

μITRON入門

T-Engine Forum
T-Engine 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

2 トロンと組込みシステム

3 μ ITRON入門

4 μ ITRON開発手順

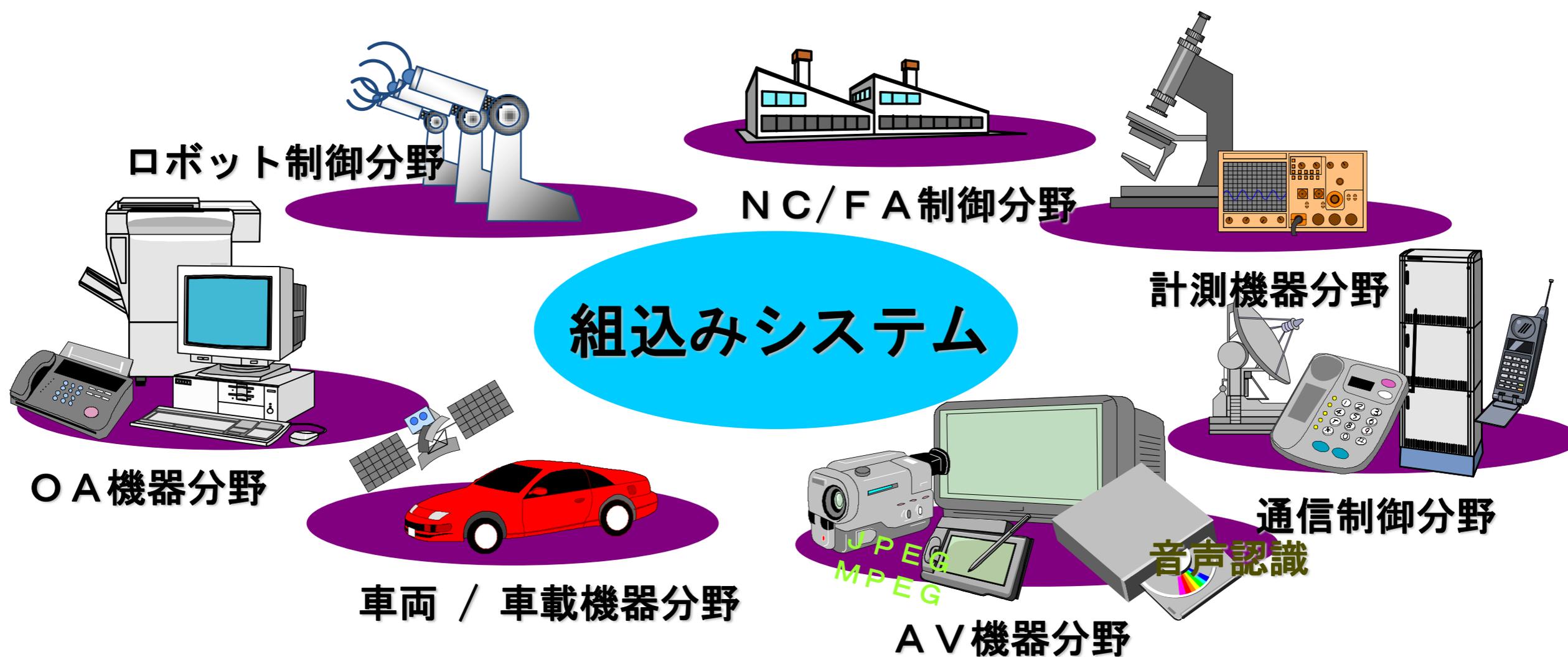
5 μ ITRONプログラミング

6 参考資料・付録など

組み込みシステムとは

- ▶ 組み込みシステム=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
- ▶ (例)
 - 家電製品の制御システム
 - ファックスやコピー機の制御
 - 自動車の制御システム
 - 携帯電話、スマートフォン
 - など...

どのようなものが 組み込みシステムか？



マイコン内蔵炊飯器、洗濯機、コピー機、FAX、
携帯情報端末、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
- ▶ ⇒ビッグデータ/オープンデータ化が進むとともに、ウェ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。

組み込みシステムの特徴(1)

- ▶ 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

- ▶ 必要最小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

組込みシステムの特徴(2)

- ▶ 専用化されたシステム
- ▶ 厳しいリソース制約
- ▶ 高い信頼性
 - パソコンがフリーズしてもお客は怒らないが、カメラがフリーズしたら、大クレーム！
- ▶ システムの改修に多大なコスト
- ▶ リアルタイム性

リアルタイム処理とは (コピー・プリンタ・ファックス複合機の例)



リアルタイム処理とは (主婦兼母親の朝)

7:00 太郎(5歳)を
起こす

7:15 花子(0歳)
おしっこで泣く
→オムツ交換

8:15 太郎スクール
バスに送る

7:05 湯が沸く
→紅茶を入れる

7:15 トースターが終了→
トーストを皿に

8:10 田中さんから電話
→夕方の買物の約束

8:28 クリーニング屋
→受け取りと支払い





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分後までに出示なさい(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

続き・・・リアルタイムシステムとは

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・・・、

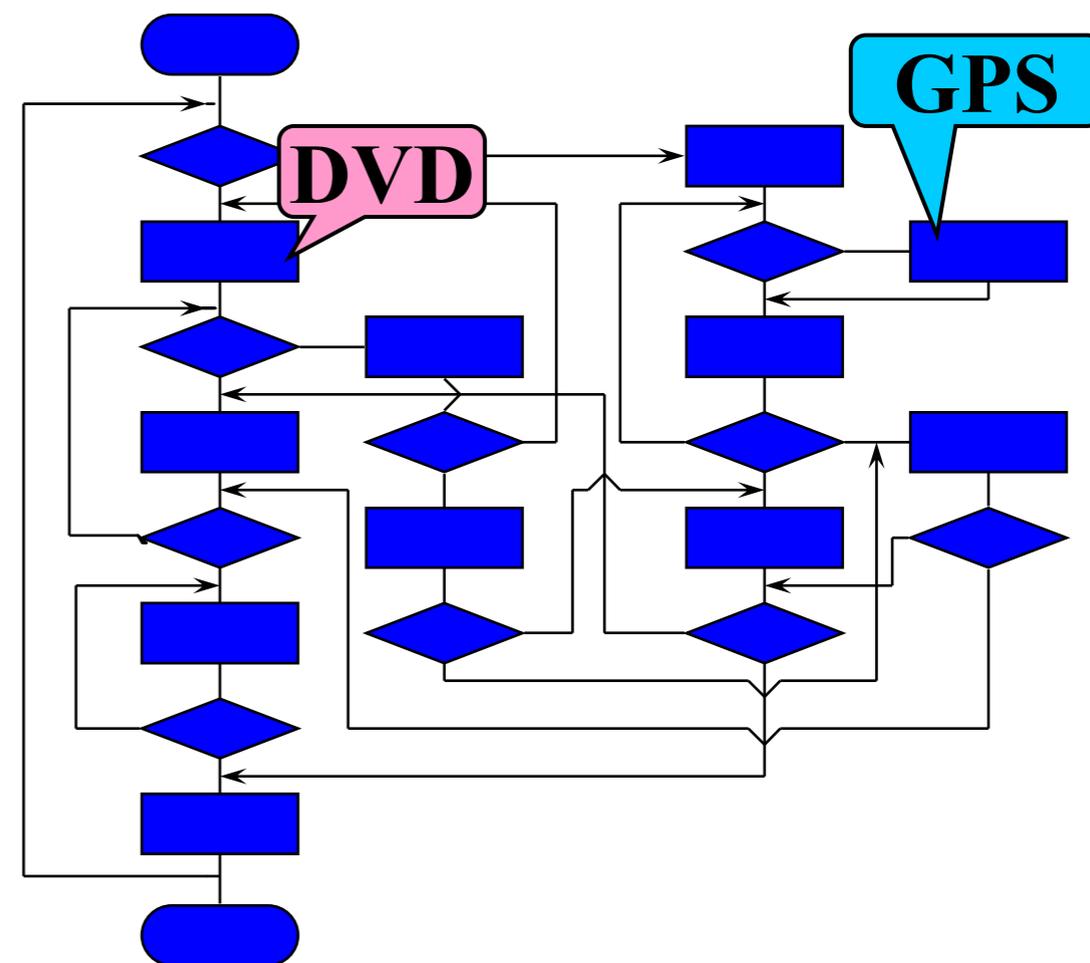
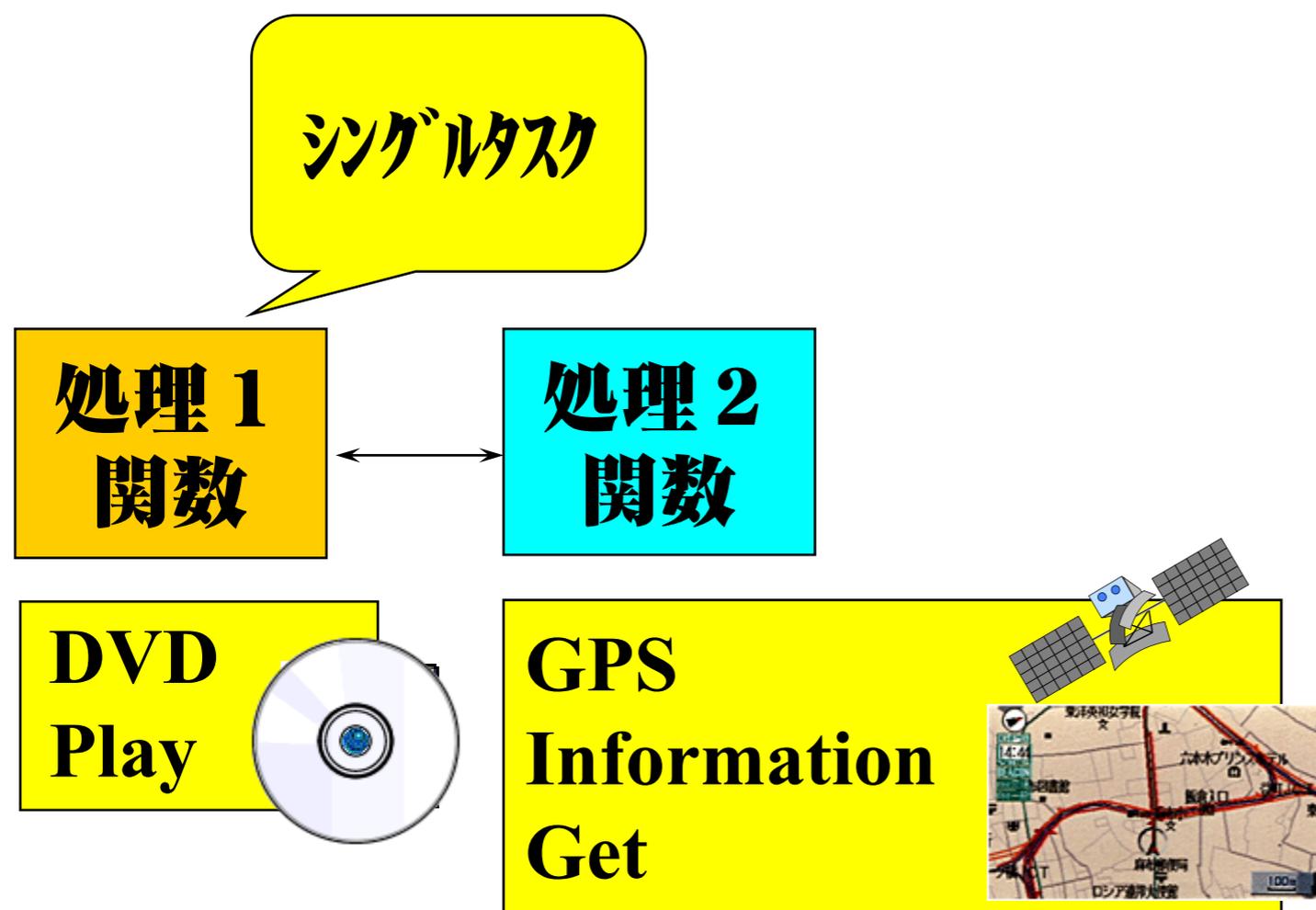


洒落にならないっ！

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

組込み機器制御 (シングルタスク)



**プログラムが複雑
→ 規模が大きくなると管理出来ない！**

リアルタイムとマルチタスク

車の運転の例

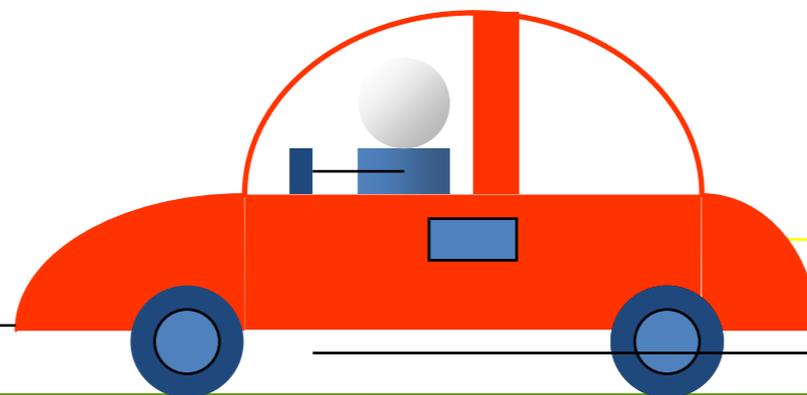
スピードが50km/h
以下になったら
アクセルを踏む
<2秒以内>

前方30m以内に
歩行者を発見したら
ブレーキを踏む
<0.1秒以内>

前方30m以内に
赤信号を発見したら
ブレーキを踏む
<1秒以内>

スピードが50km/h
以上になったら
アクセルを離す
<3秒以内>

燃料計の針が“E”
を指したらガソリン
スタンドに入る
<15分以内>



制限時間内に**複数**の処理を行う という

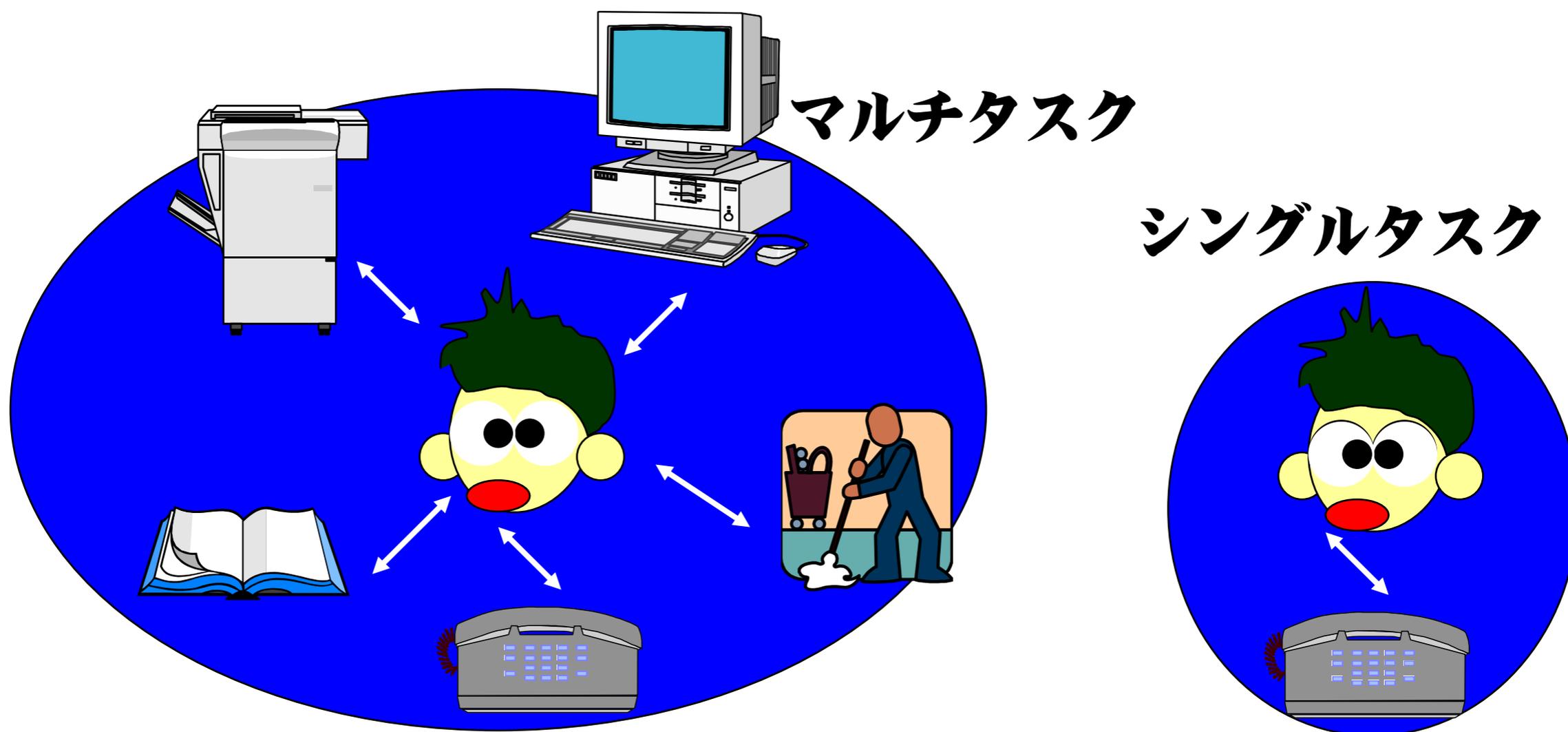
リアルタイムシステム

は、マルチタスクシステム

になる場合が多い！

マルチタスクとは

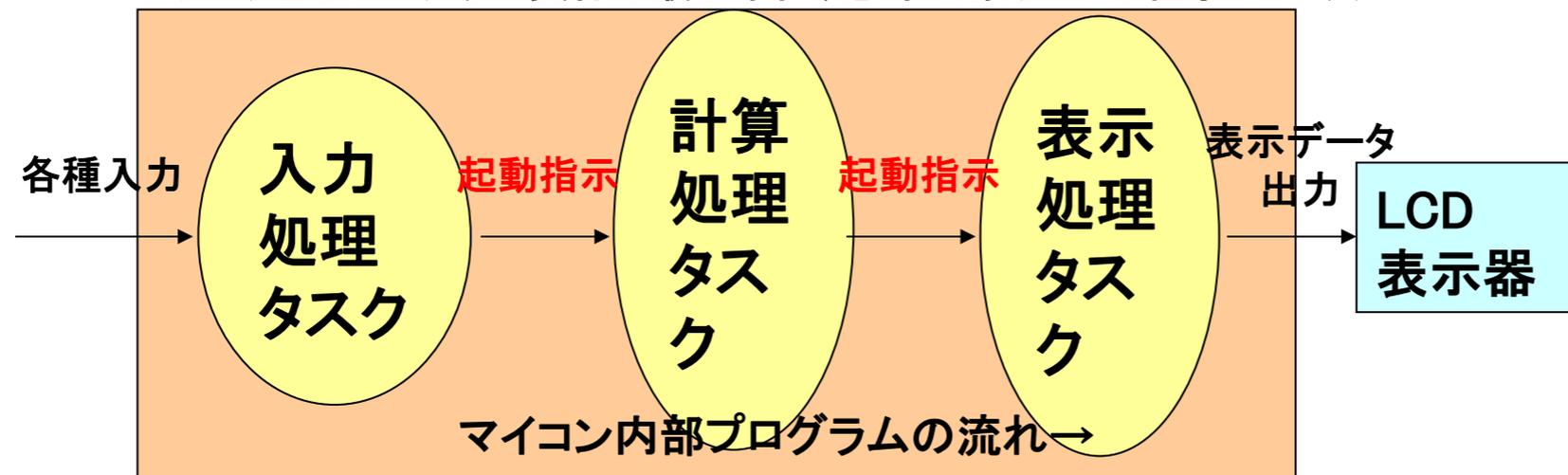
- ▶ 複数の処理を同時に実行すること=マルチタスク処理



マルチタスクとは

プログラムはタスク(Task)単位で実行

※タスクとはプログラム実行の最小単位、意味のある1つの仕事のくり(モジュールや関数のこと)



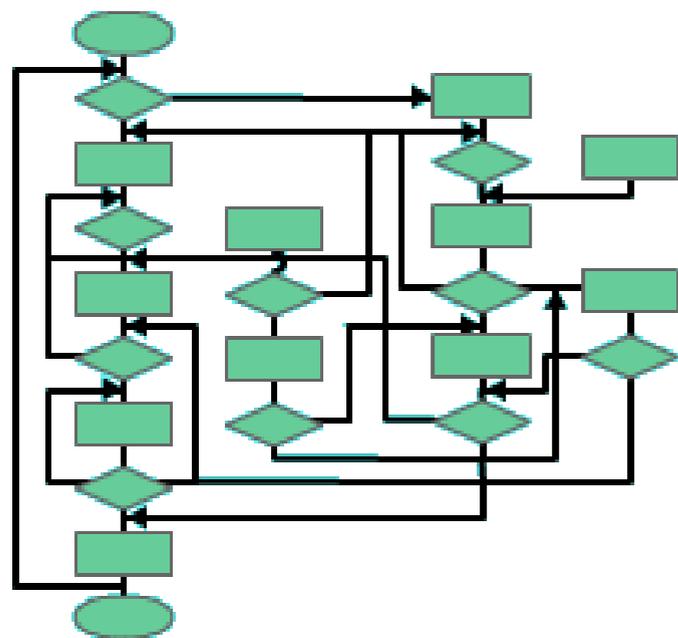
タスクスケジューリング

※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
 ※タスク分割には一般的に構造化設計手法を利用

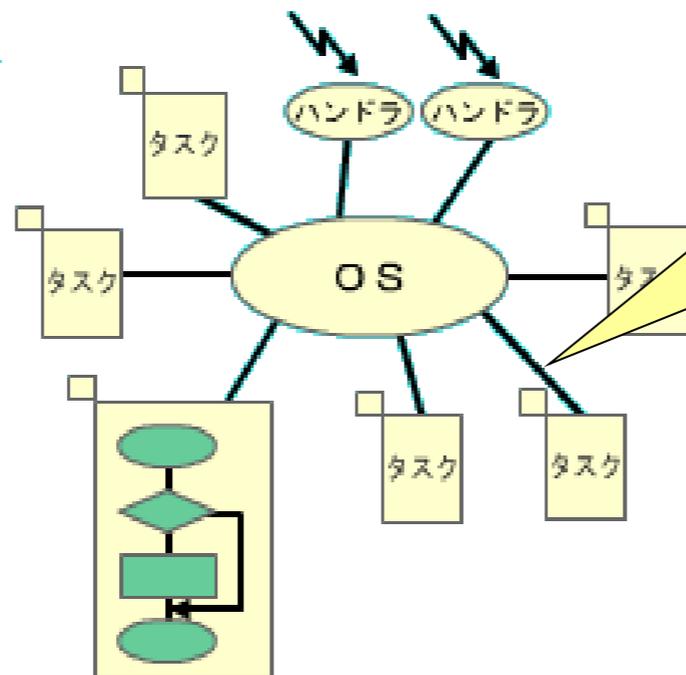
WindowsなどのOSはスケジューリング機能が無いため、ユーザがWordやExcelを起動するが、マイコンプログラムでは、あらかじめタスクに優先度をつけておき、OSはその優先度に従いタスクを実行する。よって、リアルタイムOSのことをタスクスケジューラと呼ぶことがある。

マルチタスクとは

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

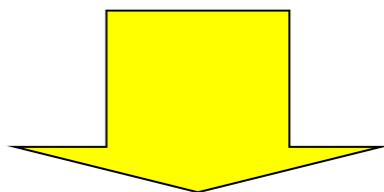
独立性/信頼性の高いタスクの取捨選択で、各種製品展開が短期間で可能となる。

Task deletion support (Task削除で対応) → HDD external connection function (HDD外部接続機能) → All CH simultaneous recording function (全CH同時録画機能) → Task addition support (Task追加で対応)

ローエンド機 ← Blu-ray レコーダ → ハイエンド機

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

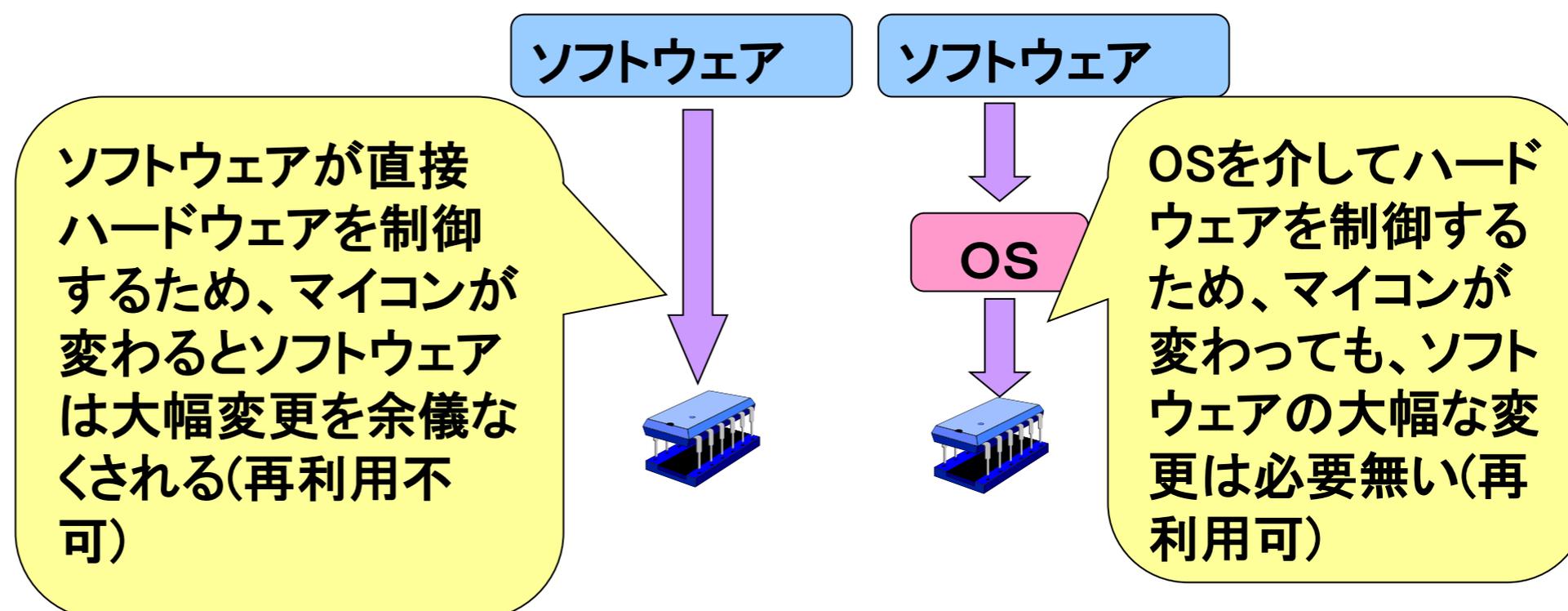
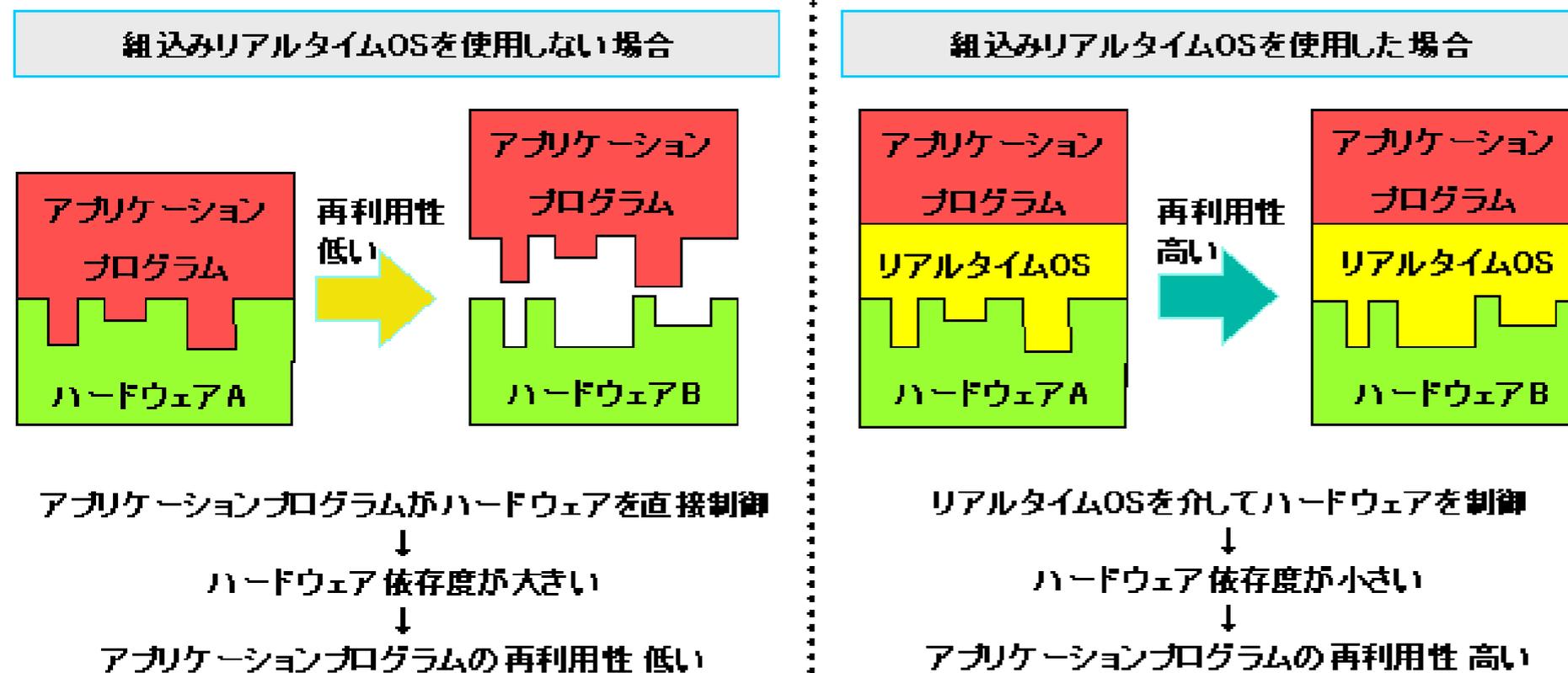


- ▶ タスクスケジューリング

RTOS導入のメリット

- ▶ 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・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すむ。
- ▶ 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



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、ソフトウェアの流用や検証などのノウハウ共有が難しい



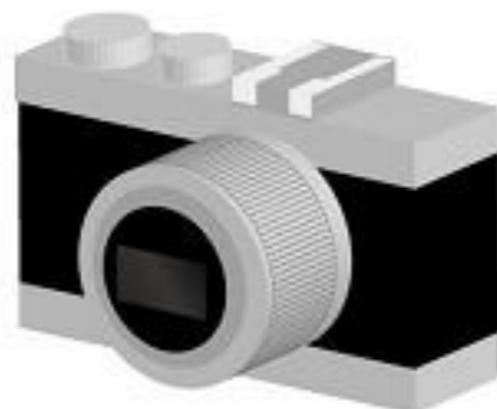
壁

アプリケーション・ソフトウェア

ミドルウェア

リアルタイムOS

ハードウェア



壁

アプリケーション・ソフトウェア

ミドルウェア

リアルタイムOS

ハードウェア



アプリケーション・ソフトウェア

ミドルウェア

リアルタイムOS

ハードウェア

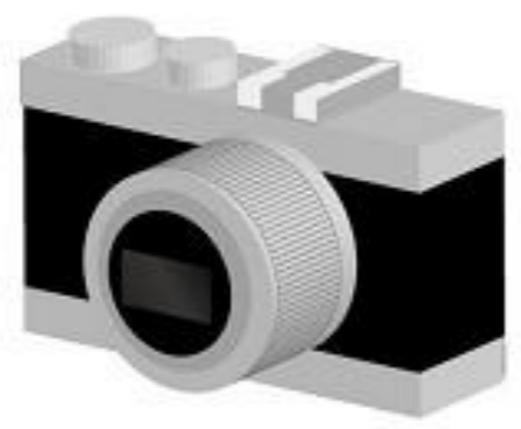
RTOS導入のメリット

～例えばこんなことができる～

壁を取り払うことで、ソフトウェアの流用や検証などのノウハウ共有が容易に



アプリケーション・ソフトウェア



アプリケーション・ソフトウェア



アプリケーション・ソフトウェア

ミドルウェア

ミドルウェア

リアルタイムOS

ハードウェア

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 μ ITRON入門

4 μ ITRON開発手順

5 μ ITRONプログラミング

6 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
- ▶ 組み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 μ ITRON 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

μITRON4.0仕様

- ・標準化仕様の強化
- ・スケーラビリティの向上
- ・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

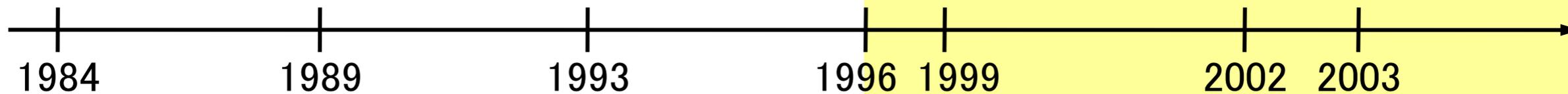
- ・保護機能拡張

【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)

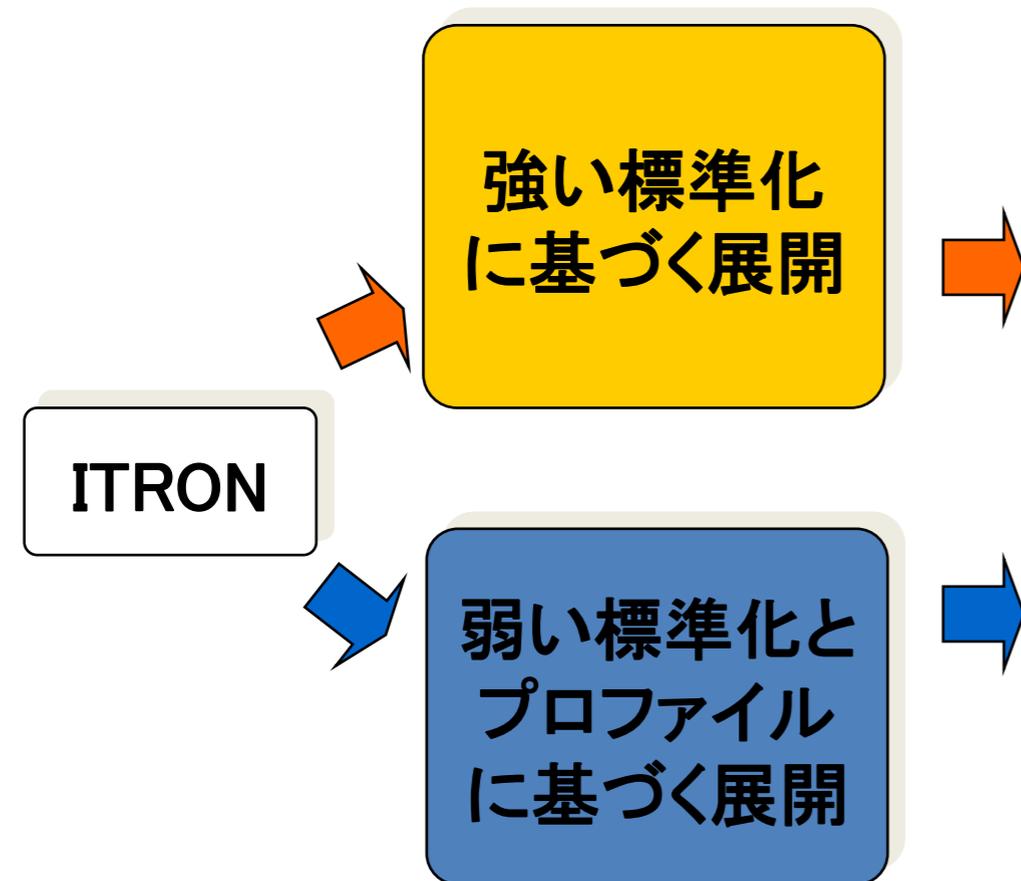
(第1フェーズ)
(第2フェーズ)

トロンプロジェクト発足



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。

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

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)

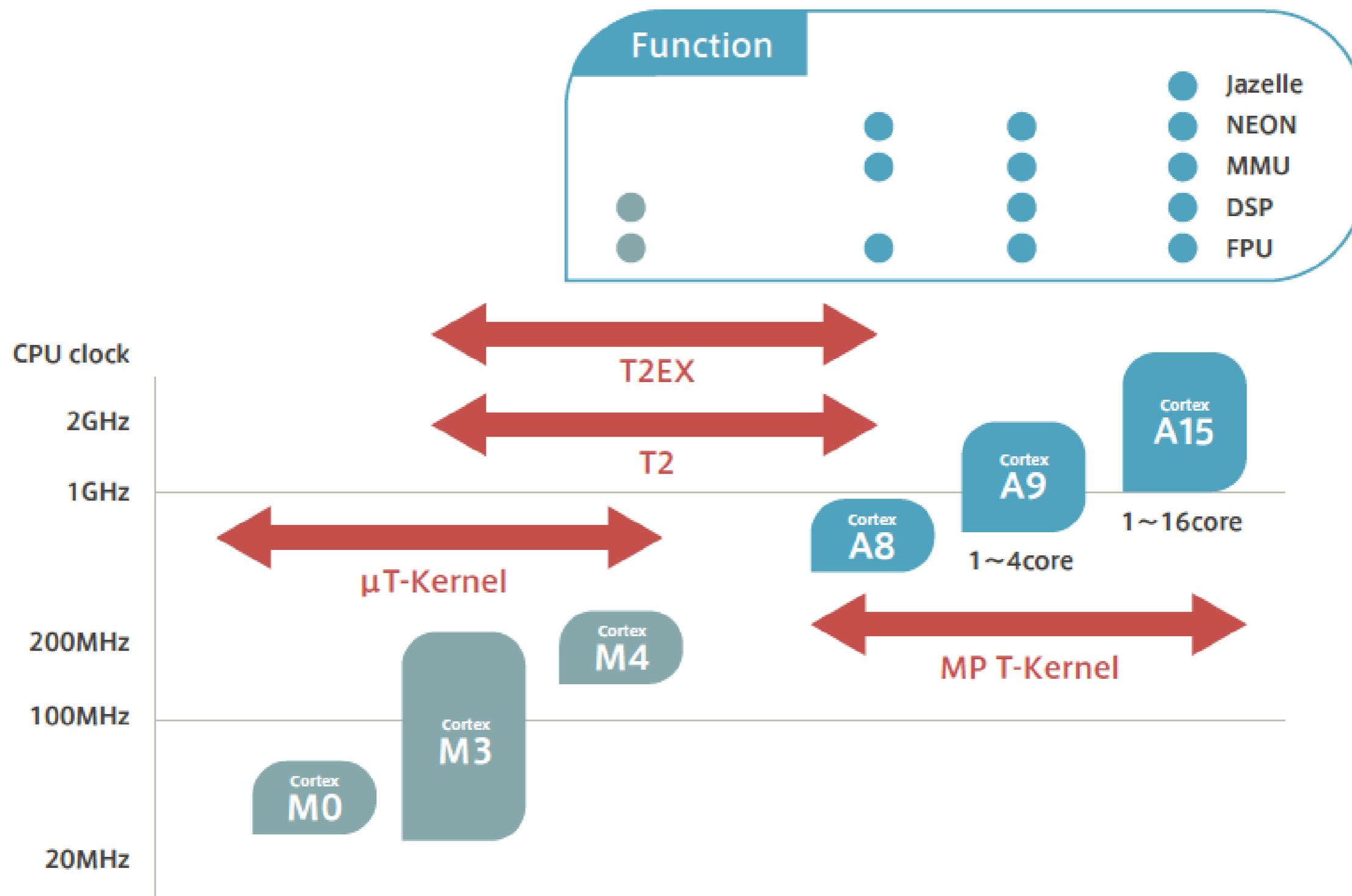
μITRON 4.0とT-Kernelの 機能比較

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はデータキュー無
拡張同期・通信機能	○	○	
メモリ・プール機能	○	○	
時間管理機能	○	△	TKはオーバラン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はサブシステムが同等
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	×	○	

uITRON適用範囲

- 8-32ビットRISC/CISC系マイコンに適用可能
- 各メーカー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)
- 各マイコンメーカーをターゲットにした製品を準備(3rd Party)
 - ・ グレープシステム
 - ・ イーソル
 - ・ イーフォース
 - ・ エーアイコーポレーション
 - ・ ミスポ
 - ・ 他(あくまで一例です)

T-Kernel適用範囲(ARM core)



Micro Controller

Application Processor

参考:その他の組み向けOS

- ▶ 組み込みLinux
 - UnixクローンOS Linuxの組み込み版
- ▶ 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

1 組み込みシステムとマルチタスク・リアルタイム処理

2 トロンと組み込みシステム

3 **μITRON**入門

4 μITRON開発手順

5 μITRONプログラミング

6 参考資料・付録など

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。

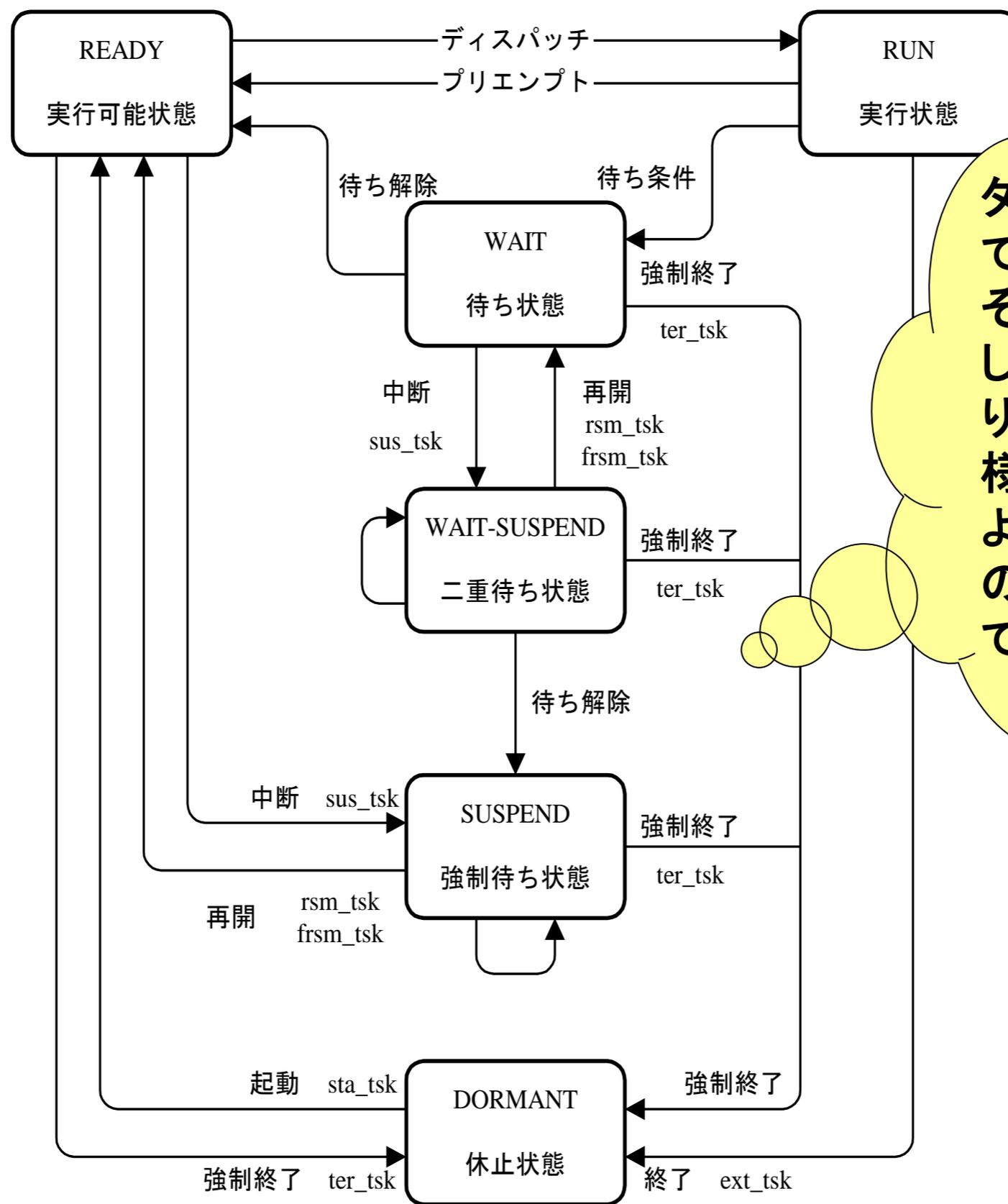
ここでは最低限の事項について記載します。

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。 μ ITRON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。

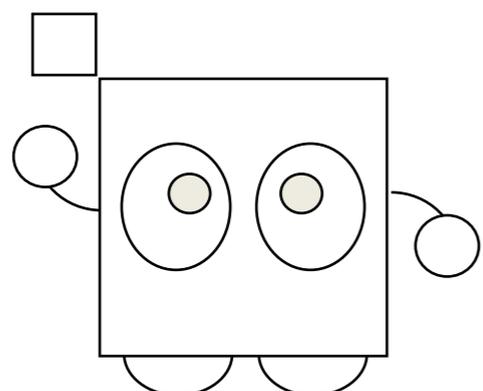
タスクの状態(1)

- ▶ **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

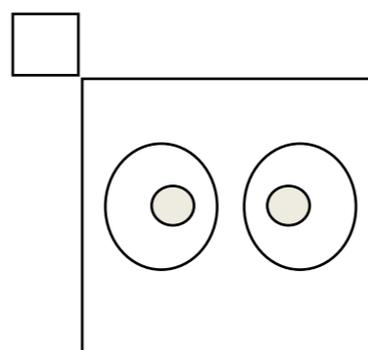
タスクの状態(2)

- ▶ **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

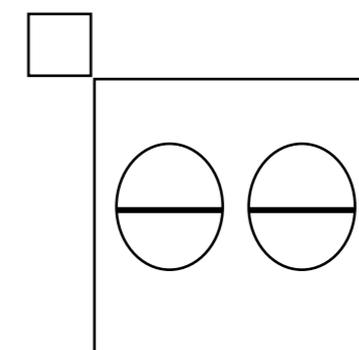
タスクの状態(アイコン)



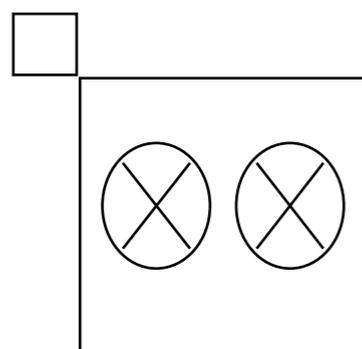
実行状態
RUNNING



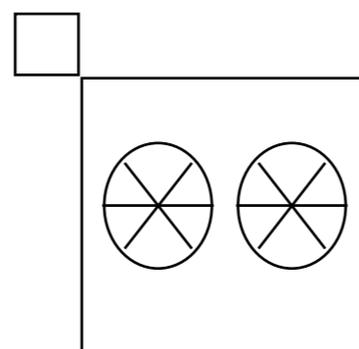
実行可能状態
READY



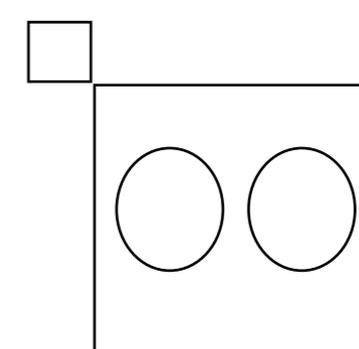
待ち状態
WAIT



強制待ち状態
SUSPEND

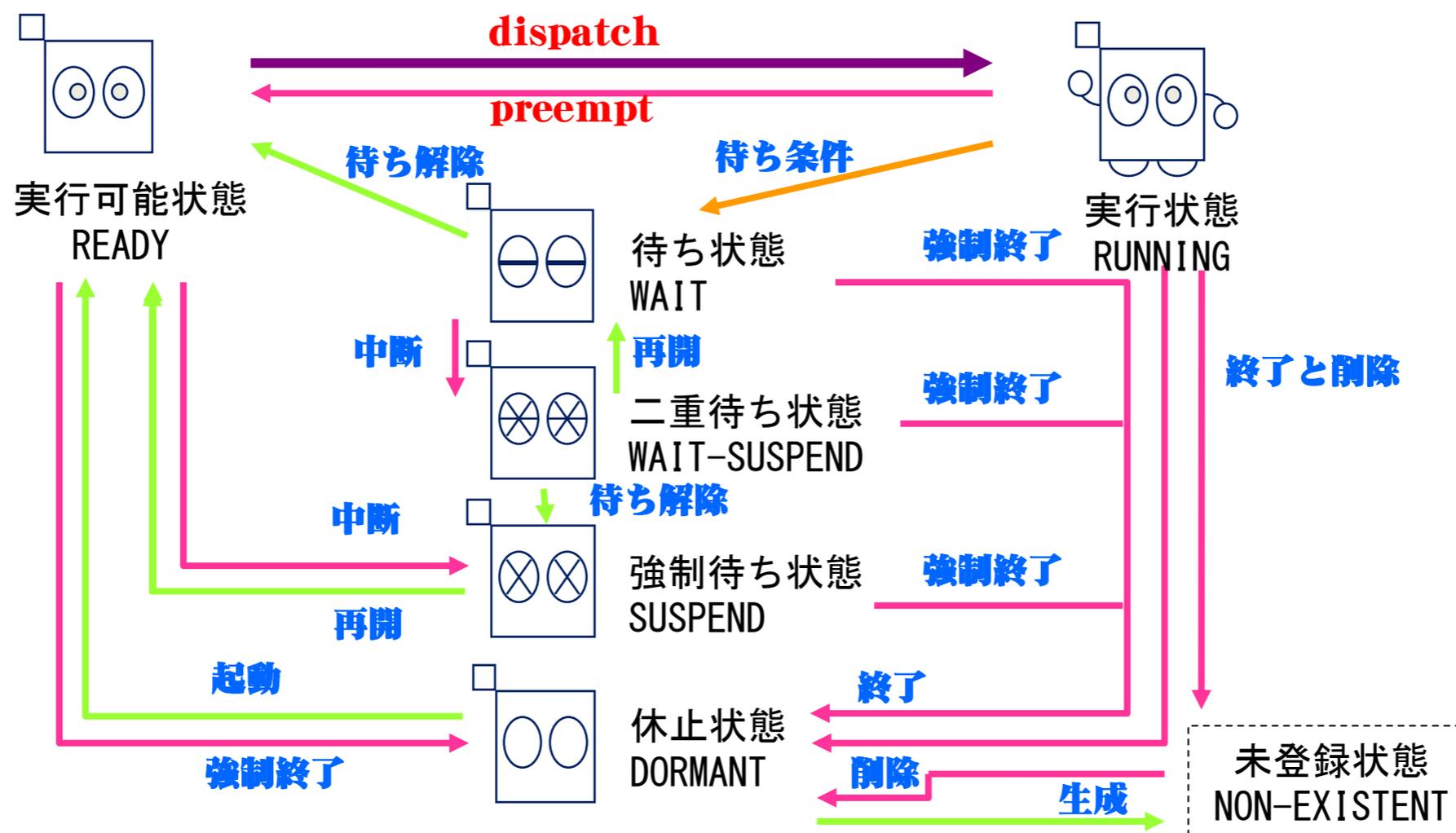


二重待ち状態
WAIT-SUSPEND

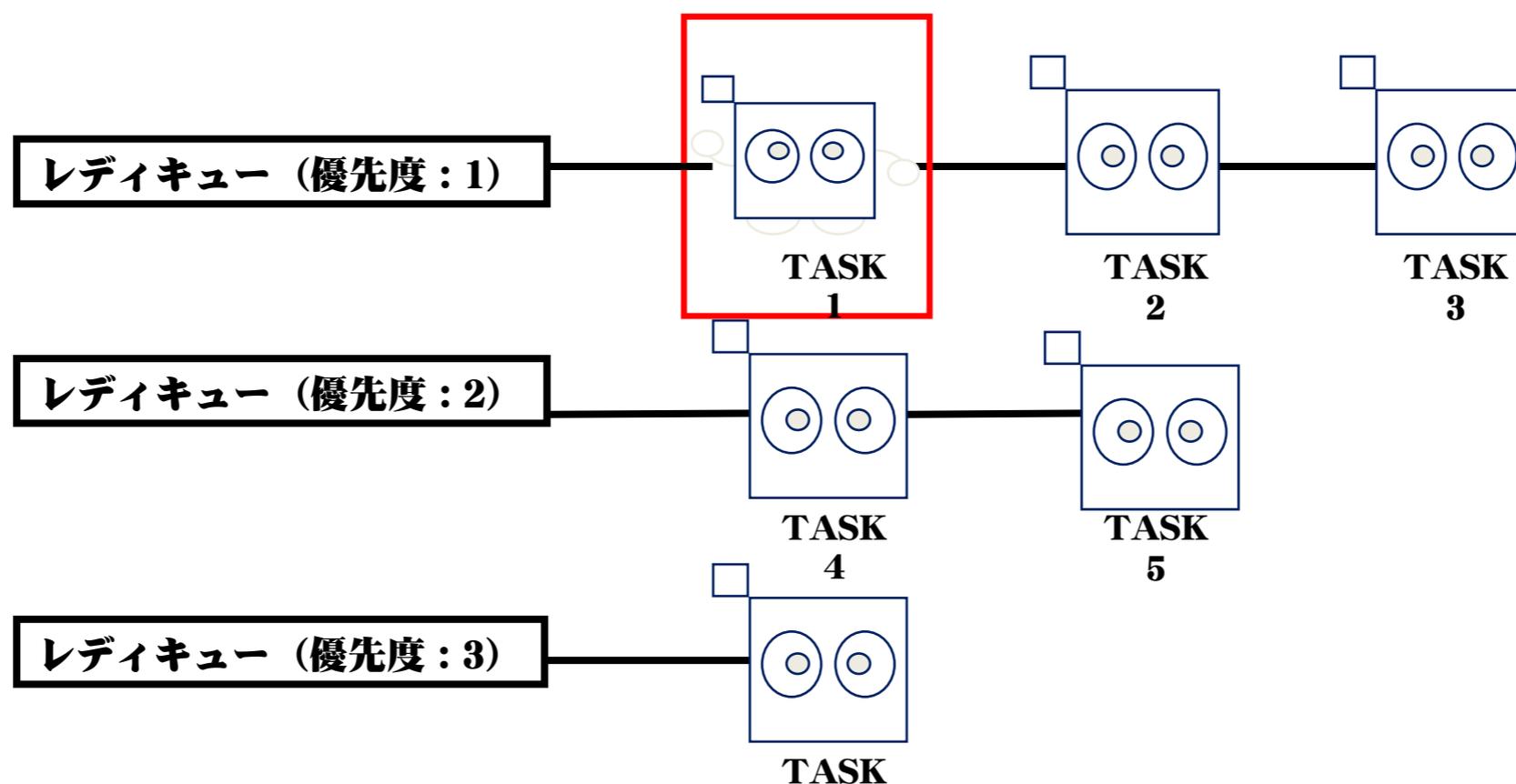


休止状態
DORMANT

タスク遷移の様子

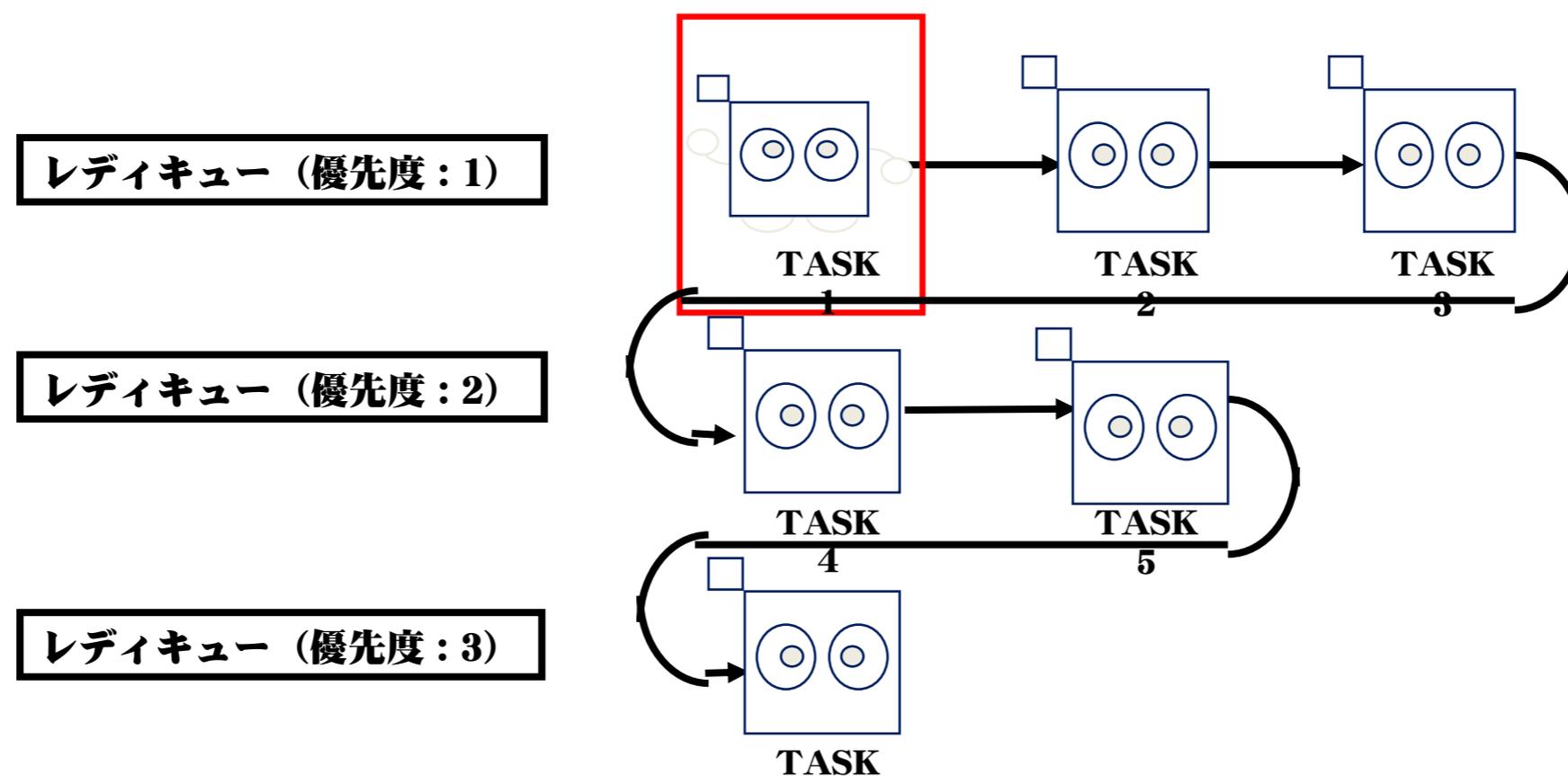


レディーキュー



- ・ **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**

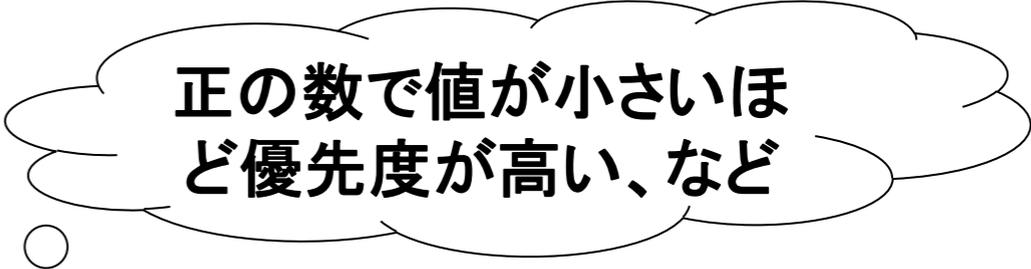
タスクの実行順序



- ・ 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ーキューに並んだ順 (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)
- ▶ μ ITRON3.0/4.0/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・・・

T-Engine フォーラム

ホーム | フォーラム案内 | TRON PROJECT | T-Kernel PROJECT | 仕様書 | イベント・セミナー | 参考資料

ホーム > OnWeb セミナー > TTVタスクシミュレータ

TTVタスクシミュレータ

TTV (Task Trace View) タスク動作シミュレータ

T-Engineフォーラムの
サイト[イベント・セミ
ナー]→[TTVタスクシ
ミュレータ]を選択

TTVは複数のタスクが状態を変えながら並列実行(マルチタスク)する様子をシミュレートする教材です。シミュレートできるタスク数は3、タスクの状態数は4(RUNNING,READY,WAITING,DORMANT(休止))、発行するコマンド数は4(Start,Wait,WakeUp,Exit)です。コマンドをタスクに発行しながら動作を見てください。各タスクの優先度も変えることができます。すべてのタスクの初期状態はDORMANTなのでSTARTコマンドを該当タスクに発行するのが最初です。実行タスクがない場合はアイドル状態となります。その他の説明はTTVのヘルプ(「?」ボタンをクリック)を参照してください。

★TTV シミュレーション

TASK TRACE VIEW (TTV)

優先度 [] [] [] [?]

ID	PRI	NAME	State
1	1	T1	D
2	2	T2	D
3	3	T3	D
		IDLE	
		RTOS	

<<HISTORY>>

各タスク実行結果 | BUTTON HISTORY COD

T-Engine Forum 2.0
リファレンス... 発売

T-Kernel 2.0を一般公開しました。

T-Engineフォーラムのホームページをリニューアルしました。

ダウンロード・仕様書更新情報

- T-Kernel2.0ダウンロード開始
- T-Kernel2.0仕様書公開
- T-Engine/SH7780 Version 3B 株式会社 ルネサス テクノロジー製コンパイラ版

プレスリリース

- T-Engineフォーラムが2010年度組込みシステムにおけるリアルタイムOSの利用動向に関するアンケート調査報告書を公開
- TRONSHOW2011プレス発表 進化するTRON
- 中国政府がT-Engine Forum Chinaを設立

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、 μ ITRON 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。

μ ITRON4.0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

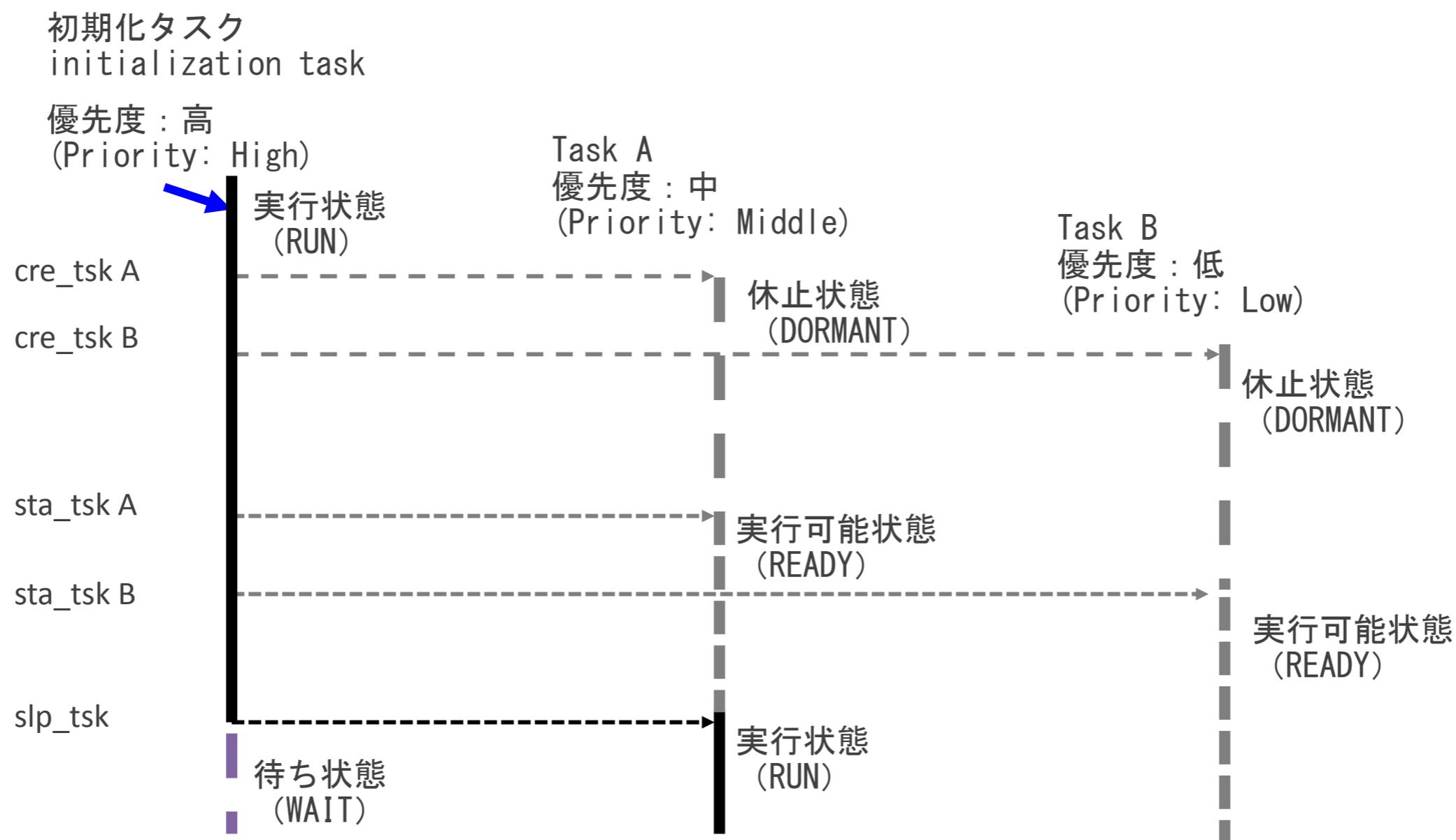
cre_tsk()	タスク生成
del_tsk()	タスク削除
sta_tsk()	タスク起動
ext_tsk()	自タスク終了
exd_tsk()	自タスク終了と削除
ter_tsk()	他タスク強制終了
ref_tsk()	タスク状態参照

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。

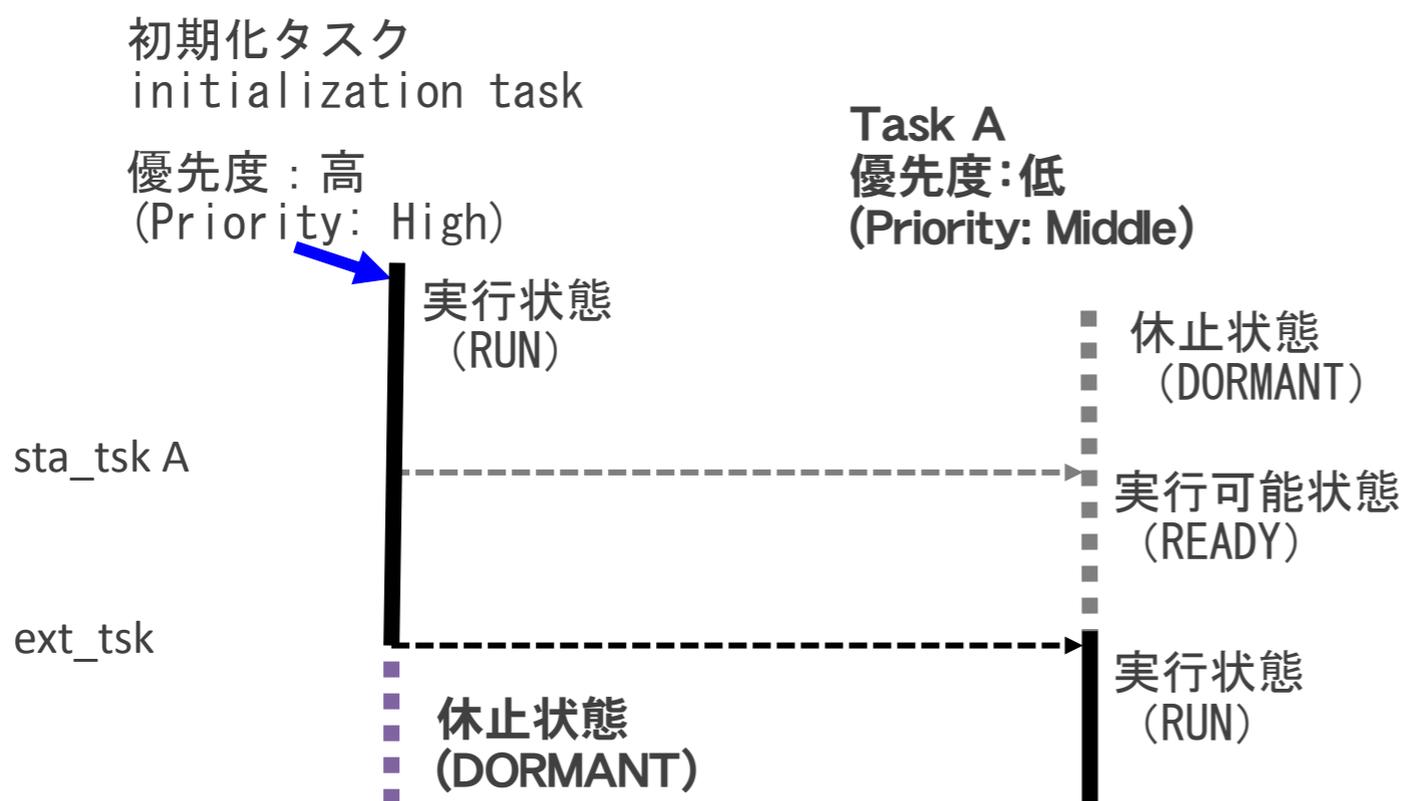
初期化タスクの流れ



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。

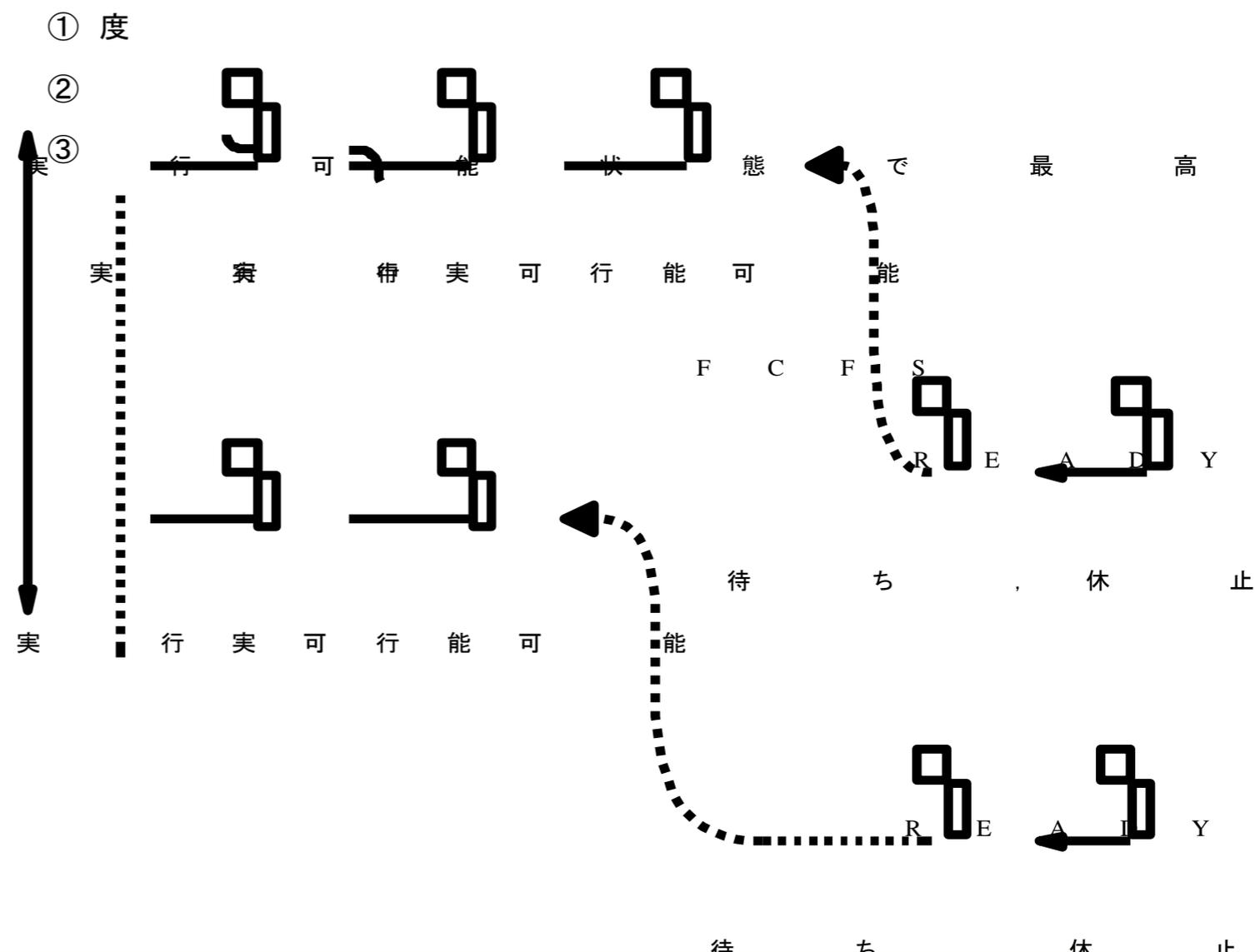
次はタスクを強制終了させる場合です。タスク実行中に異常が発生し、タスクの実行を中断しなければならないとき、`ter_tsk`を発行して、他のタスクを強制的に終了させることができます。これは、**READY**、**WAIT**、**WAIT-SUSPEND**または**SUSPEND**状態のタスクを**DORMANT**状態にします。

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がり変えられます。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

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

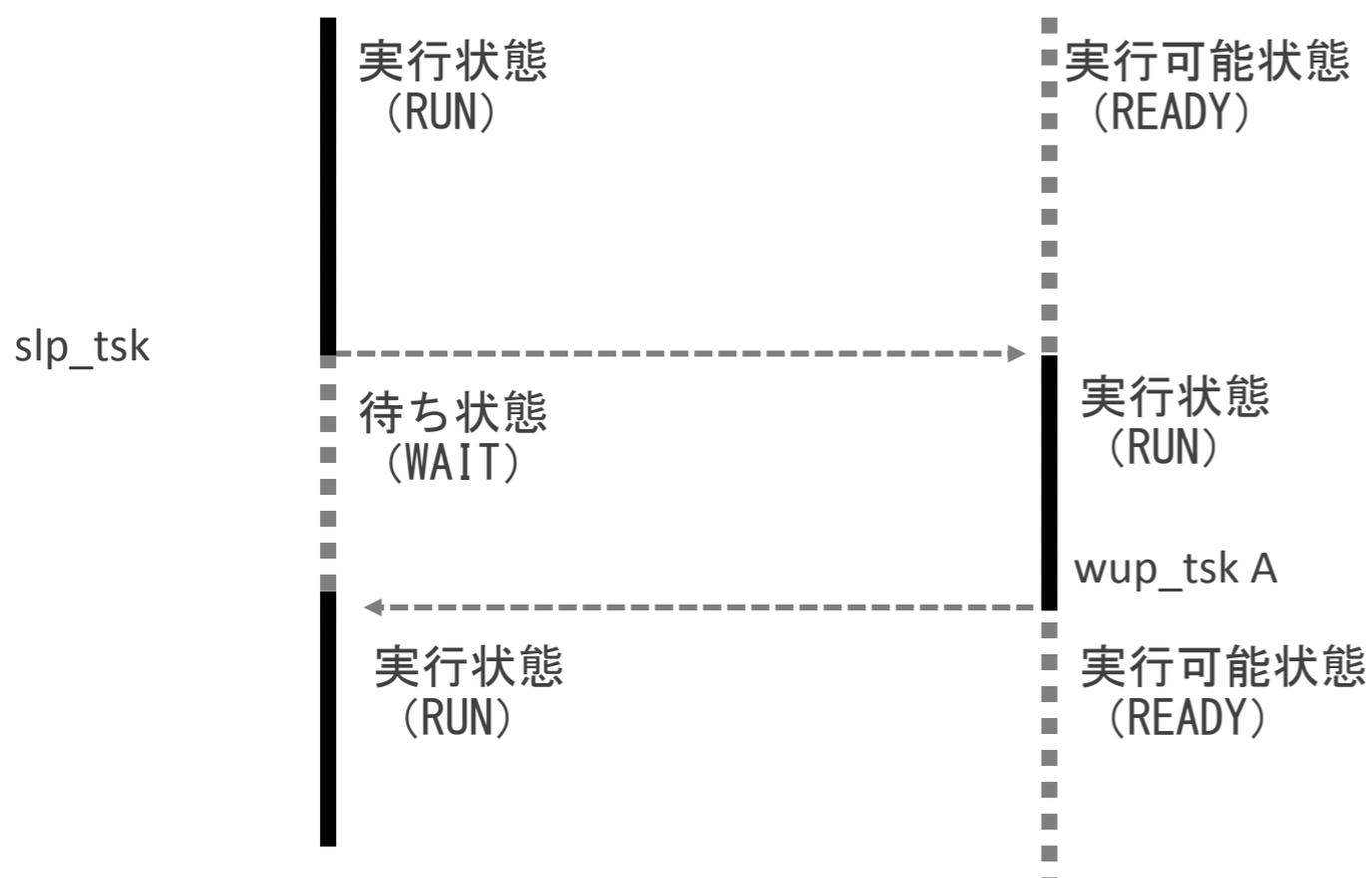
タスク付属同期機能

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

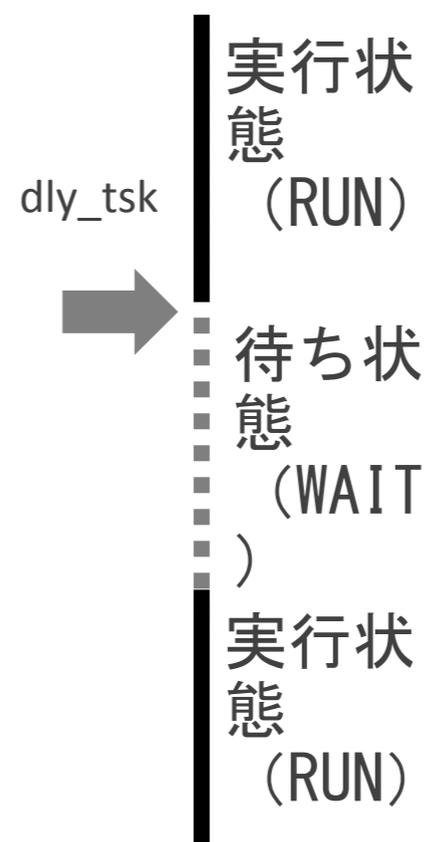
Task A
 優先度: 高
 (Priority = High)

Task B
 優先度: 低
 (Priority = Low)



例: Delay task

Task A



指定した時間が経過
 Specified time has
 past.

μ ITRON4.0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

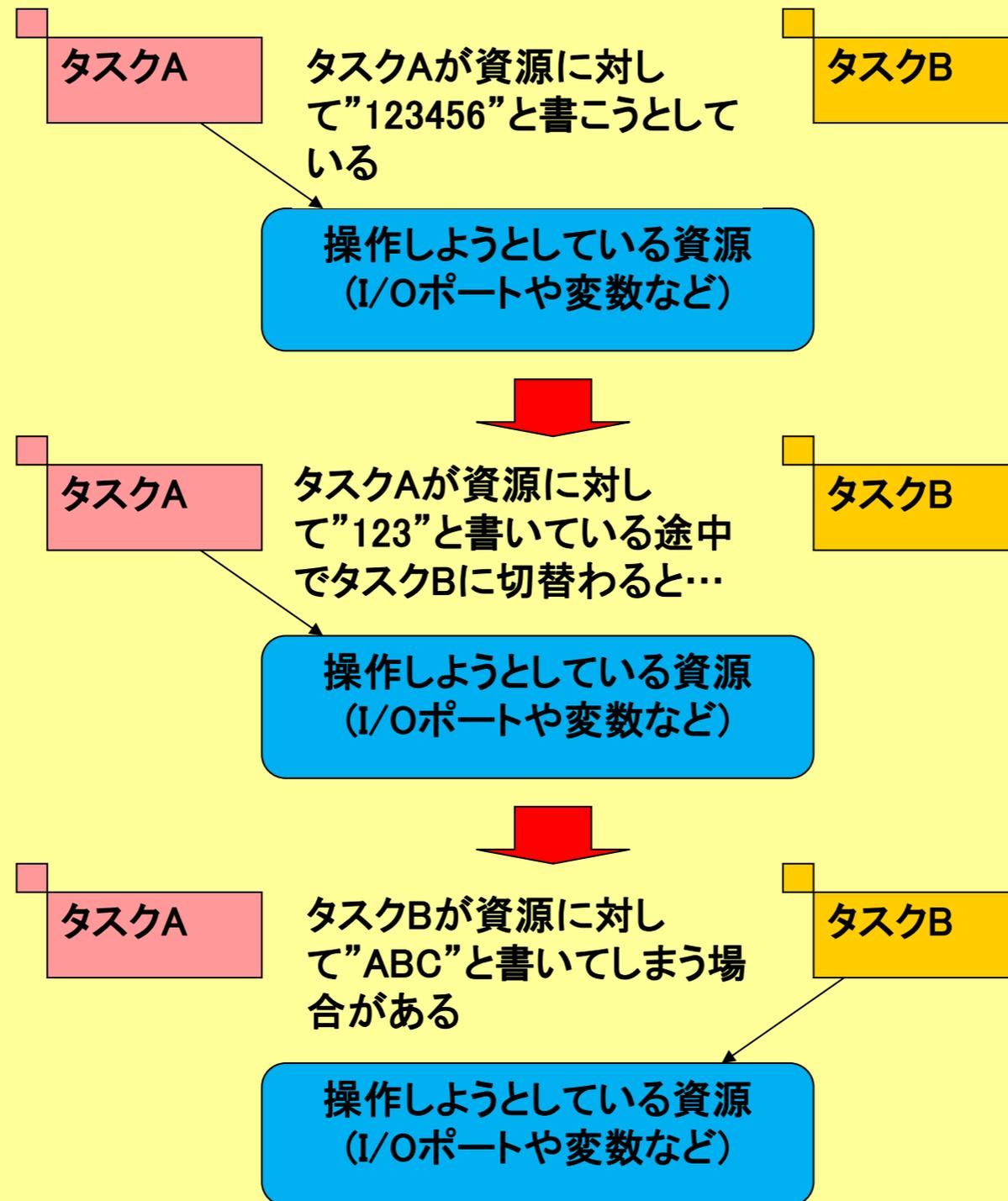
同期・通信機能

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、メールボックス機能、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。

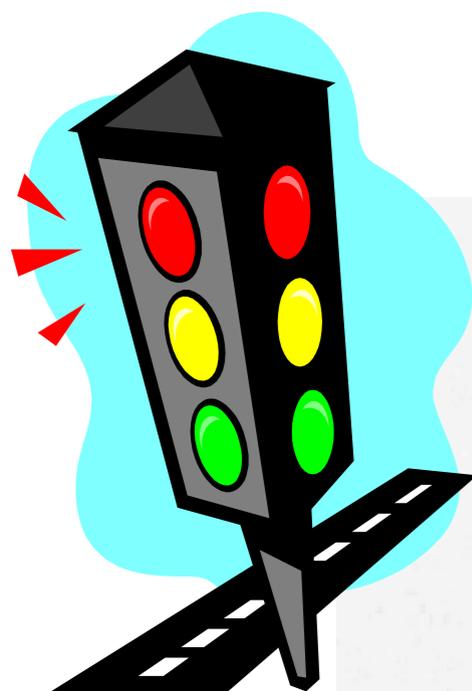
- ▶ セマフォ (Semaphore)
 - 利用可能な共有資源の数をカウンタで表し、共有資源の排他制御を行うオブジェクト
- ▶ メールボックス (Mail Box)
 - 共有メモリ上に置かれたメッセージを受け渡しして、同期通信を行うオブジェクト
- ▶ イベントフラグ (Event Flag)
 - イベントの有無をビット毎のフラグで表現することにより同期するオブジェクト
- ▶ データキュー (Data queues)
 - 1ワードのメッセージを受け渡しすることにより、同期と通信を行うためのオブジェクト

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。

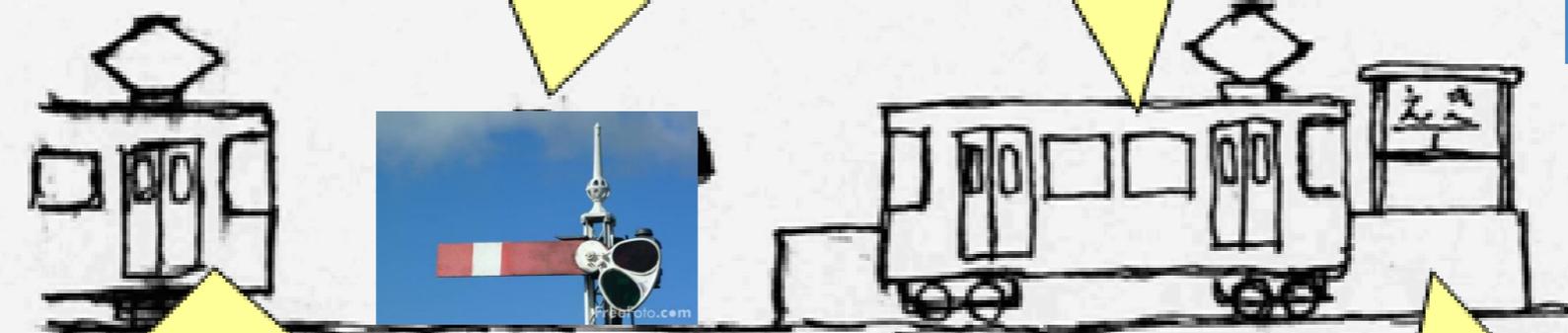
同期・通信機能（セマフォ）



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。

セマフォは信号機で、列車の入線を管理

先着列車は資源（ホーム）使用中



後続列車は資源が使われているので、セマフォのところで待たされる

資源を駅のホームと考えます



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。

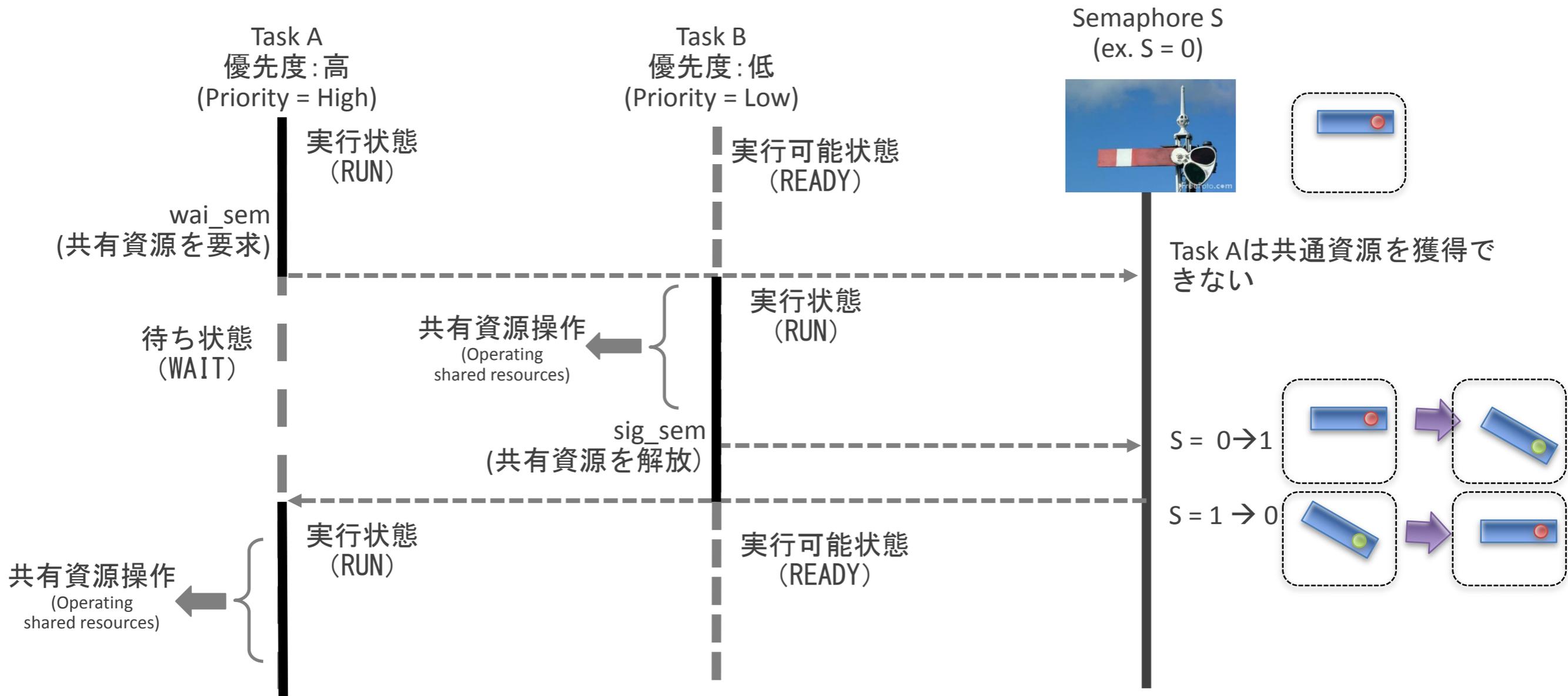
同期・通信機能（セマフォ）

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)
cre_sem()	セマフォ生成
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。

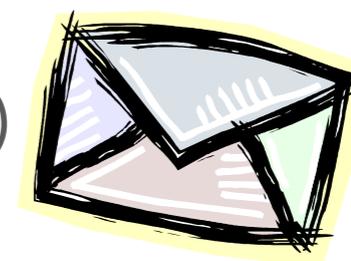
セマフォ (Semaphore) の実行例



セマフォを使った排他制御のサンプル

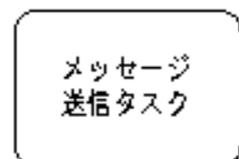
```
#include "kernel.h"
#include "kernel_id.h"
/*****/
/*   ポートP0へアクセスするタスク1                               */
/*****/
void task1(VP_INT exinf)
{
    ER   ercd;
    char num;
    while(1)
    {
        ercd = wai_sem(ID_SEM1); /* セマフォを獲得 */
        /* ここに排他制御が必要な処理を記述します */
        num = P0;
        P0 = 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);
    }
}
```

同期・通信機能（メールボックス）

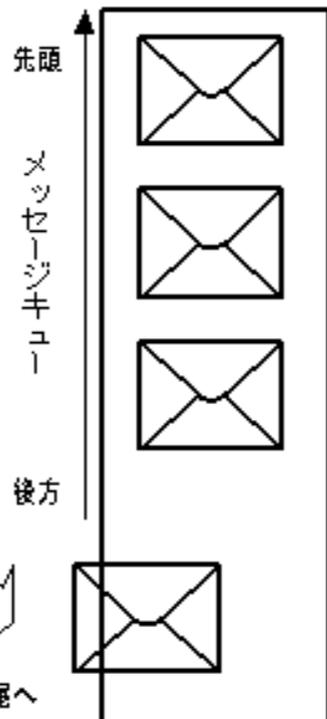


タスク間でやり取りするデータが格納されているアドレスをメッセージとして送る

メッセージの送信



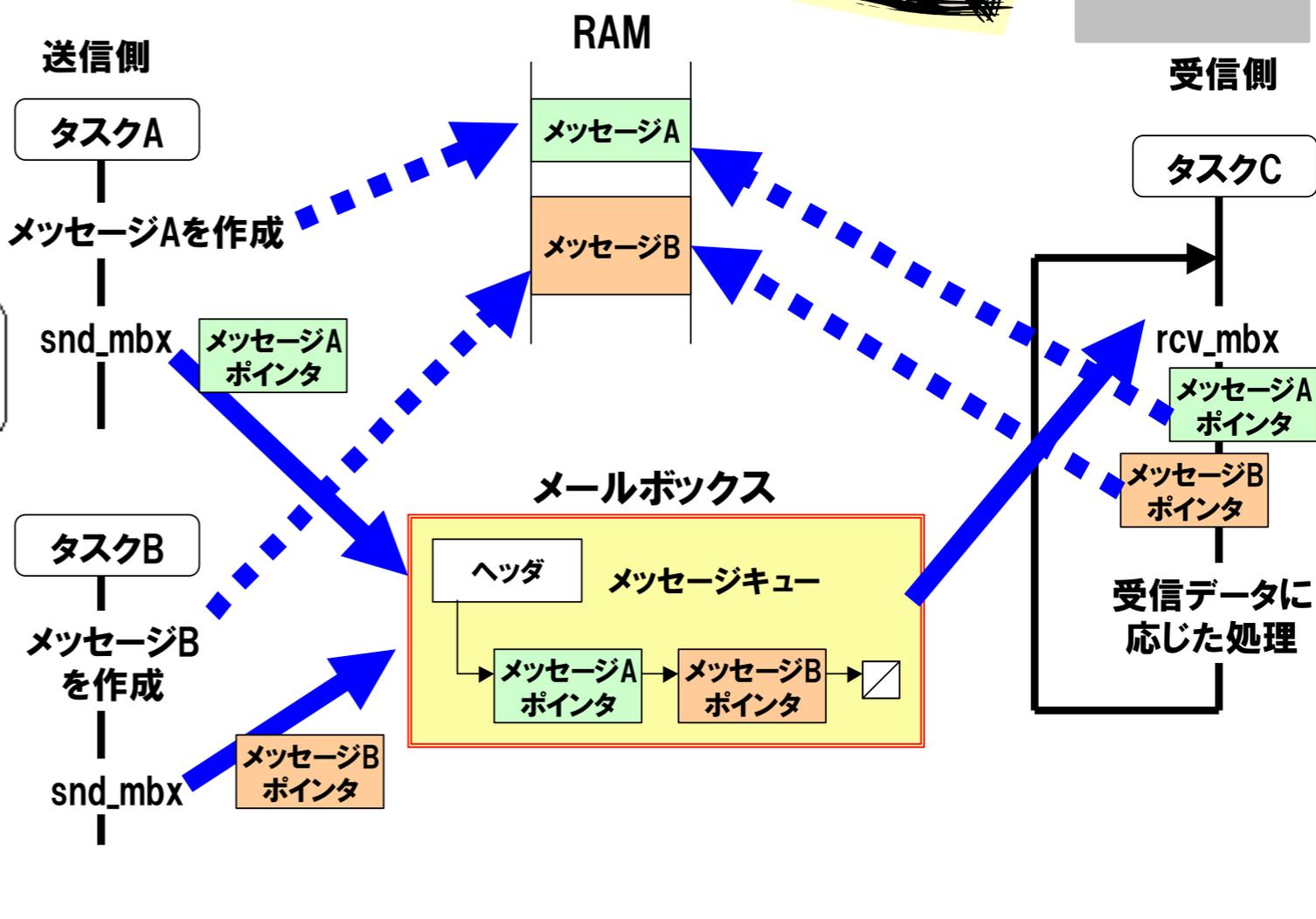
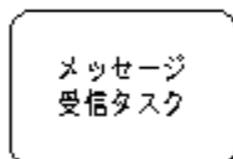
メールボックス



FIFO

先頭から

メッセージの受信



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)がメー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。

メールボックスには、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）

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)
cre_mbx()	メールボックス生成
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

補足：メールボックス(システムコール)

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、メー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。

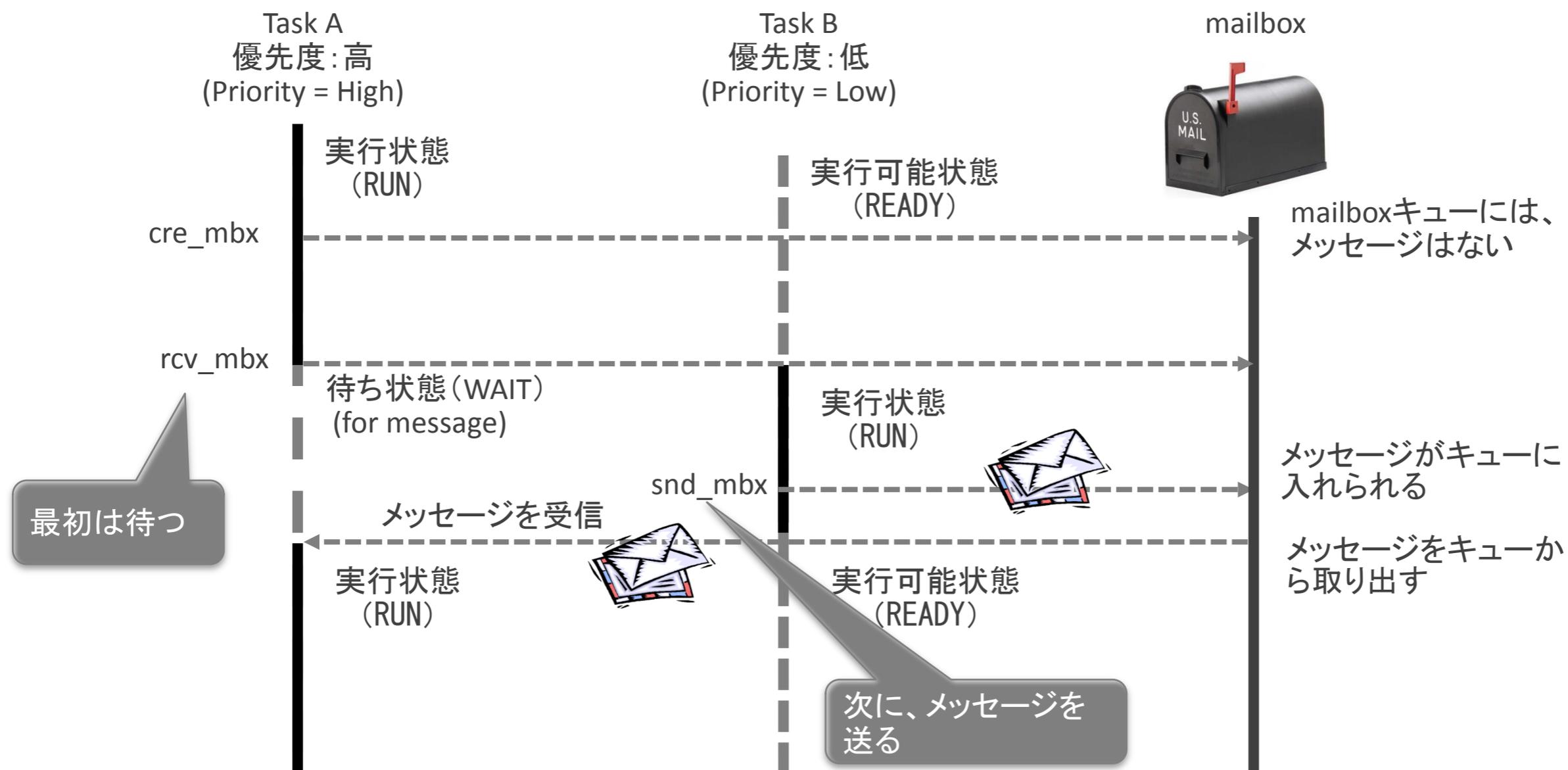
メールボックスに送られてきたメッセージはメールボックス内でFIFO順の待ち行列につながれます。これをメッセージキューと呼びます。

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。

メールボックス (Mailbox) の実行例



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 */

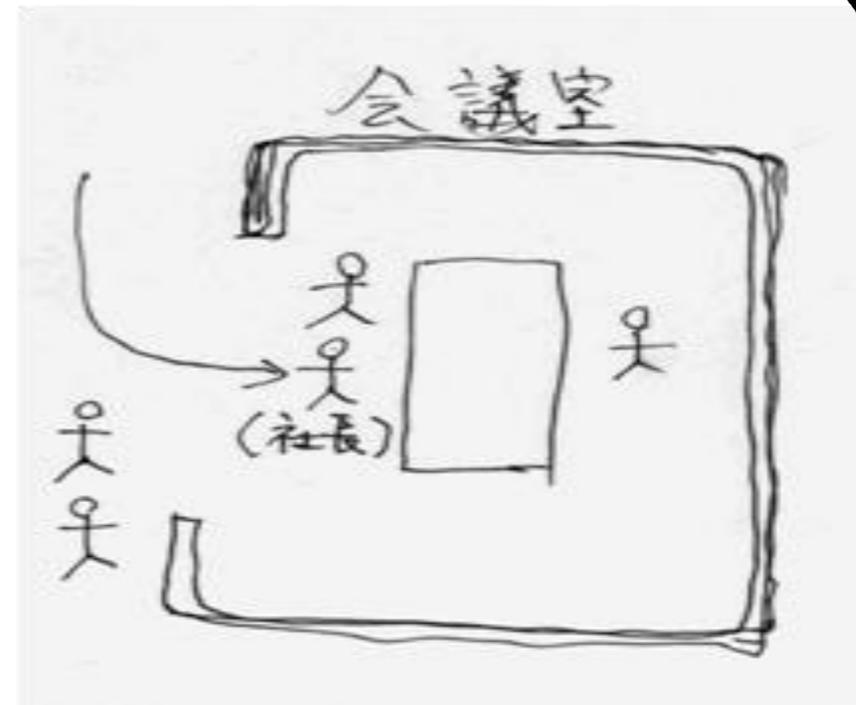
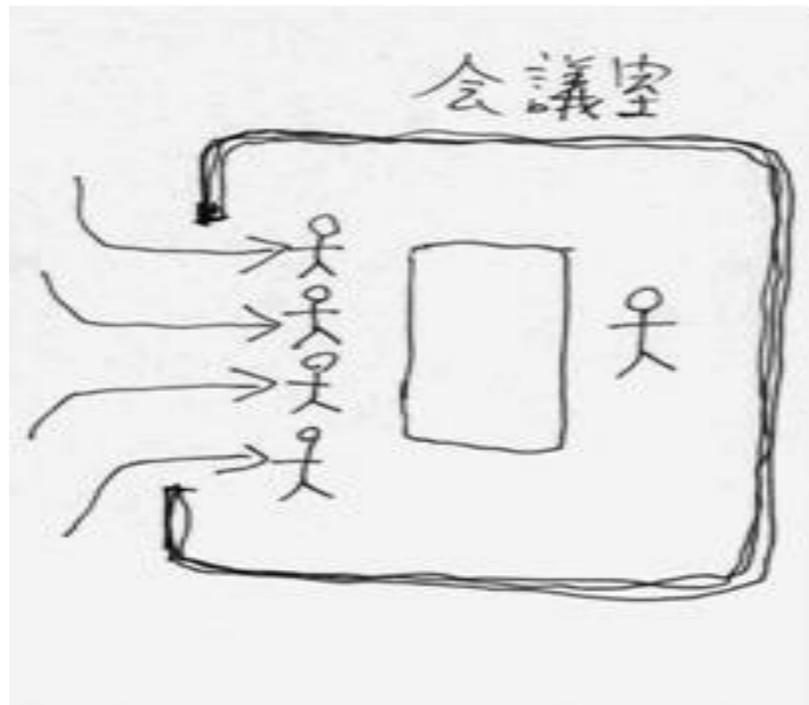
    while(1)
    {
        ercd = rcv_mbx(ID_MBX2, (T_MSG **)&pk_msg);    /* ID_MBX2から受信 */

        /* 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 */
    }
}

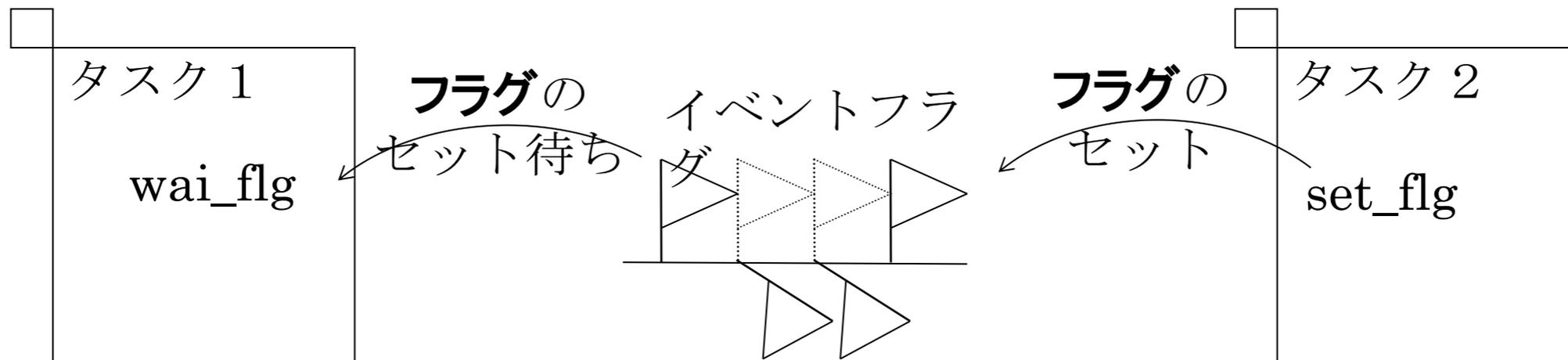
(C) 2014 T-Engine Forum, All Rights Reserved.
```

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を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。

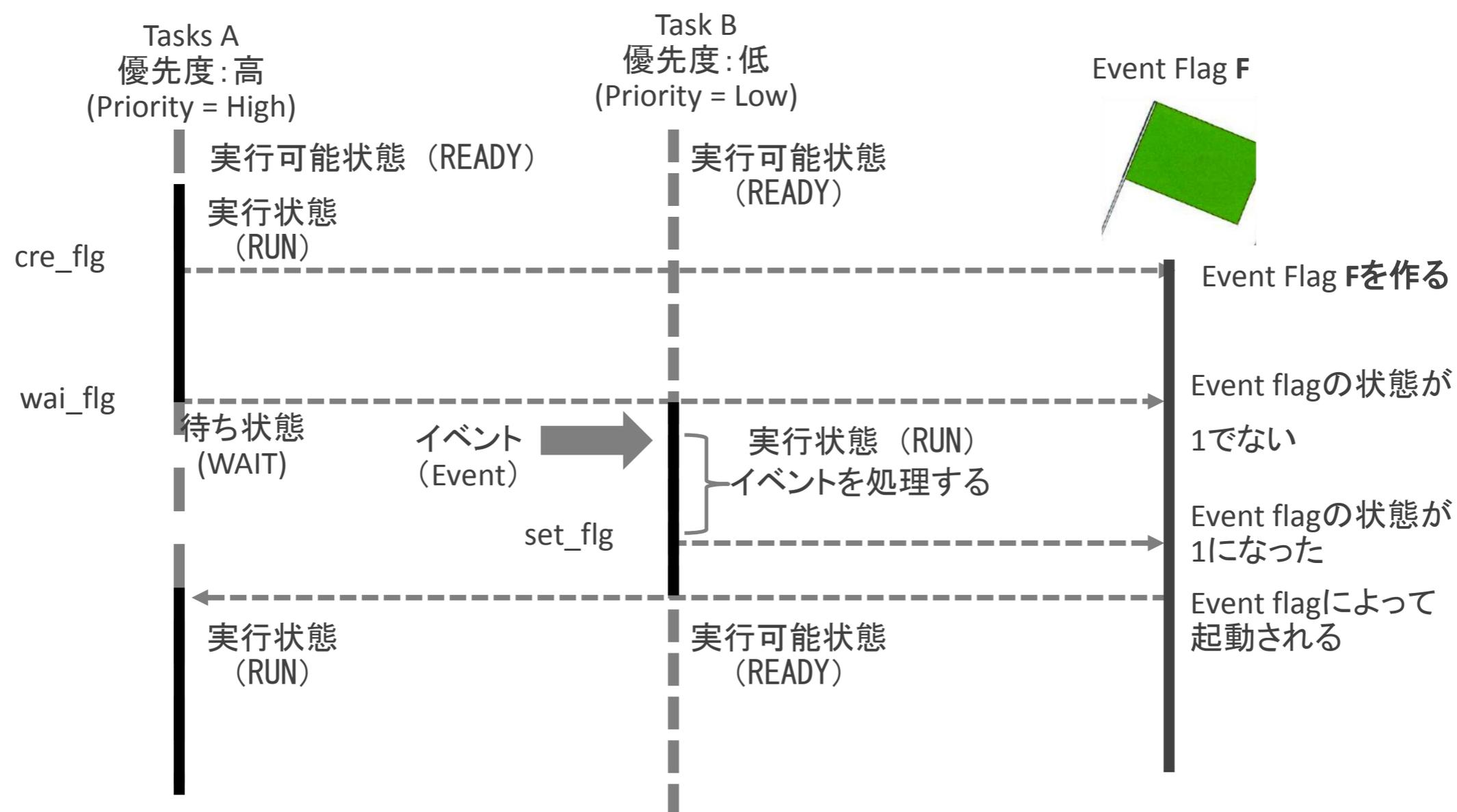


同期・通信機能（イベントフラグ）

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)
cre_flg()	イベントフラグ生成
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



```
#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 */

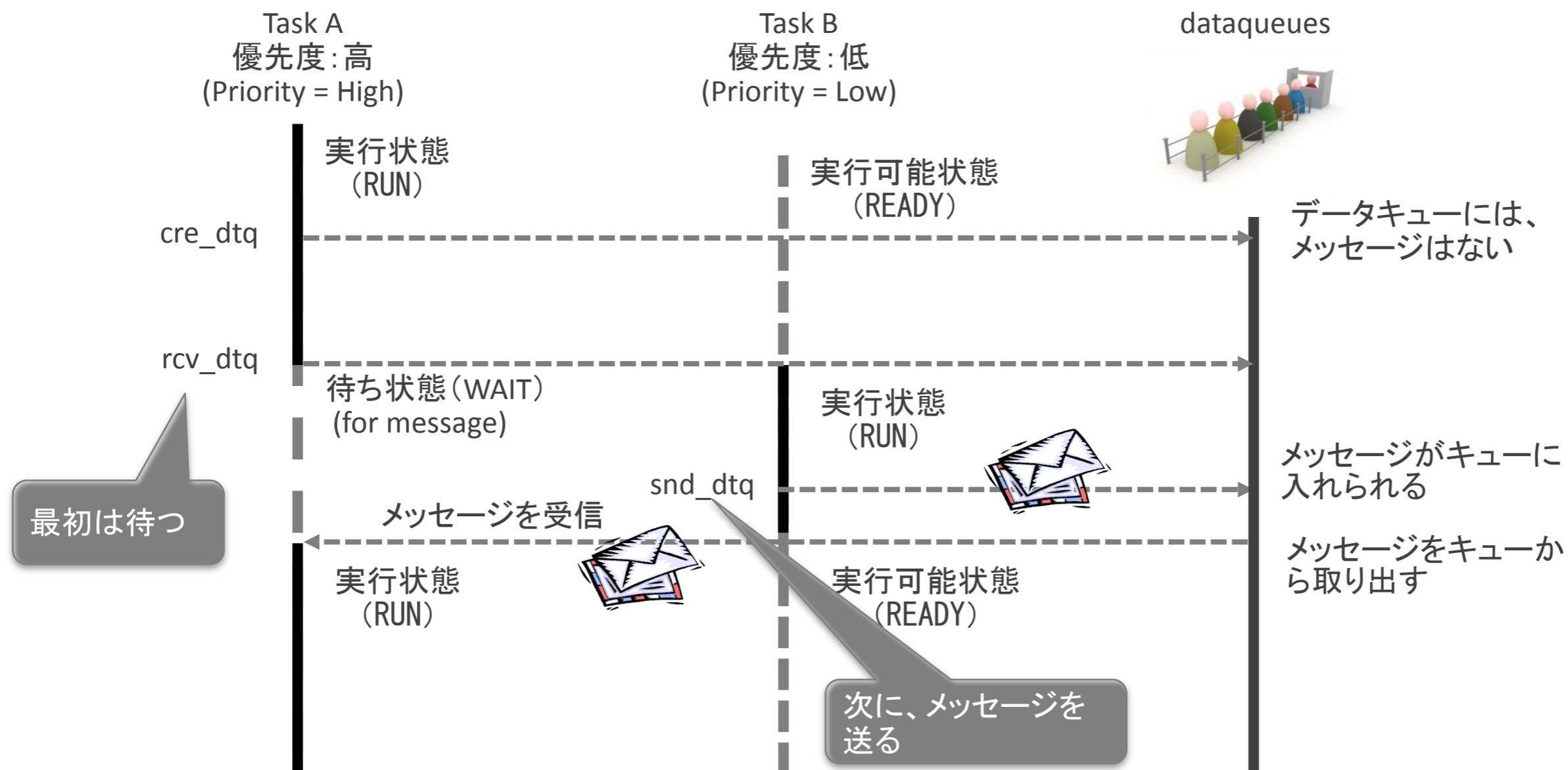
    ercd = set_flg (ID_FLG1, (FLGPTN) 1);
}
```

同期・通信機能（データキュー）

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)
cre_dtq()	データキュー生成
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。

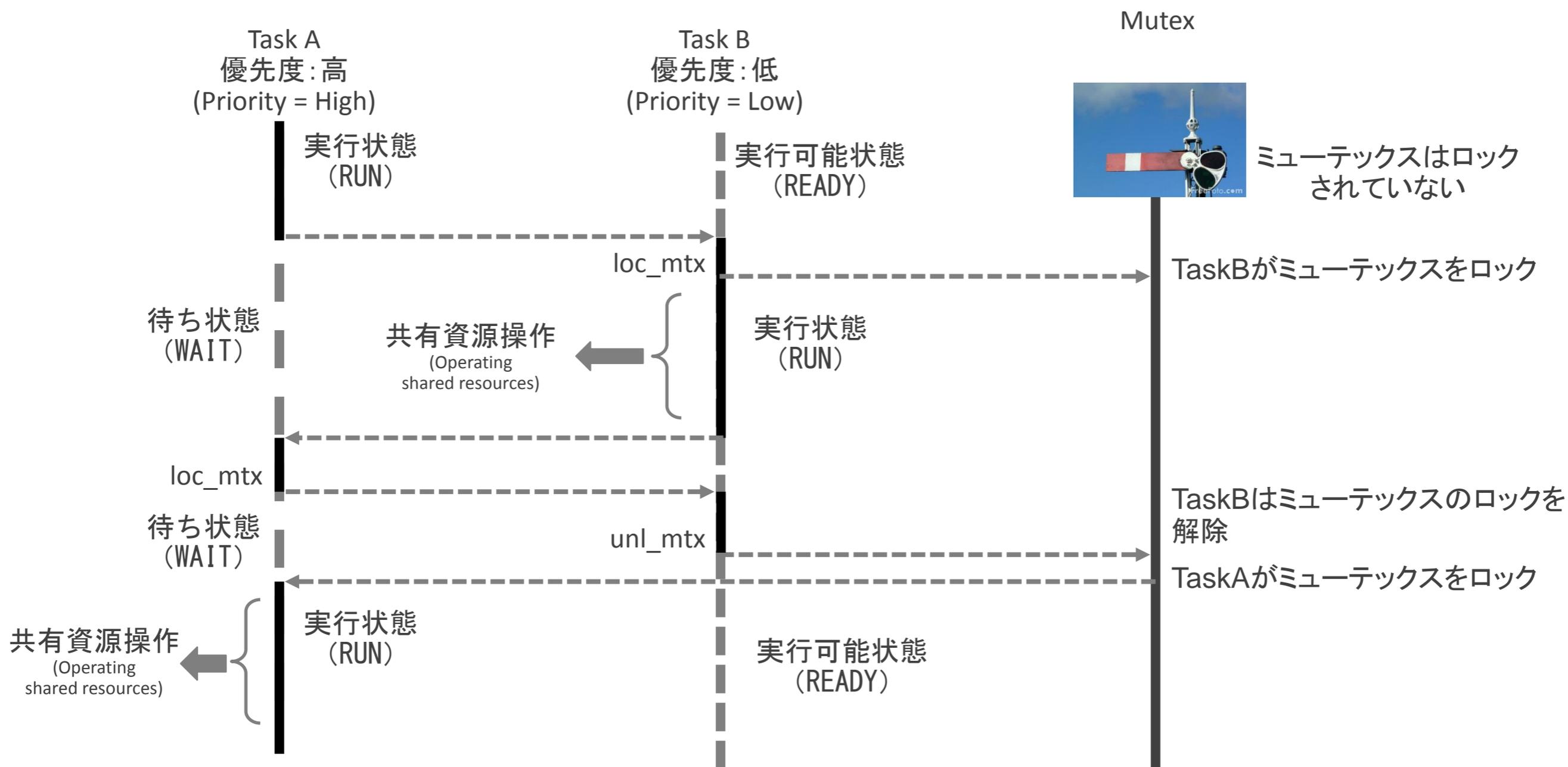
拡張同期・通信機能（ミューテックス）

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。

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

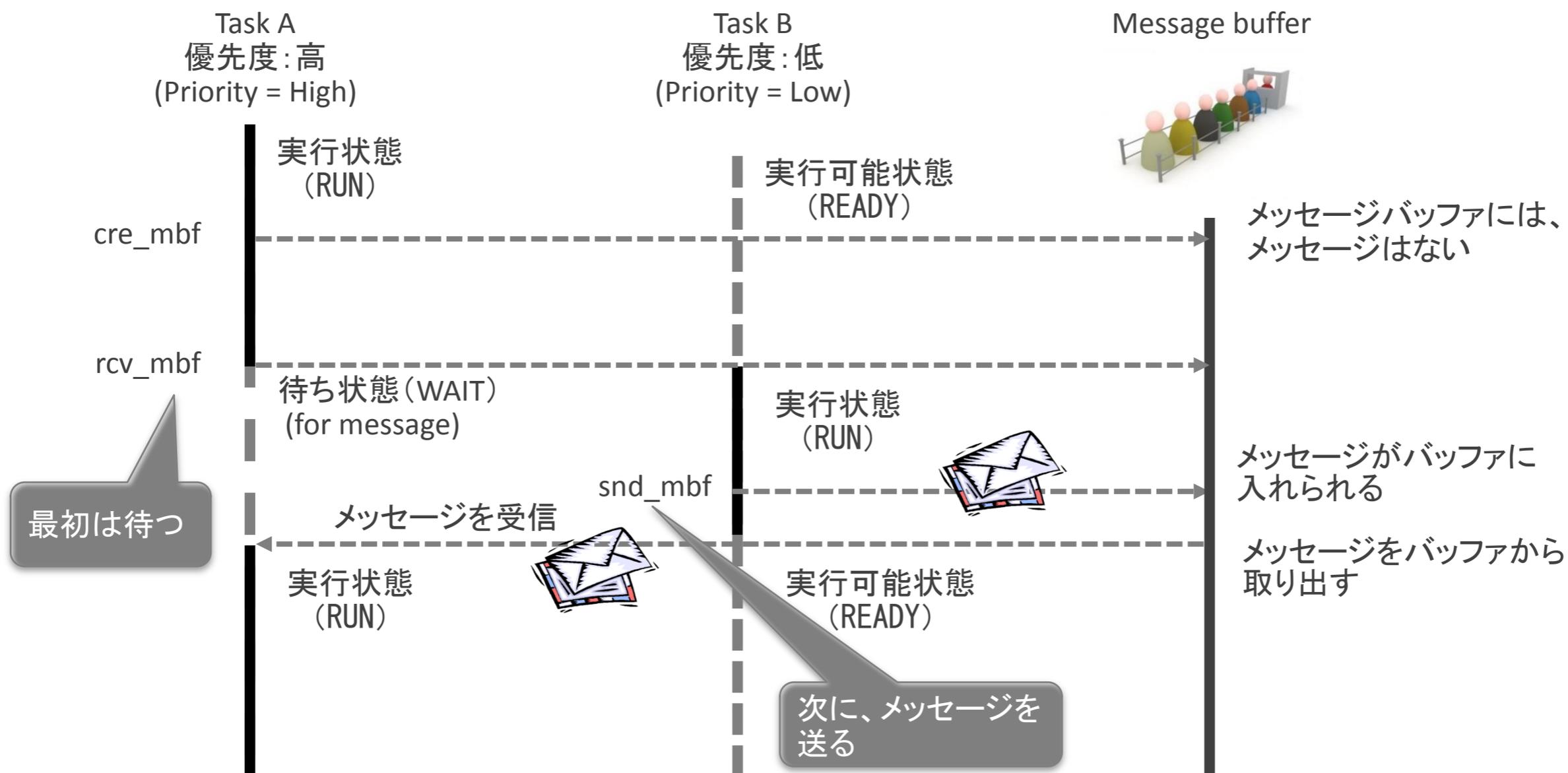


拡張同期・通信機能（メッセージバッファ）

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)
cre_mbf()	メッセージバッファ生成
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

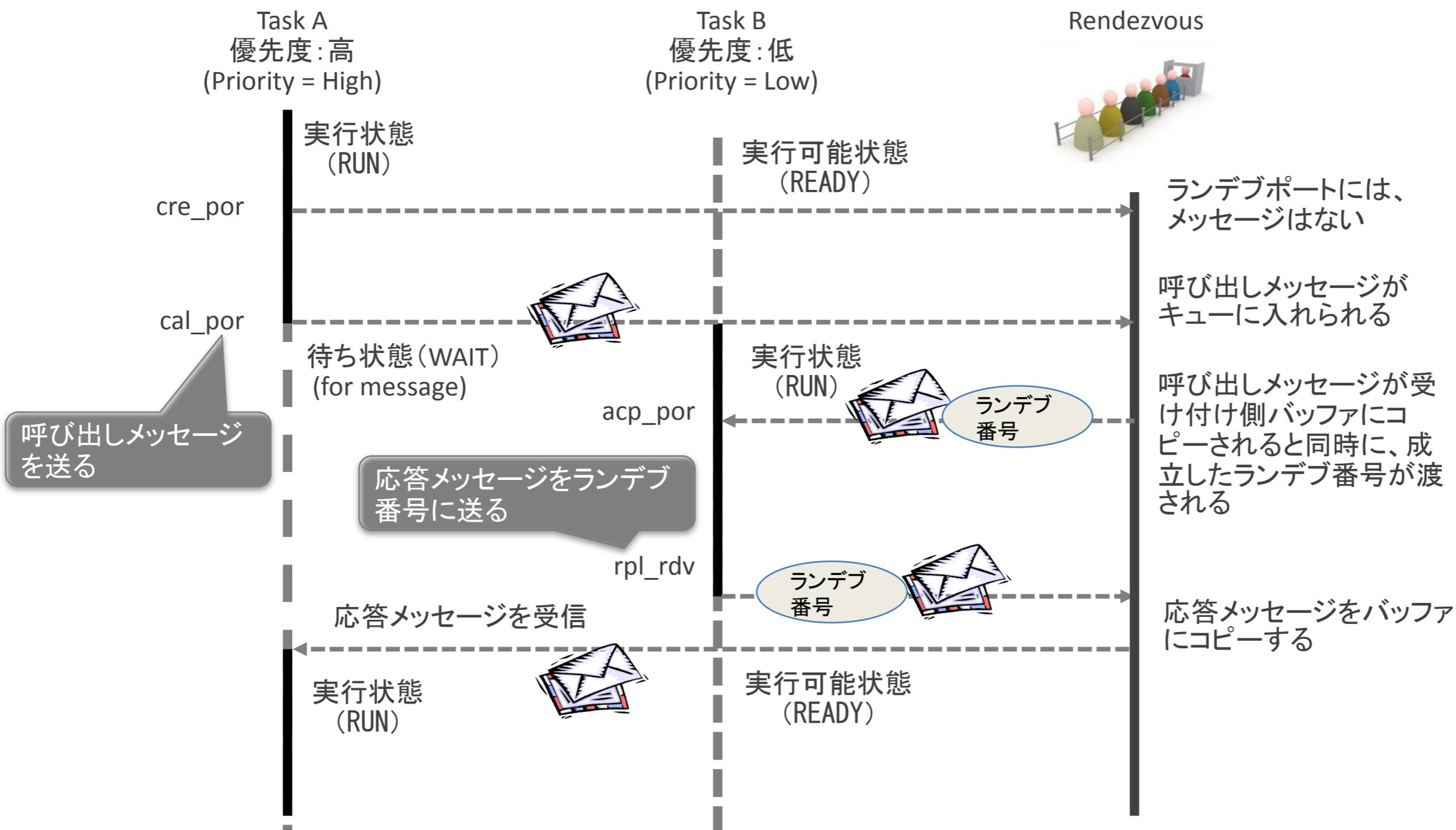


拡張同期・通信機能（ランデブ）

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)
cre_por()	ランデブポート生成
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

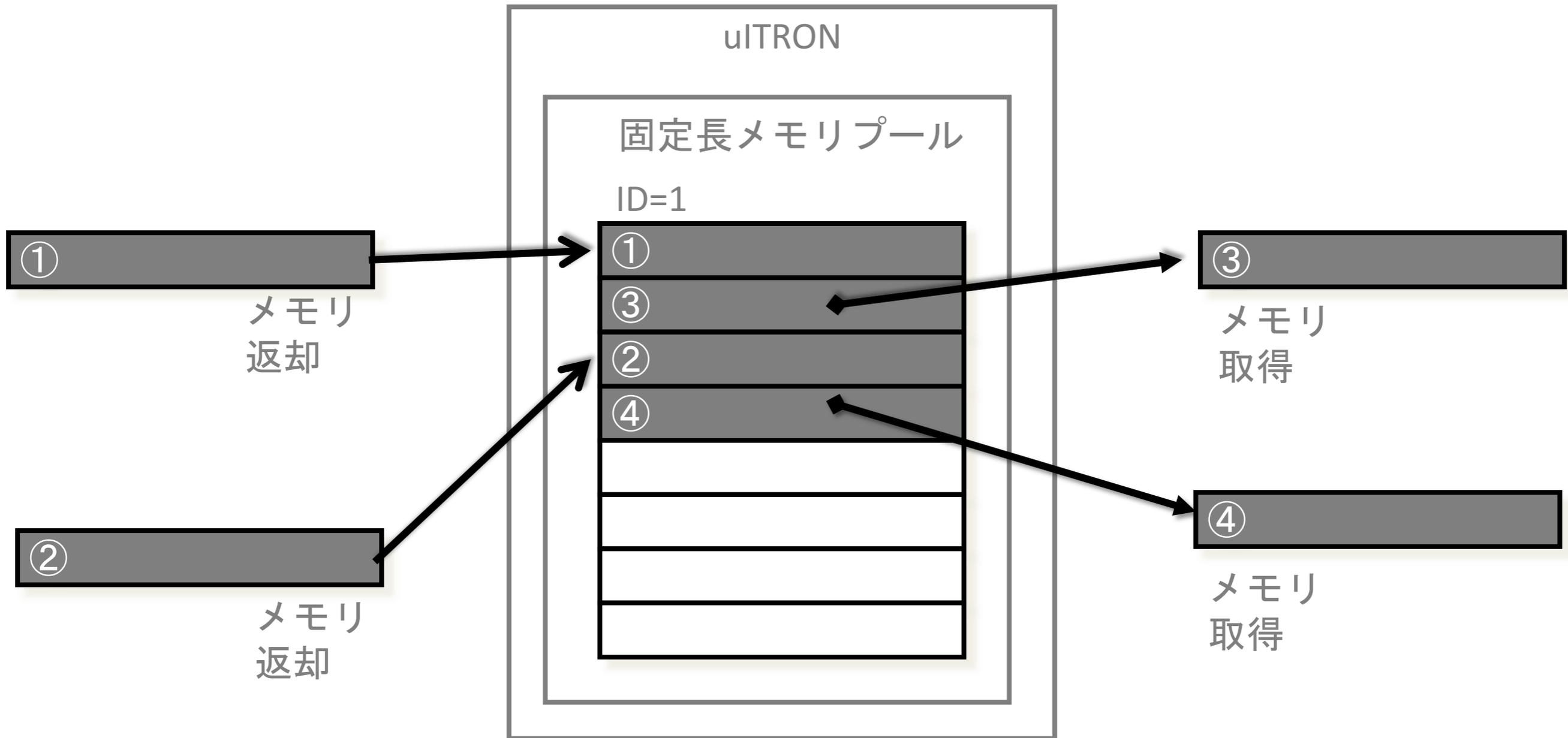
- 固定長メモリプール
- 可変長メモリプール
- ▶ 割り込み管理機能
- ▶ 時間管理機能
 - システム時刻管理
 - 周期ハンドラ
 - アラームハンドラ
 - オーバランハンドラ
- ▶ システム状態管理機能
- ▶ サービスコール管理機能
- ▶ システム構成管理機能

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



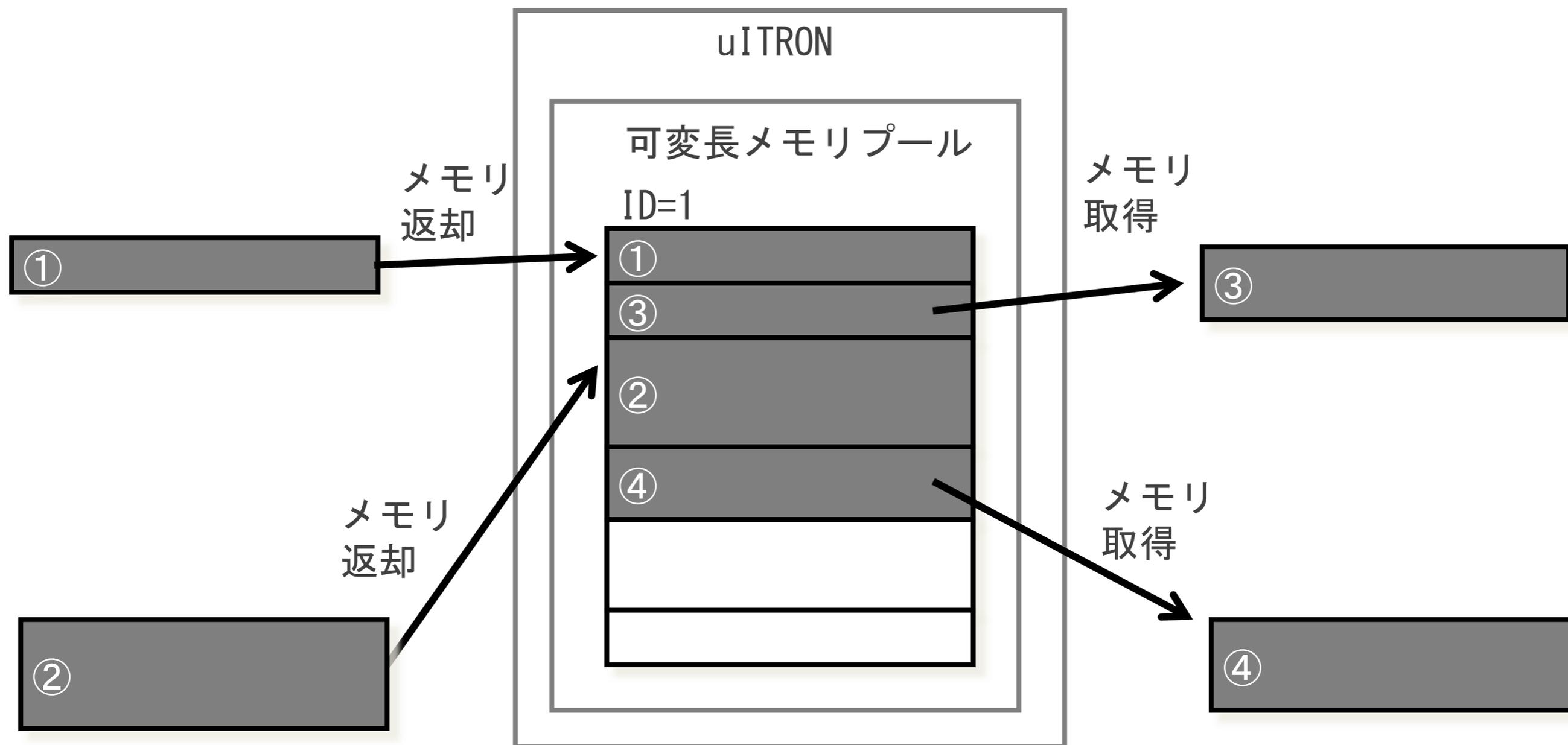
取得時のメモリブロックサイズが固定サイズ
可変長メモリプールよりは柔軟性に欠けますが、
分断の心配はなく、かつ高速です。

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。

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

割り込み管理機能

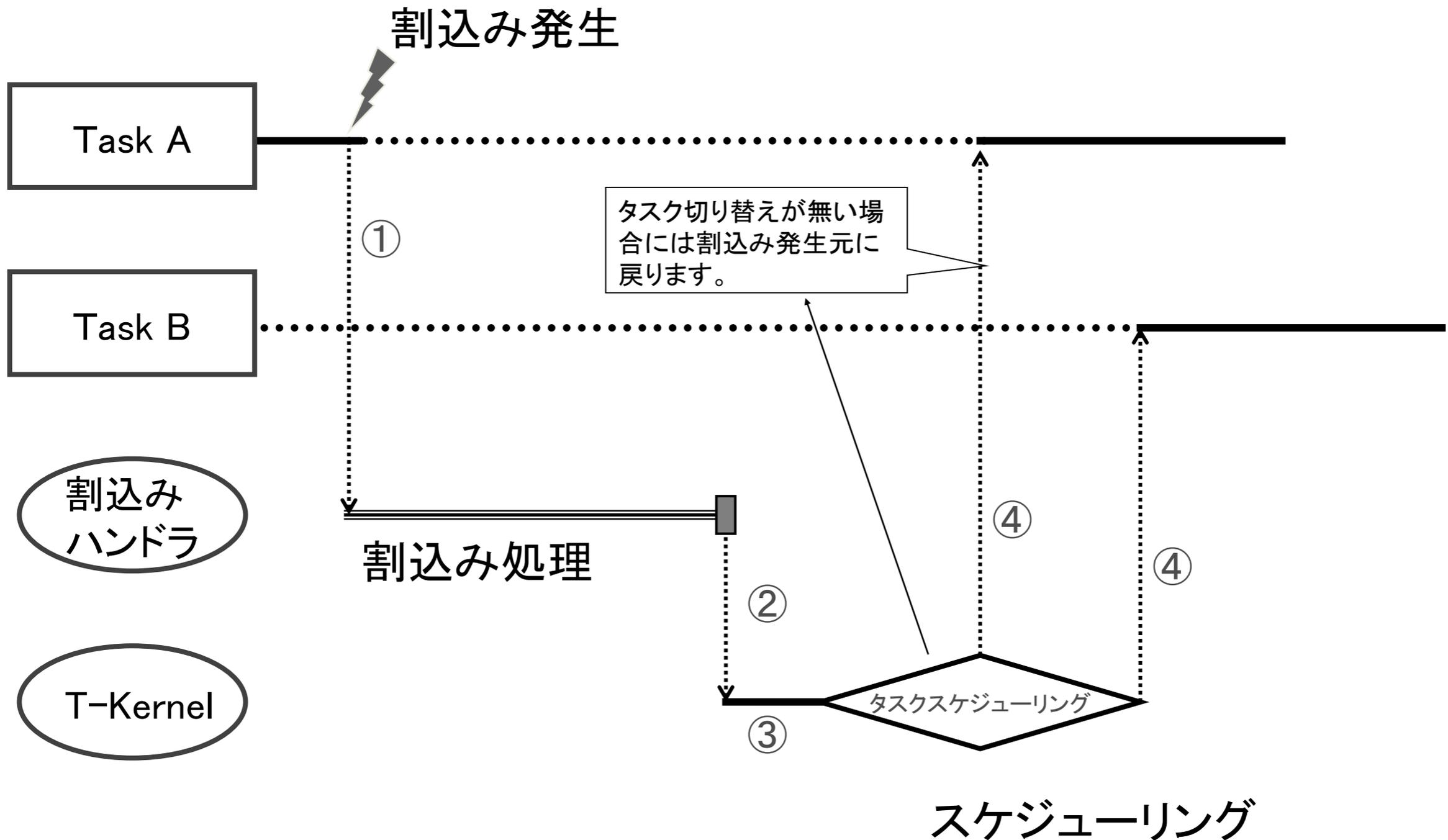
外部割り込みによって起動される割り込みハンドラおよび割り込みサービスルーチンを管理するための機能です。

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

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。

割り込み処理動作(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できます。

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



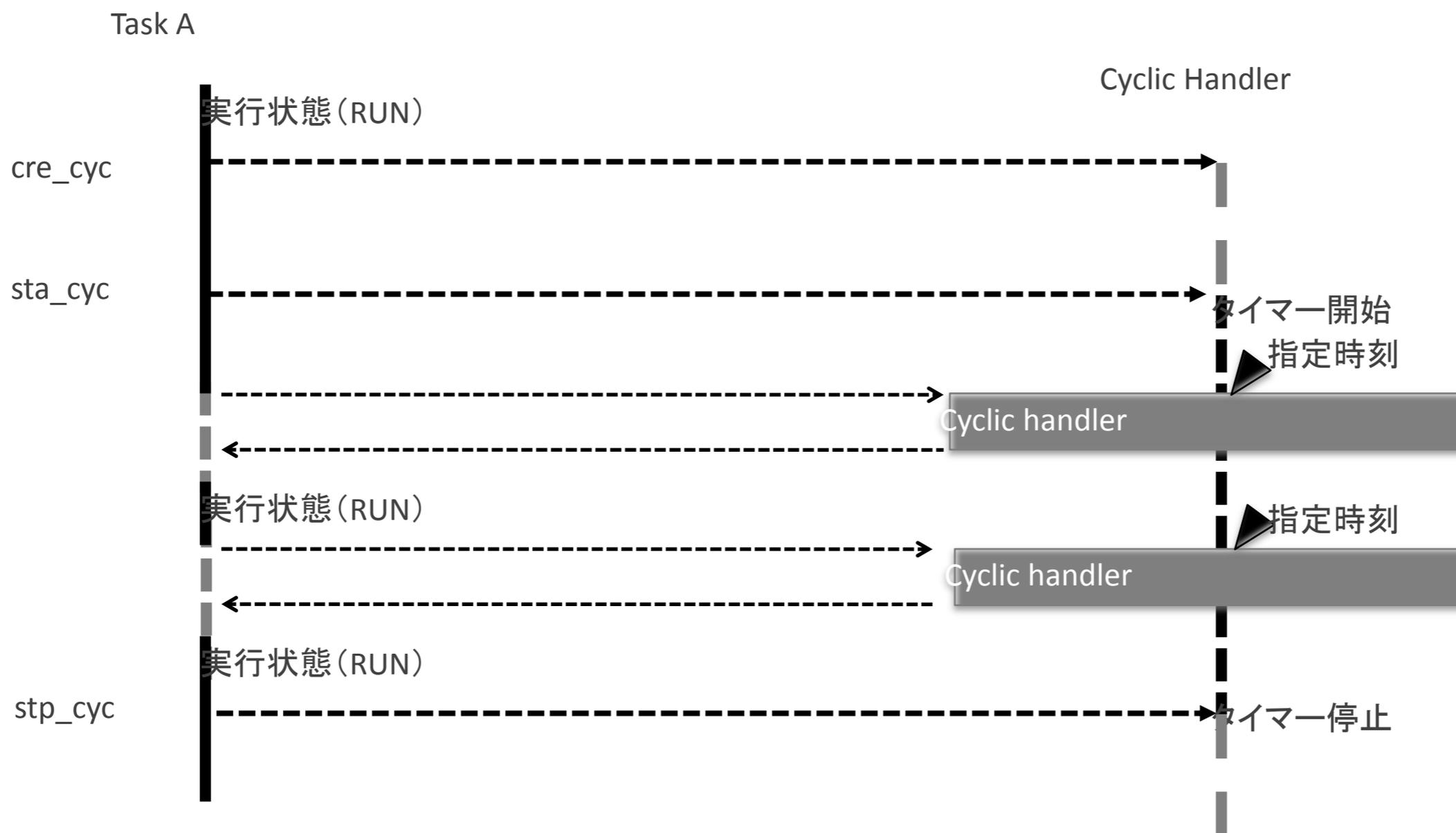
時間管理機能(周期ハンドラ)

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

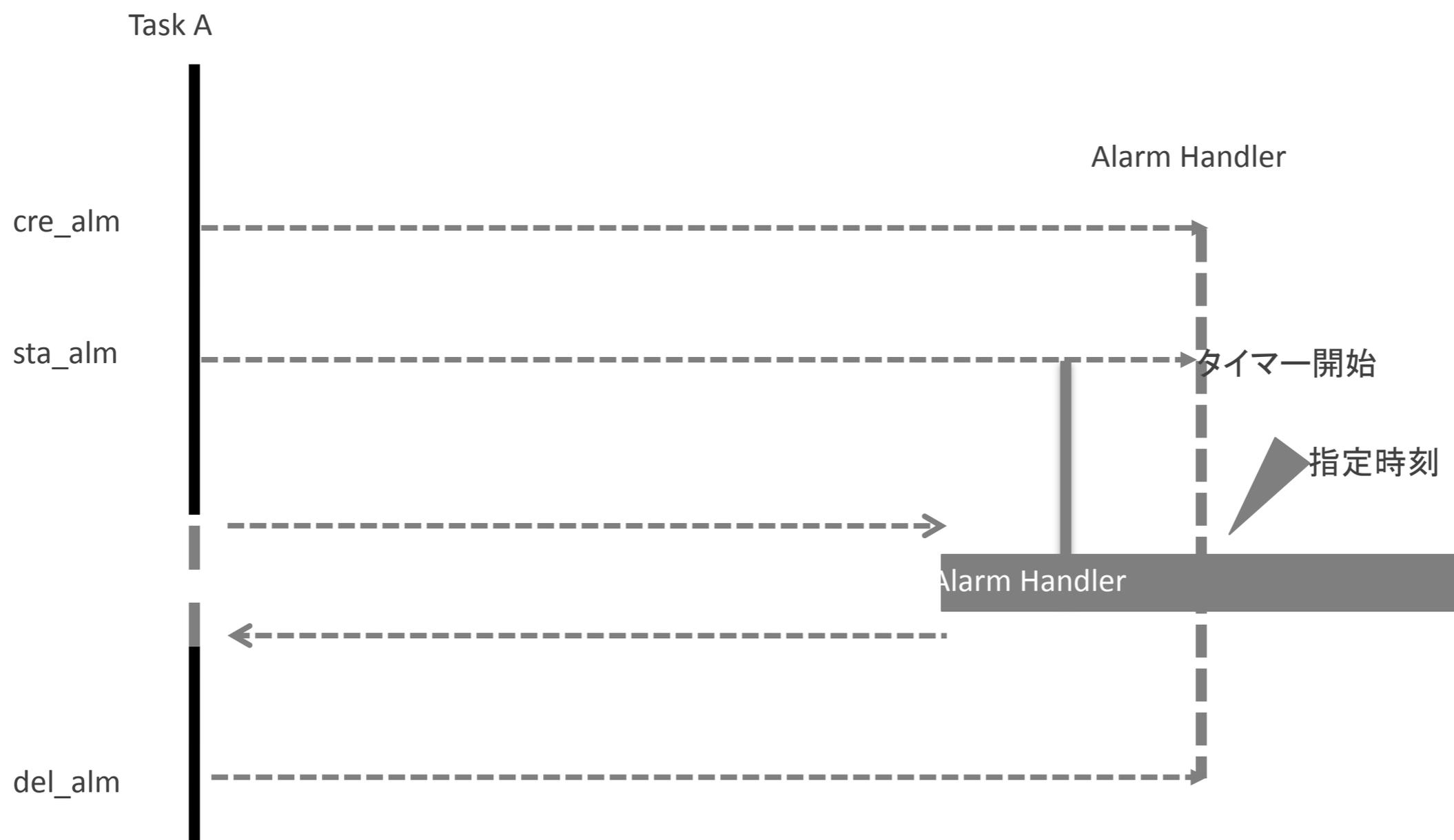
時間管理機能(アラームハンドラ)

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_alm()	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



- 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 μ ITRON入門

4 μ ITRON開発手順

- 5 μ ITRONプログラミング
- 6 参考資料・付録など

コンフィギュレーション

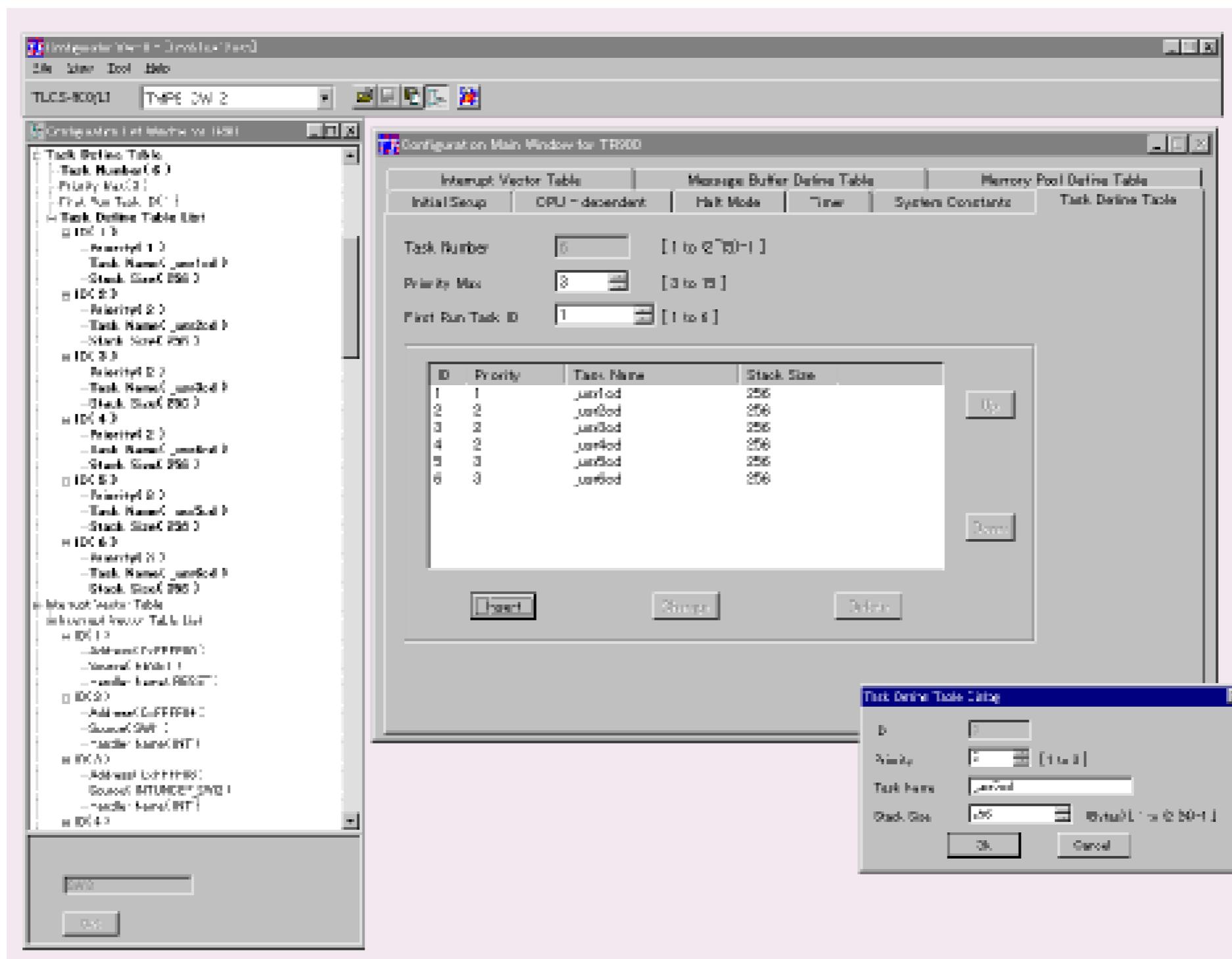
- ▶ RTOSの動作環境をコンフィギュレーションファイルに記述
- ▶ 静的APIによってオブジェクトを生成
 - ID番号の指定
 - 最大オブジェクト数の指定など
- ▶ 割込みハンドラを指定
- ▶ コンフィギュレータを使用してヘッダファイルとカーネル情報ファイルの出力

- ・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

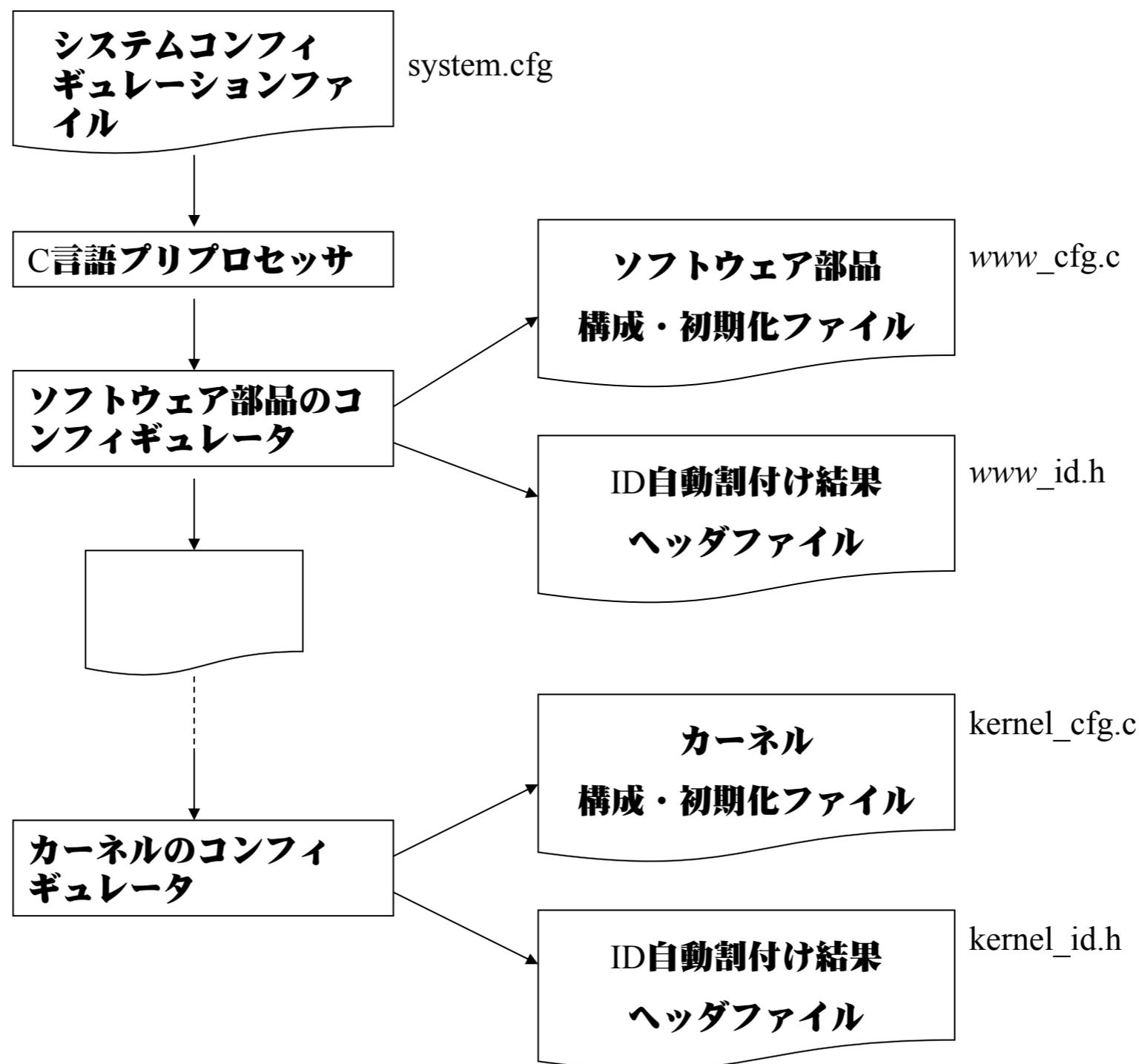
コンフィギュレーションファイルの例

```
/*  
*****  
/* Sample configuration file for uITRON 4.0  */  
*****  
  
INCLUDE ("demo.h");  
  
#define STACK_SIZE 2048  
CRE_TSK (TASK1, {TA_HLNG|TA_ACT, TASK1, task1, 15, STACK_SIZE, NULL});  
CRE_TSK (TASK2, {TA_HLNG|TA_ACT, TASK2, task2, 15, STACK_SIZE, NULL});  
  
CRE_FLG (ID_FLG1, {TA_TFIFO|TA_CLR|TA_WMUL, 0});  
CRE_SEM (SEM1, {TA_TFIFO, 0, 10});  
CRE_DTQ (DTQ1, {TA_TPRI, 10, NULL});  
  
CRE_CYC (ID_CYC1, {TA_HLNG|TA_STA, ID_CYC1, cyc_func, 4000, 2000});
```

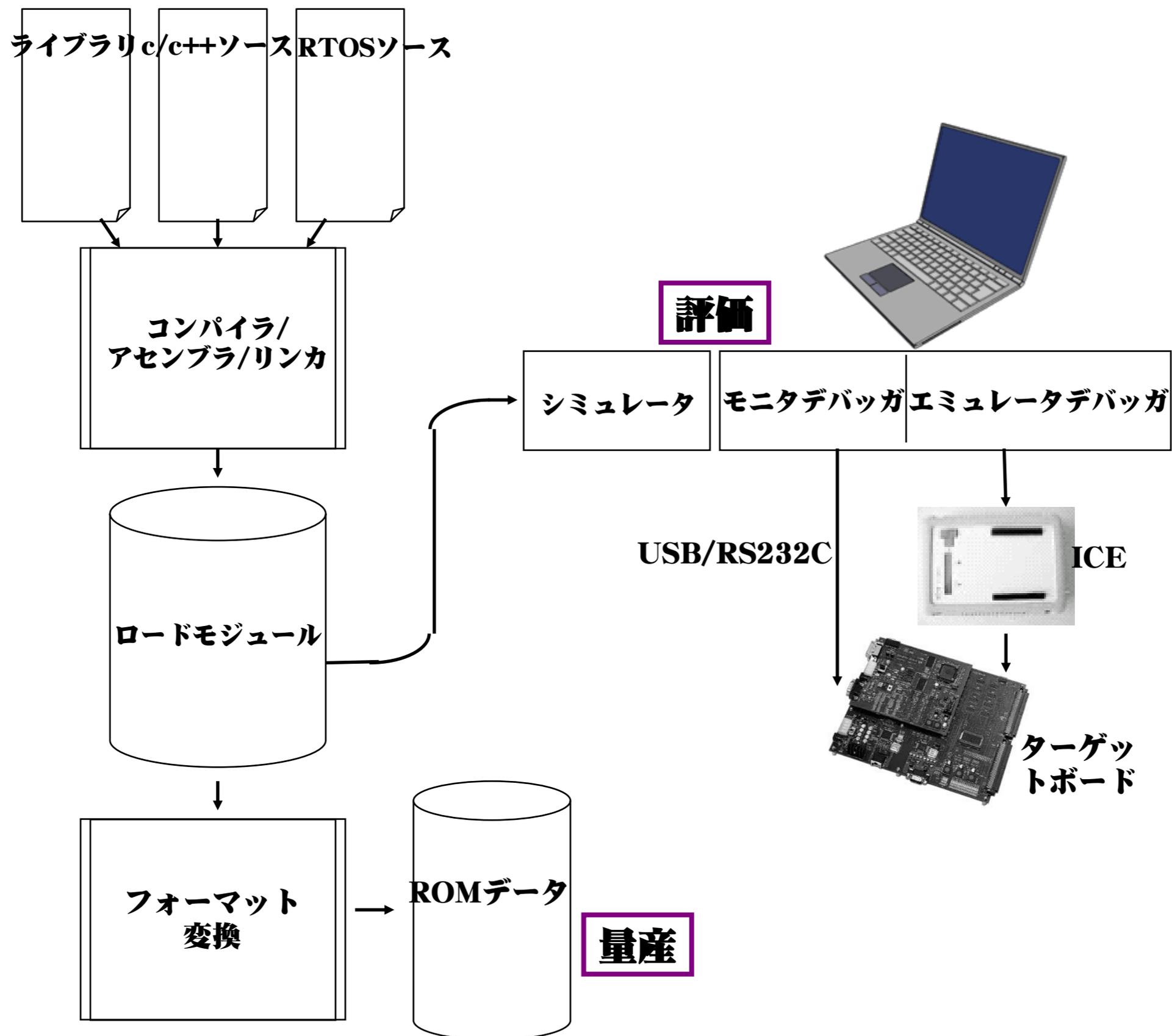
コンフィギュレーションファイルの例(GUI)



コンフィギュレーションの流れ



開発環境の例



- 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 μ ITRON入門
- 4 μ ITRON開発手順

5 μ ITRONプログラミング

- 6 参考資料・付録など

一般的なC言語プログラミングと違う点(1)

- ▶ μ ITRON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
 - 割り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

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

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

アプリケーションプログラムの開発手順は大別して次の7段階で構成されます。

- (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) リロケータブルオブジェクトファイルの作成
- (7) RTOSライブラリとアプリケーションとのリンク

以上により作成した実行オブジェクトファイルをデバッグすることにより、アプリケーションプログラムの動作検証を行います。

上記の各開発手順における概要の説明を以下に示します。

(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で設計します。

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。

(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) リロケータブル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をコンパイル/アセンブルして、リロケータブルオブジェクトファイルを作成します。アプリケーションプログラムファイル一つに対して、一つのリロケータブルオブジェクトファイルが作成されます。システム環境定義ファイルも一つのリロケータブルオブジェクトファイルとして作成されます。

(7) RTOSのライブラリとアプリケーションのリンク

リンケージエディタを使用して、RTOSのライブラリと、(6)で作成したリロケータブルオブジェクトファイルをリンクして、一つの実行オブジェクトファイルに変換します。

- 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 μ ITRON入門
- 4 μ ITRON開発手順
- 5 μ ITRONプログラミング

6 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が出来ます。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。

CC
SOME RIGHTS RESERVED

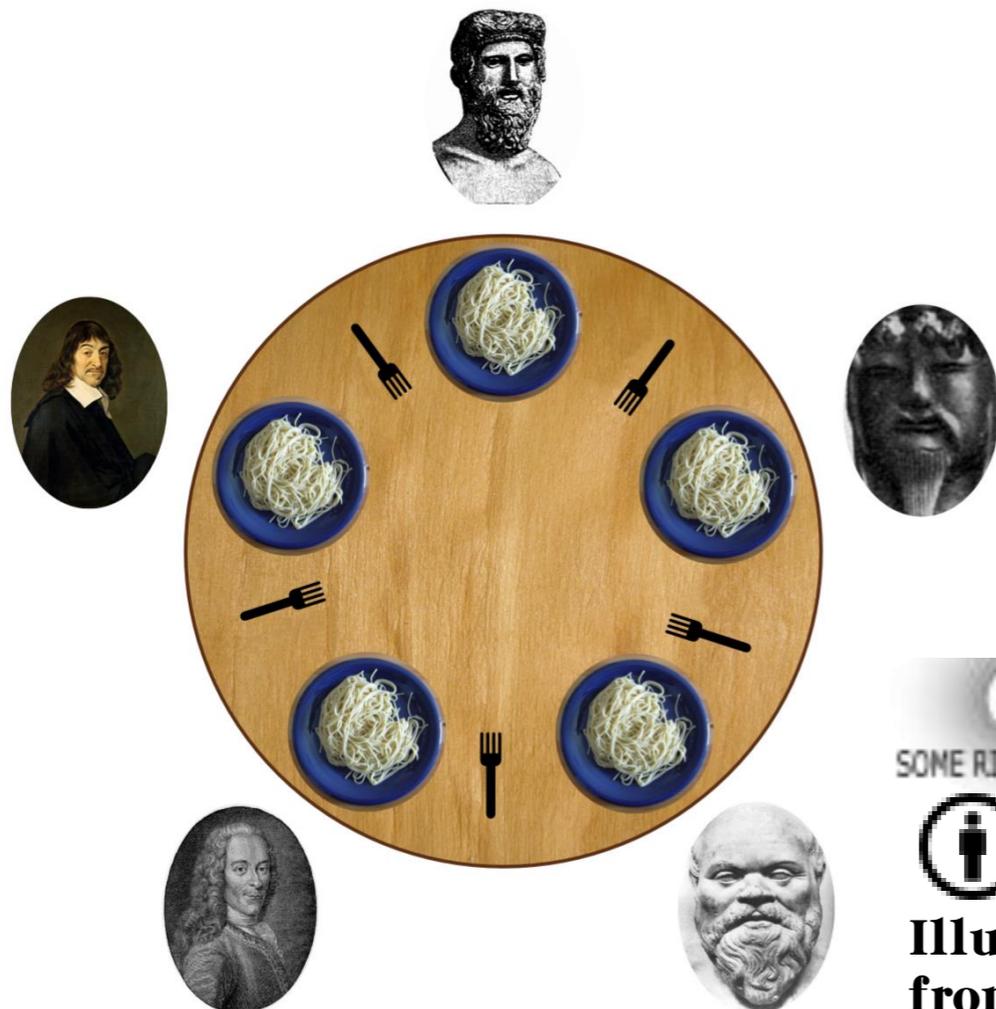


<http://creativecommons.org/licenses/by-sa/3.0/deed.ja>

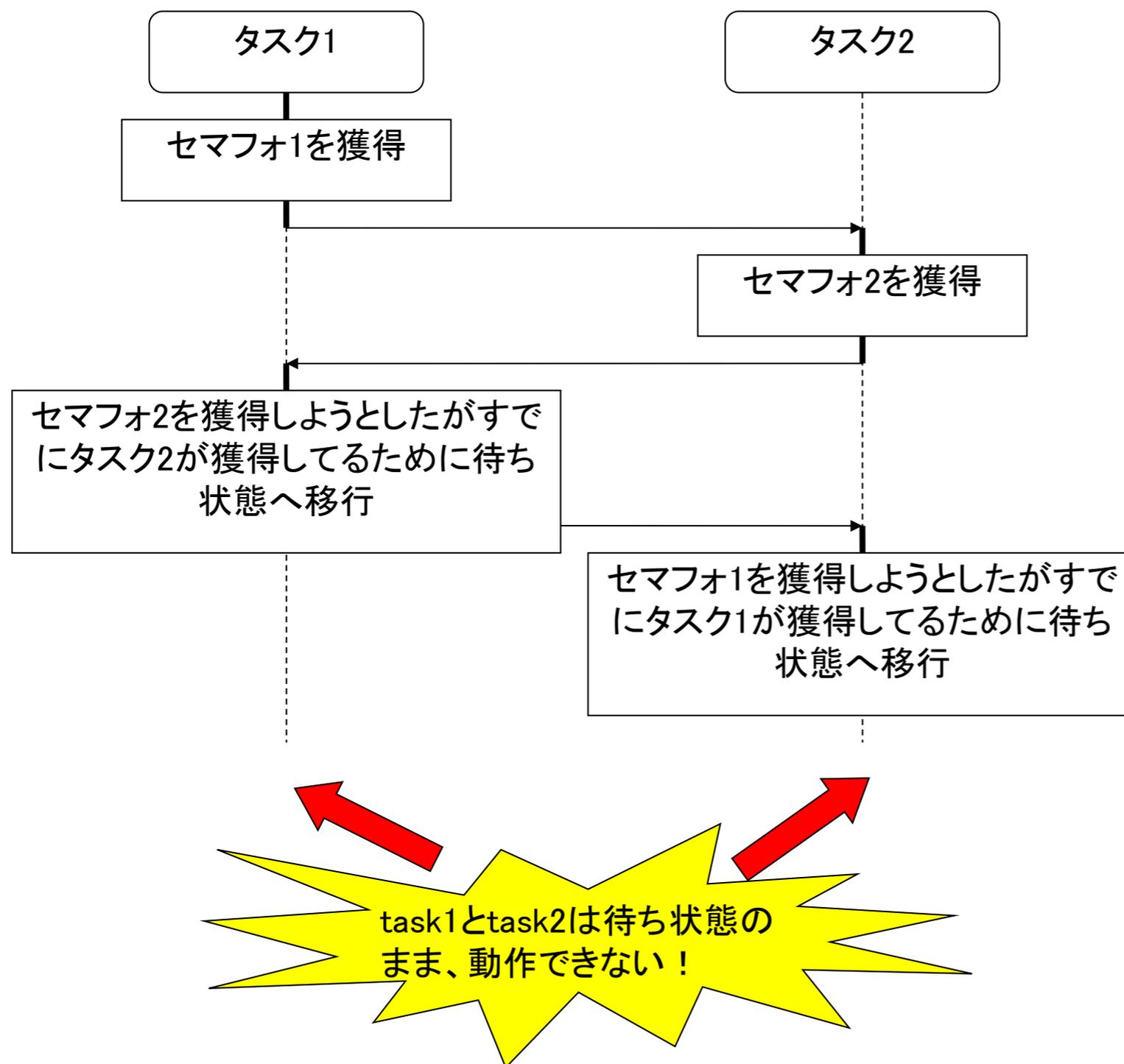
Illustrated by Benjamin D. Esham.

from http://en.wikipedia.org/wiki/File:Dining_philosophers.png

(C) 2014 T-Engine Forum, All Rights Reserved.



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がありますが、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。

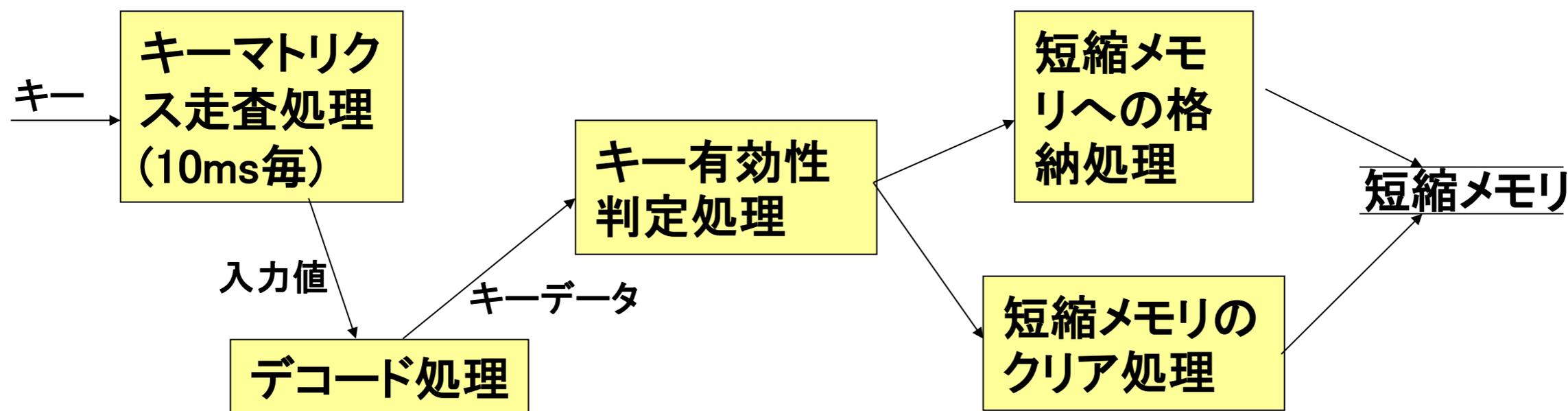
デッドロックを回避するためには、獲得したカーネルオブジェクトとは逆の順序で開放するというルールを守ることが重要です。

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、デバッグ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(タスク分割)

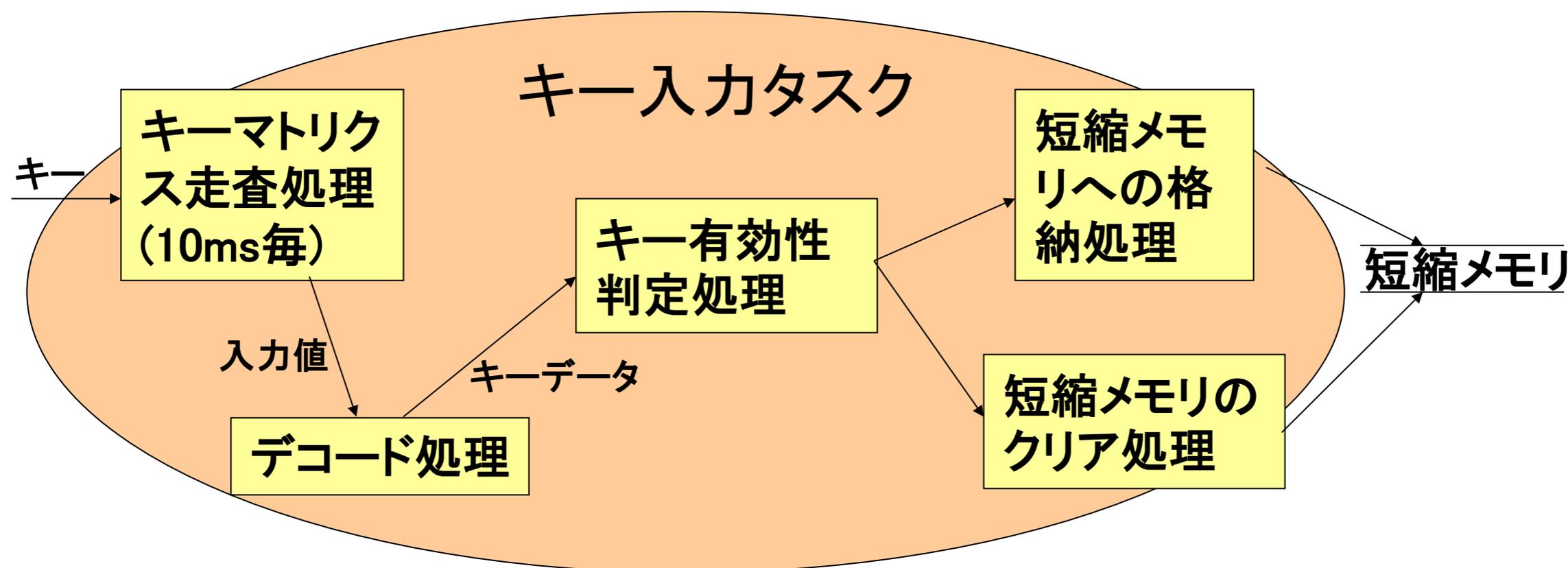
リアルタイム・システムを構築する際、タスク分割が必要です。ここではその一例と注意すべき点を記載します。

- ▶ 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。



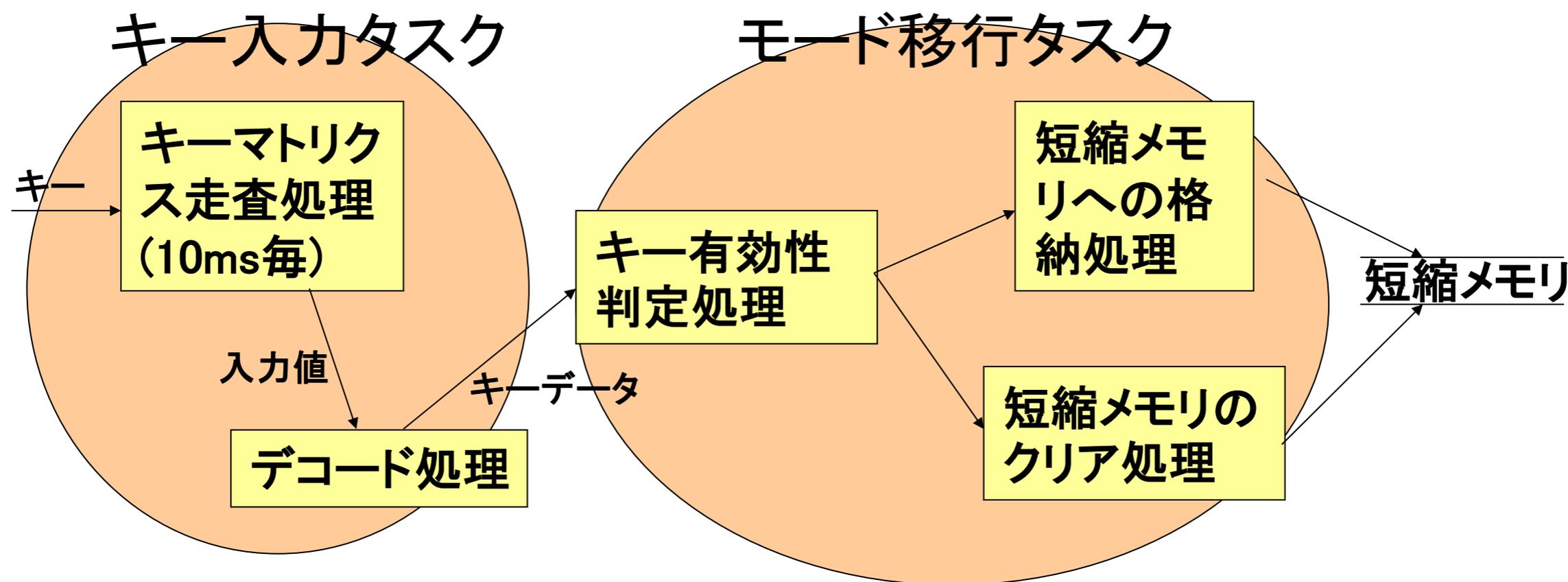
おまけ(タスク分割)

- ▶ 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う。
- ▶ 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。

参考:各仕様の比較

- ▶ 本資料は、 μ ITRON3.0/ μ ITRON4.0/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、 μ ITRON4.0仕様にに基づいてます

出典元

文書名：『 μ ITRON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
メールボックス	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
<p>C言語記述形式</p> <pre>void task(INT stacd) { ; }</pre>	<p>C言語記述形式</p> <pre>void task(VP_INT exinf) { ; }</pre> <p>exinf: sta_tskで起動した場合stacd act_tskで起動した場合exinf</p>	<p>C言語記述形式</p> <pre>void task(INT stacd, VP exinf) { ; }</pre>
<p>タスクの起動方法 システムコールsta_tsk</p>	<p>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</p>	<p>タスクの起動方法 システムコールsta_tsk</p>
<p>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</p>	<p>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</p>	<p>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)</p>
		<p>ラウンドロビンスケジューリングをサポート</p>

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	
メールボックスのメッセージ管理がリンク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	irmsm_tsk		
強制待ち状態からの強制再開	ifrsn_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	irotd_rdq	irotd_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
- ▶ μ ITRON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

T-Kernel Line-upについて(参考)

μ T-Kernel

μ T-Kernel は、小規模組込みシステムをターゲットとしたリアルタイムOSです。ARMのプロセッサで言えば、Cortex-Mがターゲットとする応用分野に適したOSです。非常にコンパクトなOSであり、たとえばROMが8KB、RAMが4KB程度しかないハードウェアでも動作するように作られています。一般的に組込みLinuxと呼ばれているものでは、どんなに小さく設定しても数MB以上のRAMを使用しますが、これと比較してもいかにコンパクトなOSであるかがよく解ると思います。

これだけコンパクトなOSではありますが、他のRTOSと比較してもかなり充実したAPIを備えています。特に、タスク関連のAPIや、タスク間同期・通信関連のAPIについては、他のRTOS（例えばFreeRTOSやμC/OS-II）と比べてもかなり充実しています。これは、RTOSの本質がCPUの計算時間を実行したい処理に最適に割り当てることにあるために非常に重要なことです。μ T-Kernelを動作させるような小規模な組込みシステムでは、一般的に省電力・省コストのローエンドなプロセッサが用いられる想定ですが、この時間の割り当てに関する制御が効率良くプログラムから行えるようにするためには、それ相応のAPI群が必要だからです。

具体的に、μ T-Kernelでは以下のような機能を提供しています。使わない機能については、ROMおよびRAMの節約のために、機能を個別に取り外すことも可能になっています。

タスク管理機能

タスク付属同期機能

同期・通信機能

拡張同期・通信機能

メモリプール管理機能

時間管理機能

割込み管理機能

サブシステム管理機能

デバイス管理機能

デバッグサポート機能

T-Kernel 2.0

T-Kernel 2.0 を一言で表すと、高性能・高機能な組み込みシステムのためのRTOSとって良いでしょう。ARM のプロセッサで言えば、Cortex-A がターゲットとする応用分野に適しています。ちょうど、RTOS と情報系OS の「いいところ取り」をしたような設計となっており、 μ T-Kernelと同様なRTOSの機能を基本とした上で、情報系OS の機能をサポートできるように設計・実装がなされたものです。

情報系OSとの「いいところ取り」という観点から重要なのが「サブシステム管理機能」です。これは、T-Kernel 2.0 というRTOS の上にOS の機能を追加するためのメカニズムで、これを用いることでRTOS をベースとして、例えばファイル管理機能やネットワーク通信機能といった情報系の機能を、OS 機能として追加することが可能となります。さらには完全な情報系OS の機能をOS 機能として追加することすら可能で、実際にT-Kernel Standard Extension は完全な情報系OS の機能を実現したOS 拡張ですし、その他にも会員企業によってPOSIX 拡張等が開発された例もあります。

なお、前節で説明した μ T-Kernel にも実は「サブシステム管理」という機能が列挙されていますが、T-Kernel 2.0 とは異なりこれはあくまで簡易的なもので、互換性のために提供されるものです。このように、T-Kernel ファミリでは、実現方法や内部的な動作、構成等がいずれも大幅に異なっていますが、それぞれの間でのAPI の互換性が考慮されて設計されています。これにより、「T-Kernel」という共通のAPI を、小さな組み込みシステムから大規模なものにまで、スケラブルに利用することができるようになっていきます。

現在の μ T-Kernel は、T-Kernel 2.0 の前身であるT-Kernel 1.0 を、互換性を保ちながら軽量かつ簡易な構成とする形で設計されたもので、ほぼT-Kernel と共通になっています。ただし、T-Kernel 2.0 は情報系OS の機能を必要とするような比較的大きな構成での利用を想定しているため、 μ T-Kernel に対して以下のような機能を追加したものとなっています。

マイクロ秒単位の時間管理

組み込みシステムにおけるデバイスの制御においては、マイクロ秒単位のきめ細かい時間管理が必要となることも多く、これを実現するためにマイクロ秒単位の時間管理をサポートした。

Cortex-A 系のCPU にみられるような処理の高速化に伴い、マイクロ秒単位の制御が実用化できる環境が整った。

大容量デバイスのサポート

SD カード等を始めとして、組み込み機器においても大容量のデバイスを利用するケースが増えており、これらの大容量デバイスを扱うための機能を追加した。

標準化範囲の拡大

物理タイマやキャッシュ制御等、T-Kernel 2.0 の仕様で規定される機能範囲を拡張した。

これにより、これらの機能を利用する際のAPI が共通化され、ミドルウェアの流通性が向上する。

T2EX

T2EX はT-Kernel 2.0 Extension の略で、直訳すると「T-Kernel 2.0 (の) 拡張」です。その名が示す通り、T2EX はT-Kernel 2.0 のためのOS 拡張機能であり、RTOSに情報系OSのいいところを持ってこられるようにするための、OSのアドオン（拡張機能）のコレクションです。

「T-Engine プロジェクトとT-Kernel のシリーズ」の節では、組込みシステムが高機能化・高性能化、あるいは機能分散化しているという話に触れましたが、ちょうどそのあたりを狙った設計となっています。高度なOS 機能を情報系OS よりも軽量の形で追加することで組込みシステムの高機能化・高性能化をサポートしつつ、ネットワーク通信機能により機能分散化をOS レベルからサポートできるようになっています。

T2EXの主な特徴は以下のとおりです。

高度な組込みシステムのための情報系OS機能の追加

ファイル管理機能やネットワーク通信機能といった、情報系OSの機能を追加。これによって、高機能な組込みシステムの開発を強力にサポートする。

アドオン・アーキテクチャ

限られたROM/RAM の中で高度な組込みシステムを実現するため、OS機能を独立させる。

ビルド時のコンフィギュレーションにより、必要なものだけを取捨選択して開発可能。たとえば、T-Kernel 2.0にネットワーク通信機能だけ追加した構成で開発したい、といったように、拡張機能をそれぞれ独立して付け加えたり取り外したりしてシステムを構築し、コンパクトな組込みシステムを効率的に開発することが可能である。

メモリ保護機能のサポート

高機能な組込みシステムの開発では、OSをアプリケーションから保護する機能が不可欠となる。

情報系OSとは異なる単一空間上のメモリ保護を実現し、効率的かつ安全なシステムの開発を可能にした。

コンパクト

情報系OSの機能を実現しながらも、非常にコンパクトな組込みシステムが開発可能である。

例えば、100KB以内のRAM でファイルシステムを含む機器（デジタルカメラなど）を開発することも可能である。

Linux 等では少なくとも数MB は消費するため、同じ処理を実現する上で極めてコンパクト性が高いと言える。

T2EXの主な機能を列挙すると以下の通りです。これらはT-Kernel 2.0のAPI と混在させて利用することが可能であり、高いリアルタイム性を実現しながらも高度な処理を実現できます。

[ネットワーク通信機能](#)

[ファイル管理機能](#)

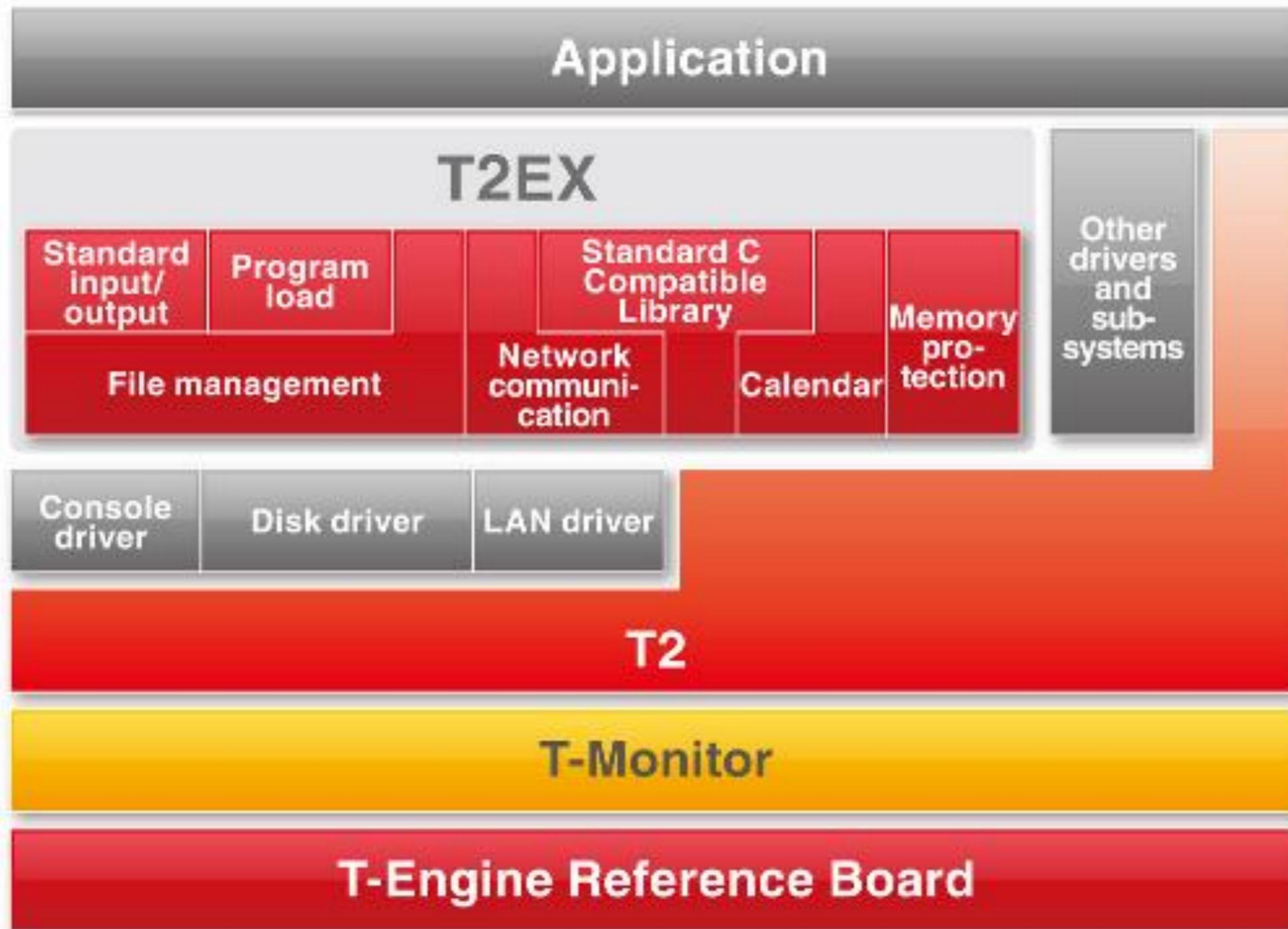
[標準C互換ライブラリ（標準入出力機能を含む）](#)

[カレンダー機能](#)

[プログラムロード機能](#)

[メモリ保護機能](#)

なお、T2EX のアーキテクチャは以下に示す通りで、図に示される機能間の依存関係を満たす範囲で前述の各機能を付け加えたり取り外したりといったことが可能になっています。



【講座】T-Kernel/ITRON入門テキスト「ITRON入門」

著者 T-Engine Forum

本テキストは、クリエイティブ・コモンズ 表示 - 継承 4.0 国際 ライセンスの下に提供されています。

<http://creativecommons.org/licenses/by-sa/4.0>



Copyright ©2014 T-Engine Forum

【ご注意およびお願い】

- 1.本テキストの中で第三者が著作権等の権利を有している箇所については、利用者の方が当該第三者から利用許諾を得てください。
- 2.本テキストの内容については、その正確性、網羅性、特定目的への適合性等、一切の保証をしないほか、本テキストを利用したことにより損害が生じても著者は責任を負いません。
- 3.本テキストをご利用いただく際、可能であれば office@t-engine.org までご利用者のお名前、ご所属、ご連絡先メールアドレスをご連絡いただければ幸いです。