Ethernet スタータキット RX65N
ソフトウェア編　取扱説明書(2)

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

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

CD 内の

SOURCE¥144_100PIN¥RX65N_UDP
SOURCE¥176PIN¥RX65N_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、
・マイコンボードから他の機器に対して UDP データ（テキストメッセージ）を送信する
・他の機器からマイコンボードに対して飛んできた UDP データ（テキストメッセージ）を表示する
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
Config_CMT0	コンペアマッチタイマ（100ms 毎）

（→RX65N_MAGIC_PACKET/RX65N_MAGIC_PACKET2 プロジェクトと同じ）

・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
arp	arp.c	ARP のプロトコルを取り扱う関数
	arp.h	上記ヘッダ
client	**text_client.c**	UDP のテキストメッセージの送信プログラム（クライアント）
	text_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
ethernet	ethernet.c	イーサネットヘッダを取り扱う関数
	ethernet.h	上記ヘッダ
interrupt	ether_interrupt.c	割り込み関数
	ether_interrupt.h	上記ヘッダ
ip	ip.c	IP(InternetProtocol)を扱う関数
	ip.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	**text_server.c**	UDP のテキストメッセージの待ち受けプログラム（サーバ）
	text_server.h	上記ヘッダ
udp	**udp.c**	UDP を取り扱う関数
	udp.h	上記ヘッダ

太字のフォルダ、ファイルが新規に追加されたものです。

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 UDP text client/server sample program.

COMMAND:
   t : send text message set
   p : send text message print
   s : send text message

-setting address-
---
communication target IP address -> 192.168.0.81
this board IP address                -> 192.168.0.80 (MAC address -> 00-0D-76-00-40-01)
---
-setting port-
---
UDP server listen port -> 30000
UDP client dest      port -> 30000
UDP client source port -> 50000
---

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

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

UDP クライアントで送信元として使用するポート　50000

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/MAC address settings change --

COMMAND:
   1 : communication target IP address
   2 : this board IP address
   3 : this board MAC address
   4 : UDP server listen port (default = 30000)
   5 : UDP client dest port      (default = 30000)
   6 : UDP client source port (default = 50000)
   p : print setting
   e : exit setting
```

1~6 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。

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!
Ether0: LINK-UP
```

・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

//通信相手のMACアドレス
unsigned char g_dst_MAC[6] = {0};//プログラム内で通信相手に応じて代入

//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};

//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UDPでテキストメッセージを送る
unsigned char g_dst_IP[4] = {192,168,0,81};

//クライアント時：送信元（本ボード）ポート番号
unsigned short g_client_src_port = 50000;

//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_client_dst_port = 30000;

//サーバ時：送信元ポート番号（通信相手が使用するポート番号，任意）
unsigned short g_server_src_port = 0;//プログラム内で通信相手に応じて代入

//サーバ時：宛先ポート番号（本ボード，この番号で待ち受け）
unsigned short g_server_dst_port = 30000;
```

IP アドレスや MAC 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。

7.1. 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
・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
・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
・Ether の初期化
・コールバック関数の登録
・Ether のスタンバイ解除
・端子設定
・Ether の動作開始
・メインループ

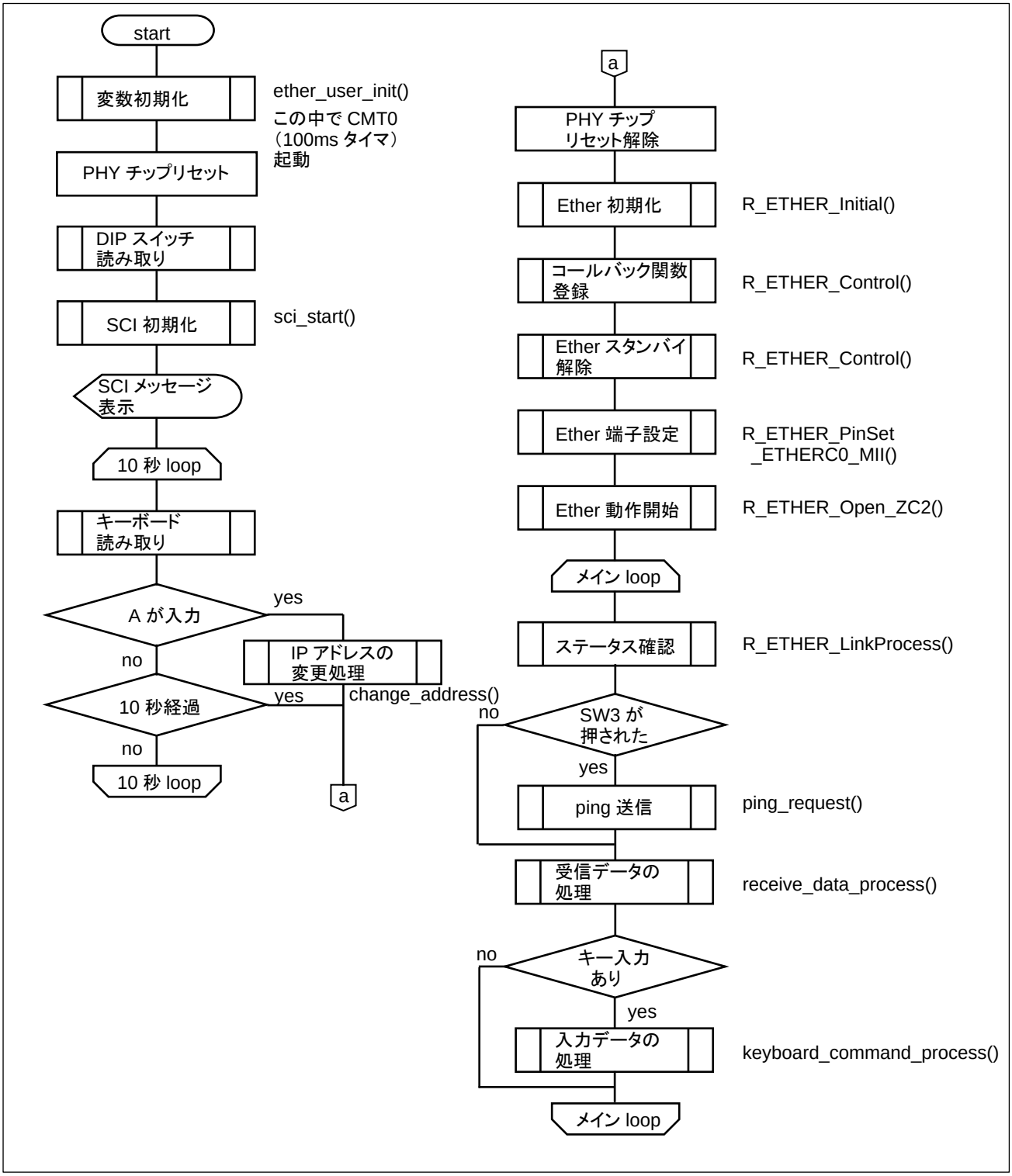
の処理を行っています。

メインループでは、

・API 関数のステータスチェック
・SW3 が押されている場合は ping 送信
・受信データの処理
・キーボードからのコマンドの処理

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 s を入力すると、UDP でメッセージを送信します。

SW3 は、通信相手に設定した IP アドレスに ping を送信します。

7.2. UDP の送信処理

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
send text message to 192.168.0.81 port 30000

send target 192.168.0.81 -> MAC address not resolved.
ARP request (192.168.0.81-> MAC address?) send
ARP reply received from 192.168.0.81, MAC address = 00-0D-76-00-40-02
UDP text data send to 192.168.0.81:30000
```

赤字の部分は、初回だけ表示されます。

初回は送信先（ここでは、192.168.0.81）に対応する MAC アドレスを知らないので、ARP リクエストで MAC アドレスの解決（192.168.0.81 に設定されている機器の MAC アドレスを調べる）を行います。MAC アドレス判明後に、UDP でメッセージを送信します。

送信するメッセージは、デフォルトでは"hello,world　　"です。t 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。）

メッセージの送信は、
text_packet_send()
関数で行っています。

・text_client.c

```
int text_packet_send(unsigned char *dst_IP, unsigned short dst_port, unsigned char *text_data)
{
    //UDPテキストメッセージ送信関数

    //戻り値
    //  ETHER_SUCESS(0) : 正常終了
    //  ETHER_RET_SEND_QUEUE(1) : 送信キューに入った（条件が満たされた段階で送信）
    //  ETHER_RET_FIFO_OVERFLOW(-13) : 前回の呼び出しの送信が未完了（送信待ち）

    unsigned short i;

    if (g_delay_execution_count != 0)
    {
        //遅延実行がスケジュールされているので関数呼び出しは無効とする

        sci_write_str("¥nMessage send scheduled, so skip.¥n");

        return ETHER_RET_FIFO_OVERFLOW;
    }

    //メッセージ送信先のMACアドレス
    if ((g_dst_MAC[0] == 0) && (g_dst_MAC[1] == 0) && (g_dst_MAC[2] == 0))
    {
        /*
         * 宛先MACアドレスの先頭3バイト（ベンダコード）が0x00の場合は
         * IPアドレスとMACアドレスの対応が判明していないのでARPリクエストを送る
         */

         sci_write_str("¥nsend target ");
         sci_print_IP(dst_IP);
         sci_write_str(" -> MAC address not resolved.¥n");

        ARP_request(dst_IP);

        //今回はARPリクエストを送って終了とする
        //（タイマでメッセージ送信関数を遅延実行しIP-MAC対応が判明している場合はUDPメッセージを送る）

        //遅延実行をスケジュール
        g_delay_execution_count = 20;//100ms毎に最大20回メッセージ送信を試行する

        //遅延実行時に参照される引数
        for (i=0; i<4; i++) g_delay_execution_dst_IP[i] = *dst_IP++;
        g_delay_execution_dst_port = dst_port;
        for (i=0; i<17; i++) g_delay_execution_text_data[i] = *text_data++;

        return ETHER_RET_SEND_QUEUE;
    }

    //宛先MACアドレスが判明している場合はメッセージ送信を行う
    text_packet_send_core(dst_IP, dst_port, text_data);

    return ETHER_SUCCESS;
}
```

UDP メッセージ送信関数は上記の様になっています。

メッセージ送信先の IP アドレスに対する MAC アドレスが不明な場合は、ARP リクエスト（ARP_request()）を実行して、本関数を抜けます。その後、100ms 毎のタイマ割り込みルーチン内で、MAC アドレスの解決ができていれば、UDP でメッセージを送信します。

本関数実行のタイミングで MAC アドレスの解決ができていれば、UDP メッセージ送信関数の本体である text_packet_send_core()を実行します。

メッセージ送信関数の本体である、text_packet_send_core()は、以下の様になっています。

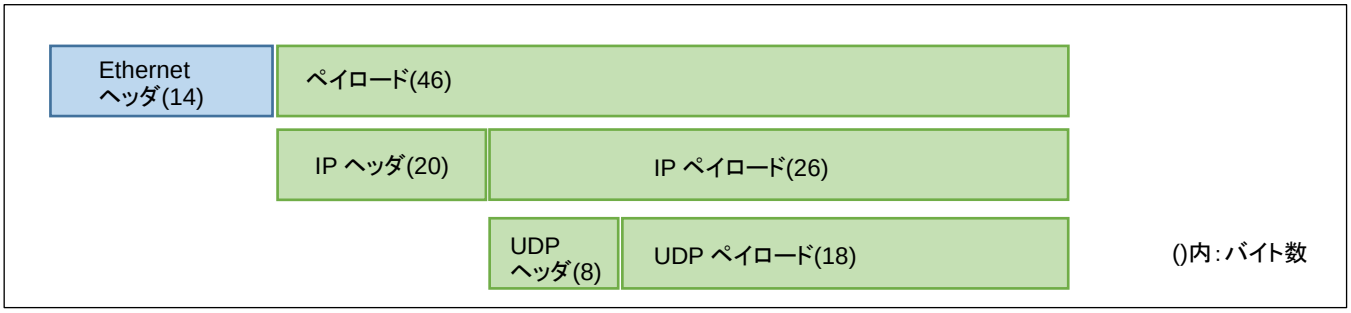
・text_client.c

```
static void text_packet_send_core(unsigned char *dst_IP, unsigned short dst_port, unsigned char
*text_data)
{
    unsigned short i;

    ETHERNET_HEADER ethernet_packet;
    IP_HEADER ip_packet;
    UDP_HEADER udp_packet;
    PSEUDO_HEADER udp_pseudo_packet;

    ether_return_t ret;

    unsigned char send_data[60] = {0};

    //Ethrnetパケットの設定//

    //宛先MACアドレスはARPリクエストで判明したMACアドレスを設定，送信元は本ボードのMACアドレスを指定
    for (i=0; i<6; i++)
    {
        ethernet_packet.eth_dst_MAC[i] = g_dst_MAC[i];
        ethernet_packet.eth_src_MAC[i] = g_src_MAC[i];
    }

    //イーサネットタイプ(IP)
    ethernet_packet.eth_ethernet_type = 0x0800;

    //IPヘッダの設定//

    //バージョン及びヘッダ長
    ip_packet.ip_version_length = 0x45;

    //パケット優先度
    ip_packet.ip_service_type = 0x00;

    //IPデータグラムの長さ
    ip_packet.ip_total_length = 20 + 8 + 16;

    //パケットの識別子
    ip_packet.ip_id = 0xABCD;//本プログラムでは固定値

    //パケットの分割有無
    ip_packet.ip_flags_fragment_offset = 0x4000;

    //ルータ中継回数の上限
    ip_packet.ip_time_to_live = 0x80;

    //上位プロトコルの種類(UDP)
    ip_packet.ip_protocol = 0x11;

    //チェックサム（ここでは仮値）
    ip_packet.ip_checksum = 0x0000;

    //送信元は本ボードのIPアドレスを指定，宛先IPアドレスは引数で指定されたIPアドレスを設定
    for (i=0; i<4; i++)
    {
        ip_packet.ip_src_IP[i] = g_src_IP[i];
        ip_packet.ip_dst_IP[i] = dst_IP[i];
    }

    //チェックサムの計算
    ip_packet.ip_checksum = ip_header_sum(&ip_packet);
```

・text_client.c（続き）

```
    //UDPヘッダの設定//

    //送信元ポート
    udp_packet.udp_src_port = g_client_src_port;

    //宛先ポート
    udp_packet.udp_dst_port = dst_port;

    //セグメントの長さ
    udp_packet.udp_length = 8 + 16;//ヘッダ8バイト，メッセージ16バイト

    //チェックサム（ここでは仮値）
    udp_packet.udp_checksum = 0x0000;

    //UDPチェックサム計算のためUDP疑似ヘッダ
    for (i=0; i<4; i++)
    {
        udp_pseudo_packet.ip_src_IP[i] = ip_packet.ip_src_IP[i];//送信元IPアドレス
        udp_pseudo_packet.ip_dst_IP[i] = ip_packet.ip_dst_IP[i];//送信先IPアドレス
    }
    udp_pseudo_packet.zero_padding = 0;//ゼロパディング
    udp_pseudo_packet.ip_protocol = ip_packet.ip_protocol;//上位プロトコルの種類(UDP)
    udp_pseudo_packet.length = udp_packet.udp_length;//UDPセグメントの長さ

    //チェックサムの計算
    udp_packet.udp_checksum = udp_sum(&udp_pseudo_packet, &udp_packet, text_data, 16);

    //UDPメッセージパケット = ETHERNETヘッダ(14) + IPヘッダ(20) + UDPヘッダ(8) + UDPデータ(16) ※ここで
は16バイトのメッセージを送る処理とする
    ETHERNET_header_to_stream(&ethernet_packet, send_data);
    IP_header_to_stream(&ip_packet, &send_data[14]);
    UDP_header_to_stream(&udp_packet, &send_data[14+20]);

    //UDPデータの設定
    for (i=0; i<16; i++)
    {
        send_data[14+20+8+i] = text_data[i];
    }
    //合計58バイト，送信データは60バイト

    //UDPパケット送信
    ret = R_ETHER_Write(g_ether_channel, (void *)send_data, sizeof(send_data));

    sci_write_str("UDP text data send to ");
    sci_print_IP(dst_IP);
    sci_write_str(":");
    sci_write_uint16(dst_port);

    if (ret != ETHER_SUCCESS)
    {
        sci_write_str(", [failed]");
    }
    sci_write_str("¥n");
}
```

－送信時に設定するデーター



Ethernet フレームとしては 60 バイトのデータ。Ethernet フレームのペイロードは、IP ヘッダ＋IP ペイロードで構成。IP のペイロードは、さらに UDP のヘッダ＋UDP ペイロードで構成される形です。

－通信パケット例－

・Ethernet ヘッダ(14)

00	0D	76	00	40	02	00	0D	76	00	40	01	08	00
宛先 MAC アドレス						送信元 MAC アドレス						Ethernet type (IPv4)	

・IP ヘッダ(20)

45	00	00	2C	AB	CD	40	00	80	11	??	??	C0	A8	00	50
IPv4 ヘッダ 20B	Service type	Total Length 44 バイト		ID (*1)		Fragment		TTL	Protocol UDP	Checksum (*2)		送信元 IP アドレス (192.168.0.80)			

C0	A8	00	51
送信先 IP アドレス (192.168.0.81)			

・UDP ヘッダ(8)

C3	50	75	30	00	18	??	??
Source port (50000)		Destination Port (30000)		length (24 バイト)		Checksum (*3)	

・UDP データ(16)

68	65	6C	6C	6F	2C	77	6F	72	6C	64	20	20	20	20	20
'h'	'e'	'l'	'l'	'o'	','	'w'	'o'	'r'	'l'	'd'	' '	' '	' '	' '	' '

・パディング(2)　　…Ethernet フレーム最低 60 バイトのためパディングデータ追加

00	00

(*1)IP ヘッダ内の ID は、「本プログラムでは 0xABCD に固定」しています

(*2)IP ヘッダ内のチェックサムに関しては、5 章(5.6)で計算方法を示しています。

(*3)UDP 内のチェックサムの計算方法に関しては、以下の様になります。

・疑似ヘッダ（12 バイト）

UDP ヘッダとは別に、疑似ヘッダを生成します。

フィールド	bit 数	内容	データ例
Source IP Address	32	送信元 IP アドレス	C0 A8 00 50
Distination IP Address	32	送信先 IP アドレス	C0 A8 00 51
Zero padding	8	パディング(0x00)	00
Protocol	8	プロトコル（UDP の場合は 0x11）	11
Length	16	UDP セグメントの長さ（ヘッダ＋データ）[バイト]	00 18　（24 バイト）

12 バイトの疑似ヘッダを 2 バイト単位で区切ります。

C0A8	0050	C0A8	0051	0011	0018

・数値を単純に加算していく

C0A8	+	0050	=	C0F8
C0F8	+	C0A8	=	181A0
81A1	+	0051	=	81F2
81F2	+	0011	=	8203
8203	+	0018	=	821B

→81A1（bit16 を 0 にして、1 を加算する）

・0xFFFF を超えた場合は、0xFFFF を超えた部分の最上位桁は消して、1 を加算する(0x181A0-0x10000+0x1)
→1 の補数和, one_complement_sum()

・UDP ヘッダ（8 バイト）

チェックサム対象のデータを 2 バイト毎に区切る

C350	7330	0018	0000

（チェックサムを計算する場合はチェックサムのフィールドは 0x0000 で計算）

・疑似ヘッダの加算結果に数値を単純に加算していく

821B	+	C350	=	1456B
456C	+	7530	=	BA9C
BA9C	+	0018	=	BAB4
BAB4	+	0000	=	BAB4

→456C

チェックサムは 0x0000 で計算

・UDP データ（今回は 16 バイト）

チェックサム対象のデータを 2 バイト毎に区切る

6865	6C6C	6F2C	776F	726C	6420	2020	2020

・疑似ヘッダ＋UDP ヘッダの加算結果に数値を単純に加算していく

BAB4	+	6865	=	12319	→231A
231A	+	6C6C	=	8F86	
8F86	+	6F2C	=	FEB2	
FEB2	+	776F	=	17621	→7622
7622	+	726C	=	E88E	
E88E	+	6420	=	14CAE	→4CAF
4CAF	+	2020	=	6CCF	
6CCF	+	2020	=	8CEF	
~8CEF			=	7310	

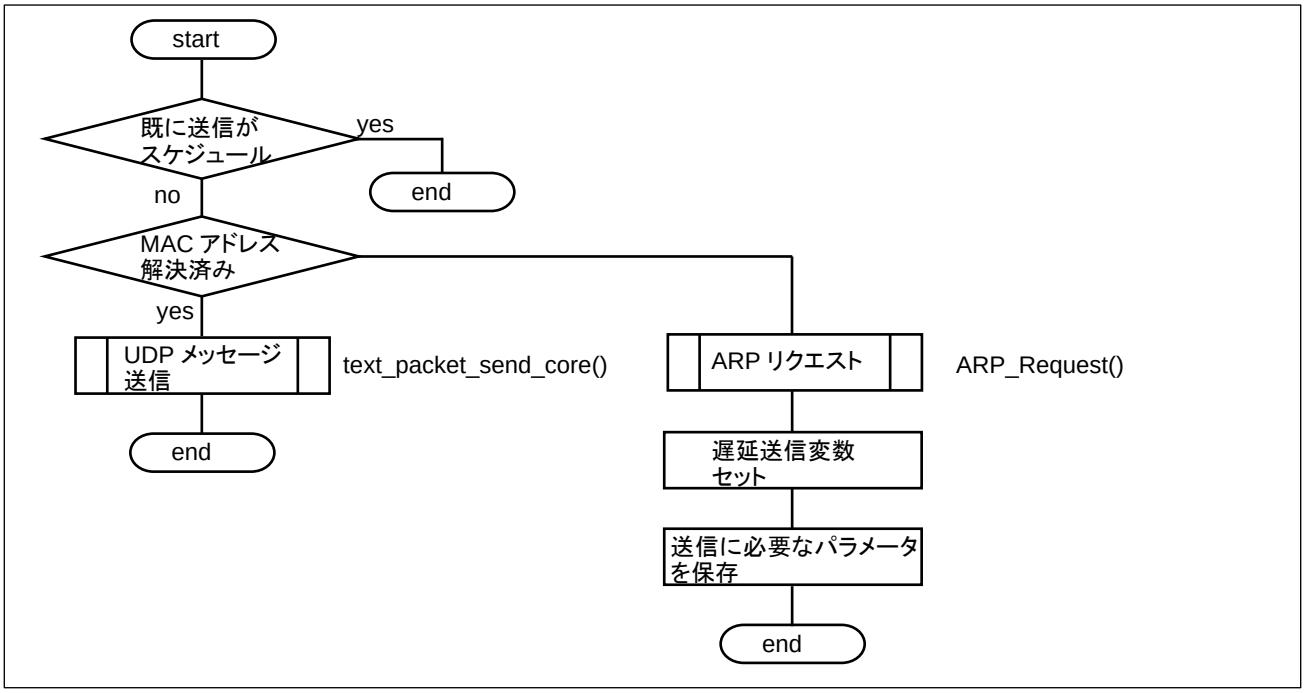
UDP ヘッダ内に埋め込まれるチェックサムの値は、0x7310 となります。

IP ヘッダ内のチェックサムは単純に 1 の補数和の反転となりますが、UDP のチェックサムは疑似ヘッダを計算に含める点が異なります。

後は、Ethernet ヘッダ＋IP ヘッダ＋UDP ヘッダ＋UDP データをバイト列に格納して、API 関数(R_ETHER_Write())で送信するという形です。

text_packet_send()関数では、MAC アドレス解決済みの場合は直ぐにメッセージを送信。そうでない場合は、本関数内では ARP リクエストを実行して、メッセージの送信は後で送信する様にスケジュールの設定を行って関数を抜けます。

・text_packet_send() （text_client.c 内）



ARP リクエストの結果回収は、メインループ内の受信関数(receive_data_process())内で行われます。
スケジュールされたメッセージの送信は、100ms 毎の割り込み関数内で実行されます。

・ether_interrupt.c

```
void ether_interrupt_cmt0(void)
{
    //100ms割り込み関数

    extern unsigned short g_sw_read_counter;

    //スイッチの読み取りを一定時間経過後に有効にする処理
    if (g_sw_read_counter != 0)g_sw_read_counter--;

    //UDPメッセージ送信の実行がスケジュールされている場合は実行する
    text_packet_send_schedule();
}
```

ether_interrupt_cmt0 は、コンペアマッチタイマ(CMT0)で 100ms 毎に実行される関数です。

text_packet_send_schedule()は、遅延送信がスケジュールされているかを確認し、スケジュールされていて、かつ MAC アドレスの解決ができていれば、UDP メッセージを送信します。

・text_client.c

```
void text_packet_send_schedule(void)
{
    //UDPテキストメッセージ送信関数（遅延実行）

    //戻り値
    //  なし

    //引数
    //  なし

    //説明
    //  text_packet_send()関数が呼ばれたがIPアドレスとMACアドレスの解決ができていない場合、後から本関数を
呼んでメッセージの送信を処理する

    //補足
    //  タイマ割り込みで呼ばれる関数で、外部から呼ばれる関数なので static 指定はしていないがユーザ関数では
ない

    if (g_delay_execution_count == 0) return;//遅延実行がスケジュールされていない

    //メッセージ送信先のMACアドレスをチェック
    if ((g_dst_MAC[0] == 0) && (g_dst_MAC[1] == 0) && (g_dst_MAC[2] == 0))
    {
        //MACアドレスが解決していない

        //送信試行回数を減らす
        g_delay_execution_count--;

        if (g_delay_execution_count == 0)
        {
        //最大試行回数まで達したが、MACアドレスの解決ができていない
        sci_write_str("¥ntext_packet_send : Retry count over error.¥n");
        }

        return;
    }

    //宛先MACアドレスが判明している場合はメッセージ送信を行う
    text_packet_send_core(g_delay_execution_dst_IP,g_delay_execution_dst_port,
g_delay_execution_text_data);

    //これ以降の送信試行はキャンセル
    g_delay_execution_count = 0;
}
```

text_packet_send()関数内で、g_delay_execution_count は 20 がセットされますので、100ms 毎に最大 20 回送信を試みます。100ms×20（2 秒）以内に、MAC アドレスの解決（ARP リプライの受信）ができない場合は、送信を諦めます。

UDP の場合、単純にデータを送信するだけですが、MAC アドレスの解決ができている場合（即時送信）と、できていない場合（遅延送信をスケジュール）で処理を分けています。

なお、本プロジェクトでは、1 回 ARP リプライを受信して MAC アドレスの解決ができた場合、永続的に MAC アドレスの値を記憶します。

PC などでは、MAC アドレスの解決ができた場合、一時的に MAC アドレス値を記憶し、数分程度でデータを破棄して再度 ARP リクエストを発行するのが一般的です。（IP アドレスと MAC アドレスの関係は動的なものとして管理されています。）

7.3. UDP の受信処理

RX65N_PING プロジェクトでは、Ethernet ヘッダ内の Ethernet タイプや IP ヘッダ内のプロトコルのフィールドを見て、ARP のデータなのか ping のデータなのかを判定して、処理を分けていました。

本プロジェクトにおいても、IP ヘッダ内のプロトコルフィールドが 0x11(17:UDP)である場合の条件分けを追加しているだけで、本質的には RX65N_PING プロジェクトの受信処理と異なる訳ではありません。

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が、何か特別に待ち受けプログラムを走らせている訳でもありません。

メッセージを受信した場合

```
ARP reply send (this board MAC address = 00-0D-76-00-40-01) to 00-0D-76-00-40-02
UDP text data received -> "hello,world          " from 192.168.0.81:50000
```

赤字の部分は、相手から ARP リクエストが来たときのみ表示されます。
（通信相手がマイコンボードの場合は、初回通信の 1 回だけ。通信相手が PC の場合は、初回通信の時以外でも定期的に表示されます。）

・main.c,（receive_data_process() 抜粋）

```
void reveive_data_process(void)
{
    //受信データを処理する関数

    unsigned char *addr, *addr2, *addr3;
    unsigned short flag;
    int ret;

    unsigned char text_message[UDP_MESSAGE_SIZE];
    unsigned short from_port;

    unsigned short i;

    ret = ether_recv_data_access(&addr, &flag);//受信リングバッファに格納されているデータを参照

    if (ret == ETHER_NO_DATA) return;//受信データなし

    ETHERNET_HEADER ethernet_packet;

    stream_to_ETHERNET_header(addr, &ethernet_packet);//受信データのEthernetヘッダを分解

    addr2 = addr + 14;//Ethernetヘッダのオフセット（14バイト目以降がEthernetヘッダを除いたデータ）

    switch (ethernet_packet.eth_ethernet_type)//Ethernetタイプで処理を分ける
    {
（中略）
        case 0x0800: //IPパケットの時
        {
            IP_HEADER ip_packet;

            addr3 = addr2 + 20;//IPデータのオフセット（IPヘッダが20バイト）

            stream_to_IP_header(addr2, &ip_packet);

            if (IP_compare(ip_packet.ip_dst_IP, g_src_IP) == 0)//宛先IPアドレスが本ボードのIPアドレスに
一致
            {
                switch (ip_packet.ip_protocol)
                {
（中略）
                    case 17://UDPパケットの時
                        ret = text_packet_receive(addr3, g_server_dst_port, &from_port,
text_message);
                        if (ret != 0)
                        {
                            sci_write_str("UDP text data received -> ¥"");
                            for (i=0; i<ret; i++) sci_write_char(text_message[i]);
                            sci_write_str("¥" from ");
                            sci_print_IP(ip_packet.ip_src_IP);
                            sci_write_str(":");
                            sci_write_uint16(from_port);
                            sci_write_str("¥n");
                        }
                        break;
                }
            }
        }
        break;
    }

    //受信データの処理が終わったので読み出しのキューを進める
    ether_recv_data_dequeue();
}
```

受信した Ethernet フレームが、IP パケット、かつ UDP パケットの際に text_packet_receive()関数にデータを渡しています。

・text_server.c

```
int text_packet_receive(unsigned char *udp_stream, unsigned short port, unsigned short *from_port,
unsigned char *text_data)
{
    //UDPテキストメッセージ受信関数

    //戻り値
    //  0 : メッセージなし
    //  >0 : 戻り値のバイト数のデータを受信（本プログラムでは最大16バイトに制限）

    //引数
    //  *udp_stream : 受信したUDPデータフレーム
    //  port : データ待ち受けポート
    //  *from_port: 送信元ポート（戻り値）
    //  *text_data : 受信したデータ（戻り値）※16バイト以上のunsigned char配列を与える

    int i;

    UDP_HEADER udp_header;

    stream_to_UDP_header(udp_stream, &udp_header);

    if (udp_header.udp_dst_port == port)
    {
        //対象ポートに対するデータ

        *from_port = udp_header.udp_src_port;//送信元ポート

        for (i=0; i<udp_header.udp_length; i++)
        {
            if (i >= 16) break;//本プログラムでは最大16バイトのテキストメッセージを受け取る
            *text_data++ = udp_stream[8 + i];
        }
        return i;
    }

    return 0;
}
```

text_packet_receive()関数では、UDP のパケットを受け取り、宛先ポート番号が受信に設定したポート番号（デフォルト 30000）に一致した場合、送信元ポート番号と UDP データを変数に格納してリターンします。

本ボードの、IP アドレスが 192.168.0.80、MAC アドレスが 00-0D-76-00-40-01 の場合、ネットワークインタフェースに到達したデータにより

MAC アドレス	IP アドレス	プロトコル	宛先 ポート番号	取り扱い
01-23-45-67-89-AB	任意	任意	任意	データを受信しない （受信割り込みまで行かない） (*)
00-0D-76-00-40-01	192.168.0.1	任意	任意	IP アドレスが一致していないので receive_data_process()内で受信データとして処理しない
00-0D-76-00-40-01	192.168.0.80	0x11(17)	10000	text_packet_receive()内で受信データとして 処理しない
00-0D-76-00-40-01	192.168.0.80	0x11(17)	30000	テキストメッセージを画面に表示

MAC アドレスが一致しない場合は、マイコンの Ethernet コントローラがデータを受信しません（受信割り込みには飛んできません）。MAC アドレスが一致（もしくはブロードキャスト）の場合は、受信割り込みに来るので receive_data_process()でデータが処理されます。

MAC, IP アドレスが一致して、UDP のパケットで宛先ポート番号が一致しない場合は、text_packet_receive()内で受信データとして取り扱われません。ポート番号まで一致した場合、受信メッセージが画面に表示されます。

(*)MAC アドレスが不一致な場合ですが、一般的にはスイッチングハブ（Layer2 スイッチ）のレベルで、本ボードにはパケットを送ってこないという動作となります。

7.4. ユーザ作成関数

text_packet_send

概要:UDP データの送信
宣言:int text_packet_send(unsigned char *dst_IP, unsigned short dst_port, unsigned char *text_data)
説明:
・UDP でのデータ送信
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
　unsigned char *text_data: 送信データ
戻り値:
　0: 正常終了（データを送信）
　1: MAC アドレスの解決ができていないので送信待ちに入った
　-13: 前回の送信待ちが解消されていない（未送信で待機状態にある）
補足:本関数では送信データは 16 バイト固定です
　　本関数呼び出し時点で、MAC アドレスの解決ができていない場合は、*dst_IP に対して ARP リクエストを送り
　　実行に必要な引数をグローバル変数に保存します（その場合は、定期的に text_packet_send_schedule()
　　関数を呼び出す事で、MAC アドレス解決後にデータが送信されます）

text_packet_receive

概要:UDP データの受信
宣言:int text_packet_receive(unsigned char *udp_stream, unsigned short port, unsigned short *from_port, unsigned char *text_data)
説明:
・受信した UDP パケットからデータの抽出
を行います
引数:
　unsigned char *udp_stream: 受信した UDP のストリームデータ
　unsigned short port: サーバ待ち受けポート
　unsigned short *from_port: 送信元ポート（戻り値）
　unsigned char *text_data: 受信データ
戻り値:
　0: 受信データなし
　>0: 受信バイト数

補足：本関数では受信データは最大 16 バイトです、*text_data は 16 バイト以上の領域を確保して下さい
　　port に指定した引数と受信したデータの宛先ポートが一致しない場合は受信データなしとなります

8. RX65N_TCP プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_TCP
SOURCE¥176PIN¥RX65N_TCP

（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、
・マイコンボードから TCP サーバに接続を行う（クライアント動作）
・接続済みのサーバに対してデータ（テキストメッセージ）を送信する
・他の機器からの TCP の接続要求を受け付ける（サーバ動作）
・接続済みの他の機器からマイコンボードに対して飛んできたデータ（テキストメッセージ）を表示する
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
Config_CMT0	コンペアマッチタイマ（100ms 毎）
Config_CMT1	**コンペアマッチタイマ（1ms 毎）**

（→RX65N_UDP プロジェクトに対し、CMT1 が追加）

・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
arp	arp.c	ARP のプロトコルを取り扱う関数
	arp.h	上記ヘッダ
client	text_client.c	TCP のテキストメッセージの送信プログラム（クライアント）
	text_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
ethernet	ethernet.c	イーサネットヘッダを取り扱う関数
	ethernet.h	上記ヘッダ
interrupt	ether_interrupt.c	割り込み関数
	ether_interrupt.h	上記ヘッダ
ip	ip.c	IP(InternetProtocol)を扱う関数
	ip.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	text_server.c	UDP のテキストメッセージの待ち受けプログラム（サーバ）
	text_server.h	上記ヘッダ

・ユーザ作成のプログラムコード（続き）

フォルダ（カテゴリ）	ファイル名	備考
tcp	**tcp.c**	TCP を取り扱う関数
	tcp.h	上記ヘッダ

太字のフォルダ、ファイルが新規に追加されたものです。本プロジェクトでは、udp.c, udp.h の代わりに tcp.c, tcp.h が追加となっています。

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 TCP text client/server sample program.

COMMAND:
   t : send text message set
   r : response text message set
   p : send/response text message print
   s : send text message
   c : connect tcp session
   d : disconnect tcp session
   D : force disconnect tcp session

-setting address-
---
communication target IP address -> 192.168.0.81
this board IP address                -> 192.168.0.80 (MAC address -> 00-0D-76-00-40-01)
---
-setting port-
---
TCP server listen port -> 20000
TCP client dest     port -> 20000
TCP client source port -> 50000
---

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

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

TCP クライアントで送信元として使用するポート　50000

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/MAC address settings change --

COMMAND:
   1 : communication target IP address
   2 : this board IP address
   3 : this board MAC address
   4 : TCP server listen port (default = 20000)
   5 : TCP client dest port     (default = 20000)
   6 : TCP client source port (default = 50000)
   p : print setting
   e : exit setting
```

1~6 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。

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!
Ether0: LINK-UP
```

・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。

IP アドレスや MAC アドレレス、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。

8.1. 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
・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
・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
・Ether の初期化
・コールバック関数の登録
・Ether のスタンバイ解除
・端子設定
・Ether の動作開始
・サーバの待ち受け設定
・メインループ

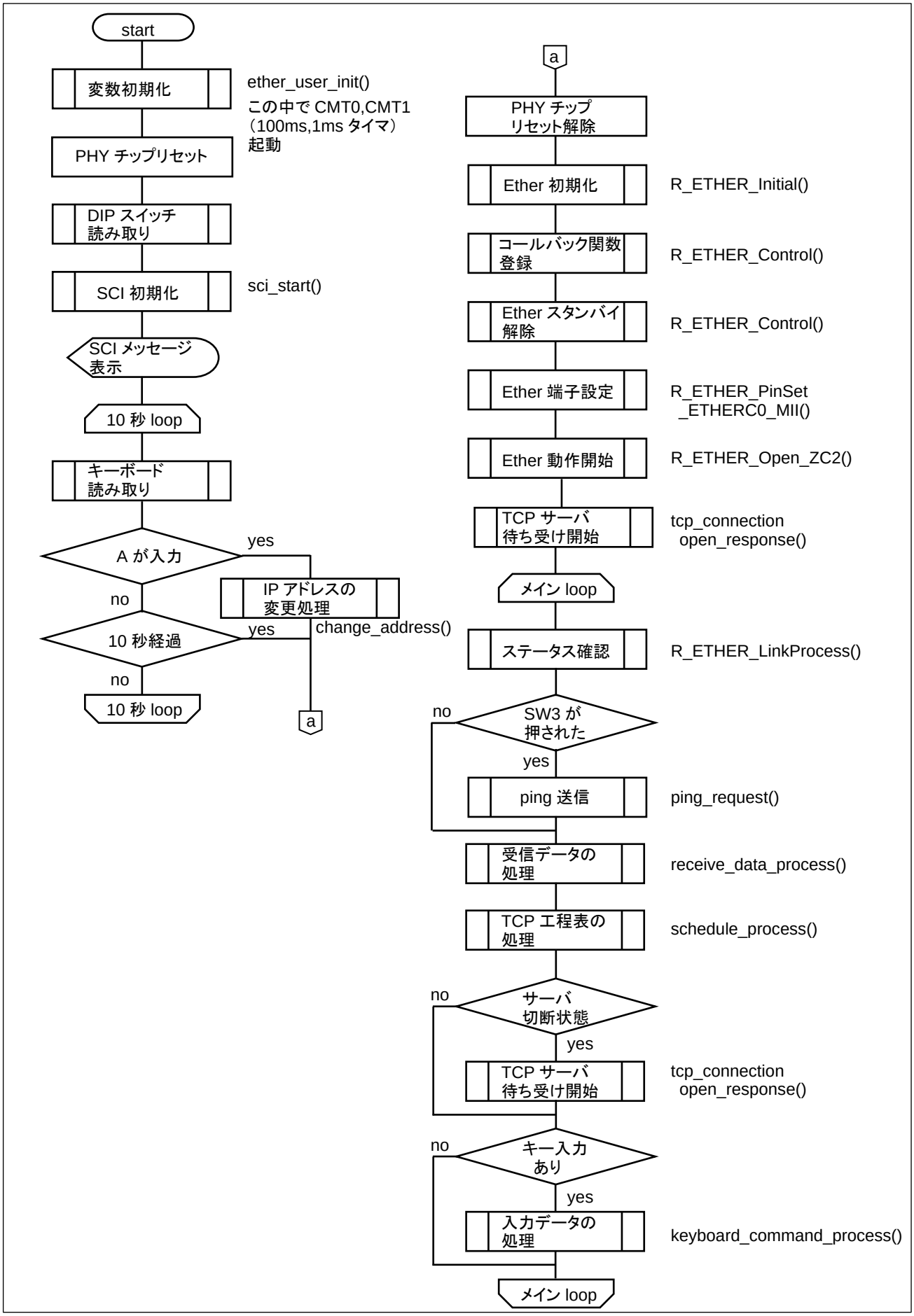
の処理を行っています。

メインループでは、

・API 関数のステータスチェック
・SW3 が押されている場合は ping 送信
・受信データの処理
・TCP 工程表の処理
・キーボードからのコマンドの処理

となっています。

・ether_main() （main.c 内）



本プロジェクトでは、「TCP サーバの待ち受け開始」「TCP 工程表の処理」が RX65N_UDP プロジェクトに対して追加されています。

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。（RX65N_UDP プロジェクトと同じです）

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、c を入力すると TCP での接続を確立します。TCP 接続確立後に s を入力すると、TCP でメッセージを送信します。
SW3 は、通信相手に設定した IP アドレスに ping を送信します。

8.2. 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_start: 192.168.0.81 MAC address?
ARP request (192.168.0.81-> MAC address?) send
ARP reply received from 192.168.0.81, MAC address = 00-0D-76-00-40-02
TCP flag packet SYN send to 192.168.0.81:20000
TCP handshake ACK & SYN flag received OK.
TCP flag packet ACK send to 192.168.0.81:20000
-- CONNECTION ESTABLISHED --
```

赤字の部分は ARP リクエストに対して ARP リプライを受信している部分です。
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
send text message to 192.168.0.81 port 20000
TCP data packet (40 bytes) (flag = ACK PSH ) send to 192.168.0.81:20000
TCP frame received from 192.168.0.81:20000 Flag = ACK PSH , 40 bytes data
>OK(RX65N)
TCP flag packet ACK send to 192.168.0.81:20000
```

OK(RX65N)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,world RX65N TCP message "です。t 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 flag packet ACK FIN send to 192.168.0.81:20000
TCP handshake ACK flag received OK.
TCP handshake ACK & FIN flag received OK.
TCP flag packet ACK send to 192.168.0.81:20000
-- 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。

8.3. 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

```
ARP reply send (this board MAC address = 00-0D-76-00-40-01) to 00-0D-76-00-40-02
tcp_connection_start: 192.168.0.81:50000
TCP handshake SYN flag received OK.
TCP flag packet ACK SYN send to 192.168.0.81:50000
TCP handshake ACK flag received OK.
-- CONNECTION ESTABLISHED --
```

赤字の部分は接続相手の ARP リクエストに対して ARP リプライを送信している部分です。
CONNECTION ESTABLISHED と表示されれば、TCP での接続は成功しています。

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

```
TCP frame received from 192.168.0.81:50000 Flag = ACK PSH , 40 bytes data
>hello,world RX65N TCP message
TCP data packet (40 bytes) (flag = ACK PSH ) send to 192.168.0.81:50000
TCP frame received from 192.168.0.81:50000 Flag = ACK
```

"hello,world RX65N TCP message "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(RX65N)"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 -> hello,world RX65N TCP message
response text message is -> OK(RX65N)
```

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 frame received from 192.168.0.81:50000 Flag = ACK FIN
tcp_connection_idle: message: FIN flag received.
TCP flag packet ACK send to 192.168.0.81:50000
TCP flag packet ACK FIN send to 192.168.0.81:50000
TCP handshake ACK flag received OK.
-- CONNECTION CLOSED --
```

何行かのメッセージの表示と、CONNECTION CLOSED 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。）

8.4. TCP 受信処理

・main.c,（receive_data_process() 抜粋）

```
void reveive_data_process(void)
{
    //受信データを処理する関数

    unsigned char *addr, *addr2, *addr3;
    unsigned short flag;
    int ret;

    unsigned short i;

    ret = ether_recv_data_access(&addr, &flag);//受信リングバッファに格納されているデータを参照

    if (ret == ETHER_NO_DATA) return;//受信データなし

    ETHERNET_HEADER ethernet_packet;

    stream_to_ETHERNET_header(addr, &ethernet_packet);//受信データのEthernetヘッダを分解

    addr2 = addr + 14;//Ethernetヘッダのオフセット（14バイト目以降がEthernetヘッダを除いたデータ）

    switch (ethernet_packet.eth_ethernet_type)//Ethernetタイプで処理を分ける
    {
（中略）
        case 0x0800: //IPパケットの時
        {
            IP_HEADER ip_packet;

            addr3 = addr2 + 20;//IPデータのオフセット（IPヘッダが20バイト）

            stream_to_IP_header(addr2, &ip_packet);

            if (IP_compare(ip_packet.ip_dst_IP, g_src_IP) == 0)//宛先IPアドレスが本ボードのIPアドレスに一致
            {
                switch (ip_packet.ip_protocol)
                {
（中略）
                    case 6://TCPパケットの時
                    {
                        TCP_HEADER tcp_header;

                        stream_to_TCP_header(addr3, &tcp_header);

                        //TCPクライアント接続時の処理
                        if ((g_tcp_client.connect != TCP_DISCONNECT) && (tcp_header.tcp_dst_port ==
g_tcp_client.src_port))
                        {
                            tcp_recv_data_enqueue(&addr, ret, g_tcp_client.session_serial);
//クライアント時は、Ethernetヘッダ(14)，IPヘッダ(20)を除いたデータしか使わないが全データを保存しておく
                        }

                        //TCPサーバ接続時の処理
                        if ((g_tcp_server.connect != TCP_DISCONNECT) && (tcp_header.tcp_dst_port ==
g_tcp_server.src_port))
                        {
                            tcp_recv_data_enqueue(&addr, ret, g_tcp_server.session_serial);
//サーバー時はEthernetヘッダ(14)，IPヘッダ(20)の情報も使用する
                        }
                    }
                    break;
（中略）
    //受信データの処理が終わったので読み出しのキューを進める
    ether_recv_data_dequeue();
}
```

受信した Ethernet フレームが、

・IP パケット

・宛先 IP アドレスが本ボードのアドレスに一致

・TCP パケット

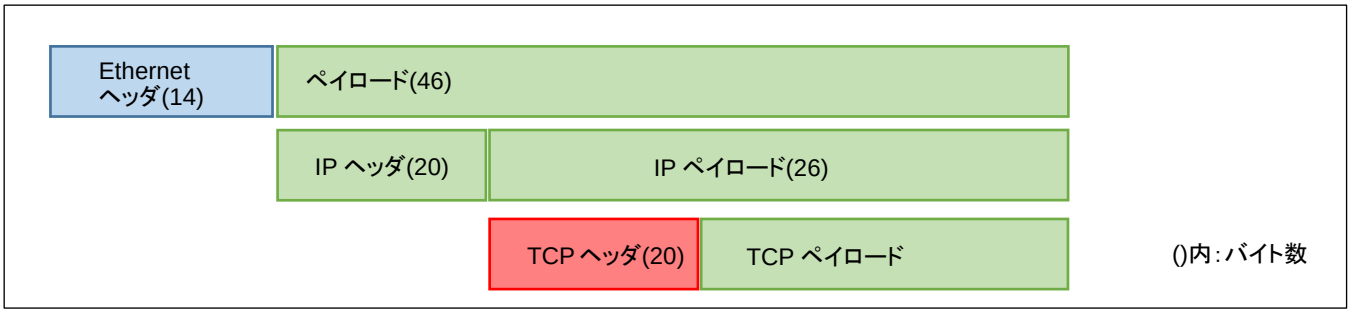
・状態が未接続(TCP_DISCONNECT)ではない

・宛先ポートが使用しているポート番号に一致

上記全ての条件が満たされる場合に、tcp_recv_data_enqueue()で受信データを保存する処理を行います。

保存したデータは、メイン関数のメインループ内の schedue_process()関数内で使用されます。

8.5. TCP ヘッダ



・Ethernet ヘッダ(14)

00	0D	76	00	40	02	00	0D	76	00	40	01	08	00
宛先 MAC アドレス						送信元 MAC アドレス						Ethernet type (IPv4)	

・IP ヘッダ(20)

45	00	00	28	00	00	40	00	80	06	78	CE	C0	A8	00	50
IPv4 ヘッダ 20B	Service type	Total Length 40 バイト		ID		Fragment		TTL	Protocol TCP	Checksum		送信元 IP アドレス (192.168.0.80)			

C0	A8	00	51
送信先 IP アドレス (192.168.0.81)			

・TCP ヘッダ(20)

C3	50	4E	20	00	00	23	18	00	00	00	00	50	02	05	B4
Source port (50000)		Destination Port (20000)		Sequence Number				Acknowledgment Number				Headersize + Flags		Window 1460	

F3	A3	00	00
Checksum		Urgent Pointer	

・TCP ペイロード

→パディング(6)　　…Ethernet フレーム最低 60 バイトのためパディングデータ追加

00	00	00	00	00	00

TCP ヘッダは 20 バイト（20 バイトかそれ以上のサイズ）で、

・送信元ポート(Source port)

・宛先ポート(Destination port)

・シーケンス番号(Sequence Number)

・応答確認番号(Acknowledgement Number)

・ヘッダサイズとフラグ(Headersize + Flags)

・ウィンドウサイズ(Window)

・チェックサム(Checksum)

・緊急ポインタ(Urgent Pointer)
から構成されています。

（ヘッダサイズ＋フラグが 0x5xxx→上位 4 ビットが 0x5 の時は、ヘッダサイズが 0x5×4=20 バイトです。本プロジェクトでは、マイコンボード側で生成するヘッダサイズは常に 20 バイトです。）

このうち、シーケンス番号と応答確認番号は、決められたルールで値を管理する必要があります。

8.5.1. ポート番号

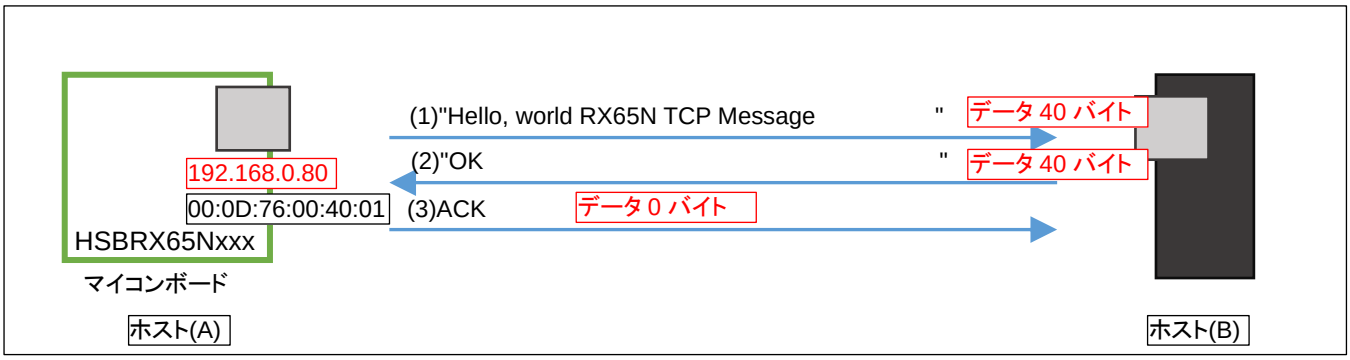
TCP 通信では、IP アドレスの他にポート番号を使用して交通整理を行います。一般的にサーバ（待ち受け）る側のポート番号は固定で使用します。サーバに接続する側のポート番号は、PC アプリなどでは動的に（ランダムに）決定されるのが普通です。

本プロジェクトでは、待ち受け側はデフォルトで 20000 番。接続する側は、50000 番を使う設定です。

8.5.2. シーケンス番号と応答確認番号

Ethernet の通信では、送信した Ethernet フレームが決められた時間以内に到達する事や、送信した順番通りに到達する事は保証されていません。TCP の通信では、フレームが欠落したり、順番が入れ替わった場合でも正しくデータを構成できる様に、パケット毎に所定の番号（4 バイトの数値）を埋め込んで送信する決まりになっています。

シーケンス番号は、初回の通信では通常ランダムな値とします。その後は、送信したデータのバイト数を加算した値を送ります。
応答確認番号は、相手が送ってきたシーケンス番号に受信したデータのバイト数を加算した値を送ります。



ここで、ホスト(A)からホスト(B)に対して 40 バイトのデータ送信。ホスト(B)からホスト(A)に対して、40 バイトのデータを送信。ホスト(A)からホスト(B)に対して ACK（データは空）を送った場合を考えてみます。

	(1)で送信	(2)で送信	(3)で送信
シーケンス番号	00001234	0000ABCD	0000125C
応答確認番号	0000ABCD	0000125C →受信したシーケンス番号+ 受信データバイト数(0x28)	0000ABF5 →受信したシーケンス番号+ 受信データバイト数(0x28)

データ送信処理開始時点の、シーケンス番号と応答確認番号の値は 0x00001234 と 0x0000ABCD と仮定します。

ホスト(A)とホスト(B)は、それぞれシーケンス番号と応答確認番号を管理しています。

・ホスト(A)側の処理

	送信する シーケンス番号	送信する 応答確認番号
処理開始時点	00001234	0000ABCD
(1)で 40 バイトのデータを送信	00001234	0000ABCD
データ送信後	0000125C (+0x28)	0000ABCD
(2)で 40 バイトのデータを受信	0000125C	0000ABCD
データ受信後	0000125C	0000ABF5 (+0x28)
(3)で ACK を送信	0000125C	0000ABF5
ACK 送信後	0000125C	0000ABF5

(1)で送信するのは、0x00001234 と 0x0000ABCD です。ここで、40 バイトのデータを送っているので、シーケンス番号に 40(=0x28)を加算します。次の送信で使用するシーケンス番号は、0x0000125C となります。

(2)で相手（ホスト(B)）からのデータを受信しますが、この時点ではシーケンス番号 0x0000ABCD、応答確認番号 0x0000125C が送られてくるのが期待値です。（ホスト(A)とホスト(B)のシーケンス番号と応答確認は逆となります。）

この時、40 バイトのデータを受信したので、次に送る応答確認番号は、0000ABCD+0x28 = 0x0000ABF5 となります。

(3)では、シーケンス番号として 0x0000125C、応答確認番号として 0x0000ABF5 を送ります。

ACK フラグの送信（送信データバイト数=0）の場合は、シーケンス番号、応答確認番号の増加はありません。

例外として、接続時や切断時のフラグ送信(Syn, Fin)の際は、応答確認番号が+1 されるルールがあります。

－応答確認番号に関して－

・ホスト(A)側の処理

	送信する シーケンス番号	送信する 応答確認番号
処理開始時点	00001234	0000ABCD
(1)で 40 バイトのデータを送信	00001234	0000ABCD
データ送信後	0000125C　(+0x28)	0000ABCD
(2)で 40 バイトのデータを受信	0000125C	0000ABCD(*1)
データ受信後	0000125C	**0000ABCD** + 0x28 ↓ 0000ABF5
(3)で ACK を送信	0000125C	0000ABF5
ACK 送信後	0000125C	0000ABF5

上記表で赤字の部分の 0000ABCD ですが、どこからやってくるのか。

・元々ホスト(A)側で保持していた応答確認番号(*1)

・(2)で受信したパケットに埋め込まれていたホスト(B)が送ってきたシーケンス番号

後者のデータを使うのが正解だと思います。（パケットの順番が入れ替わらない限り、前者と後者は同じ値ですが）

(3)で送信する ACK は、(2)に対する応答です。(2)のパケットを正しく受け取った旨の応答確認番号ですので、ベースになるのは(2)の TCP ヘッダ内のシーケンス番号であるべきだと思われます。

プログラム上は、以下の様に設定しています。

シーケンス番号

→自分自身で管理している値、データ送信毎に、送信バイト数を加算していく

応答確認番号

→相手が送ってきたシーケンス番号＋データバイト数とする

8.5.3. TCP フラグ

ヘッダサイズ＋フラグは、0x5002=0b0101 0000 0000 0010 の場合、以下の様になります。

0b**0101** 0000 0000 0010	b15-b12:ヘッダサイズ	5×4=20 バイト
0b0101 **000**0 0000 0010	b11-b9:予約	
0b0101 000**0** 0000 0010	b8:Accurate ECN	
0b0101 0000 **0**000 0010	b7:CongestionWindowReduced	
0b0101 0000 0**0**00 0010	b6:ECN-Echo	
0b0101 0000 00**0**0 0010	b5:Urgent	
0b0101 0000 000**0** 0010	b4:Acknowledgment	応答
0b0101 0000 0000 **0**010	b3:Push	データを送信（バッファリングせずアプリに送る）
0b0101 0000 0000 0**0**10	b2:Reset	強制切断
0b0101 0000 0000 00**1**0	b1:Syn	接続の開始
0b0101 0000 0000 001**0**	b0:Fin	接続の終了

このフラグの状態により、接続の要求なのか、データの送信なのか、切断の要求なのか等の情報をやり取りします。

b0: Fin　切断要求
b1: Syn　接続要求
b2: Reset　異常時の強制切断
b3: Push　データの送信
b4: ACK　受け取った旨の応答
のフラグに関して、以下の処理で使用しています。

Syn フラグを立てて送信→TCP フラグのフィールドに、0x5002 を埋め込んで送信
Syn &ACK フラグを立てて送信→TCP フラグのフィールドに、0x5012 を埋め込んで送信
といったデータとなります。

8.5.4. ウィンドウサイズ

1 回のやり取りで送信されるデータの最大サイズです。

マイコンボード側では、1460 バイトに固定しています。
（Ethernet フレーム – Ethernet ヘッダ – IP ヘッダ – TCP ヘッダ
1514 – 14 – 20 – 20 = 1460）

8.5.5. チェックサム

UDP の際のチェックサムの計算方法（疑似ヘッダを用いた計算）と同じアルゴリズムで計算した値です。

7.2 節を参照してください。

8.5.6. 緊急ポインタ

本プロジェクトでは、常に 0x0000 です。

8.6. TCP 構造体

TCP では接続を確立する過程や接続確立中、切断する過程における、一連のセッションで TCP の接続に必要な情報を（シーケンス番号や応答確認番号）構造体で管理しています。

・tcp.h

```
typedef struct
{
    //TCP接続状態
    unsigned short connect;//0:未接続，1:接続試行中，2:接続中
    unsigned short sequence;//工程表の何処にいるか
    unsigned long seq_no;//シーケンシャル番号
    unsigned long ack_no;//確認応答番号
    unsigned char src_MAC[6];//本ボードのMACアドレス
    unsigned char dst_MAC[6];//接続先のMACアドレス
    unsigned char src_IP[4];//本ボードのIPアドレス
    unsigned char dst_IP[4];//接続先のIPアドレス
    unsigned short src_port;//本ボード側の使用ポート
    unsigned short dst_port;//接続先のポート
    unsigned short id;//TCPヘッダ内のID（本プログラムでは送信毎に単純にインクリメントする）
    unsigned short session_serial;//セッション毎のシリアル番号（メイン関数に登録する関数番号）
    void (*push_action)();//データを受信した際のアクションの関数ポインタ（クライアント:何もしない（画面表
示のみ），サーバ:相手にOKを返す）
    uint64_t time;//処理が行われた時間を記録
    uint64_t timeout;//接続タイムアウト（未来の時間を設定，設定時間が来るとタイムアウト）0:タイムアウト処
理を行わない
    uint64_t wait_time;//ウェイト（未来の時間を設定，設定時間が来るまでは実行をスキップする）
}TCP_CONNECTION;
```

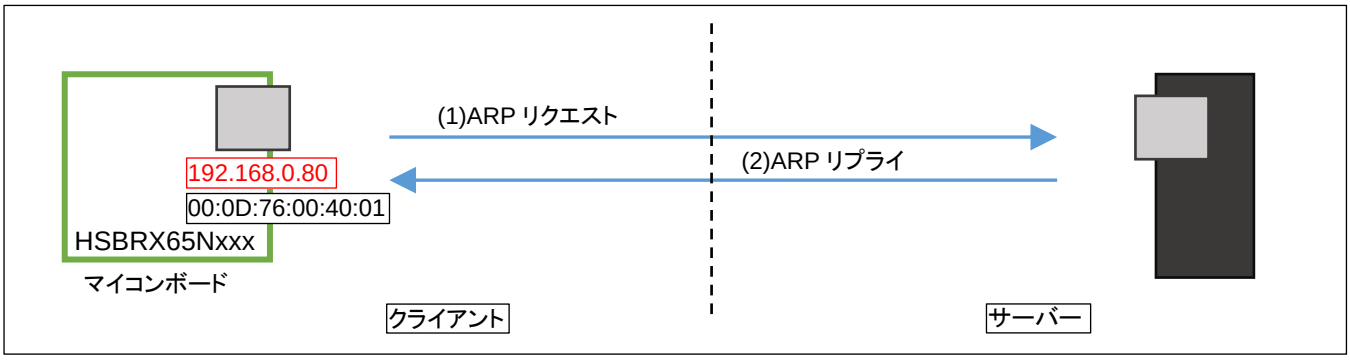
構造体のメンバは上記の様にしています。MAC アドレス、IP アドレス、ポート番号に関しては特段説明は不要かと思います。その他のメンバは、プログラム内で実際に使われている場面に応じて説明します。

8.7. TCP 工程管理

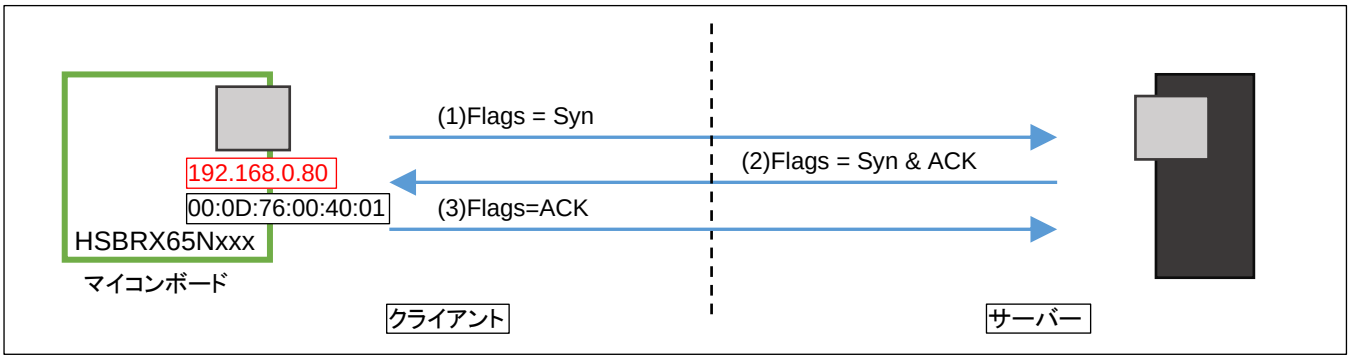
TCP では、接続時は 3 ウェイハンドシェイクと言って合計 3 回のデータのやり取りが必要です。同じく切断時は 4 ウェイハンドシェイクで 4 回のデータのやり取りとなります。

これらのデータのやり取りは、相手からの応答の後で次のパケットを送信する必要があり、動作を自分自身のタイミングで決められません。

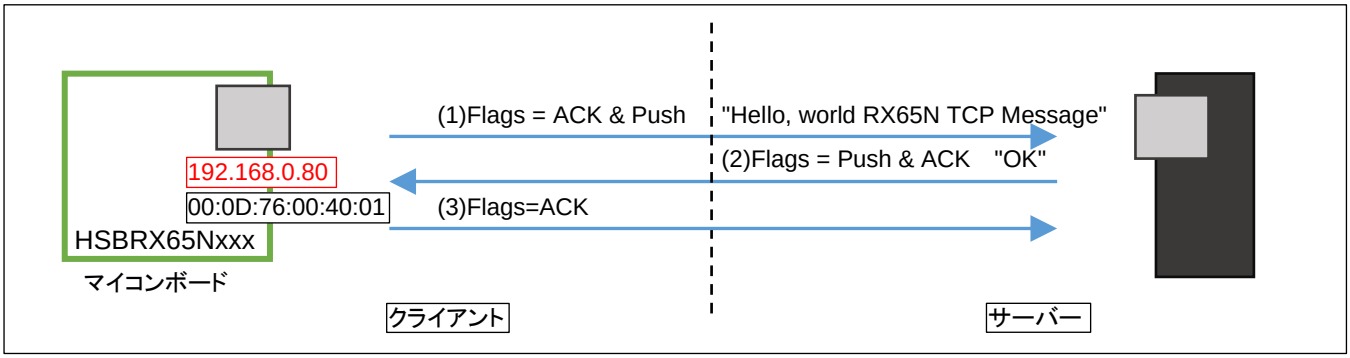
・接続前の IP アドレスーMAC アドレス解決



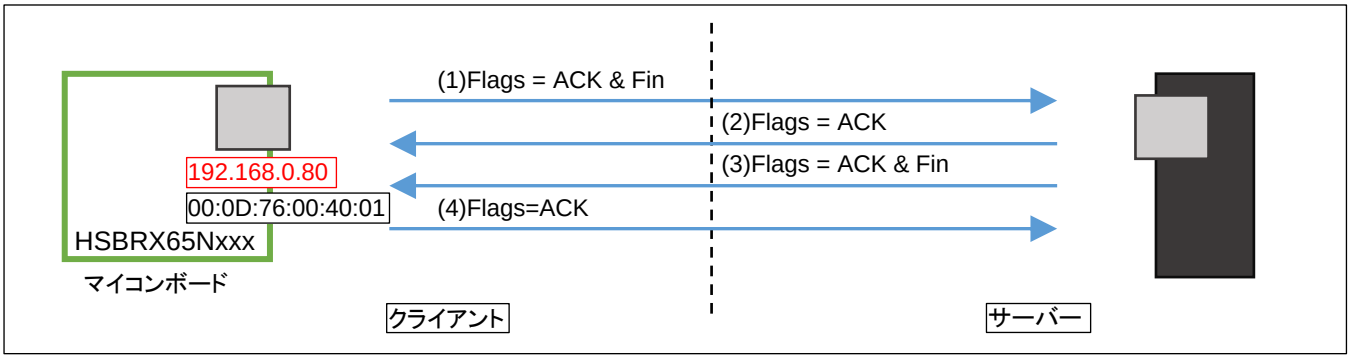
・接続確立時の通信（3 ウェイハンドシェイク）



・データ送信



・切断時の通信（4 ウェイハンドシェイク）



それぞれの通信で上記の様なパケットのやり取りが発生します。

例えばクライアント動作時、接続を確立する場合、3 ウェイハンドシェイク(1)を送信後、受信データを監視して、3 ウェイハンドシェイク(2)を受信した場合、3 ウェイハンドシェイク(3)の送信に進めます。相手からの返信を待つという過程が入る点が、プログラム上厄介な点です。自分自身のタイミングで次へ進む事ができません。

本プログラムでは、工程番号で、プログラムがどこの過程に居るかを管理します。

工程番号	内容	クライアント動作時	サーバ動作時
0	初期値	起動後 c コマンド入力前	起動後、待ち受け関数実行で 2 に移行
1	MAC アドレス解決	ARP リクエスト送信 ARP リプライ受信待ち(*)	なし
2	3 ウェイハンドシェイク(1)	パケット送信	パケット受信待ち(*)
3	3 ウェイハンドシェイク(2)	パケット受信待ち(*)	パケット送信
4	3 ウェイハンドシェイク(3)	パケット送信	パケット受信待ち(*)
5	接続が確立された状態	s コマンド入力待ち(*)	データ受信待ち(*)
6	4 ウェイハンドシェイク(1)	パケット送信	なし（5 から 7 に移行）
7	4 ウェイハンドシェイク(2)	パケット受信待ち(*)	パケット送信
8	4 ウェイハンドシェイク(3)	パケット受信待ち(*)	パケット送信
9	4 ウェイハンドシェイク(4)	パケット送信	パケット受信待ち(*)
10	終了処理	変数のクリアなど	変数のクリアなど
99	タイムアウト	タイムアウト処理	タイムアウト処理

(*)の部分は、相手からデータが来るまで待っている処理となります。

上記の様な工程番号を定義し、クライアント動作時は、2 の工程は自分主体で先へ進める（パケット送信後、工程を 3 に進める）が、3 の工程は相手からデータが来た時点で 4 の工程に進むといった様な動作としています。

TCP の工程管理用に、tcp.c 内で
TCP_CONNECTION g_tcp_client;
TCP_CONNECTION g_tcp_server;
クライアント動作時向け(g_tcp_client)とサーバ動作時向け(g_tcp_server)という構造体を定義しています。

8.7.1. クライアント接続処理

クライアント動作時は、c コマンドで TCP の接続を確立しようと動き出すのか一連の動作のスタートとなります。起動後 c コマンド入力前は、
g_tcp_client.connect = 0
の状態で、この時はクライアント向けの TCP のデータを受信しても処理しない動作です。

c コマンド入力時に、
tcp_connection_open_request_queue()
関数が実行されます。

・tcp.c

```
int tcp_connection_open_request(unsigned char *IP, unsigned short port, TCP_CONNECTION *tcp_session,
unsigned short session_serial)
{
    //TCP接続の開始を要求する関数を実行キューに登録する関数

    //戻り値
    //  0 : 正常終了（接続開始処理関数をキューに登録）
    //  -1 : 接続済み、または接続試行中

    //引数
    //  *IP : 接続先IPアドレス
    //  port : 接続先ポート
    //  tcp_session : TCP接続情報
    //  session_serial: セッションのシリアル番号

    unsigned short i;

    if (tcp_session->connect != TCP_DISCONNECT)
    {
        sci_write_str("tcp_connection_open: error: TCP connection is already open or try to open.¥n");
        return -1;
    }

    tcp_session->sequence = 1;//工程の最初からスタート

    tcp_session->connect = TCP_CONNECT_ATTEMPT;//接続試行中
    tcp_session->id = 0;//最初はID=0からスタート
    tcp_session->wait_time = 0;//初期化（ウェイトなし）
    tcp_session->time = g_timer_counter;//処理が始まった時間を記録
    tcp_session->timeout = g_timer_counter + 3000;//接続タイムアウト設定，ARPによるIP-MAC解決のタイムアウト
（3000ms）

    //MACアドレスの設定（保存，初期化）
    for (i=0; i<6; i++)
    {
        tcp_session->src_MAC[i] = g_src_MAC[i];//ether_operation.c で定義されている本ボードのMACアドレス
        tcp_session->dst_MAC[i] = 0x00;//初期化
    }

    //IPアドレスの設定（保存）
    for (i=0; i<4; i++)
    {
        tcp_session->src_IP[i] = g_src_IP[i];//ether_operation.c で定義されている本ボードのIPアドレス
        tcp_session->dst_IP[i] = IP[i];//接続先のIPアドレス
    }

    //使用ポート
    tcp_session->src_port = g_client_src_port;//ether_operation.c で定義されている使用ポート（デフォルト
50000）

    //接続先ポート
    tcp_session->dst_port = port;

    //シーケンス番号
    tcp_session->seq_no = (unsigned long)rand();//シーケンス番号はランダムに決める（0-7FFFの範囲であるかOKと
する）

    //応答確認番号
    tcp_session->ack_no = 0;//応答確認番号は最初は0にする

    //セッションのシリアル番号
    tcp_session->session_serial = session_serial;

    //PUSHフラグを受信した際のアクション
    tcp_session->push_action = NULL;//何もしない（ACKを返すだけ）

    //メイン関数で実行される関数に工程実行関数を登録
    tcp_session_function[tcp_session->session_serial] = tcp_connection_open_request_queue;

    return 0;
}
```

この関数では、g_tcp_client のメンバを適切に初期化するのが主な目的です。

tcp_session->sequence = 1
→工程表の最初のステップである IP-MAC アドレス解決の工程にセット

tcp_session->connect = TCP_CONNECT_ATTEMPT; //(=1)
→セッションの状態を接続試行中にセット（これにより、メイン関数内の受信関数でデータを受信する様になります）

tcp_session->id = 0;//最初は ID=0 からスタート
→TCP ヘッダの中に埋め込まれる ID 値です
　ID 値は本プログラムでは、送信の度に単純にインクリメントする仕様としています

tcp_session->wait_time = 0;//初期化（ウェイトなし）
→工程表を進めるにあたり次の工程に進めるウェイト時間の設定（ここではウェイトなし）

tcp_session->time = g_timer_counter;//処理が始まった時間を記録
→工程表のタイムアウトを管理するため、現時点での時間（カウンタ値）を保存
　カウンタ値は、CMT1 の割り込みで 1ms 毎にインクリメントされます
　工程表で変化があった（次の工程に進んだ）際に、この値は更新されます

tcp_session->timeout = g_timer_counter + 3000;//接続タイムアウト設定, ARP による IP-MAC 解決のタイムアウト（3000ms）
→次の工程に進むタイムアウト時間を設定
　ここでは、ARP リクエストによる IP-MAC アドレスの解決に 3000ms 以上掛かった場合、タイムアウトで処理される様設定しています

その後、IP アドレス、MAC アドレス、ポート番号の設定をします。通信相手の MAC アドレスはこの時点では判明していないので、0x00 で埋めています。

tcp_session->seq_no = (unsigned long)rand();//シーケンス番号はランダムに決める（0-7FFF の範囲であるか OK とする）
→シーケンス番号の設定
　乱数で初期値を決める事とします
　シーケンス番号は 4 バイトですが、rand()関数を使った場合は、0-7FFF の範囲となりますが OK とします
　（完全な 4 バイトの乱数とはしていません）
　また、rand()の種の設定は行っていないので、毎回同じ数から始まります
　（特にこの値がランダムである事を重視していません）

tcp_session->ack_no = 0;
→応答確認番号の設定
　応答確認番号は、最初は 0 からスタートする事になっているので 0 で初期化。

tcp_session->session_serial = session_serial;
→TCP セッションの固有番号
　本プロジェクトでは、常に session_serial = TCP_CLIENT_NUM(=0)で呼ばれます
　tcp_connection_open_request()は、クライアント向けの関数で、本プロジェクトではクライアントは 1 セッションのみ使用しています
　複数のクライアントセッションを動かす場合は、固有の値でセッションを区別する必要があります

tcp_session->push_action = NULL;//何もしない（ACK を返すだけ）
→TCP の接続確立後に、Push フラグのデータを受け取ったときにどのようにアクションするかを決める値
　現在は接続確立前なので、何もしない（ACK は返す）という設定です
　接続確立後にアクションを登録します
　右辺値は関数名（関数アドレス）となります

tcp_session_function[tcp_session->session_serial] = tcp_connection_open_request_queue;
→メイン関数内のメインループ内で実行される schedule_process()内で実行される関数
　右辺値は関数名（関数アドレス）となります
　ここで定義した関数はメインループ内で連続的に実行されます

　関数ポインタを使っていて、この時点でメインループ内で
tcp_connection_open_request_queue()
が連続的に実行される動作となります。

・tcp.c

```
void tcp_connection_open_request_queue(TCP_CONNECTION *tcp_session)
{
    //TCP接続の開始を要求する実際の関数

    //戻り値
    //  なし

    //引数
    // *tcp_session : TCP_CONNECT構造体

    //補足
    //  本関数はメインループで何度も呼ばれる事を想定した作り

    /* 工程表
     * 1. 接続先のMACアドレスの解決を行う
     * 2. 3ウェイハンドシェイクの1個目のパケットを送信する（本ボードからの送信1回目）
     * 3. 3ウェイハンドシェイクの接続相手からの2個目のパケットの応答を待つ（本ボード受信1回目）
     * 4. 3ウェイハンドシェイクの3個目のパケットを送信する（本ボードからの送信2回目）
     * 5. 接続が確立された状態でアイドル
     * 99.タイムアウト処理
     */

     unsigned short i;
     int ret;

     TCP_HEADER tcp_header;

    //タイムアウト処理（設定した時間が来るとタイムアウト）
    if ((tcp_session->timeout != 0 ) && (tcp_session->timeout <= g_timer_counter))
    {
        sci_write_str("tcp_connection_open: error: timeout.¥n");
        sci_write_str("  sequence -> ");
        sci_write_uint16(tcp_session->sequence);
        sci_write_str("¥n");

        tcp_session->sequence = 99;//処理の終了
    }

    //ウェイトの処理（設定した時間までは待ち）
    if (tcp_session->wait_time > g_timer_counter)
    {
        return;
    }
```

本関数は、tcp_connection_open_request_queue()を実行したことにより、メインループ内で実行される関数に登録されて、メインループ内で連続的に実行されます。

予め設定したタイムアウト時間が来た場合は、工程 99（タイムアウト）に移行します。また、予め設定したウェイト時間がある場合は、ウェイト時間経過するまでは直ぐに本関数を抜けます。

tcp_connection_open_request_queue()内では、
・タイムアウト 3 秒(3000ms)
・ウェイトなし
の設定としています。

3 秒以内にタイムアウト設定を無効化するか、タイムアウト時間を更新しない場合は本関数の先頭のところでタイムアウト処理に移行（工程表を 99 のタイムアウトに設定する）となります。

・tcp.c,（tcp_connection_open_request_queue() 続き）

```
    switch (tcp_session->sequence)
    {
        case 1:
            //接続先のMACアドレスの解決（工程表の中で本工程が複数回実行される可能性あり）

            if ((g_dst_MAC[0] == 0) && (g_dst_MAC[1] == 0) && (g_dst_MAC[2] == 0))
            {
/*
 * g_dst_MAC（共通で使うIP-MACアドレス解決の結果変数） の先頭3バイト（ベンダコード）が0x00の場合は
  * IPアドレスとMACアドレスの対応が判明していないのでARPリクエストを送る
 * ARPリクエストの結果は、g_dst_MAC 変数に保存される
 * g_dst_MAC 変数はMACアドレス解決に共通で使用する
 * ※複数のIPアドレスーMACアドレスの解決を同時に行う場合は g_dst_MAC に相当する変数を拡張要
 * 現状はARPリクエスト（IPアドレスに対するMACアドレスの問い合わせ）から結果の回収までの間
 * 複数のリクエストが発生しない前提
  */

                sci_write_str("tcp_connection_start: ");
                sci_print_IP(tcp_session->dst_IP);
                sci_write_str(" MAC address?¥n");

                ARP_request(tcp_session->dst_IP);

                tcp_session->wait_time = g_timer_counter + 500;//短時間で何回もARPリクエストを送る事を抑止す
るため、500ms間は次のARPリクエストは送らない
            }
            else
            {
                //IPアドレス-MACアドレスの対応が判明したので保存して次の工程に進む
                for (i=0; i<6; i++)
                {
                    tcp_session->dst_MAC[i] = g_dst_MAC[i];
                    g_dst_MAC[i] = 0;//共通で使用するMACアドレス変数は初期化
                }

                 tcp_session->sequence = 2;//次の工程に進む
                 tcp_session->time = g_timer_counter;//処理を行った時間を記録
             }
             break;
```

工程 1

```
        case 2:
                //3ウェイハンドシェイクの1個目のフレームを送信する（工程表の中で1回しか実行されない）

                tcp_connection_flag_send(tcp_session, TCP_FLAG_SYN);//SYNフラグ送信

                tcp_session->timeout = g_timer_counter + 10000;//3ウェイハンドシェイクの相手の応答タイムアウト
（10秒）

                tcp_session->sequence = 3;//次の工程に進む
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
                break;
```

工程 2

工程 1 では、MAC アドレスの解決を行います。ARP リクエストを送り、MAC アドレスの解決ができた時点で工程 2 に進めます。工程 2 では、3 ウェイハンドシェイクの 1 回目のデータを送信します。

・tcp.c,（tcp_connection_open_request_queue() 続き）

```
        case 3:
        //3ウェイハンドシェイクの2個目となる応答フレームを受信する（工程表の中で本工程が複数回実行される可能性あり）

        ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデータ読み出し
        if (ret != 0)
        {
            stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解（Ethernetヘッダ(14)+IPヘッダ(20)を除いた部分）
            if (tcp_header.tcp_ack_no != (tcp_session->seq_no + 1UL))//こちらが送信したシーケンシャル番号+1が相手が返す確認応答番号の期待値
            {
//確認応答番号が一致しない
                sci_write_str("tcp_connection_open: warning: handshake Acknowledgement number is incorrect.¥n");
                sci_write_str("  received  -> 0x");
                sci_write_uint32_hex(tcp_header.tcp_ack_no);
                sci_write_str("¥n");
                sci_write_str("  expection -> 0x");
                sci_write_uint32_hex(tcp_session->seq_no + 1UL);
                sci_write_str("¥n");
            }
            if (!((tcp_header.tcp_flag & TCP_FLAG_ACK) && (tcp_header.tcp_flag & TCP_FLAG_SYN)))
            {
                //ACKフラグとSYNフラグが立っていない場合
                sci_write_str("tcp_connection_open: error: handshake ACK & SYN flag is not set.¥n");
                break;
            }

            sci_write_str("TCP handshake ACK & SYN flag received OK.¥n");

            tcp_session->seq_no = tcp_header.tcp_ack_no;//次に送るシーケンス番号は相手の確認応答番号
            tcp_session->ack_no = tcp_header.tcp_seq_no + 1UL;//次に送る確認応答番号は相手のシーケンシャル番号+1
            tcp_session->sequence = 4;//次の工程に進む
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
        }
        break;
```

工程 3

```
        case 4:
            //3ウェイハンドシェイクの3個目のフレームを送信する（工程表の中で1回しか実行されない）

            tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK);//ACKフラグ送信

            tcp_session->connect = TCP_CONNECT;//接続試行中→接続

            tcp_session->sequence = 5;//次の工程に進む
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 4

```
        case 5:
            //終了処理（工程表の中で1回しか実行されない）->アイドル状態に移行

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

            tcp_session->timeout = 0;//tcp_sessionは切断まで使いまわすので、タイムアウトなしに設定しておく

            tcp_session_function[tcp_session->session_serial] = tcp_connection_idle_queue;//ループでの実行を tcp_connection_idle_queue() に移行
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 5

工程 3 は 3 ウェイハンドシェイクの 2 回目のデータの受信待ちを行います。工程 4 は 3 ウェイハンドシェイク 3 回目のデータを送信します。工程 5 は接続が完了したので、データ送受信モードへ移行させます。

・tcp.c,（tcp_connection_open_request_queue() 続き）

```
        case 99:
            //異常終了（タイムアウト）

            tcp_session->connect = TCP_DISCONNECT;//接続試行中→切断
            tcp_session_function[tcp_session->session_serial] = NULL;//メイン関数のループ処理関数の登録を解除
            tcp_session->sequence = 0;//工程はリセット
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
        break;

        default:
            sci_write_str("Prograss table undefined.¥n");
            tcp_session_function[tcp_session->session_serial] = NULL;//メイン関数のループ処理関数の登録を解除
            tcp_session->sequence = 0;//工程はリセット
            break;
    }
}
```

タイムアウト

タイムアウトや工程表にない工程番号となった場合のエラー処理です。

工程 1（複数回呼ばれる、相手からの通信を待つ工程）

・MAC アドレスの解決が済んでいるかの確認

→済んでいれば工程 2 に進む

・ARP リクエストを送る

・500ms のウェイトを設定

MAC アドレスの解決ができていなければ、ARP リクエストを送って本関数を抜けます。本関数は、メインループ内の schedule_process()内で連続的に実行されます。（マイコンが割り込みや受信データの処理など他のタスクを実行していない時は、短時間で何回も実行されます。）そのため、500ms のウェイトを設定して 500ms 間は本関数を呼ばれた場合直ぐに抜ける様になります。

工程 2（1 回しか実行されない、相手に依存しない）

・TCP ヘッダ内の SYN（接続要求）フラグを立ててデータ送信（TCP のペイロードは空）

・タイムアウトを 10 秒に設定

・工程 3 に進む

工程 3（複数回呼ばれる、相手からの通信を待つ工程）

・TCP のデータを受信しているかを確認して所定（SYN, ACK フラグ）のデータを受信したかの確認

・工程 4 に進む

工程 4（1 回しか実行されない、相手に依存しない）

・TCP ヘッダ内の ACK（返答）フラグを立ててデータ送信（TCP のペイロードは空）

・工程 5 に進む

工程 5(1 回しか実行されない、相手に依存しない)

・タイムアウトを 0(タイムアウトなし)に設定

→工程 3 から工程 5 までで 10 秒以内に完了しないとタイムアウトとなる

・メインループで呼ばれる関数を tcp_connection_idle_queue()に設定

→現時点ではメインループで呼ばれる関数に登録されているのは本関数(tcp_connection_open_request_queue())

ここから、別な関数に処理を引き継ぐ形となります

8.7.2. クライアント通信処理

接続完了となると、tcp_connection_idle_queue()がメインループ内で連続的に実行される様になります。(工程 5)

・tcp.c

```
void tcp_connection_idle_queue(TCP_CONNECTION *tcp_session)
{
    //TCPの接続を維持監視する関数

    //戻り値
    //  なし

    //引数
    // *tcp_session : TCP_CONNECT構造体

    //補足
    //  本関数はメインループで何度も呼ばれる事を想定した作り
    //  本関数は
    //  tcp_session->sequence == 5
    //  の時に実行される想定

    unsigned short i;
    int ret;
     unsigned short header_size;
     unsigned short data_size;

     TCP_HEADER tcp_header;
    IP_HEADER ip_header;

    //タイムアウト処理(設定した時間が来るとタイムアウト)
    if ((tcp_session->timeout != 0 ) && (tcp_session->timeout <= g_timer_counter))
    {
        sci_write_str("tcp_connection_idle: error: timeout.¥n");
        sci_write_str("  sequence -> ");
        sci_write_uint16(tcp_session->sequence);
        sci_write_str("¥n");

        tcp_session->connect = TCP_DISCONNECT;//接続→切断
        tcp_session->sequence = 0;//工程表はリセット

        tcp_session_function[0] = NULL;//ループでの実行をキャンセル
        tcp_session->time = g_timer_counter;//処理を行った時間を記録

        return;
    }

    //ウェイトの処理(設定した時間までは待ち)
    if (tcp_session->wait_time > g_timer_counter)
    {
        return;
    }
```

タイムアウトとウェイトは、tcp_connection_open_request_queue()と同じです。

・tcp.c,（tcp_connection_idle_queue() 続き）

```
    //接続相手からのデータの待ち受けを行いデータの種類に応じた処理

    ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデータ読み出し
    if (ret != 0)
    {
    stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解
    stream_to_IP_header(&tcp_buf[14], &ip_header);//受信したストームをIPヘッダに分解

    //送信データサイズ
    header_size = (tcp_header.tcp_flag & 0xF000) >> 10;//フラグの上位4bit（b15-b12)×4バイト（12bit右シフトの後4を掛ける→10bit右シフト）
    data_size = ip_header.ip_total_length - 20 - header_size;//IPヘッダ内のデータ長からIPヘッダ(20)，TCPヘッダ(可変)を差し引く

    //相手が送ってくる応答確認番号
    /*
     * こちらが送信したシーケンシャル番号+送信したデータサイズが相手が返す確認応答番号の期待値
     * データ送信の際に、tcp_session->seq_no に送信したデータサイズを加算済み
     */
    if (tcp_header.tcp_ack_no != tcp_session->seq_no)
    {
        //確認応答番号が一致しない
        sci_write_str("tcp_connection_idle: warning: Acknowledgement number is incorrect.¥n");
        sci_write_str("  received  -> 0x");
        sci_write_uint32_hex(tcp_header.tcp_ack_no);
        sci_write_str("¥n");
        sci_write_str("  expection -> 0x");
        sci_write_uint32_hex(tcp_session->seq_no + 1UL);
        sci_write_str("¥n");
    }

    if (tcp_header.tcp_flag & TCP_FLAG_RST)
    {
        sci_write_str("tcp_connection_idle: warning: RESET flag received.¥n");
        sci_write_str("-- CONNECTION CLOSED (RESET!)--¥n");

        //強制切断

        tcp_session->connect = TCP_DISCONNECT;//接続→切断
        tcp_session->sequence = 0;

        tcp_session_function[tcp_session->session_serial] = NULL;//ループでの実行をキャンセル

        return;
    }

    sci_write_str("TCP frame received from ");
    sci_print_IP(tcp_session->dst_IP);
    sci_write_str(":");
    sci_write_uint16(tcp_session->dst_port);
    sci_write_str(" Flag = ");
    sci_print_tcp_flag_short_format(tcp_header.tcp_flag);
    if (data_size > 0)
    {
        sci_write_str(", ");
        sci_write_uint16(data_size);
        sci_write_str(" bytes data¥n");
        sci_write_str(">");
        for (i=0; i<data_size; i++)
        {
            sci_write_char(tcp_buf[14 + 20 + header_size + i]);//Ethernetヘッダ(14)+IPヘッダ+TCPヘッダ(可変)の後がTCPのデータ
        }
    }
    sci_write_str("¥n");
```

・tcp.c,（tcp_connection_idle_queue() 続き）

```
    tcp_session->seq_no = tcp_header.tcp_ack_no;//次に送るシーケンス番号は相手の確認応答番号
    tcp_session->ack_no = tcp_header.tcp_seq_no + (unsigned long)data_size;//次に送る確認応答番号（相手が送
ってきたシーケンス番号+データサイズ）

    if (tcp_header.tcp_flag & TCP_FLAG_FIN)
    {
        //相手がFIN(切断)フラグを送ってきた

        sci_write_str("tcp_connection_idle: message: FIN flag received.¥n");

        //通常の切断処理

        tcp_session->sequence = 7;
        tcp_session_function[tcp_session->session_serial] = tcp_connection_close_response_queue;//接続を切
断するシーケンスを実行
    }
    else if (tcp_header.tcp_flag & TCP_FLAG_PSH)
    {
        //Push（相手がデータを送ってきた）の場合

        if (tcp_session->push_action == NULL)
        {
            //データを受け取った際のアクション未登録の場合->受け取った旨の返信を行う

            tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK);
        }
        else
        {
            //予め登録済みの関数を実行する（この関数内でACKフラグ付きで返信する）

            tcp_session->push_action();
        }
    }

    tcp_session->time = g_timer_counter;//処理を行った時間を記録
    }
}
```

クライアント動作

サーバ動作

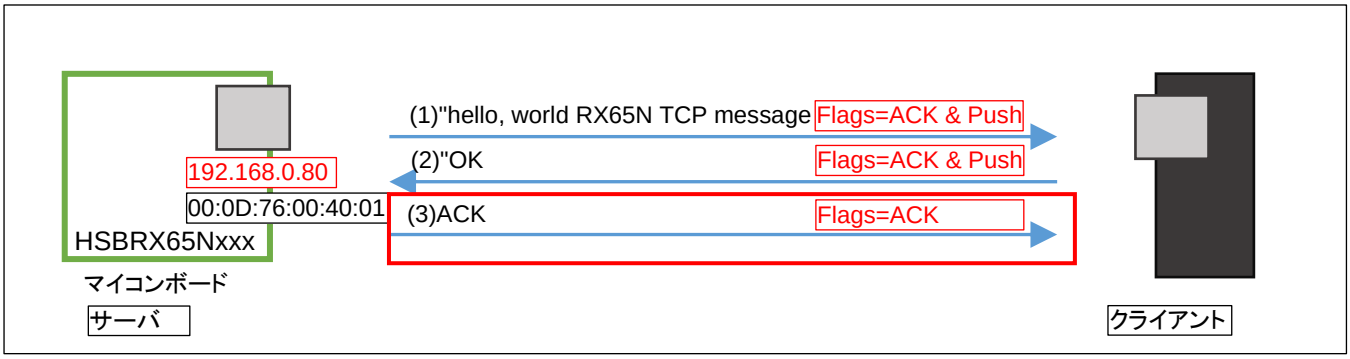
接続確立後は、本関数がループで呼ばれている状態です。

Reset フラグを受け取った場合は、即切断状態に移行します。

Fin フラグを受け取った場合は、工程 7 に進みます。（基本的にはサーバ動作時）

Push フラグを受け取た場合、データ受信処理を行います。

クライアント動作の場合は、データを受け取った通知(ACK)を返します（下図赤枠部）。サーバ動作時は、予め設定された関数を実行します（後述）。



8.7.3. クライアントからのデータ送信

接続済みの状態で、s コマンドを入力するとサーバにメッセージを送信します。
接続済みの状態は、工程では工程 5 です。

s コマンドでは、text_packet_send()関数が実行されます。

・text_client.c

```
void text_packet_send(void)
{
    //TCPテキストメッセージ送信関数

    //戻り値
    //  なし

    //引数
    //  なし

    //補足
    //  g_tcp_client で開かれている接続に対し、g_send_text_data を返送する

    if (g_tcp_client.connect != TCP_CONNECT)
    {
        //TCPの通信が開いていない

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

        return;
    }

        tcp_send_data(&g_tcp_client, g_send_text_data, TCP_MESSAGE_SIZE, TCP_FLAG_ACK | TCP_FLAG_PSH);
//ACKとPUSHフラグを立てて送信
}
```

本関数は、接続が確立していれば、tcp_send_data()関数を呼び出します。

・tcp.c

```
void tcp_send_data(TCP_CONNECTION *tcp_session, unsigned char *data, unsigned short size, unsigned
short tcp_flag)
{
    //TCPでデータ送信する関数

    //戻り値
    //  なし

    //引数
    // *tcp_session : TCP_CONNECT構造体
    // *send_data : 送信データ
    // size : 送信データサイズ
    // flag : TCPフラグ

    ETHERNET_HEADER ethernet_header;
    IP_HEADER ip_header;
    TCP_HEADER tcp_header;
    PSEUDO_HEADER pseudo_header;

    unsigned short data_size;
    unsigned short send_data_size;
```

・tcp.h

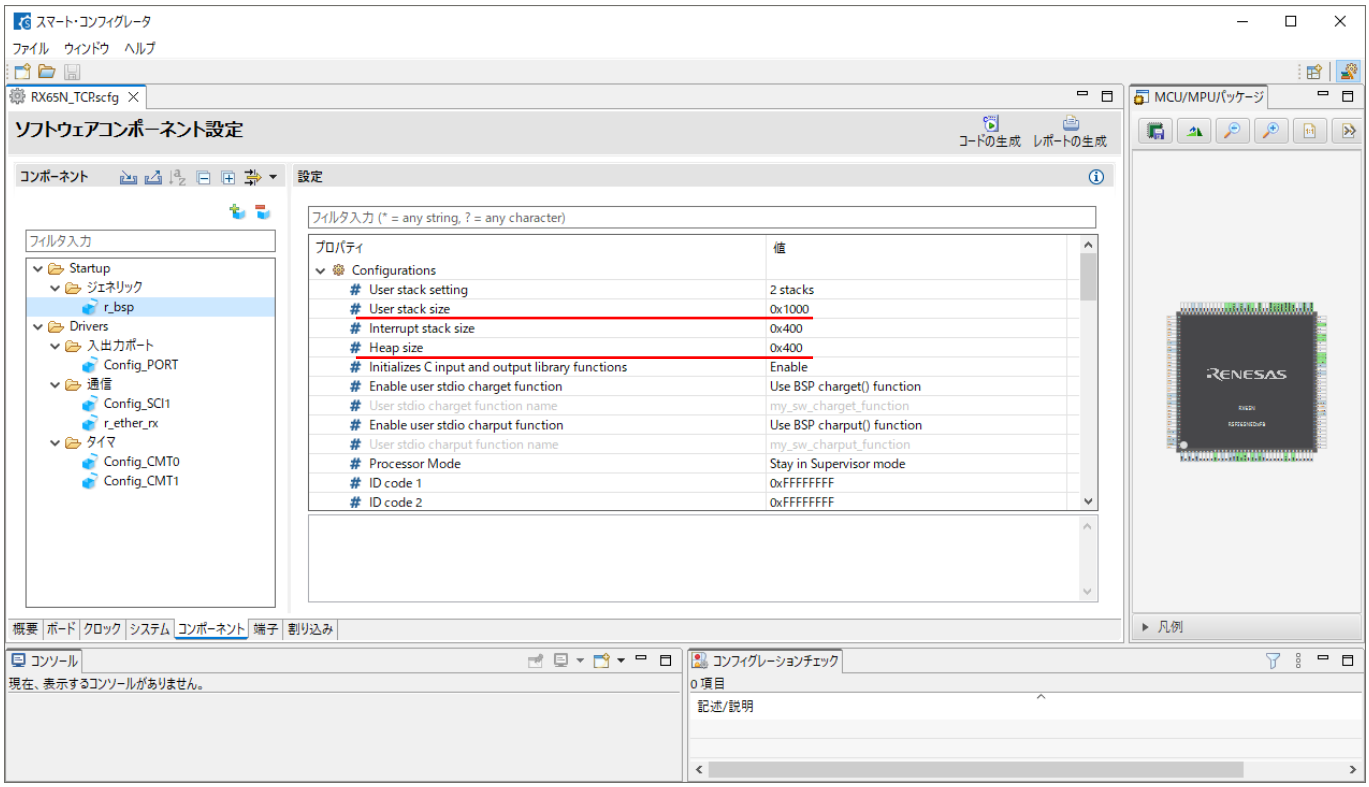
```
//ワークメモリ
#define TCP_WORK_MEMORY_HEAP  (1)//ヒープメモリ上に確保（デフォルト1kB）,使用する分だけメモリを使用
#define TCP_WORK_MEMORY_USER_STACK  (2) //ユーザスタック上に確保（デフォルト4kBだが色々なバッファが使用する，常に1514バイト）
#define TCP_WORK_MEMORY_STATIC  (3) //静的変数として確保（常に1514バイト）

#define TCP_WORK_MEMORYTCP_WORK_MEMORY_HEAP //デフォルトはヒープ上に動的に確保
```

本関数は一時的な変数上で送信データを展開してから送信となります。データをどこに確保するかで 3 種類の方法を定義しています。

(1)calloc(), free()を使い、ヒープメモリ上に確保する

(2)ユーザスタック上（自動変数）に確保する

(3)static 変数として確保する



ライブラリ関数の calloc(), free()を使う場合は、ヒープ領域からメモリを確保します。ヒープ領域は、デフォルトでは 0x400(1kB)です。本プロジェクトでは、40 バイトのデータしか取り扱わないので送信データのサイズは 94 バイトとなります。（Ethernet ヘッダ(14)+IP ヘッダ(20)+TCP ヘッダ(20)+データ(40) = 94 バイト）

Ethernet フレームの最大サイズ（1514 バイト）のデータを取り扱う際は、スマート・コンフィグレータの設定で Heap size を増やしてください。

ユーザスタック上に確保する場合は、ユーザスタックはデフォルトでは 0x1000(4kB)ですが、ここには様々なデータが確保されます。（トータルで 4kB で足りるかの試算が必要になってきます。）

static で確保した場合は、常に同じアドレスに確保され、そのメモリ領域は送信バッファ専用になります。

デフォルトは(1)が選択されています。(2)(3)を使った場合は、送信データサイズに拘わらず、1514 バイトの送信バッファが確保されます。(1)の場合は、送信データサイズに応じたサイズのメモリが使用されます。

・tcp.c（tcp_send_data() 続き）

```
//送信バッファの領域確保

#if (TCP_WORK_MEMORY == TCP_WORK_MEMORY_HEAP)

    //ヒープ上に動的に確保（ヒープメモリサイズと送信データサイズに注意）
    unsigned char *send_data;

#elif (TCP_WORK_MEMORY == TCP_WORK_MEMORY_USER_STACK)

    //ユーザスタック上に最大値で確保
    unsigned char send_data[1514] = {0};

#endif

    ether_return_t ret;

    unsigned short i;

    //最大1460bytes（Ethernetフレームの最大(1518) - Ethernetヘッダ(14) - IPヘッダ(20) - TCPヘッダ(20) -
FCS(4)）
    if (size > 1460) return;

    //送信するサイズは偶数バイトとする
    if ((size % 2) != 0)
    {
        data_size = size + 1;
    }
    else
    {
        data_size = size;
    }

    //送信データサイズ
    if (data_size >= 6)
    {
        //Ethernetヘッダ(14) + IPヘッダ(20) + TCPヘッダ(20) + 送信データ
        send_data_size = 14 + 20 + 20 + data_size;
    }
    else
    {
        //Ethernetフレームは最小60バイト
        send_data_size = 60;
    }

    //送信バッファ
#if (TCP_WORK_MEMORY ==TCP_WORK_MEMORY_HEAP)

    send_data = calloc(send_data_size, 1);

    if (send_data == NULL)
    {
        sci_write_str("*** insufficient memory ***¥n");
        return;
    }
    /*
     * callocは、ヒープメモリから領域を確保しますので
     * 大きなサイズのデータを送る際は
     * スマート・コンフィグレータの「コンポーネント」タブ
     * r_bsp の #Heap size 0x400（デフォルト1024バイト）
     * を大きくしてください
     */
```

・tcp.c（tcp_send_data() 続き）

```
#elif (TCP_WORK_MEMORY ==TCP_WORK_MEMORY_STATIC)

    if (data_size < 6)
    {
        for (i=54; i<60; i++)
        {
            send_data[i] = 0x00;
        }

    /*
     * static で send_data を確保した場合、トータルの送信サイズが60バイト未満の場合
     * パディングデータを0x00としたいのでパディングデータのゼロ埋めを行っておく
     */
#endif

    //Ethrnetヘッダの設定// 14bytes

    //MACアドレス
    for (i=0; i<6; i++)
    {
    ethernet_header.eth_dst_MAC[i] = tcp_session->dst_MAC[i];
    ethernet_header.eth_src_MAC[i] = tcp_session->src_MAC[i];
    }
    ethernet_header.eth_ethernet_type = 0x0800;//上位プロトコルの種類(IP)

    //IPヘッダの設定// 20bytes
    ip_header.ip_version_length = 0x45;//バージョン及びヘッダ長
    ip_header.ip_service_type = 0x00;//パケット優先度
    ip_header.ip_total_length = (unsigned short)(40 + data_size);//IPデータグラムの長さ，IPヘッダ(20) +
TCPヘッダ(20) + 送信データ
    ip_header.ip_id = tcp_session->id++;//パケットの識別子(ID)，送信毎にインクリメント
    ip_header.ip_flags_fragment_offset = 0x4000;//パケットの分割有無（分割なし）
    ip_header.ip_time_to_live = 0x80;//ルータ中継回数の上限
    ip_header.ip_protocol = 0x6;//プロトコルの種類(TCP)
    ip_header.ip_checksum = 0x0000;//チェックサム（仮値）

    //IPアドレス
    for (i=0; i<4; i++)
    {
        ip_header.ip_src_IP[i] = tcp_session->src_IP[i];
        ip_header.ip_dst_IP[i] = tcp_session->dst_IP[i];
    }

    //チェックサムの計算
    ip_header.ip_checksum = ip_header_sum(&ip_header);

    //TCPヘッダの設定//

    tcp_header.tcp_src_port = tcp_session->src_port;//送信元ポート
    tcp_header.tcp_dst_port = tcp_session->dst_port;//宛先ポート
    tcp_header.tcp_seq_no = tcp_session->seq_no;//シーケンス番号
    tcp_header.tcp_ack_no = tcp_session->ack_no;//確認応答番号
    tcp_header.tcp_flag = TCP_HEADER_SIZE | (tcp_flag & 0x01ff);//ヘッダ長+フラグ(b8-b0)
    tcp_header.tcp_window = 1460;//ウィンドウサイズ
    tcp_header.tcp_checksum = 0x0000;//チェックサム（仮値）
    tcp_header.tcp_urgent_pointer = 0x0000;//緊急ポインタ

    //チェックサム計算
    for (i=0; i<4; i++)
    {
        pseudo_header.ip_src_IP[i] = tcp_session->src_IP[i];
        pseudo_header.ip_dst_IP[i] = tcp_session->dst_IP[i];
    }
    pseudo_header.zero_padding = 0x00;
    pseudo_header.ip_protocol = 0x06;//TCP
    pseudo_header.length = 20 + data_size;//TCPヘッダ+データ
```

・tcp.c（tcp_send_data() 続き）

```
    tcp_header.tcp_checksum = tcp_sum(&pseudo_header, &tcp_header, data, size);

    //送信データ
    //ETHERNETヘッダ(14) + IPヘッダ(20) + TCPヘッダ(20) + データ
    ETHERNET_header_to_stream(&ethernet_header, send_data);
    IP_header_to_stream(&ip_header, &send_data[14]);
    TCP_header_to_stream(&tcp_header, &send_data[14+20]);
    memcpy(&send_data[14+20+20], data, size);

    //TCPパケット送信
    ret = R_ETHER_Write(g_ether_channel, (void *)send_data, send_data_size);

    sci_write_str("TCP data packet (");
    sci_write_uint16(data_size);
    sci_write_str(" bytes) (flag = ");

    sci_print_tcp_flag_short_format(tcp_flag);

    sci_write_str(") send to ");
    sci_print_IP(tcp_session->dst_IP);
    sci_write_str(":");
    sci_write_uint16(tcp_session->dst_port);

    if (ret != ETHER_SUCCESS)
    {
        sci_write_str(", [failed]");
    }
    sci_write_str("¥n");

    tcp_session->seq_no += (unsigned long)size;
    /*
     * 次に相手が送ってくる次の応答確認番号はシーケンシャル番号+送信データサイズなので
     * ここで、シーケンシャル番号に送信データサイズを加算しておく
     * （次にデータを受信した場合は、送られてきた応答確認番号と tcp_session->seq_no を比較すれば良い
     */

#if (TCP_WORK_MEMORY ==TCP_WORK_MEMORY_HEAP)
    free(send_data);
#endif
}
```

今までの、ARP や ping、UDP でのデータ送信の時と同様

・Ethernet ヘッダ

・IP ヘッダ

・TCP ヘッダ

・TCP データ

を送信用の変数に順に設定していき、API 関数(R_ETHER_Write())で送信するだけです。

各ヘッダがどのようなデータで構成されているのか、チェックサムをどの様に計算するかが判っていれば送信データ列を作るのは難しくありません。

TCP の場合は、シーケンシャル番号を TCP ヘッダ内に含める必要があるので、最後のところで

tcp_session->seq_no

を、送信バイト（この場合は、TCP データの部分のバイト数）分加算しています。

8.7.4. クライアント切断処理

接続中に d コマンドで切断した場合の動作です。

・tcp.c

```
int tcp_connection_close_request(TCP_CONNECTION *tcp_session)
{
    //TCP接続の終了を要求する関数を実行キューに登録する関数

    //戻り値
    //  0 : 正常終了（接続終了処理関数をキューに登録）
    //  -1 : 未接続、または接続試行中

    //引数
    //  tcp_session : TCP接続情報

    if (tcp_session->connect != TCP_CONNECT)
    {
        sci_write_str("tcp_connection_close: error: TCP connection is not established.¥n");
        return -1;
    }

    tcp_session->sequence = 6;//工程6からスタート

    tcp_session->wait_time = 0;//初期化（ウェイトなし）
    tcp_session->time = g_timer_counter;//処理が始まった時間を記録

    //メイン関数で実行される関数に工程実行関数を登録
    tcp_session_function[tcp_session->session_serial] = tcp_connection_close_request_queue;

    return 0;
}
```

・工程を 6 に設定

・メインループで実行される関数を tcp_connection_close_request_queue()

とします。

・tcp.c

```
void tcp_connection_close_request_queue(TCP_CONNECTION *tcp_session)
{
    //TCP接続の終了を要求する実際の関数

    //戻り値
    //  なし

    //引数
    // *tcp_session : TCP_CONNECT構造体

    //補足
    //  本関数はメインループで何度も呼ばれる事を想定した作り

    /* 工程表
     * 6. 4ウェイハンドシェイクの1個目のパケットを送信する（本ボードから送信1回目）
     * 7. 4ウェイハンドシェイクの接続相手からの2個目の応答を待つ（本ボード受信1回目）
     * 8. 4ウェイハンドシェイクの接続相手からの3個目の応答を待つ（本ボード受信2回目）
     * 9. 4ウェイハンドシェイクの4個目のパケットを送信する（本ボードから送信2回目）
     * 10.終了処理
     * 99.タイムアウト処理
     */

    int ret;

    TCP_HEADER tcp_header;

（タイムアウト処理とウェイト処理省略）

    switch (tcp_session->sequence)
    {
        case 6:
            //4ウェイハンドシェイクの1個目のフレームを送信する（工程表の中で1回しか実行されない）

            tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK | TCP_FLAG_FIN);//ACKとFINフラグ送信

            tcp_session->timeout = g_timer_counter + 10000;//4ウェイハンドシェイクの相手の応答タイムアウト
（10秒）

            tcp_session->sequence = 7;//次の工程に進む
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 6

```
        case 7:
        //4ウェイハンドシェイクの2個目となる応答フレーム(1)を受信する（工程表の中で本工程が複数回実行される可能
性あり）
            ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデ
ータ読み出し
            if (ret != 0)
             {
                stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解
                if (tcp_header.tcp_ack_no != (tcp_session->seq_no + 1UL))//こちらが送信したシーケンシャル
番号+1が相手が返す確認応答番号の期待値
                {
                    //確認応答番号が一致しない
（エラー表示、中略）
                }
                if (!(tcp_header.tcp_flag & TCP_FLAG_ACK))
                {
                    //ACKフラグが立っていない場合
                    sci_write_str("tcp_connection_open: error: handshake ACK flag is not set.¥n");
                    break;
                 }

                sci_write_str("TCP handshake ACK flag received OK.¥n");
```

工程 7

・tcp.c（tcp_connection_close_request_queue() 続き）

```
                    tcp_session->seq_no += 1UL;//次に送るシーケンス番号は+1
                    tcp_session->ack_no = tcp_header.tcp_seq_no + 1UL;//次に送る確認応答番号は相手のシーケンシャル番号+1
                    tcp_session->sequence = 8;//次の工程に進む
                    tcp_session->time = g_timer_counter;//処理を行った時間を記録
                }
                break;
```

工程 7

```
            case 8:
                //4ウェイハンドシェイクの3個目となる応答フレーム(2)を受信する（工程表の中で本工程が複数回実行される可能性あり）

                ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデータ読み出し
                if (ret != 0)
                {
                    stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解

                    if (tcp_header.tcp_ack_no != tcp_session->seq_no)//こちらが送信したシーケンシャル番号が相手が返す確認応答番号の期待値(*)
                    {
                        //確認応答番号が一致しない
（エラー表示、中略）
                    }

                    if (!((tcp_header.tcp_flag & TCP_FLAG_ACK) && (tcp_header.tcp_flag & TCP_FLAG_FIN)))
                    {
                        //ACKフラグとFINフラグが立っていない場合
                        sci_write_str("tcp_connection_open: error: handshake ACK & FIN flag is not set.¥n");
                        break;
                    }

                    sci_write_str("TCP handshake ACK & FIN flag received OK.¥n");

                    //tcp_session->seq_no//次に送るシーケンス番号は変更なし
                    tcp_session->ack_no = tcp_header.tcp_seq_no + 1UL;//次に送る確認応答番号は相手のシーケンシャル番号+1
                    tcp_session->sequence = 9;//次の工程に進む
                    tcp_session->time = g_timer_counter;//処理を行った時間を記録
                }
                break;
```

工程 8

```
            case 9:
                //4ウェイハンドシェイクの4個目のフレームを送信する（工程表の中で1回しか実行されない）

                tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK);//ACKフラグ送信

                tcp_session->sequence = 10;//次の工程に進む
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
                break;
```

工程 9

```
            case 10:
                //終了処理（1回しか来ない）
                sci_write_str("-- CONNECTION CLOSED --¥n");

                tcp_session->connect = TCP_DISCONNECT;//切断
                tcp_session_function[tcp_session->session_serial] = NULL;//メイン関数のループ処理関数の登録を解除
                tcp_session->sequence = 0;//工程はリセット
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
                break;
```

工程 10

```
（タイムアウト処理、後略）
```

工程 6（1 回しか実行されない、相手に依存しない）
・タイムアウトを 10 秒に設定
・FIN & ACK フラグの送信
・工程 7 に進む

工程 7（複数回呼ばれる、相手からの通信を待つ工程）
・ACK フラグの受信待ち
・工程 8 に進む

工程 8（複数回呼ばれる、相手からの通信を待つ工程）
・FIN & ACK フラグの受信待ち
・工程 9 に進む

工程 9（1 回しか実行されない、相手に依存しない）
・ACK フラグの送信
・工程 10 に進む

工程 10（1 回しか実行されない、相手に依存しない）
・状態変数のセット
・メインループで実行される関数を空にする

切断処理は、4 ウェイハンドシェイクとなり、工程 6-9 となります。すべて終了すると、メインループで連続的に実行される関数を空にします。（現状は、tcp_connection_close_request_queue()が実行されている状態）

8.7.5. サーバ待ち受け処理

サーバ側の動作は、メインループに入る前に、tcp_connection_open_response()が呼ばれる事で、接続の待ち受けを行います。
また、一度接続したクライアント側から FIN（切断要求）が来て、切断した後も、同じ関数が呼ばれて、再度待ち受け状態となります。

・tcp.c

```
int tcp_connection_open_response(unsigned short port, TCP_CONNECTION *tcp_session, unsigned short
session_serial, void (*function)())
{
    //TCP接続の待ち受けを行う関数を実行キューに登録する関数

    //戻り値
    //  0 : 正常終了（接続開始処理関数をキューに登録）
    //  -1 : 接続済み、または接続試行中

    //引数
    //  port : 待ち受けポート
    //  tcp_session : TCP接続情報
    //  session_serial: セッションのシリアル番号
    //function() : PUSH（データ受信時）のアクション関数登録

    unsigned short i;

    if (tcp_session->connect != TCP_DISCONNECT)
    {
        sci_write_str("tcp_connection_open: error: TCP connection is already open or try to open.¥n");
        return -1;
    }

    tcp_session->sequence = 2;//工程の最初からスタート（1.IPアドレス-MACアドレス解決は待ち受けの場合は行わない
ので2からスタート）

    tcp_session->connect = TCP_CONNECT_ATTEMPT;//接続試行中
    tcp_session->id = 0;//最初はID=0からスタート
    tcp_session->wait_time = 0;//初期化（ウェイトなし）
    tcp_session->time = g_timer_counter;//処理が始まった時間を記録
    tcp_session->timeout = 0;//本関数が呼ばれてから接続要求が来るまでのタイムアウトはなし

    //MACアドレスの設定（保存，初期化）
    for (i=0; i<6; i++)
    {
        tcp_session->src_MAC[i] = g_src_MAC[i];//ether_operation.c で定義されている本ボードのMACアドレス
        tcp_session->dst_MAC[i] = 0x00;//初期化（接続要求時に判明）
    }

    //IPアドレスの設定（保存，初期化）
    for (i=0; i<4; i++)
    {
        tcp_session->src_IP[i] = g_src_IP[i];//ether_operation.c で定義されている本ボードのIPアドレス
        tcp_session->dst_IP[i] = 0x00;//初期化（接続要求時に判明）
    }

    //使用ポート
    tcp_session->src_port = port;//待ち受けで使用するポート（引数で与える）

    //接続先ポート
    tcp_session->dst_port = 0;//初期化（接続要求時に判明）

    //シーケンス番号
    tcp_session->seq_no = (unsigned long)rand();//シーケンス番号はランダムに決める（0-7FFFの範囲であるかOKと
する）

    //応答確認番号
    tcp_session->ack_no = 0;//応答確認番号初期化（接続要求時に決まる）

    //セッションのシリアル番号
    tcp_session->session_serial = session_serial;

    //PUSHフラグを受信した際のアクション
    tcp_session->push_action = function;//アクション関数を登録

    //メイン関数で実行される関数に工程実行関数を登録
    tcp_session_function[tcp_session->session_serial] = tcp_connection_open_response_queue;

    return 0;
}
```

各種変数の初期化と、メインループで実行される関数として、tcp_connection_open_response_queue()を登録します。

接続待ちを開始する本関数では、工程は 2 からスタートとしています。

→本ボードから接続相手に対して IP-MAC アドレスの解決（ARP リクエスト）は発行しません

→接続相手から、ARP リクエストが飛んでくるが、それは TCP とは無関係（ARP の応答ルーチン）で処理されます

→接続相手からの Syn フラグが送られてきた際のパケット内に、接続相手の MAC アドレスと IP アドレスの情報が含まれます、こちらからはそのパケット内に含まれる MAC アドレスが正しいものとして Ehertnet ヘッダを生成します（接続相手の IP アドレスに対して、ARP リクエストを投げる動作は行わない）

なお、

tcp_session->push_action = function;

の部分は、接続が完了した後の工程 5 で関わってきます。工程 5（クライアントからのデータ待ち）の際に、データが来た場合（Push フラグが立っているデータが来た場合）この部分で指定した関数（本関数の引数で与えられた、function）を実行します。

→"OK(RX65N)"のメッセージを返信する動作の関数です

・tcp.c

```
void tcp_connection_open_response_queue(TCP_CONNECTION *tcp_session)
{
    //TCP接続の待ち受けを行う実際の関数

    //戻り値
    //  なし

    //引数
    // *tcp_session : TCP_CONNECT構造体

    //補足
    //  本関数はメインループで何度も呼ばれる事を想定した作り

    /* 工程表
     * 1.未定義
     * 2.3ウェイハンドシェイクの1個目のパケットを受信する
     * 3.3ウェイハンドシェイクの2個目となる応答パケットを送信する
     * 4.3ウェイハンドシェイクの3個目のパケットを受信する
     * 5.接続が確立された状態でアイドル
     * 99.タイムアウト処理
     */

    unsigned short i;
    int ret;

    ETHERNET_HEADER ethernet_header;
    IP_HEADER ip_header;
    TCP_HEADER tcp_header;

（タイムアウトとウェイト処理、省略）
```

・tcp.c,（tcp_connection_open_response_queue() 続き）

```
    switch (tcp_session->sequence)
    {
        case 2:
        //3ウェイハンドシェイクの1個目の接続要求フレームを受信する（工程表の中で本工程が複数回実行される可能性あり）

            ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデータ読み出し
            if (ret != 0)
            {
                stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解
                if (tcp_header.tcp_ack_no != 0)//一般的には最初の確認応答番号の期待値は0
                {
                    //確認応答番号が一致しない
（エラー表示、省略）
                }
                if (!(tcp_header.tcp_flag & TCP_FLAG_SYN))
                {
                    //SYNフラグが立っていない場合
                    sci_write_str("tcp_connection_open: error: handshake SYN flag is not set.¥n");
                    break;
                }

                stream_to_ETHERNET_header(tcp_buf, &ethernet_header);//受信したストームをEthernetヘッダに分解
                for (i=0; i<6; i++)
                {
                    tcp_session->dst_MAC[i] = ethernet_header.eth_src_MAC[i];//受信したEthernetパケットに含まれるMACアドレスを送信先に設定
                }

                stream_to_IP_header(&tcp_buf[14], &ip_header);//受信したストームをIPヘッダに分解
                for (i=0; i<4; i++)
                {
                    tcp_session->dst_IP[i] = ip_header.ip_src_IP[i];//要求元のIPアドレスを送信先に設定
                }

                tcp_session->dst_port = tcp_header.tcp_src_port;//送り先のポートは送信元のポート

                sci_write_str("tcp_connection_start: ");
                sci_print_IP(tcp_session->dst_IP);
                sci_write_str(":");
                sci_write_uint16(tcp_session->dst_port);
                sci_write_str("¥n");

                sci_write_str("TCP handshake SYN flag received OK.¥n");

                //tcp_session->seq_no//次に送るシーケンス番号は最初にランダムで決めた番号（設定済み）
                tcp_session->ack_no = tcp_header.tcp_seq_no + 1UL;//次に送る確認応答番号は相手のシーケンシャル番号+1
                tcp_session->sequence = 3;//次の工程に進む
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
            }

            break;
```

工程 2

```
        case 3:
            //3ウェイハンドシェイクの2個目のフレームを送信する（工程表の中で1回しか実行されない）

            tcp_connection_flag_send(tcp_session, TCP_FLAG_SYN | TCP_FLAG_ACK);//SYN & ACKフラグ送信

            tcp_session->timeout = g_timer_counter + 10000;//3ウェイハンドシェイクの相手の応答タイムアウト（10秒）

            tcp_session->sequence = 4;//次の工程に進む
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 3

・tcp.c,（tcp_connection_open_response_queue() 続き）

```
        case 4:
            //3ウェイハンドシェイクの3個目となる応答フレームを受信する（工程表の中で本工程が複数回実行される可能性あり）

            ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデータ読み出し
            if (ret != 0)
            {
                stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解
                if (tcp_header.tcp_ack_no != (tcp_session->seq_no + 1UL))//こちらが送信したシーケンシャル番号+1が相手が返す確認応答番号の期待値
                {
//確認応答番号が一致しない
（エラー表示、省略）
                }
                if (!(tcp_header.tcp_flag & TCP_FLAG_ACK))
                {
                    //ACKフラグ立っていない場合
                    sci_write_str("tcp_connection_open: error: handshake ACK flag is not set.¥n");
                    break;
                }

                sci_write_str("TCP handshake ACK flag received OK.¥n");

                tcp_session->seq_no += 1UL;//次に送るシーケンス番号は+1
                tcp_session->ack_no = tcp_header.tcp_seq_no + 1UL;//次に送る確認応答番号は相手のシーケンシャル番号+1
                tcp_session->sequence = 5;//次の工程に進む
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
            }
            break;
```

工程 4

```
        case 5:
                //終了処理（工程表の中で1回しか実行されない）->アイドル状態に移行

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

                tcp_session->connect = TCP_CONNECT;//接続に移行
                tcp_session->timeout = 0;//tcp_sessionは切断まで使いまわすので、タイムアウトなしに設定しておく
                tcp_session_function[tcp_session->session_serial] = tcp_connection_idle_queue;//ループでの実行を tcp_connection_idle() に移行
                tcp_session->sequence = 0;//工程はリセット
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
                break;
```

工程 5

```
（後略）
```

工程 2（複数回呼ばれる、相手からの通信を待つ工程）

・SYN フラグの受信待ち

・工程 3 に進む

工程 3（1 回しか実行されない、相手に依存しない）

・SYN & ACK フラグの送信

・タイムアウト 10 秒の設定

・工程 4 に進む

工程 4（複数回呼ばれる、相手からの通信を待つ工程）

・ACK フラグの受信待ち

・工程 5 に進む

工程 5（1 回しか実行されない、相手に依存しない）

・タイムアウトを 0（タイムアウトなし）に設定

→工程 3 から工程 5 までで 10 秒以内に完了しないとタイムアウトとなる

・メインループで呼ばれる関数を tcp_connection_idle_queue()に設定

→現時点ではメインループで呼ばれる関数に登録されているのは本関数(tcp_connection_open_response_queue())

ここから、別な関数に処理を引き継ぐ形となります

→メインループで呼ばれるのはクライアント通信処理と同じ関数です

8.7.6. サーバ通信処理

ここでは、メインループで、tcp_connection_idle_queue()が呼ばれます。

・tcp.c,（tcp_connection_idle_queue() 抜粋）

```
    else if (tcp_header.tcp_flag & TCP_FLAG_PSH)
    {
        //Push（相手がデータを送ってきた）の場合

        if (tcp_session->push_action == NULL)
        {
            //データを受け取った際のアクション未登録の場合->受け取った旨の返信を行う

            tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK);
        }
        else
        {
            //予め登録済みの関数を実行する（この関数内でACKフラグ付きで返信する）

            tcp_session->push_action();
        }
    }

    tcp_session->time = g_timer_counter;//処理を行った時間を記録
    }
}
```

クライアント動作

サーバ動作

サーバ動作時は、Push フラグのデータを受け取った場合、tcp_connection_open_response()で設定した関数が呼ばれます。

・main.c, main_ether()内

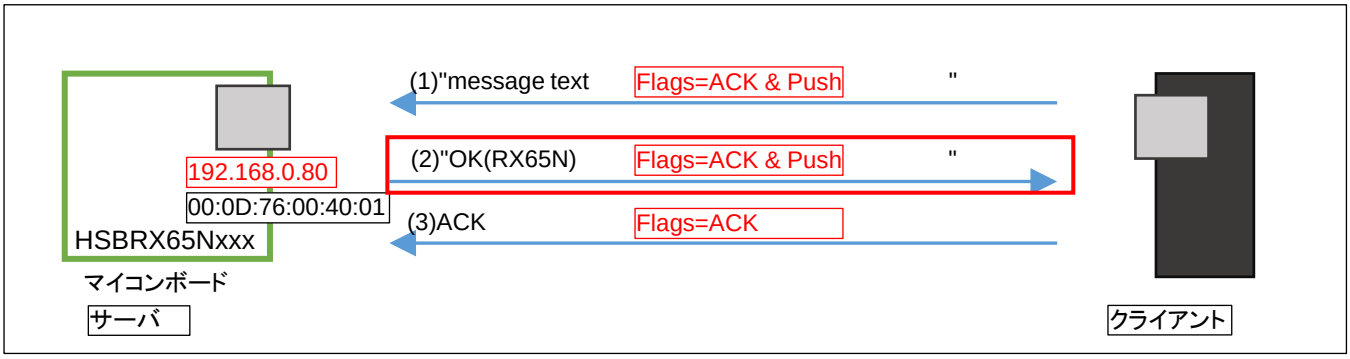
```
//サーバの待ち受け開始
tcp_connection_open_response(g_server_dst_port, &g_tcp_server, TCP_SERVER_NUM, text_packet_response);
```

サーバの待ち受け開始時に、データを受信した際は text_packet_response()関数を呼ぶ様に設定しています。

・text_server.c

```
void text_packet_response(void)
{
    //TCPテキストメッセージ応答（返信）関数

    //戻り値
    //  なし

    //引数
    //  なし

    //補足
    //  g_tcp_server で開かれている接続に対し、g_response_text_data を返送する

    if (g_tcp_server.connect != TCP_CONNECT)
    {
        //TCPの通信が開いていない

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

        return;
    }

    tcp_send_data(&g_tcp_server, g_response_text_data, TCP_MESSAGE_SIZE, TCP_FLAG_ACK | TCP_FLAG_PSH);
//ACKとPUSHフラグを立てて送信
}
```

サーバ動作時は、データを受信した際、返信メッセージ（デフォルトでは、"OK(RX65N)"）を、データ送信関数で送信します。その際、ACK と Push のフラグ付きでの送信となります。



ここで送信されるのは、(2)のパケットです。

8.7.7. サーバ切断処理

メインループで、tcp_connection_idle_queue()が呼ばれている時に、FIN フラグのデータを受信した場合に、切断処理に移行します。

・tcp.c,（tcp_connection_idle_queue() 抜粋）

```
    if (tcp_header.tcp_flag & TCP_FLAG_FIN)
    {
        //相手がFIN(切断)フラグを送ってきた

        sci_write_str("tcp_connection_idle: message: FIN flag received.¥n");

        //通常の切断処理

        tcp_session->sequence = 7;
        tcp_session_function[tcp_session->session_serial] = tcp_connection_close_response_queue;//接続を切
断するシーケンスを実行
    }
```

FIN フラグ受信時は、

・工程を 7

・メインループで実行される関数を tcp_connection_close_response_queue()

に設定します。

・tcp.c

```
void tcp_connection_close_response_queue(TCP_CONNECTION *tcp_session)
{
    //TCP接続の終了を処理する実際の関数

    //戻り値
    //  なし

    //引数
    // *tcp_session : TCP_CONNECT構造体

    //補足
    //  本関数はメインループで何度も呼ばれる事を想定した作り

    /* 工程表
     * (6. 4ウェイハンドシェイクの接続相手からの1個目の受信を待つ（本ボード受信1回目）[既に受信済み])
     * →4ウェイハンドシェイクの接続相手からの1個目の応答に関してはアイドル状態で受信した結果本関数が呼ばれるので、
本関数としては工程7からのスタート
     *
     * 7. 4ウェイハンドシェイクの2個目のパケットを送信する（本ボードから送信1回目）
     * 8. 4ウェイハンドシェイクの3個目のパケットを送信する（本ボードから送信2回目）
     * 9. 4ウェイハンドシェイクの接続相手からの4個目の応答を待つ（本ボード受信2回目）
     * 10.終了処理
     * 99.タイムアウト処理
     */

    int ret;

    TCP_HEADER tcp_header;

(タイムアウト処理とウェイト処理、省略)
```

・tcp.c,（tcp_connection_close_response_queue() 続き）

```
    switch (tcp_session->sequence)
    {
        case 7:
            //4ウェイハンドシェイクの2個目のフレームを送信する（1回しか来ない）

            tcp_session->ack_no += 1UL;//応答確認番号は前回相手が送ってきた値+1

            tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK);//ACKフラグ送信

            tcp_session->timeout = g_timer_counter + 10000;//4ウェイハンドシェイクの相手の応答タイムアウト
（10秒）

            tcp_session->sequence = 8;//次の工程に進む
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 7

```
        case 8:
        //4ウェイハンドシェイクの3個目のフレームを送信する（1回しか来ない）

            tcp_connection_flag_send(tcp_session, TCP_FLAG_ACK | TCP_FLAG_FIN);//ACKとFINフラグ送信

            tcp_session->sequence = 9;//次の工程に進む
            tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 8

```
        case 9:
            //4ウェイハンドシェイクの4個目となる応答フレーム(2)を受信する（何度も来る可能性あり）

            ret = tcp_recv_data_dequeue(tcp_buf, tcp_session->session_serial);//受信リングバッファからデータ
読み出し
            if (ret != 0)
            {
                stream_to_TCP_header(&tcp_buf[14 + 20], &tcp_header);//受信したストームをTCPヘッダに分解
                if (tcp_header.tcp_ack_no != (tcp_session->seq_no + 1UL))//こちらが送信したシーケンシャル番号
+1が相手が返す確認応答番号の期待値
                {
                    //確認応答番号が一致しない
（エラー表示、省略）
                }
                if (!(tcp_header.tcp_flag & TCP_FLAG_ACK))
                {
                    //ACKフラグが立っていない場合
                    sci_write_str("tcp_connection_open: error: handshake ACK flag is not set.¥n");
                    break;
                }

                sci_write_str("TCP handshake ACK flag received OK.¥n");

                tcp_session->seq_no += 1UL;//次に送るシーケンス番号は+1※最後のパケットなので今後こちらから送る
事は無いが値は記録しておく
                tcp_session->ack_no = tcp_header.tcp_seq_no + 1UL;//次に送る確認応答番号は相手のシーケンシャ
ル番号+1※最後のパケットなので今後こちらから送る事は無いが値は記録しておく
                tcp_session->sequence = 10;//次の工程に進む
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
            }
            break;
```

工程 9

```
        case 10:
                //終了処理（1回しか来ない）
                sci_write_str("-- CONNECTION CLOSED --¥n");

                tcp_session->connect = TCP_DISCONNECT;//切断
                tcp_session_function[tcp_session->session_serial] = NULL;//メイン関数のループ処理関数の登録を
解除
                tcp_session->sequence = 0;//工程はリセット
                tcp_session->time = g_timer_counter;//処理を行った時間を記録
            break;
```

工程 10

```
（タイムアウト処理、後略）
```

工程 7（1 回しか実行されない、相手に依存しない）

・ACK フラグの送信

・工程 8 に進む

工程 8（1 回しか実行されない、相手に依存しない）

・FIN & ACK フラグの送信

・工程 9 に進む

工程 9（複数回呼ばれる、相手からの通信を待つ工程）

・ACK フラグの受信待ち

・工程 10 に進む

工程 10（1 回しか実行されない、相手に依存しない）

・状態変数のセット

・メインループで実行される関数を空にする

→メインループ内で切断状態を検出して、再度待ち受け状態にする関数(tcp_connection_open_response())が呼ばれます

8.7.1. 工程管理のまとめ

工程番号	内容	クライアント動作時	サーバ動作時
0	初期値	起動後 c コマンド入力前	起動後、待ち受け関数実行で 2 に移行
1	MAC アドレス解決	ARP リクエスト送信 ARP リプライ受信待ち	なし
2	3 ウェイハンドシェイク(1)	パケット送信	パケット受信待ち
3	3 ウェイハンドシェイク(2)	パケット受信待ち	パケット送信
4	3 ウェイハンドシェイク(3)	パケット送信	パケット受信待ち
5	接続が確立された状態	s コマンド入力待ち	データ受信待ち
6	4 ウェイハンドシェイク(1)	パケット送信	なし（5 から 7 に移行）
7	4 ウェイハンドシェイク(2)	パケット受信待ち	パケット送信
8	4 ウェイハンドシェイク(3)	パケット受信待ち	パケット送信
9	4 ウェイハンドシェイク(4)	パケット送信	パケット受信待ち
10	終了処理	変数のクリアなど	変数のクリアなど
99	タイムアウト	タイムアウト処理	タイムアウト処理

接続の工程が 1~4、切断の工程が 6~10 です。

接続が完了してデータの送受信が可能なのは、工程 5 です。

・クライアント動作時

工程番号	トリガ	メインループで実行される関数
0→1	c コマンドの入力 tcp_connection_open_request()の実行	tcp_connection_open_request_queue() [工程 1-4]
1→2	MAC アドレスの解決	
2→3	機械的に進む （3 ウェイハンドシェイク(1)パケットの送信後）	
3→4	3 ウェイハンドシェイク(2)受信	
4→5	機械的に進む （3 ウェイハンドシェイク(3)パケットの送信後）	tcp_connection_idle_queue() [工程 5]
5→6	d コマンドの入力 tcp_connection_close_request()の実行	tcp_connection_close_request_queue() [工程 6-10]
6→7	機械的に進む （4 ウェイハンドシェイク(1)パケットの送信後）	
7→8	4 ウェイハンドシェイク(2)の受信	
8→9	4 ウェイハンドシェイク(3)の受信	
9→10	機械的に進む （4 ウェイハンドシェイク(4)パケットの送信後）	
10→0	機械的に進む	実行されない[工程 0]

・サーバ動作時

工程番号	トリガ	メインループで実行される関数
0→2	メイン関数で tcp_connection_open_response()の実行	tcp_connection_open_response_queue() [工程 2-4]
2→3	3 ウェイハンドシェイク(1)受信	
3→4	機械的に進む （3 ウェイハンドシェイク(2)パケットの送信後）	
4→5	3 ウェイハンドシェイク(3)受信	tcp_connection_idle_queue()[工程 5]
5→7	4 ウェイハンドシェイク(1)の受信	tcp_connection_close_response_queue() [工程 7-10]
7→8	機械的に進む （4 ウェイハンドシェイク(2)パケットの送信後）	
8→9	機械的に進む （4 ウェイハンドシェイク(3)パケットの送信後）	
9→10	4 ウェイハンドシェイク(2)の受信	
10→0	機械的に進む	実行されない[工程 0]

接続が完了している状態では、サーバ・クライアント共、メイン関数内で tcp_connection_idle_queue()が実行されます。接続と切断の過程では、サーバとクライアントで別な関数が実行されます。（接続と切断時は、要求を出す側と、要求に応答する側で、送信と受信のタイミングが逆になるので別関数としています。）

－TCP で使用されるユーザ関数－

・text_client.c

関数名	内容
text_packet_send	TCP データの送信

・text_server.c

関数名	内容
text_packet_response	TCP データの送信

・tcp.c

関数名	内容
tcp_recv_data_enqueue	TCP 向けのデータバッファへの格納
tcp_recv_data_dequeue	TCP 向けのデータバッファからの読み出し
tcp_connection_open_request	接続（クライアント向け）
tcp_connection_close_request	切断（クライアント向け）
tcp_connection_force_close_request	強制切断
tcp_connection_open_response	接続待ち（サーバ向け）
tcp_send_data	データ送信

・tcp.c　※ユーザ実行関数ではなく、メイン関数内のメインループ内で実行される関数

関数名	内容
tcp_connection_open_request_queue	接続要求時に実行される関数（クライアント）
tcp_connection_close_request_queue	切断要求時に実行される関数（クライアント）
tcp_connection_open_response_queue	接続要求時に実行される関数（サーバ）
tcp_connection_close_response_queue	切断要求時に実行される関数（サーバ）
tcp_connection_idle_queue	接続後に実行される関数（クライアント、サーバ共）

ここで、TCP 構造体に関してもまとめとして記載します。

・tcp.h

```
typedef struct
{
    //TCP接続状態
    unsigned short connect;//0:未接続，1:接続試行中，2:接続中
    unsigned short sequence;//工程表の何処にいるか
    unsigned long seq_no;//シーケンシャル番号
    unsigned long ack_no;//確認応答番号
    unsigned char src_MAC[6];//本ボードのMACアドレス
    unsigned char dst_MAC[6];//接続先のMACアドレス
    unsigned char src_IP[4];//本ボードのIPアドレス
    unsigned char dst_IP[4];//接続先のIPアドレス
    unsigned short src_port;//本ボード側の使用ポート
    unsigned short dst_port;//接続先のポート
    unsigned short id;//TCPヘッダ内のID（本プログラムでは送信毎に単純にインクリメントする）
    unsigned short session_serial;//セッション毎のシリアル番号（メイン関数に登録する関数番号）
    void (*push_action)();//データを受信した際のアクションの関数ポインタ（クライアント:何もしない（画面表
示のみ），サーバ:相手にOKを返す）
    uint64_t time;//処理が行われた時間を記録
    uint64_t timeout;//接続タイムアウト（未来の時間を設定，設定時間が来るとタイムアウト）0:タイムアウト処
理を行わない
    uint64_t wait_time;//ウェイト（未来の時間を設定，設定時間が来るまでは実行をスキップする）
}TCP_CONNECTION;
```

メンバ名	内容	値の例
connect	接続状態を表す	TCP_DISCONNECT　→切断 TCP_CONNECT_ATTEMPT　→接続試行中 TCP_CONNECT　→接続中
sequence	工程表の番号	1　→ARP リクエスト送信 5　→接続確立中通信データ待ち　など
sec_no	シーケンス番号	0x12345678
ack_no	応答確認番号	0x12345678
src_MAC	本ボードの MAC アドレス	{0x00, 0x0D, 0x76, 0x00, 0x40, 0x01}
dst_MAC	通信相手の MAC アドレス クライアント動作時は ARP リクエストで調べる サーバ動作時は相手が送ってきた値を保持	{0x00, 0x0D, 0x76, 0x00, 0x40, 0x02}

メンバ名	内容	値の例
src_MAC	本ボードの IP アドレス	{192, 168, 0, 80}
dst_MAC	通信相手の IP アドレス	{192, 168, 0, 81}
id	TCP ヘッダに含める ID 値 送信毎に単純にインクリメント	0x0001
session_serial	TCP セッションの管理番号(*1)	TCP_CLIENT_NUM(=0) TCP_SERVER_NUM(=1) のいずれか
(*push_action)()	Push フラグを受け取った際のアクションを登録(*2)	NULL →クライアント動作時 text_packet_response →サーバ動作時
time	最後に処理を行った時間	g_timer_counter
timeout	タイムアウト時間の設定(*3)	g_timer_counter + 10000
wait_time	待ち時間の設定(*4)	g_timer_counter + 500

(*1)本プロジェクトでは、クライアント動作用とサーバ動作用に 0, 1 を割り当てて使用。

サーバを複数ポートで同時に多数のセッションの待ち受けをする、クライアントのセッション数を増やして複数のサーバに同時に接続できるようにするなどの拡張を行う際には、セッション毎に重複しない番号を割り振って、セッションを管理する必要があります。

(*2)相手から Push フラグを受け取った際の動作を設定する部分です。

本プロジェクトでは、以下の様にしています。

クライアント動作時は、ACK を返すだけとしているので、NULL を指定しています。

サーバ動作時は、"OK(RX65N)"というデータを送信する動作としているので、その関数(text_packet_response)を指定しています。ここで指定するのは、関数名です。

(*3)例えば工程 1 から工程 5 までのタイムアウトを 10 秒に設定する場合、工程 1 の部分で、timeout=g_timer_counter + 10000 を代入します。また、工程 5 の部分で、timeout=0 を代入します。指定する単位は、[ms]です。g_timer_counter は、現時点の時間です。timeout に指定するのは未来の時間です。timeout=0 はタイムアウトなしです。

(*4)例えば、何か処理を実行して、500ms 待ってから次のアクションに進みたいという場合は、wait_time=g_timer_counter + 500 を代入します。メインループで連続的に実行される関数内で、g_timer_counter（1ms 毎にインクリメント）が wait_time を超えた場合に、処理が進みます。wait_time は未来の時間を指定します。0（または、現在の時間や過去の時間）を指定すると、ウェイトなしです。

8.8. ユーザ作成関数

tcp_recv_data_enqueue

概要：TCP データの一時保存
宣言：tcp_recv_data_enqueue(unsigned char **src_address, unsigned short size, unsigned short num)
説明：
・TCP で受信したデータの一時保存
を行います
引数：
　unsigned char **src_address: データ格納先頭アドレスポインタ
　unsigned short size: データサイズ
　unsigned short num: バッファ番号（TCP セッション毎）
戻り値：
　0: 正常終了（データを保存）
　-4: 引数エラー
　-13: バッファフル
補足：メイン関数内の受信ルーチン内で TCP ヘッダ内の宛先ポートに応じて、本関数で受信データを一時的に保存
　　tcp_connection_???_queue()関数内で受信データの参照を行います
　　本プロジェクトでは、クライアントのデータはセッション番号 0（バッファ番号 num=0）のバッファに格納されます
　　サーバのデータはセッション番号 1（バッファ番号 num=1）のバッファに格納されます
　　TCP セッション毎に、バッファが別となります（本プロジェクトでは 2 セッションまで取り扱う）

tcp_recv_data_dequeue

概要：一時保存 TCP データの参照
宣言：int tcp_recv_data_dequeue(unsigned char *data, unsigned short num)
説明：
・tcp_recv_data_enqueue で保存したデータの参照、破棄
を行います
引数：
　unsigned char *data: データのアドレス
　unsigned short num: バッファ番号（TCP セッション毎）
戻り値：
　0: 一時保存されたデータなし
　>0: 一時保存されているデータのバイト数
補足：一時保存バッファはリングバッファとして構成されており、本関数実行時に読み出し済みキューが進みます
　　（読み出し済みとして処理）

tcp_connection_open_request

概要:TCP 接続
宣言: int tcp_connection_open_request(unsigned char *IP, unsigned short port, TCP_CONNECTION *tcp_session, unsigned short session_serial)
説明:
・TCP 接続の開始を要求する関数を実行キューに登録
を行います
引数:
unsigned char *IP: 接続先 IP アドレス
unsigned short port: 接続先ポート
TCP_CONNECTION *tcp_session: TCP セッション構造体
unsigned short session_serial: TCP セッションのシリアル番号
戻り値:
0: 正常終了
-1: 接続済み、または接続試行中
補足:クライアント側が呼び出す事を想定した関数です

tcp_connection_close_request

概要:TCP 切断
宣言:int tcp_connection_close_request(TCP_CONNECTION *tcp_session)
説明:
・TCP 切断処理の開始を要求する関数を実行キューに登録
を行います
引数:
TCP_CONNECTION *tcp_session: TCP セッション構造体
戻り値:
0: 正常終了
-1: 接続済み、または接続試行中
補足:クライアント側が呼び出す事を想定した関数です

tcp_connection_force_close_request

概要:TCP 強制切断
宣言:int tcp_connection_force_close_request(TCP_CONNECTION *tcp_session)
説明:
・TCP 接続の強制的な切断
を行います

引数:
TCP_CONNECTION *tcp_session: TCP セッション構造体
戻り値:
0: 正常終了
補足:接続相手に Reset フラグを立てたパケットを投げる関数です

tcp_connection_open_response

概要:TCP 接続待ち受け
宣言:int tcp_connection_open_response(unsigned short port, TCP_CONNECTION *tcp_session, unsigned short session_serial, void (*function)()
説明:
・TCP 接続の待ち受け動作の関数を実行キューに登録
を行います
引数:
port: 待ち受けポート
TCP_CONNECTION *tcp_session: TCP セッション構造体
unsigned short session_serial: TCP セッションのシリアル番号
void (*function)(): 接続後にデータを受信した際に実行される関数
戻り値:
0: 正常終了
補足:サーバ側が呼び出す事を想定した関数です
第 4 引数は、接続完了した後、データ受信した際にどのようなアクションを行うかの関数を指定する引数です
本プロジェクトでは、text_packet_response（応答メッセージを返信する関数）を与えています

tcp_send_data

概要:TCP データ送信
宣言: void tcp_send_data(TCP_CONNECTION *tcp_session, unsigned char *data, unsigned short size, unsigned short tcp_flag)
説明:
・TCP でデータの送信
・TCP フラグの送信
を行います
引数:
TCP_CONNECTION *tcp_session: TCP セッション構造体
unsigned char *data: 送信データ
unsigned short size: 送信サイズ
unsigned short tcp_flag; TCP フラグ

戻り値:なし

補足:データは空、フラグのみ送信する場合は*data=NULL, size=0 で本関数を実行してください

8.9. まとめ

TCP でのデータ送信は、UDP に対して非常に煩雑となります。

通信前に接続処理がある事やシーケンス番号と応答確認番号の管理、フラグの返送（パケットを受信した際に ACK を返す必要がある）など、UDP でのデータ送信（準備は不要で 1 つのパケットを送出するだけ）と比べると TCP は面倒だという事が（本マニュアルのボリュームを見るだけでも）伝わるかと思います。

なお、本プロジェクトの TCP は（結構なことをやっていますが）あくまで簡易的な TCP プロトコルを実装しているに過ぎません。TCP では、本プロジェクトでは実装していない、パケットの順番が入れ替わったケースの並び替えやパケットが届かなかった場合の再送などの機能があります。

次に示すプロジェクトでは、TCP や UDP のプロトコルスタックを使用した内容となります。

（UDP であれば、ユーザプログラムで実装しても良いかとも思いますが、TCP で実用的なプログラムを作成する場合は、プロトコルスタックの使用が前提となるのではないかと思います。）

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 スタータキット RX65N
ソフトウェア編　取扱説明書(2)

株式会社 北斗電子
 2026 年 6 月 8 日改訂 REV.1.0.0.0 (260417)