEvent & EVENT_SERIN) {
/* Handle serial reception   */
}
}
}
void TimerKey(void) {
/* More code to find out if key has been pressed          */
OS_SignalEvent(EVENT_SERIN, &TCB0); /* Notify Task that key was pressed */
}
void InitTask(void) {
OS_CREATETASK(&TCB0, 0, Task0, 100, Stack0);          /* Create Task0 */
}
```

タスクが単にキーが押されるのを待機しているだけだとしたら、OS_GetMail()を呼び出せばいいでしょう。そしてタスクはキーが押されるまで非アクティブ化されるでしょう。この例のようにタスクが複数のメールボックスを扱う必要がある場合、イベントがよい選択肢となります。

- 10.2.6 OS_GetEventsOccurred()

説明

指定されたタスクのために起きたイベントのリストを返します。

プロトタイプ

```
char OS_GetEventsOccurred (OS_TASK* pTask);
```

Table 10.7: OS_GetEventsOccured() parameter list

戻り値

実際に起こったイベントのイベントマスク。

補足情報

この関数を呼び出すことによって、実際のイベントが送信されたままの状態になります。イベントメモリはクリアされません。この方法は、タスクがどのイベントが送信されたかを判断する方法の一つです。イベントが送信されていない場合タスクは中断されません。

- 10.2.7 OS_ClearEvents()

説明

実際のイベントの状態を返し、その後指定されたタスクのイベントをクリアします。

プロトタイプ

```
char  OS_ClearEvents  (OS_TASK*  pTask);
```

Table 10.8: OS_ClearEvents() parameter list

戻り値

クリアされる前に実際に送信されたイベント

- Chapter 11 イベントオブジェクト

- 11.1 導入

イベントオブジェクトは、もう一つのオブジェクト間の通信あるいは同期です。 タスクイベントと違い、イベントオブジェクトはどのタスクにも所有されていない独立のオブジェクトです。イベントオブジェクトの目的は、一つあるいは複数のタスクが指定されたイベントが起こるのを待機することを可能にすることです。タスクはイベントが別のタスク、S/W タイマ、あるいは割り込みハンドラによって設定されるまで中断できます。イベントは、ソフトウェアがどのような方法であれ知らされているものになりえます。例えば、入力信号の変化、タイマの満了、キーを押す、文字の受信、または完全なコマンドなどです。

タスクイベントと異なり、シグナリング関数は、どのタスクがイベントを待機しているのかを知る必要はありません。

- 11.2 API 関数

Table 11.1: Event object API functions

▪ 11.2.1 OS_EVENT_Create()

説明

イベントオブジェクトを作成しリセットします。

プロトタイプ

void OS_EVENT_Create （OS_EVENT* pEvent）

Table 11.2: OS_EVENT_Create() parameter list

補足情報

イベントオブジェクトは使用できるようになる前に、まず OS_EVENT_Create()を呼び出して作成される必要があります。t 作成時には、イベントはノンシグナル状態に設定され、待機しているタスクのリストは消去されます。よって、OS_EVENT_Create()は既に作成されたイベントオブジェクトのために呼び出すことができません。 embOS のデバッグバージョンでは、指定されたイベントオブジェクトが既に作成されているかどうかを確認し、OS_EVENT_Create()の呼び出しにの前に既にイベントオブジェクトが作成されている場合はエラーコード OS_ERR_2USE_EVENTOBJ で OS_Error() を呼び出します。

使用例

```
OS_EVENT _HW_Event;
OS_EVENT_Create(&HW_Event);        /* Create and initialize event object */
```

▪ 11.2.2 OS_EVENT_Wait()

説明

イベントを待機し、イベントが送信されていない場合は呼び出し元のタスクを中断します。

プロトタイプ

void OS_EVENT_Wait （OS_EVENT* pEvent）

Table 11.3: OS_EVENT_Wait() parameter list

補足情報

指定されたイベントオブジェクトが既に設定されている場合、呼び出し元のタスクはイベントをリセットし実行を継続します。

指定されたイベントオブジェクトが設定されていない場合、呼び出し元のタスクはイベントオブジェクトに信号が送られるまで中断されます。タスクが再開するときにはイベントはリセットされません。

pEvent は、OS_EVENT_Wait()を呼び出す前に既に作成されているイベントオブジェクトをアドレス指定する必要があります。embOS のデバッグバージョンでは pEvent が有効なイベントオブジェクトをアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

重要

この関数は割り込みハンドラあるいはソフトウェアタイマからは呼び出せません。

使用例

```
OS_EVENT_Wait(&_HW_Event);        // Wait for event object
OS_EVENT_Reset(&_HW_Event);       // Reset the event
```

▪ 11.2.3 OS_EVENT_WaitTimed()

説明

イベントを待機し、イベントが送信されない場合は指定された時間呼び出し元のタスクを中断します。

プロトタイプ

```
char OS_EVENT_WaitTimed (OS_EVENT* pEvent, OS_TIME Timeout)
```

Table 11.4: OS_EVENT_WaitTimed() parameter list

戻り値

0 成功。イベントは指定された時間内に送信されました。

1 イベントが送信されずタイムアウトが発生しました。

補足情報

指定されたイベントオブジェクトが既に設定されている場合、呼び出し元のタスクはイベントをリセットし実行を継続します。指定されたイベントオブジェクトが設定されていない場合、呼び出し元のタスクは、イベントオブジェクトが信号を受け取るかタイムアウト時間が期限切れになるまで中断されます。イベントオブジェクトが指定されたタイムアウト時間内に信号を受け取った場合、イベントはリセットされ関数はタイムアウト結果なしの値を返します。

pEvent は OS_EVENT_WaitTimed()の呼び出しの前に作成された実在するイベントオブジェクトにアドレス指定する必要があります。embOS のデバッグバージョンでは pEvent が有効なイベントオブジェクトにアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

重要

この関数は割り込みハンドラあるいはソフトウェアタイマからは呼び出せません。

使用例

```
if (OS_EVENT_WaitTimed(&_HW_Event, 10) == 0) {
/* event was signaled within timeout time, handle event */
...
} else {
/* event was not signaled within timeout time, handle timeout */
...
}
```

▪ 11.2.4 OS_EVENT_Set()

説明

イベントオブジェクトを指定された状態に設定、あるいはイベントオブジェクトで待機しているタスクを再開します。

プロトタイプ

```
void OS_EVENT_Set (OS_EVENT* pEvent)
```

Table 11.5: OS_EVENT_Set() parameter list

補足情報

イベントオブジェクトを待機しているタスクが存在しない場合、イベントオブジェクトは指定された状態に設定されます。少なくとも一つのタスクが既にイベントオブジェクトを待機している場合、待機中のタスクは全て再開されイベントオブジェクトは通知された状態には設定されません。pEvent は OS_EVENT_Create()の呼び出によって既に作成された実在するイベントオブジェクトにアドレス指定する必要があります。embOS のデバッグバージョンでは pEvent が有効なイベントオブジェクトにアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

使用例

OS_EVENT_Set()関数の使用法は“Examples”で説明しています。

- 11.2.5 OS_EVENT_Reset()

説明

指定されたイベントオブジェクトをノンシグナル状態にリセットします。

プロトタイプ

void OS_EVENT_Reset (OS_EVENT* pEvent)

Table 11.6: OS_EVENT_Reset() parameter list

補足情報

pEvent は、OS_EVENT_Create()の呼び出しによって既に作成されているイベントオブジェクトにアドレス指定する必要があります。embOS のデバッグバージョンでは pEvent が有効なイベントオブジェクトにアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

使用例

```
OS_EVENT_Reset(&_HW_Event); /* Reset event object to non-signaled state */
```

- 11.2.6 OS_EVENT_Pulse()

説明

イベントオブジェクトに信号を出して大気中のタスクを再開し、その後イベントオブジェクトをノンシグナル状態にリセットします。

プロトタイプ

void　OS_EVENT_Pulse　(OS_EVENT*　pEvent);

Table 11.7: OS_EVENT_Pulse() parameter list

補足情報

イベントオブジェクトで待機中のタスクがあれば、そのタスクは再開されます。イベントオブジェクトはノンシグナル状態のままになります。embOS のデバッグバージョンでは、pEvent が有効なイベントオブジェクトにアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

- 11.2.7　OS_EVENT_Get()

説明

イベントオブジェクトの状態を返します。

プロトタイプ

unsigned　char　OS_EVENT_Get　(OS_EVENT*　pEvent);

Table 11.8: OS_EVENT_Get() parameter list

戻り値

0: イベントオブジェクトは信号が送られた状態に設定されていません。

1: イベントオブジェクトは信号が送られた状態に設定されています。

補足情報

この関数の呼び出しによって、イベントオブジェクトの実際の状態は変更されません。pEvent は、OS_EVENT_Create()の呼び出しによって既に作成されているイベントオブジェクトにアドレス指定する必要があります。embOS のデバッグバージョンでは pEvent が有効なイベントオブジェクトにアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード

OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

▪ 11.2.8 OS_EVENT_Delete()

説明

イベントオブジェクトを消去します。

プロトタイプ

void OS_EVENT_Delete (OS_EVENT* pEvent);

Table 11.9: OS_EVENT_Delete() parameter list

補足情報

システムを完全に動的に保つには、イベントオブジェクトが動的に作成できる必要があります。これは、イベントオブジェクトが不要になったときに消去する方法が存在しなければならないことを意味します。よって、イベントオブジェクトの制御構造に使用されていたメモリは再利用あるいは再度割り当てできます。

以下のことはユーザーの責任です:

· プログラムが消去されるイベントオブジェクトを使用していないこと。

· 消去されるイベントオブジェクトが実際に存在する（事前に作成されている）こと。

· .イベントオブジェクトが消去されたときそれを待機しているタスクが存在しないこと。

pEvent は、OS_EVENT_Create()の呼び出しによって既に作成されているイベントオブジェクトにアドレス指定する必要があります。

embOS のデバッグバージョンでは、pEvent が有効なイベントオブジェクトにアドレス指定しているかどうかを確認し、エラーの場合にはエラーコード OS_ERR_EVENT_INVALID で OS_Error()を呼び出します。

消去されたイベントオブジェクトで待機しているタスクが存在する場合、embOS のデバッグバージョンはエラーコード OS_ERR_EVENT_DELETE で OS_Error()を呼び出します。

問題が発生するのを避けるため、イベントオブジェクトを一般のアプリケーション内で消去しないで下さい。

▪ 11.3 イベントオブジェクトの使用例

この章ではアプリケーション内でのイベントオブジェクトの使用法の一部を紹介します。

- 11.3.1 イベントオブジェクトによる割り込みからのタスクのアクティブ化

下記のサンプルコードは、タスクをアクティブ化するために ISR ハンドラから信号を送られたイベントオブジェクトの使用法を紹介します。.待機中のタスクは待機が終了した後イベントをリセットする必要があります。

```
static OS_EVENT _HW_Event;

/*************************************************************
*
*           _ISRhandler
*/
static void _ISRhandler(void) {
//
// Perform some simple & fast processing in ISR          //
//
...
//
// Wake up task to do the rest of the work
//
OS_EVENT_Set(&_Event);
}


/*************************************************************
*
*           _Task
*/
static void _Task(void) {
while (1) {
OS_EVENT_Wait(&_Event);
OS_EVENT_Reset(&_Event);
//
// Do the rest of the work (which has not been done in the ISR)
//
```

}

}

- 11.3.2 一つのイベントオブジェクトを使用しての複数のタスクのアクティブ化

下記のサンプルプログラムで一つのイベントオブジェクトで複数のタスクを同期する方法を紹介します。このサンプルプログラムは embOS に付属し“Application” または“Samples”フォルダにあります。

```
/*********************************************************************
* SEGGER MICROCONTROLLER SYSTEME GmbH
* Solutions for real time microcontroller applications
**********************************************************************
File        : Main_EVENT.c
Purpose : Sample program for embOS using EVENT object
--------- END-OF-HEADER -------------------------------------------*/

#include "RTOS.h"


OS_STACKPTR int StackHP[128], StackLP[128]; /* Task stacks */
OS_TASK TCBHP, TCBLP;  /* Task-control-blocks */

/*********************************************************************/

/****** Interface to HW module ***************************************/

void HW_Wait(void);
void HW_Free(void);
void HW_Init(void);
```

```
/*********************************************************************/

/****** HW module ***************************************************/

OS_STACKPTR int _StackHW[128];     /* Task stack  */
OS_TASK _TCBHW;              /* Task-control-block  */

/****** local data **************************************************/

static OS_EVENT _HW_Event;

/****** local functions *********************************************/
static void _HWTask(void) {
/* Initialize HW functionallity             */
OS_Delay(100);
/* Init done, send broadcast to waiting tasks */
HW_Free();
while (1) {
OS_Delay (40);
}
}

/****** global functions ********************************************/
void HW_Wait(void) {
OS_EVENT_Wait(&_HW_Event);
}

void HW_Free(void) {
OS_EVENT_Set(&_HW_Event);
}

void HW_Init(void) {
OS_CREATETASK(&_TCBHW, "HWTask", _HWTask, 25, _StackHW);
```

```
OS_EVENT_Create(&_HW_Event);
}
/*********************************************************************/

/****** Main application ********************************************/

static void HPTask(void) {
HW_Wait();          /* Wait until HW module is set up */
while (1) {
OS_Delay (50);
}
}

static void LPTask(void) {
HW_Wait();          /* Wait until HW module is set up */
while (1) {
OS_Delay (200);
}
}

/*********************************************************************
*
*          main
*
*********************************************************************/

int main(void) {
OS_IncDI();         /* Initially disable interrupts  */
OS_InitKern();      /* Initialize OS      */
OS_InitHW();        /* Initialize Hardware for OS           */
HW_Init();          /* Initialize HW module       */
/* You need to create at least one task before calling OS_Start() */
OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
```

```
OS_CREATETASK(&TCBLP, "LP Task", LPTask, 50, StackLP);
OS_SendString("Start project will start multitasking !\n");
OS_Start();          /* Start multitasking          */
return 0;
}
```

- Chapter 12　　ヒープ型のメモリ管理

- 導入

ANSI C では、いくつかの基本的な動的メモリ管理関数を提供しています。これらは、malloc、free、realloc です。

残念ながらこれらのルーチンは、特別なスレッドセーフな実装がコンパイラに依存するランタイムライブラリに存在しない限り、スレッドセーフではありません。これらは 1 つのタスクからのみ、または順番に呼び出された場合複数のタスクで使用できます。よって、embOS はこれらのルーチンのタスクセーフなバージョンを提供します。これらのバージョンは、ANSI で対応するものと同じ名前を持っていますが、OS_で既に固定されており、OS_malloc（）OS_free（)、OS_realloc（）と呼ばれます。 embOS が提供するスレッドセーフなバージョンは、標準 ANSI ルーチンを使用していますが、呼び出しはリソースセマフォを使用してシリアル化されることを保証します。

ヒープメモリ管理が特定の CPU 用の標準 C ライブラリでサポートされていない場合、embOS ヒープメモリ管理は実装されません。

ヒープ型のメモリ管理は embOS ライブラリの一部です。アプリケーションに参照されていない場合（すなわちアプリケーションがメモリ管理 API 関数をどれも使用しない場合)、リソースを使用しません。

他にもこれらのルーチンに問題があることを覚えておいてください。: この関数に使用されるメモリ（ヒープとして知られている）は細分化する可能性があります。メモリの合計量が十分であるような状況になることもありますが、割り当て要求を満たすための単一ブロック内の使用可能なメモリは十分ではありません。

- 12.2 API 関数

Table 12.1: Heap type memory manager API functions

Chapter 13　固定ブロックサイズメモリプール

13.1 導入

固定ブロックサイズのメモリプールには、メモリの固定サイズのブロックの特定の番号が含まれています。プールのメモリの位置、各ブロックのサイズ、ブロックの数は、OS_MEMF_CREATE()関数の呼び出しによって実行時にアプリケーションによって設定されます。固定メモリプールの利点は、非常に短い決められた時間内に任意のタスクの中からメモリのブロックを割り当てることができるということです。

13.2 API 関数

固定ブロックサイズのメモリプール API 関数は全て OS_MEMF_によって事前に固定されています。

Table 13.1: Memory pools API functions

13.2.1 OS_MEMF_Create()

説明

固定ブロックサイズメモリプールを作成し、初期化します。

プロトタイプ

```
void　OS_MEMF_Create　(OS_MEMF*　pMEMF,
void*　pPool,
OS_UINT　NumBlocks,
OS_UINT　BlockSize);
```

Table 13.2: OS_MEMF_Create() parameter list

補足情報

OS_MEMF_SIZEOF_BLOCKCONTROL は、制御およびデバッグの目的で使用されるバイト数を示します。これは、リリースまたはスタックチェックビルドでは 0 であることが保証されています。メモリプールを使用する前に、作成する必要があります。ライブラリのデバッグバージョンは、作成・削除されたメモリプールを追跡します。この機能はリリ

ースおよびスタックチェックのバージョンにはありません。
ブロックの最大数と最大ブロックサイズは、16 ビット CPU 用では 32768 用で 32 ビット CPU 用では 2147483648 です。

▪ 13.2.2 OS_MEMF_Delete()

説明
固定ブロックサイズメモリプールを消去します。消去された後は、メモリプールとその中のメモリブロックは使用できません。

プロトタイプ
```
void  OS_MEMF_Delete  (OS_MEMF*  pMEMF);
```

Table 13.3: OS_MEMF_Delete() parameter list

補足情報
このルーチンは、完全性のために提供されています。メモリプールを動的に作成・削除する必要はありませんので、大部分のアプリケーションでは使用されません。
ほとんどのアプリケーションでは、それは静的メモリプールのデザインであることが望まれます。メモリプールは、起動時（OS_Start()よりも前）に作成され、削除されることはありません。ライブラリのデバッグバージョンでは、メモリプールが削除されたと記録します。

▪ 13.2.3 OS_MEMF_Alloc()

説明
メモリブロックの割り当てを要求します。メモリのブロックが使用可能になるまで待機します。

プロトタイプ
```
void*  OS_MEMF_Alloc  (OS_MEMF*  pMEMF,
int          Purpose);
```

Table 13.4: OS_MEMF_Alloc() parameter list

戻り値
割り当てられたブロックへのポインタ。

補足情報
プール内に空きメモリブロックが存在しない場合、呼び出し元のタスクはメモリブロックが使用可能になるまで中断されます。取得したポインタはメモリブロックを解放するパラメータとして OS_MEMF_Release()に渡されなければなりません。ポインタを変更することはできません。

▪ 13.2.4 OS_MEMF_AllocTimed()

説明
メモリブロックの割り当てを要求します。メモリのブロックが使用可能になるかタイムアウトが期限切れになるまで待機します。

プロトタイプ
```
void* OS_MEMF_AllocTimed (OS_MEMF* pMEMF,
OS_TIME Timeout,
int Purpose);
```

Table 13.5: OS_MEMF_AllocTimed()

戻り値
!=割り当てられたブロックへの NULL ポインタ
ブロックが割り当てられていない場合は NULL を返します。

補足情報
プール内に空きメモリブロックが存在しない場合、呼び出し元のタスクは、メモリブロックが使用可能になるかタイムアウトが期限切れになるまで中断されます。取得したポインタはメモリブロックを解放するパラメータとして OS_MEMF_Release()に渡されなければなりません。ポインタを変更することはできません。

13.2.5 OS_MEMF_Request()

説明

メモリブロックの割り当てを要求します。どのような場合においても実行を継続します。

プロトタイプ

```
void*  OS_MEMF_Request  (OS_MEMF*  pMEMF,
int          Purpose);
```

Table 13.6: OS_MEMF_Request() parameter list

戻り値

!=割り当てられたブロックへの NULL ポインタ。

割り当てられたブロックが無い場合は NULL を返します。

補足情報

呼び出し元のタスクは OS_MEMF_Request()の呼び出しによって中断されることはありません。取得したポインタはメモリブロックを解放するパラメータとして OS_MEMF_Release()に渡されなければなりません。ポインタを変更することはできません。

13.2.6 OS_MEMF_Release()

説明

事前に割り当てられたメモリブロックを開放します。

プロトタイプ

```
void  OS_MEMF_Release  (OS_MEMF*  pMEMF,
void*       pMemBlock);
```

Table 13.7: OS_MEMF_Release() parameter list

補足情報

pMemBlock ポインタは、上記のいずれかの検索機能から配信されたものでなければなりません。ポインタは割り当てと解放との間に変更することはできません。メモリブロック

はプールからメモリブロックを待っている他のタスクで使用可能になります。いずれかのタスクが固定メモリブロックを待っている場合、それはスケジューラの規則に従って起動されます。

▪ 13.2.7 OS_MEMF_FreeBlock()

説明

事前に割り当てられたメモリブロックを開放します。メモリプールは表示される必要はありません。

プロトタイプ

void OS_MEMF_FreeBlock (void* pMemBlock);

Table 13.8: OS_MEMF_FreeBlock() parameter list

補足情報

pMemBlock ポインタはフォームに記載された検索機能から配信されたものでなければなりません。ポインタが割り当てと解放との間に変更することはできません。この関数は、OS_MEMF_Releas()の代わりに使用することができます。この関数は、1 つのパラメーターしか必要ないという利点があります。 embOS 自体は、関連付けられたメモリプールを見つけるでしょう。メモリブロックは、プールからメモリブロックを待っている他のタスクで使用可能になります。いずれかのタスクが固定メモリブロックを待っている場合、それはスケジューラの規則に従って起動されます。

▪ 13.2.8 OS_MEMF_GetNumBlocks()

説明

プール内の使用可能な全メモリブロックの数を検出するインフォメーションルーチン。

プロトタイプ

int OS_MEMF_GetNumFreeBlocks (OS_MEMF* pMEMF);

Table 13.9: OS_MEMF_GetNumBlocks() parameter list

戻り値
指定されたメモリプールにあるブロックの数を返します。メモリプールの作成時パラメータとして指定された値です。

▪ 13.2.9 OS_MEMF_GetBlockSize()

説明
プール内のあるメモリブロックのサイズを検出するインフォメーションルーチン。「

プロトタイプ
int OS_MEMF_GetBlockSize (OS_MEMF* pMEMF);

Table 13.10: OS_MEMF_GetBlockSize() parameter list

戻り値
指定したメモリプール内の 1 つのメモリブロックのサイズ（バイト単位)。これは、メモリプールが作成されたパラメータの値です。

▪ 13.2.10 OS_MEMF_GetNumFreeBlocks()

説明
プール内の空きメモリブロックの数を検出するインフォメーションルーチン。

プロトタイプ
int OS_MEMF_GetNumFreeBlocks (OS_MEMF* pMEMF);

Table 13.11: OS_MEMF_GetNumFreeBlocks() parameter list

戻り値
指定されたメモリプール内で実際に使用可能な空のブロックの数。

- 13.2.11 OS_MEMF_GetMaxUsed()

説明

プールの作成以降に同時に使用されたプール内のメモリブロックの量を調べるためのインフォメーションルーチン。

プロトタイプ

int　OS_MEMF_GetMaxUsed　(OS_MEMF*　pMEMF);

Table 13.12: OS_MEMF_GetMaxUsed() parameter list

戻り値

プールの作成以降に同時に使用された特定のメモリプール内のブロックの最大数。

- 13.2.12 OS_MEMF_IsInPool()

説明

メモリブロック参照ポインタが指定したメモリプールに属しているかどうかを検討するインフォメーションルーチン。

プロトタイプ

char　OS_MEMF_IsInPool　(OS_MEMF*　pMEMF,

void*　pMemBlock);

Table 13.13: OS_MEMF_IsInPool() parameter list

戻り値

0: ポインタはメモリプールにありません

1: ポインタがプールにあります。

- Chapter 14　　スタック

14.1 導入

スタックは、一時的な記憶だけでなく、関数呼び出し、パラメータ、ローカル変数のリターンアドレスを格納するために使用されるメモリ領域です。割り込みルーチンでも、CPUが割り込み関数のための独立したスタックを持っている場合を除いて、リターンアドレス及びフラグレジスタを保存するスタックを使用します。お使いのプロセッサのスタックの詳細については、embOS 説明書の CPU&コンパイラマニュアルを参照してください。"通常の"シングルタスクプログラムは、厳密に 1 つのスタックを必要とします。マルチタスクシステムでは、それぞれのタスクが自身のスタックを持っている必要があります。スタックは、最悪の場合のネスティングにおけるルーチンのスタック使用量の合計により決定される最小サイズを持っているある必要があります。スタックが小さすぎる場合、スタックのために予約されていないメモリのセクションは上書きされ、重大なプログラム障害が発生する可能性が非常に高くなります。 embOS は、スタックサイズ（及び、可能であれば、デバッグバージョンで割り込みスタックサイズ）を監視し、スタックオーバーフローを検出した場合、故障ルーチン OS_Error()を呼び出します。しかし、embOS はスタックオーバーフローを確実に検出することはできません。

スタックが必要以上に大きく定義されていても傷つくことはありませんが、メモリを無駄にしています。スタックオーバーフローを検出するために、embOS のデバッグとスタックチェックビルドでは、スタックが作成されたときに制御文字を使用してスタックに記入し、タスクが非アクティブ化されるたびにその制御文字をチェックします。オーバーフローが検出された場合、OS_Error()が呼び出されます。

14.1.1 システムスタック

embOS がコントロールを引き継ぐ前（OS_Start()がコールされる前に）に、プログラムは、いわゆるシステムスタックを使用しています。これは、この CPU が利用する embOS でないプログラムが使用するのと同じスタックです。OS_Start()の呼び出しによって embOS スケジューラにコントロールを転送した後、次の目的で実行されているタスクが存在しない場合システムスタックが使用されます。

·embOS スケジューラ

·embOS ソフトウェアタイマー（およびコールバック）。

お使いのシステムスタックに必要なサイズに関する詳細については、embOS マニュアルの CPU&コンパイラ細目を参照してください。

14.1.2 タスクスタック

embOS の各タスクには独立したスタックがあります。このスタックの位置とサイズはタスク作成時に定義されています。タスクスタックの最小サイズは CPU やコンパイラに大きく依存します。詳細については、embOS マニュアルの CPU&コンパイラマニュアルを参照してください。

14.1.3 割り込みスタック

マルチタスク環境でのスタックサイズを削減するために、プロセッサの一部は、割り込みサービスルーチン（ハードウェア割り込みスタックと呼ぶ）用に特定のスタック領域を使用します。割り込みスタックが存在しない場合は、各タスクのスタックに割込みサービスルーチンのスタック要件を追加する必要があります。

CPU がハードウェア割り込みスタックをサポートしていない場合でも、embOS は、その割り込みサービス･ルーチンの冒頭で OS_LeaveIntStack()OS_EnterIntStack()を呼び出し、最後に OS_EnterIntStack()を呼び出すことによって、割り込みの個別のスタックをサポートすることができます。CPU がすでにハードウェア割り込みスタックをサポートしている、もしくは独立した割り込みスタックがまったくサポートされていない場合、これらの関数呼び出しは空のマクロとして実装されています。

お使いの特定の CPU を利用しているコードが割り込みスタックをサポートしている別の CPU または embOS の新しいバージョンでスタックサイズを減少させるかもしれないので、他に利点がなかったとしても OS_EnterIntStack()と OS_LeaveIntStack()を使用することをお勧めします。割り込みスタックの詳細については、embOS マニュアルの CPU&コンパイラマニュアルを参照してください。

14.2 API 関数

Table 14.1: Stacks API functions

14.2.1 OS_GetStackBase()

説明

タスクスタックのベースへのポインタを返します。

プロトタイプ

OS_STACKPTR* OS_GetStackBase (OS_TASK* pTask);

Table 14.2: OS_GetStackBase() parameter list

戻り値

タスクスタックのベースアドレスへのポインタ。

補足情報

この関数は、スタックチェックビルドのみがこれらのタスクに使用されるスタック領域を初期化するので、embOS のデバッグとスタックチェックビルドでのみ使用可能です。

使用例

```
void CheckSpace(void) {
printf("Addr Stack[0] %x", OS_GetStackBase(&TCB[0]);
OS_Delay(1000);
printf("Addr Stack[1] %x", OS_GetStackBase(&TCB[1]);
OS_Delay(1000);
}
```

▪ 14.2.2 OS_GetStackSize()

説明

タスクスタックのサイズを返します。

プロトタイプ

int OS_GetStackSize (OS_TASK* pTask);

Table 14.3: OS_GetStackSize() parameter list

戻り値
タスクスタックのサイズをバイト単位で返します。

補足情報
この関数は、スタックチェックビルドのみがこれらのタスクに使用されるスタック領域を初期化するので、embOS のデバッグとスタックチェックビルドでのみ使用可能です。

使用例
```
void CheckSpace(void) {
printf("Size Stack[0] %d", OS_GetStackSize(&TCB[0]);
OS_Delay(1000);
printf("Size Stack[1] %d", OS_GetStackSize(&TCB[1]);
OS_Delay(1000);
}
```

▪ 14.2.3 OS_GetStackSpace()

説明
タスクスタックの未使用部分を返します。

プロトタイプ
```
int OS_GetStackSpace (OS_TASK* pTask);
```

Table 14.4: OS_GetStackSpace() parameter list

戻り値
タスクスタックの未使用部分をバイト単位で返します。

補足情報
最悪の場合のネスティングを計算するのにはかなりの時間がかかる上、計算自体が困難であるため、ほとんどの場合タスクが必要とするスタックサイズを簡単に計算することはできません。
ただし、スタック上の未使用バイト数を返す関数 OS_GetStackSpace()を使って必要なス

タックサイズを計算することができます。スペースがたくさん残っている場合は、このスタックのサイズを減らし、逆に増やすこともできます。
この関数は、スタックチェックビルドのみがこれらのタスクに使用されるスタック領域を初期化するので、embOS のデバッグとスタックチェックビルドでのみ使用可能です。

重要
このルーチンはスタックの変更されたバイト数を検出することしかできないので、スタックの空き領域の量を確実に検出することはできません。残念ながら、レジスタストレージやローカル変数に使われる領域は必ずしも変更されるわけではありません。殆どの場合、このルーチンはスタックのバイト数を性格に検出しますが、疑わしい場合は、スタックのバイト数に関しては大目に見るか、割り当てられたスタック領域が十分であることを保証する他の手段を使ってください。

使用例
```
void CheckSpace(void) {
printf("Unused Stack[0] %d", OS_GetStackSpace(&TCB[0]);
OS_Delay(1000);
printf("Unused Stack[1] %d", OS_GetStackSpace(&TCB[1]);
OS_Delay(1000);
}
```

- 14.2.4 OS_GetStackUsed()

説明
タスクスタックの使用されている部分を返します。

プロトタイプ
```
int OS_GetStackUsed (OS_TASK* pTask);
```

Table 14.5: OS_GetStackUsed() parameter list

戻り値

タスクスタックの使用されている部分をバイト数で返す。

補足情報

最悪の場合のネスティングを計算するのにはかなりの時間がかかることと、計算自体が難しいことから、ほとんどの場合、タスクに必要なスタックサイズは簡単に計算することはできません。ですが、必要なスタックサイズは、スタックの使用されたバイト数を返すOS_GetStackUsed()関数を使用することで計算することができます。多くの領域が残っている場合、このスタックのサイズを減らしたり逆に増やしたりすることもできます。
この関数は、embOS のデバッグ及びスタックチェックビルドでのみ使用可能です。これらのビルドのみがタスクで使用されたスタック領域を初期化することができるからです。

重要

このルーチンは唯一、スタックで変更されたバイト数を検出することが可能ですが、使用するスタック領域の量を確実に検出することはできません。残念なことに、レジスタストレージまたはローカル変数のために使用されるスペースは常に変更されるわけではありません。このルーチンは殆どの場合スタックバイトの正しい量を検出しますが、怪しい場合には、スタックスペースに関しては大目に見るか、または割り当てられたスタック領域が十分であるかどうか確認するために他の手段を使用してください。

使用例

```
void CheckSpace(void) {
printf("Used Stack[0] %d", OS_GetStackUsed(&TCB[0]);
OS_Delay(1000);
printf("Used Stack[1] %d", OS_GetStackUsed(&TCB[1]);
OS_Delay(1000);
}
```

- Chapter 15　割り込み

この章では、embOS を利用して割り込みサービスルーチン（ISR）を使用する方法につい

て説明します。ご使用の CPU とコンパイラについては、具体的な詳細は embOS マニュアルの CPU&コンパイラさいめマニュアルをご参照ください。

▪ 15.1 割り込みとは

割り込みとは、ハードウェアによってプログラムが遮られることを言います。割り込みが起こると、CPU はレジスタを保存し、割り込みサービスルーチン(ISR)と呼ばれるサブルーチンを実行します。ISR が終了すると、プログラムは READY 状態にあるもののうち最も優先順位の高いタスクを返します。標準的な割り込みはマスクすることが可能、つまり CPU が「割り込みを無効にする」という指示を出して無効化されない限り、いつでも起こりえます。ISR はネスト可能、即ち他の ISR の中で認識され実行することができます。
割り込みルーチンを使用することには様々な利点があります。割り込みルーチンは、入力のステータス変更、ハードウェアタイマの期限切れ、シリアルインターフェースを通じた文字の受け取りや転送のような外部のイベントに対して非常に早く応答します。
割り込みが起こると、イベントが効率的に処理できるようになります。

▪ 15.2 割り込みレイテンシ

割り込みレイテンシは、割り込み要求が発生してから、割り込みサービスルーチンの最初の命令を実行するまでの時間です。
すべてのコンピュータシステムが割り込みレイテンシを持ちます。レイテンシは様々な要因に依存し、同じコンピュータシステムの中でも異なります。一般的に考慮される値は、最悪の場合の割り込みレイテンシです。
割り込みレイテンシは、以下に説明する別の小さい遅延の合計です。

▪ 15.2.1 割り込みレイテンシの要因

·最初の遅延は、一般的にはハードウェアで起こります。割り込み要求信号は、CPU のクロックに同期する必要があります。割り込み要求が CPU コアに到達する前に、同期ロジックに応じて通常最大 3 CPU サイクルが失われる可能性があります。
·CPU は通常、現在の命令を完了します。この命令は、多くのサイクルを取ることができ、ほとんどのシステムでは、除算、プッシュ倍数、またはメモリ·コピーの命令は、ほとんどのクロックサイクルを必要とするものです。 CPU に必要なサイクルの上には、殆どの場合メモリアクセスに必要な追加のサイクルがあります。 ARM7 のシステムでは、STMDB SP!,{R0-R11,LR};（プッシュパラメータと perm.レジスタ）の命令は、通常、最悪の場合の命令です。これは、スタック上に 13 個の 32 ビットレジスタを格納します。

CPU は 15 クロックサイクルが必要です。

·メモリシステムは、待機状態の追加サイクルを必要とする場合があります。

現在の命令が完了した後、CPU はモードスイッチを実行するか、またはスタックにレジスタ（一般的には PC とフラグレジスタ）をプッシュします。一般的に、最近の CPU は（ARM など）レジスタ保存よりも少ない CPU サイクルを必要とするモード·スイッチを実行します。

·パイプラインフィル

最近のほとんどの CPU はパイプライン化されています。パイプラインのさまざまな段階で命令が発されます。パイプラインの最終段階に達したときに命令が実行されます。モードスイッチが、パイプラインをフラッシュしたので、余分なサイクルでパイプラインを再充填する必要があります。

▪ 15.2.2 その他の割り込みレイテンシの要因

割り込みレイテンシの要因は他にも考えられます。

これらは、使用するシステムのタイプによって異なりますが、いくつかをリストします。

・レイテンシがキャッシュラインフィルによって起こっている。

メモリシステムが 1 つまたは複数のキャッシュを持っている場合、必要なデータを持つことはできません。この場合、必要なデータがメモリからロードされるだけでなく、大概、メモリから複数のワードを読み、完全なラインフィルが実行される必要があります。

・キャッシュの書き戻しによるレイテンシ

キャッシュミスにより、ラインが置き換えられる可能性があります。このラインがダーティとしてマークされている場合、メインメモリに書き戻される必要があり、それがさらに遅延を引き起こします。

・MMU 変換テーブルウォークによって生じるレイテンシ。

変換テーブルウォークは、特に潜在的に低速なメインメモリアクセスを伴う場合、かなりの時間がかかることがあります。リアルタイム割り込みハンドラでは、TLB によって引き起こされるアクセスするハンドラやデータを含まない変換テーブルウォークは、割り込みレイテンシをかなり増加させる可能性があります。

・アプリケーションプログラム。

アプリケーションプログラムは割り込みを無効にすることによって、当然新たにレイテンシを引き起こす可能性があります。これが意味を持つこともありますが、当然レイテンシを引き起こします。

・割り込みルーチン。

ほとんどのシステムでは、一つの割り込みは他の割り込みを無効にします。ISR で割り込みが再有効化されている場合でも、新たにいくつかの命令を受け取ります、レイテンシを引き起こします。

・RTOS（リアルタイム･オペレーティング･システム)。

RTOS も、RTOS の API-関数を呼び出すことができる割り込みを一時的に無効にする必要があります。RTOS のなかには、すべての割り込みを無効化し、事実上すべての割り込みに対する割り込みレイテンシを増加させるものもあり、他にも（embOS など)、優先順位の低い割り込みのみを無効にすることにより、優先度の高い割り込みのレイテンシに影響を与えないものもあります。

▪ 15.3 ゼロ割り込みレイテンシ

厳密な意味でゼロ割り込みレイテンシは、上記で説明したものではありません。我々が"ゼロ割り込みレイテンシ"と呼んでいるものは、RTOS に影響されない優先度の高い割り込みのレイテンシです。embOS を使用しているシステムは、embOS なしで稼働するシステムでの、優先度の高い割り込みで最悪の場合の割り込みレイテンシと同ものを持つことになります

なぜゼロレイテンシが重要なのでしょうか？

いくつかのシステムでは、最大割り込み応答時間やレイテンシを明確に定義することができます。この最大レイテンシは、非常に正確なタイミングを必要とするプロトコルやソフトウェア UART の実行の最大応答時間などの要件から生じる可能性があります。

ある顧客が、48MHz の ARM7 で FIQ（高速割り込み）を使用してソフトウェアで最高 800kHz までで受信する UART を実装しました。しかしこれは、短時間であっても高速割り込みが無効になった場合には不可能でしょう。

多くの組込みシステムでは、製品の品質は応答時間とそれによるレイテンシに依存します。典型的な例としては、定期的に高速で A/D 変換器から値を読み取るシステムがあり、精度がタイミングの正確さに依存しています。ジッタ少ない方ががより良い製品です。

優先順位の高い ISR が OS の API を使用できないのではなぜしょうか？

embOS データ構造が変更されたとき embOS は、優先順位の低い割り込みを無効にします。この間に、embOS データを変更する embOS 関数を呼び出す場合は、優先順位の高い ISR が有効になっていると、embOS のデータ構造が破損するおそれがあります。

高優先度 ISR はタスクと通信することができますか？

最も一般的な方法は、グローバル変数を使用することです。例えば、周期的に ADC から読み込まれ、結果はグローバル変数に格納されます。もう一つの方法は、高優先度 ISR 内で低優先度 ISR の割り込み要求を設定し、そうすることでその後の通信や、1 つ以上のタスクを起動させることを可能にします。これは、高優先度 ISR でいくつかのデータを受け取ることをお望みの場合に有用であり、低優先度 ISR はメッセージキューにデータバイトを受信します。

▪ 15.4 優先順位の高い/低い割り込み

ほとんどの CPU では異なる優先順位を持つ割り込みをサポートします。優先順位が異なると、2 つの効果があります：

·別々の割り込みが同時に発生した場合、優先順位の高い割り込みが優先され、その ISR は最初に実行されます。

·割り込みは、優先順位の同じまたは低い他の割り込みによって中断されることはありません。

どれだけの異なるレベルの割り込みが存在するかは CPU と割り込みコントローラに依存します。詳細は、CPU/ MCU/ SOC マニュアルや embOS の CPU&コンパイラマニュアルで説明されています。embOS は、割り込みの 2 つの異なるレベル（高/低優先度）を区別します。 embOS ポート固有のドキュメントでは、どの割り込みの優先順位が高いとみなされ、どの割り込みの優先順位が低いと見なされるのかという"線引き"がどこでされているのかを説明しています。一般的には、違いは次のとおりです。

低優先度の割り込み

・embOS の API 関数を呼び出すことができる

・embOS によって引き起こされるレイテンシ

高優先度の割り込み

・embOS API 関数を呼び出すことはできない

・embOS によるレイテンシなし（ゼロレイテンシ）

割り込み優先度が異なる例

割り込み優先レベルを 8 段階サポートする CPU があると仮定しましょう。 embOS では、優先順位の高い方から 3 段階が"高優先度の割り込み"として扱われます。 ARM の CPU で

は通常の割り込み（IRQ）と高速割り込み（FIQ）をサポートしています。embOS 使用すると、FIQ は"優先度の高い割り込み"として扱われます。ほとんどの実装では、高優先度の閾値は調整可能です。詳細については、プロセッサ固有の embOS マニュアルを参照してください。

- 15.4.1 優先順位の高い割り込みからの OS 関数の使用

優先順位の高い割り込みは embOS の関数を一切使用することはできません。これは、ゼロレイテンシの結果としておこる制限です：embOS は、優先順位の高い割り込みを無効にすることはありません。つまり、優先順位の高い割り込みは、関連リストの修正や二重関連リストなどのクリティカルな状況でも、オペレーティングシステムをいつでも中断するということです。これは、優先順位の高い割り込みのゼロ割り込みレイテンシは通常 OS の関数を呼び出す機能よりも重要であることから、決められている設計上の決定です。
それでも、間接的に優先度の高い割り込みから OS の関数を使用する方法もあります：
優先順位の高い割り込みが割り込み要求フラグを設定することにより、優先順位の低い割り込みをトリガします。するとこの優先順位の低い割り込みは OS の関数を呼び出すことができます。

タスク 1 は優先順位の高いの割り込みによって中断されました。この優先順位の高い割り込みは embOS の API 関数を直接呼び出すことはできません。したがって、優先順位の高い割り込みが embOS の API 関数を呼び出すことが許可されている優先順位の低い割り込みをトリガします。優先順位の低い割り込みは、タスク 2 を再開するために embOS の API 関数を呼び出します。

- 15.5 割り込みハンドラの規則

- 15.5.1 一般的な規則

割り込みサービスルーチン（ISR）にはいくつかの一般規則あります。これらのルールは、embOS を使用するシングルタスクプログラミングだけでなく、マルチタスクプログラミングにも適用されます。
·ISR はすべてのレジスタを保存する。

割り込みハンドラは完全にタスクの環境を復元する必要があります。この環境は、通常レジスタのみで構成されており、ISR は割り込み実行中に変更されたすべてのレジスタが最初に保存されていて、割り込みルーチンの最後で復元されることを確認する必要があります。

·割り込みハンドラは素早く終了しなければならない。

集中的な計算は、割り込みハンドラの外でなされなければなりません。割り込みハンドラは、受信した値を格納するか、一般のプログラム（タスク）の動作をトリガする目的のみに使用されなければなりません。いかなる状態でもポーリング操作を待機したり、実行したりしないでください。

▪ 15.5.2 その他のプリエンプティブマルチタスキングの規則

embOS ようなプリエンプティブマルチタスクシステムは、実行しているプログラムが現在のタスクの一部なのか割り込みハンドラなのかを知る必要があります。これは、embOS が ISR の実行中にタスクスイッチを実行することができず、ISR の終了時にのみ行うことができるためです。

タスクスイッチが ISR の実行中に発生した場合、割り込みされたタスクが再び現在のタスクになるとすぐに、ISR 実行を継続します。これは、これ以上割り込みを許可（割り込みを有効に）せず、embOS 関数を呼び出さないという、割り込みハンドラの問題ではありません。

これにより、次の規則が導かれます：

割り込みを再度有効にする、または embOS 関数を使用する ISR は、他のコマンドを実行する前に、最初に OS_EnterInterrupt()を呼び出し、また戻す前に、最後のコマンドとして OS_LeaveInterrupt()を呼び出す必要があります。

ISR によって優先順位の高いタスク準備されている場合、ルーチン OS_LeaveInterrupt()のなかでタスクスイッチが発生します。割り込みされたタスクが再度 READY 状態になったとき、後で ISR は終了されます。割り込みルーチンをデバッグする場合、混乱しないでください。これは、割り込みサービスルーチン内からタスクスイッチを開始するための最も効率的な方法です。

▪ 15.6 API 関数

embOS 関数を ISR 内から呼び出す前に、embOS は割り込みサービスルーチンが実行されていることを通知される必要があります。

Table 15.1: Interrupt API functions

▪ 15.6.1 OS_CallISR()

説明

embOS 割り込みハンドラで使用するためのエントリ関数です。ネスト可能な割り込みが無効になっています。

プロトタイプ

```
void        OS_CallISR  (void  (*pRoutine)(void));
```

Table 15.2: OS_CallISR() parameter list

補足情報

対応する割り込みが別の embOS 割り込みによって中断されるべきではない場合、embOS の割り込みハンドラ内で OS_CallISR()をエントリ関数として使用することができます。OS_CallISR()は、CPU の割り込み優先順位を、ユーザーが定義可能な"速い"割り込みレベルに設定し、他の embOS の割り込みをロックします。高速の割り込みは禁止されていません。

注：いくつかの特定の CPU の OS_CallISR()は、OS_EnterInterrupt()および OS_LeaveInterrupt()が利用できないので、割り込みハンドラを呼び出すために使用されなければならない場合があります。CPU のマニュアルを参照してください。

使用例

```
#pragma interrupt void OS_ISR_Tick(void) {
OS_CallISR(_IsrTickHandler);
}
```

▪ 15.6.2 OS_CallNestableISR()

説明

embOS の割り込みハンドラのエントリ関数です。ネスト可能な割り込みが有効化されます。

プロトタイプ

void　　OS_CallNestableISR （void （*pRoutine)(void));

Table 15.3: OS_CallNestableISR() parameter list

補足情報

OS_CallNestableISR()は、優先順位の高い embOS 割り込みによる割り込みを許可するときに、embOS 割り込みハンドラ内でエントリ関数として使用することができます。

OS_CallNestableISR ()は CPU の割り込み優先順位を変更することはなく、全ての割り込みを優先順位の高い状態に保つことができます。

注：いくつかの特定の CPU では、OS_EnterNestableInterrupt() および OS_LeaveNestableInterrupt()が使用できないため、があります。

割り込みハンドラを呼び出すために OS_CallNestableISR()を使用しなければならない場合があります。

CPU のマニュアルを参照してください。

使用例

```
#pragma interrupt void OS_ISR_Tick(void) {
OS_CallNestableISR(_IsrTickHandler);
}
```

- 15.6.3 OS_EnterInterrupt()

注:この関数はすべてのポートで使用できる訳ではありません。

説明

embOS に割り込みコードが実行されていることを通知します。

プロトタイプ

void OS_EnterInterrupt （void);

補足情報

OS_EnterInterrupt()が使用される場合、割り込みハンドラで呼び出される関数の中で一

番最初でなければなりません。また、最後に呼び出す関数として OS_LeaveInterrupt()とともに使用する必要があります。この関数の使用は、下記のような効果があります：

· タスクスイッチを無効にします

· 内部ルーチンで割り込みを無効に保ちます。

▪ 15.6.4 OS_LeaveInterrupt()

注:この関数は全てのポートで使用可能なわけではありません。

説明

embOS に割り込みルーチンが終端に達したことを通知します。ISR 内でタスクスイッチが起こります。

プロトタイプ

void OS_LeaveInterrupt (void);

補足情報

OS_LeaveInterrupt()が使用される場合、割り込みハンドラ内で呼び出される関数のうち最後でなければなりません。割り込みによちタスクスイッチが発生した場合、(割り込みされたプログラムが危険領域にあった場合を除き) 実行されます。

▪ 15.6.5 OS_EnterInterrupt()/OS_LeaveInterrupt()の使用例

OS_EnterInterrupt()/OS_LeaveInterrupt()を使用した割り込みルーチンの例:

```
__interrupt void ISR_Timer(void) {
OS_EnterInterrupt();
OS_SignalEvent(1,&Task);/* Any functionality could be here */
OS_LeaveInterrupt();
}
```

15.7 C からの割り込み有効化および無効化

タスクの実行中、マスク可能な割り込みは通常有効になっています。しかし、プログラムの特定の領域では、割り込みできないアトミック操作の領域を作るための短い時間のために、割り込みを無効化する必要がある場合があります。下記の例は、8/16 ビット CPU 上で長いタイプのグローバル volatile 変数へのアクセスとなっています。必要とされている 2 つ以上のアクセスの間で値が変化しないことを確認するために、割り込みを一時的に無効化する必要があります：

悪い例：

```
volatile long lvar;
void routine (void) {
lvar ++;
}
```

割り込みを無効化あるいは再有効化することに伴う問題は、割り込みを有効あるいは無効にする機能をネストすることができないということです。

お使いの C コンパイラは、割り込みの有効化と無効化のための 2 つの本質的な関数を提供しています。これらの関数はまだ使用できますが、embOS が提供する関数（これらは関数のように見えますが、実際にはマクロです）を使用することをお勧めします。

これらの推奨されている embOS 関数を使用しない場合は、コードの一部が割り込みを無効にして実行する必要があるルーチンがネストされている、あるいは OS ルーチンを呼び出す場合、問題が生じるでしょう。

可能であれば、いかに短時間であっても割り込みを無効化することをお勧めします。さらに、長いレイテンシをもたらす可能性がある（割り込み無効化が長いほど、割り込みレイテンシが長い）ため、割り込みが無効化されたときはルーチンを呼び出すべきではありません。割り込みを有効化して embOS の関数を呼び出している限り、安全に割り込みを無効にするためにコンパイラが提供する組み込み関数を使用することができます。

15.7.1 OS_IncDI() / OS_DecRI()

次の関数は、実際に RTOS.h に定義されたマクロなので、非常に迅速に実行され、非常に効率的です。これらは、最初 OS_IncDI()、その後 OS_DecRI()のペアとして使用されていることが重要です。

OS_IncDI()

インクリメントおよび割り込み無効化の短縮形です。割り込み無効カウンタ（OS_DICnt）を増加させて割り込みを無効化します。

OS_DecRI()

デクリメントおよび割り込み復元の短縮形です。カウンタを減少させ、カウンタが 0 になった場合は割り込み有効にします。

使用例

```
volatile long lvar;
void routine (void) {
OS_IncDI();
lvar ++;
OS_DecRI();
}
```

OS_DecRI()は OS 全体のために使用されている割り込み無効カウンタを減少させ、それによっていかなるルーチンも呼び出すことができ、かつ OS_DecRI()のマッチングが実行される前に割り込みがオンにされないという点でプログラムの残りの部分と一致させます。

上記の例のように、ルーチンが一切呼び出されていないところでほんの短い間でもの割り込みを無効にする必要がある場合、あなたも OS_DI()と OS_RestoreI()を組み合わせて使用することができます。

これらは、割り込み無効カウンタ OS_DICnt()が 2 回変更されないため、より効率的ですが、一度しかチェックされません。また、OS_DICnt()の状態が実際には変更されていないためルーチンで動作しないという欠点を持っているため、細心の注意を払って使用する必要があります。疑わしい場合は、OS_IncDI()と OS_DecRI()を使用してください。

▪ 15.7.2 OS_DI() / OS_EI() / OS_RestoreI()

OS_DI()

割り込み禁止の短縮形です。割り込みを禁止します。割り込み禁止カウンタは変更されません。

OS_EI()

割り込み有効化の短縮形です。割り込み無効カウンタを考慮しないので、割り込み有効カウントが 0 の値であることを確認しない限り、この関数を直接使用しないでください。

OS_RestoreI()
割り込み復元の短縮形です。割り込み無効カウンタに基づいて割り込みフラグのステータスを復元します。

使用例

```
volatile long lvar;
void routine (void) {
OS_DI();
lvar++;
OS_RestoreI();
}
```

- 15.8 割り込み制御マクロの定義(RTOS.h)

```
#define OS_IncDI()          { OS_ASSERT_DICnt(); OS_DI(); OS_DICnt++; }
#define OS_DecRI()          { OS_ASSERT_DICnt(); if (--OS_DICnt==0) OS_EI(); }
#define OS_RestoreI()       { OS_ASSERT_DICnt(); if (OS_DICnt==0)
        OS_EI(); }
```

- 15.9 割り込みルーチンのネスト

デフォルトでは、CPU が割り込みハンドラの実行で割り込みを無効にするため、割込みは ISR で無効になっています。割り込みハンドラ内で割り込みを再度有効にすると、現在の割り込みと優先度が等しいか、または優先度の高い他の割り込みを実行することができます。これらは以下のダイヤグラムで示した割り込みとして知られています：

（略）

短い割り込みレイテンシを必要とするアプリケーションでは、割り込みハンドラ内で OS_EnterNestableInterrupt()と OS_LeaveNestableInterrupt()を使用して ISR の内部割り込みを再度有効にすることができます。
ネストされた割り込みは、追跡が困難な問題にを引き起こす可能性があるので、割り込み

ハンドラ内で割り込みを有効にすることは推奨されません。embOS が、割り込み有効または無効フラグのステータスを追跡することは重要であるため、ISR 内からの割り込みの有効化および無効化は、embOS がこの目的のために提供している関数を使用して行う必要があります。ルーチン OS_EnterNestableInterrupt()は、ISR 内で割り込みを有効にし、さらにそれ以上のタスクスイッチを防ぐことができます。OS_LeaveNestableInterrupt()は割り込みルーチンを終了する直前に割り込みを無効にし、それによってデフォルトの状態を復元します。
割り込みを再度有効にすると、embOS スケジューラがすぐにその ISR を中断するために割り込みをすることができます。このケースでは、embOS は別の ISR がまだ実行していて、タスクスイッチを実行することはできないということを知っている必要があります。

- 15.9.1 OS_EnterNestableInterrupt()

注:この関数は全てのポートで利用可能なわけではありません。
説明
割り込みを再度可能にして embOS の内部危険領域カウンタを増加させます。それにより、タスクスイッチをこれ以上行えないようにします。

プロトタイプ
void OS_EnterNestableInterrupt (void);

補足情報
.ネストされた割り込みが必要な場合、この関数は割り込みハンドラ内のうち最初の呼び出しである必要があります。関数 OS_EnterNestableInterrupt()はマクロとして実装され、OS_EnterInterrupt()に OS_DecRI()を組み合わせたものと同じ機能を提供していますが、より小さく速いコードで、それゆえにより効率的です。

使用例
OS_LeaveNestableInterrupt()の例をご参照ください。

- 15.9.2 OS_LeaveNestableInterrupt()

注:この関数は全てのポートで利用可能というわけではありません。

説明

これ以降割り込みを不可能にし、embOS の内部危険領域カウントを減らします。カウンタが再びゼロに達した場合は、タスクスイッチを再度有効にします。

プロトタイプ

void OS_LeaveNestableInterrupt (void);

補足情報

この関数は、OS_EnterNestableInterrupt()と対応しており、ネストされた割り込みが以前 OS_EnterNestableInterrupt()によって有効になっていたときに、割り込みハンドラ内の最後の関数呼び出しである必要があります。関数 OS_LeaveNestableInterrupt()はマクロとして実装されており、OS_IncDI()との組み合わせで OS_LeaveInterrupt()おなじ役割を果たしますが、より小さく、より速いコードであるため、より効率的です。

- 15.10 マスク不可能割り込み (NMI)

embOS は、割り込みを無効にすることで、アトミック操作（不可分操作）を実行します。しかし、マスク不可能割り込み（NMI）を無効にすることはできず、すなわちアトミック操作を中断することができます。

したがって、NMI は細心の注意を払って使用されるべきであり、いかなる場合でも embOS のルーチンを呼び出すことができます。

- Chapter 16 危険領域

- 16.1 導入

危険領域はスケジューラがスイッチオフされている間のプログラムセクションであり、つまり実行中のタスクが待機しなければならない状況以外では、タスクスイッチとソフトウェアタイマの実行は許可されません。実際には、プリエンプションがオフになります。

クリティカル領域の典型的な例は、タイムクリティカルなハードウェアアクセス（例えば、バイトが一定の時間で書かれていなければならない領域で EEPROM に複数バイトを書き

込むど）を処理するプログラムセクションの実行、あるいは異なるタスクによって使用されるグローバル変数にデータを書き込むので、データに一貫性があることを確認する必要があるセクションです。
クリティカル領域は、タスクの実行中にどこでも定義することができます。重要な領域はネストすることができます。最も外側のループが残された後にスケジューラが再びオンになります。割り込みは、危険領域ではまだ合法です。ソフトウェアタイマと割り込みはいかなる方法でも危険領域として実行されるので、そのような宣言の仕方だと、害はありませんがよい影響もありません。危険領域の実行中にタスクスイッチの期限がくる場合は、その領域が終了した直後に実行されます。

16.2 API 関数

Table 16.1: Critical regions API functions

16.2.1 OS_EnterRegion()

説明
OS に対し、危険領域の開始を示します。

プロトタイプ
void OS_EnterRegion (void);

補足情報
OS_EnterRegion() は実際には関数ではなくマクロです。しかし、関数のように振る舞い、しかもより効率的です。このマクロを使用し、embOS に危険領域の開始を示します。危険領域カウンタ(OS_RegionCnt)はデフォルトでは 0 であり、ルーチンがネストされると増加します。, カウンタは OS_LeaveRegion()の呼び出しによって減少されます。カウンタが再び 0 になった場合は、危険領域が終了します。 割り込みが OS_EnterRegion()の使用によって無効化されることはありませんが、割り込みの無効化によってプリエンプティブタスクスイッチを無効にします。

使用例
```
void SubRoutine(void) {
```

```
OS_EnterRegion();
/* this code will not be interrupted by the OS */
OS_LeaveRegion();
}
```

- 16.2.2 OS_LeaveRegion()

説明

OS に危険領域の終了を示します。

プロトタイプ

```
void OS_LeaveRegion (void);
```

補足情報

OS_LeaveRegion() は実際には関数ではなくマクロです。しかし、関数と同じように機能し、より効率的です。 このマクロを使用することによって、embOS に危険領域の終了を示します。危険領域カウンタ (OS_RegionCnt)はデフォルトでは 0 ですが、減少されます。このカウンタが再び 0 になると、危険領域が終了します。

使用例

OS_EnterRegion()の例をご参照ください。

- Chapter 17 タイム測定

embOS は 2 タイプの時間測定を用意しています。

· 低解像度(タイム変数使用)

· 高解像度(ハードウェアタイマ使用)

双方ともこの章で説明しています。

▪ 17.1 導入

embOS では、ユーザーコードのセクションの実行時間を計算するために使用するランタイム測定の、二つの基本的なタイプをサポートしています。

高分解能測定がサイクルと呼ばれる時間単位に基づいていながら、低解像度の測定は、ティックのタイムベースを使用しています。サイクルの長さは、タイマクロック周波数に依存します。

▪ 17.2 低解像度測定

システムタイム変数 OS_Time ティックまたはミリ秒で測定されています。低解像度関数 OS_GetTime() および OS_GetTime32()は、この変数の実行中コンテンツを返すことに使われます低解像度測定の基本的な発想は非常に単純です。システムタイムは、コードの領域が測定される前と後に一度ずつ帰され、コードが実行されるのにかかった時間を保持するために最初の値は次の値から引かれます。低解像度という用語は、返されるタイム値が完成されたティックで測定されるという理由で使用されます。次のことを考えてみてください。通常のティックが 1 ミリ秒で、変数 OS_Time がティック割り込みあるいは 1 秒ごとに増加する状況を考えてみてください。これは、実際のシステムタイムが低解像度関数の返すものよりも大きいということです。(例えば、割り込みが 1.4 ティックで起こった場合、システムは 1 ティックしか経過していないという測定をします）実行時間測定の場合には、システムタイムは二度測定される必要があるため問題がさらに大きくなります。それぞれの測定で実際のタイムよりも、せいぜい 1 ティック少なくなる程度なので、二つの測定の違いは理論上高々2 ティックです。

次のダイアグラムでは、低解像度測定の動きを図示しています。コードの領域が実際に 0.5 ミリ秒で開始し 5.2 ミリ秒で終了しているのがわかり、これは実際の実行時間が （5.2 - 0.5） = 4.7 ミリ秒であることを意味します。しかし、1 ミリ秒のティックでは、最初の OS_GetTime()の呼び出しで 0 を返し、次の呼び出しで 5 を返します。よって、コードの実行時間は(5 - 0) = 5 ミリ秒となります。

多くのアプリケーションにとって、低解像度測定はユーザーのニーズを満たすのに十分です。使いやすさや計算時間の短さの点で高解像度測定よりも望ましい場合もあります。

- 17.2.1 API 関数

Table 17.1: Low-resolution measurement API functions

- 17.2.1.1 OS_GetTime()

説明

実行中のシステムタイムをティックで返します。

プロトタイプ

int OS_GetTime (void);

戻り値

16 または 32 ビット整数としてのシステム変数 OS_Time。

補足情報

この関数は 8/16 ビット CPU では 16 ビット値として、また 32 ビット CPU では 32 ビット値としてシステム単位を返します。OS_Time 変数は 32 ビット値です。よって、戻り値が 32 ビットの場合、OS_Time 変数のコンテンツです。戻り値が 16 ビットの場合は、OS_Time 変数低いほうの 16 ビットです。

- 17.2.1.2 OS_GetTime32()

説明

実行中のシステムタイムをティックで、32 ビット値として返します。

プロトタイプ

int OS_GetTime32 (void);

戻り値

32 ビット整数値でのシステム変数。

補足情報

この関数は、常にシステムタイムを 32 ビット値として返します。OS_Time 変数は 32 ビット値でもあるので、戻り値は単に OS_Time 値のコンテンツです。

17.3 高解像度測定

高解像度測定では embOS のプロファイリングビルドで使用されているものと同じルーチンを使用しており、タイム測定の高精度名チューニングが可能です。システム解像は使用されている CPU に依存しますが、一般的には約 1 µs であり、高解像度測定は低解像度測定の 10 倍正確です。与えられた時間に終了したティックの数を測定する代わりに、終了したサイクルの数の内部カウントが保存されます。以下の図をご覧ください。低解像度計算で使用されたものと同じコードで実行時間を測定しています。この例では、CPU が 10MH z で動作するタイマを保持して計算していることを想定しています。よって、ティックごとのサイクルの数は(10 MHz / 1kHz) = 10,000 です。これは、割り込みごとに、タイマは 0 で再し始動し 10,000 までカウントされます。

OS_Timing_Start() の呼び出しにより、5,000 サイクルで初期値を計算するのに対し、OS_Timing_End() の呼び出しでは 52,000 で終了値を計算します (いずれの値も内部のトラックに保存されます。)。 よって、この例で測定されたコードの実行時間は(52,000 - 5,000) = 47,000 サイクル、これが 4.7 ms と一致します。

関数 OS_Timing_GetCycles()は実行時間を上記のサイクルで返すのに使用されますが、マイクロ秒単位(µs)で返す関数 OS_Timing_Getus を使用するほうがより一般的です。上記の例では、戻り値は 4,700 µs です。

データ構造

高解像度ルーチンはすべて、以下で定義されるような OS_TIMING のデータ構造へのポインタをパラメータとしてとる必要があります:

```
#define  OS_TIMING  OS_U32
```

17.3.1 API 関数

Table 17.2: High-resolution measurement API functions

17.3.1.1 OS_Timing_Start()

説明

測定されるコードの領域の開始を記録します。

プロトタイプ

void　OS_Timing_Start　(OS_TIMING*　pCycle);

Table 17.3: OS_TimingStart() parameter list

補足情報

この関数は OS_Timing_End()とともに使用される必要があります。

- 17.3.1.2 OS_Timing_End()

説明

測定されるコードの領域の終了を記録します。

プロトタイプ

void　OS_Timing_End　(OS_TIMING*　pCycle);

Table 17.4: OS_TimingEnd() parameter list

補足情報

この関数は OS_Timing_Start()とともに使用する必要があります。

- 17.3.1.3 OS_Timing_Getus()

説明

OS_Timing_Start()と OS_Timing_End()の間のコードの実行時間をマイクロ秒で返します。

プロトタイプ

OS_U32　OS_Timing_Getus　(OS_TIMING*　pCycle);

Table 17.5: OS_Timing_Getus() parameter list

補足情報

ビット整数値でのマイクロ秒 (µs)単位での実行時間。

- 17.3.1.4 OS_Timing_GetCycles()

説明

OS_Timing_Start()と OS_Timing_End()の間のコードの実行時間を返します。

プロトタイプ

OS_U32 OS_Timing_GetCycles (OS_TIMING* pCycle);

Table 17.6: OS_Timing_GetCycles() parameter list

戻り値

サイクルにおける実行時間を 32 ビット整数で返します。

補足情報

サイクル長はタイマクロック周波数に依存します。

- 17.4 使用例

下記の例では、コードの領域の実行時間を返す低解像度および高解像度での測定を示しています。:

```
/*********************************************************
*          SEGGER MICROCONTROLLER SYSTEME GmbH
*          Solutions for real time microcontroller applications
**********************************************************
File        : SampleHiRes.c
Purpose     : Demonstration of embOS Hires Timer
----------END-OF-HEADER-----------------------------*/
#include "RTOS.H"
#include <stdio.h>
OS_STACKPTR int Stack[1000]; /* Task stacks */
OS_TASK TCB;    /* Task-control-blocks */
volatile int Dummy;
void UserCode(void) {
```

```
for (Dummy=0; Dummy < 11000; Dummy++);  /* Burn some time  */
}
/*
* Measure the execution time with low resolution and return it in ms (ticks)
*/
int BenchmarkLoRes(void) {
int t;
t = OS_GetTime();
UserCode();          /* Execute the user code to be benchmarked */
t = OS_GetTime() - t;
return t;
}
/*
* Measure the execution time with hi resolution and return it in us
*/
OS_U32 BenchmarkHiRes(void) {
OS_U32 t;
OS_Timing_Start(&t);
UserCode();          /* Execute the user code to be benchmarked */
OS_Timing_End(&t);
return OS_Timing_Getus(&t);
}
void Task(void) {
int tLo;
OS_U32 tHi;
char ac[80];
while (1) {
tLo = BenchmarkLoRes();
tHi = BenchmarkHiRes();
sprintf(ac, "LoRes: %d ms\n", tLo);
OS_SendString(ac);
sprintf(ac, "HiRes: %d us\n", tHi);
OS_SendString(ac);
```

```
}
}
/*********************************************************
*
*          main
*
*********************************************************/
void main(void) {
OS_InitKern();      /* Initialize OS    */
OS_InitHW();        /* Initialize Hardware for OS        */
/* You need to create at least one task here !           */
OS_CREATETASK(&TCB, "HP Task", Task, 100, Stack);
OS_Start();         /* Start multitasking        */
}
```

アウトプットは次のとおりです。:

```
LoRes: 7  ms
HiRes: 6641  us
LoRes: 7  ms
HiRes: 6641  us
LoRes: 6  ms
```

▪ Chapter 18　システム変数

システム関数は、ここでは embOS の動きとデバッグをよりよく理解するために説明されています。

▪ 18.1 導入

注:どのシステム変数の値も変更しないでください。

これらの変数は、アクセス可能であり定数宣言されませんが、embOS 関数のみによって

変更される必要があります。しかし、これらの変数の中には、時間変数に非常に役立つものもあります。

18.2 タイム変数

18.2.1 OS_Global

OS_Global は embOS の内部変数を含む構造です。次の関数 OS_Time a および OS_TimeDex は OS_Global の一部です。 OS_Global のほかの部分はどれも、embOS を使用するのに必要とされないため、ここでは説明しません。

18.2.2 OS_Time

説明

実行中のシステムタイムをティック（通常は 1 ミリ秒と同等）で含むタイム変数です。

補足情報

タイム変数は 1 タイム単位の解像であり、通常は 1/1000 秒
(1 ミリ秒) で embOS タイマ割り込みハンドラを二つ連続で呼び出す間のタイムです。この変数に直接アクセスする代わりに、241 ページの Time measurement の章で説明されている OS_GetTime() または OS_GetTime32() を使用してください。

18.2.3 OS_TimeDex

基本的には内部使用にか限ります。次のタスクスイッチあるいはタイマのアクティブ化の期限切れのタイムを含みます。 ((int)(OS_Time - OS_TimeDex)) >= 0 の場合、タスクリストおよびタイマリストは、タスクあるいはタイマをアクティブ化するためにチェックされっます。 アクティブ化後は、OS_TimeDex は次のタスクあるいはタイマがアクティブ化されるタイムスタンプに割り当てられます。

18.3 OS 内部変数およびデータ構造

embOS 内部変数は embOS の使用においては必要とされないためここでは説明しません。embOS の将来のバージョンっでも変更されないことが保証されているのは 記録された

API 関数のみであるので、 お使いのアプリケーションは内部変数に依存してはいけません。

重要
いずれのシステム変数も変更しないでください。

Chapter 19　　システムティック

この章では、ハードウェアタイマとそれで使用可能なすべてのオプションによって生成されたシステムティックの概念を説明しています。

19.1　導入

通常、ハードウェアタイマは OS のタイムベースとして使用される定期割り込みを生成します。割り込みサービスルーチンは OS のティックハンドラのひとつを呼び出します。embOS は、さまざまな機能性だけでなく、システムティックハンドラ内からフック関数を呼び出す方法でさまざまな機能のティックハンドラを提供しています。
タイマ割り込みの生成
ハードウェアタイマは通常、BSP で配信される OS_InitHW()関数で初期化されます。BSP はハードウェアタイマ割り込みに呼び出される割り込みハンドラも含んでいます。この割り込みハンドラは、この章で説明する embOS のシステムティックハンドラ関数のひとつを呼び出す必要があります。

19.2　ティックハンドラ

タイムベースとして使用される割り込みサービスルーチンは、ティックハンドラを呼び出す必要があります。利用可能なティックハンドラはいろいろありますが、そのうちひとつを呼び出す必要があります。様々なティックハンドラが存在する理由は簡単です。性能やコードの大きさ、実行速度が異なるのです。ほとんどのアプリケーションでは、スタンダードなティックハンドラ OS_TICK_Handle()を使用しており、これは呼び出されるごとにティックカウントを一ずつ増加させます。ティックハンドラは小さくて効率的ですが、割り込み速度がティック速度と異なるような状況には対処できません。OS_TICK_HandleEx() はティックあたり 1.6 割り込みというような端数の割り込み速度で

も扱うことができます。

- 19.2.1 API 関数

Table 19.1: API functions

- 19.2.1.1 OS_TICK_Handle()

説明

て通常ハードウェアタイマ割り込みハンドラに呼び出されるデフォルトの embOS タイマティックハンドラです。

プロトタイプ

```
void OS_TICK_Handle ( void );
```

補足情報

embOS ティックハンドラはアプリケーションから呼び出すことができず、割り込みハンドラから呼び出されなければなりません。

OS ティックハンドラを呼び出す前に OS_EnterInterrupt() および OS_EnterNestableInterrupt()を呼び出す必要があります。

使用例

```
/* Example of a timer interrupt handler */
/*********************************************************************
*
*          OS_ISR_Tick
*/
__interrupt void OS_ISR_Tick(void) {
OS_EnterNestableInterrupt();
OS_TICK_Handle();
OS_LeaveNestableInterrupt();
}
```

- 19.2.1.2 OS_TICK_HandleEx()

説明

通常のティックハンドラの変わりに使用される代替ティックハンドラです。基本的なタイマ割込み間隔（ティック）が 1 ミリ秒の倍数で、遅れのパラメータとして使用されているタイム値がまだタイムベースとして 1 ミリ秒を使用する必要がある場合に使用できます。

プロトタイプ

```
void  OS_TICK_HandleEx ( void );
```

補足情報

embOS ティックハンドラはアプリケーションから呼び出すことができず、割り込みハンドラから呼び出されなければなりません。

OS ティックハンドラを呼び出す前に OS_EnterInterrupt() および OS_EnterNestableInterrupt()を呼び出す必要があります。

OS_TICK_HandleEx()の設定方法については 265 ページの OS_TICK_Config()のセクションをご参照ください。

使用例

```
/*  Example of a timer interrupt handler using OS_HandleTickEx */
/*********************************************************************
*
*          OS_ISR_Tick
*/
__interrupt void OS_ISR_Tick(void) {
OS_EnterNestableInterrupt();
OS_TICK_HandleEx();
OS_LeaveNestableInterrupt();
}
```

ハードウェアタイマが周波数 500Hz で動いていると仮定すると、システムを 2 ミリ秒ごとに割り込みすることになり、OS_TICK_Config のセクションの「使用例」で説明しているように、embOS ティックハンドラの設定関数 OS_TICK_Config()を呼び出す必要があります。これは OS_InitHW()の間、embOS タイマが開始する前に行う必要があります。

- 19.2.1.3 OS_TICK_HandleNoHook()

説明

通常ハードウェアタイマ割り込みハンドラに呼び出されるフック関数なしの最適化された embOS タイマティックハンドラの代替速度。

プロトタイプ

```
void  OS_TICK_HandleNoHook  (  void  );
```

補足情報

ティックハンドラはアプリケーションから呼び出すことができず、割り込みハンドラから呼び出されなければなりません。

OS ティックハンドラを呼び出す前に OS_EnterInterrupt()および OS_EnterNestableInterrupt()を呼び出す必要があります。

使用例

```
/*  Example  of  a  timer  interrupt  handler  */

/*********************************************************************
*
*            OS_ISR_Tick
*/
__interrupt  void  OS_ISR_Tick(void)  {
OS_EnterNestableInterrupt();
OS_TICK_HandleNoHook();
OS_LeaveNestableInterrupt();
}
```

- 19.2.1.4 OS_TICK_Config()

説明

ティックの割り込み比率を設定します。 "通常の"ティックハンドラ OS_TICK_Handle()は 1:1 の比率、つまりひとつの割り込みがティックカウント（OS_Time）を 1 ずつ増やすこ

とを想定しています。他の比率については、OS_TICK_HandleEx()が使用される必要があります。比率は OS_TICK_Config() を呼び出すことによって定義されます。

プロトタイプ

void OS_TICK_Config (unsigned FractPerInt, unsigned FractPerTick);

Table 19.2: OS_TICK_Config() parameter list

補足情報

FractPerInt/FractPerTick = 2 ティック割り込み間のタイム/ 1 ティックのタイム

割り込みが 1.6 ミリ秒ごとに生成されるような場合、端数値はティックが 1 ミリ秒というようにサポートされることに注意してください。これは、FractPerInt および FractPerTick が次のようになるということです。:

FractPerInt = 16;

FractPerTick = 10;

または

FractPerInt = 8;

FractPerTick = 5;

例:

```
OS_TICK_Config(2, 1);       // 500 Hz interrupts (2ms), 1ms tick
OS_TICK_Config(8, 5);       // Interrupts once per 1.6ms, 1ms tick
OS_TICK_Config(1, 10);      // 10 kHz interrupts (0.1ms), 1ms tick
OS_TICK_Config(1, 1);       // 10 kHz interrupts (0.1ms), 0.1 ms tick
OS_TICK_Config(1, 100);     // 10 kHz interrupts (0.1ms), 1 us tick
```

▪ 19.3 システムティックへのフッキング

ティックハンドラから関数を呼び出すことが望ましい状況はいろいろありますが、例としては下記のものがあります。:

· ウォッチドッグアップデート

· 定期ステータスチェック
· 定期 I/O アップデート
高優先度タスク 1 ティックピリオドタイムでのソフトウェアタイマで同様の機能が達成されます。

フック関数を使用する利点
フック関数は embOS のタイマ割り込みハンドラから直接呼び出され、かつコンテキストスイッチを起こさないため、フック関数を使用するとタスクスイッチやソフトウェアタイマのアクティブ化よりも断然早くなります。

19.3.1 API 関数

Table 19.3: API functions

19.3.1.1 OS_TICK_AddHook()

説明
ティックフックハンドラを追加します。

プロトタイプ
void OS_TICK_AddHook (OS_TICK_HOOK * pHook,
OS_TICK_HOOK_ROUTINE * pfUser);

Table 19.4: OS_TICK_AddHook() parameter list

補足情報
フック関数は割り込みハンドラから直接呼び出されます。よって、この関数はできる限り早く実行されなければなりません。ティックフックによって呼び出される関数は割り込みを再度有効化することはできません。

19.3.1.2 OS_TICK_RemoveHook()

説明
ティックフックハンドラを削除します。

プロトタイプ

void OS_TICK_RemoveHook (OS_TICK_HOOK * pHook);

Table 19.5: OS_TICK_RemoveHook() parameter list

補足情報

この関数は OS_TICK_AddHook()の呼び出しによってインストールされたティックフック関数を動的に除去するために呼び出されます。

Chapter 20 ターゲットシステムの設定(BSP)

20.1 導入

ユーザーは embOS を始める際に何も設定する必要はありません。供給されたスタートプロジェクトがシステム上で実行されます。設定内の小さな変化は、システムの周波数の後の時点、または付属の embOSView との通信に使用される UART のために、必要となります。

ファイル RTOSInit.c はソースコードで提供されており、ターゲットハードウェアのニーズに合わせて変更することができます。それはお使いのアプリケーションプログラムでコンパイルされ、リンクします。

20.2 ハードウェア固有ルーチン

Table 20.1: Hardware specific routines

20.2.1 OS_Idle()

embOS 関数 OS_Idle()は実行の準備ができているタスクが存在しないときに呼び出されます。関数 OS_Idle()は embOS とともに提供されるターゲット CPU 固有の RTOSInit.c ファイルの一部です。通常、機能なしのエンドレスループとしてプログラムされます。ほとんどの embOS ポートでは、ターゲット CPU の電源保存スリープモードをアクティブ化します。

embOS の OS_Idle() 関数はタスクではないので、タスクコンテキストや固有スタックを持

ちません。 OS_Idle() 関数はカーネルに使われている通常の CSTACK 上で動きます。OS_Idle がタスクスイッチを起こさない限りは、例外や割り込みがあっても問題はありません。これらは OS_Idle に戻り、コードは中断されたところから続行します。OS_Idle()の実行中にタスクスイッチが起きた場合, OS_Idle() 関数は中断され再びアクティブ化されるまで実行されません。 OS_Idle()がアクティブ化されるときは常に、最初から始まります。 割り込みされたコードは続行されません。システム内で最も優先順位で動くご自分のアイドルタスクを作成することができます。そのアイドルタスクにおいてブロッキング関数やサスペンディング関数を呼び出さない場合、OS_Idle()にたどり着くことはないでしょう。
これは、割り込み及びタスクスイッチで、短い反応時間を維持するための推奨されるソリューションです。
OS_Idle()で doStuff()の実行中にタスクスイッチを避けるため、OS_EnterRegion()か OS_LeaveRegio()のどちらかを使用する場合があります。
危険領域で実行しても割り込みをブロックしませんが、OS_LeaveRegion()が呼び出されるまでタスクスイッチが無効になります。
OS_Idle()の中で危険領域を使用すると、タスクの起動時間に影響を与えますが、割り込みレイテンシに影響することはありません。

▪ 20.3 設定の定義

ほとんどの組み込みシステムでは、設定は単純に定義を変更することによってなされており、RTOSInit.c ファイルの最上位にあります。

Table 20.2: Configuration defines overview

▪ 20.4 設定の変更方法

変更しなければならない可能性のあるファイルは RTOSInit.c のみです。このファイルはハードウェア固有のルーチンを全て含んでいます。一つの例外は、embOS のいくつかのポートでは追加の割り込みベクトルテーブルファイルを必要とするということです（詳細は embOS の CPU & コンパイラ固有マニュアル をご参照ください)。

- 20.4.1 システム周波数 OS_FSYS の設定

関連する定義

OS_FSYS

関連ルーチン

OS_ConvertCycles2us()（プロファイリングのみで使用される）

ほとんどのシステムでは RTOSInit.c の上部で OS_FSYS 定義を変更するのに十分でなければならない。プロファイリングを使用する場合、特定の値が OS_ConvertCycles2us()の変更を必要とする場合があります。 RTOSInit.c ファイルには、どういった場合にこれが必要で、何が行われる必要があるかという詳細情報が含まれています。

- 20.4.2 embOS 用ティック割り込みを生成するための別のタイマの使用

関連ルーチン

OS_InitHW()

embOS は通常、1 ミリ秒に等しいタイマ割り込みまたはティックにより、1 ミリ秒あたり 1 割り込みを生成します。これは、ルーチン OS_InitHW()で初期化されたタイマによって行われます。お使いアプリケーションのための別のタイマを使用する必要がある場合、適切なタイマを初期化するために OS_InitHW()を変更する必要があります。初期化の詳細については、RTOSInit.c のコメントをお読みください。

- 20.4.3 embOSView の異なる UART あるいはボーレートの使用

関連する定義

OS_UART

OS_BAUDRATE

関連ルーチン:

OS_COM_Init()

OS_COM_Send1()

OS_ISR_rx()

OS_ISR_tx()

いくつかのケースでは、単に定義 OS_UART を変更することによって行われます。お使いの CPU でサポートされている UARTS の詳細については、RTOSInit.c ファイルの内容を参照してください。

▪ 20.4.4 ティック周波数の変更

関連する定義

OS_FSYS

上記の通り、embOS では通常 1 ミリ秒あたり 1 の割り込みを生成します。 OS_FSYS はお使いのシステムのクロック周波数を Hz で定義します。OS_FSYS の値は 1000 割り込み/秒のシステムタイマのご所望のリロードカウンタの値を計算するために使用されます。従って、割り込み周波数は、通常 1 kHz です。異なる（低いまたは高い）割り込みレートが可能です。1 kHz とは異なる割り込み周波数を選択された場合は、時間変数 OS_Time の値は、1 ミリ秒の倍数に相当しません。しかし、ティック時間として 1 ミリ秒の倍数を使用なさる場合は、基本的な時間単位は（オプションの）コンフィギュレーションマクロ OS_CONFIG()（µbelow を参照ください）を使用して 1 ミリ秒とすることが可能です。基本的な時間単位は 1 ミリ秒である必要はなく、100 マイクロ秒または 10 ミリ秒またはそれ以外の値であるかもしれません。ほとんどのアプリケーションでは、1 ミリ秒が適切な値です。

▪ 20.5 STOP / HALT / IDLE モード

ほとんどの CPU では、省電力のための STOP、HALT または IDLE モードをサポートしています。これらのタイプのモードを使用するのは、アイドル時間中の消費電力を節約する可能な方法の一つです。タイマ割り込み embOS ティックごとにシステムを起動する限り、また他の割り込みがタスクをアクティブにする限り、これらのモードは、消費電力を節約するために使用することができます。

必要な場合は、アイドル時間中に CPU を省電力モードに切り替えるために、ハードウェアに依存するモジュール RTOSInit.c の一部である OS_Idle()ルーチンを変更することができます。お使いのプロセッサの詳細については、embOS ドキュメントのマニュアル CPU &コンパイラマニュアルをご参照ください。

▪ Chapter 21 プロファイリング

この章では、アプリケーションで使用できるプロファイリング関数についてご説明いたします。

- 21.0.1 API 関数

Table 21.1: API functions

- 21.0.1.1 OS_STAT_Sample()

説明

OS_STAT_Sample()は プロファイリングを開始し OS_STAT_Sample()の最後の呼び出しからのタスクの全実行時間を計算します。

プロトタイプ

void OS_STAT_Sample (void);

補足情報

OS_STAT_Sample()はプロファイリングを 5 秒で開始し、次の OS_STAT_Sample() ノ呼び出しは 5 秒以内に行われる必要があります。特定の CPU 負荷を 1/10%で得るためには embOS 関数 OS_STAT_GetLoad() を使用してください。

- 21.0.1.2 OS_STAT_GetLoad()

説明

OS_STAT_GetLoad() は実行中のタスクの CPU 負荷を 1/10 %で計算します。

プロトタイプ

int OS_STAT_GetLoad(OS_TASK * pTask);

Table 21.2: OS_STAT_GetLoad() parameter list

戻り値

OS_STAT_GetLoad は実行中のタスクの CPU 負荷を 1/10%で返します。

補足情報

OS_STAT_GetLoad() は、OS_STAT_Sample() が定期的に呼び出されていることを必要とします。

- 21.0.1.3 OS_STAT_Sample() および OS_STAT_GetLoad()のサンプルアプリケーション

```
#include "RTOS.h"
#include "stdio.h"
OS_STACKPTR int StackHP[128], StackLP[128], StackMP[128]; /* Task stacks */
OS_TASK TCBHP, TCBLP, TCBMP; /* Task-control-blocks */
static void HPTask(void) {
volatile int r;
while (1) {
OS_Delay (1000);
OS_STAT_Sample();
r = OS_STAT_GetLoad(&TCBMP);
printf("CPU Usage of MP Task: %d\n", r);
}
}
static void MPTask(void) {
while (1) {
}
}
static void LPTask(void) {
while (1) {
}
}

int main(void) {
OS_IncDI();          /* Initially disable interrupts  */
OS_InitKern();       /* Initialize OS     */
OS_InitHW();         /* Initialize Hardware for OS          */
/* You need to create at least one task before calling OS_Start() */
OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
OS_CREATETASK(&TCBMP, "MP Task", MPTask,          50, StackMP);
OS_CREATETASK(&TCBLP, "LP Task", LPTask,          50, StackLP);
```

```
OS_Start();          /* Start multitasking        */
return 0;
}
```

Output:

```
500
499
501
500
500
...
```

- Chapter 22 embOSView: プロファイリングと分析

- 22.1 概要

embOSView は embOS を使用して実行中のアプリケーションの状態を表示します。シリアルインタフェース（UART）は通常、ターゲットとの通信に使用されます。ハードウェア依存のルーチン及び embOSView との通信に使用可能な定義は RTOSInit.c に位置しています。このファイルには、適切に設定する必要があります。
このファイルを設定する方法の詳細については、embOS マニュアルの CPU&コンパイラマニュアルを参照してください。 embOSView ユーティリティは、embOS とともに embOSView.exe として出荷されており、Windows 9x/ NT /2000 で動作します。最新バージョンは当社のウェブサイト www.segger.com で提供されています。embOSView は、実行中のターゲット·アプリケーションの分析のための非常に有用なツールです。

- 22.2 タスク一覧ウィンドウ

embOSView は、タスク一覧ウィンドウにある目的のアプリケーションの作成されたタスクの状態をすべて示しています。表示される情報は、お使いのアプリケーションで使用されるライブラリに依存しています。

Table 22.1: Task list window overview

タスク一覧ウィンドウは、実行中のタスクごとのスタック使用料及び CPU 負荷を分析するのに有用です。

22.3 システム環境変数ウィンドウ

embOSView は、システム環境変数ウィンドウ内の主要なシステム変数の実際の状態を示しています。表示される情報は、お使いのアプリケーションで使用されているライブラリに依存します：

Table 22.2: System variables window overview

22.4 Sharing the SIO for terminal I/O

embOSView で使用されるシリアル入力/出力（SIO）は、入力と出力の両方に同じ時にアプリケーションで使用することができます。これは非常に役に立ちます。端末入力は、多くの場合、端末出力がデバッグメッセージを出力するために使用することができるキーボード入力として使用されます。

入力と出力は、メニューから View/ Terminal を選択することにより表示することができ、ターミナルウィンドウを介して行われます。ビューアの機能と並行して、ターミナルウィンドウを介しての通信を確保するために、アプリケーションは、ターミナルウィンドウと 1 バイトを受信する受信ルーチンをフックする関数 OS_SetRxCallback()に文字列を送信するための関数 OS_SendString()を使用します。

22.5 API 関数

Table 22.3: Shared SIO API functions

22.5.1 OS_SendString()

説明

ターミナルウィンドウに SIO 以上の文字列を送信します。

プロトタイプ

void OS_SendString (const char* s);

Table 22.4: OS_SendString() parameter list

補足情報
この関数は、 RTOSInit.c.で定義されている OS_COM_Send1()を使用しています。

- 22.5.2 OS_SetRxCallback()

説明
1 文字を受信するためのルーチンにコールバックフックを設定します。

プロトタイプ
typedef void OS_RX_CALLBACK (OS_U8 Data)
OS_RX_CALLBACK* OS_SetRxCallback (OS_RX_CALLBACK* cb);

Table 22.5: OS_SetRxCallback() parameter list

戻り値
上記のような OS_RX_CALLBACK* です。呼び出しの前にフックされていたコールバック関数へのポインタです。

補足情報
ユーザー関数は embOS から呼び出されます。受信された文字はパラメータと認識されます。下記の例をご覧ください。

使用例
```
void GUI_X_OnRx(OS_U8 Data); /* Callback ... called from Rx-interrupt */
void GUI_X_Init(void) {
OS_SetRxCallback( &GUI_X_OnRx);
}
```

- 22.6 API トレースの使用

embOS バージョン 3.06 以降では、API 呼び出しのトレース機能が含まれています。これは、ターゲットのアプリケーションでトレースビルドライブラリを使用する必要があります。
トレースビルドライブラリは 100 トレースエントリのためのバッファを実装します。 API 呼び出しのトレースは、関数 OS_TraceEnable()と OS_TraceDiasable()を使用して、embOSView から Trace メニューを介してあるいはアプリケーション内から開始および停止できます。個々のフィルタは、API 呼び出しがタスクごとにトレースされるべきなのか、あるいは割り込みやタイマルーチン内からトレースされるべきなのかかを決定するために定義することができます。
トレースが開始されると、API 呼び出しが定期的に embOSView によって読み取られ、トレースバッファに記録されます。結果は、トレースウィンドウに表示されます：

トレースリスト内のすべてのエントリは、実際のシステム時刻で記録されています。タスクからの呼び出しやイベントの場合は、タスク ID（タスク ID）とタスク名（タスク名、15 文字まで）も記録されます。 API 呼び出しのパラメータは可能であれば記録され、APIName 列の一部として表示されます。上記の例では、OS_Delay（3）で見ることができます。トレースバッファが一杯になると、トレースは自動的に停止します。
トレースリストおよびバッファーは embOSView からクリアすることができます。

embOSView からのトレースの設定
トレース用の 3 つの異なる種類のトレースフィルタが定義されています。これらのフィルタは、オプション/セットアップ/トレースのメニューを介して embOSView から設定できます。
フィルタ 0 は、タスク固有のものではなく、タスクに関わらず指定されたすべてのイベントを記録します。アイドルループはタスクではないので、アイドルループ内からの呼び出しはトレースされません。
フィルタ 1 は、割り込みすべてのサービスルーチン、ソフトウェアタイマ、および実行中のタスクの外で発生し呼び出しに固有のものです。これらの呼び出しは、アイドルループ中、あるいはタスクが実行されていない起動時に来るかもません。
フィルタ 2 から 4 は、指定されたタスクからの API 呼び出しのトレースを可能にします。
フィルタを有効または無効にするためには、単に Filter 4 Enable から Filter 0 Enable

と書かれたチェックボックスをオンまたはオフにするだけです。これらの 5 つのフィルタのいずれかについて、個々の API 関数は、リスト内の対応するチェックボックスをオンまたはオフにすることで、有効または無効にできます。プロセスをスピードアップするために、利用可能なボタンは２つあります。

・現在有効な（チェックされた）フィルタにすべての API 関数のトレース使用可能を選択する。

・現在有効な（チェックされた）フィルタのすべての API 関数のトレース不可能を選択解除する。

フィルタ 2、フィルタ 3、フィルタ 4 は、タスク固有の API 呼び出しのトレースを可能にします。よって、タスク名はこれらのフィルタをそれぞれ指定することができます。上記の例では、フィルタ 4 は MainTask というタスクから OS_Delay()の呼び出しをトレースするように構成されます。設定が（適用または OK ボタンを介して）保存された後、新しい設定はターゲットアプリケーションに送信されます。

22.7 トレースフィルタセットアップ関数

API やユーザー関数呼び出しのトレースは embOSView から開始または停止することができます。デフォルトでは、トレースは最初にアプリケーションプログラムで無効になっています。次の関数を使用して、アプリケーションから直接トレースイベントの記録を制御することは非常に便利かもしれません。

22.8 API 関数

Table 22.6: Trace filter API functions

22.8.1 OS_TraceEnable()

説明

フィルタのかかった API 関数のトレースを可能にします。

プロトタイプ

void　OS_TraceEnable　(void);

補足情報

トレースフィルタ条件は、この関数を呼び出す前に設定されている必要があります。この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しは、プリプロセッサによって削除されます。

▪ 22.8.2 OS_TraceDisable()

説明

API とユーザー関数呼び出しのトレースを無効にします。

プロトタイプ

void　OS_TraceDisable　(void);

補足情報

この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しは、プリプロセッサによって削除されます。

▪ 22.8.3 OS_TraceEnableAll()

説明

フィルタ 0（どのタスクでも）を設定し、すべての API 呼び出しのトレースを有効にし、トレース関数を有効にします。

プロトタイプ

void　OS_TraceEnableAll　(void);

補足情報

他のトレースフィルタの状態は影響を受けません。

この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しは、プリプロセッサによって削除されます。

▪ 22.8.4 OS_TraceDisableAll()

説明

フィルタ 0（どのタスクでも）を設定し、すべての API 呼び出しのトレースを無効にし、トレース関数を無効にします。

プロトタイプ

void　OS_TraceDisableAll （void);

補足情報

他のトレースフィルタの状態は影響を受けませんが、トレースは停止されます。

この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しは、プリプロセッサによって削除されます。

- 22.8.5　OS_TraceEnableId()

説明

フィルタ 0 で（どのタスクでも）指定された ID 値を設定し、指定された関数のトレースを有効にしますが、トレースを開始はしません。

プロトタイプ

void　OS_TraceEnableId （OS_U8　Id);

Table 22.7: OS_TraceEnableId() parameter list

補足情報

embOS の API 関数のトレースを有効にするためには、正しい ID 値を使用する必要があります。これらの値は RTOS.h でシンボリック定数として定義されます。こも関数はお使いのユーザー関数のトレースを有効にすることも可能です。この機能はトレースビルド内のみで使用可能であり、トレースビルド外では、API 呼び出しはプリプロセッサによって除かれます。

- 22.8.6　OS_TraceDisableId()

説明

フィルタ 0 で（どのタスクでも）指定された ID 値をリセットし、指定された関数のトレースを無効にしますが、トレースを停止はしません。

プロトタイプ

void　OS_TraceDisableId　(OS_U8　Id);

Table 22.8: OS_TraceDisabledId() parameter list

補足情報

特定の embOS API 関数のトレースを無効にするには、正しい ID 値を使用する必要があります。これらの値は RTOS.h.内のシンボリック定数として定義されています。この関数は、独自の関数のトレースを無効にするために使用することがでます。この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しはプリプロセッサによって削除されます。

- 22.8.7　OS_TraceEnableFilterId()

説明

指定されたとレースフィルタに指定された ID 値を設定し、指定された関数のトレースを有効にします。トレースを開始はしません。

プロトタイプ

void　OS_TraceEnableFilterId　(OS_U8　FilterIndex, OS_U8　Id)

Table 22.9: OS_TraceEnabledFilterId() parameter list

補足情報

特定の embOS API 関数のトレースを無効にするには、正しい ID 値を使用する必要があります。これらの値は RTOS.h.内のシンボリック定数として定義されています。この関数は、独自の関数のトレースを無効にするために使用することがでます。この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しはプリプロセッサによって削除されます。

- 22.8.8 OS_TraceDisableFilterId()

説明

指定されたトレースフィルタ内で指定された ID 値をリセットし、特定の関数のトレースを無効にします。トレースを停止させはしません。

プロトタイプ

void OS_TraceDisableFilterId (OS_U8 FilterIndex, OS_U8 Id)

Table 22.10: OS_TraceDisableFilterId() parameter list

補足情報

指定された embOS の API 関数のトレースをも高にするためには、正しい ID 値をご使用になる必要があります。この値は RTOS.h でシンボリック定数として定義されます。この関数はユーザー定義関数のトレースを無効にするためにも使用することができます。
この機能はトレースビルド内のみで使用可能です。トレースビルド外では、API 呼び出しはプリプロセッサによって除かれます。

- 22.9 トレース記録関数

下記の関数は、データをトレースバッファーに書き込む（記録する）ために使用されます。embOS の API 呼び出しのみが記録される必要がある場合に限り、これらの関数は内部で、トレースビルドライブラリによって使用されます。もし何らかの理由でユーザー定義パラメータを持つユーザー定義関数をトレースする必要がある場合、これらのルーチンのうち一つを呼び出すことができます。これらの関数には次のような共通点があります。:
・データを記録するには、トレースは有効にする必要があります。
・ID パラメータとして 100 から 127 までの ID 値はを使用する必要があります。0 から 99 までの ID 値は embOS 内部で予約されています。
ID として特定されるイベントはいかなるトレースフィルタ内においても有効にされる必要があります。アクティブシステムタイムおよび実行中のタスクは指定されたイベントとともに自動的に記録されます。

- 22.10 API 関数

Table 22.11: Trace record API functions

- 22.10.1 OS_TraceVoid()

説明

トレースバッファに、ID のみによって識別されるエントリを書き込みます。

プロトタイプ

```
void   OS_TraceVoid   (OS_U8   Id);
```

Table 22.12: OS_TraceVoid() parameter list

補足情報

この機能はトレースビルド内のみで使用可能であり、API 呼び出しはプリプロセッサによって除かれません。

- 22.10.2 OS_TracePtr()

説明

エントリを ID で、またポインタをパラメータでトレースバッファ内に書き込みます。

プロトタイプ

```
void   OS_TracePtr   (OS_U8   Id,
void*   p);
```

Table 22.13: OS_TracePtr() parameter list

補足情報

パラメータとして渡されるポインタは、embOSView のトレース一覧ウィンドウに表示されます。この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しはプリプロセッサによって削除されます。

- 22.10.3 OS_TraceData()

説明

エントリを ID で、また整数値をパラメータとしてトレースバッファ内に書き込みます。

プロタイプ

```
void   OS_TraceData   (OS_U8   Id,   int              v);
```

Table 22.14: OS_TraceData() parameter list

補足情報

パラメータとして渡されるポインタは、embOSView のトレース一覧ウィンドウに表示されます。この機能は、トレースビルドのみで利用可能です。非トレースビルドでは、API 呼び出しはプリプロセッサによって削除されます。

- 22.10.4 OS_TraceDataPtr()

説明

エントリを ID で、また整数値およびポインタをパラメータとしてトレースバッファ内に書き込みます。

プロトタイプ

```
void   OS_TraceDataPtr   (OS_U8   Id,
int          v,
void*   p);
```

Table 22.15: OS_TraceDataPtr() parameter list

補足情報

パラメータとして認識される値は embOS のトレースリストウィンドウに表示されます。

- 22.10.5 OS_TraceU32Ptr()

説明

エントリを ID で、またパラメータとして 32 ビットの符号なし整数およびポインタをトレースバッファ内に書き込みます。

プロトタイプ

```
void   OS_TraceU32Ptr  (OS_U8          Id,
OS_U32   p0,
void*       p1);
```

Table 22.16: OS_TraceU32Ptr() parameter list

補足情報

この関数は２つのポインタを記録するために使用されます。パラメータとして認識される値は embOSView のトレースリストウィンドウに表示されます。この機能はトレースビルドのみで使用可能です。トレースビルド以外では、API 呼び出しはプリプロセッサによって除かれます。

- 22.11 アプリケーション制御トレースサンプル

前のセクションで説明したように、ユーザアプリケーションは embOSView からの接続またはコマンドを指定せずにトレース条件を有効または設定することができます。トレース記録関数は、100 から 127 までの ID 番号を使用して、トレースバッファにデータを書き込むためにどのユーザー関数からでも呼び出すことができます。アプリケーションからのトレース制御は、アプリケーションを起動した直後で、embOSView へのコミュニケーションがまだ使用できない場合や embOSView のセットアップが完了していない場合に API とユーザ関数をトレースするのに非常に役立ちます。

以下の例では、トレースフィルタをアプリケーションによって設定する方法を示しています。関数 OS_TraceEnableID()は、実行中のタスクからの呼び出しに影響するトレースフィルタ 0 を設定します。

よって、その時点で実行中のタスクが存在しないため、この例では SetState()の最初の呼び出しはトレースされません。追加のフィルタ設定ルーチン OS_TraceEnableFilterId()がフィルタ 1 で呼び出され、外で実行中のタスクからの呼び出しをトレースします。

サンプルコード

```
#include  "RTOS.h"
#ifndef  OS_TRACE_FROM_START
#define  OS_TRACE_FROM_START  1
#endif
```

```
/*          Application specific trace id numbers          */
#define APP_TRACE_ID_SETSTATE  100
char MainState;
/*  Sample of application routine with trace  */
          void SetState(char* pState, char Value) {
          #if OS_TRACE
          OS_TraceDataPtr(APP_TRACE_ID_SETSTATE, Value, pState);
          #endif
          * pState = Value;
}
/*  Sample main routine, that enables and setup API and function call trace
from start          */
          void main(void) {
          OS_InitKern();
          OS_InitHW();
          #if (OS_TRACE && OS_TRACE_FROM_START)
          /*          OS_TRACE is defined in trace builds of the library    */
          OS_TraceDisableAll();          /*          Disable all API trace calls */
          OS_TraceEnableId(APP_TRACE_ID_SETSTATE);          /*
          User trace          */
          OS_TraceEnableFilterId(APP_TRACE_ID_SETSTATE);  /*
          User trace          */
          OS_TraceEnable();
          #endif
          /*          Application specific initilisation          */
          SetState(&MainState, 1);
          OS_CREATETASK(&TCBMain, "MainTask", MainTask, PRIO_MAIN, Mai
nStack);
          OS_Start();          /* Start multitasking          -> MainTask() */
          }
```

デフォルトでは、embOSView はとレース一覧ウィンドウのユーザー関数のトレースをルーチンとしてリスト化し、指定された ID および 16 進法値の 2 つのパラメータで追跡されます。上記のれいは次のような結果となります:

Routine100(0xabcd, 0x01)
0xabcd がポインタアドレスであり、 0x01 が OS_TraceDataPtr()に記録されたパラメータです。

- 22.12 ユーザー定義関数

アプリケーションプログラムのユーザ関数用に、組み込みのトレース（embOS のトレースビルドでも利用可能）を使用するために、embOSView をカスタマイズすることができます。このカスタマイズは、セットアップファイル embOS.ini で行われます。このセットアップファイルは embOSView の起動時に解析されます。これはオプションであり、もし見つからない場合には、エラーメッセージは表示されません。
ユーザ関数のトレースの設定を有効にするためには、embOSView は ID 番号、関数名およびトレース可能な 2 つのオプションパラメータのタイプを知る必要があります。形式は、次の embOS.ini ファイルのサンプルで説明いたします：

サンプルコード

```
# File: embOS.ini
#
# embOSView Setup file
#
# embOSView loads this file at startup. It has to reside in the same
# directory as the execuatble itself.
#
# Note: The file is not required to run embOSView. You will not get
# an error message if it is not found. However, you will get an error
message
# if the contents of the file are invalid.

#
# Define add. API functions.
# Syntax: API( <Index>, <Routinename> [parameters])
# Index: Integer, between 100 and 127
# Routinename: Identifier for the routine. Should be no more than 32
```

characters

parameters: Optional paramters. A max. of 2 parameters can be specified.

Valid parameters are:

int

ptr

Every parameter has to be preceeded by a colon.

#

API(100, "Routine100")

API(101, "Routine101", int)

API(102, "Routine102", int, ptr)

Chapter 23 性能及びリソースの使用

この章では embOS の性能およびリソースの使用について説明します。embOS の標準化に関する説明、およびほとんどのターゲットシステムの十分な評価に使用できる典型的なシステムのメモリ要求に関する情報があります。

23.1 導入

リソースの使用が少ない状況での高性能化は、常に主要な設計上の考慮事項となっています。 embOS は、8/16/32 ビット CPU で実行されます。どの機能が使用されているかによって、2 キロバイト以下の ROM、1 キロバイトの RAM を搭載したシングルチップシステムが embOS によってサポートされています。実際のパフォーマンスとリソースの使用率は、多くの要因（CPU、コンパイラ、メモリモデル、最適化、構成など）に依存します。

23.2 メモリ要求

embOS のメモリ要求 (RAM and ROM) は使用されているライブラリの機能によります。次の表では様々なモジュールでのメモリ要求を示しています。

Table 23.1: embOS memory requirements

* これらの値は 32 ビット CPU の典型的な値であり、ご使用の CPU やコンパイラ、ライブラリのモデルによって異なります。

23.3 性能

次のセクションでは得られたサンプルプログラムで embOS をどのように基準化するかを示しています。

23.4 基準化

embOS は、高速コンテキストスイッチを実行するように設計されています。このセクションでは、優先順位の低いタスクから優先度の高いタスクへのコンテキストスイッチの実行時間を計算するための 2 つの異なる方法を説明します。

最初の方法は、ポートピンを使用するもので、オシロスコープが必要です。

２つ目の方法は、高分解能の測定機能を使用しています。2 つの方法のサンプルプログラムが、お使いの embOS の\Sample ディレクトリに提供されています。

SEGGER は、embOS のパフォーマンスを基準化するために、これらのプログラムを使用しています。基準化の結果を評価するために、これらのサンプルを使用することができます。実際のパフォーマンスは多くの要因（CPU、クロック速度、ツールチェーン、メモリモデル、最適化、構成など）に依存することに注意してください。

サイクル数は、命令のと等しくないことに注意してください。 ARM7 の上の多くの命令はゼロの待ち状態で 2 または 3 サイクルを必要とします。例えば必要 LDR は 3 サイクルを必要とします。次の表は、メモリの種類と CPU モードによるコンテキストスイッチの時間変動の概要を説明します。

Table 23.2: embOS コンテキスト変換タイム

次のセクション内のすべてのサンプルパフォーマンス値は次のようなシステム構成で決定されています：

すべてのソースは IAR Embedded Workbench バージョン 5.40 でコンパイルされ、サムまたはアームモード、XR ライブラリと高レベルの最適化を使用しています。 embOS バージョン 3.82 が使用されており、値は異なるビルドごとに異なる場合があります。

23.4.1 ポートピン及びオシロスコープでの測定

サンプルファイル MeasureCST_Scope.c ではポートピンを設定しクリアするために LED.c module モジュールを使用しています。これにより、オシロスコープでコンテキストスイッチ時間を測ることができます。次のソースコードは MeasureCST_Scope.c の抜粋です:

```
#include "RTOS.h"
#include "LED.h"
static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK TCBHP, TCBLP;     // Task-control-blocks
/*****************************************************************
*
*          HPTask
*/
static void HPTask(void) {
while (1) {
OS_Suspend(NULL);          // Suspend high priority task
LED_ClrLED0();   // Stop measurement
}
}
/*****************************************************************
*
*          LPTask
*/
static void LPTask(void) {
while (1) {
OS_Delay(100);     // Syncronize to tick to avoid jitter
//
// Display measurement overhead
//
LED_SetLED0();
LED_ClrLED0();
//
// Perform measurement
```

```
  //
  LED_SetLED0();   // Start measurement
  OS_Resume(&TCBHP); // Resume high priority task to force task switc
h
  }
  }
  /*********************************************************************
  *
  *          main
  */
  int main(void) {
  OS_IncDI();          // Initially disable interrupts
  OS_InitKern();       // Initialize OS
  OS_InitHW();         // Initialize Hardware for OS
  LED_Init();          // Initialize LED ports
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,         99, StackLP);
  OS_Start();          // Start multitasking
  return 0;
  }
```

- 23.4.1.1 オシロスコープ分析

コンテキスト変換タイムは LED をオンからオフに変換する間のタイムです。LED がアクティブ高シグナルでオンにされた場合、コンテキスト変換タイムはシグナルのエッジが上ってからおりるまでのタイムです。LED がアクティブ低シグナルでオンにされた場合、シグナルの差が逆になります。シグナルは LED のオンとオフの変換全般を含んでいるため、実際のコンテキスト変換タイムはより短いです。このタイムは、タスク変換タイムの表示の前に小さいピークライトとしても表示され、表示されたコンテキスト変換タイムから引かれる必要があります。下記の図では単純なオシロスコープのシグナルをアクティブ低 LED シグナル（低いとは LED が点灯しているということです）で示しています。
これらは決定すべき変換ポイントです:
·A = LED は全体の測定がオンにされています

·B = LED は全体の測定がオフにされています
·C = LED は優先順位の低いタスクのコンテキスト変換直前にオンにされています
·D = LED は優先順位の低いタスクのコンテキスト変換直前にオフにされています
サブルーチンで LED をオン、オフに変換するために必要な時間はタイム tAB として記録されます。サブルーチンで LED をオン、オフに変換するのに必要なタイムを含む完全なコンテキスト変換はタイム tCD として記録されます。
コンテキスト変換タイム tCS は次のように計算されます:
tCS = tCD - tAB

- 23.4.1.2 サンプル測定 AT91SAM7S、RAM の ARM コード

タスク変換タイムは以下の一覧のパラメータで計算されます:
embOS Version V3.82
アプリケーションプログラム: MeasureCST_Scope.c
ハードウェア: AT91SAM7SE512 processor with 48MHz
プログラムは RAM で実行されます
ARM モードが使用されます
使用されるコンパイラ: IAR V5.40
CPU 周波数 (fCPU): 47.9232MHz
CPU クロックサイクル (tCycle):
tCycle = 1 / fCPU = 1 / 47.9232MHz = 20,866ns
tAB と tCD の測定

tAB は 312ns と計算されます。
サイクル数は次のように計算されます:
CyclesAB = tAB / tCycle
=332ns / 20.866ns
= 15.911Cycles
=> 16Cycles

tCD は 4420.0ns として計算されます
CyclesCD = tCD / tCycle
= 4420.0ns / 20.866ns

= 211.83Cycles

=> 212Cycles

コンテキスト変換タイム及びサイクル数を結果として表示します。

単なるコンテキストスイッチに必要な時間は:

tCS = tCD - tAB = 212Cycles - 16Cycles = 196Cycles

=> 196Cycles (4.09us @48MHz).

- 23.4.1.3 サンプル測定 AT91SAM7S、 FLASH でのサムコード

タスクスイッチングタイムは下記の一覧のパラメータで測定されています。:

embOS バージョン V3.82

アプリケーションプログラム: MeasureCST_Scope.c

ハードウェア: AT91SAM7E512 processor with 48MHz

Program is executing in FLASH

サムモードが使用されている

使用コンパイラ: IAR V5.40

CPU 周波数 (fCPU): 47.9232MHz

CPU クロックサイクル (tCycle):

tCycle = 1 / fCPU = 1 / 47.9232MHz = 20,866ns

tAB および tCD の測定

tAB は 436.8ns として測定されます

サイクルの数は次のように計算されます:

CyclesAB = tAB / tCycle

=416.0ns / 20.866ns

= 19.937Cycles

=> 20Cycles

tCD は 7250ns として計算されている.

サイクル数は次のように計算されます:

CyclesCD = tCD / tCycle

= 7250ns / 20.866ns

= 347.46Cycles

=> 347Cycles

コンテキストスイッチングタイムおよびサイクルの数の結果算出
単純なコンテキストスイッチに必要な時間は:
tCS = tCD - tAB = 347Cycles - 20Cycles = 327Cycles
=> 327Cycles (6.83us @48MHz).

- 23.4.1.4 高解像度タイマでの測定

コンテキストスイッチタイムは高解像度タイマで測定することができます。embOS の高解像度測定の詳細については 247 ページの高解像度測定のセクションをご参照ください。サンプル測定 CST_HRTimer_embOSView.c では、低優先度タスクから高優先度タスクまでのコンテキストスイッチタイムを測定し embOSView に結果を表示するために高解像度タイマを使用しています。

```
#include "RTOS.h"
#include "stdio.h"
static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK TCBHP, TCBLP;     // Task-control-blocks
static OS_U32 _Time;         // Timer values
/*********************************************************************
*
*          HPTask
*/
static void HPTask(void) {
while (1) {
OS_Suspend(NULL);          // Suspend high priority task
OS_Timing_End(&_Time);     // Stop measurement
}
}
/*********************************************************************
*
*          LPTask
*/
```

```
static void LPTask(void) {
char       acBuffer[100];       // Output buffer
OS_U32 MeasureOverhead; // Time for Measure Overhead
OS_U32 v;
//
// Measure Overhead for time measurement so we can take
// this into account by subtracting it
//
OS_Timing_Start(&MeasureOverhead);
OS_Timing_End(&MeasureOverhead);
//
// Perform measurements in endless loop
//
while (1) {
OS_Delay(100);     // Sync. to tick to avoid jitter
OS_Timing_Start(&_Time);  // Start measurement
OS_Resume(&TCBHP);
// Resume high priority task to force task switch
v = OS_Timing_GetCycles(&_Time) -
 OS_Timing_GetCycles(&MeasureOverhead);
v = OS_ConvertCycles2us(1000 * v);  // Convert cycles to nano-seconds
sprintf(acBuffer, "Context switch time: %u.%.3u usec\r", v / 1000, v %
1000);
OS_SendString(acBuffer);
}
}
```

サンプルプログラムでは自身の測定を計算し減算するので、これを行う必要はありません。結果は embOSView に転送されるので、UART から embOSView への通信をサポートするそれぞれのターゲットでサンプルが実行されます。サンプルプログラム MeasureCST_HRTimer_Printf.c はサンプルプログラム MeasureCST_HRTimer_embOSView.c と同等ですが、終端結果エミュレーションをサポートするデバッガーでは 関数 printf() で結果を表示します。

Chapter 24　デバッグ

24.1　実行時間エラー

実行時間中に発見されるエラー状態もあります。それらは:

·初期化されていないデータ構造の使用

·無効なポインタ

·過去そのタスクに使用されたことのないリソース

·OS_LeaveRegion() が OS_EnterRegion()よりも頻繁に呼び出される

·スタックオーバーフロー (この機能が使用可能でないプロセッサもあります)

どの実行時間エラーが発見されるかはチェックがどの程度行われているかによります。残念ながら、チェックを追加するとメモリ及びスピードを消費します（さほど重要ではありませんが、違いはあります)。embOS が実行時間エラーを発見した場合、次のようなルーチンを呼び出します:

```
void OS_Error(int ErrCode);
```

このルーチンはモジュール OS_Error.c の一部としてのソースコードとして提供されます。これは単に次のように割り込みや無限ループを有効にした後これ以上のタスク変換を無効にします。:

例

```
/*
Run time error reaction
*/
void OS_Error(int ErrCode) {
OS_EnterRegion(); /* Avoid further task switches */
OS_DICnt =0;      /* Allow interrupts so we can communicate */
OS_EI();
OS_Status = ErrCode;
while (OS_Status);
}
```

embOSView をご使用の場合、システム変数ウィンドウで

OS_Status の値と意味をご覧頂けます。エミュレーターを使用する場合、このルーチンの開始時点でブレークポイントを設定するか、失敗のあとにプログラムを停止する必要があります。
エラーコードは関数にパラメータとして渡されます。お使いのハードウェアを調整するためにルーチンを変更することができ、つまり目的のハードウェアはエラーを示す LED を設定するかディスプレイにメッセージを表示します。;
注: OS_Error()ルーチンを変更する場合、最初のステートメントは OS_EnterRegion()を介してスケジューラを無効にする必要があります。最後のステートメントは無限ループである必要があります。
OS_Error() ルーチンをご覧になると、必要以上に複雑であることがお分かりになるでしょう。実際のエラーコードはグローバル変数 OS_Status に割り当てられます。プログラムはこの変数がリセットされるのを待機します。単にサーキットエミュレーターでこの変数を 0 にリセットすると、問題の原因となっている一連のプログラムに簡単に戻ることができます。ほとんどの時間、プログラムの一部を見ると問題が明確に分かります。

- 24.1.1 OS_DEBUG_LEVEL

定義 OS_DEBUG_LEVEL では embOS のデバッグレベルを定義します。デフォルトの値は 1d です。高デバッグレベルにはより多くのデバッグコードが含まれています。デバッグレベル 2 は OS_RegionCnt が容量を超えているかどうかをチェックします。

- 24.2 エラーコード一覧

Table 24.1: Error code list

定義されたエラーリストの最新のバージョンは、ソースファイル OS_Error.c での OS_Error()関数宣言の直前のコメントの一部です。

- Chapter 25 サポートされた開発ツール

- 25.1 概要

embOS は選択されたターゲットプロセッサのための特定の C コンパイラバージョンで開発されています。詳細は RELEASE.HTML ファイルをご参照ください。他のコンパイラ

は異なる呼び出しのコンバージョン（目的のファイルフォーマットとは相容れない）を使用するので、embOS は指定された C コンパイラのみで実行されされます。しかし、異なる C コンパイラをご使用になりたい場合は、ご連絡いただければできる限り早い時間でご希望に対応できるよう努力いたします。

再登録

同時に異なるタスクから使用できる全てのルーチンは完全に再登録される必要があります。ルーチンは、呼び出された時点から返されるまで、またはタスク終了が呼び出されるまで使用されます。お使いの RTOS システムでサポートされる全てのルーチンは完全に再登録されます。何らかの理由で、一つ以上のタスクから使用されるプログラムで再登録されていないルーチンを持つ必要がある場合、このような問題を避けるためにリソースセマフォをご使用になることが推奨されます。

C ルーチンおよび再登録

通常、C コンパイラは完全に再登録されコードを生成します。しかし、コンパイラが無理矢理再登録されていないコードを生成するオプションも存在します。これらのオプションを使用することはは特定の状況下では可能ですが、推奨されません。

アセンブリルーチンおよび再登録

これらのアセンブリ関数がローカル変数及びパラメータのみにアクセスする場合に限り、すべて完全に再登録されます。他のものにはすべて注意する必要があります。

▪ Chapter 26　制限

embOS には次のような制限が存在します。:

Max. no. of tasks:	limited by available RAM only
Max. no. of priorities:	255
Max. no. of semaphores:	limited by available RAM only
Max. no. of mailboxes:	limited by available RAM only
Max. no. of queues:	limited by available RAM only
Max. size. of queues:	limited by available RAM only
Max. no. of timers	limited by available RAM only

Task specific Event flags: 8 bits / task

可能な関数の追加に関するフィードバックを歓迎いたします。またそれらの関数が目的にかなう場合は実行するためにベストを尽くします。
遠慮なくご連絡ください。embOS への変更が必要である場合、すべてのソースコードがご使用いただけます。

- Chapter 27 カーネル及びライブラリのソースコード

- 27.1 導入

sembOS は 2 つのバージョンで使用可能です:

1. オブジェクトバージョン: Object code + hardware initialization source.
2. フルソースバージョン: Complete source code.

この説明書ではオブジェクトバージョンについて説明しているので、内部データ構造の詳細は説明していません。オブジェクトバージョンは、サポートされたコンパイラ、上記のデバッグライブラリ、およびアイドルタスクとハードウェアの初期化のソースコードのメモリモデルを含む embOS の機能を提供しています。しかし、オブジェクトバージョンではライブラリルーチン及びカーネルのソースレベルでのデバッグを許可していません。
フルソースバージョンでは無制限のオプションを提供できます: embOS は、様々なデータサイズで再コンパイルすることができます。様々なコンパイルオプションで、システムを様々な用途のために最適化したりメモリ要求を最小化したりすることを可能にし、生成されたコードを完全に成業することができます。システム全体をデバッグしたり新しいメモリモデルや他の CPU に変更したりできます。embOS のソースコード配分には次のような追加のファイルが含まれています:

· CPU フォルダは、embOS の設計に使われる全ての CPU 及びコンパイラに固有のソースコード及びヘッダーファイルを含んでいます。また、embOS のデモプロジェクト用に Start フォルダに提供されるサンプルスタートプロジェクト、ワークスペース、及びソースファイルも含んでいます。通常、CPU フォルダにあるファイルはどれも変更することはできません。

· GenOSSrc フォルダは、次のセクションで説明するバッチモードでコンパイルするために使用されるすべての embOS ソースおよびバッチファイルを含んでいます。

- 27.2 embOS ライブラリの設計

embOS ライブラリは embOS のソースコードバージョンを購入された場合に限り作成されます。embOS のルートパスでは、DOS のバッチファイル PREP.BAT があり、これはお使いの C コンパイラ のインストールディレクトリと合致するように変更する必要があります。一度これを行うと、バッチファイル M.BAT を呼び出して全ての embOS ライブラリを作成することができます。

注:M.bat を使用して embOS ライブラリを再ビルドすると、Start フォルダ全体が消去され再ビルドされます。Start フォルダにあるご自身のプロジェクトを変更したり再ビルドする場合は、embOS を再ビルドする前にお使いの Start フォルダのコピーをお取りください。ビルドプロセスはエラーや警告メッセージなしで実行される必要があります。ビルドプロセスが何か問題を報告した場合、次のことを確認してください:

· ファイル RELEASE.HTML で言及したものと同じコンパイラバージョンを使用していますか?

· PREP.BAT を実行した後に単純なテストファイルをコンパイルすることができ、それは本当に指定したコンパイラバージョンを使用していますか?

· RELEASE.HTML にある可能なコンパイラの警告について言及されていますか?

それでも何か問題がある場合には、ご一報ください。

ビルドプロセス全体は、お使いのソースコード配分のルートディレクトリにあるいくつかのバッチファイルとともに制御されています:

· Prep.bat: コンパイラ、アセンブラ、及びリンカーの環境を設定します。このファイルがお使いのコンパイラに必要なパスおよび追加ディレクトリを設定することを確認してください。通常、このバッチファイルは唯一 embOS ライブラリをビルドするために変更する必要があります。通常、このファイルはすべてのライブラリのビルドプロセス内で M.bat ファイルから呼び出されます。

· Clean.bat: embOS のライブラリビルドプロセスの全ての出力を消去します。新しいライブラリが生成される前に、ビルドプロセス内で自動的に呼び出されます。通常、Start フォルダを消去します。よって、このバッチファイルを間違って呼び出してしまわないように注意してください。通常、このファイルは全てのライブラリのビルドプロセス中に最初に M.bat で呼び出されます。

· cc.bat: このバッチファイルはコンパイラを呼び出し、一つの embOS ソースファイルをデバッグ情報の出力なしにコンパイルするために使用されます。ほとんどのコンパイラオプションはこのファイルで定義されており、通常は変更できません。ご使用目的のために、デバッグ出力をアクティブにしたり、最適レベルを変更したりすることもできます。変更を行う際には注意してください。通常、このファイルは embOS の内部バッチファイル

CC_OS.bat から呼び出され、直接呼び出すことはできません。

· ccd.bat: このバッチファイルはコンパイラを呼び出しすべてのグローバル関数を含む OS_Global.c をコンパイルするために使用できます。すべてのコンパイラ設定は、embOS ライブラリを使用する際にグローバル変数のデバッグを有効にするためにデバッグ出力がアクティブ化される以外は、cc.bat で使用されているものと同等です。通常、このファイルは embOS の内部バッチファイル CC_OS.bat から呼び出され、直接呼び出すことはできません。

· asm.bat: このバッチファイルはアセンブラを呼び出し、通常タスク変換機能を含む embOS のアセンブパートをアセンブルするのに使用されます。通常、このファイルは embOS の内部バッチファイル CC_OS.bat から呼び出され、直接呼び出すことはできません。

· MakeH.bat: CPU/コンパイラ固有部分 OS_Chip.h および総称部分 OS_RAW.h で作成される embOS のヘッダファイル RTOS.h をビルドします。通常、RTOS.h はサブフォルダ Start\Inc の出力です。

· M1.bat: このバッチファイルは M.bat から呼び出され、ある固有の embOS ライブラリをビルドするのに使用され、直接呼び出すことはできません。

· M.bat:このバッチファイルはすべての embOS ライブラリを生成するために呼び出される必要があります。これは最初に Clean.bat を呼び出し、よって Start フォルダ全体を消去します。生成されたライブラリはスタートプロジェクトやライブラリ、ヘッダ、サンプルスターとプログラムを含む新しい Start フォルダに配置されます。

▪ 27.3 主要なコンパイルタイム変換

embOS の多くの機能はコンパイルタイム変換によって変更されます。これらは全て事前に embOS の配分の適当な値に定義されています。

コンパイルタイム変換は RTOS.h で変更することはできません。embOS の何らかの機能を変更するためにコンパイルタイム変換を行う必要がある場合、その変換は OS_RAW.h 内で行われるかライブラリビルドプロセス中にパラメータとして認識される必要があります。embOS ソースは再コンパイルされる必要があり、RTOS.h はスイッチとともに再ビルドされる必要があります。

▪ 27.3.1 OS_RR_SUPPORTED

このスイッチはラウンドロビンスケジューリングアルゴリズムがサポートされているかどうかを定義しています。embOS のすべてのバージョンでは、デフォルトでラウンドロビ

ンスケジューリングが有効にされています。ラウンドロビンスケジューリングをご使用にならずにすべてのタスクををれぞれの優先順位で行う場合、このスイッチを 0 に定義することでラウンドロビンスケジューリングを無効にすることができます。これにより、RAM と ROM を節約しタスク変換プロセスの速度を上げることができます。ラウンドロビンスケジューリングを無効にしている時にお使いのタスクが同じ優先順位にないことを確認してください。このコンパイルタイム変換は RTOS.h で変更することはできません。embOS ライブラリが再ビルドされる前に OS_RAW.h で変更される必要があります。

- 27.3.2 OS_SUPPORT_CLEANUP_ON_TERMINATE

このコンパイルタイム変換は embOS のバージョン 3.26 以降の新しいものです。有効になると、リソースセマフォを要求したり同期目的で中断されるタスクの終了を許可します。
注:デフォルトでは、この変換は 16 および 32 ビット CPU でオンになります。8 ビット CPU ではオフになります。
オーバーヘッドが最小で実行時間が重大な影響を及ぼされない場合であっても、お使いのアプリケーションでタスクを終了しない場合にはこのスイッチをゼロに定義することができます。また、アプリケーションの保証があれば、そのタスクは同期オブジェクトで中断されたり終了時にリソースセマフォを呼び出したりすることはありません。
このスイッチを無効にするとタスク制御構造で RAM をいくらか節約することになり、同期オブジェクトの待機関数の速度を上げることになります。
8 ビット CPU をお使いの場合、同期オブジェクトで中断されたりリソースセマフォを要求したりするタスクの終了を有効にするために、この変換（0 ではない値に定義する）を可能にする必要があります。
このコンパイルタイム変換は RTOS.h で変更することはできません。OS_RAW.h 内行うか、あるいは embOS ライブラリが再ビルドされるビルドプロセス内で定義として認識される必要があります。

- Chapter 28 FAQ (frequently asked questions、よくある質問)

Q:異なる優先順位のスケジューリングアルゴリズムを実装することはできますか？
A:はい、システムは完全に動的であり、システム実行中でも(OS_SetPriority()を使用して)タスクの優先順位は変更できます。この機能は、基本的に全ての望まれるアルゴリズムが

実装できるような方法で優先順位を変更するために使用できます。ひとつの方法は、動的に優先順位を変更する他の全てのタスクよりも高い優先順位のタスク制御タスクを持つことです。通常、優先順位制御ラウンドロビンアルゴリズムはリアルタイムアプリケーションに対応しています。

Q: embOS の異なる割り込みソースを使用することはできますか？
A: はい。どの定期シグナルでも、つまりどの内部タイマ使用することができますが、外部シグナルでも使用することができます。

Q: プログラム使用の割り込みのために、いずれの割り込み優先順位を使用すればいいでしょうか？
A: いずれでも問題ありません。

- Chapter 29　用語解

協力マルチタスキング
それぞれのタスクが CPU を終了するまで実行を許可されるスケジューリングシステム; ISR は優先順位の高いタスクを READY 状態にすることができますが、割り込みされたタスクは返され最初に終了します。

カウンティングセマフォ
複数のリソースを追跡するセマフォのタイプ。タスクが一回より多くシグナルを送られるものを待機する必要があるときに使用されます。

CPU
セントラルプロセシングユニット（Central Processing Unit)。マイクロコントローラの「脳」にあたります。；プロセッサの指示を出す部分です。

危険領域
割り込みなしで実行される必要があるコードの領域。

イベント
シグナルに送信されるメッセージで、何かが起きた時に指定されるタスク。タスクは

READY 状態になります。

割り込みハンドラ
割り込みサービスルーチン。ルーチンは割り込みが認識された時点でプロセッサにより自動的に呼び出されます。ISR はタスクのコンテキスト全体（すべてのレジスタ）を保存する必要があります。

ISR
割り込みサービスルーチン。ルーチンは割り込みが認識された時点でプロセッサにより自動的に呼び出されます。
ISR はタスクのコンテキスト全体（全てのレジスタ）を保存する必要があります。

メールボックス
RTOS により管理されるデーバッファーで、タスクまたは割り込みハンドラにメッセージを送信するために使用されます。

メッセージ
データの項目（メールボックス、キュー、あるいは他のデータのコンテナに送信されます)。

マルチタスキング
互いに独立な複数のソフトウェアルーチンの実行。OS は異なるルーチン（タスク）が同時に起こるようにプロセッサのタイムを分割します。

NMI
マスク不可能割り込み（Non-Maskable Interrupt)。ソフトウェアによってマスクできない（無効にできない）割り込み。
例: ウォッチドッグタイマ割り込み

プリエンプティブマルチタスキング
常に優先順位の高いタスクが実行されるスケジューリングシステム。ISR が優先順位の高いタスクを READY 状態にする場合、そのタスクは割り込まれたタスクが返される前に実行されます。

プロセス
プロセスは、自身のメモリを配置するタスクです。2 つのプロセスは通常同じメモリ位置にはアクセスできません。一般的に、異なるプロセスは異なるアクセス権限を持ち、（NMU の場合）異なる翻訳テーブルを持ちます。

プロセッサ
マイクロプロセッサの略。コントローラーの CPU コア。

優先順位
あるタスクの別のタスクに対する相対的な重要度。RTOS にあるそれぞれのタスクは優先順位を持ちます。

優先順位の逆転
優先順位の低いタスクに使用されているシェアされたリソースへのアクセスを待機している間に優先順位の高いタスクが遅延する状況。優先順位の低いタスクは、リソースを解放するまで一時的に優先順位が高くなります。

キュー
メールボックスと同様ですが、より大きなメッセージやそれぞれ別のサイズのメッセージを、タスクや割り込みハンドラに送信するために使用されます。

Ready
“ready 状態” にあるいずれのタスクも他のより優先順位の高いタスクが“ready 状態” にない場合アクティブになります。

リソース
コンピュータシステム内にあり使用を制限されているもの（例えば、メモリ、タイマ、計算タイム）。本来は、タスクに使用されるもの。

リソースセマフォ
一つのタスクのみが一度に一つのリソースにアクセスすることを保証することによって、リソースを管理するために使用されるセマフォのタイプ。

RTOS
リアルタイムオペレーティングシステム

ランニングタスク
与えられた時間には、一つタスクのみが実行可能です。現在実行しているタスクはランニングタスクと呼ばれます。

スケジューラ
どのタスクが ready 状態にあるか、相対的な優先順位、および使用されているスケジューリングシステムに基づき、アクティブのタスクを選択する RTOS のプログラムセクション。

セマフォ
タスクの同期のために使用されるデータ構造。

ソフトウェアタイマ
特定の遅延のあと、ユーザー固有ルーチンを呼び出すデータ構造。

スタック
LIFO のパラメータ、自動変数、戻りアドレス、および関数呼び出しで維持される必要がある情報を格納するメモリのエリア。

スーパーループ
無限ループの中で実行されリアルタイムカーネルを使用しないプログラム。ISR はソフトウェアのリアルタイムパートに使用されます。

タスク
プロセッサ上で実行されるプログラム。マルチタスキングシステムは複数のタスクが互いに独立で実行することを許可します。

スレッド
スレッドは、同じメモリのレイアウトを共有するタスクです。2 つのスレッドは同じメモリ位置にアクセスできます。仮想のメモリが使用される場合、同じ仮想メモリから実在の

メモリへの変換およびアクセス権限が使用されます。

(-> Thread, Process)

ティック

OS タイマ割り込み。通常 1 ms。

タイムスライス

ラウンドロビンタスク変換が起こるまでにタスクが実行されるタイム（ティックの数）