μITRON4.0 仕様の例外処理のための機能とその評価

本田晋也[†] 高田広章[†]

ITRON リアルタイムカーネル仕様の最新バージョンである μITRON4.0 仕様では，アプリケーションプログラムのポータビリティを確保しつつ，多様な組込みシステムへの適応性を向上させるために，必要な機能を実現するための最小限のプリミティブのみをカーネルに持たせるという設計方針をとっている．本論文では，μITRON4.0 仕様の設計にあたっての考え方と設計方針について述べ，それに従って設計された例外処理機能とオーバーランハンドラ機能について解説する．また，μITRON4.0 仕様に準拠したカーネルを実装し，それらが設計目標を満たしていることを評価する．

Exception Handling Functions of the μITRON4.0 Specification and Their Evaluation

Shinya Honda[†] and Hiroaki Takada[†]

In order to raise the adaptability to a wide variety of embedded systems while realizing the portability of application programs, the μITRON4.0 Specification, the recent version of the ITRON real-time kernel specifications, is designed so that the kernel has a minimum set of primitives with which necessary functions can be realized. This paper presents the design concepts and policies of the μITRON4.0 Specification, and describes its exception handling function and overrun handler function which are designed according to the policies. We also evaluate that these functions achieve the design goal of the specification with a kernel which is implemented based on the specification.

1. はじめに

μITRON 仕様は，組込みシステム用のリアルタイムカーネル（以下，単にカーネルと呼ぶ）の標準仕様である．μITRON 仕様に準拠して数多くのカーネルが実装されており，また広く適用されている[1)]．

μITRON4.0 仕様[2)]は，ITRON リアルタイムカーネル仕様としては第 4 世代に位置付けられるもので，1997 年からの約 2 年間の標準化検討を経て，1999 年 6 月に公開した．μITRON4.0 仕様を策定した目的は，アプリケーションソフトウェアのポータビリティの向上，ソフトウェア部品向けの機能の追加，新しい要求・検討成果の反映，半導体技術の進歩への対応の 4 点に集約できる．

組込みシステムは，組込み対象の機器を制御するという 1 つの目的に特化されたシステムであり，アプリケーションソフトウェアが入れ替えられることはないのが通常である．そのため，機器に組み込まれるカーネルには，アプリケーションが必要とする機能のみを持たせればよい．一方，多くの組込みシステムに共通する特性として，厳しいリソース制約があげられる．リアルタイムカーネル（またはその仕様）を多様な組込みシステムに広く適用できるものとし，かつ使用するリソースを最小限にするためには，カーネルの機能や実装を適用対象のアプリケーションやターゲットハードウェアに最適化できることがきわめて重要となる．この性質を適応性と呼ぶ．

μITRON 仕様の重要な設計コンセプトとして，「弱い標準化」があげられる．これは，仕様の適応性を向上させるために，カーネル仕様を厳格に標準化することを避け，実装ごとの最適化の余地を残すことをいう．たとえば μITRON 仕様では，カーネルの各種の機能の API を標準化しているが，どの API を実装しなければならないかや，その実装方法については規定していない．カーネルの実装者は，μITRON 仕様に規定された API の中から，適用対象となるアプリケーションにおける要求事項や使用できるリソース，ターゲットハードウェアの性質などを考慮して，最適な API セットとその実装方法を決定することができる．実際，

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

μITRON 仕様が 8-bit から 64-bit までの多くのプロセッサ用に実装され，多種多様な組込みシステムに適用されている[1)]ことからも，μITRON 仕様が仕様として高い適応性を実現しているといえる．

一方で弱い標準化は，μITRON 仕様に準拠したカーネルであってもそれぞれ実装している API セットが異なることを意味しており，アプリケーションプログラムのポータビリティを阻害する要因となる．近年の組込みソフトウェアの大規模化・複雑化により，アプリケーションソフトウェアの再利用やソフトウェア部品の流通が不可欠となってきており，カーネル上で動作するソフトウェアのポータビリティ（以下，単にポータビリティと呼ぶ）がこれまで以上に重要となっている．このような背景から，ポータビリティの向上を μITRON4.0 仕様の策定目的の 1 つとした．

ところが，ポータビリティの確保と適応性の向上は一般には相反する．すなわち，ポータビリティを向上させるためにはカーネルの機能とその API を厳密に標準化する必要があるが，それにより，アプリケーションの性質などに応じてカーネルの実装を最適化できる余地は狭まる．

ポータビリティと適応性を両立させる方法として，カーネルを構成可能（configurable）に実装する方法がある．これにより，カーネルの実装すべき機能が厳密に標準化されていても，カーネルを最適な構成で機器に組み込むことが可能になり，ポータビリティと適応性が両立できる．たとえば，カーネルには多くの機能を持たせておき，その中から必要な機能のみを選んでシステムに組み込む（いい換えると，使わない機能を取り外す）ことで，ポータビリティを確保しつつ，カーネルの使用リソースの削減が可能になる．ただし，カーネルを構成可能にすると，カーネルの検証性が悪くなるという問題がある．

μITRON4.0 仕様は，スタンダードプロファイルと呼ばれる規定により，カーネルが標準的に実装すべき API セットとその機能を厳密に規定し，ポータビリティを確保している．さらに，適応性向上のために，ライブラリリンクの機構を用いてカーネルを構成可能とすることを想定し，必要な機能のみを組み込むことが容易な API 仕様とすることを目指した．ライブラリリンクによる構成を想定したのは，この方式が検証性の面で利点が大きいことと，それゆえにこれまで開発された μITRON 仕様カーネルの多くがこの方式をとっているためである．必要な機能のみを組み込めるようにするためのアプローチとして，μITRON4.0 仕様では，以前のバージョンでの設計方針に加えて，カーネルには必要な機能を実現するための最小限のプリミティブのみを持たせるという設計方針をとった．この設計方針を「RISC 的なカーネル仕様」と呼ぶ．カーネルの構成可能性と検証性の問題ならびに μITRON4.0 仕様の設計方針については，2 章で詳しく議論する．

本論文の目的は，μITRON4.0 仕様の設計方針とそれに従って策定された仕様の有効性を検証することである．具体的には，RISC 的なカーネル仕様という設計方針が反映している機能として，μITRON4.0 仕様で新たに規定された例外処理機能とオーバーランハンドラ機能を取り上げ，それらの仕様について解説するとともに，μITRON4.0 仕様に準拠したカーネルを実装することで，それらが設計目標を達成しているかを評価する．

2. リアルタイムカーネルの構成可能性と μITRON4.0 仕様の設計方針

2.1 ライブラリリンクによる構成

前述のとおり，これまでに実装された μITRON 仕様カーネルの多くが，ライブラリリンクの機構を用いてカーネルを構成可能としている．具体的には，カーネルのオブジェクトファイルをライブラリの形で提供し，アプリケーションプログラムにリンクする☆．これにより，アプリケーションが用いているシステムコールを含むモジュールのみが，システムに組み込まれることになる．

ライブラリリンクによる構成は，アプリケーションからの参照情報に基づいて組み込むモジュールをリンカが自動的に決定するため，構成のための手間がかからない反面，リンカの機能のみを用いることによる限界がある．具体的には，静的な参照情報のみに基づいてリンクするモジュールを決定するため，実際には実行されることがないプログラムが組み込まれてしまう場合がある．また，1 つのモジュールに対して複数の実装を用意し，その中で最適なものを選択することはできない．

カーネルのソースコード中に条件コンパイル記述を入れる，カーネルをオブジェクト指向で設計しオブジェクト単位で構成するなど，カーネルをより小さい粒度で構成可能にする方法もある．ところが，小さい粒度での構成可能性は，カーネルの検証コストを増大させるという問題がある．たとえば，2 つの機能にそれぞれ 2 通りの実装が用意されており，それぞれ最適

☆ 多くの μITRON 仕様カーネルで，アプリケーションプログラムをカーネルと同一のメモリ空間・特権モードで実行し，システムコール呼び出しを通常のサブルーチンコールで実現している．

なものを選択できるとすると，その組合せで 4 通りのカーネルが構成できることになる．2 つの機能が関連している場合には，4 通りすべてについて検証を行わなければならない．そのためカーネルの検証コストは，構成可能なポイントの数に対して指数的に増加することになる．

それに対してライブラリリンクによる構成は，検証性の面で利点が大きい．これは，いくつかの例外的な状況を除いては，呼び出されない関数がリンクされているかどうかで，システムの動作が変化しないためである．そのため，参照していないモジュールをリンクしないという条件でカーネルの検証を行えば，あるモジュールをリンクしているがそこに含まれる機能を使わない場合の動作も検証したことになる[3]．

2.2 μITRON4.0 仕様の設計方針

前節で述べたように，ライブラリリンクによる構成と小さい粒度での構成には，一長一短がある．リアルタイム OS の開発の中で検証作業の占める割合が大きいことから，μITRON 仕様の設計にあたっては，ライブラリリンクによる構成を想定することとした．すなわち，カーネルがライブラリリンクによって構成可能となっていることを前提に，できる限り必要な機能のみがシステムに組み込まれるような API 仕様となるよう考慮した．小さい粒度での構成は，リアルタイム OS の API 仕様の設計にかかわらず可能であるため，特別な配慮はしていない．

他の標準化された OS API 仕様の中で，OSEK OS 仕様[4]はコンフォーマンスクラス規定やスケジューリングポリシーの選択によって，EL/IX[5]は機能のレベル化によって適応性を実現しているが，ライブラリリンクによる構成は考慮していない．そのため，構成可能な実装を行おうとすると，小さい粒度での構成をとるしかない．実際，OSEK OS については可能な組合せが多く，検証にコストがかかることが問題となっている．また，コンポーネントベースの構成可能な OS である MMLite[6]でも，検証性の問題は考えられていない．

以下では，ライブラリリンクによる構成が有効に働くように，μITRON4.0 仕様でとった設計方針について述べる．この中の最初の 2 つは，以前のバージョンから採用していた方針である[7]のに対して，3 つめの設計方針は μITRON4.0 仕様において新たに設定したものである．

- システムコールの単機能化

1 つのシステムコールには，なるべく 1 つの機能のみを持たせるような設計とする．1 つのシステムコールに複数の機能を持たせると，その中の一部の機能のみが必要な場合にも，そのシステムコールの持つすべての機能が組み込まれてしまう．

- 同期・通信オブジェクトのタスク独立化

セマフォやイベントフラグ，メールボックスなどのタスク間同期・通信のための機構を，タスクとは独立したオブジェクトとする．これにより，アプリケーションで使わないオブジェクトについては，オブジェクトを操作するためのシステムコールやオブジェクトを管理するためのデータ領域を，ライブラリリンクによる構成で取り外すことができる．

- RISC 的なカーネル仕様

スタンダードプロファイルに含まれる機能であっても，アプリケーションが使用しない機能は，機器への組み込み時には取り外すことができる．そのため，上の 2 つの設計方針により取り外せる機能については，スタンダードプロファイルに含めてもデメリットは少ない．

それに対して，スケジューラなどカーネルの必須部分に組み込まれるか，そこから呼び出されるコードは，ライブラリリンクによる構成では取り外すことができない．この場合，必要な機能を実現するための最小限のプリミティブのみをスタンダードプロファイルに含め，より複雑な機能が必要な場合には，カーネルの上で実現できるようにする．逆にその機能が不要な場合にも，カーネルに含まれるのは必要最小限の機能であるため，無駄に使われるリソースを小さくおさえることができる．

なお，カーネルには必要な機能を実現するための最小限のプリミティブのみを持たせるという方針を RISC 的なカーネル仕様と名付けたのは，必要な機能を実現するための最小限のプリミティブのみをハードウェアで実現するという RISC プロセッサの設計方針と類似しているためである．

RISC 的なリアルタイムカーネル仕様の設計方針は，いわゆるマイクロカーネルの考え方と類似しているが，何を必要最小限のプリミティブとするかが大きく異なる．具体的には，μITRON4.0 仕様では，アプリケーションのポータビリティを保つために，プロセッサの違いを隠蔽するためのプリミティブを必要最小限ととらえている．それに対して，Pebble[8]，Exokernel[9]，L4[10]などのマイクロカーネルでは，保護機能を実現するために，特権モードで実行しなければならないプリミティブを必要最小限としている．

RISC 的なリアルタイムカーネル仕様の設計方針は，

表 1 ターゲットプロセッサの仕様
Table 1 Specification of the target processors.

	MC68LC040	SH7709
アーキテクチャ	CISC 型	RISC 型
内部周波数	66 MHz	80 MHz
周辺モジュール周波数	33 MHz	40 MHz
キャッシュ容量	4 kbyte	8 kbyte

主に μITRON4.0 仕様で新たに標準化した機能の設計に反映されている．具体的には，CPU 例外ハンドラとタスク例外処理機能からなる例外処理機能と，タスクが定められたプロセッサ時間を使い切ったことを検出するためのオーバーランハンドラ機能は，この方針を考慮して設計した．

3. 評価環境および評価方法

この章では，評価のため実装を行ったターゲットハードウェアおよびカーネルと，評価項目について述べる．

3.1 ターゲットハードウェア

ターゲットハードウェアとして，プロセッサに M68040[11]（モトローラ製 MC68LC040）を用いた DVE68K/40（電産製）と，SH-3[12]（日立製 SH7709）を用いた MU200-RSH3（三菱電機マイコン機器ソフトウエア製）を用いた．プロセッサの仕様を**表 1** に示す．2 種類のターゲットを用いたのは，性能評価結果の一般性を高め，CISC 型と RISC 型のアーキテクチャによる実装方法と性能の違いを確認するためである．

3.2 カーネル

カーネルは我々の研究室で開発している μITRON4.0 仕様準拠の TOPPERS/JSP カーネル[13]（以下，JSP カーネル）を用いた．JSP カーネルは，スタンダードプロファイルに含まれる機能のみを実装している．スタンダードプロファイルには，今回評価する機能の 1 つであるオーバーランハンドラ機能は含まれていないため，JSP カーネルに追加機能として実装した．

JSP カーネルは，カーネルとアプリケーションを 1 つのロードモジュールにリンクする構成をとっている．また，タスクやセマフォなどのカーネルオブジェクトを，システム実行中に動的に生成する機能は持たない．カーネルオブジェクトの生成は，静的 API と呼ばれる記法により，生成に必要な情報をコンフィギュレーションファイルに記述することで行う．オブジェクトの初期化情報は，このファイルに記述された情報から，システム構成時に C 言語のソースファイルの形で生成する．

プロセッサの割込みおよび例外処理の方式は，今回評価を行う機能の実装方法に大きく影響する．M68040 は，割込みや例外が発生するとベクタテーブルを参照して個別のハンドラを実行する．それに対して，SH-3 は，割込み発生時はベクタレジスタ（以下，VBR）+0x600 番地から，例外発生時は VBR+0x100 番地から☆実行する．割込みまたは例外の発生要因の判定は，要因レジスタにセットされる要因コードにより行い，対応するハンドラへの分岐はソフトウェアで行う必要がある．

3.3 評価項目

評価項目として，まずそれぞれの機能の実行時間と，その実装に必要なメモリサイズを評価する．

RISC 的なカーネル仕様という設計方針は，機能が不要な場合にも，無駄に使われるリソースを小さくおさえることを狙ったものであった．そこで，カーネルに機能を実装することで，それぞれの機能を使用しない場合の実行時間と使用メモリサイズがどの程度増加するを評価する．

また，例外処理機能については，複雑な例外処理を実現するためのプリミティブとして十分であるかについても評価する．

実行時間については，タスクディスパッチ時間と割込み処理時間が，機能を実装しない場合と比べてどれだけ長くなるかを測定する．タスクディスパッチ時間は，同一優先度のタスクを 2 つ用意し，一方のタスクからその優先度のレディーキューを回転させるシステムコールを発行後，ディスパッチ処理が行われてもう一方のタスクの実行が始まるまでの時間を測定する．割込み処理時間は，カーネルに 1 msec ごとに入るシステムタイマ割込みハンドラの処理時間☆☆を測定する．具体的には，システムタイマ割込みしか発生しない状況で，短い処理の実行時間を計測すると，途中でシステムタイマ割込みが入る場合があるために，処理時間の分布が大きく 2 つに分かれる．その差が，割込みハンドラの処理時間である．

実行時間の測定は，測定する処理の前後でハードウェアタイマを参照することで行う．ハードウェアタイマの参照にかかるオーバヘッドは事前に測定しておき，測定結果から減ずる．なお，ハードウェアタイマのカウント周期は，DVE68K/40 では 1 μsec，MU200-RSH3 では 0.8 μsec である．実際的な最悪値を求めるために，キャッシュは有効とし☆☆☆，測定前にキャッシュをパージする．

☆ TLB ミス発生時は VBR+0x400 番地から実行するが，JSP カーネルはサポートしていない．
☆☆ システムタイマ割込みハンドラがカーネルのタイムティックの更新のみを行う場合．
☆☆☆ SH-3 はライトスルーモード．

表 2 JSP カーネルの実行時間
Table 2 Execution times of JSP Kernel.

プロセッサ	タスクディスパッチ時間	割込み処理時間
M68040	20 μsec	16 μsec
SH-3	14 μsec	11 μsec

表 3 JSP カーネルのメモリサイズ
Table 3 Memory sizes of JSP Kernel.

プロセッサ	プログラム	ROM データ	RAM データ
M68040	16072 byte	430 byte	1232 byte
SH-3	20538 byte	545 byte	1932 byte

オリジナルの JSP カーネルのタスクディスパッチ時間と割込み処理時間を，**表 2** に示す．

メモリサイズについては，機能の実装に必要なメモリサイズを，ライブラリリンクによる構成で取り外せる部分と取り外せない部分に分けて示す．取り外せない部分のメモリサイズとは，カーネルの必須モジュールの変更による使用メモリサイズの増加分である．JSP カーネルでは，タスクディスパッチャ，割込みハンドラおよび例外ハンドラの出入口処理，タスクごとに必要な制御ブロックなどが，取り外せない部分に該当する．タスクごとの制御ブロックは，値が変化しない部分をタスク初期化ブロック（以下，TIB）として ROM に置き，RAM に置くタスク制御ブロック（以下，TCB）と分離している．オリジナルの JSP カーネルでの TIB と TCB のサイズは，ともに 8 word である．一方，システムコールの処理ルーチンは，ライブラリリンクによる構成で取り外すことができる．

表 3 に，例外処理機能を含むすべてのモジュールをリンクしたときに，オリジナルの JSP カーネルが固定的に使用する（いい換えると，カーネルオブジェクトを生成するごとに必要な分を除く）メモリサイズを示す．

4. 例外処理機能とその評価

この章では，μITRON4.0 仕様の例外処理機能の仕様と特徴，JSP カーネルにおける実装方法を述べ，3.3 節で述べた各項目について評価する．また，この機能がプリミティブとして十分であるかを評価するため，標準化された API のみで複雑な例外処理機能を実現できることを示す．

4.1 仕様と特徴

μITRON4.0 仕様では，例外処理のための機能を，CPU 例外ハンドラ機能とタスク例外処理機能に分けて提供している．

4.1.1 CPU 例外ハンドラ機能

プロセッサが例外を検出した場合，カーネルは例外の種類に応じてシステムに登録された CPU 例外ハンドラを起動する．1 つの例外要因に対する CPU 例外ハンドラはシステムで 1 つである．CPU 例外ハンドラ機能の API として，スタンダードプロファイルでは，それぞれの例外要因に対して CPU 例外ハンドラを登録する静的 API のみが用意されている（**表 4**）．

表 4 CPU 例外ハンドラ機能の API
Table 4 API of CPU exception handler functions.

API	説明
DEF_EXC	CPU 例外ハンドラの登録（静的 API）

表 5 タスク例外処理機能の API
Table 5 API of task exception handling functions.

API	説明
DEF_TEX	タスク例外処理ハンドラの登録（静的 API）
ras_tex	タスク例外処理の要求
iras_tex	同上（非タスクコンテキスト用）
dis_tex	タスク例外処理の禁止
ena_tex	タスク例外処理の許可
sns_tex	タスク例外処理禁止状態の参照

4.1.2 タスク例外処理機能

タスク例外処理機能は，POSIX のシグナル機能[14]に相当する機能である．タスクに対してタスク例外処理が要求されると，次にそのタスクがスケジュールされたときに，そのタスクのコンテキスト内でタスク例外処理ルーチンが起動される．POSIX のシグナルハンドラと異なり，タスク例外処理ルーチンは，各タスクに対して 1 つのみ登録できる．

発生した例外の種類を判別するために，タスク例外処理ルーチンに対しては，タスク例外処理が要求された際に指定されたタスク例外要因が渡される．タスク例外処理ルーチンが起動される前にタスク例外処理が複数回要求された場合には，それらのタスク例外要因のビットごとの論理和をとった値が渡される．カーネルは，タスクごとに，要求されたがまだ処理されていないタスク例外要因を管理する．これを保留例外要因と呼ぶ．

タスク例外処理ルーチンが多重に起動されるのを防ぐために，タスク例外処理ルーチンの起動時にタスク例外処理禁止状態とし，リターン時にタスク例外処理許可状態に戻す．また，タスク例外処理禁止および許可状態へ移行させるシステムコールが用意されている．スタンダードプロファイルに含まれるタスク例外処理機能の API の一覧を**表 5** に示す．

4.1.3 μITRON4.0 仕様の例外処理機能の特徴

μITRON4.0 仕様の例外処理機能の特徴を，POSIX のシグナル機能と比較して説明する．

POSIX のシグナル機能では，プロセッサが何らかの例外条件（たとえば，ゼロ除算や不正アドレスへのアクセス）を検出した場合，例外を発生させたプロセスのシグナルハンドラが起動される．組込みシステムでは，例外が発生した場合に，原因となったタスクによらずに同じ処理（たとえばシステムをリセットする）を行う場合も多い．POSIX のシグナル機能では，そのような場合にもすべてのプロセスに対して個別にシグナルハンドラを登録しなければならず，無駄が大きい．また，このような場合にも，プロセスにシグナルを通知する処理を取り外すことはできない仕様になっている．

μITRON4.0 仕様の例外処理機能では，プロセッサが検出した例外を処理するための CPU 例外ハンドラ機能と，タスクに例外的な条件を通知するためのタスク例外処理機能を明確に分離している☆．そのため，たとえば，プロセッサがゼロ除算を検出した場合にシステムをリセットしたいのであれば，アプリケーションで用意するゼロ除算を処理する CPU 例外ハンドラでシステムをリセットする処理を行えばよい．また，ゼロ除算をタスクに例外の形で通知したい場合には，CPU 例外ハンドラでタスク例外処理を要求すればよい．このように 2 つの機能を分離することで，アプリケーションシステムに必要な機能だけを組み込むことが可能になる．

前述のとおり，μITRON4.0 仕様のタスク例外処理機能では，各タスクに対して登録できるタスク例外処理ルーチンは 1 つである．これは，POSIX のシグナル機能が 1 つのプロセスに対して例外要因ごとにハンドラが登録できるのと比較して，必要最小限の機能のみをカーネルに持たせているという意味で，RISC 的なカーネル仕様の例となっている☆☆．また，POSIX のシグナル機能では，例外要因ごとにマスクできるのに対して，タスク例外処理機能では，例外処理の禁止か許可の 2 状態しか持たない．後に 4.3.3 項で，このように簡略化されたタスク例外処理機能の上に，POSIX のシグナル機能が実現できることを述べる．

4.2 実装方法

4.2.1 CPU 例外ハンドラ機能

プロセッサが例外を検出すると，その時点のコンテキストを保存し，非タスクコンテキストへと切り替えた後に，検出した例外に対する CPU 例外ハンドラを呼び出す．CPU 例外ハンドラから戻ると，コンテキストを元に戻す．また，CPU 例外ハンドラ実行中にタスクディスパッチが要求される場合があるため，CPU 例外ハンドラの出口ではディスパッチが必要か調べ，必要であればタスクディスパッチャを呼び出す．これらの処理を行うルーチンを，出入口処理ルーチンと呼ぶ．

M68040 では，出入口処理ルーチンをマクロとして定義し，コンフィグレーションファイルに記述された CPU 例外ハンドラの数だけ展開する．また，CPU 例外ハンドラの先頭番地をベクタテーブルに登録する．

一方 SH-3 では，VBR+0x100 番地に出入口処理ルーチンを置く．また，例外の要因コードをオフセットとして，CPU 例外ハンドラの先頭番地を登録したテーブルを用意する．

4.2.2 タスク例外処理機能

タスク例外処理機能を実現するために，タスク例外処理ルーチンの属性と先頭番地を TIB に，タスク例外処理許可状態と保留例外要因を TCB に置く．

タスク例外処理ルーチンの起動は，対象のタスクが実行状態である，タスク例外処理許可状態である，保留例外要因がある，非タスクコンテキストを実行していないという 4 つの起動条件が揃ったときに行う．そのため，以下のタイミングで，タスク例外処理ルーチンの起動を行う必要がある．

(1) タスクディスパッチの直後
新しく実行状態になったタスクが起動条件を満たしている場合．

(2) タスク例外処理の許可
タスク例外処理許可状態に移行した結果，自タスクが起動条件を満たす場合．

(3) 自タスクに対するタスク例外処理の要求
自タスクに対してタスク例外処理を要求した結果，自タスクが起動条件を満たす場合．

(4) ハンドラの出口
割込みハンドラおよび CPU 例外ハンドラ（ここではハンドラと総称する）内で，実行状態のタスクに対してタスク例外処理が要求され，戻り先のタスクが起動条件を満す場合．

(1) より，タスクディスパッチャに，タスク例外処理ルーチンの起動処理を含める必要がある．また (4) より，ハンドラの出口処理ルーチンにもタスク例外処理ルーチンの起動処理が必要である．ハンドラの出口処理ではタスクディスパッチが必要となる場合もあるため，ディスパッチ要求を示すフラグとタスク例外

☆ 分離が可能なのは，μITRON4.0 ではアプリケーションプログラムにシステム特権レベルでのプログラミングを許可しているためである．

☆☆ 実際，μITRON4.0 仕様のタスク例外処理機能と POSIX のシグナル機能の違いは，RISC 型プロセッサと CISC 型のプロセッサの割込み処理機能の違いに類似している．

表 6 例外処理の起動オーバヘッド
Table 6 Overhead of exception handlings.

プロセッサ	CPU 例外ハンドラ	タスク例外処理ルーチン
M68040	8 μsec	3 μsec
SH-3	6 μsec	3 μsec

処理ルーチンの起動要求を示すフラグを共通化し，いずれかが要求されていた場合に，それらの処理を行うルーチンに分岐するようにした．これらのルーチンは，カーネルの必須モジュールであり，ライブラリリンクによる構成で取り外すことはできない．

(2) と (3) の起動タイミングは，それぞれタスク例外処理許可状態にするシステムコールとタスク例外処理を要求するシステムコール内で発生するため，タスク例外処理ルーチンの起動処理は，これらのシステムコール処理ルーチンに含める．システムコール処理ルーチンは，そのシステムコールを使わない場合には，ライブラリリンクによる構成で取り外される．

4.3 評価結果

4.3.1 実行時間の評価

CPU 例外ハンドラ機能とタスク例外処理機能の実行時間として，例外が発生した後 CPU 例外ハンドラの実行が始まるまでの時間と，タスクディスパッチ後タスク例外処理ルーチンの実行が始まるまでの時間を**表 6** に示す．

タスク例外処理機能を実装することによる，この機能を使用しない場合の実行時間の増加は微少（実行時間の測定精度である 1 μsec 未満）である．具体的には，タスクディスパッチャとハンドラの出口処理の実行時間のいずれも，変数を参照して条件分岐する処理（M68040 で 2 命令，SH-3 で 3 命令）の実行時間分長くなる．CPU 例外ハンドラ機能の実装により，これらの実行時間は変化しない．

以上より，例外処理機能を使用しない場合の実行時間の増加は十分小さく，最小限のオーバヘッドで例外処理機能を実現できている．また，これらの機能自身の実行時間も十分高速であった．

4.3.2 メモリサイズの評価

CPU 例外ハンドラ機能の実装によるメモリサイズの増加量を**表 7** に示す．表中の n は，システムに登録される CPU 例外ハンドラの数である．これらはすべて，CPU 例外ハンドラ機能を使用しない場合には，ライブラリンクによる構成により取り外すことができる．

M68040 では，CPU 例外ハンドラの数だけ出入口処理ルーチンのマクロを展開するため，CPU 例外ハンドラの数に応じてプログラムサイズが増加する．また，CPU 例外ハンドラごとに，その登録情報を置くために ROM データが 12 byte 増加する．なお SH-3 では，CPU 例外ハンドラの先頭番地を登録するテーブルを，RAM に置いている．

表 7 CPU 例外ハンドラ機能によるメモリサイズの増加量
Table 7 Increased memory sizes with CPU exception handler function.

プロセッサ	プログラム	ROM	RAM
M68040	212+56·n byte	4+12·n byte	
SH-3	304 byte	4+12·n byte	64 byte

表 8 タスク例外処理機能によるプログラムサイズの増加量
Table 8 Increased program sizes with task exception handling function.

プロセッサ	取り外せる部分	取り外せない部分
M68040	604 byte	152 byte
SH-3	708 byte	124 byte

表 9 タスク例外処理機能によるタスクごとのデータサイズの増加量
Table 9 Increased data sizes for each task with task exception handling function.

名称	TIB	TCB
タスク例外処理ルーチン属性	1 word	
タスク例外処理ルーチン先頭番地	1 word	
タスク例外処理許可状態		1 bit
保留例外要因		1 word

表 8 には，タスク例外処理機能の実装によるプログラムサイズの増加量を，ライブラリリンクによる構成で取り外せる部分と取り外せない部分に分けて示す．取り外せない主な部分は，タスクディスパッチャとハンドラの出口処理に含まれるタスク例外処理ルーチンを起動する処理である．

タスク例外処理機能の実装による，タスクごとのデータサイズの増加量を**表 9** に示す．タスク例外処理機能は，TIB の 25%，TCB の 13%を使用している．これらは，ライブラリリンクによる構成で取り外すことはできない．

以上より，例外処理機能の実装によるプログラムサイズの増加のうち，ライブラリリンクによる構成で取り外せない部分はきわめて小さく，それらの機能を使わない場合にも許容できるサイズである．それに対して，タスク例外処理機能によるデータサイズの増加率はやや大きいが，絶対的なサイズは小さく，アプリケーションにもよるが，許容範囲内と考えられる．

4.3.3 POSIX のシグナル機能の実現

μITRON4.0 の例外処理機能が，より複雑な例外処理を実現するのに十分なプリミティブであることを確認するために，標準化された API のみを用いて，POSIX のシグナル機能を実現した[15]．

シグナルの種類は，タスク例外要因により通知する．また，シグナルごとのマスクは，次のように実現できる．タスク例外処理機能では，タスク例外要因ごとにタスク例外処理を禁止/許可することはできないため，クリティカルセクションを除いてつねにタスク例外処理許可状態とする．マスクされているシグナルに対してタスク例外処理ルーチンが起動された場合には，シグナル処理を保留して，すぐにリターンする．保留されたシグナル処理は，マスクが解除された時点で行う．

ここで，POSIX のシグナル機能を実現するプログラムは，標準化された API のみを用いているため，異なるプロセッサへのポータビリティが確保されていることに注意されたい．POSIX のシグナル機能には含まれないが，BSD UNIX などが持っている `sigstack` システムコールは，標準化された API のみで実現することはできない．スタックを直接操作して実現することは可能だが，この方法では異なるプロセッサへのポータビリティが確保されない．すなわち，`sigstack` システムコールを実現するためには，μITRON4.0 仕様の例外処理機能は，プリミティブとして不十分であることになる．

このことから，μITRON4.0 仕様の例外処理機能は，POSIX のシグナル機能程度の例外処理を実現するのに十分なプリミティブを提供しているといえる．

5. オーバーランハンドラ機能とその評価

この章では，μITRON4.0 仕様のオーバーランハンドラ機能の仕様と特徴，JSP カーネルにおける実装方法を述べ，3.3 節で述べた項目について評価する．また，本論文の主目的とは外れるが，プロセッサ時間の計測精度についても評価する．

5.1 仕様と特徴

μITRON4.0 仕様を，多様な組込みシステムに幅広く適用できるものとするためには，非周期タスクを実行するための各種のサーバアルゴリズム[16]や，プロセッサ時間の保護機能[17]が利用できることが望まれる．ところが，これらの機能はいずれも，複雑であるうえにバリエーションが多く，標準的な仕様に取り込むのは難しい．そこで，これらの機能を実現するための必要最小限のプリミティブとして，オーバーランハンドラ機能を導入した．この意味で，RISC 的なカーネル仕様の 1 つの例となっている．

オーバーランハンドラ機能は，各タスクが使用したプロセッサ時間を計測し，タスクが設定された時間を超えて実行されると，あらかじめ登録されたオーバーランハンドラを実行する機能である．

表 10 オーバーランハンドラ機能の API
Table 10 API of overrun handler functions.

API	説明
`DEF_OVR`	オーバーランハンドラの登録（静的 API）
`sta_ovr`	上限プロセッサ時間の設定
`stp_ovr`	上限プロセッサ時間の設定解除

タスクが使用できる総プロセッサ時間の上限値を上限プロセッサ時間と呼び，タスクごとに設定できる．カーネルは，上限プロセッサ時間が設定されたタスクについて，使用したプロセッサ時間を計測し，その累積値が上限プロセッサ時間を超えるとオーバーランハンドラを起動する．オーバーランハンドラの主な API を**表 10** に示す．

カーネルのデータサイズを小さくするために，システムに登録可能なオーバーランハンドラは 1 つだけとしている．プロセッサ時間を使いきったタスクを判別するために，オーバーランハンドラにはそのタスクの ID 番号を渡す．プロセッサ時間を使いきったときの処理をタスクごとに行いたい場合は，タスク例外処理機能を併用すればよい．この点でも，提供するプリミティブを最小限のものとすることを意識している．

5.2 実装方法

タスクのプロセッサ使用時間の計測は，ハードウェアタイマにより行い，タイムアウト割込みにより起動される割込みハンドラから，設定されたオーバーランハンドラを起動する．ここでは，このハードウェアタイマを単にタイマと呼ぶ．上限プロセッサ時間の設定の有無と使用プロセッサ時間の累積値は，TCB に記憶する．

5.2.1 ハンドラの入口処理

割込みハンドラおよび例外ハンドラ（以下，ハンドラと総称する）の処理時間は，タスクの使用プロセッサ時間に含めないため，ハンドラの入口処理でタイマを停止する．その際，ハンドラ起動前に実行中のタスクに対して上限プロセッサ時間が設定されていなければ，タイマを停止する必要はない．しかし，上限プロセッサ時間の設定の有無の確認後にタイマを停止すると，オーバヘッドや後述の計測誤差が増加する．そこで，停止しているタイマを再度停止させても問題はないため，無条件にタイマを停止する．

5.2.2 ハンドラの出口処理

ハンドラの出口処理では，ハンドラの実行中にディスパッチが要求された場合には，次に述べるタスクディスパッチャを呼び出し，後述の処理を行う．ディスパッチが必要ない場合には，戻り先のタスクに上限プロセッサ時間が設定されている場合のみ，タイマを

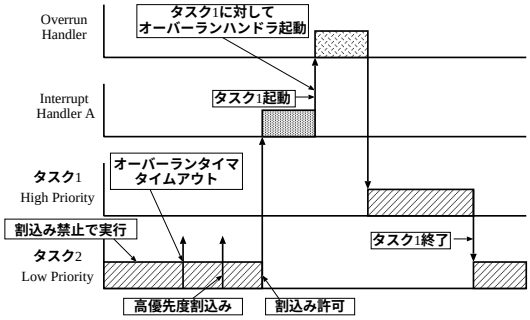

図 1 高優先度の割込みが存在する場合の問題
Fig. 1 A problem with a higher priority interrupt handler.

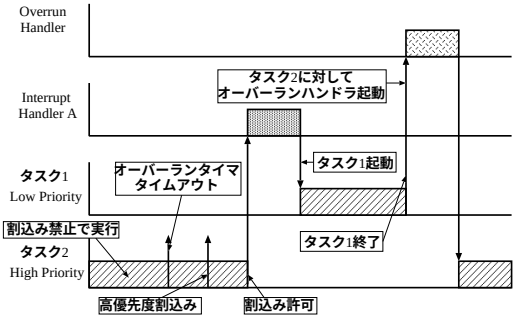

図 2 正しい処理
Fig. 2 Correct behavior.

スタートさせる．

5.2.3 タスクディスパッチャ

タスクディスパッチャでは，ディスパッチ元のタスクのコンテキストを保存する際に，前述したのと同じ理由で無条件にタイマを停止する．その後，そのタスクに上限プロセッサ時間が設定されていれば，タイマのカウント値を使用プロセッサ時間の累積値としてTCB に保存する．また，ディスパッチ先のタスクのコンテキストを復帰する際に，そのタスクに上限プロセッサ時間が設定されていれば，TCB 中の使用プロセッサ時間の累積値をタイマのカウンタに設定し，タイマをスタートする．

5.2.4 問 題 点

ハードウェアタイマによる割込みでオーバーランハンドラを起動している関係から，上記の処理だけでは次の 2 つの問題が発生する．

1 つめの問題は，タイマの割込みより優先度の高い割込みが存在する場合に発生する．**図 1** で，タスク 2 には上限プロセッサ時間が設定されており，割込み禁止状態で実行しているものとする．この間にタイマによる割込みと，それより優先度の高い割込みの両方が発生した場合を考える．タスク 2 が割込みを許可すると，まず，後者の割込みを処理するハンドラ A が実行される．ハンドラ A の中でタスク 1 が起動されると，ハンドラ A の出口処理でタスクディスパッチを行う．タスクディスパッチは割込みを禁止して行うため，タイマの割込みは，タスク 1 の実行が始まるまで受け付けられない．その結果，タスク 2 に対するオーバーランハンドラの起動要求が，タスク 1 の実行中に発生する．

同様の問題は，タスクから呼び出されたシステムコール中で，タスクディスパッチを行う場合にも発生する．これは，タスクディスパッチャが割込み禁止状態で呼ばれるために，タスクディスパッチャがタイマを停止するまでの間に，タイマがタイムアウトする可能性があるためである．

これらの問題を回避するためには，タイマを停止させた後，カウント値を TCB に保存する前に，タイマによる割込み要求が入っているかを調べ，要求があればそれをクリアする．また，TCB にはタイムアウト直前のカウント値を保存する．これにより図 1 の状況では，次にタスク 2 が実行された直後に，タスク 2 に対してオーバーランハンドラが起動される（**図 2**）．

表 11 上限プロセッサ時間設定時の実行時間の増加
Table 11 Increased execution times when limit processor time is set.

プロセッサ	タスクディスパッチ時間	割込み処理時間
M68040	6 μsec（30%）	2 μsec（13%）
SH-3	4 μsec（25%）	3 μsec（27%）

表 12 上限プロセッサ時間未設定時の実行時間の増加
Table 12 Increased execution times when limit processor time is not set.

プロセッサ	タスクディスパッチ時間	割込み処理時間
M68040	2 μsec（5%）	1 μsec（6%）
SH-3	1 μsec（7%）	2 μsec（18%）

5.3 評 価 結 果

5.3.1 実行時間の評価

オーバーランハンドラ機能の実行時間として，タスクに上限プロセッサ時間を設定した場合のタスクディスパッチ時間と割込み処理時間の増加を，**表 11** に示す．

また，タスクに上限プロセッサ時間を設定しない場合のこれらの時間の増加を**表 12** に示す．この表より，オーバーランハンドラ機能を使用しない場合には，オーバーランハンドラの実装によるタスクディスパッチ時間の増加は 10%以下である．SH-3 の割込み処理時間の増加が 18%と大きくなった理由は，後述のハン

表 13 オーバーランハンドラ機能によるプログラムサイズの増加量
Table 13 Increased program sizes with overrun handler function.

プロセッサ	取り外せる部分	取り外せない部分
M68040	1062 + 36·n byte	46 byte
SH-3	1162 byte	78 byte

表 14 オーバーランハンドラ機能によるタスクごとのデータサイズの増加量
Table 14 Increased data sizes for each task with overrun handler function.

名称	TCB
オーバーランハンドラの動作状態	1 bit
累積プロセッサ時間	1 word

ドラの実行による計測誤差を小さくするためにハンドラの出口処理を大きく変更したためであり，計測精度とのトレードオフとなっている．

以上より，オーバーランハンドラ機能を使用しない場合の実行時間の増加は，増加率としてはやや大きい場合もあるが，絶対値としては十分に小さく，設計目標を達成しているといえる．また，オーバーランハンドラ機能を使用した場合の実行時間も，十分に実用的な速度であると考えられる．

5.3.2 メモリサイズの評価

表 13 には，オーバーランハンドラ機能の実装によるプログラムサイズの増加量を，ライブラリリンクによる構成で取り外せる部分と取り外せない部分に分けて示す．表中の n は，システムに登録した割込みハンドラおよび CPU 例外ハンドラの総数である．M68040 では，割込みハンドラの出入口処理ルーチンも，CPU 例外ハンドラと同様に展開するため，プログラムサイズの増加量が，割込みハンドラおよび CPU 例外ハンドラの総数に応じて増加する．

オーバーランハンドラ機能の実装による，タスクごとのデータサイズの増加量を**表 14** に示す．オーバーランハンドラ機能により，TCB のサイズは 13%増加する．

以上より，オーバーランハンドラ機能の実装によるプログラムサイズの増加のうち，ライブラリリンクによる構成で取り外せない部分はきわめて小さく，機能を使わない場合にも許容できるサイズである．また，データサイズの増加もわずかで，許容範囲内といえる．

5.3.3 計測精度

上限プロセッサ時間が設定されているタスクの実行中に，割込みや例外が発生すると，ハンドラの入口処理でタイマを停止するまでの時間と，出口処理でタイマをスタートさせてからタスクの実行を再開するまでの時間は，タスクが実行されていないにもかかわらず，使用プロセッサ時間に算入される．本論文の主目的とは外れるが，以下ではこの計測誤差の測定について述べる．

具体的な測定方法は次のとおりである．上限プロセッサ時間を設定したタスクにおいて，変数をインクリメントし続ける処理を行う．1 回のインクリメントに必要な時間は事前に調べておく．このタスクに対してオーバーランハンドラが呼び出されるまでの変数のインクリメント回数を，オーバーランハンドラを起動するハードウェアタイマ以外の割込みを禁止した状態と，すべての割込みを許可した状態で測定する．割込みを許可した場合，1 msec ごとのシステムタイマ割込みハンドラによる計測誤差により，割込みを禁止した場合と比較して，オーバーランハンドラが呼び出されるまでのインクリメント回数が減少する．その減少分を割込みの回数で割ることにより，計測誤差を求めることができる．

測定の結果，計測誤差は M68040 で 4.5 μsec，SH-3 で 0.6 μsec となった．どちらのプロセッサでも，ハンドラの入口処理の最初でタイマを停止し，タスクの実行を再開する直前にスタートさせている．それにもかかわらず，計測誤差が 1 桁異なる値となったのは，プロセッサアーキテクチャの違いが原因であると考えられる．M68040 は，割込みが発生すると，割込みベクタの取得，ベクタテーブルの読み出し，プログラムカウンタなどのメモリへの保存といった処理を，ハードウェアで行う．逆に割込み処理の終了時には，プログラムカウンタなどの復帰処理を行う．これが，μsec オーダの計測誤差につながっている．それに対して SH-3 は，最小限のレジスタを専用の退避レジスタに保存し，固定番地より実行するため，メモリアクセスがいっさい行われない．そのため，M68040 と比較して計測誤差が小さくなる．

計測誤差が一定であれば補正も可能であるが，計測誤差はキャッシュの状態で変化するため，完全な補正は不可能である．また，補正を行うことにより，オーバヘッドが増加するという問題もある．

6. ま と め

本論文では，アプリケーションソフトウェアのポータビリティを確保しつつ，多様な組込みシステムへの適応性を向上させるために，μITRON4.0 仕様で採用したアプローチについて述べ，その有効性を評価した．

μITRON4.0 仕様では，ライブラリリンクによる構成を想定して，必要な機能をポータブルに実現するた

めの最小限のプリミティブを提供するというRISC的なカーネル仕様という設計方針をとった．そして，この設計方針に従って設計された例外処理機能とオーバーランハンドラ機能の仕様とその実装方法について述べ，それらの実行時間やメモリサイズを測定した．その結果，いずれの機能も十分小さいオーバヘッドで実現できること，それらの機能が不要な場合にも無駄に使われるリソースはきわめて小さいこと，それらの機能を使ってより複雑な処理が実現できることを示した．これらの結果は，これらの機能のAPI仕様に，RISC的なカーネル仕様という設計方針が有効に働いていることを意味しており，仕様が設計目標を達成していることを確認できた．

謝辞 μITRON4.0仕様の策定作業にご尽力下さったμITRON4.0仕様研究会カーネル仕様検討WGのメンバ各位に感謝します．

参 考 文 献

1) 高田広章，田丸喜一郎：ITRONサブプロジェクト第2フェーズへの展開，情報処理，Vol.40, No.3, pp.223–228 (1999).
2) 坂村　健（監修），高田広章（編）：μITRON4.0仕様 Ver.4.00.00，トロン協会 (1999).
3) 高田広章：構成可能なRTOSとμITRON仕様におけるアプローチ，ソフトウェアシンポジウム2000論文集，pp.49–54 (2000).
4) OSEK/VDX: OSEK/VDX Operating system, version 2.0 revision 1 (1997).
5) Garnett, N.: EL/IX Base API Specification DRAFT–V1.2 (2000). Available from `http://sources.redhat.com/elix/api/current/api.pdf`.
6) J., H. and A., F.: MMLite: A Highly Componentized System Architecture, *Proc. 8th ACM SIGOPS European Workshop*. Sintra, Portugal (1998).
7) 高田広章，坂村　健：μITRON仕様における適応性と標準化，信学技報（1994年実時間処理に関するワークショップ），Vol.93, No.516, pp.71–78 (1994).
8) Gabber, E., Bruno, J., Brustoloni, J., Silberschatz, A. and Small, C.: The Pebble component-based operating system, *Proc. 1999 USENIX Technical Conference*, pp.55–65 (1999).
9) Engler, D., Kaashoek, M. and Jr, J.: Exokernel: an operating system architecture for application-specific resource management, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp.251–266 (1995).
10) Hartig, H., Hohmuth, M., Liedtke, J., Schonberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *Proc.16th ACM Symposium on Operating Systems Principles*, pp.66–77 (1997).
11) 日本モトローラ：M68040ユーザーズマニュアル，1st edition, 日本モトローラ半導体事業部 (1992).
12) 日立製作所：SH7709ハードウェアマニュアル，4th edition, 日立製作所半導体グループ電子統括営業本部 (1998).
13) TOPPERS/JSP Projectホームページ．`http://www.ertl.ics.tut.ac.jp/TOPPERS/`.
14) IEEE 1003.1-1990: *POSIX-Part 1: System Application Program Interface (API) [C Language]*, IEEE (1990).
15) 前田三希緒：μITRON4.0仕様のタスク例外処理機能の実装と評価，修士論文，豊橋技術科学大学情報工学系 (2000).
16) Buttazzo, G.C.: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers (1997).
17) 安田吉幸，今井和彦，高田広章：保護機構を持ったRTOS，SWEST1予稿集，pp.31–32 (1999).

(平成12年12月18日受付)
(平成13年 4月 6日採録)

本田　晋也

2000年豊橋技術科学大学工学部情報工学課程卒業．現在，同大学院情報工学専攻に在学中．リアルタイムOS，ソフトウェア/ハードウェアコデザインの研究に従事．

高田　広章（正会員）

豊橋技術科学大学情報工学系講師．1988年東京大学大学院理学系研究科情報科学専攻修士課程修了．同学科の助手等を経て，1997年12月より現職．リアルタイムOS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．ITRON仕様の標準化活動に，中心的メンバとして参加．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．