RENESAS

HW-RTOS

Real Time OS in Hardware

従来 RTOS の リアルタイム性能面における課題と HW-RTOS による性能向上

組込みシステムでは、マルチタスク環境を実現するため RTOS を利用することが多い。しかし、従来のソフトウェア RTOS を導入した結果、要求されるリアルタイム性能を満足できなくなってしまうことがある。これは RTOS のオーバヘッドや割込み禁止期間によるものである。本ホワイトペーパーではオーバーヘッドや割り込み禁止期間が発生するメカニズムを明確にし、測定により確認を行った。また HW-RTOS でも同様な測定を行い、極めて良好な結果を得た。

R70WP0002JJ0100　2018年9月

1. 概要

リアルタイム組込みシステムにリアルタイム OS（RTOS）を導入するとき、そのオーバヘッドによる影響が問題になることがある。1 ミリ秒以上のオーバーヘッドを許容できるシステムであればほとんどの場合問題ないが、マイクロ秒オーダーのオーバーヘッドが問題であるとき、チューニングをしたり、あるいは RTOS の利用をあきらめたりする場合もある。本ホワイトペーパーでは RTOS がシステムのリアルタイム性能を劣化させる原因を特定し、測定によりこれを確認した。測定結果は一般の組込みソフトウェア技術者が考えている以上にリアルタイム性能への影響が大きいことが確認でき、また最悪値を保証することが極めて難しいことが理解できる。一方、HW-RTOS でも同じ測定を行い、リアルタイム性能に与える影響がほとんどないことが確認できた。HW-RTOS では、最悪値を設計時に定義できることから、ソフトウェア開発者の設計負担を軽減し、容易にリアルタイムシステムを開発し、またその性能を保証することができることもあわせて説明する。

2. RTOS の性能を決める要因

2.1. RTOS の利便性と不都合

組込みシステムにおいて RTOS が多く使用されているが、この理由は以下の通りである。RTOS を利用することにより、マルチタスク環境が簡単に得られ、またセマフォやイベントフラグを利用して、簡単にタスク間同期・タスク間通信を容易に実現することができるようになる。結果的にソフトウェアの部品化・再利用化が容易になり、ソフトウェア開発の生産性が向上し、またシステムの信頼性も向上する。

しかし、RTOS を導入してみるとシステムとしての目標の性能を達成できないことがある。一つは RTOS を導入した途端「重くなった」と感じることである。製品としての機能は問題ないがとにかく遅い、あるいは動きが重いと言うことである。もう一つは必須とされる時間的条件を満足できなくなってしまったということである。これは要するに要求されるリアルタイム性能を満足できていない言うことである。

これらの原因を検討すると 2 つの原因に行き着く。前者の原因は RTOS のオーバヘッドであり、後者の原因は RTOS が発生する割込み禁止期間である。逆に言うとこれら 2 つが RTOS の性能を決める要因である。以下、これらについてもう少し詳しく検討する。

2.2. RTOS のオーバヘッド

RTOS は CPU 上で動作するソフトウェアの一部である。すなわち RTOS とアプリケーションソフトウェアは CPU をシェアする。したがって、RTOS の CPU 占有率が高くなればなるほどアプリケーションの動作が「重くなった」と感じることになる。全体に対する RTOS の占有率が多いか少ないかはアプリケーションの種類に依存する。例えばネットワークプロトコル処理は RTOS の機能を頻繁に利用する。これはネットワークプロトコルがマルチタスクで実現され、これらのタスクが RTOS のタスク間通信・タスク間同期を頻繁に利用するためである。したがって、ネットワークプロトコル処理を行っているときは CPU の RTOS 占有率が高くなる。Low エンドプロセッサを使用するとネットワークのスループットが中々上がらないと感じたことはないだろうか。これは RTOS のオーバヘッドに起因するものである。

以上のように RTOS はできるだけそのオーバヘッドが少ないことが望ましい。「3. 従来のソフトウェア RTOS の性能」では、具体的にどのくらいのオーバヘッドであるか、測定結果を示す。

2.3. RTOS による割込み禁止期間

まず、リアルタイムとはなにかについて述べる。「ある一定時間以内に、何らかの反応を要求するシステム」をリアルタイムシステムという。「ある一定時間」はそのアプリケーションにより異なる。たとえばテレビのリモコンの反応速度は 100m 秒であれば十分であろうが、電子オルガンで鍵盤をたたいてから音が出るまで 100m 秒では遅すぎると考えられる。また、サーボモータ制御では極めて高い反応速度を求められる。高速なサーボ制御では、1 マイクロ秒以下の反応速度を求められるものもある。

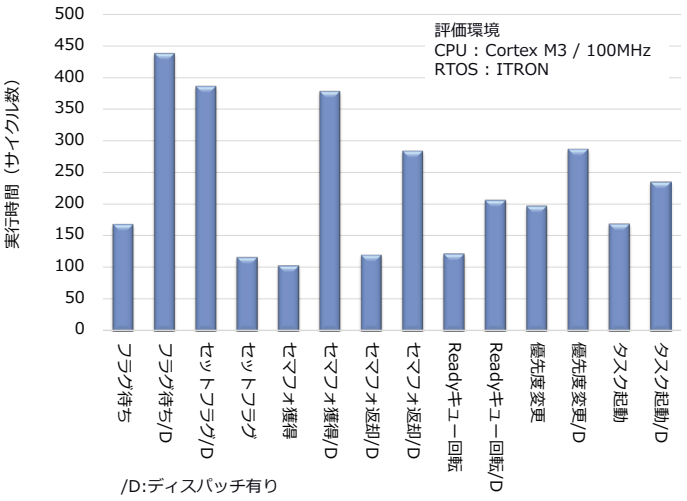

図3-1　システムコール実行時間

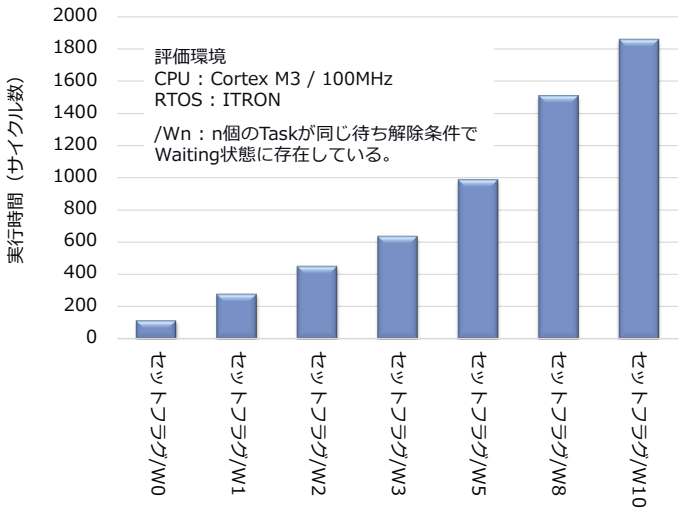

図3-2　システムコール実行時間

RTOS とはその名の通り、リアルタイムシステムにおいて利用可能な OS である。しかし、実際には高い反応時間を必要とするシステムでは RTOS を利用することが困難である。一般的に RTOS を実装した場合、問題なく動作できる反応速度は 100 マイクロ秒程度である。

この理由は、RTOS による割り込み禁止期間である。RTOS の処理は一貫性を必要とする。したがって RTOS 実行中のほとんどの期間は割り込み禁止期間になる。このため RTOS を導入すると反応速度が劣化する。この割り込み禁止期間であるが、一定であればそれを見越してシステムを構築することも可能であろう。しかし、RTOS の内部の状態により、割り込み禁止期間は変動する。さらに割り込み禁止期間の最悪値を求めることは大変難しい。「3. 従来のソフトウェア RTOS の性能」では、割り込み禁止期間の測定値を示すと共に、RTOS の構造の説明を通して、割り込み禁止期間の最悪値を求めることが大変難しいことを示す。

3.　従来のソフトウェア RTOS の性能

「RTOS によるオーバヘッド」は RTOS が動作している時間に相当する。RTOS が動作している時間とは、システムコールを実行している時間と割り込みが発生したとき、割込みハンドラを立ち上げるまでに RTOS が動作した期間、および Tick 処理が動作している期間であり、これが RTOS のオーバヘッドとなる。

一方「RTOS による割り込み禁止期間」であるが、これは上記 RTOS が動作している期間のほとんどが割込み禁止期間となる。しかし、所々で割込みを許可させることをすることにより割込み禁止期間を短くする工夫をしている RTOS が多いようである。ただし、Tick 処理や、内部のキューを操作している期間はかなり長期にわたり割込みを禁止せざるを得ない。

以上から、この章ではシステムコール実行時間の測定結果について述べ、さらに Tick 処理や、内部のキューを操作している期間のオーバヘッドと割込み禁止状態について詳しく言及する。

3.1.　システムコール実行時間

図 3-1 は ITRON のシステムコール実行時間を測定したものである。/D は、システムコール実行の結果、ディスパッチを伴うケースを示している。図 3-1 は静的な状態での測定を示している。しかし、RTOS の実行時間はそのときの内部状態に依存し、ダイナミックに変動する。図 3-2 はこの例を示している。図はセットフラグシステムコールを実行したときの実行時間であるが、同じイベントを待っているタスクの数によってシステムコール実行時間が大きく異なっている。この理由は以下の通りである。

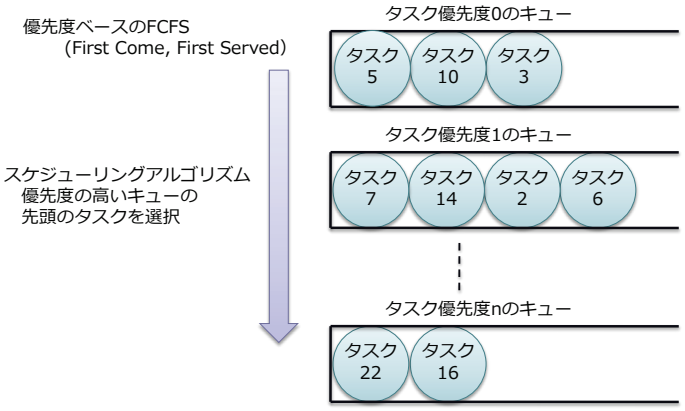

図3-3　Waitキュー

図 3-3 は RTOS で利用されている Wait キューを示しており、優先度ベースの FCFS (First Come, First Served)と呼ばれるアルゴリズムを実現している。具体的には、各オブジェクトがタスクの優先度毎にキューを持っている（ここでは優先度 0 が最高優先度、優先度 n が最低優先度と仮定する)。例えば、セマフォを実現するためにセマフォ識別子毎に n 個のキューを実装する。セマフォを獲得したいタスクはこのキューで待つ。あるタスクがセマフォを解放すると優先度の一番高いキューの先頭のタスクが選択され、このタスクにセマフォが渡される。

セットフラグシステムコールの場合は、それだけの処理に留まらない。フラグ処理の場合、待っているフラグパターンと、セットされたあとのイベントテーブルのフラグパターンを比較し、待っているタスクの条件を満たしているかどうか判断する必要がある。したがって、RTOS は図 3-3 のキューを先頭から順次サーチし、フラグパターンを比較する処理を行わなければならない。このため、図 3-2 に示すようにセットフラグシステムコールの場合は、同じフラグで待っているタスクの数に比例して処理時間が大幅に増える。このようにシステムコール実行に伴う RTOS のオーバヘッドは、RTOS のそのときの内部状態に応じて動的に変化することがわかる。

一般に設計者はシステムコールを実行してせいぜい数 100 クロック後には結果を得ることができると思っている。しかし、ある状況が重なるとこのように大幅な時間を消費していると言うことは理解していないのが現状である。したがって、何らかの状況が重なって、上記の様にオーバヘッドが急速に増加した場合、ある一定の時間内に処理が終了できなくなり、リアルタイムシステムとしての破綻を期することもあり得る。

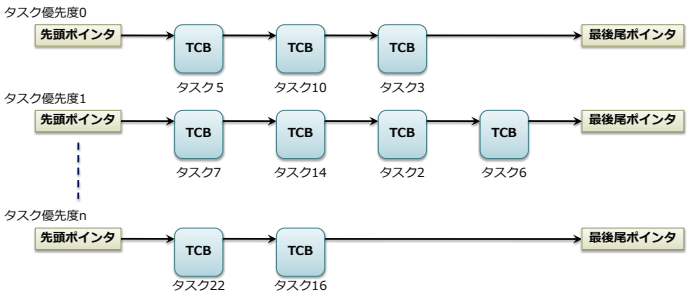

図3-4　Waitキューの実現方法 (1)

3.2. キュー処理の影響

図 3-3 で Wait キューの論理構造について説明したように、RTOS ではオブジェクト毎に優先度キューが必要である。あるオブジェクトに注目し、どのようにキューを実現しているかを説明する。

一般的にソフトウェアではキューを実現するためにリスト構造を使用する。RTOS では TCB をリスト構造に接続することにより Wait キューを実現する（TCB : Task Control Block。各タスクのための情報の集合体であり、タスク識別子毎に存在する)。図 3-3 のキューは図 3-4 のように実現できる。

Wait キューからタスクを取り出すときは、一番優先度の高いキューの先頭のタスクを取り出す。この図ではタスク 5 になる。この図の場合、「優先度 0」にタスクが接続されているが、優先度 0 のキューにはタスクがない場合がある。したがって、ソフトウェアは優先度の高いキューから順にタスクが接続されているかどうかをチェックして、接続されている優先順位を探し当てなければならない。したがって、そのときのキューの状況により処理時間は大きく異なってくる。もう一つの問題点はポインタである。RTOS では膨大な量のキューを実装している。例えばセマフォ識別子が

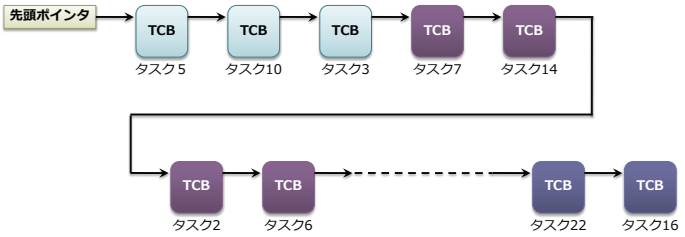

図3-5　Waitキューの実現方法 (2)

64 個、イベント識別子が 64 個、優先度が 16 個であったとするとキューの総数は、

64×2×16 = 2,048 個

となる。したがって高々上記のオブジェクトの数でさえポインタだけのために多くのリソースを必要とすることになる。

これを改善したのが図 3-5 である。各優先度の最後尾と次の優先度の先頭の TCB を接続している。このようにすると、Wait キューからのタスクの取り出し方法は極めて簡単であり、常にポインタが示すタスクを選択すれば良いことになる。またポインタの数も激減する。一方でよくよく考えてみると、この改善案はタスクをキューに接続するときの処理が大変であることがわかる。例えば優先度 7 のタスクをキューに接続したい場合、先頭からサーチを開始し、優先度 7 のタスクを探し出し、その最後尾に接続しなければならない。

いずれにせよ、キュー処理は多くの処理時間を要し、またそのときのキューの状況により処理時間が変動することが理解できる。また、キュー処理に共通して言えることは、大変クリティカルな処理であるため、その期間割込み禁止となる。以下、キュー処理にどのくらいの時間が費やされ、またリアルタイム性にどのくらいの影響を及ぼしているかを評価した。

評価する RTOS のキュー構造は、図 3-5 の形態であることがわかっている。このためキューにタスクを接続するとき、その時点でキューに既に何個のタスクが接続されているかにより処理時間が大きく変わってくることが予想できる。また、優先度毎のキューにそれぞれいくつのタスクが接続されているかということも

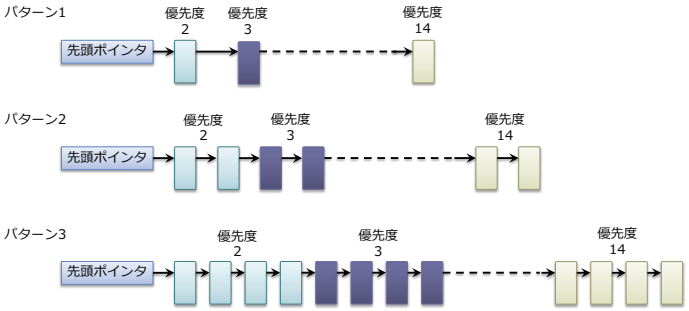

図3-6　キュー処理時間測定パターン

処理時間に影響することが予想できる。このため、図 3-6 で示した 3 つのパターンで測定を実施した。パターン 1 は既に接続されているタスクが、優先度 2 から 14 までそれぞれ一つ、パターン 2 はそれぞれ 2 つ、パターン 3 はそれぞれ 4 つとした。このとき、接続するタスクの優先度を変化させ、処理時間の違いを測定した。具体的には、あるセマフォ識別子を指定して、「セマフォ獲得」システムコールを繰り返し発行することにより、あらかじめ図 3-6 の各パターンを生成する。そして最後に同様に「セマフォ獲得」システムコールを発行し、この最後のシステムコール実行時間を測定した。

結果を図 3-7 に示す。この図で示すように、キューに既にタスクが多く接続されているパターンほど実行時間が大きく、また優先度が低いタスクをキューに接続する方が多く時間がかかっていることがわかる。

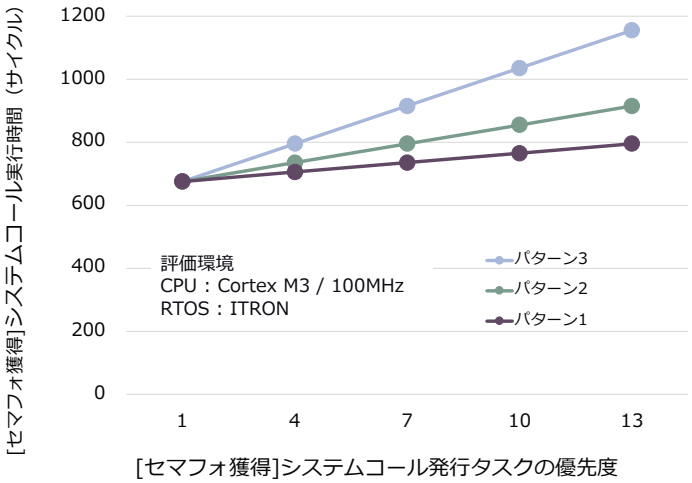

図3-7　キュー処理による実行時間の変動

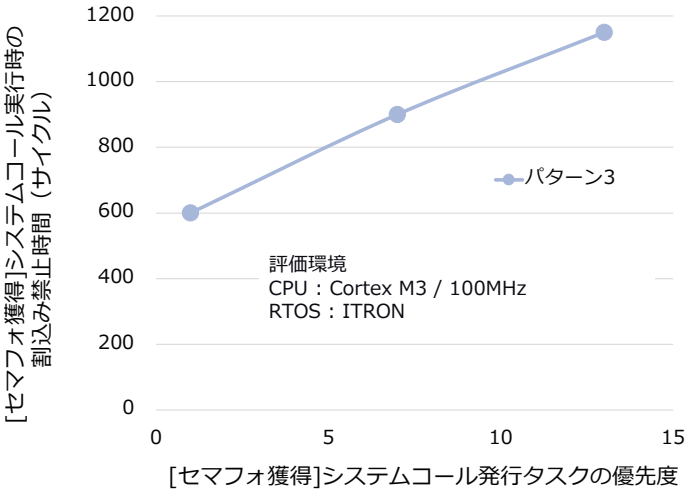

図3-8　キュー処理にともなう割込み禁止期間

図 3-8 はパターン 3 に関し、この処理の期間どのくらい割込み禁止になっているかを測定した結果である。この図からシステムコール処理をしているほとんどの期間割込み禁止になっていることがわかる。

以上から次のことが明確になった。キュー処理を伴うシステムコール処理時間はそのときの RTOS のキューの状態により大きく変動する。またこれにより生じる割込み禁止期間はシステムコール処理時間とほぼ同じであるため、割込み禁止期間も RTOS のキューの内部状態によりダイナミックに変化する。

このように、キューに既に多くのタスクが接続されているときのキュー処理において、思わぬオーバヘッドや思わぬ長期間の割込み禁止状態が発生し、リアルタイムシステムに思わぬ瑕疵を発生させる可能性がある。従来、アプリケーション・ソフトウェア設計者は、RTOS 内部のキューにどのくらいのタスクが接続されているのかといったことはほとんど考慮することはなかったと思われるが、こうしたことが発生しうることは事前に認識しておく必要がある。

3.3.　Tick 処理の影響

次に Tick 処理の影響について述べる。Tick 処理は RTOS の時間管理にかかわる処理を実行する。具体的には、

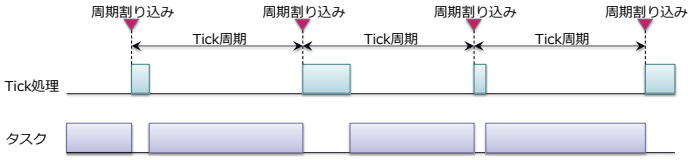

図3-9　Tick処理

- Wait 状態タスクのタイムアウトをチェックし、タイムアウトしていたらそのタスクを Wait 状態から Ready 状態に変更する。
- サイクリックハンドラ起動時刻になったかどうかをチェックし、起動時刻であったらサイクリックハンドラを起動する。

の 2 つである。図 3-9 で示すように、Tick 処理は周期割込みで起動され、上記処理を行う。この周期割込みの間隔を Tick 周期と呼び、この周期が Tick 処理の基本単位となる。したがって、タイムアウトやサイクリックハンドラの周期は Tick 周期の整数倍となる。

以上のように Tick 処理は RTOS にとって必須となる機能であるが、以下の様な問題点を有する。

1. 図 3-9 で示すように、周期的にアプリケーションが中断され、したがって CPU の使用効率が低下する。
2. 上記の様に Tick 期間はクリティカルな処理を行うため、割込み禁止となる。したがって、割込み応答性能が劣化する。
3. 上記の様に Tick 周期がタイムアウトやサイクリックハンドラの周期の単位となるため、Tick 周期を短くした方が時間管理の精度が上がる。しかし Tick 周期を短くすると、CPU 時間の Tick 処理占有率が上がり、アプリケーションのための CPU 効率が低下する。

このように、Tick 処理は RTOS 性能において、オーバーヘッドの原因にもなり、また割込み禁止期間のに対しても大きな影響をもたらす。以下 Tick 処理に伴うオーバヘッド時間と、Tick 処理により発生する割込み禁止期間を測定した。

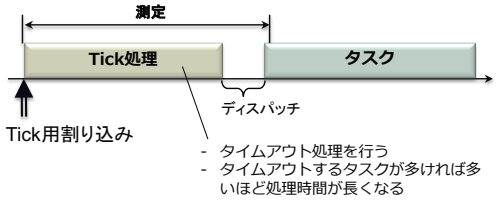

図3-10　Tick処理時間測定

上記の様に、Tick 処理では、Wait 状態のタスクがタイムアウトしたかどうかをチェックしている。したがって同時にタイムアウトするタスクが多いほど処理が増えると想定した。このためタイムアウトするタスクのタイミングを調整し、同じ Tick 処理で、n 個のタスクが同時にタイムアウトするようにソフトウェアを設定し、Tick 処理にかかわる時間を測定した。具体的には図 3-10 で示すように、Tick 起動割込み発生から、Tick 処理が終了し次のタスクがディスパッチされるまでの時間を測定した。このとき、n は 1、8、16 と変化させた。この結果を図 3-11 に示す。また、この Tick 処理による割込み禁止期間を測定した。その結果を図 3-12 に示す。

これらの図から言えることは以下の通りである。まず図 3-11 と図 3-12 がほぼ同じと言うことは Tick 処理中は割込み禁止であることを示していることがわかる。また、タイムアウト処理が一つ増える毎に約 567 サイクルのオーバヘッド（または割込み禁止期間）が増えていることがわかる。

この結果からリアルタイム処理を保証する上で、ソフトウェア技術者は極めて慎重に開発を進めなければならないことがわかる。実際に 16 個のタスクが同時にタイムアウトする確率は極めて小さい。しかし従来ソフトウェア技術者は同時にタイムアウトしたときの影響をほとんど考慮していないのではないだろうか。例えば何らかの異常状態が重なったとき、複数のタスクが同時にタイムアウトし、本来異常状態ではリアルタイム性を担保する必要があるのに、逆にリアルタイム性を保証することができなくなると言ったことが起きうる。

以上、従来のソフトウェア RTOS について、オーバヘッドと割込み禁止期間について述べた。従来の RTOS においては、例えば単純なシステムコールの実行時間などではそれほど大きなオーバヘッドは見受けられないものの、RTOS の内部状態によっては同じシステムコールでも大きく実行時間が変動し、またこれに伴い割込み禁止時間もダイナミックに変化する。また Tick 処理では処理内容により同様にオーバヘッドや割込み禁止期間がダイナミックに変化する。100MHz の CPU で、場合によっては数十マイクロ秒以上の割込み禁止状態が発生することもあり、リアルタイム処理においては十分な注意が必要である。

この章で測定した RTOS は、市販の RTOS の一つの例ではあり、それぞれの RTOS の実装方法により結

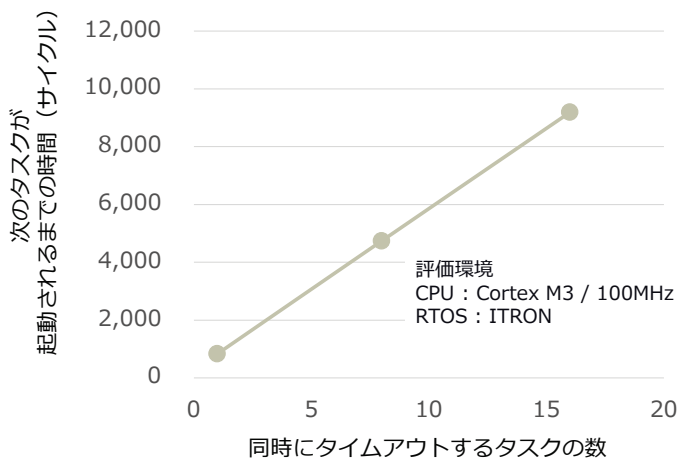

図3-11　Tickによるオーバヘッド

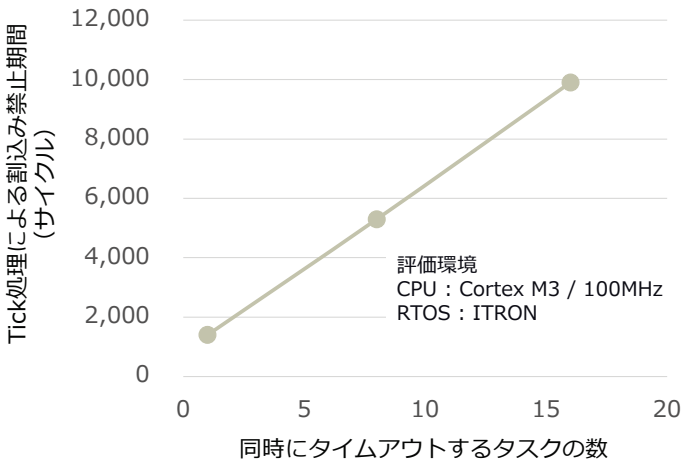

図3-12　Tick処理に伴う割込み禁止期間

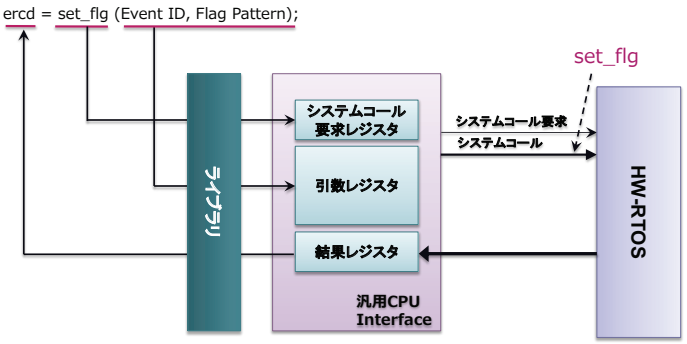

図4-1　HW-RTOSのシステムコール発行方法

果は異なる。しかし、他の RTOS を使用したとしても抜本的な解決策はあり得ず、傾向としては同様な結果が得られると考える。

4.　HW-RTOS の改善策と性能

4.1.　システムコール実行時間の改善

ここからは、HW-RTOS において、今まで示してきた問題をどのように改善しているのかを述べる。

図 4-1 に HW-RTOS のシステムコール発行方法を示す。この図のようにアプリケーションから発行されたシステムコールはライブラリソフトウェアを介して HW-RTOS にハードウェアの信号としてし通知され、また戻り値もライブラリソフトウェアを介して受け取る。またディスパッチ処理は HW-RTOS の指示に従ってライブラリソフトウェアが実行する。図 4-2 に HW-RTOS と従来のソフトウェアの RTOS（以下 SW RTOS と呼ぶ）の実行時間の測定結果を示す。

HW-RTOS のシステムコール実行処理時間は大変高速で、ほとんどのシステムコール処理時間は 10 サイクル以下で完了する。しかしライブラリソフトウェアのオーバヘッドにより、図 4-2 で示す実行時間となる。HW-RTOS の場合、SW RTOS で示した「セットフラグ」システムコールのように、そのときの RTOS の状況によって同じシステムコールでも実行時間が変化すると言うことはない。またシステムコールの最悪値をデータシートにより明示できるのが大きな特徴である。以下、この理由を説明する。

4.2.　キュー処理の改善

HW-RTOS では、ルネサスオリジナル技術である「仮想キュー」という回路によりキューを実現している[1] [3]。仮想キューは、タスクのキューへの接続、キューからのタスクの取り出し、またキューの途中にあるタスクの取り出しもすべて 2 サイクルで実現する。このためそのときのキューの状態に依存することなく一定の時間で処理を完了することができる。

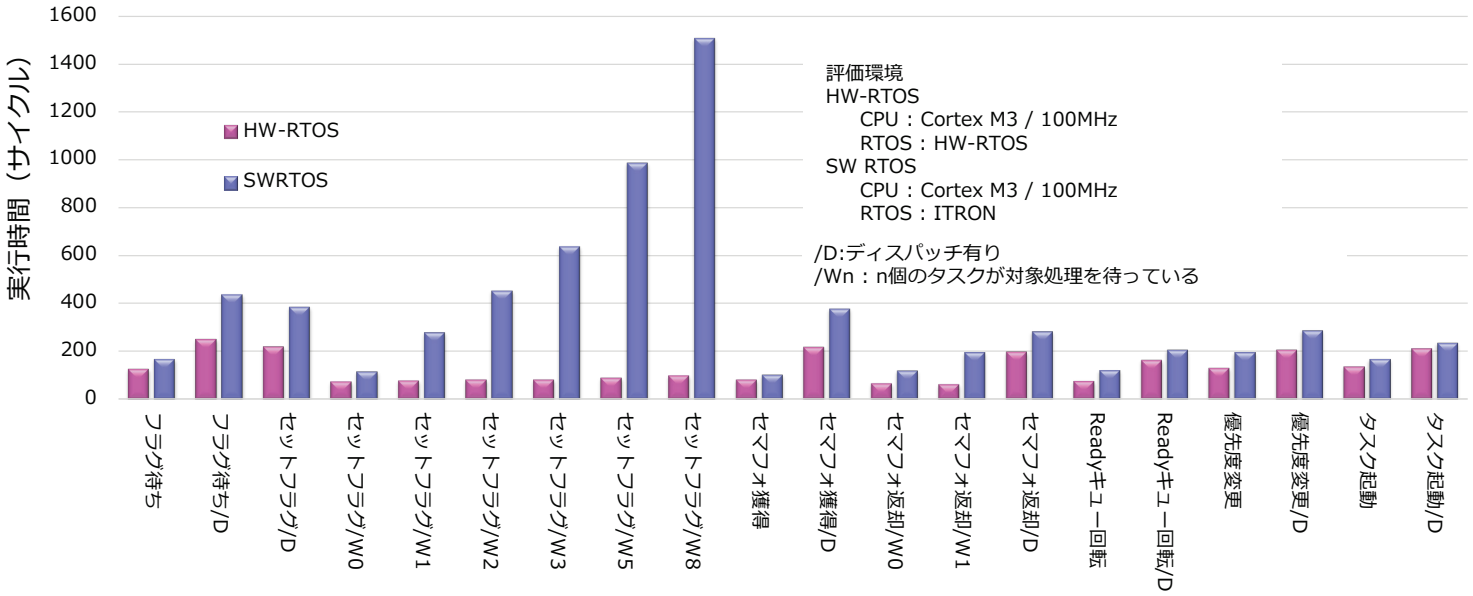

図4-2　システムコール実行時間

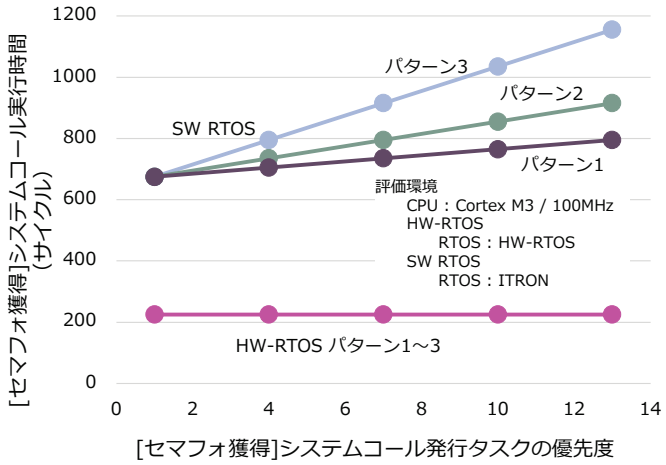

図4-3　キュー処理とオーバーヘッド

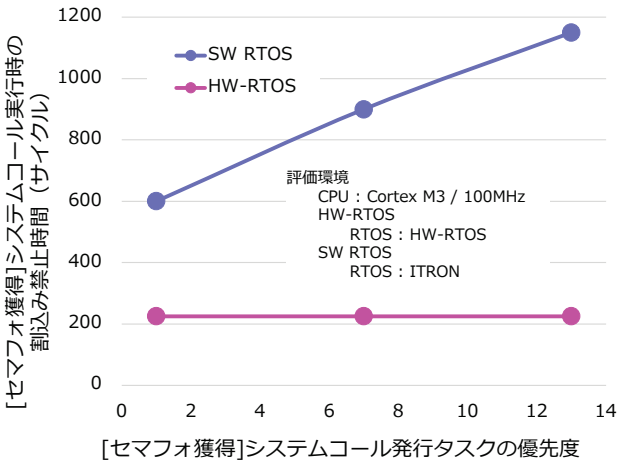

図4-4　キュー処理による割込み禁止期間

図 4-3 は、3.2 で測定した「セマフォ獲得」システムコールの実行時間を HW-RTOS でも測定したものである。この測定結果より、HW-RTOS はどのようなキューの状態においてもシステムコール実行時間が一定であることが確認できる。さらに図 4-4 ではパターン 3 の割込み禁止期間について測定しているが、こちらも HW-RTOS は一定である。以上から、仮想キューを採用している HW-RTOS では、キューのその時々の状態に依存することなく常に一定の処理時間を実現することができ、これにより、一定のオーバヘッド、一定の割込み禁止期間を実現していることがわかる。したがってアプリケーション・ソフトウェア設計者が予想しなかったようなオーバヘッドや割込み禁止期間が発生することは一切ない。また、HW-RTOS では上記システムコールの実行時間および割込み禁止期間に関し最悪値を明確に示すことができる。

4.3.　Tick 処理の改善

次に HW-RTOS における Tick 処理の改善について説明する。HW-RTOS では Tick 処理を完全にハードウェア化した[2] [5]。この機能を Tick オフローディングという。図 4-5 に Tick オフローディングを示す。この図で示されるように、Tick 機能は完全にハードウェア化され、HW-RTOS の中に実装されている。図 3-9 で示したように、従来は周期割込みにより Tick 処理を起動していたが、図 4-5 で示すように HW-RTOS では Tick 処理のための周期割込みは必要なく、また CPU では一切 Tick 処理を実行する必要がないためアプリケーションソフトウェアは Tick 中も処理を継続できる。アプリケーションが中断するのは（オーバヘッドが発生するのは）タイムアウトが発生しタスク切換が起きるときのみである。

また Tick 処理による割込み禁止期間はほとんどゼロであり、割込み禁止期間が生じるのはオーバヘッドと同様に、タイムアウトが発生しタスク切換が起きるときのみである。さらに Tick 処理が高速であるため、Tick 周期を短くすることができ、したがってタイムアウトやサイクリックハンドラの精度を大幅に向上させることができる。

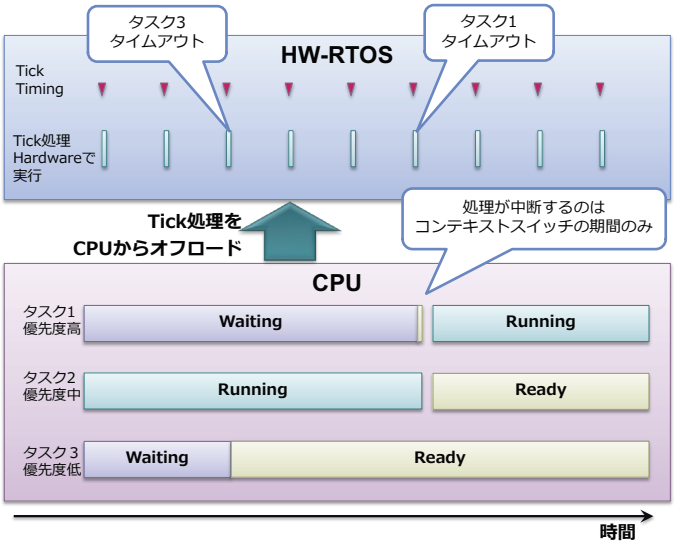

図4-5　Tickオフローディング

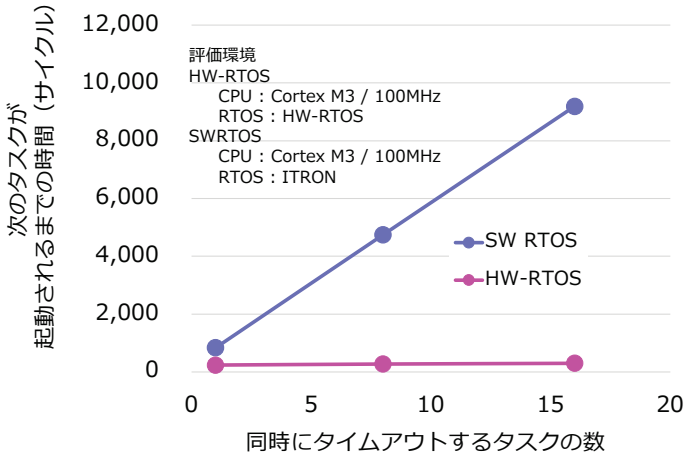

図4-6　Tick処理によるオーバヘッド

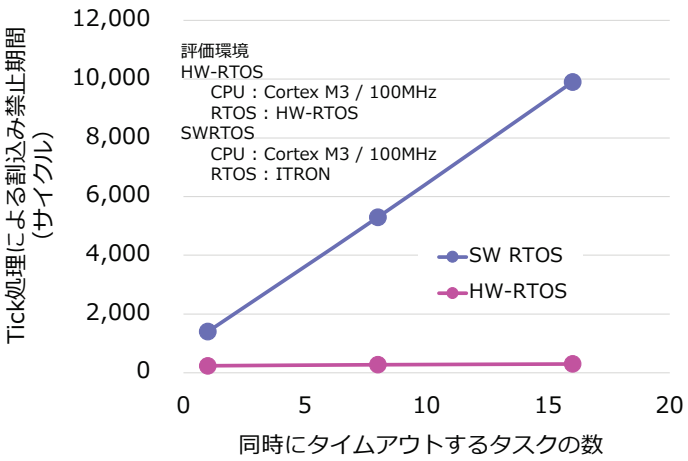

図4-7　Tick処理による割込み禁止期間

以上のように Tick オフローディング機能により、先に述べた従来の Tick 処理による 3 つの問題点が全て解決されることになる。

図 4-6 は、3.3 で示した Tick オーバヘッドの試験を HW-RTOS に適用した結果でる。すなわち、ある Tick で 1 つのタスク、8 タスク、16 タスクが同時にタイムアウトしたときのタイムアウト処理におけるオーバヘッドを測定したものである。HW-RTOS では全てオーバヘッドは 225 サイクルで、タイムアウトするタスクの数に依存しないことがこの図からわかる。

図 4-7 は同じ環境で割込み禁止期間を測定したものであるが、オーバヘッドと同様に割込み禁止期間も HW-RTOS ではタイムアウトするタスクの数に依存せず一定であることが確認できる。

以上のように SW RTOS では Tick 処理において Wait タスクのタイムアウト処理により大きなオーバヘッドが生じ、またこの期間割込み禁止になるが、HW-RTOS ではオーバヘッド、割込み禁止期間とも SW RTOS に比較し大幅に少なくまた一定である。

タイムアウトのタイミングが同時になることは希ではあるが、設計者は同時になるかどうか、同時になったときどのくらいのオーバヘッドや割込み禁止期間が発生するかを事前に知ることが難しい。このため複数のタイムアウトが同時に発生してもオーバヘッド、割込み禁止期間とも一定である HW-RTOS はリアルタイムシステムにおいて極めて有効な手段であるといえる。

5.　結論

従来のソフトウェア RTOS において、オーバヘッドの最悪値や割込み禁止期間の最悪値を事前に知ることは困難である。その理由は、キューの処理時間や Tick での処理時間が RTOS の内部状態により大幅に変動するからであり、その原因となる Wait キューでの待ちタスクの数や同時にタイムアウトするタスクの数などは常に変動しており、設計時に知ることは不可能に近いからである。こうしたことから従来のソフトウェア RTOS ではある状況が重なったとき、リアルタイム要求性能を満足せず、リアルタイムシステムにおいて致命的な結果をもたらすこともあり得る。

一方 HW-RTOS は、オーバヘッドの最悪値や割込み禁止期間の最悪値を事前に定義することが可能であり、アプリケーション・ソフトウェア設計者は設計段階でリアルタイム性を確定することができる。HW-RTOS を使用することによりソフトウェア設計者の開発負担を大幅に減らすことが可能であり、また信頼性の極めて高いリアルタイムシステムを容易に構築することが可能である。また割り込み禁止期間が極めて小さいことから、高精度なリアルタイムシステムを構築することが容易になる。

6.　参考文献

[1] N. Maruyama, T. Ishihara, H. Yasuura,“ An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing ”in Proc. of IEEE Symposium on Application Specific Processors (SASP), 2010, pp. 13-18.

[2] N. Maruyama, T. Ishikawa, S. Honda, H. Takada, K. Suzuki, ”ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems”, in Proc. of Operating Systems Platforms for Embedded Real-Time applications (OSPERT’14), 2014, pp. 9-16.

[3] 丸山修孝, 石原亨, 安浦寛人 :” RTOS のハードウェア化によるソフトウェアベー ス TCP/IP 処理の高速化と低消費電力化”, 電子情報通信学会論文誌 A Vol.J94-A No.9 pp.692-701 (2011).

[4] 丸山修孝,一場利幸, 本田晋也, 高田広章 :”マルチコア対応 RTOS のハードウェア 化による性能向上”, 電子情報通信学会論文誌 D Vol. J96-D No.10 pp.2150-2162 (2013)

[5] 丸山修孝, 石川拓也, 本田晋也, 高田広章, 鈴木克信 :”疎結合ハードウェア RTOS 搭載産業ネットワーク用 SoC”, 電子情報通信学会論文誌 D, Vol.J98-D No.4 pp.661-673 (2015).