Ethernet スタータキット RX65N
ソフトウェア編　取扱説明書(3)

ルネサス エレクトロニクス社　RX マイコン搭載
HSB シリーズマイコンボード　評価キット

-本書を必ずよく読み、ご理解された上でご利用ください

株式会社北斗電子
REV.1.0.0.0

ー目　次ー

注意事項 **1**

安全上のご注意 **2**

1. 本キットのサンプルプログラムに関して **4**

2. RX65N_MAGIC_PACKET プロジェクト **4**

3. RX65N_MAGIC_PACKET2 プロジェクト **4**

4. RX65N_ETHER_NOAPI プロジェクト **4**

5. RX65N_PING プロジェクト **4**

6. 構造体定義 **4**

7. RX65N_UDP プロジェクト **4**

8. RX65N_TCP プロジェクト **4**

9. RX65N_LWIP プロジェクト **5**

9.1. lwIP(Light weight Internet Protocol)とは 5
9.2. スマート・コンフィグレータの追加コンポーネント 6
9.3. ユーザ作成コード 13
9.4. プログラムの動作 13
9.4.1. TCP クライアントとしての動作 16
9.4.2. TCP サーバとしての動作 17
9.4.3. UDP の送信動作 18
9.4.4. UDP の受信動作 19
9.5. メイン関数での処理 20
9.5.1. LWIP の初期化 22
9.5.1. メインループ内で呼び出す LWIP の API 関数 22
9.6. TCP クライアント 23
9.6.1. 接続処理 23
9.6.2. データ送信処理 28
9.6.1. 切断処理 29
9.7. TCP サーバ 30
9.7.1. 待ち受け開始処理 30
9.8. UDP クライアント 34
9.8.1. データ送信処理 34
9.9. UDP サーバ 36
9.9.1. 待ち受け開始処理 36
9.10. ping 38
9.10.1. ping リクエスト 38
9.10.2. ping リプライ 40
9.11. ユーザ作成関数 42

9.12. MAC アドレスの設定に関して 46
10. RX65N_UART_ETHER_UDP プロジェクト 48
10.1. メイン関数での処理 52
10.2. UDP での送信処理 54
10.3. UDP での受信処理 55
10.4. ユーザ作成関数 57
11. RX65N_UART_ETHER_TCP_DMAC プロジェクト 59
11.1. メイン関数での処理 64
11.2. TCP の接続処理（クライアント動作） 66
11.3. TCP の送信処理（クライアント動作） 67
11.4. TCP の受信処理（サーバ動作） 69
11.5. ユーザ作成関数 71
取扱説明書改定記録 74
お問合せ窓口 74

注意事項

本書を必ずよく読み、ご理解された上でご利用ください

【ご利用にあたって】

1. 本製品をご利用になる前には必ず取扱説明書をよく読んで下さい。また、本書は必ず保管し、使用上不明な点がある場合は再読し、よく理解して使用して下さい。
2. 本書は株式会社北斗電子製マイコンボードの使用方法について説明するものであり、ユーザシステムは対象ではありません。
3. 本書及び製品は著作権及び工業所有権によって保護されており、全ての権利は弊社に帰属します。本書の無断複写・複製・転載はできません。
4. 弊社のマイコンボードの仕様は全て使用しているマイコンの仕様に準じております。マイコンの仕様に関しましては製造元にお問い合わせ下さい。弊社製品のデザイン・機能・仕様は性能や安全性の向上を目的に、予告無しに変更することがあります。また価格を変更する場合や本書の図は実物と異なる場合もありますので、御了承下さい。
5. 本製品のご使用にあたっては、十分に評価の上ご使用下さい。
6. 未実装の部品に関してはサポート対象外です。お客様の責任においてご使用下さい。

【限定保証】

1. 弊社は本製品が頒布されているご利用条件に従って製造されたもので、本書に記載された動作を保証致します。
2. 本製品の保証期間は購入戴いた日から1年間です。

【保証規定】

保証期間内でも次のような場合は保証対象外となり有料修理となります

1. 火災・地震・第三者による行為その他の事故により本製品に不具合が生じた場合
2. お客様の故意・過失・誤用・異常な条件でのご利用で本製品に不具合が生じた場合
3. 本製品及び付属品のご利用方法に起因した損害が発生した場合
4. お客様によって本製品及び付属品へ改造・修理がなされた場合

【免責事項】

弊社は特定の目的･用途に関する保証や特許権侵害に対する保証等、本保証条件以外のものは明示・黙示に拘わらず一切の保証は致し兼ねます。また、直接的・間接的損害金もしくは欠陥製品や製品の使用方法に起因する損失金・費用には一切責任を負いません。損害の発生についてあらかじめ知らされていた場合でも保証は致し兼ねます。
ただし、明示的に保証責任または担保責任を負う場合でも、その理由のいかんを問わず、累積的な損害賠償責任は、弊社が受領した対価を上限とします。本製品は「現状」で販売されているものであり、使用に際してはお客様がその結果に一切の責任を負うものとします。弊社は使用または使用不能から生ずる損害に関して一切責任を負いません。
保証は最初の購入者であるお客様ご本人にのみ適用され、お客様が転売された第三者には適用されません。よって転売による第三者またはその為になすお客様からのいかなる請求についても責任を負いません。
本製品を使った二次製品の保証は致し兼ねます。

安全上のご注意

製品を安全にお使いいただくための項目を次のように記載しています。絵表示の意味をよく理解した上でお読み下さい。

表記の意味

警告	取扱を誤った場合、人が死亡または重傷を負う危険が切迫して生じる可能性がある事が想定される
注意	取扱を誤った場合、人が軽傷を負う可能性又は、物的損害のみを引き起こすが可能性がある事が想定される

絵記号の意味

	**一般指示** 使用者に対して指示に基づく行為を強制するものを示します		**一般禁止** 一般的な禁止事項を示します
	**電源プラグを抜く** 使用者に対して電源プラグをコンセントから抜くように指示します		**一般注意** 一般的な注意を示しています

警告

以下の警告に反する操作をされた場合、本製品及びユーザシステムの破壊・発煙・発火の危険があります。マイコン内蔵プログラムを破壊する場合もあります。

1. 本製品及びユーザシステムに電源が入ったままケーブルの抜き差しを行わないでください。
2. 本製品及びユーザシステムに電源が入ったままで、ユーザシステム上に実装されたマイコンまたはIC等の抜き差しを行わないでください。
3. 本製品及びユーザシステムは規定の電圧範囲でご利用ください。
4. 本製品及びユーザシステムは、コネクタのピン番号及びユーザシステム上のマイコンとの接続を確認の上正しく扱ってください。

発煙・異音・異臭にお気付きの際はすぐに使用を中止してください。

電源がある場合は電源を切って、コンセントから電源プラグを抜いてください。そのままご使用すると火災や感電の原因になります。

⚠注意

以下のことをされると故障の原因となる場合があります。

1. 静電気が流れ、部品が破壊される恐れがありますので、ボード製品のコネクタ部分や部品面には直接手を触れないでください。
2. 次の様な場所での使用、保管をしないでください。
   ホコリが多い場所、長時間直射日光があたる場所、不安定な場所、衝撃や振動が加わる場所、落下の可能性がある場所、水分や湿気の多い場所、磁気を発するものの近く
3. 落としたり、衝撃を与えたり、重いものを乗せないでください。
4. 製品の上に水などの液体や、クリップなどの金属を置かないでください。
5. 製品の傍で飲食や喫煙をしないでください。

ボード製品では、裏面にハンダ付けの跡があり、尖っている場合があります。

取り付け、取り外しの際は製品の両端を持ってください。裏面のハンダ付け跡で、誤って手など怪我をする場合があります。

CD メディア、フロッピーディスク付属の製品では、故障に備えてバックアップ(複製)をお取りください。

製品をご使用中にデータなどが消失した場合、データなどの保証は一切致しかねます。

アクセスランプがある製品では、アクセスランプの点灯中に電源を切ったり、パソコンをリセットをしないでください。

製品の故障や、データ消失の原因となります。

本製品は、医療、航空宇宙、原子力、輸送などの人命に関わる機器やシステム及び高度な信頼性を必要とする設備や機器などに用いられる事を目的として、設計及び製造されておりません。

医療、航空宇宙、原子力、輸送などの設備や機器、システムなどに本製品を使用され、本製品の故障により、人身や火災事故、社会的な損害などが生じても、弊社では責任を負いかねます。お客様ご自身にて対策を期されるようご注意ください。

1. 本キットのサンプルプログラムに関して

2. RX65N_MAGIC_PACKET プロジェクト

3. RX65N_MAGIC_PACKET2 プロジェクト

4. RX65N_ETHER_NOAPI プロジェクト

5. RX65N_PING プロジェクト

6. 構造体定義

上記の内容は、Ethernet スタータキット RX65N ソフトウェア編　取扱説明書(1)に記載しています。

7. RX65N_UDP プロジェクト

8. RX65N_TCP プロジェクト

上記の内容は、Ethernet スタータキット RX65N ソフトウェア編　取扱説明書(2)に記載しています。

9. RX65N_LWIP プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_LWIP
SOURCE¥176PIN¥RX65N_LWIP

（使用しているマイコンボードのピン数に応じたフォルダ）を PC のストレージにコピーしてください。

本プロジェクトは、
RX65N_UDP
RX65N_TCP
プロジェクトと同等の動作を行いますが、UDP/TCP のプロトコルの取り扱いは、lwIP(LWIP)のプロトコルスタックに置き換えています。

RX シリーズの TCP/IP のプロトコルスタックとしては、長年 M3S-T4-Tiny が使用されてきましたが、2025 年末でサポート終了となりました。M3S-T4-Tiny の後継としては lwIP（もしくは市販のプロトコルスタック）が推奨となっています。

9.1. lwIP(Light weight Internet Protocol)とは

・オープンソース（BSD ライセンス）
・組み込み用に軽量に設計されたプロトコルスタック
・IP, ICMP（ping 等）, UDP, TCP などをサポート
が特徴のプロトコルスタック（→ソフトウェアのライブラリ）です。

RX 以外のマイコンでも世界中で幅広く使用されており、PC に比べて圧倒的に使用可能なメモリが少ないマイコンの世界でも動作する様に設計されています。

lwIP を使う場合、
・raw API
・netconn API
・socket API
の API が用意されています。下に行くほど抽象度は高くなります（＝泥臭くなくなります）。

操作は面倒だが、軽量でサクッと動くのが raw API。PC のネットワークプログラミングに近いのが socket API となります。socket API を使う場合は、OS との組み合わせが前提となります。

本プロジェクトでは、raw API を使います。

raw AIP は socket API に比べると、煩雑な点がありますが、RX65N_TCP のプロジェクトと比べると、かなりすっきりとした作りになるのではないかと思います。

OS は導入せずに TCP/IP を使いたいという場合、lwIP の raw API を使用するというのが、現状では第一の選択肢となるのではないかと思います。（もちろん、市販のプロトコルスタックを購入するという方法もあります。lwIP ですと、追加費用なしで試せますので、とりあえずは lwIP の raw API でやってみてから判断する形でも良いと考えます。）

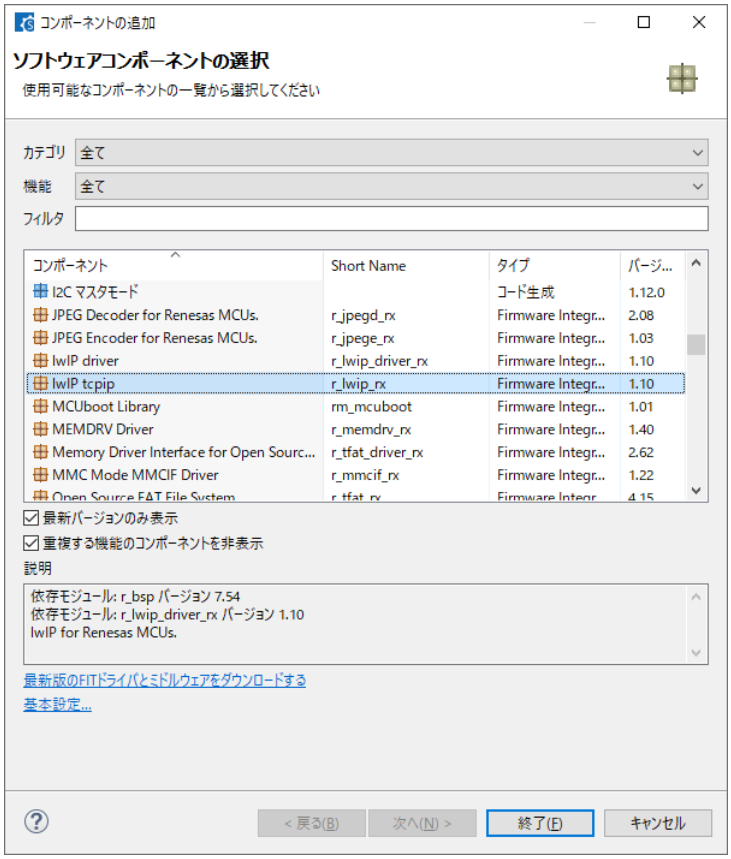
9.2. スマート・コンフィグレータの追加コンポーネント

・スマート・コンフィグレータで追加するコンポーネント

コンポーネント	備考
Config_PORT	SW と LED のポート設定
Config_SCI1	SCI(UART)通信
r_ether_rx	Ethernet
r_cmt_rx	コンペアマッチタイマ
r_lwip_driver_rx	lwIP ドライバ
r_lwip_rx	lwIP TCP/IP

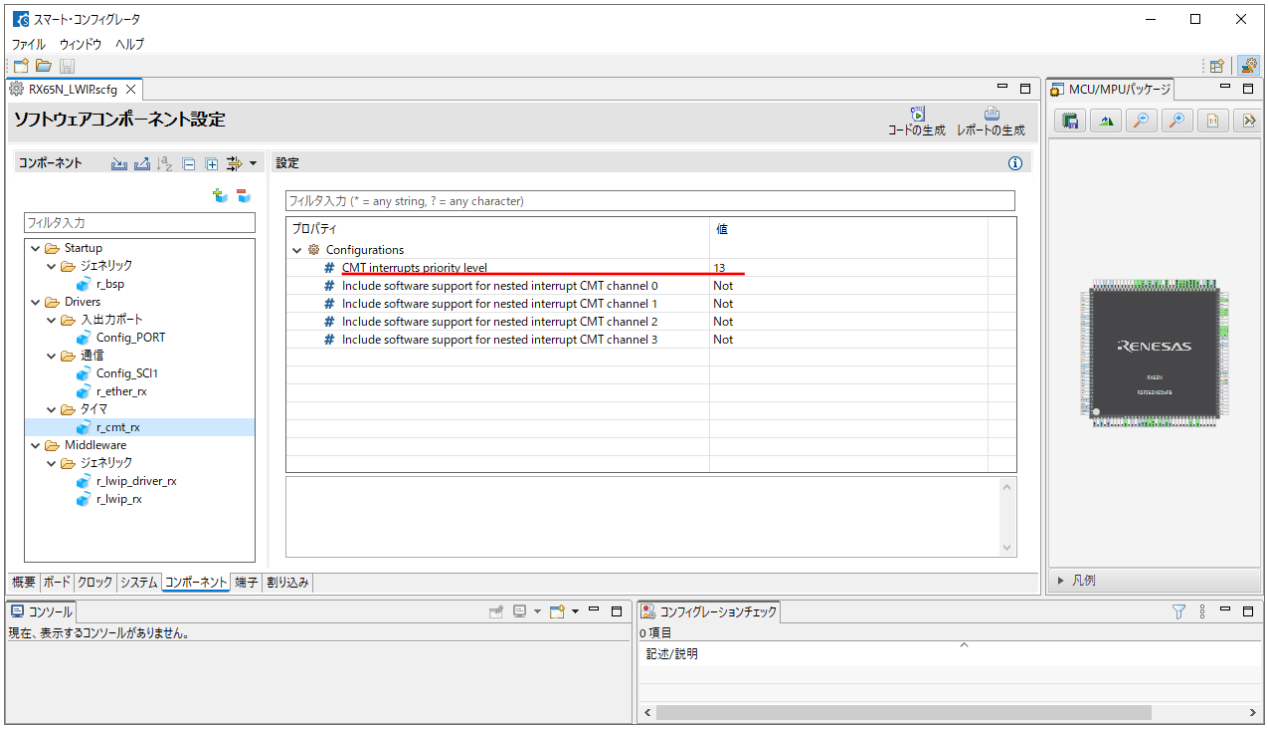
r_lwip_rx(lwIP tcpip)を追加すると、自動的に r_lwip_driver_rx, r_cmt_rx が追加されます。
（r_ether_rx は、先に追加しておくのが良いです。）

コンポーネントの追加で、lwIP tcpip を追加してください。



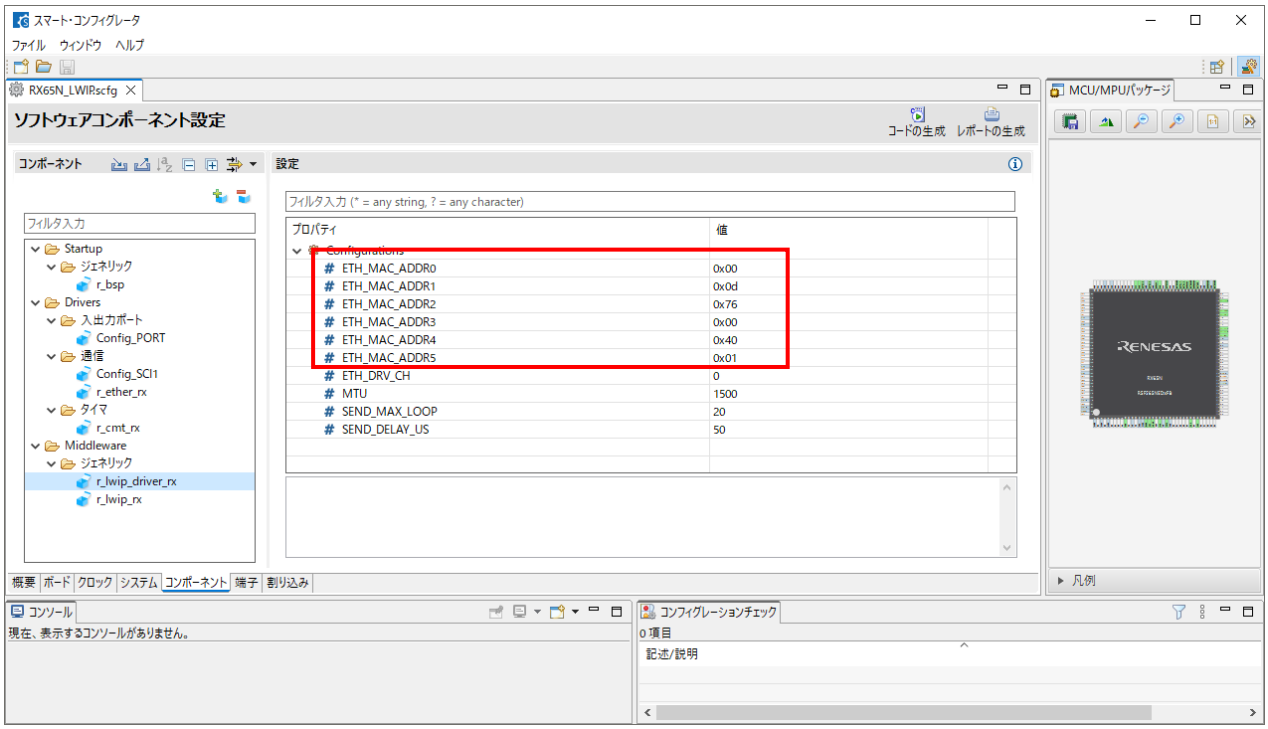
一覧に表示されない場合は、「最新版の FIT ドライバとミドルウェアをダウンロードする」からダウンロードしてください。バージョンは、1.10 以上である必要があります。

・r_cmt_rx



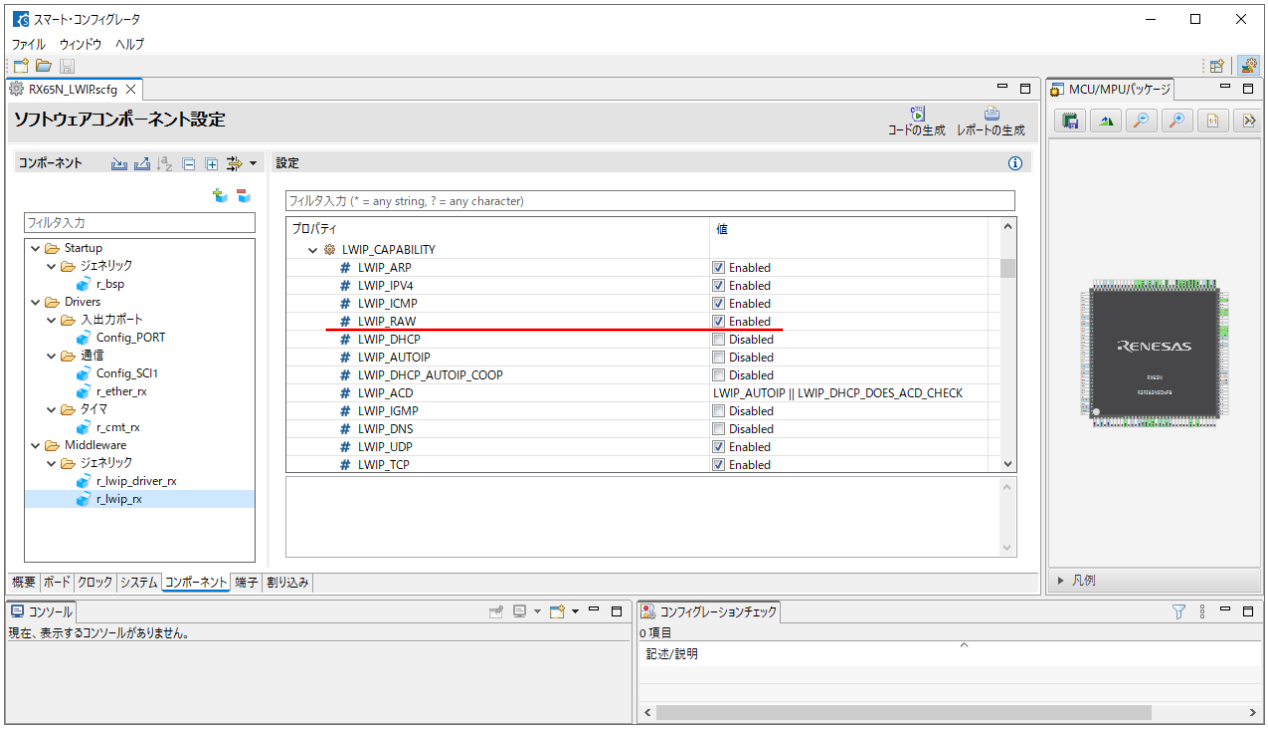
本プロジェクトでは、コンペアマッチタイマの割り込み優先度を 13（高め）に設定しています。
→デフォルトは 5 です、設定値は任意です

・r_lwip_driver



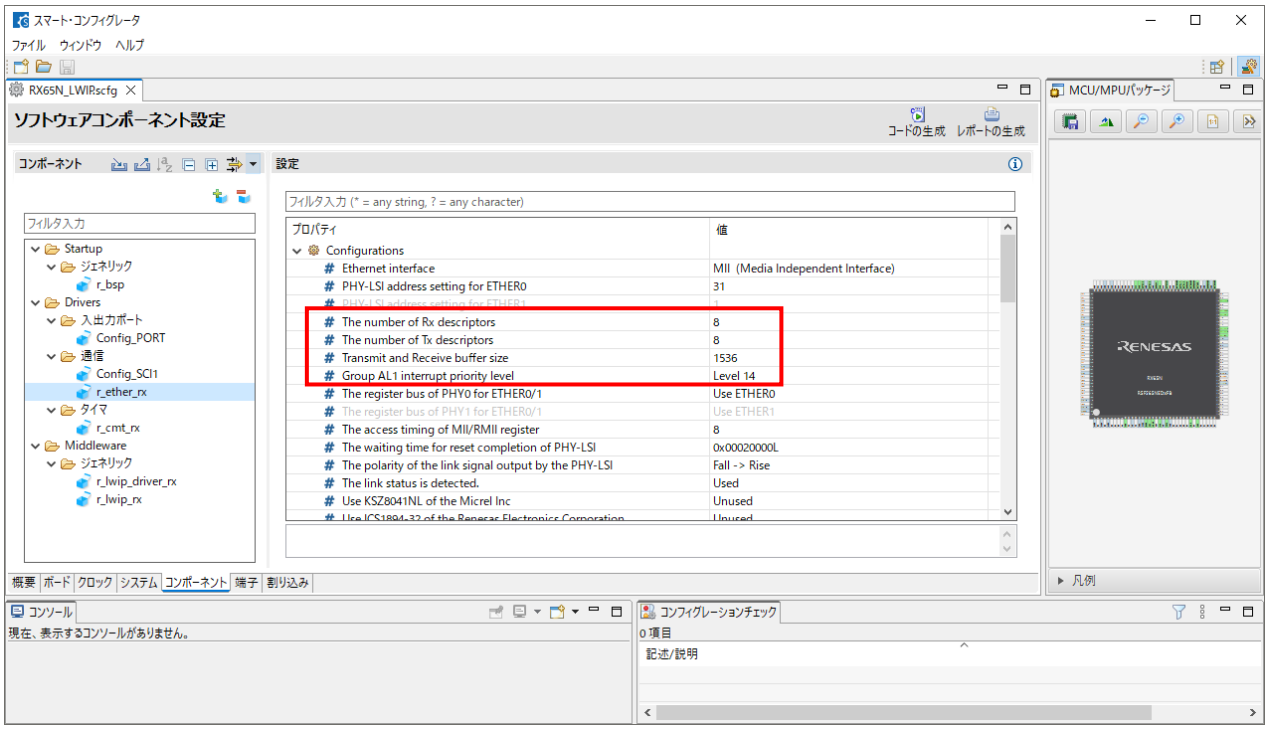
赤枠内に、ボードに設定する MAC アドレスを設定してください。今までのプロジェクトでは、ether_operation.c 内に値を定義していましたが、r_lwip_driver_rx コンポーネントを使った場合、上記で設定する必要があります。

・r_lwip_rx



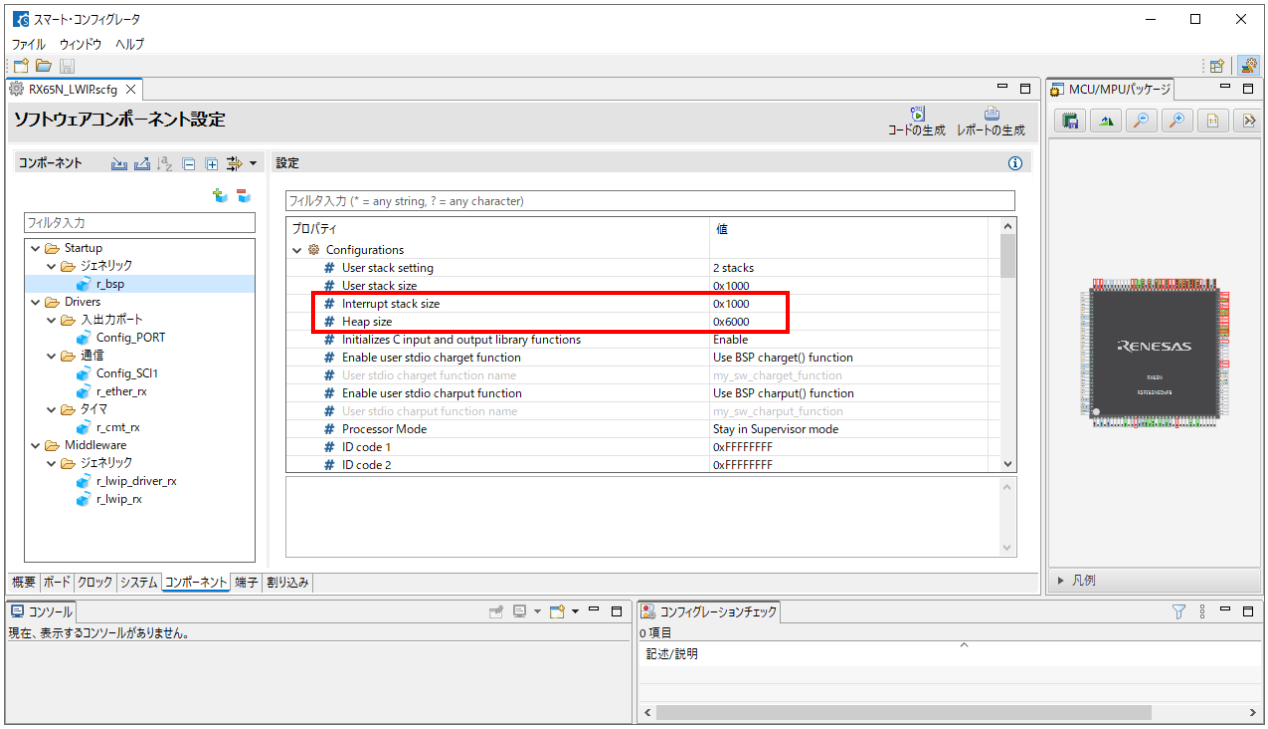
LWIP_RAW にチェックを追加してください。(Enabled にチェック)

・r_ether_rx



使用するディスクリプタの数には制約はありませんが、本プロジェクトでは、どちらも 8 に設定しています。また割り込み優先度は 14 に設定しています。(どちらも任意)

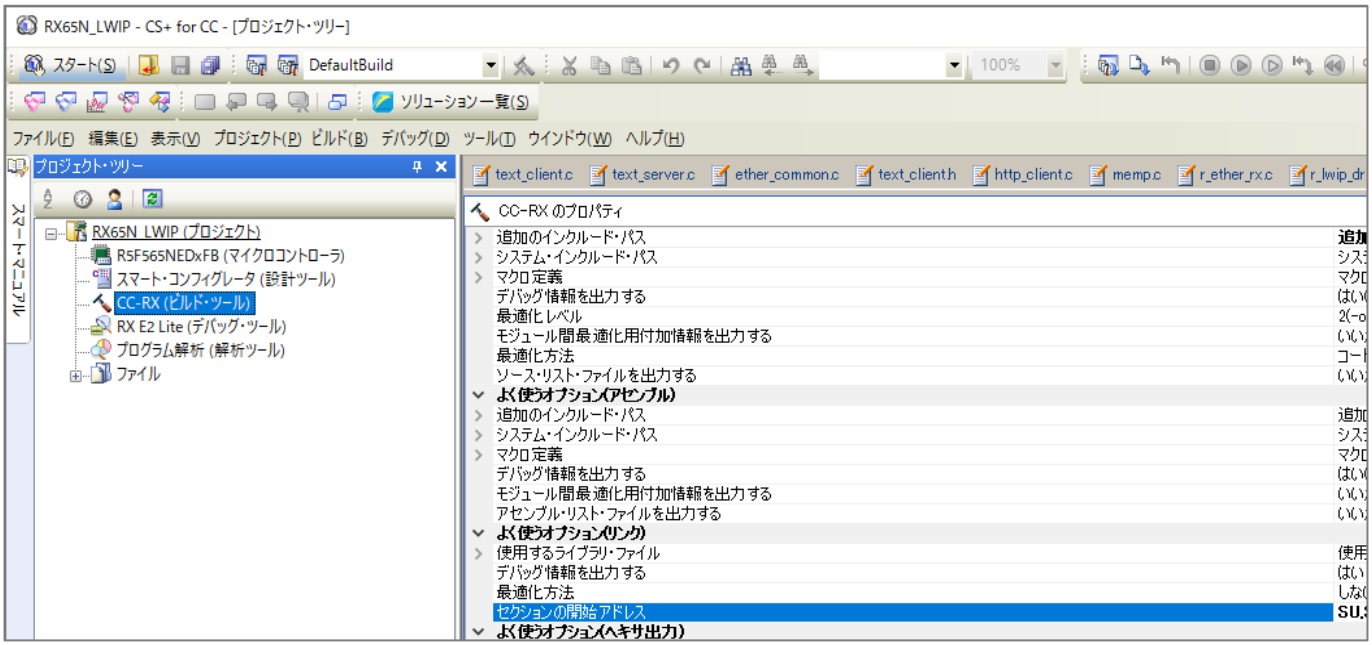
・r_bsp

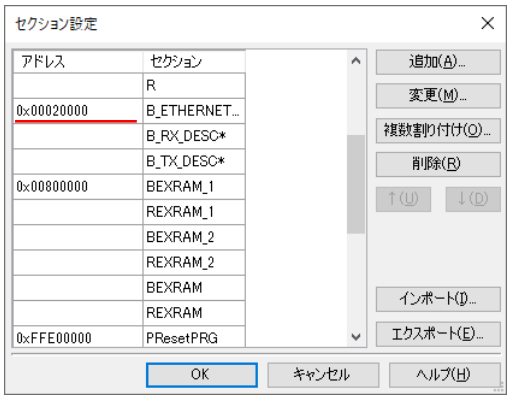


interrupt stack size はデフォルト 0x400 ですが、0x1000 以上に増やしてください。

Heap size は、デフォルト 0x400 ですが、20kB 以上に増やしてください（0x6000 など）

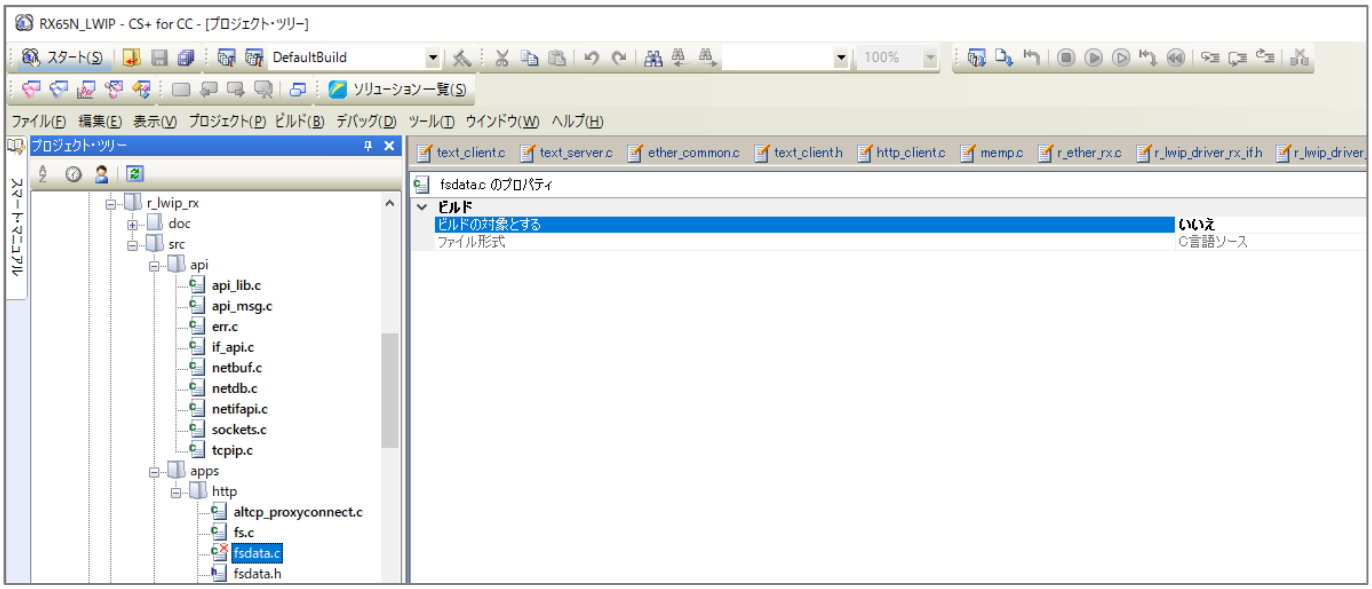
・セクションアドレス





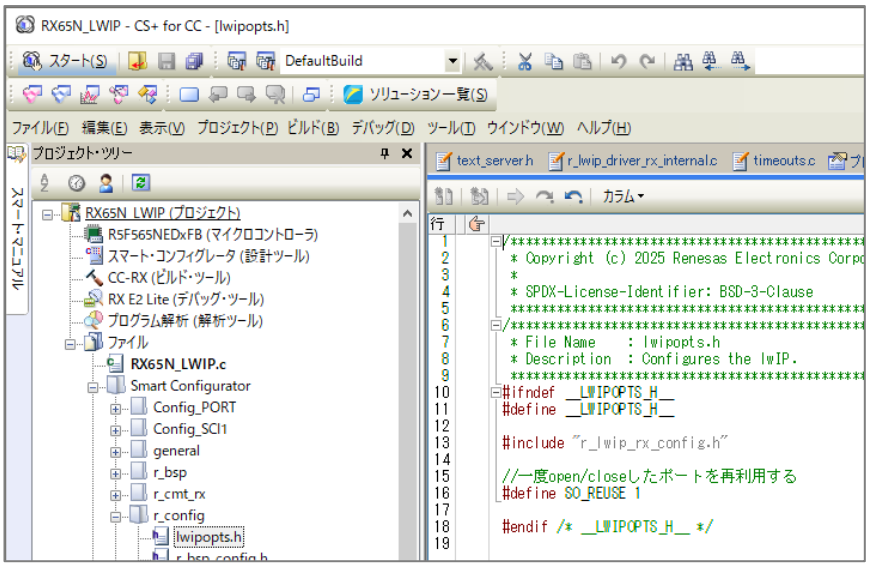
B_ETHERNET が割り当てられているセクションの開始アドレスを、0x20000 等にずらしてください。
（デフォルトは、0x10000）

各種設定後、「コード生成」を押して、コード生成後、プロジェクトツリーから



r_lwip_rx¥src¥apps¥http¥fsdata.c を選択して、ビルド対象から外してください。

・生成コードへの追加



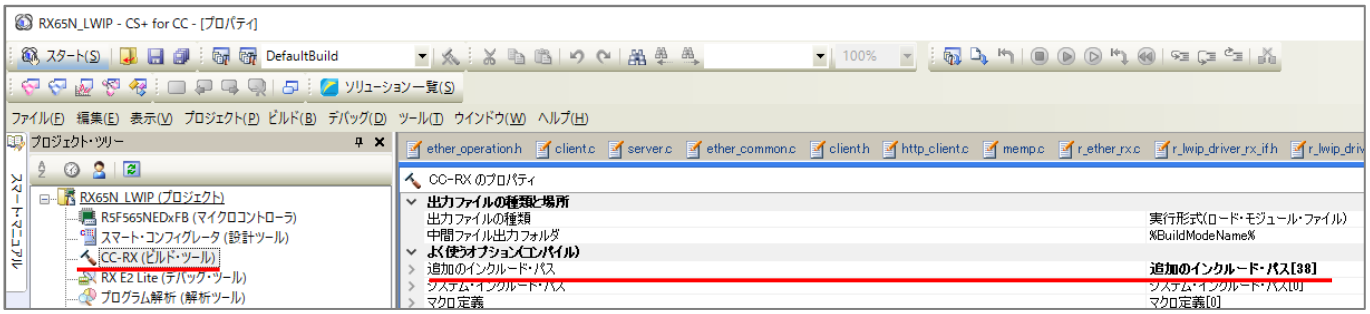
r_config¥lwipopts.h に

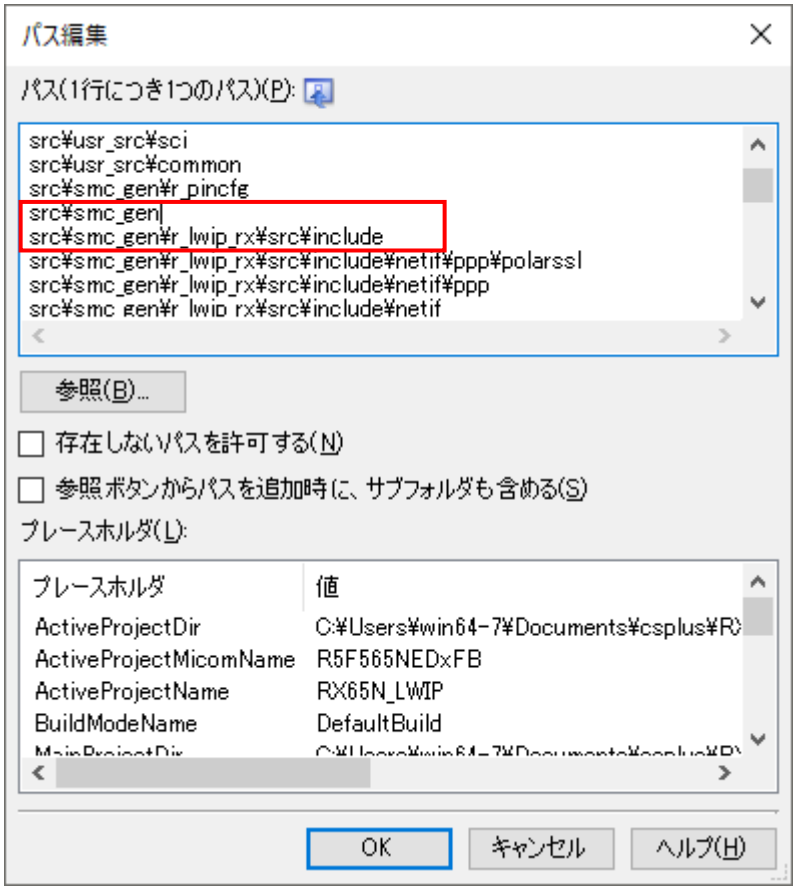
・lwipopts.h

```
//一度open/closeしたポートを再利用する
#define SO_REUSE 1
```

を追加してください。（サーバ側動作で、一度切断した後で再度接続する場合に必要です。）

・インクルードパスの追加





src¥smc_gen

src¥smc_gen¥r_lwip_rx¥src¥include

をインクルードパスに追加。

（PDF ファイルの上記 2 行をコピー&ペーストすると、！が付き「指定したパスに存在しないフォルダが含まれます」というエラーとなります。PDF ファイル上は、¥→UTF8 で 0xC2A5 という文字コードになるので、コピー&ペーストする場合は¥はキーボードから上書きしてください。ファイルパスの¥は ASCII で文字コード 0x5C である必要があります。）

9.3. ユーザ作成コード

・ユーザ作成のプログラムコード

フォルダ（カテゴリ）	ファイル名	備考
client	client.c	クライアント動作のコード
	client.h	上記ヘッダ
common	ether_common.c	Ether のプログラムで共通に使用する関数
	ether_common.c	上記ヘッダ
	ether_operation.c	IP アドレスなどの定義
	ether_operation.h	上記ヘッダ
main	main.c	メイン関数
ping	ping.c	ping を取り扱う関数
	ping.h	上記ヘッダ
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ
server	server.c	サーバト動作のコード
	server.h	上記ヘッダ

IP や UDP, TCP が lwIP 側で処理されますので、ユーザ作成コードは大分スリムになります。

9.4. プログラムの動作

起動後、以下の様なメッセージが表示されます。

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether LwIP sample program.

COMMAND:
   --message setting command
   m : send text message set(TCP)
   r : response text message set(TCP)
   M : send text message set(UDP)
   p : send text message print
   --TCP command
   s : send text message(TCP)
   c : connect tcp session
   d : disconnect tcp session
   D : force disconnect tcp session
   --UDP command
   S : send text message(UDP)
   X : udp listen end
   Z : udp listen restart

-setting address-
---
this board IP address                -> 192.168.0.80 (MAC address -> 00-0D-76-00-40-01)
netmask                                 -> 255.255.255.0
gateway IP address                   -> 192.168.0.1
communication target IP address -> 192.168.0.81
---

address setting change -> Please input 'A' within 10 seconds.
>
```

デフォルトでは、

本ボードの IP アドレス　192.168.0.80 （MAC アドレス　00-0D-76-00-40-01）

ネットマスク　255.255.255.0

ゲートウェイアドレス　192.168.0.1
通信相手の IP アドレス　192.168.0.81
TCP サーバの待ち受けポート　20000
UDP サーバの待ち受けポート　30000
TCP クライアントの接続先ポート　20000
UDP クライアントの接続先ポート　30000

です。(起動時に DIP-SW の 3 が ON になっていると、送信先と本ボードの IP アドレスが逆に設定されます。)

これらの値を変えたい場合は、起動後 10 秒以内に、キーボードから A を入力してください。

A 入力時は、

```
-- IP address settings change --

COMMAND:
  1 : this board
  2 : netmask
  3 : gateway
  4 : communication target
  5 : TCP server port (default = 20000)
  6 : UDP servrt port (default = 30000)
  7 : TCP client port (default = 20000)
  8 : UDP client port (default = 30000)
  p : print setting
  e : exit setting
```

1~8 で上記のアドレスとポート番号の設定値を変更可能です。
※通信相手の IP アドレス変更は 4 番です(前のプロジェクトでは 1)

10 秒経過後、または IP アドレスなどの設定後に、動作開始します。

```
network operation start!
-- TCP SERVER LISTEN START (port = 20000) --
-- UDP SERVER LISTEN START (port = 30000) --
```

・SW3 で ping を送る
・ARP リクエストが来た場合は ARP リプライを返す
・ping が飛んできた場合は応答を返す
という動作は、RX65N_PING プロジェクトから引き継がれて、本プロジェクトでも有効です。

・ether_operation.c

```
//本ボードのMACアドレス
unsigned char g_src_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x01};//00-0D-76 HokuotoDenshi vender code,
本値は設定されない（スマート・コンフィグレータでの r_lwip_driver_rx で設定した値が有効）
unsigned char g_src_alt_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x02};

//本ボードのIPアドレス（,で区切る事に注意）※接続するネットワーク応じて変更
unsigned char g_src_IP[4] = {192,168,0,80};
unsigned char g_src_alt_IP[4] = {192,168,0,81};

//通信相手のIPアドレス（,で区切る事に注意）※本プログラムでは、ここで設定した相手にTCP/UDPでテキストメッセ
ージやり取りする
unsigned char g_dst_IP[4] = {192,168,0,81};
unsigned char g_dst_alt_IP[4] = {192,168,0,80};

/*
 * 起動時ボードのDIP-SWでIPアドレスとMACアドレスの設定が変わります
 *
 * SW2-3(UPSEL)が OFF側の場合は
 * スマート・コンフィグレータで設定したMACアドレス，g_src_IP，g_dst_IP が使われます
 *
 * SW2-3(UPSEL)が ON側の場合は
 * g_src_alt_MAC，g_src_alt_IP，g_dst_alt_IP が使われます
 *
 */

//ネットマスク
unsigned char g_netmask[4] = {255,255,255,0};

//ゲートウェイIPアドレス
unsigned char g_gateway_IP[4] = {192,168,0,1};

//TCPクライアント時：宛先ポート番号（サーバ側はこの番号で待ち受けている事を想定）
unsigned short g_tcp_client_dst_port = 20000;

//TCPサーバ時：相手から見た宛先ポート番号（本ボード，この番号で待ち受け）
unsigned short g_tcp_server_listen_port = 20000;

//UDPクライアント時：宛先ポート番号（サーバ側はこの番号で待ち受けている事を想定）
unsigned short g_udp_client_dst_port = 30000;

//UDPサーバ時：相手から見た宛先ポート番号（本ボード，この番号で待ち受け）
unsigned short g_udp_server_listen_port = 30000;
```

IP アドレス、使用ポート番号などの初期値は、ether_operation.c 内に定義されています。デフォルト値を変える場合は、このファイル内の値を変更してビルドしてください。

DIP-SW(3)=OFF の場合は、
g_src_IP　ボードの IP アドレス
g_dst_IP　通信相手の IP アドレス
の設定が使用されます。

DIP-SW(3)=ON の場合は、
g_src_alt_MAC　本ボードの MAC アドレス
g_src_alt_IP　ボードの IP アドレス
g_dst_alt_IP　通信相手の IP アドレス
の設定が使用されます。

g_src_MAC の値は、起動時の画面表示にのみ使用されます。

動作上、MAC アドレスを変えたい場合は、
(1)スマート・コンフィグレータの r_lwip_driver_rx 設定を変更する
(2)g_src_MAC の値を設定
main.c 内の static int mac_address_replace = FALSE;を TRUE に変更
のどちらかで変更可能です。

9.4.1. TCP クライアントとしての動作

キーボードから c コマンドを入力すると、、設定した送信先と TCP での接続を確立します。

c コマンドの入力

```
>command=c
connect 192.168.0.81 port 20000
-- TCP CONNECTION ESTABLISHED --
```

TCP CONNECTION ESTABLISHED と表示されれば、TCP での接続は成功しています。

キーボードから s コマンドを入力すると、接続確立した相手に TCP でメッセージを送信します。

s コマンドの入力

```
>command=s
TCP send text message to 192.168.0.81 port 20000
send message.
message received (TCP) ->
OK,      (RX65)     (TCP server)
```

"OK,　　(RX65)　(TCP server)　　"の部分は、こちらからのメッセージに関して相手が返信してきた部分です。
（接続相手が PC の場合は、この部分のデフォルトのメッセージは変わります。）
送信するメッセージは、デフォルトでは"hello, (RX65)　(TCP client)　　"です。
m コマンドで、送信メッセージは変更できます。
（本プログラムでは、メッセージの長さは 40 文字までとしています。送信するバイト数も常に 40 バイトとしています。）

d コマンドの入力

```
>command=d
disconnect
-- TCP CONNECTION CLOSED --
```

d コマンドは、TCP の接続を切断するコマンドです。
切断後は、s コマンドでメッセージを送る事はできません。

c コマンドで接続を確立
s コマンドでメッセージの送信
…
d コマンドで接続の切断

というのが、一連の動作となります。

9.4.2. TCP サーバとしての動作

起動後、デフォルトでは 20000 番のポートで TCP の接続を待ち受ける動作となります。(起動後 10 秒以内に A コマンドを入力する事で、待ち受けポート番号は変更できます。デフォルト値を変える場合は、ether_operation.c 内に記載されているポート番号を変更してください。)

ネットワーク動作開始後、別な機器の TCP クライアントから接続要求が来た場合は、以下の様に表示が出ます。

接続要求に対する反応

```
-- TCP CONNECTION ACCEPTED --
(connected from 192.168.0.81:52362)
```

TCP CONNECTION ACCEPTED と表示されれば、TCP での接続は成功しています。from の表示は、接続元の IP アドレスとポート番号です。

この状態で接続相手からのメッセージを受信した場合は、画面に表示されます。

メッセージを受信した場合

```
message received (TCP) ->
hello, (RX65)     (TCP client)
send response.
```

"hello, (RX65)　(TCP client)　　"が相手が送信してきたメッセージです。

この時、本ボードは相手に対して、"OK,　(RX65)　(TCP server)　　"というメッセージを送り返しています。送り返すメッセージは r コマンドで変更可能です。p コマンドで送信するメッセージを確認できます。

p コマンド

```
>command=p
send text message is ->
TCP client : "hello, (RX65)     (TCP client)                "
TCP server : "OK,       (RX65)     (TCP server)                "
UDP : "hello(RX65)(UDP)"
```

TCP クライアント側が送るメッセージ　m コマンドで変更　(前のプロジェクトでは t でしたが変わりました)
TCP サーバ側が送るメッセージ　r コマンドで変更
(UDP で送るメッセージ　M コマンドで変更)※UDP の動作は後述

クライアント動作時、s コマンドで送信するメッセージと、サーバ時返信するメッセージの確認。

サーバ動作時、接続相手から切断要求が来た場合

```
-- TCP SERVER LISTEN END --
-- TCP SERVER LISTEN START (port = 20000) --
```

TCP SERVER LISTEN END
で、現在の TCP セッションは切れます。その後、
TCP SERVER LISTEN START
で新しい接続待ちとなります。

切断後の状態は、起動時と同じ状態で、再度接続を受け付ける事ができます。
（1 回目と 2 回目で別なホストから接続要求が来ても問題ありません。）

本プロジェクトでは、待ち受けは 1 セッションのみで、同時に複数のクライアントからの接続要求を受け入れる事はできません。

サーバ動作とクライアントの動作は独立しているので、サーバ接続状態でもクライアント側がサーバ側で接続している機器、もしくは別な機器（別な IP アドレスの機器）と接続を確立できます。

c（接続）、s（送信）、d（切断）はクライアント側のコマンドです。サーバ動作時はサーバ側主体で「データ送信」「切断」するコマンドは用意していませんが、TCP の通信という観点では、サーバ側からデータを送信したり、切断しても問題はありません。

（一般的には、サーバ側は決められたポートで接続を待っているというだけで、接続が確立した後は、どちらかが主でもう一方は従であるといった関係性はありません。）

9.4.3. UDP の送信動作

キーボードから S コマンドを入力すると、設定した送信先に UDP でメッセージを送信します。

S コマンドの入力

```
>command=S
UDP send text message to 192.168.0.81 port 30000
```

送信するメッセージは、デフォルトでは"hello(RX65)(UDP)"です。M コマンドで、送信メッセージは変更できます。
（本プログラムでは、メッセージの長さは 16 文字までとしています。送信するバイト数も常に 16 バイトとしています。）

9.4.4. UDP の受信動作

UDP のデータ受信では、待ち受けポートを指定して、デフォルトでは 30000 番のポートで待ち受けています。待ち受けポートにデータが来た場合、下記の様な表示が出ます。

メッセージを受信した場合

```
message received (UDP) ->
hello(RX65)(UDP) (from 192.168.0.81:62404)
```

9.5. メイン関数での処理

main.c 内の ether_main()では、

・変数の初期化
・PHY チップのリセット
・DIP-SW のチェック
→DIP-SW の 3 番が ON の場合は、IP アドレス=192.168.0.81, MAC アドレス=00-0D-76-00-40-02 に設定
・SCI 初期化
・起動時のメッセージ表示
・LWIP ドライバ動作開始
・キーボードからの入力待ち（10 秒でタイムアウトして先へ進む）
・PHY チップのリセット解除
・LWIP 初期化
・ネットワークインタフェース設定
（・MAC アドレスの変更[変更する場合のみ]）
・Ether のリンク確認
・ネットワークインタフェース有効化
・メインループ

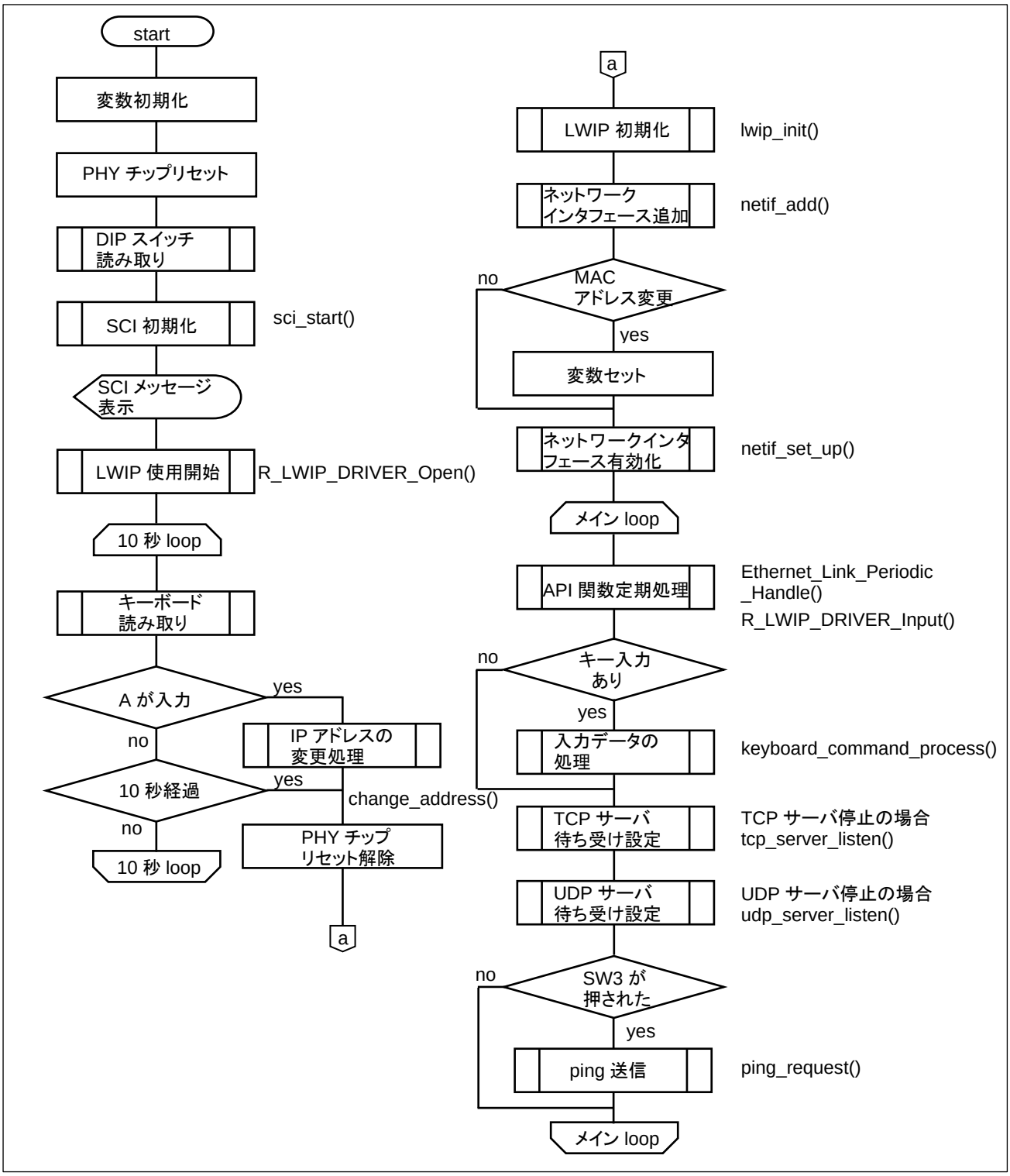
の処理を行っています。

メインループでは、

・API 関数の定期処理
・キーボードからのコマンドの処理
・TCP サーバ待ち受け処理
・UDP サーバ待ち受け処理
・SW3 が押されている場合は ping 送信

となっています。

・ether_main() （main.c 内）



本プロジェクトでは、起動後 10 秒経過すると、ネットワークの動作を開始します。起動後 10 秒以内に、キーボードから A を入力すると、アドレス設定変更処理に移行します。

また、メインループ内でキーボードからの入力を受け付けます。メインループ内でコマンドを入力すると、コマンドに応じた動作を行います。

SW3 は、通信相手に設定した IP アドレスに ping を送信します。

9.5.1. LWIP の初期化

・main.c 内（抜粋、一部変更）

```
//LWIP初期化
lwip_init();

//Netif設定
IP_ADDR4(&ipaddr, 192, 168, 0, 80);
IP_ADDR4(&netmask, 255, 255, 255, 0);
IP_ADDR4(&gw, 192, 168, 0, 1);

//ネットワークインタフェースを追加
netif_add(&s_netif, &ipaddr, &netmask, &gw, NULL, &r_lwip_driver_ethernetif_init, &netif_input);

//作成したネットワークインタフェースをデフォルトで使用
netif_set_default(&s_netif);

//ネットワークインタフェース有効化
netif_set_up(&s_netif);
```

ボードの IP アドレス、ネットマスク、ゲートウェイアドレスを設定して、ネットワークインタフェースを有効化します。

9.5.1. メインループ内で呼び出す LWIP の API 関数

メイン関数では各種設定のあと、while(1)で無限ループさせていますが、その中で呼び出す必要がある API 関数が

R_LWIP_DRIVER_Input()

R_LWIP_DRIVER_EthernetLinkCheck()

となります。前者は受信処理、後者は接続状態のチェックを行う関数です。

・main.c 内（抜粋、一部変更）

```
//メインループ
while (1)
{
    Ethernet_Link_Periodic_Handle();

    R_LWIP_DRIVER_Input(&s_netif);

 （省略）
}


void Ethernet_Link_Periodic_Handle(void)
{
    //100ms毎に実行
unsigned long current_timer = sys_now();

    if ((current_timer - s_ethernet_link_timer) >= 100)
    {
        s_ethernet_link_timer = current_timer;
        R_LWIP_DRIVER_EthernetLinkCheck(&s_netif);
    }
}
```

メインループ内では、Ethernet_Link_Periodic_Handle()と R_LWIP_DRIVER_Input()を実行しており、Ethernet_Link_Periodic_Handle()の中で、R_LWIP_DRIVER_EthernetLinkCheck()を呼び出しています。

sys_now()は、r_cmt_rx（コンペアマッチタイマ）とも関連しており、1ms 毎にインクリメントされる値（＝現在の時間）を返します。Ethernet_Link_Periodic_Handle()はメインループ内で連続的に実行されますが、前回時間値を取得してから 100 カウント後、すなわち 100ms 毎に R_LWIP_DRIVER_EthernetLinkCheck()が実行されます。

9.6. TCP クライアント

9.6.1. 接続処理

c コマンドを入力したときの動作です。

・client.c

```
void tcp_client_connect(unsigned char *dst_IP, unsigned short dst_port)
{
    //TCP接続関数

    //戻り値
    //  なし

    //引数
    //  *dst_IP : 接続先IPアドレス
    //  dst_port : 接続先ポート

    ip_addr_t dst_ipaddr;

    //TCP pcb作成
    g_client_tcp_pcb = tcp_new();

    if (g_client_tcp_pcb != NULL)
    {
        IP4_ADDR(&dst_ipaddr, dst_IP[0], dst_IP[1], dst_IP[2], dst_IP[3]);

        //接続
        tcp_connect(g_client_tcp_pcb, &dst_ipaddr, dst_port, tcp_client_connected);
    }
}
```

IP4_ADDR は、IP アドレス（unsigned char dst_IP[4]）を LWIP の関数で扱う形式に変換しているだけです。

pcb は protocol control block で LWIP で使用する TCP 接続を扱う変数（構造体）です。

```
struct tcp_pcb *g_client_tcp_pcb = NULL;
```

tcp_new()で新しい pcb を作成して、TCP の接続関数 tcp_connect()の引数として渡します。tcp_connect の 2,3 番目の引数は接続先の IP アドレスとポートですが、4 番目に指定している tcp_client_connected は関数です。接続が完了した際に、4 番目の引数で指定した関数が呼ばれる動作となります。

・client.c

```
static err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    //TCPクライアント接続時コールバック関数を登録する関数

    g_client_es.pcb = tpcb;//各種コールバック関数で使用する引数

    tcp_arg(tpcb, &g_client_es);

    tcp_recv(tpcb, tcp_client_recv);

    tcp_sent(tpcb, tcp_client_sent);

    tcp_poll(tpcb, tcp_client_poll, 1);

    tcp_err(tpcb, tcp_client_err);

    g_client_es.tcp_connection = TCP_CONNECT;

    sci_write_str("-- TCP CONNECTION ESTABLISHED --¥n");

    return 0;
}
```

接続完了時には、

tcp_recv　受信時

tcp_sent　送信完了時

tcp_poll　接続中の定期処理

tcp_err　エラー時

に呼び出される関数（コールバック関数）を登録する処理を行っています。

tcp_arg は、各種コールバック関数に引数を渡すためのおまじないです。

g_client_es.tcp_connection = TCP_CONNECT;

は、接続状態管理のための代入です。（未接続でメッセージ送信した場合はエラー表示させるため）（LWIP の動作に必要なものではありません）

・client.c

```
static err_t tcp_client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    //TCPクライアント受信関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    unsigned char buf[TCP_MESSAGE_SIZE + 1];
    unsigned short len;

    if (p == NULL)
    {
        //空のTCPフレームを受信→接続を切断，ここではFINフラグ受信に対してACKを返す（後でこちらからも
tcp_server_poll() 内でFINを送信）
        es->tcp_connection = TCP_CONNECT_CLOSEING;

    return ERR_OK;
    }

    //データを受信
    len = p->len;
    if (len > TCP_MESSAGE_SIZE) len = TCP_MESSAGE_SIZE;// bufが41バイトしか確保していないのでサイズ超
過の場合は先頭の40バイトのみ表示

    memcpy(buf, p->payload, len);
    buf[TCP_MESSAGE_SIZE] = '¥0';//終端文字

    //スタックに受信サイズ通知
    tcp_recved(tpcb, p->tot_len);

    //バッファメモリの解放
    pbuf_free(p);

    sci_write_str("message received (TCP) ->¥n");
    sci_write_str((const char *)buf);
    sci_write_str("¥n");

    return ERR_OK;
}

static err_t tcp_client_sent(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
    //TCPクライアント送信完了関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    if (es->p_tx != NULL)
        {
            tcp_client_send(tpcb, es);
        }

    return ERR_OK;
}
```

データを受信した場合(tcp_client_recv())は、ユーザ側で用意したバッファにコピーした後で、LWIP で使用しているバッファメモリは解放します。また、端末にメッセージを表示させます。

データの送信完了時は、送信バッファにデータが残っている場合は、次の送信を促します(tcp_client_send())。

・client.c

```
static err_t tcp_client_poll(void *arg, struct tcp_pcb *tpcb)
{
    //TCPクライアントポーリング関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    if (es != NULL)
    {
        if (es->p_tx != NULL)
        {
            tcp_client_send(tpcb, es);
        }
        else
        {
            if (es->tcp_connection == TCP_CONNECT_CLOSEING)
            {
                //切断のFINを送信
                tcp_client_connection_close(tpcb, es);
            }
        }
    }

    return ERR_OK;
}

static void tcp_client_err(void *arg, err_t err)
{
    //TCPクライアントエラー処理関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    switch (err)
    {
        case ERR_RST:
        case ERR_CLSD:
            tcp_client_connection_close(es->pcb, es);
            break;
    }

    sci_write_str("tcp client error function called, code = ");
    sci_write_uint16(err);
    sci_write_str("¥n");
}
```

TCP 接続中、定期的に呼ばれるのがポーリング関数(tcp_client_poll())です。送信バッファに、未送信のデータが残っている場合は、送信を促します。切断要求を受け取った場合は、切断処理を行います。

エラーコールバック関数(tcp_client_err())が呼ばれた場合は、エラーコードを表示する様にしています。

・client.c

```
static void tcp_client_send(struct tcp_pcb *tpcb, struct TCP_ARG *es)
{
    //TCPクライアントの送信バッファを処理する関数

    struct pbuf *ptr = NULL;
    err_t err = ERR_OK;

    while ((err == ERR_OK) && (es->p_tx != NULL) && (es->p_tx->len <= tcp_sndbuf(tpcb)))
    {
        ptr = es->p_tx;

        err = tcp_write(tpcb, ptr->payload, ptr->len, 1);

        if (err == ERR_OK)
        {
            es->p_tx = ptr->next;

            if (es->p_tx != NULL)
            {
                pbuf_ref(es->p_tx);
            }

            pbuf_free(ptr);
        }
        else if (err == ERR_MEM)
        {
            es->p_tx = ptr;
         }
        else
        {
         //その他のエラー
        }
    }
}
```

エラーが起こっていない　未送信のデータがある　バッファメモリの空きが len 以上

次のバッファを参照

参照カウンタインクリメント

メモリの開放

tcp_client_send()は、送信完了のコールバック関数やポーリングのコールバック関数で呼ばれる関数です。
未送信のデータがある場合の処理です。

9.6.2. データ送信処理

接続後 s コマンドを入力した場合の動作です。

・client.c

```
void text_packet_send_tcp(struct tcp_pcb *tpcb, unsigned char *message, unsigned short size)
{
    //TCPテキストメッセージ送信関数

    //戻り値
    //  なし

    //引数
    //  tpcb : TCPを取り扱う構造体
    //  message : 送信メッセージ
    //  size : 送信サイズ

    if (g_client_es.tcp_connection == TCP_DISCONNECT)
    {
        //TCPの通信が開いていない

        sci_write_str("TCP NOT CONNECTED. Please open TCP session first.¥n");

        return;
    }

    err_t err;

    sci_write_str("send message.¥n");

    err = tcp_write(tpcb, message, size, TCP_WRITE_FLAG_COPY); //TCP_WRITE_FLAG_COPY(1):バッファにコピーを行う

    if (err != ERR_OK)
    {
        sci_write_str("... ERROR¥n");
    }
}
```

tcp_write()がデータを送信する API 関数です。

4 番目の引数は、TCP_WRITE_FLAG_COPY(=1)とした場合、データ（このケースでは message）を LWIP が管理しているバッファにコピーを行います。その場合は、messege はデータ送信完了まで内容を保持する必要はありません。

9.6.1. 切断処理

接続後 d コマンドを入力した場合の動作です。

・client.c

```
void tcp_client_connection_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
{
    //TCP接続切断関数

    //戻り値
    //  なし

    //引数
    //  *tpcb : TCPを取り扱う構造体
    //  *es : 引数の構造体

    tcp_arg(tpcb, NULL);
    tcp_recv(tpcb, NULL);
    tcp_sent(tpcb, NULL);
    tcp_poll(tpcb, NULL, 0);
    tcp_err(tpcb, NULL);

    tcp_close(tpcb);

    es->tcp_connection = TCP_DISCONNECT;

    sci_write_str("-- TCP CONNECTION CLOSED --¥n");
}
```

各種コールバック関数を NULL に設定して、tcp_close()を呼び出します。

9.7. TCP サーバ

9.7.1. 待ち受け開始処理

・main.c, ether_main()内

```
//メインループ
while (1)
{
 （中略）

    //TCPサーバ待ち受け設定
    if (g_server_es.tcp_connection == TCP_DISCONNECT)
    {
        ret = netif_is_link_up(&s_netif);
        if (ret)
        {
            tcp_server_listen(g_tcp_server_listen_port);
        }
    }
```

g_server_es.tcp_connection は、ユーザが管理している変数値です。サーバが立ち上がっていない場合に、待ち受け開始の処理を行います。起動後や、一度接続して切断した後で待ち受け開始処理は実行されます。

netif_is_link_up()は、ネットワークが立ち上がっているかの確認です。

tcp_server_listen()が待ち受け開始関数です。（g_tcp_server_listen_port はデフォルトでは 20000 です）

・server.c

```
void tcp_server_listen(unsigned short listen_port)
{
    //TCP待ち受け関数

    //戻り値
    //  なし

    //引数
    //  listen_port : 待ち受けポート

    err_t err;

    //TCP pcb作成
    g_server_tcp_pcb = tcp_new();

    g_server_tcp_pcb->so_options |= SOF_REUSEADDR;//一度使用したポートを再利用する

    /*
     * smc_gen¥r_config¥lwipopts.h 内で
     *
     * #define SO_REUSE 1
     *
     * の定義をしておかないと、SOF_REUSEADDR の設定は有効にならないので注意
     */

    if (g_server_tcp_pcb != NULL)
    {
        err = tcp_bind(g_server_tcp_pcb, IP_ADDR_ANY, listen_port);

        if (err !=  ERR_OK)
        {
            sci_write_str("tcp_bind error¥n");

            tcp_close(g_server_tcp_pcb);

            return;
        }

        g_server_tcp_pcb = tcp_listen(g_server_tcp_pcb);//tcp_listen() は pcb を返す
 （g_server_tcp_pcbが上書きされて更新）

        tcp_accept(g_server_tcp_pcb, tcp_server_connected);//接続を受け入れた際に、
tcp_server_connected() が呼ばれる設定

        g_server_es.tcp_connection = TCP_CONNECT_ATTEMPT;//クライアントからの接続待ち状態

        sci_write_str("-- TCP SERVER LISTEN START (port = ");
        sci_write_uint16(listen_port);
        sci_write_str(") --¥n");
    }
}
```

サーバ立ち上げの際も、tcp_new で TCP pcb 変数（構造体）を初期化します。

g_server_tcp_pcb->so_options の設定は、20000 ポートで待ち受け、接続、通信、切断。その後、再度 20000 ポートで待ち受けする場合に必要な設定です。このポートを再利用する設定は、lwipopts.h の書き換えも必要です。（9.2 節で lwipopts.h 内に指定が必要な事項に関しては記載しています）

tcp_bind()　pcb と待ち受けポート番号の関連付け
tcp_listen()　待ち受け動作
tcp_accept()　接続時に呼ばれる関数の設定

使用する API 関数は上記です。

tcp_accept()の 2 番目の引数は関数で、接続時にここで指定した関数が呼ばれます。

・server.c

```
static err_t tcp_server_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    //TCPサーバ接続時コールバック関数を登録する関数

    g_server_es.tcp_connection = TCP_CONNECT;

    g_server_es.pcb = tpcb;//各種コールバック関数で使用する引数

    tcp_arg(tpcb, &g_server_es);

    tcp_recv(tpcb, tcp_server_recv);

    tcp_sent(tpcb, tcp_server_sent);

    tcp_poll(tpcb, tcp_server_poll, 1);

    tcp_err(tpcb, tcp_server_err);

    sci_write_str("-- TCP CONNECTION ACCEPTED --¥n");

    //接続元
    unsigned char remote_ip[4];

    remote_ip[0] = (unsigned char)(tpcb->remote_ip.addr & 0xff);//remote_ip->addr はリトルエンディアンで格納される
    remote_ip[1] = (unsigned char)((tpcb->remote_ip.addr >> 8) & 0xff);
    remote_ip[2] = (unsigned char)((tpcb->remote_ip.addr >> 16) & 0xff);
    remote_ip[3] = (unsigned char)((tpcb->remote_ip.addr >> 24) & 0xff);

    sci_write_str("(connected from ");
    sci_print_IP(remote_ip);
    sci_write_str(":");
    sci_write_uint16(tpcb->remote_port);
    sci_write_str(")¥n");

    return 0;
}
```

g_server_es.tcp_connection はユーザ側で管理している変数（LWIP で使用する訳ではない）。

接続完了時には、
　tcp_recv　受信時
　tcp_sent　送信完了時
　tcp_poll　接続中の定期処理
　tcp_err　エラー時
に呼び出される関数（コールバック関数）を登録する処理を行っています。このあたりは、クライアントの接続時の関数と同じです。

・server.c

```
static err_t tcp_server_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    //TCPサーバ受信関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    unsigned char buf[TCP_MESSAGE_SIZE + 1];
    unsigned short len;

    extern unsigned char g_tcp_response_message[];//サーバ動作時に返信するメッセージ

    if (p == NULL)
    {
        //空のTCPフレームを受信→接続を切断，ここではFINフラグ受信に対してACKを返す（後でこちらからも
tcp_server_poll() 内でFINを送信）
        es->tcp_connection = TCP_CONNECT_CLOSEING;

        return ERR_OK;
    }

    //データを受信
    len = p->len;
    if (len > TCP_MESSAGE_SIZE) len = TCP_MESSAGE_SIZE;// bufが41バイトしか確保していないのでサイズ超
過の場合は先頭の40バイトのみ表示

    memcpy(buf, p->payload, len);
    buf[TCP_MESSAGE_SIZE] = '¥0';//終端文字

    //スタックに受信サイズ通知
    tcp_recved(tpcb, p->tot_len);

    //バッファメモリの解放
    pbuf_free(p);

    sci_write_str("message received (TCP) ->¥n");
    sci_write_str((const char*)buf);
    sci_write_str("¥n");

    sci_write_str("send response.¥n");

    err = tcp_write(tpcb, g_tcp_response_message, TCP_MESSAGE_SIZE, TCP_WRITE_FLAG_COPY);//
TCP_WRITE_FLAG_COPY(1):バッファにコピーを行う，レスポンス返送

    if (err != ERR_OK)
    {
        sci_write_str("... ERROR¥n");
    }

     return ERR_OK;
}
```

データの受信コールバック関数(tcp_server_recv())は、赤枠部が TCP クライアントの受信コールバック関数との相違です。サーバ側での動作の場合、データを受信した際" OK,　(RX65)　(TCP server)"というメッセージを返信します。それ以外のコールバック関数に関しては、クライアントのコールバック関数と変わりません。

サーバ動作時、データを受信した場合は、上記の tcp_server_recv()のコールバック関数での処理。切断要求を受け取った場合は、tcp_server_recv()と tcp_server_poll()のコールバック関数により 4 ウェイハンドシェイクでの切断処理が行われます。

なお切断後は、メインループ内で再度 TCP サーバの待ち受け動作が開始されます。

9.8. UDP クライアント

9.8.1. データ送信処理

Ｓコマンドを入力したときの動作です。

・client.c

```
void text_packet_send_udp(unsigned char *dst_IP, unsigned short dst_port, unsigned char *message,
unsigned short size)
{
    //UDPテキストメッセージ送信関数

    //戻り値
    //  なし

    //引数
    //  *dst_IP: 送信先IPアドレス
    //  dst_port: 送信先ポート
    //  *message : 送信メッセージ
    //  size : 送信サイズ

    struct udp_pcb *pcb = NULL;
    ip_addr_t dst_ipaddr;
    err_t err;
    struct pbuf *p = NULL;;

    //UDP pcb作成
    pcb = udp_new();

    if (pcb != NULL)
    {
        IP4_ADDR(&dst_ipaddr, dst_IP[0], dst_IP[1], dst_IP[2], dst_IP[3]);

        //接続
        err = udp_connect(pcb, &dst_ipaddr, dst_port);

        if (err != ERR_OK)
        {
            sci_write_str("udp_connect error.¥n");

            udp_remove(pcb);

            return;
        }

    p = pbuf_alloc(PBUF_TRANSPORT, size, PBUF_RAM);

    if (p != NULL)
    {
        pbuf_take(p, message, size);

        err = udp_send(pcb, p);

        if (err != ERR_OK)
        {
            sci_write_str("udp_send error.¥n");
        }

        pbuf_free(p);
    }

    udp_disconnect(pcb);
    udp_remove(pcb);
}
```

UDP でのデータ送信の場合、
pcb = udp_new()
で、通信に使用する pcb（通信に使用する構造体変数）を、この段階で作成します。
（TCP 通信の場合は、接続開始時に pcb を作成して、通信時は作成済みの pcb を使ってデータを送信する流れでした。UDP の場合は通信前に接続を確立しておく必要はないので、送信パケットを投げる度ごとに pcb を作る形としています。）

udp_connect()　接続
udp_send()　送信
udp_disconnect()　切断
udp_remove()　pcb の削除

上記の流れで、API 関数を呼び出しています。UDP の場合は、送信が終わるとポートを開放するという意味でも、udp_disconnect()して、pcb 自体も削除してしまいます。

udp_connect()で、接続先の IPアドレスとポート番号を指定して接続、usb_send()では pcb を指定して送信（この時点では pcb には、接続先の IPアドレスとポート番号が紐付け済み）という形です。

tcp のデータ送信関数(tcp_write())とちょっと違う点があり、

pbuf_alloc()　送信用のバッファメモリを確保
pbuf_take()　送信用のバッファにデータを格納
pbuf_free()　送信用のバッファメモリの開放

の API 関数を使っています。

tcp_write()は、引数に送信データと送信バイト数を指定する関数です。それに対し、udp_send()は、引数に pbuf 構造体（LWIP が使用するデータ送受信用のバッファメモリ）を指定する関数になっています。そのため、ユーザプログラムで pbuf 構造体に送信データを格納してから udp_send()を呼び出すようにしています。

※tcp_write()と udp_send()のインタフェースの違いに関しては何とも言えませんが、tcp_write()でも内部的には pbuf を確保して pbuf から送信処理を行っていると思われます。
udp は udp_send()を呼び出すと即時送信が行われ、udp_send()後に pbuf_free()を実行しても問題ありません。それに対し、tcp_write()の方は即時送信ではなく LWIP のプロトコルスタックの都合の良いタイミングでの送信となります。pbuf_free()できるタイミングはプロトコルスタックでなければ判らない形です。そのため、API 関数としてはデータを受け取っておき、裏で pbuf にコピー、（遅延がある）、送信、pbuf 解放(pbuf_free())を行うインタフェースとしているのではないかと思います。
なお TCP の場合でも、tcp_write()後に、tcp_output()を実行すると、tcp_write()で送信キューに入れられたデータが即時送信されるという動作となります。（本プロジェクトでは、tcp_output()は使っていません。）

9.9. UDP サーバ

9.9.1. 待ち受け開始処理

・main.c, ether_main()内

```
//メインループ
while (1)
{
 （中略）

    //UDPサーバ待ち受け設定
    if (g_server_udp_pcb == NULL)
    {
        ret = netif_is_link_up(&s_netif);
        if (ret)
        {
            udp_server_listen(g_udp_server_listen_port);
        }
    }
```

g_server_udp_pcb は、UDP サーバ向けの pcb です。これが作成されていない場合は、待ち受け開始の処理を行います。起動後、待ち受け開始処理が実行されると ==NULL ではなくなります。

udp_server_listen()が待ち受け開始関数です。（g_udp_server_listen_port はデフォルトでは 30000 です）

・server.c

```
void udp_server_listen(unsigned short listen_port)
{
    //UDP待ち受け関数

    //戻り値
    //  なし

    //引数
    //  listen_port : 待ち受けポート

    err_t err;

    //UDP pcb作成
    g_server_udp_pcb = udp_new();

    if (g_server_udp_pcb != NULL)
    {
        err = udp_bind(g_server_udp_pcb, IP_ADDR_ANY, listen_port);

        if (err !=  ERR_OK)
        {
            sci_write_str("udp_bind error¥n");
            udp_remove(g_server_udp_pcb);

            return;
        }

        //受信コールバック関数の登録
        udp_recv(g_server_udp_pcb, udp_server_recv, NULL);

        sci_write_str("-- UDP SERVER LISTEN START (port = ");
        sci_write_uint16(listen_port);
        sci_write_str(") --¥n");
    }
}
```

udp_new()　pcb の作成。
tcp_bind()　ポート番号と pcb の紐付け
udp_recv()　受信コールバック関数の登録
を行っています。

udp_recv()の引数は、「pcb」、「登録する受信コールバック関数」、3 番目の引数が「コールバック関数に渡したいデータのポインタ」です。特に、受信コールバック関数に渡したいデータはないので、第 3 引数は NULL で本関数を呼び出しています。

・server.c

```
static void udp_server_recv(void *arg, struct udp_pcb *tpcb, struct pbuf *p, const ip_addr_t
*addr, u16_t port)
{
    //UDPサーバ受信関数

    unsigned char buf[UDP_MESSAGE_SIZE + 1];
    unsigned short len;

    //データを受信
    len = p->len;
    if (len > UDP_MESSAGE_SIZE) len = UDP_MESSAGE_SIZE;// bufが17バイトしか確保していないのでサイズ超
過の場合は先頭の16バイトのみ表示

    memcpy(buf, p->payload, len);

    //受信バッファメモリの開放
    pbuf_free(p);

    buf[UDP_MESSAGE_SIZE] = '¥0';//終端文字

    sci_write_str("message received (UDP) ->¥n");
    sci_write_str((const char*)buf);
    sci_write_str(" (from ");
    sci_print_IP((unsigned char *)addr);
    sci_write_str(":");
    sci_write_uint16(port);
    sci_write_str(")¥n");
}
```

データを受信した場合は、自前で確保したバッファ(buf)へのデータコピー、プロトコルスタック側で使用した受信バッファの開放(pbuf_free())と、受信メッセージの表示を行っています。

9.10.ping

本プロジェクトでも、SW3 を押すと通信相手に ping を打ちます。また、外部から本ボードに対して ping を打つと、本ボードは応答します。

ping は、プロトコルとしては ICMP(Internet Control Massage Protocol)となりますので、スマート・コンフィグレータの r_lwip_rx の設定で IWIP_ICMP の設定は Enabled（チェック[デフォルト]）に設定しておく必要があります。

9.10.1.ping リクエスト

・ping.c

```
void ping_request(unsigned char *dst_IP)
{
    //ping送信関数

    //戻り値
    //  なし

    //引数
    //  *dst_addr : ping送信先IPアドレス

    struct pbuf *p = NULL;
    struct icmp_echo_hdr *iecho = NULL;
    ip_addr_t ipaddr;

    int i;
    unsigned char *d;

    //PINGのIDは毎回変更
    ping_id++;

    //前回使用した g_ping_pcb が残っている場合一旦開放
    if (g_ping_pcb != NULL)
    {
        raw_remove(g_ping_pcb);
    }

    //pcbを作成
    g_ping_pcb = raw_new(IP_PROTO_ICMP);
    if (g_ping_pcb == NULL) return;

    //受信コールバック関数を設定
    raw_recv(g_ping_pcb, ping_recv, NULL);

    raw_bind(g_ping_pcb, IP_ADDR_ANY);

    //パケットバッファの確保
    p = pbuf_alloc(PBUF_LINK, sizeof(struct icmp_echo_hdr) + PING_DATA_SIZE, PBUF_RAM);
    if (p == NULL)
    {
        raw_remove(g_ping_pcb);
        return;
    }

    //ICMPヘッダ
    iecho = (struct icmp_echo_hdr *)p->payload;
    ICMPH_TYPE_SET(iecho, ICMP_ECHO);
    ICMPH_CODE_SET(iecho, 0);
    iecho->chksum = 0;
    iecho->id = ping_id;
    iecho->seqno = htons(1);
```

・ping.c, ping_request()（続き）

```
    /*
 * リトルエンディアンの環境では、IDの初期値は0x3512(=0x1234 +1 を上位下位反転）となり
 * 0x3612->0x3712->0x3812の様に変化する動作となる（0x1234をインクリメントして上位と下位を逆にした値がIDとなる）     */

    //pingデータ（32バイト）:abcdefghijklmnopqrstuvwabcdefghi（Windowsのpingが送るデータに合わせているだけで、この並びで送らなければならないという訳ではない）
    d = p->payload;
    d += sizeof(struct icmp_echo_hdr);
    for (i=0; i<23; i++) *d++  = (unsigned char)('a' + i);
    for (i=0; i<9; i++) *d++ = (unsigned char)('a' + i);

    // チェックサムの計算
    iecho->chksum = inet_chksum(p->payload, p->len);

    //ping送信
    IP4_ADDR(&ipaddr, dst_IP[0], dst_IP[1], dst_IP[2], dst_IP[3]);
    raw_sendto(g_ping_pcb, p, &ipaddr);
    sci_write_str("ping request send to ");
    sci_print_IP(dst_IP);
    sci_write_str("¥n");

    //バッファは解放
    pbuf_free(p);
}
```

ping リクエストに関しては、

g_ping_pcb = raw_new(IP_PROTO_ICMP)　ping 用の pcb を作成

raw_recv(g_ping_pcb, ping_recv, NULL)　受信コールバック関数を登録

raw_bind(g_ping_pcb, IP_ADDR_ANY)　pcb と IP アドレスの紐付け

raw_sendto(g_ping_pcb, p, &ipaddr)　ping リクエストパケットの送信

となりますが、送信するデータ（バイト列）を作って、チェックサムを計算して埋め込んでなどと、前のプロジェクトと似ているところがあります。

※送信データをセットしているところ

iecho->id = ping_id;

ですが、ここで ping_id=0xABCD の場合、送信されるデータは 0xCDAB となります。

RX はバイエンディアン（データをリトルエンディアン、ビッグエンディアンのどちらかを選択して扱える CPU）ですが、本プロジェクトでは、リトルエンディアンとしています。

Ethernet の世界では、ビッグエンディアンで通信する決まりです。リトルエンディアンの CPU の場合、思っていたのと上位と下位バイトが逆になる（事もある）と認識してください。

（RX65N_TCP までのプロジェクトでは、ID などのデータを上位・下位の順番でバイトストリームに変していましたので、0xABCD が 0xABCD として送信される形でした。LWIP の環境では、メモリ格納順（リトルエンディアンでは、下位バイトから）送信される仕様の様です。）

・ping.c

```
static u8_t ping_recv(void *arg, struct raw_pcb *pcb, struct pbuf *p, const ip_addr_t *addr)
{
    //ping応答受信時のコールバック関数

    struct icmp_echo_hdr *iecho;

    // ICMPヘッダーを取得
    if (pbuf_header(p, -PBUF_IP_HLEN) == 0)
    {
        iecho = (struct icmp_echo_hdr *)p->payload;
        if (iecho->id == ping_id) //PING IDが一致した場合は本ボードのpingに対する応答であるとみなす
        {
            pbuf_free(p);

            //pingが返ってきた場合はLED点灯
            LED = 1;

            sci_write_str("-> ping reply received from ");
            sci_print_IP((unsigned char *)addr);
            sci_write_str("¥n");

            //pingリプライが返ってきた場合g_ping_pcbを破棄する
            raw_remove(g_ping_pcb);

            return 1;
        }
    }

    pbuf_free(p);

    //本ボードのpingに対する応答でない場合もg_ping_pcbは破棄する
    raw_remove(g_ping_pcb);

    return 0;
}
```

受信コールバック関数は上記の様にしています。受信したデータ（ping リプライであると期待）の、ping ID の値を見て、本ボードが送った ID と一致すれば、

・ボード上の LED の点灯

・画面にメッセージ（-> ping reply received from xxx.xxx.xxx.xxx）を表示

します。

その後、g_ping_pcb を破棄します（＝コールバック関数の登録も消します）。

g_ping_pcb を破棄するのが、ping 送信（＝コールバック関数登録）の後の 1 回目の受信時なので、たまたまそのタイミングで別な ICMP メッセージ（ping リクエスト）を受信した場合は、ping 応答は行いません。（相手から見ると ping が通りません。）

9.10.2.ping リプライ

ping リプライに関しては、ユーザプログラムでは行っていません。LWIP プロトコルスタック内で処理されます。

但し、ping 送信後コールバック関数(ping_recv())が登録されているタイミングで ping リクエストを受信した場合は、ping_recv()関数が ping リクエストのパケットを受信します。

ping_recv()関数では、ping リプライの送信を行う処理を記載していないので、その場合は本ボードが ping 応答を返さない事となります。

時系列		ICMP データ受信先	動作
1	ping リクエスト受信	LWIP プロトコルスタック	ping リプライ送信
2	ping 送出	-	ping_revc()登録
3	ping リプライ受信	ping_revc()コールバック関数	メッセージ表示、LED 点灯 ping_recv()破棄
4	ping リクエスト受信	LWIP プロトコルスタック	ping リプライ送信

起動後は、LWIP プロトコルスタックが ping のリクエストを処理します。SW3 を押して ping を送信した場合、ICMP パケットの受信先は ping_recv()となり、このタイミングで ping リプライを受信すれば、ping 応答を受信したものとして処理されます。

ここで、ping 送出先のホストが応答しない場合を考えます。

時系列		ICMP データ受信先	動作
1	ping リクエスト受信	LWIP プロトコルスタック	ping リプライ送信
2	ping 送出	-	ping_recv()登録
~~3~~	~~ping リプライ受信~~	~~ping_revc()コールバック関数~~	~~メッセージ表示、LED 点灯~~ ~~ping_recv()破棄~~
3	ping リクエスト受信	ping_recv()コールバック関数	ping_recv()破棄
4	ping リクエスト受信	LWIP プロトコルスタック	ping リプライ送信

その場合は、時系列 3 の動作がないので、その後の外部からの ping リクエストに対して、ping_recv()が受信します。ping_recv()には、ping リプライ送信のルーチンを実装していないために ping は返りません。その後、ping リクエストが来た場合は、LWIP のプロトコルスタックが受信するので、ping は返ります。

PC から ping を打った場合の例（応答しないホストに ping を打った後の挙動）

```
C:¥Users¥win64-7>ping 192.168.0.80

192.168.0.80 に ping を送信しています 32 バイトのデータ:
要求がタイムアウトしました。
192.168.0.80 からの応答: バイト数 =32 時間 <1ms TTL=255
192.168.0.80 からの応答: バイト数 =32 時間 <1ms TTL=255
192.168.0.80 からの応答: バイト数 =32 時間 <1ms TTL=255

192.168.0.80 の ping 統計:
    パケット数: 送信 = 4、受信 = 3、損失 = 1 (25% の損失)、
ラウンド トリップの概算時間 (ミリ秒):
    最小 = 0ms、最大 = 0ms、平均 = 0ms
```

上記の様に、1 回目の ping には応答せず、2 回目以降は応答します。

※ping_recv()内に、ping リプライの機能を実装するという手もあります。本プロジェクトでは、ping リプライは LWIP のプロトコルスタックに任せるという手法としています。

9.11.ユーザ作成関数

text_packet_send_tcp

概要:TCP データの送信
宣言:void text_packet_send_tcp(struct tcp_pcb *tpcb, unsigned char *message, unsigned short size)
説明:
・TCP でのデータ送信
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　unsigned short dst_port: 送信先ポート
　unsigned char *message: 送信データ
　unsigned short size: 送信バイト数
戻り値:なし
補足:通信で使用する tcp_pcb 構造体は、作成済みで接続済みである必要があります

tcp_client_connect

概要:TCP 接続
宣言:void tcp_client_connect(unsigned char *dst_IP, unsigned short dst_port)
説明:
・TCP の接続の確立
を行います
引数:
　unsigned char *dst_IP 接続先 IP アドレス
　unsigned short dst_port: 接続先ポート
戻り値:なし

tcp_client_connection_close

概要:TCP 切断
宣言:void tcp_client_connection_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明:
・TCP の接続の切断
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　struct TCP_ARG *es: TCP で使用する引数の構造体

戻り値:なし

tcp_client_force_connection_close

概要:TCP 強制切断
宣言:void tcp_client_connection_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明:
・TCP の接続を強制的に切断
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　u struct TCP_ARG *es: TCP で使用する引数の構造体
戻り値:なし
補足:RESET フラグを送信して、通信相手と強制的に切断します

text_packet_send_udp

概要:UDP データの送信
宣言:void text_packet_send_udp(unsigned char *dst_IP, unsigned short dst_port, unsigned char *message, unsigned short size)
説明:
・UDP でのデータ送信
を行います
引数:
　unsigned char *dst_IP 接続先 IP アドレス
　unsigned short dst_port: 送信先ポート
　unsigned char *message: 送信データ
　unsigned short size: 送信バイト数
戻り値:なし

tcp_server_listen

概要:TCP サーバの起動
宣言:void tcp_server_listen(unsigned short listen_port)
説明:
・TCP での接続の待ち受け開始
を行います
引数:
　unsigned short listen_port: サーバ待ち受けポート
戻り値:なし

tcp_server_listen_close

概要:TCP サーバの切断
宣言:tcp_server_listen_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明:
・TCP サーバ側からの切断
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　u struct TCP_ARG *es: TCP で使用する引数の構造体
戻り値:なし
補足:本プロジェクトでは未使用(クライアント側からの接続ー切断を想定した作り)

tcp_server_force_listen_close

概要:TCP サーバの強制切断
宣言:tcp_server_listen_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明:
・TCP サーバ側からの強制切断
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　u struct TCP_ARG *es: TCP で使用する引数の構造体
戻り値:なし
補足:RESET フラグを送信して、通信相手と強制的に切断します
　　本プロジェクトでは未使用(クライアント側からの接続ー切断を想定した作り)

udp_server_listen

概要:UDP サーバの起動
宣言:void udp_server_listen(unsigned short listen_port)
説明:
・UDP でのデータ受信の待ち受け開始
を行います
引数:
　unsigned short listen_port: サーバ待ち受けポート
戻り値:なし

udp_server_listen_close

概要:UDP サーバの終了
宣言:void udp_server_listen_close(struct udp_pcb *tpcb)
説明:
・UDP サーバの終了
を行います
引数:
　struct ucp_pcb *tpcb: ucp_pcb 構造体
戻り値:なし
補足:本プロジェクトでは X コマンドで本関数を呼び出して UDP サーバを落とせるようになっています
　　（UDP サーバを再度立ち上げる場合は、Z コマンドです）

ping_request

概要:ping 送信
宣言:void ping_request(unsigned char *dst_IP)
説明:
・ping の送信
を行います
引数:
　unsigned char *dst_IP: 送信先 IP アドレス
戻り値:なし

9.12.MAC アドレスの設定に関して

RX65N_TCP までのプロジェクトでは、MAC アドレスは、ether_opetration.c 内で定義した値を使用していました。

LWIP のプロトコルスタックを使用する場合、r_lwip_rx_driver コンポーネントの GUI 画面でボードの MAC アドレスを設定する形となります。

本プロジェクトでは、スマート・コンフィグレータの設定で、MAC アドレスを 00-0D-76-00-40-01 に設定しています。

なお、DIP-SW(3)を ON にして起動した場合は、MAC アドレスが 00-0D-76-00-40-02 に設定します。

・ether_operation.c

```
//本ボードのMACアドレス
unsigned char g_src_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x01};//00-0D-76 HokuotoDenshi vender code,
本値は設定されない（スマート・コンフィグレータでの r_lwip_driver_rx で設定した値が有効）
unsigned char g_src_alt_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x02};

//本ボードのIPアドレス（,で区切る事に注意）※接続するネットワーク応じて変更
unsigned char g_src_IP[4] = {192,168,0,80};
unsigned char g_src_alt_IP[4] = {192,168,0,81};

//通信相手のIPアドレス（,で区切る事に注意）※本プログラムでは、ここで設定した相手にTCP/UDPでテキストメッセ
ージやり取りする
unsigned char g_dst_IP[4] = {192,168,0,81};
unsigned char g_dst_alt_IP[4] = {192,168,0,80};

/*
 * 起動時ボードのDIP-SWでIPアドレスとMACアドレスの設定が変わります
 *
 * SW2-3(UPSEL)が OFF側の場合は
 * スマート・コンフィグレータで設定したMACアドレス, g_src_IP, g_dst_IP が使われます
 *
 * SW2-3(UPSEL)が ON側の場合は
 * g_src_alt_MAC, g_src_alt_IP, g_dst_alt_IP が使われます
 *
 */

//ネットマスク
unsigned char g_netmask[4] = {255,255,255,0};

//ゲートウェイIPアドレス
unsigned char g_gateway_IP[4] = {192,168,0,1};

//TCPクライアント時：宛先ポート番号（サーバ側はこの番号で待ち受けている事を想定）
unsigned short g_tcp_client_dst_port = 20000;

//TCPサーバ時：相手から見た宛先ポート番号（本ボード，この番号で待ち受け）
unsigned short g_tcp_server_listen_port = 20000;

//UDPクライアント時：宛先ポート番号（サーバ側はこの番号で待ち受けている事を想定）
unsigned short g_udp_client_dst_port = 30000;

//UDPサーバ時：相手から見た宛先ポート番号（本ボード，この番号で待ち受け）
unsigned short g_udp_server_listen_port = 30000;

//使用するEtherのch(Ether0)
uint32_t g_ether_channel = ETHER_CHANNEL_0;
```

MAC アドレス設定を GUI で設定した値から変更するにあたり、多少イレギュラーな事を行っています。

(1)LWIP が管理している変数の書き換え

・main.c

```
//MACアドレスの設定（スマート・コンフィグレータでの r_lwip_driver_rx での設定を上書き
for (i=0; i<6; i++) s_netif.hwaddr[i] = g_src_MAC[i];
```

(2)ETHER0 のレジスタ値の書き換え

・main.c

```
ETHERC0.MAHR = mac_address_mahr;
ETHERC0.MALR.BIT.MA = mac_address_malr;
```

上記(1)(2)の値を変更する事により、GUI で設定した MAC アドレスの値を上書きする事は可能です。

MAC アドレスを変更するにあたり、正規の手順は

(1)r_lwip_driver_rx の GUI 値を変更

(2)コード生成ボタンを押して再度コード生成する

(3)ビルド

です。

本プロジェクトでは、DIP-SW で IP, MAC アドレスを変えたかったので、多少イレギュラーな方法で MAC アドレスを変更しています。GUI の設定を変更せず MAC アドレス値を任意値に設定したい場合は、

・ether_operation.c

```
//本ボードのMACアドレス
unsigned char g_src_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x01};
```

→g_src_MAC に設定したい MAC アドレスを記載

・main.c

```
//MACアドレス強制設定
static int mac_address_replace = FALSE;
```

↓

```
//MACアドレス強制設定
static int mac_address_replace = TRUE;
```

→mac_address_replace の値を TRUE に変更。

という方法もありますが、基本的には GUI 上で変更する事を推奨致します。

10.RX65N_UART_ETHER_UDP プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_UART_ETHER_UDP
SOURCE¥176PIN¥RX65N_ UART_ETHER_UDP

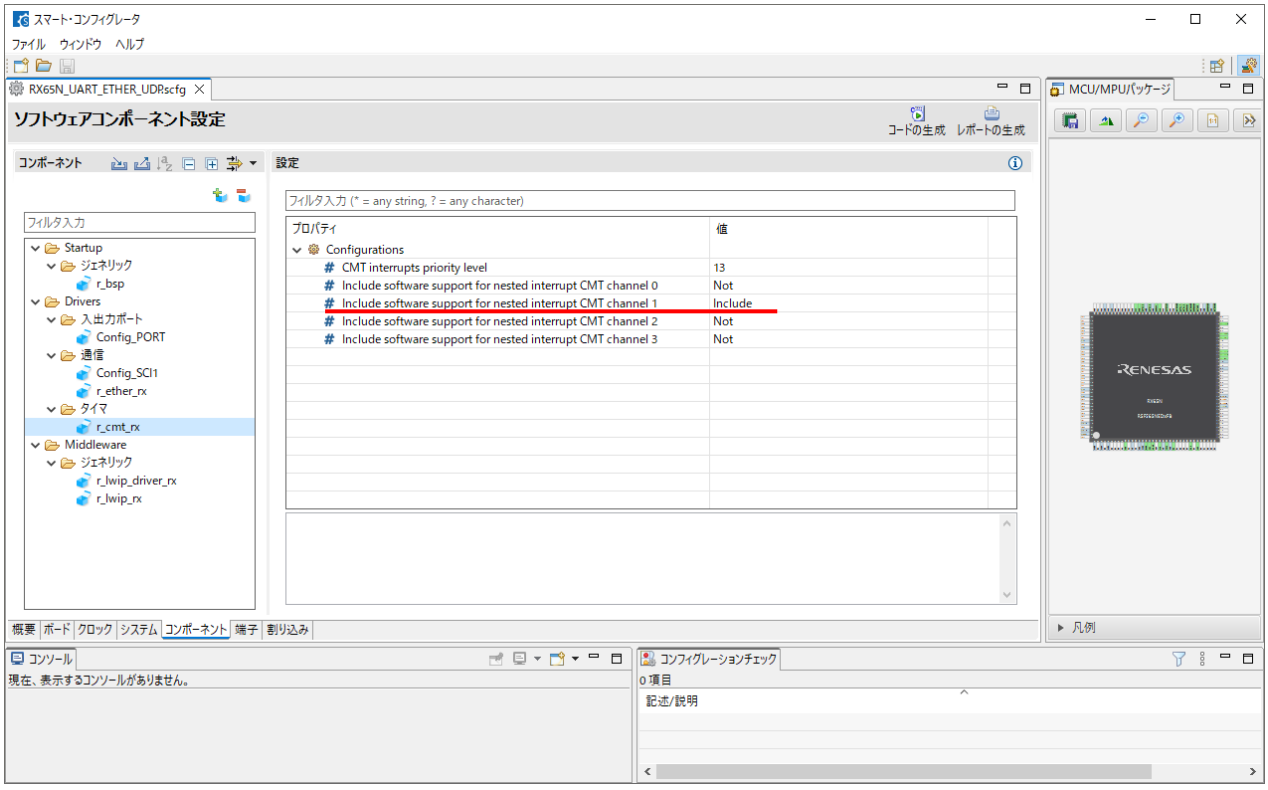
（使用しているマイコンボードのピン数に応じたフォルダ）を PC のストレージにコピーしてください。

本プロジェクトは、
・マイコンボードが UART-ETHER ブリッジ(UDP)として動作する
事を行います。

・スマート・コンフィグレータで追加するコンポーネント

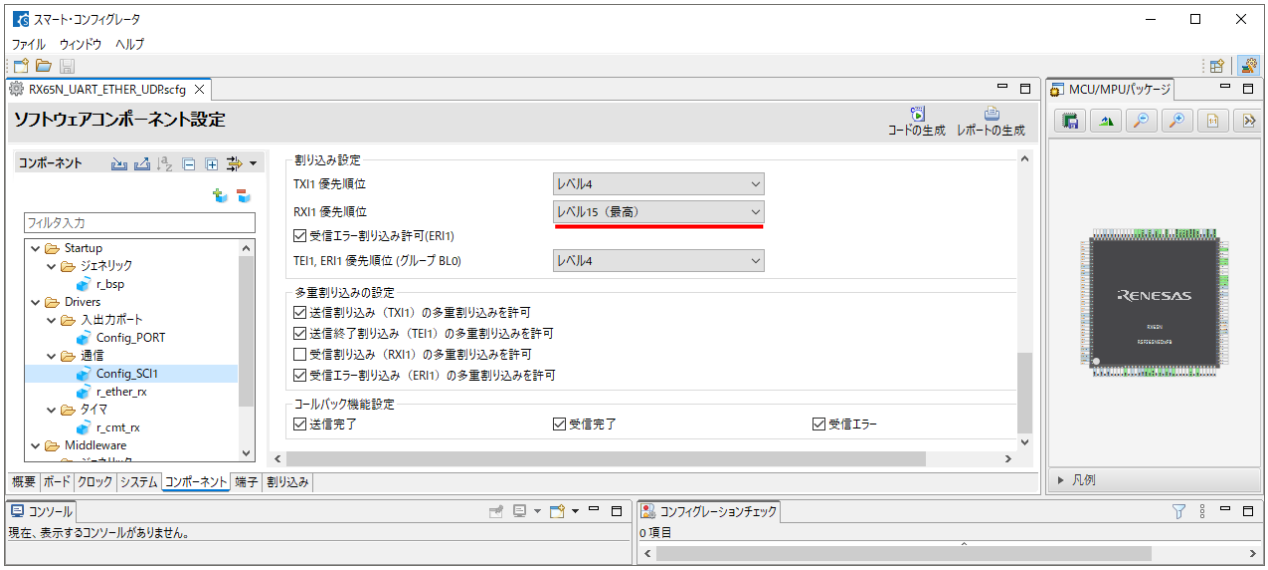
コンポーネント	備考
Config_PORT	SW と LED のポート設定
Config_SCI1	SCI(UART)通信
r_ether_rx	Ethernet
r_cmt_rx	コンペアマッチタイマ
r_lwip_driver_rx	lwIP ドライバ
r_lwip_rx	lwIP TCP/IP

・r_cmt_rx



本プロジェクトでは、LWIP プロトコルスタックが使用する CMT0 に加え、ユーザ側で CMT1 を使っているので、CMT channel1 を"Include"に変更してください。

・Config_SCI1



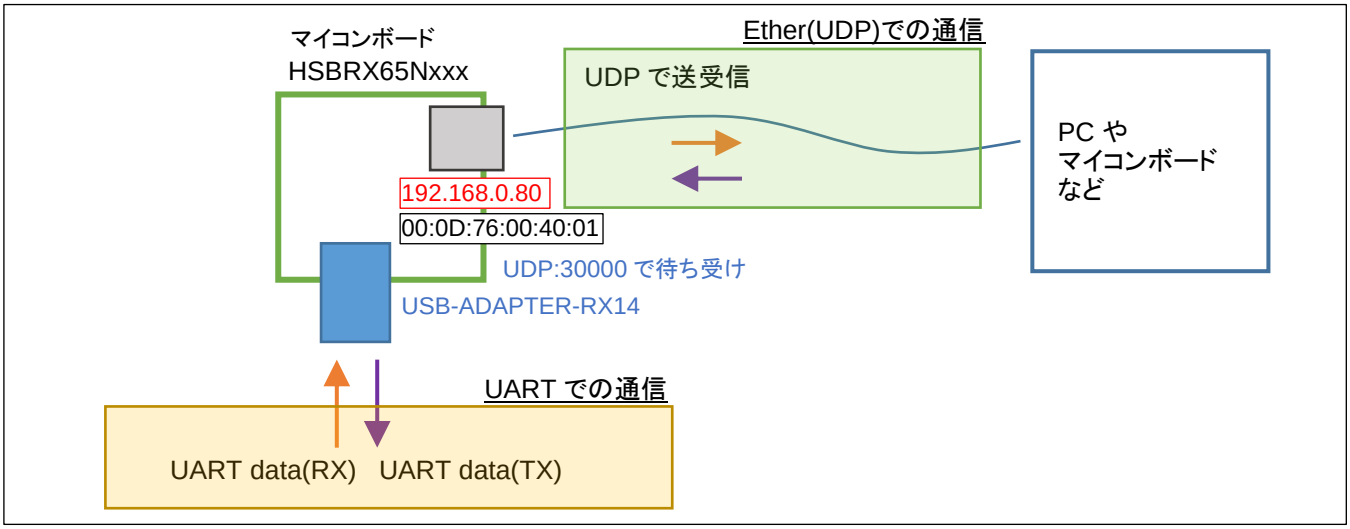
本プロジェクトでは、SCI の受信割り込みの優先度を高く設定しています。
→従来のプロジェクトでは SCI の受信は、キーボードからの読み取りに使用していたので、優先度 4（使用している割り込みの中で最低）に設定していましたが、本プロジェクトでファイル転送を行った場合、連続して SCI の受信が発生するので、高優先度に設定しています。
→SCI の速度、1152,00bps の場合は、1 バイトのデータ転送で 1/115200×10=約 87us となります。（×10 は、スタート・ストップビット＋データ 8bit です）。87us で次々と SCI の受信データを処理しなければならないので、（87us 以内にデータをリードしない場合は、オーバランエラーとなる）優先度を上げています。
→SCI のデータ受信に、DMAC や DTC を使用する。FIFO バッファ付きの SCI を使用するなど、SCI の受信処理に掛かるリソースを低減させる方策はあります。ここでは、単純に割り込み優先度を上げる事としています。

・ユーザ作成のプログラムコード

フォルダ（カテゴリ）	ファイル名	備考
client	client.c	クライアント動作のコード
	client.h	上記ヘッダ
common	ether_common.c	Ether のプログラムで共通に使用する関数
	ether_common.c	上記ヘッダ
	ether_operation.c	IP アドレスなどの定義
	ether_operation.h	上記ヘッダ
main	main.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ
server	server.c	サーバト動作のコード
	server.h	上記ヘッダ

※本プロジェクトでは、ping の送信機能はありません（外部からの ping には応答します）

・本プロジェクトの動作



本プロジェクトのプログラム（mot ファイル）をマイコンボードに書き込むと、マイコンボードは UART(SCI)と Ether(UDP)の変換機器として動作します。

USB-ADAPTER-RX14 の受信端子（11 番ピン）に来た UART のデータは、UDP パケットに変換されて送信されます。また、UDP で本ボードが受信したデータは、USB-ADAPTER-RX14 の送信端子（5 番ピン）から UART で送信されます。

※USB-ADAPTER-RX14 の接続先は、マイコンから見ると SCI1 です

※UDP の送信先は、30000 番ポート、UDP の受信ポートも 30000 番のポートです

HSBRX65N のマイコンボードを 2 台用意して同じプログラムを書き込むと、UART-Ether-UART のブリッジとして動作します。

起動後、以下の様なメッセージが表示されます。

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether UART Ether UDP sample program.
-setting address-
---
this board IP address                -> 192.168.0.80 (MAC address -> 00-0D-76-00-40-01 )
netmask                                 -> 255.255.255.0
gateway IP address                   -> 192.168.0.1
communication target IP address -> 192.168.0.81
---
-- UDP SERVER LISTEN START (port = 30000) --
```

※デバッグ表示モード有効時に、上記メッセージが表示されます

・ether_common.h

```
#define SCI_DEBUG//UART(SCI)によるデバッグ表示を行う
```

ether_common.h 内の上記行をコメントアウトすれば、起動時にメッセージは表示されなくなります。

→純粋な UART-Ether 変換機器として動作します

2 台のマイコンボードで UART-Ether-UART ブリッジを構成する場合は、片方のマイコンボードの DIPSW(3)を ON に設定してください。その場合は、IP アドレス=192.168.0.81, MAC アドレス=00-0D-76-00-40-02 に設定されます。）

起動時に、接続先の IP アドレスなどを変えたい場合は、ボード上の SW3 を押したままで、電源投入またはリセットボタンを押してください。（SW3 は画面にメッセージが表示されるまで押したままとしてください。）

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether UART Ether UDP sample program.

-- IP address settings change --

COMMAND:
   1 : this board
   2 : netmask
   3 : gateway
   4 : communication target
   5 : UDP servrt port (default = 30000)
   6 : UDP client port (default = 30000)
   p : print setting
   e : exit setting
```

その場合は、キーボードからアドレスやポート番号の設定ができる様になります。

デフォルトでは、

通信相手の IP アドレス　192.168.0.81

本ボードの IP アドレス　192.168.0.80

本ボードの MAC アドレス　00-0D-76-00-40-01

UDP サーバの待ち受けポート　30000

UDP クライアントのメッセージの送信先ポート　30000

です。（起動時に DIP-SW の 3 が ON になっていると、送信先と本ボードの IP アドレスが逆に設定されます。）

10.1.メイン関数での処理

main.c 内の ether_main()では、

・変数の初期化
・PHY チップのリセット
・SCI 初期化
・起動時のメッセージ表示
・DIP-SW のチェック
→DIP-SW の 3 番が ON の場合は、IP アドレス=192.168.0.81, MAC アドレス=00-0D-76-00-40-02 に設定
・SW3 のチェック（押されている時は IP アドレス変更）
・LWIP ドライバ動作開始
・PHY チップのリセット解除
・LWIP 初期化
・ネットワークインタフェース設定
（・MAC アドレスの変更[変更する場合のみ]）
・Ether のリンク確認
・ネットワークインタフェース有効化
・CMT1 開始
・メインループ

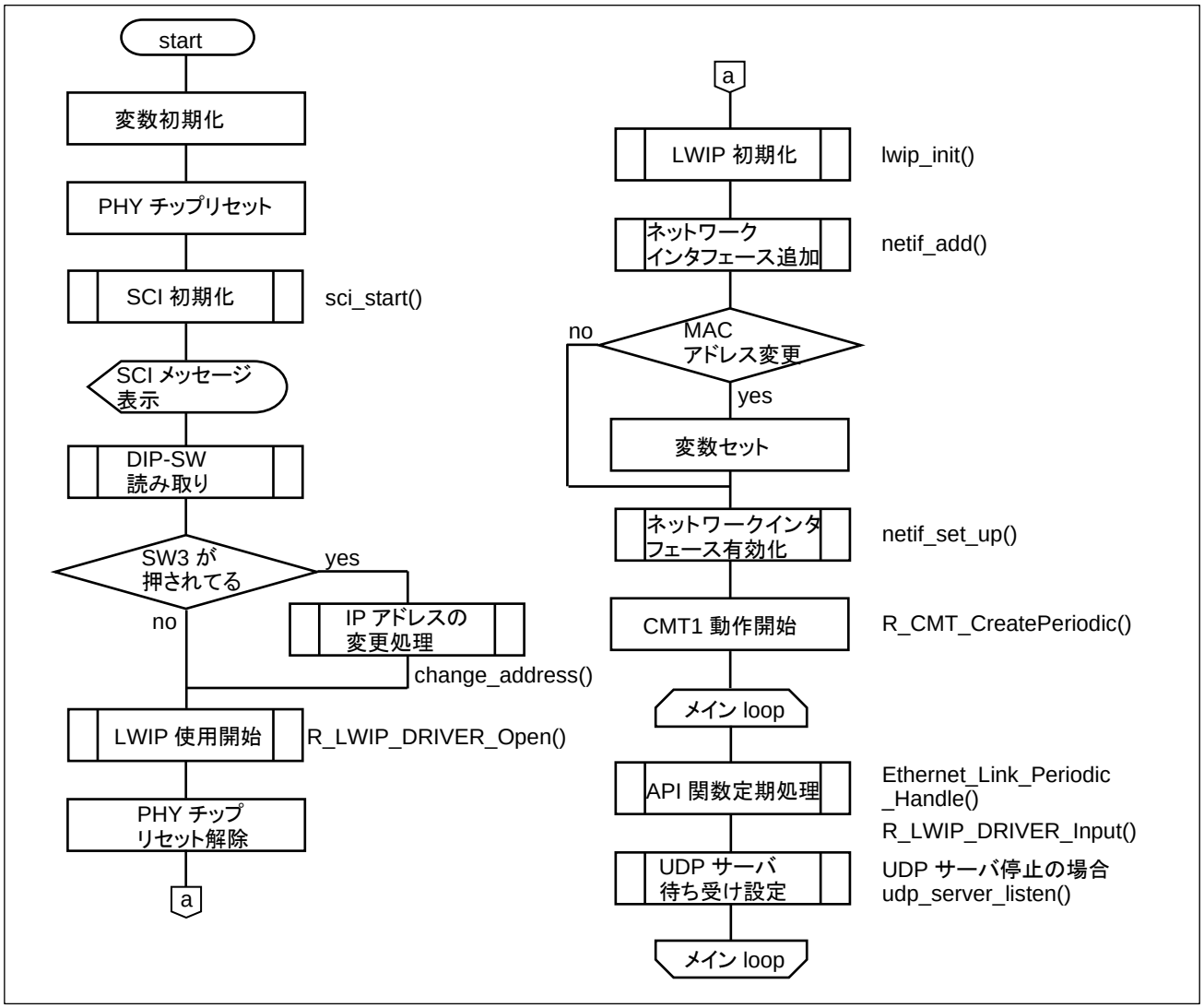
の処理を行っています。

メインループでは、

・API 関数の定期処理
・UDP サーバ待ち受け処理

となっています。

・ether_main()（main.c 内）



10.2.UDP での送信処理

本プロジェクトでは、CMT1 の定期処理が追加されており、100ms に 1 回以下の処理が実行されます。

・main.c

```
void Uart_Periodic_Handle(void *pdata)
{
    //UARTの定期処理(100ms毎)

    if (g_client_udp_pcb == NULL)
    {
        udp_client_init(g_dst_IP, g_udp_client_dst_port);

        if (g_client_udp_pcb == NULL) return;//UDPpcb作成失敗
    }

    err_t err;
    unsigned short size;
    char *buf = NULL;
    struct pbuf *p = NULL;

    //現時点でSCIの受信バッファに溜まっているサイズを取得
    size = sci_read_data_size();

    if (size == 0) return;//データなし

    //1472バイト以上の場合は、今回は1472バイト送信する（分割送信とする）
    if (size > UDP_MAX_PACET_SIZE)
    {
        size = UDP_MAX_PACET_SIZE;
    }

    buf = malloc(size);

    if (buf != NULL)
    {
        sci_read_str(buf, size);//bufに受信内容をコピー

        //UDP送信バッファを確保
        p = pbuf_alloc(PBUF_TRANSPORT, size, PBUF_RAM);

        if (p != NULL)
        {
            //送信データセット
            pbuf_take(p, buf, size);

            //UDPで送信
            err = udp_send(g_client_udp_pcb, p);

            if (err == ERR_OK)
            {
                debug_sci_write_str("¥n-- ");
                debug_sci_write_uint16(size);
                debug_sci_write_str(" bytes send --¥n");
            }
            else
            {
                debug_sci_write_str("udp_send error.¥n");
            }

            //バッファメモリ開放
            pbuf_free(p);
        }

        free(buf);
    }
}
```

定期的（100ms に 1 回）、SCI の受信バッファを確認してデータがあれば、UDP での送信を行います。

SCI バッファに、1472（=1514-14（Ethernet ヘッダ）-20（IP ヘッダ）-8（UDP ヘッダ））バイト以上のデータがある場合は、今回は 1472 バイト送信し、残りは次回の送信に回されます。

SCI の受信バッファはリングバッファになっているので、例えば"UDP message"というデータが

アドレス	0	1	2	3	4	5	6		....		7FB	7FC	7FD	7FE	7FF
データ	'e'	's'	's'	'a'	'g'	'e'					'U'	'D'	'P'	' '	'm'

というように格納されている事が考えられます。
（アドレスは、SCI の受信バッファの先頭アドレスからのオフセット。SCI の受信バッファはデフォルトで、2kB(0x800)。）
この場合は、pbuf_take()で UDP の送信バッファにコピーできないので、一旦 buf を malloc で確保（buf はヒープ領域に確保される）して、buf に送信データを格納。buf から pbuf_take()で UDP の送信バッファにコピーとしています。
→SCI の受信バッファの構成が、キーボードからの入力を前提に作成したものを使いまわしているので、効率が悪いです。
→SCI の受信バッファを 2 面構成（格納用と読み出し用）として、常にオフセットアドレス 0 からデータが格納される様にする（＝SCI の受信バッファから直接 UDP の送信バッファにコピーできる様にする）。UDP の送信バッファにコピーする際、0x7FB-0x7FF, 0x0-0x5 の 2 回コピーで済む様に修正するなど、一時的に使用している buf が不要となる様な形が望ましいとは思います。
→単純に、アドレス a からアドレス b までのデータを、アドレス c にコピーする処理であれば、コンパイル結果が SMOVF 命令（順方向ストリング転送命令）に置き換わる様なコードとすれば、高速に処理ができます。
※ここでは、効率改善の余地があるという事を示唆するに留めて、具体的な改善後のコードは提示しません

UDP でデータを送信するところは、前のプロジェクト(RX65N_LWIP)と同じです。
但し、RX65N_LWIP プロジェクトでは、pcb(protocol control block)を毎回作成、送信後に削除としていましたが、本プロジェクトでは最初に 1 回作成して、以降同じ pcb で送信を繰り返す動作としています。
→送信の度に送信先の IP アドレスやポート番号を設定する動作とはしていません

10.3.UDP での受信処理

Ethernet のデータ受信処理は、メインループ内の R_LWIP_DRIVER_Input()関数内で行われています。

30000 番ポートに来た UDP のデータは、udp_server_listen()関数内で登録したコールバック関数 udp_server_recv()内で処理されます。

・server.c

```
static void udp_server_recv(void *arg, struct udp_pcb *tpcb, struct pbuf *p, const ip_addr_t
*addr, u16_t port)
{
    //UDPサーバ受信関数

    unsigned char *buf = NULL;
    unsigned short len;

    //データを受信
    len = p->len;

    buf = malloc(len);
    if (buf != NULL)
    {
        memcpy(buf, p->payload, len);
    }

    //バッファメモリの解放
    pbuf_free(p);

    if (buf != NULL)
    {
        unsigned short i;

        for (i=0; i<len; i++) sci_write_char(buf[i]);

        free(buf);

#ifdef ETHER_DEBUG_LEVEL_0
        debug_sci_write_str("¥n-- ");
        debug_sci_write_uint16(len);
        debug_sci_write_str(" bytes received --¥n");
#endif
    }
}
```

受信側の処理も、一時的に buf を確保して、SCI の送信処理（送信バッファにデータを渡す）形としています。
→SCI の送信側の処理も、デバッグメッセージの表示用に作成した sci の関数を使いまわしているので、非常に効率が悪いです。
→11 章で説明するプロジェクトでは、上記 sci_write_char で 1 バイトずつコピーするところを、一括でコピーする形にしています。

ETHER_DEBUG_LEVEL_0 定数は、ether_operation.h 内で、コメントアウトで定義されています。この定数を定義すると、受信の度に何バイト受信したか、表示されます（受信データに「何バイト受信したか」の情報が付与されます）。
→有効にするとファイル転送を行う際は余計なデータが挟まる形になります
（あくまでデバッグ用です）

10.4.ユーザ作成関数

udp_client_init

概要:UDP クライアントの初期化
宣言:void udp_client_init(unsigned char *dst_IP, unsigned short dst_port)
説明:
・UDP クライアント(データ送信側)の初期化
を行います
引数:
　unsigned char *dst_IP: 送信先 IP アドレス
　unsigned short dst_port: 送信先ポート
戻り値:なし
補足:通信で使用する udp_pcb 構造体を準備して、IP アドレスとポート番号の紐づけを行います

udp_client_close

概要:UDP クライアントの終了処理
宣言:void udp_client_close(void)
説明:
・UDP クライアント(データ送信側)の pcb を閉じる処理
を行います
引数:なし
戻り値:なし
補足:本プロジェクトでは未使用です

udp_server_listen

概要:UDP サーバの立ち上げ
宣言:void udp_server_listen(unsigned short listen_port)
説明:
・UDP サーバの通信待ち受けの開始
を行います
引数:
　unsigned short listen_port: 受信データ待ち受けポート
戻り値:なし

udp_server_listen_close

概要:UDP サーバの終了

宣言:void udp_server_listen_close(struct udp_pcb *tpcb)

説明:

・UDP サーバ待ち受けの終了

を行います

引数:

　struct udp_pcb *tpcb: udp_pcb 構造体

戻り値:なし

11.RX65N_UART_ETHER_TCP_DMAC プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_UART_ETHER_TCP_DMAC
SOURCE¥176PIN¥RX65N_UART_ETHER_TCP_DMAC

（使用しているマイコンボードのピン数に応じたフォルダ）を PC のストレージにコピーしてください。

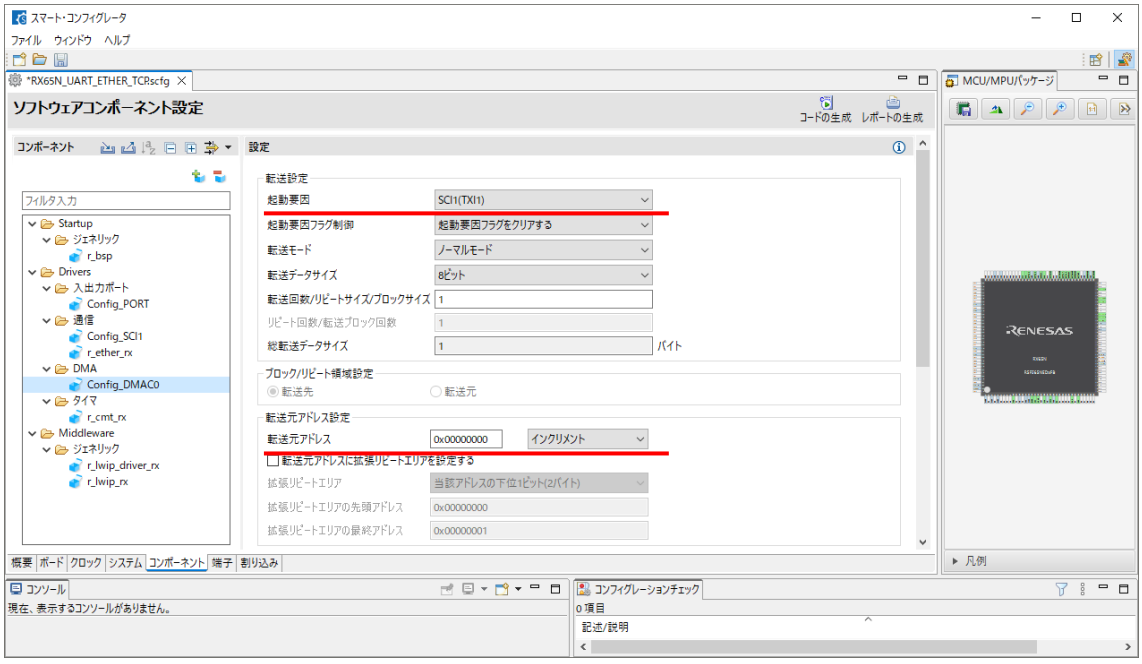
本プロジェクトは、
・マイコンボードが UART-ETHER ブリッジ(TCP)として動作する
事を行います。

RX65N_UART_ETHER_UDP プロジェクトでは、Ether の通信に UDP を使用していましたが、本プロジェクトでは TCP の通信に置き換わっています。（RX65N_UART_ETHER_UDP の TCP 版です）

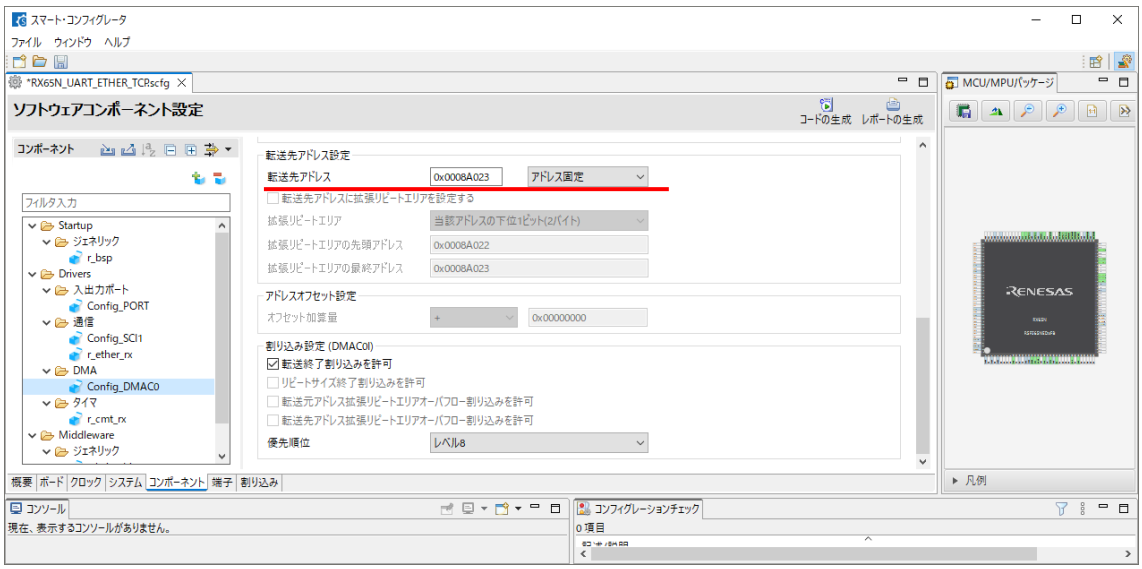
・スマート・コンフィグレータで追加するコンポーネント

コンポーネント	備考
Config_PORT	SW と LED のポート設定
Config_SCI1	SCI(UART)通信
r_ether_rx	Ethernet
Config_DMAC0	DMAC
r_cmt_rx	コンペアマッチタイマ
r_lwip_driver_rx	lwIP ドライバ
r_lwip_rx	lwIP TCP/IP

・Config_DMAC0



起動要因は、SCI1(TXI1)。転送元アドレスは後で代入しますので値は仮値です。インクリメントを選択してください。



転送先アドレスは、0x0008A023 を指定（SCI1 の TDR（送信データレジスタ）の値とします。

本プロジェクトでは、SCI の送信側（Ether でデータを受信した際に SCI で送信）に、DMAC(Direct Memory Access Controller)を使用しています。

今までのプロジェクトでは、SCI の送信は割り込みで処理していました。本プロジェクトでは、送信に DMAC を使う事とします。

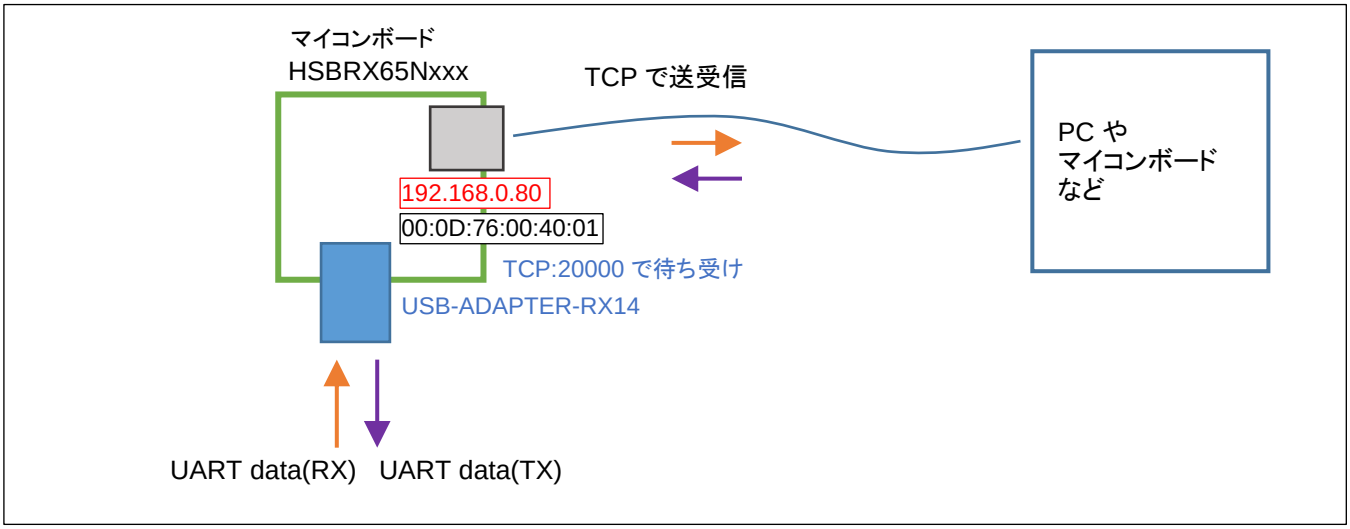
転送元アドレスは、0x0 としていますが、実際の DMAC 使用時に適切なアドレスを設定します。

転送先アドレスは、SCI1 の TX のデータバッファとなるので、固定値となります。

・ユーザ作成のプログラムコード

フォルダ（カテゴリ）	ファイル名	備考
client	client.c	クライアント動作のコード
	client.h	上記ヘッダ
common	ether_common.c	Ether のプログラムで共通に使用する関数
	ether_common.c	上記ヘッダ
	ether_operation.c	IP アドレスなどの定義
	ether_operation.h	上記ヘッダ
main	main.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ
server	server.c	サーバト動作のコード
	server.h	上記ヘッダ

・本プロジェクトの動作



RX65N_UART_UDP プロジェクトの、UDP での通信の部分が TCP に変わっただけです。

起動後、以下の様なメッセージが表示されます。

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether UART Ether TCP sample program (with SCI1-TX->DMAC).
-setting address-
---
this board IP address                -> 192.168.0.80 (MAC address -> 00-0D-76-00-40-01 )
netmask                                 -> 255.255.255.0
gateway IP address                   -> 192.168.0.1
communication target IP address -> 192.168.0.81
---
-- TCP SERVER LISTEN START (port = 20000) --
```

※デバッグ表示モード有効時に、上記メッセージが表示されます

・ether_common.h

```
#define SCI_DEBUG//UART(SCI)によるデバッグ表示を行う
```

ether_common.h 内の上記行をコメントアウトすれば、起動時にメッセージは表示されなくなります。

→純粋な UART-Ether 変換機器として動作します

2 台のマイコンボードで UART-Ether-UART ブリッジを構成する場合は、片方のマイコンボードの DIPSW(3)を ON に設定してください。その場合は、IP アドレス=192.168.0.81, MAC アドレス=00-0D-76-00-40-02 に設定されます。）

起動時に、接続先の IP アドレスなどを変えたい場合は、ボード上の SW3 を押したままで、電源投入またはリセットボタンを押してください。(SW3 は画面にメッセージが表示されるまで押したままとしてください。)

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether UART Ether TCP sample program (with SCI1-TX->DMAC).

-- IP address settings change --

COMMAND:
  1 : this board
  2 : netmask
  3 : gateway
  4 : communication target
  5 : TCP servrt port (default = 20000)
  6 : TCP client port (default = 20000)
  p : print setting
  e : exit setting
```

その場合は、キーボードからアドレスやポート番号の設定ができる様になります。

デフォルトでは、

通信相手の IP アドレス　192.168.0.81

本ボードの IP アドレス　192.168.0.80

本ボードの MAC アドレス　00-0D-76-00-40-01

TCP サーバの待ち受けポート　20000

TCP クライアントのメッセージの送信先ポート　20000

です。(起動時に DIP-SW の 3 が ON になっていると、送信先と本ボードの IP アドレスが逆に設定されます。)

起動後、

```
-- TCP SERVER LISTEN START (port = 20000) --
```

のメッセージが表示されると、接続可能状態となります。通信相手から本ボードに接続する場合は、接続処理を行ってください。(PC アプリの場合は、接続ボタンを押す)

通信相手からの接続が完了した場合は、

```
-- TCP SERVER LISTEN START (port = 20000) –
-- TCP CONNECTION ACCEPTED --
(connected from 192.168.0.81:52362)
```

赤字の表示が出ます。

本ボードから、別な機器(ボードや PC)に接続する場合は、SW3 を押してください。従来のプロジェクトでは、接続はキーボードから c コマンドを入力して接続していましたが、本プロジェクトは UART の信号を互いに疎通させるものですので、UART でコマンドを入力する様にはしていません。

ボードの SW3 を押すと、（かつ、通信相手が接続可能な状態となっている場合）

```
-- TCP SERVER LISTEN START (port = 20000) –
-- TCP CONNECTION ESTABLISHED --
```

赤字の表示が出るはずです。また、ボード上の LED が点灯します。
→LED 点灯はデータ送信の準備が整ったという意味です

ether_common.h 内の
//#define SCI_DEBUG
をコメントアウトした場合、画面にはメッセージが表示されなくなり、純粋な UART-Ether コンバータとして動作しますので、その場合は LED の点灯を確認してください。

通信相手から接続後に、本ボードの 20000 番ポートに到達したデータは、UART(SCI1)で外部に送信されます。通信相手と接続後に、UART 経由でデータを流し込むと、通信相手に TCP でデータを送り付けます。

・通信相手と相互に接続
・送信と受信で、それぞれ別の通信相手と接続
・送信と受信の一方のみ使用
どのような形でも問題ありません。

11.1.メイン関数での処理

main.c 内の ether_main()では、

・変数の初期化
・PHY チップのリセット
・SCI 初期化
・起動時のメッセージ表示
・DIP-SW のチェック
→DIP-SW の 3 番が ON の場合は、IP アドレス=192.168.0.81, MAC アドレス=00-0D-76-00-40-02 に設定
・SW3 のチェック（押されている時は IP アドレス変更）
・LWIP ドライバ動作開始
・PHY チップのリセット解除
・LWIP 初期化
・ネットワークインタフェース設定
（・MAC アドレスの変更[変更する場合のみ]）
・Ether のリンク確認
・ネットワークインタフェース有効化
・CMT1 開始
・メインループ

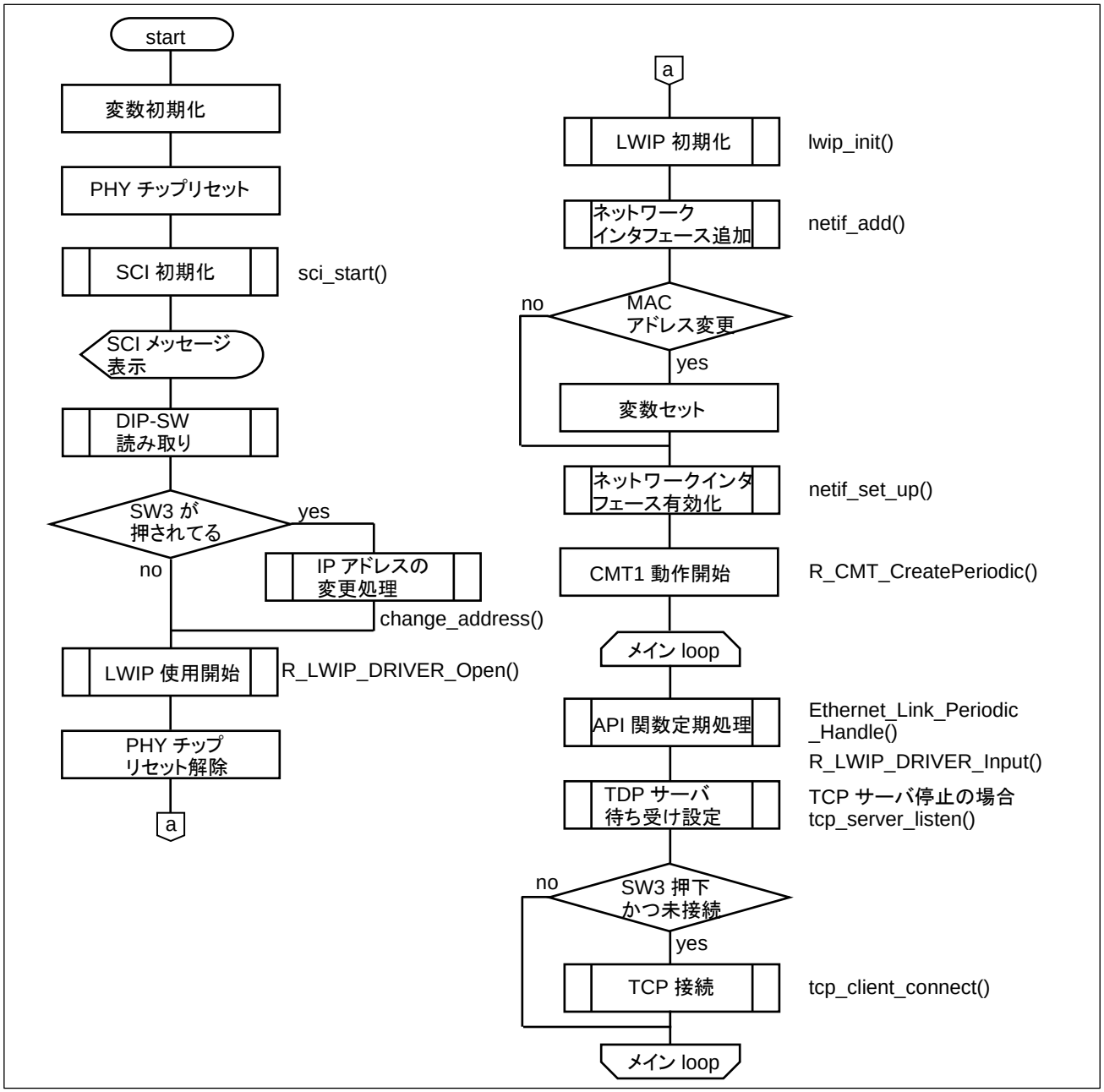
の処理を行っています。

メインループでは、

・API 関数の定期処理
・TCP サーバ待ち受け処理
・SW3 を監視して TCP の接続処理

となっています。

・ether_main() （main.c 内）



11.2.TCP の接続処理（クライアント動作）

SW3 が押された場合で、未接続の場合は以下の関数が実行されます。

・client.c

```
void tcp_client_connect(unsigned char *dst_IP, unsigned short dst_port)
{
    //TCP接続関数

    //戻り値
    //  なし

    //引数
    //  *dst_IP : 接続先IPアドレス
    //  dst_port : 接続先ポート

    ip_addr_t dst_ipaddr;

    //TCP pcb作成
    g_client_tcp_pcb = tcp_new();

    if (g_client_tcp_pcb != NULL)
    {
        IP4_ADDR(&dst_ipaddr, dst_IP[0], dst_IP[1], dst_IP[2], dst_IP[3]);

        //接続
        tcp_connect(g_client_tcp_pcb, &dst_ipaddr, dst_port, tcp_client_connected);
    }
}
```

TCP 用の pcb(protocol controll block)を作成して、API 関数の tcp_connect()を呼ぶ形で、RX65N_LWIP プロジェクトでの c コマンド実行時と特に変わりません。

tcp_client_connected()では、各種コールバック関数の登録を行っています。（9.6.1 節を参照）

・client.c

```
static err_t tcp_client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    //TCPクライアント受信関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    if (p == NULL)
    {
        //空のTCPフレームを受信→接続を切断，ここではFINフラグ受信に対してACKを返す（後でこちらからも
tcp_server_poll() 内でFINを送信）
        es->tcp_connection = TCP_CONNECT_CLOSEING;

        return ERR_OK;
    }

    //スタックに受信サイズ通知
    tcp_recved(tpcb, p->tot_len);

    //バッファメモリの解放
    pbuf_free(p);

    //クライアントで受信した場合は受信データに対して何も処理しない

    return ERR_OK;
}
```

受信コールバック関数(tcp_client_recv())を、RX65N_LWIP プロジェクトとは多少変えており、
・RX65N_LWIP プロジェクト： 受信メッセージを表示
・本プロジェクト：何もしない（受信バッファのメモリ開放のみおこなう）
という形です。本プロジェクトでは、クライアント接続ポートにサーバからデータが送られてきても何もしない動作となります。（UART ブリッジとしての動作は、サーバ側に送られてきたデータを SCI1-TX から送信。）

11.3.TCP の送信処理（クライアント動作）

SCI1 の RX で受信したデータを TCP サーバに送る動作です。

本プロジェクトでは、CMT1 の定期処理が追加されており、100ms に 1 回以下の処理が実行されます。

・main.c

```
void Uart_Periodic_Handle(void *pdata)
{
    //UARTの定期処理(100ms毎)

    err_t err;
    unsigned short size;
    char *buf = NULL;

    if (g_client_es.tcp_connection == TCP_DISCONNECT) return;//TCPコネクションが成立していない

    //現時点でSCIの受信バッファに溜まっているサイズを取得
    size = sci_read_data_size();

    if (size == 0) return;//データなし

    //1460バイト以上の場合は、今回は1460バイト送信する（分割送信とする）
    if (size > TCP_MAX_PACET_SIZE)
    {
        size = TCP_MAX_PACET_SIZE;
    }

    buf = malloc(size);

    if (buf != NULL)
    {
        sci_read_str(buf, size);

        //TCPで送信
        err = tcp_write(g_client_tcp_pcb, buf, size, 1);//1:バッファにコピーを行う

        free(buf);

        if (err == ERR_OK)
        {
            debug_sci_write_str("¥n-- ");
            debug_sci_write_uint16(size);
            debug_sci_write_str(" bytes send --¥n");
        }
        else
        {
            debug_sci_write_str("tcp_send error.¥n");
        }
    }
}
```

SCI バッファに、1460（=1514-14（Ethernet ヘッダ）-20（IP ヘッダ）-20（TCP ヘッダ））バイト以上のデータがある場合は、今回は 1460 バイト送信し、残りは次回の送信に回されます。

送信の処理は、RX65N_UART_ETHER_UDP プロジェクトと変わりません。

11.4.TCP の受信処理（サーバ動作）

サーバ側で受信したデータは、SCI の TX に送られます。

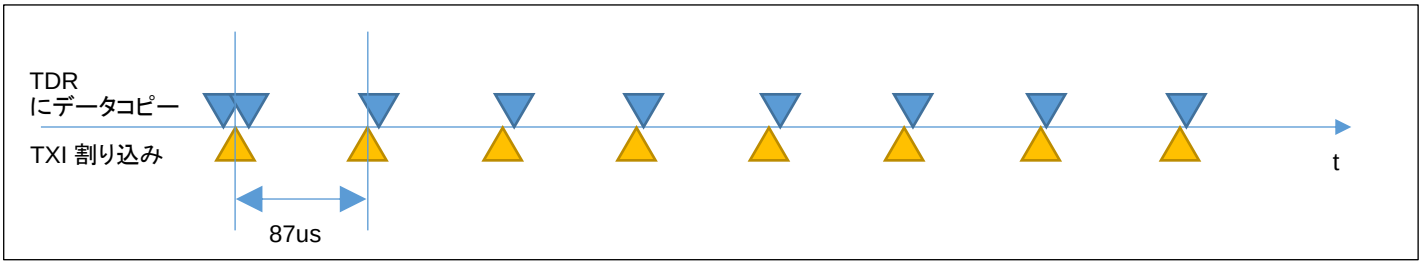
Ethernet のデータ受信処理は、メインループ内の R_LWIP_DRIVER_Input()関数内で行われています。
20000 番ポートに来た TDP のデータは、tcp_server_connected()関数内で登録したコールバック関数 tcp_server_recv()内で処理されます。

・server.c

```
static err_t tcp_server_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    //TCPサーバ受信関数

    struct TCP_ARG *es = (struct TCP_ARG *)arg;

    unsigned char *buf = NULL;
    unsigned short len;

    if (p == NULL)
    {
        //空のTCPフレームを受信→接続を切断，ここではFINフラグ受信に対してACKを返す（後でこちらからも
tcp_server_poll() 内でFINを送信）
        es->tcp_connection = TCP_CONNECT_CLOSEING;

        return ERR_OK;
    }

    //データを受信
    len = p->len;

    buf = malloc(len);

    if (buf != NULL)
    {
        memcpy(buf, p->payload, len);
    }

    //スタックに受信サイズ通知
    tcp_recved(tpcb, p->tot_len);

    //バッファメモリの解放
    pbuf_free(p);

    if (buf != NULL)
    {

        //SCI送信バッファにまとめてコピー
        sci_write_data(buf, len);

        free(buf);

#ifdef ETHER_DEBUG_LEVEL_0
        debug_sci_write_str("¥n-- ");
        debug_sci_write_uint16(len);
        debug_sci_write_str("(");
        debug_sci_write_uint16(p->tot_len);
        debug_sci_write_str(")");
        debug_sci_write_str(" bytes received --¥n");
#endif
    }

    return ERR_OK;
}
```

```
//SCI送信バッファに1バイトずつコピー
//unsigned short i;
//for (i=0; i<len; i++) sci_write_char(buf[i]);
```

RX65N_UART_ETHER_UDP プロジェクトでは、sci_write_char()関数で 1 バイトずつコピーしていましたが、本プロジェクトでは sci_write_data()関数で、一括でのコピーとしています。
また、RX65N_UART_ETHER_UDP プロジェクトでは割り込みでデータを送信していました。



SCI1 の TDR（送信バッファ）にデータをコピーした際、初回は TDR が直ぐに空になり、TXI（送信バッファが空いた）割り込みが入ります。TXI 割り込みルーチン内で、次のデータを TDR にコピーする動作となります。2 回目以降は、約 87us に 1 回 TXI 割り込みが入る形になります。（87us→115,200bps での 10bit 時間）その度に TXI の割り込みルーチンに飛んできて、TDR にデータをコピーする動作が繰り返されます。

この動作は、Ether や CMT の割り込みに邪魔されても問題はありません。SCI の TX から送信されるデータ間隔に隙間が空くだけでデータの消失には直接つながりません。（TCP で受信するデータレート(Bytes/s)に対し、TXI 割り込みのレート（割り込み回数/s）の平均値が間に合っていれば良いです。）
（※SCI の受信側、RXI 割り込みに関しては、平均のレートで間に合っていても NG です。SCI の RX に 87us 間隔でデータが来る場合は、87us 以内に前の割り込み処理が終わらない場合は、オーバランエラー（データの取りこぼし）が生じます。）

本プロジェクトでは、TX のデータ送信に関して定期的（87us 毎）な割り込みによる処理ではなく、DMAC で処理する様にしています。その場合、1 バイトのデータを送信して TDR が空になったタイミングで DMAC が起動されて、SCI の送信バッファ（ユーザプログラムで確保したメモリ上の領域）から、SCI1 の TDR にデータがコピーされます。CPU がそのタイミングで別な処理を行っていても邪魔をすることがありません。但し、CPU 内部のデータバスは RAM から TDR にコピーするデータが流れますので、そのタイミングで RAM アクセスなどがあった場合は、アクセス待ちが入ります。

（送信バッファにコピーした後、SCI の TX からデータを出力する処理は、ユーザプログラムで何かしたいことはなく、機械的に処理してくれれば良いので、DMAC の使用に適しています。また、送信バッファにデータをコピーした段階で、何回 DMAC を起動すれば良いかが確定します。）

（SCI の受信側に DMAC を使うという方法もあります。受信側の場合は、SCI でデータを受信した後で、Ether(TCP)の送信処理にデータを渡す必要があります。受信して終わりではなく、受信したことを検出して、次の処理（関数など）を起動する必要があります。また、DMAC を起動するタイミングで、何回 DMAC を起動すべきなのかが判りません。例えば、DMAC の起動（リピート）回数を 100 回に設定すると、100 バイトのデータが溜まるまで次のアクションを起こせません。DMAC の起動回数を 1 回に設定するのであれば、DMAC を使うメリットがありません（＝SCI の受信割り込みは 1 バイト毎に発生、DMAC の割り込みも 1 バイト毎に発生）。）

（そこで、DMAC の起動（リピート）回数は 100 や 1000 等のある程度大きな値としておき、かつタイマで一定時間毎に、その時点で受信しているデータを次の処理に渡すといった方法が考えられます。その場合は、設定したリピート回数のデータを受信したタイミングとタイマで設定したタイミングで、Ether の送信にデータを渡す事ができます。かつ、1 バイト毎の受信割り込みが不要となります。）

※本プロジェクトでは、SCI の送信に DMAC を使い、受信は割り込みで処理する形です。

11.5.ユーザ作成関数

tcp_client_connect

概要：TCP 接続（クライアント側で実行する関数）
宣言：void tcp_client_connect(unsigned char *dst_IP, unsigned short dst_port)
説明：
・TCP サーバへの接続
を行います
引数：
　unsigned char *dst_IP: 接続先 IP アドレス
　unsigned short dst_port: 接続先ポート
戻り値：なし

tcp_client_connection_close

概要：TCP 切断（クライアント側で実行する関数）
宣言：void tcp_client_connection_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明：
・TCP サーバからの切断
を行います
引数：
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　struct TCP_ARG *es: 引数を渡すための構造体
戻り値：なし

tcp_client_force_connection_close

概要：TCP 強制切断（クライアント側で実行する関数）
宣言：void tcp_client_force_connection_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)

説明:
・TCP サーバからの強制切断
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　struct TCP_ARG *es: 引数を渡すための構造体
戻り値:なし

tcp_server_listen

概要:TCP サーバ起動
宣言:void tcp_server_listen(unsigned short listen_port)
説明:
・TCP サーバの待ち受け開始
を行います
引数:
　unsigned short listen_port: 待ち受けポート
戻り値:なし

tcp_server_listen_close

概要:TCP 切断（サーバ側で実行する関数）
宣言:void tcp_server_listen_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明:
・TCP サーバ側からの切断
・TCP サーバの待ち受け終了
を行います
引数:
　struct tcp_pcb *tpcb: tcp_pcb 構造体
　struct TCP_ARG *es: 引数を渡すための構造体
戻り値:なし

tcp_server_force_listen_close

概要:TCP 強制切断（サーバ側で実行する関数）
宣言:void tcp_server_listen_close(struct tcp_pcb *tpcb, struct TCP_ARG *es)
説明:
・TCP サーバ側からの強制切断
・TCP サーバの待ち受け終了
を行います

引数:

struct tcp_pcb *tpcb: tcp_pcb 構造体

struct TCP_ARG *es: 引数を渡すための構造体

戻り値:なし

取扱説明書改定記録

バージョン	発行日	ページ	改定内容
REV.1.0.0.0	2026.6.8	—	初版発行

お問合せ窓口

最新情報については弊社ホームページをご活用ください。

ご不明点は弊社サポート窓口までお問合せください。

株式会社 北斗電子

〒060-0042 札幌市中央区大通西 16 丁目 3 番地 7

TEL 011-640-8800　FAX 011-640-8801

e-mail：support@hokutodenshi.co.jp (サポート用)、order@hokutodenshi.co.jp (ご注文用)

URL:https://www.hokutodenshi.co.jp

ルネサス エレクトロニクス RX マイコン搭載
HSB シリーズマイコンボード 評価キット

Ethernet スターターキット RX65N
ソフトウェア編　取扱説明書(3)

株式会社 北斗電子

 2026 年 6 月 8 日改訂 REV.1.0.0.0 (260608)