μITRON4.0仕様の概要

15th July 1998

高田 広章
ITRON専門委員会／豊橋技術科学大学

hiro@ertl.ics.tut.ac.jp
http://ertl.ics.tut.ac.jp/~hiro/

ITRONプロジェクト ー 第2ステージへ

- 組込みシステム用のリアルタイムOSとそれに関連する仕様の標準化を行う

第1ステージ　1984年～

- リアルタイムカーネル仕様の標準化に注力

第2ステージ　1996年頃～

- 周辺仕様まで含めた標準化へ

← 組込みシステムの大規模化・複合化

↓

- ソフトウェア部品 (ソフトウェアIP, 実行時ソフトウェア)
  - ソフトウェア部品が流通する前提条件の整備
  - ソフトウェア部品の API の標準化（分野毎）
- 開発環境・言語
- 応用分野に特化した標準化

μITRON4.0仕様 ー 策定の必要性

ソフトウェアの移植性の重視

- 組み込みソフトウェアの大規模化により移植性が重視される傾向
- 移植性の向上はソフトウェア部品流通の前提条件
  - ! 「弱い標準化」により移植性が阻害されているという指摘

リアルタイム性保証のための機構

- リアルタイム性を持ったソフトウェア部品
  - → どのようにしてソフトウェア部品とアプリケーションのリアルタイム制約を両立させるか？
  - ↓
- リアルタイム性の保証を容易にするための機構

半導体技術の進歩への対応

- μITRON3.0を公開してから5年が経過

ソフトウェアの移植性の向上

- 基本的には標準化の度合いを強くすればよい
- 弱い標準化の利点は保つ
  - スケーラビリティの確保
  - プロセッサ（ハードウェア）の特徴を活かす
  - アプリケーション毎の要求への対応

スケーラビリティは大前提

- 移植性が重視されるのは比較的大規模なシステム
- 小規模なシステムにおいては性能が重視される傾向

- 対象のシステム規模に応じて標準化の強さを変える

cf. ITRON2 と μITRON（Ver.2）

スケーラブルな標準化

スタンダードプロファイルの導入

コンセプト

- μITRONの適用分野の中で比較的大規模なシステムを想定し，*標準的な機能セットを「強く」標準化*
- 性能重視の小規模なシステムにはサブセットで対応
- 標準を越える要求のために拡張機能も定義

μITRON4.0仕様全体	… 弱い標準化
スタンダードプロファイル	… 強い標準化

言い換えると，

- 移植性を重視するソフトウェア（例: ソフトウェア部品）はスタンダードプロファイルの機能のみを用いる
- ソフトウェアの移植性を重視する分野向けのカーネルはスタンダードプロファイルに準拠して実装する

プロファイル内でも弱い標準化の利点を！

- プロセッサの特徴を殺さない範囲でどこまで強く標準化できるかを追求
  - *カーネルの機能ではなく，カーネルのAPIを定義するというスタンスを明確に*
  - 標準化する部分とできない部分をより明確に
- 豊富な機能を用意した上で，ライブラリリンクの機構で必要のない機能がリンクされないように工夫
  - システムコールの単機能化
  - 属性による多機能化を避ける

  - スケーラビリティの確保
  - アプリケーション毎の要求への対応

より広いスケーラビリティの実現

3.0仕様よりも高機能化

- 例外処理のための機能
- オブジェクトIDの自動割り付け
- ハードリアルタイム対応

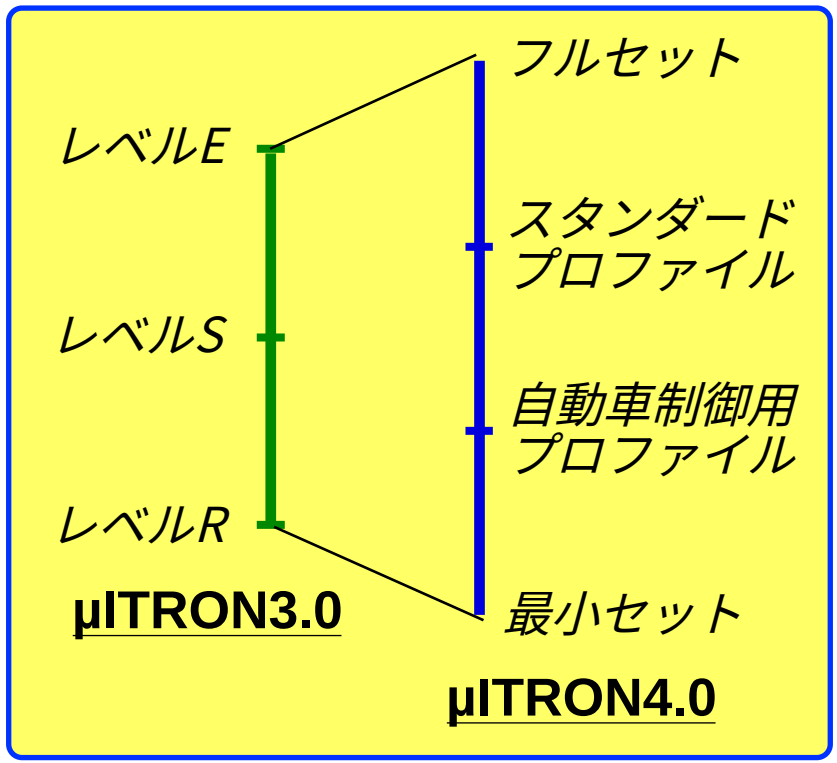
自動車制御用プロファイル

- アプリケーション毎の要求への対応
- 仕様本体とは別に定義

待ち状態をオプションに

（休止状態は必須）

- より小規模なシステムへの対応



検討の枠組みとスケジュール

要求事項の整理（1997年度）

- ハードリアルタイムサポート研究会
  カーネル仕様検討WG
  - スタンダードプロファイルに必要な機能
  - ハードリアルタイムサポート機能
- RTOS自動車応用技術委員会
  - 自動車制御用カーネル仕様
  - 待ち状態のないカーネル仕様

仕様のとりまとめ（1998年度）

- μITRON4.0仕様研究会
  カーネル仕様WG

→ **1998年内に完成予定**

→ *いずれもオープンな研究会/委員会*

スタンダードプロファイルの概要

想定するシステムイメージ

- ハイエンド 16bit ～ 32bit プロセッサ
- カーネルサイズ 10KB程度（全機能を使った場合）
- *システム全体が１つのモジュールにリンク*
  - システムコールはサブルーチンコールする
  - プロテクションの機構は持たない
- *カーネルオブジェクトは静的に生成*

基本的には，

- μITRON3.0仕様のレベルSのほとんど + 追加機能
- カーネルオブジェクトの生成情報の記述の標準化
- 実装依存性の削減，概念・用語の整理

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

静的API

- カーネルオブジェクトは静的に生成
  - → 静的に生成するための記述方法を標準化
- 通常のAPI（動的API）を大文字で書く

例）
```
CRE_TSK(tskid, { exinf, tskatr, task,
                 itskpri, stack, stksz, ... });
```
例）
```
DEF_INT(dintno, { intatr, intadr, ... });
```

- ソフトウェアを別のカーネルへ移植するのが楽に
- ソフトウェア部品の組み込みが容易に
- 動的APIと静的APIを個別に覚える必要がない

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

3.0仕様のレベルSから変更される機能

- 起動コードの渡せる sta_tsk は拡張機能に
  - → 代わりに，キューイングされる act_tsk を導入
- API名の変更
  - `preq_sem` → `pol_sem`
  - `snd_msg` → `snd_mbx`，`rcv_msg` → `rcv_mbx`
- 自タスクに対する `sus_tsk` を可能に
- C言語からの `ret_int` APIは廃止

3.0仕様のレベルEで採用される機能

- 固定長メモリプール
- タイムアウト付きのAPI
- 周期ハンドラ → 機能的には若干の変更あり，名称も変更

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

新たに追加される機能・API

- データキュー機能 → 後で詳説
- 例外処理のための機能 → 後で詳説
- act_can（act_tskのキャンセル），isig_tim（TICKを伝える）

実装依存性の削除

- C言語による割込みハンドラの記述方法を強く標準化
- 割込みハンドラから呼べる（iXXX_YYY）APIを規定 → 後で
- メイルボックスをポインタによる実装に限定
- スケジューリング規則をより強く標準化
- 3.0のレベルX機能をスタンダードと拡張機能に分ける
  - イベントフラグの複数タスク待ちは拡張機能に
- システムの初期化手順を標準化 → 後で詳説

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

概念・用語の整理

- タスク状態の名称を整理
  - RUN → RUNNING
  - WAIT → WAITING
  - SUSPEND → SUSPENDED
- システムの実行状態を整理
  - カーネル
  - 割込みハンドラ
  - CPU例外ハンドラ
  - タスク
    - タスクメイン処理ルーチン
    - タスク例外処理ルーチン
- 優先順位の整理

  ハンドラ ＞ カーネルのタスクディスパッチャ ＞ タスク

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

データキュー機能

- 1ワードデータのメッセージ通信機構（3.0仕様のリングバッファで実装されたメイルボックスに相当）
- 自動車制御応用から強い要求，他の応用でも有用
- 使わない場合にはリンクされない
- 追加されるAPI

  CRE_DTQ … 生成（静的API）
  rcv_dtq, prcv_dtq, trcv_dtq … 受信
  snd_dtq, psnd_dtq, tsnd_dtq … 送信（ブロックあり）
  isnd_dtq … 割込みハンドラかもデータ送信可能
  fsnd_dtq … バッファフルの時に最も古いデータを上書き
  cre_dtq, del_dtq, ref_dtq … 生成・削除・参照（拡張機能）

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

例外処理のための機能

! 例外処理の枠組みを定義

CPU例外ハンドラ ← 例外発生時にプロセッサが起動

（例外種別毎にアプリケーションで定義）

↓ 必要に応じて例外処理をタスクに任せる

タスク例外処理ルーチン

（タスク毎にアプリケーションで定義）

▸ タスク例外処理ルーチン

UNIXのシグナルハンドラを軽くしたような機能

- `ras_tex` を呼び出して明示的に起動要求
- タスクが次にスケジュールされる時に起動
- タスクと同じコンテキストで実行
- タスク毎に１つのみで，ネストして起動されない

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

割込みハンドラから呼べるAPI

`iact_tsk`, `irot_rdq`, `irel_wai`, `iwup_tsk`
`isig_sem`, `iset_flg`, `isnd_dtq`
`iras_tex`, `isig_tim`, `get_tid`

システムの初期化手順

ハードウェア依存の初期化処理 ← リセット

↓（ユーザが作成，カーネルの管理外）

カーネルの初期化，静的APIの処理

↓

初期化ルーチンの実行

↓（ユーザが作成，静的APIで定義）

カーネル起動，タスクの実行が開始

割込みが許可，システムクロックが動作開始

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

拡張機能の概要

3.0仕様のレベルE機能

- カーネルオブジェクトの動的生成，削除，状態参照
- 拡張同期・通信（メッセージバッファ，ランデブ）
- 可変長メモリプール
- アラームハンドラ　→ 機能的には変更あり
- exd_tsk，frsm_tsk など

追加される機能

- オブジェクトIDの自動割り付け
- 優先度継承・上限をサポートする排他制御機構
  - POSIX リアルタイム拡張の mutex に相当
- オーバランの検出

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

自動車制御用プロファイルの概要

3.0仕様のレベルSとの機能的な差

- SUSPENDED状態，rot_rdq
- メイルボックス
+ データキュー
+ 周期ハンドラ

スタック共有機構 ➡ メモリ節約のため

- 複数のタスクでスタックエリアを共有
- スタックを共有しているタスクは待ち状態に入れない
- スタンダードプロファイルでも動作するプログラムの書き方を明示

! この制約下でスタンダードプロファイルが上位互換に

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

最小の機能セット

待ち状態のないカーネル

- WAITING状態を必須から外し，RUNNING，READY，DORMANT状態を必須に
  - ! DORMANT状態のタスクはスタックを解放できる

- すべてのタスクが１本のスタックエリアを共有できる

最小の機能セット

- プリエンプティブな優先度ベーススケジューリング
- RUNNING，READY，DORMANT状態
- `CRE_TSK`，`DEF_INT`
- `act_tsk`，`iact_tsk`，`loc_cpu`，`unl_cpu`
- `ter_tsk`（メイン関数からのリターンで代用可）

§ このOHPは今後の議論で変わる可能性のある内容を含んでいます

μITRON4.0仕様研究会の検討課題

カーネル仕様WG

- μITRON4.0リアルタイムカーネル仕様の検討

デバッギングインタフェース仕様WG

目的:

- μITRON4.0仕様のリアルタイムカーネルとデバッグ環境（デバッガ，ICE，ロジアナ）の**RTOSサポート機能**とのインタフェースの標準化

目標:

- カーネルのオーバヘッドが最小限
- 異なるデバッグ環境とのインタフェースをできる限り共通化
- 他のRTOSやソフトウェア部品にも適用可能

デバッギングインタフェース仕様WG ― 続き

アプローチ:

- デバッグ環境側からの**処理要求**を、あるITRON仕様カーネルで行う場合の処理内容を、カーネルメーカ側が**プログラムの形**で提供
- デバッグ環境における**処理のプリミティブ**を標準化

例) デバッグ環境側からの処理要求

- あるタスク/オブジェクトの状態を読み出す
- タスク切替をトレースする
- あるタスクにブレークポイントをかける

例) デバッグ環境における処理のプリミティブ

- 指定した番地のメモリの内容を読み出す
- 指定した番地にブレークポイントをかける
- 指定した番地へのアクセスをトレースする

アプリケーション設計ガイドラインWG

- リアルタイム性を持ったアプリケーションをリアルタイムカーネル上に構築する場合
  - **タスク分割**の方法
  - **優先度**の付け方

  ➡ リアルタイム性確保の観点から1つの指針

- ソフトウェア部品を用いる場合のリアルタイム性の保証

  ソフトウェア部品の提供者

  … ソフトウェア部品の性質 (使用リソース, リアルタイム制約) を**パラメータ化**して提供

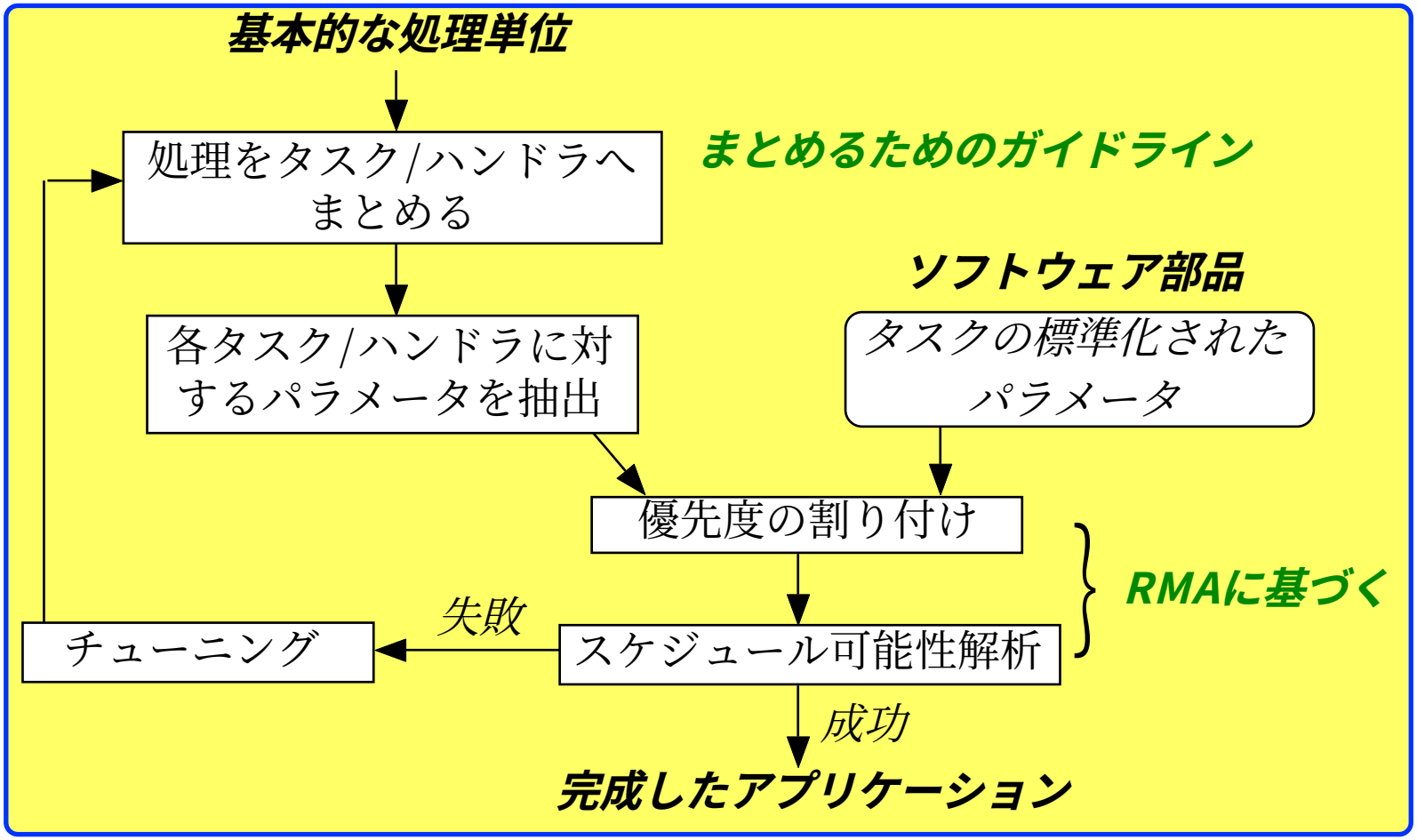
  ソフトウェア部品のユーザ

  … 提供された**パラメータ**を用いてシステム全体のリアルタイム性を検証

  ➡ パラメータの標準化

アプリケーション設計ガイドラインWG ― 続き

ガイドラインの流れ:



デバイスドライバ設計ガイドラインWG

- 「デバイスドライバ」という用語は人によってイメージが異なるので避けた方が無難

Device Interface Component (DIC) *仮称*

  - ハードウェアデバイスを扱うための最低限のインタフェースソフトウェア．OS のための抽象化は提供しない
  - ハードウェアデバイスの違いを過度に隠蔽しない (デバイスの特徴を引き出すことができる)
  - アプリケーションプログラマが直接呼び出す
  - デバイスメーカが提供してくれると理想的

→ *移植性の確保が極めて重要*

デバイスドライバ設計ガイドラインWG ― 続き

アプローチ:

- デバイスの (機能の) モデル化
  - 初期化
  - 同期処理要求
  - 非同期処理要求
  - 処理完了
  - 処理キャンセル
  - 例外事象通知
- DIC のソフトウェアアーキテクチャ
  - 共有ライブラリ型
  - タスク型
  - 組み合わせ型
  - ……
- DIC の実装手法のガイドライン
- DIC の API

デバイスの特性

データの有無，待ちの有無
キューイングの有無
データ転送方法
イベントの種類　などなど

更なる発展性

ソフトウェア部品仕様との関連

- ITRON TCP/IP API 仕様，JTRON 仕様は，μITRON4.0 と整合性が取れるようにバージョンアップ

さらに先の課題

- テストスイート/検証スイートの整備
- C++言語バインディングの標準化
- 他のソフトウェア部品APIの標準化
- 他の応用分野の要求への対応

最新情報は，

ITRONホームページ
http://tron.um.u-tokyo.ac.jp/TRON/ITRON