

μ T-Kernel3.0

TCP/IPプロトコルスタック [M3S-T4-Tiny]

構築手順書

Version. 01. 00. 09

2024. 9. 13

目次

| | |
|-------------------------------------|----|
| 1. 概要 | 4 |
| 1.1 目的 | 4 |
| 1.2 実装の基本方針 | 4 |
| 1.3 バージョン情報 | 4 |
| 1.4 関連ドキュメント | 4 |
| 1.5 ソースコード構成 | 5 |
| 1.6 プロジェクト・ファイル | 5 |
| 1.7 イーサネット未実装ボードへの対応 | 6 |
| 2. T4 が使用する資源 | 8 |
| 2.1 使用する資源の概要 | 8 |
| 2.2 タスク (ether_tsk) | 8 |
| 2.3 周期ハンドラ (ether_cychdr) | 8 |
| 2.4 割込みハンドラ (ether_eint0) | 9 |
| 2.5 グループ割込み (GroupAL1Handler) | 9 |
| 2.6 割込みハンドラ (ether_pmgioi) | 10 |
| 2.7 選択型割込み A (PERIA) | 10 |
| 3. T4 のコンフィグレーション | 11 |
| 3.1 T4 のコンフィグレーション・ファイル | 11 |
| 3.2 タスクの優先度と割込みハンドラの割込み優先度 | 11 |
| 3.3 DHCP 関連 | 11 |
| 3.4 TCP 受付け口のポート番号関連 | 13 |
| 3.5 TCP 通信端点関連 | 13 |
| 3.6 その他の TCP 関連 | 14 |
| 3.7 UDP 通信端点関連 | 15 |
| 3.8 その他の UDP 関連 | 16 |
| 4. API 仕様 | 17 |
| 4.1 lan_open | 17 |
| 4.2 lan_close | 18 |
| 4.3 tcpudp_open | 19 |
| 4.4 tcpudp_close | 20 |
| 4.5 tcpudp_reset | 21 |
| 4.6 tcpudp_get_ramsize | 22 |
| 4.7 tcp_acp_cep | 23 |
| 4.8 tcp_con_cep | 24 |
| 4.9 tcp_sht_cep | 26 |
| 4.10 tcp_cls_cep | 27 |

| | | |
|-------|-------------------------------|----|
| 4.11 | tcp_snd_dat | 28 |
| 4.12 | tcp_rcv_dat | 29 |
| 4.13 | udp_snd_dat | 30 |
| 4.14 | udp_rcv_dat | 32 |
| 5. | スタックサイズ | 34 |
| 5.1 | スタック見積もりツール | 34 |
| 5.2 | ether_tsk のスタックサイズ | 34 |
| 5.3 | ether_cychdr のスタックサイズ | 34 |
| 5.4 | ether_eint0 のスタックサイズ | 35 |
| 5.5 | PMGI 対応版のスタックサイズ | 36 |
| 6. | サンプルプログラム | 37 |
| 6.1 | サンプルプログラムの概要 | 37 |
| 6.2 | クライアント側のサンプルプログラム | 38 |
| 6.2.1 | usermain 関数 | 38 |
| 6.2.2 | client_tsk | 38 |
| 6.2.3 | コンフィグレーション・ファイル（主要項目のみ） | 39 |
| 6.3 | サーバー側のサンプルプログラム | 40 |
| 6.3.1 | usermain 関数 | 40 |
| 6.3.2 | server_tsk | 41 |
| 6.3.3 | コンフィグレーション・ファイル（主要項目のみ） | 41 |
| 6.4 | 実行結果 | 42 |
| 6.5 | NTP クライアントのサンプルプログラム | 43 |
| 6.5.1 | usermain 関数 | 43 |
| 6.5.2 | ntp_client_tsk | 44 |
| 6.5.3 | コンフィグレーション・ファイル（主要項目のみ） | 45 |
| 7. | 問い合わせ先 | 45 |

1. 概要

1.1 目的

本構築手順書はルネサスエレクトロニクス社のRX用に改変した μ T-Kernel3.0 (V3.00.00)のソースコードにルネサスエレクトロニクス社のTCP/IPプロトコルスタック【M3S-T4-Tiny】を実装するための手順を記載した仕様書です。

1.2 実装の基本方針

本実装の基本方針を以下に示します。

- TCP/IPプロトコルスタック【M3S-T4-Tiny】（以降、本書ではT4と記す）のソースコードやAPI仕様は極力変更しない。 μ T-Kernel3.0対応に変更が必要なもののみリコーディングを行う。
- APIはブロッキングインタフェースで提供し、 μ T-Kernel3.0の機能により待ち状態を使ってAPIの完了待ちを行う。
- TCP/IPのプロトコル処理はタスクとして実装する。ただし、 μ T-Kernel3.0仕様におけるサブシステム管理機能は使用せず、単なるタスクとして実装する。

1.3 バージョン情報

本実装に利用したT4のバージョン情報は以下の通りです。

Ver.: V.2.10 Release 00

リリース日: 2021/4/19

1.4 関連ドキュメント

以下に本構築仕様書の関連するドキュメントを示します。

| 分類 | 名称 | 発行 |
|------------------|--|------------------------------------|
| OS仕様 | μ T-Kernel3.0仕様書 (Ver. 3.00.00) | TRONフォーラム TEF020-S004-3.00.00 |
| T-Monitor | T-Monitor仕様書 | TRONフォーラム TEF-020-S002-01.00.01 |
| 共通実装仕様 | uTK3.0共通実装仕様書 | |
| ハードウェア 依存実装仕様 | uTK3.0_AP-RX63N-0A実装仕様書 | |
| | uTK3.0_AP-RX65N-0A実装仕様書 | |
| | uTK3.0_AP-RX72N-0A実装仕様書 | |
| | uTK3.0_GR-SAKURA実装仕様書 | |
| | uTK3.0_TB-RX65N実装仕様書 | |
| | uTK3.0_TB-RX66N実装仕様書 | |
| | uTK3.0_EK-RX72N実装仕様書 | |
| 構築手順 | uTK3.0_AP-RX63N-0A構築手順書 | |

| | | |
|--|-------------------------|--|
| | uTK3.0_AP-RX65N-0A構築手順書 | |
| | uTK3.0_AP-RX72N-0A構築手順書 | |
| | uTK3.0_GR-SAKURA構築手順書 | |
| | uTK3.0_TB-RX65N構築手順書 | |
| | uTK3.0_TB-RX66N構築手順書 | |
| | uTK3.0_EK-RX72N構築手順書 | |

| 分類 | 名称 | 発行 |
|-------|---|---------------------------------|
| マニュアル | 組み込み用TCP/IP M3S-T4-Tiny Ethernetドライバインタフェース仕様書 | ルネサスエレクトロニクス R20UW0032JJ0109 |
| | 組み込み用TCP/IP M3S-T4-Tiny ユーザズマニュアル | ルネサスエレクトロニクス R20UW0031JJ0111 |
| | 組み込み用TCP/IP M3S-T4-Tiny導入ガイド | ルネサスエレクトロニクス R20AN0051JJ0210 |

1.5 ソースコード構成

本実装のソースコードのディレクトリ構成を以下に示します。

| | |
|---------------|----------------------|
| └ device | デバイスドライバ・ミドルウェア |
| └ ether | TCP/IPプロトコルスタック |
| ├ └ config | コンフィグレーション |
| ├ └ sysdepend | 実装依存定義 |
| ├ └ src | ソースコード |
| └ include | アプリケーション用のインクルードファイル |

機種依存定義 sysdepend ディレクトリは以下のように構成されます。

| | |
|-------------|------------|
| └ sysdepend | 実装依存定義 |
| ├ <ターゲット1> | ターゲット1 依存部 |
| ├ : | |
| └ <ターゲットn> | ターゲットn 依存部 |

1.6 プロジェクト・ファイル

本実装を利用するためのプロジェクト・ファイルは「mtkernel_3/ide/cs」のフォルダにあります。

アイコンの名称が「****_T4.mtpj」となっているものがT4を利用可能なμT-Kernel3.0のプロジェクト・ファイルです。ダブルクリックすればCS+が起動されます。



図1.1 T4対応のプロジェクト・ファイル（上記以外もあります）

1.7 イーサネット未実装ボードへの対応

TB-RX65NやTB-RX66Nはイーサネットが未実装です。これらのターゲットに関してはLAN8720 ETH Boardをジャンパで接続して対応します。以下、LAN8720 ETH Boardとの接続を示します。

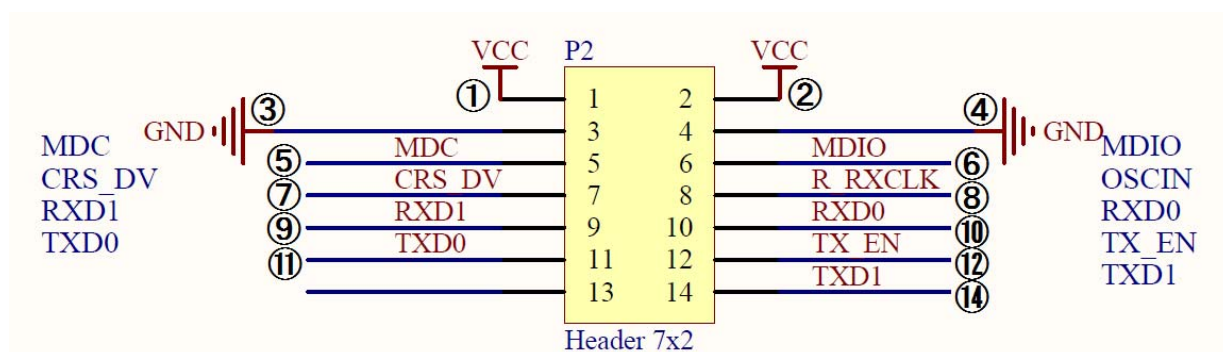


図1.2 LAN8720 ETH Boardの端子番号

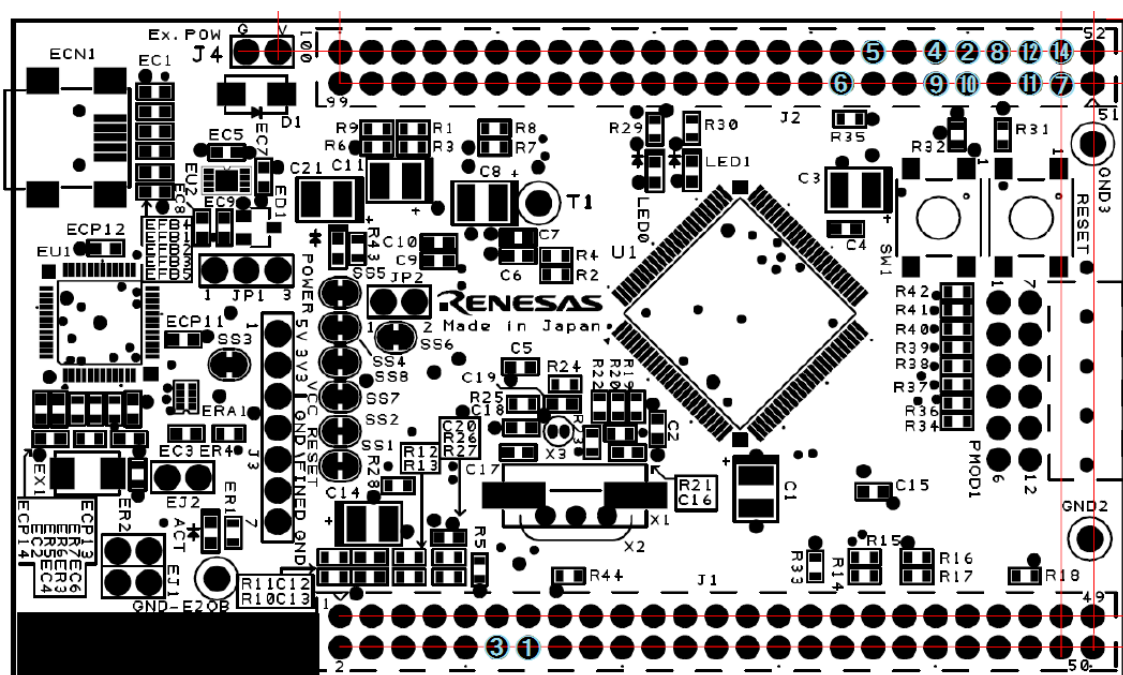


図1.3 TB-RX65Nの接続先

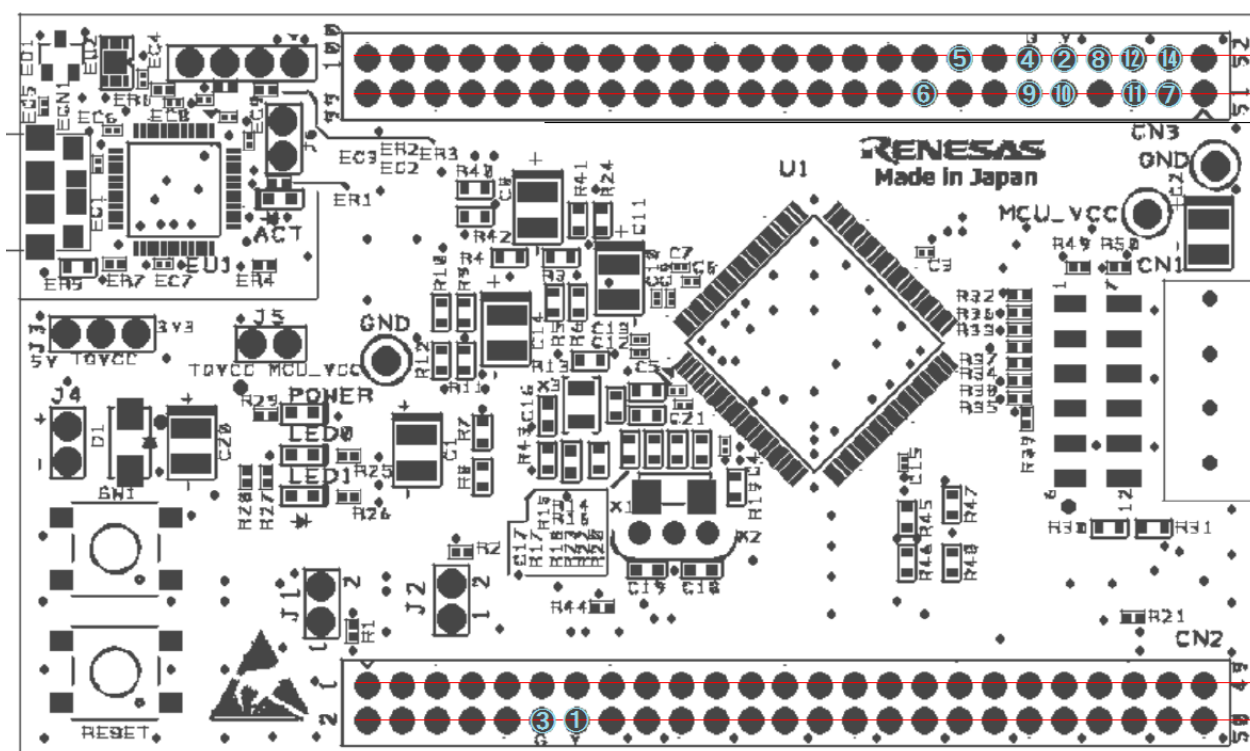


図1.4 TB-RX66Nの接続先

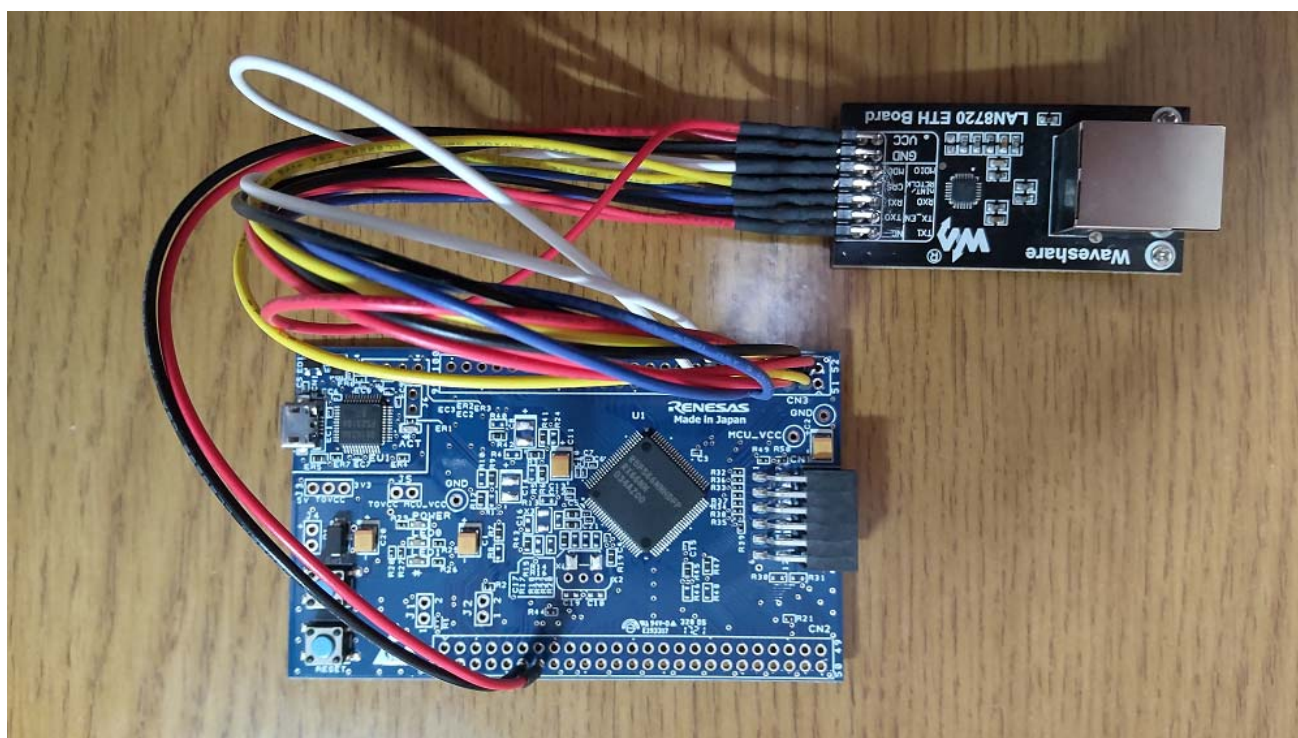


図1.5 接続イメージ

2. T4 が使用する資源

2.1 使用する資源の概要

T4はTCP/IPプロトコル処理を実施するに当たり、以下の2つの μ T-Kernel3.0のオブジェクトと1つの割り込みを利用します。これらの資源は第5章で紹介するlan_open関数のAPIを呼び出すと生成または定義されます。

| 使用する資源 | 名称 | 優先度 |
|----------|--------------|--|
| タスク | ether_tsk | タスクの優先度は ETHER_CFG_TASK_PRIORITY |
| 周期ハンドラ | ether_cychdr | 周期ハンドラの割り込み優先度は TIM_INT_PRI |
| 割り込みハンドラ | ether_eint0 | 割り込みハンドラの割り込み優先度は ETHER_CFG_INT_PRIORITY |
| | ether_pmg0i | 割り込みハンドラの割り込み優先度は ETHER_CFG_INT_PRIORITY |

注：ETHER_CFG_TASK_PRIORITY と ETHER_CFG_INT_PRIORITY は次節を参照

TIM_INT_PRI は各ターゲットの実装仕様書を参照

ether_pmg0i はターゲットがEK-RX72Nの時のみ定義されます

2.2 タスク (ether_tsk)

TCP/IPのプロトコル処理を行うタスクです。ユーザアプリケーションからのAPIの受け付け、LANソケット（イーサコントローラ）から割り込み、タイムアウト等による再送制御を行います。

| 項目 | 値 |
|------------|----------------------------------|
| 拡張情報 | 0 |
| タスク属性 | TA_HLNG TA_DSNAME TA_USERBUF |
| タスク起動アドレス | ether_tsk |
| タスク起動時優先度 | ETHER_CFG_TASK_PRIORITY |
| スタックサイズ | 512 |
| DSオブジェクト名称 | "ether_t" |

備考：

TA_DSNAME、TA_USERBUFのタスク属性はカーネルのコンフィグレーションによって未サポートとなる場合があります。

2.3 周期ハンドラ (ether_cychdr)

10ms周期で起動し、ether_tskを起床する周期ハンドラです。

| 項目 | 値 |
|----------|---------------------|
| 拡張情報 | 0 |
| 周期ハンドラ属性 | TA_HLNG TA_DSNAME |

| | |
|--------------|--------------|
| 周期ハンドラ起動アドレス | ether_cychdr |
| DSオブジェクト名称 | "ether_c" |
| 起動周期 | 10ms |
| 起動位相 | 10ms |

備考：

TA_DSNAMEの周期ハンドラ属性はカーネルのコンフィグレーションによって未サポートとなる場合があります。

2.4 割込みハンドラ (ether_eint0)

イーサコントローラ (ETHERC) の割込みハンドラです。割込みの発生によりether_tskを起床します。

| 項目 | ターゲット | 値 |
|---------------|-------------|------------------------|
| 割込み番号 | AP-RX63N-0A | 32 (Ether EINT) |
| | AP-RX65N-0A | 113 (GROUPAL1) |
| | AP-RX72N-0A | 113 (GROUPAL1) |
| | GR-SAKURA | 32 (Ether EINT) |
| | TB-RX65N | 113 (GROUPAL1) |
| | TB-RX66N | 113 (GROUPAL1) |
| | EK-RX72N | 113 (GROUPAL1) |
| 割込み優先度 | — | ETHER_CFG_INT_PRIORITY |
| 割込みハンドラ起動アドレス | — | ether_eint0 |

ether_eint0割込みハンドラでは、関数ポインタ経由でlan_inthdr関数が呼び出されます。スタックサイズを算出する際はlan_inthdr関数が呼び出されることを加味してください。詳細は第6章で説明します。

2.5 グループ割込み (GroupAL1Handler)

イーサコントローラ (ETHERC) の割込みがグループ割込みの場合 (RX65NやRX72N等の場合)、割込み番号 (ベクタ番号) 113の割込みハンドラを定義することになります。ただし、システムで割込み番号 113のグループ割込みに分類された割込みを利用する場合、イーサコントローラの割込みと共存する必要があります。

現在のether_subsystem.cのサンプルでは、このグループ割込みに対して、デバイスドライバ共通のグループ割込みハンドラ (GroupAL1Handler) を利用しています。ソースファイルはgroupal1.cです。グループ割込みハンドラでは関数ポインタ経由で割込み要求に対応した割込みハンドラを呼び出します。

GroupAL1Handlerグループ割込みハンドラ

```
EXPORT void (*GroupAL1Table[32])(UINT dintno);
EXPORT void GroupAL1Handler(UINT dintno)
{
    INT i;
    UW intreqflg;
    intreqflg = ICU.GRPAL1.LONG;
    for( i=0 ; !( intreqflg & 0xFF ) ; i+=8 )
        intreqflg >>= 8;
    for( ; intreqflg ; i++, intreqflg >>= 1 )
        if( intreqflg & 1 )
            GroupAL1Table[i]( dintno );
}
```

GroupAL1Handlerグループ割込みハンドラの中でGRPAL1レジスタの各ステータスフラグをチェックし、各割込みハンドラを呼び出します。

2.6 割込みハンドラ (ether_pmg0i)

PHYマネージメントインタフェース (PMGI) の割込みハンドラです。T4内部でPHY-LSIの操作時に利用されます。この割込みハンドラはPMGIを内蔵し、なお且つPHY-LSIとのインタフェースにMIIを利用している場合のみ利用されます。現状サポートしているターゲットではEK-RX72Nのみとなります。

| 項目 | ターゲット | 値 |
|---------------|----------|------------------------|
| 割込み番号 | EK-RX72N | 208 (PERIA INTA208) |
| 割込み優先度 | — | ETHER_CFG_INT_PRIORITY |
| 割込みハンドラ起動アドレス | — | ether_pmg0i |

ether_pmg0i割込みハンドラでは、関数ポインタ経由でpmgi0_callback関数を呼び出しますが、スタック使用量は4バイトと極小なのでスタックサイズを算出する際はpmgi0_callback関数の呼び出しは無視して構いません。詳細は第6章で説明します。

2.7 選択型割込みA (PERIA)

ether_pmg0i 割込みハンドラの割込み要因は選択型割込みAです。選択型割込みAではベクタ番号208～255までが選択可能ですが、ether_pmg0i 割込みハンドラではベクタ番号208利用しており、現在の実装ではコンフィグレーション等で利用するベクタ番号を変更することはできません（将来は実装を変更する可能性があります）。

3. T4 のコンフィグレーション

3.1 T4 のコンフィグレーション・ファイル

T4のコンフィグレーション・ファイルは、`mtkernel_3¥device¥ether¥config¥config_tcpudp.h` です。また、設定項目の殆どの内容はT4のマニュアル（組み込み用TCP/IP M3S-T4-Tiny導入ガイドの2.7節）に関連情報が記載されています。以降記載内容と合わせて参照ください。

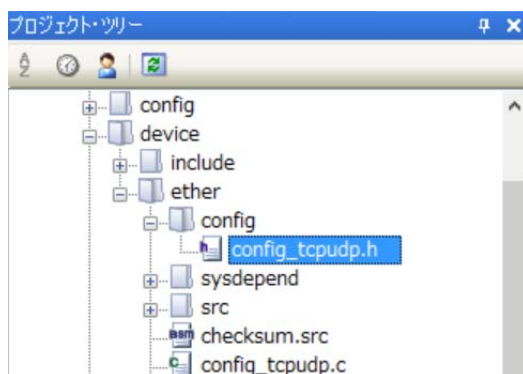


図3.1 T4のコンフィグレーション・ファイル

3.2 タスクの優先度と割り込みハンドラの割り込み優先度

● ETHER_CFG_TASK_PRIORITY

TCP/IPのプロトコル処理を実行するタスク（`ether_tsk`）のタスク優先度です。

システムの都合に応じて、1 ～ `CFN_MAX_TSKPRI`（タスク優先度の最大値）の範囲で変更できます。最低でもTCP/IPを利用するタスクの優先度よりも高く設定することを推奨します。デフォルトは 3 に設定されています。

● ETHER_CFG_INT_PRIORITY

内蔵周辺機能のイーサネットコントローラ（ETHERC）の割り込み優先度です。

システムの都合に応じて、1 ～ `MAX_INT_PRI`（最高外部割り込み優先度）の範囲で変更できます。`TIM_INT_PRI`（システムタイマの割り込み優先度）よりも低く設定することを推奨します。デフォルトでは 8 に設定されています。

```
/* Ether task priority. */  
#define ETHER_CFG_TASK_PRIORITY (3)  
  
/* EINT interrupt priority level. */  
#define ETHER_CFG_INT_PRIORITY (8)
```

3.3 DHCP 関連

● CFG_SYSTEM_DHCP

DHCP機能の有効/無効を設定します（1=有効、0=無効）。

本設定は、_t4_dhcp_enable に反映されます。

● CFG_FIXED_IP_ADDRESS_CHO (CFG_SYSTEM_DHCPが1の場合は設定不要です)

LANソケット毎のIPアドレスを設定します。

バイト間はコンマ(,)で区切ってください。記述例：192, 168, 1, 200

本設定は、tcpudp_env[].ipaddr に反映されます。

● CFG_FIXED_SABNET_MASK_CHO (CFG_SYSTEM_DHCPが1の場合は設定不要です)

LANソケット毎のサブネットマスクを設定します。

バイト間はコンマ(,)で区切ってください。記述例：255, 255, 255, 0

本設定は、tcpudp_env[].maskaddr に反映されます。

● CFG_FIXED_SABNET_MASK_CHO (CFG_SYSTEM_DHCPが1の場合は設定不要です)

LANソケット毎のゲートウェイアドレスを設定します。

バイト間はコンマ(,)で区切ってください。記述例：192, 168, 1, 1 ゲートウェイを使用しない場合は全てゼロを設定してください。

本設定は、tcpudp_env[].gwaddr に反映されます。

● CFG_ETHER_CHO_MAC_ADDRESS

LANソケット毎のMACアドレスを設定します。

バイト間はコンマ(,)で区切ってください。記述例：0x74, 0x90, 0x50, 0x10, 0xFE, 0x77

本設定は、_myethaddr に反映されます。EEPROM等からMACアドレスを読み出して利用する場合は全てゼロに設定し、LANソケットをオープンする前に二次元配列 _myethaddr に設定してください。

以下の4つはLANソケットが2チャンネル存在するターゲット用です。意味はチャンネル0用の設定と同じですが、現状サポートされているターゲットには存在しませんので設定は不要です。

● CFG_FIXED_IP_ADDRESS_CH1

● CFG_FIXED_SABNET_MASK_CH1

● CFG_FIXED_SABNET_MASK_CH1

● CFG_ETHER_CH1_MAC_ADDRESS

```
#define CFG_SYSTEM_DHCP (1)
#define CFG_FIXED_IP_ADDRESS_CHO 192, 168, 1, 200
#define CFG_FIXED_SABNET_MASK_CHO 255, 255, 255, 0
#define CFG_FIXED_GATEWAY_ADDRESS_CHO 192, 168, 1, 1
#define CFG_ETHER_CHO_MAC_ADDRESS 0x00, 0x00, 0x00, 0x00, 0x00, 0x08
#define CFG_FIXED_IP_ADDRESS_CH1 192, 168, 0, 150
#define CFG_FIXED_SABNET_MASK_CH1 255, 255, 255, 0
#define CFG_FIXED_GATEWAY_ADDRESS_CH1 192, 168, 0, 1
#define CFG_ETHER_CH1_MAC_ADDRESS 0x00, 0x00, 0x00, 0x00, 0x00, 0x04
```

3.4 TCP 受付け口のポート番号関連

● CFG_TCP_REPID_NUM

使用するTCPの受付口のポート番号の個数を設定します（1～4まで指定可能）。

● CFG_TCP_REPID*_PORT_NUMBER（*は1～4）

TCPの受付口のポート番号を設定します。上記の CFG_TCP_REPID_NUM の個数分、設定値が有効となります。

本設定は、tcp_crep[].myaddr.portnoに反映されます。

```
#define CFG_TCP_REPID_NUM (1)
#define CFG_TCP_REPID1_PORT_NUMBER (1024)
#define CFG_TCP_REPID2_PORT_NUMBER (1025)
#define CFG_TCP_REPID3_PORT_NUMBER (1026)
#define CFG_TCP_REPID4_PORT_NUMBER (1027)
```

上記の場合、CFG_TCP_REPID_NUM が 1 ですから、CFG_TCP_REPID1_PORT_NUMBER のみが有効な値となります。

3.5 TCP 通信端点関連

● CFG_TCP_CEPID_NUM

使用するTCP通信端点の個数を設定します（1～6まで指定可能）。

以下の3つは CFG_TCP_CEPID_NUM の個数分、設定値が有効となります。

CFG_TCP_CEPID*_CHANNEL（*は1～6）

CFG_TCP_CEPID*_RECEIVE_WINDOW_SIZE（*は1～6）

CFG_TCP_CEPID*_KEEPALIVE_ENABLE（*は1～6）

● CFG_TCP_CEPID*_CHANNEL（*は1～6）

TCP通信端点毎に紐づけするLANソケットのチャネル番号を設定します。

範囲は 0 ～ 1ですが、現状サポートされているターゲットでは全て 0 となります。

本設定は、tcp_ccep[].lan_port_number に反映されます。

● CFG_TCP_CEPID*_RECEIVE_WINDOW_SIZE（*は1～6）

TCP通信端点毎の受信ウィンドウサイズを設定します。

本設定は、tcp_ccep[].rbufsz に反映されます。

● CFG_TCP_CEPID*_KEEPALIVE_ENABLE（*は1～6）

TCP通信端点毎のKEEPALIVE機能（接続が有効であることを確認したり、無通信により切断されるのを防ぐために定期的に短い通信を行う機能）の有効/無効を設定します。

0=無効、1=有効

```

#define CFG_TCP_CEPID_NUM (1)
#define CFG_TCP_CEPID1_CHANNEL (0)
#define CFG_TCP_CEPID1_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID1_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID2_CHANNEL (0)
#define CFG_TCP_CEPID2_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID2_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID3_CHANNEL (0)
#define CFG_TCP_CEPID3_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID3_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID4_CHANNEL (0)
#define CFG_TCP_CEPID4_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID4_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID5_CHANNEL (0)
#define CFG_TCP_CEPID5_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID5_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID6_CHANNEL (0)
#define CFG_TCP_CEPID6_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID6_KEEPALIVE_ENABLE (0)

```

上記の場合、CFG_TCP_CEPID_NUM が 1 ですから、CFG_TCP_CEPID1_CHANNEL、CFG_TCP_CEPID1_RECEIVE_WINDOW_SIZE、CFG_TCP_CEPID1_KEEPALIVE_ENABLE が有効な値となります。

3.6 その他の TCP 関連

● CFG_TCP_MSS

TCP通信のMSS値（最大セグメントサイズ）を設定します。

本設定は、_tcp_mss に反映されます。デフォルトは1460バイトに設定されています。

● CFG_TCP_2MSL_TIME

TCP通信の2MSL時間（TCPセグメントがネットワーク内に存在可能な最大時間）を設定します。

本設定は、_tcp_2msl に反映されます。デフォルトは60秒に設定されています。

● CFG_TCP_MAX_TIMEOUT_PERIOD

TCP通信の再送タイムアウト時間を設定します。

本設定は、_tcp_rt_tmo_rst に反映されます。デフォルトは600秒に設定されています。

● CFG_TCP_DIVIDE_SENDING_PACKET

TCP通信の多重送信機能を設定します。本設定を有効にするとACKの応答待ちを削減するため、送信データは分割して送信されます。

1=有効、0=無効

本設定は、_tcp_dack に反映されます。デフォルトは有効に設定されています。

● CFG_TCP_KEEPALIVE_START

KEEPALIVEの送信開始時間を設定します。設定可能な範囲は 1 ～ 86400 です。

本設定は、_tcp_keepalive_start に反映されます。デフォルトは7200秒に設定されています。

● CFG_TCP_KEEPALIVE_INTERVAL

KEEPALIVEの再送信間隔時間を設定します。設定可能な範囲は 1 ～ 86400 です。

本設定は、_tcp_keepalive_interval に反映されます。デフォルトは10秒に設定されています。

● CFG_TCP_KEEPALIVE_COUNT

KEEPALIVEパケット送信の最大回数を設定します。設定可能な範囲は 0 ～ 0xFFFFFFFF です。

本設定は、_tcp_keepalive_count に反映されます。デフォルトは10回に設定されています。

```
#define CFG_TCP_MSS (1460)
#define CFG_TCP_2MSL_TIME (60)
#define CFG_TCP_MAX_TIMEOUT_PERIOD (600)
#define CFG_TCP_DIVIDE_SENDING_PACKET (1)
#define CFG_TCP_KEEPALIVE_START (7200)
#define CFG_TCP_KEEPALIVE_INTERVAL (10)
#define CFG_TCP_KEEPALIVE_COUNT (10)
```

3.7 UDP 通信端点関連

● CFG_UDP_CEPID_NUM

使用するUDP通信端点の個数を設定します（1～6まで指定可能）。

以下の2つは CFG_UDP_CEPID_NUM の個数分、設定値が有効となります。

CFG_UDP_CEPID*_CHANNEL（*は1～6）

CFG_UDP_CEPID*_PORT_NUMBER（*は1～6）

● CFG_UDP_CEPID*_CHANNEL（*は1～6）

UDP通信端点毎に紐づけするLANソケットのチャネル番号を設定します。

範囲は 0 ～ 1ですが、現状サポートされているターゲットでは全て 0 となります。

本設定は、udp_ccep[].lan_port_number に反映されます。

● CFG_UDP_CEPID*_PORT_NUMBER（*は1～6）

UDP通信端点毎のポート番号を設定します。

本設定は、udp_ccep[].myaddr.portno に反映されます。

```
#define CFG_UDP_CEPID_NUM (1)
#define CFG_UDP_CEPID1_CHANNEL (0)
#define CFG_UDP_CEPID1_PORT_NUMBER (1365)
#define CFG_UDP_CEPID2_CHANNEL (0)
#define CFG_UDP_CEPID2_PORT_NUMBER (1366)
#define CFG_UDP_CEPID3_CHANNEL (0)
#define CFG_UDP_CEPID3_PORT_NUMBER (1367)
#define CFG_UDP_CEPID4_CHANNEL (0)
```

```
#define CFG_UDP_CEPID4_PORT_NUMBER (1368)
#define CFG_UDP_CEPID5_CHANNEL (0)
#define CFG_UDP_CEPID5_PORT_NUMBER (1369)
#define CFG_UDP_CEPID6_CHANNEL (0)
#define CFG_UDP_CEPID6_PORT_NUMBER (1370)
```

3.8 その他のUDP 関連

● CFG_UDP_MULTICAST_TTL

マルチキャスト宛に送信するIPデータグラムのTTLを設定します。

本設定、は__multi_TTL に反映されます。デフォルトは 1 に設定されています。

● CFG_UDP_BEHAVIOR_OF_RECEIVED_ZERO_CHECKSUM

UDPゼロチェックサムを受信したときの動作を設定します。

0=正常パケットとして扱う 1=異常パケットとして破棄する

本設定は、_udp_enable_zerochecksum に反映されます。デフォルトは 0 に設定されています。

● CFG_IP_ARP_CACHE_TABLE_COUNT

ARPのキャッシュエントリ数を設定します。「同時に通信する可能性のあるホスト数+1」以上の値を推奨します。

本設定は、_ip_tblcnt に反映されます。デフォルトは 3 に設定されています。

```
#define CFG_UDP_MULTICAST_TTL (1)
#define CFG_UDP_BEHAVIOR_OF_RECEIVED_ZERO_CHECKSUM (0)
#define CFG_IP_ARP_CACHE_TABLE_COUNT (3)
```


4. API 仕様

4.1 lan_open

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = lan_open(void);
```

<機能>

LANソケットの初期化を行い、他のAPI関数が見える状態にします。また、受信バッファを初期化します。TCP/IPプロトコル処理を行うタスクや周期ハンドラの生成、割込みハンドラの定義も、本APIの内部で行われます。

<パラメータ>

なし

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|------|------|
| E_OK | 正常終了 |
| 負の値 | 異常終了 |

4.2 lan_close

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = lan_close(void);
```

<機能>

LANソケットの動作を停止します。

<パラメータ>

なし

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|------|------|
| E_OK | 正常終了 |
| 負の値 | 異常終了 |

4.3 tcpudp_open

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcpudp_open(UW *work);
```

<機能>

T4ライブラリの初期化を行います。ライブラリの初期化処理では、内部管理領域のメモリ割り当てとその初期化およびライブラリが使用するTCP/IP周期処理関数の起動を行います。

T4が使用するワーク領域の先頭アドレスをパラメータ work に指定します。必要なワーク領域のサイズは、tcpudp_get_ramsize() のリターンパラメータで得られます。

<パラメータ>

| | | |
|-----|------|--------------|
| UW* | work | ワーク領域の先頭アドレス |
|-----|------|--------------|

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|------|-------|
| E_OK | 正常終了 |
| 負の値 | 初期化失敗 |

4.4 tcpudp_close

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcpudp_close(void);
```

<機能>

T4ライブラリの終了処理を行います。ライブラリの終了処理では、ライブラリが使用するTCP/IP周期処理関数を停止します。

本APIを呼び出す前に、TCP通信端点を切断・未使用状態にしてください。

本APIの実行後、ライブラリ・オープン関数 `tcpudp_open()` で指定したワーク領域は開放されますのでアプリケーションプログラムでワーク領域が使用可能となります。

<パラメータ>

なし

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|------|--------|
| E_OK | 正常終了 |
| 負の値 | 終了処理失敗 |

4.5 tcpudp_reset

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcpudp_reset(UB channel);
```

<機能>

T4ライブラリで使用中の指定チャンネルのリセットを行います。このリセット処理では、内部管理領域のメモリの初期化を行います。ユーザはパラメータにリセットチャンネルを指定します。

<パラメータ>

| | | |
|----|---------|---------------|
| UB | channel | LANソケットのチャンネル |
|----|---------|---------------|

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|------|------|
| E_OK | 正常終了 |
|------|------|

4.6 tcpudp_get_ramsize

<API書式>

```
#include <dev_ether.h>
```

```
W size = tcpudp_get_ramsize(void);
```

<機能>

T4が使用するワーク領域のサイズ（RAMサイズ）を戻り値として返します。

ワーク領域は、TCPの受信ウィンドウなどに使用するメモリ領域のことであり、プログラム中で確保し、APIの初期化関数 `tcpudp_open` のパラメータで指定する必要があります。ワーク領域の先頭アドレスは、4byte境界に配置してください。

<パラメータ>

なし

<リターンパラメータ>

W size T4が使用するワーク領域のサイズ（単位：バイト）

例：`tcpudp_get_ramsize`関数の戻り値が100の場合、T4が使用するワーク領域workは以下のように定義することができます。

```
UW work[100/sizeof(UW)];
```

本APIは、以下の目的で使用できます。

1. 静的な配列でワーク領域を確保する場合

本APIでは、T4のコンフィグレーション・ファイルで設定した内容により、T4で使用するワーク領域のサイズを算出しますが、予めユーザ自身でこのサイズを算出するのは困難です。従って、T4のコンフィグレーション・ファイルの内容が決定した時点で、プログラムをビルドしてデバッガで本APIを実行して戻り値を調べます。そして、戻り値が示すサイズ分、ワーク領域のメモリ配列を確保するようにします。

なお、T4のコンフィグレーション・ファイルを変更した場合には、必要なワーク領域のサイズが変わりますので、本APIを用いてワーク領域のサイズを再計算してください。またエラー判定処理として、初期設定の処理の中で必ず本APIを呼び出して、戻り値と（ユーザが最終的に決めた）ワーク領域のサイズを比較し、異なっていた場合はエラー処理に分岐させて、デバッグやテストの段階で間違いが発見できるようにしておいてください。

2. 動的メモリからワーク領域を確保する場合

アプリケーションプログラムの初期設定で本APIを呼び出してワーク領域のサイズを算出します。算出したサイズのワーク領域を動的メモリから確保し、初期化関数`tcpudp_open()`に渡して初期化してください。

なお、本バージョンの μ T-Kernel3.0の実装仕様では、`TK_SUPPORT_MEMLIB` が `FALSE` です。このため、メモリ割当てライブラリが使用できません。動的メモリとしては可変長メモリプールを使用することになります。

4.7 tcp_acp_cep

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcp_acp_cep(ID cepid, ID repid, T_IPV4EP *p_dstaddr, TMO tmout);
```

<機能>

TCP受付口 repid に対する接続要求を待ちます。接続要求があった場合には、指定したTCP通信端点 cepid を用いて接続を確立し、接続を要求してきた相手のIPアドレスとポート番号をリターンパラメータ p_dstaddr が指す領域に格納して返します。

タイムアウト指定 tmout に TMO_FEVR を指定した場合は、接続が確立するまでは待ち状態となりますが、この待ち時間に対してタイムアウト指定することができます。指定時間内に接続が確立されない場合、エラーコード E_TMOUT を返します。

接続が確立された場合、戻り値 E_OK が返されます。確立されない場合は、各原因により上記 E_OK 以外のエラーコードを返します。

<パラメータ>

| | | |
|-----------|-----------|----------------------------------|
| ID | cepid | TCP通信端点ID (1~6までの任意設定値) |
| ID | repid | TCP受付口ID (1~4までの任意設定値) |
| T_IPV4EP* | p_dstaddr | リターンパラメータ dstaddr を返す領域へのポインタ |
| TMO | tmout | タイムアウト指定 |
| | | 正の値：接続が完了するのを待つ時間 (時間の単位は10msec) |
| | | TMO_FEVR：接続が完了するまで待ち状態 (永久待ち) |

<リターンパラメータ>

| | | |
|----------|---------|--------------------------|
| ER | ercd | エラーコード |
| T_IPV4EP | dstaddr | 接続を要求してきた相手のIPアドレスとポート番号 |

<エラーコード>

| | |
|---------|-------------------------------------|
| E_OK | 正常終了 (接続が確立) |
| E_PAR | パラメータエラー (cepid、tmoutが不正な値) |
| E_QOVR | キューイングオーバーフロー (同一端点に対し複数のAPIが発行された) |
| E_OBJ | オブジェクト状態エラー (使用中の通信端点IDを指定) |
| E_TMOUT | タイムアウト (tmoutに設定した時間を経過) |
| E_SYS | システムエラー |

4.8 tcp_con_cep

<API書式>

```
#include <dev_ether.h>
```

```
ER tcp_con_cep(ID cepid, T_IPV4EP *p_myaddr, T_IPV4EP *p_dstaddr, TMO tmout);
```

<機能>

TCP 通信端点 cepid を用いて、接続したい相手の IP アドレス/ポート番号に対して接続を要求し、タイムアウト指定 tmout に TMO_FEVR を指定した場合は、接続が完了するまで待ち状態となります。

接続が確立するまでは待ち状態となりますが、この待ち時間に対してタイムアウト指定することができます。指定時間内に接続が確立されない場合、E_TMOUT を返します。

タイムアウトにより接続の要求がキャンセルされた場合、および相手がサポートしていないポート番号を指定するなど接続が拒否された場合には、通信端点 cepid は未使用状態に戻ります。

この API がコールされるとまず通信端点が使用中でないかをチェックし、以下の設定を行います。

自局の IP アドレスには、変数 tcpudp_env（3章参照）に設定された自局の IP アドレスが設定されます。

自局のポート番号に 0 以外の値を指定した場合、その値が設定されます。また、自局のポート番号に TCP_PORTANY を指定した場合、T4 で 49152～65535 番の範囲の中から割り当てます。

自局の IP アドレス/ポート番号 p_myaddr に NADR を指定した場合には、自局の IP アドレスには変数 tcpudp_env に設定された IP アドレスを、自分側のポート番号には T4 で 49152～65535 番の範囲の中からポート番号を割り当てます。

接続が確立された場合、戻り値 E_OK が返されます。確立されない場合は、各原因により上記 E_OK 以外のエラーコードを返します。

<パラメータ>

| | | |
|-----------|-----------|---------------------------------|
| ID | cepid | TCP通信端点ID（1～6までの任意設定値） |
| T_IPV4EP* | p_myaddr | 自局の IP アドレスとポート番号 |
| T_IPV4EP* | p_dstaddr | 接続したい相手側の IP アドレスとポート番号 |
| TMO | tmout | タイムアウト指定 |
| | | 正の値：接続が完了するのを待つ時間（時間の単位は10msec） |
| | | TMO_FEVR：接続が完了するまで待ち状態（永久待ち） |

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|---------|--------------------------------------|
| E_OK | 正常終了（接続が確立） |
| E_PAR | パラメータエラー（cepid、tmout が不正な値） |
| E_QOVR | キューイングオーバーフロー（同一端点に対し複数の API が発行された） |
| E_OBJ | オブジェクト状態エラー（使用中の通信端点 ID を指定） |
| E_TMOUT | タイムアウト（tmout に設定した時間を経過） |
| E_CLS | 接続の失敗（接続が拒否された） |
| E_SYS | システムエラー |

<備考>

IP アドレスとポート番号を T_IP_V4EP 構造体に格納して本関数に指定してください。

<使用例>

IP アドレス 192. 168. 123. 250、ポート 80 番に接続する場合

```
T_IPV4EP dst;
```

```
dst.ipaddr = 0xc0a87bfa;      // 192 = 0xc0, 168 = 0xa8, 123 = 0x7b, 250 = 0xfa
```

```
dst.portno = 80;
```

```
tcp_con_cep(1, NADR, &dst, TMO_FEVR);
```

4.9 tcp_sht_cep

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcp_sht_cep(ID cepid);
```

<機能>

TCP 通信端点 cepid に対する接続の切断処理の準備として、データ送信の終了手続きを行います。具体的には、送信したデータに対する ACK を受信した時点で FIN を送信します。本 API は切断処理の手配を行うだけのため、待ち状態にはなりません。

本 API 発行後、指定した TCP 通信端点 cepid に対してデータを送信することはできず、送信しようとした場合、エラーコード E_OBJ を返します。データの受信は可能です。正常にデータ転送が行われた場合は、E_OK を返します。データ転送が失敗した場合は各原因により上記 E_OK 以外のエラーコードを返します。

本 API はシャットダウンパケットを送信すると完了します。

<パラメータ>

| | | |
|----|-------|----------------------------|
| ID | cepid | TCP 通信端点 ID (1~6 までの任意設定値) |
|----|-------|----------------------------|

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|--------|---------------------------------------|
| E_OK | 正常終了 (データ送信が終了) |
| E_PAR | パラメータエラー (cepid が不正な値) |
| E_QOVR | キューイングオーバーフロー (同一端点に対し複数の API が発行された) |
| E_OBJ | オブジェクト状態エラー (指定した通信端点が未接続) |
| E_SYS | システムエラー |

4.10 tcp_cls_cep

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcp_cls_cep(ID cepid, TMO tmout);
```

<機能>

TCP 通信端点 cepid の接続を切断します。本 API を発行後は、相手から送信されたデータを破棄します。タイムアウト指定 tmout に TMO_FEVR を指定した場合、タイムアウトにより接続の切断処理がキャンセルされた場合は本 API で指定した TCP 通信端点から RST を送信し、強制的に接続を切断します。この場合、正常切断ではないため、エラーコード E_TMOUT を返します。

本 API は、タイムアウト指定 tmout に TMO_FEVR を指定した場合、正常切断、強制切断のどちらの場合も通信端点が未使用状態になるのを待って API からリターンするため、本 API からリターンした後は TCP 通信端点 cepid をすぐに利用することが可能です。TCP 通信端点は、接続が完全に終了するまで未使用状態となりません。TCP/IP の規格に従うと接続が完全に終了するまでに TIME_WAIT 状態に留まる場合があります。TIME_WAIT 状態の時間 2MSL は、T4 のコンフィグレーション・ファイルで設定できます（4 章参照）。

接続が正常に切断された場合、戻り値 E_OK が返されます。タイムアウトにより強制的に切断された場合は、エラーコード E_TMOUT を返します。これらのコードが返された場合、接続は完全に終了しているため、指定した TCP 通信端点は未使用状態となります。また、通信端点が未接続の場合には本 API は E_OBJ を返します。

tcp_sht_cep() 発行後、ケーブルが切断された等で通信不可の状態になったときに tcp_cls_cep() を TMO_FEVR 指定で発行すると、待ち状態が継続される場合があります。tcp_cls_cep() はタイムアウト指定、またはキャンセル API を発行してください。

<パラメータ>

| | | |
|-----|-------|--|
| ID | cepid | TCP通信端点ID（1～6までの任意設定値） |
| TMO | tmout | タイムアウト指定 正の値：接続がクローズするのを待つ時間（ 時間の単位は10msec ） TMO_FEVR：接続がクローズするまで待ち状態（永久待ち） |

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|---------|--------------------------------------|
| E_OK | 正常終了（接続が正常切断） |
| E_PAR | パラメータエラー（cepid、tmout が不正な値） |
| E_QOVR | キューイングオーバーフロー（同一端点に対し複数の API が発行された） |
| E_OBJ | オブジェクト状態エラー（指定した通信端点が未接続） |
| E_TMOUT | タイムアウト（tmout に設定した時間を経過。接続を強制切断） |
| E_SYS | システムエラー |

4.11 tcp_snd_dat

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcp_snd_dat(ID cepid, void *data, INT len, TMO tmout);
```

<機能>

TCP 通信端点 cepid からデータを送信します。正常に送信した場合、API は送信したデータサイズを返します。

T4 では、RAM 使用の効率化、送信速度の向上のため、ITRON TCP 仕様と異なる点があります。本ライブラリでは、送信データが格納されている領域（以下、ユーザ送信バッファと呼びます）が、ITRON TCP 仕様で定義されている送信ウィンドウを兼ねています。同様に、送信ウィンドウサイズは、送信データの長さとなり、本 API の使用状況によりサイズが異なります。

ITRON TCP 仕様では、ユーザ送信バッファのデータを送信ウィンドウにコピーした時点で、API からリターンしますが、本 API ではタイムアウト指定 tmout に TMO_FEVR を指定した場合、データを送信し、送信データに対する ACK を受信した後、API からリターンします。具体的には、MSS や相手の受信ウィンドウサイズにより TCP レベルの分割が発生した場合、1 つ目の分割データの送信に対する ACKではなく、すべてのデータの送信に対する ACK を受信した時点で API からリターンします。

ITRON TCP 仕様では、送信ウィンドウの空き領域のサイズにより、送信データのサイズが決まります。このため、API の正常終了時の戻り値は必ず第3引数 len と一致するわけではありません。T4 以外の ITRON TCP/IP API 仕様準拠のプロトコルスタック上に移植される場合は、ご注意ください。

tcp_snd_dat()実行中に tcp_can_cep()を発行した後は、ユーザが tcp_cls_cep()を発行して端点を切断してください。

<パラメータ>

| | | |
|-------|-------|---------------------------------|
| ID | cepid | TCP 通信端点 ID (1~6 までの任意設定値) |
| void* | data | 送信データの先頭アドレス |
| INT | len | 送信データの長さ |
| TMO | tmout | タイムアウト指定 |
| | | 正の値：送信が完了するのを待つ時間（時間の単位は10msec） |
| | | TMO_FEVR：送信が完了するまで待ち状態（永久待ち） |

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|---------|---------------------------------------|
| 正の値 | 正常終了（送信したデータのサイズ=第3引数 len の値。単位：byte） |
| E_PAR | パラメータエラー（cepid、tmout が不正な値） |
| E_QOVR | キューイングオーバーフロー（同一端点に対し複数の API が発行された） |
| E_OBJ | オブジェクト状態エラー（未接続、送信終了） |
| E_TMOUT | タイムアウト（tmout に設定した時間を経過） |
| E_CLS | 相手から接続を切断された（RST の送受信により異常切断した場合） |
| E_SYS | システムエラー |

4.12 tcp_rcv_dat

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = tcp_rcv_dat(ID cepid, void *data, INT len, TMO tmout);
```

<機能>

TCP 通信端点 cepid からデータを受信します。

TMO_FEVR を指定した場合は、本 API は、受信ウィンドウからパラメータ data が指すユーザ領域にデータをコピー（以下、データの取り出しと呼びます）し、リターンします。受信ウィンドウが空の場合は、データを受信するまで本 API は待ち状態となります。

受信ウィンドウに入っているデータのサイズが、受信しようとしたデータのサイズ len よりも短い場合、受信ウィンドウが空になるまでデータを取り出し、取り出したデータのサイズを戻り値として返します。

相手から接続が正常に切断され、受信ウィンドウのデータをすべて取り出し、データがなくなると、API から 0 が返ります。

データを受信した場合、受信したデータのサイズが返されます。受信に失敗した場合は、各原因により上記エラーコードを返します。

<パラメータ>

| | | |
|-------|-------|---------------------------------|
| ID | cepid | TCP 通信端点 ID（1～6 までの任意設定値） |
| void* | data | 受信データを格納する領域の先頭アドレス |
| INT | len | 受信データを格納する最大サイズ |
| TMO | tmout | タイムアウト指定 |
| | | 正の値：受信が完了するのを待つ時間（時間の単位は10msec） |
| | | TMO_FEVR：受信が完了するまで待ち状態（永久待ち） |

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|---------|--|
| 正の値 | 正常終了（受信したデータのサイズ。単位：byte） |
| E_OK | データ終了（接続が正常切断。切断までのデータはすべて受信） |
| E_PAR | パラメータエラー（cepid、tmout が不正な値） |
| E_QOVR | キューイングオーバーフロー（同一端点に対し複数の API が発行された） |
| E_OBJ | オブジェクト状態エラー（未接続） |
| E_TMOUT | タイムアウト（tmout に設定した時間を経過） |
| E_CLS | 接続を切断され、受信ウィンドウが空（RST の送受信により異常切断した場合） |
| E_SYS | システムエラー |

4.13 udp_snd_dat

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = udp_snd_dat(ID cepid, T_IPV4EP *p_dstaddr, void *data, INT len, TMO tmout);
```

<機能>

UDP 通信端点 cepid から、相手側の IP アドレスとポート番号を指定して、UDP データグラムを送信します。

UDP データグラムを送信バッファに入れた時点で本 API からリターンします。ここで指す送信バッファとは、T4 内部で確保したものではなく、Ethernet コントローラ内部の送信バッファ（Ethernet の場合）、シリアルドライバで確保した送信バッファ（PPP の場合）のことです。データが送信バッファに格納された場合、送信したデータのサイズ（第 4 引数 len の値）が返されます。送信に失敗した場合は、各原因により上記エラーコードを返します。

本 API では、宛先アドレスにユニキャストアドレス、または、マルチキャストアドレス、ブロードキャストアドレス、ディレクテッドブロードキャストアドレスを指定してデータを送信することができます。

T4 では、1 度に送信できるデータサイズの最大値は、送信バッファのサイズに依存します。ドライバ・インタフェース関数 lan_write, ppp_write により、最大サイズが M (byte) の Ethernet フレーム、PPP フレームを送信（送信バッファに格納）できる場合、1 度に送信可能なデータの最大サイズ N (byte) は、

$$N = M - \text{フレーム・ヘッダサイズ}(F) - \text{IP ヘッダの最小サイズ}(I) - \text{UDP ヘッダサイズ}(U)$$
となります。ここで、M, F, I, U の各値は以下の通りです。

M ≤ 1514 (Ethernet の場合)、M ≤ 1504 (PPP の場合)

F = 14 (Ethernet の場合)、F = 4 (PPP の場合)

I = 20

U = 8

T4 では、IP フラグメントに対応していないため、これより大きなサイズのデータを送信する場合には、データを分割し、N byte 以下にする必要があります。N byte 以上のデータサイズを指定した場合の動作、戻り値は不定です。

タイムアウト指定 tmout に TMO_FEVR を指定した場合は、送信バッファにデータが入るまで待ち状態になります。タイムアウト指定 tmout に正の値を指定した場合は、指定時間まで待ち状態になります。指定時間までに送信バッファにデータが入らなかった場合には、エラーコード E_TMOUT を返します。データが送信バッファに格納された場合、格納されたデータのサイズを返します。

<パラメータ>

| | | |
|-----------|-----------|---|
| ID | cepid | UDP 通信端点 ID (1~6 までの任意設定値) |
| T_IPV4EP* | p_dstaddr | 送信したい相手側の IP アドレスとポート番号 |
| void* | data | 送信データの先頭アドレス |
| INT | len | 送信データの長さ (0 以上 1472 以下) |
| TMO | tmout | タイムアウト指定 正の値：送信が完了するのを待つ時間（時間の単位は10msec） TMO_FEVR：送信が完了するまで待ち状態（永久待ち） |

<リターンパラメータ>

| | | |
|----|------|--------|
| ER | ercd | エラーコード |
|----|------|--------|

<エラーコード>

| | |
|-----------|---|
| E_OK, 正の値 | 正常終了（送信したデータのサイズ=第4引数 len の値。単位 : byte） |
| E_PAR | パラメータエラー（cepid、tmout が不正な値） |
| E_QOVR | キューイングオーバーフロー（同一端点に対し複数の API が発行された） |
| E_TMOUT | タイムアウト（tmout に設定した時間を経過） |
| E_CLS | 接続の失敗（ARP 交換ができなかった） |
| E_SYS | システムエラー |

4.14 udp_rcv_dat

<API書式>

```
#include <dev_ether.h>
```

```
ER ercd = udp_rcv_dat(ID cepid, T_IPV4EP *p_dstaddr, void *data, INT len, TMO tmout);
```

<機能>

UDP 通信端点 cepid から UDP データグラムを受信し、相手側の IP アドレスとポート番号を取得します。データを受信した場合、受信したデータのサイズが返されます。受信に失敗した場合は、各原因により上記エラーコードを返します。

T4 では、1 度に受信できるデータサイズの最大値は、ドライバの受信バッファのサイズに依存します。ドライバ・インタフェース関数 lan_read, ppp_read により、最大サイズが M (byte) の Ethernet フレーム、PPP フレームを受信（受信バッファに格納）できる場合、1 度に受信可能なデータの最大サイズ N (byte) は、

$$N = M - \text{フレーム・ヘッダサイズ}(F) - \text{IP ヘッダの最小サイズ}(I) - \text{UDP ヘッダサイズ}(U)$$
となります。ここで、M, F, I, U の各値は以下の通りです。

M ≤ 1514 (Ethernet の場合)、M ≤ 1504 (PPP の場合)

F = 14 (Ethernet の場合)、F = 4 (PPP の場合)

I = 20

U = 8

T4 では、IP フラグメントに対応していないため、これより大きなサイズのデータを受信した場合には破棄されます。

タイムアウト指定 tmout に TMO_FEVR を指定した場合は、UDP データグラムを受信するまで待ち状態になります。タイムアウト指定 tmout に正の値を指定した場合は、指定時間まで待ち状態になります。指定時間までに UDP データグラムを受信できなかった場合、エラーコード E_TMOUT を返します。

受信 API が発行された状態で UDP データを受信すると、受信した UDP データは API で指定したデータ領域にコピーされます。

受信したデータのサイズが API で指定したサイズと等しい、または指定したサイズより小さい場合、受信したデータのすべてはコピーされます。受信したデータのサイズが指定したサイズより大きい場合、受信したデータは指定されたデータサイズと等しい byte 数コピーされ、残りは破棄されます。

前者の場合、戻り値は受信データ長です。後者の場合、戻り値は E_BOVR です。

本 API では、宛先 IP アドレスがユニキャストアドレス、または、マルチキャストアドレス (224.0.0.0~239.255.255.255)、または、ブロードキャストアドレス (255.255.255.255) の UDP データグラムを受信することができます。また、ディレクテッドブロードキャストアドレス（例：192.168.0.0/24 の場合、192.168.0.255）も受信可能です。

<パラメータ>

| | | |
|-----------|-----------|---|
| ID | cepid | UDP 通信端点 ID (1~6 までの任意設定値) |
| T_IPV4EP* | p_dstaddr | リターンパラメータ dstaddr を返す領域へのポインタ |
| void* | data | 受信データを格納する領域の先頭アドレス |
| INT | len | 受信データを格納する最大サイズ |
| TMO | tmout | タイムアウト指定 正の値：受信が完了するのを待つ時間（時間の単位は10msec） |

TMO_FEVR : 受信が完了するまで待ち状態（永久待ち）

<リターンパラメータ>

| | | |
|----------|---------|-----------------------------|
| ER | ercd | エラーコード |
| T_IPV4EP | dstaddr | データを送信してきた相手の IP アドレスとポート番号 |

<エラーコード>

| | |
|-----------|--------------------------------------|
| E_OK, 正の値 | 正常終了（受信したデータのサイズ。単位 : byte） |
| E_PAR | パラメータエラー（cepid、tmout が不正な値） |
| E_QOVR | キューイングオーバーフロー（同一端点に対し複数の API が発行された） |
| E_TMOUT | タイムアウト（tmout に設定した時間を経過） |
| E_BOVR | バッファオーバーフロー（受信データを格納する領域以上のデータが届いた） |
| E_SYS | システムエラー |

5. スタックサイズ

5.1 スタック見積もりツール

本実装においてもスタック見積もりツールを使用することが可能です。スタック見積もりツールの起動方法やリアルタイムOSオプションの設定等はターゲット毎の構築仕様書の第5章を参照ください。

以降はスタック見積もりツールを使用することを前提に各スタックサイズを紹介します。以下にスタック見積もりツールの表示例を示します。

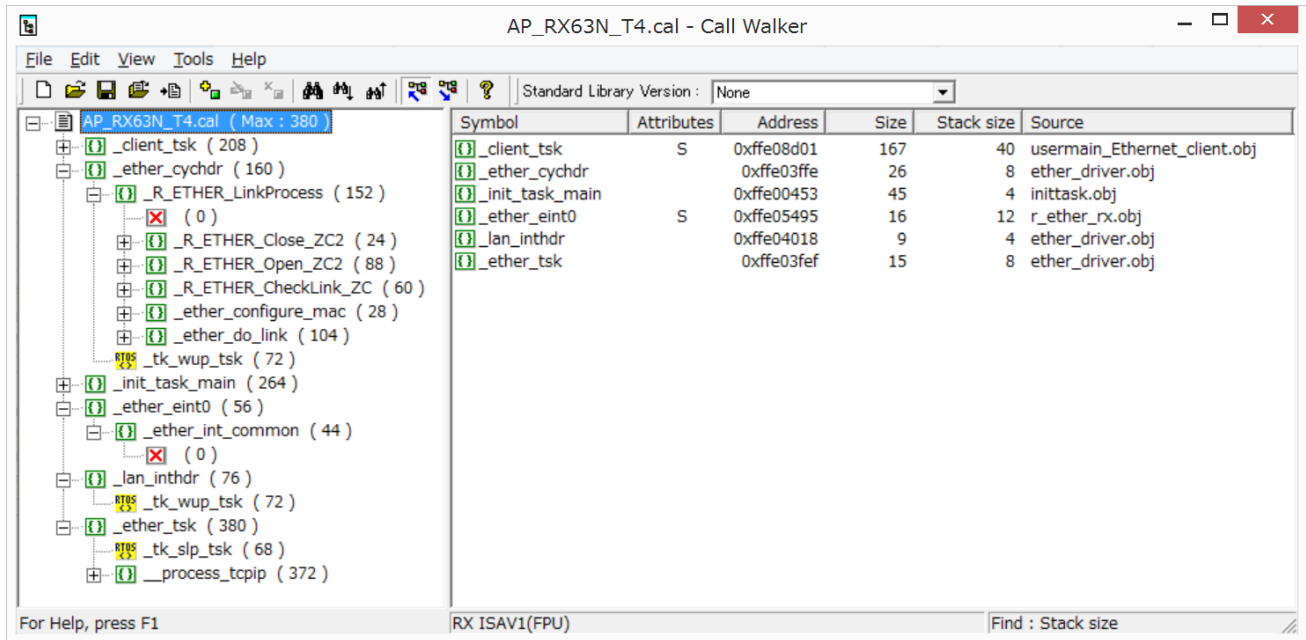


図5.1 T4のスタック見積もり結果

5.2 ether_tsk のスタックサイズ

ether_tskタスクが独自に使用するスタックサイズは380バイトです。コンパイラのバージョン等が変化しても、それほど大きな違いはないはずです。このサイズにタスクコンテキストのサイズを加えても、**現状のサンプルで確保されている512バイトは超えない**と考えて構いません。

ただし、カーネル管理外割込みを複数レベル使用すると512バイトを超える可能性があります。もし、カーネル管理外割込みを複数レベル使用する場合はターゲット毎の構築仕様書の第5章の内容に従ってスタックサイズを算出し、mtkernel_3¥device¥ether¥sysdepend¥ターゲット¥ether_subsystem.cで生成・確保されているスタックサイズを変更してください。

```
u.t_ctsk.stksz = 512;           // Set Task StackSize
u.t_ctsk.itskpri = ETHER_CFG_TASK_PRIORITY; // Set Task Priority
```

5.3 ether_cychdr のスタックサイズ

ether_cychdr周期ハンドラのスタックサイズはスタック見積もりツールの表示結果を使ってください。図5.1の例ならば160バイトとなります。内部で関数ポインタによりcallback_ether関数を呼び出し

ており、未解決シンボルの表示がありますが、表示されているスタック最深部を超えませんから無視して構いません。

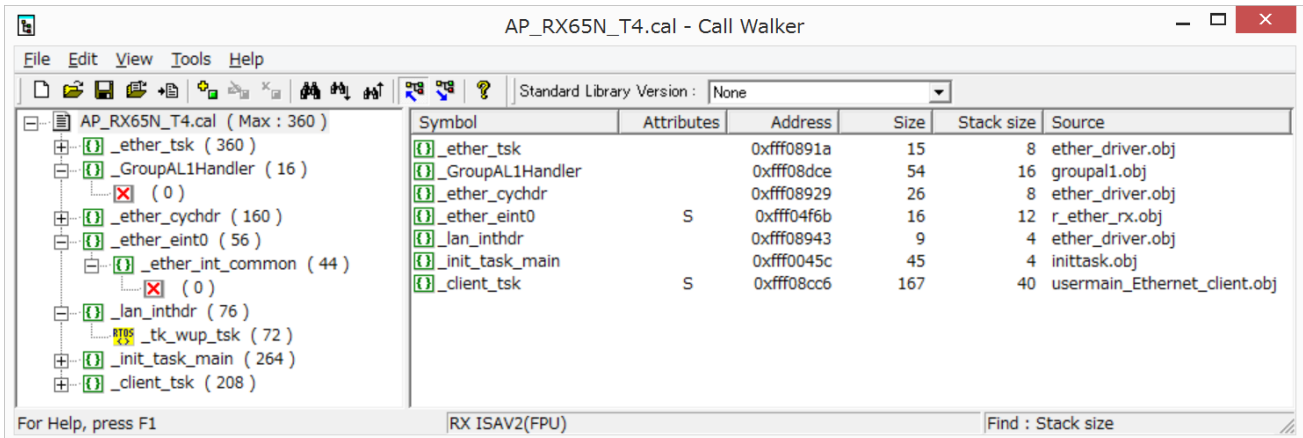
周期ハンドラのスタックサイズはシステムタイマ割込みの延長上で実行されます。システムタイマ割込みのスタックサイズの算出には、ether_cychdrが使用するスタックサイズを加味してください。システムタイマ割込みのスタックサイズの算出方法はターゲット毎の構築仕様書の第5章を参照してください。

5.4 ether_eint0 のスタックサイズ

ether_eint0割込みハンドラは、内部で関数ポインタによりlan_inthdr関数を呼び出します。結果、ether_eint0割込みハンドラは表示されているスタックサイズにlan_inthdr関数を使用するサイズを加算したサイズとなります。図5.1の例ならば、 $56 + 76 = 132$ バイトとなります。

このサイズがT4のコンフィグレーションで指定した ETHER_CFG_INT_PRIORITY 割込み優先度で実行される割込みハンドラのスタックサイズとなります。5.3節のether_cychdrのスタックサイズと共に、これらの数値を使って例外スタックのサイズを算出してください。詳しくはターゲット毎の構築仕様書の第5章を参照してください。

また、RX65NやRX72N等のようにether_eint0割込みハンドラがGroupAL1Handlerグループ割込みハンドラ経由で実行される場合、スタック見積もりツールの表示結果は図5.2のようになります。



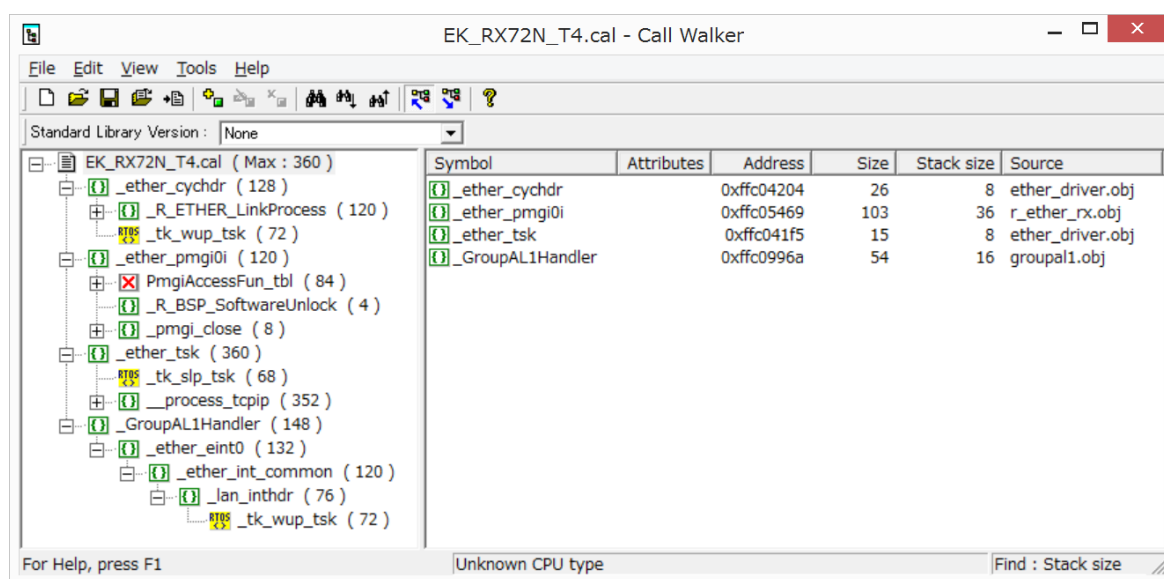
| Symbol | Attributes | Address | Size | Stack size | Source |
|------------------|------------|-------------|------|------------|------------------------------|
| _ether_tsk | | 0xffff0891a | 15 | 8 | ether_driver.obj |
| _GroupAL1Handler | | 0xffff08dce | 54 | 16 | groupal1.obj |
| _ether_cychdr | | 0xffff08929 | 26 | 8 | ether_driver.obj |
| _ether_eint0 | S | 0xffff04f6b | 16 | 12 | r_ether_rx.obj |
| _lan_inthdr | | 0xffff08943 | 9 | 4 | ether_driver.obj |
| _init_task_main | | 0xffff0045c | 45 | 4 | inittask.obj |
| _client_tsk | S | 0xffff08cc6 | 167 | 40 | usermain_Ethernet_client.obj |

図5.2 GroupAL1Handlerグループ割込みハンドラを使用する場合のスタック見積もり結果

上記の場合、ether_eint0割込みハンドラはGroupAL1Handlerグループ割込みハンドラから関数ポインタ経由で実行され、呼び出されたether_eint0割込みハンドラは更に関数ポインタ経由でlan_inthdr関数を呼び出します。結果、GroupAL1Handlerグループ割込みハンドラを利用する場合、GroupAL1Handlerグループ割込みハンドラは表示されているスタックサイズにlan_inthdr関数とether_eint0割込みハンドラが使用するサイズを加算したサイズとなります。図5.2の例ならば、 $16 + 56 + 76 = 148$ バイトとなります。

5.5 PMGI 対応版のスタックサイズ

EK-RX72N ターゲットのように PMGI（PHY マネジメントインタフェース）を内蔵し、PHY-LSI と MII で接続されている場合、ether_pmgioi 割込みハンドラを利用した構造となります。**ether_pmgioi 割込みハンドラは、ether_eint0 割込みハンドラと同じ ETHER_CFG_INT_PRIORITY 割込み優先度で実行されます。**従って、スタックサイズの計算においては、ether_pmgioi 割込みハンドラと GroupAL1Handler グループ割込みハンドラ経由で実行される ether_eint0 割込みハンドラの 2 つの内、より多くスタックサイズを使用する方を選択することになります。以下に PMGI 対応版のスタック見積もりツールの表示例を示します。なお、解析結果における未参照シンボルは全て関数のドラッグ&ドロップで正しい解析結果に修正してあります。



| Symbol | Attributes | Address | Size | Stack size | Source |
|------------------|------------|------------|------|------------|------------------|
| _ether_cychdr | | 0xffc04204 | 26 | 8 | ether_driver.obj |
| _ether_pmgioi | | 0xffc05469 | 103 | 36 | r_ether_rx.obj |
| _ether_tsk | | 0xffc041f5 | 15 | 8 | ether_driver.obj |
| _GroupAL1Handler | | 0xffc0996a | 54 | 16 | groupal1.obj |

図5.3 PMGI対応版のスタック見積もり結果

解析結果を見ると、ether_pmgioi 割込みハンドラ（GroupAL1Handler グループ割込みハンドラ経由）と ether_eint0 割込みハンドラでは、ether_eint0 割込みハンドラの方が、より多くのスタックサイズを必要とします。以上のことから、**PMGI 対応版であっても、T4 利用時のスタックサイズの計算方法は変わらないこと**になります。

- ・ ether_tsk → 表示結果がタスクが独自に使用するスタックサイズとなります。
上記の例ならば、360 バイトとなります。
- ・ ether_cychdr → 表示結果が周期ハンドラが独自に使用するスタックサイズとなります。
上記の例ならば、128 バイトとなります。
- ・ ether_eint0 → GroupAL1Handler 割込みハンドラ + 表示結果 + lan_inthdr 関数 の値が
割込みハンドラが独自に使用するスタックサイズとなります。
上記の例ならば、148 バイトとなります。
- ・ ether_pmgioi → スタックサイズの計算には利用しません。

6. サンプルプログラム

6.1 サンプルプログラムの概要

本実装ではサーバー側（接続要求の受信側）のサンプルプログラムとクライアント側（接続要求の送信側）のサンプルプログラムを準備しています。

【サーバー側】

```
mtkernel_3¥kernel¥usermain_ethernet¥ usermain_ethernet_server.c
```

【クライアント側】

```
mtkernel_3¥kernel¥usermain_ethernet¥ usermain_ethernet_client.c
```

プロジェクト立ち上げ時、クライアント側のソースプログラムがアクティブになっており、サーバー側のサンプルプログラムはビルドから除外されています。必要に応じてビルドの対象を変更してお使いください。

なお、初期状態のT4コンフィグレーション・ファイルの内容はクライアント側の設定になっています。サーバー側のサンプルプログラムを使用する場合、7.3節に記載の内容に従ってT4コンフィグレーション・ファイルの内容を修正ください。

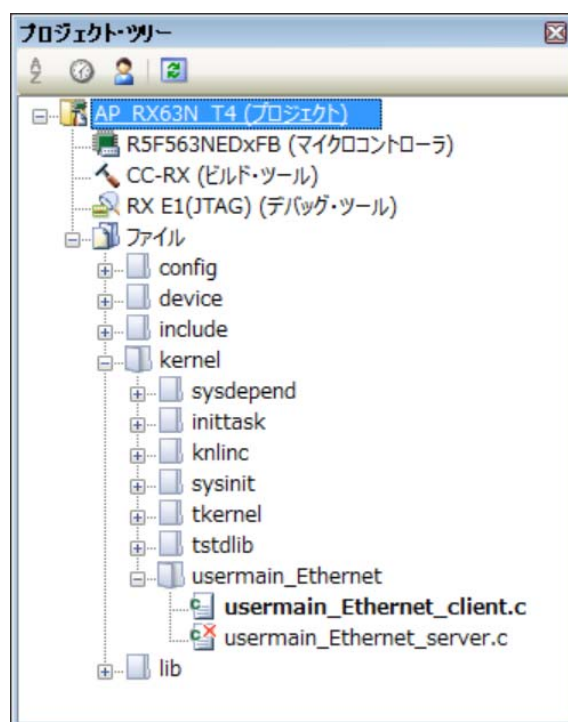


図6.1 サンプルプログラムの初期状態

なお、サンプルプログラムでは「CLANGSPEC」のマクロ定義を利用しています。「CLANGSPEC」のマクロ定義に関しては各ターゲットの構築仕様書を参照してください。

6.2 クライアント側のサンプルプログラム

6.2.1 usermain 関数

```
#include <string.h>
#include <tk/tkernel.h>
#include "dev_ether.h"

#define T4_WORK_SIZE (2796)
LOCAL UW tcpudp_work[(T4_WORK_SIZE / sizeof(UW)) + 1];

void usermain(void)
{
    ID objid;
    T_CTSK t_ctsk;
    if( lan_open() != E_OK )                // LANソケットの動作開始
        goto Err;
    if( tcpudp_get_ramsize() > sizeof( tcpudp_work ) ) // ワークエリアのサイズを確認
        goto Err;
    if( tcpudp_open( tcpudp_work ) != E_OK ) // TCP/UDP通信を開始
        goto Err;
    t_ctsk.exinf = (void*)tk_get_tid();      // 自タスクのID番号を拡張情報に設定
    t_ctsk.tskatr = TA_HLNG | TA_DSNAME;    // タスクの属性を設定
    t_ctsk.stksz = 512;                     // タスクのスタックサイズを設定
    t_ctsk.itskpri = 10;                    // タスクの優先度を設定 (任意)
    t_ctsk.task = client_tsk;               // タスクの起動番地を設定
    strcpy( t_ctsk.dsname, "client" );      // タスクのデバッガサポート名を設定
    if( (objid = tk_cre_tsk( &t_ctsk )) <= E_OK ) // クライアントタスクの生成
        goto Err;
    if( tk_sta_tsk( objid, 1 ) < E_OK )      // クライアントタスクの起動
        goto Err;                          // 通信端点IDはタスクの起動コードに設定
    // 通信端点ID(1)の詳細は config_tcpudp.h の CFG_TCP_CEPID1_* で定義
    tk_slp_tsk( TMO_FEVR );                 // 通信の終了を待つ
    tcpudp_close();                         // TCP/UDP通信を終了
    lan_close();                            // LANソケットの動作停止
Err:
    while( 1 ) ;
}
```

usermain関数では、LANソケットのオープン・クローズ、TCP/UDP通信のオープン・クローズ、クライアント側のタスクの生成と起動を行っています。生成したクライアントが側のタスクには拡張情報で自身のタスクのID番号、起動コードで使用する通信端点IDを渡し、自身はtk_slp_tskでクライアント側のタスクの処理の終了を待ちます。

6.2.2 client_tsk

```
#include <tk/tkernel.h>
#include <tm/tmonitor.h>
#include "dev_ether.h"

LOCAL char sbuf[] = "ABCDEFGHJKLMNOPQRSTUVWXYZ"; // 送信バッファ
LOCAL char rbuf[1460];                          // 受信バッファ

LOCAL void client_tsk(INT stacd, void *exinf)
{
    LOCAL T_IPV4EP dstaddr, myaddr;
    ID cepid = stacd; ER ercd;
    INT i, len;
```

```

dstaddr.ipaddr = (192U<<24) + (168<<16) + (1<<8) + 200; // サーバーのIPアドレスを設定
dstaddr.portno = 1024; // サーバーのポート番号を設定
if( tcp_con_cep( cepid, &myaddr, &dstaddr, 1000 ) != E_OK ) // 接続要求の発行
    goto Err;
for( i=0 ; i<10 ; i++ ) {
    tcp_snd_dat( cepid, sbuf, sizeof(sbuf)-1-i, TMO_FEVR ); // サーバーにデータ送信
    // パケットが分割されない時の受信手順。分割されないなら一度の受信要求で完了
    len = tcp_rcv_dat( cepid, rbuf, 1460, TMO_FEVR );
    // パケットが分割される可能性がある時の受信手順。以下の例は300msのタイムアウトで判断
    // 最初のパケットは永久待ちで受信要求を行い、2回目以降は300msのタイムアウトを設定
    len = tcp_rcv_dat( cepid, rbuf, 1460, TMO_FEVR );
    while( (ercd=tcp_rcv_dat( cepid, &rbuf[len], 1460, 30 )) > E_OK )
        len += ercd;
    rbuf[len] = '\0'; // 受信データにヌルコードを設定
    tm_printf( "%s\n", rbuf ); // ターミナルに表示
    tk_dly_tsk( 5000 ); // 5秒待ち
}
tcp_sht_cep( cepid ); // 切断準備
tcp_cls_cep( cepid, TMO_FEVR ); // 通信切断
Err:
    tk_wup_tsk( (ID)exinf ); // 初期化タスクを起床
    tk_ext_tsk( ); // タスクの終了
}

```

起動されたら、IPアドレス192.168.1.200、ポート番号1024のサーバーに接続要求を発行します。

接続後は、10回、データの送信と受信を行います。送信は1回毎に送信データの長さを-1します。サーバー側からは送信したデータと同じデータがエコーバックされる予定です。

T4の場合、CFG_TCP_DIVIDE_SENDING_PACKETの設定により、送信データは分割される可能性があります。分割されないのであれば、現在コメントになっている1度のtcp_rcv_datで受信可能ですが、分割される可能性がある場合、上位プロトコルの仕様で分割されたデータを連結する必要があります。上記の例ではプロトコル仕様を決めていないため、300msの期間内に到着したパケットはデータを連結するものと仮定して処理を構成しています。受信後はターミナルに受信データを表示し、tk_dly_tskで5秒の待ち時間を確保後、再度送信と受信を繰り返します。

10回、データの送信と受信が完了したら、通信を切断し、拡張情報を使って初期化タスクを起床し、自タスクを終了します。

6.2.3 コンフィグレーション・ファイル（主要項目のみ）

```

#define CFG_SYSTEM_DHCP (1)
#define CFG_TCP_CEPID_NUM (1)
#define CFG_TCP_CEPID1_CHANNEL (0)
#define CFG_TCP_CEPID1_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID1_KEEPALIVE_ENABLE (0)
#define CFG_TCP_DIVIDE_SENDING_PACKET (1)

```

DHCPは有効です。サンプルプログラムを実行するネットワーク環境にDHCPサーバーが必要となります。TCPの通信端点は1つしか使用しません。ID=1の通信端点は、LANソケット（ETHERC）のチャンネルは0、受信ウィンドウサイズは1460バイト、KEEPALIVE機能は無効です。

また、CFG_TCP_DIVIDE_SENDING_PACKETが有効です。このため、送信データは分割されて送信されることになります。本設定値は無効にしても、サンプルプログラムは正常に動作します。

6.3 サーバー側のサンプルプログラム

6.3.1 usermain 関数

```
#include <string.h>
#include <tk/tkernel.h>
#include "dev_ether.h"

#define T4_WORK_SIZE (5112)
LOCAL UW tcpudp_work[T4_WORK_SIZE/sizeof(UW) + 1];

void usermain(void)
{
    int i;
    ID objid;
    T_CTSK t_ctsk;
    if( lan_open() != E_OK )                // LANソケットの動作開始
        goto Err;
    if( tcpudp_get_ramsize() > sizeof( tcpudp_work ) ) // ワークエリアのサイズを確認
        goto Err;
    if( tcpudp_open( tcpudp_work ) != E_OK ) // TCP/UDP通信を開始
        goto Err;
    t_ctsk.exinf = (void*)1;                // 受け口ポートIDを拡張情報に設定
    // 受け口ポートID(1)の詳細は config_tcpudp.h の CFG_TCP_REPID1_PORT_NUMBER で定義
    t_ctsk.tskatr = TA_HLNG | TA_DSNAME;    // タスクの属性を設定
    t_ctsk.stksiz = 512;                    // タスクのスタックサイズを設定
    t_ctsk.itkpri = 10;                     // タスクの優先度を設定 (任意)
    t_ctsk.task = server_tsk;               // タスクの起動番地を設定
    strcpy( t_ctsk.dsname, "server" );      // タスクのデバッグサポート名を設定
    for( i=0 ; i<3 ; i++ ) {               // 用意した通信端点数分のループ
        // 用意した通信端点の個数は config_tcpudp.h の CFG_TCP_CEPID_NUM で定義
        if( (objid = tk_cre_tsk( &t_ctsk )) <= E_OK ) // サーバータスクの生成
            goto Err;
        if( tk_sta_tsk( objid, i ) < E_OK ) // サーバータスクの起動
            goto Err;                       // 通信端点IDはタスクの起動コードに設定
    }
    tk_slp_tsk( TMO_FEVR );                 // サーバータスクに制御を渡す
    tcpudp_close();                         // TCP/UDP通信を終了
    lan_close();                            // LANソケットの動作停止
Err:
    while( 1 ) ;
}
```

usermain関数では、LANソケットのオープン、TCP/UDP通信のオープン、サーバー側のタスクの生成と起動を行っています。サーバー側のタスクは3つ生成しています。つまり、同時に3つのクライアント側からの接続要求が受け付け可能です。受け付けポート番号は1024の同じ値を利用するため、拡張情報に受け付け口ID番号を設定します。TCP通信端点IDはタスクの起動コードに設定します。

本サンプルでは、初期化タスクはtk_slp_tskで起床待ちとなってサーバータスクに制御を渡した後に起床待ちが解除されることはありませんが、もし起床待ちが解除された時はTCP/UDP通信のクローズ、LANソケットのクローズを行います。

6.3.2 server_tsk

```
#include <tk/tkernel.h>
#include <tm/tmonitor.h>
#include "dev_ether.h"

LOCAL char buf[3][1460];          // 受信バッファ（通信端点数分確保）

LOCAL void server_tsk(INT stacd, void *exinf)
{
    T_IPV4EP dstaddr;
    ID cepid = stacd+1, repid = (ID)exinf;
    INT len; ER ercd;
    while( 1 ) {
        tcp_acp_cep( cepid, repid, &dstaddr, TMO_FEVR );
        // 接続要求待ち、通信端点IDはタスクの起動コード、受け口ポートIDはタスクの拡張情報から入手
        while( 1 ) {
            // パケットが分割されない時の受信手順。分割されないなら一度の受信要求で完了
            if( (len = tcp_rcv_dat( cepid, buf[stacd], 1460, TMO_FEVR )) <= E_OK )
                break;
            // パケットが分割される可能性がある時の受信手順。以下の例は300msのタイムアウトで判断
            // 最初のパケットは永久待ちで受信要求を行い、2回目以降は300msのタイムアウトを設定
            len = tcp_rcv_dat( cepid, buf[stacd], 1460, TMO_FEVR );
            while( (ercd=tcp_rcv_dat( cepid, &buf[stacd][len], 1460, 30 )) > E_OK )
                len += ercd;
            if( ercd != E_TMOUT )                // エラーコードがタイムアウトでなければ通信終了
                break;                          // 無限ループを抜け出す
            buf[stacd][len] = '\0';              // 受信バッファにヌルコードを設定
            tm_printf("cepid = %d : %s\n", cepid, buf[stacd] ); // ターミナルに表示
            tcp_snd_dat( cepid, buf[stacd], len, TMO_FEVR );    // クライアントにエコーバック
        }
        tcp_sht_cep( cepid );                  // 切断準備
        tcp_cls_cep( cepid, TMO_FEVR );        // 通信を切断
    }
}
```

起動されたら、起動コードと拡張情報で渡された通信端点ID、受け口ポートIDを使って、クライアントからの接続要求を待ちます。クライアントからの接続要求を受付けたら、通信が切断されるまで以下の処理を繰り返します。

クライアントからのデータ受信を行い、受信後は受信したデータをクライアントにエコーバックします。T4の場合、CFG_TCP_DIVIDE_SENDING_PACKETの設定により、送信データは分割される可能性があります。分割されないのであれば、現在コメントになっている1度のtcp_rcv_datで受信可能ですが、分割される可能性がある場合、上位プロトコルの仕様で分割されたデータを連結する必要があります。上記の例ではプロトコル仕様を決めていないため、300msの期間内に到着したパケットはデータを連結するものと仮定して処理を構成しています。受信後はターミナルに受信データを表示し、クライアントに受信データをエコーバックします。

通信が切断されたら、こちらも切断処理を行い、再度クライアントからの接続要求を待ちます。

6.3.3 コンフィグレーション・ファイル（主要項目のみ）

```
#define CFG_SYSTEM_DHCP                (0)
#define CFG_FIXED_IP_ADDRESS_CHO      192, 168, 1, 200
```

```

#define CFG_FIXED_SUBNET_MASK_CHO 255, 255, 255, 0
#define CFG_FIXED_GATEWAY_ADDRESS_CHO 192, 168, 1, 1

#define CFG_TCP_REPID_NUM (1)
#define CFG_TCP_REPID1_PORT_NUMBER (1024)

#define CFG_TCP_CEPID_NUM (3)
#define CFG_TCP_CEPID1_CHANNEL (0)
#define CFG_TCP_CEPID1_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID1_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID2_CHANNEL (0)
#define CFG_TCP_CEPID2_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID2_KEEPALIVE_ENABLE (0)
#define CFG_TCP_CEPID3_CHANNEL (0)
#define CFG_TCP_CEPID3_RECEIVE_WINDOW_SIZE (1460)
#define CFG_TCP_CEPID3_KEEPALIVE_ENABLE (0)

#define CFG_TCP_DIVIDE_SENDING_PACKET (1)

```

DHCPは無効です。このためLANソケットチャネル0のIPアドレス、サブネットマスク、ゲートウェイアドレスの値が有効です。受け口ポート番号は1024の1つしか使いません。TCPの通信端点は3つ使用します（生成するサーバタスクの個数分）。結果、通信端点ID=1から3が有効となります。どの通信端点もLANソケット（ETHERC）のチャネルは0、受信ウィンドウサイズは1460バイト、KEEPALIVE機能は無効です。

また、CFG_TCP_DIVIDE_SENDING_PACKETが有効です。このため、送信データは分割されて送信されることになります。本設定値は無効にしても、サンプルプログラムは正常に動作します。

6.4 実行結果

同一ネットワーク上にサーバ側のターゲットを1台、クライアント側のターゲットを3台、同時に実行した時のサーバ側のターミナル表示を示します。

```

microT-Kernel Version 3.00

cepid = 1 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
cepid = 1 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
cepid = 1 : ABCDEFGHIJKLMNOPQRSTUVWX
cepid = 1 : ABCDEFGHIJKLMNOPQRSTUVW
cepid = 2 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
cepid = 1 : ABCDEFGHIJKLMNOPQRSTUV
cepid = 2 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
cepid = 1 : ABCDEFGHIJKLMNOPQRSTU
cepid = 3 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
cepid = 2 : ABCDEFGHIJKLMNOPQRSTUVWX
cepid = 1 : ABCDEFGHIJKLMNOPQRST
cepid = 3 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
cepid = 2 : ABCDEFGHIJKLMNOPQRSTUVW
cepid = 1 : ABCDEFGHIJKLMNOPQRS
cepid = 3 : ABCDEFGHIJKLMNOPQRSTUVWX
cepid = 2 : ABCDEFGHIJKLMNOPQRSTUV
cepid = 1 : ABCDEFGHIJKLMNOPQR
cepid = 3 : ABCDEFGHIJKLMNOPQRSTUVW
cepid = 2 : ABCDEFGHIJKLMNOPQRSTU
cepid = 1 : ABCDEFGHIJKLMNOPQ
cepid = 3 : ABCDEFGHIJKLMNOPQRSTUV

```

```

cepid = 2 : ABCDEFGHIJKLMNOPQRST
cepid = 3 : ABCDEFGHIJKLMNOPQRSTU
cepid = 2 : ABCDEFGHIJKLMNOPQRS
cepid = 3 : ABCDEFGHIJKLMNOPQRST
cepid = 2 : ABCDEFGHIJKLMNOPQR
cepid = 3 : ABCDEFGHIJKLMNOPQRS
cepid = 2 : ABCDEFGHIJKLMNOPQ
cepid = 3 : ABCDEFGHIJKLMNOPQR
cepid = 3 : ABCDEFGHIJKLMNOPQ

```

6.5 NTP クライアントのサンプルプログラム

RFC2030 に準拠したシンプルネットワークタイムプロトコル (SNTP) のサンプルプログラムです。実行後、1 日間隔で NTP サーバーに現在時刻を問い合わせ、 μ T-Kernel3.0 のシステム時刻を更新します。

6.5.1 usermain 関数

```

#include <string.h>
#include <tk/tkernel.h>
#include <tm/tmonitor.h>
#include "dev_ether.h"

#define T4_WORK_SIZE (2796)
LOCAL UW tcpudp_work[(T4_WORK_SIZE / sizeof(UW))];

void usermain(void)
{
    ID objid;
    T_CTSK t_ctsk;
    if( lan_open() != E_OK )                // LANソケットの動作開始
        goto Err;
    if( tcpudp_get_ramsize() > sizeof( tcpudp_work ) ) // ワークエリアのサイズを確認
        goto Err;
    if( tcpudp_open( tcpudp_work ) != E_OK ) // TCP/UDP通信を開始
        goto Err;
    t_ctsk.exinf = (void*)tk_get_tid();      // 自タスクのID番号を拡張情報に設定
    t_ctsk.tskatr = TA_HLNG | TA_DSNAME;    // タスクの属性を設定
    t_ctsk.stksz = 356;                     // タスクのスタックサイズを設定
    t_ctsk.itskpri = 10;                    // タスクの優先度を設定 (任意)
    t_ctsk.task = ntp_client_tsk;           // タスクの起動番地を設定
    strcpy( t_ctsk.dsname, "ntp_cli" );     // タスクのデバッガサポート名を設定
    if( (objid = tk_cre_tsk( &t_ctsk )) <= E_OK ) // クライアントタスクの生成
        goto Err;
    if( tk_sta_tsk( objid, 1 ) < E_OK )      // クライアントタスクの起動
        goto Err;
    tk_slp_tsk( TMO_FEVR );                 // 通信端点IDはタスクの起動コードに設定
    tcpudp_close();                         // 通信の終了を待つ
    lan_close();                            // TCP/UDP通信を終了
    Err:
    while( 1 ) ;
}

```

usermain関数では、LANソケットのオープン、TCP/UDP通信のオープン、NTPクライアントのタスク生成と起動を行っています。使用するUDP通信端点IDはタスクの起動コード、拡張情報に初期化タスクのID番号を設定しています。

本サンプルでは、初期化タスクはtk_slp_tskで起床待ちとなります。何らかの要因によりNTPクライ

ントのタスクが終了すると起床待ちが解除され、TCP/UDP通信のクローズ、LANソケットのクローズを行います。

6.5.2 ntp_client_tsk

```
// UTC時刻のオーバーフローを管理するマクロ名です。本プログラム起動時の時刻が
// 2036年2月6日6時28分15秒を超えていない場合は 0 を指定します。
// 2036年2月6日6時28分15秒を超える場合は 1 を指定します。
// 以後、4, 294, 967, 295秒を超える毎に +1 を指定します。
#define UTCBASE 0 // 現在は2036年2月6日6時28分15秒 未満
// 日本の標準時刻を使用するか、グリニッチ標準時刻を使用するかを指定します。
// グリニッチ標準時刻を使用の場合は以下のマクロ名をコメントアウトします。
#define JPN TIME // 日本の標準時刻を使用
// NTPサーバのIPアドレスです。1つだけマクロ名を有効にします。
#define NTPSERVER 0x85F3EEF3 // 133. 243. 238. 243 NICT(独立行政法人情報通信研究機構)
// #define NTPSERVER 0x850F4008 // 133. 15. 64. 8 豊橋技術科学大学

typedef struct { // NTPパケット構造
    UB li_vn_mode, stratum, poll, precision;
    UW rootDelay, rootDispersion, refId;
    UW refTm_s, refTm_f, origTm_s, origTm_f, rxTm_s, rxTm_f, txTm_s, txTm_f;
} NTP_PAKET;
LOCAL NTP_PAKET buf; // 送受信バッファ(NTPパケット)

#ifdef __LIT
#define TS(x) __revl(x)
#else
#define TS(x) (x)
#endif

LOCAL void ntp_client_tsk(INT stacd, void *exinf)
{
    LOCAL T_IPV4EP dstaddr;
    LOCAL SYSTIM tim;
    ID cepid = stacd;
    D t1, t2, t3, t4;
    dstaddr.ipaddr = NTPSERVER; // サーバのIPアドレスを設定
    dstaddr.portno = 123; // サーバのポート番号を設定
    // 自局のポート番号はCFG_UDP_CEPID#_PORT_NUMBERで123を指定
    t1 = 4294967296000 * UTCBASE; // UTCのベース時刻を設定
    tim.hi = t1 >> 32; tim.lo = t1; // システム時刻用の変数を初期化
    tk_set_utc( &tim ); // UTC時刻を初期化
    while( 1 ) {
        buf.li_vn_mode = 0x23; // LI, VN, MODEを設定
        buf.poll = 17; // Pollを設定
        buf.stratum = buf.precision = buf.rootDelay = buf.rootDispersion = buf.refId = 0; // UTC時刻を参照
        tk_get_utc( &tim ); // UTC時刻を参照
        UTCtoTS( tim.hi, tim.lo, &buf.txTm_s, &buf.txTm_f ); // 送信のタイムスタンプを設定
        t1 = ( ( t1 = tim.hi ) << 32 ) + tim.lo; // t1のUTC時刻(64ビット)を設定
        if( udp_snd_dat( cepid, &dstaddr, &buf, sizeof(buf), 3000 ) != sizeof(buf) ) // サーバへデータ送信
            break;
        if( udp_rcv_dat( cepid, &dstaddr, &buf, sizeof(buf), 3000 ) != sizeof(buf) ) // サーバからデータ受信
            break;
        tk_get_utc( &tim ); // t4のUTC時刻を参照
        UTCtoTS( tim.hi, tim.lo, &buf.refTm_s, &buf.refTm_f ); // t4のタイムスタンプを設定
        t4 = ( ( t4 = tim.hi ) << 32 ) + tim.lo; // t4のUTC時刻(64ビット)を設定
        TStoUTC( buf.rxTm_s, buf.rxTm_f, &tim.hi, &tim.lo ); // t2をUTC時刻に変換
        t2 = ( ( t2 = tim.hi ) << 32 ) + tim.lo; // t2のUTC時刻(64ビット)を設定
        TStoUTC( buf.txTm_s, buf.txTm_f, &tim.hi, &tim.lo ); // t3をUTC時刻に変換
        t3 = ( ( t3 = tim.hi ) << 32 ) + tim.lo; // t3のUTC時刻(64ビット)を設定
        t1 = ( ( t2 - t1 ) + ( t3 - t4 ) ) / 2; // オフセットを算出
        tk_get_utc( &tim ); // UTC時刻を参照
        t1 += t4 = ( ( t4 = tim.hi ) << 32 ) + tim.lo; // 同期UTC時刻を算出
        t1 += t4 = ( t4 / 1000 >> 32 << 32 ) * 1000; // 2036年問題を補正
    }
}
```

```

        tim.hi = t1 >> 32; tim.lo = t1; // 同期UTC時刻を設定
        tk_set_utc( &tim ); // 同期UTC時刻を設定
        UTCtoDATETIM( tim.hi, tim.lo ); // UTC時刻を表示
        buf.origTm_s = buf.txTm_s; buf.origTm_f = buf.txTm_f; // 次回のt3をバッファに設定
        buf.rxTm_s = buf.refTm_s; buf.rxTm_f = buf.refTm_f; // 次回のt4をバッファに設定
        UTCtoTS( tim.hi, tim.lo, &buf.refTm_s, &buf.refTm_f ); // タイムスタンプを設定
        tk_dly_tsk( 1000 * 60 * 60 * 24 ); // 24時間待ち
    }
    tk_wup_tsk( (ID)exinf ); // 初期化タスクを起床
    tk_ext_tsk( ); // タスクの終了
}

```

起動されたら、UTC時刻でシステム時刻を初期化後、NTP起動コードで渡されたUDP通信端点IDを使って、NTPサーバーを利用した時刻合わせを24時間毎に行います。

NTPサーバーのIPアドレスはNTPSERVERマクロで定義します。Web等で検索し、利用可能なNTPサーバーのIPアドレスを指定してください。

何らかの原因によりUDPパケットの送受信が行えない場合は拡張情報で指定された初期化タスクを起床し、自タスクを終了します。

6.5.3 コンフィグレーション・ファイル（主要項目のみ）

```

#define CFG_SYSTEM_DHCP          (1)
#define CFG_UDP_CEPID_NUM        (1)
#define CFG_UDP_CEPID1_CHANNEL   (0)
#define CFG_UDP_CEPID1_PORT_NUMBER (123)

```

DHCPは有効です。UDPの通信端点が最低でも1つは必要であり、ポート番号には123を指定します。

7. 問い合わせ先

本実装に関する問い合わせや他のRXファミリへのポーティングに関する相談は以下のメールアドレス宛にお願い致します。

yuji_katori@yahoo.co.jp

トロンフォーラム学術・教育WGメンバ

鹿取 祐二（かとり ゆうじ）

なお、上記のメールアドレスは余儀なく変更される場合がありますが、その際はご了承ください。

以上