リアルタイムOSのハードウェア化による
マルチコア組込みシステムの高速化

丸　山　修　孝

2015年1月9日

概要

本論文のメインテーマはRTOS（リアルタイムOS）のハードウェア化である．目的は組込みシステムにおけるアプリケーションの高速化である．特にネットワークプロトコル処理や機械制御処理の高速化を目指す．組込みシステムにおいて，低コストおよび低消費電力は必須条件である．従来技術による高速化手法は，主にコアの性能を向上させることにより実現しており，この手法では消費電力やコストの増加を招いてしまう．本研究では，RTOSをハードウェア化することにより，低コスト・低消費電力を実現しつつこれらのアプリケーションの高速化を実現する．ネットワーク処理においては全処理量に対するRTOSの処理の比率が高く，また機械制御においては割り込み応答の高速化が要求されているため，RTOSのハードウェア化がアプリケーションの高速化に大きく貢献する．

本研究ではまず，シングルコア対応のハードウェア化RTOSを開発した．処理をハードウェア化することにより高速なOS API（以下では単にAPIと呼ぶ）実行性能を実現した．またオリジナルコアを同時に開発した．オリジナルコアはハードウェア化RTOSと密結合で接続するための専用のインターフェースを有し，これにより極めて高速なAPI発行と高速なコンテキストスイッチを実現した．また組込みシステムに使用するRTOSは一般に数千個のキューを内蔵するが，従来技術によりこれをハードウェアで実現すると膨大なハードウェア量になってしまう．本研究では仮想キューという新しい手法を提案し，極めて少ない回路量で数千個のキューを作り出した．以上のような新しい概念をアーキテクチャに取り込むことにより，API実行時間において，

従来のソフトウェア RTOS の 30〜60 倍の性能を達成した．また本アーキテクチャに基づき ASIC を試作し TCP/IP プロトコルソフトウェアを実装した結果，従来のシステムに比較し同じ動作クロックレートで十倍以上の TCP/IP スループットを実現した．

次に前記開発したハードウェア化 RTOS に，高速化した割り込み機能を追加した．具体的には，tick 処理のハードウェア化および ISR 処理の定型化によるハードウェア化である．この結果割り込み応答性能において従来のソフトウェア RTOS システムと比較し，38〜131 倍の性能を実現した．また割り込み応答時間の最小値と最大値の変動幅を大幅に縮小した．これはリアルタイム処理において大きなアドバンテージである．

次に汎用コアである ARM コアからハードウェア化 RTOS を利用できるよう改良を実施した．上記研究はハードウェア化 RTOS とオリジナルコアを密結合で接続していたが，オリジナルコアではなく汎用コアでハードウェア化 RTOS を利用したいというニーズが多く，この要望に応えた．従来の密結合方式のハードウェア化 RTOS を改良し，汎用コアのバスを使用してコアとハードウェア化 RTOS を接続する疎結合アーキテクチャを提案した．さらにこのアーキテクチャに基づき，ルネサスエレクトロニクス（株）において，FA ネットワークコントロール用 ASSP を開発した．

最後にマルチコア対応ハードウェア化 RTOS を開発した．このシステムでは密結合接続を採用した．マルチコアシステムでは RTOS の管理情報を各コアからアクセスするため，そのロック単位としてジャイアントロック方式と細粒度ロック方式が存在する．検討した結果，最大アクセス時間は双方ともほぼ同じであるのに対し，ハードウェア量がジャイアントロックの方が圧倒的に少ないため，本研究ではジャイアントロック方式を採用した．また本研究においては，従来のマルチコア対応ソフトウェア RTOS において問題となっていたスタベーションやコア内・コア間排他制御の問題を防ぐ回路を実装し，その結果 API の最悪実行時間を定義することができた．性能評価では，コアを跨がった API 実行時間においてソフトウェア RTOS に比較し，30〜40 倍の性能を実現した．

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

序論

1.1 研究の背景

近年，様々な機器がコンピュータにより制御されている．ロボットなどの工場内の産業機器をはじめ，自動車，航空機，電車等の移動機器，ネットワーク機器，医療機器，また冷蔵庫，電子レンジ，掃除機などの白物家電等，様々な機器がコンピュータにより制御される．このように特定の機能を実現するため，機器に組み込まれるコンピュータシステムのことを組込みシステムと呼んでいる．

本研究の目的は，組込みシステムにおけるアプリケーションの高速化である．特にOS機能を必須とするが，リアルタイムOS (RTOS) を使用しても，RTOSのオーバヘッド（実行時間や割り込み応答時間）が，高速化の妨げになるアプリケーションである．このようなアプリケーションとして，ネットワークプロトコル処理や機械制御処理を挙げることができる．本論文では，従来技術による高速化の障害や問題点を明確化すると共に，問題を解決し高速化を実現する新たな手段としてRTOSのハードウェア化を提案する．ハードウェア化RTOSにより，頻繁にRTOSを呼び出すアプリケーションでは，RTOS実行時間の高速化により飛躍的な性能向上が期待できる．また従来RTOSのオーバーヘッドのため要求される割り込み応答性能を実現できず，

RTOS を利用することが難しかったアプリケーションにおいても RTOS を搭載できるようになるため，ソフトウェア開発の飛躍的な生産性の向上が期待できる．

以下，組込みシステムにおいて要求される条件とそれに対する従来技術の課題，本論文でこの課題に対しハードウェア RTOS を提案する理由について述べる．

まず，組込みシステムにおける低コストと低消費電力の必要性と課題について述べる．多くの組込みシステムで要求される共通の条件は低消費電力と低コストである．コンシューマ機器において低コスト・低消費電力が要求されるのは当然であるが，産業機器等においても同様である．例えば工場内の産業機器はモーター制御のためにコントローラが必要であり，工場内では膨大な数のコントローラを配備しなければならない．工場全体として低コスト・低消費電力を実現するためには，個々のコントローラの低コスト化と低消費電力化が必須である．また自動車においても乗用車では数十のプロセッサが搭載されており，同じ理由で低コスト・低消費電力が重要である．さらに自動車の場合は発熱に対する対応が必要であり，単に電力消費を少なくすると言う意味以外において低消費電力化が重要である．また最近ではコンシューマ機器，産業機器を問わずモバイル機器が増加しており，バッテリーを長持ちさせるためさらなる低消費電力化が求められている．従って，本研究の目的である組込みシステムの高速化においては，低コスト化と低消費電力化という条件も同時に満たす必要がある．

低コスト・低消費電力が求められる一方、アプリケーションサイドからはその高速化が常に求められている．従来技術によるアプリケーションの高速化は，コアの高速化により実現する手法が主流である．ネットワーク処理では，高速処理を実現する高性能ネットワーク処理専用プロセッサが存在するが，高速な動作クロックレートのため消費電力も大きく，また高価である．機械制御においては，高速なリアルタイム応答性実現するためのみの目的で，高い動作クロックレートのコアを使用することも少なくない．コアのクロックレートの高速化は，周辺のメモリやバスシステムの高速

化をも要求するため，コアのコスト・消費電力を増大させるだけでなく，システム全体のコスト・消費電力の増加を招く．したがって従来手法により低消費電力化・低コスト化という条件を満たしつつアプリケーションの高速化を実現することは困難である．

次に，RTOS の有用性と課題について述べる．組込みシステムはコンピュータシステムであり，したがってソフトウェアで動作をコントロールするシステムである．組込みシステム用の制御ソフトウェアであっても制御すべき内容は複雑であり，一般的には複数のソフトウェア・モジュールより構成される．ソフトウェア・モジュールの数が増えると，モジュール間の関連が複雑になりシステムの信頼性が低下する．RTOS はこの問題を解決する．RTOS 上に構築されるアプリケーション・ソフトウェア・モジュールをタスクと呼ぶ．RTOS は複数のタスクのスケジューリングを行い，またタスク間での同期機能や通信機能など共通で使用できる機能を提供する．RTOS はこうした機能を API (Application Programming Interface) を介してタスクに提供する．RTOS を利用することにより次のようなメリットを得ることができる．

(1)　定義された API 上にアプリケーション・ソフトウェアを開発するため，ソフトウェアの部品化，再利用化が容易であり，また複数の開発機関での開発が容易になるため，ソフトウェア開発生産性が向上する．

(2)　スケジューリング機能，同期機能や通信機能など複雑な機能が既に RTOS に実装されており，システムの信頼性が向上する．

このため組込みシステムでは RTOS を利用することが望ましい．企業や業界では，RTOS を含む共通プラットフォームを定義することにより，開発生産性や信頼性の向上を目指す動きも多く見られる．一方で RTOS は複雑な共通機能を提供するため，そのオーバヘッド時間が問題になる場合がある．特にネットワークプロトコル処理や機械制御などは RTOS のオーバヘッド時間により性能が劣化する．以下，これらのアプリケーションについて RTOS のオーバヘッドについて具体的に説明する．

ネットワークプロトコル処理は複数の処理が同時に発生するため，RTOS 上にマル

チタスク環境が必要である．例えば，送信と受信の要求は独立して発生するため処理も独立にすべきである．またパケット毎に処理すべきものとコネクション単位で処理すべきものは別タスクとすべきである．こうしたことからネットワークプロトコルはマルチタスクとして実現されることが多く，またこのため頻繁にRTOS機能を使用してタスク間の同期通信を行っている．したがって，ネットワーク処理におけるRTOS処理のCPU占有率は極めて高く，実験結果では純然たるプロトコル処理の3倍ものCPU時間をRTOS処理が占有している．

機械制御においては高度なリアルタイム処理を実現することが重要であり，RTOSのオーバヘッドがリアルタイム処理を実現する上で障害になる．機械制御におけるリアルタイム処理とは,「周期的かつ確実な実施」を要求する処理である．周期を短くすることにより高精度なモーター制御が可能になる．周期を短くするためには高速な割り込み応答が必須である．要求性能が厳しいアプリケーションの例では，割り込み発生から1μ秒以内に周期処理ソフトウェアが立ち上がらなければならない．RTOSを利用すると，そのオーバヘッドのため，このような厳しい割り込み応答性能を実現することは困難である．

以上から，RTOSの性能を向上させることにより，ネットワークプロトコル処理および機械制御とも性能向上をさせることができると考えられる．すなわちネットワーク処理においては，純然たるプロトコル処理の3倍ものCPU時間をRTOSが占有していると言うことは，RTOS処理性能を著しく向上させ，仮にCPU占有時間をゼロに近づけることができれば，ネットワーク処理性能は約4倍になる．また機械制御においてもRTOSの性能を向上させることにより，割り込み応答時間が大幅に改善され，モーターコントロールの精度を向上させることができる．このアプローチでは，従来技術のように単純にコアの高性能化によるアプリケーション高速化のアプローチと全く異なり，コアの動作クロックの高速化なしにアプリケーションの高速化が実現可能である．

本研究では，RTOS性能を著しく向上させため，RTOSのハードウェア化を提案す

る．RTOS のハードウェア化により，コアの性能を上げることなく，すなわち従来のシステムに比較し大幅に消費電力・コストを増加させることなく，アプリケーションの性能を大幅に向上させることができる．

アプリケーションの高速化を実現する方法として，従来技術としてはコアの高速化以外の方法としてマルチコアを挙げることができる．半導体技術の向上により 1 チップの中に容易に複数のコアを実装することが可能になっていることから，マルチコアは大きなコスト増につながらない．また動作クロックを上げずに性能を向上することができることから，大幅な消費電力増加に結びつかず，従って組込みシステム採用での最低条件をクリアすることができる．一方マルチコア対応の RTOS を使用した場合，RTOS 管理情報を共有するためにロックシステムを採用せざるを得ず，ロックによるオーバヘッドが性能劣化をもたらす．またスタベーション，コア内・コア間排他制御の問題と言ったマルチコア対応 RTOS 特有の問題があり，これらが原因で従来の RTOS においては最悪 RTOS 実行時間を定義できず，リアルタイム処理において大きな課題になっている．本研究では，ハードウェア化した RTOS をマルチコア対応に拡張し，さらに RTOS 特有の問題の発生を防ぐ機能を実装することにより上記性能劣化や最悪実行時間が定義できない問題を解決する．これにより，マルチコア組込みシステムにおいても大幅に消費電力・コストを増加させることなく，アプリケーションの高速化を実現し，かつリアルタイム処理系でのニーズに答える．

1.2 研究の概要

まず RTOS の種類と言葉の定義を行う．従来のソフトウェアにより実装された RTOS を SWROTS，ハードウェア化された RTOS を HWRTOS と言う．Fig.1.1 に示すよう，HWRTOS とコアの接続方法は 2 種類あり，一つは専用インターフェースで接続される密結合型（TC-HWRTOS : tightly coupled type of HWRTOS)，もう一つはシステムバスにより接続される疎結合型（LC-HWRTOS : loosely coupled type of HWRTOS) で

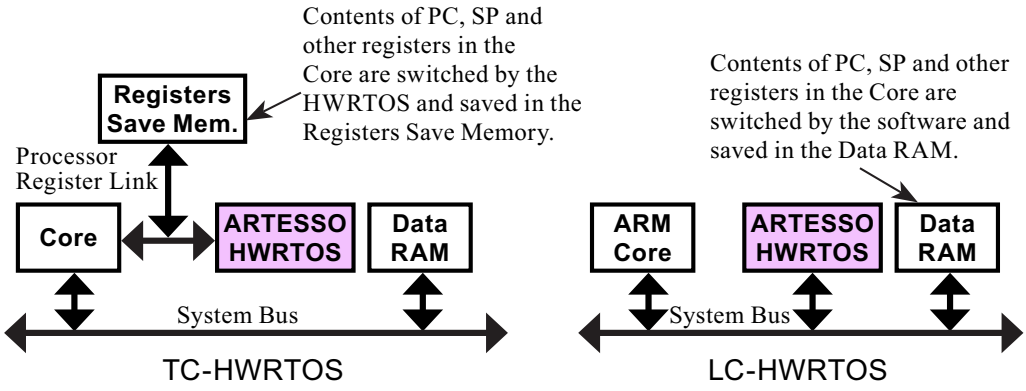


Figure 1.1: HWRTOS とコアの接続方法

ある．我々が提案，開発した HWRTOS とコアを含むプラットフォームを ARTESSO と呼ぶ．また我々が開発した HWRTOS を ARTESSO HWRTOS，TC-HWRTOS のために我々が開発したコアを ARTESSO Core と言う．

第 6 章では，従来のソフトウェアによるシングルコア用 RTOS を SoftSingle-RTOS，マルチコア対応 RTOS を SoftMulti-RTOS と呼ぶ．一方，ハードウェアで実現したシングルコア用 RTOS を HardSingle-RTOS，マルチコア対応 RTOS を HardMulti-RTOS と呼ぶ．我々が開発したシングルコア用 HWRTOS を Single-ARTESSO，マルチコア対応 HWRTOS を HardMulti-ARTESSO と呼ぶ．Single-ARTESSO は第 3 章，第 4 章の機能を含む ARTESSO と定義し，第 5 章の機能は含まないものと定義する．

次に研究の概要について説明する．

（1） シングルコア対応 RTOS のハードウェア化による TCP/IP 処理の高速化と低消費電力化

本研究では，シングルコア対応 RTOS をハードウェア化し，TCP/IP スループットの向上を実現した．本システムのためにオリジナルのコア ”ARTESSO Core” を開発し，ARTESSO HWRTOS と ARTESSO Core 間に専用インターフェースを定義し，TC-HWRTOS を実現した．ARTESSO HWRTOS は 40 種類の ITRON 仕様 [1] の API を実

装し，既存の ITRON ベースで開発されたソフトウェアの容易なポーティングを実現する．

RTOS は多数のキューを必要とする．例えばセマフォ，イベント，メールボックスの識別子が各 32 個，優先度が 16 個の場合，1,536 個のキューが必要である．従来技術でキューをハードウェアで実現しようとした場合，ハードウェア FIFO を使用するのが一般的であるが，上記多量のキューを LSI に実装することはコスト増につながる．本研究では仮想キューという新しいキューイングシステムを提案した．仮想キューはキュー情報の可逆性のある圧縮技術であり，数千個のキューを少ない回路量で実現し，また極めて短時間でのキュー操作を実現する．

ARTESSO HWRTOS の性能を測定したところ，従来の SWRTOS の API 実行時間が約 300 サイクル程度から 600 サイクル程度であるのに対し，ARTESSO HWRTOS の性能は 10 サイクル時間前後であった．従って従来比約 30〜60 倍の性能を達成した．また本研究では 90nm プロセスを使用した ASIC を試作開発した．この ASIC 上に TCP/IP ファームウェアを実装し TCP/IP スループットを測定した．この結果 50MHz の動作クロックにおいて，従来の SWRTOS を使用した汎用 CPU では 11Mbps であったのに対し，ARTESSO では 125Mbps であった．また 100Mbps のスループット時の消費電力は従来技術の 1/7 であり，ARTESSO が低消費電力であることを実証した．

本研究は，ARTESSO HWRTOS によって低い動作クロックレートで高い TCP/IP スループットを達成，つまり低コスト・低消費電力で高いネットワーク性能を実現した．

（2） 超高速応答を実現するハードウェア割り込み処理機構

本研究では（1）で開発した HWRTOS に 2 つの機能拡張をし，割り込み応答性能を向上させた．一つは tick オフローディング機構，もう一つは IIA (Interrupt Invoked API) オフローディング機構である．

まず tick オフローディングについて説明する．割り込み応答性能を劣化させる要

因として割り込み禁止状態がある．長い割り込み禁止状態を作り出す原因は tick 処理である．tick 処理は，RTOS で使用しているウエイトタイマの更新，タイムアウトの検出，タイムアウト時の処理等を行う．従って tick 処理は RTOS の重要な管理情報を更新する処理であり，他の処理に比較し優先度を高くしかつ割り込み禁止で実行される．このため割り込み応答時間に大きな影響を与える．本研究では tick 処理を完全にハードウェア化し，ソフトウェアによる tick 処理を不要とする tick オフローディング機構を提案した．これによりコアの動作クロックを高速にすることなく，割り込み応答性能を大幅に向上させ，また割り込み応答時間の変動もほとんどなくなった．

次に IIA オフローディングについて説明する．割り込み発生時における従来の SWRTOS の処理の流れは以下の通りである．まず現在実行中のタスクを中断し，ISR (Interrupt Service Routine) が起動され，ISR が割り込み処理を実行し，完了すると再びタスク処理に戻る.「割り込み発生から再びタスク処理に戻るまでにかかる時間」は，SWRTOS では数百〜数千サイクル時間を要していた．本研究では ISR 機能を定型化し，ハードウェア化することによりこの時間を大幅に短縮する手法を提案した．これを IIA オフローディングと呼ぶ．IIA オフローディングにより，コアの動作クロックを高速にすることなく,「割り込み発生から再びタスク処理に戻るまでの時間」は数百〜数千サイクルから 14〜17 サイクル時間に短縮した．また SWRTOS に比較しこの時間の変動幅を大幅に圧縮した.

本研究により，RTOS を使用した環境においても，高速な動作クロックを使用することなく，すなわちコスト・消費電力の増加を伴うことなく，割り込み応答性能の大幅な向上を実現し，組込みシステムにおける機械制御処理の高速化実現を可能にした.

（3） シングルコア対応疎結合ハードウェア RTOS 搭載 産業ネットワーク用 SOC

本研究では（2）で開発した ARTESSO HWRTOS を改造，機能追加を行い，汎用プロセッサである ARM コアで ARTESSO HWRTOS を利用できるようした．またこ

のプラットフォームを ASSP に適用し，ルネサスエレクトロニクス（株）が FA 用ネットワーク処理用 ASSP を開発，現在量産中である．

（1）,（2）はコアとしてオリジナルコアである ARTESSO Core を使用する．しかし組込みシステム用の汎用 LSI においては，ユーザは ARM 社のコアの実装を求めている．これは既に開発したソフトウェア資産を利用しやすいこと，今まで使用していた開発環境を利用できること，コアとしてのスケーラビリティがありまた信頼性が高いことが理由である．ARTESSO Core と ARTESSO HWRTOS は専用インターフェースで接続され TC-HWRTOS を構成するが，ARM コアは専用インターフェースがない．このため本研究では，ARM バスを利用して ARM コアと ARTESSO HWRTOS を接続する LC-HWRTOS を提案した。さらに（2）で提案した tick オフローディングおよび IIA オフローディング機構がコアと並列実行できるように改造を行い，コアのアベイラビリティを向上させた．

開発・量産した ASSP (R-IN32M3) は ARM コアとして Cortex-M3 100MHz を採用，産業用 Ethernet を実現するため Ethernet を 2 ポート実装した．また機械制御用に CAN を 2 ポート，I2C を 2 ポート，CC-Link を 1 ポート実装した．

（4） マルチコア対応 RTOS のハードウェア化による性能向上

本研究ではマルチコア対応の TC-HWRTOS を提案した．まずマルチコア RTOS として必要であるロックシステムに関し，RTOS が共有する管理情報全体をロックするジャイアントロック方式と，資源毎にロックする細粒度ロック方式を比較検討し，ジャイアントロック方式を採用した．そしてジャイアントロック方式に基づくアーキテクチャ設計を行った．アーキテクチャ設計にあたっては（2）をベースとし，またマルチコア RTOS 特有の課題であるスタベーション，排他制御の課題を防止する回路を実装した．

アーキテクチャ設計に基づき，FPGA を使用して ARTESSO Core を 8 個実装した試作システムを開発し，機能・性能評価を行った．この結果，比較対象の SWRTOS

のコアを跨いだ API 実行時間が 850〜900 サイクル時間であったのに対し，開発したマルチコア対応 TC-HWRTOS は 25〜27 サイクルであった．

本研究により，マルチコア組込みシステムにおいても，HWRTOS 上での動作環境を実現した．すなわち低動作クロックレート，つまり大幅に消費電力・コストを増加させることなく，マルチコア組込みシステムでネットワーク処理，機械制御処理の高速化の実現を可能にした．

1.3 論文の構成

本論文の構成は以下の通りである．まず，第 1 章では本研究の背景と概要について述べた．第 2 章では，組込みシステムにおける RTOS の必要性を説明した後，組込みシステムで要求されている高速化について説明し，高速化を実現するための技術的課題を明確にする．第 3 章では，高速化要求を実現する上での一手段である RTOS のハードウェア化について提案を行い，ASIC を開発，その評価結果を示す．第 4 章では，第 3 章で開発した HWRTOS の割り込み応答性能をさらに向上する方式を提案し，評価する．第 5 章では，第 4 章までに開発した HWRTOS を汎用プロセッサである ARM コアから使用できるよう実施した機能追加について述べ，開発した ASSP について説明し，性能評価を示す．第 6 章では，マルチコア対応の HWRTOS を提案し，実施した試作開発について説明し，性能評価を示す．最後に第 7 章で本研究の結論と今後の課題について述べる．

CHAPTER 2

組込みシステムの高速化技術

第 2 章では組込みシステムの基本的な概念，RTOS の基本的な機能，RTOS の典型的な性能について説明し，また組込みシステムにおいて RTOS を使用するメリットについて述べる．さらにネットワーク処理や機械制御アプリケーションにおける高速化ニーズとそれを実現するための課題および RTOS を使用したときの問題点について説明する．また，組込みシステムの高速化ニーズを満足させる技術として，RTOS のハードウェア化とマルチコア化を挙げ，これらを実現するための課題を明確にする．

2.1　組込みシステム

2.1.1　組込みシステムとは

組込みシステムとは，特定の機能を実現するため機器に組み込まれるコンピュータシステムのことを言い，パソコン，サーバ等汎用的なシステムと区別されている．現在までに様々な組込みシステムが開発され，商品化されている．ロボットなどの工場内の産業機器，自動車，航空機，電車等の輸送機器の制御装置，医療機器をはじめ冷蔵庫，電子レンジ，掃除機など白物家電等も組込みシステムが内蔵され，ソフトウェアによりコントロールされている．自動車などでは数十のプロセッサが搭載され，

これらが連携して目的を達成している．工場内においてもライン上に数多くの工作用モーターが配置されており，組込みシステムによりこれらモーターがコントロールされ，全体として目的を達成している．またこのように近年複数のプロセッサがネットワークに接続され，協調して動作することにより目的を達成するシステムが増加してきており，このためネットワーク機能を必要とする組込みシステムが多くなってきている．

組込みシステムの共通した要求として低コスト，低消費電力を挙げることができる．コンシューマ機器では低コスト・低消費電力が重要であるが，コンシューマ機器でなくても組込みシステムでは低コスト・低消費電力は必要な条件である．上記自動車や工場のように多量に組込みシステムを使用する分野では，全体として低コスト・低消費電力が求められており，これを実現するためには個々の組込み装置の低コスト・低消費電力化を実現する必要がある．さらにコンシューマ機器，産業機器を問わずモバイルシステムが増加しており，バッテリー動作のため低消費電力化が強く望まれる．また自動車等のアプリケーションでは発熱を極力嫌う傾向にあり，発熱を抑えるという意味において低消費電力化が求められている．以上のように，低コスト，低消費電力化は，組込みシステムの共通した必要条件である．

2.1.2 リアルタイム組込みシステム

組込みシステムのうちリアルタイム性能を必要とするシステムをリアルタイム組込みシステムと言う．リアルタイムとはアプリケーションが時間的制限を持つことを意味する．これはパソコンなどによるビジネス系アプリケーションでは平常時のパフォーマンスを求められていることと大きく異なる点である．機械制御系のアプリケーションの多くはリアルタイム性を必要とする．たとえば自動車のパワーステアリングシステムは，センサが運転者のステアリング操作を検知し，モーターを駆動することにより運転者の負担を軽くしている．この制御はリアルタイムシステム，すなわち時間的制約を持ったシステムである．なぜなら運転手がステアリングを廻しきって

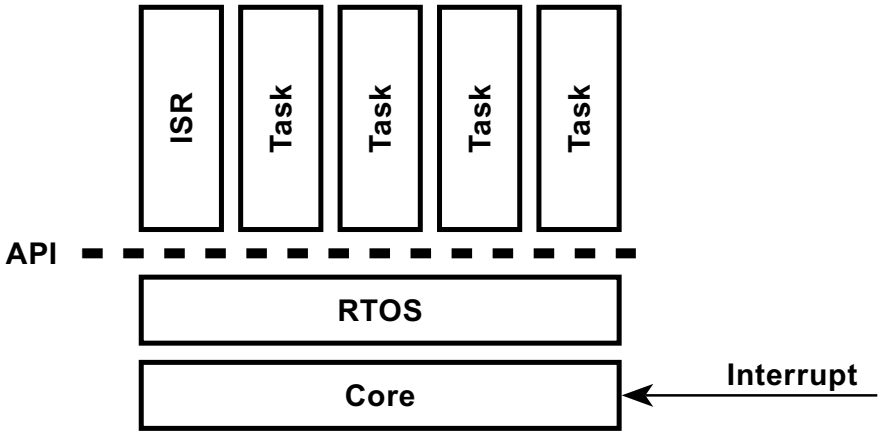


Figure 2.1: RTOS を使用した典型的な組込みシステムの構成

からモーターが動き出したのでは遅く，廻し出したのを即座に検知しモーターを駆動しなければならないからである．実際のパワーステアリング制御では 100μ 秒毎に検出を行い，その検出値に基づき 100μ 秒毎にモーターをコントロールしている．この間隔が変動すると正しくモーターがコントロールされない．したがってステアリングコントロールシステムは時間的制約を持つシステムである．このようにリアルタイム組込みシステムとは，アプリケーションの時間的制約に対応することのできる組込みシステムを言う．

2.2 リアルタイム OS

2.2.1 マルチタスク構成の必要性と RTOS の果たす役割

一般の組込みシステムは複数のソフトウェア・モジュールから構成され，これらソフトウェア・モジュールが関連し合って動作する．ソフトウェア・モジュールの数が増えてくるとソフトウェア・モジュール間の関連が複雑になる．RTOS を使用することによりこの問題を解決することができる．以下これを説明する．

Fig.2.1 に典型的な RTOS を使用した組込みシステムの構成図を示す．RTOS を使

Table 2.1: 典型的な API

カテゴリ	API
タスク同期機能	- イベントフラグ生成，イベントフラグ削除，イベントフラグクリア，イベントフラグ待ち，イベントフラグセット - セマフォ生成，セマフォ削除，セマフォ資源獲得，セマフォ資源返却 - メールボックス生成，メールボックス削除，メールボックス送信，メールボックス受信
タスク管理機能	自タスク終了削除，タスク強制終了，タスク起動，タスク優先度変更
システム状態管理	- CPU ロック，CPU ロック解除，ディスパッチ禁止，ディスパッチ禁止解除 - 実行状態タスク参照，タスク優先順位回転
時間管理	システム時刻参照，システム時刻設定
タスク付属同期機能	起床待ち，タスク起床，待ち状態強制解除

用したシステムでは，アプリケーションのためのソフトウェア・モジュールはタスクと呼ばれ，割り込みが発生したときに起動されるソフトウェア・モジュールは ISR (Interrupt Service Routine) と呼ばれる．RTOS は実行すべきタスクまたは ISR を選択し実行させる．これをスケジューリングといい，RTOS の重要な機能の一つである．図で示すように，RTOS とタスクまたは ISR の間には API (Application Programming Interface) が定義されている．API は，タスクまたは ISR が RTOS の提供する機能を利用するためのインターフェースである．典型的な API の例を Table2.1 に示す．

RTOS を使用する利点は次の通りである．第一にアプリケーション・ソフトウェアをタスクとして定義するため，ソフトウェアのモジュール化を促進することができる．第二に共通した API 上でソフトウェアを設計することができるため，同じ RTOS を使った他製品への展開が容易になり，ソフトウェアの再利用が容易になる．第三に API が厳密に定義されているため複数の技術者，複数の開発機関での分散開発が容易になる．第四に，タスク間同期通信機能，スケジューリング機能など共通で使用する複雑かつ重要な機能が RTOS に実装されているため，アプリケーションソフトではこれらの複雑な機能を開発する必要がなくなり，ソフトウェア障害を減少させることができ，ひいてはシステムの信頼性が向上する．このように組込みシステムにおいては

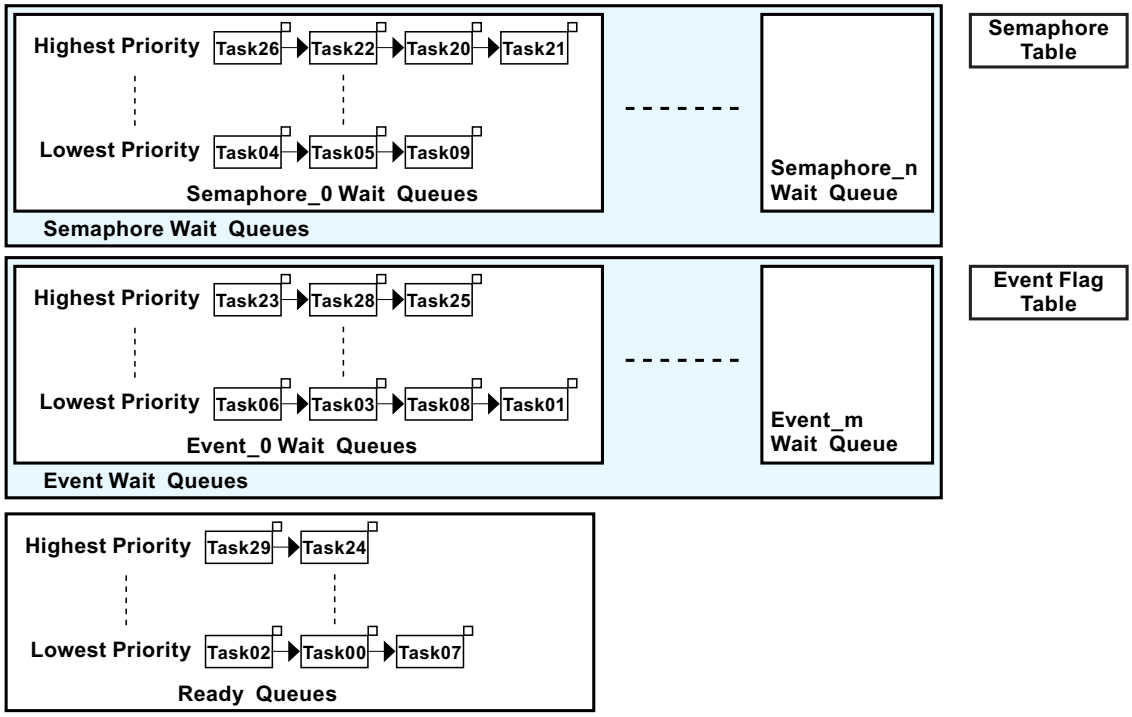


Figure 2.2: RTOS の構造

RTOS を使用することによって得られる利益は大きい.

2.2.2 RTOS の構造

次に RTOS の構造について説明する．Fig.2.2 に RTOS の構造を示す．従来の一般的な SWRTOS では，タスクの状態として少なくとも「IDLE」状態,「WAIT」状態,「READY」状態,「RUN」状態の 4 つが定義されており，各タスクは常にいずれかの状態に属している．また全てのタスクには優先度が与えられている．RUN 状態は現在実行中のタスク，READY 状態は実行することが可能になっているタスクである．一つのコアでは一時期に一つのソフトウェアしか実行できない．このため RTOS は READY 状態のタスクから一つ選んで RUN 状態にし，このタスクを実行させる．これをスケジューリングという．多くの RTOS が採用しているスケジューリングアルゴリズムは, 優先度付き First-come, First-served (FCFS) 方式である．FCFS とは先に要求されたもの

を先に処理すると言うことであり，このアルゴリズムを実現するためにキューを使用する．キューは優先度の数だけ用意される．優先度付き FCFS アルゴリズムとは一番優先度の高いキューの先頭のタスクを選択すると言うことである．READY 状態から RUN 状態に遷移する時に使用するのが Ready キューであり，Fig.2.2 の Ready Queues に示すよう，優先度付き FCFS を実現するため，タスクの優先度の数だけキューが存在する．

スケジューリングにより RUN 状態のタスクや ISR を切り換えるためには，コアのレジスタ（プログラムカウンタ，スタックポインタ，フラグレジスタ，汎用レジスタ）を入れ替える必要がある．各タスクや ISR 毎のレジスタの内容をコンテキストと言うことから，RUN 状態のタスクや ISR の切り換えに伴うレジスタの入れ替えをコンテキストスイッチという．すなわちコンテキストスイッチの実施は RUN 状態のタスクまたは ISR の切り換えを意味する．

セマフォやイベントフラグ機能を実現するためにもキューを使用する（Fig.2.2）．イベントおよびセマフォは，識別子毎にキューを持っている．また各識別子ごとに，タスクで定義された優先度の数だけキューが存在する．RUN 状態のタスクが「セマフォ資源獲得」API を発行し，セマフォ資源を他のタスクが使用中であった場合，発行したタスクは指定されたセマフォ識別子かつこのタスクの優先度の Wait キューに接続される．このときこのタスクは RUN 状態から WAIT 状態に遷移したことになる．

一方，RUN 状態のタスクが「セマフォ資源返却」API を発行し，このセマフォ識別子のセマフォ獲得を待っているタスクがある場合（すなわちこのセマフォ識別子の Wait キューにタスクがキューイングされている場合），優先度付き FCFS のルールに従って一つのタスクを Wait キューから取り出し，同じ優先度の Ready キューに接続する．すなわちこのタスクは WAIT 状態から READY 状態に遷移したことになる．このとき現在 RUN 状態のタスク，すなわち「セマフォ資源返却」API を発行したタスクの優先度がセマフォを待っていたタスクの優先度以上であれば，RUN 状態のタスクは変更しない．逆に API を発行したタスクの優先度がセマフォを待っていたタスク

の優先度より低ければ，API を発行したタスクを READY 状態にし，セマフォを待っていたタスクを RUN 状態にする．すなわち RUN 状態のタスクが切り換わるため，コンテキストスイッチが実施される．

RUN 状態のタスクが「待ち状態強制解除」API を発行したとき，RTOS は引数で指定されたタスクを WAIT 状態から READY 状態に遷移させる．このとき FCFS に従わずキューの途中から該当タスクを取り出すことになる．その後上記と同じように優先度比較を行い必要に応じてコンテキストスイッチが実行される．

「セマフォ資源獲得」や「イベントフラグ待ち」API ではタイムアウトを設定することができる．発行したタスクが WAIT 状態になりセマフォ獲得やイベントフラグセットの前にタイムアウトになると，このタスクは WAIT 状態から READY 状態に遷移する．このときも FCFS に従わずキューの途中から該当タスクを取り出す．その後現在 RUN 状態のタスクの優先度と比較を行い，タイムアウトしたタスクの方が優先度が高い場合，コンテキストスイッチを行う．

以上のように RTOS は FCFS スケジューリングシステムや豊富な API 機能を提供することにより，複数のソフトウェア・モジュールを使用したシステム，すなわちマルチタスクシステムにおいてソフトウェア開発を単純化し，ソフトウェアのモジュール化・再利用化を促進し，ソフトウェア開発生産性を向上させ，さらにシステム信頼性をも向上させる．また RTOS を実現するためには多くのキューの実装が必要であり，さらにキューは単に FCFS を実現するだけではなく，あるときは途中から取り出すなど複雑な構造を伴う．

2.2.3 SWRTOS の性能

上記のように RTOS は様々な重要な共通機能を提供する．従来技術の RTOS はソフトウェアで構成されており（すなわち SWRTOS），アプリケーション・ソフトウェアと同じコア上で動作するため，SWRTOS が動作している期間はアプリケーションから見るとオーバヘッド時間である．Fig. 2.3 は SWRTOS によるオーバヘッド時間を示

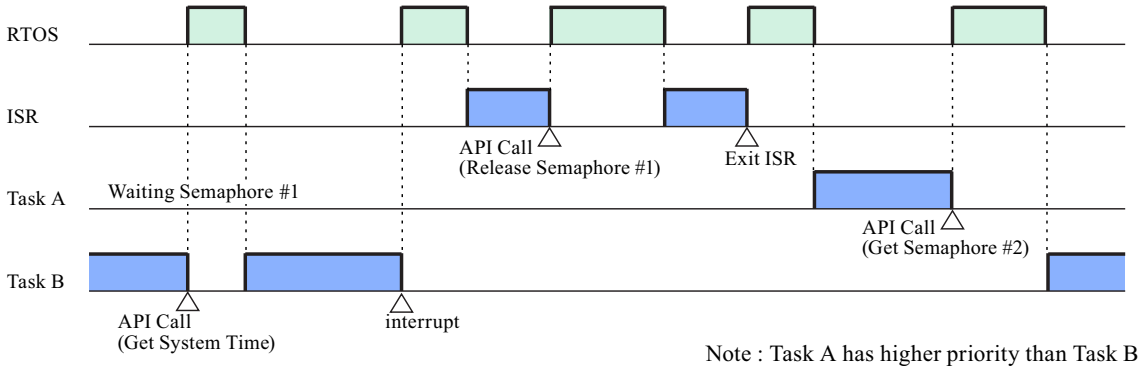


Figure 2.3: SWRTOS オーバヘッド時間

Table 2.2: 市販 SWRTOS の API 実行時間・割り込み応答時間

処理内容	コンテキストスイッチ	実行時間
API 処理		
起床待ち	有	628
タスク起床	有	496
タスク優先度変更	有	541
セマフォ資源獲得	無	216
セマフォ資源獲得	有	558
セマフォ資源返却	無	344
セマフォ資源返却	有	536
割り込み応答	有	92 〜 428

単位 : Cycles

している．頻繁に API 発行をしたり，頻繁にインタラプトが発生するアプリケーションでは，頻繁に処理が SWRTOS に移るため，SWRTOS のオーバヘッド時間が全体の性能に与える影響が大きい．したがって RTOS 内での処理時間は短ければ短いほど良い．RTOS の性能を示す指標として，アプリケーションが API 実行要求を RTOS に発行してから戻ってくるまでの時間（API 実行時間）を挙げることができる．Table2.2 に市販 SWRTOS の API 実行時間の測定値を示す．単位はコアの動作クロックのサイクル数である．使用したコアは 32 ビットオリジナル RISC コア（ARTESSO Core），使用した RTOS は商用の ITRON 仕様 SWRTOS である．もう一つの性能指標は割り

込み応答時間である．これは割り込みが発生してからISRが実行されるまでの時間であり，特にリアルタイムアプリケーションでは重要な指標である．同じ環境で測定を行った結果，応答時間は，最小92サイクル，最大428サイクルであった．これはSWRTOSの内部状況により，その時々で応答時間に変動があることを示している．

次節では，アプリケーションの高速化について言及し，その中でRTOS性能がアプリケーションの高速化に与える影響について説明する．

2.3 アプリケーションの高速化ニーズ

組込みシステムにおいて，特に高速化ニーズが高いアプリケーションが，ネットワークプロトコル処理と機械制御である．本節ではこれら二つのアプリケーションについてなぜ高速化ニーズが高いのか，また高速化を実現する上での問題点は何なのかを明確にする．

2.3.1 ネットワークプロトコル処理

2.3.1.1 ネットワーク高速化ニーズ

近年様々な機器がネットワークに接続されるようになってきており，有線ではEthernet，無線では802.11仕様の無線LANやLTE等高速公衆無線網が利用されている．Ethernetは当初10Mbpsであったが，現在では最大100Gbps仕様まで標準化されており，さらに400Gbpsの仕様の標準化が進められている．802.11無線LANは当初1Mbps仕様であったが，現在は最大6.9Gbpsの仕様の標準化を完了している．こうした状況の中，実際に組込みシステムで使用されている速度はコストや消費電力の制約からEthernetでは100M〜1Gbps，802.11では56Mbps〜433Mbps程度である．

このようにPHYレイヤ，MACレイヤにおいては高速データ転送できる環境が実現されており，その上位レイヤにあたるTCP/IPにおいても高いスループットの実現

が期待される．しかし TCP/IP プロトコルの実行には多量の CPU 負荷を必要とする．パソコンやサーバはプロセッサの性能が極めて高いため，Gbps レベルの TCP/IP スループットを容易に達成することができる．しかし組込みシステムで使用されるプロセッサは，一般的にパソコンクラスのプロセッサに比較し性能が桁違いに低い．組込みシステムではコスト面，消費電力面の制限から高速なプロセッサを使用することが困難である．組込みシステムでよく使用される ARM9 上に市販の TCP/IP を実装しスループットを測定したところ，100MHz の動作クロックで 22Mbps のスループットであった．したがって単純に計算すると，100Mbps の TCP/IP スループットを実現するためには 450MHz，1Gbps を実現するためには 4.5GHz もの動作クロックが必要となる．組込みシステムでこのような高性能プロセサを使用することはコスト的にも消費電力的にも現実的でない．

2.3.1.2　従来のネットワーク処理高速化手法

TCP/IP の処理性能を上げる単純な方法は，コアの性能を上げることである．コアの性能を上げる一つの方法は動作クロックを上げる方法であり，もう一つの方法はマルチコアによる性能向上である．コアの動作クロックを上げると当然消費電力も増加する．またコアの動作クロックを上げるだけではトータル性能の向上は頭打ちになるため，キャッシュメモリ，周辺メモリ，バス等の性能もコア性能に比例して上げる必要があり，システムとしての消費電力の増加はコアの消費電力増加の数倍以上となる．さらに動作クロックを上げると部品コスト自体も増加する．

一方マルチコア手法は，コアの動作クロックを上げることなく性能向上を実現できる可能性がある．しかしマルチコアにおいて最大性能を実現するためには，ソフトウェアの各コアへのアサインメントという課題があり，またさらに 2.4.2 で示すように，RTOS 性能劣化を招くマルチコア特有の問題が存在する．しかしこれらの課題が克服できればマルチコアは，低コスト・低消費電力を維持しつつ高速化を実現する有力な手段である．

高スループットの TCP/IP を実現するためには Cavium 社等がネットワーク処理専用のプロセッサを出荷しており，動作クロック 500MHz〜1GHz，2〜16 コア構成により数 Gbps の TCP/IP を実現している．しかしこのようなネットワーク処理専用プロセッサは高価であり，また消費電力も極めて大きく組込みシステムにおいて汎用的に使用されているわけではない．

TCP/IP の処理性能を上げるもう一つの方法として TOE（TCP/IP Offload Engine）を使用する方法がある．TOE を使用することにより速度的な問題も消費電力的な問題も解決することができるが，プロトコル処理全体をハードウェア化するため，拡張性，保守性に問題がある．たとえばバグがあった場合 LSI を作り替えざるを得ない，TCP/IP のパラメタの拡張がソフトウェアでできない，TCP/IP の規格の変化に対応できない，と言った問題があり積極的には使用されていない．

2.3.1.3　ネットワーク処理高速化阻害要因の明確化

次に，TCP/IP 処理が高い CPU 負荷を必要とする原因を説明する．TCP/IP が実行されているとき，どのような処理が実行されているのか，TCP/IP プロトコルスタックを使用し各処理のコアの占有率を測定した．Fig.2.4（上：市販 TCP/IP）にこの結果を示す．測定条件は以下の通りである．

- プロトコルスタックは市販の TCP/IP スタックを使用．
- RTOS は市販の ITRON 準拠 SWRTOS を使用．
- コア上では TCP 処理，IP 処理および Ethernet ドライバ処理のみを実行．
- 送信側 TCP は 1,460 バイトフレームを送信，送信 2 フレームに対し 1 フレームの ACK フレームを返信．
- コアとして 32bit RISC プロセッサを使用．

同図において注目すべきは，プロトコル処理に割かれている時間が高々10%程度であると言うことである．その他の処理としては RTOS 処理（ROTS Processing），ヘッダ並び替え（Header Rearrangement），TCP チェックサム（TCP checksum），メモリコピー（Mem. copy）であり，これらの処理に約 9 割の CPU 時間を費やしている．

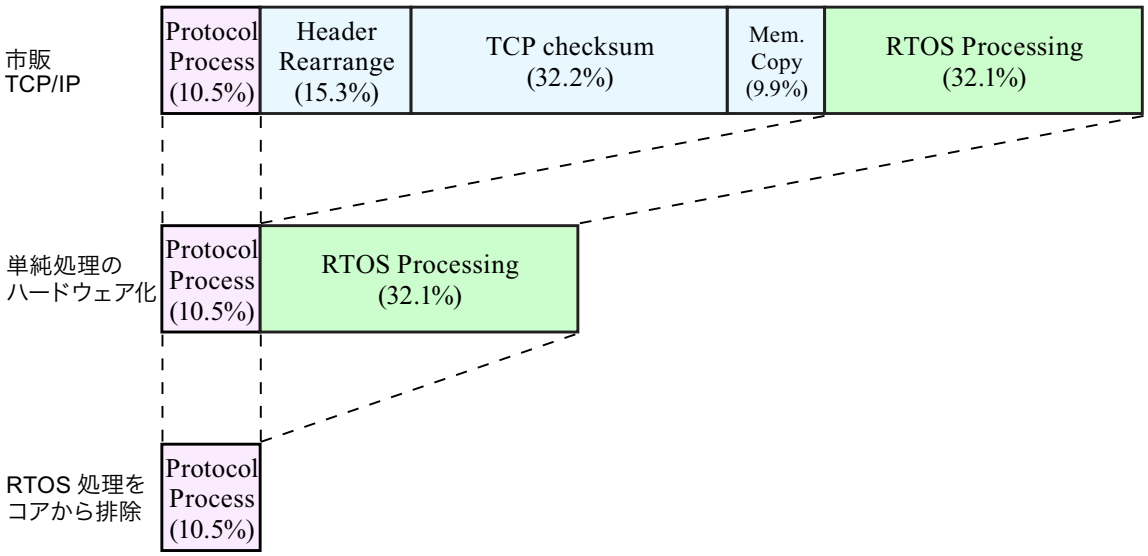


Figure 2.4: TCP/IP における各処理の CPU 占有率

メモリコピーとはメモリ内の単純なデータの移動である．ヘッダ並べ替えとは,「プロトコルで規定されたヘッダ部のイメージ」と,「ソフトウェアが処理しやすい形のイメージ」との変換処理である．TCP チェックサムは TCP プロトコルで規定されているチェックサム処理である．これらの処理は極めて単純であるが多くの CPU パワーを消費している．

もう一つコアに大きな負荷をかけているのが RTOS 処理である．ネットワーク処理は並列性の高い処理が随所にあるため，複数のタスクに分解し RTOS 上で処理を行った方がコアを効率的に使用でき，性能も向上する．一例を示すと,「送信 IP/Ethernet 処理タスク」,「受信 IP/Ethernet 処理タスク」,「送信 TCP 処理タスク」,「受信 TCP 処理タスク」といったタスク割り当てが考えられる．TCP はコネクション単位の処理であり，IP と Ethernet はパケット単位の処理であるためタスクは分割した方が良い．また送信と受信は別事象で発生し，同時に行われることも考えられるため別タスクの方が良い．このように並列化できる処理をタスクに分割することにより性能が向上する．一方でこのようにタスクを分割することによりタスク間通信やリソースの奪い合

いを制御するためのセマフォ機能を利用することになりRTOSの処理が増加する．またプロトコル処理はDMAの完了割り込みやEthernetの送受信完了割り込み，共有バスからの割り込み等が頻繁に発生する．割り込みが発生する度にRTOSが呼び出されるため，これもRTOS処理の割合が増加する要因となる．この問題を解決するためにTCP/IPをシングルタスクで実装するアプローチも存在するが，複数のプロセスが同時に動くメカニズムをシングルタスクの中に組み込む必要があり，限定的な効果しか期待できない．

以上より，次の結論が導かれる．ヘッダ並び替え，TCPチェックサム，メモリコピーは単純処理であり，ハードウェア化しコアからこれらの処理を切り離すことは比較的容易である．実際，TCPチェックサム回路を実装したEthernetコントローラも市販されている．コアからこれらの処理を切り離すことにより約57%のオーバヘッドを取り去ることができる．しかしこれだけでは依然RTOSの処理時間がプロトコル処理の3倍もの時間を必要としている（Fig.2.4中：単純処理のハードウェア化）．したがってRTOS処理時間を高速化することによりネットワーク処理性能は劇的に改善されることがわかる．つまりFig.2.4下図（RTOS処理をコアから排除）のように，仮にRTOS処理を完全にコアから取り去ることができれば，トータルで約90%のオーバヘッドを取り去ることができたことになり，これは同じ動作クロックのコアを使用しても10倍のスループットを実現できると言うことである．言い換えると，コスト・消費電力を増加することなく10倍の性能を得ることが可能である．しかし，今日に至るまで多くのRTOSベンダがSWRTOS処理性能の向上のための工夫をし尽くしており，SWRTOS実行時間の高速化はほぼ限界に達している．したがって現在のソフトウェア技術を使用する限りにおいては，RTOS処理のさらなる高速化は困難である．

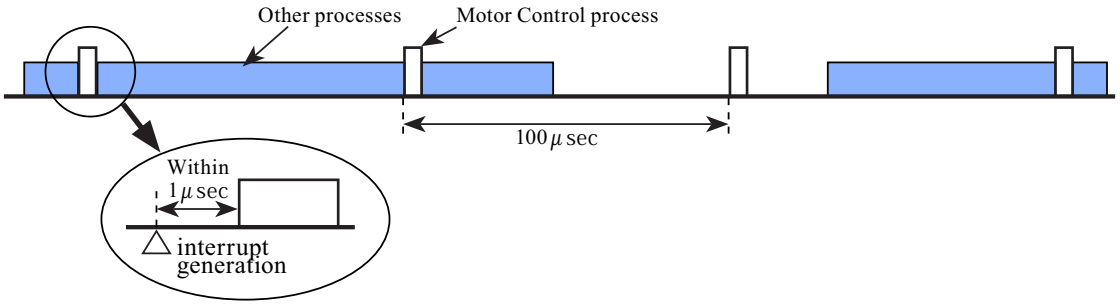


Figure 2.5: モーター制御の一例

2.3.2 機械制御

2.3.2.1 機械処理高速化ニーズ

工場内の機器をはじめ自動車，家電など様々なアプリケーションにおいてモーターが使用されており，これらのモーターはソフトウェアで制御されている．特に精密機器においては高精度なモーター制御を必要とする．Fig.2.5 は自動車のパワーステアリングのモーター制御を示している．モーターを制御するソフトウェアが 100μ 秒毎に立ち上がる．機械制御ではこのようなリアルタイム処理，すなわち周期的かつ確実な実行（歯抜けになってはならない）を求められる．このようなリアルタイム処理においては，割り込み応答が速ければ速いほど良い．割り込み応答性能が高速になれば周期を短くできる．周期を半分にできればモーター制御の精度は 2 倍になる．

Fig.2.5 のモーター制御ソフトウェアは割り込み発生から 1μ 秒以内に起動されなければならない．このような厳しい割り込み応答性能を必要とするアプリケーションでは SWRTOS を使用することは難しい．2.2.3 で示した実験結果によると，その環境での割り込み応答時間は最大 428 サイクルであった．したがって割り込み応答時間を 1μ 秒にするためには 428MHz の動作クロックのコアを使用する必要がある．割り込み応答性能は環境により変化するためマージンを考慮しなければならず，実際には 428MHz よりさらに速いクロックで動作させる必要がある．これは消費電力的にもコスト的にも問題である．さらにマージンをどの程度取れば良いかという問題もあ

り，現実的にはこのような環境では RTOS は使用せず，いわゆる「割り込みベース」のソフトウェアでシステムを構築している．従って 2.2.1 で示した，ソフトウェアのモジュール化・再利用化の促進，開発効率の向上，信頼性の向上と言った RTOS のメリットを享受することができない．しかし近年ではソフトウェア開発の生産性の向上や信頼性の向上のため，RTOS を利用したいというニーズが増えてきている，このようなニーズに応え，機械制御において RTOS を使用する場合，その割り込み応答性能を高速にし，また変動を小さく抑える必要がある．

次に工場内のネットワークについて説明する．工場内には様々な制御機器が存在する．たとえばラインに産業用機器が並んで設置されており，ライン上の製品を順次加工し組み立ててゆく．したがって，これら産業用機器は機器同士同期を取っている．さらに産業用機器の中には数個〜数十個のモーターが存在し，これらも同期を取って動作している．これら工場内の機器間，機器内の情報伝達に使用されるのが FA ネットワークである．ネットワークを介して同期を取るために，モーター制御と同様リアルタイム性を持った情報転送が必要である．リアルタイム性を持った情報転送とは，周期的かつ確実なデータ転送である（以下，このような転送をリアルタイム転送と呼ぶ.）．FA ネットワークには，従来 CAN (Controller Area Network) などの規格が使用されていたが，転送速度が低速であり（最高で 1Mbps 程度），近年 Ethernet タイプのネットワークに切り替わりつつある．Ethernet 仕様はリアルタイム転送を保証していないため，Ethernet の上位層に TCP/IP に代わるプロトコルを定義し，このプロトコルでリアルタイム転送を実現している．TCP/IP に代わる新たなプロトコル規格は複数定義されており，既に導入されている．たとえば，PROFINET (PROcess FIeld NETwork), EtherNet/IP (Ethernet industrial protocol)，ModbusTCP，EtherCAT (Ethernet for control automation technology) である [2]．リアルタイム転送は主にソフトウェアにより実現される．したがってリアルタイム転送を実現するには割り込み応答性能の高速化が重要である．割り込み応答性能が高速であればあるほど周期を短くすることができる．周期が 1/2 になれば工場の生産性は 2 倍になる．

2.3.2.2 従来の機械処理高速化手法

2.3.1.2 と同じく，従来技術による機械制御を高速化する方法はコアの高性能化である．すなわちコアの動作クロックを上げるか，マルチコアによる性能向上である．したがって課題も 2.3.1.2 と全く同じ課題となる．また FA ネットワークでは，2.3.1.2 で示したハードウェアオフロードエンジンの使用は現実的でない．上記のように FA ネットワークでは様々なプロトコルが規定されており，これら全てをハードウェア化すると言うことは大きなコストを伴うからである．

2.3.2.3 機械制御高速化阻害要因の明確化

前記のように，機械制御や FA ネットワークにおいてはリアルタイム処理，すなわち周期的かつ確実な実行処理が必須である．しかし現状の SWRTOS を使用した場合，割り込み応答時間が遅いだけでなく，割り込み応答時間が RTOS の内部状況に依存して大きく変動し，最悪値を求めることが非常に難しい．周期的かつ確実な実行処理を実現するためには，割り込み応答時間を短くするだけでなく応答時間の変動の縮小と最悪値を規定できることが重要である．一方この問題を従来の高速化手法，すなわちコアの高性能化により解決しようとすると消費電力やコストが問題になる．したがって，消費電力やコストを増加することなく，RTOS の割り込み応答時間の短縮を実現することが望ましい．

2.4 組込みシステムの高速化技術と課題

前節ではネットワーク処理と機械制御の高速化要求とその課題について説明した．特に組込みシステムにおいてはコストおよび消費電力を増加させることなく高速化要求を満たすことが重要であり，ポイントとなるのは RTOS 性能の向上，すなわち API の実行時間の高速化および割り込み応答性能の高速化であることを説明した．しかし現状の技術では SWRTOS の性能改善は既にし尽くされており大幅な性能向上を期待

することはできない．本研究においては，RTOS 性能を劇的に向上させるため RTOS をハードウェア化することを提案する．

一方，組込みシステムにおいて高速化要求を実現するために有効なもう一つの方法はマルチコアである．先にも述べたように組込みシステムにおいて重要なのはコストと消費電力の増加を抑えつつ高速化を目指さなければならないということである．シングルコアシステムにおいて単純に性能向上を実現するためには動作クロックを上げることになるが，これはコストと消費電力の制約から組込みシステムでの採用には限界がある．マルチコアシステムでは周辺回路の動作クロックを上げる必要がないためシステム全体としてのコスト増加，消費電力増加が少ない．また半導体技術の向上に伴い一つの半導体に複数のコアを実装することは容易であり，この点においてもコスト・消費電力増加要因を排除できる．さらに，上記 RTOS のハードウェア化による性能向上は極めて有効な手段であるものの，もし RTOS によるオーバヘッドをコアから完全に取り除くことができた場合，それ以上の性能向上はない．したがってそれ以上の性能向上は，マルチコアに頼るのが最善の方法である．

以上から組込みシステムの高速化技術として，RTOS のハードウェア化とマルチコア化の二つをあげることができ，双方を組み合わせることによりさらに大きな成果を期待できる．しかし双方とも組込みシステムで使用する上で様々な課題が存在する．以下では，RTOS のハードウェア化における課題と組込みシステムにおけるマルチコアシステムの課題を明確にする．

2.4.1 既存のハードウェア RTOS とその課題

RTOS を高速化するにはコアの性能を向上させることが一つの方法であるが，様々な問題があることを既に説明した．過去において，RTOS の性能を向上させるために様々な研究がなされている．いくつかの研究では RTOS の機能をハードウェアとソフトウェアに分割し性能向上を図っている [3]-[8]．またいくつかの研究では大部分の RTOS 機能をハードウェア化している [9]-[17]．以下 RTOS をハードウェア化する上

での課題について説明する．

RTOS をハードウェア化する上で大きな障害になるのが，キューの数である．実用的な RTOS においては各タスクが少なくとも STOP，RUN，READY，WAIT の 4 つの状態を有する．この中で WAIT 状態にあるタスクは何かのトリガを待っている．たとえばタイムアウト，セマフォの解放，イベント通知，メールボックスからのメッセージの受信などである．同じトリガを待っているタスクが複数存在する場合，優先度付き FCFS アルゴリズムにしたがってタスクが選択される．2.2 で示したように優先度付き FCFS アルゴリズムを実現するためにはキューが必要である．もし RTOS をハードウェア化しようとした場合，キューを実現するためにはハードウェア FIFO を使用するのが一般的である．2.2 で示したように Wait キューは，セマフォ，イベント，メールボックス等のオブジェクト識別子毎にキューが存在し，さらにオブジェクト毎優先度の数だけキューを用意しなければならない．たとえば識別子としてセマフォ 32 個，イベント 32 個，メールボックス 32 個の実装が必要であった場合で，優先度の数は 16 であった場合，必要となるキューの数は $(32 + 32 + 32) \times 16 = 1{,}536$ である．これだけ多量のキューをハードウェア FIFO で実現すると巨大な回路になる．

キューの基本機能は「入った順序で取り出すことができる機能」である．しかし RTOS で使用するキューはこの基本機能だけでは不十分である．2.2.2 で示したように RTOS ではキューの途中から指定したタスクを取り出す機能が必要である．具体例としてはタイムアウト付きの API（「セマフォ資源獲得」や「イベント待ち」など）を挙げることができる．WAIT キューに入っているタスクがタイムアウトすると，このタスクはキューの先頭でなくてもこのキューから取り出されなければならない．もう一つの例としては，WAIT キューから直接タスクを取り出す API を挙げることができる．例えば「待ち状態の強制解除」API は Wait 中のタスクを指定しキューから取り出すための API である．

上述のように RTOS では WAIT キューの途中から指定したタスクを探し出し，取り出す機能が必要である．「検索・取り出し」機能は RTOS 内部では頻繁に利用され，

かつ重要な機能である．ハードウェア FIFO は単に最後尾からデータを挿入し，先頭からデータを取り出す機能のみであり,「検索・取り出し」機能は実装されていない．またこの「検索・取り出し」機能をハードウェア FIFO に実装しようとすると回路量が増えるばかりでなく，実行動作時間が増加する．

従来の SWRTOS はリスト構造のキューを使っている．キューへの書き込みはリストの最後部にエレメントを付け加えることにより行われる．同様にキューからの読み出しは先頭のエレメントを取り外すことにより達成される．一番時間を消費するのは,「検索・取り出し」機能である．リストに沿って目的のエレメントを先頭からサーチしてゆき，エレメントを取り外しリストを構成し直す．このためリスト構造によるキューは動作に時間を消費するだけでなく，キューの長さにより処理時間が大きく変化し，リアルタイム性の高い処理に影響を与える．

以上のように RTOS では膨大な量のキューを使用し，また先頭から取り出すのみではなく「検索・取り出し」機能が必要である．このようなキューシステムをコストパフォーマンスを維持してハードウェア化することは従来の技術では困難である．過去における RTOS のハードウェア化の研究事例は RTU[9]-[11]，Silicon TRON [14]-[15], fido100[17]，HartOS[16] 等である．これらの事例におけるキューの数は限定的である．

Silicon TRON はイベント識別子，セマフォ識別子が共に 3 個実装されているのみであり，実装されている API の数は 21 種類である．fido100 は，タスクの優先度による単純なスケジューリングを実行するスケジューラである．タスクの優先度のみにより RUN 状態が決定されるため Ready キューは存在しない．また fido100 は，イベント，セマフォ，メールボックス等の機能自体存在せず，一般の RTOS が有するこうした機能を利用するための API は存在しない．したがって Wait キューも存在しない．HartOS は最大で 512 個のセマフォを実装できるが，リスト構造を使ったキューであるため実行時間が遅く，またセマフォ数，コンテキスト数，現在の内部状態により実行時間は大きく変動する．またイベントフラグやメールボックス等の API は実装されていない．

2.4.2 マルチコアの組込みシステムの適用と課題

既に説明したように，マルチコアシステムは組込みシステムの高速化要求に応える一つの有効な手法である．マルチコアシステムにおいてもオペレーシングシステムを実装することが望ましく，特にリアルタイム組込みシステムではマルチコアに対応したRTOSを必要とする．2.2においてRTOSの概要について説明したが．これはシングルコア用のRTOSの説明である．本節ではマルチコア対応RTOSの概要を説明すると共に，リアルタイム性の実現という観点からの課題を明確にする．

2.4.2.1 マルチコアとオペレーシングシステム

マルチコアシステムに実装するOSは，次の3つのタイプに分類できる．

- 疎結合型マルチコアシステム
- 対称型マルチコアシステム
- 機能分散型マルチコアシステム

疎結合型マルチコアシステムは，各コア毎完全に独立して動作するシステムである．したがって各コア毎にシングルプロセッサ対応OSが独立に実装され，OSは「マルチコア対応」である必要はない．したがって、本節ではマルチコア対応OSの特性，課題について検討するため，対象外とする．

対称型マルチコアシステムは，各タスクがどのコアで実行されるかをスケジューリング時点においてOSが決定する．パソコンやサーバなどはこのタイプである．対称型マルチコアシステムは，コヒーレントキャッシュが必要になり高価なシステムとなる．また動的な負荷分散を行うことからリアルタイム性を実現することが困難である．

機能分散型マルチコアシステムは，各タスクをどのコアで実行するか設計時にエンジニアが決定する．機能分散型マルチコアシステムでは，全てのタスクは初期化時においていずれかのコアに割り当てられ，したがってタスクのスケジューリングは各

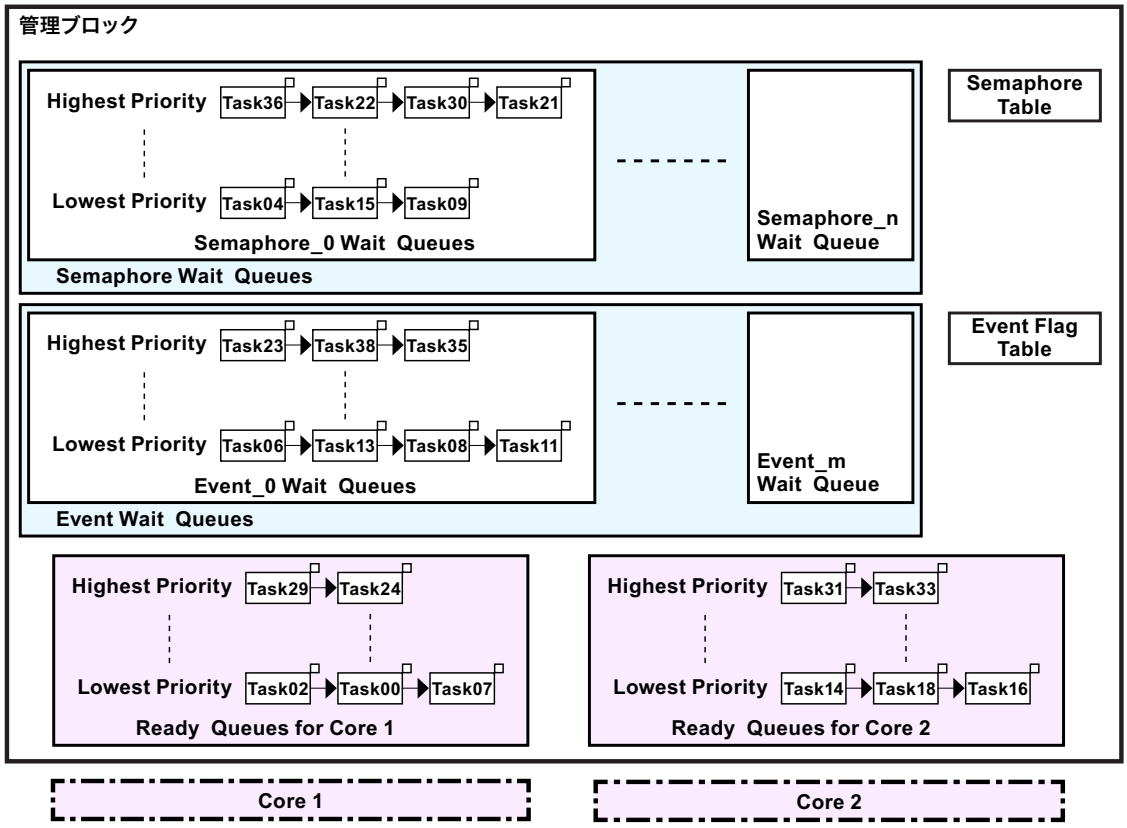


Figure 2.6: マルチコアシステムにおける RTOS の構造

コア毎に実行される．一方資源管理やタスク間同期通信等はコアを跨いだ実行が可能である．分散機能型マルチコアシステムでは，プロセッサ間でコード共有を行わないため，コヒーレントキャッシュの必要が無く安価なシステムの実現が可能であり，タスクをコアに固定できることからリアルタイムシステムを構築しやすい．したがってリアルタイム組込みシステムに適している．

以上から，本研究では機能分散型マルチコア対応の RTOS を検討するものとし，以下マルチコア対応 RTOS とは全て，機能分散型マルチコア対応の RTOS を示すものとする．

2.4.2.2　マルチコア対応 RTOS の構造

Fig.2.6 に典型的なマルチコア対応 RTOS の機能構造を示す．タスクはそれぞれのコアに割り当てられ，コアごと独立にスケジューリングされる．したがって Ready キューはコアごとに存在する．API は他のコアのタスクに対して発行可能である．すなわち,「タスク強制終了」API,「タスク起動」API,「タスク優先度変更」API 等を使用して他のコアで定義されているタスクに対し，強制終了，起動，優先度の変更等を実行することが可能である．

一方セマフォ，イベント，メールボックス等オブジェクトはコアには割り付けられておらず，したがって Wait キューもコアに割り付けられていない．オブジェクトはどのコアからも操作可能である．すなわち，セマフォ，イベント，メールボックスはコア間で共有でき，したがってこれらの API を使用してコア間同期，コア間通信を実現できる．

割り込みが発生したとき起動する ISR は初期化時に特定のコアに割り当てる．Ready キュー，Wait キュー，セマフォテーブル，イベントテーブルなど RTOS のオブジェクトを管理するデータ構造を管理ブロックという．

2.4.2.3　マルチコアにおける高速化阻害要因の明確化

（1）　ロック

あるコアに属するタスクがセマフォ待ちをしており，他のコアがこのセマフォを解放する API を発行するようなことがある．このようなコアを跨いだ API 処理を実現するためには，コアが他のコアの管理ブロックを参照，変更しなければならない．管理ブロックを参照，変更する方法として，API を発行するコアが管理ブロックを直接操作する直接操作法を採った場合，競合によるデータの矛盾が生じないようロックにより排他制御をおこなう．ロックはスピンロックで実現するのが一般的である．

排他制御を行うリソースの単位をロック単位という．ロック単位が管理ブロック

全体であるシステムをジャイアントロックという．一方管理ブロック内の資源ごとにロックを行うシステムを細粒度ロックという．ロックの競合の発生頻度を考慮すると細粒度ロックが望ましいが，あまり細かい粒度でロック単位を設定すると，API 処理の際に複数のロックを取得する必要があるためオーバヘッドが増大する．

ロック処理のためシングルコア用 SWRTOS に比較しマルチコア対応 SWRTOS では API 実行時間が増加する．例えばセマフォ資源返却（コンテキストスイッチあり）を実行したとき，シングルコア用 SWRTOS では 300 サイクルであったが，マルチコア対応 SWRTOS では 650 サイクルであった [18].

（2） ロック取得におけるスタベーション

複数のコアから API が発行され管理ブロックへのアクセスが頻発して発生したとき，あるコアの API が長期間処理されない状況が続くことをスタベーションという．したがってスタベーションが発生する RTOS では API の最悪実行時間を定めることができない．上記のように API を処理するためにはロックの取得を行う．単純なスピンロックとして TAS (Test And Set) スピンロックがあるが，ロックが取得できる順番はランダムに決まるためスタベーションが発生する可能性がある．ロック取得の順番を管理するためキューイングロックアルゴリズムが提案されているが，ソフトウェアで実現するとオーバヘッドが大きく，高速化するためには専用のハードウェアが必要である [19]．このため一般には TAS スピンロックが使用され，したがってこの場合 API の最悪実行時間を定めることができない．

（3） コア間とコア内の排他制御の問題

API を実行し管理ブロックにアクセスするとき，同一コアでの排他制御のために割り込み禁止を使用し，異なるコア間での排他制御のためにスピンロックを使用する．ロックの取得→割り込み禁止という順に実行すると，ロック取得から割り込み禁止までの間に割り込みが発生する可能性がある．この場合ロックをかけたまま割り込み処

理を行うため，他のコアの API を待たせてしまうことになり，この結果 API 実行時間の最悪値を定めることができない．一方，割り込み禁止→ロックの取得という順に実行をすると，ロックの取得を試みる期間割り込み禁止となるため，スタベーションを回避させていない場合割り込み応答時間の最悪値を定めることができない．上記のようにどちらを先に実行しても問題が発生する．ロックと割り込み禁止が同時に実行できれば問題は発生しないが，1 命令でロックの取得と割り込み禁止を実現できる汎用 CPU は存在しない [18].

2.4.3 本研究でのアプローチ

以上，ネットワーク処理や機械制御のためには RTOS の高速化が重要なポイントであり，一つの解決策として RTOS のハードウェア化を挙げた．しかし RTOS をハードウェア化するためにはキューの問題があることを述べた．さらにもう一つの高速化手法としてマルチコア化があるが，従来のマルチコア対応 RTOS ではマルチコア化による性能劣化や実行時間の最悪値を規定できないという問題があることを述べた．

本研究でのアプローチは以下の通りである．第 3 章では RTOS のハードウェア化の提案をおこない，低コスト・低消費電力で高速な TCP/IP スループットを実現し，実験結果を示す．この中で特に，仮想キューというキューイングシステムを提案し，膨大な量のキューを少ないハードウェアで実現し，かつ高速なキュー操作をも実現する．第 4 章では第 3 章で開発した HWRTOS 上に新たな割り込み機構を提案し，コスト・消費電力を増やすことなく割り込み応答性能を大幅に改善する．第 5 章では第 3 章，第 4 章の技術を使用し，かつ汎用プロセッサである ARM コアとのインターフェースを実装した量産型プロセッサを開発しこの性能評価を行った結果について述べる．第 6 章ではもう一つの高速化手法であるマルチコア化に対応したハードウェア化 RTOS を提案し，マルチコア特有の問題点を解決し，実施した性能評価の結果を示す．

CHAPTER 3

シングルコア対応RTOSの ハードウェア化による TCP/IP処理の高速化と低消費電力化

3.1 概要

第2章で述べたように，ネットワークの物理速度は既にGbpsレベルに達しており，これに伴いTCP/IPなど上位層のスループット向上に対する要望がある．しかし組込みシステムにおいて現状では，TCP/IP等上位層においてGbpsレベルのスループットを達成することは極めて困難な状況である．その理由は，TCP/IPプロトコルの処理には多くのCPU時間を消費するにもかかわらず，組込みシステムではコスト，消費電力の問題から高性能なプロセッサを使用できないからである．第2章ではTCP/IPプロトコルの特性上，RTOS処理をプロセッサから切り離すことでTCP/IP等ネットワークプロトコル処理を高速化できることを示した．

本章ではRTOS処理をコアから切り離すため，RTOSのハードウェア化を実現し，またこのHWRTOSを実装したプロセッサを提案する．このプロセッサは高スループッ

トの TCP/IP 等ネットワークプロトコル処理を実現し，一方で低いクロックレートで動作するため，結果として低消費電力を達成する．また本研究ではこのプロセッサの ASIC での試作を行い，その評価結果を示す．このプロセッサすなわち HWRTOS およびコアを含むシステムを ARTESSO (Advanced Real Time Embedded Silicon System Operator) と呼ぶ．ARTESSO HWRTOS は商用 RTOS と同等の機能を有している．たとえばスケジューラは FCFS (First-come and First-serve) スケジューラを実装し，また組込みシステムに必要とされる十分な数のタスク，セマフォ，イベントフラグ，メールボックスを提供する．

第 2 章において，RTOS のハードウェア化に関する従来の研究について言及したが，従来の SWRTOS と同等な機能を持った HWRTOS が無かった理由は，十分な数のキューをハードウェア的に用意できないことが理由であることを説明した．また「検索・取り出し」機能をハードウェア的に有するキューを実装することはさらに困難であることを説明した．本研究では多量のキューをハードウェアで実現する「仮想キュー（Virtual Queue）」という全く新しい技術を提案する．仮想キューは極めて少ないハードウェア量で多量の待ち行列を実現し，また「検索・取り出し」機能も実現している．

ARTESSO のアプローチは TCP/IP をハードウェア化するいわゆる TCP/IP オフロードエンジン（TOE）と異なり，ほとんどのプロトコル処理をソフトウェアで実装したまま，SWRTOS を使用したシステムと比較して，大幅な消費電力の増加を伴うことなく劇的な性能向上を実現した．これは柔軟性を必要とするプロトコル処理にとって極めて有効な手法である．

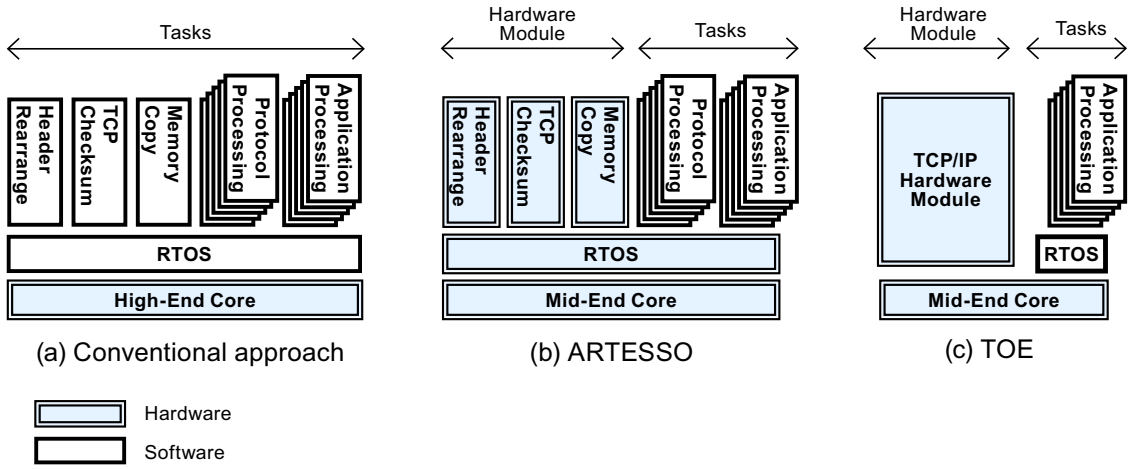


Figure 3.1: 従来のアプローチとの比較

3.2 ARTESSO

3.2.1 ARTESSO の概念

Fig.3.1 は従来の方法と ARTESSO のモジュールの構成比較を示している．Fig.2.4 で示したように従来のシステムにおいて TCP/IP 処理の実行時の解析すると，プロセッサは「プロトコル処理」の他,「メモリコピー」,「TCP チェックサム」,「ヘッダ並べ替え」,「RTOS 処理」を実行する．従ってこの構成は Fig.3.1(a) のように表される．従来の方式では上記処理のために多くの CPU 時間を消費し，したがって高スループットを得るためには，ハイエンド・コアが必要である．ARTESSO では，この解析に基づき「メモリコピー」,「TCP チェックサム」,「ヘッダ並べ替え」および「RTOS 処理」をハードウェア化することにより，Fig.2.4 のように TCP/IP 処理の CPU 負荷を大幅に削減した．この構成は Fig.3.1(b) で表される．Fig.2.4 から ARTESSO は同じプロセッサで約 10 倍のスループットを実現できるため，要求性能が従来と同等であればミッドエンド・コアの使用が可能であり，コストメリットを享受できる．また ARTESSO のプロトコル処理は従来と同じように RTOS 上にファームウェアで実現されているため，柔軟性を維持することができる．したがってプロトコルスタックの障害や規格の

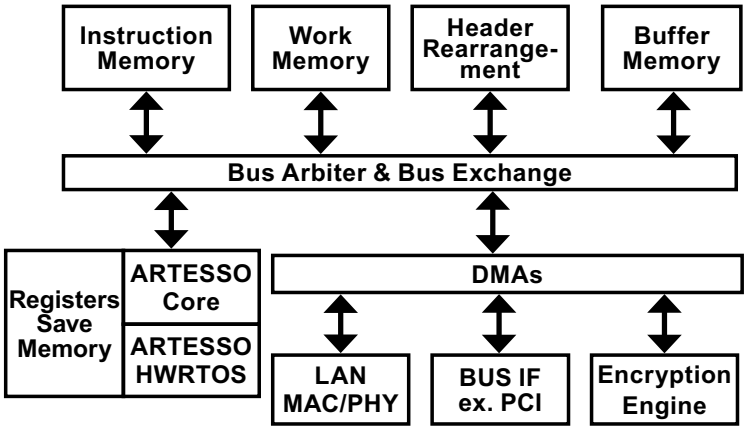


Figure 3.2: ARTESSO の全体構成

変更，ポート数の増加等の機能拡張に柔軟に対応可能である．また TCP/IP だけでなく UDP やその他のプロトコルもファームウェアとして追加実装できるばかりでなく，これらも TCP/IP と同様性能が向上する．同図 (c) はハードウェア TOE を使用した構成例である．TOE を使用することにより性能，消費電力面で要求を満たすことが可能である．しかしプロトコル処理全体がハードウェア化されているため柔軟性が著しく欠落し，バグの修正や機能追加等に対応できない．また TCP/IP 以外のプロトコルを実装したい場合は，各プロトコルに対応したハードウェアエンジンを新たに設計するか，従来通りコアの性能を上げることでしか高性能化に対応することはできない．

3.2.2 ARTESSO の高速化アプローチ

Fig.3.2 に ARTESSO の構成図を示す．ARTESSO の目的は 2.3.1.3 で示したように TCP/IP の処理において，コアがプロトコル処理に専念できる環境を提供することである．ARTESSO は 2.3.1.3 で示した TCP/IP 処理上のボトルネックを以下のように解決した．

(1) RTOS を完全にハードウェア化．

(2) 「ヘッダ並べ替え」をハードウェアで実装．

(3) ハードウェア化したTCPチェックサム計算回路をEthernet MACに実装.

(4) メモリコピーは全てDMAで行う．DMAは「バスアービター／バススイッチ（Bus Arbiter & Bus Exchange）」により，コアと並列動作可能な構造とした.

このような構造にすることにより，コアはFig.2.4に示した「プロトコル処理以外の処理」から解放され,「プロトコル処理」にほとんどのCPU時間を消費することができるようになった．上記のうち(3)に関してはARTESSOに限らずすでに一部の半導体に実装されており[20]，(4)に関しても従来からいろいろな形で検討，実施されている．(2)に関しては新しいアイディアであるが，ハードウェアでの実装は容易である．しかし(1)に関しては，2.4.1で言及したように，RTOSを高性能かつコンパクトな回路で実現することは極めて困難である．以下，ARTESSO HWRTOSについて説明する.

3.2.3 ARTESSO HWRTOSの詳細

3.2.3.1 ARTESSO HWRTOSの構造とコアの接続

Fig.3.3にARTESSOにおけるコアとHWRTOSの構成図を示す．ARTESSO HWRTOSはRTOS処理をハードウェアで実行するモジュールである．ARTESSO Coreはオリジナル設計の32ビットRISCプロセッサであり，汎用RISCとの違いはARTESSO HWRTOSとの間に専用のインターフェースを有することである．Registers Save MemoryはARTESSO Core内のレジスタ（プロクラムカウンタ，スタックポインタ，フラグレジスタ，汎用レジスタ：以降これらを合わせてProcRegと呼ぶ）をタスク毎に保管するメモリである.

ARTESSO HWRTOSの内部について説明する．Main Controller（以降，MCと呼ぶ）は各API処理を実行する回路であり，ハードウェア・ステートマシンにより実現されている．Task Control Blockは各タスクごとの情報，例えば「現在のタスクの状態」,「待ち理由」,「待ちセマフォID」などの情報を維持管理しており，情報はSelector

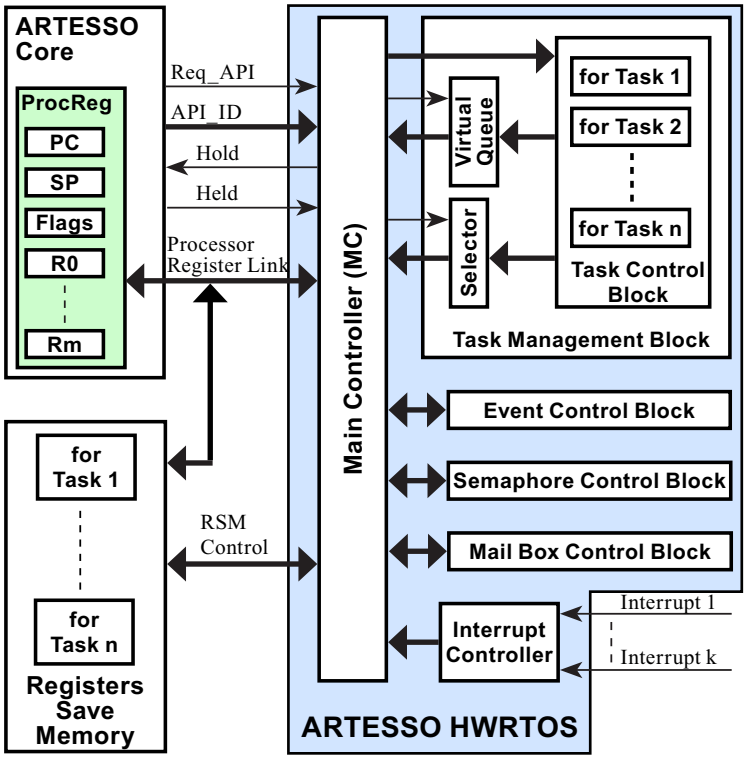


Figure 3.3: ARTESSO HWRTOS とコア

で選択され MC に提供される．Virtual Queue モジュールは RTOS で使用する全てのキューを仮想キューで実現している．Event Control Block，Semaphore Control Block，Mail Box Control Blok はそれぞれイベント機能，セマフォ機能，メールボックス機能を実現するための情報を維持管理している．MC はこれらの情報を使用し，API を実行する．

3.2.3.2　サポート API

Table3.1 に，ARTESSO HWRTOS がサポートしている API を示す．ARTESSO HWRTOS は 40 種類の ITRON 仕様 API を提供する [1]．このうちイベントフラグ待ち，セマフォ資源獲得，メールボックス受信の各 API はタイムアウトオプション，ポーリングオプションの設定が都度可能である．また，イベントフラグセット，セマフォ資源

Table 3.1: 提供される API

カテゴリ	API
同期通信機能	イベントフラグ生成、イベントフラグ削除、イベントフラグセット、 イベントフラグクリア、イベントフラグ待ち、セマフォ生成、セマフォ削除、 セマフォ資源返却、セマフォ資源獲得、メールボックス生成、 メールボックス削除、メールボックス送信、メールボックス受信、
タスク管理機能	タスク起動、自タスク終了、タスク強制終了、タスク優先度変更
システム状態管理機能	タスク優先順位回転、実行タスク ID 参照、CPU ロック、CPU アンロック、 ディスパッチ禁止、ディスパッチ許可
時間管理機能	システム時刻設定、システム時刻参照
タスク付属同期機能	起床待ち、タスク起床、待ち状態の強制解除

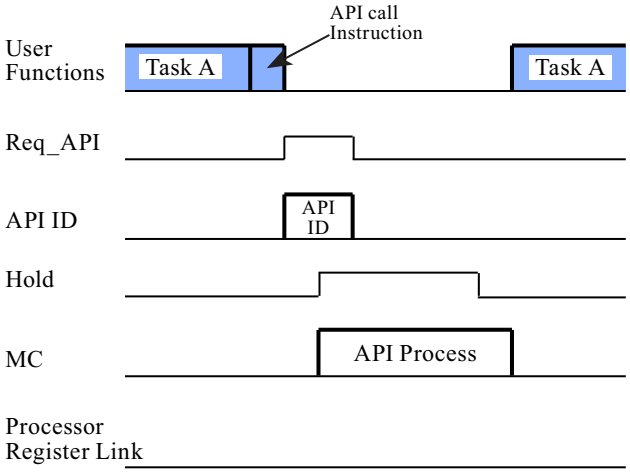


Figure 3.4: API コールのタイミング（コンテキストスイッチなし）

返却，メールボックス送信，タスク起動，自タスク終了，実行タスク ID 参照，待ち状態の強制解除，タスク優先順位回転，ロック CPU，アンロック CPU，システム時刻設定，システム時刻参照の各 API コールは ISR から発行可能である．

3.2.3.3 API 発行方法と割り込み発生時の動作

次にコアが API を発行する時の動作を説明する．コアが API 処理を MC に依頼するには，通常の関数呼び出し命令を使用し，呼び出し先として特定のアドレスを指

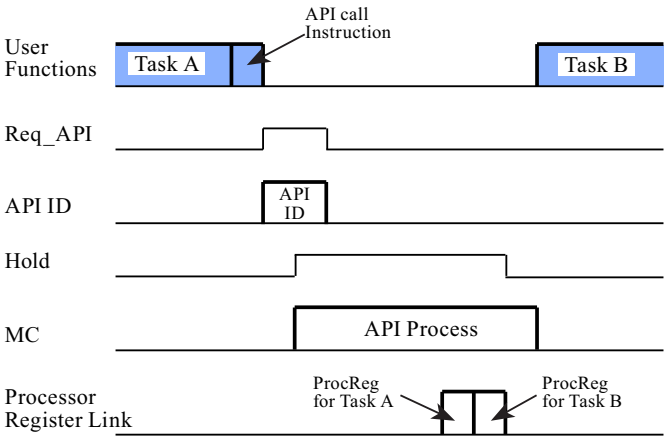


Figure 3.5: API コールのタイミング（コンテキストスイッチあり）

定することにより一般の関数呼び出しと区別をする．具体的には，0x8000 0000 番地〜0x8000 00FF 番地までを API 発行用番地と定義し，call 0x8000 0000 は「タスク起動」，call 0x8000 0001 は「自タスク終了」などと定義する．API の引数および戻り値はコアの汎用レジスタの一部を使用してやりとりする．具体的には，コアの汎用レジスタ R4〜R7 を引数用，R0 を戻り値用のレジスタとして定義する．MC は Processor Register Link を介して R4〜R7 に書かれている引数を読み出し，また戻り値を R0 に書き込む．

Fig.3.4 にコンテキストスイッチがない場合の API コールのタイミングチャートを示す．コアは API コール命令をフェッチすると，Req_API 信号を 1 にし，API_ID 信号により API の種類を MC に伝える．Req_API 信号により API 要求が伝えられると，MC は Hold 信号を 1 にすると共に，API_ID 信号で指定された API 処理を行う（Hold 信号は後述するように 1 → 0 でコアの動作再開をコアに指示するために使用する）．また API が引数を伴う場合，MC は引数のために割り当てられたコア内の汎用レジスタ（R4〜R7）を参照する．MC は API 処理を終了すると，戻り値をコア内のあらかじめ指定された汎用レジスタ（R0）に書き込む．次に Hold 信号を 1 → 0 とし，コアはこれを認識し次のプログラムカウンタ（以降 PC という）から実行を再開する．

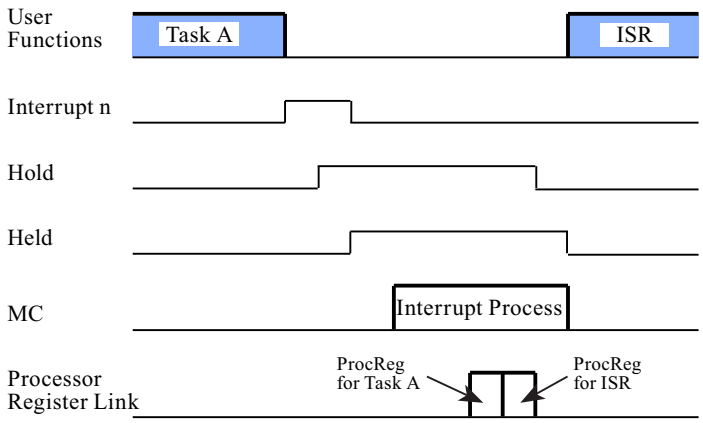


Figure 3.6: 割り込み時のタイミング

API 処理の結果，タスク切換の必要がある場合（Fig.3.5 ），MC は Hold 信号を 0 に戻す前に，Processor Register Link を介して現在のコア内の ProcReg をそのタスク ID に対応した Registers Save Memory に退避し，新しいタスクのレジスタセットを Registers Save Memory からコアの ProcReg にロードする．このあと Hold 信号を 0 にすることにより，書き換えられた PC を含むレジスタに従ってコアは動作を開始する．

次に割り込みが発生したときの動作を説明する（Fig.3.6）．割り込み発生は Interrupt Controller を介して MC に通知される．MC は Hold 信号を 1 にすることにより，コアに停止を要求する．コアは停止が完了すると Held 信号を 1 にする．コアが停止すると MC はタスク切換と同じ手順で現在のコア内の ProcReg の内容を待避し，ISR に対応するレジスタセットを Registers Save Memory からコアの ProcReg にロードする．このあと Hold 信号を 0 にすることにより，コアは ISR に対応したレジスタセットに従って動作を開始する．

3.2.3.4　仮想キュー

仮想キューは今までにない新しい概念のキューの実現方法である．まず従来の FIFO の構造と仮想キューの構造を比較することにより，仮想キューが少ない回路規模で構

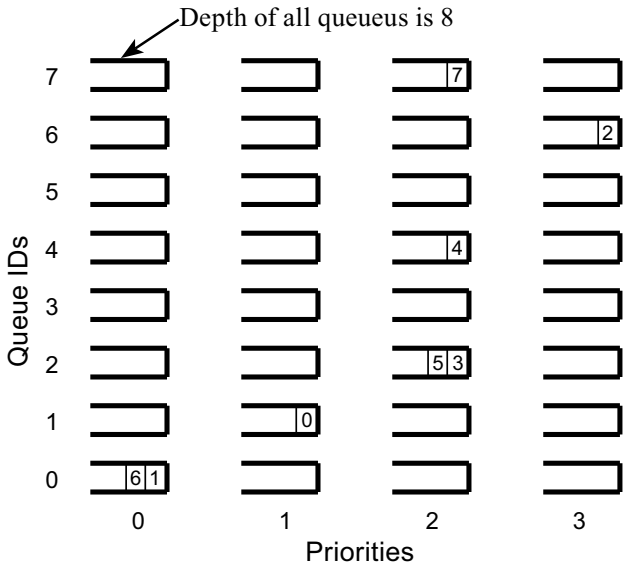


Figure 3.7: ハードウェア FIFO によるキューシステム

成できることを説明する．Fig.3.7 は従来のハードウェア FIFO を使った RTOS のキューシステムの構造を示している．この例のキューシステムは，タスクの数が 8 個，セマフォ ID が 8 個，タスクの優先度の数は 4 つであると仮定する．したがってセマフォ待ちの FIFO の数は 8 × 4 で 32 個必要である．また各 FIFO の深さはタスクの最大数，すなわち 8 必要である．各タスクはセマフォを獲得するために図のようにキューの中に入っていく．タスクの数は 8 個しかないため全てのタスクがキューに入ったとしてもにキューは常に閑散としている．一方，一つのキューに待ちが集中することも考えられ，キューごとに最大数のタスクが入っても問題ないよう，最大数の深さを用意しておく必要がある．つまり蓄積部分の最大使用率は 1 / 32 になる．蓄積部分の最大使用率はタスクの数には関係がない．一般的な表現をすると，RTOS で n 個のキューが存在すれば，FIFO の蓄積部分の最大使用率は 1 / n になり，極めて使用効率が悪いことがわかる．

本研究において試作した ASIC では，セマフォ ID 数 32 個，イベントフラグ ID 数 32 個，メールボックス ID 数 192 個実装されており，キューの総数は，

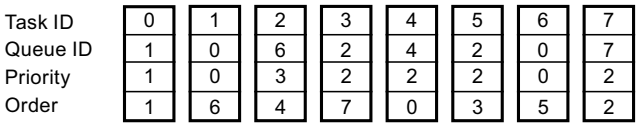


Figure 3.8: Queue control registers

$(32 + 32 + 192) \times 16 = 4{,}096$ 個

である．したがってもし従来のハードウェア FIFO で構成したとするとこの蓄積部分の最大使用率は 1 / 4,096 となる．

仮想キューの考え方は従来のキューシステムの考え方と全く異なる．たとえば Fig.3.7 においてタスク 5 に着目する．タスク 5 は Queue ID = 2, Priority = 2 のキューの中におり，先頭から 2 番目に存在する．すなわち，Queue ID，Priority，キュー内の順序の 3 つのパラメタが確定すればそのタスクがキューシステムの中の何処に存在するかを識別できる．キューの数が膨大であってもこの 3 つのパラメタが確定できればキューの位置を確定できる．仮想キューはこの考え方に基づいている．Fig.3.8 は仮想キューのデータ構造を示している．すなわちタスク識別子毎に，現在属している「キュー識別子（Queue ID）」,「優先度（Priority）」,「キューに入った順序（Order）」を維持管理する.「キューに入った順序（Order）」はここのキューに入った順序でなく，キューシステム全体での順序を示している．維持するデータはこれだけであるため，Fig.3.7 に示す従来の FIFO に比べ無駄な蓄積領域を確保する必要が無い．また従来の FIFO 型ではキュー識別子の数が 2 倍になると必要となるリソースも 2 倍になったが，仮想キューではキュー識別子が 2 倍になってもキュー識別子を維持するメモリエリアが 1 ビット増えるだけである．したがってキューの数が大きくなればなるほど効果は大きい．

Fig.3.9 は仮想キューシステムの構成図である．MC から Q_ID および COMMAND の値を指定することによりキューオペレーションを実行する．キューオペレーションの種類は「エンキュー」,「デキュー」,「検索・取り出し」であり，出力部からは常に

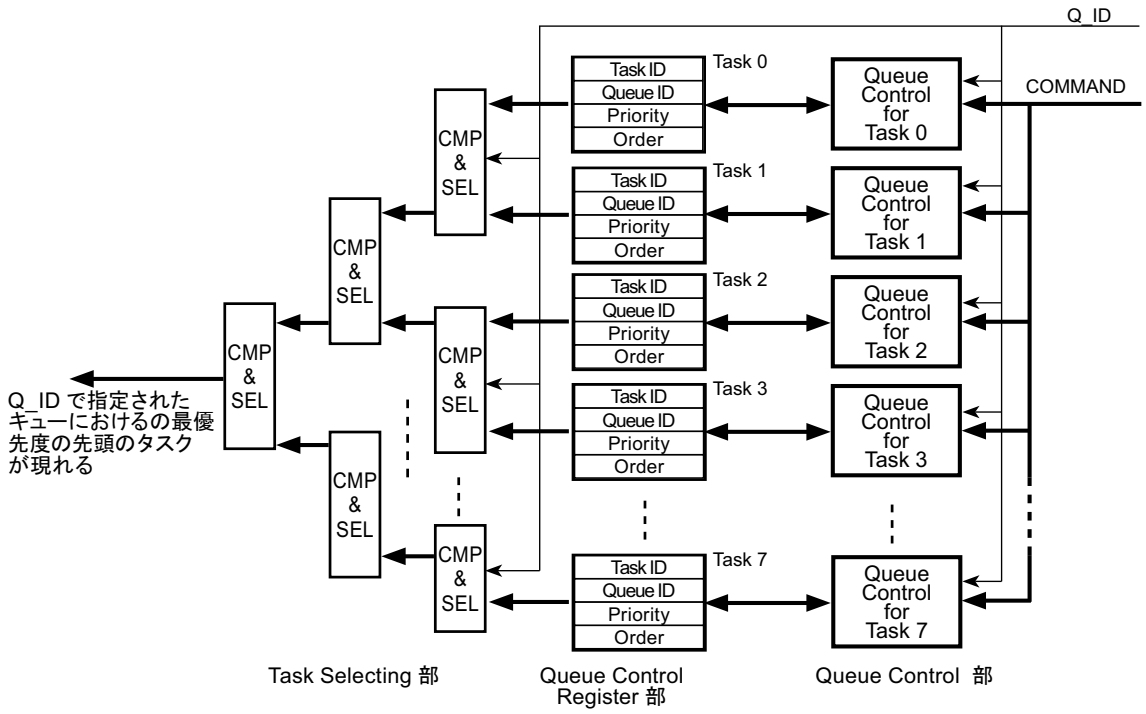


Figure 3.9: 仮想キューのアーキテクチャ

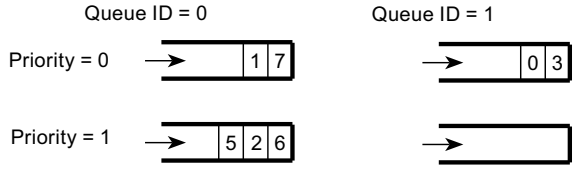
Q_ID で指定された識別子に属する最高優先度のキューにおける先頭のタスクが出力されている.「エンキュー」とはキューの最後尾に接続させること,「デキュー」とはキューの先頭から取り出すことを意味する．また「検索・取り出し」とはタスク ID を指定して，そのタスクをキューシステムの中から探し出し，そのタスクをキューから取り出すことを示す.

仮想キューは「Queue Control 部」,「Queue Control Register 部」,「Task Selecting 部」の 3 つの部分から構成されている.「Queue Control Register 部」は Fig.3.8 で示したタスク毎のキュー管理情報を記憶するレジスタである.「Task Selecting 部」は指定されたキュー識別子（Queue ID）に属するタスクを，プライオリティベース FCFS にしたがって選択し出力する．Task Selecting 部は複数の”CMP & SEL”モジュールが Fig.3.9 のように接続されている．この接続によりトーナメント回路を実現している．各”CMP & SEL”モジュールは，優先度の高い方のタスク識別子を後段に送る．優先度が同じ場

合は「キューに入った順序」が先であるタスクが選択し，後段に送る．したがって最終段から得られるタスク識別子はプライオリティベース FCFS に基づいた結果が得られる．キュー識別子の指定は”Q_ID”信号によって MC から指定される．この信号により初段の”CMP & SEL”モジュールにおいて，関連する Queue ID のみ後段に伝えられる．従って”Q_ID”信号でセマフォ ID : 23 を指定すると，セマフォ 23 のキューにおけるプライオリティベース FCFS によるタスクが選択されることになる．仮想キューはセマフォ，イベント，メールボックス等を区別しておらず，単にキュー ID のみで識別しいてるため，回路としてはセマフォ，イベント，メールボックスを別々に用意する必要はなく，また Wait キューのみならず Ready キューも回路を共有できる．「Queue Control 部」についてはキューオペレーションと密接な関連があるため，キューオペレーションの説明を先に行う．

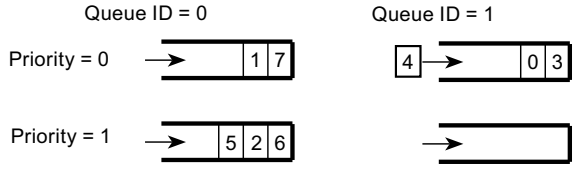
Fig.3.10 を使用してキューオペレーションの説明を行う．説明を簡単化するため，Queue ID は 2 個，Priority が 2 個のキューシステムを使って説明を行う．Priority は 0 の方が 1 より優先度が高いものと定義する．Fig.3.10(a) はキューシステムの本説明における最初の状態を示している．Order Register の値はこのキューシステムにエンキューされた順序を示しており，最大の値のものが一番古く，0 のタスクが一番最近エンキューされたことを示している．したがって，タスク 3 が一番最初にエンキューされ，一番最近エンキューされたのはタスク 5 であることがわかる．Queue ID Register はエンキューされたキューの ID，Priority Register はこのタスクの優先度を示している．タスク 4 はキューイングされていないため，Order Register と Queue ID Register が”Non”という”値”である．また Priority Register はタスク毎に定義され，タスク 4 は優先度 1 と定義されている．

同図 (b) は，(a) の状態からタスク 4 が Queue ID 1 にエンキューされたときの状態を示している．タスク 4 の Queue ID レジスタには 1 が書き込まれる．またタスク 4 の Order Register には 0 が書き込まれる．なぜならばタスク 4 が一番最近にエンキューされたタスクになるからである．他のタスクの Order Register は「一つ古くなる」た



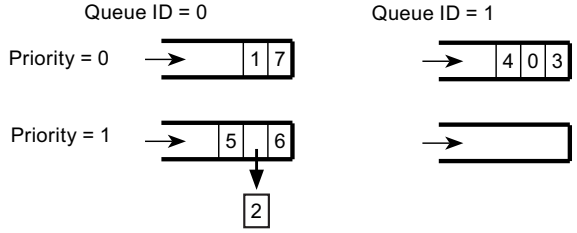
Task ID Register	Order Register	Queue ID Register	Priority Register
0	4	1	0
1	1	0	0
2	3	0	1
3	6	1	0
4	Non	Non	1
5	0	0	1
6	5	0	1
7	2	0	0

(a) 最初の状態



Task ID Register	Order Register	Queue ID Register	Priority Register
0	4 → 5	1	0
1	1 → 2	0	0
2	3 → 4	0	1
3	6 → 7	1	0
4	Non → 0	Non → 1	1
5	0 → 1	0	1
6	5 → 6	0	1
7	2 → 3	0	0

(b) コンテキスト "4" をエンキュー



Task ID Register	Order Register	Queue ID Register	Priority Register
0	5 → 4	1	0
1	2	0	0
2	4 → Non	0 → Non	1
3	7 → 6	1	0
4	0	1	0
5	1	0	1
6	6 → 5	0	1
7	3	0	0

(c) コンテキスト "2" を検索・取り出し

Figure 3.10: キューオペレーション

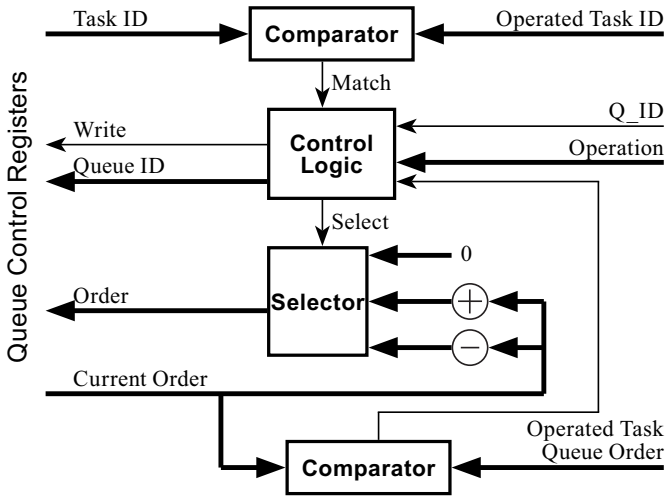


Figure 3.11: Queue Control ブロック

め，1 加算された値になる．

同図 (c) は (b) の状態においてタスク 2 を検索し，取り出すオペレーションである．このオペレーションではキューシステムの中からタスク 2 を探し出し，これをキューから外す．Queue Control Register 自体がタスク ID 順になっているため，タスク 2 を検索することは簡単である．タスク 2 はキューから外されるため，Order Register と Queue ID Register には”Non”が書き込まれる．その他のタスクの Order Register は，タスク 2 の順序番号より大きい順序番号は 1 減算，タスク 2 の順序番号より小さい順序番号はその値が維持される．このような処理をすることにより検索・取り出しオペレーションが実行される．

デキューは検索・取り出しオペレーションの一変形になる．たとえば同図 (b) において，Queue ID 0 の先頭のタスクをデキューしたい場合のオペレーションは以下の通りである．まず MC は Q_ID 信号に 0 を設定すると，CMP & SEL の最終段からはタスク 7 を得ることができる．その後 MC はタスク 7 を指定し，検索・取り出しオペレーションを実行することによりデキューを達成する．

Fig.3.11 は各タスクごとに配置されている Queue Control モジュールの内部構造を示

している．各 Queue Control モジュールは，各タスクの Queue Control Register の書き換えを実行することにより，先に説明したキューオペレーションを実現する．”Operation”信号，”Operated Task ID”信号，”Operated Task Queue Order”信号は，Fig.3.9 で示した COMMAND 信号の要素であり，MC がこれらの値を指定する．”Operation”信号は「エンキュー」,「デキュー」,「検索・取り出し」等のオペレーションを示している．また”Operated Task ID”信号，”Operated Task Queue Order”信号はそれぞれオペレーションを行う対象のタスク識別子，キューの順序番号を示している．これらの信号と属しているタスク，現在の順序番号を比較したデータが Control Logic に与えられる．Control Logic は”Operation”信号が発生したとき，これらの情報に基づき Queue Control Register を書き換える．

3.3 評価

3.3.1 実験方法

この章では ARTESSO と従来の解決方法の比較を行い，結果を示す．RTOS 性能に関しては以下のように実験を行った．従来の RTOS として NORTi を選択した．NORTi は ITRON 仕様の商用 RTOS であり，RTOS カーネルと TCP/IP モジュールを含んでいる．NORTi を ARM926 評価ボード上で走らせ，各 API に要するクロックサイクル数を測定した．一方 ARTESSO は Verilog シミュレータにより各 API に費やされるクロックサイクル数を求めた．

次に TCP/IP 処理の CPU 時間占有率について以下のように求めた．従来の性能は，ARTESSO Core シミュレータで NORTi を走らせ，プロファイリングを作成し，求めた．このとき，ARTESSO Core 以外の機能，ARTESSO HWRTOS や Register Save Memory は非活性化した．ARTESSO の CPU 時間占有率は，ARTESSO シミュレータ上で TCP/IP を走らせプロファイリングを作成し，求めた．このとき ARTESSO 機能

Table 3.2: RTOS 性能比較

単位：クロックサイクル

API	ディス パッチ	NORTi (ARM926)	Silicon TRON	ARTESSO HWRTOS
起床待ち	有	628	64	10
タスク起床	有	496	82	10
タスク優先度変更	有	541	103	11
メールボックス受信	無	224	-	7
メールボックス受信	有	591	-	11
メールボックス送信	無	360	-	8
メールボックス送信	有	541	-	11
セマフォ資源獲得	無	216	-	6
セマフォ資源獲得	有	558	83	9
セマフォ資源返却	無	344	-	7
セマフォ資源返却	有	536	83	11

注 : "-" は未実装または情報なし

を全て活性化している．双方とも送信側 TCP は 1,460 バイトフレームを送信するものとし，送信 2 フレームに対し 1 フレームの ACK フレームを受信側が送信するものとした．

TCP/IP スループットの計測は以下のように行った．従来の TCP/IP 性能は ARM926 評価ボードに NORTi を実装し動作クロック 50MHz で測定した．ARTESSO は 150nm プロセスで開発したゲートアレイを使用し，動作クロックを同じく 50MHz に設定し，測定した．

3.3.2 実験結果

Table3.2 に従来の SWRTOS と ARTESSO HWRTOS の性能比較を示した．また 2.4.1 で述べた Silicon TRON の性能も示した．数値は各 API 実行に必要なクロックサイクル数を示している．ARTESSO HWRTOS は従来の SWRTOS に比較して 30〜60 倍高速であることがわかる．また Silicon TRON に比較しても 6〜9 倍高速である．

Table 3.3: CPU 占有時間

Process Category	Time Occupancy (%)	
	NORTi	ARTESSO
Protocol processing	10.5	92.7
RTOS	32.1	2.8
Header Rearrangement	15.3	0.0
TCP checksum	32.2	0.0
Memory copy	9.9	4.5
Total	100.0	100.0

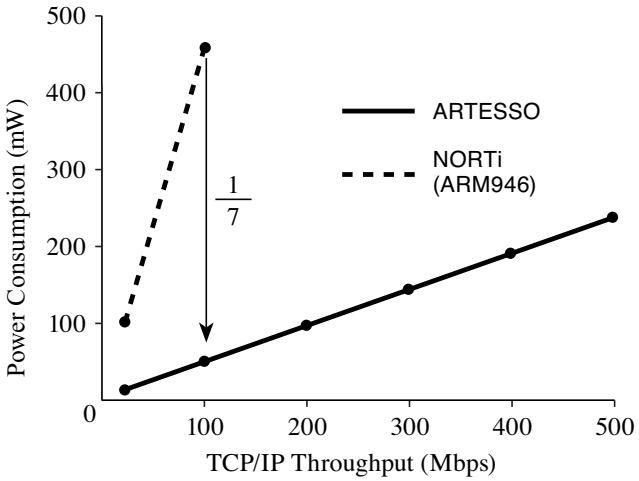


Figure 3.12: スループットと消費電力の関係

Table3.3 は TCP/IP の各処理における CPU 占有時間を示している．ARTESSO では RTOS，ヘッダ並べ替え，TCP チェックサム，メモリコピー処理からコアを解放したため，プロトコル処理に 92.7%の時間をコアに割り当てることができた．

TCP/IP のスループット測定の結果は，NORTi が動作クロック 50MHz の ARM926 で動作した場合 11Mbps，一方 ARTESSO は同じ動作クロックで 125Mbps であった．言い換えると，ARTESSO はミドルエンドコアでも 100Mbps 超の TCP/IP スループットを達成できるが，従来の方法では 100Mbps のスループットを達成するためにはハイエンドコアが必要であると言える．

Fig.3.12 は ARTESSO と ARM926 上で動作させた NORTi の消費電力を示してい

る．ARTESSO は Verilog シミュレータによるゲートレベルシミュレーションと電力計算ツール（Power Compiler）を使用して測定した．この消費電力にはメモリコピー，ヘッダ並べ替え，TCP チェックサム回路の消費電力および TCP/IP を動作させるために必要なメモリの消費電力も含んでいる．NORTi の消費電力は ARM926 のデータシートから計算した．この消費電力は TCP/IP を動作させるために必要なメモリ消費電力も含んでいる．また双方とも TCP/IP スループットが動作クロックに比例すると仮定している．ARTESSO の命令メモリ，ワークメモリは全て LSI 内部に存在し，キャッシュメモリ等は使用していない．従ってコアパフォーマンスは完全に動作クロックに比例し，これは論理シミュレーションでも実証されている．一方 ARM 等の汎用コアにおいては最良条件においてコアパフォーマンスは動作クロックに比例する．この比較では比較対象（ARM）を最良条件で比較した．この結果 TCP/IP スループットが 100Mbps のとき，NORTi / ARM926 は 454mW，ARTESSO は 65mW であり，ARTESSO は従来技術の 1/7 の消費電力で同じ TCP/IP 性能を実現した．

3.3.3 ASIC での実現

90nm プロセスの ASIC を使用して ARTESSO を実現した．Fig.3.13 に ASIC の写真及び機能の配置を示す．この ASIC では Fig.3.2 に示した機能のほかに PCI バス，USB インターフェースを追加している．またこの ASIC では ARTESSO HWRTOS として 32 タスク，優先度は 16 を実装している．またセマフォとイベント識別子は 32 個，メールボックスの識別子は 192 個実装されている．すなわちこの ASIC に実装されているキューの数は 4,096 個である．この ASIC は最大動作周波数 150MHz で，このとき 515Mbps の TCP/IP スループット達成を測定により確認した．

Table3.4 はこの ASIC の各機能の面積を示している．ASIC の総面積が 27,601,284μm^2 であるのに対し，ハードウェア RTOS の面積は 540,170μm^2 であった．また仮想キューの面積はわずか 30,213μm^2 であり，これはゲート換算で約 38,000 ゲートである．上記のように本 ASIC には 4,096 個ものキューが実装されており，この結果仮想キュー

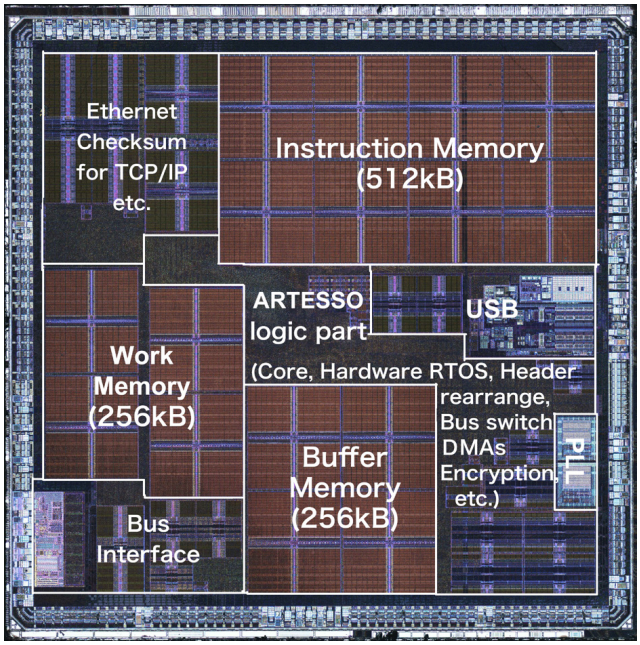


Figure 3.13: ARTESSO 90nm ASIC

Table 3.4: 構成要素毎の面積

	Area (μm^2)
Total area	27,601,284
RAM for Instruction, Data, Buffer	14,851,332
ARTESSO	2,919,025
RTOS	540,170
Logic	372,964
Virtual Queue	30,213
Others	342,751
RAM	167,206
Others (Core, Header Rearrange, DMA, Encryption, etc.)	2,378,855
Others (Ethernet, USB, Bus interface)	9,830,927

が極めて小規模の回路で多量のキューを実現することが実証された.

3.4 むすび

従来の方法で数百 Mbps の TCP/IP スループットを達成しようとすると，ハイエンドコアを使用し，GHz クラスの動作クロックで動作させる必要があった．この場合消費電力は数 W 以上になり，モバイルシステムやコンシューマプロダクトでは受け入れがたい．さらにこうしたハイエンドプロセッサは高価であり，また熱対策のための配慮にもコストがかかる．ARTESSO は性能を落とすことなく低いクロックレートでの動作を実現することにより，圧倒的な低消費電力を実現した．またその際 TOE のようなシステムの硬直化を招くことなく，従来通りファームウェアでネットワークプロトコルを実現することによりシステム柔軟性を維持した．

ARTESSO HWRTOS の中核技術は「仮想キュー」である．仮想キューは極めて少ない回路で膨大な量のキューを実現する．ASIC 実現およびそのデータより，これが検証された．

CHAPTER 4

超高速応答を実現する ハードウェア割り込み処理機構

4.1 概要

第 2 章で示したように，モーター制御ではリアルタイム処理すなわち周期的かつ確実な処理の実行が要求される．自動車のステアリングコントロールでは 100μ 秒周期の割り込みを使用し，割り込み発生から 1μ 秒以内に制御ソフトウェアが実行されることが要求されることを示した．SWRTOS を使用した現状技術ではこのような高速割り込み応答性能を実現することができないため，RTOS を活用できず，この結果 RTOS が提供する利点，すなわちソフトウェアの開発生産性の向上，ソフトウェアの信頼性の向上と言ったメリットを享受できていない．

第 2 章において，リアルタイム処理を必要とするアプリケーションでは周期を短くすることが性能向上を意味し，周期を短くするためには割り込み応答性能を向上させることが重要であることを示した．また FA ネットワークにおいてもネットワークを介してリアルタイム処理が必要であり，周期を短くするためには RTOS の割り込み応答性能の向上が必要であることを示した．

一方第 3 章において，RTOS をハードウェア化することにより割り込み発生から ISR 起動までの時間を著しく向上させることが可能になることを示した．

本章では，従来の SWRTOS を使用したシステムでは割り込み応答性能に変動があること，この原因が tick 処理であることを示し，この処理のハードウェア化により ARTESSO HWRTOS の割り込み応答時間をさらに短くするだけでなく，変動を最小に抑えることができることを示す．また ISR 処理をハードウェア化することにより，さらに割り込み応答性能を向上させ，割り込み時のオーバヘッドを大幅に削減し，特に割り込みが頻発するアプリケーションにおいてさらなる性能向上を実現できることを示す．

4.2 TICK オフローディング

4.2.1 従来の TICK 処理の問題点

割り込み応答性能に影響を与える一つの要因が「tick 処理」である．tick 処理は周期割り込みにより起動され，RTOS で使用しているウエイトタイマの更新，タイムアウトの検出，タイムアウト時の処理を行う．従って tick 処理は RTOS がリアルタイム機能を提供する上で極めて重要な機能である．一方 tick 処理は上記の RTOS 内の排他的な管理データの参照・変更を行うため，割り込み禁止状態で実行される．またリアルタイム性を維持するために tick 処理の優先度を高く設定することが必要であり，このため周期割り込み以外の割り込み処理の応答時間を大きく変動させる．

過去の研究では，[21] においては RTOS のボトルネックを分析し，その一つに tick 処理を挙げ，tick 処理のハードウェア化を行っている．但しこのシステムは，API 処理自体はソフトウェアで行っており，性能改善は限定的である．

4.2.2 従来の TICK 処理の動作

次に従来の SWRTOS の tick 処理の実際の動作を説明する．一般に各タスクは TCB (Task Control Block) と呼ばれる各タスク固有の情報をまとめた構造体を有している．「イベントフラグ待ち」,「セマフォ資源獲得」,「メールボックス受信」などタイムアウトオプションを有する API を発行すると，引数で指定されたタイムアウト値がこの構造体のタイムアウトフィールドに書き込まれる．またこのような API を発行してこのタスクが WAIT 状態になると言うことは，この TCB が Fig.2.2 で示されるように該当する WAIT キューの最後尾に接続されるということである．

tick プロセスは，tick 専用の周期的割り込みによりが起動される．tick プロセスは各 WAIT キューの先頭から順次 TCB を見て行き，各 TCB のタイムアウトフィールドの値から周期値を減算する．もしタイムアウトフィールドの計算結果が 0 以下になったら，このタスクを WAIT キューから外し，READY キューに接続する．WAIT キューに接続されている全ての TCB のタイムアウトフィールドの減算が終わると，tick プロセスを終了する．

4.2.3 ハードウェア TICK 処理

ARTESSO HWRTOS では上記処理をハードウェア化した．これを tick オフローディングと呼ぶ．ARTESSO HWRTOS の構造は Fig.3.3 に示したとおりである．Task Management Block の中に Task Control Block が各タスク毎に存在する．基本的に各 Task Control Block はレジスタ群で実装されており，タスクタイプ，タスク状態，初期タスク優先度，カレントタスク優先度，待ち理由などの情報が維持されている．タイムアウト用のレジスタも TCB で維持管理されている．タイムアウトレジスタはハードウェア・ダウンカウンタで構成される．従って，何らかの値が書き込まれると自動的にデクレメントされる．ダウンカウンタの値が 0 になると MC に通知される．

コアが「イベントフラグ待ち」等タイムアウトオプションを有する API を発生し，

MC がこの API を処理する過程でこのタスクを WAIT 状態に遷移させると判断したとき，MC はこの API のタイムアウト引数を取り出し，API を発行したタスクの Task Management Block 中のタイムアウトレジスタにこの引数の値を初期値として書き込む．上記のようにタイムアウトレジスタ値は自動的に減算され，待ち状態の解除要因が発生する前に 0 になると MC に通知される．MC は割り込み発生時と同様に Hold 信号を 1 にすることによりコアを停止させ，Held 信号によりコアの停止を確認したらタイムアウトしたタスクを WAIT キューから取り外し，READY キューに接続する．最後にディスパッチ処理を行い，プライオリティベース FCFS に基づきタスクを選択して RUN 状態にし，Hold 信号を 0 にすることによりコアを再起動する．

以上のような構成にすることにより，tick 処理をソフトウェアで実行する必要が全くなくなった．したがって割り込み禁止期間は MC が，タイムアウトレジスタが 0 になった通知を受け取ってからディスパッチが完了するまでの期間のうち割り込みを認識できない数サイクル時間のみとなり，割り込み応答時間が極めて短くなっただけでなく，割り込み応答時間の変動幅を大幅に削減することができた．またハードウェアタイマの採用により，タイマデクレメントの解像度が 3 桁向上し，この結果アプリケーションのリアルタイム精度を大幅に向上した．一方，従来のソフトウェア tick に比較しハードウェア回路の増加以外不利益になる点は特にない．tick オフローディングのための論理回路の増加量は約 14k ゲートであった．

4.3 IIA オフローディング

本節では，割り込み処理を完全にハードウェア化し割り込み時のオーバヘッドを大幅に削減するシステム，IIA (Interrupt Invoked API) オフローディング機構について述べる．

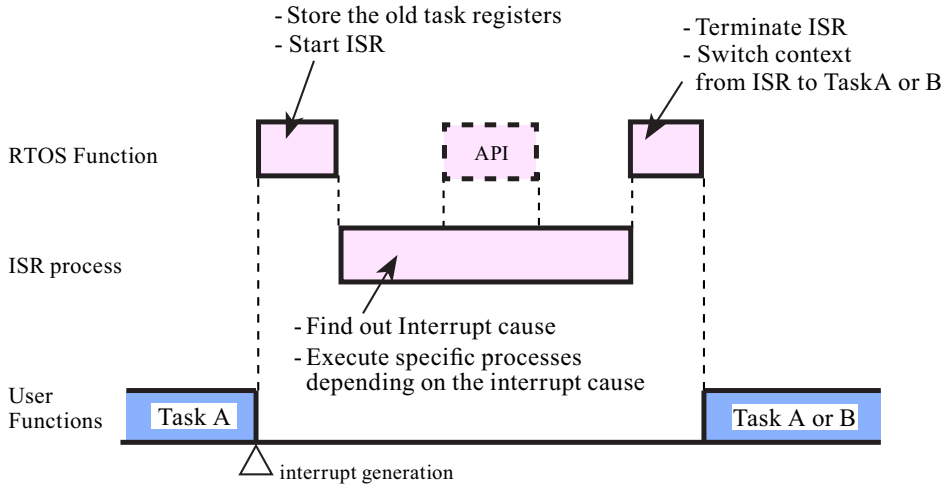


Figure 4.1: 典型的な割り込み処理

4.3.1 SWRTOS の割り込み処理

まず SWRTOS を使用した場合の典型的な割り込み時の処理の流れについて Fig.4.1 を使用して説明する．タスク A が実行されているときに割り込みが発生すると，RTOS に処理が移る．RTOS ではコンテキストのスイッチを行い，ISR (Interrupt Service Routine) を起動する．ISR では割り込み要因が何であるかを確定し，要因に対応した処理を行う．ISR は他のタスクより優先度が高く，割り込み禁止状態で行われるため，ごく簡単な処理のみを行うのが一般的である．割り込みに対応した長い処理がある場合，割り込みに対応したタスクを起動し，このタスク内で長い処理を実行するなどの手法を採る．ISR 内で API（たとえば「イベントフラグセット」,「セマフォ資源獲得」,「待ち状態の強制解除」など ISR 内で発行を許されている API）を発行することにより割り込み要因に対応したタスクを WAIT 状態から READY 状態にすることができ，ISR 自体は短い時間で終了することができる．ISR が終了すると RTOS はタスク A に戻るか他のタスク B がタスク A より優先度が高くなったためこれを実行させるかどうかを判断しいずれかのタスクを RUN 状態にする．

上記の様に一度割り込みが発生すると様々な処理を行わなければならない．割り

込みが発生し，ISR 内で API を一つ発行するだけの単純な処理でも TaskA が中断されてから次に TaskA または B が実行を開始するまでの時間は数百〜千数百サイクルの時間が消費されることがわかっている．タスクから見るとこれらの処理は単なるオーバヘッドであり，できるだけ短くすることが望ましい．特に割り込みが頻繁に発生するネットワーク処理や機械制御ではこのオーバヘッド時間を短縮することにより，同じシステム性能を低い性能のコアで実現することが可能となり，低コスト化，低消費電力化に大きく貢献できる．

4.3.2 関連研究

[22] では，ISR をハードウェア化し性能を向上させる概念が記載されているが具体的な実現方法が記されておらず，また割り込み応答性能に関するデータは全く記載されていない．従来の研究にあっては，RTOS の性能向上による CPU 時間の RTOS 占有率の低減化にフォーカスされており，割り込み応答性能に関して具体的な数値を示した研究が見あたらない．

4.3.3 ISR のハードウェア処理化

以上のように割り込み発生は多くのオーバヘッドを伴うため，割り込みが発生してから次にタスクが実行できるまでの処理全体をハードウェア化し，オーバヘッドを最小化することを試みた．まずハードウェア化を実現するため ISR 処理を定型化した．これを Fig.4.2 (1) に示す．割り込みが発生すると，処理が RTOS に処理が移る．RTOS ではコンテキストのスイッチを行い，ISR を起動する．ISR は割り込み要因を確定し，要因に対応した API を発行し，RTOS で API 処理を行う．API 処理が終了すると ISR に処理が移動し，必要であればさらに他の API を発行する．そのあと他の割り込み要因がないかどうかなどの確認を行い，なければ ISR を終了し RTOS に戻る．RTOS はこのあとディスパッチ処理を行う．以上のように割り込み発生時の処理を定

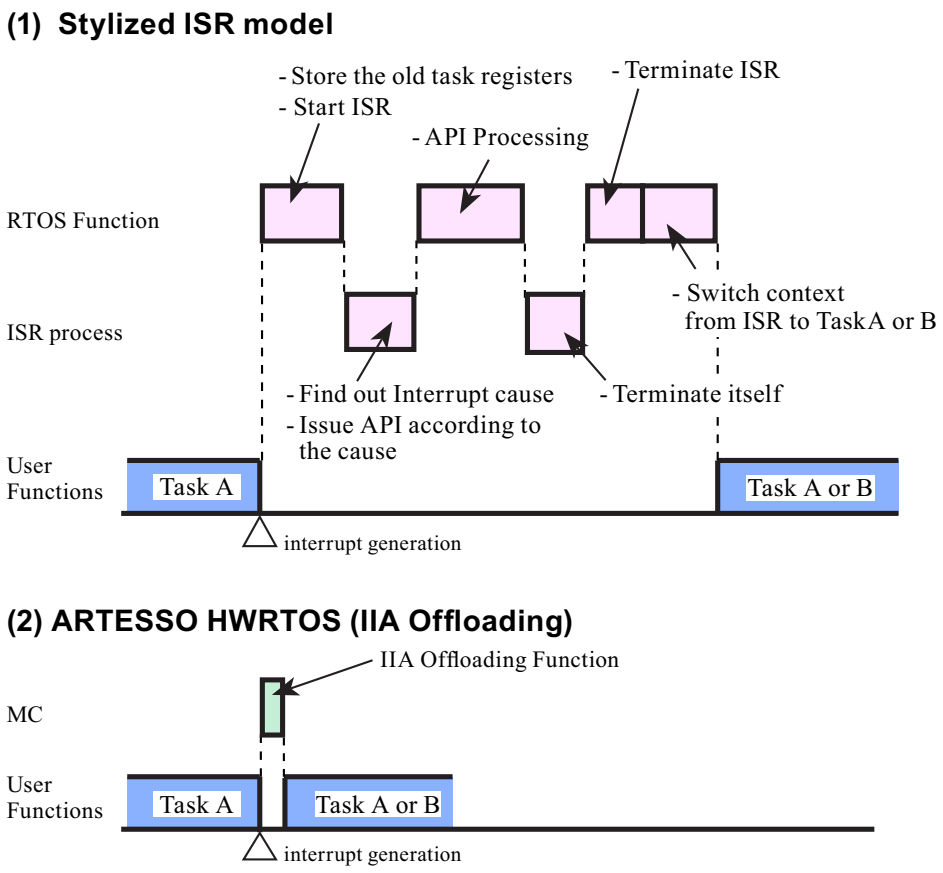


Figure 4.2: IIA オフローディング

型化し，ハードウェア化した．この機能を IIA (Interrupt Invoked API) オフローディングという．Fig.4.2 (2) にこれを示す．割り込みが発生すると MC は Hold 信号を使用してコアを停止させる．Held 信号によりコア停止が確認出来ると上記処理を実行する．終了すると Hold 信号を 0 にしてコアの実行を再開する．このとき TaskA 処理に戻るときは ProcReg の値はそのまま，TaskB に遷移するときは ProcReg の中身を交換してから Hold 信号を 0 にすることによりコンテキストスイッチを実現する．

Fig.4.2 (1) に示す処理のうち，RTOS の処理は既にハードウェア化している．IIA オフローディングを実現するために追加した機能は，割り込み要因に対応した API をハードウェアが発行する機能である．IIA オフローディングでは，IIA オフローディングを使用するか，従来の ISR を使用するかを，各割り込み要因毎にプログラマブル

に選択できるようにした．また割り込み要因毎に発行する API の設定もプログラマブルにした．発行可能な API は「イベントフラグセット」,「セマフォ資源解放」,「待ち状態の強制解除」,「タスク起床」の各 API である．また一つの割り込み要因で複数の API の発行が可能である．

IIA オフローディングの短所はソフトウェア ISR 処理のように ISR 処理の中で任意の処理を実行できないことである．ISR 内で実行できるのは API 発行のみである．しかし IIA オフローディングを使用すると，割り込みが発生してから次にタスクが実行されるまでの時間は，SWRTOS システムにおいて割り込みが発生してから ISR が実行されるまでの時間に比較し圧倒的に短い．このため IIA オフローディングにより起床したタスク内で個別処理をソフトウェアで実行しても問題無いと考えた．またどうしても ISR 内で処理したい場合は，その割り込み要因は IIA オフローディングでなく ISR で処理するようにレジスタを設定することにより従来通り ISR によりソフトウェア処理が可能である．この場合であっても ISR の起動は HWRTOS によって行われるため，極めて高速な応答になる．

以上のように IIA オフローディングは従来のソフトウェア ISR ほどの柔軟性は持ち合わせていないものの，ISR で発行する API およびパラメタの登録をプログラマブルにすることにより柔軟性を確保しつつ最大限の高速化を実現した．IIA オフローディング処理のために追加した回路の規模は割り込み要因数 8 のとき，12k ゲートであった．

4.4 評価

本節では tick オフローディングと IIA オフローディング機能を実装した ARTESSO HWRTOS と一般の SWRTOS の割り込み応答性能の比較を行い，結果を示す．

4.4.1 実験環境

従来の SWRTOS の性能測定には以下の環境を使用した.

- コア : V850 (μPD70F3318YGJ)

- コア内部クロック : 20MHz

- RTOS : μITRON4.0 (TOPPERS JSP カーネル)

測定方法は以下の通りである．外部より割り込み信号を入力し，測定したいタイムポイントでソフトウェアから IO ポートに信号を出力した．これらの信号をオシロスコープにより表示し，割り込み応答時間を計測した．結果は内部クロック周波数 20MHz との積をとり，サイクル時間に換算した.

一方 ARTESSO HWRTOS は Verilog シミュレータにより各処理に費やされる時間を求めた．クロック設定は上記 V850 コアクロックと同じ 20MHz に設定した.

4.4.2 実験内容

実験では以下で示す 2 つの時間を測定し従来の SWRTOS と ARTESSO HWRTOS の割り込み応答性能を比較する.

- 割り込み発生から ISR 起動まで

- 割り込み発生からタスク A または B 起動まで

SWRTOS の場合，そのとき動作しているタスク数，キューに接続されているタスクの数，その他その時点の RTOS の内部状況により，応答時間が異なることが予想される．本実験ではいくつかの状況を設定し，上記予想が正しいことを示す．また ARTESSO HWRTOS の場合動作しているタスク数やキューの状況に応答時間が影響しないことを示す.

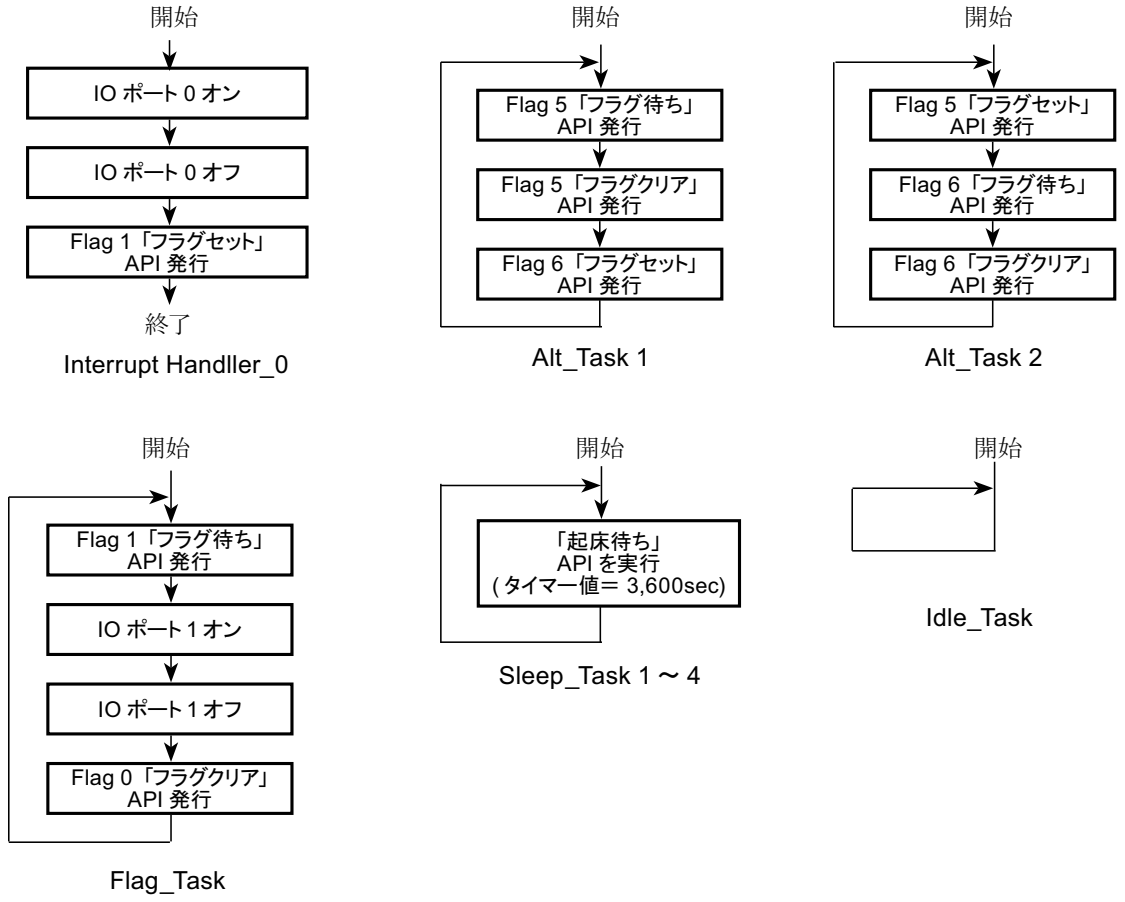


Figure 4.3: 各タスクの処理

4.4.2.1 SWRTOS の実験方法

4.4.1 に示した ITRON 環境上に以下のソフトウェアを実装し性能を測定した．使用する ISR およびタスクは Interrupt_Handler_0，Cyclic Handler，Flag_Task，Alt_Task 1～2，Sleep_Task 1～4，Idle_Task である．Cyclic Handler は ITRON で定義されており，tick 処理を行っている．その他は Fig.4.3 に示すとおりである．また各タスクの優先度は Flag_Task：優先度 5，Alt_Task 1～2：優先度 12，Sleep_Task 1～4：優先度 6，Idle_Task：優先度 12 とした（小さい数ほど優先度が高い）．外部より 200μ 秒の周期で割り込みを入力し，この割り込み信号により Interrupt_Handller_0 が起動されるものとする．また Cyclic Handler はプロセッサ内部タイマにより 1 ミリ秒の周期で起動さ

れる．実験の手順は以下の通りである．

実験 1:初期化時 Flag_Task を生成する．外部割り込みが発生する度に Interrupt_Handller_0 (ISR に相当）が起動され，Flag 1 をセット．これにより Flag_Task が起床する．IO ポート 0，1 の信号をオシロスコープで 15 分間観察することにより「割り込み発生から ISR 起動までの時間」,「割り込み発生から Flag_Task 起動までの時間」を測定する．

実験 2：実験 1 に追加し，さらに Idle_Task を生成．各 15 分間オシロスコープで観察し，同様の測定を行う．

実験 3：実験 1 に追加し，さらに Alt_Task 1 および Alt_Task 2 を生成．各 15 分間オシロスコープで観察し，同様の測定を行う．

実験 4：実験 1 に追加し，さらに Sleep_Task1〜4 を生成．各 15 分間オシロスコープで観察し，同様の測定を行う．

4.4.2.2　ARTESSO HWRTOS の実験方法

Verilog シミュレーションにより実験 1〜4 における「割り込み発生から ISR 起動までの時間」,「割り込み発生から Flag_Task 起動までの時間」を計測する．ソフトウェアはシミュレータの ROM 上に実現する．ソフトウェアによる tick 処理の必要はないので，Cyclic Handler は実装しない．

4.4.3　実験結果

Table 4.1 に実験結果を示す．まず「割り込みから ISR までの時間」の結果について考察する．実験 1 では割り込み発生時，全くタスクが走っていない．それにもかかわらず SWRTOS における ISR 起動までの時間は一定でなく，334 サイクルもの変動がある．これは tick 処理の影響と考えられる．一方 ARTESSO HWRTOS は常に一定の時間で ISR が起動している．

実験 2 では Idle Task が常時起動している．したがって，Idle Task 実行中に割り込みが発生する．SWRTOS においては実験 1 に比較し ISR 起動までの時間が 70〜90 サ

Table 4.1: 割り込み応答性能

単位：クロックサイクル

実験	生成タスク	割り込みから ISR 起動までの時間						割り込みから Flag_Task 起動までの時間					
		SWRTOS			ARTESSO HWRTOS			SWRTOS			ARTESSO HWRTOS		
		最小	最大	変動幅	最小	最大	変動幅	最小	最大	変動幅	最小	最大	変動幅
1	Flag_Task	94	428	334	5	5	0	532	1,456	924	14	14	0
2	Idle_Task, Flag_Task	97	524	427	9	9	0	756	1,676	916	14	14	0
3	Alt_Task1, 2, Flag_Task	96	504	408	9	13	4	772	1,960	1,188	14	17	3
4	Sleep_Task1 ～ 4, Flag_Task	92	500	408	5	5	0	752	1,832	1,080	14	14	0

イクルも増加しており，変動幅は 400 サイクル以上に拡大した．

実験 3 では Alt_Task 1 と Alt_Task 2 が交互に起動する．したがって割り込みは Alt_Task 1 または Alt_Task 2 タスクの実行中，もしくは RTOS 内で API 処理中に発生する．SWRTOS の結果は実験 2 とほぼ同じ数値であり，またばらつきもある．ARTESSO HWRTOS の場合，最小値と最大値では 4 サイクルの変動が生じた．最小値はタスク実行中の割り込み，最大値は RTOS 内での処理中割り込みが発生したことを示している．しかし RTOS 処理中で割り込みを受け付けられない期間が最大で 4 サイクルであるため，変動幅が 4 サイクルに収まっている．

実験 4 は常時スリープ状態のタスクが 4 つ存在する．これらのタスクは 3,600 秒のタイマでスリープしているため，実験中には起床しない．したがって割り込みは実験 1 と同様全くタスクが走っていない状態で発生する．SWRTOS では最小値最大値が実験 1 より大幅に大きくなっている．この理由は tick 処理内においてタイマを減算する処理が加算されるためと考えられる．一方 ARTESSO HWRTOS は tick 処理が行われないため，実験 1 と数値が全く同じになる．

「割り込み発生から Flag_Task 起動までの時間」に関しては,「割り込みから ISR までの時間」に比較し SWRTOS では最小値，変動幅とも大幅に増加しているのに対し，ARTESSO HWRTOS では最小値の増加も微増である．変動幅は縮小しているが，これは HWRTOS が割り込み発生時ソフトウェア ISR を起動する回路と IIA オフローディング実行回路は全く異なる回路であることに起因する．

以上から，SWRTOS では，生成するタスクの数や待ち状態の種類，現在の RTOS の内部状態，tick 処理等が割り込み応答性能や割り込み応答時間の変動に大きな影響を与えていることが数値的に理解できる．また SWRTOS においては正確な割り込み応答時間を設計時点で見積もってシステム設計を行うことが困難である．さらに同じプログラムを使用したとしても異なったコンパイラを使用したり，コンパイラのバージョンが変わることによって応答時間が変化することも予想できる．

一方，ARTESSO HWRTOS の割り込み応答性能はソフトウェアに比較し数十倍以上である．変動幅もほとんどが 0 であり，また 0 でないものも数サイクルであり，さらに設計時に予測できる値である．具体的な性能向上は ISR 起動までの応答時間では 18〜100 倍，次のタスク起動までは 38〜131 倍であった．ARTESSO HWRTOS の場合，データシートに「どのようなコンフィグレーションであれば割り込み応答性能は何サイクル」と定義をすることができ，設計者が設計時に割り込み応答性能を確定することができる．また，コンパイラの種類やバージョンに依存することもない．

4.5 むすび

高度な割り込み応答性能を必要とするアプリケーションを従来の SWRTOS 上に構築することは困難であることを実験により数値的に示した．一方 tick オフローディングや IIA オフローディングを実装した ARTESSO HWRTOS では超高速割り込み応答性能を実現し，従って高度な割り込み応答性能を必要とするアプリケーションを RTOS 上のタスクとして実装することが可能であることを示した．

以上本章においては，ARTESSO HWRTOS が，高度な割り込み応答性能を必要とするアプリケーションにおいても RTOS を利用できる環境を提供できることから，ソフトウェア信頼性・安全性の向上，ソフトウェアの部品化・再利用化によるソフトウェア開発生産性向上に大きく寄与可能であることを示した．

CHAPTER 5

シングルコア対応
疎結合ハードウェアRTOS搭載
産業ネットワーク用SoC

5.1　概要

工場内のネットワークはその使用目的に応じ階層化されている．Fig.5.1にFAネットワークの階層構造を示す．最上位階層は情報系ネットワークであり，製造スケジューリング，工程管理など工場内全体の管理に利用される．第二階層はPLC（Programmable Logic Controller）間やRC（Robot Controller）間を接続するコントロールレベル・ネットワークである．第三階層はPLCやRCとモーター・コントローラ，アクチュエータ・コントローラなどを接続するフィールドレベル・ネットワークである．なお本論文では上記モーター・コントローラなどネットワーク機能を持った機械制御用コントローラをFAコントローラと言う．

情報系ネットワークと異なりコントロール・レベルネットワーク，フィールドレベル・ネットワークはリアルタイム性能が要求される．ネットワークのリアルタイム性

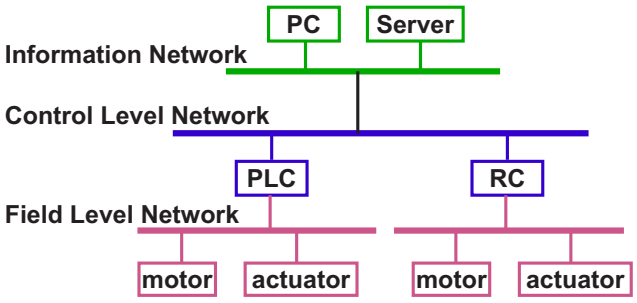


Figure 5.1: FA ネットワーク

とは言い換えると，リアルタイム転送すなわち周期的かつ確実なデータ転送であり，ネットワークで接続された複数の機器が同期して動作するために必須である．従来コントロールレベル・ネットワークやフィールドレベル・ネットワークではCAN 等の低速ネットワーク（数百 kbps）が使用されていたが，2000 年ごろからコントロールレベル・ネットワークが，近年ではフィールドレベル・ネットワークも高速な Ethernet タイプ（10/100/1000Mbps）に置き換わりつつある．Ethernet 上でネットワークのリアルタイム性を実現するため PROFINET (PROcess FIeld NET- work) ，EtherNet/IP (Ethernet industrial protocol) ，ModbusTCP，EtherCAT (Ethernet for control automation technology) など様々なプロトコルが考案され，FA コントローラに実装されている [2].

一方，FA コントローラのコアはモーター制御のためのリアルタイム処理をも実行する．しかし上記のようにネットワークの高速化に伴い，ネットワークプロトコル処理のために占有されるコアの CPU 時間が大幅に増大し，機械制御のためのリアルタイム処理に影響を与えている．本研究ではこの問題を解決するため，FA コントローラの制御に HWRTOS の採用した．第 1 章 Fig.1.1 に示したように，HWRTOS とコアの接続方法は 2 種類あり，一つは専用インターフェースで接続される密結合型（TC-HWRTOS)，もう一つはシステムバスにより接続される疎結合型（LC-HWRTOS) である．FA コントローラにおいては ARM コア搭載が顧客ニーズとして必須であり，汎用コアとの接続が容易である LC-HWRTOS を選択した．

本章における主要な貢献は以下の通りである．

1. FA コントローラのための LC-HWRTOS アーキテクチャを提案する．

2. コアと HWRTOS の並行動作による性能向上を提案する．

3. 本アーキテクチャを使用した FA コントローラのための SoC を設計・作製する．

4. SoC による LC-HWRTOS と SWRTOS における，RTOS 性能およびネットワーク性能比較を行う．

評価した結果，従来の SWRTOS に比較し，LC-HWRTOS を実装した本 SoC においては，API 実行性能は 1.4〜2.9 倍，UDP/IP 性能は 1.67 倍であった．

5.2 FA コントローラの要件

本節では FA コントローラが必要とする条件について述べる．FA コントローラはモーター制御などを行うためリアルタイム処理を必要とするが，FA コントローラに高速ネットワークインターフェースを実装すると，同じコア上で動作する機械制御用のリアルタイム処理に影響を及ぼす．これはネットワークの高速化によりネットワークプロトコル処理が増大しコアに大きな負荷をかけるからである．したがって，FA コントローラではネットワークプロトコル処理の負荷軽減が要求される．

次に，先に述べたようにフィールドレベル・ネットワークでは複数のプロトコル規格が採用されており，またリアルタイム性能が要求されている．このため FA コントローラではマルチプロトコル対応と，リアルタイム処理性能の向上が要求される．

また FA コントローラでは，ARM コアの使用が要求される．これは ARM コアが事実上業界標準になっており，既に ARM コア上で動作するソフトウェア資産をユーザが有していること，従来から使用している ARM 用開発環境をそのまま使用できること，ARM コアファミリーはスケイラビリティーがあり機器性能に応じたコアの選択が可能であること，市場実績が多くコアとしての信頼性が非常に高いことが理由である．

さらに FA コントローラにおいても，低消費電力，低コストが要求される．これは

第 2 章で述べたよう，大量の FA コントローラがフィールドレベル・ネットワークに接続されるため，個々の低消費電力化・低コスト化が，システム全体の消費電力・コストに大きく影響を及ぼすからである．

以上より，

(1) ネットワークプロトコル処理の負荷軽減

(2) リアルタイム処理性能の向上

(3) マルチプロトコルの対応

(4) ARM コアの使用

(5) 低消費電力，低コスト

の 5 項目を FA コントローラの要件として挙げることができる．

要件 (1) を満たす方法としては TCP/IP オフロードエンジンを利用し，プロトコルの負荷をコアから切り離す方法がある．しかし同時に要件 (3) に対応するためにはプロトコル毎に複数のエンジンを設計・実装する必要があり，コスト増加，開発期間の大幅な増加につながる．もう一つの方法は動作クロック周波数を上げ，コアの性能を向上させる方法である．この方法では要件 (1) と (3) の双方を満足できるが，消費電力，コストとも増加するため要件 (5) を満足しない．

上記システム要件を満足させるため，FA コントローラに実装する RTOS に HWRTOS を採用した．プロトコル処理は頻繁に RTOS 機能を呼び出すため，全体の処理時間に対する RTOS 処理の時間の割合が高い．したがって RTOS をハードウェア化し RTOS の実行速度が高速になればプロトコル処理の負荷は軽減され，要件 (1) が解決される．また，RTOS のハードウェア化により割り込み応答性能が向上し，また RTOS 実行時間が減少するためインタラプト禁止状態の期間も減少することから要件 (2) を満たすことができる．次に HWRTOS はプロトコルの種類によらず各プロトコルが共通に利用でき，どのプロトコルも性能向上することから要件 (3) が解決される．さらに HWRTOS の利用によりコアの性能を上げる必要がなくなるため (5) も解決できる．(4) の解決方法については 5.4 で詳細に説明する．

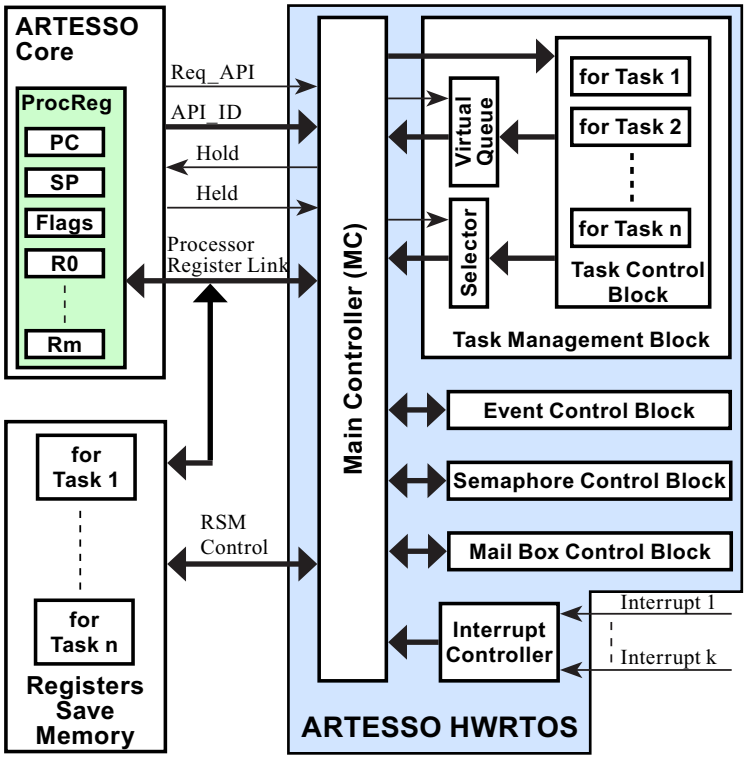


Figure 5.2: ARTESSO HWRTOS とコア

5.3 TC-HWRTOS

先に述べたようにFA コントローラではLC-HWRTOS を採用する．本節ではLC-HWRTOS とTC-HWRTOS の違いを明確にするため，TC-HWRTOS について再度簡単に説明する．

第3章で説明したARTESSOは，オリジナルコアであるARTESSO Core とARTESSO HWRTOS を密結合で接合したTC-HWRTOS である．第3章に第4章の機能を追加したARTESSO システムを本章ではオリジナルARTESSO システムと呼び，このARTESSO HWRTOS をオリジナルARTESSO HWRTOS と呼ぶ．以下TC-HWRTOS であるオリジナルARTESSO について Fig.5.2（Fig.3.3 と同一）の構成図を使用して説明する．

TC-HWRTOS では，コアとARTESSO HWRTOS は専用インターフェースで接続される．このインターフェースにはAPI を発行するための信号線や，HWRTOS がコ

ンテキストスイッチを実行するための制御信号線が含まれる．

ARTESSO Core は ARTESSO HWRTOS と接続するための専用インターフェースを有する独自設計の 32bit RISC である．MC は各 API 処理等の RTOS 処理を実行する回路であり，ハードウェア・ステートマシンで構成される．Task Control Block は各タスクごとの情報，例えば「現在のタスクの状態」,「優先度」,「待ち要因」などの情報を維持管理しており，情報は Selector で選択され MC に提供される．Virtual Queue モジュールは RTOS で使用する全てのキューを仮想キューで実現している．Event Control Block，Semaphore Control Block，Mail Box Control Blok はそれぞれイベント機能，セマフォ機能，メールボックス機能を実現するための情報を維持管理している．

コンテキストスイッチは，コアの ProcReg の内容を Registers Save Mem. との間で交換することにより実行される．Registers Save Mem. はタスクまたは ISR 毎に ProcReg の内容を保管する．ARTESSO HWRTOS は，ProcReg の内容を Registers Save Mem. の対応するタスク（または ISR）のエリアに保管し，Registers Save Mem. から次に実行すべきタスク（または ISR）のレジスタデータを ProcReg にロードすることにより，コンテキストスイッチを実現する．レジスタの内容の交換は Processor Register Link を使用して 1 サイクルで実行される．また第 4 章で示した，IIA オフローディング機能や tick オフローディング機能を追加することにより割り込み応答性能を向上させ，また割り込み応答時間の変動を大幅に減少させている．

市販の SWRTOS と ARTESSO HWRTOS との API の実行時間の比較は以下のとおりである，”起床待ち” API は SWRTOS の 628 サイクルに対し ARTESSO HWRTOS が 10 サイクル，”セマフォ資源返却” API で，344 サイクルに対し 7 サイクルであった（Table3.2）．また 4.4.3 に示すよう割り込み応答性能は，ISR 起動までの応答時間では 18〜100 倍，次のタスク起動までは 38〜131 倍であった．

以上のように，オリジナル ARTESSO システムでは，ARTESSO HWRTOS，ARTESSO Core および Register Save Memory により TC-HWRTOS を実現した．この構成により SWRTOS に比較し，上記のように極めて高速な API 処理，割り込み応答，コンテキ

Table 5.1: RTOS タイプとコアの組合せ

Combination Type	(A) TC -HWRTOS + Purpose-built	(B) TC -HWRTOS + modified ARM	(C) LC -HWRTOS + ARM	(D) SWRTOS + ARM
API execution time	Low		Middle	High
Tick management offloading	Available			Unavailable
IIA offloading	Available			Unavailable
Core Reliability	Middle		High	
Core Scalability	No	Low	Yes	
Core verification cost	High	Middle	Low	
Standard Tools	Unusable	Usable		
LSI development cost	Middle	High	Middle	Low

ストスイッチを実現した.

5.4 FA コントローラ対応 HWRTOS アーキテクチャ

この章では要件 (4)，すなわち「ARM コアの使用」を満たすために LC-HWRTOS を採用した理由を示す.

Table5.1 に RTOS のタイプとコアの組合せを示す.

(A) HWRTOS + 専用コア　(TC-HWRTOS)

(B) HWRTOS + カスタマイズド ARM コア　(TC-HWRTOS)

(C) HWRTOS + ARM コア　(LC-HWRTOS)

(D) SWRTOS + ARM コア　(SWRTOS)

組合せタイプ (A)，(B) は TC-HWRTOS であり，Fig.5.3（Fig.1.1 と同一）に示すよう TC-HWRTOS は HWRTOS とコアの間に専用インターフェースが必要である．(A) はオリジナル ARTESSO システムである．(B) は ARM コアを使用し TC-HWRTOS を実現する方法で，ARTESSO HWRTOS との専用インターフェースを実現するため，ARM コアの改造を行う．(C) は ARM コアを使用した LC-HWRTOS である.

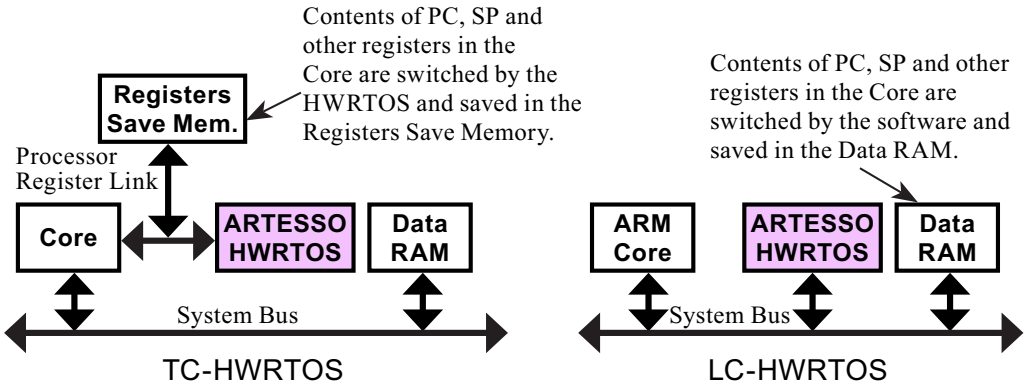
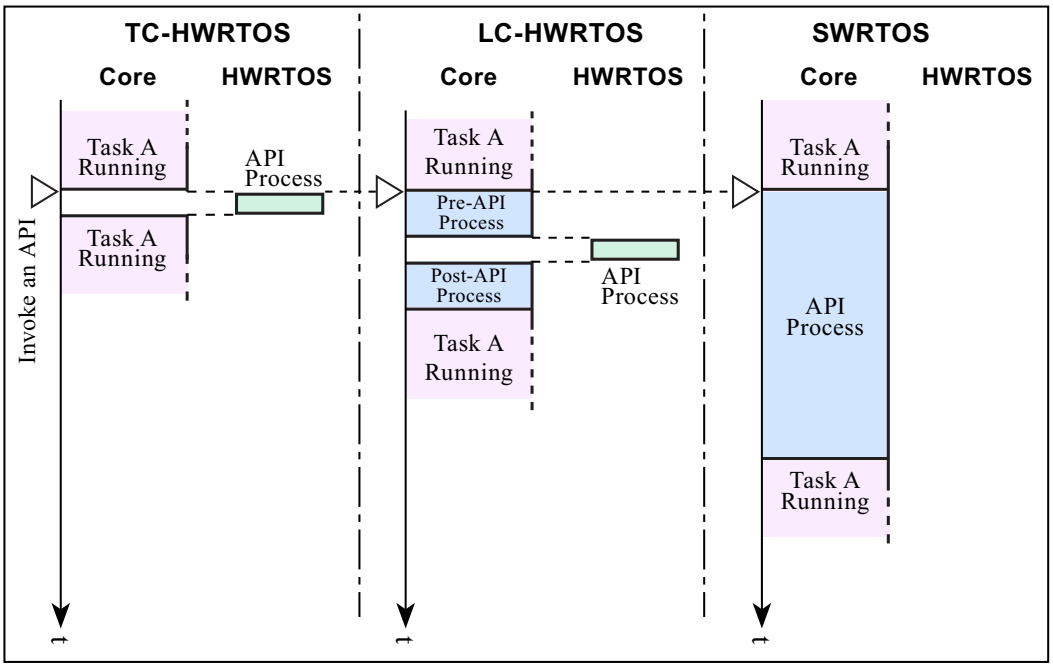


Figure 5.3: HWRTOS とコアの接続方法

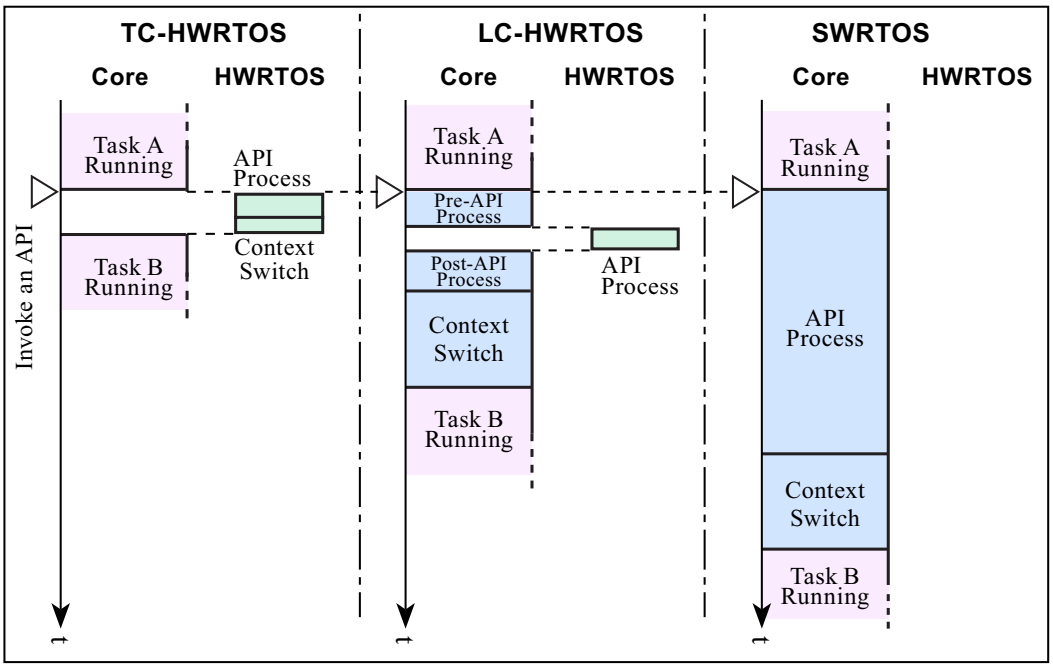
Fig.5.3 で示したように LC-HWRTOS ではシステムバス上に HWRTOS を実装し，コアはシステムバスを介して API を発行する．このため ARM コアの改造は必要無い．LC-HWRTOS は API をハードウェアで実行するため高速であるが，コンテキストスイッチの処理は SWRTOS と同様にソフトウェア処理で実行する必要がある．(D) は従来の SWRTOS による実現方法である．

Fig.5.4(a) に TC-HWRTOS，LC-HWRTOS および SWRTOS の各 API 処理シーケンス（コンテキストスイッチなしの場合）を示す．TC-HWRTOS では，API を発行すると短時間で HWRTOS が API 処理を実行し，終了すると即座にタスク処理が再開される．LC-HWRTOS ではシステムバスを介して API を発行する手順や，また API の結果を読み出す手順等，ソフトウェアによる前処理と後処理が必要である．これは 5.5 に示すよう LC-HWRTOS では引数と戻り値は ARTESSO HWRTOS とシステムバスの間にあるレジスタを介してやりとりするためである．一方 TC-HWRTOS では 5.3 に示したように ProcReg を介して行うため，前処理，後処理の必要がない．

Fig.5.4(b) はコンテキストスイッチがある場合の API 処理シーケンス図である．TC-HWRTOS ではハードウェアによりコンテキストスイッチが行われるため極めて短い時間で実行されるが，LC-HWRTOS では前処理，後処理以外に，コンテキストスイッチ処理をソフトウェアで実行する必要がある．



(a) API without context switch



(b) API with context switch

Figure 5.4: API 実行シーケンス

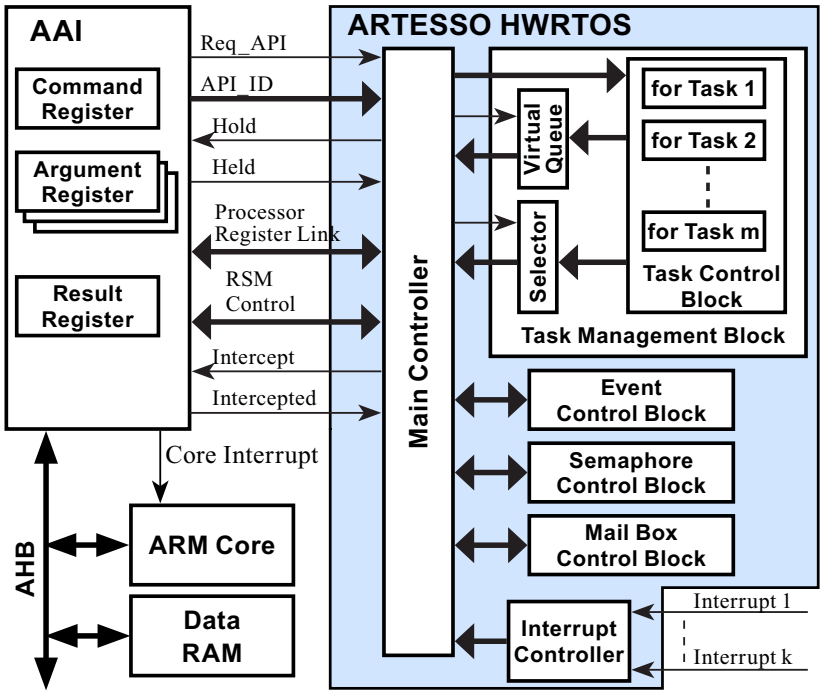


Figure 5.5: LC-HWRTOS 構成図

要件 (4) より ARM コアを使用する必要があるため，Table5.1 の (B) または (C) を選択する必要がある．(C) は (B) と比較すると API 発行にかかる時間が増加する．しかし (B) は ARM コアを改造する必要があり，開発費が高価になる．一方 (C) は ARM をそのまま使用できるためコスト的に (B) よりメリットがあり，(D) と比較すると (B) と同様 API 処理をハードウェアで実現しているため API 実行時間が極めて高速であるだけでなく，第 4 章で示した IIA オフローディングや tick オフローディングが実装されているためリアルタイム性が向上し，また割り込み回数も減少するなど大幅な性能向上が予想できる．以上の検討より本 FA コントローラの開発では (C) の LC-HWRTOS を選択した．

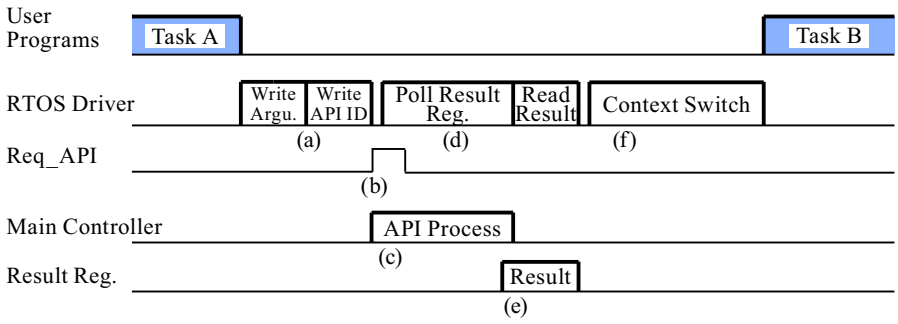


Figure 5.6: LC-HWRTOS における API 実行シーケンス

5.5 LC-HWRTOS

Fig.5.5 は，FA コントローラ向けの LC-HWRTOS の構成である．オリジナル ARTESSO システムは TC-HWRTOS であるため，これを改造し LC-HWRTOS を実現した．AAI (ARTESSO AHB Interface) モジュールを新たに設計，実装した．AAI は”Command Register”，”Argument Register”，”Result Register”を実装する．ARM コアは AAI 内のこれらのレジスタにアクセスすることにより，ARTESSO HWRTOS に対し API を発行する．MC は，AAI への接続および，5.5.2 で示すコアと HWRTOS の並列動作を実現するための改造を行った．

5.5.1 API 発行手順

API 発行時のシーケンスを Fig.5.6 に示す．ARM コア上で動作するソフトウェアは”User Programs”と”RTOS Driver”である．User Program が API を発行すると RTOS Driver がコールされる．RTOS Driver は引数となる値を Argument レジスタに書き込み，そのあと Command レジスタに発行すべき API 識別子を書き込む (a)．AAI はこれをトリガとし，Req_API 信号を 1 にして，API 処理要求を MC に伝える (b)．このとき API 識別子は API_ID により通知される．MC は API 処理を開始し (c)，API 処理が完了すると，戻り値が結果レジスタに書き込まれる (e)．ARM コアは API 発行後結果

レジスタをポーリングし (d)，結果が書き込まれると戻り値を読み出す．

API 処理の結果コンテキストスイッチが必要な場合，結果レジスタにより戻り値と共に，コンテキストスイッチが必要である旨およびコンテキストスイッチ後のタスク ID が伝えられる．このとき ARM コアはソフトウェアによりコンテキスト切換を行う (f)．すなわち SWRTOS と同じ手順で内部レジスタを Data RAM に待避，新しいタスクまたは ISR 用のレジスタセットを Data RAM から ARM コア内のレジスタにロードする．

5.5.2 コアと ARTESSO HWRTOS の並列動作

本節ではコアと HWRTOS の並列動作について述べる．SWRTOS においては，SWRTOS とユーザ・プログラムは両者共にコアで実行されており，またユーザ・プログラムは API を発行したあと API の戻り値を待つため，SWRTOS が動作している期間ユーザ・プログラムは停止する．オリジナル ARTESSO システムではこの考え方を踏襲している．すなわちオリジナル ARTESSO HWRTOS が動作しているときは ARTESSO Core は停止する．これを SOHC (Serial Operation of a HWRTOS and a core) という．先に述べたように，オリジナル ARTESSO HWRTOS は API 処理のみでなく IIA オフローディング機能や tick オフローディング機能が実装されている．上記のように HWRTOS が動作しているときコアは停止すると言うルールを踏襲しているため，IIA オフローディング機能が動作している期間，tick オフローディングによりタイムアウト値が 0 になりキュー操作を行う期間，コアは停止する．しかし詳細に検討するとこれらの処理は処理要求元がユーザ・プログラムでないためユーザ・プログラムと並行に処理が可能である．以上の結果より MC を改造することにより IIA オフローディング機能と tick オフローディング機能がユーザ・プログラムと並行に動作させるようにした．この方式を POHC (Parallel Operation of a HWRTOS and a core) という．

Fig.5.7 は SOHC を LC-HWRTOS に適用したと仮定した場合の IIA オフローディング実行時の動作を示している．Interrupt k が発生すると (t1)，User Program を停止さ

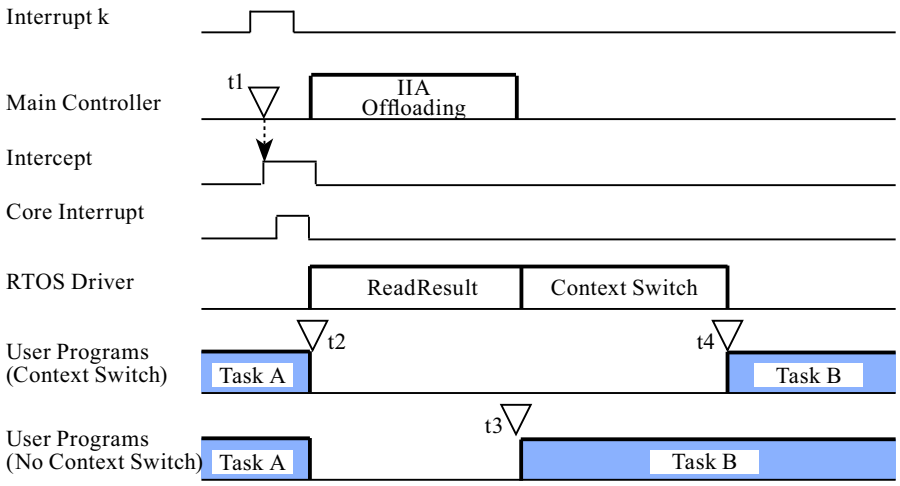
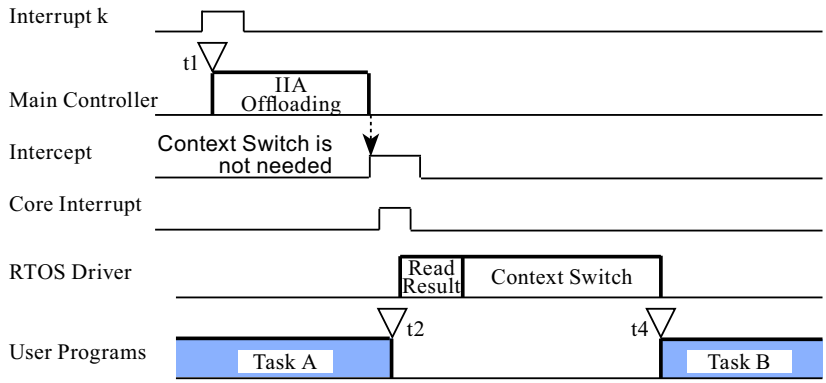


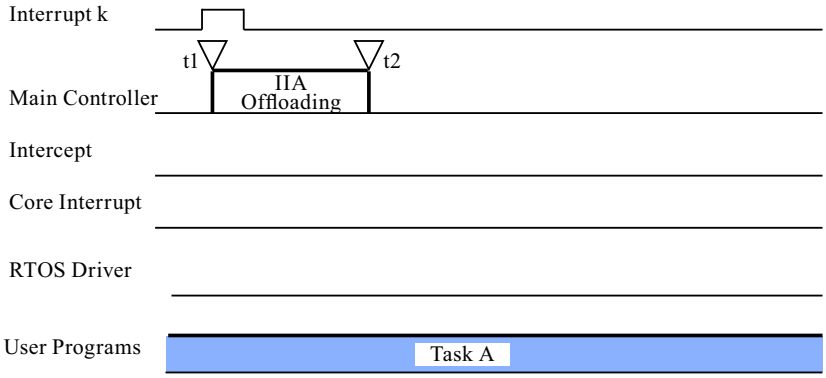
Figure 5.7: SOHC での IIA 実行シーケンス

せ，停止したのち IIA Offloading を MC が実行する (t2)．IIA オフローディングが完了すると結果が Result レジスタに書き込まれるため，これを RTOS Driver が読み，コンテキストスイッチの指示がなければもとのタスクが実行再開し (t3)，コンテキストスイッチの指示があればこれを行い，新しいタスクが動作開始になる (t4)．

Fig.5.8 は POHC を適用した場合の IIA オフローディングのユーザ・プログラムとの並行動作を示している．同図 (a) はコンテキストスイッチが発生する場合である．Interrupt k が発生すると (t1)，IIA オフローディング機能が MC で実行される．この間ユーザ・プログラムは IIA オフローディングと並列に動作している．この処理が終了し (t2)，コンテキストスイッチが必要な場合，MC は ARM コアに割り込みによりコンテキストスイッチの要求を知らせる．ARM コアでは Task A が停止し，RTOS Driver が動作し，AAI の Result Register を参照しコンテキストスイッチ後のタスク ID（Task B）を得る．RTOS Driver はレジスタを入れ替え，Task B を起動する (t3)．(b) はコンテキストスイッチが発生しない場合である．t2 において MC がコンテキストスイッチの必要がないと判断したが場合はそのまま処理が完了する．従って Task A は処理を中断されることなく，動作し続ける．以上のように POHC を採用することにより MC とコアの並列動作を実現できるだけでなく，コンテキストスイッチが必要無い場合は



(a) POHC with context switch



(b) POHC without context switch

Figure 5.8: POHC での IIA 実行シーケンス

User Program が継続して動作することができるようになるため，コアの利用効率が向上する．

以上コアと HWRTOS の並行処理の改造について IIA オフローディングで説明したが，タイムアウト処理も同様に並列処理が可能である．すなわちタイムアウト処理は User Program と並列動作可能であり，その結果コンテキストスイッチの必要が無ければ User Program はタスク処理を継続できる．このようにコアと ARTESSO HWRTOS の並列動作を可能にしたことにより，コアへの割り込みが大幅に削減し，さらにコアのアベイラビリティが向上する．

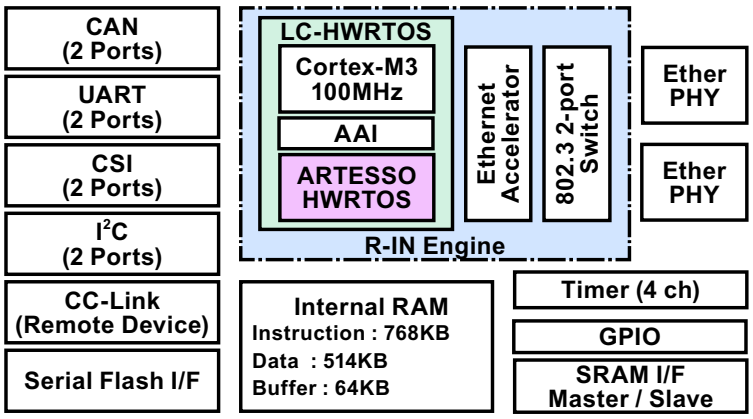


Figure 5.9: R-IN32M3 SoC 構成図

5.6 R-IN32M3 の構成

前節までに，FA コントローラ向け SoC に必要な要件を検討し，これを満足させるために LC-HWRTOS を採用し，そのアーキテクチャを説明した．本節ではこのアーキテクチャに基づき設計・開発した FA コントローラ向け SoC，R-IN32M3 について説明する．R-IN32M3 は既にルネサスエレクトロニクス（株）で量産しており，また R-IN32M3 用 RTOS Driver やプロトコルスタック等のライブラリは同社 web サイトより入手可能である [23].

Fig.5.9 に R-IN32M3 の全体構成図を示す．外部インターフェースとして，CAN, UART，CSI，I2C インターフェースをそれぞれ 2 ポートずつ実装した．また CC-Link の Remote Device を実装した．さらに Ethernet PHY を 2 ポート実装し産業用 Ethernet として利用できる仕様とした．

内部メモリとして，Instruction Memory 768KB，Data Memory 512KB，Buffer Memory 64KB を実装した．外部 SRAM も実装可能だが，内部メモリのみでシステムを構築することができれば，コスト削減，および消費電力の削減となる．

R-IN Engine は産業用ネットワーク機能を実現するための処理モジュールであり，単なるマイクロプロセッサでなく，高付加価値かつ高速処理を提供する．以下に R-IN

Engine の説明をする．

1) LC-HWRTOS : ARTESSO HWRTOS，AAI，コアで構成される．コアは ARM の Cortex-M3 100MHz を実装した．詳細は 5.5 で示したとおりである．

2) 802.3 2-port Switch : このモジュールは産業用 Ethernet における 2 つの PHY を使用したデイジーチェイン接続を実現する．ハードウェア構造は使用するプロトコルによって異なり，EtherCAT / slave または CC-Link IE / Field の選択が可能である．

3) Ethernet Accelerator : Ethernet Accelerator はネットワーク処理に関する以下の 3 つの処理をハードウェアで実現している．第一は TCP および IP のチェックサム機能である．第二は TCP/IP および Ethernet MAC ヘッダ並べ替え機能であり，これは圧縮して配置されたフレーム・ヘッダ上の情報をソフトウェアがアクセスしやすいように並べ替える機能である．第三は，ハードウェアによるバッファ管理機能であり，パケットを蓄えるためのバッファ領域のユーザへの割り当て・解放機能である．第一および第二の機能は特に TCP/IP で有利な機能である．産業用ネットワークプロトコルでは，独自制御用プロトコルに並行して TCP/IP もプロトコルスタックとして定義されている．

5.7 性能評価

5.7.1 評価項目

RTOS の性能評価および，RTOS の種類によるネットワーク性能の影響について評価を行う．下記，1)〜5) は RTOS の性能評価，6) はネットワーク性能評価である．

1) API 実行時間 : ”起動タスク”，”起床タスク”，”待ち状態強制解除”の各 API 処理の実行時間を測定し，SWRTOS に比較し LC-HWRTOS で性能が改善されていることを確認する．

2) 割り込み応答 : IIA オフローディング機能によりインタラプト応答性能が向上

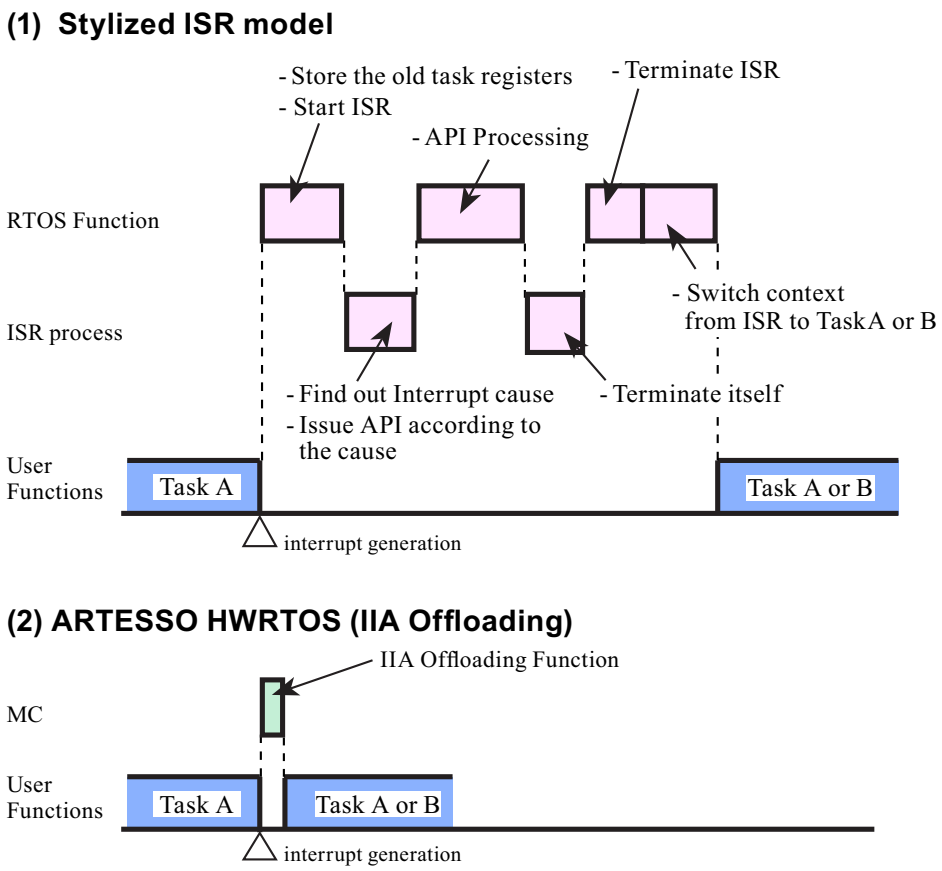


Figure 5.10: IIA オフローディング

したことを確認する．SWRTOS では Fig.5.10 (1)（Fig.4.2 (1) と同一）において割り込み発生から Task B が起動されるまでの時間，TC-HWRTOS では Fig.5.10 (2) において割り込み発生から Task B が起動されるまでの時間，LC-HWRTOS では Fig.5.8 (a) の t1 から t4 の期間を測定する．

3) IIA オーバヘッド時間 : コンテキストスイッチが発生しない場合の POHC の効果について測定する．SWRTOS では Fig.5.10 (1) において割り込み発生から Task A が起動されるまでの時間，TC-HWRTOS では Fig.5.10 (2) において割り込み発生から Task A が起動されるまでの時間，LC-HWRTOS では Fig.5.8 (b) で示されるよう，Task A が処理を続けていること，すなわちオーバヘッド時間がゼロであることを確認する．

4) tick オフローディングの効果 : tick オフローディング機能により，割り込み応答

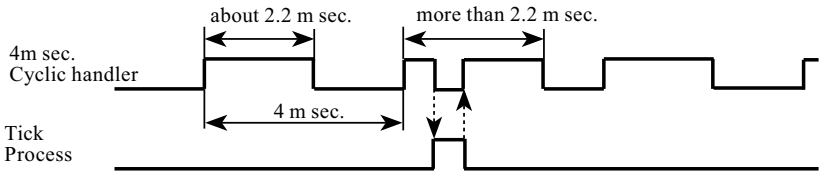


Figure 5.11: tick プロセスの影響

の高速化と応答時間変動幅の縮小を確認する．3 つの周期タスクを起動する．それぞれの周期は 4 m 秒，10 m 秒，15 m 秒とした．各周期タスクは単純な for ループ 20,000 回繰り返すものであり，実行時間は約 2.2 m 秒である．SWRTOS では 1m 秒毎にタイムティック割り込みが発生し，タイムティック処理が実行される．LC-HWRTOS は 4.2 で示したようにタイマ処理をハードウェアで行っており，ソフトウェアでタイムティック処理を実行する必要はない．以上の環境で，4 m 秒の周期タスクの実行時間を測定する．タイムティック割り込みの影響がなければ実行時間は常に一定の約 2.2 m 秒になる（Fig.5.11）．この測定を 10,000 回行った．

5) 起床実行時間 : 仮想キューによるリアルタイム性の向上について評価をおこなう．タイムアウト待ちタスクが複数存在する場合，その一つを起床させたとき，従来の SWRTOS では起床時間と待ちキューの長さに相関があるが，HWRTOS ではオペレーション時間はキューの長さに依存せず一定である．n 個のタスクを起床待ち状態にさせ，そののち一つのタスクを起床した時点からこのタスクがディスパッチされるまでの時間を測定した．この測定は各 n について 100 万回繰り返し，処理時間の最低値と最大値を測定した．

6) UDP/IP スループット : UDP/IP プロトコルを実装し，LC-HWRTOS を使用したほうが SWRTOS より性能が向上していることを確認する．

5.7.2 測定方法

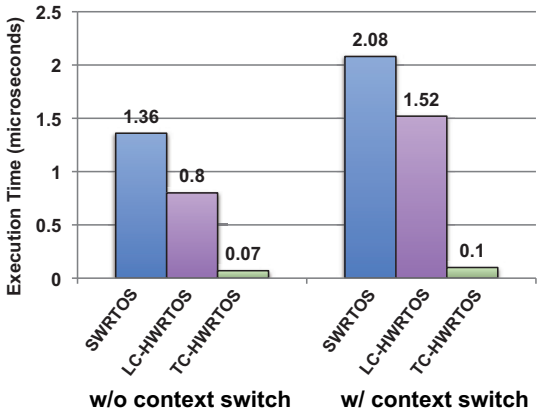
RTOS 性能評価は，TC-HWRTOS，LC-HWRTOS，SWRTOS についてそれぞれ測定を行った．LC-HWRTOS は R-IN32M3 評価ボード（動作クロックはコア，HWRTOS とも 100MHz）を使用し，評価用ソフトウェアを実装して測定を行った．SWRTOS については，同ボードの HWRTOS 機能を停止させ，オープンソース SWRTOS の TOPPERS を実装し，評価用ソフトウェアを実装して測定を行った．同一の環境を使用することにより，測定条件をできる限り同じにした．TC-HWRTOS は ARTESSO HWROTS を Verilog シミュレータ上で動作させ，処理時間を計測した．

ネットワークプロトコル処理性能評価は LC-HWRTOS および SWRTOS のみ評価を実施し，上記 RTOS 評価と同じ環境で UDP/IP プロトコルスタックを実装し，UDP ペイロードが 18，318，558，918，1,158 バイトのフレームについてそれぞれスループットおよび転送フレームレートを測定した．

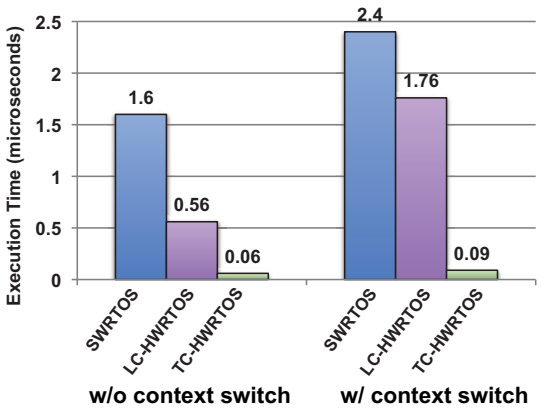
5.7.3 測定結果

1) API 実行時間 : Fig.5.12a)〜c) にそれぞれ”起動タスク”，”起床タスク”，”待ち状態強制解除”の各 API 実行時間の測定結果を示した．コンテキストスイッチがない場合 API 実行時間は，SWRTOS は LC-HWRTOS の 1.4 倍，1.5 倍，1.4 倍であった．一方コンテキストスイッチがある場合はそれぞれ，1.7 倍，2.7 倍，2.9 倍であった．コンテキストスイッチがある方が開きが少ないのは，コンテキストスイッチをソフトウェアで実行しているためソフトウェア処理が全体に占める割合が大きくなるためである．一方 TC-HWRTOS は LC-HWRTOS より実行時間が小さいが，これは TC-HWRTOS は API 発行時の前処理・後処理およびコンテキストスイッチのソフトウェア処理を必要としないからである．

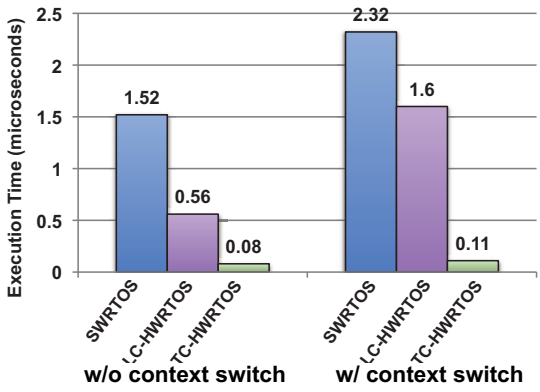
2) 割り込み応答 : Fig.5.13 の a) に結果を示す．SWRTOS の実行時間は LC-HWRTOS に比較し 2.3 倍であった．



a) 起動タスク API 実行時間



b) 起床タスク API 実行時間

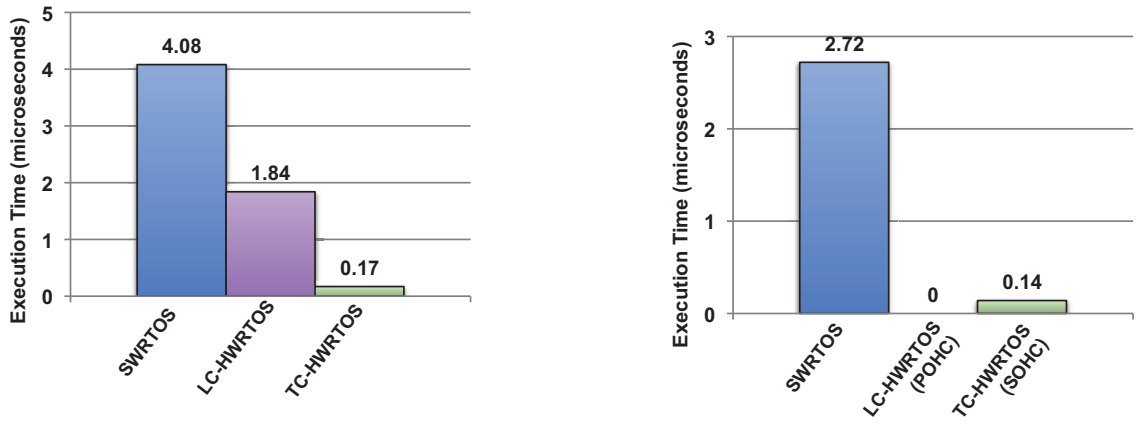


c) 待ち状態強制解除 API 実行時間

Figure 5.12: 評価結果 (1)

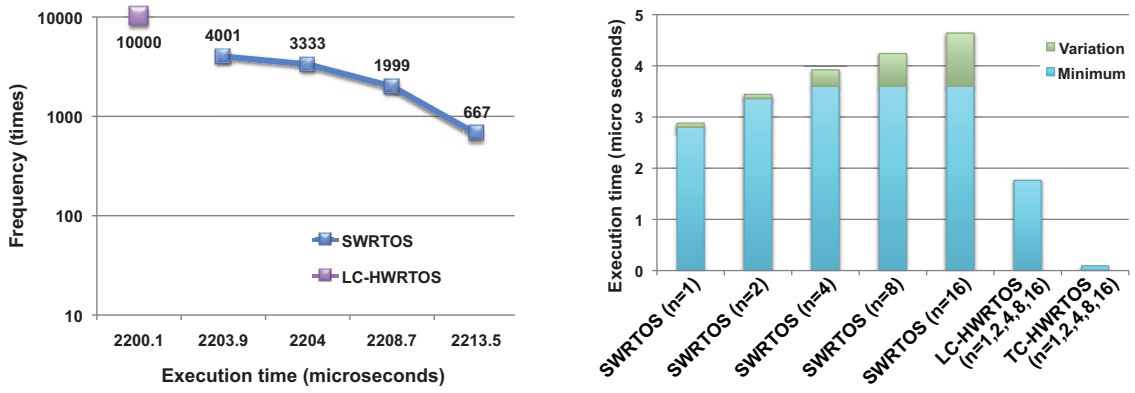
3) IIA オーバヘッド時間 : Fig.5.13 の b) に結果を示す．LC-HWRTOS では POHC を採用したため，オーバヘッド時間はゼロになることが確認できた．

4) tick オフローディングの効果 : Fig.5.14 の a) に結果を示す．横軸は周期が 4 m 秒の周期タスクの実行時間である．10,000 回の試行の結果，SWRTOS ではタイムティックの割り込みにより周期タスクの実行時間は 2.2039〜2.2135m 秒の間で変動し，したがって変動幅は最大 9.6μ 秒であった．一方 LC-HWRTOS は内部の tick 処理に依存することなく，各タスクは全て 2.2001m 秒で完了した．この結果 SWRTOS では tick 処



a) **Interrupt response**　　b) **Influence of interrupts**

Figure 5.13: 評価結果 (2)



a) tick の影響　　b) 起床タスク API 実行時間

Figure 5.14: 評価結果 (3)

理により割り込み処理が最大で 9.6μ 秒も遅延し，また遅延時間もばらつくのに対し，LC-HWRTOS ではタイムアウト処理による割り込み処理の遅延は全く発生しないことが確認できた．LC-HWRTOS は正確な時間での割り込み処理を実現でき，リアルタイム性に極めて優れていることが立証された．

5) 起床実行時間 : Fig.5.14 の b) に結果を示す．SWRTOS では n が大きくなるほど最大値が増え，またバラツキも大きくなる．n=16 のときのバラツキ時間幅は 1.04μ 秒

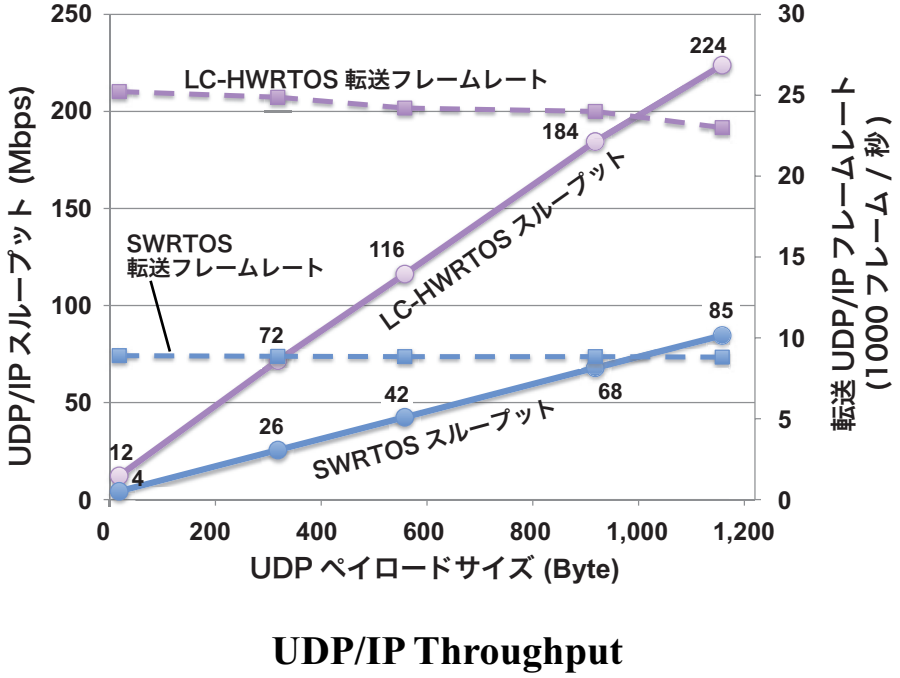


Figure 5.15: 評価結果 (4)

であった．一方 LC-HWRTOS は常に処理時間が一定であり，n= 16 のとき処理時間は SWRTOS の 1/2.6 であった．この実験から SWRTOS は待ちキューの長さによりタスク起床処理時間が変動するが，LC-HWRTOS は TC-HWRTOS と同様に処理時間が高速なだけでなく，処理時間が待ちキューの長さに依存しないことが確認できた．すなわち LC-HWRTOS は起動したい時間に正確にタスクを起動できる，つまりリアルタイム性に極めて優れているということができる．

6) UDP/IP スループット : Fig.5.15 のに結果を示す．双方の相違は単に RTOS の種類のみであり，この結果から SWRTOS から LC-HWRTOS に変更するだけで平均 2.74 倍の性能を得ることができた．これは必要とするネットワーク性能を得るために，従来の 36%の CPU 負荷で達成できるということである．よって残りの CPU 時間を, 他の処理に割り当てることが可能となる．以上より，LC-HWRTOS が産業ネットワークシステムの応用において有効であることを確認することができた．

5.8 むすび

FA コントローラ向け R-IN32M3 SoC を開発し，性能を評価した．FA コントローラ向け SoC では (1) ネットワークプロトコル処理の負荷軽減，(2) リアルタイム処理性能の向上，(3) マルチプロトコルの対応，(4)ARM コアの使用，(5) 低消費電力・低コストの 5 項目が要件であるが，これらを満たすため，Cortex-M3 コアを使用し，RTOS として LC-HWRTOS を使用した．LC-HWRTOS は単に SWRTOS に比較し高速なだけでなく，POHC 方式を取り入れることにより割り込み頻度を低減させコアのアベイラビリティを向上させた．評価結果としては，API 実行時間が大幅に短縮されただけでなく，割り込み性能も向上したことが確認でき，POHC の効果も確認できた．またネットワーク性能においては単に SWRTOS から LC-HWRTOS に変更するだけで UDP/IP 性能を 2.74 倍に向上させることができ，またこれは必要とするネットワーク性能を得るために，従来の 36%の CPU 負荷で達成できることを示しており，当初の目標を達成することができた．

CHAPTER 6

マルチコア対応RTOSの ハードウェア化による性能向上

6.1 概要

近年組込みシステムにおいてマルチコアの利用が進んでおり，マルチコア対応のRTOSが必要とされている．マルチコア対応RTOS上にソフトウェアを構築することによりシングルコアと同様のメリットを享受できるからである．一方2.4.2に示したように，API処理時間や割り込み応答時間はSoftSingle-RTOSに比較しSoftMulti-RTOSでは増大し，さらにSoftMulti-RTOSではAPI実行時間や割り込み応答時間の最悪値を定めることができないというマルチコア特有の問題が存在する．

本章では上記課題を解決するため，Single-ARTESSOを拡張してマルチコアに対応したMulti-ARTESSOを提案する（Single-ARTESSOは第3章，第4章の機能を含むシングルコア用のARTESSOであり，第5章の機能を含まないものとする）．開発したMulti-ARTESSOは，日本で広く使われているITRON仕様と同等の仕様を実現するとともに，仮想キューなどSingle-ARTESSOの技術を活かすことにより，様々な

組込みシステムに適用できる機能を提供した．さらに前記 SoftMulti-RTOS の問題を解決することにより，API 実行時間や割り込み応答時間が高速であるばかりでなくこれらの最悪値を定めることを可能とした．

6.2 マルチコア対応ハードウェア RTOS

本研究では第 3 章で開発した Single-ARTESSO をベースに HardMulti-RTOS を開発した．Multi-ARTESSO は 2.4.2 で示した SoftMulti-RTOS が有する問題を解決した．以下，実用化に耐えうる Multi-ARTESSO のシステム要件を 6.2.1 で示し，6.2.2 でこのシステム要件を満たすシステム構成を示す．なお Multi-ARTESSO のコア数は 2 個から 16 個程度を対象とする．

6.2.1 システム要件

Multi-ARTESSO の開発にあたり，以下をシステム要件とした．

(1) 産業界で使用されている RTOS と同等の機能を提供すること．

(2) RTOS 内で使用する各種 Wait キューを効率的に実現すること．

(3) シングルコアでも使用でき，その場合 Single-ARTESSO と比較して性能低下が少ないこと．

(4) API 実行時間および割り込み応答時間の最悪値が定まること．また実行時間の変動幅が少ないこと．

(5) 可能な限りコアと HWRTOS との実行並列性を実現すること．

(1) は様々な組込みシステムで使用するために重要である．(2) は，2.4.1 で述べたように従来技術を使用した HardSingle-RTOS では十分な数のオブジェクトに対応できず，従って従来技術を使用して HardMulti-RTOS を実現したとしても同様に十分な数のオブジェクトに対応できないと予想できることから，Multi-ARTESSO では仮想キューによりこの課題を解消するとともに少ない回路量で実現することを目標とした．(3) は Single-ARTESSO と Multi-ARTESSO でハードウェアを共通化することにより

ソースコードの一元化を目指す．共通化によるデメリットを最少化するため，Single-ARTESSO と Multi-ARTESSO をシングルコアで使用した場合の性能が著しく劣化しないことを目標とした．(4) に関しては，2.4.2.3 で述べたように，SoftMulti-RTOS ではスタベーションおよびコア間・コア内排他制御の問題から，API 実行時間および割り込み応答時間の最悪値の両方を定めることができておらず，Multi-ARTESSO では最悪値を定義できることを目標とした．またただ単に最悪値が定められるだけでなく，利用する上で変動幅すなわち最悪値と最小値の幅が少ないことが重要であり，あわせて変動幅が少ないことを目標とした．(5) に関しては以下の通りである．Single-ARTESSO ではコアと HWRTOS は並列動作しなかったが，RTOS 処理がハードウェアで実行され高速であり問題にならなかった．しかしマルチコアの場合，コアと HWRTOS に並列実行性がないと HWRTOS が動作している期間全てのコアが停止することになり性能の劣化が予想される．このため本研究ではコアと HWRTOS が可能な限り並列化して動作することを目指す．

6.2.2 システム構成

6.2.2.1 ハードウェア基本構造の決定

2.4.2.3 において管理ブロックの実現方法としてジャイアントロックと細粒度ロックがあることを説明した．Multi-ARTESSO において管理ブロックの実現方法を検討するとジャイアントロックとなる構造と細粒度ロックとなる構造ではハードウェア基本構造が異なる．本節ではこれら 2 つの構造を提案検討し，Multi-ARTESSO のハードウェア基本構造を決定する．

Fig.6.1 に細粒度ロックとなる構造を示す（以下，本構造を FGL 構造と呼ぶ）．ハードウェア化した RTOS を各コアに一つずつ配置する．Hardware RTOS Control 1〜n モジュールはそれぞれ第 3 章で示した Fig.3.3 の Registers Save Memory 部および MC 部の機能から成る．Hardware RTOS Control 1〜n は並列に動作することが可能である．ロッ

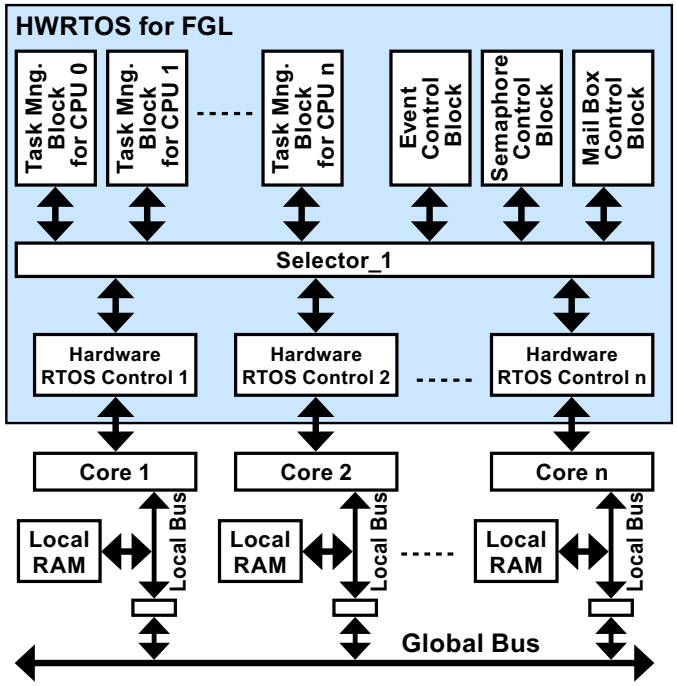


Figure 6.1: Multi-ARTESSO の基本構造（FGL 構造）

クされるべき資源（管理ブロック）は Fig.3.3 で示した Task Management block，Event Control Block, Semaphore Control Block, Mail Box Control Block である．Hardware RTOS Control 1〜n は API 実行にともない，Selector_1 を介し必要とする管理ブロックにアクセスする．Selector_1 は管理ブロックごとにロックする機能を持つ．例えば Hardware RTOS Control 0 が Event Control Block をアクセスすると，他の Hardware RTOS Control n が同ブロックをアクセスしようとしても待たせる．

次に Fig.6.2 にジャイアントロックとなる構造を示す（以下，本構造を GL 構造と呼ぶ）．この構造では RTOS は一つしか存在せず，複数のコアが同時に API を発行しようとした場合 Selector_2 により調停が行われる．すなわちあるコアからの API を実行している間は他のコアからの API 要求は待たされる．Hardware RTOS Control モジュールは FGL 構造と同じように Registers Save Memory，MC の機能から成る．MC と管理ブロックの接続は Fig.3.3 と同一である．したがって MC が管理ブロックをア

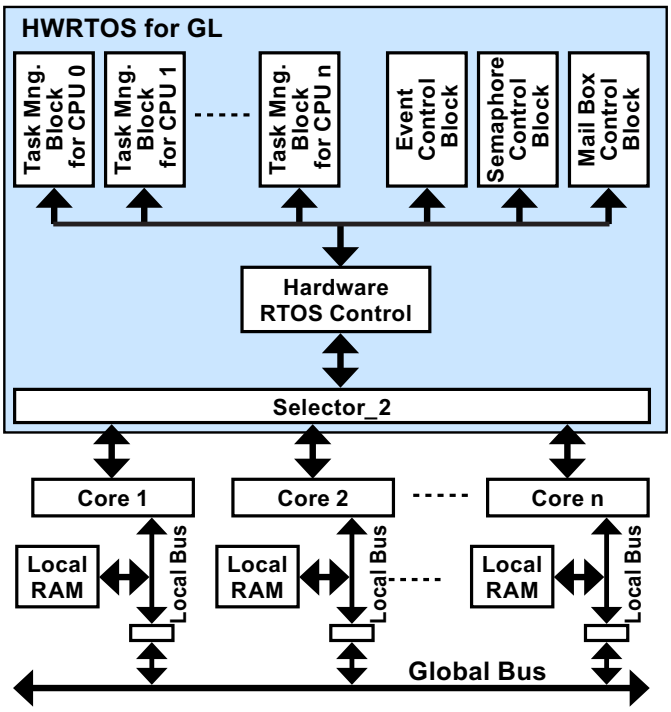


Figure 6.2: Multi-ARTESSO の基本構造（GL 構造）

クセスする上で排他制御は必要無い．

これら 2 方式を実現するハードウェアを実際に開発し評価することはコストがかかるため机上で性能比較および回路量の比較を行った（Table 6.1）．まず性能比較を示す．API 最小実行時間は，複数のコアからの API 発行が競合しない場合である．コアと Hardware RTOS Control の間の遅延および Hardware RTOS Control から管理ブロックまでの遅延を比較すればよい．競合がないという前提では，Selector_1 および Selector_2 において 3 サイクル程度の競合制御サイクル数，その他の箇所ではゲート遅延のみが発生する．このため双方の回路遅延はほぼ同じである．したがって両者における API 最小実行時間の差は無い．API 最大実行時間は，スタベーションが発生しない場合，全てのコアが同時に同じ API を発行しかつ同じ管理ブロックにアクセスしたときであり，全てのコアが管理ブロックへのアクセスする時間の総和になるため，こちらの差もほとんどない．以上より API 最小実行時間，API 最大実行時間とも両方

Table 6.1: 各ロック方式の比較

Items to compare	Result
API max execution time	FGL ＝ GL
API min execution time	FGL ＝ GL
Probability of contention	FGL < GL （GL is also low）
Hardware volume	FGL > GL
Operation frequency	FGL < GL

式の性能はほぼ同じである．

GL 構造は FGL 構造に比較すると競合発生確率は高い．ただし GL 構造であっても，SoftMulti-RTOS と比較して API 処理の時間が短いため絶対数値としての競合発生確率は低い．

次にハードウェア量の比較を示す．管理ブロックの回路は両方式ともほぼ同じ回路である．異なるのは FGL 構造の Hardware RTOS Control 1〜n と GL 構造の Hardware RTOS Control，FGL 構造の Selector_1 と GL 構造の Selector_2 である．

まず GL 構造の Hardware RTOS Control と Hardware RTOS Control 1〜n は Fig.3.3 の Register Save Memory と MC の機能に相当するが，Register Save Memory はタスクの総量に比例するため回路量は同じになる．MC は，FGL 構造では n 個実装されており，GL 構造の約 n 倍の回路量となる．

Selector_1 は n × (k+m) のスイッチ，Selector_2 は n × 1 のスイッチで構成される．n はコアの個数である．Selector_1 の式において m は分割した粒度の数であり，k はタスク数である．Selector_1 は k+m 個の多重化回路と調停回路が必要であるのに対し，Selector_2 は多重化回路および調停回路は 1 つである．以上よりハードウェア量の比較では GL 構造の方が有利である．また回路化において，n × (k+m) のスイッチと n × 1 のスイッチでは前者の方が，論理回路が多段になるため最大動作周波数は低くなり，この見地からも GL 構造が有利である．

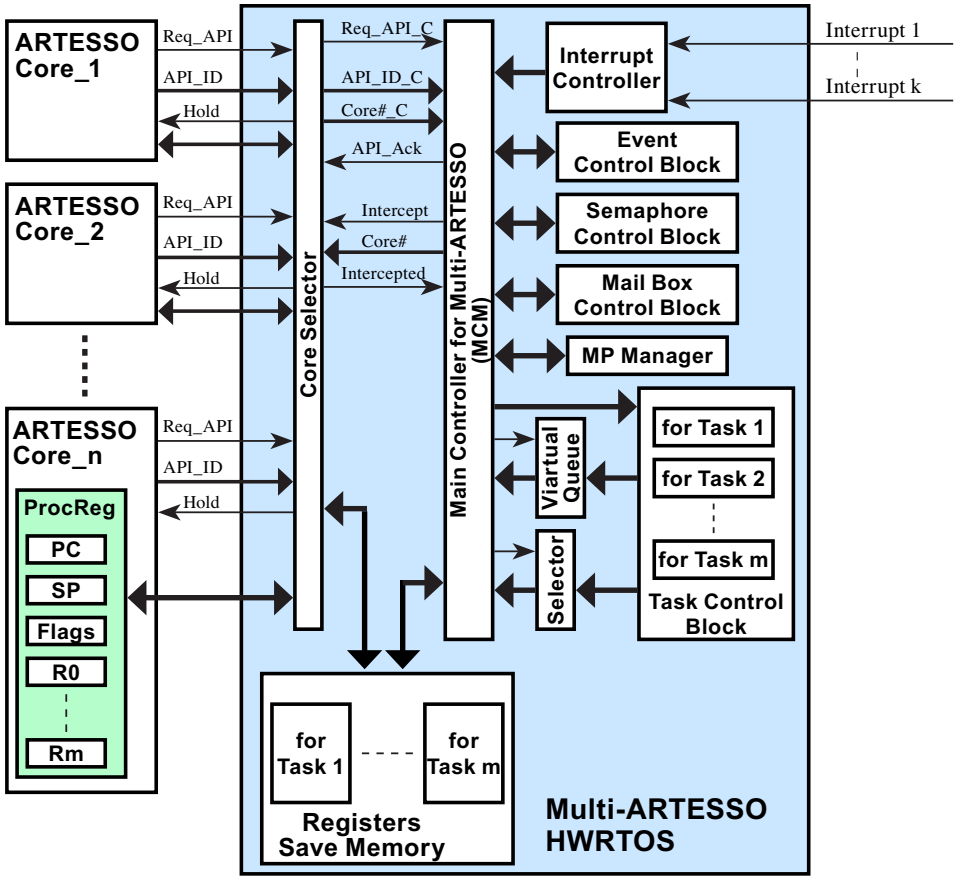


Figure 6.3: Multi-ARTESSO におけるシステム構造

以上の比較検討結果より Multi-ARTESSO においては GL 構造を採用するという決定をした.

6.2.2.2 MULTI-ARTESSO の詳細設計

6.2.2.1 のハードウェア基本構造に基づき Multi-ARTESSO の詳細設計を行った. Fig.6.3 は Multi-ARTESSO の全体構成図である. 各コアは Single-ARTESSO と同じ ARTESSO Core を流用する. 従って各コアと Multi-ARTESSO HWRTOS とのインターフェースは同一である. 第 2 章 Fig.2.6 において SoftMulti-RTOS の機能構造を示したが, ハードウェア化にあたり Multi-ARTESSO では SoftMulti-RTOS と同等の機能を実

現する必要がある．また 6.2.1 で示したシステム要件を満たさなければならない．このため以下の条件を詳細設計に取り入れた．

(a) 各コアからの API 要求を確実に処理
(b) セマフォ，イベント，メールボックス等のオブジェクトはコアに属さず，各コア上のタスクがオブジェクトを共有可能
(c) 各タスクおよびレディーキューは各コアごとに管理
(d) 他のコアのディスパッチを伴う API（コアを跨いだ API）の実現
(e) API 実行時間および割り込み応答時間の最悪値を定めることができる

上記条件をどのように実現しているかを以下に示す．条件 (a) の実現方法を示す．各コアからの API 処理要求は，Core Selector で競合を調停し，Req_API_C 信号および API_ID_C 信号により MC for Multi-ARTESSO（以降，MCM と呼ぶ）に伝えられる．Req_API_C 信号および API_ID_C 信号は，Single-ARTESSO の Req_API 信号および API_ID 信号と同じ意味である．またどのコアの要求かはコア#_C 信号により伝えられる．MCM はこれを受け付けると API_Ack 信号で Core Selector に受け付けを伝える．競合が発生していない場合 API 発行時の Core Selector の遅延は往復で 3 サイクルである．MCM は機能拡張を行っているが，各 API 実行のサイクル数の増加はない．従って競合が発生していない場合，Single-ARTESSO に比較して Multi-ARTESSO の API 実行時間は全て 3 サイクルの増加となる．

条件 (b) を満たすために，MCM ではオブジェクトをコアと独立に管理する．したがって各コア上のタスクはオブジェクトを共有することができる．

条件 (c) から，各タスクおよびレディーキューは各コアごとに管理しなければならない．タスク識別子の上位ビットをコア識別子に割り当てることによりタスクが割り付けられているコアを判別する．またレディーキューをコアの数だけ用意し，コア識別子とキュー識別子を関連づけることにより，コアごとのレディーキューを実現した．さらに，コアごとに一つずつ Run 状態であるタスク（カレントタスク）が存在するが，MP Manager モジュールを新たに開発し，MP Manager により各コアのカレントタスクに関する情報を一元管理した．これにより MCM では処理の負荷が軽くなる

ばかりでなく，コアごとの管理情報を持つ必要がなくなるため MCM の回路記述はコアの数に依存しない記述とすることができた．

次に (d) の条件の実現について述べる．前述のようにタスク識別子の上位ビットはコア識別子であるが，MCM 内部ではディスパッチ処理以外はコア識別子を意識せず単にタスク識別子として処理を行っている．このためコアを跨がる API であってもコアに関係なく処理がなされる．ディスパッチ処理ではコアを考慮し，他のコアのタスクスイッチを行う場合，該当コアを停止させタスクスイッチを実行する．

タスクスイッチは次のように行う．MCM が Core#信号出力によりコアを指定し，Intercept 信号によりコアに停止要求を出す．Core Selector は指定されたコアに対し Hold 信号を 1 にし，コアを停止させる．コアが停止すると MCM はコア内のレジスタを Registers Save Memory に退避し，次に実行するタスクのレジスタセットを Registers Save Memory からコアに書き込む．MCM が Intercept 信号を 0 にするとそのコアへの Hold 信号が 0 に戻り，コアは動作を再開する．以上のようにタスクスイッチを実現した．

条件 (e) については，6.2.2.5 および 6.2.2.6 で述べる．

6.2.2.3　ITRON 仕様の採用

Single-ARTESSO では日本で広く使用されている ITRON 仕様を選択したが，Multi-ARTESSO では同様に ITRON 仕様をマルチコアに拡張した仕様を採用した．

6.2.2.4　RTOS とコアの並列処理の向上

6.2.1 のシステム要件 (5) にあるよう Multi-ARTESSO ではコアと RTOS の並行処理を実現した．Multi-ARTESSO では Fig.6.4 で示すよう，API を発行したコアのみを停止するよう MCM を設計した．API 処理の結果，他のコアのタスクスイッチが必要な場合のみ，MCM は該当するコアを停止させタスクスイッチを行う（Fig.6.4 では Core_2）．Fig.6.4 において Core_1 に割り当てられているタスクは Task 1x，Core_2 に

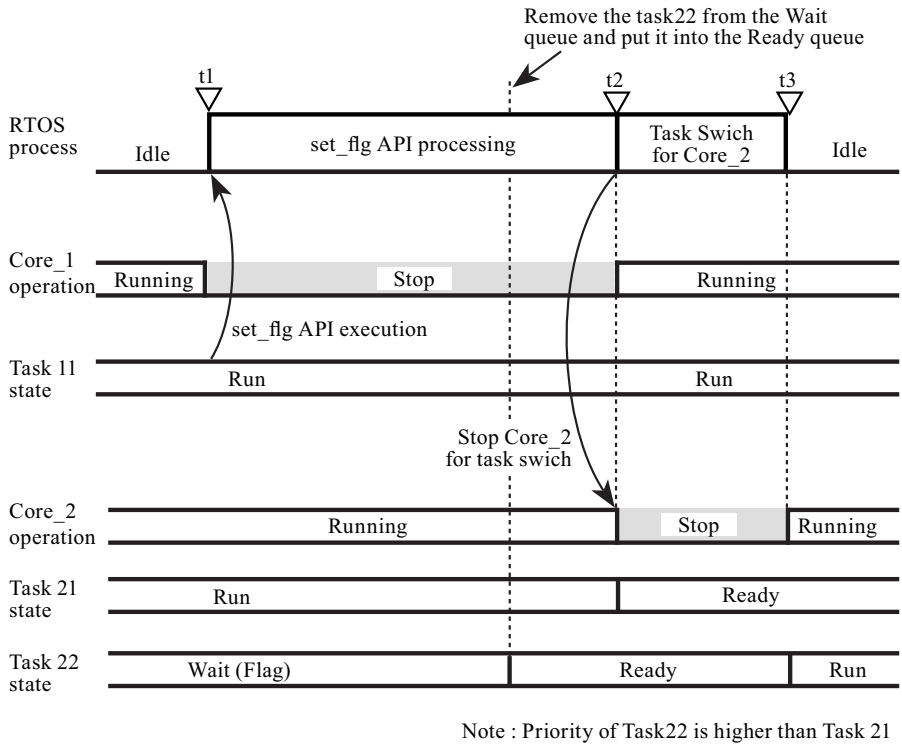


Figure 6.4: Multi-ARTESSO におけるタスクスイッチ

割り当てられているタスクは Task 2x と示した．以降の図でも同様とする．

Fig.6.4 の動作を説明する．Core_1 で Task 11 が RUN 状態であり，Core_2 では Task 21 が RUN 状態，Task 22 がフラグを待っている．Task 22 の優先度は Task 21 より高いものとする．Task 11 が Task 22 の待っているフラグに対し set_flg を発行する (t1)．MCM はこの API を処理し，Core_2 を停止させる (t2)．Task 21 から Task 22 にコンテキストスイッチを行い，Task 22 が RUN 状態になる (t3)．このようにコアを跨がった API 処理においても，コアと MCM の並行動作により，コアが停止するのはコンテキストスイッチの期間のみである．

割り込み発生時のコアと MCM の並行動作は以下のとおりである．Interrupt Controller が MCM に割り込み発生を通知したとき，MCM は ISR が動作すべきコアを指定して，前記タスクスイッチ手順で実行中タスクを ISR に切り換える．したがってコ

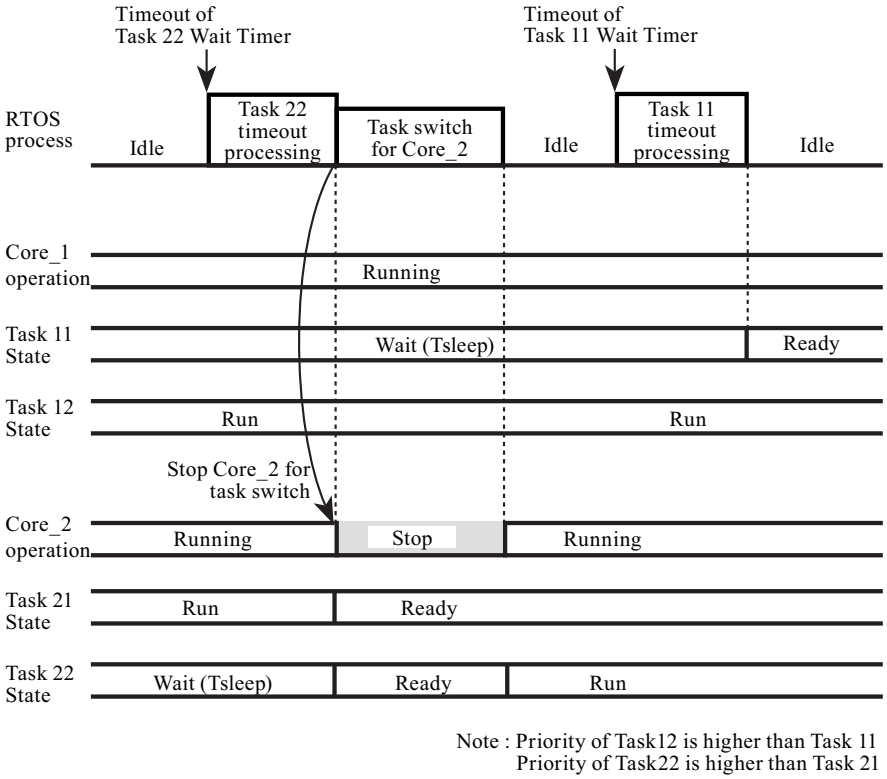


Figure 6.5: コアと RTOS の並行処理

アが停止するのは，ISR が起動されるコアのタスク切換処理の期間のみとなる．

さらにこの並列化メカニズムを tick オフローディング，IIA オフローディング処理にも適用した．すなわち MCM で実行される tick オフローディング，ハードウェア ISR 処理は全てのコアと並列に処理し，処理の結果タスクスイッチが必要なときのみ前記手順でタスクスイッチを行うようにした．Fig.6.5 は tick オフローディングの例を示している．Task22 の tick オフローディングでは，Run 状態にあった Task21 より Task22 の方が優先度が高いためタスクスイッチが発生する．したがってこの期間のみコアが停止する．しかし Task11 のタイムアウトでは，Task11 より Run 状態の Task12 の方が優先度が高いためタスクスイッチが発生せず，したがってコアの処理は全く中断されない．

以上のように RTOS とコアの処理をできるだけ並列化し，タスクスイッチが発生

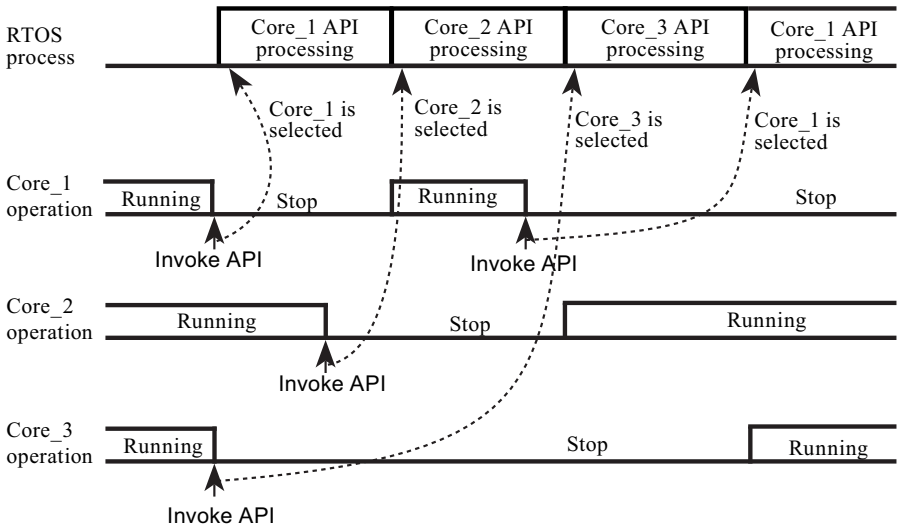


Figure 6.6: スタベーション回避アルゴリズム

しない時はコアを停止せず，またタスクスイッチがあった場合であっても該当するコアのみを停止させるようにしたため，コアの中断時間を大幅に削減できた．

6.2.2.5　スタベーションの回避

Core Selector の動作を以下のようにすることによりスタベーションを回避した．Core Selector は API 発行の競合を検知すると，コア番号の小さいものから順次 API を処理する．処理が終わったコアは再度 API を発行しても他のコアの処理が全て終了するまで「処理の候補」から外れる．この様子を Fig.6.6 に示す．ジャイアントロックであるためスタベーション回避回路は Core Selector の内部に一つ実装されているだけであり，全体の回路規模を押し上げる要因にはなっていない．

6.2.2.6　コア間とコア内の排他制御の問題

SoftMulti-RTOS ではロックの後割り込み禁止を行う前に割り込みが発生すると，ロック期間を定めることができないことが問題であった．一方割り込み禁止にしてからロックを行うと，スタベーションが回避されていないとロック期間を定めることが

できず，割り込み応答時間を定められなかった．Multi-ARTESSO では，MCM が管理ブロックへのアクセスや割り込みのハンドリングを集中管理している．MCM は管理ブロックのロックと割り込み禁止を同時に実行するよう設計されており，上記問題が発生しない．また API 処理がハードウェア化されており，割り込みを受け付けない期間は数サイクルで終了するため，割り込み処理開始までの時間を定めることができる．以上より Multi-ARTESSO では MultiSoft-RTOS でのコア間とコア内の排他制御の問題は生じない．

6.2.2.7　統計用カウンタ

Task Control Block には各タスクごとに統計用のカウンタを実装した．各タスクごとに STOP，WAIT，READY，RUN の状態に留まった累計時間を測定することができる．この機能によりタスクの分割や割り当てコアの変更のための情報を得ることができるようになった．

6.3　評価および要件への対応

本章では 6.2 に基づき回路を開発し評価を行うとともに，要件への対応を示す．

6.3.1　評価項目

6.3.1.1　回路量の評価

FPGA により Fig.6.3 のシステムを実現し，回路量を評価する．システムの仕様は以下の通りである．

- コア 8 個
- タスク数 64 個
- Event ID 数 128 個

- Semaphore ID 数 128 個
- MailBoxID 数 128 個
- 最大 Mail 数 256 個

6.3.1.2 API 実行時間の評価

タスク起床 API およびセマフォ資源返却 API を実行したときの SoftSingle-RTOS, SoftMulti-RTOS，Single-ARTESSO，Multi-ARTESSO それぞれの実行時間を測定する．各 API とも，ディスパッチ有り／無し，マルチコアの場合は API 処理が同一コアで閉じる場合とコアを跨ぐ場合の測定を行う．ただし複数のコアからの同時 API 発行による競合はないものとする．

6.3.1.3 最悪 API 実行時間の評価

スタベーションが発生していると API 実行時間の最悪値を定めることができない．このためまず前述のスタベーション回避回路が確実に動作しており，スタベーションが発生しないことを確認する．

次に，Multi-ARTESSO における API 実行時間の最悪値の測定を行う．複数のコアが同時に API を発行するとこれらの API は MCM で順次処理される．最悪 API 実行時間は，全てのコアが同時に API を発行した場合における，最後に処理される API の終了までの時間である．本評価では各コアから同時にタスク起床 API を発行させ，API の発行から最後に実行された API の完了までの時間を測定する．API 処理が同一コアで閉じる場合とコアを跨ぐ場合の双方を，またディスパッチがある場合とない場合について測定を行う．同時に API を発行するコアは 1 個〜8 個と変化させる．

6.3.2 評価方法

「6.3.2.1 回路量と最大動作周波数の評価」では，XILINX 社製 FPGA，XCV6VLX760（スピードグレード 2）に Fig.6.3 の回路を実現し，回路量を評価する．「6.3.2.2 API 実行

Table 6.2: Multi-ARTESSO の回路量

Modles	CLB Slices	FF (bit)	Max operational frequency (MHz)
Main Controller	317	745	301
Task Control Block	11,007	15,212	214
Virtual Queue / Selector	329	450	-
Hardware ISR	548	1,611	266
Mail Box Control Block	1,871	5,857	213
Regsters Save Memory	32	0	-
MP Manager	79	120	1,319
Core Selector	656	71	300
Others	23	67	-
Subtotal of Hardware RTOS	14,862	24,133	92
Core	1,353	1,976	72
Subtotal of 8 Cores	10,821	15,808	72
Total	25,683	39,941	56

時間」における SoftSingle-RTOS および SoftMulti-RTOS の評価は [18] のデータを使用する．また Single-ARTESSO および Multi-ARTESSO の評価は Verilog シミュレータにより測定・確認を行う.「6.3.2.3 最悪 API 実行時間の評価」における Single-ARTESSO および Multi-ARTESSO の評価は Verilog シミュレータにより測定・確認を行う.

6.3.2.1 回路量と最大動作周波数の評価

Table6.2 に Multi-ARTESSO の FPGA への実装結果を示す．CLB スライスとは FPGA の論理モジュールの基本単位であり，使用したスライス数が組合せ回路の大きさに比例する．また FF は実装されているフリップフロップの数である．

SoftMulti-RTOS と Multi-ARTESSO の回路量の比較をする．SoftMulti-RTOS と比較して Multi-ARTESSO で増加する回路は Hardware RTOS 部で示される値であり，したがって本構成（サポートするオブジェクトと 8 コア）においては Hardware RTOS により回路量が 137%増加する（Table6.2 における”Subtotal of Hardware RTOS”と ”Subtotal of 8 Cores”の CLB スライス比）．Core 部の回路量はコア数に比例するが，Hardware

Table 6.3: API 実行時間

API	ディスパッチ	ディスパッチするタスクのコア	Soft-Single RTOS	Soft-Multi RTOS	Single ARTESSO	Multi ARTESSO
タスク起床	無	同じ	250	450	8	11
		異なる	-	450	-	11
	有	同じ	350	550	11	14
		異なる	-	850	-	25
セマフォ資源返却	無	同じ	250	500	10	13
		異なる	-	500	-	13
	有	同じ	300	650	13	16
		異なる	-	900	-	27

(単位 : クロックサイクル)

RTOS は 6.3.2.1 に記載したシステム仕様（サポートするオブジェクトの数）に依存する．特に影響が大きいのはタスク数であり，Task Control Block と Virtual Queue / Selector モジュールの回路量は実装されるタスク数に比例する．また Mail Box Control Block はメールボックスおよび最大メール数に依存する．Table6.2 の値を利用し，開発しようとする LSI における回路の増加量の概算値を求めることが可能である．

各モジュールの最大動作周波数を”Maximum operational frequency”の欄に示す．この値は回路合成後，配置配線前の値である．モジュール内のフリップフロップが 1 段以下のモジュールは最大動作周波数を算出できないため”-”で示す．Single-ARTESSO は ASIC で実装し，Multi-ARTESSO は FPGA で実装した．このため両者の回路量の直接比較はできず，これは今後の課題である．

6.3.2.2 API 実行時間

結果を Table6.3 に示す．結果が示すように Multi-ARTESSO は SoftMulti-RTOS に比較し 30〜40 倍高速である．以下，セマフォ資源返却 API を例にとり細部を確認する．SWRTOS の場合でディスパッチ無しのとき SoftSingle-RTOS から SoftMulti-RTOS で 2 倍であるのに対し，Single-ARTESSO から Multi-ARTESSO では一定量の増加（3 サイ

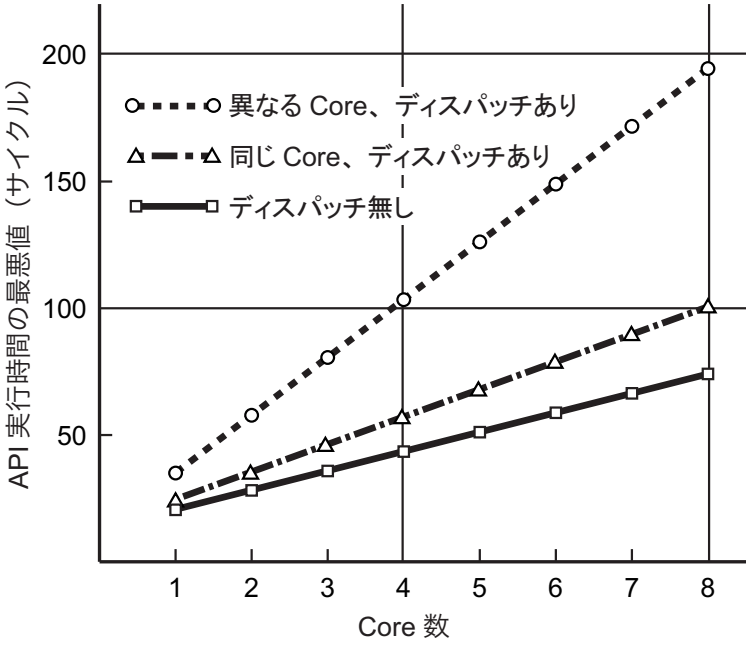


Figure 6.7: Multi-ARTESSO 最悪 API 実行時間

クル）である．これは Core Selector で生じる一定の遅延である．またディスパッチがあるとき SoftSingle-RTOS から SoftMulti-RTOS で 2.17 倍であったが Single-ARTESSO から Multi-ARTESSO では「同じく一定量の増加（3 サイクル）であった．但しディスパッチ先が同一コアの場合と別コアの場合を比較すると（ディスパッチありの列の「同じ / 異なる」），SoftMulti-RTOS では 1.38 倍であるのに対し Multi-ARTESSO では 1.69 倍に増加している．理由は以下の通りである．この場合，API を発行したコアのタスクに対し戻り値を返すなどの処理と，タスクスイッチを行うコアでのディスパッチ処理の 2 つの処理を行う．SoftMulti-RTOS ではそれぞれのコアにおいて RTOS のコードが実行されるため 2 つの処理がパラレルに実行されるが，Multi-ARTESSO では MCM がシリアルに実行する．このため倍率比較では Multi-ARTESSO の方が劣っている．

6.3.2.3 最悪 API 実行時間の評価

シミュレータにより，Fig.6.6 で示される機能が確実に動作することを確認し，スタベーションが発生していないことを確認した．次に競合時 API 実行時間の評価を実施し，この結果を Fig.6.7 に示す．グラフが示すよう実行時間の最悪値はコアの数に正比例する．8 個のコアにおけるタスク起床 API での最悪値は 195 サイクルであった．変動幅は，ディスパッチのある場合を例に取ると，最悪値 195 と Table6.3 のタスク起床 API の最小値 11 との差であるから 184 サイクルであり，リアルタイムアプリケーションソフトウエアを作成する上で制約にならない十分小さな値である．

6.3.3 要件への対応

システム要件 (1)：Multi-ARTESSO においても ITRON 仕様をマルチコアに拡張した API を採用した．

システム要件 (2)：Single-ARTESSO と同等の仮想キューを使用し，複数のコアが共有するキュー，コア別に存在するコアのキューの双方を実現した．回路増加量は実装するタスク数に応じた増加であり，マルチコア化に伴う増加はほとんどない．

システム要件 (3)：Multi-ARTESSO のシングルコアでの使用が可能な構成とした．Single-ARTESSO との性能比較では，Core Selector での遅延が増えるのみであった（全ての処理において 3 サイクル増加）．

システム要件 (4)：Multi-ARTESSO では，6.2.2.5，6.2.2.6 で示したようにスタベーションおよびコア間・コア内排他制御の問題は発生せず，API 実行時間の最悪値，割り込み応答時間の最悪値を定めることができた．変動幅については定量的な評価が困難であるため定性的な評価とする．SoftMulti-RTOS では，第 2 章 2.4.2.3 に示した「スタベーション」および「コア間とコア内の排他制御の問題」により，API 実行時間の最悪値を定めることができない．一方 Multi-ARTESSO では最悪値を定めることができており，したがって API 実行時間の変動幅を一定値以内に抑えたと言える．

システム要件 (5)：6.2.2.4 で示したように，コアと RTOS が並列動作できるようにした．これにより各コアのアベイラビリティ，システム全体のアベイラビリティが向上した．

以上のように全ての要件を満たした．

6.4 むすび

組込みシステムにおいてもマルチコア化が進展しており，マルチコアシステムにおいても高度なリアルタイム性を実現するため本研究では Multi-ARTESSO を開発した．研究結果としては，SoftMulti-RTOS と比較し絶対的な数値が優れているだけでなく，SoftSingle-RTOS から SoftMulti-RTOS に拡張する上で問題であった「スタベーションの問題」や「コア間とコア内の排他制御の問題」を解決した．本研究による Multi-ARTESSO は，単に API 処理時間や割り込み応答処理時間が高速なだけでなく，これらの最悪値を定めることができることから，高速かつ高度なリアルタイムマルチコア環境を必要とするアプリケーションに対し RTOS 環境への適用を可能にした．さらに最悪処理時間がコアの数に正比例することから，将来的にコアの数を増やしても，最悪処理時間を事前に見積もることが可能になった．

CHAPTER 7

結論

7.1 まとめ

本論文のメインテーマはRTOSのハードウェア化である．目的は組込みシステムにおけるアプリケーションの高速化である．特にネットワークプロトコル処理や機械制御処理においては，従来技術の延長線上での性能改善方法では消費電力やコストの増加を招いてしまう．本研究では，RTOSをハードウェア化することにより，コスト・消費電力の大幅な増加を伴うことなくこれらのアプリケーションの劇的な性能向上を実現した．ネットワーク処理においては全処理量に対するRTOSの処理の比率が高く，また機械制御においては高速化要求すべき箇所が割り込み応答であるため，このような分野においてはRTOSのハードウェア化がこれらアプリケーションの性能向上において極めて有効な手段である．

本論文ではまず，シングルコア対応のHWRTOSを開発した（ARTESSO HWRTOS）．処理をハードウェア化することにより高速なAPI実行を実現した．API仕様はITRONと同等な仕様とし，既存のITRON上で開発されたソフトウェア資源を活用できるよう配慮した．またオリジナルコアであるARTESSO Coreを同時に開発した．ARTESSO CoreはARTESSO HWRTOSと密結合で接続するための専用のイン

ターフェースを有し，これにより極めて高速な API 発行と高速なコンテキストスイッチを実現した．また組込みシステムに使用する RTOS は一般に数千個のキューを内蔵するが，従来技術でこれをハードウェアで実現すると膨大なハードウェア量になってしまう．本研究では仮想キューという新しいアイディアを提案し，極めて少ない回路量で数千個のキューを作り出し，またキューのオペレーション時間を 1 サイクル時間で実現した．以上のような新しい概念をアーキテクチャに取り込むことにより，API 実行時間においては，従来のソフトウェア RTOS では 200〜600 サイクル程度であった処理時間が，6〜11 サイクル時間と大幅に性能向上した．また本アーキテクチャに基づき ASIC を試作し TCP/IP プロトコルソフトウェアを実装し，スループットを測定した結果，双方 50MHz の動作クロックにおいて，従来のソフトウェア組込みシステムでは 11Mbps であったのに対し，ARTESSO では 125Mbps のスループットを実現した．また 100Mbps のスループット時の消費電力は従来技術の 1/7 であり，ARTESSO が低消費電力であることを実証した．

次に前記開発した ARTESSO HWRTOS に，高速化した割り込み機能を追加した．tick 処理をハードウェア化した tick オフローディング機構，および ISR 処理を定型化しハードウェア化した IIA オフローディング機構を ARTESSO HWRTOS に組み入れた．この結果割り込み発生から ISR 起動までの時間においては，従来のソフトウェア RTOS では 92〜524 サイクル時間であったのに対し，ARTESSO では 5〜13 サイクル時間となった．一方割り込み発生から次にタスクが起動されるまでの時間においては，従来のソフトウェア RTOS では 532〜1960 サイクル時間であったのに対し，ARTESSO では 14〜17 サイクル時間となった．双方とも大幅に応答時間を短縮化を実現しただけでなく，最小値と最大値の変動幅を大幅に縮小した．これはリアルタイム処理において大きなアドバンテージである．

次に汎用コアである ARM コアから ARTESSO HWRTOS を利用するシステムを開発した．これまでの研究は ARTESSO HWRTOS と ARTESSO Core を専用インターフェースで接続し，高速な API 発行および高速なコンテキストスイッチを実現して

いた．一方専用コアではなく，汎用コアで ARTESSO HWRTOS を利用したいというニーズが多く，この要望に応えた．専用インターフェース方式である TC-HWRTOS を改造し，また新たに機能を追加し ARM バスから ARTESSO HWRTOS にアクセスできる LC-HWRTOS アーキテクチャを提案した．さらにこのアーキテクチャに基づき，ルネサスエレクトロニクス（株）において，FA ネットワークコントロール用 ASSP を開発し，現在量産中である．LC-HWRTOS は TC-HWRTOS の性能には劣るものの，SWRTOS に比較して十分アドバンテージを得られる性能を出すことができた．また本研究では tick オフローディングおよび IIA オフローディング処理とコアの並列処理を実現することにより，コアへのインタラプトを大幅に削減することを可能にした．

最後にマルチコア対応 HWRTOS を開発した．このシステムはコアとして ARTESSO HWRTOS を採用し，アーキテクチャは TC-HWRTOS である．マルチコアのシステムでは RTOS の管理情報を各コアからアクセスするため，そのロック単位としてジャイアントロック方式と細粒度ロック方式が存在する．検討した結果，最大アクセス時間は双方ともほぼ同じであるのに対し，ハードウェア量がジャイアントロックの方が圧倒的に少ないため，本開発ではジャイアントロック方式を採用した．また本開発においては，従来のマルチコア対応 SWRTOS において問題となっていたスタベーションやコア内・コア間排他制御の問題を防ぐ回路を実装し，その結果 API の最悪実行時間を定義することができた．性能評価では，コアを跨がった API 実行時間において，SWRTOS では 850〜900 サイクル時間であったのに対し，ARTESSO では 25〜27 サイクル時間と大幅に改善した．

以上のように本研究を通して，従来技術のように高速化のためにコスト・消費電力増加を伴うことなく，ネットワーク処理性能の向上，機械制御処理性能の向上を達成することができた．また，割り込み応答性能劣化のため RTOS を使用できなかったようなアプリケーションにおいても RTOS が使用できるようになり，このためソフトウェア開発の生産性の飛躍的向上，システム安全性の大幅な向上に貢献できた．

7.2 今後の課題

本研究により，HWRTOS によりネットワーク処理や機械制御処理において，大幅な消費電力・コスト増を伴うことなく，飛躍的な性能向上を実現した．第 5 章で述べたように既にルネサスエレクトロニクス（株）において一品種の ASSP が量産化されており，さらにラインアップの拡張を予定している．これらは FA ネットワークアプリケーションに特化しているが，これに留まらずネットワーク処理や機械制御処理における幅広い分野で商品化されるよう追加研究開発を行ってゆく．またさらに，API 実行処理時間・割り込み応答時間が高速であるという有利性を活かし，上記以外の分野での商品化を実現できるよう努力する．こうした目標を達成するために，ARTESSO HWRTOS の機能の充実，ITRON 以外への展開が重要である．

ARTESSO HWROTS の機能充実として第一にサイクリックハンドラ機能を実装する．現在サイクリックハンドラは割り込み機能によりソフトウェアで実行させているが，これを完全ハードウェア化しリアルタイム性能をさらに向上させ，特に機械制御処理での利用促進を目指す．第二に保護機能の充実である．先進的な RTOS ではアプリケーションの不良動作が他のアプリケーションに影響を及ぼすことがないよう，時間保護機構やメモリ保護機構が実装されている．従来技術においてこのような機能を導入した場合 RTOS のオーバヘッドに関し十分留意する必要があった．HWRTOS では従来の RTOS に比較しオーバヘッド時間が極めて小さく，とくに TC-HWRTOS ではコンテキストスイッチが高速であるため，保護機構の効率的な動作が可能である．以上のように，ARTESSO HWRTOS の特徴である高速性という特徴を活かした機能追加を行うことにより，採用分野を広げる．

次に ITRON 以外の RTOS への展開について述べる．ITRON 以外への RTOS への展開方法は二つある．一つ目の方法は他の RTOS の API を実現するソフトウェア・ラッパーを開発することである．現在 ARTESSO HWRTOS の内部構造は ITRON 仕様に準拠しているものの，ITRON の C 言語 API を実現するためにソフトウェアのラッパー

を実装している．このラッパー部分を改造することにより，比較的容易に他の ITRON に API が類似している RTOS の API を実現可能である．もう一つの方法は HWRTOS のハードウェア構造自体を他の RTOS に対応する方法である．ITRON と API や構造が似ていない場合，この方法を採る．

Micrium 社では μC/OS というオリジナルの RTOS を開発，販売している．同社では μC/OS 対応の ARTESSO 用ラッパーを開発し，この結果 5 章で紹介した R-IN32 上で ARTESSO HWRTOS を使用した μC/OS の実行が可能となった．今後他の RTOS メーカにも同様のアプローチを行う．

AUTOSAR OS は自動車の ECU で使用される国際標準 RTOS であるが，構造，API とも ITRON と大きく異なる．たとえば，イベント，セマフォなどのオブジェクトにおいて ARTESSO HWRTOS (ITRON) では，ウエイト状態からの起床は FCFS アルゴリズムに則って動作するが，AUTOSAR OS のアルゴリズムは全く異なる．したがって上記ラッパー方式で実現した場合，ITRON との差異のため，大幅な性能向上，特に割り込み応答での性能向上が困難である．AUTOSAR OS に HWRTOS を適用するためには AUTOSAR OS 用に HWRTOS 自体を改造することが望ましい．今後ニーズを見ながら開発すべきかどうかを見極める．

また，近年様々な組込みシステムにおいて採用されているオープンソースの FreeRTOS に ARTESSO HWRTOS を適用させるためには以下の手法での実現を検討中である．FreeRTOS には ITRON のセマフォやメールボックスに類似した同期通信機能があり，この部分はソフトウェア・ラッパーでの対応が可能である．しかし FreeRTOS のスケジューラは優先度付き FCFS ではない．このため主にスケジューラ部分のハードウェア構造の改造のみを行うことでコストパフォーマンスの高いシステムを得ることができる．

最後に適用分野の拡大について述べる．本論文で述べているよう ARTESSO HWRTOS の効果的な利用分野はネットワーク処理や機械制御処理である．しかし，現時点で商品化に至っているのは FA ネットワークの分野のみであり，幅広くネットワーク

処理や機械制御処理分野での採用を目指している．特に FA ネットワーク + 産業機械制御での利用が興味をもたれており，この場合 Dual Core システムになるため（ネットワーク制御用と機械制御用にそれぞれ独立してコアをアサイン），マルチコアタイプの ARTESSO HWRTOS を適用する．この技術は産業機械だけでなく輸送機械，プリンター等への適用が可能であり，適用分野の広範な展開のベースとなりうる．

一方コンシューマ機器を始め，16 ビット級コアを使用した比較的単純な制御を実行する分野もターゲットの一つである．この分野においてもネットワーク化が浸透しつつあり，ネットワークプロトコルを実装するとその処理負荷の増加のためコアの性能を上げざるを得ない．またこのため消費電力やコストも高くならざるを得ないといった事例が数多くある．ARTESSO HWRTOS を使用することによりネットワークプロトコルを実装しても従来通りの処理性能を維持し，かつ消費電力やコストの大幅な増加なくシステムを実現できる．十分な性能を提供するためには TC-HWRTOS が必須であり，半導体メーカとの共同開発等が必要である．今後 TC-HWRTOS を見据えた半導体メーカとのコラボレーション等の機会を積極的に創造する．

謝辞

本研究に際しまた本論文執筆に際し，終始親身なご指導，ご助言を賜りました名古屋大学大学院情報科学研究科，高田広章教授および本田晋也准教授に深く感謝申し上げます．また本研究に際し，ご協力をいただいた同研究科一場利幸氏，石川拓也氏に感謝申し上げます．さらに本論文とりまとめにあたり，丁寧なご助言を賜りました同研究科枝廣正人教授に感謝申し上げます．

名古屋大学大学院入学前二年間にわたり，本研究の論文執筆，投稿に際し懇切丁寧なるご指導ご助言を賜りました九州大学安浦寛人副学長，京都大学大学院情報学研究科石原亨准教授に心より感謝申し上げます．

本研究の成果の論文化にあたり，既に本研究の一部成果を採用していただいている製品の性能データの採取，提供等ご協力を賜りましたルネサスエレクトロニクス株式会社第二ソリューション事業本部傳田明事業部長，鈴木克信部長に対し厚く御礼申し上げます．

最後に，本研究実施にあたり，経済産業省「平成 22・23 年度戦略的基盤技術高度化支援事業」，独立行政法人新エネルギー・産業技術総合開発機構「平成 17・18 年度産業技術実用化開発事業」および同「平成 24 年度イノベーション実用化ベンチャー支援事業」のご支援を賜りました．同省および同法人に深く御礼申し上げます．

参考文献

[1] TRON ASSOCIATION, "μITRON4.0 Specification", 1999.

[2] Felser M., "Real-time Ethernet – industry prospective", Proceedings of the IEEE, Volume 93, Issue 6, June 2005, pp. 1118–1129.

[3] Kohout P., Ganesh B., Jacob B., "Hardware support for real-time operating systems", in Proc. of the 1st International Conference on Hardware/Software Codesign and System Synthesis, pp. 45–51, Oct. 2003

[4] Chandra S., Regazzoni F., Lajolo M., "Hardware/software partitioning of operating systems: a behavioral synthesis approach", in Proc. of the 16th ACM Great Lakes Symposium on VLSI, pp. 324–329, 2006

[5] Parisoto A., Souza A. Jr, Carro L., Pontremoli M., Pereira C., Suzim A., "F-Timer: dedicated FPGA to real-time systems design support", in Proc. of 9th Euromicro Workshop on Real-Time Systems, pp. 35–40, 1997

[6] Mooney III V., Lee J., Daleby A., Ingstrom K., Klevin T., Lindth L., "A comparison of the RTU hardware RTOS with a hardware/software RTOS", in Proc. of Design Automation Conference (DAC'03), 2003, pp. 683–688.

[7] Mooney III V.J., Blough D.M., "A hardware-software real-time operating system framework for SoCs", IEEE Design & Test of Computers, 44–51, 2002.

[8] Mooney III. V., ”Hardware/software partitioning of operating systems”, in Proc. of Design Automation and Test in Europe Conference (DATE’03), 2003, pp. 338–339.

[9] Lindh L., ”Fastchart – a fast time deterministic CPU and hardware based real-time kernel”, in Proc. of Euromicro Workshop on Real Time Systems, pp. 36–40, Jun, 1991

[10] L. Lindh, J. Starner, J. Furunas, , “ From single to multiprocessor real-time kernels in hardware, ” in Proc. of the Real-Time Technology and Applications Symposium, pp. 42-43, May 1995.

[11] Adomat J., Furunas J., Lindh L., Starner J., ”Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems”, in Proc. of the 8th Euromicro Workshop, pp. 164–168, Jun 1996

[12] Nordstrom S., Lindh L., Johansson L., Skoglund T., ”Application specific real-time microkernel in hardware”, in Proc. of Real Time Conference, 2005.

[13] Samuelsson T., Åkerholm M., Nygren P., Johan Stärner J., Lindh L., ”Comparison of multiprocessor real-time operating systems implemented in hardware and software”, in Proc. of Int’l Workshop on Advanced Real-Time Operating System Services (ARTOSS’03), 2003.

[14] Nakano T., Utama A., Itabashi M., Shiomi A., Imai M., ”Hardware implementation of a real-time operating system”, in Proc. of 12th TRON Project International Symposium (TORN’95), pp. 34044, 1995.

[15] T. Nakano, Y. Komatsudaira, A. Shiome, and M. Imai, “ Performance Evaluation of STRON: A Hardware Implementation of a Real-Time OS, ” in IEICE Trans. Fundamentals, pp. 2375-2382, 1999

[16] Lange A. B., Andersen K. H., ”HartOS-A hardware implemented RTOS for hard real-time applications”, in Proc. of the 11th IFAC Conference on Programmable Devices and Embedded Systems, Volume# 11, Part# 1, May 23, 2012.

[17] Innovasic, Inc., ”fido100 User Guide”, May 6, 2010.

[18] 本田晋也, 高田広章：”ITRON 仕様 OS の機能分散マルチプロセッサ拡張”, 電子情報通信学会論文誌 D Vol.J91-D No.4 pp.934-944 (2008).

[19] 一場利幸，松原豊，本田晋也, 高田広章：”中断可能な優先度継承キューイングスピンロックとそのハードウェア実装”, 情報処理学会論文誌 コンピューティングシステム Vol. 4 No. 3 133-146 (May 2011).

[20] Intel, ”82571EB/82572EI Gigabit Ethernet Controller Datasheet, Dec. 2006.

[21] P. Kohout, B. Ganesh, B. Jacob,“ Hardware support for real-time operating systems, ”in Proc. of the 1st International Conference on Hardware/Software Codesign and System Synthesis, pp.45-51, Oct. 2003.

[22] M. Song, S. H. Hong, Y. Chung, ”Reducing the Overhead of Real-Time Operating System through Reconfigurable Hardware”, 10th Euromicro Conference on Digital System Design Architectures, Aug. 2007.

[23] Renesas Electronics, ”R-IN32M3-Series Data Sheet”, Dec 9, 2013.

研究業績

主論文に関連する研究業績

査読付きの学術雑誌論文

1. 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).

2. 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)

3. 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 Apr. 2015.（採録決定）

査読付きの国際会議論文

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.

学会での口頭発表

1. 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”, 第 23 回 回路とシステム軽井沢ワークショップ論文集, pp.370-375, Apr. 2010.

2. 丸山修孝, 石原亨, 安浦寛人：”仮想キューによる高性能ハードウェア RTOS の実現”, 情報科学技術フォーラム講演論文集 9(1), 115-120, 2010-08-20

3. 丸山修孝, 石原亨, 高田広章, 安浦寛人：”超高速応答を実現するハードウェア割り込み処理機構”, 電子情報通信学会技術研究報告. VLD, VLSI 設計技術 111(324), 31-36, 2011-11-21

特許

登録済みまたは査定済み特許

日本

1. 特許 4119945　”タスク処理装置”
2. 特許 4127848　”タスク処理装置”
3. 特許 4131983　”メモリ管理装置”
4. 特許 4088335　”仮想キュー処理回路およびタスク処理装置”
5. 特許 5155336　”タスク処理装置”
6. 特許 5204740　”タスク処理装置”

米国

1. US 8,060,723 B2 ”Memory Management Device”
2. US 8,327,379 B2 ”Method for switching a selected task to be executed according with an output from task selecting circuit”
3. US 8,341,641 B2 ”Task Processor”
4. US 8,776,079 B2 ” Task Processing Device”

台湾

1. TWI 416413 号　”Task Processor ”
2. TWI 426451 号　”Task Processing Device ”
3. TWI 426452 号　”Task Processing Device ”
4. TWI 438679 号　”Virtual Queue Processing Circuit and Task Processor”

中国

1. CN101529383　”Task Processing Device”
2. CN101796487　”Virtual Queue Processing Circuit and Task Processor”

審査中特許

米国

1. 12/281,333　” Task Processing Device”
2. 12/281,990　” Virtual Queue Processing Cirquit and Task Processor”
3. 13/334,042　” Virtual Queue Processing Cirquit and Task Processor”
4. 14/243,801　” Task Processing Device”
5. 14/543,288　” Task Processing Device”

インド

1. 1043/DELNP/2009　”Task Processing Device”

2. 3151/DELNP/2014 ”Task Processing Device”

台湾

1. TW98101355 ”Task Processor”
2. TW103103900 ”Virtual Queue Processing Circuit and Task Processor”
3. TW103114180 ”Virtual Queue Processing Circuit and Task Processor”

中国

1. CN201110345010A ”Task Treatment Device”
2. CN201280072694A ”Task Processing Device”

その他特許

PCT

1. PCT/JP2012/63334 タスク処理装置

その他の研究業績

査読付きの学術雑誌論文

なし

査読付きの国際会議論文

なし

学会での口頭発表

なし

商業雑誌等

なし

受賞等

なし