組み込みシステムに適したQoS制御方式 ——QoSテーブル方式の提案と実装

菅原智義† 高野陽介†

この論文では，組み込みシステムに適した新しいQoS制御方式，「QoSテーブル方式」を提案する．本方式は，制御系とマルチメディア系のタスクが混在するような組み込みシステムを対象にしたものである．組み込みシステムではタスクが要求する資源量は予測可能であることが，多くの場合に仮定できる．本方式では，この点に注目して，タスクに割り当てる資源量をテーブルに記載しておき，それに基づいて資源配分を決定する．本方式はQoS制御の手順が簡単であり，高速にQoS制御できるという特長を持つ．また，テーブルを使うことにより，さまざまなQoS制御方針を実現することが可能である．本論文では，QoSテーブル方式をCPU時間に適用したQoS制御機構の設計と，μITRON3.0準拠のRTOS上への実装と性能評価について述べる．

Table-based QoS Control for Embedded Systems

Tomoyoshi Sugawara† and Yosuke Takano†

This paper proposes a new QoS control scheme that is suited for embedded real-time systems. Our scheme focused on the real-time system which both device control and multimedia processing are required. The scheme uses a table which describes resource requirements of all tasks. Resource allocations for tasks are determined by the table. The scheme has advantages that the process of QoS control is very simple and fast. Various QoS control policies can be implemented using a QoS table. This paper also presents implementation and evaluation of the scheme.

1. はじめに

ここ数年，マルチメディア機能を重視した小型の組み込みシステムがにわかに脚光を浴びている．その一例がカーナビゲーションシステムを中心とした車載組み込みシステムや，ディスク装置を使ったデジタルビデオレコーダ（DVR）である．従来の組み込みシステムとは異なり，これらのシステムには制御系とマルチメディア系という性質の異なるタスクが共存している点が特徴である．このようなシステムは，失敗が許されない制御系タスクに関しては最悪時でも実行を保証しつつ，処理量が大きく変動するマルチメディア系のタスクもできるだけ良い品質で実行できることが望ましい．たとえば，車載組み込みシステムでは走行距離センサやGPSからのデータを処理するタスクを優先的に実行しながら，地図の表示や音声対話などの処理を行うタスクはできるかぎり良いサービスを提供できるべきである．また，DVRでは特殊再生やエラーリトライなどによりディスク装置の入出力量が変動する状況でも，録画ストリームの品質を保つことが求められる．これらの要求を満たすためには，CPU時間やディスク装置のバンド幅の利用量に関して品質の保証，すなわちQoS（Quality of Service）の保証をする必要がある．

QoS制御は，資源利用における「QoSの保証」を実現する機能であり，ネットワーク，CPU時間，ディスク装置など資源としてとらえられるものに対して広く適用することができる．しかし，これまでは分散マルチメディアシステムのネットワークとCPU時間を対象にしたQoS制御方式の研究が中心であり，小型の組み込みシステムに適用できるようなQoS制御方式は存在しなかった．

分散マルチメディアシステムでは，一般に不特定多数のユーザが利用することを考慮してさまざまな機能が設計されており，そこで採用されているQoS制御方式も同様の特徴を持つ．まず第1に，資源割当てのプロセスはすべてネゴシエーションにより行わなけれ

† NECインキュベーションセンター
Incubation Center, NEC Corporation

ばならない．これは，システムがタスクが要求する資源量に関してほとんど予備知識も持たないからである．第2に，タスク間の資源割当ての優先順序を指定するための機能がなく，QoSを変更されるタスクの選定はユーザプログラムによって調停されることを前提としているものがほとんどである．これは，タスクが資源割当てに関して異なる優先度を持つことを考慮していないからある．第3に，タスクごとの資源割当て量の遵守が強要され，割当て量を超えて資源を利用することは原則的には許されていない．これは，予期しない資源不足の発生を防ぐためである．

一方，組み込みシステムでは，ユーザ数は固定で数人程度が普通であり，実行されるタスクの資源要求量も設計時に見積もれることが，多くの場合に仮定できる．また，資源割当ての優先度もタスクの優先度同様，設計時に明示的に指定できなくてはならない．さらに，場合によっては優先度の高いタスクが割当て量を超えて資源を利用することも許す必要がある．資源割当て量も資源割当ての優先度も明らかであれば，資源割当てのプロセスにおいてネゴシエーションは必要でなく，システム側で一意に決めることができる．また，一方，組み込みシステムに用いられるOSのほとんどがタスクごとの資源利用の統計をとるような高度な機能を持たないため，タスクごとに資源割当て量を遵守させることは難しい．

本論文では，以上のような特徴を持つ組み込みシステムに適したQoS制御方式として，「QoSテーブル方式」を提案する．QoSテーブル方式は設計時に作成した資源割当て量を記述したテーブルに基づいて，実行時にタスクへの資源割当て量を決定することを特徴とするQoS制御方式である．この方式は分散マルチメディアシステム用のQoS制御に比べてきわめて簡単な方式であるが，高速なQoS制御が可能であり，小型の組み込みシステムに対して十分に適用できる程度のフットプリントの小ささである．本論文では，まず，2章で，このQoSテーブル方式の関連研究について触れる．3章ではQoSテーブル方式の概要と必要機能について説明し，4章ではQoSテーブル方式の適用例として，CPU時間のQoS制御の設計およびRTOS上への実装と評価について述べる．5章では，QoSテーブル方式のスケーラビリティに関して考察する．最後に，6章では本論文のまとめと今後の展開について述べる．

2. 関連研究

分散マルチメディアシステムではQoS制御に関する研究がさかんで，さまざまな方式[1)~3)]が提案されている．これらのQoS制御方式の目的は，マルチメディアタスクと非リアルタイムタスクが混在するシステムにおいて，マルチメディアタスクどうし，あるいは，マルチメディアタスクと非リアルタイムタスクとの間の資源割当ての競合を解決することである．このため，これらの方式で用いられる資源割当て方針としては公平に資源を配分するFair shareポリシーや比率に従って資源を配分するProportional shareポリシーが用いられている．また，タスクに割り当てた資源を他のタスクに使用されることを防ぐために，タスクごとに資源利用の超過を厳しく検査している．組み込みシステムでは，タスクごとに資源割当て方針が異なるため画一的な資源割当て方針は適さない．また，組み込みシステム用のRTOSは資源利用の統計をとるような機能を持たないし，システムのオーバヘッドを考慮するとこの機能を追加することも避けたい．このため，分散マルチメディアシステム用のQoS制御方式をここで想定している組み込みシステムに適用することには問題がある．

また，Leeらは文献4)で，RT-Machの資源予約機能[5)]を利用したQoS制御方式を提案している．この方式ではタスクの実行時間や実行間隔といった資源予約の属性だけでなく，品質調整ポリシー（quality adjustment policy）とAdmissionコントロール・ポリシーとオーバランコントロール・ポリシーを指定できる．さらに，品質調整の優先度を各資源予約に関して指定でき，どの資源予約のQoSを先に変更するか優先順位を決めることができる．しかし，品質調整の優先度だけでは複雑なQoS制御の順序関係までは表現できないことが問題である．

Rosuらは文献6)で防空管制システムなどの高性能分散リアルタイムシステムのための適応的資源割当て（Adaptive Resource Allocation）モデルを提案している．このモデルでは，各タスクは資源要求を含む複数のコンフィグレーションを持ち，タスクの時間制約を満たすような1つのコンフィグレーションが選択される．たとえば，複数のプロセッサがある場合には余裕のある方のプロセッサや能力の高いプロセッサを使用するようなコンフィグレーションが選択される．この方法はあらかじめ決められた資源要求から実行時に選択する点がQoSテーブル方式に似ているが，タスクへの資源割当てに優先度が付けられないことと，他のタスクの品質を落としてまで，特定のタスクの資源割当てを保証することまでは考慮していない点で，QoSテーブル方式とは異なっている．

西尾らは文献 7) で，3 階層化した QoS 指定と層間の QoS 翻訳機構を導入してシステムが定量値を扱えるようにし，さらに，QoS 指定に単調性の制約をつけることによりセッション間調停を単純化した QoS 制御方式を提案している．セッション間調停の方針として，個人主義と公共の福祉の 2 種類のポリシーを用意しており，個人主義のポリシーを持つセッションはつねに可能な限り高位の QoS を追求し，公共の福祉のポリシーを持つセッションは自分を犠牲にしても存続するセッション数を維持もしくは増やそうとする．この方式は，各セッションの QoS 指定に単調性の制約を課して調停を単純化している点が QoS テーブル方式と類似している．しかし，QoS 変更に関する優先順位が個人主義のセッションと公共の福祉のセッションとの間で固定されている点が，QoS 変更に関する順序関係をテーブルで表現する QoS テーブル方式とは異なっている．

3. QoS テーブル方式の提案

本章では，我々が提案する QoS テーブル方式ついて具体的な例を示して説明する．

QoS テーブル方式は，タスクごとの資源割当て量を記載したテーブルを定義し，それに基づいて資源配分を決定する QoS 制御方式である．具体的には，「QoS テーブル」と呼ばれる**図 1** に示すようなテーブルを定義する．このテーブルには，タスクごとにいくつかの資源量が記載されており，横方向には資源を利用するタスクが並べられ，縦方向のインデックスはタスクの QoS のレベルを表す．たとえば，図 1 のテーブルの場合，タスク A には最高（インデックスは 0）で 50%の資源が割り当てられ，ついで 30%の資源が割り当てられ，最低（インデックスが 6）でも 20%の資源が割り当てられる．このインデックスはシステム全体で一意に管理される「QoS レベル」と呼ばれる整数値に対応しており，タスクへの資源割当て量は QoS レベルに対応する要素の値に決定される．たとえば，QoS レベルが 2 であれば，タスク A にはシステム全体の資源の 30%，タスク B には 30%，タスク C には 10%，タスク D には 20%がそれぞれ割り当てられる．

QoS レベル	タスク A	タスク B	タスク C	タスク D
0	50%	30%	10%	40%
1	30%	↓	↓	↓
2	↓	↓	↓	20%
3	↓	20%	↓	↓
4	↓	10%	↓	↓
5	20%	↓	↓	↓
6	↓	↓	↓	10%

図 1 QoS テーブルの例（↓ は表の上で 1 つ上のレベルの数値を引き継ぐことを意味する）
Fig. 1 Sample of QoS table.

また，QoS テーブルはタスクの資源割当て量を示すだけではなく，資源割当て量の減増方針を示すものでもある．たとえば，図 1 の QoS テーブルの場合，タスク C は増減なしのポリシーを持ち，システム全体の負荷がどのように変動しようとも，一定の資源割当て量を保証されている．

このような性質を持つ QoS テーブルを使うことにより，QoS 制御の手順は以下のように単純化することができる．

(1) 資源の過不足が発生した場合あるいは発生することが予測される場合は，必要なだけ QoS レベルを変更する．すなわち，資源不足のときには QoS レベルを下げ，資源に余裕があるときは QoS レベルを上げる．
(2) QoS レベルを決定したら，QoS テーブルの QoS レベルに対応する要素が示す資源量を各タスクに割り当てる．

なお，QoS テーブル方式では資源割当て量の変更にともなうタスクの処理内容の変更方法に関しては特に規定しない．基本的にはタスクのプログラムの一部をコールバックすることにより，処理内容を変更することを考えているが，同種のタスクが多いシステムであれば，自動的にタスクの処理内容を変更することも可能であると考えている．

このように簡素な手順で QoS 制御を実現できるため，QoS テーブル方式は従来の QoS 制御方法に比べて，QoS 制御に要する処理時間を短縮できると考えられる．しかし，QoS テーブル方式にはできないこともある．まず，1 つはあるタスクへの資源割当て量を減らしつつ，別のタスクへの資源割当て量を増やすことはできないことである．QoS レベルが上昇すればすべてのタスクの資源割当ては単調増加し，QoS レベルが下降すればすべてのタスクの資源割当ては単調減少するだけである．また，もう 1 つは複数の資源を同時に QoS 制御対象として扱うことも原則的にできないことである．複数の資源が同様に増減するのであれば，複数の資源を組にして QoS テーブルを作ることにより QoS テーブル方式でも扱うことができると考えられるが，複数の資源が相反して増減するようなタスクがある場合は QoS テーブルで扱うことは困難である．たとえば，ネットワークのバンド幅と CPU 時間を資源として扱うような場合，ネットワークのバンド幅を削減するためにデータを圧縮すると，CPU 時間が増加

する可能性があるが，このような場合には QoS テーブル方式を使うことはできない．この 2 つの問題については現在の QoS テーブル方式を拡張することにより対処できると考えているが，検討中の段階であるのでここでは論じない．今後，別の機会に報告したいと考えている．

それでは，ここで図 1 の QoS テーブルを使って，QoS テーブル方式による QoS 制御手順を説明する．ここで表中の数字は一番左の列が QoS レベルを表し，それ以外の列は資源利用率を表す．資源利用率とはその資源を利用する割合を表すもので，資源の全体量に対する比率☆で表現される．なお，ここでは資源利用率の合計が 100%を超えない限りはすべてのタスクが実行可能であるという前提で話を進める．

最初は，タスク A と B と C だけが動いているものとする．この状態では QoS レベル = 0 でも，この 3 つのタスクの資源利用率は 50 + 30 + 10 = 90（%）であるので，QoS レベル = 0 で実行される．しかし，さらにタスク D が実行可能になると，QoS レベル = 0 のままでは資源利用率の合計が 130%になってしまい，資源不足が起こることが予想される．この場合，QoS レベルは資源利用率の合計が 100%以下になるまで下げられる．QoS テーブルを参照すると，QoS レベルを 2 にすれば，資源利用率の合計は 30 + 30 + 10 + 20 = 90（%）となり，資源不足は解消することが分かる．そこで，QoS レベルは 2 に下げられ，タスク A の資源割当て量は 50%から 30%に減らされ，タスク D には新規に 20%の資源が割り当てられる．

また，いずれかのタスクが要求した資源利用率以上に資源を使用した場合には一時的な資源の利用超過状態が起こり，資源不足が起こる可能性がある．たとえば，タスク A と B と C が QoS レベル = 0 で実行中に資源不足が検出されたとする．この場合，QoS テーブルから計算される資源利用率の合計は 90%であり 100%を超えていないので，合計を 100%以下にするという前述の方法は使えない．そこで，資源利用の超過状態が解決するまで繰り返し QoS レベルを下げる．QoS レベルを 1 にすれば資源利用率の合計は 70%に減るので，まず，QoS レベルを 1 に下げる．それでも，まだ，資源不足が検出される場合には，さらに QoS レベルを下げる．

一方，タスクが資源を解放したり，一時的な資源の利用超過状態が解決したことにより，システム全体の資源利用率に余裕ができた場合には，逆に QoS レベルを上げて，各タスクへの資源割当て量を増やす．

以上が QoS テーブル方式による QoS 制御の手順である．この方法を実現するためには以下のような機能をシステムが備えればよい．

(1) QoS レベルごとの各タスクの資源要求を参照する機能．前出の QoS テーブルがこれに相当する．
(2) 資源の予約あるいは解放の際にシステム全体で要求される資源量に基づいて Admission テストを行い，その結果により QoS レベルを上げ下げする機能．
(3) 資源の不足あるいは余剰を検出し，QoS レベルを上げ下げする機能．
(4) QoS レベルに対応した資源割当てを行う機能．
(5) QoS レベルの変更に対応して，タスクの処理内容を変更する機能．

これらの機能のうち (1)，(2) は対象とする資源によらずほぼ共通の処理により実現できるが，(3)，(4)，(5) は対象とする資源によって実現方法が異なる．また，(4) の資源割当ては組み込みシステムでは，システムによるオーバヘッドを減らすために，各タスクが自分の宣言した資源量を守るように紳士的な資源確保を行うのが一般的であり，システムは関与しないものとする．(5) は各タスクの処理内容に依存するため QoS 制御されるタスクのプログラムによって実現されるべき機能であり，やはり，システムは関与しない．(3) の資源の過不足の検出方法に関しては，次章で CPU 時間を対象とした実例により詳しく説明する．

QoS テーブル方式のもう 1 つの利点は，QoS テーブルによりさまざまなポリシーを表現することができることである．たとえば，**図 2** の QoS テーブルは，Fair Share ポリシーを表現し，**図 3** の QoS テーブルは固定優先度的なポリシーを表現している．さらに，QoS テーブルは数式や優先度では表現することが難しい複雑な QoS 制御ポリシーまで表現できる．たとえば，**図 4** の QoS テーブルはタスク A と B には固定優先度的なポリシーを適用し，タスク C と D には Fair Share ポリシーを適用したものである．

4. QoS テーブル方式を用いた CPU 時間の QoS 制御

本章では，前章で説明した QoS テーブル方式を CPU 時間に適用した QoS 制御機構の設計について述べる．また，本方式を RTOS の 1 つである μITRON3.0[8] 上に実装したので，その性能評価を示す．

☆ 本論文中の表では百分率で表記する．

QoS レベル	タスク A	タスク B	タスク C	タスク D
0	100%	100%	100%	100%
1	60%	60%	60%	60%
2	50%	50%	50%	50%
3	40%	40%	40%	40%
4	33%	33%	33%	33%
5	25%	25%	25%	25%
6	20%	20%	20%	20%

図 2 Fair-share ポリシー的な QoS テーブルの例
Fig. 2 Sample of QoS table with fair-share policy.

QoS レベル	タスク A	タスク B	タスク C	タスク D
0	50%	30%	20%	100%
1	↓	↓	↓	50%
2	↓	↓	↓	0%
3	↓	↓	10%	↓
4	↓	↓	0%	↓
5	↓	20%	↓	↓
6	↓	10%	↓	↓

図 3 固定優先度的な QoS テーブルの例
Fig. 3 Sample of QoS table with fixed priority.

QoS レベル	タスク A	タスク B	タスク C	タスク D
0	50%	50%	100%	100%
1	↓	↓	50%	50%
2	↓	↓	25%	25%
3	↓	↓	20%	20%
4	↓	30%	10%	10%
5	↓	↓	5%	5%
6	↓	↓	0%	0%

図 4 固定優先度と Fair-Share ポリシーを合わせた QoS テーブルの例
Fig. 4 Sample of QoS table with mixed policy.

4.1 設　　計

設計の前提として，ここではスケジューリング方式として Rate Monotonic スケジューリングのように CPU 利用率の合計からスケジューリング可能性を判定できるものだけを対象にする．このような前提を置く理由は，CPU 利用率の合計からスケジューリング可能性を判定できないような一般的なスケジューリング方式を用いた場合，最適な QoS レベルを見つけるのに QoS テーブル全体を探索しなければならなくなり，前章で説明した方式を適用できないからである．QoS テーブル方式を改造することで一般的なスケジューリング方式に対応することも可能だが，その場合，QoS テーブル方式の利点である高速性が損なわれることになるので，QoS テーブル全体を探索する方式は採用しないことにした．

この前提を置くことにより，CPU 時間の不足はデッドラインミスの発生により，特定することができる．また，CPU 時間が余っていることはアイドル時間の増加によって直接特定することができる．タスクの開始や終了時に Admission テストを行うことにより，近い将来，CPU 時間の過不足が起こることを予測することもできる．

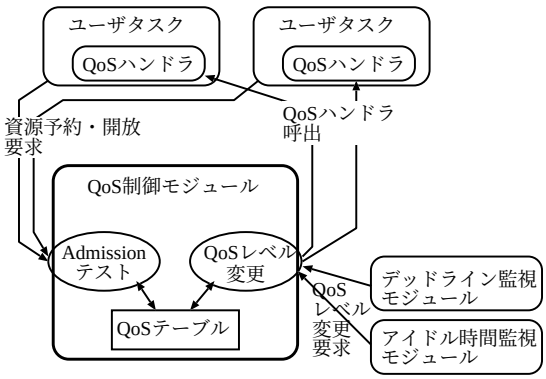

図 5 CPU 時間の QoS 制御の実現機構
Fig. 5 QoS control mechanism for CPU time.

以上のような考察から，CPU 時間の QoS 制御機構を以下のモジュールから構成する．また，これらのモジュールの関係を**図 5** に示す．

- QoS 制御モジュール
  QoS テーブルの管理と QoS テーブルに基づいた資源割当てを行う．さらに，タスクからの資源予約や解放の要求を受け取り，Admission テストを行い，必要ならば QoS レベルを変更する．また，外部からの要求を受け取り，QoS レベルを変更する．
- QoS ハンドラ
  QoS レベルが変更された場合に，実際に各タスクの処理内容を変更する関数である．タスク内に存在し，QoS 制御モジュールにより呼び出される．
- デッドライン監視モジュール
  CPU 時間の不足を検出するモジュールである．デッドラインミスが起きたかどうかを監視し，デッドラインミスが起きた場合には，QoS レベルを下げる要求を QoS 制御モジュールに対して行う．
- アイドル時間監視モジュール
  CPU 時間の余剰を検出するモジュールである．CPU 時間がどのタスクにも利用されない時間を検出し，それがある一定値を超えた時点で，QoS レベルを上げる要求を QoS 制御モジュールに対して行う．

4.2 実　　装

(1) ターゲット

実装のターゲットは，μITRON3.0[8] 準拠の RTOS である NEC RX830（以降，RX830）である．この OS を選んだのは，μITRON3.0 ベースの RTOS は国内では小型から中型の組み込みシステムに広く利用されて

おり，QoS テーブル方式の適用に向くと考えたためである．

（2） モジュールの実装

本来，カーネル機能の一部として実装すべきであるが，可搬性を考慮し，QoS 制御モジュールを，独立した μITRON のタスク（以降，QoS 制御タスク）として実装した．RX830 では固定優先度スケジューリングが提供されているので，QoS 制御タスクは QoS 制御の対象となるユーザタスクよりも高い優先度にした．これは QoS 制御タスクより高い優先度のユーザタスクが実行されることにより，長時間，QoS 制御できないことを防ぐためである．また，アイドル時間監視モジュールもタスク（以降，アイドル時間監視タスク）として実装した．アイドル時間監視タスクは他に実行されるタスクが何もないときだけ実行されるべきなので，システム内で最も優先度の低いタスクにした．

（3） デッドラインミス検出機能の実現方法

RX830 にはタスクの時限起床や周期実行の機能は用意されているが，デッドラインミスを検出するための機能が提供されていない．そこで，RX830 で用意されている周期ハンドラの機能を利用して，デッドラインミスの検出機能を実装した．周期ハンドラとは一定時間ごとに指定の関数を呼び出す機能であり，RX830 ではタスクの周期実行もこの周期ハンドラを使って実現する☆．デッドラインミスの検出機能の実現方法は，タスクの起床あるいは時限起床を要求する直前に相対デッドライン時間を周期パラメータとして持つ周期ハンドラの起床設定をするだけである．このデッドラインミス検出用の周期ハンドラ（以降，デッドラインハンドラ）の中には，それぞれのタスクのデッドラインミス処理を記述しておく．こうすることにより，タスクの実行がデッドラインまでに終わらなかった場合にはこの周期ハンドラが呼ばれてデッドラインミス処理が行われる．一方，デッドラインより前にタスクの実行が終わった場合には周期ハンドラの起床設定を解除し，余計なデッドラインミス処理を行わないようにしている．

なお，割り当てられた以上の CPU 時間を使わないようにするため機能であるポリーシングは，その実現のために CPU の横取りが起きるすべてのタイミングでタスクごとの CPU 時間の集計を行わねばならず，それを RX830 の改造なしには実現できなかった．今回の実装では RX830 を改造しない方針であったので，今回はポリーシングの実装を見送った．

☆ 周期ハンドラの詳細は文献 8) を参照のこと．

（4） QoS レベル変更の工夫

現在の実装では，外部からの要求で QoS レベルを変更する場合に，ヒント値を指定できるようにしている．ヒント値として指定できるのは不足あるいは余剰分の CPU 利用率である．ヒント値を指定できるようにしたのは，QoS レベルを 1 つずつ変化させるだけでは，なかなか目標の QoS レベルに達しない可能性があるからである．このヒント値は，たとえば，デッドラインミスが起こった場合で，デッドラインミスを起こしたタスクがまったく実行されなかった場合には，そのタスクに割り当てられている CPU 利用率分をヒント値にするというような使い方をされている．

4.3 評　　価

ここでは QoS テーブル方式の評価について述べる．ここで示す評価は，QoS 制御の効果とオーバヘッドに関してである．ここで効果とは CPU 利用率の向上とデッドラインミスの削減を意味している．これらを調べることにより QoS テーブル方式の導入によるメリットとデメリットを評価することができる．

なお，本節の評価に用いたシステムの構成は以下のとおりである．

- 評価ボード
  RTE-V831-PC（マイダス・ラボ社製）
  CPU：V831（V830 コア）100 MHz 118 MIPS
  キャッシュ　4 KB（命令）4 KB（データ）
  バスクロック：33 MHz　DRAM：8 MB
  EPROM：128 KB　FlashROM：8 MB
- ROM エミュレータ
  PARTNER-ETII（京都マイコン社製）

プログラムはフロントエンドの PC からダウンロードされて，実行される．フロントエンドの PC ではデバッガや性能解析ツールが動作し，評価ボード上のプログラムと連携する．

4.3.1 QoS 制御の効果

ここでは QoS テーブル方式を導入することにより，得られる効果について測定結果を基に評価する．

CPU 時間の QoS 制御の目的は CPU 時間を最大限に利用しつつ，デッドラインミスをできる限り減らすことである．そこで，QoS 制御の効果を評価する基準として，デッドラインミスの回数とシステム全体の有効 CPU 利用率を調べることにした．ここで，有効 CPU 利用率とは，タスクがデッドラインミスを起こさないで最後まで実行できた場合の CPU 利用率のことを示している．今回の評価ではデッドラインミスを起こした場合，途中までの計算は利用されず破棄されるものとする．

そして，この評価では，QoS制御が必要になるであろう典型的なモデルケースを想定して，負荷を与えるタスク群を設定した．そのモデルケースとは以下のようなものである．

- 割込みには継続して周期的に発生するセンサからの入力と，たまにバースト的に発生するネットワークに対する入出力がある．
- センサ入力に基づいてアクチュエータを制御するような機器制御のためのタスクがいくつかある．これらのタスクは厳しいリアルタイム性を要求され，デッドラインミスは極力少なくされるべきである．さらに，QoS制御もできないものとする．
- センサやネットワークからのデータを視覚化するためのマルチメディア処理タスクがいくつかある．これらのタスクは比較的リアルタイム性が緩く，たまにデッドラインミスが発生しても許される．さらに，フレームレートや解像度の変更によりQoSの変更が可能である．

このモデルケースは，マルチメディア処理を含んだ小規模なリアルタイムシステムを模倣したものである．このようなモデルケースにおけるQoSテーブル方式の効果を評価するために次のようなプログラムを作成した．

このプログラムは通常状態，すなわち過負荷になっていない状態では

(1) 周期的な割込み処理に相当するタイマ割込みハンドラ1個と，
(2) 機器制御を行うタスク相当の負荷を与える高優先度の周期タスク☆（以下，高優先度タスク）1個と，
(3) マルチメディア処理を行うタスク相当の負荷を与える低優先度の周期タスク（以下，低優先度タスク）4個，

を継続的に実行する．

この基本セットに外乱として，2パターンの負荷をそれぞれ追加して，以下の(1)，(2)の一時的な過負荷状態を作り出す．この過負荷状態は500ミリ秒間継続し，それ以降は基本セットだけの負荷状態に戻る．

(1) 割込みの多発による過負荷状態
基本セットに対して，割込み処理相当の負荷を与える周期ハンドラ（以下，疑似割込みハンドラ）を1個追加する．
(2) 新規の高優先度タスクの開始による過負荷状態
基本セットに対して，基本セットの(2)のような高優先度タスクを1個追加する．この高優先度タスクの実行優先度はQoS制御タスクよりも低く，基本セットの高優先度タスクよりも高い．

なお，実行の優先度は高い方から，タイマ割込みハンドラ，疑似割込みハンドラ，QoS制御タスク，高優先度タスク，低優先度タスク，アイドル時間監視タスクの順である．

このプログラムを(1) QoSテーブル方式を用いた場合，(2) QoS制御は用いないが最悪時でもすべてのタスク実行を保証する場合（以下，完全保証方式），(3) QoS制御を用いず，通常状態でのCPU利用効率を最大にする場合（以下，無保証方式）の，3つについて動作させ，その結果を比較した．評価プログラムに含まれるタスク，ハンドラの各種パラメータおよびQoSテーブルの内容を**図6**に示す．なお，完全保証方式では，追加分の負荷を収容する余裕を作るために，無保証方式よりも低優先度タスクのCPU利用率を下げている．

本評価で用いるRX830にはシステム全体の有効CPU利用率を調べる手段がない．そこで，次のようにして，タスクごとの有効CPU利用率（ユーザタイムおよびシステムタイムを含む）を計算し，その合計からシステム全体の有効CPU利用率を求めた．評価プログラムの疑似割込みハンドラと高優先度および低優先度タスクは，決められた回数だけループのカウントをしている．そして，このループのカウント数とループ1回の実行時間，経過時間からタスクの実際のCPU利用率☆☆を計算する．ループ1回の実行時間は初期化時に調べている．また，デッドラインミスの回数はデッドラインハンドラの中でカウンタを使って計数した．

結果1：割込みの多発による過負荷の場合

最初の評価は割込みが多発して過負荷状態になる場合を想定したものである．一般に割込みはいつ発生するかを予測することができないので，デッドラインミスが起きてはじめて，QoS制御が行われるという特徴がある．この結果を**表1**に示す．

この表で上の2行はそれぞれ追加の割込みが発生していない状態と追加の割込みが発生している状態のシステム全体のCPU利用率を表している．この結果で，追加の割込み負荷は基本セットが動作した後300ミリ秒の時点で1回だけ投入されており，過負荷状態のCPU利用率とは追加の割込み負荷が投入されている

☆ 周期タスクは周期ハンドラによって起床され，1周期分の実行が終了したら自発的にスリープするタスクとして実現している．

☆☆ CPU利用率 = (カウント数 × ループ1回の実行時間)/経過時間

タスクの種類	周期	デッドライン	CPU 利用率
タイマ割込み処理	1 ms	–	3%
評価 (1) で追加する疑似割込みハンドラ	1 ms	–	35%
高優先度タスク	10 ms	–	15%
評価 (2) で追加する高優先度タスク	10 ms	–	35%
低優先度タスク 0-3（完全保証方式）	50 ms	49 ms	11%
低優先度タスク 0-3（無保証方式）	50 ms	49 ms	20%

QoS レベル	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
タイマ割込み	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%	3%
追加の疑似割込みハンドラ	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%
高優先度タスク	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%	15%
追加の高優先度タスク	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%	35%
低優先度タスク 0	20%	20%	20%	20%	15%	15%	15%	15%	10%	10%	10%	10%	5%	5%	5%	5%
低優先度タスク 1	20%	20%	20%	15%	15%	15%	15%	10%	10%	10%	10%	5%	5%	5%	5%	0%
低優先度タスク 2	20%	20%	15%	15%	15%	15%	10%	10%	10%	10%	5%	5%	5%	5%	0%	0%
低優先度タスク 3	20%	15%	15%	15%	15%	10%	10%	10%	10%	5%	5%	5%	5%	0%	0%	0%

低優先度タスクの優先度はタスク 0 が一番高く，タスク 3 が一番低いものとする．

図 6 評価プログラムで使用するタスクの各種パラメータと QoS テーブル
Fig. 6 Task attributes and QoS table of evaluation programs.

表 1 割込みによる過負荷状態の場合
Table 1 Results under influence of interrupt.

	QoS テーブル方式	完全保証方式	無保証方式
通常負荷状態の CPU 利用率（%）	93.9	58.9	94.9
過負荷状態の CPU 利用率（%）	91.4	93.9	89.9
デッドラインミス（回/回/回/回）	0/0/1/4	0/0/0/0	0/0/10/10

500 ミリ秒間の CPU 利用率の平均であり，通常負荷状態の CPU 利用率とは追加の割込み負荷を取り除いた後の 500 ミリ秒間の CPU 利用率の平均を示している．一番下の行は低優先度タスクのデッドラインミスの回数をタスク 0，1，2，3 の順に表している．

無保証方式の場合は，割込みによる過負荷になる前から CPU 時間をほぼ一杯（約 95%）まで使う．このため，追加の割込みによる負荷に CPU 時間を奪われたことにより，低優先度タスク 2 と 3 は CPU 時間をほとんど割り当てられず，過負荷状態の 500 ミリ秒間ずっとデッドラインミスを起こしていることが分かる．それに対し，完全保証方式の場合はデッドラインミスは起きていない．

しかし，CPU 利用率については，無保証方式の場合は通常負荷状態でほぼすべての CPU 時間を使い切っているのに対し，完全保証方式の場合は約 60%しか CPU 時間を使っておらず，つねに 40%近くの CPU 時間が無駄になっている．過負荷状態では，完全保証方式の場合も過負荷状態でほぼ全部の CPU 時間を使い切っていることが分かる．無保証方式の場合は逆に CPU 利用率が下がっているが，これは低優先度タスク 2 の実行の途中でデッドラインミスが起こるために途中までの実行時間が有効な CPU 利用率として計上されないからである．

一方，QoS テーブル方式の場合はデッドラインミスは低優先度タスク 2 が 1 回，タスク 3 が 4 回である．デッドラインミスが起きるまで QoS 制御が行われないので，それぞれ 1 回のデッドラインミスは仕方ないが，タスク 3 で 4 回のデッドラインミスが起こるのは問題である．この理由についてはすぐ後で詳述する．また，CPU 利用率は通常状態で 93.9%，過負荷状態でも 91.4%と非常に高い値になっており，QoS 制御により CPU 利用率が改善されたことが分かる．

ここでタスク 3 でデッドラインミスが 4 回起こった理由について説明する．過負荷状態の前後も含めた見かけの CPU 利用率と QoS レベルがどう変化しているかを**図 7** に示す．「見かけの CPU 利用率」というのは，実行されているタスクが要求する CPU 利用率の総和を意味している．この値が 100%を超えているときは過負荷状態であり，デッドラインミスが起きることになる．これを見ると，疑似割込みハンドラによる負荷が投入されると，QoS レベルは最初に 4 レベル分一気に下がった後は 1 つずつ下がっている．これは，実装の QoS レベル変更の工夫のところで述べたヒント値の与え方の違いによるものである．もしも，デッドラインミスが起きた時点でそのタスクが周期の

はじめからどれだけのCPU時間を消費したかが分かれば，タスク3のデッドラインミスを1回にすることもできる．ただし，タスクごとにCPU時間の消費量を調べることはオーバヘッドの増加につながるので，結果としてCPU利用率を改善できるとは限らない．

結果2：高優先度タスクの実行による過負荷の場合

高優先度のタスクが実行されることにより過負荷状態が引き起こされる場合の結果を示す．この場合は過負荷状態になることはタスクを実行開始する時点で検出できるので，過負荷状態を予測できる点が前の評価とは異なる．この結果を**表2**に示す．なお，表中の数字の意味は表1と同様である．

完全保証方式と無保証方式の場合は，割込みの多発による過負荷状態の場合と同じ結果になっている．これは低優先度タスクにとっては，割込みハンドラも高優先度タスクもCPU時間を横取りするものという点から見れば，ほぼ等価であることを示している．

一方，QoSテーブル方式の場合はデッドラインミスは一度も起きず，通常負荷のときも過負荷状態のときもCPU利用率は95%と，ほぼCPU時間を使い切っていることが分かる．このように理想的な結果が得られたのは，割込みの多発による過負荷状態の場合と違い，過負荷状態になることが前もって予測でき，かつ，CPU利用率がどの程度不足するかも計算できるからである．

ただし，これは評価プログラムのタスクがQoSテーブルに設定されたCPU利用率どおりに実行時間を変えることができるからであり，実際のアプリケーションでは必ずしもこのように理想的な結果が得られるとは限らない．

以上の2つの結果からQoSテーブル方式には，QoS制御を用いない場合に比べて，デッドラインミスを減らし，CPU利用率を高める効果があることが確認された．特に明示的にタスクの開始や終了が指定された場合には，CPU時間をほとんど無駄にしないことが分かった．

4.3.2 QoS制御のオーバヘッド

QoS制御のオーバヘッドを調べるために，QoSテーブルの登録あるいは削除要求を受け取ってから，その処理を終えて管理タスクが再び要求待ちになるまでの時間（以降，単にレイテンシと呼ぶ）を測定した．**図8**にその結果を示す．

このグラフは，タスク数を10個から100個に増やした場合のレイテンシの変化を示しており，横軸はQoSテーブルのタスク数，縦軸はQoS制御のレイテンシを表している．4本のグラフはそれぞれ登録のレイテンシの最大値と最小値，削除のレイテンシの最大値と最小値を表す．ここで100タスクまでの結果を示したのは評価ボードのCPU上では能力的に100個以上のタスクを動かすことはないと考えたからである．

まず，ここでのQoSテーブルの登録および削除処理の内訳を示す．

(1) QoSテーブルにエントリを追加あるいは削除

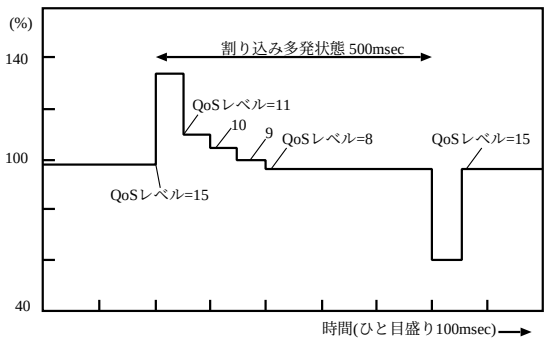

図7 見かけのCPU利用率の遷移
Fig. 7 Transition of expected CPU utiliazation.

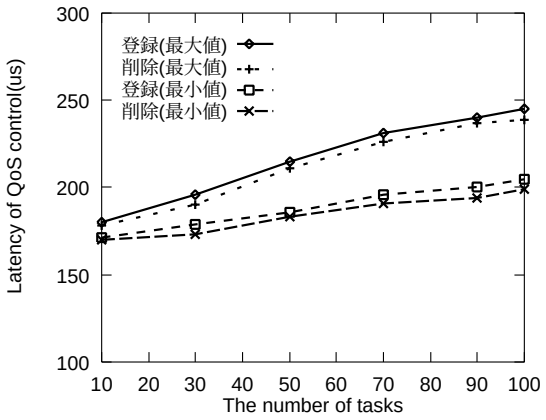

図8 タスク数に対するQoS制御のオーバヘッド
Fig. 8 QoS control overhead vs. number of tasks.

表2 高優先度タスクの新規実行による過負荷状態の場合
Table 2 Result under influence of high priority task.

	QoSテーブル方式	完全保証方式	無保証方式
通常負荷状態のCPU利用率（%）	94.9	58.9	94.9
過負荷状態のCPU利用率（%）	94.9	93.9	89.9
デッドラインミス（回/回/回/回）	0/0/0/0	0/0/0/0	0/0/10/10

する．
(2) QoS レベルを 1 つ下げるか上げる．
(3) Admission テストを行う．
(4) 資源割当て量が変更されるタスクを 1 つ探し出す．
(5) そのタスクの QoS ハンドラを呼び出す．QoS ハンドラの処理は，変数を 1 つ書き換えるだけである．

このように，ここでの QoS テーブルの登録および削除処理は QoS 制御の処理をひととおり含んでおり，QoS 制御タスクの処理の中ではレイテンシが比較的大きい．

さて，図 8 に戻ると，(1) から (5) の処理を行うのに 10 タスクで約 180 マイクロ秒，100 タスクの場合には最大で 250 マイクロ秒を要している．評価ボードの性能から，10 ミリ秒以下の周期タスクを動かすことはほとんどないと考えられるので，十分にオーバヘッドは小さいと考えられる．

また，最大値と最小値の差は QoS テーブルの検索時間によるものであると考えられる．したがって，頻繁に参照および更新されるの QoS テーブルのエントリを最も早く検索される場所に置くことでレイテンシの平均値は 100 タスクでも 200 マイクロ秒程度に抑えられると考えられる．

5. スケーラビリティに関する考察

QoS テーブル方式の 1 つの特長はフットプリントの小ささである．しかし，これは QoS テーブルのサイズが小さいときにだけいえることであり，システムの規模が多くなりタスク数が増えると，QoS テーブルのサイズが最悪，二乗オーダで増加していく．実際のシステムでは，タスク数が増えることがそのまま QoS レベルの増加につながるとは限らないが，同時に動作するタスク数が増えた状況で，タスク数が少ない場合と同等のきめ細かさで QoS 制御を行うためには QoS レベルの数を増やす必要があると考えられる．そのため，最悪ではタスク数の二乗オーダで増加すると考えられる．

たとえば，タスク数が 10 個，QoS レベル数が 10 の場合，テーブルの要素数は 100 である．これは 1 要素のサイズが 32 ビット長だとすると 400 バイトに相当する．これは数十 K バイトのメモリ容量しか持たない小型の組み込みシステムでも許容可能なサイズである．これに対し，タスク数が 100 個，レベル数が 100 の場合，要素数 10000 で 40 K バイト，さらにタスク数が 500 個，レベル数が 500 になると 1 M バイトを QoS テーブルだけで消費することになる．これは数十 M バイトのメモリ容量を持つ大型のシステムでも利用がためらわれるサイズと考えられる．

また，オーバヘッドの評価で示したように，タスク数が増加すると，QoS 制御の処理時間も比例して増加する．さらに，それだけの大きさの QoS テーブルを設計すること自体がかなりの手間のかかる作業であると考えられる．

以上のような理由から，QoS テーブル方式は小型の組み込みシステムには向いているが，大規模な組み込みシステムに適用することは難しいといえる．スケーラビリティに関しては今後の課題として，解決を目指したい．

6. おわりに

本論文では，小型の組み込みシステムに適した QoS 制御方式である，「QoS テーブル方式」を提案した．本方式は，前もってタスクが利用する資源量をある程度は見積もることができるシステムに適用するものであり，その見積もった資源量を記載したテーブルに基づいて，タスクへの資源割当て量を決定する方式である．本方式では，資源割当て量を決定するプロセスにおいてネゴシエーションを必要としないため，高速な QoS 制御が可能になっている．また，さまざまな QoS 制御方針をテーブルを使って表現することが可能である．

また，この QoS テーブル方式を CPU 時間の QoS 制御に適用し，μITRON3.0 準拠の RTOS 上に実装した．さらに，QoS 制御の効果とオーバヘッドについて評価した．その結果，本方式を用いることにより，割込みの発生やタスクの起動により負荷が大きく変動するような環境においても，デッドラインミス回数を低く抑えつつ，過負荷時においても 91%という高い CPU 利用率を実現できることを確認した．また，オーバヘッドはタスク数の増加にともない上昇するものの，想定されるタスクの周期などに比べて十分に小さいことが分かった．これらの結果から我々は，QoS テーブル方式を小型の組み込みシステムの QoS 制御方式として採用することにより，資源の効率的利用に大きな効果を期待できると考えている．

今後は，QoS テーブル方式を CPU 時間以外の計算機資源，たとえばディスクのバンド幅やメモリなどに適用していくことを考えている．また，本方式を大規模組み込みシステムにも適用していくために，スケーラビリティを向上する工夫と QoS テーブルの構築方法の検討を進めていきたい．

参考文献

1) Coulson, G. and Blair, G.: Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems, *ACM Operating Systems Review*, Vol.29, No.4, pp.17–24 (1995).
2) Fujita, H., Nakajima, T. and Tezuka, H.: A Processor Reservation System supporting Dynamic QoS control, *International Workshop on Real-Time Computing Systems and Applications* (1995).
3) Gopalakrishnan, R. and Parulkar, G.: A Real-time Upcall Facility for Protocol Processing with QoS Guarantees, *ACM Symposium on Operating Systems Principles* (1995).
4) Lee, C., Rajkumar, R. and Mercer, C.: Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach, *International Symposium on Multimedia Systems* (1996).
5) Mercer, C., Savage, S. and Tokuda, H.: Processor Capacity Reserves: OS Support for Multimedia Applications, *IEEE International Conference on Multimedia Computing Systems* (1994).
6) Rosu, D., Schwan, K., Yalamanchili, S. and Jha, R.: On Adaptive Resource Allocation for Complex Real-Time Applications, *IEEE Real-time System Symposium*, pp.320–329 (1997).
7) 西尾信彦，徳田英幸：QOSの3階層指定とその翻訳を用いたセッションの単純化調停方式，情報処理学会論文誌，Vol.39, No.2, pp.328–341 (1998).
8) 社団法人トロン協会：μITRON3.0標準ハンドブック改訂新版，パーソナルメディア (1998).

(平成11年12月13日受付)
(平成12年 4 月 6 日採録)

菅原 智義（正会員）

昭和42年生．平成5年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．同年NEC入社．現在，NECインキュベーションセンター勤務．並列分散OS，リアルタイムOSの研究開発に従事．ソフトウェア科学会会員．

高野 陽介（正会員）

昭和37年生．平成元年筑波大学大学院工学研究科卒業．同年NEC入社．現在，NECインキュベーションセンター勤務．OSおよびシステムソフトウェアの研究開発に従事．工学博士．