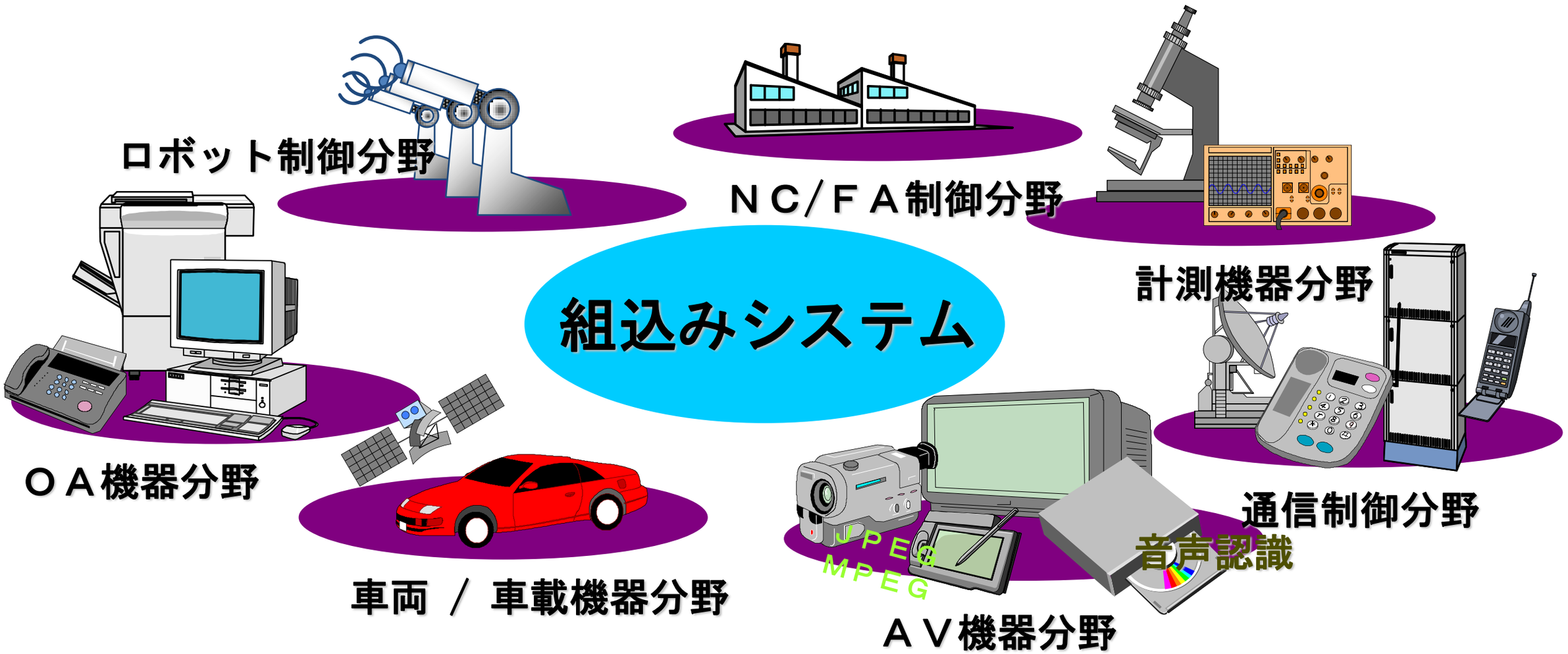
μITRON入門

TRON Forum
TRONフォーラム

1 組込みシステムとマルチタスク・リアルタイム処理

組込みシステムとは

- 組込みシステム=センサやアクチュエータ、他の機械システム等と協調して動作するコンピュータシステム



IH炊飯器、洗濯機、コピー機、ＦＡＸ、医療機器
ウエアラブル端末、電子楽器、自動車、ロボットなど

組込みシステムこそメジャー

- パソコン・ワークステーション等の出荷数
  - 約2億台/年
- 組込みコンピュータの出荷数
  - 200億個/年以上と言われる
- ⇒ビックデータ/オープンデータ化が進むとともに、ウエアラブル端末の普及などで、組込みコンピュータは今後も増えていくだろう。

組込みシステムの特徴

- 計算処理よりも、入出力処理、通信処理が中心
  - イベント処理プログラム
  - 実時間（リアルタイム）処理プログラム
- 専用化されたシステム
  - 厳しいリソース制限
  - 必要最低限のハードウェア資源にチューニングする⇒コストを極力下げる
  - 限られた基板面積、マイコンのFlash容量が小さく程コストがかからない、など
- 高い信頼性
  - パソコンがフリーズしてもお客は起こらないが、ビデオカメラがフリーズしたら、大クレーム
  - システム改修に多大なコスト⇒リコール改修費用など
- リアルタイムシステムであること

リアルタイムシステムとは

- 「リアルタイムシステム」では、計算結果があっていることだけでなく、決められた時間内に計算が終わることも保証しなければならない

(例)

- システムへの要求＝「127＋382の答えを求めなさい。答えは3分後までに出しなさい(12時23分)」
  - (答)509　(12時30分) ⇒リアルタイムシステムでは計算失敗の例となる
  - (答)509　(12時24分) ⇒リアルタイムシステムでも計算成功の例

リアルタイム処理とは（車の場合）

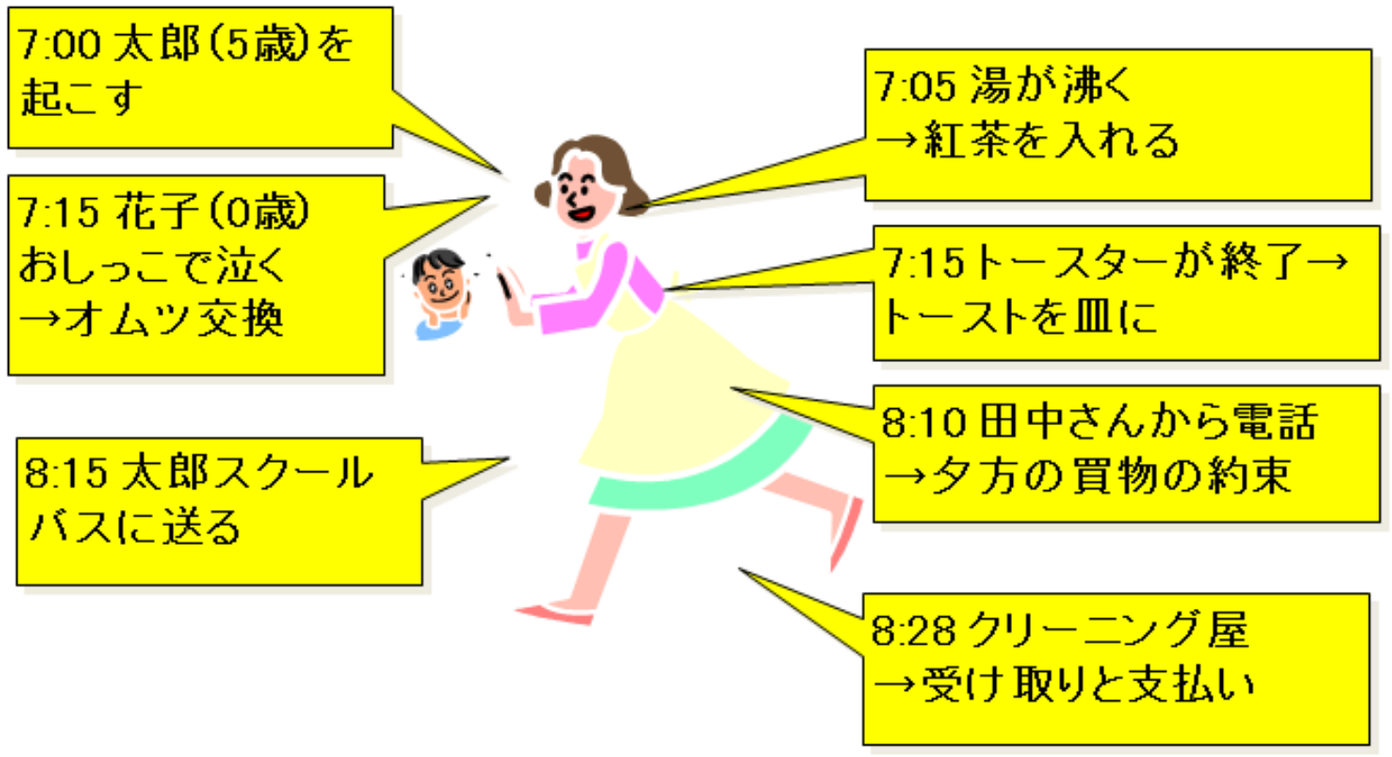
リアルタイム＝要求された時間内に決められた処理を行うこと

つまり、Windowsなどのパソコン用OSにリアルタイム性が無いため

車が衝突してエアバックが開く際、
画面に砂時計が出るなんてことに・・・、

リアルタイム処理とは

主婦兼母親の朝



コピー・プリンタ・ファックス複合機の例

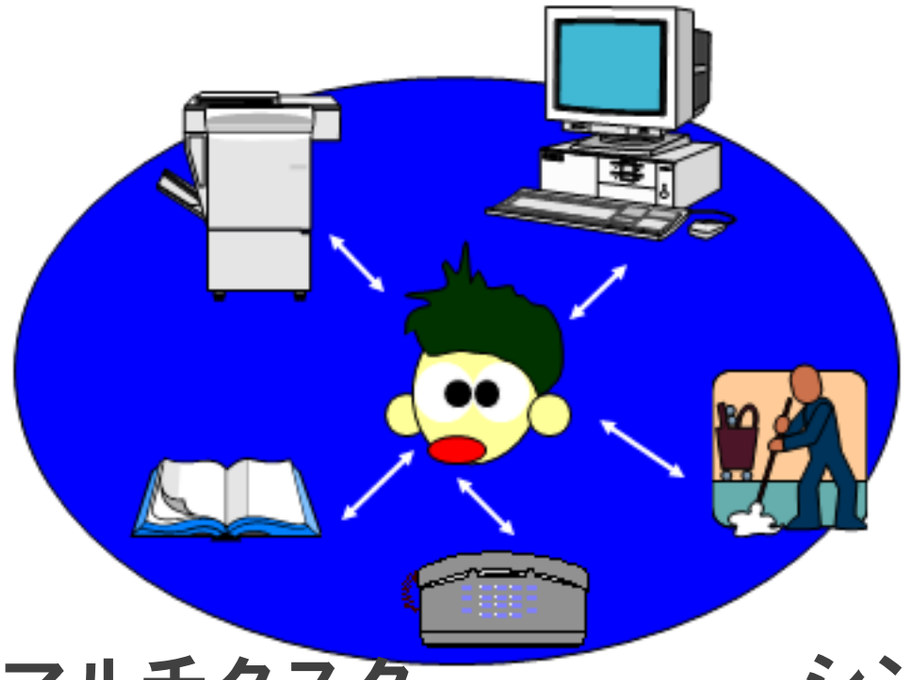
リアルタイムシステムは複雑

- 実世界とのかかわり
  - ランダムに起きる事象への対応
  - 実時間を扱う必要性
- 複雑な処理が要求されるため、処理するための技術が必要となる
  - 並行処理(タスク、スケジューリング)
  - 同期・通信
  - 実時間処理
  - 記憶管理

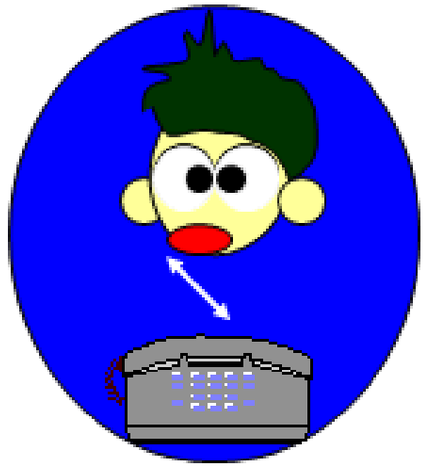
そこで、マルチタスク

- 複数の処理を同時に実行すること

  =マルチタスク処理

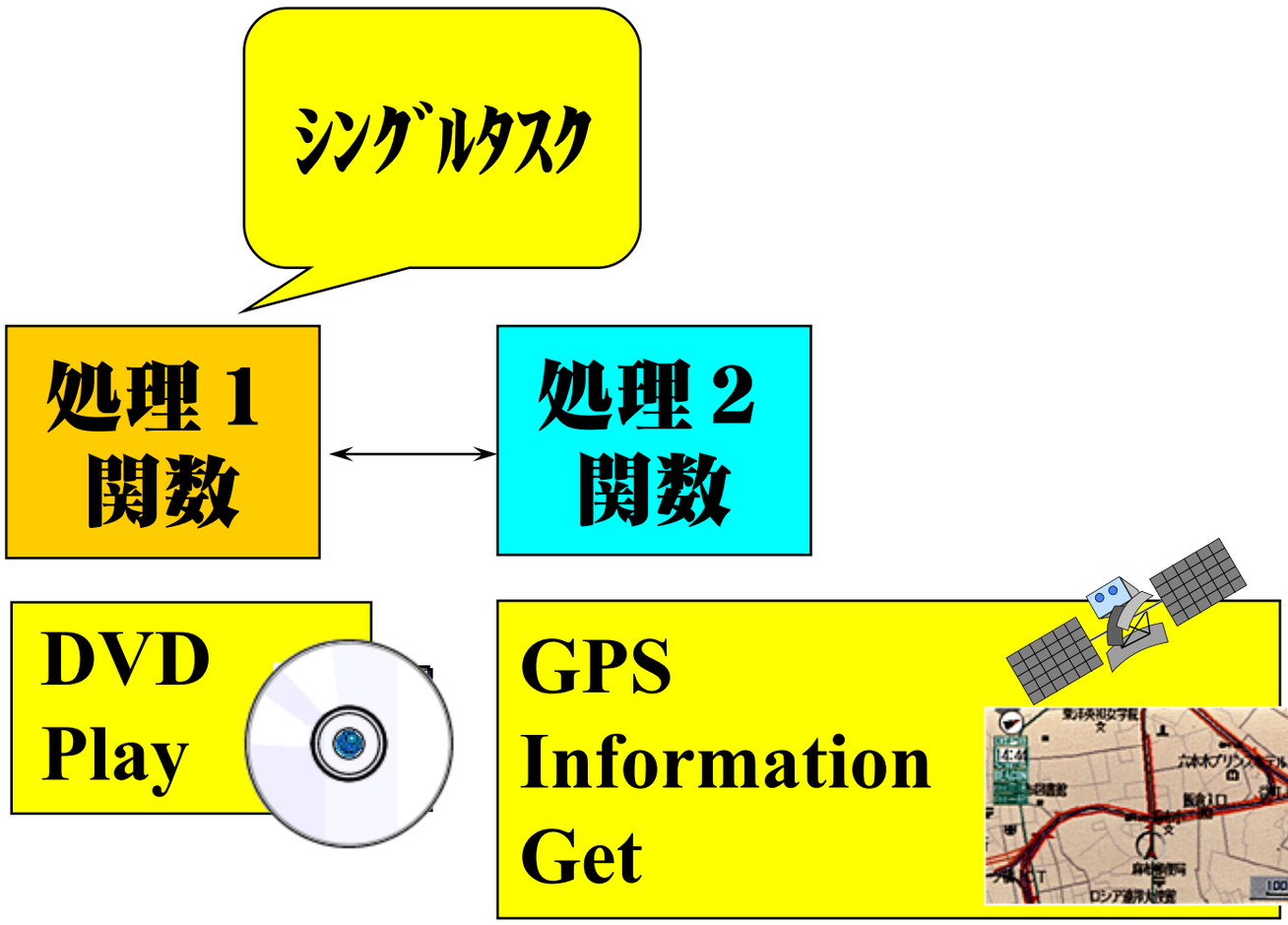


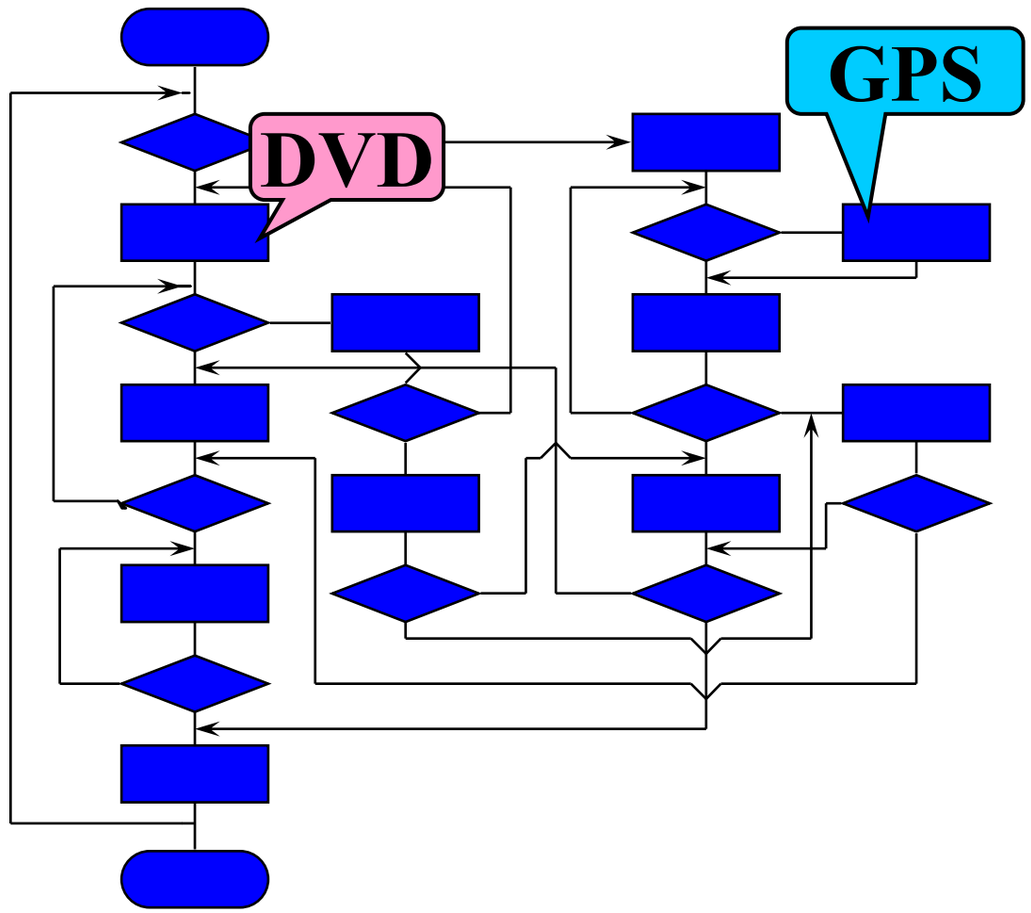
マルチタスク



シングルタスク

組込み機器制御
(シングルタスク)

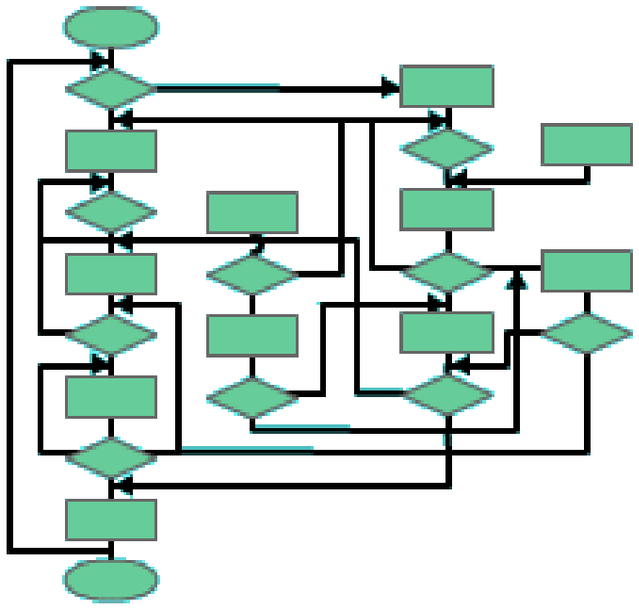




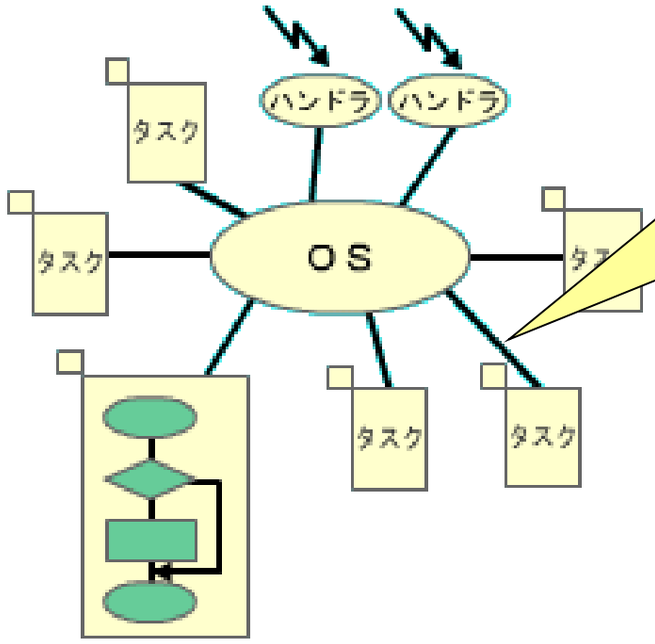
カーナビをシングルタスクで処理…
**　プログラムが複雑**
→規模が大きくなると管理出来ない！

組込み機器制御 (マルチタスク)

マルチタスクOSを使用しないシステム



マルチタスクOSを使用したシステム



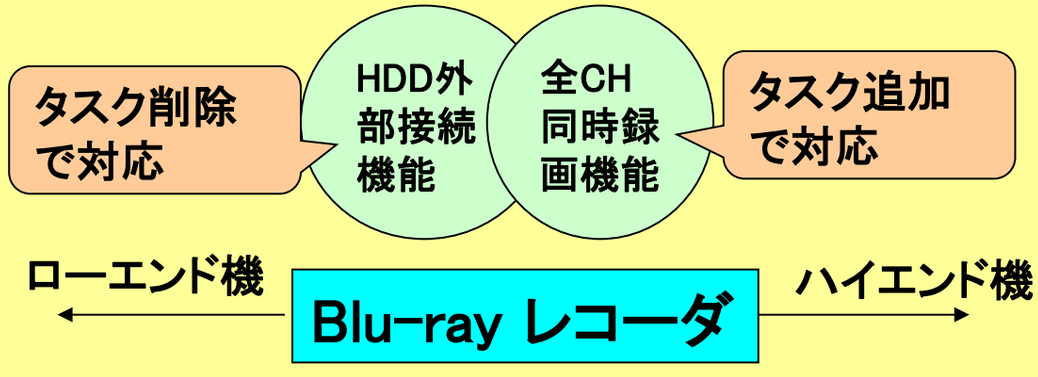
見通しの良いプログラム設計が可能となる。タスク(モジュール)再利用率が向上する(ミドルウェア等の利用可能)。

複数のタスクを動作させることから、リアルタイムOSのことをマルチタスクOSと呼ぶことがある。

不具合発生しにくい

開発効率の向上

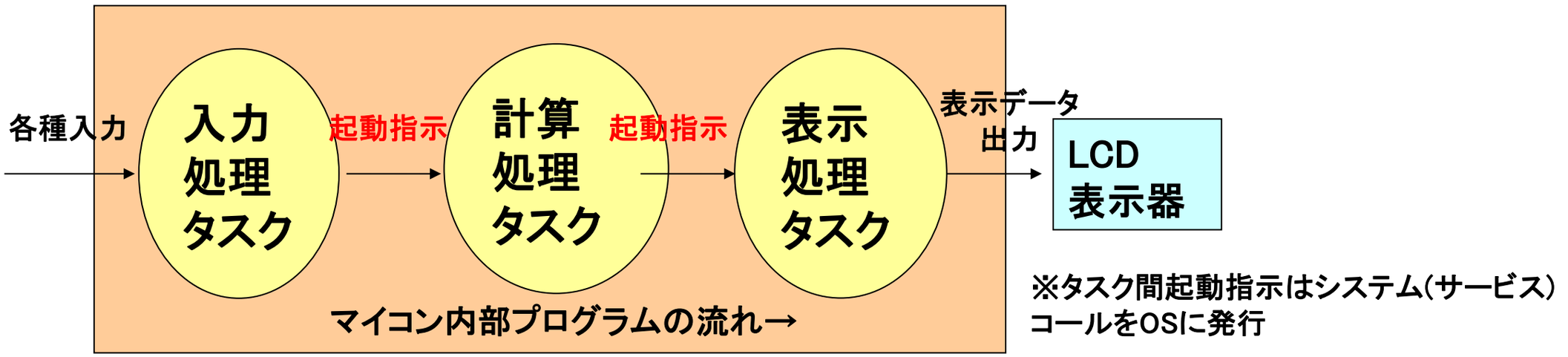
独立性/信頼性の高いタスクの取捨選択で、各種製品展開が短期間で可能となる。



ところで、タスクとは

プログラム実行の最小単位、意味のある1つの仕事のくくり （Moduleや関数）

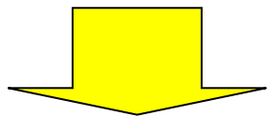
プログラムはタスク(Task)単位で実行



※タスク間起動指示はシステム(サービス)コールをOSに発行

タスクスケジューリング

- 同時に複数の処理をさせる(マルチタスク)
- しかしCPUは一つしかない…
- CPUを使う時間帯を複数のタスクに割り振る



- タスクスケジューリング

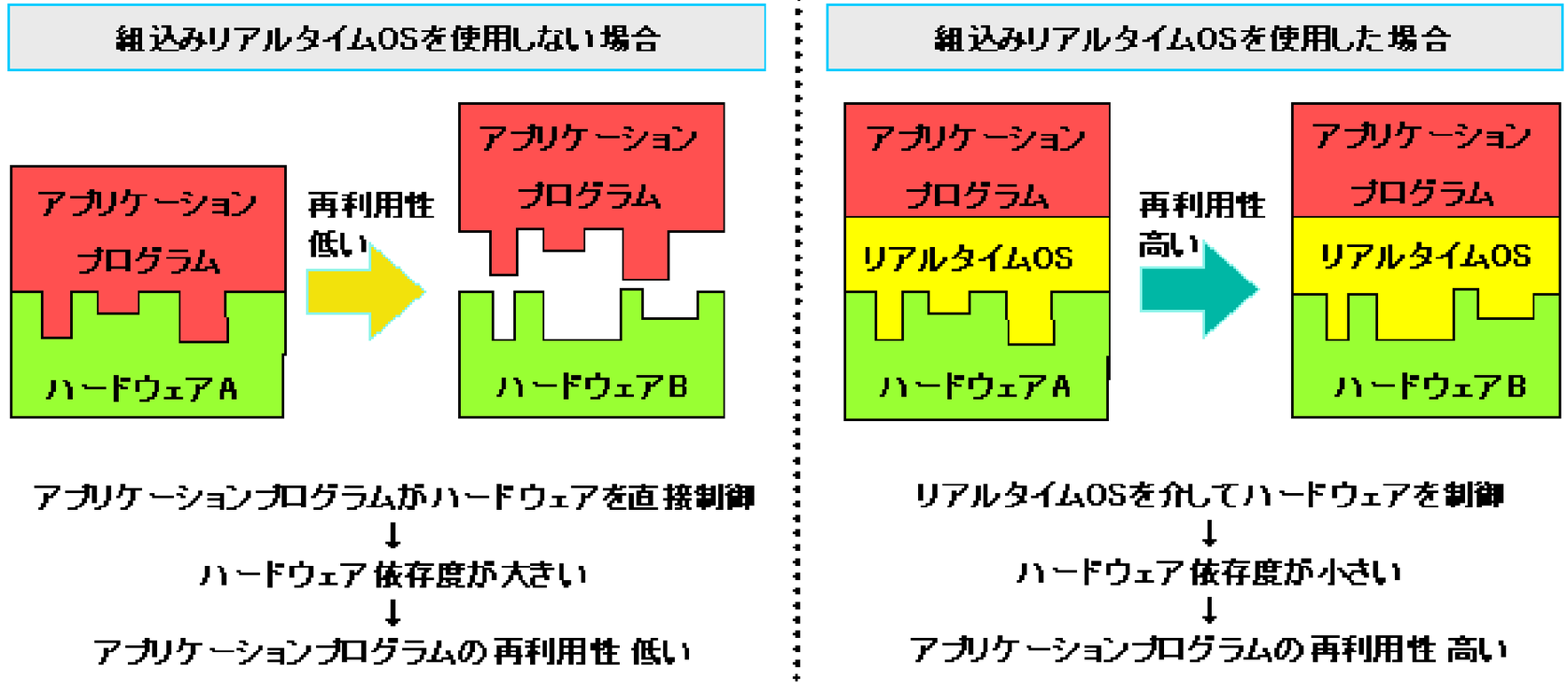
メモ：例えばWindowsなどのOSはスケジューリング機能が無いため、ユーザがWordやExcelを起動するが、リアルタイムOSでは、あらかじめタスクに優先度をつけておき、OSはその優先度に従いタスクを実行する。
よって、リアルタイムOSのことをタスクスケジューラと呼ぶことがある。

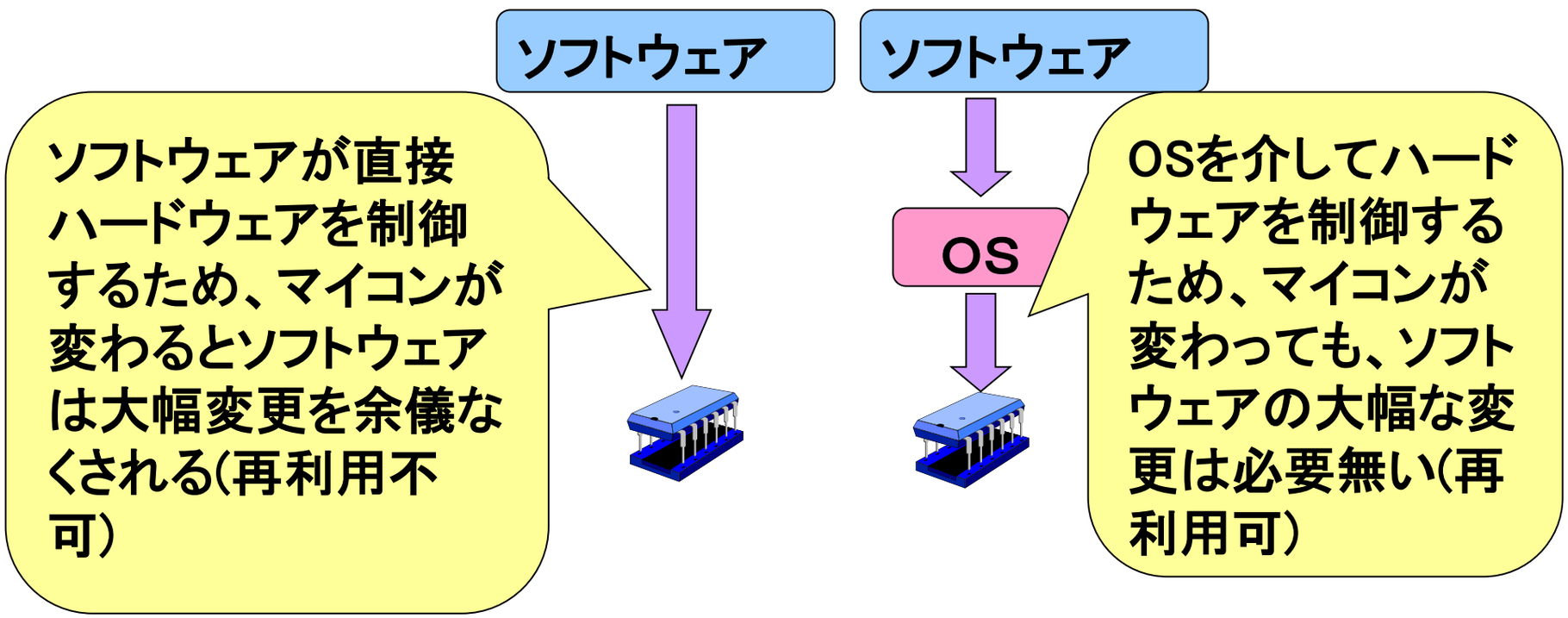
リアルタイムOS導入のメリット（リアルタイム/マルチタスク処理）

- リアルタイム処理の基本処理を扱ってくれる=プログラム作成が容易に
  - **並行処理・タスクスケジューリングのサポート**
    - タスク分割設計が容易に、効率的なI/Oも簡単に実現
  - **同期処理のサポート**
    - タスク間の通信・同期の実現が容易に
  - **記憶管理のサポート**
    - 記憶管理の細部に関らないでも済む

リアルタイムOS導入のメリット（プログラムの保守、再利用性）

- プログラムの再利用性の向上
- 保守性・拡張性の向上





RTOS導入のメリット
～例えばこんなことができる～

壁があることで、ソフトウエアの流用や検証などのノウハウ共有が難しい

壁

壁

アプリケーション・ソフトウエア

ミドルウエア

リアルタイムOS

ハードウエア

アプリケーション・ソフトウエア

ミドルウエア

リアルタイムOS

ハードウエア

アプリケーション・ソフトウエア

ミドルウエア

リアルタイムOS

ハードウエア

RTOS導入のメリット
～例えばこんなことができる～

壁を取り払うことで、ソフトウエアの流用や検証などのノウハウ共有が容易に

1 組込みシステムとマルチタスク･リアルタイム処理

2 トロンと組込みシステム

3 μITRON入門

4 μITRON開発手順

5 参考資料･付録など

トロンプロジェクトとITRON

- TRON Project since 1984
  - トロンプロジェクトはUbiquitous Computing研究開発の先駆け
- トロン(TRON)
  - The Real-time Operating system Nucleus
- 組込みシステム向けに、ITRON(Industrial TRON) Specification を公開(1989)
  - 現在のバージョンはμITRON 4.0仕様

トロンの有効性

- トロン(The Real-time Operating system Nucleus)の特長
  - 仕様が公開されている(オープン)
  - 実装が自由(ライセンス料がかからない)
  - 国内各社で利用されている(ITRON/T-Kernel)
- なぜ各社で利用されているのか？
  - ソフトウェアの流通性・再利用性の確保
  - 企業内のソフトウェア開発の標準化
  - 技術者教育の標準化

ITRONの歴史

ITRON(Industrial TRON)

機器組込み制御システム用の
オペレーティングシステム

- リアルタイム性、コンパクト
- オープンアーキテクチャ
- 弱い標準化による柔軟な
  ハードウェア適応化

ソフトウェア部品

ITRON／
FILE

カーネル仕様

ITRON1仕様

- 最初の標準化仕様
- 8、16bit MPU対象
- 標準化仕様の
  草分け的存在

ITRON2仕様

- 大規模システム用
- 主に32bit MPU対象
- ITRON1仕様に対し、
  大幅に機能拡張

μITRON2.0仕様

- 小規模組込み機器用
- 8、16bit MPU対象
- RTOSとして基本的な
  機能のみサポート

μITRON3.0仕様

- 中大規模システム用
- 8、16、32bit MPU包含
- RTOSとして豊富な
  機能をサポート
- ITRON仕様の集大成

【拡張された機能】
- オブジェクトの動的生成
- メッセージバッファ機能
- 周期/アラームハンドラ
- CPUロック/アンロック 他

ITRON
TCP/IP API

JTRON
2.1

μITRON4.0仕様

- 標準化仕様の強化
- スケーラビリティーの向上
- 構築方法の標準化

【効果】
- ソフトウェア部品
  (ミドルウェア)の流通
  促進
- 応用分野の拡大
- アプリケーション
  開発効率の向上

μITRON4.0/PX仕様

- 保護機能拡張

【効果】
- メモリへのアクセス保護
  （信頼性・安全性）
- カーネルオブジェクトへの
  アクセス保護（セキュリティ）

**最新バージョンはVer.
4.03.03**

トロンプロジェクト発足

1984 1989 1993 1996 1999 2002 2003

（第1フェーズ）

（第2フェーズ）

ITRONとT-Kernel

ITRONの成果を生かしつつ、標準化の範囲を拡大し、高度な技術を取り入れることで、短期間で高度な組込みシステムを作るためのソリューションの整備。

ITRON

強い標準化に基づく展開

T-Engineプロジェクト

ユビキタスコンピューティング環境構築を目指して、オープンなリアルタイム標準開発環境を整備するプロジェクト。 ハードウェア、OS、ミドルウェア、開発環境について、強い標準化が進められている。

- T-Kernel2.0
- uT-Kernel2.0

弱い標準化とプロファイルに基づく展開

- ITRON4.0仕様 → ITRON4.0/PX仕様

- 同一プロファイル規定のOS間で互換性を保証（スタンダードプロファイル、自動車プロファイル）
- 保護機能拡張(ITRON4.0/PX仕様)によるMMUサポート(メモリ保護を主目的としている)

ITRONの成果をベースとして、互換性や厳密性の向上を図り、生産性向上や再利用性、移植性を高めることを目的に仕様を整理。

μITRON 4.0とT-Kernelの機能比較

機　能	μITRON 4.0	T-Kernel	機能差
タスク管理機能	○	○	
タスク付属同期機能	○	○	
タスク例外処理機能	○	○	
同期・通信機能	○	△	TKはﾃﾞｰﾀｷｭｰ無
拡張同期・通信機能	○	○	
メモリ・プール機能	○	○	
時間管理機能	○	△	TKはｵｰﾊﾞﾗﾝH無
システム状態管理機能	○	○	
割込み管理機能	○	○	
サービスコール管理機能	○	×	TKはｻﾌﾞｼｽﾃﾑが同等
システム構成管理機能	○	○	
サブシステム管理機能	×	○	
システム・メモリ管理機能	×	○	
アドレス空間管理機能	×	○	
デバイス管理機能	×	○	
I/Oポート・アクセスサポート機能	×	○	
省電力機能	×	○	

μITRON適用範囲

●8-32ビットRISC/CISC系マイコンに適用可能

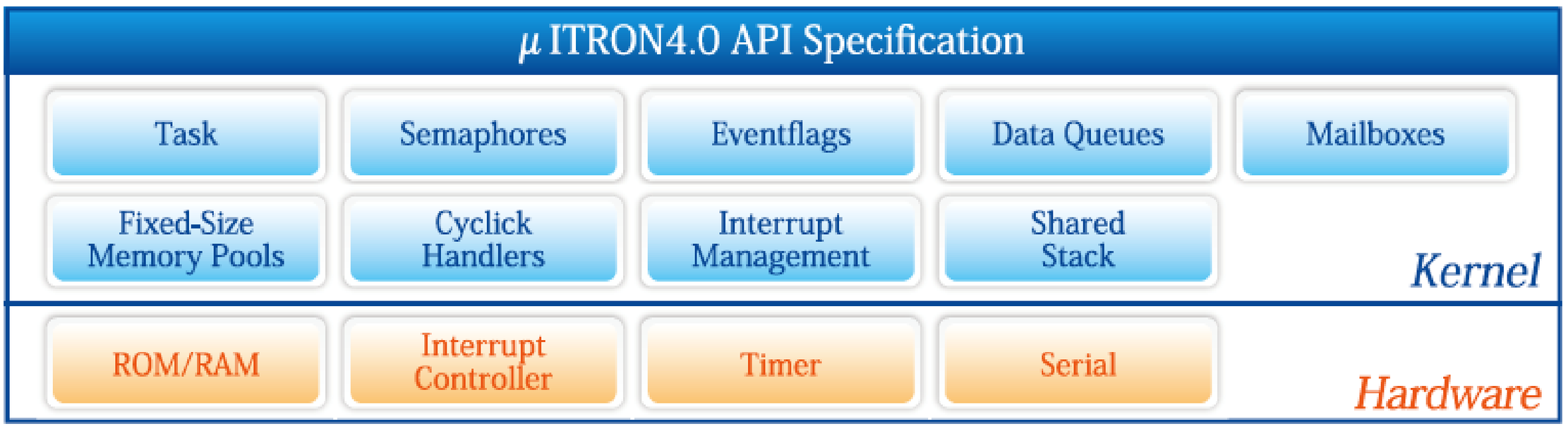
●各メーカリリースのオリジナルマイコンに合わせた製品を準備
- ルネサス(旧日立/旧三菱/旧NEC)用OS
- 富士通
- 東芝
- 他(あくまで一例です)

●各マイコンメーカをターゲットにした製品を準備(3rd Party)
- グレープシステム GRAPE SYSTEMS
- イーソル
- イーフォース eForce
- エーアイコーポレーション
- ミスポ MiSPO 株式会社ミスポ
- 他(あくまで一例です)

資料提供：イー・フォース様

例1：μITRON仕様(イーフォース様の場合)

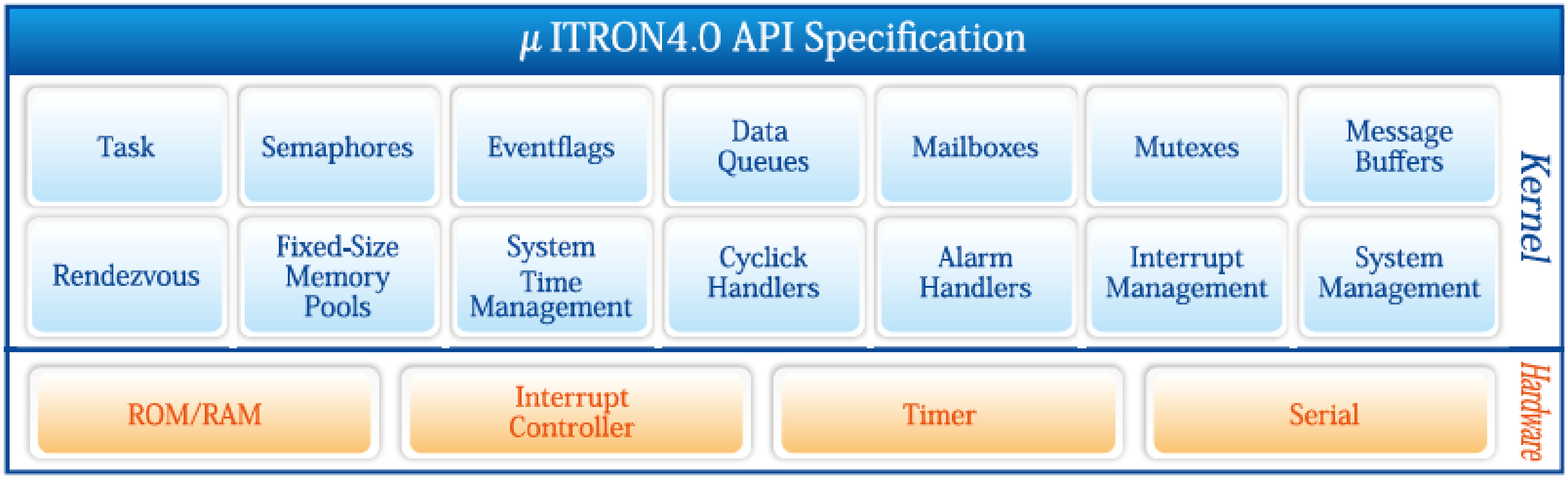
μC3(マイクロ・シー・キューブ)/Compactはマイコン内蔵の小さなメモリだけで動作するように最適化されたコンパクトなμITRON4.0仕様のRTOSです。
ソースコードに直接コンフィグレーションを行うのではなく付属のコンフィグレータによりGUIベースでRTOS、TCP/IP、デバイスのコンフィグレーションからベースコードの自動生成まで行います。
国内では最初にARM Cortex®-Mコアに対応した ITRON仕様のOSで多くの採用実績があります。



資料提供：イー・フォース様

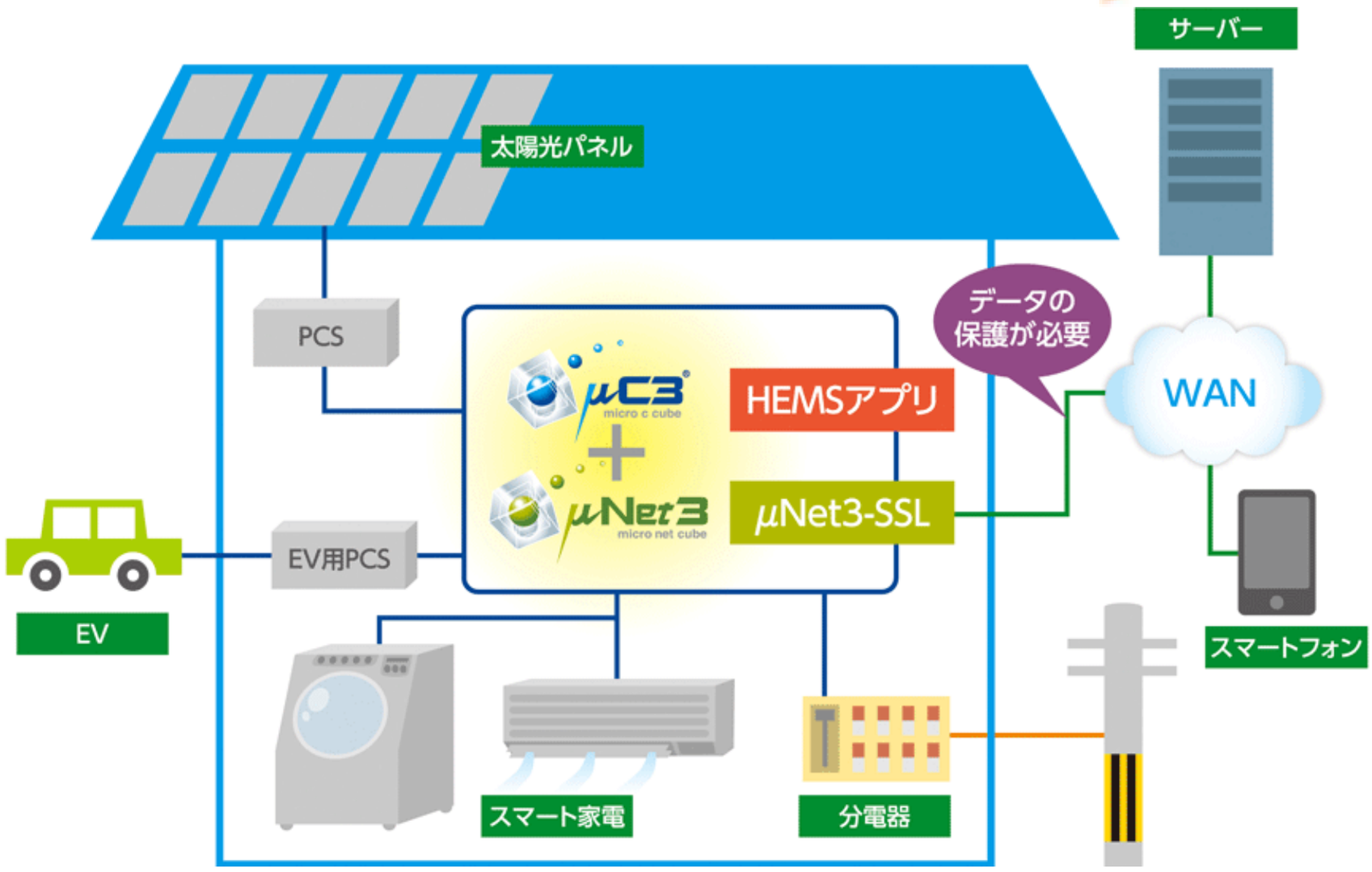
例2：μITRON仕様(イーフォース様の場合)

μC3(マイクロ・シー・キューブ)/StndardはμITRON4.0のスタンダードプロファイルをベースに、32ビットプロセッサが搭載された組込みシステム向けのRTOSです。
高性能プロセッサが、より高度なリアルタイム制御に耐えられるよう、割込み禁止時間を極力なくし、割込み応答性を最重要課題として設計したRTOSです。
ARM Cortex®-Aシリーズ、ARM Cortex®-Rシリーズなど多くの最新プロセッサをサポートしています。



資料提供：イー・フォース様

例1：μITRON採用事例(イーフォース様の場合)



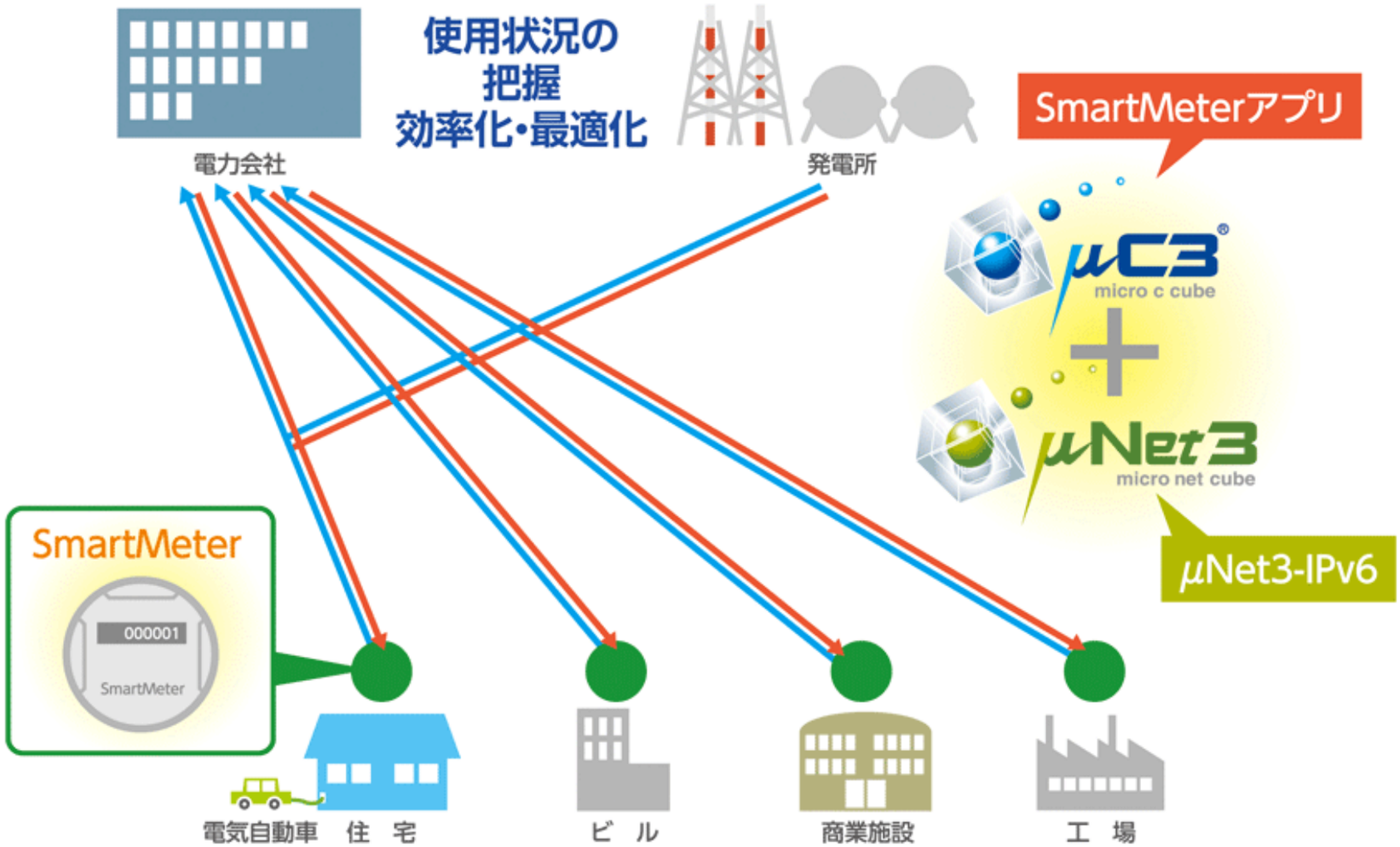
家庭内で家電機器の電力表示を行ったり、遠隔的に運転を制御したり、情報をサーバーに送ったりするHEMS(Home Energy Management System、**家庭内エネルギー管理システム**)。

外部のインターネット網に接続するため、セキュリティ機能が必須でした。

セキュリティ通信を行う場合、従来 Linux**ベースなどの比較的規模が大きなシステムで開発する必要がありましたが、**μNet3-SSL**を使うことで安価な**Cortex-M**マイコンの内蔵メモリのみで**SSL**の通信が実現できました。**

資料提供：イー・フォース様

例2：μITRON採用事例(イーフォース様の場合)



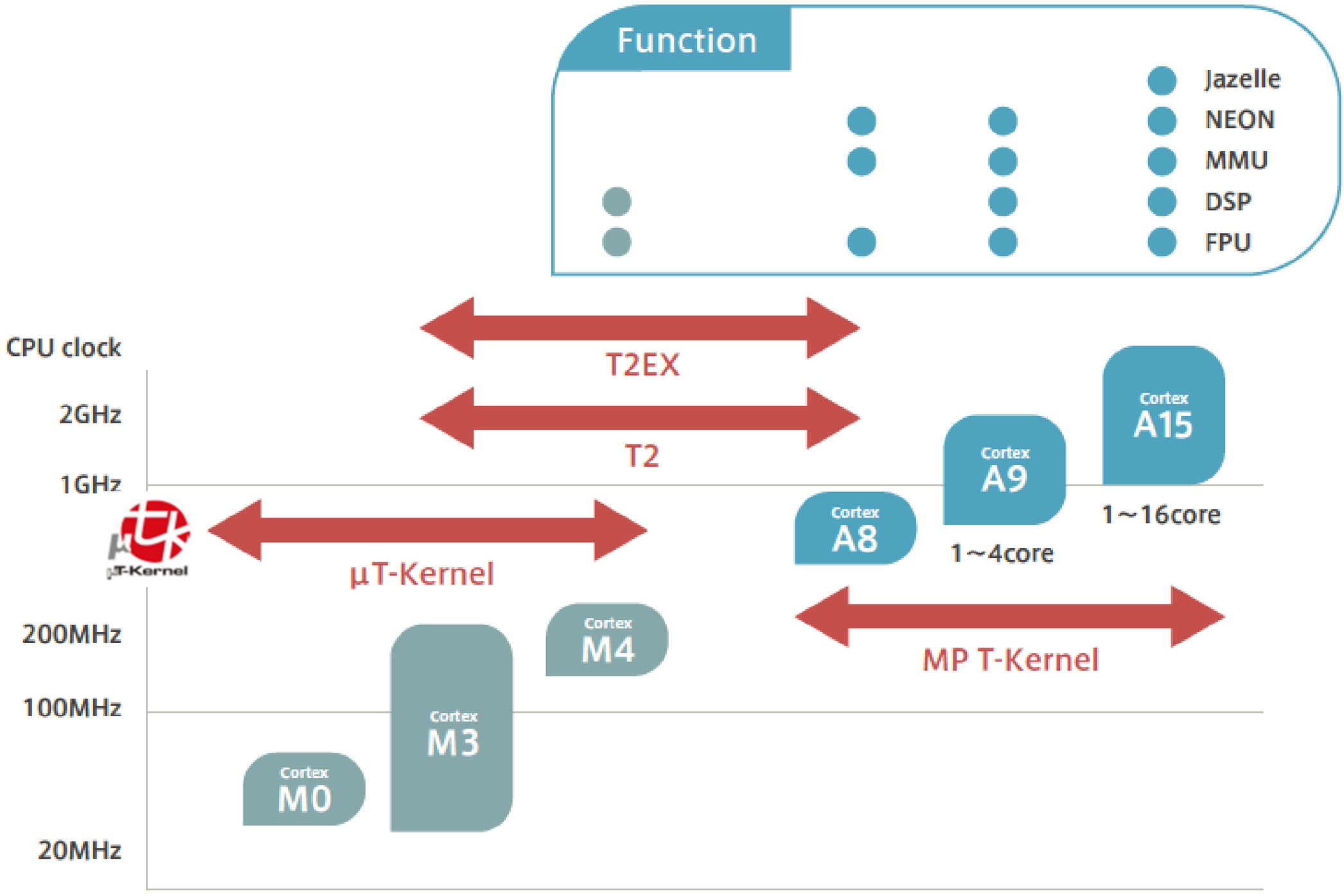
使用状況に合わせた電力やサービスの提供

今後登場する高機能なスマートメーターでは、事業所内や家庭内のエアコンや照明、温度計、セキュリティ機器などの制御まで行うことが構想されています。

スマートメーターはかなりの数の端末装置が、直接インターネットに接続できるようにIPv6の機能が必須でした。

低消費電力でローコストを実現するために、Cortex-Mマイコンと省メモリで動作するμNet3-IPv6が採用されました。

T-Kernel適用範囲(ARM core)



参考:その他の組込み向けOS

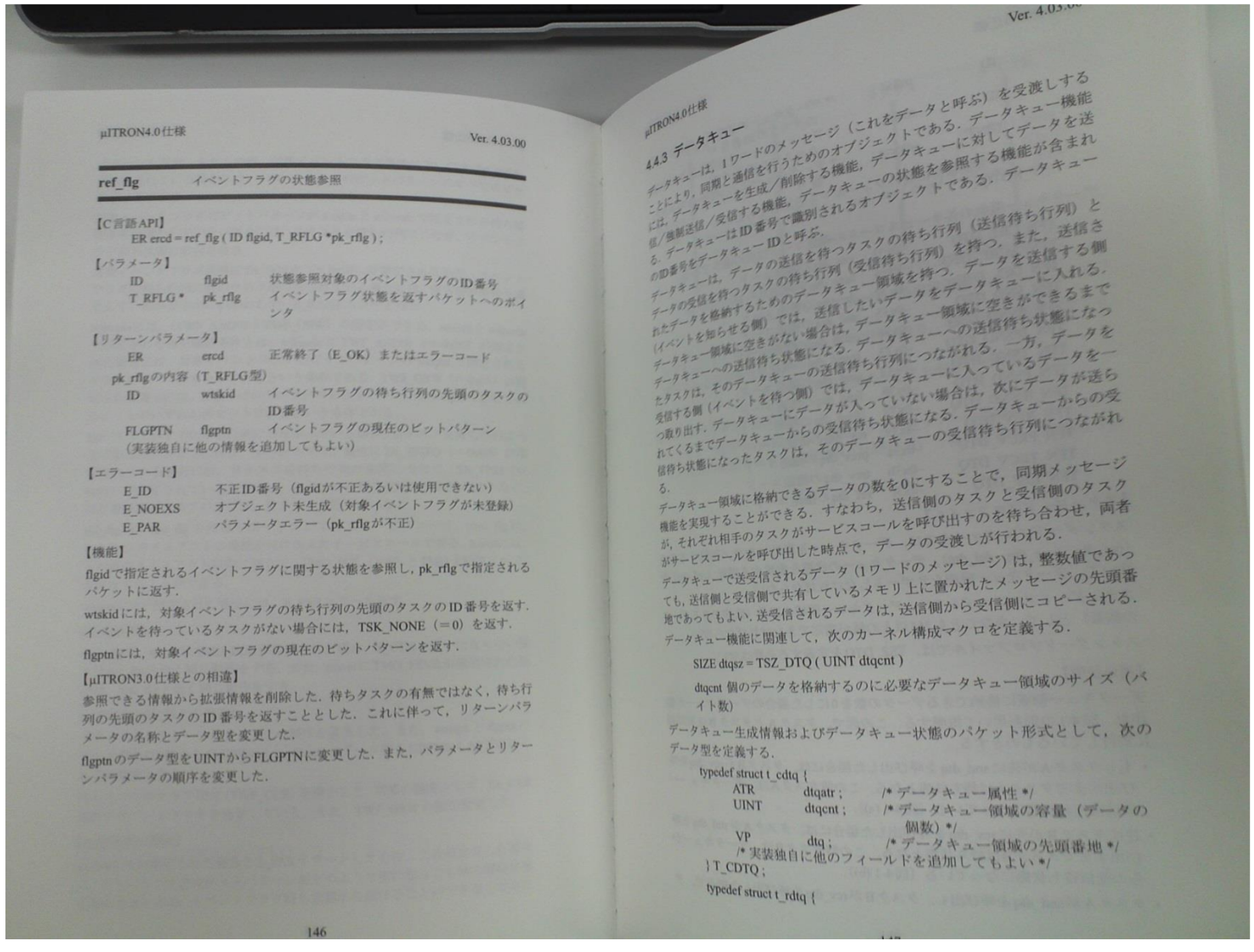
- 組込みLinux
  - UnixクローンOS Linuxの組込み版
- OSEK/VDX
  - 車載向け、欧州
- Symbian OS/Android OS/iOS
  - 携帯電話/スマートフォン向け
- VxWORKS
  - ソフトウェアベンダ系

3 μITRON入門

uITRON4.0仕様の概念と共通定義

uITRON仕様では、基本的な用語の意味やタスク状態、タスクスケジューリング、割込み処理モデルなどが共通の定義として仕様書に記載されています。

ここでは最低限の事項について記載します。



μITRON4.0仕様 Ver. 4.03.00

ref_flg　　イベントフラグの状態参照

【C言語API】
ER ercd = ref_flg (ID flgid, T_RFLG *pk_rflg);

【パラメータ】

ID	flgid	状態参照対象のイベントフラグのID番号
T_RFLG *	pk_rflg	イベントフラグ状態を返すパケットへのポインタ

【リターンパラメータ】

ER	ercd	正常終了（E_OK）またはエラーコード

pk_rflgの内容（T_RFLG型）

ID	wtskid	イベントフラグの待ち行列の先頭のタスクのID番号
FLGPTN	flgptn	イベントフラグの現在のビットパターン

（実装独自に他の情報を追加してもよい）

【エラーコード】

E_ID	不正ID番号（flgidが不正あるいは使用できない）
E_NOEXS	オブジェクト未生成（対象イベントフラグが未登録）
E_PAR	パラメータエラー（pk_rflgが不正）

【機能】
flgidで指定されるイベントフラグに関する状態を参照し，pk_rflgで指定されるパケットに返す．

wtskidには，対象イベントフラグの待ち行列の先頭のタスクのID番号を返す．イベントを待っているタスクがない場合には，TSK_NONE（＝0）を返す．

flgptnには，対象イベントフラグの現在のビットパターンを返す．

【μITRON3.0仕様との相違】
参照できる情報から拡張情報を削除した．待ちタスクの有無ではなく，待ち行列の先頭のタスクのID番号を返すこととした．これに伴って，リターンパラメータの名称とデータ型を変更した．

flgptnのデータ型をUINTからFLGPTNに変更した．また，パラメータとリターンパラメータの順序を変更した．

146

μITRON4.0仕様

4.4.3 データキュー

データキューは，1ワードのメッセージ（これをデータと呼ぶ）を受渡しすることにより，同期と通信を行うためのオブジェクトである．データキュー機能には，データキューを生成／削除する機能，データキューに対してデータを送信／強制送信／受信する機能，データキューの状態を参照する機能が含まれる．データキューはID番号で識別されるオブジェクトである．データキューのID番号をデータキューIDと呼ぶ．

データキューは，データの送信を待つタスクの待ち行列（送信待ち行列）とデータの受信を待つタスクの待ち行列（受信待ち行列）を持つ．また，送信されたデータを格納するためのデータキュー領域を持つ．データを送信する側（イベントを知らせる側）では，送信したいデータをデータキューに入れる．データキュー領域に空きがない場合は，データキューへの送信待ち状態になる．データキューへの送信待ち状態になる．データキューの送信待ち行列につながれる．一方，データを受信する側（イベントを待つ側）では，データキューに入っているデータを一つ取り出す．データキューにデータが入っていない場合は，次にデータが送られてくるまでデータキューからの受信待ち状態になる．データキューからの受信待ち状態になったタスクは，そのデータキューの受信待ち行列につながれる．

データキュー領域に格納できるデータの数を0にすることで，同期メッセージ機能を実現することができる．すなわち，送信側のタスクと受信側のタスクが，それぞれ相手のタスクがサービスコールを呼び出すのを待ち合わせ，両者がサービスコールを呼び出した時点で，データの受渡しが行われる．

データキューで送受信されるデータ（1ワードのメッセージ）は，整数値であっても，送信側と受信側で共有しているメモリ上に置かれたメッセージの先頭番地であってもよい．送受信されるデータは，送信側から受信側にコピーされる．

データキュー機能に関連して，次のカーネル構成マクロを定義する．

SIZE dtqsz = TSZ_DTQ (UINT dtqcnt)

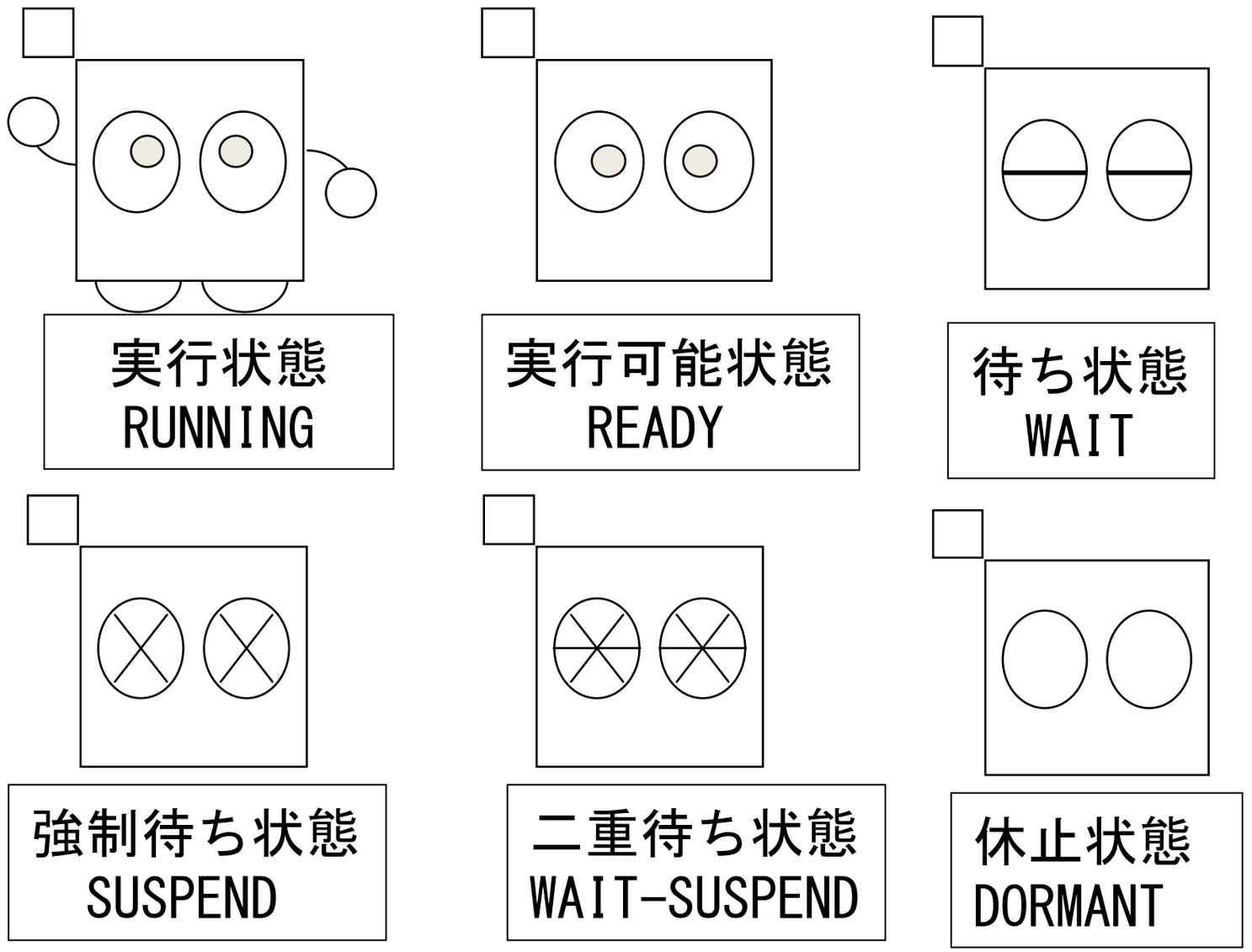
dtqcnt 個のデータを格納するのに必要なデータキュー領域のサイズ（バイト数）

データキュー生成情報およびデータキュー状態のパケット形式として，次のデータ型を定義する．

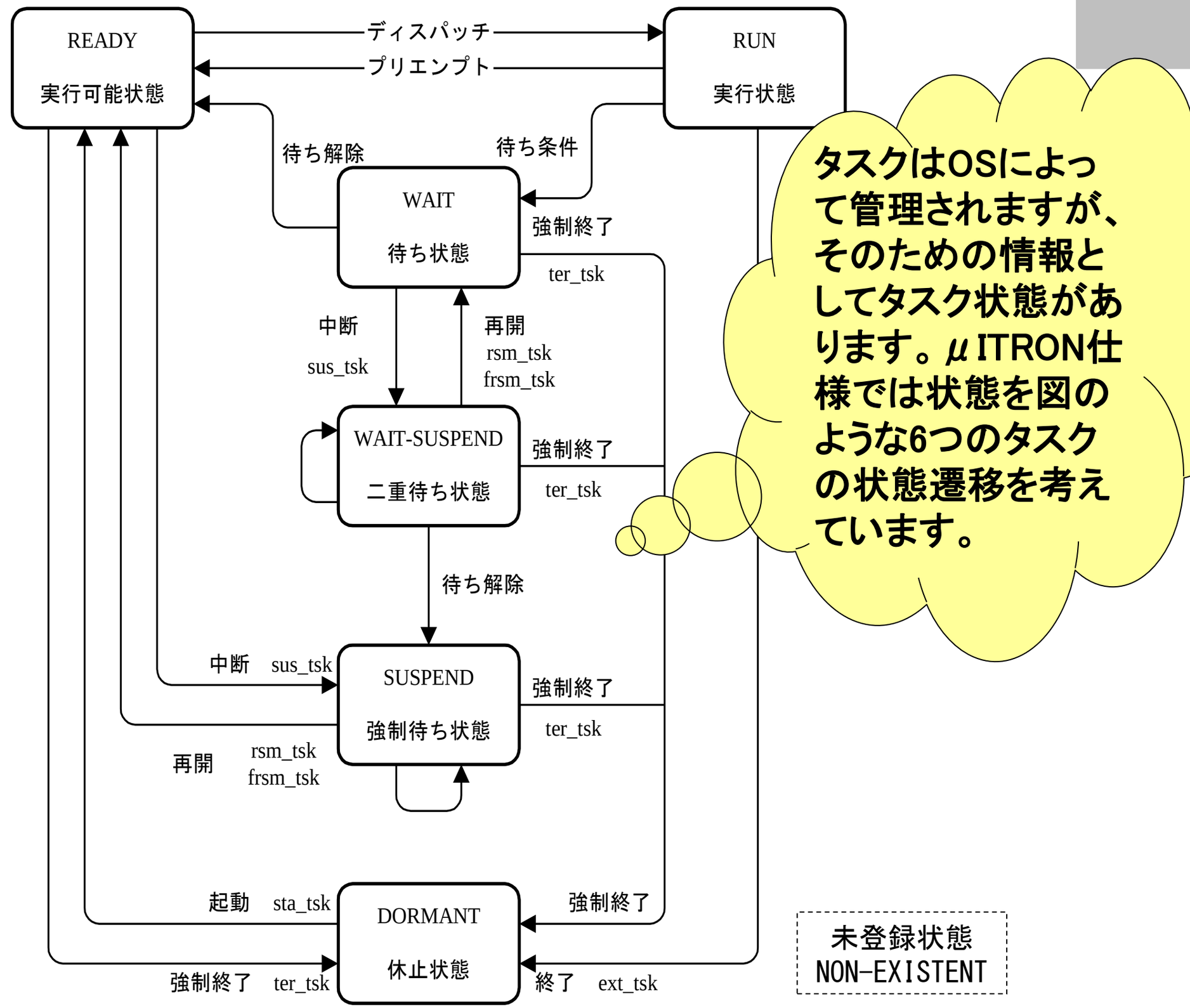
```
typedef struct t_cdtq {
    ATR     dtqatr;     /* データキュー属性 */
    UINT    dtqcnt;     /* データキュー領域の容量（データの
                           個数）*/
    VP      dtq;        /* データキュー領域の先頭番地 */
    /* 実装独自に他のフィールドを追加してもよい */
} T_CDTQ;

typedef struct t_rdtq {
```

タスクとは

- RTOS上で最小実行単位となるサブルーチン(関数)
- 各タスクには優先度を指定
- タスクには以下の状態があり、順次切り替えて並行動作を行う(タスクスケジューリング)
  - ■ 実行可能状態
  - ■ 実行状態
  - ■ 待ち状態
  - ■ 強制待ち状態
  - ■ 二重待ち状態
  - ■ 休止状態
  - ■ 未登録状態



タスクの状態遷移図



タスクはOSによって管理されますが、そのための情報としてタスク状態があります。μITRON仕様では状態を図のような6つのタスクの状態遷移を考えています。

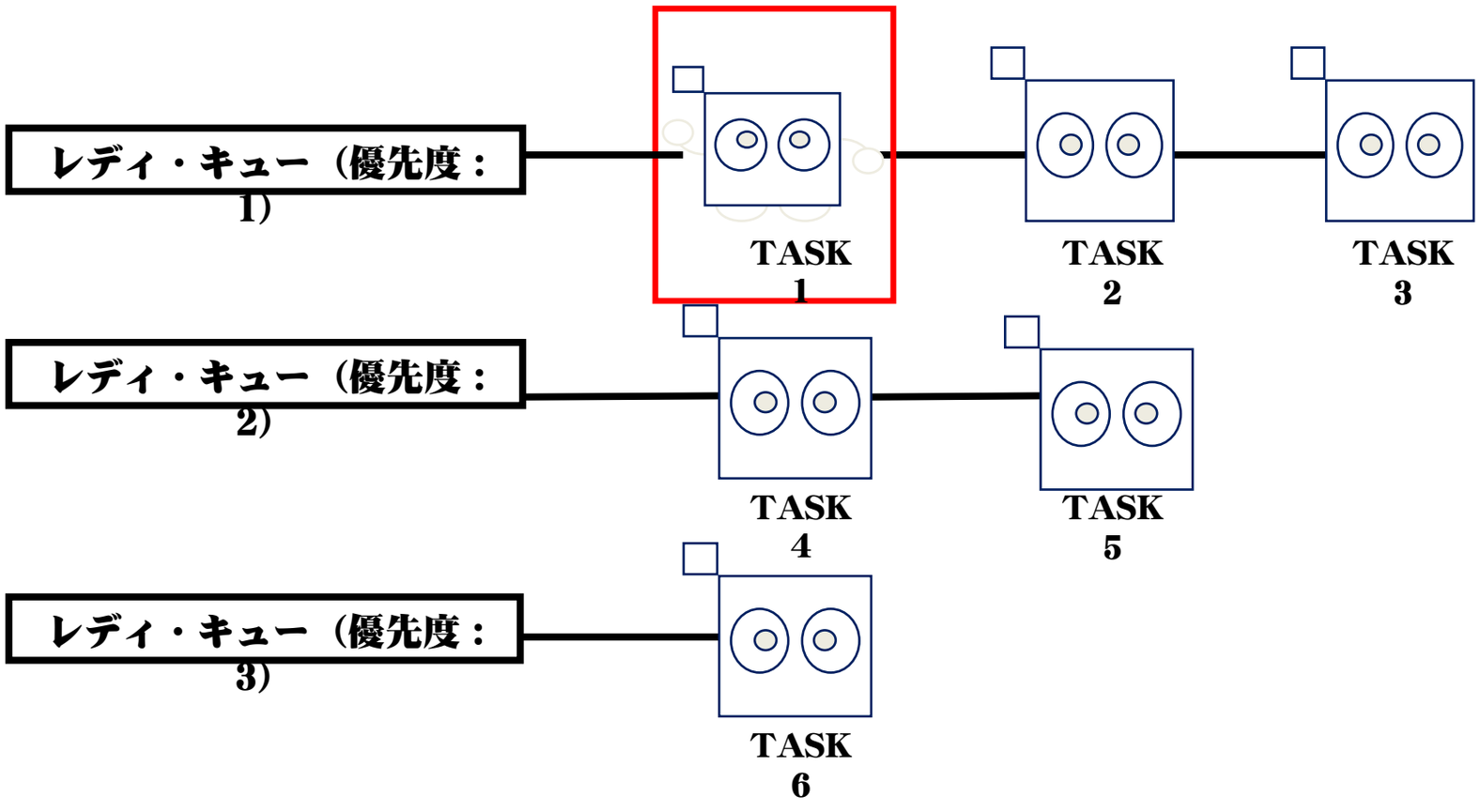
タスクの状態(1)

- 実行状態（RUNNING）
  - 現在タスクが実行中の状態
  - 実行可能状態のタスクの中で最も優先順位の高いタスクの状態
- 実行可能状態（READY）
  - sta_tsk等により実行可能になった状態
- 待ち状態（WAITING）
  - 待ち解除の条件が満たされるのを待っている状態
  - 条件が満たされると実行可能状態に移行
- 強制待ち状態（SUSPENDED）
  - 他タスクから強制的に中断された状態
  - スケジューリングの対象から強制的にはずされたタスクの状態

タスクの状態(2)

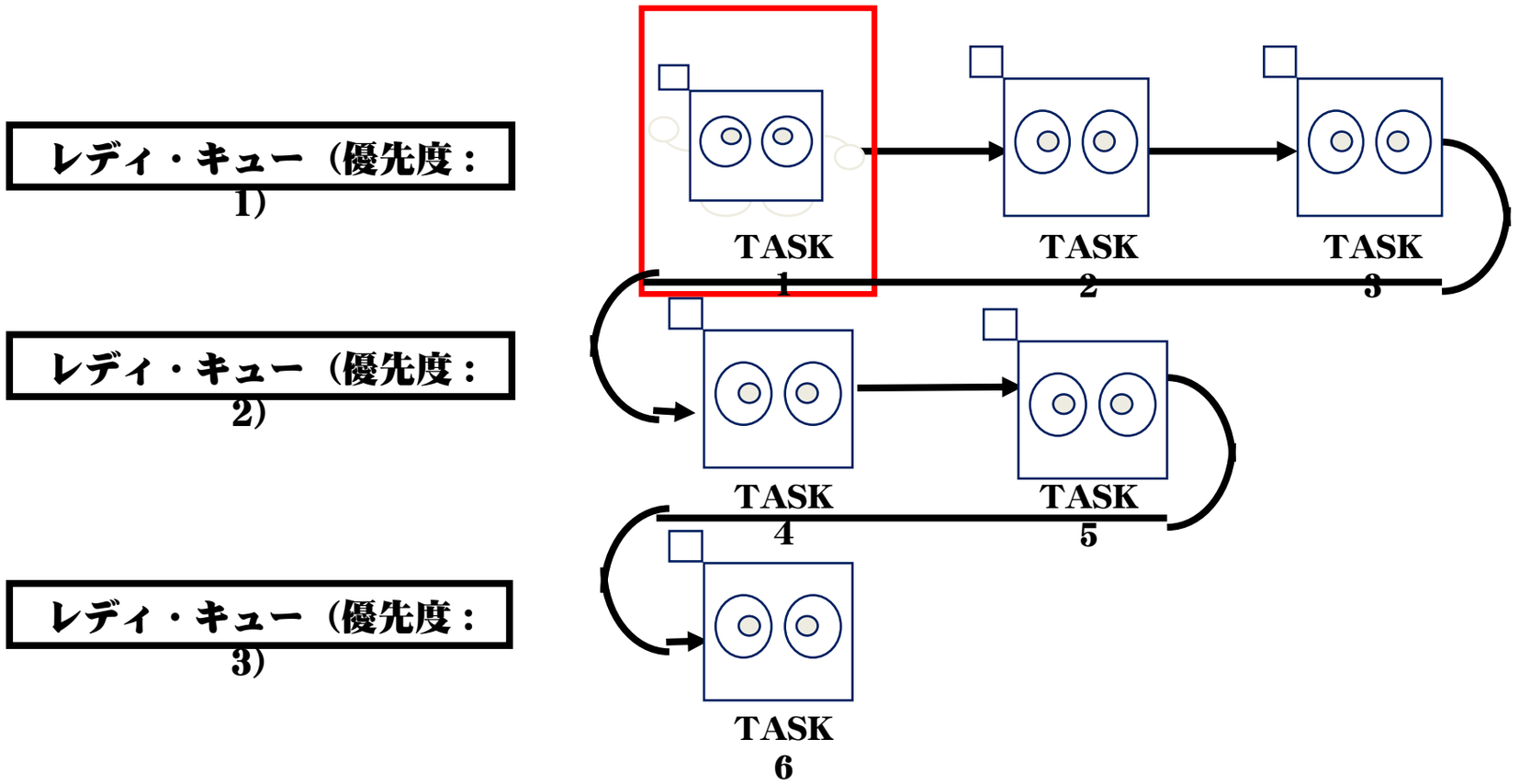
- 二重待ち状態（WAITING-SUSPENDED）
  - 待ち状態と強制待ち状態の2つの待ち状態が重なった状態
  - 待ち状態のタスクに対して、sus_tskが発行された状態
- 休止状態（DORMANT)
  - タスクが起動されるのを待っている状態
  - タスクの起動前、および タスク終了後の状態
- 未登録状態（NON-EXISTENT)
  - タスクが登録されていない状態
  - システムから削除された状態

レディ･キュー



- 実行可能状態にあるタスクのつながる行列
- 最も優先順位の高いタスクが実行状態になる

タスクの実行順序



- 優先順（優先度の数字が小さい順）
- 同じ優先度なら先にレディ・キューに並んだ順（FIFO順）

タスク動作以外の規定

- 用語、ID番号、優先度、エラーコード等
- ITRON共通規定として用意されている
  - ソフトウェア部品仕様にも共通に適用可能
  - 規定を理解しておくことで、仕様全体も理解しやすい(プログラム可読性も向上)
- μITRON3.0/4.0/T-Kernelにおいて、基本的な概念や扱いは共通

正の数で値が小さいほど優先度が高い、など

タスクの動きを勉強したい方は・・・

トロンフォーラムのサイトから、[セミナー]→[TTVタスクシミュレータ]を選択

TTVタスクシミュレータ画面

API名称の決まり

サービスコールで使用されている略語

　ITRON仕様、T-Kernel仕様のカーネルのサービスコールの名称は，xxx_yyy の形を基本としており、xxxで操作の方法，yyyで操作の対象をあらわしています。xxx,yyyの識別名は2～4文字程度の略語が使用されています。以下に代表的なものを示します。

略語	元になった英語
can	cancel
chg	change
clr	clear
cre	create
del	delete
slp	sleep
dtq	data queue
sem	semaphore
tsk	task

　その他の略語と元になった英語については、μITRON 4.0仕様書の「2.2 APIの名称に関する原則」にありますので、くわしくは仕様書を参照してください。

μITRON4.0仕様の機能(サービスコール)

- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- 割り込み管理機能
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

※赤字を中心に記載。

μITRON4.0仕様の機能(サービスコール)

- **タスク管理機能**
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- 割り込み管理機能
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

タスク管理機能

タスクの状態を直接的に操作/参照する機能

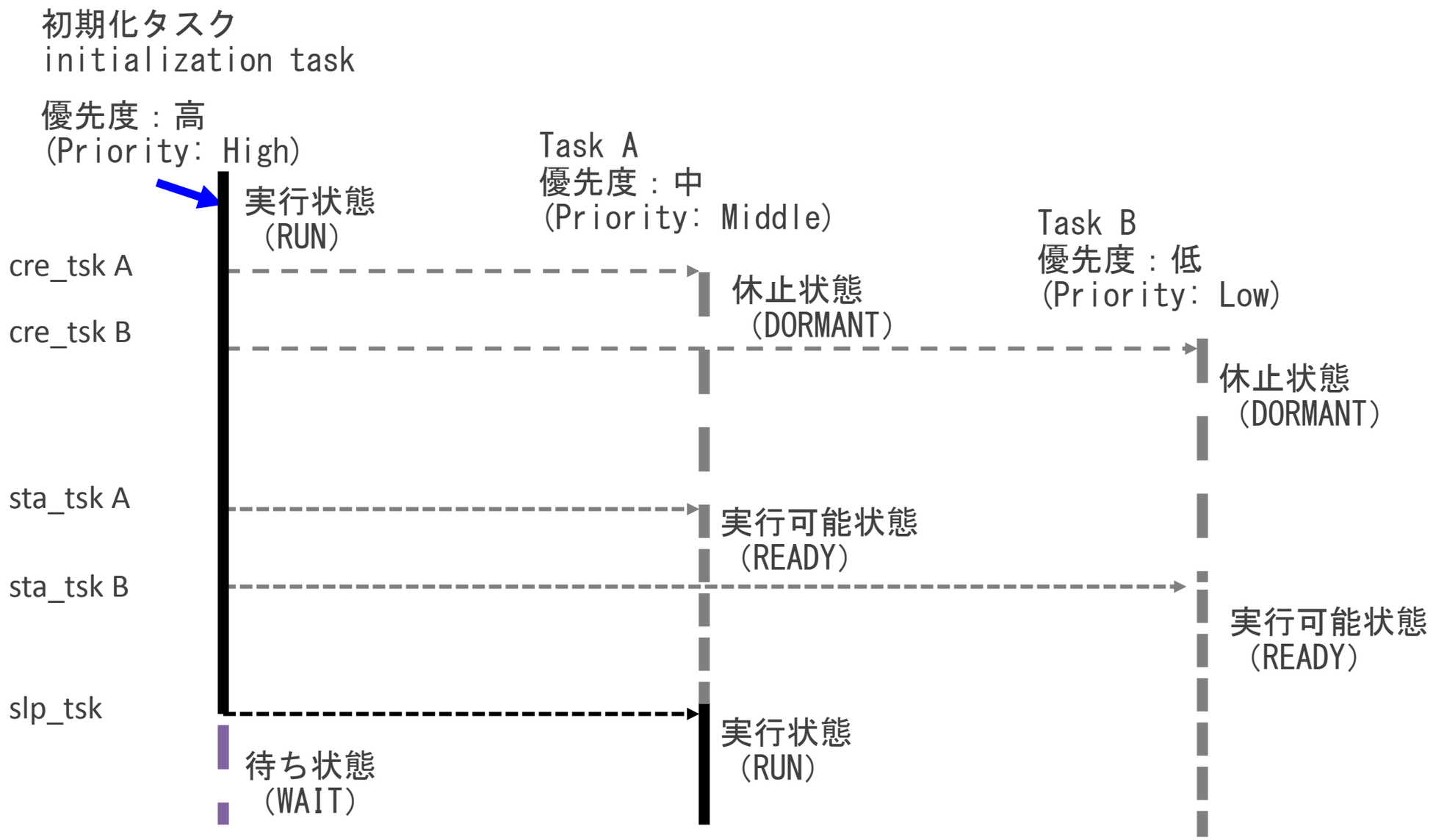
cre_tsk()	タスク生成
del_tsk()	タスク削除
sta_tsk()	タスク起動
ext_tsk()	自タスク終了
exd_tsk()	自タスク終了と削除
ter_tsk()	他タスク強制終了
ref_tsk()	タスク状態参照

初期化タスクでの処理例

```
{
  ER        ercd;
  T_CTSK    ctsk_A, ctsk_B;                 /* タスク生成情報 */
  ID        tskid_A, tskid_B;               /* タスクID       */
     ここでタスク構造体 ctsk_A, ctsk_B を設定
  tskid_A = cre_tsk(&ctsk_A);               /* タスク生成     */
  tskid_B = cre_tsk(&ctsk_B);
  ercd  = sta_tsk(tskid_A, mbxId);          /* タスク起動     */
  ercd  = sta_tsk(tskid_B, mbxId);
  ercd  = slp_tsk(TMO_FEVR);        /* タスクスリープ */
}
```

※エラー処理は入っていません。

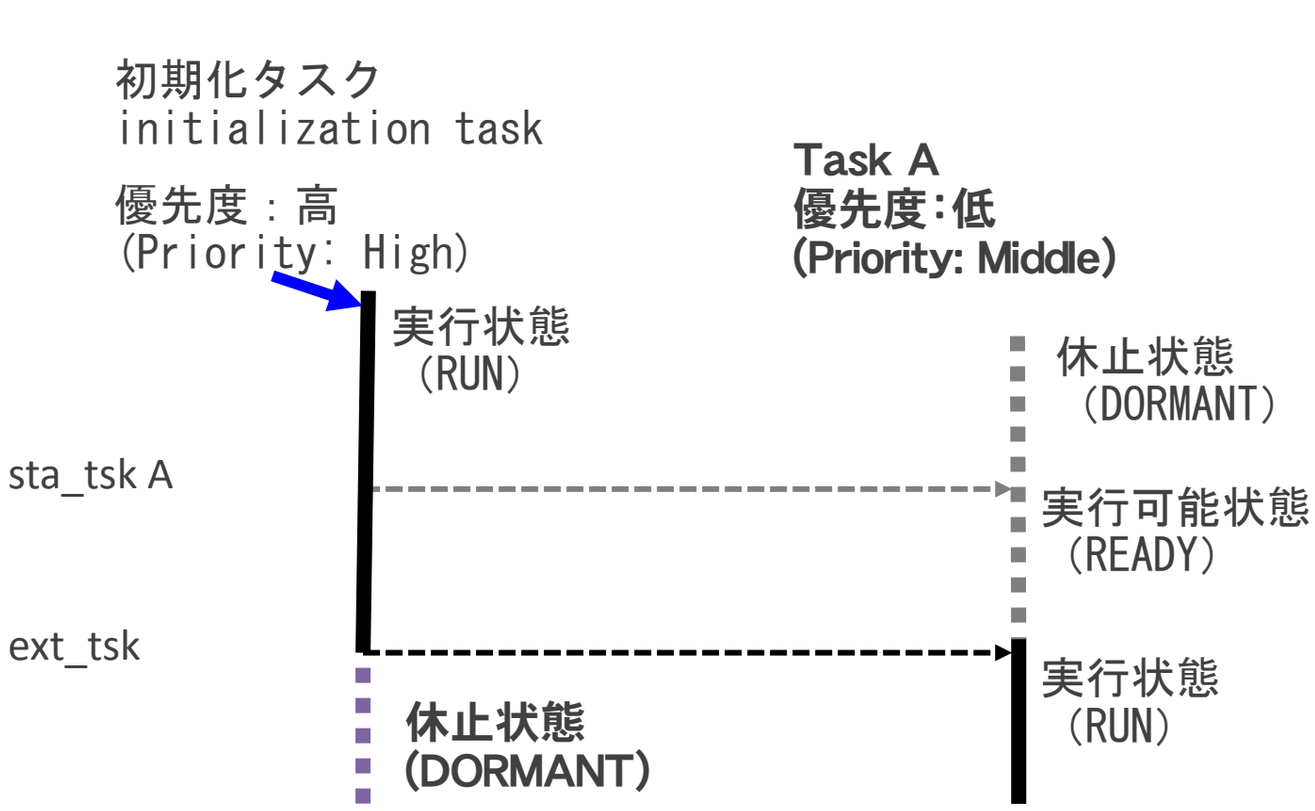
初期化タスクの流れ



タスク管理機能

```
{
  sta_tsk(TaskA);          /* タスク起動      */
  ext_tsk(Initial); /* タスクスリープ  */
}
```

※エラー処理は入っていません。



ユーザが作成したタスクを実行するには、まずタスクを起動し、**READY**(実行可能)状態にします。具体的には、起動したい対象タスクに対して**sta_tsk**を発行し、**DORMANT**(休止)状態のタスクを**READY**状態にします。そして、**READY**状態になったタスクは、**RTOS**によりスケジューリングされ、ディスパッチとプリエンプトを行います。

実行中(**RUN**状態)のタスクを終了、つまり**DORMANT**状態にするには実行中のタスクの中で**ext_tsk**を発行します。

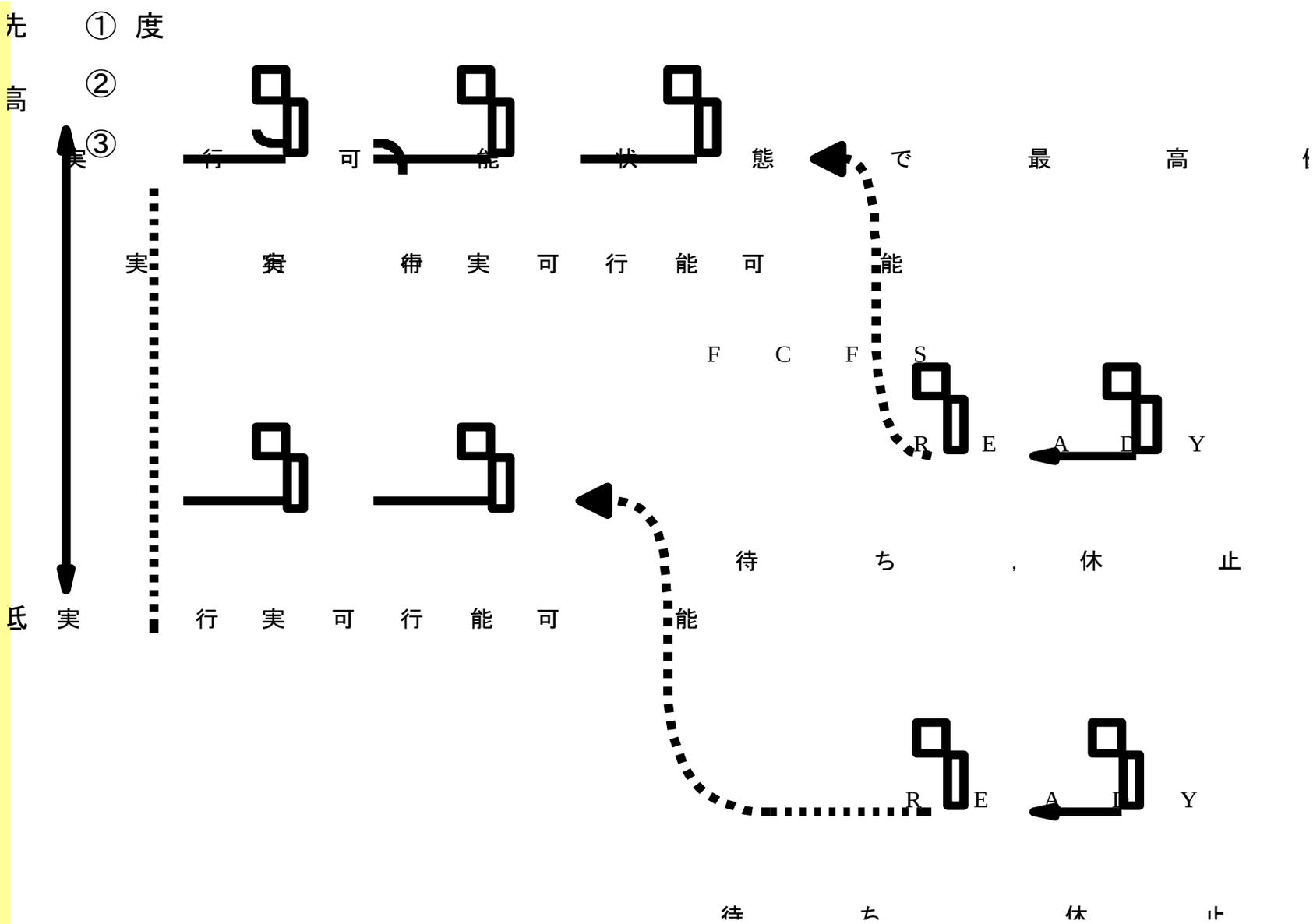
タスク実行中に異常が発生し、タスクの実行を中断しなければならないときは、**ter_tsk**を発行して、他のタスクを強制的に終了させることができます。これは、**READY**、**WAIT**、**WAIT-SUSPEND**または**SUSPEND**状態のタスクを**DORMANT**状態にします。

タスク管理機能(ご参考)

エラーや特定のイベント発生に従って、あらかじめ決めておいたタスクの実行順序を変更したい場合、次の2つの方法があります。

chg_priは、対象タスクに対して、システム起動時に指定されているタスクの優先度を一時的に変更するとき発行します。変更された優先度は、そのタスクが終了するまで有効となります。

rot_rdqシステムコールを発行すると、指定された優先度のタスクのレディキューを回転させることができます。レディキューが回転すると、先頭のタスクが最後尾につなぎ変えられます。また、このシステムコールで、自タスクの優先度を指定した場合には、自タスク自身がレディキューの最後尾につながれます。



指定したタスク(他タスク)の現在の優先度とタスク状態を知りたい場合には、**ref_tsk**を使用します。このシステムコールを発行すると、指定タスクの優先度とタスク状態がリターンパラメータとして返却されます。

μITRON4.0仕様の機能(サービスコール)

- タスク管理機能
- **タスク付属同期機能**
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- 割り込み管理機能
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

タスク付属同期機能

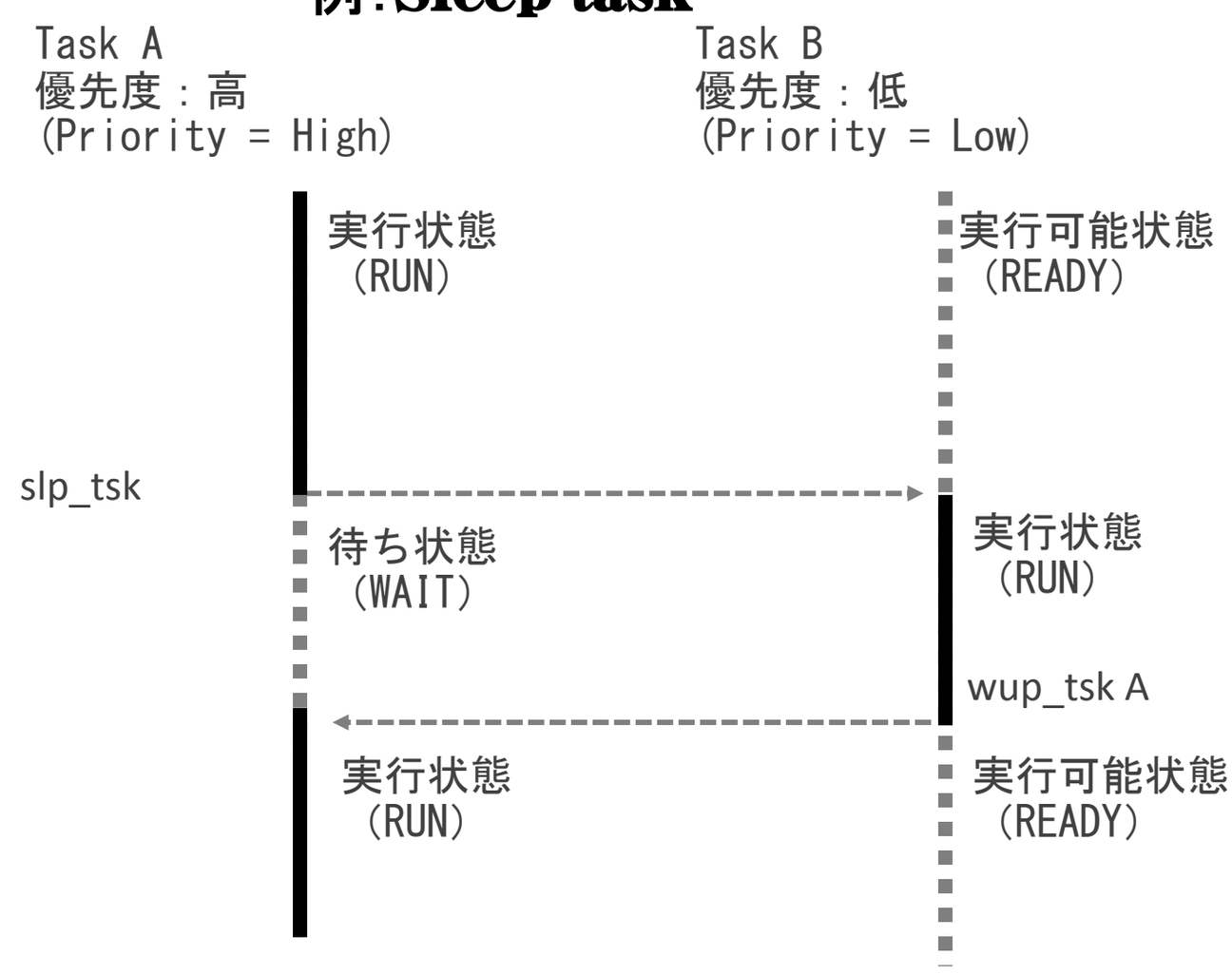
タスク状態を直接操作して同期を行う機能

slp_tsk()	自タスクを起床待ち状態へ移行
wup_tsk()	他タスクの起床
dly_tsk()	タスク遅延

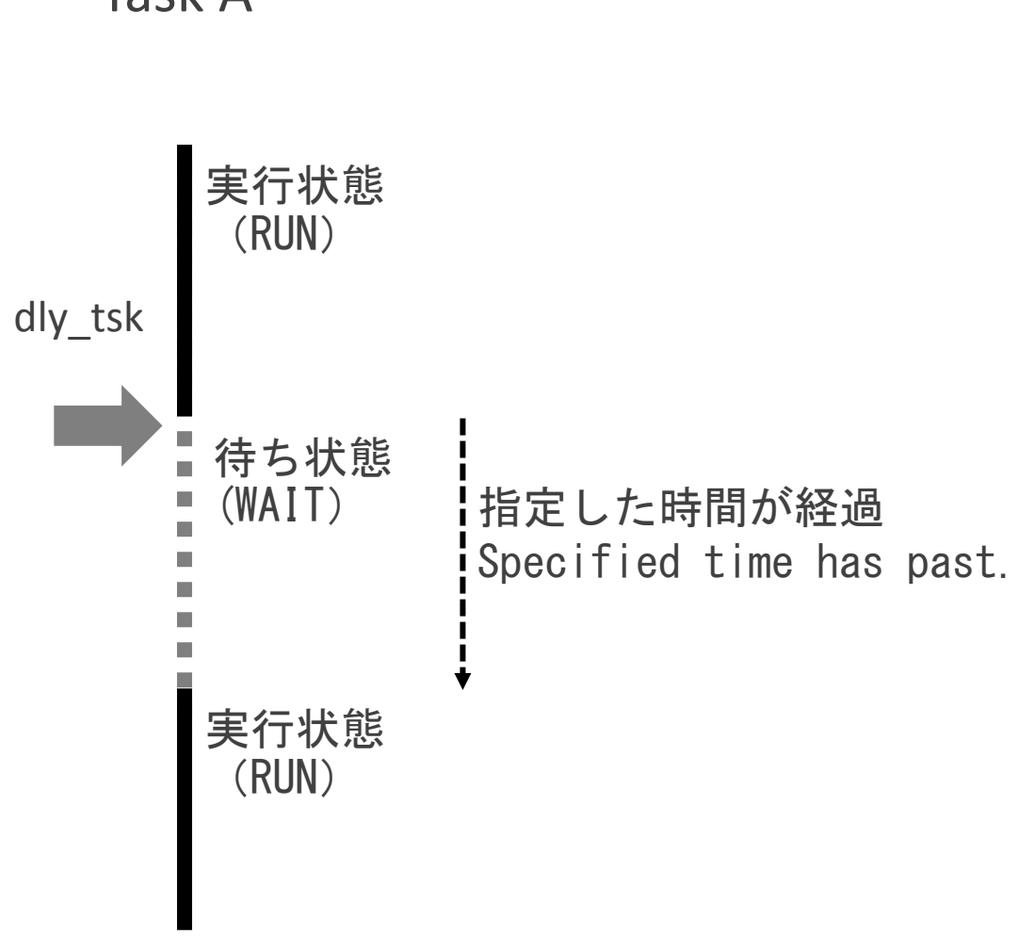
タスク付属同期機能

slp_tskはタスクの仕事が一段落して、実行中の自タスクを停止させWAIT状態にするときに発行します。
wup_tskはWAIT状態にあるタスクをREADY状態に、つまり、起床要求を出すときに発行します。
dly_tskは実行中の自タスクを一定時間WAIT状態に、つまり、一定時間止めたい場合に使用します。指定した時間が経過すると自動的にWAIT状態が解除されます。

例：Sleep task



例：Delay task



μITRON4.0仕様の機能(サービスコール)

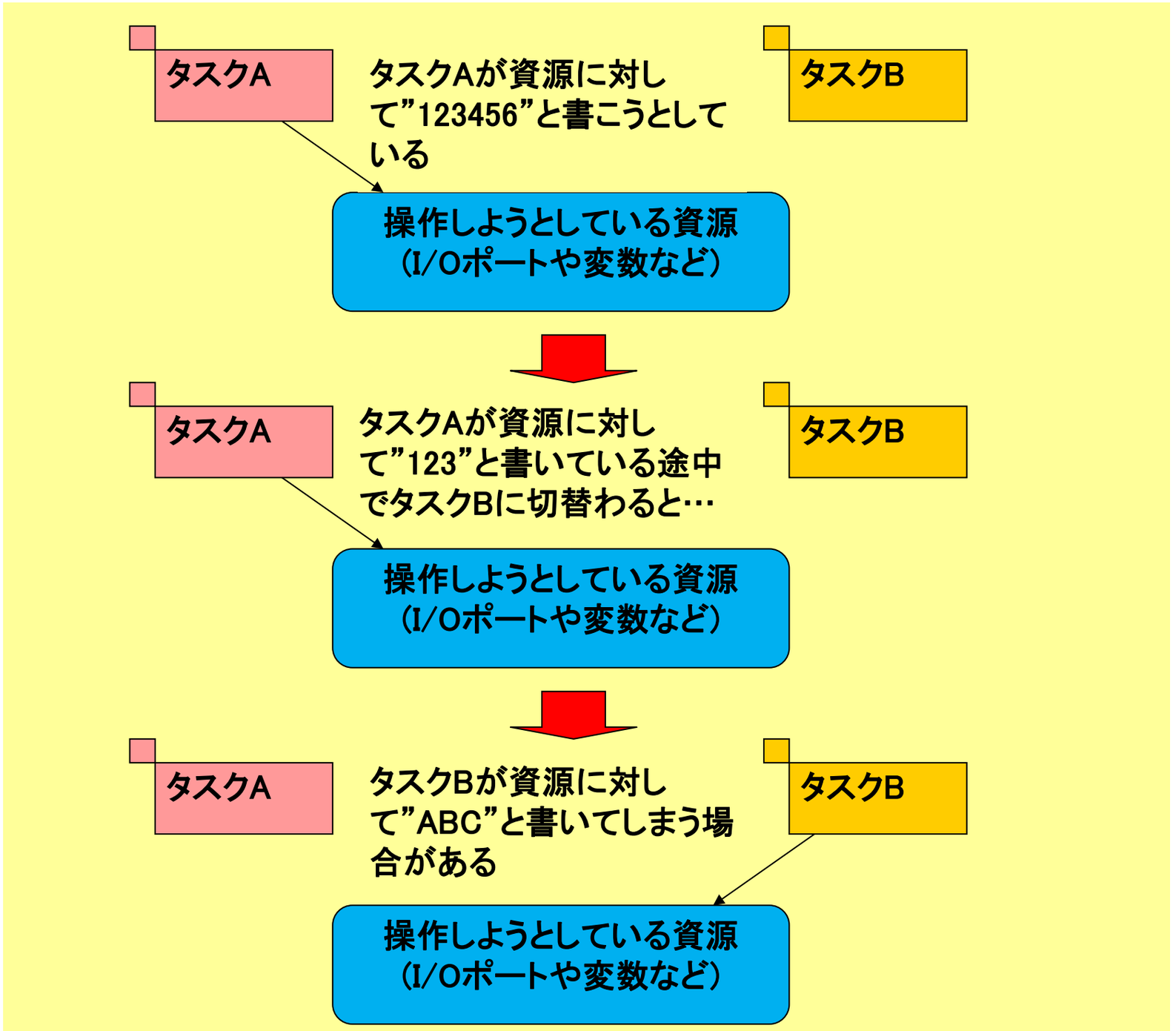
- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- **同期・通信機能**
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- 割り込み管理機能
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

同期・通信機能

マルチタスク処理を行っていると、タスク間の情報交換や同期を取ることが必要になることがあります。これを実現するのがタスク間通信・同期機能です。このために、セマフォ機能、メイルボックス機能、イベントフラグ機能などを備えています。
ユーザはこの機能を使って、タスク間において、信号やデータの送受信、同期を取るといったことを簡単に実現します。

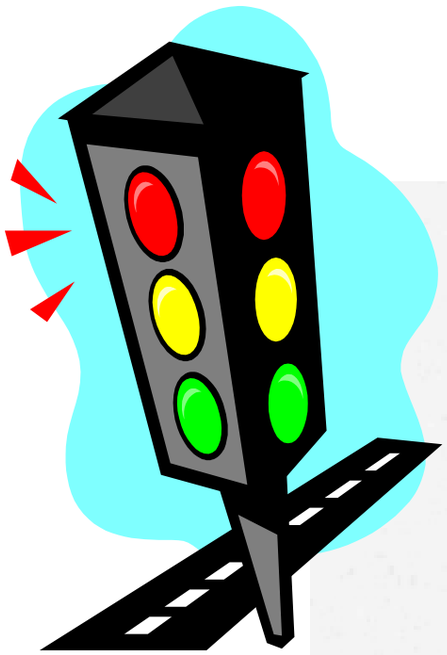
- セマフォ(Semaphore)
  - 利用可能な共有資源の数をカウンタで表し、共有資源の排他制御を行うオブジェクト
- メールボックス(Mail Box)
  - 共有メモリ上に置かれたメッセージを受け渡しして、同期通信を行うオブジェクト
- イベントフラグ(Event Flag)
  - イベントの有無をビット毎のフラグで表現することにより同期するオブジェクト
- データキュー(Data queues)
  - 1ワードのメッセージを受け渡しすることにより、同期と通信を行うためのオブジェクト

排他制御とは

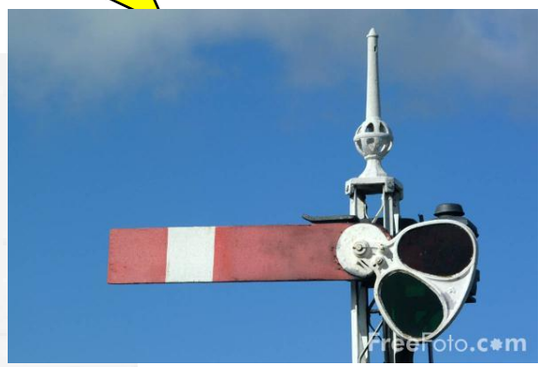


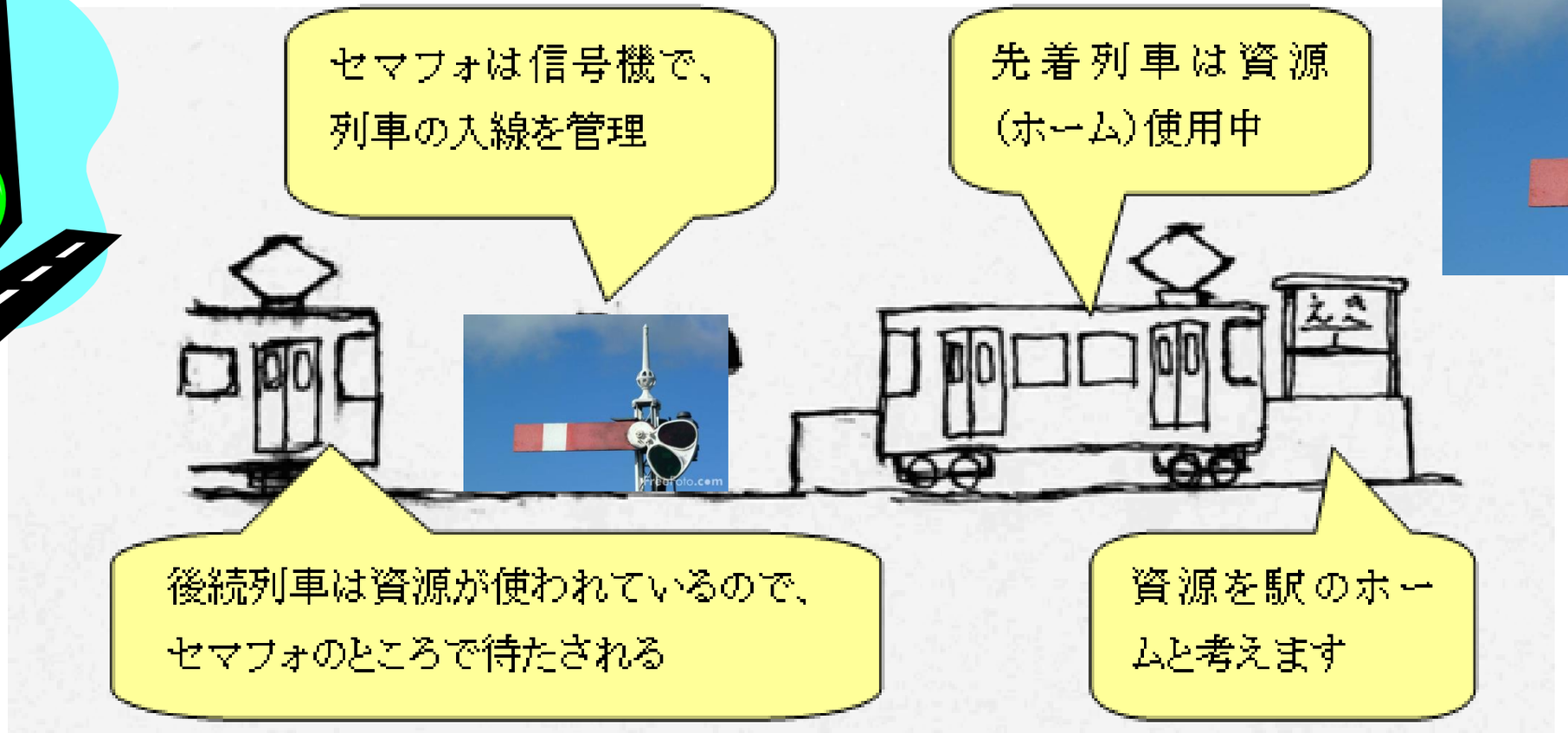
結果として、資源には”123ABC”と書いてしまうことになり、意図した結果になりません。このような場合は排他制御が必要になります。

同期・通信機能（セマフォ）



腕木式信号機、この辺だと新橋のSL広場で見ることができます。





セマフォの語源は、その昔、線路の脇にあった腕木式信号機のことです。これは線路のポイントのところにあり、列車の入線を管理していました。つまり1番線しかホームのない駅があったとすると、資源は1つ、ここに列車が入線していればセマフォのカウンタはデクリメントされ0となります。新規に到着した列車はカウンタ値が0(つまり資源の数が0)なので、セマフォのところで待たされます。

　そして、すでに入線している列車が発車したら、セマフォのカウントはインクリメントされて、セマフォのところで待っていた列車は1番線に入線できる、つまり資源を使うことができる、ということになります。こうして使える資源カウントをすることがセマフォの基本的動作です。

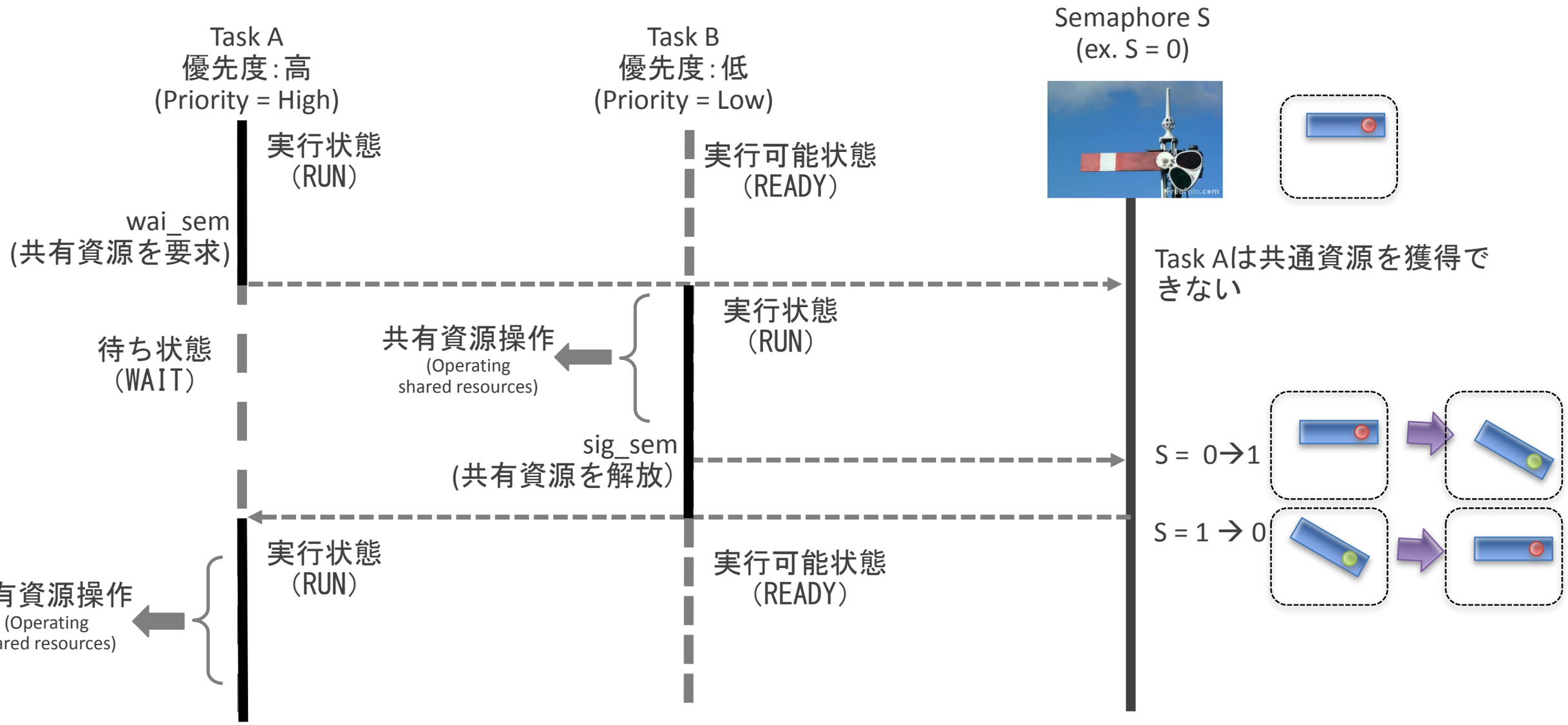
同期・通信機能（セマフォ）

セマフォは、利用可能な共有資源の数をカウンタで表し、共有資源の排他制御を行うオブジェクトです。

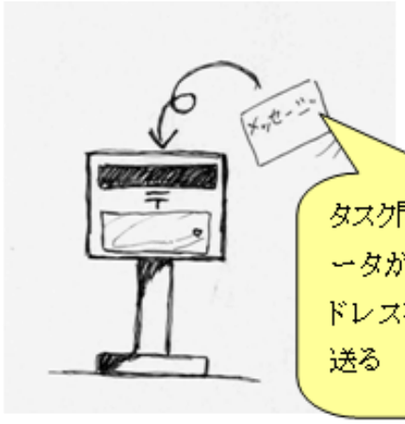
cre_sem()	セマフォ生成(静的API)
acre_sem()	セマフォ生成(ID番号自動割付け)
del_sem()	セマフォ削除
sig_sem()/isig_sem()	セマフォ資源返却
wai_sem()/pol_sem()/twai_sem()	セマフォ資源獲得

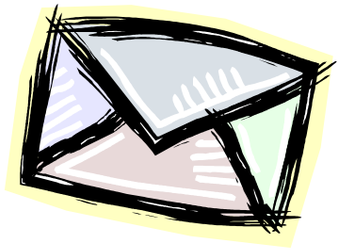
（使い方）資源を利用する前に、利用する資源の数だけセマフォのカウンタから獲得し、終わると返却します。カウンタが必要な数を持ってない場合、他タスクから返却されるのを待つことで、共有資源の排他制御を実現します。

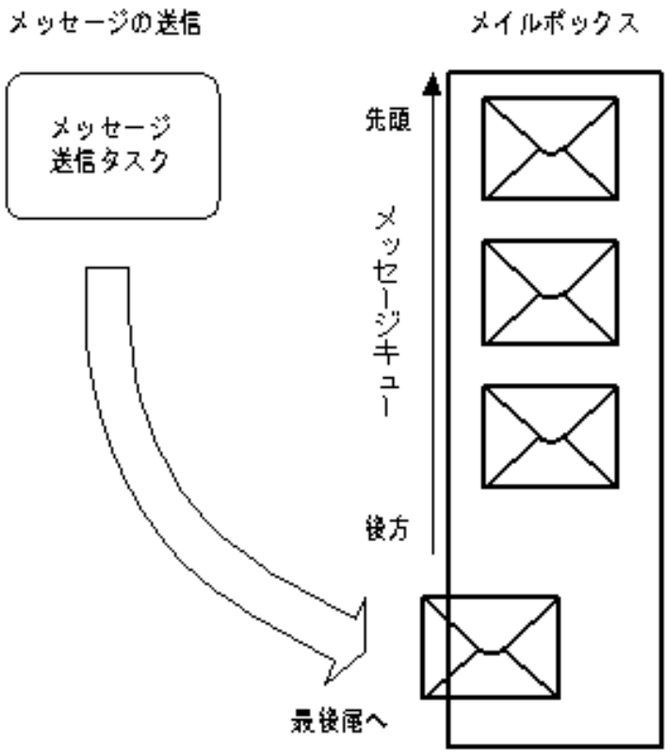
セマフォ（Semaphore）の実行例

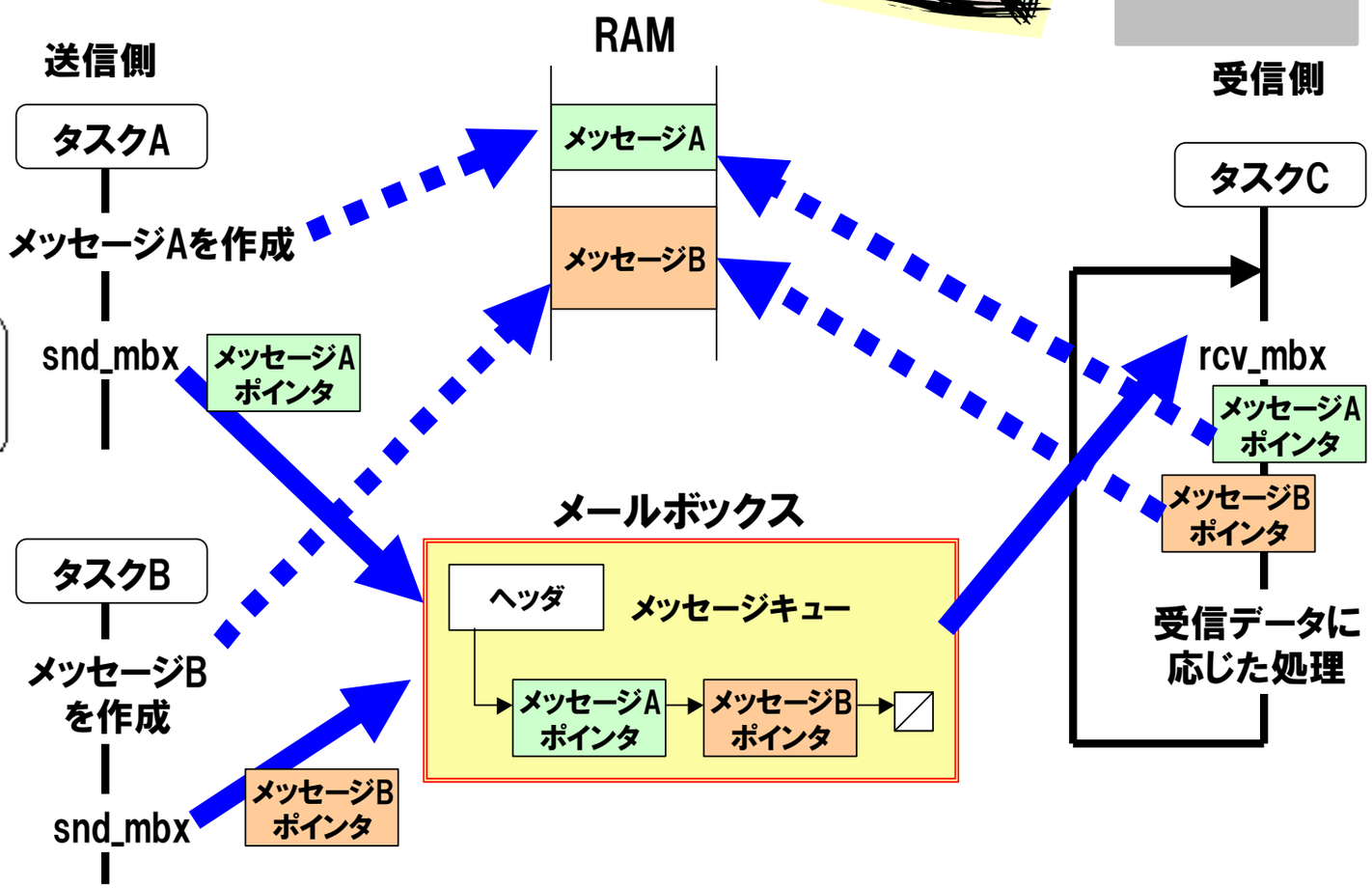


同期・通信機能（メールボックス）









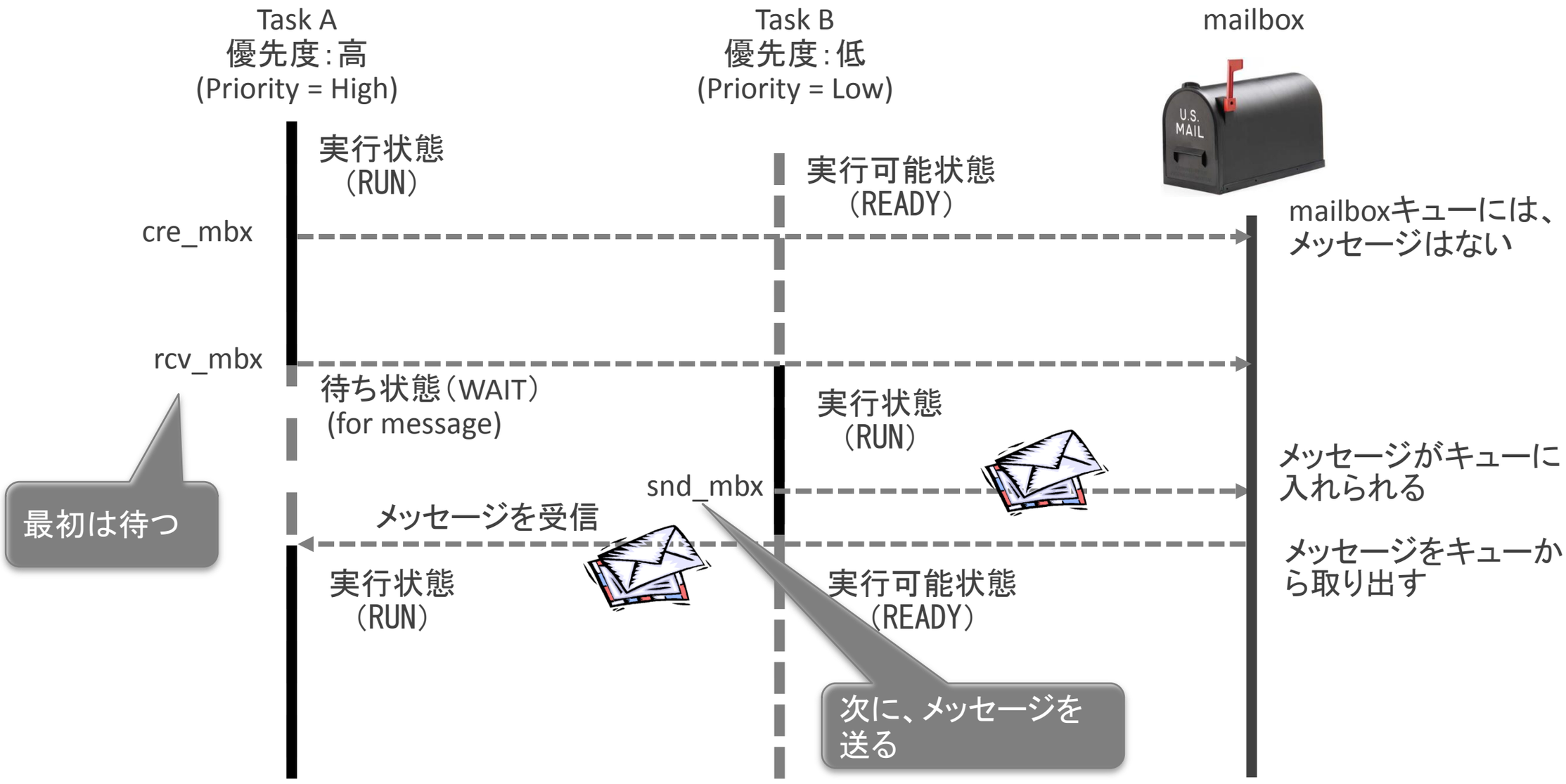
これは普段みなさんが使っている電子メールのメールボックスと同じ意味です。つまり、送信相手がタスクで、メール内容(メッセージ)がメイルボックスに入って送信相手に送られる、と書けばわかりやすいと思います。
マイコンをご存知であれば、タスク間でやり取りするメッセージとはひょっとしてデータではないかな？　と、気付くと思いますが、半分は正解です。
　メイルボックスには、実はデータでなく、そのデータが格納されているアドレスが入ります。
　もう少し掘り下げますと、メールボックスを使うと、任意のサイズのメッセージをタスク間でやりとりすることができます。メールボックスはメッセージのポインタのみが受け渡されるため、メッセージサイズが大きくなっても処理速度が低下しないという特長があります。

同期・通信機能（メールボックス）　間接型

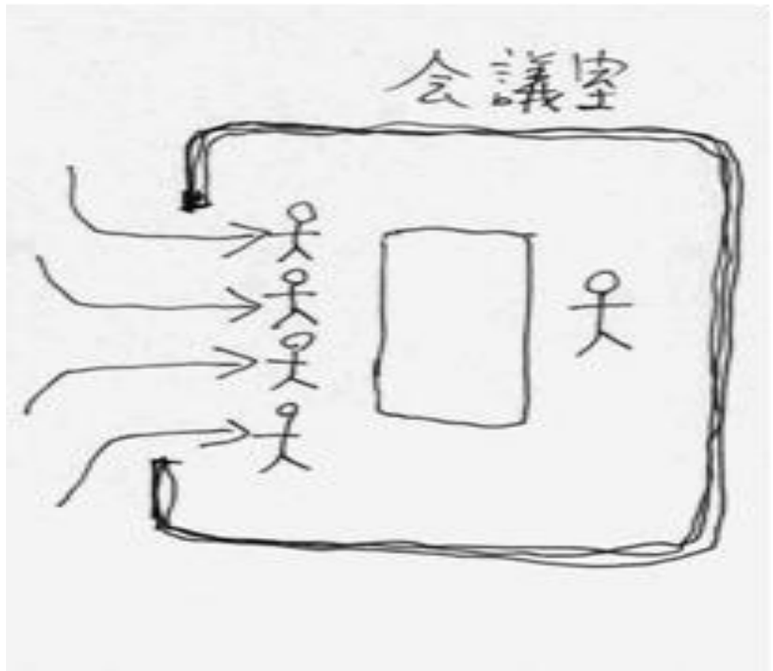
メールボックスは、共有メモリ上に置かれたメッセージを受け渡しして、同期通信を行うオブジェクト

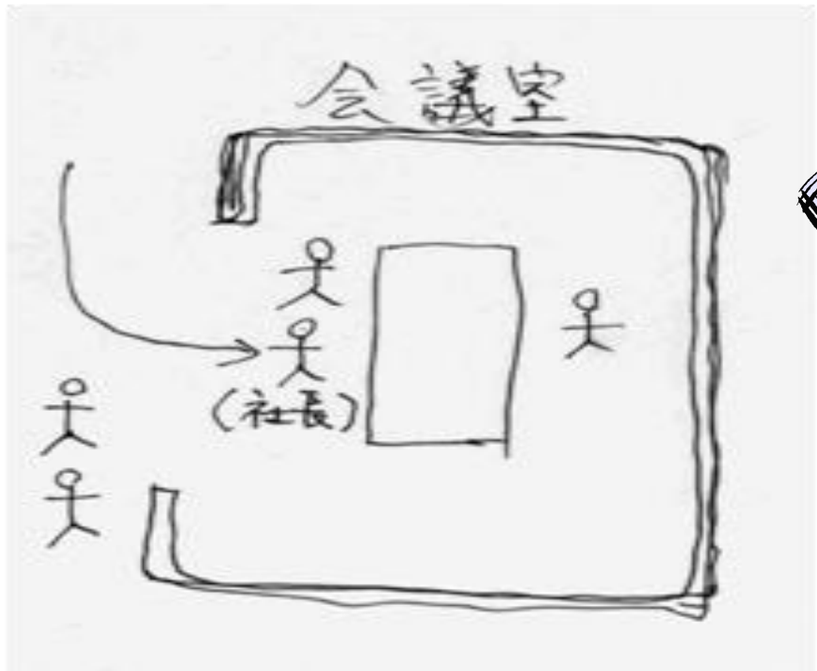
cre_mbx()	メールボックス生成(静的API)
acre_mbx()	メールボックス生成(ID番号自動割付け)
del_mbx()	メールボックス削除
snd_mbx()	メールボックスへ送信
rcv_mbx()/prcv_mbx()/trcv_mbx()	メールボックスから受信

メールボックス（Mailbox）の実行例

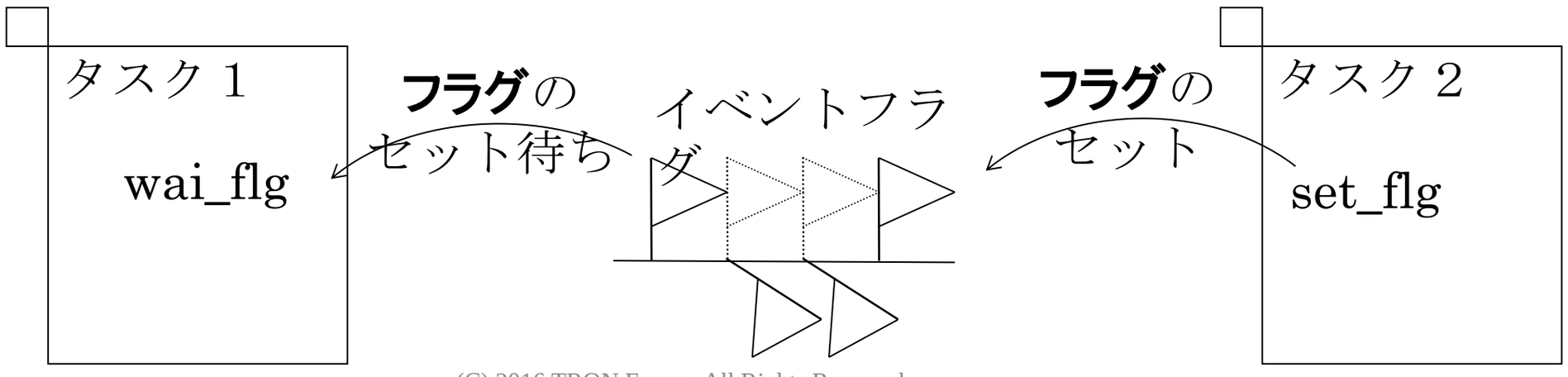


同期・通信機能（イベントフラグ）





イベントフラグは、タスクからタスクへイベントの発生を知らせるために使用します。
例えば重役会議があったとします。4人揃ったら会議開始と主催者は考えています。つまり主催者は4ビット分のイベントフラグが立てば会議を開始します。そして4人揃った、つまりフラグが4ビット立ったことを認識して、主催者は会議を開始します。このとき、4人目の人が主催者に会議開始というシステムコールを発行すればいいような気もしますが、4人目の人は会議室に到着した時点で自分が最後とは思っていないため、システムコールを発行することが出来ないのです。これがイベントフラグの基本的な動作です。
**　このように、4人揃ったらという考え方でイベントフラグを待つことをAND(論理積)待ち(左図)、4人のうち1人(社長)だけくれば会議開始という待ち方をOR(論理和)待ち(右図)とします。**

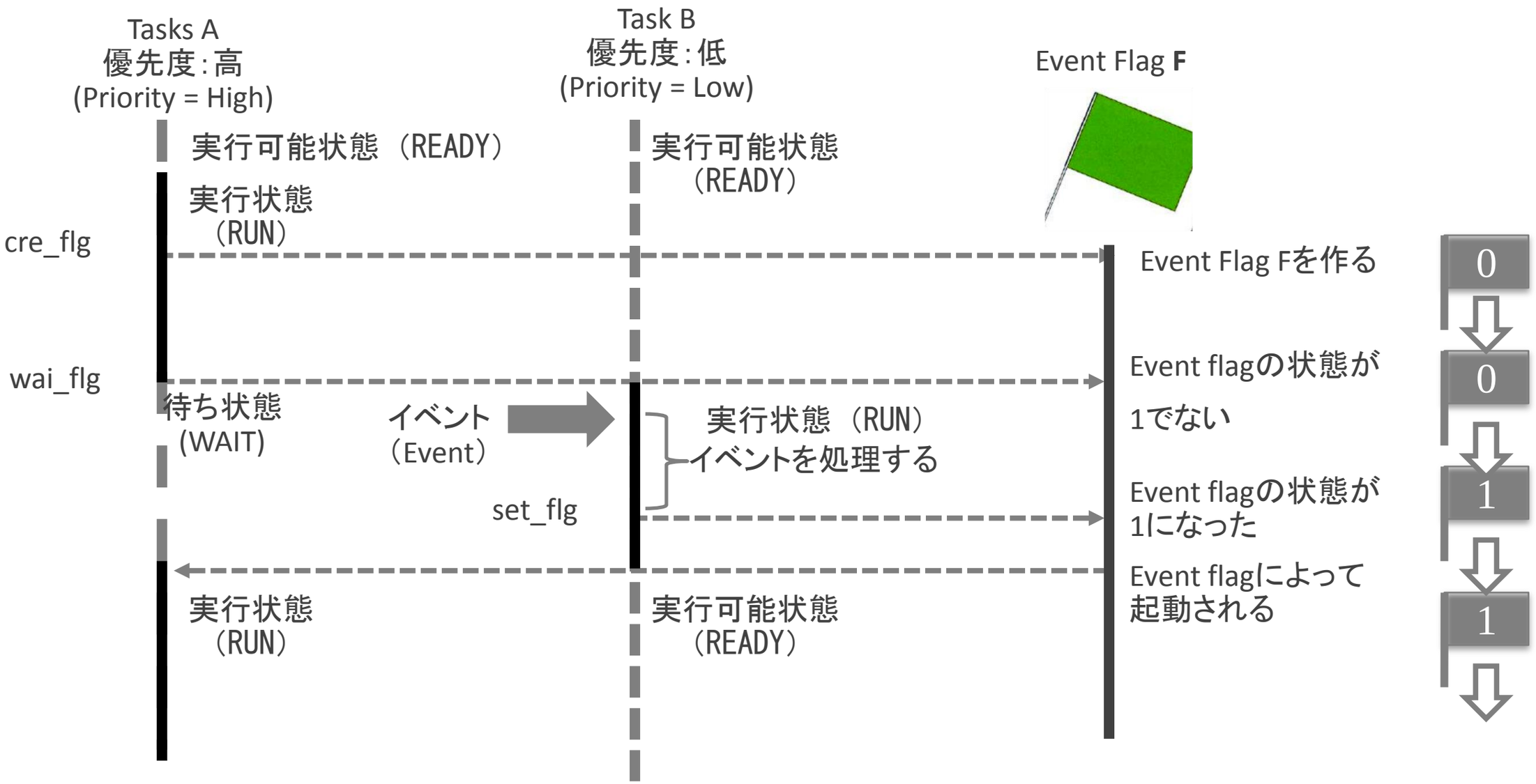


同期・通信機能（イベントフラグ）

イベントフラグは、イベントの有無をビット毎のフラグで表現することにより同期するオブジェクト

cre_flg()	イベントフラグ生成(静的API)
acre_flg()	イベントフラグ生成(ID番号自動割付け)
del_flg()	イベントフラグ削除
set_flg()/iset_flg()	イベントフラグセット
clr_flg()	イベントフラグクリア
wai_flg()/pol_flg()/twai_flg()	イベントフラグ待ち

イベントフラグ（Event Flag）の実行例

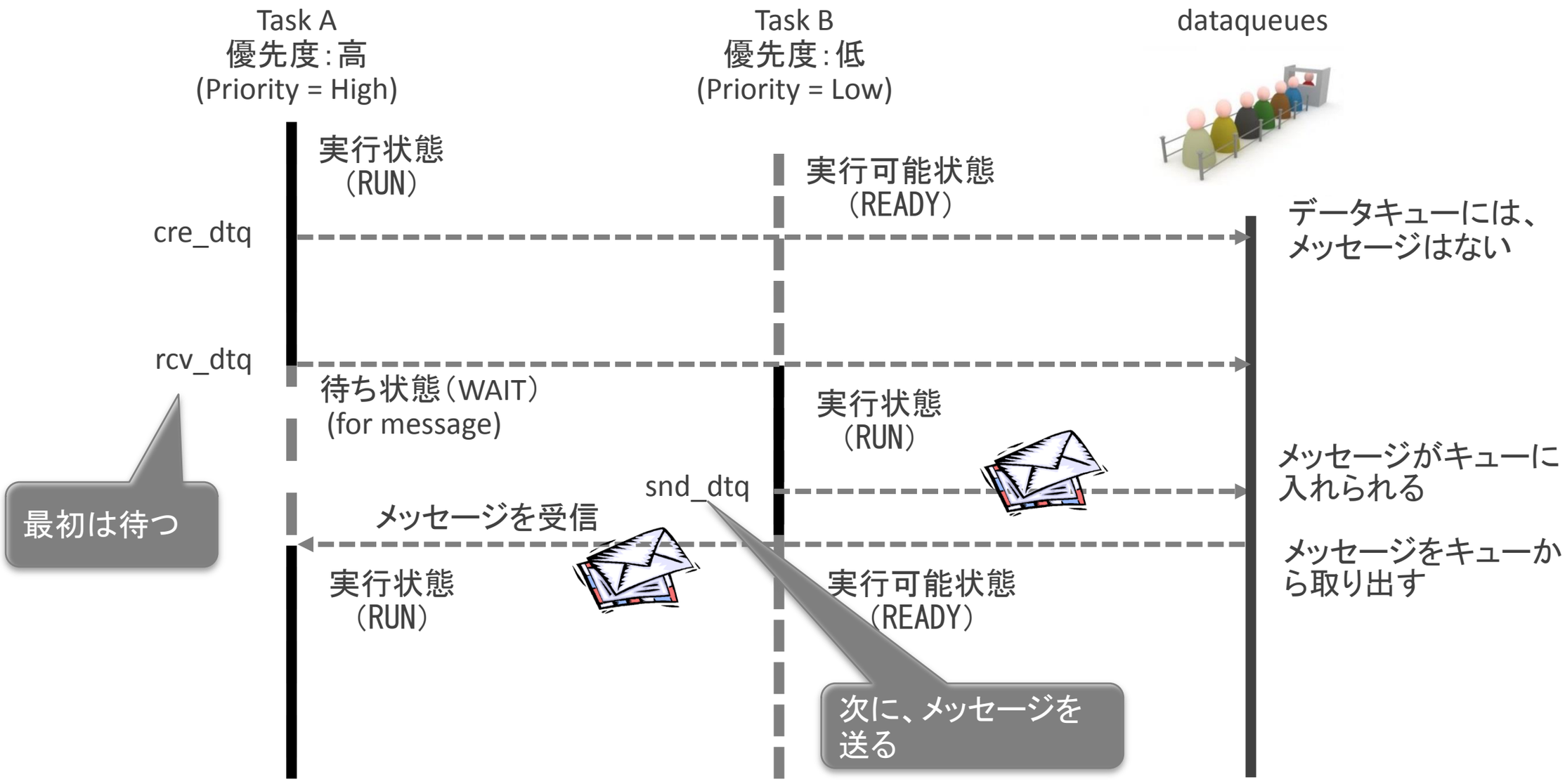


同期・通信機能（データキュー）　間接型

データキューは、1ワードのメッセージを受渡しすることにより、同期と通信を行うためのオブジェクト

cre_dtq()	データキュー生成(静的API)
acre_dtq()	データキュー生成(ID番号自動割付け)
del_dtq()	データキュー削除
snd_dtq()/psnd_dtq()/ipsnd_dtq()/tsnd_dtq()	データキューへの送信
fsnd_dtq()/ifsnd_dtq()	データキューへの強制送信
rcv_dtq()/prcv_dtq()/trcv_dtq()	データキューからの受信

データキュー（Dataqueues）の実行例



μITRON4.0仕様の機能(サービスコール)

- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- **拡張同期・通信機能**
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- 割り込み管理機能
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

拡張同期・通信機能

- ミューテックス（Mutex）
  - 共有資源に関するタスク間の排他制御を実現
  - 優先度逆転を防ぐために、優先度継承プロトコル、優先度上限プロトコルをサポートしている。
- メッセージバッファ（Message Buffer）
  - 可変長のメッセージをコピーしてやりとりする同期通信オブジェクト
  - メッセージバッファの領域サイズを調整することで、同期メッセージ、非同期メッセージの両方を実現可能
- ランデブポート（Rendezvous Port）
  - タスク間で同期通信を行うためのオブジェクト
  - ビットパターンによるランデブ条件によって、通常のクライアントサーバーモデルよりも柔軟な同期通信を実現できる。

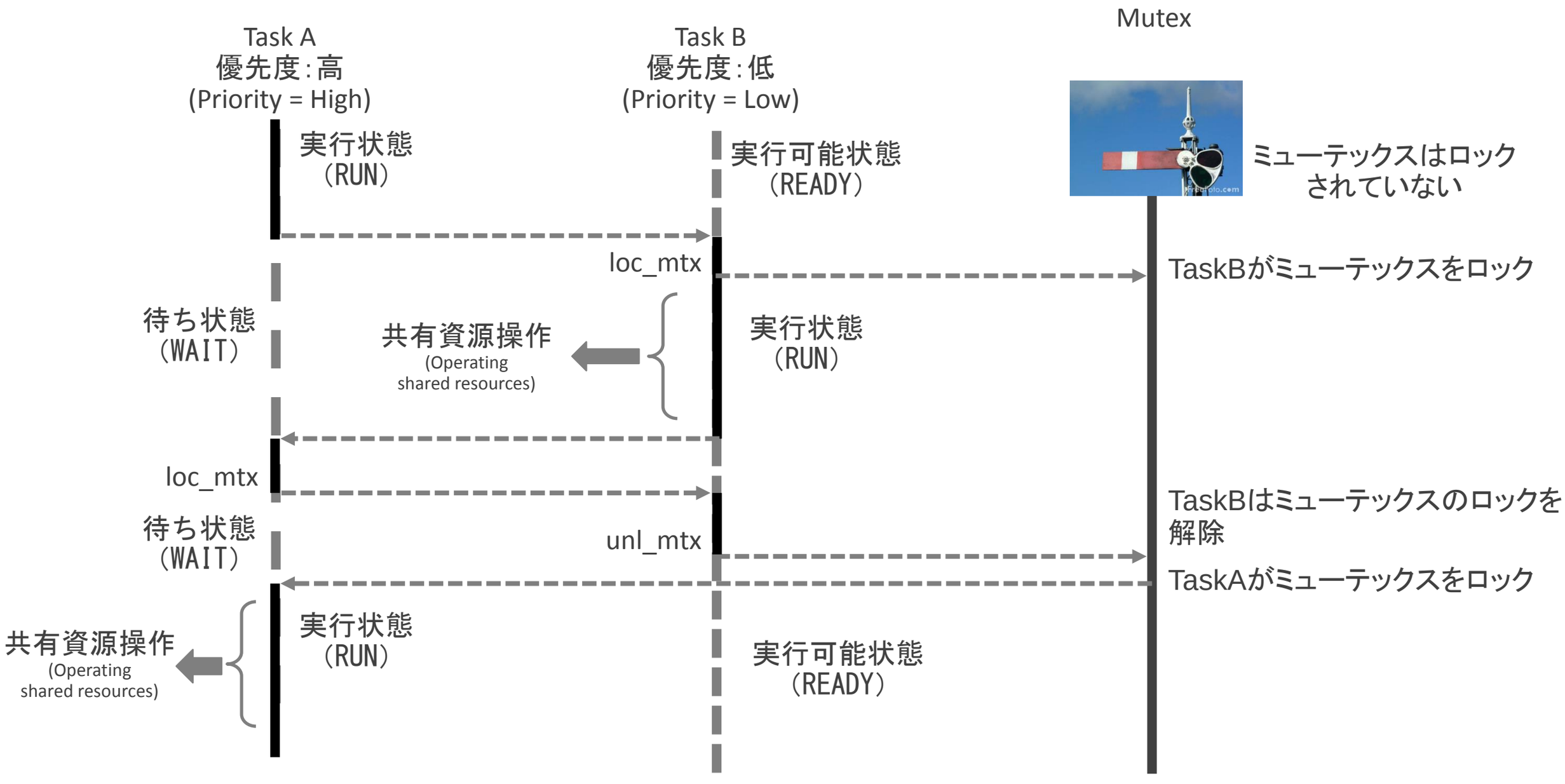
拡張同期・通信機能（ミューテックス）

ミューテックスは、優先度逆転現象を防ぐための機構をサポートした、排他制御を実現するためのオブジェクト。

優先度逆転を防ぐために、優先度継承プロトコル、優先度上限プロトコルをサポートしています。

cre_mtx()	ミューテックス生成(静的API)
acre_mtx()	ミューテックス生成(ID番号自動割付け)
del_mtx()	ミューテックス削除
loc_mtx()/ploc_mtx()/tloc_mtx()	ミューテックスのロック
unl_mtx()	ミューテックスのロック解除

ミューテックス（Mutex）の実行例

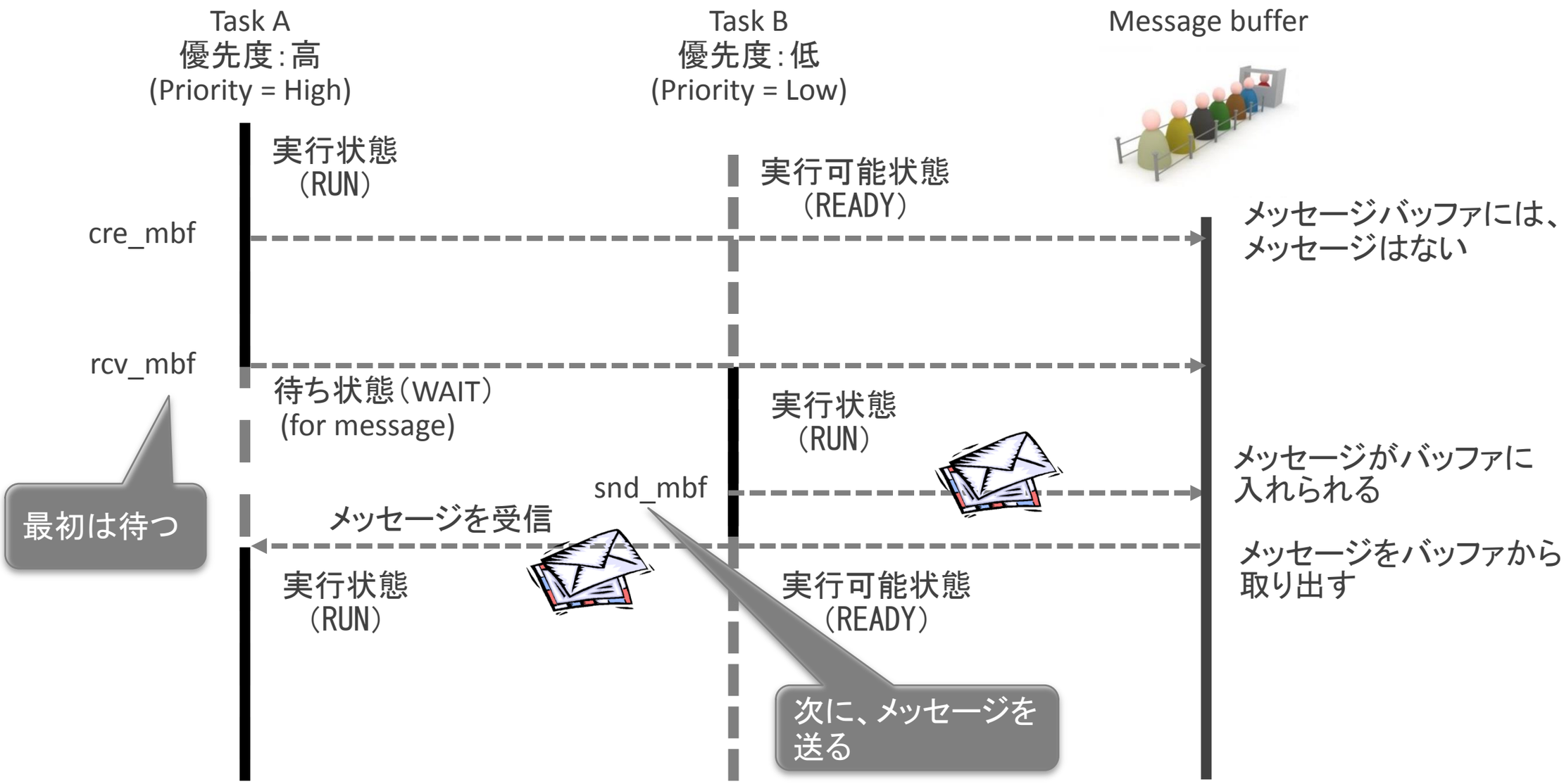


拡張同期・通信機能（メッセージバッファ）間接型

可変長のメッセージをコピーしてやりとりする同期通信オブジェクト

cre_mbf()	メッセージバッファ生成(静的API)
acre_mbf()	メッセージバッファ生成(ID番号自動割付け)
del_mbf()	メッセージバッファ削除
snd_mbf()/psnd_mbf()/tsnd_mbf()	メッセージバッファへの送信
rcv_mbf()/prcv_mbf()/trcv_mbf()	メッセージバッファからの受信

メッセージバッファ（Message Buffer）の実行例

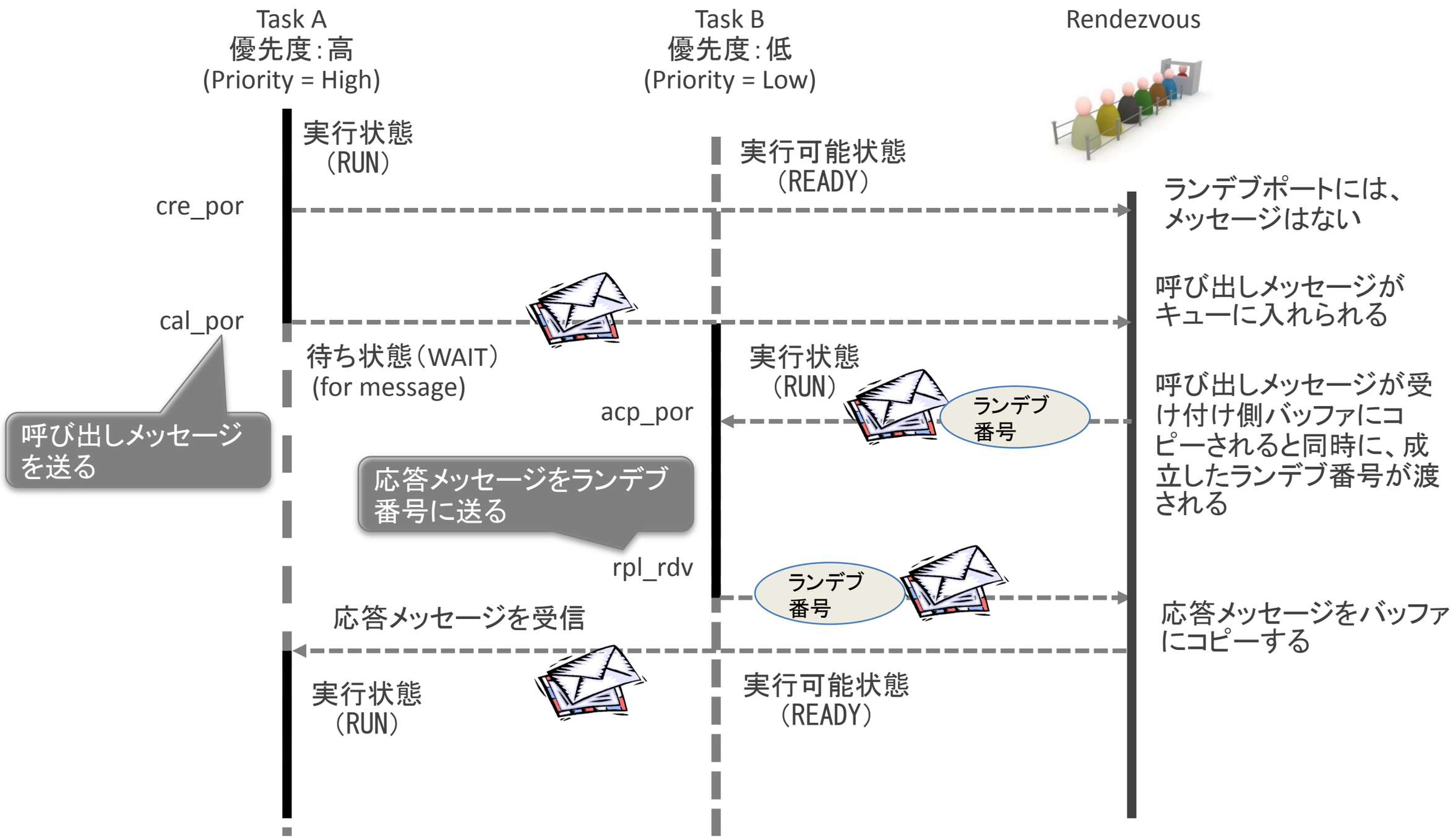


拡張同期・通信機能（ランデブ）　同期型

タスク間で同期通信を行うためのオブジェクト。ランデブポートと呼ばれる窓口を介して、呼び出しタスクと受付タスクが待ち合わせを行い、待ち合わせ(ランデブ)が成立するとお互いのメッセージを交換する。

cre_por()	ランデブポート生成(静的API)
acre_por()	ランデブポート生成(ID番号自動割付け)
del_por()	ランデブポート削除
cal_por()/tcal_por()	ランデブの呼び出し
acp_por()/pacp_por()/tacp_por()	ランデブの受け付け
fwd_por()	ランデブの回送
rpl_rdv()	ランデブの終了

ランデブ（Rendezvous）の実行例



μITRON4.0仕様の機能(サービスコール)

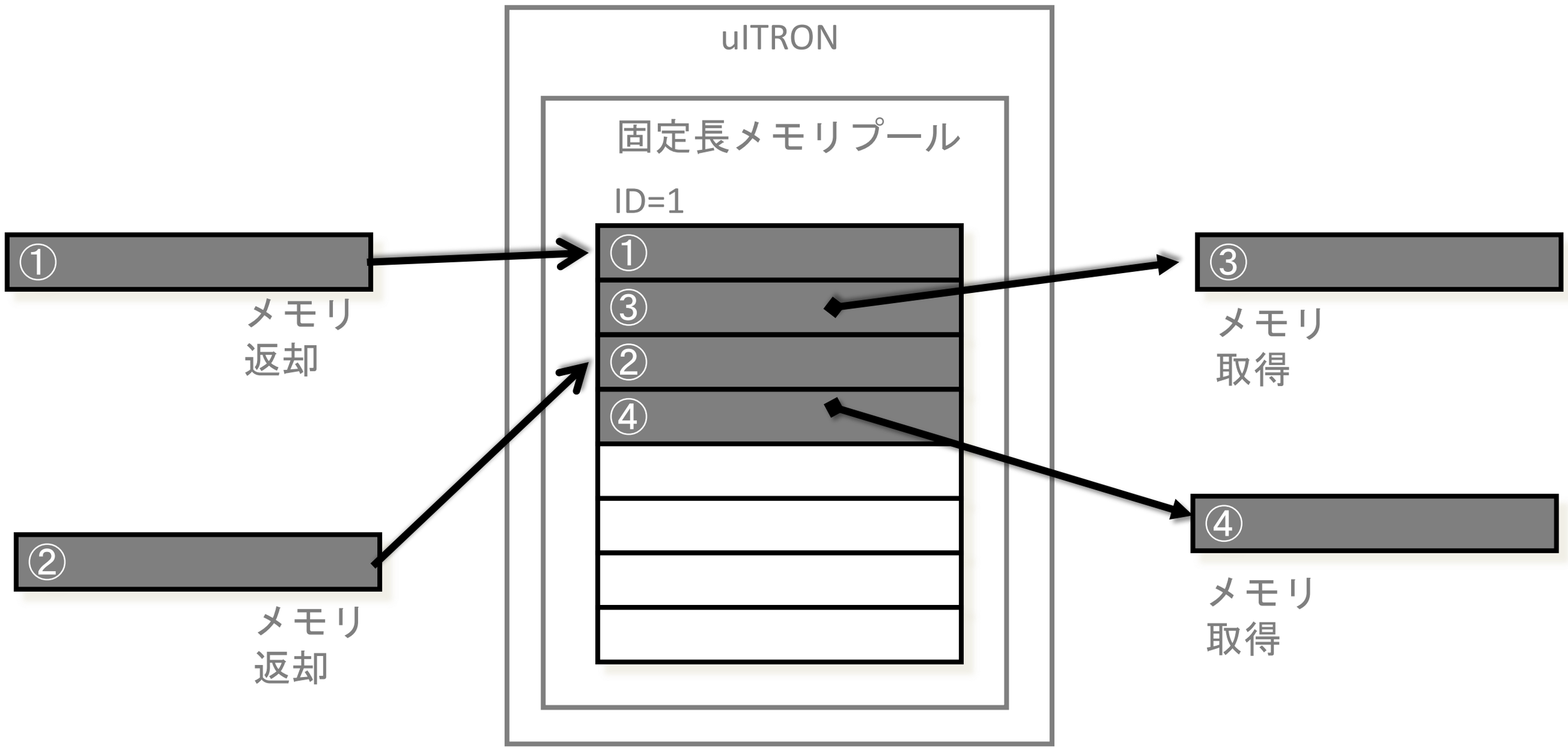
- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- **メモリプール管理機能**
  - **固定長メモリプール**
  - **可変長メモリプール**
- 割り込み管理機能
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

メモリプール管理機能(固定長)

固定長メモリプールは、固定サイズのメモリブロックを動的に管理する機能です。

cre_mpf()	固定長メモリプール生成
del_mpf()	固定長メモリプール削除
get_mpf()	固定長メモリブロック獲得
pget_mpf()	固定長メモリブロック獲得(ポーリングあり)
tget_mpf()	固定長メモリブロック獲得(タイムアウトあり)
rel_mpf()	固定長メモリブロック返却
ref_mpf()	固定長メモリプール状態参照

固定長メモリプール機能の動作



取得時のメモリブロックサイズが固定サイズ

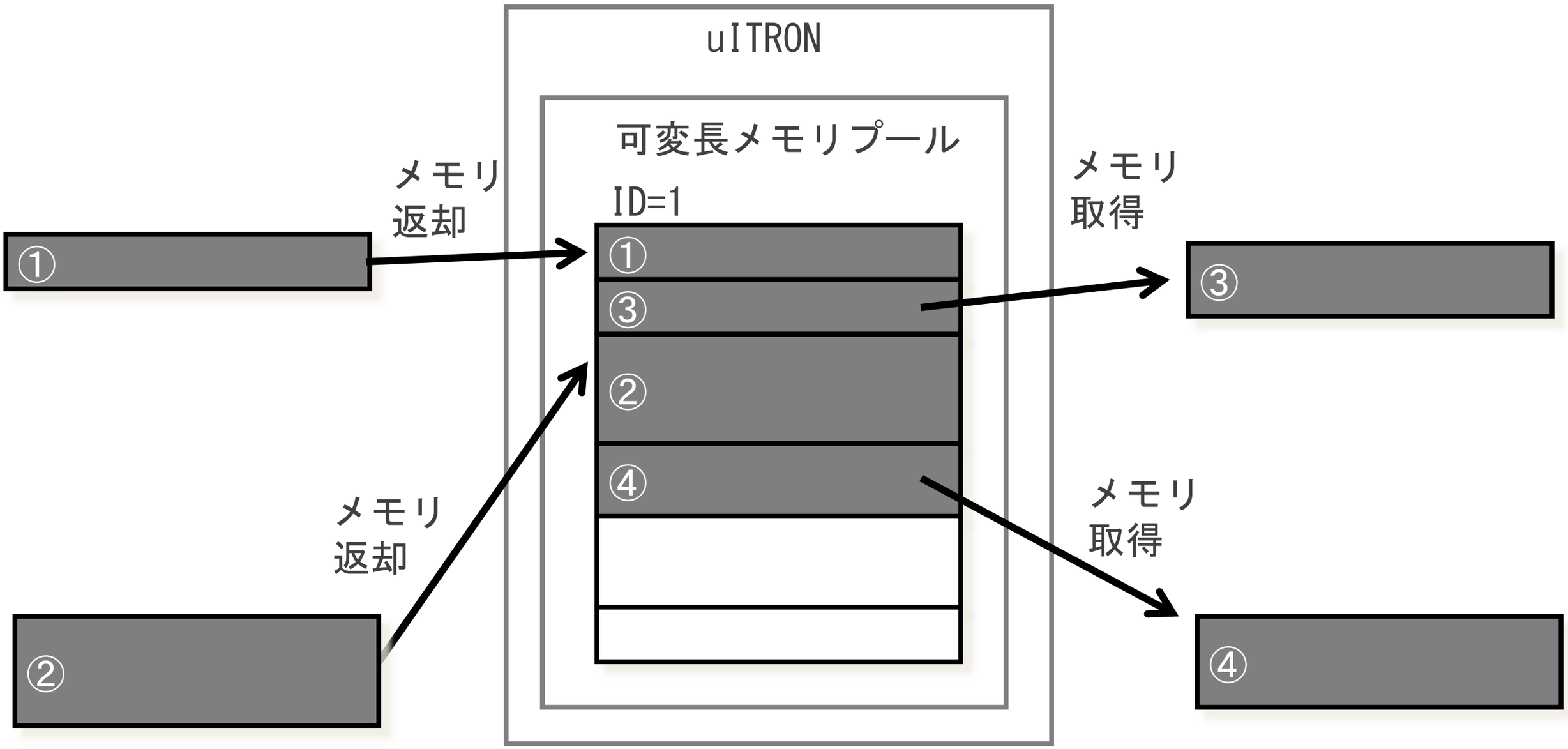
可変長メモリプールよりは柔軟性に欠けますが、
分断の心配はなく、かつ高速です。

メモリプール管理機能(可変長）

可変長メモリプールは、任意サイズのメモリブロックを動的に管理する機能です。

cre_mpl()	可変長メモリプール生成
del_mpl()	可変長メモリプール削除
get_mpl()	可変長メモリブロック獲得
pget_mpl()	可変長メモリブロック獲得(ポーリングあり)
tget_mpl()	可変長メモリブロック獲得(タイムアウトあり)
rel_mpl()	可変長メモリブロック返却
ref_mpl()	可変長メモリプール状態参照

可変長メモリプール機能の動作



取得時のメモリブロックサイズが任意サイズ

手軽ですが、獲得・解放を繰り返すうちにプール内部が分断され、
大きなサイズが獲得できなくなる可能性があります。
(T-Kernel/uITONは、分断されたメモリ領域を統合する機能までは持っていません）。

μITRON4.0仕様の機能(サービスコール)

- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- **割り込み管理機能**
- 時間管理機能
  - システム時刻管理
  - 周期ハンドラ
  - アラームハンドラ
  - オーバランハンドラ
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

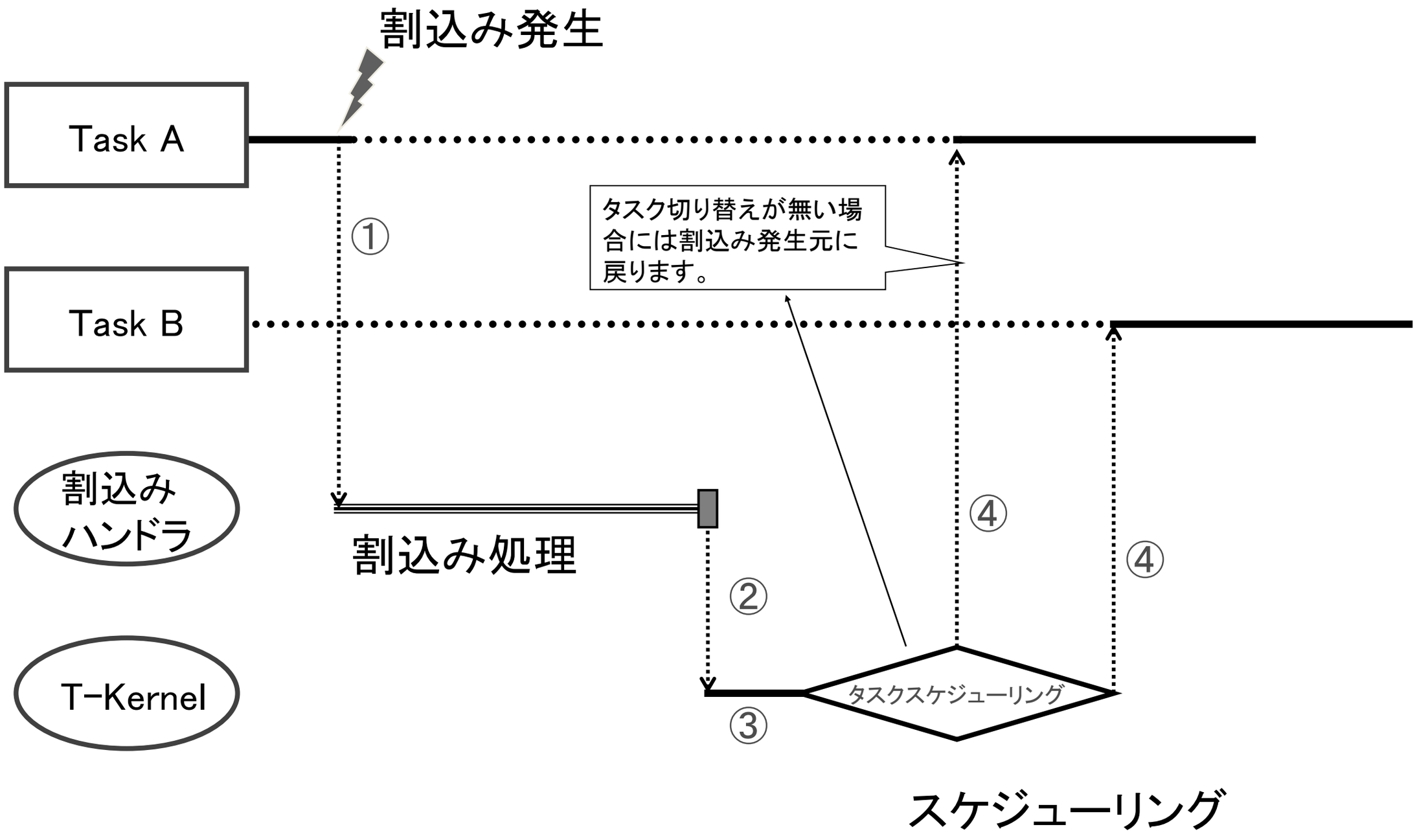
割込み管理機能

外部割込みによって起動される割込みハンドらおよび割込みサービスルーチンを管理するための機能です。

def_inh()	割込みハンドラ定義
cre_isr()	割込みサービスルーチンの生成
del_isr()	割込みサービスルーチンの削除
ref_isr()	割込みサービスルーチンの状態参照
dis_int()	割込みの禁止
ena_int()	割込みの許可
chg_ixx()	割込みマスクの変更
get_ixx()	割込みマスクの参照

リアルタイム処理では、割り込みに対する応答速度が重要です。割り込みは緊急度が高いため、通常RTOSはタスク実行を取りやめて、割り込みハンドラを起動します。これを管理するのが割り込み管理機能です。
この機能によって、割り込みハンドラを登録し、割り込みハンドラからタスクを制御することで、外部イベント要求に対し、素早いリアルタイム性を提供できます。

割込み処理動作(例)



割込み管理機能(注意点)

項目	タスク	割込みハンドラ
実行管理	ＯＳにより５つの状態で管理されます。 （ＲＥＡＤＹ、ＷＡＩＴ等）	ＯＳは実行管理を行いません。
実行スケジュール	ＲＥＡＤＹ状態のタスクの中で最高の優先度を持つタスクが実行されます。	ＯＳでは管理されず外部割込み処理で直接起動されます。
実行の切替え	優先度の高いタスクの実行要求、実行タスクの状態変化により、タスク実行が切り替えられます。	１・タスクにより処理は中断されません。 ２・割込みハンドラ同士では割込みレベルの高い方を優先します。
システムコール	割込みハンドラ専用システムコール以外の全システムコールが使用可能です。	制限があります。

割込みハンドラを作成する場合には、次の2点について注意する必要があります。

・処理時間は極力短くする。
**　割込みハンドラ実行中は、タスクは動作できません。ハンドラでの割込み処理は必要最低限にしてください。**

・使用できるシステムコールには制限があります。

μITRON4.0仕様の機能(サービスコール)

- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
  - セマフォ
  - イベントフラグ
  - メールボックス
  - データキュー
- 拡張同期・通信機能
  - ミューテックス
  - メッセージバッファ
  - ランデブポート
- メモリプール管理機能
  - 固定長メモリプール
  - 可変長メモリプール
- 割り込み管理機能
- **時間管理機能**
  - **システム時刻管理**
  - **周期ハンドラ**
  - **アラームハンドラ**
  - **オーバランハンドラ**
- システム状態管理機能
- サービスコール管理機能
- システム構成管理機能

時間管理機能

時間管理機能は、時間に依存した処理を行う機能です。μITRON仕様では、周期ハンドラ/アラームハンドラの2つのタイムイベントハンドラがあります。
タイムイベントハンドラはタスクとしてではなく、タスク独立部として実行されます。つまり、ハードウェアタイマからのタイマ割り込み処理の延長として、周期起動ハンドラが呼び出されます。これにより、タイマハンドラの起動時間がより正確になり、処理のオーバヘッドを減少させることが出来ます。

システム時刻管理

- システム時刻を操作するための機能。システム時刻を設定/参照する機能、タイムティックを供給してシステム時刻を更新する機能が含まれます。

周期ハンドラ

- 一定周期で起動されるタイムイベントハンドラ。周期ハンドラ機能には、周期ハンドラを生成/削除する機能、周期起動ハンドラの動作を開始/停止する機能、周期起動ハンドラの状態を参照する機能が含まれます。

アラームハンドラ

- 指定した時刻に起動されるタイムイベントハンドラ。アラームハンドラ機能には、アラームハンドラを生成/削除する機能、アラームハンドラ動作を開始/停止する機能、アラームハンドラの状態を参照する機能が含まれます。

時間管理機能(システム時刻管理)

システム時刻を操作する機能

set_tim()	システム時間設定
get_tim()	システム時間参照
isig_tim()	タイムティックの供給

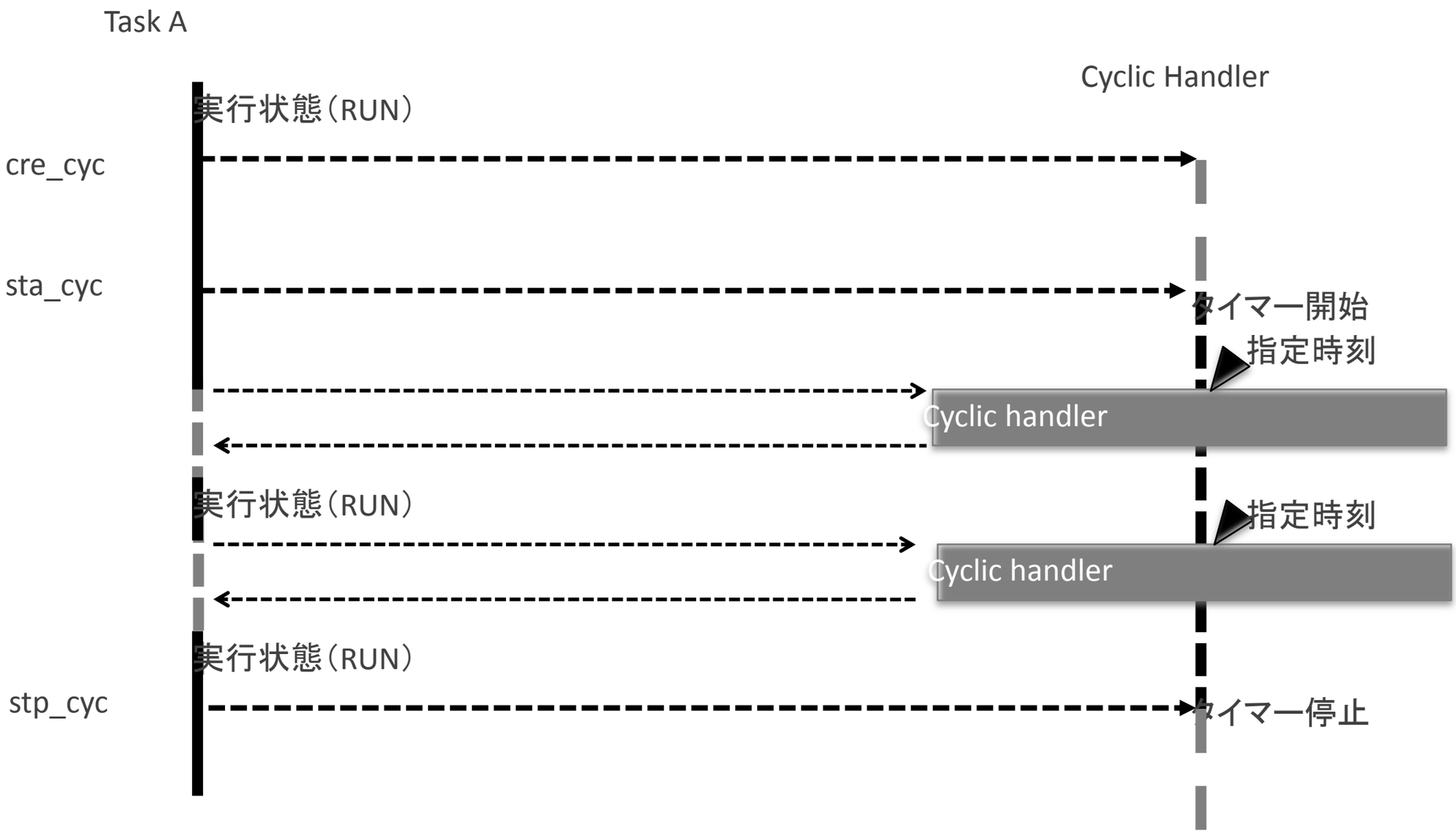
時間管理機能（周期ハンドラ）

周期ハンドラは、一定周期で起動されるタイムイベントハンドラ。非タスクコンテキストで動作します。

cre_cyc()	周期ハンドラの生成
del_cyc()	周期ハンドラの削除
sta_cyc()	周期ハンドラの動作開始
stp_cyc()	周期ハンドラの動作停止
ref_Cyc()	周期ハンドラの状態参照

！周期ハンドラは、非タスクコンテキスト（タスク独立部）で動作します。

周期起動ハンドラ（Cyclic Handler）の実行例



cre_cyc() :　　周期ハンドラの生成

sta_cyc() :　　周期ハンドラの動作開始

stp_cyc() :　　周期ハンドラの動作停止

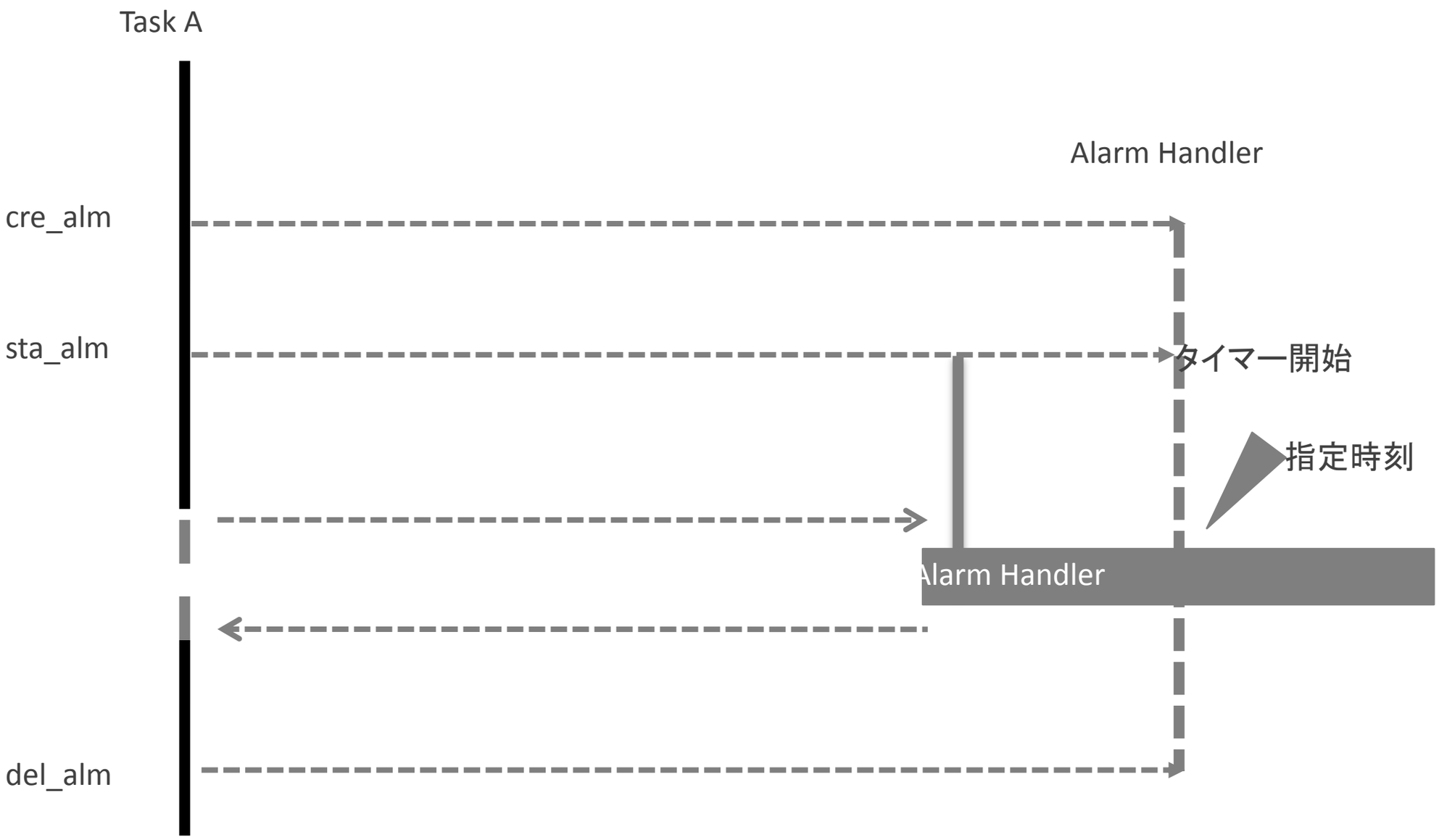
時間管理機能（アラームハンドラ）

アラームハンドラは、指定した時間に起動されるタイムイベントハンドラ

cre_alm()	アラームハンドラの生成
del_alm()	アラームハンドラの削除
sta_alm()	アラームハンドラの起動
stp_alm()	アラームハンドラの動作停止
ref_aim()	アラームハンドラの状態参照

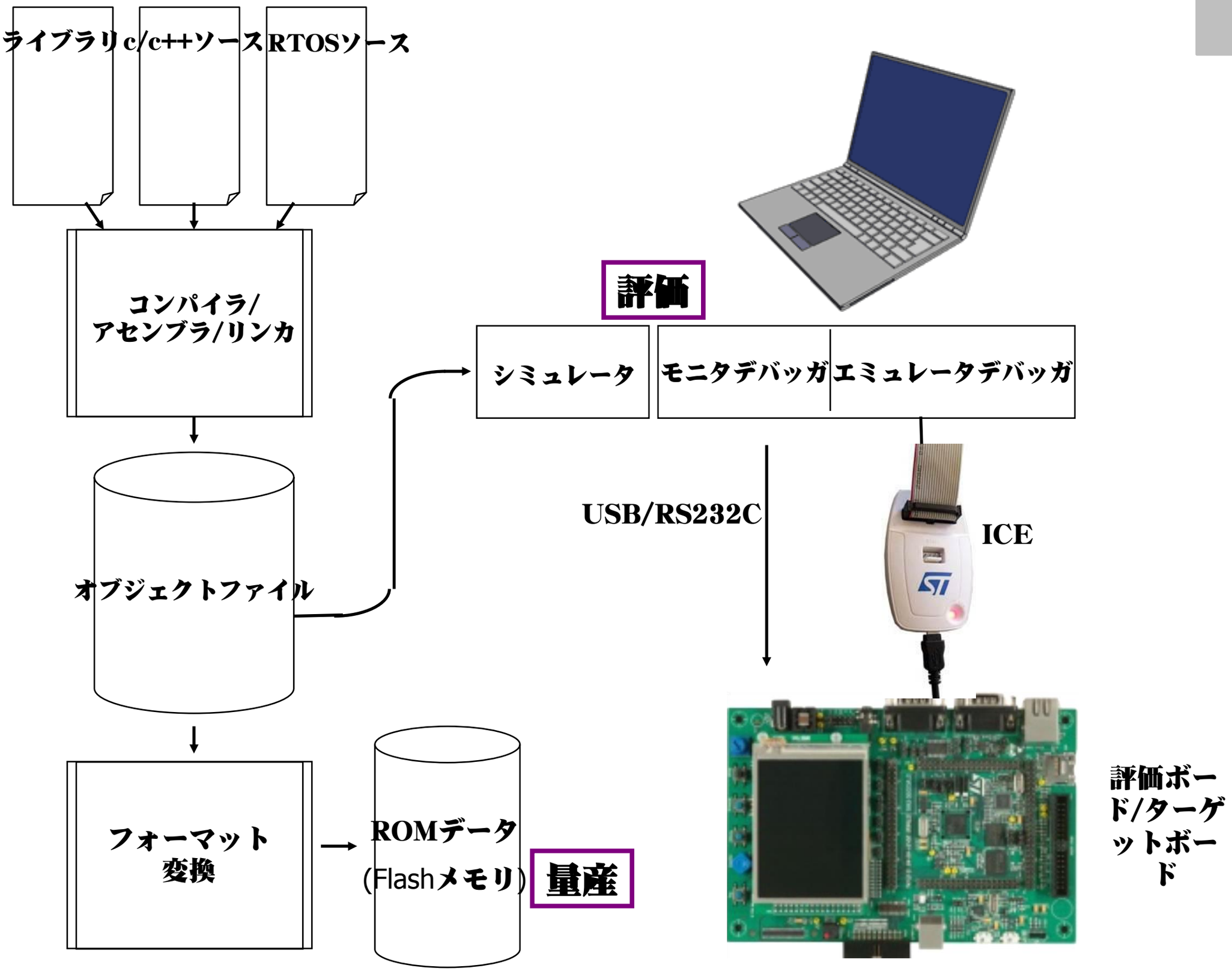
！アラームハンドラは、非タスクコンテキスト（タスク独立部）で動作します。

アラームハンドラ（Alarm Handler）の実現例



4 μITRON開発手順

開発環境の例



アプリケーション・プログラム開発の流れ(参考)

（１）処理をタスク/ハンドラに分割

↓

（２）各タスクの優先度を決定

↓

（３）各タスクで使用するシステムコールを選択

↓

（４）アプリケーションプログラムの記述

↓

（５）コンフィグレーション

↓

（６）オブジェクトファイルの作成

アプリケーション・プログラム開発の流れ(参考)

（1） 処理をタスク及びハンドラに分割

開発する製品の仕様に基づいて、プロセッサで実行したい処理をタスク/ハンドラに分割します。状況変化をとらえて動作する処理を「ハンドラ」とし、主な処理を「タスク」とします。「タスク」と「ハンドラ」の2種類の要素を組み合わせて設計します。

タスク/ハンドラ分割における設計が、リアルタイム処理性能を大きく左右します。

タスク分割を決定する際には、下記の項目を考慮してください。

（1）順次処理は同一タスク、並行処理は別タスクとする
（2）機能的な関連性が深い処理をグルーピングしてタスクとする
（3）適度な大きさおよび適度な数の処理に分割してタスクとする
（4）複数のタスク間にまたがるデータはできるだけ少なくする

※タスク分割についての一例を参考までに付録に紹介してあります。

（2） 各タスクの優先度を決定

(1)で決定したタスクにおける処理の内容を比較して、各タスクの優先度を決定します。他のタスクに実行権を奪われたくないタスクは、優先度を高くする必要があります。

（3） 各タスクで使用するシステムコールを選択

各タスク間の同期、通信を考慮して、各タスクの処理において使用するシステムコールを決定します。

アプリケーション・プログラム開発の流れ(参考)

(4) アプリケーションプログラムの記述

サービスコールを使って、アプリケーションプログラム(1で決定したハンドラとタスク)を記述します。通常、C言語を使用して、アプリケーションプログラムを記述します。

(5) コンフィグレーション

設計、記述したタスクに合わせて、RTOSを使用するためのコンフィグレーションを行い、システム環境定義ファイルを作成します。

コンフィグレーションの手段として、GUIコンフィグレーションツールを使うと便利です(OSベンダ提供)。GUIより入力されたコンフィグレーション情報から、システム環境定義ファイルを自動生成します。

(6) オブジェクトファイルの作成

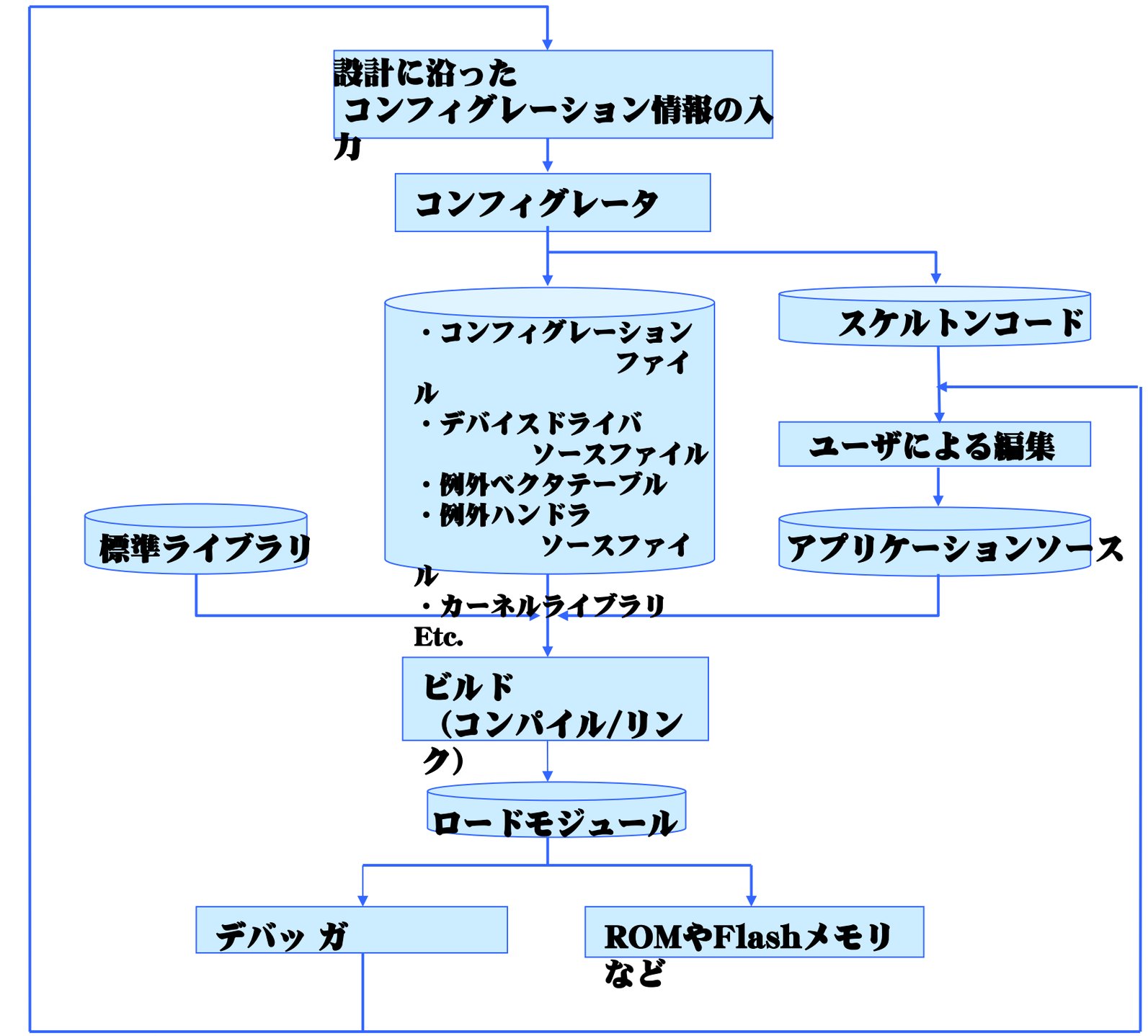
(4)で作成したアプリケーションプログラムと、(5)で作成したシステム環境定義ファイル、RTOSライブラリをコンパイル/アセンブル/リンクして、オブジェクトファイルを作成します。

コンフィギュレーション

- RTOSの動作環境をコンフィグレーションファイルに記述
- 静的APIによってオブジェクトを生成
  - ID番号の指定
  - 最大オブジェクト数の指定など
- 割込みハンドラを指定
- コンフィギュレータを使用してヘッダファイルとカーネル情報ファイルの出力

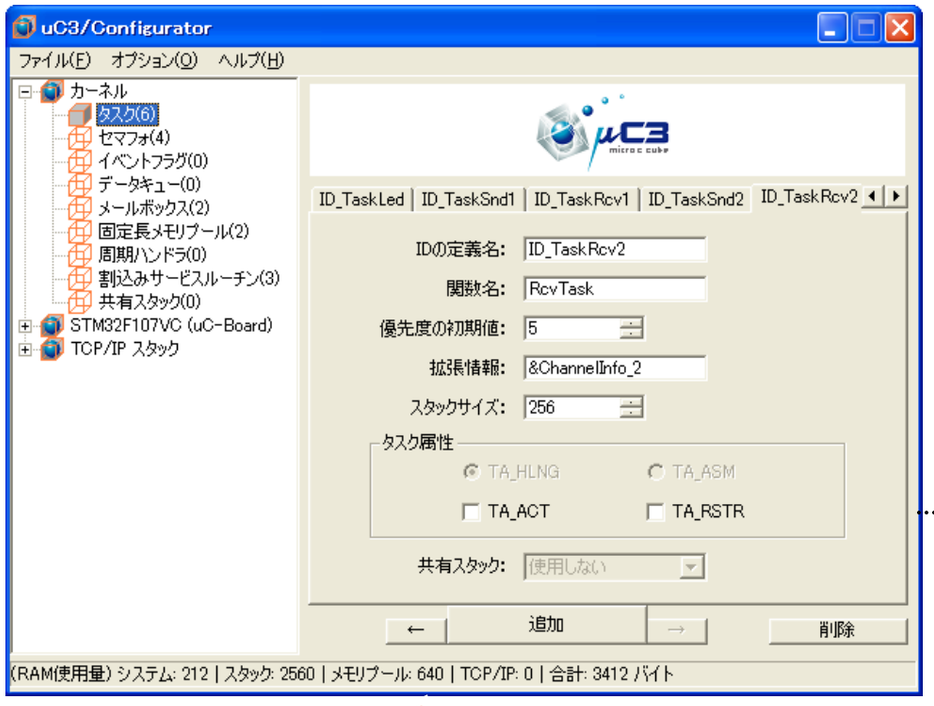
・コンフィギュレーション方法はOSによって方法などが異なる場合があります
・詳しくは各OSベンダの資料をご覧ください

μC/Compactでの開発手順

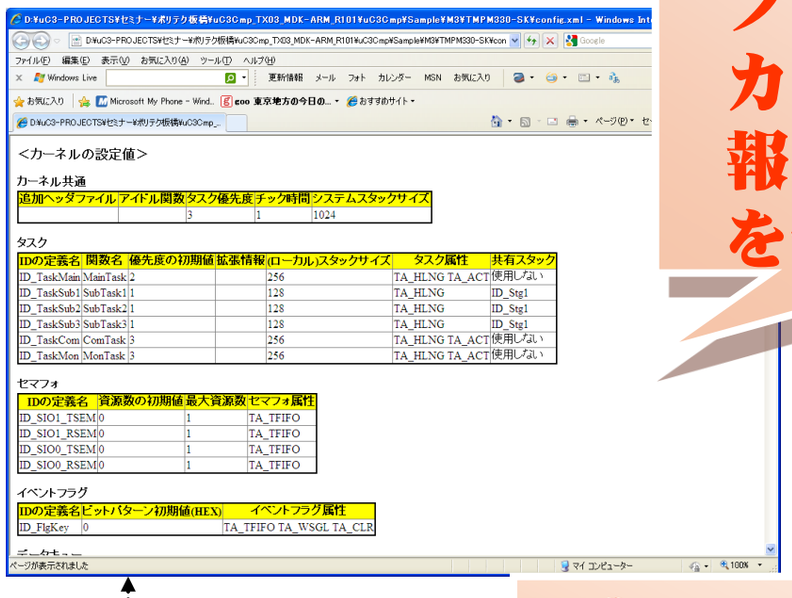


コンフィグレータ

資料提供：イー・フォース様



GUIを使って簡単に
カーネルコンフィグレーションが可能



ブートから
アプリケーションタスクが
起動するまでのコード
を
自動生成

```
051 /*************************************
052         SubTask3
053  *************************************/
054 void SubTask3(VP_INT exinf)
055 {
056 }
057
058 /*************************************
059         ComTask
060  *************************************/
061 void ComTask(VP_INT exinf)
062 {
063 }
064
065 /*************************************
066         MonTask
067  *************************************/
068 void MonTask(VP_INT exinf)
069 {
070 }
071
072 /*****************************************************
073         Cortex-M3の周辺デバイスの初期化
074  *****************************************************/
075
076 extern UW __vector_table[];
077
078 void cortex_m3_init_peripheral(void)
079 {
```

ブラウザで
カーネルリソース情報
をチェック

資料提供：イー・フォース様

コンフィグレータで生成されるファイル

必ず生成されるプロセッサに依存しないファイル

ファイル	内容
itron.h	カーネルのヘッダファイル
スタートアップ	パワーオンリセットによる初期化処理（アセンブラ言語）
ベクタテーブル	割込みベクタテーブル（アセンブラ言語）
例外ハンドラ	割込みハンドラを含めた例外ハンドラ（アセンブラ言語）
カーネルライブラリ	カーネル基本部とシステムコール群をまとめたライブラリ

必ず生成されるプロセッサに依存したファイル

ファイル	内容
kernel_id.h	オブジェクトID、デバイスID等の定義ヘッダファイル
kernel_cfg.c	カーネルのコンフィグレーション情報ファイル
kernel.h	カーネルのヘッダファイル
main.c	main()、初期設定関数、タスクやハンドラなどのスケルトンコード

デバイスドライバに依存したファイル

ファイル	内容
I/O定義ファイル	プロセッサのI/Oを定義したヘッダファイル
DDR_xxxxx.c	デバイスドライバのソースファイル
DDR_xxxxx.h	デバイスドライバのヘッダファイル
DDR_xxxxx_cfg.h	デバイスドライバのコンフィグレーションファイル

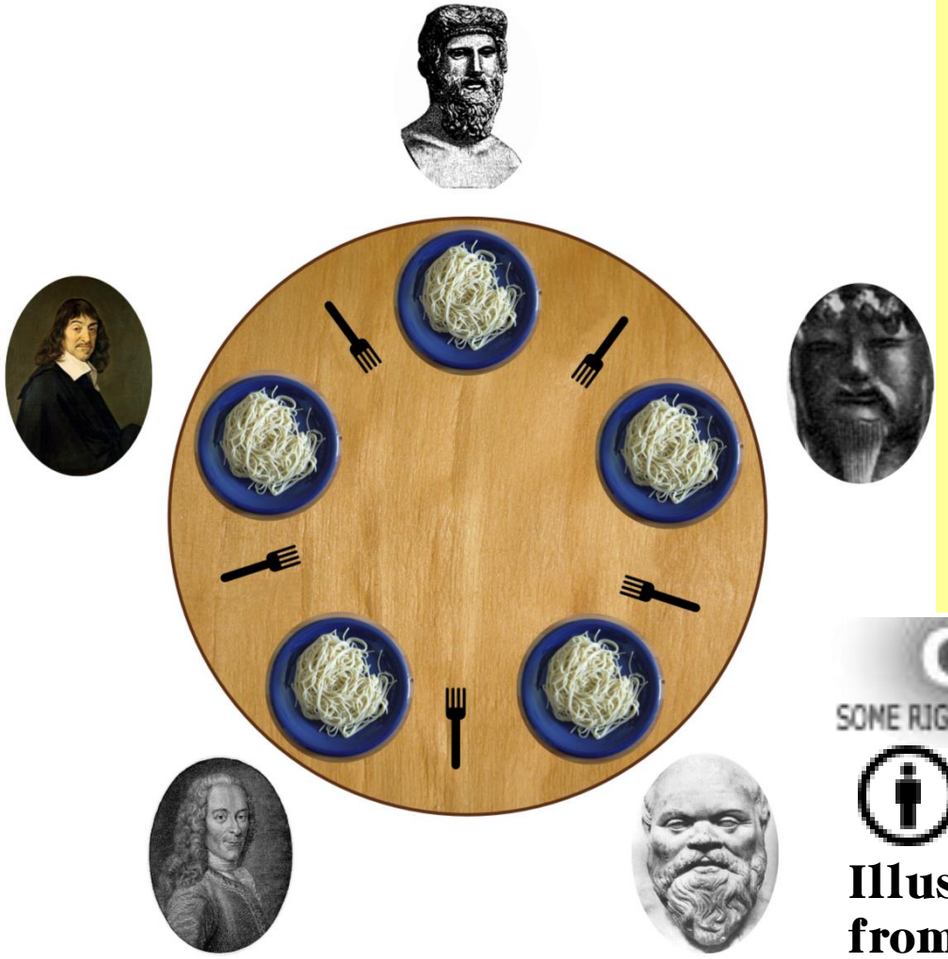
5 参考資料･付録など

おまけ(デッドロック)

マルチタスク処理で注意する必要があるのは、デッドロックやプライオリティ・インバージョン(同一優先順位のスケジューリング)です。

デッドロックは複雑な排他制御(組込みシステム上のあるハードウェア資源を、複数のタスクで共有する必要がある場合に使用)を行おうとする場合に発生します。代表的な例は「5人の哲学者」です。デッドロックを回避するには別のタスクのことも意識してプログラミングしなければならないことを示しています。

5人の哲学者は、それぞれスパゲッティ　が入った皿があるテーブルの回りに座ります。それぞれの皿の間に1本ずつ、合計5本のフォークがあります。
不特定の時刻に各哲学者はスパゲッティを食べようとします。食べるためには、まず自分の皿の横にある2本のフォークを取らなければなりません。フォークが取れるとスパゲッティを食べることかできます。一定時間後に食べ終わると、2本のフォークをテーブルに戻します。



哲学者をタスク、フォークを資源と考えます。問題はフォークが使えない結果として餓死してしまう哲学者がでないようにタスクを設計することです。

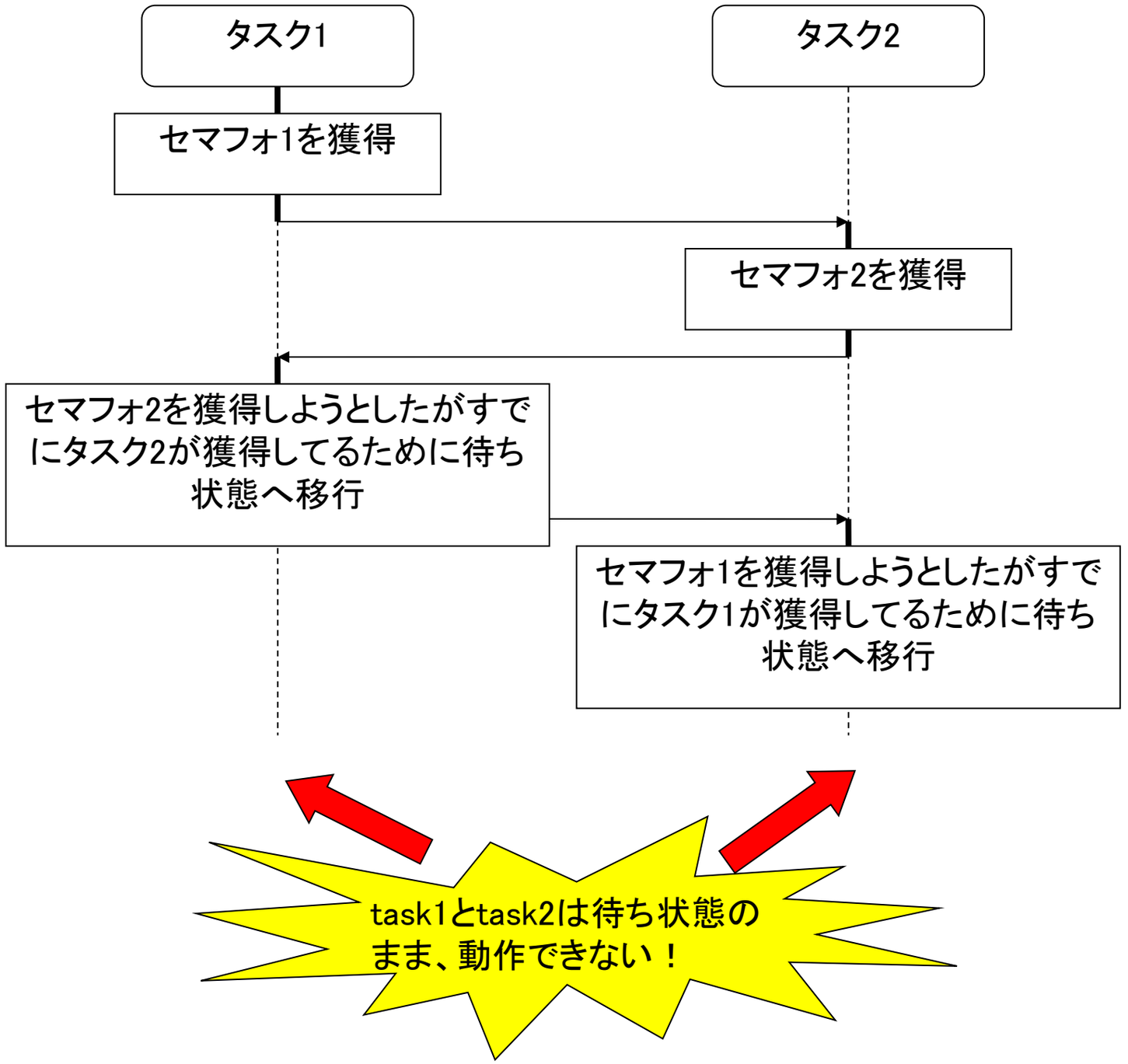
設計上、気をつけなければならないことが二つあります。

・まず、デッドロック状態を回避することです。この問題でのデッドロック状態とは、個々の哲学者がフォークを1本ずつ持ち、もう1本のフォークが開放されるのを永遠に待つことです。

・もう一つ気をつけなければならないのは2人以上の哲学者が、残りの哲学者のフォーク獲得を永遠に妨げるように共謀することがあってはならないということです。

Illustrated by Benjamin D. Esham.
from http://en.wikipedia.org/wiki/File:Dining_philosophers.png

おまけ(デッドロック)

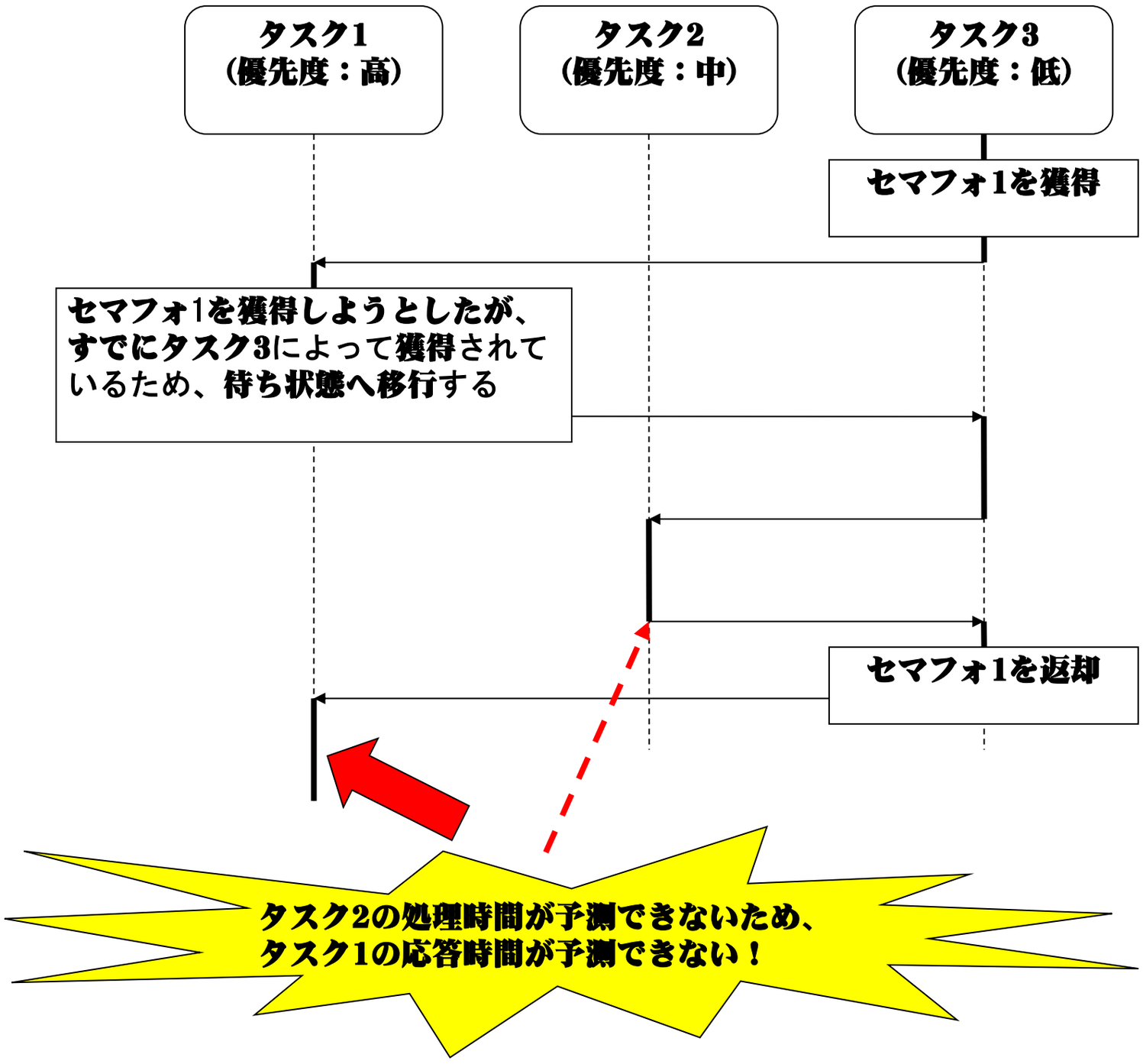


実際のプログラムでは複数の排他制御対象を同時に操作する場合もありますが、セマフォなどを複数使用して排他制御を行う場合、タイミングなどによってはデッドロックという現象を引き起こす場合がありますので、注意が必要です。
デッドロックとは、すべてのタスクが自分以外のいずれかのタスクが何かをやってくれることを待っている状態であり、正常な動作を行うことができなくなります。

デッドロックを回避するためには、獲得したカーネルオブジェクトとは逆の順序で開放するというルールを守ることが重要です。

実際の開発では、設計段階ではデッドロックが発生しないように考慮する必要がありますが、デバック作業などで処理を追加した時に、デッドロックが発生してしまう場合も多くあります。タスク数が増え、同期関係が複雑化してくると発生しやすくなりますので、注意が必要です。

おまけ(優先度逆転現象)



おまけ(優先度逆転現象)

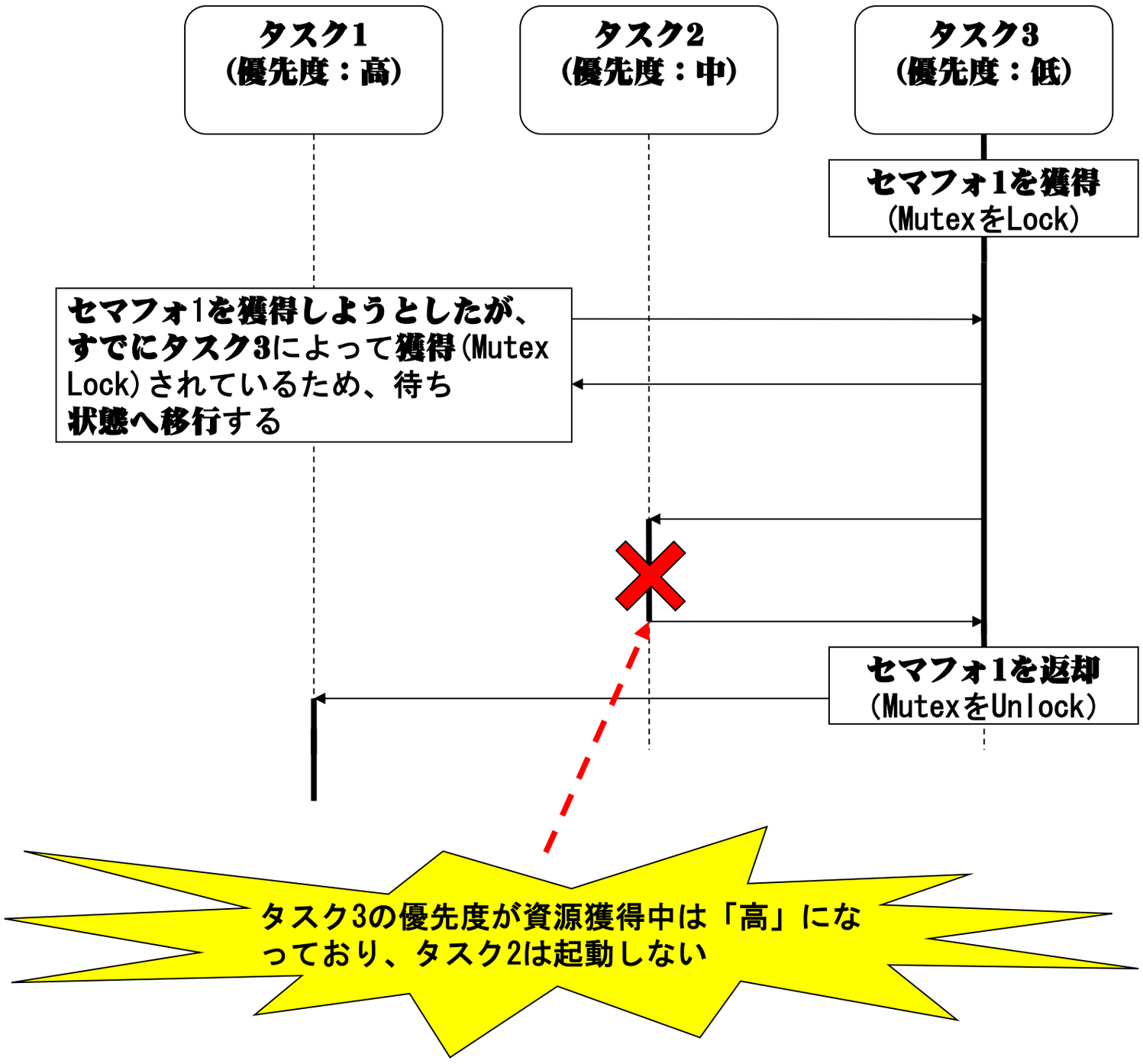
複数の排他制御対象を同時に操作する場合、デッドロックが発生しないように設計する事はもちろんですが、優先度逆転現象にも注意する必要があります。優先度逆転現象は、セマフォなどを使用して排他制御を行う場合に発生する可能性があります。

優先度逆転現象が発生すると、システムの応答性能の低下や、場合によってはシステム全体が動作不能になることもあります。また同一優先度のタスクでも発生する可能性があり、タスクそのものは動作しますので、発見が遅れることも良くあります。

発生を防ぐためには、排他制御の途中で動作するタスクが無いように優先度を設定する、優先度の高いタスクと低いタスクで同じカーネルオブジェクトをなるべく使用しない、セマフォの代わりにミューテックスを使用する、などがあります。

実際の開発では、設計段階で優先度逆転が発生しないように考慮する必要がありますが、デッドロックと同様に、デバック作業などで処理を追加した時に発生してしまう場合も多くあります。タスク数が増え、同期関係が複雑化してくると発生しやすくなりますので、注意が必要です。

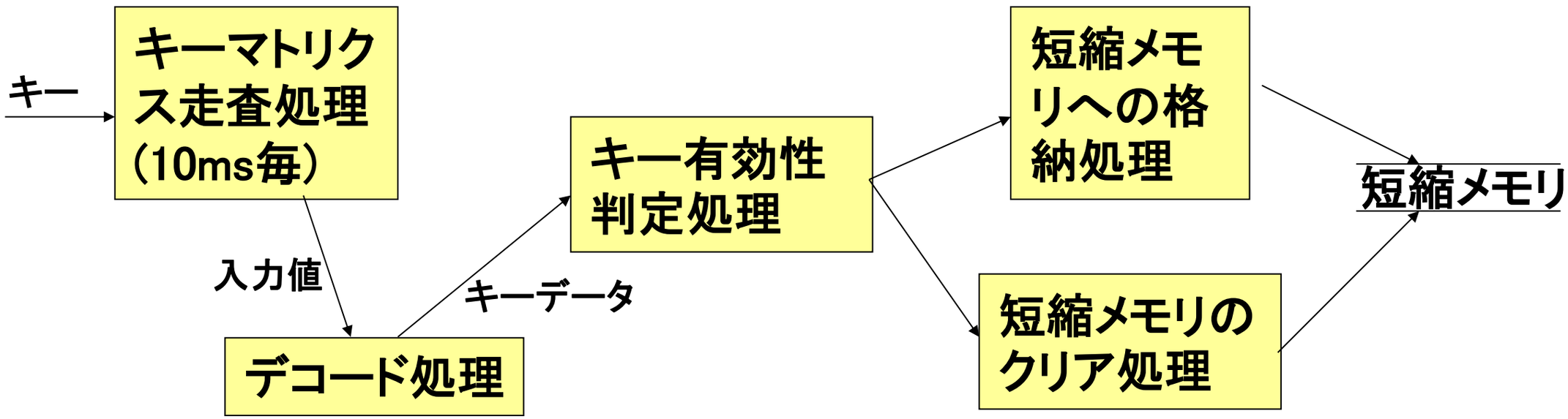
おまけ(ミューテックス使用例:優先度継承プロトコル)



おまけ(タスク分割)

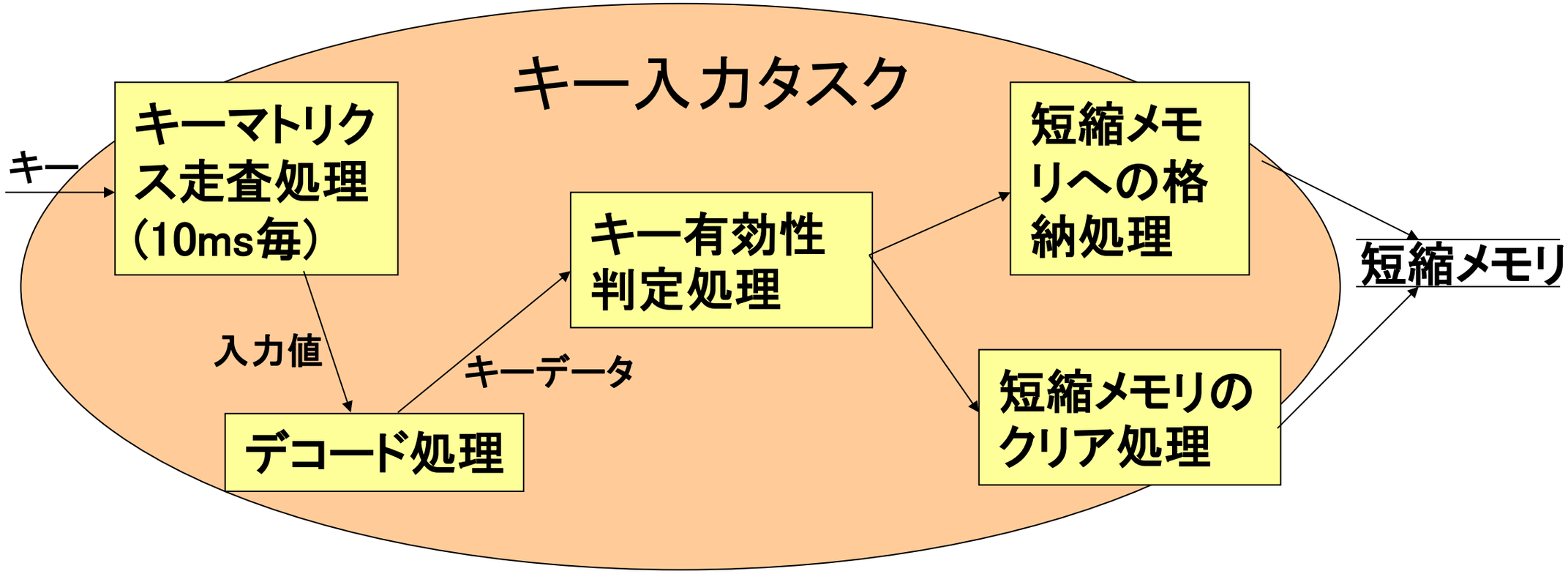
リアルタイム・システムを構築する際、タスク分割が必要です。ここではその一例と注意すべき点を記載します。

- 留守番電話機のキー入力処理にて
- キーマトリクス走査処理(10ms毎)
- 短縮メモリへ格納処理(30ms)



おまけ(タスク分割)

- キー関連処理(順次処理)をグルーピングし、1つの(キー入力)タスクとした。
- その結果、キーマトリクス走査ができなくなるというトラブルが発生した。

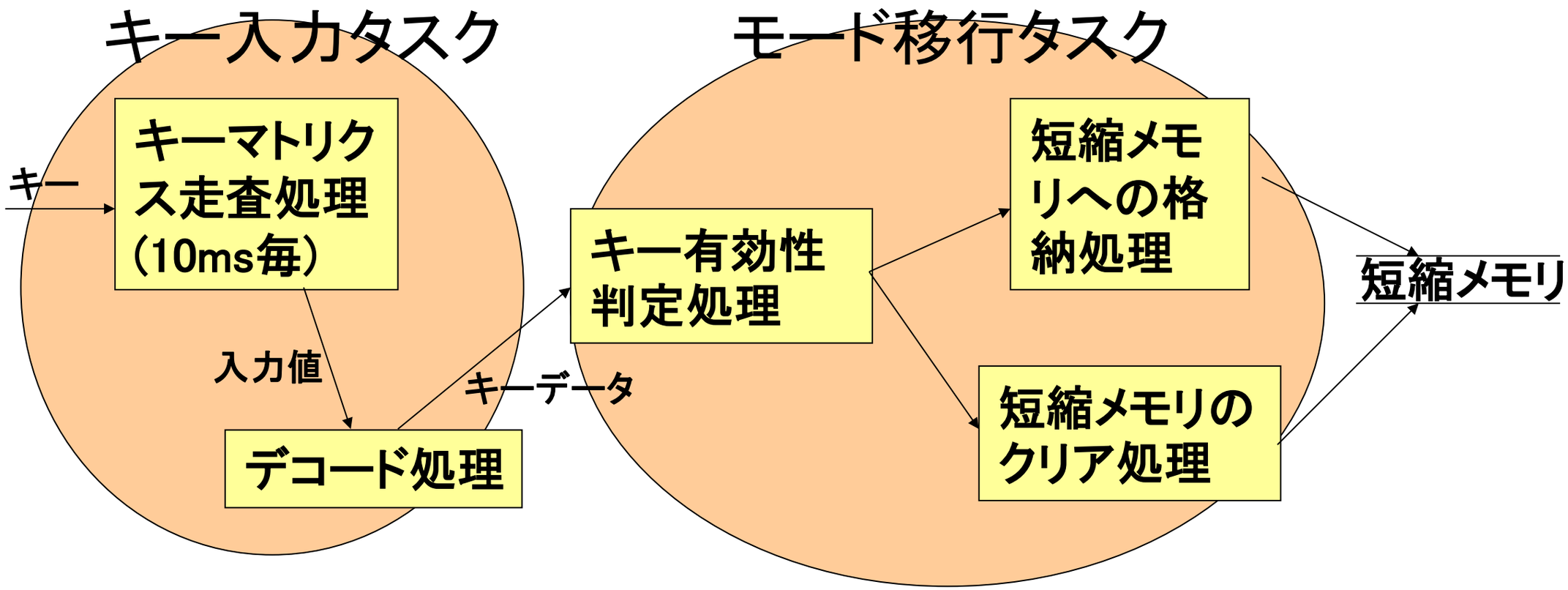


おまけ(タスク分割)

- 問題点は･･･キーマトリクス走査は10ms毎に行う必要があるが、短縮メモリへの格納に30ms掛かるため、この間、キーマトリクス走査ができなくなってしまった。
- 改善のポイントは･･･順次処理とはいえ、処理時間の異なる処理は別タスクにする。

おまけ(タスク分割)

- キーマトリクス走査処理とデコード処理で1つ、キー有効性判定処理と短縮メモリ関連処理で1つ、それぞれタスク分割した。



おまけ(タスク分割)

- 異なる時間(タイミング)で行われる処理は別タスクにし、リアルタイム性が損なわれないよう設計する必要がある。
- 順次処理をグルーピングすると、またがるデータを少なくできるなどのメリットがある。
- しかし、実は並行処理が潜んでいることは良くあり、注意深くタスク分割する必要がある。

なお、一つのシステムでどのくらいのタスク数になるのか、これも一概には言えません。あるRTOSは32767個のタスク登録が可能ですが、そこまでタスクを増やすと管理が大変です。ということで、結局、どのような規模のシステムでも、大体30～50個ぐらいのタスク数に落ち着くようです。

おまけ(一般的なC言語プログラミングと違う点 : 1)

- μITRONでは、タスクはmain()から起動するとは限らない
  - 最初に動作させるタスクは設定されているタスクの情報に基づいて決まる
  - 最初に動作させるタスクからユーザのプログラムが始まるという規則になっている

- イベントが発生しないと、タスクは切替らない
  - 割込みやサービスコールの呼出し等のイベントが発生し、動作中のタスクが待ち状態に入った時に、他のタスクに切替る

おまけ（一般的なC言語プログラミングと違う点：2）

- イベントが発生し、タスクが切り替え可能状態であれば、優先度が高いタスクに切替る(プリエンプション)
  - プリエンプション動作により、高い優先度のタスクが動作するので、高速な応答を行う必要があるタスクには、高い優先度を設定できる
  - 意識的に切り替え(ディスパッチ)を禁止することも可能

- リアルタイム性の確保はアプリケーションの作り方も重要

おまけ(イベントドリブン型プログラミング)

- リアルタイムOSでは、何か要求(イベント)がきたら動作する、というイベントドリブン型プログラミングが基本
  - 受動型プログラムとも呼ばれる
  - 発生したイベントに対する高速応答が可能
- 自ら動作を行う、能動的なプログラミングも可能
  - すべて能動的な動作では、リアルタイム性を確保するのは難しい場合が多い
- どのような形式にするかはアプリケーションによる

おまけ(セマフォを使った排他制御のサンプル)


```
#include "kernel.h"
#include "kernel_id.h"
/*****************************************************************************/
/*  ポートP0へアクセスするタスク1                                            */
/*****************************************************************************/
void task1(VP_INT exinf)
{
  ER   ercd;
  char num;
  while(1)
  {
    ercd = wai_sem(ID_SEM1); /* セマフォを獲得 */
    /* ここに排他制御が必要な処理を記述します */
    num = P0;
    P0 = num + 1;
    ercd = sig_sem(ID_SEM1); /* セマフォを返却 */
    /* ここに記述される処理は排他制御されません */
    ercd = dly_tsk(100);
  }
}
/*****************************************************************************/
/*  ポートP0へアクセスするタスク2                                            */
/*****************************************************************************/
void task2(VP_INT exinf)
{
  ER   ercd;
  while(1)
  {
    ercd = wai_sem(ID_SEM1); /* セマフォを獲得 */
    /* ここに排他制御が必要な処理を記述します */
    P0 = 0x00;
    ercd = sig_sem(ID_SEM1); /* セマフォを返却 */
    /* ここに記述される処理は排他制御されません */
    ercd = dly_tsk(100);
  }
}
```

おまけ(メールボックスを使ったサンプルプログラム)


```
#include "kernel.2h"
#include "kernel_id.h"
#include "sample_task.h"

/*************************************************************************/
/*  メッセージ形式                                                       */
/*************************************************************************/
typedef struct
{
            T_MSG header;                   /* メッセージヘッダ領域 */
    UB buf[20];                     /* メッセージ本体 */
} USER_MSG;

/*************************************************************************/
/*  メッセージを送信するタスク                                           */
/*************************************************************************/
void task1(VP_INT exinf)
{
            ER                          ercd;
            USER_MSG user_msg;      /* メッセージ実体 */
            USER_MSG *pk_msg;                       /* 受信メッセージ */

            while(1)
            {
                        /* user_msgに送信メッセージを作成 */

                        ercd = snd_mbx(ID_MBX2, (T_MSG *)&user_msg);        /* ID_MBX2へ送信 */

                        /* この間は、user_msgをアクセスしてはならない */

                        /* 送信側のメッセージ処理の完了を待つ */
                        ercd = rcv_mbx(ID_MBX1, (T_MSG **)&pk_msg);                     /* ID_MBX1から受信 */

                        /* 本例では、受信するpk_msgは必ずuser_msgと同じになります */
            }
}

/*************************************************************************/
/*  メッセージを受信するタスク                                           */
/*************************************************************************/
void task2(VP_INT exinf)
{
            ER                          ercd;
            USER_MSG *pk_msg;                       /* 受信メッセージ */

            while(1)
            {
                        ercd = rcv_mbx(ID_MBX2, (T_MSG **)&pk_msg);         /* ID_MBX2から受信 */

                        /* ここに、受信メッセージに応じた処理を記述します */

                        /* 返答メッセージを、受信したメッセージ領域に作成 */
                        ercd = snd_mbx(ID_MBX1, (T_MSG *)pk_msg);           /* ID_MBX1へ送信 */
            }
}
```

おまけ(メールボックス説明補足)

メッセージを送信するタスクはsnd_msgを発行して、特定のメールボックスにメッセージを送ります。この場合、メールボックスに送られるのはメッセージを格納したメモリ領域のアドレスだけで、メッセージそのものがコピーされて送信されるわけではありません。

一方、メッセージを受信するタスクはrcv_msgを発行して、指定したメールボックスからメッセージを受け取ります。すなわち、メールボックスからメッセージの格納アドレスを受け取り、メッセージの内容を読み出します。

このように、メイルボックスでは実際に受け渡しする情報はアドレスだけのため、少ないオーバーヘッドでメッセージの伝達を可能にしています。

メールボックスに送られてきたメッセージはメールボックス内でFIFO順の待ち行列につながれます。これをメッセージキューと呼びます。

タスクはメッセージの受信要求をした時に、メールボックスにメッセージがない場合は、それが到着するまで実行が中断されWAIT状態になり、到着待ちの待ち行列につながれます。このメールボックスに対するタスクの待ち行列もFIFO順で処理されます。

メールボックスを使用したメッセージの交換は、タスクではなくメールボックスを指定して行なわれるので、複数のタスク間でのメッセージの交換を容易に行うことができます。

また、メールボックスを使用することによって、相手のタスクを指定することなく、互いの同期が達成できます。

おまけ(イベントフラグを使ったサンプルプログラム)


```
#include "kernel.h"
#include "kernel_id.h"
void task1(VP_INT exinf)
{
 ER ercd;
 FLGPTN data;
 /* 処理A */

 ercd = wai_flg(ID_FLG1, (FLGPTN)1, TWF_ORW, &data);
 /* 処理C */
}
void task2(VP_INT exinf)
{
 ER ercd;
 /* 処理B */

 ercd = set_flg(ID_FLG1, (FLGPTN)1);
}
```

参考:各仕様の比較

- 本資料は、μITRON3.0/μITRON4.0/T-Kernelの各仕様のうち、代表的な機能とAPIの違いについて比較したものです
- 各機能の分類等については、μITRON4.0仕様に基づいてます

出典元
文書名：『μITRON仕様とT-Kernel仕様の違いについて』 第一版
著者名：エルミック・ウェスコム株式会社

用語

μITRON3.0**仕様**	μITRON4.0**仕様**	T-Kernel
仕様の準拠レベル **レベル**R(Required) **レベル**S(Standerd) **レベル**E(Extended)	**仕様の準拠レベル** **自動車制御用プロファイル** **スタンダードプロファイル**	
システムコール	**サービスコール**	**システムコール**
「タスク」を「タスク部」 **「過渡的な状態」、「タスク独立部」、「準タスク部」を合わせて「非タスク部」**	**タスクのコンテキストをタスクコンテキスト、それ以外を非タスクコンテキスト** **仕様上は過渡的な状態という用語は用いていない** **準タスク部の概念は定義していない**	**「タスク」を「タスク部」** **「過渡的な状態」、「タスク独立部」、「準タスク部」を合わせて「非タスク部」**
システムクロック	**システム時刻**	**システム時刻**
周期起動ハンドラ	**周期ハンドラ**	**周期ハンドラ**
周期起動ハンドラ/アラームハンドラを総称して、タイマハンドラと呼ぶ	**周期起ハンドラ/アラームハンドラ/オーバーランハンドラを総称して、タイムイベントハンドラと呼ぶ**	**周期起動ハンドラ/アラームハンドラを総称して、タイムイベントハンドラと呼ぶ**
メイルボックス	**メールボックス**	**メールボックス**

仕様

μITRON3.0仕様	μITRON4.0仕様	T-Kernel
オブジェクトの生成はシステムコールで要求	オブジェクトの生成は静的APIで記述する(スタンダードプロファイル) サービスコールで生成することも可能	オブジェクトの生成はシステムコールで要求
	静的APIの規定 コンフィギュレータに関する規定	
オブジェクトのID番号は利用者が指定する	オブジェクトのID番号はコンフィギュレータによる自動割付、もしくはサービスコールにより利用者が指定するか自動割付	オブジェクトのID番号は自動割付
カーネルが管理するオブジェクトには拡張情報を設定する	拡張情報を設定するのは、タスク/周期ハンドラ/アラームハンドラのみ	カーネルが管理するオブジェクトには拡張情報を設定する

タスク管理機能

μITRON3.0仕様	μITRON4.0仕様	T-Kernel

C言語記述形式
 	
void task(INT stacd)
{
　;
}
	C言語記述形式
	
	void task(VP_INT exinf)
	{
	　;
	}
	
	exinf:
	　sta_tskで起動した場合stacd
	　act_tskで起動した場合exinf
		C言語記述形式
		
		void task(INT stacd, VP exinf)
		{
		　;
		}

タスクの起動方法 　システムコールsta_tsk	タスクの起動方法 　タスク生成時の属性で起動指定 　サービスコールact_tsk/sta_tsk	タスクの起動方法 　システムコールsta_tsk
タスクのメインルーチンからリターンした場合は、動作は保障されない	タスクのメインルーチンからリターンした場合は、サービスコールext_tskを呼び出した場合と同じ振る舞いをする	関数からの単純なリターン(return)でタスクを終了することはできない(してはいけない)
		ラウンドロビンスケジューリングをサポート

タスク管理機能(API)

機能	μITRON3.0	μITRON4.0	T-Kernel
タスクの生成	cre_tsk	cre_tsk	
タスクの生成(ID**番号自動割付**)		acre_tsk	tk_cre_tsk
タスクの削除	del_tsk	del_tsk	tk_del_tsk
タスクの起動		act_tsk	
タスク起動要求のキャンセル		can_act	
タスクの起動(起動コード指定)	sta_tsk	sta_tsk	tk_sta_tsk
自タスクの終了	ext_tsk	ext_tsk	tk_ext_tsk
タスクの強制終了	ter_tsk	ter_tsk	tk_ter_tsk
タスク優先度の変更	chg_pri	chg_pri	tk_chg_pri
タスクスライスタイム変更			tk_chg_slt
タスク優先度の参照		get_pri	
タスクの状態参照	ref_tsk	ref_tsk	tk_ref_tsk
タスクの状態参照(簡易版)		ref_tst	

タスク付属同期機能

μITRON3.0仕様	μITRON4.0仕様	T-Kernel
自タスクに対し起床要求はできない	自タスクに対し起床要求ができる	自タスクに対し起床要求はできない
自タスクを強制待ちにできない	自タスクを強制待ちにできる	自タスクを強制待ちにできない
自タスクを起床待ちにする要求は永久待ち、タイムアウトありの別々のシステムコールがある	自タスクを起床待ちにする要求は永久待ち、タイムアウトありの別々のサービスコールがある	自タスクを起床待ちにするシステムコールは一つで、永久待ちまたはタイムアウトの指定を行う
		待ち状態の許可/禁止を行う機能がある

タスク付属同期機能(API)

機能	μITRON3.0	μITRON4.0	T-Kernel
起床待ち	slp_tsk	slp_tsk	tk_slp_tsk (tmout=-1)
起床待ち(タイムアウトあり)	tslp_tsk	tslp_tsk	tk_slp_tsk (tmout)
タスクの起床	wup_tsk	wup_tsk	tk_wup_tsk
タスク起床要求のキャンセル	can_wup	can_wup	tk_can_wup
強制待ち状態への移行	sus_tsk	sus_tsk	tk_sus_tsk
強制待ち状態からの再開	rsm_tsk	rsm_tsk	tk_rsm_tsk
強制待ち状態からの再開	frsm_tsk	frsm_tsk	tk_frsm_tsk
自タスクの遅延	dly_tsk	dly_tsk	tk_dly_tsk
タスクイベントの送信			tk_sig_tev
タスクイベント待ち			tk_wai_tev
タスク待ち状態の禁止			tk_dis_wai
タスク待ち状態の解除			tk_ena_wai

同期･通信機能

μITRON3.0仕様	μITRON4.0仕様	T-Kernel
セマフォの獲得/返却の資源数は1	セマフォの獲得/返却の資源数は1	セマフォの獲得/返却の資源数は要求時に指定
	スタンダードプロファイルでは、セマフォの最大資源数として65535以上の値が指定できなければならない	セマフォの最大値として少なくとも65535が指定できなければならない
セマフォの獲得待ちにする要求は永久待ち、タイムアウトありの別々のシステムコールがある	セマフォの獲得待ちにする要求は永久待ち、タイムアウトありの別々のシステムコールがある	セマフォの獲得待ちにするシステムコールは一つで、永久待ちまたはタイムアウトの指定を行う
イベントフラグ待ち時のクリア指定は待ち要求時に指定	イベントフラグ待ち時のクリア指定はイベントフラグの属性で指定	イベントフラグ待ち時のクリア指定は待ち要求時に指定
イベントフラグ待ち解除時のクリアは全ビット0	イベントフラグ待ち解除時のクリアは全ビット0	イベントフラグ待ち解除時のクリアは全ビット0か待ち条件クリアかを要求時に指定
	スタンダードプロファイルではデータキューをサポートすることを規定	
メイルボックスのメッセージ管理がリングバッファ形式かリンク形式かは実装依存	メールボックスのメッセージ管理はリンク形式	メールボックスのメッセージ管理はリンク形式

同期・通信機能(API)
セマフォ

機能	μITRON3.0	μITRON4.0	T-Kernel
セマフォの生成	cre_sem	cre_sem	
セマフォの生成(ID**番号自動割付**)		acre_sem	tk_cre_sem
セマフォの削除	del_sem	del_sem	tk_del_sem
セマフォ資源の返却	sig_sem	sig_sem	tk_sig_sem
セマフォ資源の獲得	wai_sem	wai_sem	tk_wai_sem (tmout=-1)
セマフォ資源の獲得(**ポーリング**)	preq_sem	pol_sem	tk_wai_sem (tmout=0)
セマフォ資源の獲得(**タイムアウトあり**)	twai_sem	twai_sem	tk_wai_sem (tmout)
セマフォの状態参照	ref_sem	ref_sem	tk_ref_sem

同期・通信機能(API)
イベントフラグ

機能	μITRON3.0	μITRON4.0	T-Kernel
イベントフラグの生成	cre_flg	cre_flg	
イベントフラグの生成(ID**番号自動割付**)		acre_flg	tk_cre_flg
イベントフラグの削除	del_flg	del_flg	tk_del_flg
イベントフラグのセット	set_flg	set_flg	tk_set_flg
イベントフラグのクリア	clr_flg	clr_flg	tk_clr_flg
イベントフラグ待ち	wai_flg	wai_flg	tk_wai_**flg** (tmout=-1)
イベントフラグ待ち(**ポーリング**)	pol_flg	pol_flg	tk_wai_flg (tmout=0)
イベントフラグ待ち(**タイムアウトあり**)	twai_flg	twai_flg	tk_wai_flg (tmout)
イベントフラグの状態参照	ref_flg	ref_flg	tk_ref_flg

同期・通信機能(API)
データキュー

機能	μITRON3.0	μITRON4.0	T-Kernel
データキューの生成		cre_dtq	
データキューの生成(ID**番号自動割付**)		acre_dtq	
データキューの削除		del_dtq	
データキューへの送信		snd_dtq	
データキューへの送信(ポーリング)		psnd_dtq	
データキューへの送信(タイムアウトあり)		tsnd_dtq	
データキューへの強制送信		fsnd_dtq	
データキューからの受信		rcv_dtq	
データキューからの受信(ポーリング)		prcv_dtq	
データキューからの受信(タイムアウトあり)		trcv_dtq	
データキューの状態参照		ref_dtq	

同期・通信機能(API)
メールボックス

機能	μITRON3.0	μITRON4.0	T-Kernel
メールボックスの生成	cre_mbx	cre_mbx	
メールボックスの生成(ID**番号自動割付**)		acre_mbx	tk_cre_mbx
メールボックスの削除	del_mbx	del_mbx	tk_del_mbx
メールボックスへの送信	snd_msg	snd_mbx	tk_snd_mbx
メールボックスからの受信	rcv_msg	rcv_mbx	tk_rcv_mbx (tmout=-1)
メールボックスからの受信(**ポーリング**)	prcv_msg	prcv_mbx	tk_rcv_mbx (tmout=0)
メールボックスからの受信(**タイムアウトあり**)	trcv_msg	trcv_mbx	tk_rcv_mbx (tmout)
メールボックスの状態参照	ref_mbx	ref_mbx	tk_ref_mbx

時間管理機能

μITRON3.0仕様	μITRON4.0仕様	T-Kernel
システムクロックは1985年1月1日を0とした1ミリ秒数のカウンタ	システム時刻はシステム初期化時に0に初期化したカウンタ	システムクロックは1985年1月1日を0とした1ミリ秒数のカウンタ
システムクロックを変更した場合に、それまで時間待ちしていたタスクや起動を待っていたハンドラの動作タイミングが狂う可能性がある	システム時刻を変更した場合にも、相対時間を用いて指定されたイベントの発生する実時刻は変化しない	システム時刻を変更した場合にも、相対時刻は変化しない
システムクロックのビット数を48ビットと推奨する	システム時刻のビット数に関する推奨値を定めない	システム時刻は64ビット符号付整数
周期起動ハンドラとアラームハンドラは定義する	周期ハンドラとアラームハンドラは生成する(ID番号で管理)	周期ハンドラとアラームハンドラは生成する(ID番号で管理)
	周期ハンドラに起動位相という概念を導入	周期ハンドラに起動位相という概念を導入

時間管理機能(API)

機能	μITRON3.0	μITRON4.0	T-Kernel
システム時刻の設定(実際の時間)	set_tim		tk_set_tim
システム時刻の参照(実際の時間)	get_tim		tk_get_tim
システム時刻の設定		set_tim	
システム稼働時間の参照		get_tim	tk_get_otm
周期ハンドラの生成		cre_cyc	
周期ハンドラの生成(ID番号自動割付)		acre_cyc	tk_cre_cyc
周期ハンドラの定義	def_cyc		
周期ハンドラの削除		del_cyc	tk_del_cyc
周期起動ハンドラの活性制御	act_cyc		
周期ハンドラの動作開始		sta_cyc	tk_sta_cyc
周期ハンドラの動作停止		stp_cyc	tk_stp_cyc
周期ハンドラの状態参照	ref_cyc	ref_cyc	tk_ref_cyc

システム状態管理機能(API)

機能	μITRON3.0	μITRON4.0	T-Kernel
タスク優先順位の回転	rot_rdq	rot_rdq	tk_rot_rdq
実行状態のタスクID**の参照**	get_tid	get_tid	tk_get_tid
CPU**ロック状態への移行**	loc_cpu	loc_cpu	
CPU**ロック状態の解除**	unl_cpu	unl_cpu	
ディスパッチ禁止	dis_dsp	dis_dsp	tk_dis_dsp
ディスパッチ許可	ena_dsp	ena_dsp	tk_ena_dsp
コンテキストの参照		sns_ctx	
CPU**ロック状態の参照**		sns_loc	
ディスパッチ禁止状態の参照		sns_dsp	
ディスパッチ保留状態の参照		sns_dpn	
システムの状態参照	ref_sys	ref_sys	tk_ref_sys
省電力モード設定			tk_set_pow

非タスクコンテキスト

μITRON3.0**仕様**	μITRON4.0**仕様**	T-Kernel
タスク独立部用のシステムコールの名称は、ixxx_yyy**とする**	**非タスクコンテキスト用のサービスコールの名称は、**ixxx_yyy**とする**	**タスク独立部用のシステムコールは、タスク用のシステムコールと同じ名称**
タスク独立部用のシステムコールの種類は実装依存		

非タスクコンテキスト(API)/1

機能	μITRON3.0	μITRON4.0	T-Kernel
タスクの起動		iact_tsk	
タスクの起動			tk_sta_tsk
タスク優先度の変更	ichg_pri		
タスクの起床	iwup_tsk	iwup_tsk	tk_wup_tsk
待ち状態の強制解除	irel_wai	irel_wai	tk_rel_wai
強制待ち状態への移行	isus_tsk		
強制待ち状態からの再開	irsm_tsk		
強制待ち状態からの強制再開	ifrsm_tsk		
タスク例外処理の要求		iras_tex	
セマフォ資源の返却	isig_sem	isig_sem	tk_sig_sem
イベントフラグのセット	iset_flg	iset_flg	tk_set_flg
データキューへの送信(ポーリング)		ipsnd_dtq	
データキューへの強制送信		ifsnd_dtq	
メールボックスへの送信	isnd_msg		

非タスクコンテキスト(API)/2

機能	μITRON3.0	μITRON4.0	T-Kernel
メッセージバッファへの送信	ipsnd_mbf		
固定長メモリブロックの獲得	ipget_blf		
可変長メモリブロックの獲得	ipget_blk		
タイムティックの供給		isig_tim	
タスクの優先順位の回転	irot_rdq	irot_rdq	tk_rot_rdq
実行状態のタスクID**の参照**		iget_tid	tk_get_tid
CPU**ロック状態への移行**		iloc_cpu	
CPU**ロック状態の解除**		iunl_cpu	
タスクの強制待ち			tk_sus_tsk
タスクイベントの送信			tk_sig_tev
周期ハンドラの動作開始			tk_sta_cyc
周期ハンドラの動作停止			tk_stp_cyc
アラームハンドラの動作開始			tk_sta_alm
アラームハンドラの動作停止			tk_stp_alm

デバイスドライバ

- デバイスドライバとは、特定の入出力デバイス(周辺機器などのハードウェア)を制御し、アプリケーションソフトウェアに対して抽象化したインタフェースを提供するためのソフトウェア

- μITRON仕様では、デバイスを制御するプログラムを特別に管理する機能はなく、汎用OS のデバイス･ドライバのようなものは規定されていない
  - ただしハードウェアを制御するプログラムは必要
  - デバイスドライバの形式として、DIC(Device Interface Component)という概念が提案されている
    - 参考:「組込みシステムにおけるPDIC機能ガイドライン」(T-Engine Forum)

- 開発時に規定した形式またはミドルウェア等で規定されている形式でデバイスドライバを作成する

µITRONのデバイスドライバ

- 大きく分けて、二つの形式
  - ドライバ処理の中心は割込みハンドラ
    - 利点：割込み応答性が高い、タスク動作の影響を受けにくい
    - 欠点：データ処理の時間が長い場合は他の割込み処理の応答性能に影響が出る場合もある、待ち状態を作ることが難しい
  - 割込みハンドラはデバイスドライバタスクを起動する程度で、処理の中心はデバイスドライバタスク
    - 利点：他の割込み処理を阻害する等の影響が出にくい、データ処理途中でも、他の優先度が高いタスクが動作可能
    - 欠点：割込みハンドラでデータ処理する形式よりも応答性が低い、他の優先度が高いタスクの動作によって、処理性能に差が生じる

【講座】T-Kernel/ITRON入門テキスト「μITRON入門」

著者　TRON Forum