TOPPERS新世代カーネルへのマイグレーションガイド

目次

最終更新: 2010年2月22日（作成中）

このドキュメントは，従来のTOPPERSカーネルや他のμITRON4.0仕様準拠のカーネルから，TOPPERS新世代カーネルに移行するための方法（またはヒント）を説明するものである．なお，μITRON4.0仕様とTOPPERS新世代カーネル仕様の違いについては，TOPPERS新世代カーネル統合仕様書の中に【μITRON4.0仕様との関係】として記載してある．必要に応じてそちらも参照すること．

データ型と定数の変更

TOPPERS新世代カーネル仕様では，ITRON仕様の次のデータ型を廃止している．
対応する新しいデータ型を用意しているので，単純に置き換えればよい．

廃止したデータ型	置き換えるデータ型
B	int8_t
UB	uint8_t

VB	uint8_t
H	int16_t
UH	uint16_t
VH	uint16_t
W	int32_t
UW	uint32_t
VW	uint32_t
D	int64_t
UD	uint64_t
VD	uint64_t
VP	void *
INT	int_t
UINT	uint_t
BOOL	bool_t
VP_INT	intptr_t

また，定数についても，次の通り置き換える必要がある．

廃止した定数	置き換える定数
TRUE	true
FALSE	false

なお，ITRON仕様のデータ型と定数を使いたいアプリケーションのために，
ITRON仕様との互換性を保つためのitron.hを用意しているが，使用することは 推奨しない．

オブジェクト属性の変更

TOPPERS新世代カーネル仕様では，次のオブジェクト属性は，デフォルト扱いに して廃止している．

TA_HLNG	高級言語用インタフェース
TA_TFIFO	タスクの待ち行列をFIFO順に
TA_MFIFO	メッセージキューをFIFO順に
TA_WSGL	待ちタスクは1つのみ

これらのオブジェクト属性を指定している場合には，単純に削除すればよい．

削除したことにより，指定するオブジェクト属性がなくなる場合には，TA_NULLを指定する．なお，ITRON仕様のオブジェクト属性を使いたいアプリケーションのために，ITRON仕様との互換性を保つためのitron.hを用意しているが，使用することは 推奨しない．

システムコンフィギュレーションファイルの変更

TOPPERS新世代カーネル仕様では，システムコンフィギュレーションファイルにおけるC言語プリプロセッサのディレクティブの扱いが変更になっているため，システムコンフィギュレーションファイルの変更が必要になる．

TOPPERS新世代カーネル仕様では，システムコンフィギュレーションファイル中に記述できるディレクティブが，インクルードディレクティブ（#include）と条件ディレクティブ（#if，#ifdefなど）のみとなっている．そのため，システムコンフィギュレーションファイルにマクロ定義ディレクティブ（#define）が含まれている場合には，その記述を削除し，必要であれば，#includeディレクティブでインクルードするファイル中に移動する必要がある．

μITRON4.0仕様に準拠して記述されたシステムコンフィギュレーションファイル中の#includeディレクティブは，INCLUDEディレクティブに書き換える必要がある．INCLUDEディレクティブでインクルードするファイルは，システムコンフィギュレーションファイルの一部とみなされるため，上記と同じ制限がある．逆に，μITRON4.0仕様に準拠して記述されたシステムコンフィギュレーションファイル中のINCLUDE静的APIは，#includeディレクティブに書き換える必要が ある．

オブジェクト**ID**の自動割付けへの対応

TOPPERS新世代カーネルでは，オブジェクトのID番号を自動割付けすることが基本となっている．TOPPERS新世代カーネルを用いる際のID番号の管理方法については，「TOPPERS/ASPカーネルユーザーズマニュアル」の「11.3 オブジェクトIDの管理」の節に説明があるので，まずはこれを参照すること．従来，ID番号を手動で割り付けており，手動で割り付けていたID番号を変更したくない場合には，コンフィギュレータの持つID番号の割付けをファイルから取り込む機能（--id-input-fileオプション）を用いることで，手動で割り付けたID番号を用いることができる．この場合に，手動で割り付けたID番号は，コンフィギュレータが—id-input-fileオプションで取り込むファイル中にのみ記述し，アプリケーションのソースプログラムからは，kernel_cfg.hを用いること が望ましい．

ヘッダファイルの変更

μITRON4.0仕様においては，コンフィギュレータが生成する自動割付け結果ヘッダファイルの名称がkernel_id.hであったが，TOPPERS新世代カーネル仕様では，kernel_cfg.hに変更になっている（kernel_cfg.hの方が含まれる定義が多い）．kernel_id.hをインクルードしていたアプリケーションは，kernel_cfg.hをインクルードするように変更する．

TOPPERS/JSPカーネルで，アプリケーションが用いる標準的なヘッダファイルと

して用意していたt_services.hとs_services.hは，TOPPERS新世代カーネルでは 用意されていない．t_services.hをインクルードしていたアプリケーションは，それに代えて，kernel.hをインクルードするように変更する．また，必要に応じて，t_syslog.h，t_stdlib.h，syssvc/syslog.h，syssvc/serial.hをインクルード する．ただし，t_services.hに含まれているsyscallマクロと_syscallマクロは，
アプリケーションによって適切なエラー処理方法は異なることから，TOPPERS新世代カーネルでは用意されていない．使用する場合には，アプリケーションで 用意する必要がある．s_services.hをインクルードしていたアプリケーションは，それに代えて，sil.hをインクルードするように変更する．また，必要に応じて，t_syslog.hとターゲットのハードウェア資源の定義を含むヘッダファイルをインクルードす る．

sta_taskの置き換え

TOPPERS新世代カーネルでは，タスクを起動するサービスコールとしてact_task をサポートしており，sta_taskをサポートしていない．act_taskとsta_taskの機能
を比較した場合，前者はタスク起動のキューイング機能を持つのに対して，後者はタスクに起動コードを渡す機能を持つ．そのため，sta_taskをact_taskに置き換える場合に，起動コードを渡す機能をどのように実現するかが問題となる．
起動コードを渡す機能を最も簡単に代用する方法は，起動コードを渡すためのデータキューを用意する方法である．タスクを起動する処理単位は，データキューに起動コードを送信した後，act_taskによりタスクを起動する．起動されたタスクは，データキューから起動コードを受信する．
タスク起動のキューイングが起こらないことが保証できる，言い換えると，タスクを起動する時には，対象タスクは休止状態にあることが保証できる場合には，起動コードをグローバル変数に置いて渡す方法もある．タスクを起動する処理単位がそのグローバル変数に書くのは，タスクが休止状態の間に限られ，起動されたタスクがそのグローバル変数を読むのは，タスクが実行できる状態の間に限られるため，グローバル変数に対する排他制御は必要ない． ○set_timの置き換え
TOPPERS新世代カーネルでは，システム時刻を設定するサービスコールであるset_timが使用されることは稀であると考え，サポートしないこととした． ITRON仕様と互換のset_timとget_timが必要な場合には，下に示すコードの itron_set_timとitron_get_timで代用することができる（下のコードでは，エ ラー処理は省略している）．

```
static SYSTIM  systim_offset = 0U;
void
itron_set_tim(const SYSTIM *p_systim)
{
    SYSTIM current_time;
    get_tim(&current_time);
    systim_offset = *p_systim - current_time;
}

void
itron_get_tim(SYSTIM *p_systim)
{
    SYSTIM current_time;
    get_tim(&current_time);
    *p_systim = systim_offset + current_time;
}
```

48ビットのシステム時刻

ITRON仕様準拠のカーネルでは，μITRON3.0仕様でシステム時刻を48ビットとすることを推奨していたため（μITRON4.0仕様では推奨を定めていない），システム時刻が48ビットとなっているものがある．int型，long型ともに32ビットの環境で，システム時刻を48ビットに拡張するには，カーネルを改造する方法も考えられるが，get_timのみが必要な場合には，周期ハンドラを使って上位桁を求めておく方法がある．
具体的には，まず，上位桁を求める周期ハンドラを登録するために，システムコンフィギュレーションファイルに次の記述を含める．

```
CRE_CYC(CYCHDR_SYSTIM48, { TA_STA, 0, cychdr_systim48, 1 << 30, 1 << 30 });
```

周期ハンドラ本体およびそれを用いた48ビット版のget_timは，次のように実現 することができる．

```
typedef {
    uint_16   utime;   /* システム時刻の上位16ビット */
    uint_32   ltime;   /* システム時刻の下位32ビット */
} SYSTIM48;

static SYSTIM  systim_upper = 0U;

void
cychdr_systim48(intptr_t exinf)
{
    systim_upper += 1;
}

void
itron_get_tim48(SYSTIM48 *p_systim48)
{
    SYSTIM systim;
    get_tim(&systim);
    if (((systim >> 30) & 0x3U) == (systim_upper & 0x3U) {
        p_systim48->utime = (uint_16)(systim_upper >> 2);
    }
    else {
        p_systim48->utime = (uint_16)((systim_upper >> 2) + 1);
    }
    p_systim48->ltime = (uint_32) systim;
}
```

int型が16ビットの環境では，周期ハンドラの周期として(1 << 30)を使用することができないため，周期ハンドラの登録と周期ハンドラ本体を，次のように 修正する必要がある．

```
CRE_CYC(CYCHDR_SYSTIM48, { TA_STA, 0, cychdr_systim48, 1 << 15, 1 << 15 });

static SYSTIM  systim_upper = 0U;
static SYSTIM  systim_medium = 0U;

void
cychdr_systim48(intptr_t exinf)
{
    systim_medium += 1;
    if (systim_medium == 0U) {
        systim_upper += 1;
    }
}
```

周期ハンドラの非互換性

TA_PHS属性でない周期ハンドラにおいて，sta_cycを呼び出した後，最初に周期ハンドラが起動される時刻が，μITRON4.0仕様では，sta_cycを呼び出してから周期ハンドラの起動周期（cyctim）で指定した相対時間後となっていたが，TOPPERS新世代カーネルでは，起動位相（cycphs）で指定した相対時間後とした．CRE_CYCにおいて，TA_STA属性を指定しない場合には，cycphsにcyctimと同じ値を指定することで，μITRON4.0仕様と同じ振舞いとなる（μITRON4.0仕様では，TA_STA属性とTA_PHS属性のいずれも指定しない場合には，cycphsは意味を持たない）．TA_STA属性を指定する場合には，カーネルの起動後，最初に周期ハンドラが起動されるまでの相対時間と，sta_cycを呼び出してから最初に周期ハンドラが起動されるまでの相対時間が同一でよければ，その時間をcycphsに指定すればよい．両者が同一では不都合な場合には，μITRON4.0仕様と同じ振舞いをさせることはできない．代替手段としては，以下の2つが考えられる．

1. TA_STA属性を使用せず，カーネルの起動後適切なタイミングでsta_cycを呼び出すことで，周期ハンドラを動作開始する．
2. 周期ハンドラを2つ用いる．1つをTA_STA属性とし，sta_cycで動作を制御するのをもう片方の周期ハンドラとする．

可変長メモリプールの置き換え

アプリケーションが動的メモリ管理を用いる場合に，malloc/freeの実現に可変長メモリプールが用いられることがあり，可変長メモリプールがサポートされていないことが問題となる場合がある．TOPPERS新世代カーネル仕様で可変長メモリプールをサポートしないこととしたのは，動的メモリ管理をカーネル内で実現するより，ライブラリとして実現する方が適切と考えたためである．
そこで，ここでは，アプリケーションが用いるmalloc/freeを，オープンソースのメモリ割付けライブラリであるTLSFを用いて実現する方法について述べる．
TLSFは，リアルタイムシステム向けの効率的なメモリ割付けライブラリである．TLSFのライセンス条件は，GPLとLGPLのデュアルライセンスであるが，TLSFを通常のライブラリとして用いて実装されたプログラムは，TLSFの派生物とは見なされず，GPLが適用されないことが明記されている．
TLSFは，以下のウェブサイトからダウンロードすることができる．http://www.gii.upv.es/tlsf/（動作確認は，Version 2.4.6）のアーカイブの中から，tlsf.hとtlsf.cを，アプリケーションまたはライブラリのソースファイルの置かれたディレクトリにコピーする．tlsf.hは動的メモリ管理を用いるアプリケーションからインクルードすべきヘッダファイル，tlsf.cは動的メモリ管理ライブラリの本体である．tlsf.cには以 下のようなパッチをあてる．

'patch -c tlsf.c < tlsh.patch'

```
*** tlsf.c.orig 2016-08-05 13:40:24.000000000 +0900
--- tlsf.c  2016-08-05 13:42:00.000000000 +0900
***************
*** 76,82 ****
```

```
  #if TLSF_USE_LOCKS
! #include "target.h"
  #else
  #define TLSF_CREATE_LOCK(_unused_)   do{}while(0)
  #define TLSF_DESTROY_LOCK(_unused_)  do{}while(0)
--- 76,88 ----


  #if TLSF_USE_LOCKS
! #include "kernel.h"
! #include "kernel_cfg.h"
! #define TLSF_MLOCK_T ID
! #define TLSF_CREATE_LOCK(lock)   (*lock = TLSF_SEM)
! #define TLSF_DESTROY_LOCK(lock)  ini_sem(*lock)
! #define TLSF_ACQUIRE_LOCK(lock)  wai_sem(*lock)
! #define TLSF_RELEASE_LOCK(lock)  sig_sem(*lock)
  #else
  #define TLSF_CREATE_LOCK(_unused_)   do{}while(0)
  #define TLSF_DESTROY_LOCK(_unused_)  do{}while(0)
***************
*** 169,176 ****

  #ifdef USE_PRINTF
  #include <stdio.h>
! # define PRINT_MSG(fmt, args...) printf(fmt, ## args)
! # define ERROR_MSG(fmt, args...) printf(fmt, ## args)
  #else
  # if !defined(PRINT_MSG)
  #  define PRINT_MSG(fmt, args...)
--- 175,184 ----

  #ifdef USE_PRINTF
  #include <stdio.h>
! #include <t_stddef.h>
! #include <t_syslog.h>
! # define PRINT_MSG(fmt, args...) syslog(LOG_ERROR, fmt, ## args)
! # define ERROR_MSG(fmt, args...) syslog(LOG_ERROR, fmt, ## args)
  #else
  # if !defined(PRINT_MSG)
  #  define PRINT_MSG(fmt, args...)
```

このパッチは，tlsf.cに以下の修正を加えている． * PRINT_MSGおよびERROR_MSGを，syslogを用いるように変更する． * タスク間の排他を，セマフォを用いて行う．
malloc/freeを複数のタスクから呼び出す場合には，tslf.cを-DTLSF_USE_LOCKS
オプションをつけてコンパイルし，システムコンフィギュレーションファイル に次の記述を追加する．

```
CRE_SEM(TLSF_SEM, { TA_TPRI, 1, 1 });
```

TLSFを使用するプログラムでは，メモリプールの領域を配列として確保する
（下の例は，メモリプールのために10KBの領域を確保している）．

```
#define MEMORY_POOL_SIZE  (TOPPERS_ROUND_SZ(10*1024, sizeof(intptr_t))
/* 10*1024の部分は，適切なサイズに変更する */
intptr_t memory_pool[MEMORY_POOL_SIZE / sizeof(intptr_t)];
```

次にメモリプールの初期化を行う．

```
init_memory_pool(MEMORY_POOL_SIZE, memory_pool);
```

以上により，tlsf_malloc/tlsf_freeで，malloc/freeが実現できる．　なお，
TLSFは，複数のメモリプールを用いる機能や，メモリプールのサイズを
拡張する機能を持つ．詳しくは，TLSFのアーカイブ中のREADMEを参照すること． 以上