μITRON から
μT-Kernel/T-Kernel への
移行ガイド

Version 1.00.00

(2010/11/01)

T-Engine フォーラム

目次

図目次

表目次

本書の構成と読み方

本書では移植対象としてμT-Kernelを例にとり説明しています。ただし、本書の説明は基本的にT-Kernelにも適応可能な内容になっています。μITRONからT-Kernelに移植される方も活用してください。

μT-KernelとT-Kernelの差異についてはμT-Kernel仕様書に記載されています。

第1章ではμITRONとμT-Kernelのそれぞれの特徴や違い、登場した背景について説明します。

第2章では、μITRON4.0からμT-Kernelへの移行について説明します。

一般に移行方法としては、以下のような方法があります。

- μT-Kernelのシステムコールに置換する
- ラッパ関数などを利用して移行する

この章ではμITRONからμT-Kernelに移行するための具体的手順について説明します。

どの移行方法を採用すべきか検討している開発者にとっては移行のコストや開発期間を算出する根拠として利用できるでしょう。また、実際に移行作業を行う開発者はリファレンスガイドとして利用できます。

第3章では、ラッパーを利用してのμITRON4.0からμT-Kernelへの移行について説明します。

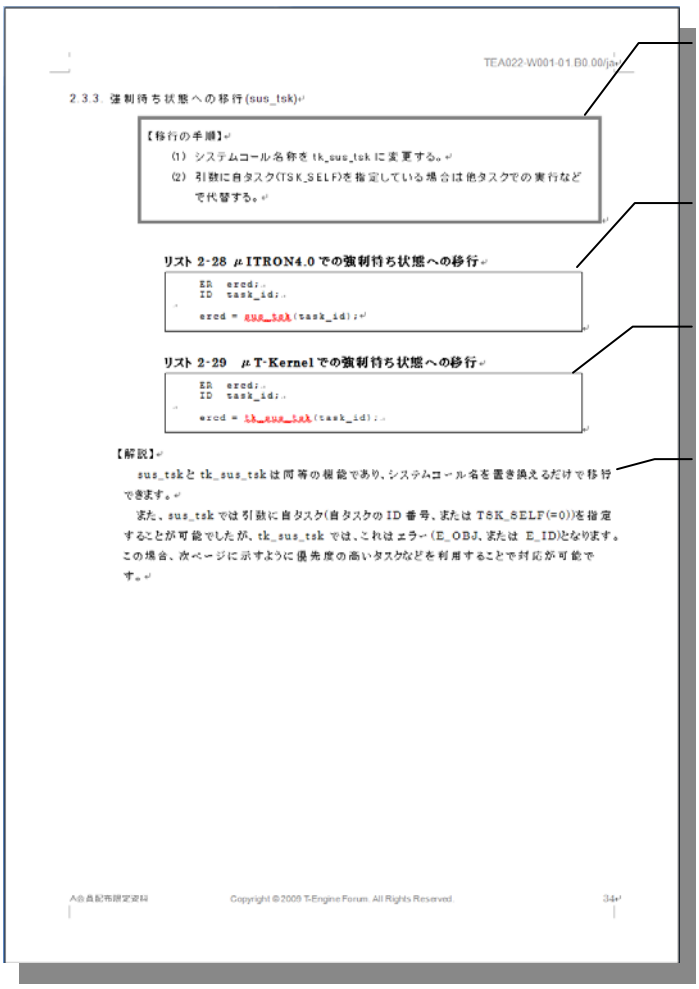
ラッパーを利用すれば、ほとんど変更することなくμITRONのプログラムをμT-Kernelで利用できるようになります。

第4章では、具体的にターゲットとする機器の開発環境においてμT-Kernelを利用する方法やラッパーなどの移行に利用できる関連製品について説明します。

開発環境については様々な組み合わせがあるため特定の開発環境の移行方法を示すことはできませんが、本章を読むことにより参考になると思います。

なお、各章に掲載されているサンプルプログラムでは、タスクIDやセマフォIDなどは特に断りなくtask_idやsem_idなどとして使用しています。これらには別途値が設定されている必要がありますが、サンプルプログラムの要点を明確にするために省略してあります。各変数に設定すべきIDについては各オブジェクトの生成に関するシステムコールの説明、または、静的APIの説明を参照してください。

各システムコールの移行方法のページ構成は以下のとおりです。



移行の手順
各システムコール毎に、μITRON から T-Kernel に移行する手順を記載しています。

移行元のソース例
移行元のμITRON における対象システムコールの記述例を記載しています。

移行後のソース例
移行後のμT-Kernel における対象システムコールの記述例を記載しています。

解説
各システムコールの移行に関する解説を記載しています。

第1章 概要

本章では、μITRON からμT-Kernel への移行を開始する前に、μITRON や μT-Kernel とはどのようなもので、どのように違うのかなどについて説明します。

1.1. μITRON と μT-Kernel

μITRON(マイクロ・アイトロン)の ITRON とは Industrial TRON(一・トロン)の略です。組込み機器などの機械を制御するための RTOS(リアルタイムオーエス)として設計されました。1987 年に最初の ITRON 仕様書が発行され、既に 20 年以上の実績を持っています。現在でも多くの組込み機器に採用されており、RTOS のデファクトスタンダードとなっています。

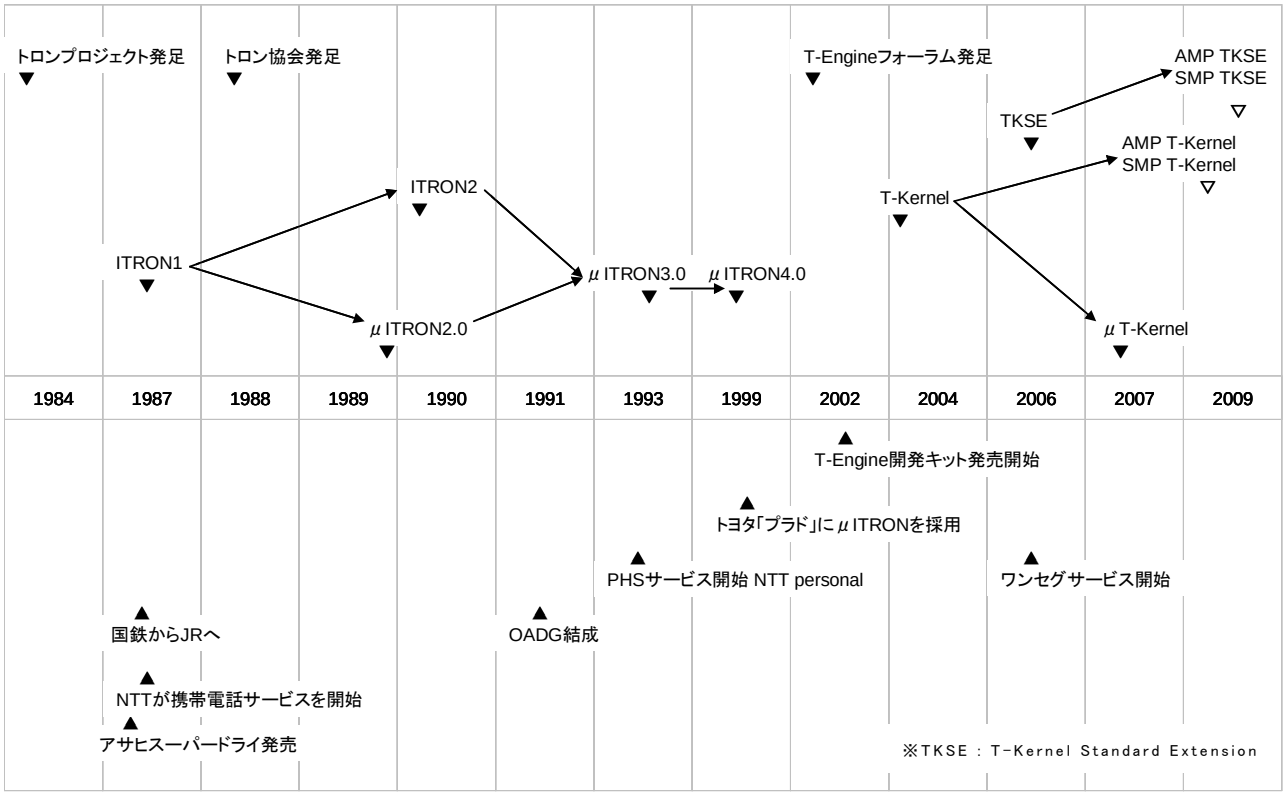


図 1-1 μITRON と T-Kernel の公開時期

μT-Kernel(マイクロ・ティー・カーネル)は ITRON の流れを汲む次世代 RTOS である T-Kernel の小型版として開発されました。μT-Kernel は、T-Engine フォーラムが仕様書とリファレンスコードを無償で公開しています。T-Kernel とμT-Kernel は ITRON と違ってリファレンスとなるソースコードを T-Engine フォーラムが無償で公開しています。リファレンスコードを公開することで仕様書だけでは規定することができない細かな部分を含めて OS の仕様を一定にし、アプリケーションやミドルウェアなどの上位ソフトウェアの流通性を向上させるようにしています。

1.2. 本書の目的

本書は、μITRONを用いてのシステム開発経験はある次のような開発者を読者として想定しています。

- μITRONからμT-Kernelへの移行を検討している開発者
- μITRONの知識があり、システムをμT-Kernelで構築しようとしている開発者

このような開発者が、既存のシステムのRTOSをμITRONからμT-Kernelに移行する場合や、μT-Kernelを用いて新規にシステムを構築しようとする場合、しばしば次のような課題に直面します。

- 移行方法が確立されていない
- 移行に伴い発生する問題点が見えない
- 移行に要する全体の作業量が見えない

本書の目的は、移行に必要な作業を具体的に示すことで、μT-Kernelを初めて利用する開発者がスムーズに作業を進められるよう支援することにあります。

そこで本書では基本的な移行作業について、以下の機能を例にとり説明します。

- タスク管理機能
- タスク付属同期機能
- セマフォ　　(同期・通信機能)
- メールボックス　　(同期・通信機能)

本書で説明する例に基づいて基本的な移行方法を理解することで、μT-Kernelが提供する他の多くの機能についても同様の手順で移行することができるようになります。

なお、本書の説明は基本的にT-Kernelにも適応可能な内容になっています。μITRONからT-Kernelに移植される方も活用してください。μT-KernelとT-Kernelの差異についてはμT-Kernel仕様書を参照してください。

1.3. μT-Kernelに移行する理由

μT-Kernelを含むT-Kernelシリーズには、次のような共通する特長があり、一度作成したアプリケーションやミドルウェアを様々なシステムに流用することができます。その結果、低コストと短納期が実現可能となります。

A.各種CPU間でのミドルウェアの高い流通性

B.T-Kernelシリーズの高い互換性

A.各種CPU間でのミドルウェアの高い流通性

T-Kernelシリーズは、ミドルウェアの流通性を高めることを目的としてOSの仕様だけでなく実装や動作が同じになるよう設計されています。また、ミドルウェアの作成方法についても規定が決められており、これに則って作成したミドルウェアはインタフェース上の問題を引き起こすことなく別のシステムに移植することが可能です。その結果、様々なミドルウェアを利用することができます。

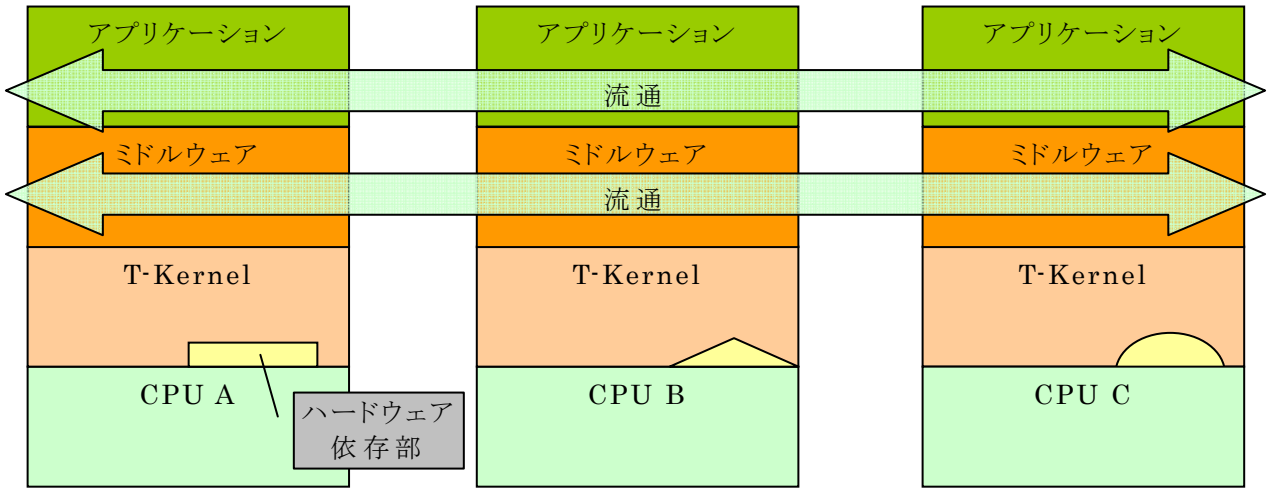


図 1-2 各種 CPU 間でのミドルウェアの高い流通性

B. T-Kernel シリーズの高い互換性

T-Kernel シリーズは、ワンチップマイコンにも対応した 16 ビット CPU 向けの μT-Kernel、32 ビット CPU 向けの T-Kernel、さらにマルチプロセッサ/マルチコアプロセッサ対応の MP T-Kernel から構成されています。また、T-Kernel、MP T-Kernel は、T-Kernel Standard Extension を組み込むことにより大規模なソフトウェア開発に対応します。T-Kernel シリーズの大きな特長として、各 OS は、API レベルで高い互換性を持っていることが挙げられます。

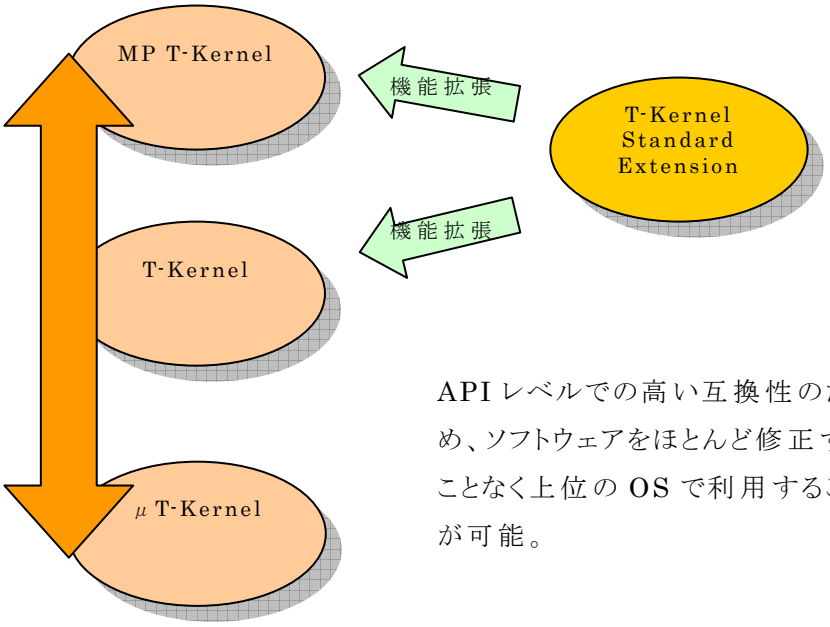


図 1-3 T-Kernel シリーズの高い互換性

そのため μT-Kernel で作成したアプリケーションやミドルウェアはわずかな修正で他の T-Kernel にも利用することができ、開発コストの低減と短納期につながります。

1.4. 対象とする ITRON

本書では、μITRON の最新バージョンであるμITRON4.0 を対象としています。

1.5. 対象とするμT-Kernel

本書の内容は、μT-Kernel 1.01.00 仕様に基づいて説明します。

本書の説明は、他のバージョンのμT-Kernel や T-Kernel にも共通的に適用できる内容になっています。他のバージョンのμT-Kernel や T-Kernel を利用される場合も参考にしてください。

第2章　μITRON4.0 からμT-Kernel への移行

本章では、μITRON4.0 からμT-Kernel への移行方法および仕様の相違点など、移行の際に注意するポイントについて説明します。

μITRON4.0 仕様RTOS上で動作していたプログラムをμT-Kernelに移行するためには、μITRON4.0 のシステムコール [1]をμT-Kernelのシステムコールに置き換えるか、またはラッパ関数を利用するなどの対応が必要となります。単純に名称を置き換えることで対応できるシステムコールもありますが、システムコールによってはパラメータや周辺のプログラムコードに対しても調整が必要となる場合もあります。

本章では、μITRON4.0 のシステムコールをμT-Kernel のシステムコールに置き換えることでμT-Kernel に移行する方法について説明します。

システムコールを置き換える方法は、利用しているシステムコールとシステムコールを利用している箇所に応じた変更が個別に必要となりますが、ラッパ関数を利用するよりも高速にシステムコールの処理が行えるので、アプリケーションの実行速度を犠牲にすることなく移行できるといった特徴があります。また、システムコールの置き換えによる移行が完了すれば、プログラムが利用するシステムコールはμT-Kernel に統一されますので、プログラムのメンテナンスが容易になります。

以下では、システムコールを置き換える際に調整が必要となる項目について、システムコール毎にまとめてあります。移行作業の参考にしてください。

[1] μITRON4.0 仕様では「サービスコール」と呼びますが、本書ではμT-Kernel に合わせて「システムコール」としてあります。

2.1. μITRON4.0 からの主な変更点

(1) システムコール名の変更
(2) オブジェクト ID の割り当て方法の変更
(3) 動的 API への統一
(4) 拡張情報 exinf の使い方の統一
(5) 同期・通信オブジェクトなどへの拡張情報 exinf の追加
(6) デバッガサポート機能の追加(dsname)
(7) 類似したシステムコールの統一

2.1.1. システムコール名の変更

【解説】

μT-Kernel ではシステムコールの先頭に”tk_”が付きます。

全てのシステムコールが“tk_”を付けることで移行できる訳ではありませんが、多くのシステムコールがこの方法で移行可能です。

2.1.2. オブジェクト ID の割り当て方法の変更

【解説】

μT-Kernel ではオブジェクトを生成する場合に、ID を自動的に割り当てます。

μITRON4.0 において ID を即値で指定してオブジェクトを生成しているプログラムでは、ID を自動生成する方法に変更してください。

2.1.3. 動的 API への統一

【移行の手順】

(1) 初期起動タスクの優先度を最高(1)に変更する。

(2) 各オブジェクトを動的生成する。

【解説】

静的生成のみを利用したμITRON4.0 で動作するプログラムをμT-Kernel に移行する場合、初期起動タスクの優先度を最高(1)に変更しておく必要があります。

μT-Kernel では動的生成のみをサポートしています。このため、TA_ACT 属性を指定してのタスク生成を行っている箇所において TA_ACT の代わりにタスクの実行(tk_sta_tsk)を呼び出す処理を追加した場合、初期起動タスクがプリエンプトされてしまい、意図した状態になる前にアプリケーションの実行が開始されてしまい、移植前とは同じ動作にならないこともあります。

以下の例では初期優先度の高い順に task3、task2、task1 と実行されることを期待しています。

リスト 2-1 μITRON4.0 の静的生成例

```
     :
     :                                                    初期優先度
     :
CRE_TSK( 1, {TA_HLNG|TA_ACT,0x00000001, task1, 3, 0x400, NULL });
CRE_TSK( 2, {TA_HLNG|TA_ACT,0x00000002, task2, 2, 0x400, NULL });
CRE_TSK( 3, {TA_HLNG|TA_ACT,0x00000003, task3, 1, 0x400, NULL });
     :
```

T-Kernel の usermain での記述例にある、初期タスクの優先度を一時的に最高優先度(1)に引き上げる記述を忘れると、task1 が最初に起動されてしまい、task1 が待ち状態か終了状態になるまで初期タスクには戻ってきません。これではμITRON4.0 の静的生成の例と同様の動作にはなりません。

リスト 2-2　μT-Kernel の usermain での記述例

```
EXPORT  INT usermain( void )
{
    T_CTSK  pk_ctsk;
    ER      ercd;
        :

    tk_chg_pri( TSK_SELF, 1 );          /* 優先度を最高に設定   */

    /* オブジェクトの生成   */
        :

    /* task1 の生成と起動       */
    pk_ctsk.tskatr  = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task    = task1;
    pk_ctsk.itskpri = 3;
    pk_ctsk.stksz   = sizeof(task_stack1);
    pk_ctsk.bufptr  = task_stack1;

    task_id = tk_cre_tsk( &pk_ctsk );
    ercd = tk_sta_tsk( task_id, 0 ); /* TA_ACT の代わり     */

    /* task2 の生成と起動       */
    pk_ctsk.tskatr  = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task    = task2;
    pk_ctsk.itskpri = 2;
    pk_ctsk.stksz   = sizeof(task_stack2);
    pk_ctsk.bufptr  = task_stack2;

    task_id = tk_cre_tsk( &pk_ctsk );
    ercd = tk_sta_tsk( task_id, 0 ); /* TA_ACT の代わり     */

    /* task3 の生成と起動       */
    pk_ctsk.tskatr  = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task    = task3;
    pk_ctsk.itskpri = 1;
    pk_ctsk.stksz   = sizeof(task_stack3);
    pk_ctsk.bufptr  = task_stack3;

    task_id = tk_cre_tsk( &pk_ctsk );
    ercd = tk_sta_tsk( task_id, 0 ); /* TA_ACT の代わり     */

        :

    tk_slp_tsk( TMO_FEVR );             /* 関数を終了させない   */
    return 0;
}
```

各タスクが個別に実行されないようにするために必要な記述。

この待ち状態になった時点で、静的生成と同様の状態を作ることができる。

2.1.4. 拡張情報 exinf の使い方の統一

【解説】

μITRON ではタスクを起動する際にはそのタスクの拡張情報が渡されます。ただし、起動コードを指定してタスクを起動するサービスコール(sta_tsk)によってタスクが起動された場合には、タスクの生成時に指定した拡張情報ではなく、指定された起動コードを渡す仕様になっています。

これに対しμT-Kernel では、タスクの生成時に指定された拡張情報 exinf とタスクの起動時に指定されたタスク起動コード stacd の両方がそれぞれ別の変数として渡されます。

リスト 2-3 μITRON でのタスクのC言語による記述形式

```
void task( VP_INT exinf )
{
    タスク本体
    ext_tsk();
}
```

μITRON4.0 仕様書 (Ver. 4.03.03) P.80 より抜粋

リスト 2-4 μT-Kernel でのタスクのC言語による記述形式

```
void task( INT stacd, VP exinf )
{
    /*
        処理
    */
    tk_ext_tsk(); または tk_exd_tsk(); /* タスクの終了 */
}
```

μT-Kernel 仕様書 (TEF020-S004-01.01.00/ja) P.35 より抜粋

なお、μT-KernelもμITRONも拡張情報 exinf の内容には関与しません。ユーザが自由に値を設定することができます。

2.1.5. 同期・通信オブジェクトなどへの拡張情報 exinf の追加

【解説】

タスク以外のオブジェクト(同期・通信のセマフォ等)にも拡張情報が追加されました。
オブジェクトの生成時や、状態参照時に拡張情報を利用することが可能になりました。

2.1.6. デバッガサポート機能の追加（dsname）

【解説】

オブジェクトの生成情報に、“dsname:DS オブジェクト名称”のメンバが追加されました。

TA_DSNAME を指定した場合に dsname は有効となり、DS オブジェクト名称として設定されます。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能です。

2.1.7. 類似したシステムコールの統一

【解説】

μT-Kernel では、類似したシステムコールが統一されました。

たとえば、待ち状態になるシステムコールにおいて、タイムアウトありのシステムコールはタイムアウトなしのシステムコールに統一されています。

例）	μITRON	slp_tsk	タイムアウト指定なし（無限待ち）
		tslp_tsk	タイムアウト指定あり
	μT-Kernel	tk_slp_tsk	タイムアウト指定あり

詳細については、各システムコールの移行の中で解説しています。

2.2. タスク管理機能

μITRON4.0 のシステムコールとμT-Kernelのシステムコールの対応は 表 2-1 の通りです。

表 2-1 μITRON4.0 と T-Kernel のシステムコールの対応

機能	μITRON4.0	μT-Kernel
タスクの生成(静的 API)	CRE_TSK	tk_cre_tsk
タスクの生成	cre_tsk	
タスクの生成(ID 番号自動割付け)	acre_tsk	
タスクの削除	del_tsk	tk_del_tsk
タスクの起動	act_tsk	tk_slp_tsk
タスクの起動	iact_tsk	tk_wup_tsk
タスク起動要求のキャンセル	can_act	tk_can_wup
タスクの起動(起動コード指定)	sta_tsk	tk_sta_tsk
自タスクの終了	ext_tsk	tk_ext_tsk
自タスクの終了と削除	exd_tsk	tk_exd_tsk
他タスクの強制終了	ter_tsk	tk_ter_tsk
タスク優先度の変更	chg_pri	tk_chg_pri
タスク優先度の参照	get_pri	tk_ref_tsk
タスクの状態参照	ref_tsk	
タスクの状態参照(簡易版)	ref_tst	

μT-Kernel ではタスク生成の機能は tk_cre_tsk に、タスクの情報を取得する機能は tk_ref_tsk に集約されます。また、タスクの起動要求をキューイングする機能はありませんので、tk_slp_tsk、tk_wup_tsk、tk_can_wup などで代用します。タスクの起動要求をキューイングすることがなければ、tk_ext_tsk、tk_sta_tsk への変更で対応できます。

その他の機能は、概ねシステムコールの名称変更で移行できます。

以下、それぞれのシステムコールについて個別に説明します。

2.2.1. タスクの生成（CRE_TSK / cre_tsk / acre_tsk）

ここでは、タスク生成の移行方法について、以下の4つに分けて説明します。

(A) acre_tsk から tk_cre_tsk へ　　ID 番号自動割付けからの移行
(B) cre_tsk から tk_cre_tsk へ　　ID 番号指定からの移行
(C) CRE_TSK から tk_cre_tsk へ　　静的 API からの移行
(D) タスクの記述形式　　登録する関数の記述について

T-Kernel はプログラムの動的なロードを前提に設計されています。このため、タスクなどのオブジェクトはプログラムのロード時に動的に生成して ID も自動的に割り付けることになっています。μT-Kernel では T-Kernel との互換性を優先し、タスクの生成の tk_cre_tsk ではタスク ID は自動割付けになっています。

tk_cre_tsk はμITRON4.0 の acre_tsk とほぼ等価のシステムコールですので、acre_tsk を利用している場合はそのままμT-Kernel に移行できます。他のシステムコール(CRE_TSK/cre_tsk)は移行に際して調整が必要になります。

また、タスクとして登録する関数の C 言語での記述形式も異なっていますが、これはテンプレートを用意して変更していけば比較的簡単に対応できます。タスクの記述形式については、各システムコールの移行方法の説明後に別途説明します。

(A) acre_tsk から tk_cre_tsk へ

【移行のポイント】
(1) システムコール名称の変更
(2) タスク属性の追加
(3) スタック指定用メンバ変数名の変更

acre_tsk と tk_cre_tsk は、ほぼ同等の機能です。
タスク属性の追加とタスク生成情報のメンバ変数名を変更することで移行できます。

リスト 2-5　μITRON4.0 でのタスクの生成　(acre_tsk)

```
    T_CTSK  pk_ctsk;
    ID      task_id;
static INT  task_stack[0x400/sizeof(INT)];

        :

    pk_ctsk.exinf   = (VP_INT)0;          /* タスクの生成情報  */
    pk_ctsk.tskatr  = TA_HLNG;
    pk_ctsk.task    = task;
    pk_ctsk.itskpri = 1;
    pk_ctsk.stksz   = sizeof(task_stack);
    pk_ctsk.stk     = task_stack;

    task_id = acre_tsk( &pk_ctsk );   /* タスクの生成と          */
                                      /* タスク ID の自動割当    */
        :
```

リスト 2-6　μT-Kernel でのタスクの生成

```
    T_CTSK  pk_ctsk;
    ID      task_id;
static INT      task_stack[0x400/sizeof(INT)];

        :

    pk_ctsk.exinf    = (VP)0;
    pk_ctsk.tskatr   = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task     = task;
    pk_ctsk.itskpri  = 1;
    pk_ctsk.stksz    = sizeof(task_stack);
    pk_ctsk.bufptr   = task_stack;

    task_id = tk_cre_tsk( &pk_ctsk );

        :
```

TA_RNG0 はタスクの保護レベルを指定しています。TA_RNGn (n = 0, 1, 2, 3)で保護レベル n で動作するプログラムであることを指定します。μT-Kernel では保護レベルは使用しませんので無視しています。このため、TA_RNG0 を指定しなくても動作しますが、T-Kernel とのプログラムの互換性を確保するため、移行の際には必ず保護レベルを指定するようにしてください。

後日、T-Kernel に移植する際、プログラムの性質を考慮したうえでどのレベルで実行させるかを検討して決定してください。T-Kernel では各保護レベルを以下のように規定しています。

表 2-2　T-Kernel の保護レベル

タスク属性	保護レベル		用途
TA_RNG0	高	レベル 0	システムソフトウェア(OS、デバイスドライバなど)
TA_RNG1	↑	レベル 1	システムアプリケーション
TA_RNG2	↓	レベル 2	未使用(予約)
TA_RNG3	低	レベル 3	ユーザアプリケーション

TA_USERBUF は生成するタスクのスタックをアプリケーション(tk_cre_tsk を発行する側)で確保している場合に指定します。このタスク属性を指定するとタスク生成情報のメンバ変数 bufptr が有効となります。

μITRON4.0 では、タスクのスタックはアプリケーションで確保する必要がありましたが、μT-Kernel ではシステムで自動的に確保されます。この場合は、タスク属性として TA_USERBUF を指定せずに、スタックサイズ(stksz)のみを指定します(bufptr の指定も不要です)。TA_USERBUF が無い場合、μT-Kernel がタスクのスタック領域として stksz バイトのメモリ領域を自動的に割当てます。

タスクとして登録する関数(pk_ctsk.task = task; で指定)の記述形式については「(D) タスクの記述形式」で説明します。

(B) cre_tsk から tk_cre_tsk へ

【移行のポイント】
(1) タスク ID 格納処理の追加
(2) システムコール名称の変更
(3) タスク属性の追加
(4) スタック指定用メンバ変数名の変更

T-Kernel ではタスク ID を自動で割り付けます。このため、割り付けられたタスク ID を変数に保存し、後で利用できるようにする処理を追加します。

リスト 2-7 μITRON4.0 でのタスクの生成 (cre_tsk)

```
    T_CTSK   pk_ctsk;
    ER       ercd;
static INT   task_stack[0x400/sizeof(INT)];

        :

    pk_ctsk.exinf    = (VP_INT)0;
    pk_ctsk.tskatr   = TA_HLNG;
    pk_ctsk.task     = task;
    pk_ctsk.itskpri  = 1;
    pk_ctsk.stksz    = sizeof(task_stack);
    pk_ctsk.stk      = task_stack;

    ercd = cre_tsk( 10, &pk_ctsk );   /* タスク ID = 10 のタスク */

        :
```

リスト 2-8 μT-Kernel でのタスクの生成

```
    T_CTSK   pk_ctsk;
    ID       task_id;
static INT   task_stack[0x400/sizeof(INT)];

        :

    pk_ctsk.exinf    = (VP)0;
    pk_ctsk.tskatr   = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task     = task;
    pk_ctsk.itskpri  = 1;
    pk_ctsk.stksz    = sizeof(task_stack);
    pk_ctsk.bufptr   = task_stack;

    task_id = tk_cre_tsk( &pk_ctsk );/* タスク ID は自動割当    */

        :
```

TA_RNG0、TA_USERBUF、bufptr については(A)を参照してください。

μITRON では cre_tsk を使用する場合、タスク ID をアプリケーション側で指定します。即値を指定できるので、システムの起動時からタスクを個別に識別できるといったメリットがありますが、システムを開発する段階で、タスク ID が一意になるように時に開発者が調整しておかなければなりません。一方の μT-Kernel ではタスク ID は自動的に割当てられますので、開発者が割り振る必要はありません。

μT-Kernel への移行に際しては、例えばグローバル変数を用意して、その変数に生成したタスクのタスク ID を指定するといった実装が必要となります。

リスト 2-9　マクロによるタスク ID の指定例（μITRON4.0）

```
#define TID_CLIENT      (10)
        :
    T_CTSK pk_ctsk;
    ER      ercd;
static INT      task_stack[0x400/sizeof(INT)];
        :
    pk_ctsk.exinf   = (VP_INT)0;
    pk_ctsk.tskatr  = TA_HLNG;
    pk_ctsk.task    = task;
    pk_ctsk.itskpri = 1;
    pk_ctsk.stksz   = sizeof(task_stack);
    pk_ctsk.stk     = task_stack;

    ercd = cre_tsk( TID_CLIENT, &pk_ctsk );

        :
```

リスト 2-10　外部変数へのタスク ID の格納例

```
static  ID          tid_client = 0;
#define TID_CLIENT  tid_client
        :
    T_CTSK  pk_ctsk;
static INT      task_stack[0x400/sizeof(INT)];

        :

    pk_ctsk.exinf   = (VP)0;
    pk_ctsk.tskatr  = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task    = task;
    pk_ctsk.itskpri = 1;
    pk_ctsk.stksz   = sizeof(task_stack);
    pk_ctsk.bufptr  = task_stack;

    tid_client = tk_cre_tsk( &pk_ctsk );

        :
```

上記のリストでは、μITRON4.0 も T-Kernel も TID_CLIENT というマクロを定義してあります。このようにすることで例えば、タスク ID を指定する必要のあるシステムコールの移行においてパラメータを変更する必要がなくなります。

なお、タスク ID を格納する変数 tid_client の名称を直接 TID_CLIENT と宣言すればマクロを定義する必要はなくなります。ただ一般には、マクロは大文字、変数名は小文字を利用することが多いようですので、上記リストのようにしました。実際の移行に際しては、各社のコーディングルールに適合するように調整してください。

(C) CRE_TSK から tk_cre_tsk へ

【移行のポイント】
- (1) 初期タスクでのタスクの生成
- (2) タスク ID 格納処理
- (3) システムコール名称の変更
- (4) タスク属性の追加
- (5) スタック指定用メンバ変数名の変更

T-Kernel は、アプリケーションプログラムの動的なロードを前提として設計されているため、静的 API はサポートしていません。μT-Kernel も T-Kernel との互換性を重視した設計になっており、静的 API はサポートしていません。μITRON4.0 の静的 API を使用している場合は、CRE_TSK を tk_cre_tsk に置き換え、システムの起動時に実行されるタスクなどでタスクを生成させることで対応します。

リスト 2-11 μITRON4.0 でのタスクの生成 (CRE_TSK)

```
CRE_TSK(tsk_id, { TA_HLNG, (VP_INT)0, task, 1, sizeof(task_stack), task_stack} );
```

リスト 2-12 μT-Kernel でのタスクの生成

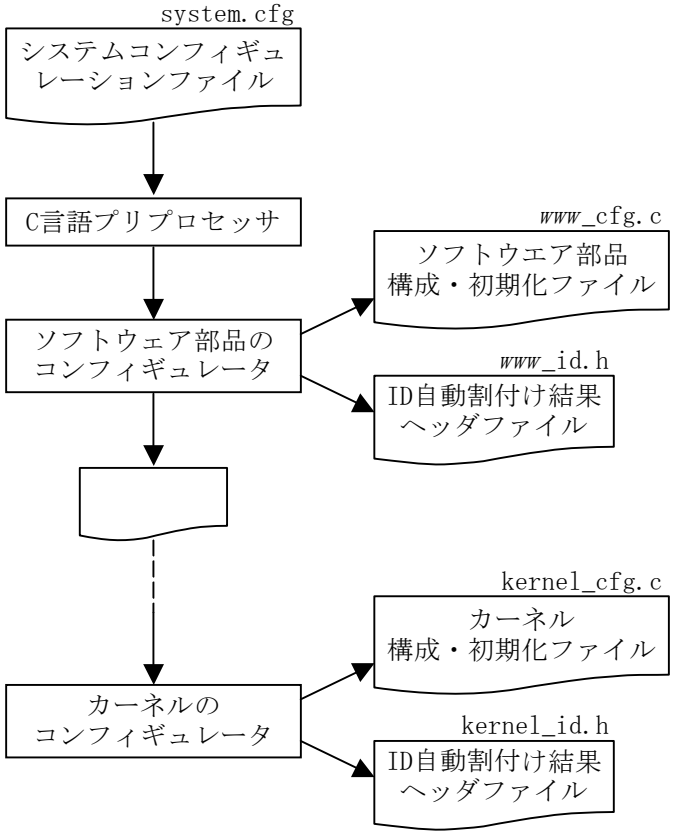
```
    T_CTSK   pk_ctsk;
    ID       task_id;
static INT   task_stack[0x400/sizeof(INT)];

        :

    pk_ctsk.exinf    = (VP)0;
    pk_ctsk.tskatr   = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task     = task;
    pk_ctsk.itskpri  = 1;
    pk_ctsk.stksz    = sizeof(task_stack);
    pk_ctsk.bufptr   = task_stack;

    task_id = tk_cre_tsk( &pk_ctsk );

        :
```

TA_RNG0、TA_USERBUF、bufptr については(A)を参照してください。

μITRON4.0 の静的 API である CRE_TSK の場合、開発者はシステムコンフィギュレーションファイルに静的に生成したいタスクの数だけ静的 API を記述し、そのファイルをコンフィギュレータに通すことでタスクを静的に生成します。

μT-Kernel では静的 API はありませんので、tk_cre_tsk を呼び出してタスクを生成することで対応します。

静的 API を使用した場合、システムが起動した段階で既にタスクが生成されています。μT-Kernel の場合、システム起動後にタスクを生成することになりますので、例えば初期タスク内で静的 API に対応する tk_cre_tsk を全て呼び出し、その後、アプリケーションタスクを起動するといった構成にすることで同じような振る舞いにさせることができます。



※図中のファイル名は例である.

図 2-1 システムコンフィギュレーションファイルの処理手順

（μITRON4.0 仕様書「2.1.10 システムコンフィギュレーションファイル」より）

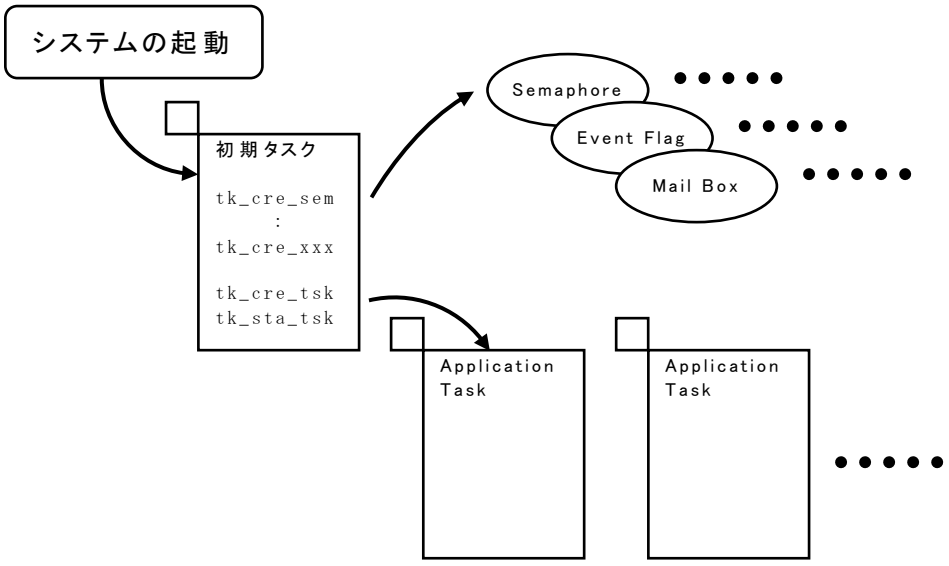


図 2-2 μT-Kernel の初期タスクを使用したオブジェクトの生成手順

μT-Kernel では、初期タスクの関数名は knl_init_task であり、このタスクから usermain という関数が呼出されます。初期タスク実行時に動作させたい処理は、この関数に記述します。

リスト 2-13　usermain での記述例

```
EXPORT  INT usermain( void )
{
    T_CTSK  pk_ctsk;
    ID      task_id1;
    ER      ercd;
        :

    tk_chg_pri( TSK_SELF, 1 );          /* 優先度を最高に設定  */

    /* オブジェクトの生成  */
        :

    /* タスクの生成        */
    pk_ctsk.tskatr  = TA_HLNG | TA_RNG0 | TA_USERBUF;
    pk_ctsk.task    = task;
    pk_ctsk.itskpri = 1;
    pk_ctsk.stksz   = sizeof(task_stack);
    pk_ctsk.bufptr  = task_stack;

    task_id1 = tk_cre_tsk( &pk_ctsk );
    if( task_id1 < 0 ){
        return 0;
    }
    ercd = tk_sta_tsk( task_id1, 0 );
    if( ercd < 0 ){
        return 0;
    }

        :

    tk_slp_tsk( TMO_FEVR );             /* 関数を終了させない  */
    return 0;
}
```

タスクやセマフォなど、必要なオブジェクトの生成処理はこの関数にまとめて記載することで、アプリケーションプログラムの本体が起動する時点で必要となるオブジェクトを用意しておくことができます。

ただし、初期タスクと生成するタスクの優先度によっては、タスクを生成・起動した時点でディスパッチが発生する場合があります。上の例ではこのようなことがないように、タスクの生成を開始する前に、tk_chg_pri(TSK_SELF, 1) によって初期タスクの優先度を最高に設定しています。

また、μT-Kernel は、usermain 関数を終了するとシステムを終了する構成になっています。このため、usermain 関数から戻らないように tk_slp_tsk(TMO_FEVR) によって初期タスクを待ち状態にしています。または、初期タスクをアプリケーションタスクの1つとして利用することも可能です。この場合、初期タスクのスタックサイズなどが、inittask_def.h で規定されていますので、適宜調整が必要となります。

必要なタスクを生成後、初期タスクを停止（リスト 2-13 では起床待ち状態に移行）するとアプリケーションタスクが動作を開始します。

(D) タスクの記述形式

μITRON4.0 とμT-Kernel のタスクの記述形式はそれぞれ以下の通りです。

リスト 2-14　μITRON4.0 のタスクの記述形式

```
void    task( VP_INT exinf )
{
    タスク本体
    ext_tsk();
}
```

(μITRON4.0 仕様 「4.1 タスク管理機能」より)

リスト 2-15　μT-Kernel のタスクの記述形式

```
void    task( INT stacd, VP exinf )
{
    /*
        処理
    */
    tk_ext_tsk();  または  tk_exd_tsk();   /* タスクの終了 */
}
```

(T-Kernel 仕様書 「タスク生成 tk_cre_tsk」より)

exinf はタスク生成時に指定する拡張情報です。どのような値を指定するかは自由であり、μITRON、μT-Kernel 共に exinf の内容については関知しません。

μT-Kernel では stacd が追加されています。stacd は tk_sta_tsk で指定します。μITRON でも sta_tsk において stacd を指定することはできますが、sta_tsk でタスクを起動した場合は exinf の値は使用できないことになります。μT-Kernel ではそれぞれ別に指定します(exinf は tk_cre_tsk、stacd は tk_sta_tsk)ので、両方のパラメータを利用することができます。

タスクの終了は自タスクの終了(ext_tsk、tk_ext_tsk)システムコールによって行います。終了と同時にタスクを削除する場合は自タスクの終了と削除(exd_tsk、tk_exd_tsk)システムコールを利用します。これらはシステムコールの名称のみ変更すれば対応できます。

ただし、μITRON4.0 仕様では「タスクのメインルーチンからリターンした場合には，ext_tsk を呼び出した場合と同じ振る舞いをする(すなわちタスクを終了する)．」と規定されており、ext_tsk がなくても良いことになっています。一方、μT-Kernel では tk_ext_tsk、または、tk_exd_tsk による終了が必須となりますので、タスクを終了させるシステムコールを省略して関数を記述している場合は対応が必要となります。

2.2.2. タスクの削除（del_tsk）

【移行のポイント】
（1）システムコール名称の変更

タスクの削除はシステムコール名を置き換えるだけで移行できます。

リスト 2-16　μITRON4.0 でのタスクの削除

```
ID  task_id;
ER  ercd;

ercd = del_tsk( task_id );
```

リスト 2-17　μT-Kernel でのタスクの削除

```
ID  task_id;
ER  ercd;

ercd = tk_del_tsk( task_id );
```

task_id には削除対象となるタスクの ID が指定されているものとします。

上記の例ではタスク ID を変数 task_id として指定していますが、μITRON4.0 の cre_tsk か CRE_TSK でタスクを生成した場合は、タスク ID を固定値として生成します。この場合、del_tsk のタスク ID も（cre_tsk や CRE_TSK で指定したのと）同じ固定値を指定します。

2.3. タスク付属同期機能

【タスク付属同期機能】

主な変更点

- タスクコンテキスト、非タスクコンテキストから発行するシステムコールを統一した。

(A)機能一覧

タスク付属同期機能の機能は次の通りです。μITRON4.0 から機能の追加・変更はありません。

（1）起床待ち
（2）タスクの起床
（3）タスク起床要求のキャンセル
（4）待ち状態の強制解除
（5）強制待ち状態への移行
（6）強制待ち状態からの(強制)再開
（7）自タスクの遅延

(B) μITRON4.0 とμT-Kernel のシステムコールの対応

μITRON4.0 のシステムコールとμT-Kernelのシステムコールの対応は 表 2-3 の通りです。

表 2-3 μITRON4.0 と T-Kernel のシステムコールの対応

機能	μITRON4.0	μT-Kernel
起床待ち	slp_tsk	tk_slp_tsk
起床待ち(タイムアウトあり)	tslp_tsk	
タスクの起床	wup_tsk	tk_wup_tsk
タスクの起床(タスク独立部 [2])	iwup_tsk	
タスク起床要求のキャンセル	can_wup	tk_can_wup
待ち状態の強制解除	rel_wai	tk_rel_wai
待ち状態の強制解除(タスク独立部)	irel_wai	
強制待ち状態への移行	sus_tsk	tk_sus_tsk
強制待ち状態からの再開	rsm_tsk	tk_rsm_tsk
強制待ち状態からの強制再開	frsm_tsk	tk_frsm_tsk
自タスクの遅延	dly_tsk	tk_dly_tsk

[2] μITRON4.0 仕様では「非タスクコンテキスト」と呼びますが、本書ではμT-Kernel に合わせて「タスク独立部」としてあります。

2.3.1. 起床待ち(slp_tsk / tslp_tsk)

【移行のポイント】
（1）システムコールを tk_slp_tsk に統一する。
（2）引数にタイムアウト時間を指定する。

T-Kernel では待ち状態に入る可能性があるシステムコールには、タイムアウト指定の機能を持たせています。μT-Kernel では T-Kernel との互換性を優先し、起床待ちに tk_slp_tsk を用います。

tslp_tsk と tk_slp_tsk は同等の機能であり、システムコール名を置き換えるだけで移行できます。また、slp_tsk と tk_slp_tsk もほぼ同等の機能であり、タイムアウト指定に永久待ち(TMO_FEVR)を指定することで移行できます。

(A) slp_tsk から tk_slp_tsk へ

【移行の手順】
（1）システムコールを tk_slp_tsk に変更する。
（2）タイムアウト値に TMO_FEVR を追加する。

リスト 2-18　μITRON4.0 での起床待ち

```
ER  ercd;
ercd = slp_tsk();
```

リスト 2-19　μT-Kernel での起床待ち

```
ER  ercd;
ercd = tk_slp_tsk(TMO_FEVR);
```

【解説】

T-Kernel では待ち状態に入る可能性があるシステムコールには、タイムアウト指定の機能を持たせているため、タイムアウト指定に永久待ち(TMO_FEVR)を指定します。

(B) tslp_tsk から tk_slp_tsk へ

【移行の手順】
（1）システムコールを tk_slp_tsk に変更する。
（2）指定する待ち時間の範囲を確認する。
（3）システムのタイムティックを確認する。

リスト 2-20　μITRON4.0 での起床待ち（タイムアウトあり）

```
ER  ercd;
ercd = tslp_tsk(100);
```

リスト 2-21　μT-Kernel での起床待ち（タイムアウトあり）

```
ER  ercd;
ercd = tk_slp_tsk(100);
```

【解説】

tslp_tsk と tk_slp_tsk は引数の型と機能については同等であるため、システムコール名を置き換えるだけで移行できますが、μITRON4.0 とμT-Kernel ではタイムアウト指定の時間単位とビット幅が異なる可能性があるため、この点は注意が必要です。

引数のタイムアウト指定は tslp_tsk と tk_slp_tsk 共に TMO 型を使用します。TMO 型はμITRON4.0 では「符号付き整数」、μT-Kernel では「符号付き 32 ビット整数」と定義されています。

μT-Kernel の時間単位は仕様によってミリ秒と規定されていますが、μITRON4.0 の時間単位は実装定義であるため、移植元の時間単位がミリ秒ではない可能性があります。このような場合には、タイムアウト時間が同等になるように調整する必要があります。

また、システムによってタイムティックが異なっている場合があります。タイムティックの設定によっては動作が異なる場合がありますので、移植元と移植先のシステムのタイムティックを確認して、事前に調整しておく必要があります。

μT-Kernel ではデフォルトで 10 ミリ秒に設定されています。設定は以下のファイルに含まれている CFN_TIMER_PERIOD ですので、適宜調整してください。

utkernel_source/config/sysdepend/[TARGET][3]/utk_config_depend.h

[3] [TARGET] と記されたディレクトリは対象となるシステムのターゲットボード略称に置き換えてください。

2.3.2. タスクの起床(wup_tsk / iwup_tsk)

【移行のポイント】
（1）システムコールを tk_wup_tsk に統一する。
（2）起床するタスクに自タスクを指定できなくなった。

(A) wup_tsk から tk_wup_tsk へ

【移行の手順】
（1）システムコール名称を tk_wup_tsk に変更する。
（2）引数に自タスク(TSK_SELF)を指定している場合はセマフォなどで代替する。

リスト 2-22　μITRON4.0 でのタスクの起床

```
ER  ercd;
ID  task_id;

ercd = wup_tsk(task_id);
```

リスト 2-23　μT-Kernel でのタスクの起床

```
ER  ercd;
ID  task_id;

ercd = tk_wup_tsk(task_id);
```

【解説】

wup_tsk と tk_wup_tsk は同等の機能であり、システムコール名を置き換えるだけで移行できます。

また、wup_tsk では引数に自タスク(TSK_SELF)を指定することが可能でしたが、tk_wup_tsk では、これはエラー(E_OBJ)となります。この場合、次ページに示すようにセマフォなどを利用することで対応が可能です。

例として、ここでは、ある条件を判定し、条件が成立していれば処理を続行、成立していなければ成立するまで起床待ち状態を保つプログラムで説明します [4]。

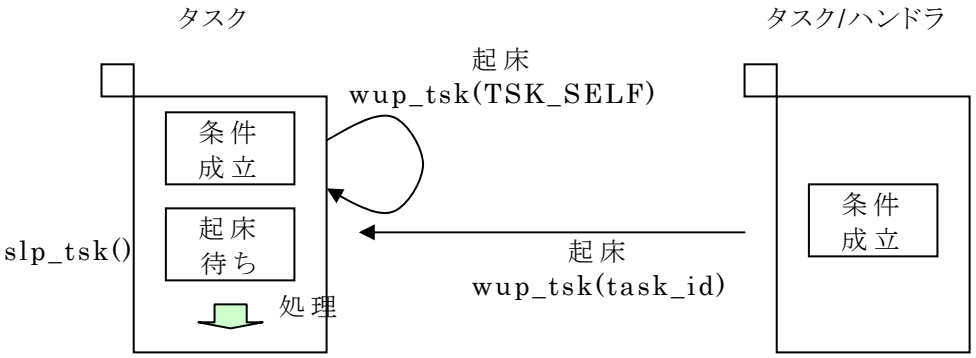


図 2-3 wup_tsk(TSK_SELF)を使った処理

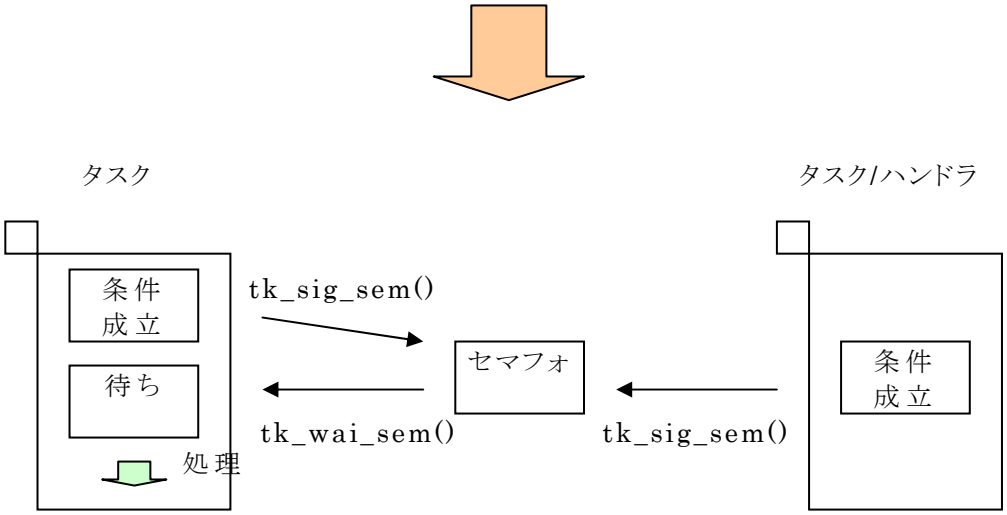


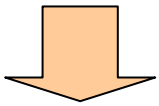
図 2-4 セマフォによる代替例

次ページにソースコード例を示します。なお、本例はセマフォで代替していますが、データの受渡しが発生する場合にはメールボックスを利用するなど、実際のプログラムに応じて適切なオブジェクトを選択してください。

[4] slp_tsk 実行時に自タスクに対する起床要求キューイング数が1以上の場合は、待ち状態に入らず実行を継続するという slp_tsk の仕様を利用しています。

リスト 2-24 wup_tsk(TSK_SELF)を使った処理例(μITRON4.0)

```
ID task_id;
...
void task(VP_INT exinf)
{
     ...
     if(条件) {
          wup_tsk(TSK_SELF);
     }
     slp_tsk();   //条件が成立している場合は待ち状態に入らない
     //何らかの処理
}

void handler(VP_INT exinf)
{
     ...
     if(条件) {
          wup_tsk(task_id);
     }
}
```



リスト 2-25 セマフォを利用した対応例(μT-Kernel)

```
ID sem_id;
...
void task(INT stacd, VP exinf)
{
     ...
     if(条件) {
          tk_sig_sem(sem_id, 1);
     }
     tk_wai_sem(sem_id, 1, TMO_FEVR);
     //何らかの処理
}

void handler(VP exinf)
{
     ...
     if(条件) {
          tk_sig_sem(sem_id, 1);
     }
}
```

(B) iwup_tsk から tk_wup_tsk へ

【移行の手順】

（1）システムコールを tk_wup_tsk に変更する。

リスト 2-26　μITRON4.0 でのタスクの起床

```
ER  ercd;
ercd = iwup_tsk( task_id );
```

リスト 2-27　μT-Kernel でのタスクの起床

```
ER  ercd;
ercd = tk_wup_tsk( task_id );
```

【解説】

tk_wup_tsk はタスク独立部およびディスパッチ禁止状態でも発行できるようになっていますので、システムコール名を置き換えるだけで移行できます。

また、システムコールの引数 task_id はタスク ID ですので、変更する必要はありません。

2.3.3. 強制待ち状態への移行(sus_tsk)

【移行の手順】

（1）システムコール名称を tk_sus_tsk に変更する。

（2）引数に自タスク(TSK_SELF)を指定している場合は他タスクでの実行などで代替する。

リスト 2-28　μITRON4.0 での強制待ち状態への移行

```
ER   ercd;
ID   task_id;

ercd = sus_tsk(task_id);
```

リスト 2-29　μT-Kernel での強制待ち状態への移行

```
ER   ercd;
ID   task_id;

ercd = tk_sus_tsk(task_id);
```

【解説】

sus_tsk と tk_sus_tsk は同等の機能であり、システムコール名を置き換えるだけで移行できます。

また、sus_tsk では引数に自タスク(自タスクの ID 番号、または TSK_SELF(=0))を指定することが可能でしたが、tk_sus_tsk では、これはエラー(E_OBJ、または E_ID)となります。この場合、次ページに示すように優先度の高いタスクなどを利用することで対応が可能です。

例として、ここでは、優先度の高いタスクを起動して、そのタスクから強制待ち状態に移行させるプログラムで説明します。

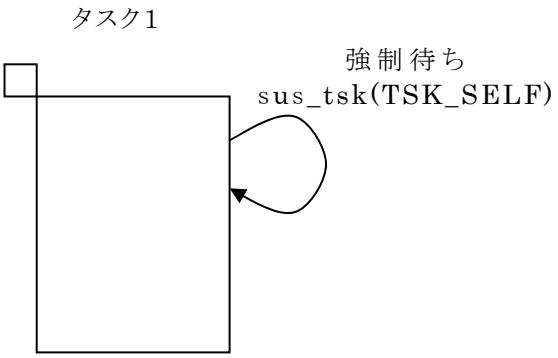


図 2-5 sus_tsk(TSK_SELF)を使った処理

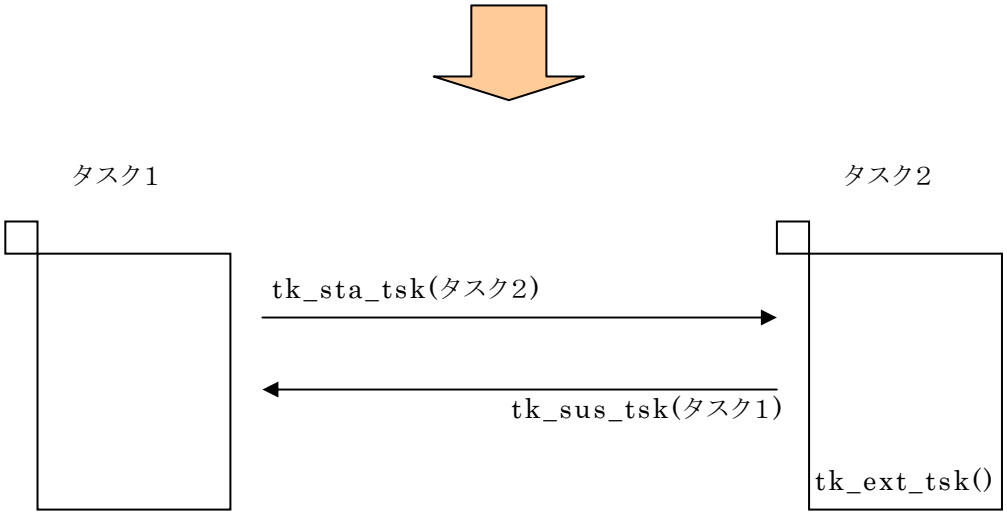
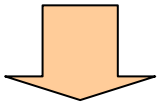


図 2-6 優先度の高いタスクによる代替例

次ページにソースコード例を示します。

リスト 2-30 sus_tsk(TSK_SELF)を使った処理例(μITRON4.0)

```
void task(VP_INT exinf)
{
    ...
    sus_tsk(TSK_SELF);
    ...
}
```



リスト 2-31 優先度の高いタスクを利用した対応例(μT-Kernel)

```
ID suspend_task_id;
...
void task(INT stacd, VP exinf)
{
    ...
    tk_sta_tsk(suspend_task_id, (INT)tk_get_tid());
    ...
}

void susupend_task(INT stacd, VP exinf)
{
    tk_sus_tsk((ID)stacd);
    tk_ext_tsk();
}
```

2.3.4. 強制待ち状態からの再開(rsm_tsk / frsm_tsk)

【移行の手順】

（1）システムコール名称を tk_rsm_tsk、tk_frsm_tsk に変更する。

リスト 2-32　μITRON4.0 での強制待ち状態からの再開

```
ER  ercd;
ID  task_id;

ercd = rsm_tsk(task_id);
```

リスト 2-33　μT-Kernel での強制待ち状態からの再開

```
ER  ercd;
ID  task_id;

ercd = tk_rsm_tsk(task_id);
```

【解説】

rsm_tsk と tk_rsm_tsk は同等の機能であり、システムコール名を置き換えるだけで移行できます。

また、frsm_tsk も同様に tk_frsm_tsk に移行できます。

2.3.5. 自タスクの遅延(dly_tsk)

【移行のポイント】

（1）待ち時間の変数のビット幅に注意する。

（2）待ち時間=ポーリングの時の動作の違いに注意する。

dly_tskとtk_dly_tskは、ほぼ同等の機能であり、システムコール名を置き換えるだけで移行できます。ただし、自タスクの遅延時間(dlytim)が0の場合は、動作が異なるので注意が必要です。

引数の待ち時間の指定はRELTIM型を使用します。RELTIM型はμITRON4.0では「符号無し整数」、μT-Kernelでは「符号無し32ビット整数」と定義されています。

μITRONではRELTIM型のビット幅が32ビットを超えている可能性があります。この場合は調整が必要となります。

(A) 待ち時間が 1 以上の場合

【移行の手順】

(1) システムコール名を tk_dly_tsk に変更する。

(2) 指定する待ち時間の範囲を確認する。

(3) システムのタイムティックを確認する。

リスト 2-34 μITRON4.0 での自タスクの遅延(遅延時間が 1 以上の場合)

```
ER  ercd;
ercd = dly_tsk(100);
```

リスト 2-35 μT-Kernel での自タスクの遅延(遅延時間が 1 以上の場合)

```
ER  ercd;
ercd = tk_dly_tsk(100);
```

【解説】

待ち時間として1以上の値が指定されている場合はシステムコールの名称を変更することで対応できます。

μT-Kernel の時間単位は仕様によってミリ秒と規定されていますが、μITRON4.0 の時間単位は実装定義であるため、移植元の時間単位がミリ秒ではない可能性があります。このような場合には、待ち時間が同等になるように調整する必要があります。

また、システムによってタイムティックが異なっている場合があります。タイムティックの設定によっては動作が異なる場合がありますので、移植元と移植先のシステムのタイムティックを確認して、事前に調整しておく必要があります。

μT-Kernel ではデフォルトで 10 ミリ秒に設定されています。設定は以下のファイルに含まれている CFN_TIMER_PERIOD ですので、適宜調整してください。

utkernel_source/config/sysdepend/[TARGET][5]/utk_config_depend.h

[5] [TARGET] と記されたディレクトリは対象となるシステムのターゲットボード略称に置き換えてください。

(B) 待ち時間が 0 の場合(ポーリング)

【移行の手順】

(1) システムコール名を tk_dly_tsk に変更する。
または、tk_rot_rdq など別のシステムコールの利用を検討する。

(2) 挙動が変わることによる影響の有無を確認する。
dly_tsk(0)…待ち状態に入る
tk_dly_tsk(0)…待ち状態には入らない

リスト 2-36 μITRON4.0 での自タスクの遅延(遅延時間が 0 の場合)

```
ER  ercd;
ercd = dly_tsk( 0 );
```

リスト 2-37 μT-Kernel での実装例(待ちを期待する場合)

```
ER  ercd;
ercd = tk_dly_tsk( 1 );
```

待ちの発生を期待する場合は、遅延時間を 1 に変更する。

リスト 2-38 μT-Kernel での実装例(実行権を放棄する場合)

```
ER  ercd;
ercd = tk_rot_rdq(TPRI_RUN);
```

【解説】

リスト 2-27 ではシステムコールの名称と遅延時間を変更しました。一般にはこの変更でも正常に動作すると考えられますが、リスト 2-26 と 2-27 とではタイムアウトが発生するタイミングが異なります(詳細は後述)。元のプログラムにおいてこの部分でどのような動作になることを期待しているのかを確認したうえで移植する必要があります。

リスト 2-28 では、タスクにいったん実行権を放棄させるためにタスクの優先順位の回転 tk_rot_rdq を利用しています。このシステムコールを発行した場合、同一優先度の実行可能状態のタスクが存在すれば、実行状態のタスクが同一優先度の待ち行列の最後尾につながれることになります。リスト 2-26 のコードが、遅延の発生ではなく、実行権の放棄を目的としている場合には、リスト 2-28 のように変更することができます。ただし、同一優先度の実行可能状態のタスクが存在しない場合は何もせず実行を継続することになりますので、μITRON4.0 の dly_tsk(0)とは異なった挙動になりますので、注意が必要です。

この他に、待ち時間(dly_tsk のパラメータ)を変数にして動的に変更しているような実装の場合は、それぞれのパラメータでどのように動作することを期待していたかをよく検討してから移植する必要があります。

【遅延時間=0の動作の違いについて】

μITRON4.0仕様では遅延時間に0を指定した場合も自タスクを待ち状態に移行させます。μITRON4.0仕様には以下のように記載されています。

> 「このサービスコールは，dlytimに0が指定された場合にも，自タスクを待ち状態に移行させる．」
>
> （μITRON4.0仕様 「dly_tsk 自タスクの遅延」より）
>
> 「相対時間に0が指定されると，サービスコールが呼出された後の，最初のタイムティックでイベント処理を行う．」
>
> （μITRON4.0仕様 「2.1.9 相対時間とシステム時刻」より）

これに対して、μT-Kernelでは遅延時間に0を指定した場合は、待ち状態に移行せずに処理を継続します（μT-Kernelの待ち状態に移行する可能性がある全てのシステムコールにおいて共通）。

また、T-Kernel、μITRON4.0共に、タイムアウトとして1以上の値が指定された場合は、指定された以上の時間が経過した後にタイムアウトが発生することを保証しなければなりません。このため、タイムアウト値として1以上の値が指定された場合は、最低でも1タイムティック分の待ちが発生することになります。

以上をまとめると下図のようになります。

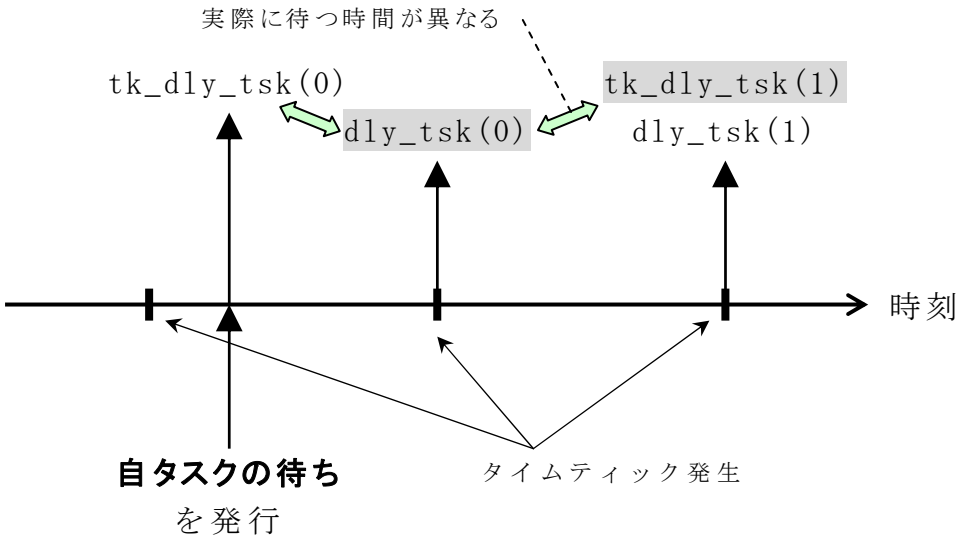


図 2-7 タイムアウトが発生するタイミング

dly_tsk(0)を発行してからタイムアウトが発生するまでの時間は一定しませんので、どの程度の待ちが発生するかは不定です。ただし、システムコールを発行したタスクがいったん待ち状態になりますので、他のタスクに実行権が移ることもあります。このような機能を利用するためにdly_tsk(0)で実装しているプログラムでは、tk_rot_rdq(TPRI_RUN)に変更するなどの調整が必要となります。

2.4. 同期・通信機能

【同期・通信機能】

主な変更点

オブジェクト	変更点
セマフォ	資源数の複数返却・確保
イベントフラグ	解除条件に一致したビットのみ 0 クリア可能
データキュー	廃止。メールボックスの機能で一部代替可能
メールボックス	優先度別のメッセージキューヘッダ領域の廃止

【解説】

同期・通信機能は、μT-Kernel ではデータキューが廃止されています。

データキューについては、メールボックスを利用することで一部代替が可能です。

2.4.1. 同期・通信機能（セマフォ）

(A)機能一覧

セマフォの機能は次の通りです。μITRON4.0 から機能の追加・変更はありません。

（1）セマフォの生成
（2）セマフォの削除
（3）セマフォ資源の返却
（4）セマフォ資源の獲得
（5）セマフォの状態参照

(B) μITRON4.0 とμT-Kernel のシステムコールの対応

μITRON4.0 とμT-Kernelのシステムコールの対応は、表 2-4 の通りです。

表 2-4　μITRON4.0 と T-Kernel のシステムコールの対応

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

(C) μT-Kernel での変更点と追加機能

μT-Kernel での変更点と追加機能を示します。

表 2-5 変更点

項目	関連する システムコール	備考
システムコール名称が変更された。	全システムコール	
セマフォ ID が自動割り当てとなった。	CRE_SEM cre_sem	§2.4.1.1 参照
静的 API が廃止された。	CRE_SEM	§2.4.1.1 参照
資源の獲得/返却個数 1 固定(指定不可) →任意の個数(指定要)となった。	sig_sem isig_sem wai_sem pol_sem twai_sem	§2.4.1.3 参照 §2.4.1.4 参照
セマフォ生成情報パケットの構成が変更された。	CRE_SEM cre_sem acre_sem	§2.4.1.1 参照
セマフォ状態パケットの構成が変更された。	ref_sem	§2.4.1.5 参照

表 2-6 追加機能

項目	備考
セマフォ属性に資源獲得の優先順の属性指定が追加された。	§2.4.1.1 参照
拡張情報 exinf が追加された。	§2.1.1.5 参照
デバッグサポート機能 dsname が追加された。	§2.1.1.6 参照

2.4.1.1. セマフォの生成（CRE_SEM / cre_sem / acre_sem）

【移行のポイント】

（1）システムコール名称が変更された。

（2）セマフォ ID が自動割り当てとなった。

（3）静的 API が廃止された。

（4）セマフォ生成情報パケットの構成が変更された。
isemcnt と maxsem の型が変更された(UINT→INT)

（5）カーネル構成定数 TMAX_MAXSEM が廃止された。

T-Kernel はプログラムの動的なロードを前提に設計されており、セマフォは動的に生成して ID も自動的に割り付けることになっています。μT-Kernel では T-Kernel との互換性を優先し、セマフォの生成には tk_cre_sem を用います。tk_cre_sem はμITRON4.0 の acre_sem とほぼ等価のシステムコールですので、あまり変更せずにμT-Kernel に移行できます。他のシステムコール(CRE_SEM/cre_sem)は移行に際して調整が必要になります。

(A) acre_sem から tk_cre_sem へ

【移行の手順】

(1) システムコールを tk_cre_sem に変更する。

(2) セマフォ属性(sematr)に TA_FIRST を追加する。

(3) isemcnt と maxsem の型を変更する。

リスト 2-39 μITRON4.0 でのセマフォの生成 (acre_sem)

```
T_CSEM   pk_csem;
ID       sem_id;

     :

pk_csem.sematr   = TA_TFIFO;
pk_csem.isemcnt  = 0U;
pk_csem.maxsem   = 10U;

sem_id = acre_sem(&pk_csem);

     :
```

リスト 2-40 μT-Kernel でのセマフォの生成

```
T_CSEM   pk_csem;
ID       sem_id;

     :
pk_csem.sematr   = TA_TFIFO | TA_FIRST;
pk_csem.isemcnt  = 0;
pk_csem.maxsem   = 10;

sem_id = tk_cre_sem(&pk_csem);

     :
```

【解説】

acre_sem と tk_cre_sem は、ほぼ同等の機能です。

セマフォ属性を追加することで移行できます。

TA_FIRST は、待ち行列先頭のタスクを優先して資源を割り当てるというセマフォ属性です。セマフォ資源の要求数に関係なく、待ち行列先頭のタスクから順に資源を割り当てていきます。

isemcnt と maxsem の型が UINT から INT に変更されています。maxsem に指定できる最大値はμITRON4.0 とμT-Kernel 共に実装定義です。μT-Kernel の仕様では、maxsem に 32768 以上の値が指定できることを保証していないため、移植元のプログラムが maxsem に 32768 以上の値を指定している場合は実装仕様書を確認する必要があります。

(B) cre_sem から tk_cre_sem へ

【移行の手順】

（1）システムコールを tk_cre_sem に変更する。

（2）セマフォ属性(sematr)に TA_FIRST を追加する。

（3）isemcnt と maxsem の型を変更する。

（4）セマフォ ID を自動割当てに変更する。

【セマフォ ID を変数に格納している場合】

リスト 2-41　μITRON4.0 でのセマフォの生成　（cre_sem）

```
T_CSEM   pk_csem;
ER       ercd;
ID       sem_id;
     :
sem_id = 10;
     :
pk_csem.sematr   = TA_TFIFO;
pk_csem.isemcnt  = 0U;
pk_csem.maxsem   = 10U;

ercd = cre_sem(sem_id, &pk_csem);

     :
```

リスト 2-42　μT-Kernel でのセマフォの生成

```
T_CSEM   pk_csem;
ID       sem_id;

     :
pk_csem.sematr   = TA_TFIFO | TA_FIRST;
pk_csem.isemcnt  = 0;
pk_csem.maxsem   = 10;

sem_id = tk_cre_sem(&pk_csem);

     :
```

【解説】

μT-Kernel ではセマフォ ID を自動で割り付けるので、tk_cre_sem が返す値をセマフォ ID 変数に保存し、後で利用できるようにする処理を追加します。

μITRON では cre_sem を使用する場合、セマフォ ID をアプリケーション側で指定します。即値を指定できるので、システムの起動時からセマフォを個別に識別できるといったメリットがありますが、システム開発する段階で、セマフォ ID が一意になるように開発者が調整しておかなければなりません。一方の μT-Kernel ではセマフォ ID は自動的に割当てられますので、開発者が割り振る必要はありません。

【セマフォ ID をマクロ定義している場合】

リスト 2-43 マクロによるセマフォ ID の指定(μITRON4.0)

```
#define SEMID_CLIENT      (10)
        :
    T_CSEM   pk_csem;
    ER       ercd;
        :
    pk_csem.sematr   = TA_TFIFO;
    pk_csem.isemcnt  = 0U;
    pk_csem.maxsem   = 10U;

    ercd = cre_sem(SEMID_CLIENT, &pk_csem);

        :
```

リスト 2-44 外部変数によるセマフォ ID の指定

```
static  ID                  semid_client = 0;
#define SEMID_CLIENT        semid_client
        :
    T_CSEM   pk_csem;
        :
    pk_csem.sematr   = TA_TFIFO | TA_FIRST;
    pk_csem.isemcnt  = 0;
    pk_csem.maxsem   = 10;

    semid_client = tk_cre_sem(&pk_csem);

        :
```

【解説】

μITRON では固定のセマフォ ID をマクロとして定義する場合が多くあります。マクロとして定義した場合、マクロを指定することでシステム全体でセマフォを一意に特定できるといったメリットがあります。μT-Kernel ではこの方法は使えませんので、移行に際してはグローバル変数を用意して、その変数に生成したセマフォのセマフォ ID を指定するといった実装が必要となります。

上記のリストでは、μITRON4.0 も T-Kernel も SEMID_CLIENT というマクロを定義してあります。このようにすることで例えば、セマフォ ID を指定する必要のあるシステムコールの移行においてパラメータを変更する必要がなくなります。

なお、セマフォ ID を格納する変数 semid_client を直接 SEMID_CLIENT とすればマクロを定義する必要はなくなります。ただ一般には、マクロは大文字、変数名は小文字を利用することが多いようですので、上記リストのようにしました。実際の移行に際しては、各社のコーディングルールに適合するように調整してください。

(C) CRE_SEM から tk_cre_sem へ

【移行の手順】

（1）システムコールを tk_cre_sem に変更する。

（2）セマフォ属性(sematr)に TA_FIRST を追加する。

（3）isemcnt と maxsem の型を変更する。

（4）セマフォ ID を自動割当てに変更する。

（5）初期タスク内にソースコードを追加する。

前述したとおり、T-Kernel はプログラムの動的なロードを前提としているため、静的 API をサポートしていません。μT-Kernel でも同様のため、μITRON4.0 でサポートしている静的 API を使用している場合は、動的生成 tk_cre_sem に置き換えます。

リスト 2-45 μITRON4.0 でのセマフォの生成 （CRE_SEM）

```
CRE_SEM(sem_id, { TA_TFIFO, 0U, 10U } );
```

リスト 2-46 μT-Kernel でのセマフォの生成

```
T_CSEM  pk_csem;
ID      sem_id;

    :
pk_csem.sematr  = TA_TFIFO | TA_FIRST;
pk_csem.isemcnt = 0;
pk_csem.maxsem  = 10;

sem_id = tk_cre_sem(&pk_csem);

    :
```

【解説】

μITRON4.0 の静的 API である CRE_SEM の場合、開発者はシステムコンフィギュレーションファイルに静的に生成したいセマフォの数だけ静的 API を記述し、そのファイルをコンフィギュレータに通すことでセマフォが静的に生成されます。μT-Kernel では静的 API の仕様そのものがないため、tk_cre_sem を呼び出すだけでセマフォを生成することができます。静的 API を使用した場合、システムが起動した段階で既にセマフォが生成されています。μT-Kernel の場合、初期タスク内で tk_cre_sem を呼び出すことで同じような振る舞いになります。

セマフォ生成のソースコードは、アプリケーションが動きだすまでに実行される位置に記述しなければなりません。

2.4.1.2. セマフォの削除（del_sem）

【移行の手順】

（1）システムコールを tk_del_sem に変更する。

リスト 2-47　μITRON4.0 でのセマフォの削除

```
ER  ercd;
    :
ercd = del_sem(sem_id);

    :
```

リスト 2-48　μT-Kernel でのセマフォの削除

```
ER  ercd;
    :
ercd = tk_del_sem(sem_id);

    :
```

【解説】

セマフォの削除はシステムコール名を置き換えるだけで移行できます。

2.4.1.3. セマフォ資源の返却（sig_sem / isig_sem）

【移行の手順】

（1）システムコールを tk_sig_sem に統一する。

（2）資源返却数(=1)をパラメータに追加する。

リスト 2-49　μITRON4.0 でのセマフォ資源の返却

```
ER  ercd;
    :
ercd = sig_sem(sem_id);

    :
```

リスト 2-50　μT-Kernel でのセマフォ資源の返却

```
ER  ercd;
    :
ercd = tk_sig_sem(sem_id, 1);

    :
```

【解説】

sig_sem および isig_sem は tk_sig_sem とほぼ同等の機能であり、返却するセマフォ資源数である第 2 引数に 1 を指定することで移行できます。

μITRON4.0 では sig_sem で一度に返却できる最大セマフォ資源数は 1 であり、バイナリ・セマフォに近い振る舞いでした。μT-Kernel では一度に複数のセマフォ資源を返却することができる汎用セマフォの振る舞いになっています。

2.4.1.4. セマフォ資源の獲得（wai_sem / pol_sem / twai_sem）

【移行のポイント】

（1）システムコールを tk_wai_sem に統一する。

（2）資源獲得数をパラメータに追加する。

（3）ポーリングの動作の違いに注意する(返値の違い)。

T-Kernel では待ち状態に入る可能性があるシステムコールには、ポーリングおよびタイムアウト指定の機能を持たせています。μT-Kernel では T-Kernel との互換性を優先し、セマフォ資源の獲得に tk_wai_sem を用います。wai_sem および pol_sem は tk_wai_sem のタイムアウト指定に永久待ち(TMO_FEVR)またはポーリング(TMO_POL)を指定することで移行できます。また twai_sem は tk_wai_sem と同等の機能であり、システムコール名を置き換えることで移行できます。

(A) wai_sem から tk_wai_sem へ

【移行の手順】

（1）システムコールを tk_wai_sem に変更する。

（2）資源獲得数(=1)をパラメータに追加する。

（3）タイムアウト値に TMO_FEVR を設定する。

リスト 2-51　μITRON4.0 でのセマフォ資源の獲得

```
ER   ercd;
     :
ercd = wai_sem(sem_id);

     :
```

リスト 2-52　μT-Kernel でのセマフォ資源の獲得

```
ER   ercd;
     :
ercd = tk_wai_sem(sem_id, 1, TMO_FEVR);

     :
```

【解説】

T-Kernel では待ち状態に入る可能性があるシステムコールには、タイムアウト指定の機能を持たせているため、タイムアウト指定に永久待ち(TMO_FEVR)を指定することで移行できます。

(B) pol_sem から tk_wai_sem へ

【移行の手順】
（1）システムコールを tk_wai_sem に変更する。
（2）資源獲得数(=1)をパラメータに追加する。
（3）タイムアウト値に TMO_POL を設定する。

リスト 2-53　μITRON4.0 でのセマフォ資源の獲得（ポーリング）

```
ER  ercd;
    :
ercd = pol_sem(sem_id);
    :
```

リスト 2-54　μT-Kernel でのセマフォ資源の獲得（ポーリング）

```
ER  ercd;
    :
ercd = tk_wai_sem(sem_id, 1, TMO_POL);
    :
```

【解説】

pol_sem と tk_wai_sem はほぼ同等の機能であり、タイムアウト指定にポーリング（TMO_POL）を指定することで移行できます。

【実装に依存する点】

タスク独立部またはディスパッチ禁止状態で pol_sem, twai_sem(TMO_POL)を発行した場合、実装によって動作が異なる可能性があるので注意が必要です。

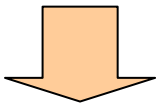
	タスク独立部または ディスパッチ禁止状態での発行	
	資源あり	資源なし
pol_sem()	E_OK	E_TMOUT
twai_sem(TMO_POL)	E_OK ｜ E_CTX	E_TMOUT ｜ E_CTX
tk_wai_sem(TMO_POL)	E_CTX	E_CTX

例えば、ハンドラ内で pol_sem を呼び出している場合、tk_wai_sem に置き換えると E_CTX が返ります。このようにタスク独立部で pol_sem を使用している場合には、リスト 2-55, 56 に示すように、新たにタスクを作成し、タスク内で tk_wai_sem を呼び出すなどの変更が必要となります。

リスト 2-55 タスク独立部で pol_sem を使っている場合の対応例(μITRON4.0)

■μITRON4.0 のソースコード例

```
ID semid;
...
void handler(VP_INT exinf)
{
    ER ercd;
    ...
    ercd = pol_sem(semid);
    if(ercd == E_OK){
        ...
    }
}
```



リスト 2-56 タスク独立部で pol_sem を使っている場合の対応例(μT-Kernel)

■μT-Kernel への移行例

```
ID semid;
ID tskid;
...
void handler(VP exinf)
{
    tk_wup_tsk(tskid);
}

void task(INT stacd, VP exinf)
{
    ER ercd;
    ...
    tk_slp_tsk(TMO_FEVR);
    ercd = tk_wai_sem(semid, 1, TMO_POL);
    if(ercd == E_OK) {
        ...
    }
    ...
}
```

(C) twai_sem から tk_wai_sem へ

【移行の手順】

（1）システムコールを tk_wai_sem に変更する。

（2）資源獲得数(=1)をパラメータに追加する。

リスト 2-57　μITRON4.0 でのセマフォ資源の獲得（タイムアウトあり）

```
ER   ercd;
     :
ercd = twai_sem(sem_id, 100);
     :
```

リスト 2-58　μT-Kernel でのセマフォ資源の獲得（タイムアウトあり）

```
ER   ercd;
     :
ercd = tk_wai_sem(sem_id, 1, 100);
     :
```

【解説】

twai_sem と tk_wai_sem は同等の機能であり、獲得するセマフォ資源数である第 2 引数に 1 を指定することで移行できます。

μITRON4.0 では twai_sem で一度に獲得できる最大セマフォ資源数は 1 であり、バイナリ・セマフォに近い振る舞いでした。μT-Kernel では一度に複数のセマフォ資源を獲得することができる汎用セマフォの振る舞いになっています。

2.4.1.5. セマフォの状態参照（ref_sem）

【移行のポイント】

（1）システムコール名称を変更する。

（2）semcnt の型を UINT→INT に変更する。

（3）「待ちタスクなし」を示すマクロ TSK_NONE(=0)が削除されているので定義する。

（4）T_RSEM 構造体のメンバ変数を変更する。

wtskid→wtsk(待ちタスクの有無またはタスク ID)

リスト 2-59　μITRON4.0 でのセマフォの状態参照

```
T_RSEM pk_rsem;
ER     ercd;
    :
ercd = ref_sem(sem_id, &pk_rsem);
if(pk_rsem.wtskid == TSK_NONE){
        :
}
```

リスト 2-60　μT-Kernel でのセマフォの状態参照

```
#define TSK_NONE    0
T_RSEM pk_rsem;
ER     ercd;
    :
ercd = tk_ref_sem(sem_id, &pk_rsem);
if(pk_rsem.wtsk == TSK_NONE) {
    :
}
```

2.4.1.6. 追加機能

μT-Kernel では次の機能が追加されました。ここでは、セマフォ資源の獲得/返却個対応について、次項で詳しく説明します。

【追加機能】

（1）セマフォ資源の獲得/返却の複数個対応

（2）拡張情報 exinf

（3）デバッガサポート機能 dsname

2.4.1.7. セマフォ資源の獲得/返却の複数個対応について

【概要】

セマフォ資源を複数個獲得/返却できる機能を追加したことに伴い、資源獲得待ちタスクの解除方法が次の 2 つになりました。

- 先頭のタスクから順に解除(従来)
- 資源を獲得できるタスクから順に解除

セマフォについてはμITRON4.0 仕様から大きく拡張されています。

μITRON4.0 仕様では、複数のセマフォ資源数を管理することはできますが、獲得する場合は 1 つずつ獲得するようになっています。それに対して、μT-Kernel 仕様では、複数のセマフォ資源数を管理することができ、かつ複数のセマフォ資源を一度に獲得できるようになっています。μT-Kernel では、下図のように一度に複数の待ちを解除するということもできます。

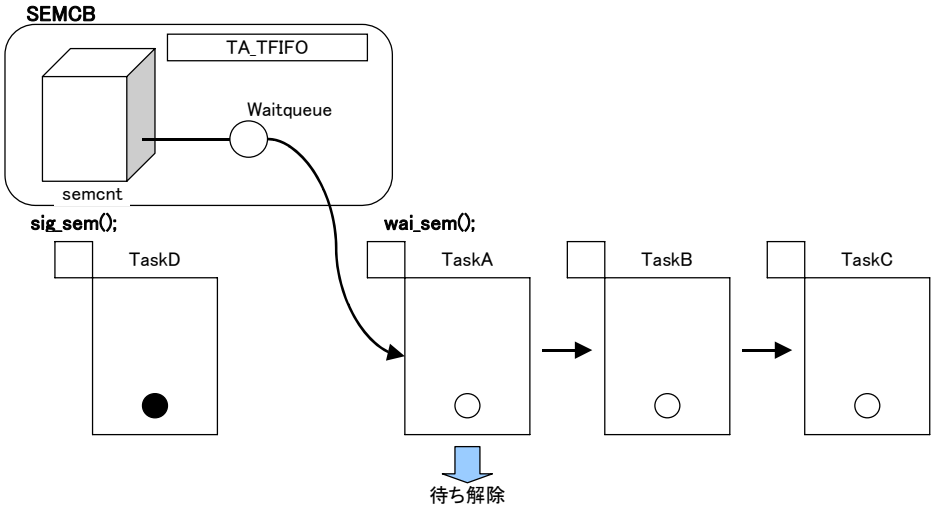


図 2-8　μITRON4.0 のセマフォ管理機能

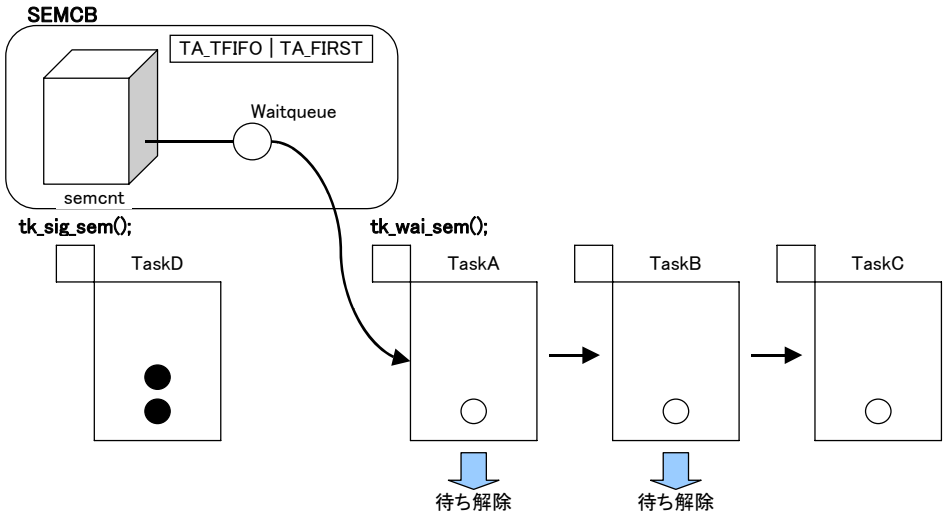


図 2-9　μT-Kernel のセマフォ管理機能

複数のセマフォ資源の獲得ができるようになったため、セマフォ属性にも変更があり、TA_FIRST、TA_CNT という資源獲得の優先順に関する新しい属性が追加されています。

- TA_FIRST…要求するセマフォ資源数に関係なく待ち行列の先頭タスクから順に解除されます。
- TA_CNT…要求したセマフォ資源数を獲得できるタスクから順に解除されます。

具体的な例を示します。

［例］

条件：

- 現在のセマフォ資源数…0

(1) セマフォ属性(TA_TFIFO | TA_FIRST)の場合

1. 獲得：TaskA が cnt=4 で tk_wai_sem()を発行する。 → 待ち状態に移行
2. 獲得：TaskB が cnt=2 で tk_wai_sem()を発行する。 → 待ち状態に移行
3. 獲得：TaskC が cnt=1 で tk_wai_sem()を発行する。 → 待ち状態に移行
4. 返却：TaskD が cnt=2 で tk_sig_sem()を発行する。

TA_FIRST で生成されたセマフォでは TaskA、TaskB、TaskC のいずれも待ち状態は解除されません。4.の時点でセマフォ資源数が 2 になりますが、TaskA の要求数よりも小さいため待ち解除されません。

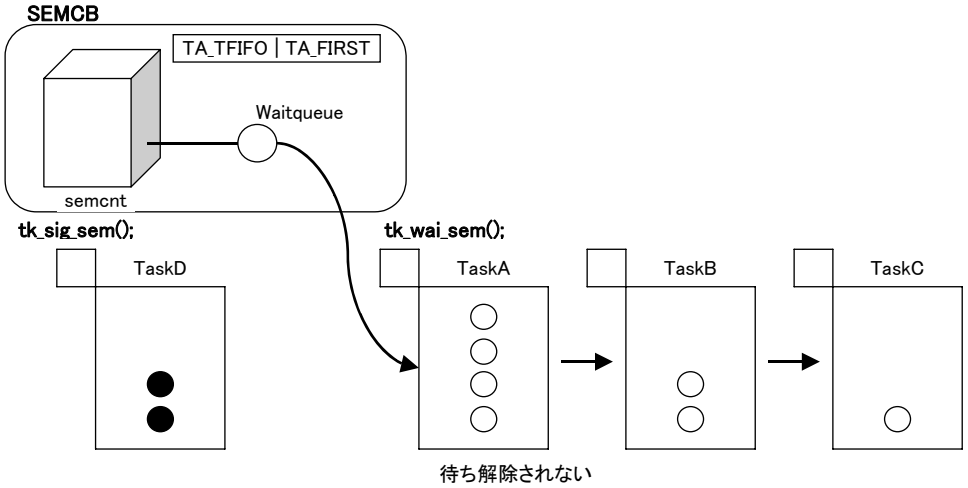


図 2-10 待ち行列先頭のタスクを優先(TA_FIRST 属性)

(2) セマフォ属性(TA_TFIFO | TA_CNT)の場合

1. 獲得: TaskA が cnt=4 で tk_wai_sem()を発行する。 → 待ち状態に移行
2. 獲得: TaskB が cnt=2 で tk_wai_sem()を発行する。 → 待ち状態に移行
3. 獲得: TaskC が cnt=1 で tk_wai_sem()を発行する。 → 待ち状態に移行
4. 返却: TaskD が cnt=2 で tk_sig_sem()を発行する。 → TaskB が待ち解除

TA_CNT で生成されたセマフォでは TaskB が待ち解除され、その他のタスクは待ち解除されません。4.の時点でセマフォ資源数が 2 になり、資源が返却された時点で待ち行列の先頭から検索して、最初に現れた、現在の資源数でサービスすることができるタスク(=TaskB)の待ち状態を解除します。

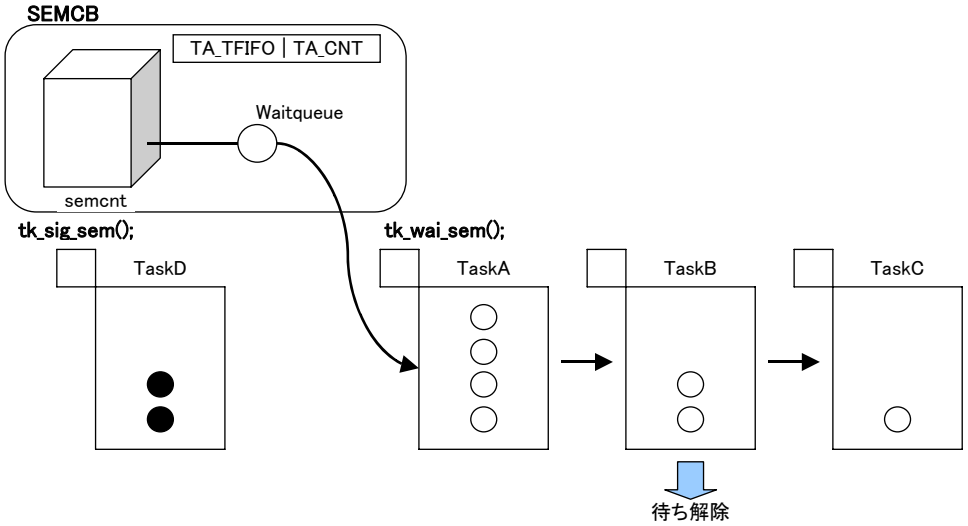


図 2-11 要求数の少ないタスクを優先(TA_CNT 属性)

2.4.2. 同期・通信機能（メールボックス）

（A）機能一覧

メールボックスの機能は次の通りです。μITRON4.0 仕様から機能の追加・変更はありません。

（1）メールボックスの生成
（2）メールボックスの削除
（3）メールボックスへの送信
（4）メールボックスからの受信
（5）メールボックスの状態参照

（B） μITRON4.0 とμT-Kernel のシステムコールの対応

μITRON4.0 のシステムコールとμT-Kernelのシステムコールの対応は 表 2-7 の通りです。

表 2-7 μITRON4.0 と T-Kernel のシステムコールの対応

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

(C) μT-Kernel での変更点と追加機能

μT-Kernel での変更点と追加機能を示します。

表 2-8 変更点

項目	関連する システムコール	備考
システムコール名称が変更された。	全システムコール	
メールボックス ID が自動割り当てとなった。	CRE_MBX cre_mbx	§2.4.2.1 参照
静的 API が廃止された。	CRE_MBX	§2.4.2.1 参照
TSZ_MPRIHD のマクロが廃止された。	マクロ	
メッセージ優先度の最大値(TMAX_MPRI)の値が変更された。	CRE_MBX cre_mbx acre_mbx snd_mbx	§2.4.2.1 参照 §2.4.2.3 参照
割込みハンドラにおけるメールボックスの受信(ポーリング)が変更された。	prcv_mbx	§2.4.2.4 参照
メールボックス生成情報パケットの構成が変更された。	CRE_MBX cre_mbx acre_mbx	§2.4.2.1 参照
メールボックス状態パケットの構成が変更された。	ref_mbx	§2.4.2.5 参照

表 2-9 追加機能

項目	備考
拡張情報 exinf が追加された。	§2.1.1.5 参照
デバッグサポート機能 dsname が追加された。	§2.1.1.6 参照

2.4.2.1. メールボックスの生成（CRE_MBX / cre_mbx / acre_mbx → tk_cre_mbx）

【移行のポイント】

（1）システムコール名称が変更された。

（2）メールボックス ID が自動割り当てとなった。

（3）静的 API が廃止された。

（4）メールボックス生成情報パケットが変更された。

maxmpri と mprihd が廃止された。

exinf と dsname が追加された。

（5）メッセージ優先度の最大値（TMAX_MPRI）の値が変更された。

μT-Kernel はプログラムの動的なロードを前提に設計されており、メールボックスは動的に生成して ID も自動的に割り付けることになっています。μT-Kernel では T-Kernel との互換性を優先し、メールボックスの生成には tk_cre_mbx を用います。tk_cre_mbx は μITRON4.0 の acre_mbx とほぼ等価のシステムコールですので、あまり変更せずに μT-Kernel に移行できます。他のシステムコール(CRE_MBX / cre_mbx)は移行に際して調整が必要になります。

(A) acre_mbx から tk_cre_mbx へ

【移行の手順】

(1) システムコール名を tk_cre_mbx に変更する。

(2) maxmpri と mprihd を利用しない。

リスト 2-61　μITRON4.0 でのメールボックスの生成　(acre_mbx)

```
T_CMBX   pk_cmbx;
ID       mbx_id;
VP       msgque;
     :
pk_cmbx.mbxatr   = TA_TFIFO | TA_MFIFO;
pk_cmbx.maxmpri  = 1;
pk_cmbx.mprihd   = (VP)msgque;
     :
mbx_id = acre_mbx(&pk_cmbx);

     :
```

リスト 2-62　μT-Kernel でのメールボックスの生成

```
T_CMBX   pk_cmbx;
ID       mbx_id;
     :
pk_cmbx.mbxatr  = TA_TFIFO | TA_MFIFO;
     :
mbx_id = tk_cre_mbx(&pk_cmbx);

     :
```

【解説】

acre_mbx と tk_cre_mbx は、ほぼ同等の機能です。

構造体 T_CMBX のメンバーを削除することで移行できます。

μT-Kernel では、構造体 T_CMBX のメンバーである送信されるメッセージの優先度の最大(maxmpri)、優先度別のメッセージキューヘッダ領域の先頭番地(mprihd)が削除されています。μITRON4.0 では、メッセージ優先度ごとにメッセージキューを持つ実装が許されていましたが、管理領域が増えてメモリ使用量が増大するため、μT-Kernel では 1 つのメッセージキューで管理するように改められました。

μITRON4.0 では、タスク以外のオブジェクトに拡張情報の機能はありませんでしたが、μT-Kernel ではタスク以外のオブジェクトに関しても拡張情報機能があります。

(B) cre_mbx から tk_cre_mbx へ

【移行の手順】

（1） システムコール名を tk_cre_mbx に変更する。

（2） maxmpri と mprihd を利用しない。

（3） メールボックス ID を自動割当てに変更する。

リスト 2-63　μITRON4.0 でのメールボックスの生成　（cre_mbx）

```
T_CMBX   pk_cmbx;
ER       ercd;
ID       mbx_id;
VP       msgque;
     :
mbx_id = 10;
     :
pk_cmbx.mbxatr   = TA_TFIFO | TA_MFIFO;
pk_cmbx.maxmpri  = 1;
pk_cmbx.mprihd   = (VP)msgque;
     :
ercd = cre_mbx(mbx_id, &pk_cmbx);

     :
```

リスト 2-64　μT-Kernel でのメールボックスの生成

```
T_CMBX   pk_cmbx;
ID       mbx_id;
     :
pk_cmbx.exinf    = (VP)1;
pk_cmbx.mbxatr   = TA_TFIFO | TA_MFIFO;
     :
mbx_id = tk_cre_mbx(&pk_cmbx);

     :
```

【解説】

T-Kernel ではメールボックス ID を自動で割り付けるので、tk_cre_mbx が返す値をメールボックス ID 変数に保存し、後で利用できるようにする処理を追加します。

μITRON では cre_mbx を使用する場合、メールボックス ID をアプリケーション側で指定します。即値を指定できるので、システムの起動時からメールボックスを個別に識別できるといったメリットがありますが、システム開発する段階で、メールボックス ID が一意になるように開発者が調整しておかなければなりません。一方のμT-Kernel ではメールボックス ID は自動的に割当てられますので、開発者が割り振る必要はありません。

【メールボックス ID をマクロ定義している場合】

リスト 2-65　マクロによるメールボックス ID の指定（μITRON4.0）

```
#define  MBXID_CLIENT      (10)
        :
    T_CMBX   pk_cmbx;
    ER       ercd;
        :
    pk_cmbx.mbxatr   = TA_TFIFO | TA_MFIFO;
    pk_cmbx.maxmpri  = 1;
    pk_cmbx.mprihd   = (VP)msgque;

    ercd = cre_mbx(MBXID_CLIENT, &pk_cmbx);

        :
```

リスト 2-66　外部変数によるメールボックス ID の指定

```
static  ID                mbxid_client = 0;
#define  MBXID_CLIENT     mbxid_client
        :
    T_CMBX   pk_cmbx;
        :
    pk_cmbx.exinf    = (VP)1;
    pk_cmbx.mbxatr   = TA_TFIFO | TA_MFIFO;

    mbxid_client     = tk_cre_mbx(&pk_cmbx);
        :
```

上記のリストでは、μITRON4.0 も T-Kernel も MBXID_CLIENT というマクロを定義してあります。このようにすることで例えば、メールボックス ID を指定する必要のあるシステムコールの移行においてパラメータを変更する必要がなくなります。

なお、メールボックス ID を格納する変数 mbxid_client を直接 MBXID_CLIENT とすればマクロを定義する必要はなくなります。ただ一般には、マクロは大文字、変数名は小文字を利用することが多いようですので、上記リストのようにしました。実際の移行に際しては、各社のコーディングルールに適合するように調整してください。

(C) CRE_MBX から tk_cre_mbx へ

【移行の手順】

(1) システムコール名を tk_cre_mbx に変更する。

(2) maxmpri と mprihd を利用しない。

(3) メールボックス ID を自動割当てに変更する。

(4) 初期タスク内にソースコードを追加する。

前述したとおり、μT-Kernel はプログラムの動的なロードを前提としているため、静的 API をサポートしていません。

静的 API を使用している場合は、動的生成 tk_cre_mbx に置き換えます。

リスト 2-67 μITRON4.0 でのメールボックスの生成 (CRE_MBX)

```
CRE_MBX(mbx_id, { (TA_TFIFO | TA_MFIFO), 1, (VP)msgque } );
```

リスト 2-68 μT-Kernel でのメールボックスの生成

```
T_CMBX   pk_cmbx;
ID       mbx_id;
    :
pk_cmbx.exinf    = (VP)1;
pk_cmbx.mbxatr   = TA_TFIFO | TA_MFIFO;
    :
mbx_id = tk_cre_mbx(&pk_cmbx);

    :
```

【解説】

μITRON4.0 の静的 API である CRE_MBX の場合、開発者はシステムコンフィギュレーションファイルに静的に生成したいメールボックスの数だけ静的 API を記述し、そのファイルをコンフィギュレータに通すことでメールボックスが静的に生成されます。μT-Kernel では tk_cre_mbx を呼び出してメールボックスを生成します。静的 API を使用した場合、システムが起動した段階で既にメールボックスが生成されています。μT-Kernel の場合、初期タスク内で tk_cre_mbx を呼び出すことで同様の振る舞いになります。

2.4.2.2. メールボックスの削除（del_mbx → tk_del_mbx）

【移行の手順】

（1）システムコール名を tk_del_mbx に変更する。

リスト 2-69　μITRON4.0 でのメールボックスの削除

```
ER  ercd;
    :
ercd = del_mbx( mbx_id );

    :
```

リスト 2-70　μT-Kernel でのメールボックスの削除

```
ER  ercd;
    :
ercd = tk_del_mbx( mbx_id );

    :
```

【解説】

メールボックスの削除はシステムコール名を置き換えるだけで移行できます。

2.4.2.3. メールボックスへの送信（snd_mbx → tk_snd_mbx）

【移行の手順】

（1）システムコール名を tk_snd_mbx に変更する。

リスト 2-71　μITRON4.0 でのメールボックスへの送信

```
T_MSG    pk_msg;
ER       ercd;
     :
ercd = snd_mbx(mbx_id, &pk_msg);

     :
```

リスト 2-72　μT-Kernel でのメールボックスへの送信

```
T_MSG    pk_msg;
ER       ercd;
     :
ercd = tk_snd_mbx(mbx_id, &pk_msg);

     :
```

【解説】

メールボックスへの送信はシステムコール名を置き換えるだけで移行できます。

2.4.2.4. メールボックスからの受信（rcv_mbx / prcv_mbx / trcv_mbx → tk_rcv_mbx）

【移行のポイント】

（1）システムコール名を tk_rcv_mbx に変更する。

（2）ポーリングの動作の違いに注意する（返値の違い）

μT-Kernel では待ち状態に入る可能性があるシステムコールには、ポーリングおよびタイムアウト指定の機能を持たせています。

基本的には tk_rcv_mbx にシステムコール名を置き換えるだけで移行できます。待ち時間指定以外の永久待ちやポーリング指定をする場合は以下のように指定してください。

・rcv_mbx → タイムアウト指定に永久待ち（TMO_FEVR）を指定する
・prcv_mbx → タイムアウト指定にポーリング（TMO_POL）を指定する
・trcv_mbx → タイムアウト指定は変更不要

(A) rcv_mbx から tk_rcv_mbx へ

【移行の手順】

(1) システムコール名を tk_rcv_mbx に変更する。

(2) タイムアウト値に TMO_FEVR を設定する。

リスト 2-73　μITRON4.0 でのメールボックスからの受信

```
T_MSG *ppk_msg;
ER ercd;
     :
ercd = rcv_mbx(mbx_id, &ppk_msg);

     :
```

リスト 2-74　μT-Kernel でのメールボックスからの受信

```
T_MSG    *ppk_msg;
ER       ercd;
     :
ercd = tk_rcv_mbx(mbx_id, &ppk_msg, TMO_FEVR);

     :
```

【解説】

μT-Kernel では待ち状態に入る可能性があるシステムコールには、タイムアウト指定の機能を持たせているため、タイムアウト指定に永久待ち(TMO_FEVR)を指定することで移行できます。

(B) prcv_mbx から tk_rcv_mbx へ

【移行の手順】

(1) システムコール名を tk_rcv_mbx に変更する。

(2) タイムアウト値に TMO_POL を設定する。

リスト 2-75　μITRON4.0 でのメールボックスからの受信（ポーリング）

```
T_MSG *ppk_msg;
ER ercd;
     :
ercd = prcv_mbx(mbx_id, &ppk_msg);

     :
```

リスト 2-76　μT-Kernel でのメールボックスからの受信（ポーリング）

```
T_MSG    *ppk_msg;
ER       ercd;
     :
ercd = tk_rcv_mbx(mbx_id, &ppk_msg, TMO_POL);

     :
```

【解説】

prcv_mbx と tk_rcv_mbx はほぼ同等の機能であり、タイムアウト指定にポーリング（TMO_POL）を指定することで移行できます。

【実装に依存する点】

タスク独立部またはディスパッチ禁止状態でμITRON4.0 の trcv_mbx(TMO_POL)を発行した場合、実装によって動作が異なる可能性があるので注意が必要です。

表 2-10 実装による返値の違い

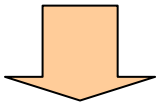
	タスク独立部または ディスパッチ禁止状態での発行	
	資源あり	資源なし
prcv_mbx()	E_OK	E_TMOUT|E_CTX
trcv_mbx(TMO_POL)	E_OK | E_CTX	E_TMOUT | E_CTX
tk_rcv_mbx(TMO_POL)	E_CTX	E_CTX

例えば、ハンドラ内で prcv_mbx を呼び出している場合、tk_rcv_mbx に置き換えると E_CTX が返ります。このようにタスク独立部で prcv_mbx や trcv_mbx の TMO_POL 指定を使用している場合には、新たにタスクを作成し、タスク内で tk_rcv_mbx を呼び出すなどの変更が必要となります。

リスト 2-77 タスク独立部で prcv_mbx を使っている場合の対応例 (μITRON4.0)

■μITRON4.0 のソースコード例

```
T_MSG *ppk_msg;
ID mbxid;
...
void handler(VP_INT exinf)
{
    ER ercd;
    ...
    ercd = prcv_mbx(mbxid, &ppk_msg);
    if(ercd == E_OK){
        ...
    }
}
```



リスト 2-78 タスク独立部で prcv_mbx を使っている場合の対応例 (μT-Kernel)

■μT-Kernel への移行例

```
T_MSG *ppk_msg;
ID mbxid;
ID tskid;
...
void handler(VP exinf)
{
    tk_wup_tsk(tskid);
}

void task(INT stacd, VP exinf)
{
    ER ercd;
    ...
    tk_slp_tsk(TMO_FEVR);
    ercd = tk_rcv_mbx(mbxid, &ppk_msg, TMO_POL);
    if(ercd == E_OK) {
        ...
    }
    ...
}
```

(C) trcv_mbx から tk_rcv_mbx へ

【移行の手順】

(1) システムコール名を tk_rcv_mbx に変更する。

リスト 2-79　μITRON4.0 でのメールボックスからの受信（タイムアウトあり）

```
T_MSG    *ppk_msg;
ER       ercd;
     :
ercd = trcv_mbx(mbx_id, &ppk_msg, 100);

     :
```

リスト 2-80　μT-Kernel でのメールボックスからの受信（タイムアウトあり）

```
T_MSG    *ppk_msg;
ER       ercd;
     :
ercd = tk_rcv_mbx(mbx_id, &ppk_msg, 100);

     :
```

【解説】

trcv_mbx と tk_rcv_mbx は同等の機能であり、システムコール名を置き換えるだけで移行できます。

2.4.2.5. メールボックスの状態参照（ref_mbx）

【移行のポイント】

（1）システムコール名称を変更する。

（2）「待ちタスクなし」を示すマクロ TS_NONE(=0)が削除されているので定義する。

（3）T_RMBX 構造体のメンバ変数を変更する。

リスト 2-81　μITRON4.0 でのメールボックスの状態参照

```
T_RMBX pk_rmbx;
ER     ercd;
    :
ercd = ref_mbx(mbx_id, &pk_rmbx);
if(pk_rmbx.wtskid == TSK_NONE){
    :
}
```

リスト 2-82　μT-Kernel でのメールボックスの状態参照

```
#define TSK_NONE    0
T_RMBX pk_rmbx;
ER     ercd;
    :
ercd = tk_ref_mbx(mbx_id, &pk_rmbx);
if(pk_rmbx.wtsk == TSK_NONE) {
    :
}
```

2.5. 相違点

2.5.1. 共通項目

本章では、データ型やエラーコードなどの共通項目における移行方法や仕様間での相違点における注意点について説明します。

2.5.1.1. データ型

汎用的なデータ型における相違点は以下のとおりです。

(A) そのまま利用できるデータ型

整数は 64 ビット整数を除き、そのまま利用できます。

表 2-11 そのまま利用できるデータ型

データ型	μITRON4.0、μT-Kernel での定義
B	符号付き 8 ビット整数
H	符号付き 16 ビット整数
W	符号付き 32 ビット整数
UB	符号無し 8 ビット整数
UH	符号無し 16 ビット整数
UW	符号無し 32 ビット整数
VB	型が一定しない 8 ビットのデータ
VH	型が一定しない 16 ビットのデータ
VW	型が一定しない 32 ビットのデータ
VP	型が一定しないデータへのポインタ
INT	プロセッサのビット幅の符号付き整数
UINT	プロセッサのビット幅の符号無し整数

(B) μT-Kernel では定義されていないデータ型

μT-Kernel で定義されていないデータ型を利用している場合には、移行時に定義を追加する必要があります。

主に定義する必要がある型は 64 ビット整数型です。

表 2-12 μT-Kernel では定義されていないデータ型

データ型	μITRON4.0 仕様の定義
D	符号付き 64 ビット整数
UD	符号無し 64 ビット整数
VD	データタイプが定まらない 64 ビットの値
STAT オブジェクトの状態	符号無し整数※1
MODE サービスコールの動作モード	符号無し整数※2
SIZE メモリ領域のサイズ	符号無し整数※3
VP_INT	データタイプが定まらないものへのポインタまたはプロセッサに自然なサイズの符号付き整数
ER_BOOL エラーコードまたは真偽値	符号付き整数
ER_ID エラーコードまたは ID 番号	符号付き整数
ER_UINT エラーコードまたは符号無し整数	符号付き整数※4
TEXPTN タスク例外要因のビットパターン	符号無し整数
FLGPTN イベントフラグのビットパターン	符号無し整数
RDVPTN ランデブ条件のビットパターン	符号無し整数
RDVNO ランデブ番号	
OVRTIM プロセッサ時間	符号無し整数
INHNO 割込みハンドラ番号	
INTNO 割込み番号	
EXCNO CPU 例外ハンドラ番号	

※1 スタンダードプロファイルでは 16 ビット以上の制約あり。
※2 スタンダードプロファイルでは 8 ビット以上の制約あり。
※3 スタンダードプロファイルではポインタと同じビット数との制約あり
※4 符号無し整数を表現する場合の有効ビット数は UINT より 1 ビット短い。

これらを利用している場合は、移植元のμITRONの仕様（実装）を確認して適宜定義し、全体でインクルードするヘッダファイルなどに追加します。

なお、D、UD、VD 型はμITRON のスタンダードプロファイルに含まれませんので、移植元のμITRON によっては対応していない場合もあります。使用していない型については、特に定義する必要はありません。

(C) μITRON4.0 仕様では実装毎に異なる定義になる可能性のあるデータ型

μITRON4.0 仕様では実装毎に異なる定義になる可能性のあるデータ型があります。

多くがプロセッサのビット幅に依存するものであるため、プロセッサのビット幅が異なるプロセッサへ移植する場合は注意が必要です。

表 2-13 システム毎に異なる定義となる可能性のあるデータ型

データ型	μITRON4.0 仕様の定義	μT-Kernel の定義
FN 機能コード	符号付き整数※1	プロセッサのビット幅の符号付き整数
ER エラーコード	符号付き整数※2	プロセッサのビット幅の符号付き整数
ID オブジェクトの ID 番号	符号付き整数※1	プロセッサのビット幅の符号付き整数
ATR オブジェクトの属性	符号無し整数※2	符号無し32ビット整数
PRI 優先度	符号付き整数	プロセッサのビット幅の符号付き整数
TMO タイムアウト指定	符号付き整数※1 時間単位は実装定義※3	符号付き 32 ビット整数 時間単位は 1 ミリ秒
RELTIM 相対時間	符号無し整数※1 時間単位は実装定義※3	符号無し 32 ビット整数 時間単位は 1 ミリ秒
SYSTIM システム時刻	符号無し整数※1 時間単位は実装定義※3	64 ビット符号付き整数 時間単位は 1 ミリ秒

※1　スタンダードプロファイルでは 16 ビット以上の制約あり。
※2　スタンダードプロファイルでは 8 ビット以上の制約あり。
※3　スタンダードプロファイルでは時間単位は 1 ミリ秒。

(D) μITRON4.0 仕様では定義されていないデータ型

μITRON4.0 仕様では定義されていないデータ型は、移行時には特に注意する必要はありませんが、μITRON で以下の型を利用する際にμT-Kernel での定義に合わせておくことで、ソースコードの移行がしやすくなります。

表 2-14 μITRON4.0 仕様では定義されていないデータ型

データ型	T-Kernel での定義
_B	volatile 宣言付の基本データ型
_H	
_W	
_UB	
_UH	
_UW	
RNO ランデブ番号	プロセッサのビット幅の符号付き整数
MSEC 時間一般(ミリ秒)	プロセッサのビット幅の符号付き整数
FUNCP 関数アドレス一般	ポインタ
TC TRON 文字コード	符号無し 16 ビット整数

2.5.1.2. エラーコード [FEI]

μITRON でいうサービスコール、T-Kernel でいうシステムコールの返値は符号付の整数で、エラーが発生した場合には負の値のエラーコード、処理を正常に終了した場合は E_OK(=0)または正の値と定義されています。

μITRON と T-Kernel のエラーコードはメインエラーコードとサブエラーコードから構成されるものとし、カーネルとソフトウェア部品でメインエラーコードを共通に利用し、エラーが発生した原因をより細かく報告できるようにしたものがサブエラーコードです。

ただし、μT-Kernel ではメインエラーコードのみを扱います。

【移行の際の注意点】

μITRON4.0 仕様と T-Kernel、μT-Kernel ではメインエラーコードが格納されているビットフィールドが下図のように異なるので注意が必要です。

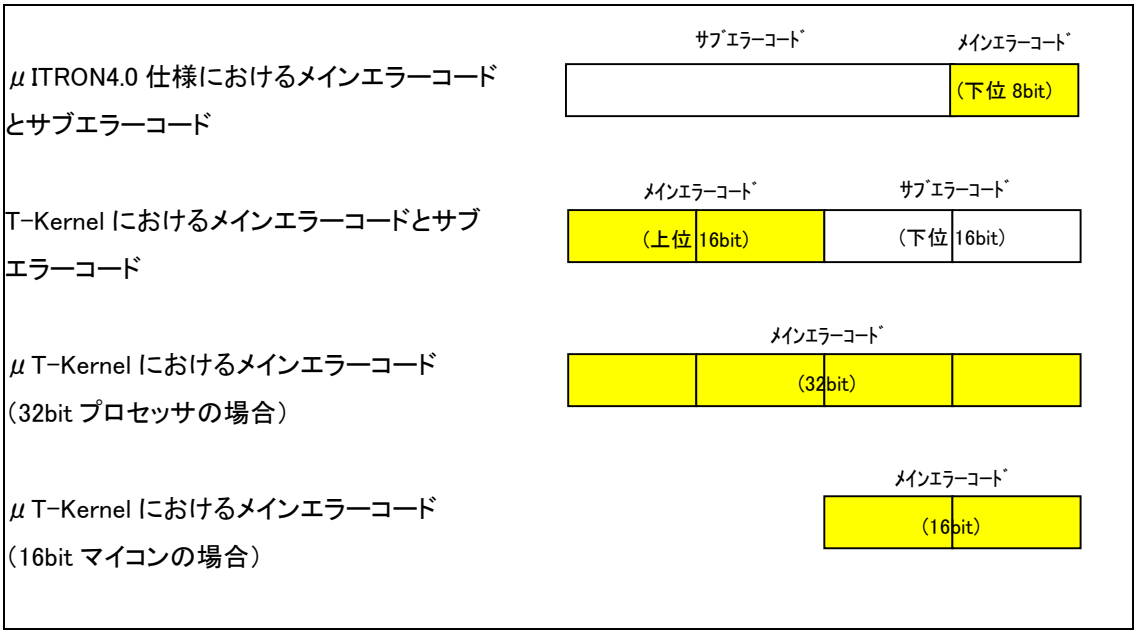


図 2-12 エラーコードの構成

μITRON4.0 ではサブエラーコードは構造としては定義されていますが、仕様上は実装定義となっており、必ずしも実装されているとは限りません。一方、T-Kernel では、構造として上図のように定義されていますが、T-Kernel/OS ではサブエラーコードは使用せず常に 0 となっています。また、μT-Kernel では、サブエラーコード自体が省略されています。

エラーコードは E_XXX というマクロで定義されています。このため、特にサブエラーコードを使用していないシステム間でプログラムを移行する場合は、エラーコードに関しては特に既存のプログラムを変える必要はありません。(リスト 2-83、リスト 2-84)

リスト 2-83　μITRON4.0 でのエラーの判定例

```
ER      ercd;

    :

ercd = ter_tsk( MOTOR_TSKID );
if( ercd < 0 ){
    if( ercd == E_ID ){
        /* 不正 ID 番号の場合の処理  */
    } else if ( ercd == E_NOEXS ) {
        /* タスクが存在しない場合の処理  */
    }
    /* その他のエラー処理 */
}
    :
```

リスト 2-84　μT-Kernel でのエラーの判定例

```
ER      ercd;

    :

ercd = tk_ter_tsk( MOTOR_TSKID );
if( ercd < 0 ){
    if( ercd == E_ID ){
        /* 不正 ID 番号の場合の処理  */
    } else if ( ercd == E_NOEXS ) {
        /* タスクが存在しない場合の処理  */
    }
    /* その他のエラー処理 */
}
    :
```

もし、サブエラーコードが実装されている場合は、μITRON4.0 ではエラーコード用のマクロ MERCD()を利用することになります。μT-Kernel では MERCD()マクロは定義されていませんので、必要に応じて MERCD()をダミーで定義するか、または、MERCD()を削除してください。

リスト 2-85　μITRON4.0 でのエラーの判定例(サブエラーコードあり)

```
ER      ercd;

    :

ercd = ter_tsk( MOTOR_TSKID );
ercd = MERCD(ercd);
if( ercd < 0 ){
    if( ercd == E_ID ){
        /* 不正 ID 番号の場合の処理  */
    } else if ( ercd == E_NOEXS ) {
        /* タスクが存在しない場合の処理  */
    }
    /* その他のエラー処理 */
}
    :
```

リスト 2-86　μT-Kernel でのエラーの判定例

```
#define MERCD(ercd) (ercd)
        :

    ER      ercd;

        :

    ercd = tk_ter_tsk( MOTOR_TSKID );
    ercd = MERCD(ercd);
    if( ercd < 0 ){
        if( ercd == E_ID ){
            /* 不正 ID 番号の場合の処理  */
        } else if ( ercd == E_NOEXS ) {
            /* タスクが存在しない場合の処理  */
        }
        /* その他のエラー処理 */
    }
        :
```

以下、メインエラーコードの定義値に着目して相違点を示します。

(A) そのまま利用できる定義

以下のマクロ定義はそのまま利用できます。

表 2-15 そのまま利用できるエラーコード

定数	値	説明
E_OK※	0	正常終了
E_SYS	-5	システムエラー
E_NOSPT	-9	未サポート機能
E_RSFN	-10	予約機能コード番号
E_RSATR	-11	予約属性
E_PAR	-17	パラメーターエラー
E_ID	-18	不正 ID 番号
E_CTX	-25	コンテキストエラー
E_MACV	-26	メモリアクセス権違反、メモリアクセス不能
E_OACV	-27	オブジェクトアクセス違反
E_ILUSE	-28	システムコール不正使用
E_NOMEM	-33	メモリ不足
E_OBJ	-41	オブジェクト状態が不正
E_NOEXS	-42	オブジェクトが存在していない
E_QOVR	-43	キューイング、またはネストのオーバフロー
E_RLWAI	-49	待ち状態の強制解除
E_TMOUT	-50	ポーリング失敗またはタイムアウト
E_DLT	-51	待ちオブジェクトの削除

※E_OK はエラーコードではありませんが、便宜上含めてあります。

(B) μT-Kernel では定義されていないエラーコード定義

以下にあげる定義はμT-Kernel では定義されていないため、使用する場合には同じ名前のマクロを定義する必要があります。

表 2-16 μT-Kernel では定義されていないエラーコード

定数	値	説明
E_NOID	-34	ID 番号不足
E_WBLK	-57	ノンブロッキング受付け
E_BOVR	-58	バッファオーバフロー

【移行の例】

例えば、専用のヘッダファイルを作成し、その中で不足しているエラーコードをマクロ定義することでコンパイルエラーを回避します。

リスト 2-87 μT-Kernel にないエラーコードの定義例

```
/* uT-Kernelでは定義されていない定義 */
#define E_NOID   (-34)
#define E_WBLK   (-57)
#define E_BOVR   (-58)
```

定義する値は任意の負の値を利用できますが、μT-Kernel の他のエラーコードと同じ値にならないように定義してください。

(C) μITRON4.0 仕様では定義されていないエラーコード定義

以下にあげる定義は、T-Kernel で新たに追加されたエラーコード定義となります。

表 2-17 μITRON4.0 仕様では定義されていないエラーコード

定数	値	説明
E_NOCOP	-6	コプロセッサ使用不可
E_LIMIT	-34	システムの制限を超過
E_DISWAI	-52	待ち禁止による待ち解除
E_IO	-57	入出力エラー
E_NOMDA	-58	メディアがない
E_BUSY	-65	ビジー状態
E_ABORT	-66	中止した
E_RONLY	-67	書込み禁止

【移行の例】

μITRON4.0 仕様では定義されていないため、特に対策は不要です。

2.5.1.3. 定数

μITRON4.0 仕様と、μT-Kernel 仕様の一方でしか定義されていない定数は以下のとおりです。

表 2-18 μITRON4.0 仕様でしか定義されていない定数

定義	説明
TA_ACT	タスクを起動された状態で生成
TA_RSTR	制約タスク
TA_CLR	待ち解除時にイベントフラグをクリア
TMO_NBLK	ノンブロッキング
TTW_SDTQ	データキューへの送信待ち状態
TTW_RDTQ	データキューからの受信待ち状態
TTEX_ENA	タスク例外処理許可状態
TTEX_DIS	タスク例外処理禁止状態
TOVR_STP	上限プロセッサ時間が設定されていない
TOVR_STA	上限プロセッサ時間が設定されている
TSK_NONE	該当するタスクがない
TPRI_SELF	自タスクのベース優先度の指定
TMIN_TPRI	タスク優先度の最小値（＝1）
TMAX_TPRI	タスク優先度の最大値
TMIN_MPRI	メッセージ優先度の最小値（＝1）
TMAX_MPRI	メッセージ優先度の最大値
TKERNEL_MAKER	カーネルのメーカコード
TKERNEL_PRID	カーネルの識別番号
TKERNEL_SPVER	ITRON 仕様のバージョン番号
TKERNEL_PRVER	カーネルのバージョン番号
TMAX_ACTCNT	タスクの起動要求キューイング数の最大値
TMAX_WUPCNT	タスクの起床要求キューイング数の最大値
TMAX_SUSCNT	タスクの強制待ち要求ネスト数の最大値
TBIT_TEXPTN	タスク例外要因のビット数
TBIT_FLGPTN	イベントフラグのビット数
TBIT_RDVPTN	ランデブ条件のビット数
TIC_NUME	タイムティックの周期の分子
TIC_DEMO	タイムティックの周期の分母
TMAX_MAXSEM	セマフォの最大資源数の最大値

表 2-19　μT-Kernel 仕様でしか定義されていない定数

定義	説明
TA_FIRST	待ち行列先頭のタスクを優先
TA_CNT	要求数の少ないタスクを優先
TA_USERBUF	スタック領域としてユーザが指定した領域を使用する
TA_DSNAME	DS オブジェクト名称を指定する
TA_RNGn	対象タスクは保護レベル n で実行する
TWF_CLR	全クリア指定
TWF_BITCLR	条件ビットのみクリア指定
TPRI_RUN	実行状態(RUNNING)にあるタスクの優先度のタスクの優先順位を回転する

第3章 ラッパーを使った移行方法

この章では、μITRON4.0 の API を使って書かれたプログラムに「ラッパー」をかぶせることによって、T-Kernel 上に移行させる方法について説明します。この方法のメリットは、元のソースプログラムをほとんど変更する必要がなく、移行に必要な作業を大幅に軽減できることです。

（用語解説）

プログラムから ITRON や T-Kernel の機能を呼び出す方法は、細かく分けると「サービスコール」、「システムコール」、「拡張 SVC」、「ライブラリ関数」、「マクロ」などさまざまな区別がありますが、この章ではこれらを特に区別せずに「API (Application Program Interface)」と総称することにします。

3.1. ラッパーとは

μITRON4.0 の API を使って書かれたプログラムを T-Kernel に移行するには、主に次の二つのアプローチが考えられます。μITRON4.0 の API は多数ありますが、ここでは API を代表して wup_tsk(タスク起床)の例で示します。他の API についても同様です。

3.1.1. ソースプログラムを直接書き変える方法

ITRONとT-Kernelの仕様の差を埋めるために、ソースプログラムを書き換えて対応する方法です。元のソースプログラム上の ITRON の wup_tsk は T-Kernel の tk_wup_tsk に書き換える形になります。

ただしタスク ID は ITRON では基本的には固定 ID(プログラムのコンパイル時にコンフィギュレータ等を使って固定の ID 値を割り当てる方法)、T-Kernel では動的 ID(プログラムの実行時にT-Kernelが動的にIDの値を割り当てる方法)ですので、その間の変換を考慮してソースプログラムを修正する必要があります。

また、T-Kernel の tk_wup_tsk は自タスクを直接起床できないため、元のソースプログラムが自タスクの起床を行っているかどうか注意深く確認して、個別に対応方法を選択する必要があります。

3.1.2. ラッパーを使う方法

ITRON と T-Kernel の仕様の差をライブラリで吸収する方法です。こうしたライブラリをラッパー(wrapper)と呼びます。固定 ID と動的 ID の変換や、自タスクかどうかの判断と対応などはすべてこのライブラリ内で自動的に行います。元のソースプログラム上の wup_tsk は変更せずに、ライブラリ関数として呼び出す形になります。

この方法のメリットは、元のソースプログラムをほとんど変更する必要がなく、大部分を自動的に移行できることです。

(用語解説)

「ラッパー(wrapper)」とは「包むもの」という意味ですが、この場合は、T-Kernel に対して、それを「包む」ことにより、変換や機能追加を行う付加的なプログラム(ライブラリ)を表わします。

3.2. ラッパー内部の処理

ITRONとT-Kernelの仕様の差を吸収するために、ラッパーの内部では主に次のような処理を行う必要があります。

3.2.1. API名の変換

ITRONのAPI名をラッパー内部でT-KernelのAPI名に変換して呼び出します。

（例） wup_tsk → tk_wup_tsk

基本的にはITRONのAPI名の先頭に「tk_」をつけたものがT-KernelのAPIになりますが、引数などに細かい違いもあります。

3.2.2. 固定IDと動的IDの変換

ITRONのオブジェクトの固定IDは、ラッパー内部で変換テーブル等を使ってT-Kernelの動的IDに変換する必要があります。

（例） T-Kernelのタスク IDの動的な値をラッパー内部で管理する配列 tid[]に保持して

wup_tsk(3) → tk_wup_tsk(tid[3 - 1])

† 配列の添字の範囲チェック等も必要です。

3.2.3. エラーコードの変換

ITRONのエラーコードとT-Kernelのエラーコードは値が異なりますので、ラッパー内部で変換する必要があります。

（例） ITRONの E_PAR = -17

→ T-Kernelでは E_PAR = (-17 << 16)

→ μT-Kernelでは E_PAR = -17

基本的にはシフト演算で対応できますが、APIによってはITRONとT-Kernelでエラーコードが一部異なる場合もありますので、個別に対応する必要があります。

3.2.4. T-Kernelにない機能、細かい仕様の違い

T-Kernelにない ITRONの機能や、細かい仕様の違いについては、できる限りITRON仕様に合わせるようにラッパー側で対応します。

（例）ITRONではact_tskでタスク起動要求をキューイングしておくことが可能です。この場合、そのタスクは終了すると同時に再起動します。この機能はT-Kernelにはないため、ラッパー内部の処理で対応する必要があります。

3.3. ラッパー内部の処理の具体例

次にラッパー内部の処理の具体例をいくつか紹介します。

（例 1） メッセージバッファ生成

T-Kernel のメッセージバッファ ID の動的な値をラッパー内部で管理する配列 mbfid[] に保持するようにします。

リスト 3-1 ITRON 用ソースプログラム（静的 API）

```
CRE_MBF(MBF_C, {TA_TFIFO, 10, 1000, NULL});
```

リスト 3-2 ラッパー内部の処理例

```
#define MBF_C  3  /* 固定 ID */

   T_CMBF cmbf = {NULL, TA_TFIFO, 1000, 10};
   mbfid[MBF_C - 1] = tk_cre_mbf(&cmbf);
```

†実際にはパラメータチェックやエラー処理等も必要です。

（例 2） メッセージバッファへのメッセージ送信

T-Kernel のメッセージバッファ ID の動的な値をラッパー内部で管理する配列 mbfid[] に保持するようにします。

リスト 3-3 ITRON 用ソースプログラム

```
    er = snd_mbf(MBF_C, msg, sizeof(msg));
```

リスト 3-4 ラッパー内部の処理例

```
er = tk_snd_mbf(mbfid[MBF_C - 1], msg, sizeof(msg), TMO_FEVR);
er >>= 16; /* T-Kernel のエラーコードを ITRON のエラーコードに */
```

（例 3） 自タスクを含むタスクの起床

μITRON4.0 では自タスクへのタスク起床要求のキューイングも可能ですが、T-Kernel の tk_wup_tsk は自タスクには対応しておらず、オブジェクト状態エラー(E_OBJ)を返します。そこでラッパー内部でいったん別のコンテキスト(他タスクまたはタスク独立部)に切り替えてから tk_wup_tsk を使うようにします。

リスト 3-5 ITRON 用ソースプログラム

```
er = wup_tsk(n);
```

リスト 3-6 ラッパー内部の処理例

```
er = tk_wup_tsk(tid[n - 1]);
if (er == E_OBJ && tid[n - 1] == 自タスク) {
     別のコンテキストに切り替える;
     er = tk_wup_tsk(tid[n - 1]);
     元のタスクのコンテキストに切り替える;
}
if (er < 0) er >>= 16; /* エラーコード変換 */
```

† 実際には n == 0 (TSK_SELF) の場合のチェック等も必要です。

このように、ラッパー全体を開発するには、相当な開発工数が必要になります。

しかし、関連製品の章で紹介紹介するような製品も利用可能です。このような製品などを利用すればラッパーを開発する必要はありません。

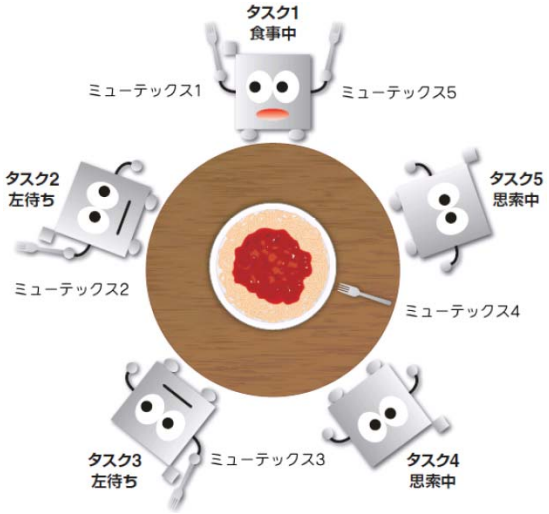
3.4. 移行の具体例

ここでは移行の具体例として、簡単な ITRON のソースプログラムを T-Kernel に移行する例を扱います。移行に必要な作業量を比較するため、ラッパーとして I-right/TK を使って移行する場合と、I-right/TK などのラッパーを使わずにソースプログラムを直接書き換えて T-Kernel に移行する場合の両方について説明します。

3.4.1. 元の ITRON 用ソースプログラム

ITRON の API を使って書かれた「哲学者の食事問題」を解くプログラムを例として取り上げます。

丸テーブルを囲む 5 人の哲学者(タスク)が思索と食事を繰り返します。食事するには自分の左右のフォーク(ミューテックス)を両方とも獲得(ロック)する必要があります。全員同時にフォークを取ろうと待ち続けてデッドロックに陥ったり、特定の哲学者だけ食事できないようなことが発生しないためには、フォークを取るルールを工夫する必要があります。ここでは「1 番めの哲学者だけは左のフォークを先に、残りの 4 人の哲学者は右のフォークを先に取る」というルールにしています。



(A) コンフィギュレーションファイル(ITRON 用)

静的 API を使ってタスクとミューテックスを 5 個ずつ定義しています。ミューテックス ID は固定値(1,2,3,4,5)としている点にご注意ください。

リスト 3-7 コンフィギュレーションファイルの例(ITRON 用)

```
CRE_TSK(TID_A, {TA_HLNG | TA_ACT, 1, task, 1, 8192, NULL});
CRE_TSK(TID_B, {TA_HLNG | TA_ACT, 2, task, 1, 8192, NULL});
CRE_TSK(TID_C, {TA_HLNG | TA_ACT, 3, task, 1, 8192, NULL});
CRE_TSK(TID_D, {TA_HLNG | TA_ACT, 4, task, 1, 8192, NULL});
CRE_TSK(TID_E, {TA_HLNG | TA_ACT, 5, task, 1, 8192, NULL});
CRE_MTX(1, {TA_TFIFO, 0});
CRE_MTX(2, {TA_TFIFO, 0});
CRE_MTX(3, {TA_TFIFO, 0});
CRE_MTX(4, {TA_TFIFO, 0});
CRE_MTX(5, {TA_TFIFO, 0});
```

(B) 哲学者タスクのソースプログラム(ITRON 用)

このソースプログラムでも自分の左右のミューテックス ID(mLeft, mRight)を求める上で、ミューテックス ID は固定値(1,2,3,4,5)であることを前提としています。

リスト 3-8 哲学者タスクのソースプログラム例(ITRON 用)

```
#include <itron.h>    /* ITRON ヘッダファイル */
#include <stdio.h>    /* printf() */

/* 哲学者タスク */
void task(VP_INT n)
{
    /* 自分の右のミューテックス ID */
    ID  mRight = n;

    /* 自分の左のミューテックス ID */
    ID  mLeft = n > 1 ? n - 1 : 5;

    W   i;
    for(i = 1; i <= 10; i++) {
        /* 思索する */
        dly_tsk(200);

        /* 左右のミューテックスが取れるまで待つ */
        loc_mtx(n == 1 ? mLeft : mRight);
        dly_tsk(10);
        loc_mtx(n == 1 ? mRight : mLeft);

        /* 食事する */
        printf("哲学者 %d : %d 回目の食事開始¥n", n, i);
        dly_tsk(100);
        printf("哲学者 %d : %d 回目の食事終了¥n", n, i);

        /* 左右のミューテックスを手放す */
        unl_mtx(mLeft);
        unl_mtx(mRight);
    }
}
```

3.4.2. I-right/TK を使って T-Kernel に移行する場合

(A) 初期化ルーチン (T-Kernel + I-right/TK 用)

I-right/TK では、ITRON の静的 API は STA_CFG と END_CFG で囲むだけで記述できます。これをプログラムの初期化ルーチンに含めます。

リスト 3-9 初期化ルーチン(T-Kernel＋I-right/TK 用)

```
#include <itron.h>    /* I-right/TK ヘッダファイル */

/* 初期化ルーチン */
ER  main(INT ac, UB *av[])
{
    STA_CFG;          /* コンフィギュレーション開始 */

    CRE_TSK(TID_A, {TA_HLNG | TA_ACT, 1, task, 1, 8192, NULL});
    CRE_TSK(TID_B, {TA_HLNG | TA_ACT, 2, task, 1, 8192, NULL});
    CRE_TSK(TID_C, {TA_HLNG | TA_ACT, 3, task, 1, 8192, NULL});
    CRE_TSK(TID_D, {TA_HLNG | TA_ACT, 4, task, 1, 8192, NULL});
    CRE_TSK(TID_E, {TA_HLNG | TA_ACT, 5, task, 1, 8192, NULL});
    CRE_MTX(1, {TA_TFIFO, 0});
    CRE_MTX(2, {TA_TFIFO, 0});
    CRE_MTX(3, {TA_TFIFO, 0});
    CRE_MTX(4, {TA_TFIFO, 0});
    CRE_MTX(5, {TA_TFIFO, 0});

    END_CFG;          /* コンフィギュレーション終了 */
    return ERR_CFG;   /* 初期化ルーチン終了 */
}
```

† 初期化ルーチンは、ここでは代表的に main という名前にしていますが、システムによっては異なる場合があります。

(B) 哲学者タスク (T-Kernel + I-right/TK 用)

このソースプログラムは、まったく書き換える必要はありません。この後で説明するラッパーを使わない場合に必要となる API の変換、ID の変換、オブジェクト生成タイミングなどはすべて I-right/TK 側で自動的に対応しますので、このままで動作します。

リスト 3-10 哲学者タスク(T-Kernel+I-right/TK 用)

```
#include <itron.h>    /* I-right/TK ヘッダファイル */
#include <stdio.h>    /* printf() */

/* 哲学者タスク */
void task(VP_INT n)
{
    /* 自分の右のミューテックス ID */
    ID  mRight = n;

    /* 自分の左のミューテックス ID */
    ID  mLeft = n > 1 ? n - 1 : 5;

    W   i;
    for(i = 1; i <= 10; i++) {
        /* 思索する */
        dly_tsk(200);

        /* 左右のミューテックスが取れるまで待つ */
        loc_mtx(n == 1 ? mLeft : mRight);
        dly_tsk(10);
        loc_mtx(n == 1 ? mRight : mLeft);

        /* 食事する */
        printf("哲学者 %d : %d 回目の食事開始¥n", n, i);
        dly_tsk(100);
        printf("哲学者 %d : %d 回目の食事終了¥n", n, i);

        /* 左右のミューテックスを手放す */
        unl_mtx(mLeft);
        unl_mtx(mRight);
    }
}
```

以上のプログラムをメイクして I-right/TK とリンクすれば、T-Kernel 上で正しく動作します。

3.4.3. ソースプログラムを書き換えて T-Kernel に移行する場合

移行に必要な作業量を比較するため、I-right/TK などのラッパーを使わずにソースプログラムを直接書き換えて T-Kernel に移行する場合について考えてみましょう。

(A) API の変換

API を ITRON から T-Kernel に変更します。

ITRON	→	T-Kernel
CRE_TSK (TA_ACT)	→	tk_cre_tsk と tk_sta_tsk
CRE_MTX	→	tk_cre_mtx
dly_tsk	→	tk_dly_tsk
loc_mtx	→	tk_loc_mtx (TMO_FEVR)
unl_mtx	→	tk_unl_mtx
(タスクからのリターン)	→	tk_ext_tsk

(B) 固定 ID と動的 ID の変換

元のソースプログラムはミューテックス ID は固定値(1,2,3,4,5)であることを前提としているため、T-Kernel の動的 ID ではそのままでは動作しませんので、ソースプログラムの修正が必要となります。

ここでは固定 ID を動的 ID に変換するために、変換テーブル(配列)を確保することにします。

(C) オブジェクトの生成タイミングの考慮

元のコンフィギュレーションファイルではタスクを生成・起動してからミューテックスを生成する順番で書かれていますが、こうした順番に関係なく、コンフィギュレーション全体が完了してから、すべてのタスクが一斉に実行可能になることが保証されていました。

一方、T-Kernel 上で初期化ルーチンの中でタスクを生成・起動後にミューテックスを生成すると、初期化ルーチンより優先度の高いタスクは起動した段階ですぐ実行開始され、その時点では他のミューテックスやタスクは未生成・未起動のため、プログラムは正しく実行されません。

この問題を回避するため、今回は初期化ルーチンの中でタスクとミューテックスをすべて生成してから、一斉にタスクを起動するように、順序を入れ替えることにします。

厳密に言えば、tk_sta_tsk を使ってタスクを一つずつ起動しただけでは「一斉に起動」したことにはならず、やはりタイミングにずれが生じます。しかし今回の場合はタスク起動後にそのタスクが tk_dly_tsk で待ち状態になり、すぐに初期化ルーチンに実行権が戻ってきて次のタスクを起動できますので、「ほぼ同時に起動」できます。

(参考)

タスクが実行開始後すぐ待ちに入らない場合は、タスクを一斉に起動するために工夫する必要があります。例えば初期化ルーチンのタスク優先度を一時的に引き上

げて、生成するタスクの優先度よりも高くしておき、すべてのタスクの生成・起動後に、初期化ルーチンのタスク優先度を元に引き下げます。ただしこの方法では、割込みハンドラや周期ハンドラなどのタスク独立部の起動は遅延できません。静的 API でこれらのハンドラを生成・起動している場合は、個別の対応が必要です。

以上のような検討を行った上で、元のソースプログラムを次のように書き換えます。

リスト 3-11 初期化ルーチン（T-Kernel 用）

```
#include <tk/tkernel.h>     /* T-Kernel ヘッダーファイル */

ID  mtxID[5];            /* ミューテックス ID テーブル */

/* 初期化ルーチン */
ER  main(INT ac, UB *av[])
{
    T_CTSK ctsk = {NULL, TA_HLNG | TA_RNG0, task, 120, 8192};
    T_CMTX cmtx = {NULL, TA_TFIFO, 0};
    ID     tskID[5]; /* タスク ID テーブル */
    W      n;

    /* タスクを生成する */
    for(n = 1; n <= 5; n++) {
        tskID[n - 1] = tk_cre_tsk(&ctsk);
    }

    /* ミューテックスを生成する */
    for(n = 1; n <= 5; n++) {
        mtxID[n - 1] = tk_cre_mtx(&cmtx);
    }

    /* タスクを起動する */
    for(n = 1; n <= 5; n++) {
        tk_sta_tsk(tskID[n - 1], n);
    }

    return E_OK;   /* 初期化ルーチン終了 */
}
```

リスト 3-12 哲学者タスク（T-Kernel 用）

```
#include <tk/tkernel.h>  /* T-Kernel ヘッダファイル */
#include <stdio.h>       /* printf() */

extern  ID  mtxID[];     /* ミューテックス ID テーブル */

/* 哲学者タスク */
void task(INT n, VP exinf)
{
    /* 自分の右のミューテックス ID */
    ID  mRight = mtxID[n - 1];

    /* 自分の左のミューテックス ID */
    ID  mLeft = mtxID[(n > 1 ? n - 1 : 5) - 1];

    W    i;
    for(i = 1; i <= 10; i++) {
        /* 思索する */
        tk_dly_tsk(200);

        /* 左右のミューテックスが取れるまで待つ */
        tk_loc_mtx(n == 1 ? mLeft : mRight, TMO_FEVR);
        tk_dly_tsk(10);
        tk_loc_mtx(n == 1 ? mRight : mLeft, TMO_FEVR);

        /* 食事する */
        printf("哲学者 %d : %d 回目の食事開始¥n", n, i);
        tk_dly_tsk(100);
        printf("哲学者 %d : %d 回目の食事終了¥n", n, i);

        /* 左右のミューテックスを手放す */
        tk_unl_mtx(mLeft);
        tk_unl_mtx(mRight);
    }

    tk_ext_tsk();         /* タスク終了 */
}
```

以上のプログラムをメイクすれば、T-Kernel 上で正しく動作します。

3.5. この章のまとめ

この章では、ラッパーを使うことにより、元の ITRON 用のソースプログラムをほとんど書き換える必要なく、最小限の手間で T-Kernel に移行できることを説明しました。

既に製品化された T-Kernel 用の ITRON ラッパーとしては、パーソナルメディア株式会社の「I-right/TK」があります。I-right/TK は、パーソナルメディアから発売されている T-Kernel 応用製品はもちろん、ユーザ自身あるいは第三者が移植した T-Kernel 上にも若干のカスタマイズにより搭載可能です。詳細についてはパーソナルメディア株式会社までお問合せください。

第4章　開発環境／関連製品

第 4 章では、μT-Kernel の開発環境の構成と構築方法について説明します。

4.1. μT-Kernel の開発環境

4.1.1. リファレンスコードの開発環境(GCC)

T-Engine フォーラムの Web ページで公開されているμT-Kernel のリファレンスコードは ARM7TDMI と H8S/2212 に対応しています。μT-Kernel の開発には GCC などの GNU ツールを使います。そのため、Windows 上で開発を行う場合には図 4-1 に示すように UNIX エミュレータの Cygwin を利用します。

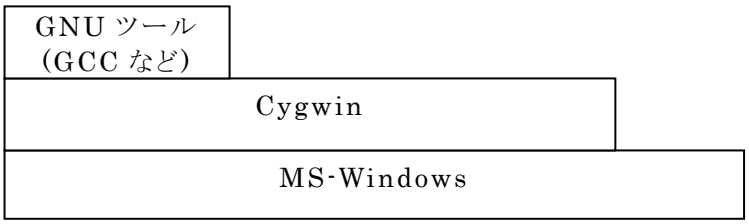


図 4-1 開発環境の構成

開発環境は図 4-2 に示す手順で構築します。

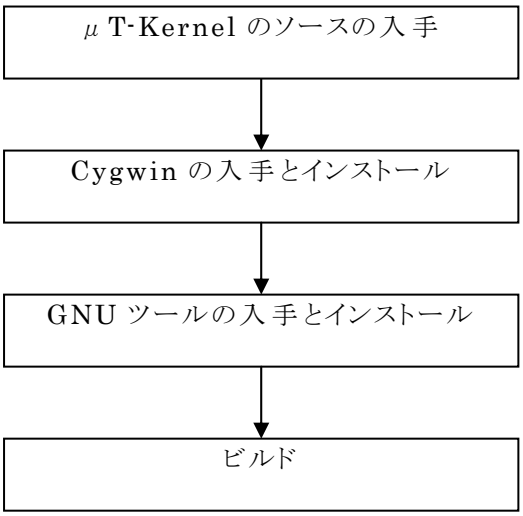


図 4-2 開発環境構築手順

(1) μT-Kernel のソースの入手

μT-Kernel ソースコードは T-Engine フォーラムの Web ページから入手できます。

URL: http://www.t-engine.org/T-Kernel/tkernel.html#utk_src

はじめに、μT-Kernel の利用申込みを行います(①)。フォームに氏名、メールアドレスなどを入力して送信すると T-Engine フォーラムからダウンロードページの ID とパスワードが発行されます。

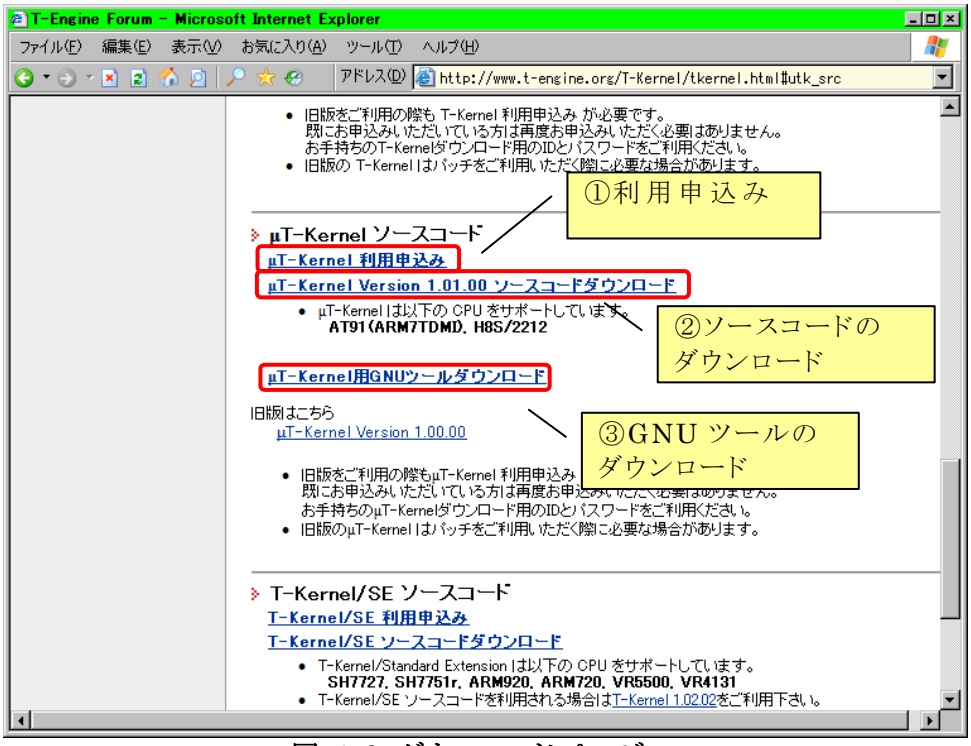


図 4-3 ダウンロードページ

次に②をクリックしてμT-Kernel ソースコードダウンロードページを開きます。この時 ID とパスワード入力画面が表示されるので T-Engine フォーラムから発行された ID とパスワードを入力してください。ソースコード utkernel.1.01.00.tar.gz を選択して保存します。

図 4-4 µT-Kernel ソースコードダウンロードページ

(2) Cygwin の入手とインストール

Cygwin は次の URL からダウンロードできます。

URL: http://www.cygwin.com/

確認に使用した各ソフトウェアのバージョンは次の通りです。

表 4-1 Cygwin ツールのバージョン

ソフトウェア	バージョン
cygwin1.dll	1.5.21
make	3.81
perl	5.8.7
bash	3.1.17

Web ページ上の Cygwin のアイコンをクリックするとインストーラが起動されます。指示に従ってインストールしてください。この時、パッケージ選択画面で図 4-5 のように devel と perl のパッケージを選択します。

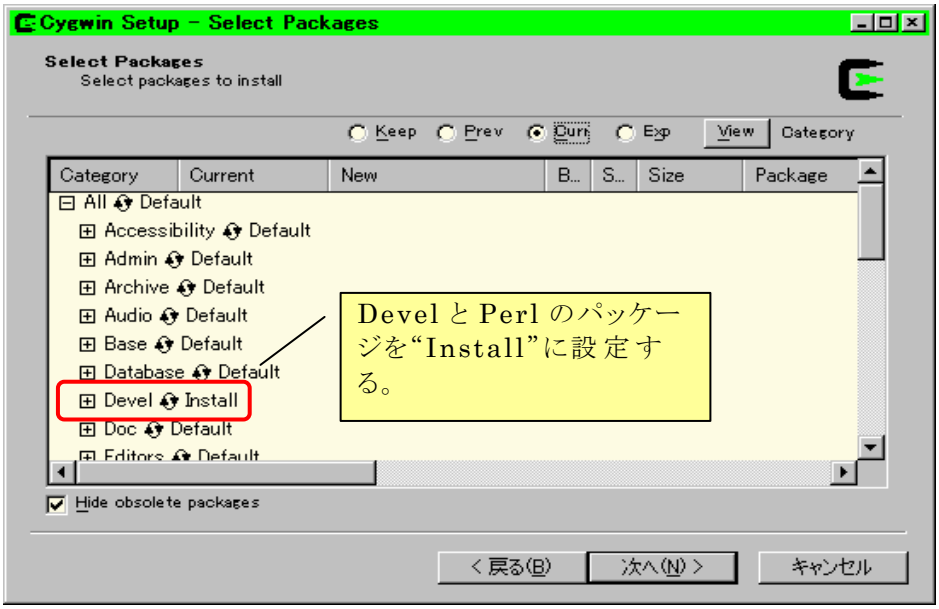


図 4-5 パッケージの選択

インストール後、Cygwin を起動し、/usr/local/bin に/usr/bin/perl へのシンボリックリンクを作成します。

```
$ ln -s /usr/local/bin /usr/bin/perl
```

(3) GNU ツールの入手とインストール

GNU ツールのダウンロードページは図 4-3 の③をクリックすると表示されます。ARM7TDMI または H8S/2212 の GNU ツールを選択して保存します。

図 4-6 GNU ツールダウンロードページ

保存した GNU ツールを/usr/local にコピーして展開すると GNU ツールがインストールされます。

```
ARM7TDMI の場合
$ cd /usr/local
$ tar xzf devenv_arm7tdmi.tgz
```

```
H8S/2212 の場合
$ cd /usr/local
$ tar xzf  devenv_h8300-elf-4.1.tgz
```

(4) 環境変数の設定

GNU ツールを実行するには次の環境変数を環境に合わせて設定する必要があります。.bashrc などに以下の内容を書き込みビルド前に環境変数が設定されるようにしてください。

【ARM7TDMI の設定例】

```
$ export BD=/usr/local/te/utkernel_source          μT-Kernel のソースコードを展開したディレクトリ
$ export GNUs=/usr                                 make があるディレクトリ
$ export GNU_BD=/usr/local/arm7tdmi                GNU ツールのベースディレクトリ
$ export GNUarm=$GNU_BD/arm-elf                    GNU 関連ツールのディレクトリ
$ export GCC_EXEC_PREFIX=$GNU_BD/lib/gcc-lib       gcc 関連ディレクトリ
```

【H8S/2212 の設定例】

```
$ export BD=/usr/local/te/utkernel_source
$ export GNUs=/usr
$ export GNU_BD=/usr/local/h8300-elf-4.1
$ export GNUh8300=$GNU_BD/h8300-elf
$ export GCC_EXEC_PREFIX=$GNU_BD/lib/gcc
```

(5) ビルド

(1)で入手したμT-Kernelのソースコードを/usr/local/te配下に展開します。

```
$ cd /usr/local
$ mkdir te
$ cd te
$ tar xzf utkernel.1.01.00.tar.gz
```

次にμT-Kernelをビルドします。ビルドが正常に終了するとmakeを行ったディレクトリに実行ファイル(kernel-rom.mot)が作成されます。

```
$ cd kernel/sysmain/build/app_at91
または
$ cd kernel/sysmain/build/app_h8s2212
$ make
…ビルド処理が表示される。
```

(6) ターゲットボードへの書き込みとデバッグ

作成した実行ファイルは、JTAGデバッガを使ってターゲットボードに書き込むことにより、デバッグを行うことができます。

図 4-7 GNU ツールダウンロードページ

4.1.2. SOFTUNE μT-REALOS/FR [FML]

SOFTUNE μT-REALOS/FR は、富士通マイクロエレクトロニクス製 FR マイコンに対応しているμT-Kernel 仕様の RTOS です。μT-REALOS/FR を利用したアプリケーションを開発するには、統合開発環境 SOFTUNE を使用します。統合開発環境 SOFTUNE は、言語ツール、デバッガツール、解析ツールを統合し、コーディング、コンパイル、デバッグを繰り返すアプリケーション開発を効率的に行えます。

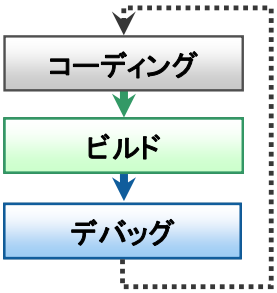


図 4-8 開発の流れ

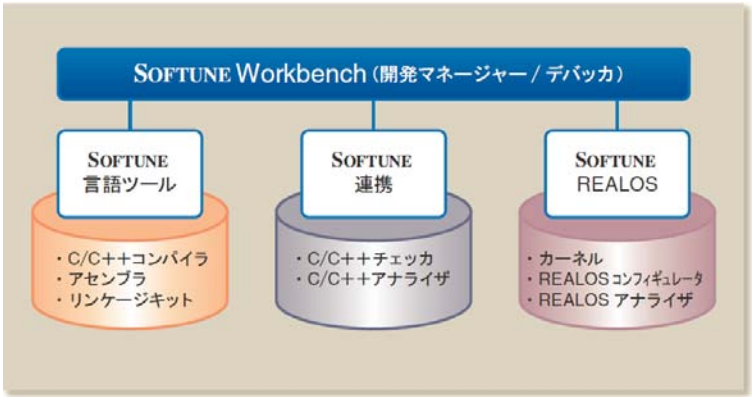


図 4-9 統合開発環境 SOFTUNE の構成

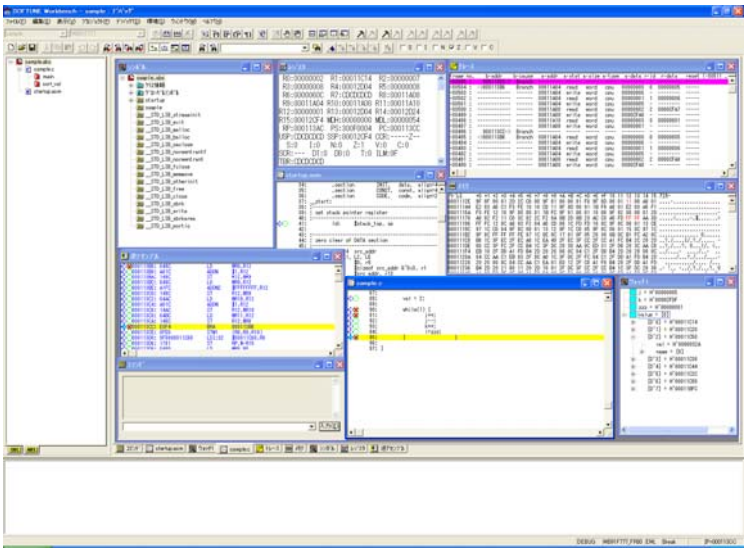


図 4-10 統合開発環境 SOFTUNE

(1) SOFTUNE μT-REALOS/FR 体験版の入手とインストール

SOFTUNE μT-REALOS/FR を使用するには、製品を購入する方法と体験版をダウンロードする方法の 2 通りあります。

体験版は製品版とは異なり、製品開発には使用できないことやサポートが受けられないなどの制限がありますが、手軽に試用することができます。使用許諾条件および制限事項の詳細については、ウェブページで確認してください。

体験版は、以下のサイトからダウンロードできます。

URL: http://jp.fujitsu.com/microelectronics/

(右下のバナー 

をクリックすると、ダウンロードサイトに移動できます。)

SOFTUNE μT-REALOS/FR をインストールする前に、統合開発環境 SOFTUNE をインストールする必要があります。同じダウンロードサイトから、評価版を入手できます。SOFTUNE μT-REALOS/FR のインストール方法については、ダウンロードサイトを参照するか、インストール説明書を参照してください。

(2) SOFTUNE μT-REALOS/FR の開発支援ツール

SOFTUNE μT-REALOS/FR では、アプリケーションを開発する時に有効な開発支援ツールを用意しています。

コンフィギュレータは、アプリケーションで使用するカーネルのオブジェクト数を最適に設定できるツールです。アプリケーション毎に使用するタスク数や機能は変わってきます。コンフィギュレータで使用するオブジェクト数および機能を指定することで、メモリ消費量の少ないカーネルが再構築できます。

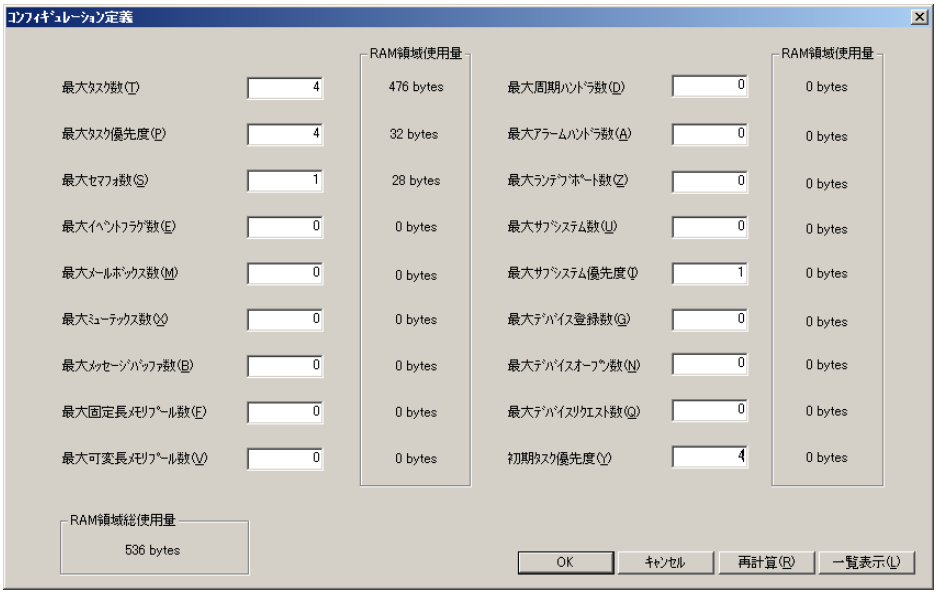


図 4-11 μT-REALOS コンフィギュレータ
(コンフィギュレーション定義)

その他にも、割込み番号とエントリ名を設定するだけの割込み登録機能、ユーザカスタマイズ可能な省電力モードの設定機能などがあります。

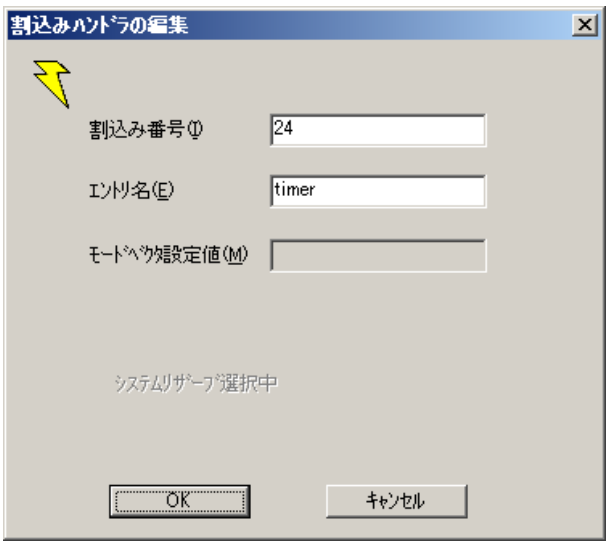


図 4-12 μT-REALOS コンフィギュレータ
（割込みハンドラの編集）

アナライザは、カーネルの状態をリアルタイムに解析し、タスク遷移図を表示するツールです。デバッグ支援ツールとして、またはチューニング支援ツールとして使うことができます。例えば、タスクやオブジェクトの状態が分かるオブジェクト状態表示機能やタスクのスタック使用量解析機能などを用意しています。

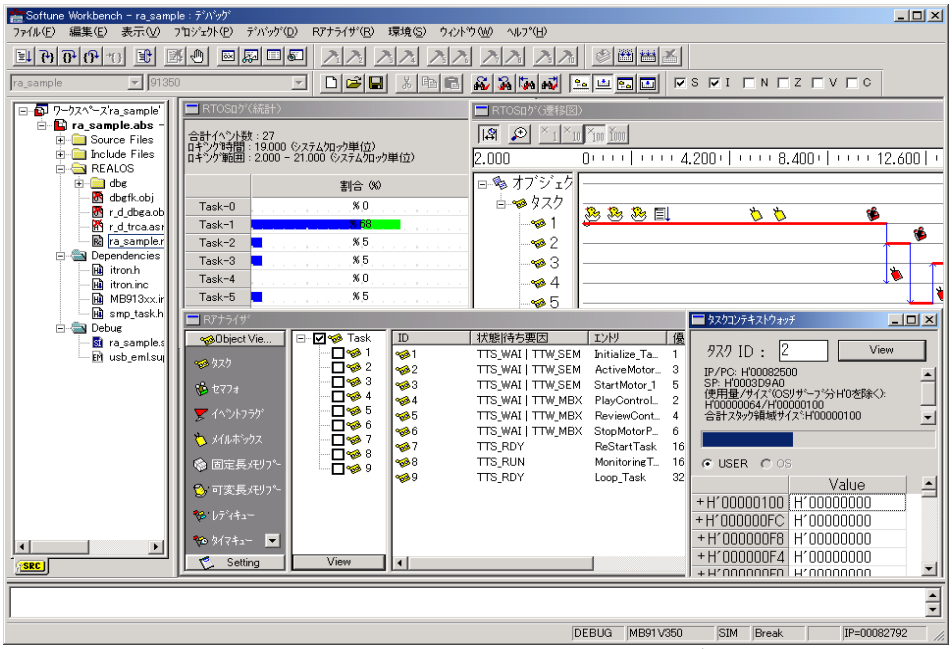


図 4-13 μT-REALOS アナライザ

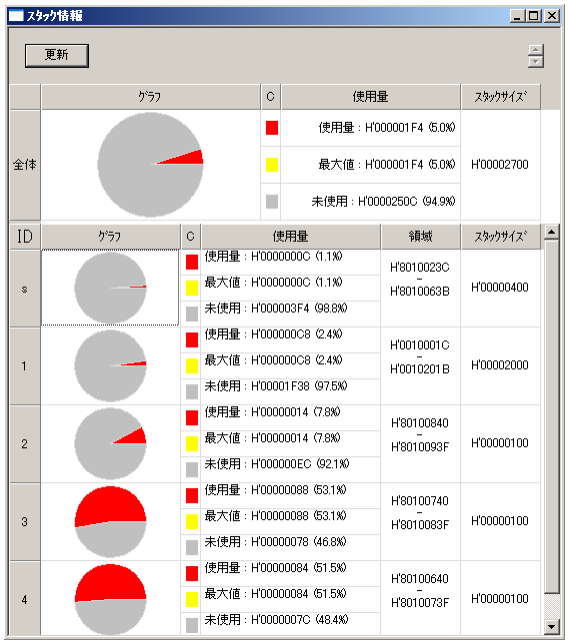


図 4-14 μT-REALOS アナライザ
（スタック情報表示機能）

(3) サンプルプログラムのビルド

SOFTUNE μT-REALOS/FR 体験版の中には、ビルドして動かすことが可能なサンプルプログラムがあります。なお、サンプルプログラムは、FR マイコンの MB91403 で動かすことを前提に作られています。

サンプルプログラムをビルドする方法は以下のとおりです。

1. 統合開発環境 SOFTUNE の起動

スタートメニューにある[Softune V6]の[FR Family Softune Workbench]をクリックすることで起動します。

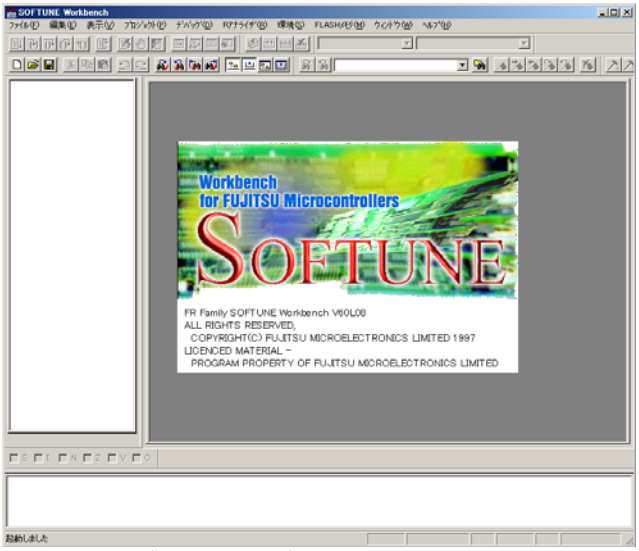


図 4-15 統合開発環境 SOFTUNE の起動画面

2. ワークスペースを開く

メニューの[ﾌｧｲﾙ(F)]－[ﾜｰｸｽﾍﾟｰｽを開く(R)]で、C:¥Softune6¥utkernel ¥911¥smpsys¥smpsys.wsp を指定します。

3. ビルド

サンプルプログラムはすぐにビルドできるようになっていますので、[ﾌﾟﾛｼﾞｪｸﾄ(P)]－[ﾋﾞﾙﾄﾞ(B)]でビルドすることができます。

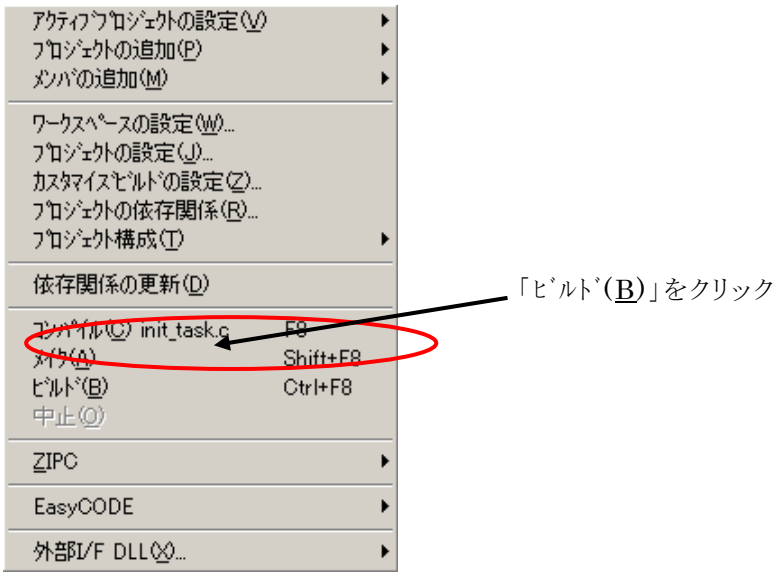


図 4-16 SOFTUNE のプロジェクトメニュー

(4) サンプルプログラムのシミュレータ環境でのデバッグ

シミュレータを使用してデバッグする方法を説明します。

1. デバッグの開始

メニューの[ﾃﾞﾊﾞｯｸﾞ(D)]－[ﾃﾞﾊﾞｯｸﾞの開始(D)]でデバッグを開始します。セットアップウィザードが立ち上がった場合は、[Simulator Debugger]を選択し、シミュレータ種別は[Nomal]を選択してください。

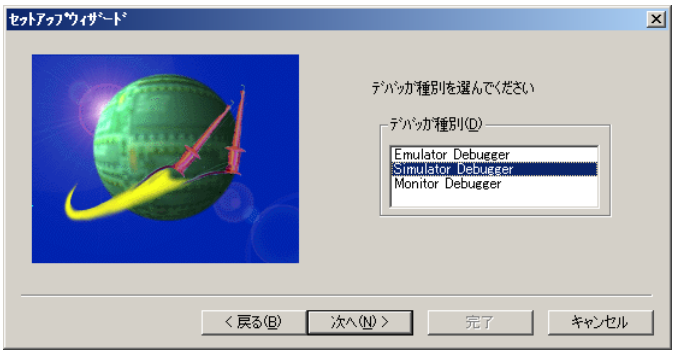


図 4-17 デバッガのセットアップウィザード(その 1)

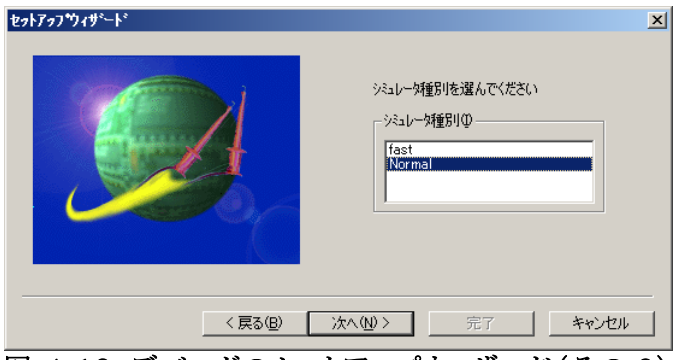


図 4-18 デバッガのセットアップウィザード(その 2)

2. ターゲットファイルのロード

ターゲットにロードするため、メニューの[ﾃﾞﾊﾞｯｸﾞ(D)]－[ﾀｰｹﾞｯﾄﾌｧｲﾙのﾛｰﾄﾞ(O)]でロードしてください。

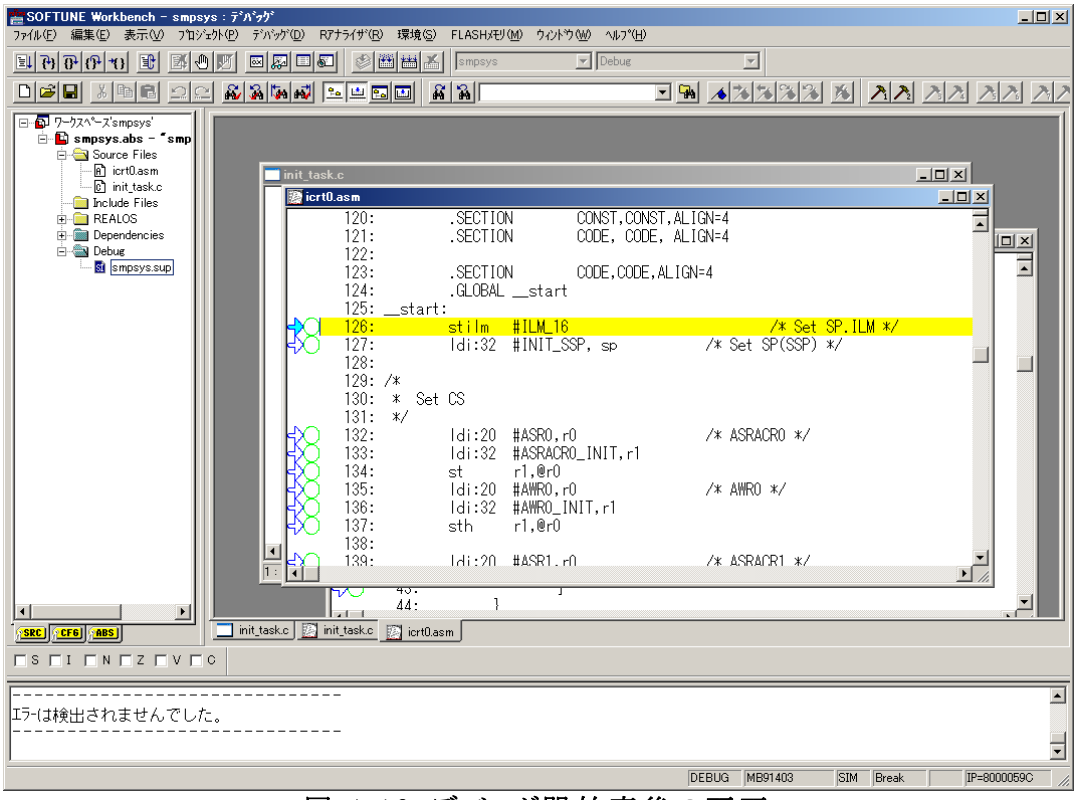


図 4-19 デバッグ開始直後の画面

3. デバッグ操作

一般的なデバッグ環境と同じように、ブレークを張るなどして、デバッグをすることができます。アナライザを起動する場合は、メニューの[R ｱﾅﾗｲｻﾞ(R)]－[R ｱﾅﾗｲｻﾞ(R)]で起動できます。

(5) サンプルプログラムの実機でのデバッグ

(4)の説明はシミュレータ環境を使用する方法でしたが、実機で動作確認したい場合は、FR マイコンのスターターキット「bits pot(ビッツ・ポット)」で動作させることができます。詳しくは、販売元(都築電産株式会社)にお問い合わせください。

都築電産株式会社: http://www.tsuzuki-densan.co.jp/bitspot/

μITRON3.0/4.0 仕様のリアルタイム OS を使用しているアプリケーションの移行方法について、富士通マイクロエレクトロニクス㈱で資料を用意していますので、ご相談ください。

4.2. 関連製品

ここではμITRON 仕様 OS で作成されたプログラムを T-Kernel で使えるようにするためのラッパーなどについて紹介します。

4.2.1. I-right/TK

I-right/TK(アイ・ライト・ティー・ケー)は、パーソナルメディア株式会社の開発した ITRON ラッパーであり、μITRON4.0 の API を T-Kernel 上に実現します。詳細は次のウェブサイトをご覧ください。

http://www.t-engine4u.com/products/iright_tk.html

I-right/TK の主な特長は次の通りです。

(A) ITRON の資産を T-Kernel で活用

これまで組込み向けに開発されてきた ITRON の豊富なプログラム資産を、最小限の修正により T-Engine プロジェクトの成果である T-Kernel 上で実行できます。また ITRON に慣れたエンジニアが、T-Kernel 上で新規のプログラムを開発する場合にも便利です。

(B) 軽量なラッパー

I-right/TK はμITRON4.0 と T-Kernel の仕様の差の吸収したり、ITRON のオブジェクト ID(固定 ID)と T-Kernel のオブジェクト ID(動的 ID)の変換を行うラッパーですが、ラッパー内部のオーバーヘッド(実行速度の低下およびコードサイズの増加)が最小限になるように工夫してあります。このため、ITRON と T-Kernel の持つリアルタイム性をどちらも損なうことなく、ITRON と T-Kernel の連携動作が可能になります。

(C) μITRON4.0 フルセットに準拠

ITRON の API を T-Kernel ベースのプログラム上から呼び出せます。ITRONAPI の仕様はμITRON4.0 仕様のフルセットにほぼ準拠しています。

(D) 固定的なオブジェクト ID 番号の指定や静的 API にも対応

T-Kernel ではタスク ID などのオブジェクト ID 番号は動的に決まるため、固定値では指定できませんが、I-right/TK では ITRON と同様に、固定的なオブジェクト ID 番号で指定可能です。また、μITRON4.0 仕様の静的 API にも対応しています。

(E) T-Kernel のミドルウェアやアプリケーションとの連携

ITRON 側から T-Kernel 上の豊富なデバイスドライバやミドルウェアを利用したり、T-Kernel 上のプログラムと連携動作を行うことが可能です。例えば T-Kernel 上で動いている TCP/IP やファイルシステム、グラフィック機能と、既存の ITRON のプログラムを組合せたシステムを容易に構築できます。

(F) 多くの CPU、多くの実行環境をサポート

T-Kernel は現在市販されている多くの CPU に対応済みであり、その成果が ITRON のプログラムからも利用できます。

またベンダー各社やユーザ自身により移植された T-Kernel やμT-Kernel に対応したカ

スタマイズ版の I-right/TK の提供も可能です。

(G) Eclipse による開発

統合開発環境「Eclipse」によるプログラム開発が可能です。

(H) 仮想環境でのシミュレーション

x86 系 CPU で動作する T-Kernel 製品「T-Kernel/x86」と組み合わせることにより、実機だけでなくパソコン上の仮想環境でのシミュレーション実行が可能です。

第5章　参考資料

[1] μT-Kernel 仕様書（Ver.1.01.00）　T-Engine フォーラム
[2] μITRON4.0 仕様 Ver. 4.03.03　T-Engine フォーラム
[3] μITRON3.0 仕様 Ver. 3.02.02　T-Engine フォーラム
[4] T-Kernel 標準ハンドブック 改訂新版　パーソナルメディア株式会社
[5] 組込みシステム実践プログラミングガイド~ITRON 仕様 OS/T-Kernel 対応~　技術評論社