Ethernet スタータキット RX65N
ソフトウェア編　取扱説明書(1)

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

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

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

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

1.1. RX65N マイコン向けサンプルプログラム

マイコン向けのサンプルプログラムは、CS+のプロジェクト形式としています。
e2studio を使用する場合は、ワークスペースにインポートを行って下さい。

CS+forCC(V8.15)以降
（e2studio　2025-12 以降）
RX スマート・コンフィグレータ　2.28.0 以降
RX Family RX Driver Package　Ver.1.50 以降（アプリケーションノート　R01AN8254）
　r_lwip_driver_rx_v1.11 以降
　r_lwip_rx_v1.11 以降

RX Family RX Driver Package(RX-FIT)の、Ver1.50 以降には、
RX Family Lightweight IP (lwIP) Driver Using Firmware Integration Technology (r_lwip_driver_rx)
RX Family Lightweight IP (lwIP) Module Using Firmware Integration Technology (r_lwip_rx)
が含まれますので、RX Family RX Driver Package(RX-FIT)をダウンロード、インストールすれば必要なものは揃います。

※RX Family RX Driver Package Ver1.49 以前を使用する場合
RX Family Lightweight IP (lwIP) Module Using Firmware Integration Technology Rev.1.11 以降
　(r_lwip_driver_rx)（アプリケーションノート　R20AN0788）
RX Family Lightweight IP (lwIP) Driver Using Firmware Integration Technology Rev.1.11 以降
　(r_lwip_rx)（アプリケーションノート　R20AN0789）
を、別途インストールしてください。

1.2. PC 向けサンプルプログラム

PC 向けのサンプルプログラムは、VisualStudio2026, C#, .NET10 で作成しています。（特段バージョンに依存する様な点はないので、古いバージョンの ViualStudio や.NET でも問題はないかと思います。）

プログラムをビルドする場合は、VisualStudio2026, C#, .NET10 の開発環境をインストールしてください。

プログラムの実行に関しては、.NET10 のランタイムをインストールしてください。

ーPC ソフトの実行に必要なランタイムに関してー

CD に格納されている、PC_APPLI 以下のソフトは、C#, .NET10 で作成されており、実行するためには、.NET10 のランタイムライブラリが PC にインストールされている必要があります。（OS は、Windows11 か Windows10 に対応）

「.NET10 ランタイム」で検索し、Microsoft の Web ページより「.NET デスクトップランタイム」(x64)をダウンロードしてインストールしてください。

2. RX65N_MAGIC_PACKET プロジェクト

RX マイコン向けのプロジェクトは、CD 内の

SOURCE¥144_100PIN¥RX65N_MAGIC_PACKET
SOURCE¥176PIN¥RX65N_MAGIC_PACKET

に格納されています。
使用しているマイコンボードが、HSBRX65N176（176 ピンのマイコンが搭載されたボード）の場合は、SOURCE¥176PIN 以下のフォルダを PC のストレージにコピーしてください。
HSBRX65N144A（144 ピンのマイコンが搭載されたボード）, HSBRX65N100A（100 ピンのマイコンが搭載されたボード）の場合は、SOURCE¥144_100PIN 以下のフォルダを PC のストレージにコピーして使用してください。

（マイコンのピン数により、使用している端子が異なりますので、プロジェクトが別になっています。）
（どちらのプロジェクトでも、使用端子以外の違いはありません。）

本プロジェクトでは、指定した MAC アドレスの機器に対して MagicPaket を送信します。

起動後、以下の様なメッセージが表示されます。
（端末は、115,200bps で開いてください。）

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

COMMAND:
   m : target MAC address set
   p : print MAC address
   S : swap [this board MAC address] <-> [target MAC address]
---
   s : Operation start with send mode
   r : Operation start with receive mode
---

USAGE:
   SW3 : MagicPacket send
   LED : MagicPacket received

-default setting address-
---
target MAC address is        -> 00-0D-76-00-40-02 : m command for change
this board MAC address is -> 00-0D-76-00-40-01 : S command for swap
---
First m, S, command -> initial setting

Please input 's' or 'r' for operation start!
>
```

最初に、必要に応じて m か S コマンドで、MAC アドレスの設定を行います。

デフォルトは、

送信先 MAC アドレス　00-0D-76-00-40-02

本ボードの MAC アドレス　00-0D-76-00-40-01

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 MAC アドレスが逆に設定されます。)

m コマンドを入力すると

```
>command=m
target MAC address set(please input 2 letters(0-9,a-f,A-F) hex x 6)

MAC[0] >
```

の表示となりますので、MagicPacket 送信先の MAC アドレスを 2 桁ずつ 6 個入力してください。

S コマンドは、送信先と本ボードの MAC アドレスを入れ替える動作を行います。

アドレス確定後、s または r コマンドで動作開始です。

s コマンドで動作開始

```
>command=s

Operation start with MagicPacket send mode.
PUSH SW3 -> send MagicPacket
Ether0: LINK-UP
```

→送信モード、ボードの SW3 を押すと MagicPacket 送信

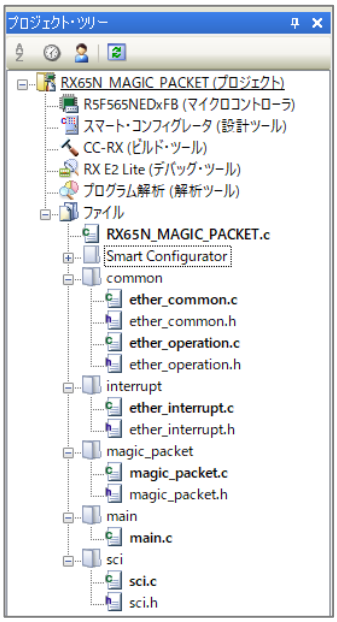
r コマンドで動作開始

```
>command=r

Operation start with MagicPacket receive mode.
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-UP
-> MagicPacket waiting ...
```

→受信モード、MagicPacket が外部から送信されるのを待機

2.1. プロジェクトツリー

プロジェクトのソースコードは、以下の様になっています。



RX65N_MAGIC_PACKET.c
→プロジェクト名.c　main 関数を呼び出すだけの内容としています

Smart Configurator 以下
→スマート・コンフィグレータ（ツール）で生成されるソースコード

common 以下
→IP アドレスの設定や共通で使用するソース

interrupt 以下
→割り込み関数を記載

magic_packet 以下
→MAGIC PACKET(WakeOnLan)の処理

main 以下
→main.c が含まれます、プロジェクトのメイン関数に相当するソースコードです

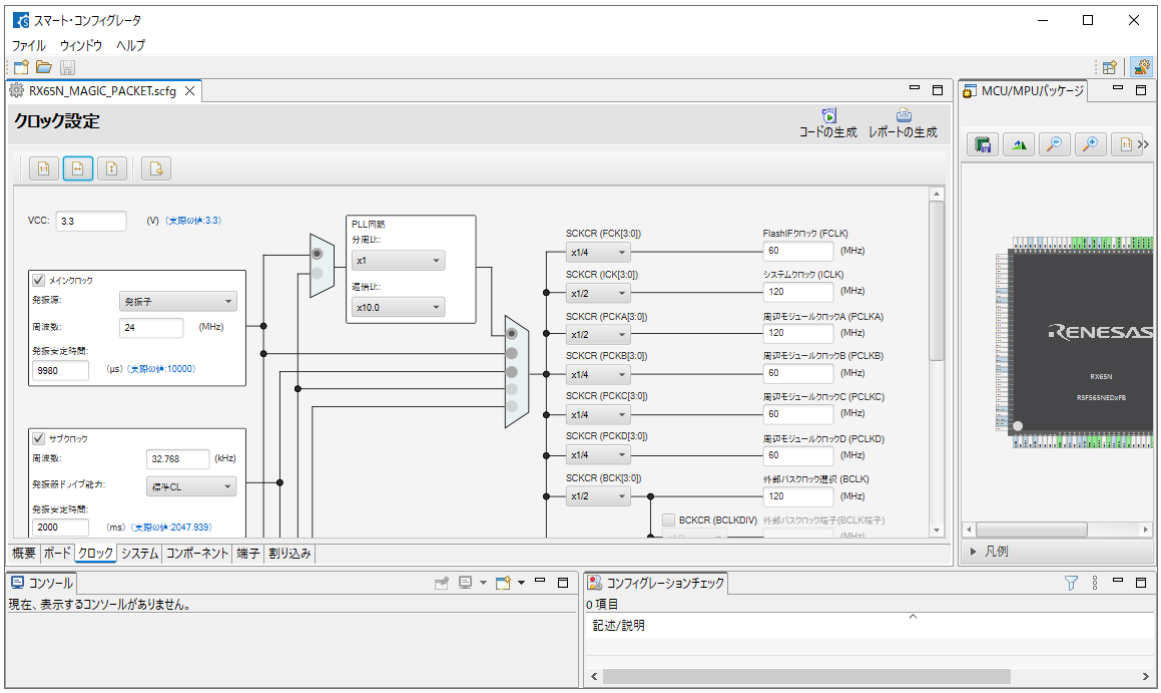
sci 以下
→端末にメッセージを表示、及びキーボードからの入力を読み取る処理

common, interrupt, magic_packet, main. sci 以下に含まれるのが、ユーザ作成のソースコード。Smart Configurator 以下は、スマート・コンフィグレータ(GUI)で設定した内容がソースコードに反映される形です。

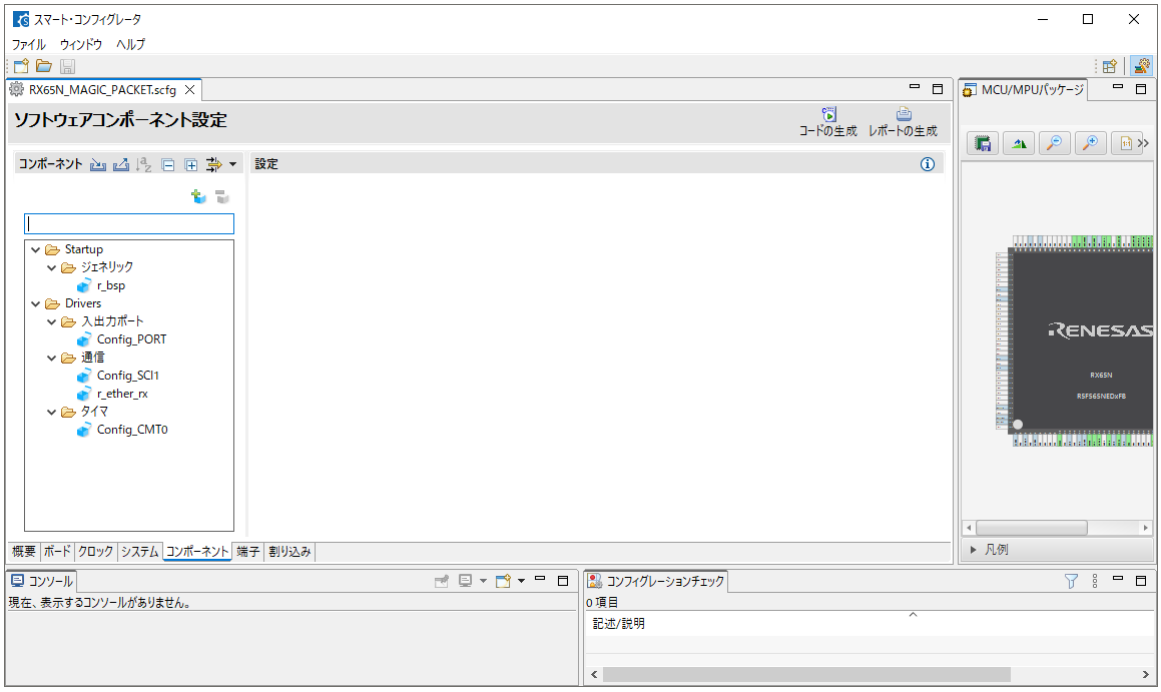
2.2. スマート・コンフィグレータ

スマート・コンフィグレータでは、クロックの設定や、マイコン内蔵の周辺機能の設定などを行います。

・クロック設定

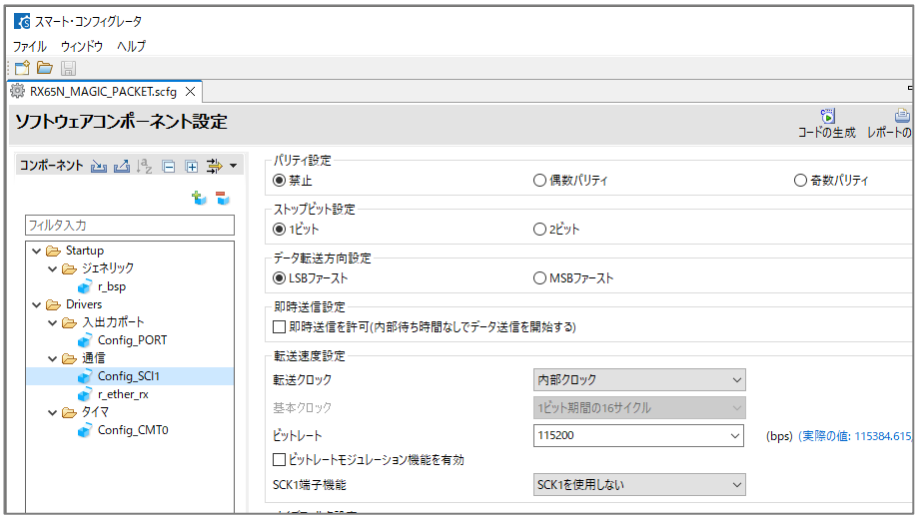


・コンポーネント（周辺機能）設定



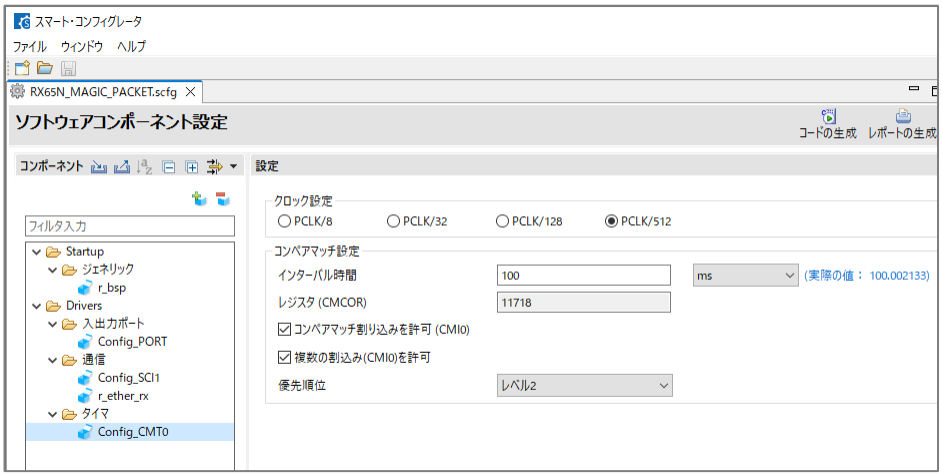
使用しているコンポーネントは、
r_bsp （最初から追加されている）
Config_PORT　I/O 端子の設定、ここではプッシュスイッチと LED ポートの設定を行っています
Config_SCI1　SCI(UART)の設定
r_ether_rx　Ethernet の設定
Config_CMT0　コンペアマッチタイマ（タイマによる定期処理）の設定
です。

例えば、Config_SCI1 は、



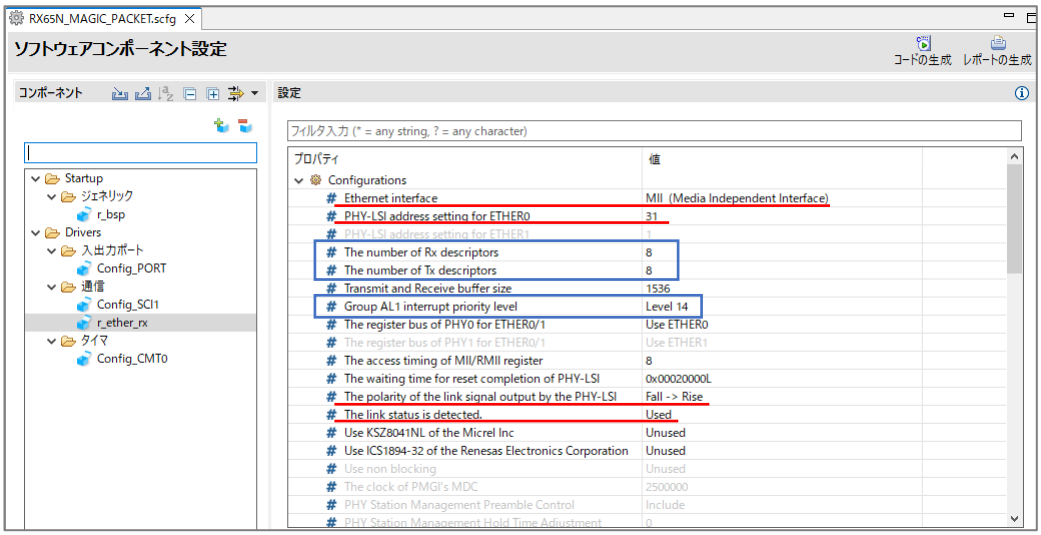
上記の様になっており、SCI の速度設定（上記では、115,200bps）などが、GUI で設定できるようになっています。
（使いたい機能を追加して設定するだけで、マイコンの周辺機能の初期化や動作させる関数が生成される。）

・タイマの設定



CMT（コンペアマッチタイマ）は、上記の様に設定しており、100ms 毎にタイマ割り込みが掛り、定期的な処理が実行されます。
（スマート・コンフィグレータを使用しない場合は、PCLKB の周波数から 100ms が何カウントであるかを計算して、11718 という値を自分でレジスタに設定する必要があります。スマート・コンフィグレータなら、100ms という値を与えるだけですので、比較的簡単にマイコンの機能を使う事ができます。）

・Ether の設定



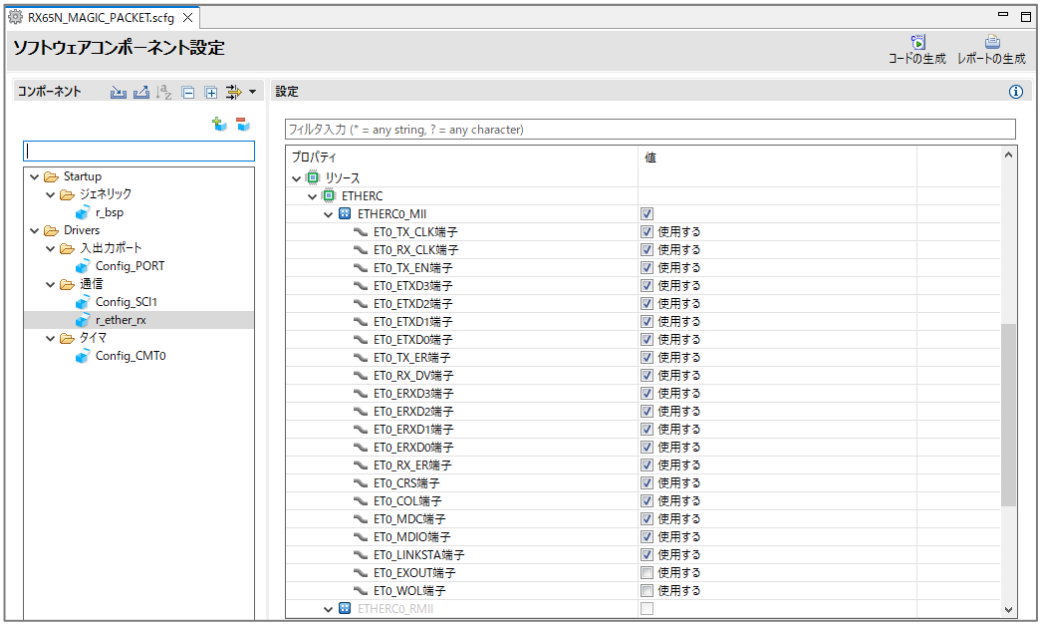
設定項目	設定値	備考
Ethernet interface	MII (*)	ボードの設計で MII を選ぶか RMII を選ぶが決まる
PHY-LSI address	31 (*)	PHY アドレスは 31（ボードの設計で決まる項目）
The number of Rx descripters	8	RX ディスクリプタ数（一例、任意の数を設定可能）
The number of Tx descripters	8	TX ディスクリプタ数（一例、任意の数を設定可能）
Group AL1 interrupt priority level	14	割り込み優先度（一例、1～15 の値を設定可能）
The polarity of the link signal	Fall->Rise (*)	接続している PHY-LSI の仕様で決まる
The link status is detected	Used (*)	ボードの設計で決まる（PHY-LSI の LINKSTA 出力を使用）

(*)当社のボードを使う場合は、この値を選択してください

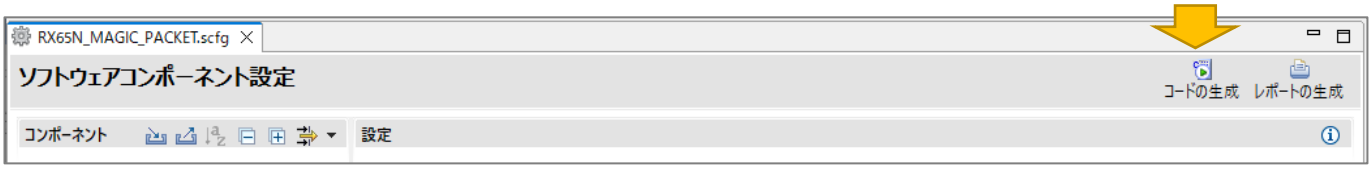
ディスクリプタに関しては後述します。

割り込み優先度は、15（最高優先度）から 1（最低優先度）まで、どの処理を優先させるかで選びます。

（上記画面の続き）

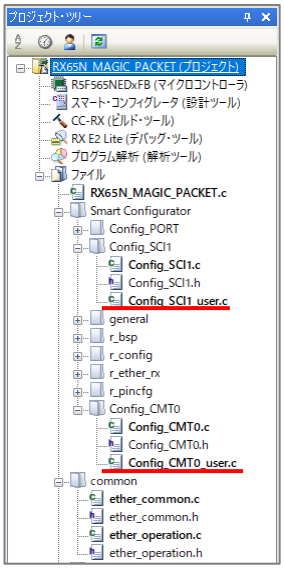


Ether でどの端子を使用するかの設定画面です。ボードの設計に依存する部分です。基本的には上記設定としてください。(ET0_EXOUT, ET0_WOL 端子以外にチェックを入れる)



スマート・コンフィグレータでコンポーネントの追加や設定の変更を行った場合は、「コード生成」のボタンを押してください。(この際に、設定がソースコードに反映されます。)

2.2.1. スマート・コンフィグレータ生成コードの変更



スマート・コンフィグレータ生成ファイルの中に、
Config_SCI1_user.c
の様に_user が付くファイルがあります。これらのファイルは、ユーザ側でコードを追加しても良いファイルとなります。

・Config_SCI1_user.c

```
/***********************************************************************************************
**********************
Includes
***********************************************************************************************
**********************/
#include "r_cg_macrodriver.h"
#include "Config_SCI1.h"
/* Start user code for include. Do not edit comment generated here */
#include "sci.h"
/* End user code. Do not edit comment generated here */
#include "r_cg_userdefine.h"
```

（中略）

```
/***********************************************************************************************
***************
* Function Name: r_Config_SCI1_callback_transmitend
* Description  : This function is a callback function when SCI1 finishes transmission
* Arguments    : None
* Return Value : None
***********************************************************************************************
**************/

static void r_Config_SCI1_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI1_callback_transmitend. Do not edit comment generated here */
    intr_sci_send_end();
    /* End user code. Do not edit comment generated here */
}

/***********************************************************************************************
***************
* Function Name: r_Config_SCI1_callback_receiveend
* Description  : This function is a callback function when SCI1 finishes reception
* Arguments    : None
* Return Value : None
***********************************************************************************************
**************/

static void r_Config_SCI1_callback_receiveend(void)
{
    /* Start user code for r_Config_SCI1_callback_receiveend. Do not edit comment generated here */
    intr_sci_receive_end();
    /* End user code. Do not edit comment generated here */
}

/***********************************************************************************************
***************
* Function Name: r_Config_SCI1_callback_receiveerror
* Description  : This function is a callback function when SCI1 reception encounters error
* Arguments    : None
* Return Value : None
***********************************************************************************************
**************/

static void r_Config_SCI1_callback_receiveerror(void)
{
    /* Start user code for r_Config_SCI1_callback_receiveerror. Do not edit comment generated here */
    intr_sci_receive_error();
    /* End user code. Do not edit comment generated here */
}
```

`/* Start user code`

`/* End user code.`

の間に、赤字で書いたコードを追加しています（合計 4 行）。

※Start-End で囲まれた以外のところに書くと、「コード生成」ボタンを押した際に、上書きされて消されてしまいます

Config_SCI1_user.c には、UART(SCI)で端末への情報表示に使用しているコードを追加しています。

同様にコンペアマッチタイマのコードには、以下を追加しています。

・Config_CMT0_user.c

```
/***********************************************************************************************
**********************
Includes
***********************************************************************************************
*********************/
#include "r_cg_macrodriver.h"
#include "Config_CMT0.h"
/* Start user code for include. Do not edit comment generated here */
#include "ether_interrupt.h"
/* End user code. Do not edit comment generated here */
#include "r_cg_userdefine.h"
```

（中略）

```
static void r_Config_CMT0_cmi0_interrupt(void)
{
    /* Set bit PSW.I = 1 to allow multiple interrupts */
    R_BSP_SETPSW_I();

    /* Start user code for r_Config_CMT0_cmi0_interrupt. Do not edit comment generated here */
    ether_interrupt_cmt0();
    /* End user code. Do not edit comment generated here */
}
```

SCI1, CMT0 のどちらも、割り込みが掛った際にユーザ側で作成した関数を呼ぶためのコード追加です。

上記以外のスマート・コンフィグレータ生成コードには手を入れる必要はありません。

2.3. ユーザ側で作成したプログラムコード

フォルダ（カテゴリ）	ファイル名	備考
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
interrupt	ether_interrupt.c	割り込み関数
	ether_interrupt.h	上記ヘッダ
magic_packet	magic_packet.c	MagicPacket(WOL)を処理する関数
	magic_packet.h	上記ヘッダ
main	main.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ

スマート・コンフィグレータ生成コード以外に、ユーザ側で作成してプロジェクトに追加したファイルは上記です。

・ether_common.c

本ファイル内では、

```
void ether_user_init(void);
void ether_eesr_error_print(unsigned long eesr);

void sci_print_IP(unsigned char *IP);
void sci_print_MAC(unsigned char *MAC);

void sci_read_IP(unsigned char *IP);
void sci_read_MAC(unsigned char *MAC);

int IP_compare(unsigned char *IP_a, unsigned char *IP_b);
int MAC_compare(unsigned char *MAC_a, unsigned char *MAC_b);

int sci_read_numeric_value(unsigned long *input, unsigned short digit);
int sci_read_hex_value(unsigned char *input);
int sci_read_one_char(char *input, char *valid_chars);
```

上記の関数が定義されています。

関数名	内容
ether_user_init	変数の初期化等
ether_eesr_error_print	エラー発生時の表示
sci_print_IP	IP アドレスの表示（本プロジェクトでは IP アドレスは取り扱いません：未使用）
sci_print_MAC	MAC アドレスの表示
sci_read_IP	IP アドレスのキーボードからの読み取り（本プロジェクトでは IP アドレスは取り扱いません：未使用）
sci_read_MAC	MAC アドレスのキーボードからの読み取り
IP_compare	IP アドレスの比較（本プロジェクトでは IP アドレスは取り扱いません：未使用）
MAC_compare	MAC アドレスの比較
sci_read_numeric_value	端末から数値の読み取り
sci_read_hex_value	端末から 16 進数の読み取り
sci_read_one_char	端末から 1 文字読み取り

本プロジェクト（MagicPacket の送受信）とは、直接関係がなく、今後も共通で使用する関数などが、ether_common.c 内で定義されています。

・ether_operation.c

本ファイル内では、

```
//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

//MagicPacket送信相手のMACアドレス
unsigned char g_dst_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x02};//（仮値）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

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

MAC アドレスや使用する Ether のチャンネル（RX65N は ch0 しかありません）等を定義しています。

本プロジェクトでは、IP アドレスは取り扱いませんが、今後のプロジェクトでは IP アドレスもこのファイル内で定義します。接続するネットワークの構成に合わせて変更する必要があるファイルです。

・ether_interrupt.c

本ファイル内では、

```
void ether_callback(void*);
void ether_int_handler(void*);

void ether_interrupt_cmt0(void);
```

上記の関数が定義されています。

関数名	内容
ether_callback	Ether のコールバック関数（詳細は後述）
ether_int_handler	Ether の割り込みハンドラ（詳細は後述）
ether_interrupt_cmt0	CMT0 タイマ割り込み

割り込みで呼ばれる関数を記載したファイルとなります。割り込みで処理される内容に関しては後述します。

・magic_packet.c

本ファイル内では、

```
void magic_packet_send(unsigned char *MAC);
```

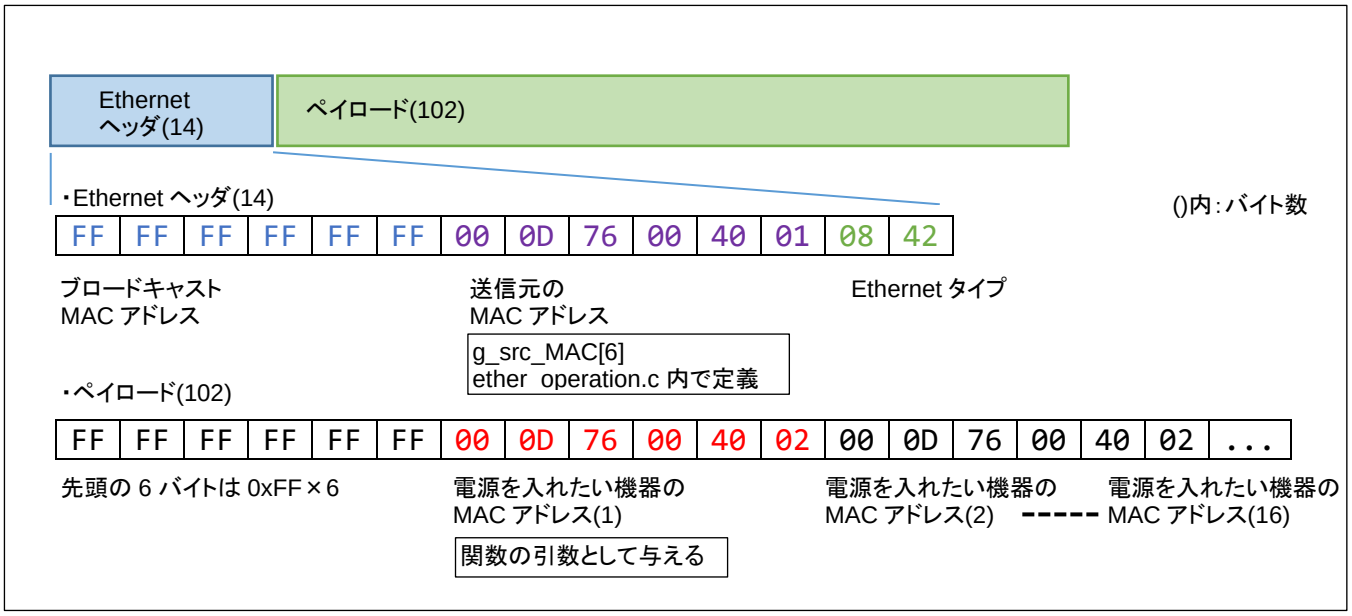
上記の関数が定義されています。

関数名	内容
magic_packet_send	MagicPacket（WOL パケット）の送信

magic_packet_send()

```
void magic_packet_send(unsigned char *MAC)
{
    //MagicPacket(112bytes)を送信する関数

    //戻り値
    //  なし

    //引数
    //  *MAC : MagicPacketに含めるMACアドレス

    //MagicPacketの構造

    //(1)純粋なWOL(MagicPacket)構造
    //--Ethernetヘッダ(14bytes)
    //ff ff ff ff ff ff : 宛先MACアドレス（ブロードキャスト）
    //xx xx xx xx xx xx : 送信元MACアドレス（本ボードのMACアドレス）
    //08 42             : Ethernetタイプ(WOL)
    //--データ(102bytes)
    //ff ff ff ff ff ff : 0xff×6
    //yy yy yy yy yy yy : MagicPaket対象（WakeOnLanで電源を入れる機器）のMACアドレス
    //yy yy yy yy yy yy : 以下6バイトのMACアドレスが合計16個
    //（以下省略）

    //※PCで受信する場合は、Raw Socketを扱えるプログラムが必要

    unsigned short i, j;
    unsigned char send_data[14 + 102];
    ether_return_t ret;

    //Ethernetヘッダ(14bytes)
    for (i=0; i<6; i++) send_data[i] = 0xff;//宛先MACアドレス（ブロードキャスト）

    for (i=0; i<6; i++) send_data[6 + i] = g_src_MAC[i];//送信元MACアドレス（本ボードのMACアドレス）

    send_data[12] = 0x08;//Ethernetタイプ 0x0842 : WOL(Wake On Lan, MagicPacket)
    send_data[13] = 0x42;

    //データ(102bytes)
    for (i=0; i<6; i++) send_data[14 + i] = 0xff;//MagicPacketの先頭6バイト

    for (i=0; i<16; i++)
    {
        for (j=0; j<6; j++) send_data[14 + 6 + i * 6 + j] = MAC[j];//MACアドレスを16個繰り返す
    }

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

    sci_write_str("MagicPacket data send (");
    sci_print_MAC(MAC);
    sci_write_str(")");

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

send_data に所定のデータを設定した後で、API 関数の R_ETHER_Write()で送信します。



magic_packet_send()関数で、上記の様な Ethernet フレームが送信されます。
（送信先の MAC アドレスの設定など、プログラムの操作方法に関しては、取扱説明書の方を参照してください。）

・main.c

プロジェクトの本当のメイン関数は、プロジェクト名.c 内で定義されており、本プロジェクトでは
RX65N_MAGIC_PACKET.c
です。

RX65N_MAGIC_PACKET.c

```
extern void main_ether(void);

void main(void)
{
    main_ether();
}
```

メイン関数内で、main_ether()を呼び出すように記載しており、main_ether()は main.c 内に記載されています。

main.c 内に記載されている main_ether()が、実質的なメイン関数です。
マイコン起動後、クロックや初期設定後に、本関数が実行されます。

main_ether()　※抜粋

```
sci_start();

sci_write_str("Copyright (C) 2026 HokutoDenshi. All Rights Reserved.¥n");
sci_write_str("RX65N Ether MagicPacket(WOL) sample program.¥n");
sci_write_str("¥nCOMMAND:¥n");
sci_write_str("  m : target MAC address set¥n");
sci_write_str("  p : print MAC address¥n");
sci_write_str("  S : swap [this board MAC address] <-> [target MAC address]¥n");
sci_write_str("---¥n");
sci_write_str("  s : Operation start with send mode¥n");
sci_write_str("  r : Operation start with receive mode¥n");
sci_write_str("---¥n");
sci_write_str("¥nUSAGE:¥n");
sci_write_str("  SW3 : MagicPacket send¥n");
sci_write_str("  LED : MagicPacket received¥n");
sci_write_str("¥n");

sci_write_str("-default setting address-¥n");
sci_write_str("---¥n");
sci_write_str("target MAC address is     -> ");
sci_print_MAC(g_dst_MAC);
sci_write_str(" : m command for change¥n");
sci_write_str("this board MAC address is -> ");
sci_print_MAC(g_src_MAC);
sci_write_str(" : S command for swap¥n");
sci_write_str("---¥n");

sci_write_str("First m, S, command -> initial setting¥n¥n");
sci_write_str("Please input 's' or 'r' for operation start! ¥n>");
```

起動後に、端末に簡単な操作方法等のメッセージを表示します。

USB-ADAPTER-RX14 に接続した端末上で、以下の様に表示されます。(115200bps)

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

COMMAND:
   m : target MAC address set
   p : print MAC address
   S : swap [this board MAC address] <-> [target MAC address]
---
   s : Operation start with send mode
   r : Operation start with receive mode
---

USAGE:
   SW3 : MagicPacket send
   LED : MagicPacket received

-default setting address-
---
target MAC address is       -> 00-0D-76-00-40-02 : m command for change
this board MAC address is -> 00-0D-76-00-40-01 : S command for swap
---
First m, S, command -> initial setting

Please input 's' or 'r' for operation start!
>
```

端末へのメッセージの表示は、
sci_start()
sci_write_str()
等の関数（sci.c 内で定義）で行えます。文字列を表示する sci_write_str 以外にも、数値を表示する関数などが用意されています。ここでは、ユーザ側で作成したファイルを順に説明していますので、sci.c に関しても後述します。

main_ether() （続き）

```
//動作モード入力待ち
while (1)
{
    //キーボードからのコマンド
    ret2 = sci_read_char(&xc);
    if (ret2 != SCI_RECEIVE_DATA_EMPTY)
    {
        keyboard_command_process(xc);
    }
    //s か r のコマンド入力時に先へ進む
    if (g_operation_mode != OPERATION_WAIT) break;
}
```

メッセージを表示した後で、キーボードからの入力を受け付けます。この部分で MagicPacket を送信する MAC アドレスを変更したり、MagicPacket の受信モードに入るのか送信モードに入るのかをキーボードからの入力で決定します。

main_ether() （続き）（エラーメッセージの表示などは省略）

```
//PHY-LSIのリセット解除
PHY_RESET = 1;

//Etherの初期化
R_ETHER_Initial();

//コールバック関数の登録（リンク状態変更）
cb_func.pcb_func = &ether_callback;
param.ether_callback = cb_func;
ret = R_ETHER_Control(CONTROL_SET_CALLBACK, param);

//コールバック関数の登録（ステータス変更）
cb_func.pcb_int_hnd = &ether_int_handler;
param.ether_callback = cb_func;
ret = R_ETHER_Control(CONTROL_SET_INT_HANDLER, param);

param.channel = g_ether_channel;//本プログラムでは、Eherのchは0(=g_ether_channel)
ret = R_ETHER_Control(CONTROL_POWER_ON, param);

//端子設定
R_ETHER_PinSet_ETHERC0_MII();

//Etherのオープン
ret = R_ETHER_Open_ZC2(g_ether_channel, g_src_MAC, pause_enable);
```

Ether を動かすための初期化の部分です。
PHY-LSI のリセット解除は、RX65N マイコンと RJ45 コネクタの間に接続されている Ethernet-PHY（RX65N の L(=0V), H(=3.3V)のデジタル信号と 100BASE-TX の信号の物理的な変換を行うチップ）を動かし始める処理です。（ここでは、PA6 を H にしているだけです）

R_ETHER_Initial()
は、スマート・コンフィグレータで追加した、r_ether_rx の初期化の API 関数です。

その後、R_ETHER_Control()で 2 種類のコールバック関数を登録しています。コールバック関数の実体は、ether_interupt.c 内に記載しています。

R_ETHER_Control(CONTROL_POWER_ON, param);
は、マイコンの ETHERC, EDMAC の動作を開始する処理です。

R_ETHER_PinSet_ETHERC0_MII();
は、Ether で使用する端子設定です。

R_ETHER_Open_ZC2(g_ether_channel, g_src_MAC, pause_enable);
は、Ether の動作開始です。

main_ether() （続き）

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

    if (g_operation_mode != OPERATION_RECEIVE)//WOL受信専用モードの時はデータ送信不可
    {
        //SW3が押された場合は本ボードからMagicPacketを送出
        if ((SW3 == 0) && sw_valid())
        {
            magic_packet_send(g_dst_MAC);
            sw_disable();//スイッチを一定期間無効化する
        }
    }
}
```

その後、プログラムはメインループ（無限ループ）に入ります。
R_ETHER_LinkProcess()は、API 関数で定期的に実行する必要があります。
MagicPacket 受信モードの時は、受信した場合の処理は割り込み関数内で処理されるので、メインループでは何もしない。送信モードの時は、ボードの SW3 が押された場合は MagicPacket の送信関数(magic_packet_send())が呼ばれます。一度 MagicPacket を送信すると、約 1 秒間はスイッチを押しても反応しない様にしています。
（100ms 刻みの CMT0 の割り込み内で、10 カウント経過後に、SW3 の読み取りが有効化されます。）

〇割り込み関数内での処理（ether_interrupt.c）

ether_callback()

```
void ether_callback(void *arg)
{
    ether_cb_arg_t *p_arg;
    p_arg = (ether_cb_arg_t *)arg;
    ether_return_t ret;

    if (p_arg->event_id == ETHER_CB_EVENT_ID_WAKEON_LAN)
    {
        sci_write_str("Ether");
        sci_write_uint8((unsigned char)g_ether_channel);
        sci_write_str(": MagicPacket received.¥n");
        LED = 1;//ボードのLEDを点灯させる
    }
    else if (p_arg->event_id == ETHER_CB_EVENT_ID_LINK_ON)
    {
        sci_write_str("Ether");
        sci_write_uint8((unsigned char)g_ether_channel);
        sci_write_str(": LINK-UP¥n");

        if (g_operation_mode == OPERATION_RECEIVE)
        {
            //LINK-UPしたらWOL(MagicPacket受信モード）に切り替える
            ret = R_ETHER_WakeOnLAN(g_ether_channel);
            if (ret == ETHER_SUCCESS)
            {
                sci_write_str("-> MagicPacket waiting ...¥n");
            }
            else
            {
                sci_write_str("NORMAL -> WOL(MagicPacket) mode change error.¥n");
            }
        }
    }
    else if (p_arg->event_id == ETHER_CB_EVENT_ID_LINK_OFF)
    {
        sci_write_str("Ether");
        sci_write_uint8((unsigned char)g_ether_channel);
        sci_write_str(": LINK-DOWN¥n");
    }
}
```

本関数は、

・MagicPacket を受信した際

・Ether がリンクアップした際

・Ether がリンクダウンした際（LAN ケーブルを抜いたり、ハブの電源を落とした場合）

に呼び出されます。

MagicPacket を受信した際には、ボードの LED(D5)を点灯させて、端末にメッセージを表示します。

リンクアップの時は、メッセージの表示と MagicPacket 受信動作を開始させます。

リンクダウンの時は、メッセージの表示を行います。

MagicPacket の受信に関しては、マイコンの機能として MagicPacket 受信機能があるので、その機能を有効化するだけです。受信したデータをプログラム内で処理（MAC アドレスの一致を確認したり）する部分はありません。

（なお、RX65N_MAGIC_PACKET2 プロジェクトでは、受信したデータの中身を見て MAC アドレスの一致を確認するコードが含まれます。本プロジェクトでは、MagicPacket の受信はマイコンの機能に任せています。）

ether_int_handler() （一部省略）

```
void ether_int_handler(void *arg)
{
    ether_cb_arg_t *p_arg;

    uint32_t ret, ret2;

    unsigned char *buffer_address;

    p_arg = (ether_cb_arg_t *)arg;

    //受信処理
    if ((p_arg->status_eesr & ETHER_EESR_FR) != 0)
    {
       while (1)
       {
          ret = R_ETHER_Read_ZC2(g_ether_channel, (void **)&buffer_address);

          if (ETHER_NO_DATA < ret)
          {
              ret2 = R_ETHER_Read_ZC2_BufRelease(g_ether_channel);//本プログラムでは受信データがある場合でもバッファの開放しか行わない

              if (ret2 != ETHER_SUCCESS)
              {
                  sci_write_str("Error: BufRelease fail.¥n");
              }
           }
           else
           {
               break;//受信データなし
           }
       }
    }
}
```

本関数は、データの受信などで生じる割り込みの処理となります。上記では、データを受信した際に読み出しのみ行い何もしない（受信データの中身を見る事は無い）処理となっています。（以降のプロジェクトでは、ここで受信したデータを使って処理を行っています。）

ether_interrupt_cmt0()

```
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--;
}
```

本関数は、CMT0 の 100ms 毎に呼び出されます。SW3 を押した際に一定期間（10 カウント）SW3 の読み取りを無効化する事を行っています。

・sci.c

本ファイル内では、

```
void sci_start(void);
void sci_stop(void);
void sci_write_char(unsigned char c);
void sci_write_uint8_hex(unsigned char c);
void sci_write_uint16_hex(unsigned short s);
void sci_write_uint32_hex(unsigned long l);
void sci_write_uint8(unsigned char num);
void sci_write_uint16(unsigned short num);
void sci_write_uint32(unsigned long num);
void sci_write_int8(char num);
void sci_write_int16(short num);
void sci_write_int32(long num);
void sci_write_str(const char *str);
void sci_write_flush(void);
unsigned short sci_read_char(unsigned char *c);
unsigned short sci_read_str(char *str, unsigned short size);
unsigned short sci_read_data_size(void);
void sci_read_buf_clear(void);
```

上記の関数が定義されています。

関数名	内容
sci_start	SCI 動作開始
sci_stop	SCI 動作停止
sci_write_char	1 文字出力
sci_write_uint8_hex	8bit 変数を 16 進数 2 桁で表示
sci_write_uint8_hex	16bit 変数を 16 進数 4 桁で表示
sci_write_uint8_hex	32bit 変数を 16 進数 8 桁で表示
sci_write_uint8	8bit 変数を 10 進数で表示（符号なし）
sci_write_uint16	16bit 変数を 10 進数で表示（符号なし）
sci_write_uint32	32bit 変数を 10 進数で表示（符号なし）
sci_write_int8	8bit 変数を 10 進数で表示（符号あり）
sci_write_int16	16bit 変数を 10 進数で表示（符号あり）
sci_write_int32	16bit 変数を 10 進数で表示（符号あり）
sci_write_str	文字列を表示
sci_write_flush	送信バッファに溜まっているデータを出力
sci_read_char	キーボードから 1 文字入力
sci_read_str	キーボードから複数文字入力
sci_read_data_size	受信バッファに溜まっているバイト数を返す
sci_read_buf_clear	受信バッファを破棄する

SCI(UART)使用時は、最初に

sci_start()

を実行します。

sci_write_str("message¥n");

→message[改行]を表示

unsigned short i = 65432;

sci_write_uint16(i);

→65432 を表示

sci_write_uint16hex(i);

→FF98 を表示

```
unsigned char c;
unsigned short ret;
while (1)
{
   ret = sci_read_char(&c);
   if (ret != SCI_RECEIVE_DATA_EMPTY)
   {
      if (c == 'e') break;
   }
}
```
→キーボードから'e'が入力されるのを待つ処理

上記の様な感じで、端末にメッセージや情報を出力したり、キーボードから入力を読み取る事が出来ます。

sci.c 内に記載されている処理は、スマート・コンフィグレータで生成した API 関数と連携して動作する様になっています。端末の速度（デフォルトでは、115,200bps）等は、スマート・コンフィグレータで設定した値となります。

これらの関数を使用する場合は、スマート・コンフィグレータで SCI の設定を行いコード生成後に、
Config_SCI1_user.c
内に、sci.c 内の割り込み関数を呼び出すように設定する必要があります。新規にプロジェクトを作成して、sci.c 内の関数を使う場合は、2.2.1 に記載されている変更を反映させてください。

以上が、RX65N_MAGIC_PACKET で使用している一連のソースコードの説明となります。

本プロジェクトでは、MagicPacket（単純な 112 バイトのデータ）しか扱っていませんので、複雑な処理がない分 Ether 関連の API 関数の基本的な呼び出し方が判り易い思います。

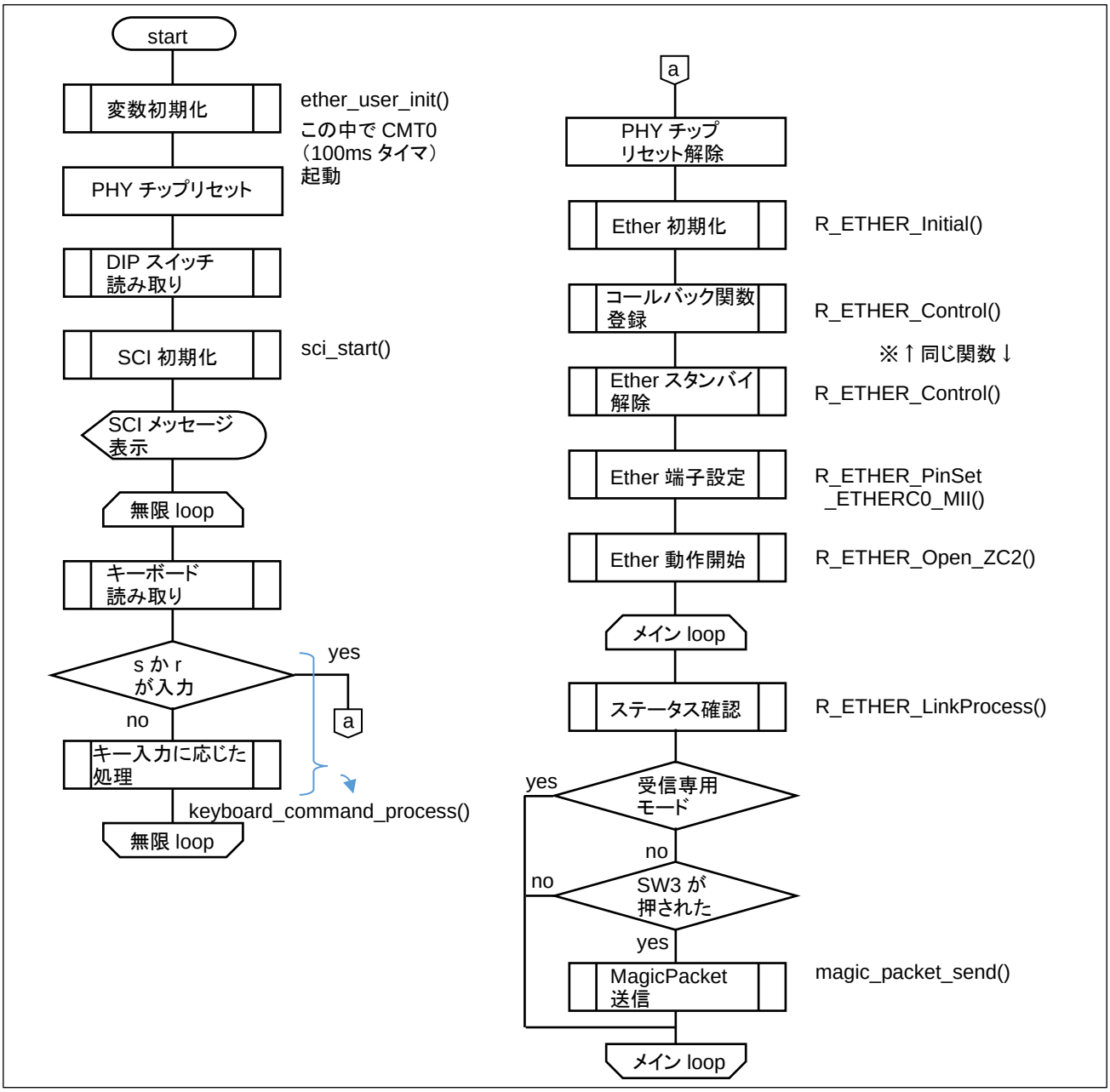
2.4. ユーザ作成関数

magic_packet_send

概要:MagicPacket（WOL パケット）の送信
宣言:void magic_packet_send(unsigned char *MAC)
説明:
・MagicPacket の送信
を行います
引数:
　unsigned char *MAC: MagicPacket 送信対象の MAC アドレス
戻り値:なし
補足:MAC アドレスは
　　unsigned char MAC[6] = {0x00, 0x0D, 0x76, 0x00, 0x40, 0x02};　//MAC address = 00-0D-76-00-40-02
　　の様に、unsigned char の 6 バイトのデータを与えます

2.5. メイン関数での処理

・ether_main() （main.c 内） フローチャート



※本プログラムでは受信データの処理をしている部分はありません
（マイコンが持つ MagicPacket 受信機能で MagicPacket を受信）

3. RX65N_MAGIC_PACKET2 プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_MAGIC_PACKET2
SOURCE¥176PIN¥RX65N_MAGIC_PACKET2

（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。

本プロジェクトでは、
・送信する MagicPacket が 2 種類（純粋な MagicPacket と UDP の MagicPacket）
・受信方式が 2 種類（マイコンの機能で受信する、受信したデータの中身を見て判断）
となります。

・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 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
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
interrupt	ether_interrupt.c	割り込み関数
	ether_interrupt.h	上記ヘッダ
magic_packet	magic_packet.c	MagicPacket(WOL)を処理する関数
	magic_packet.h	上記ヘッダ
main	main.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ

（→RX65N_MAGIC_PACKET プロジェクトと同じ）

使用しているコンポーネントとソースコードの構成は、RX65N_MAGIC_PACKET プロジェクトと同じです。

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 MagicPacket(WOL) sample program2.

COMMAND:
   m : target MAC address set
   i : target broadcast IP address set
   j : this board IP address set
   p : print MAC/IP address
   S : swap [this board MAC address] <-> [target MAC address]
   W : send packet type is WOL.
   I : send packet type is IP.(default)
---
   s : Operation start with send/receive mode
   r : Operation start with receive only mode
---

USAGE:
   SW3 : MagicPacket send
   LED : MagicPacket received

-default setting address-
---
target MAC address is        -> 00-0D-76-00-40-02 : m command for change
this board MAC address is -> 00-0D-76-00-40-01 : S command for swap
target IP address is          -> 192.168.0.255 : i command for change
this board IP address is    -> 192.168.0.80 : j command for change
---
First m, i, j, p, S, W, I command -> initial setting

Please input 's' or 'r' for operation start!
>
```

デフォルトは、

送信先 MAC アドレス　00-0D-76-00-40-02　　（m コマンドで変更）

本ボードの MAC アドレス　00-0D-76-00-40-01　（S コマンドで送信先と本ボードの MAC アドレスの交換）

送信先の IP アドレス　192.168.0.255　　（i コマンドで変更）

本ボードの IP アドレス　192.168.0.80　　（j コマンドで変更）

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 MAC アドレスが逆に設定されます。）

必要に応じて、m, S, i, j コマンドでアドレスの設定を行ってください。

また、W, I コマンドで送信する MagicPacket のタイプを変更可能です。

I コマンド（起動時のデフォルト）（大文字の i）IP アドレスに対して MagicPacket の送信

W コマンド　MAC アドレスに対して MagicPacket の送信

上記が、初期設定のコマンドです。初期設定後、s か r コマンドで動作開始です。

s コマンドで動作開始

```
>command=s

Operation start with MagicPacket send/receive mode.
PUSH SW3 -> send MagicPacket
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-UP
```

→送信・受信モード、ボードの SW3 を押すと MagicPacket 送信

→表示された（この場合は 00-0D-76-00-40-01）に対しての MagicPacket を受信

（ユーザプログラムが受信判定）

r コマンドで動作開始

```
>command=r

Operation start with MagicPacket receive only mode.
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-UP
-> MagicPacket waiting ...
```

→受信専用モード、MagicPacket が外部から送信されるのを待機

（マイコンの Ethernet モジュールが受信判定）

・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

//MagicPacket送信相手のMACアドレス
unsigned char g_dst_MAC[6] = {0x00,0x0D,0x76,0x00,0x40,0x02};//（仮値）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）
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、ここで設定したブロードキャストアドレスに対して
MagicPacketを送信
unsigned char g_dst_IP[4] = {192,168,0,255};

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

本プロジェクトでは、IP アドレスを使用して通信を行うので、IP アドレスの初期値が ether_operation.c 内に記載されています。

IP アドレスと MAC アドレスは、プログラム起動後にキーボードからの入力で変更可能ですが、初期値を変える場合は ether_operation.c 内の値を書き換えてください。

3.1. Ether の初期化

全体的な流れは、2 章で説明していますが、本章では内容に関して多少掘り下げて説明します。

・ether_main.c

```
//PHY-LSIのリセット
PHY_RESET = 0;

（中略）

//PHY-LSIのリセット解除
PHY_RESET = 1;
```

・ether_common.h

```
//PHYリセット端子
#define PHY_RESET PORTA.PODR.BIT.B6
```

PHY-LSI のリセット端子には、本ボードでは PA6 が接続されており、PA6=L で PHY-LSI をリセットします。

PA6 を出力端子に設定しているのは、スマート・コンフィグレータで追加した Config_PORT 内です。（PORTA.PDR.BIT.B6 = 1 をどこかに書けば、スマート・コンフィグレータでの設定は不要です。どこかで、PA6 を出力に設定してください。）

PHY_RESET=0 の後で、100us 以上経過してから、PHY_RESET=1 で PHY-LSI のリセットが解除され、ハードウェアの部分では動作準備が整います。プログラムでは、PHY_RESET=0 と PHY_RESET=1 の間にメッセージを表示させたり、色々な処理が入っているので 100us 以上のタイミングは確保されています。

・ether_main.c

```
//Etherの初期化
R_ETHER_Initial();
```

r_ether_rx モジュールで提供されている API 関数です。

Ethernet モジュールの初期化を行います。

・ether_main.c

```
//コールバック関数の登録（リンク状態変更）
cb_func.pcb_func = &ether_callback;
param.ether_callback = cb_func;
ret = R_ETHER_Control(CONTROL_SET_CALLBACK, param);

//コールバック関数の登録（ステータス変更）
cb_func.pcb_int_hnd = &ether_int_handler;
param.ether_callback = cb_func;
ret = R_ETHER_Control(CONTROL_SET_INT_HANDLER, param);
```

2 種類のコールバック関数を登録しています。

ether_callback()

ETHERC0 のレジスタ(ETHERC0.ECSR)が変化した際に呼ばれるコールバック関数。

→リンクステータス変化

→マイコンの MagicPacket 受信機能を使い MagicPacket を受信した

などの際に呼び出されます。

Ether 機能を動かした後、LAN ケーブルが差さっていると、Ethernet インタフェースがリンクアップして本関数に飛んでくるはずです。

リンクアップ／ダウンすると、PHY-LSI が信号を出しマイコンの ET0_LINKSTA 端子が変化して、リンクアップ／ダウンをマイコン側で検知して割り込みを出します。

ether_int_handler()

EDMAC0 のレジスタ(EDMAC0.EESR)が変化した際に呼ばれるコールバック関数。

→データ受信

→送信完了

→エラー検出

→ディスクリプタ枯渇

などの際に呼び出されます。

データを受信した場合は、本関数内で処理を行います。

コールバック関数は、API 関数の R_ETHER_Control()で行います。登録する関数名は、ユーザが自由に決められます。

・ether_main.c

```
param.channel = g_ether_channel;//本プログラムでは、Eherのchは0(=g_ether_channel)
ret = R_ETHER_Control(CONTROL_POWER_ON, param);
```

ETHERC と EDMAC のモジュールストップ解除を行います。

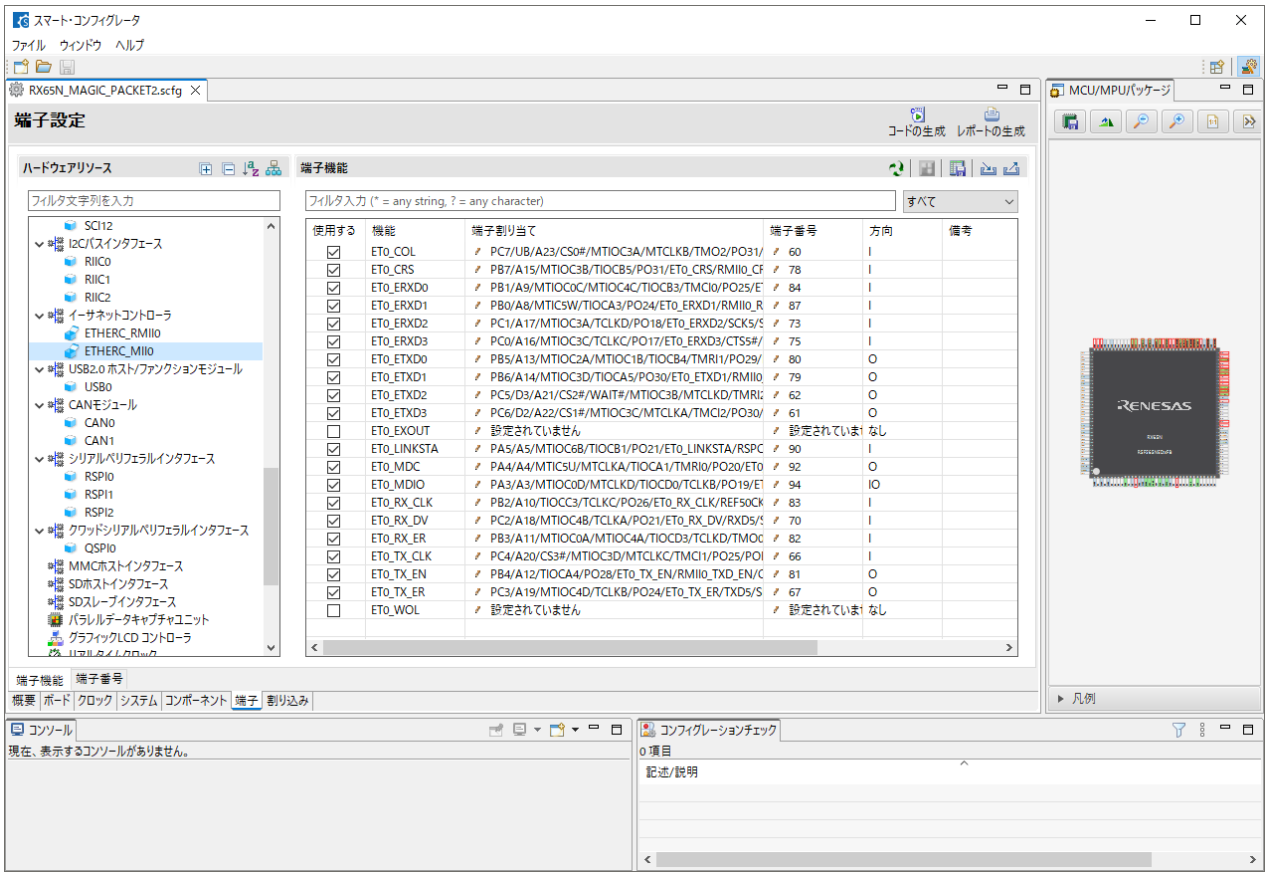
使用しているのは、コールバック関数の登録と同じ関数ですが、与える引数により動作が変わります。

g_ether_channel は、マイコンの ETHERC0/EDMAC0 を使う場合は 0（＝Ether のチャネル）を与えます。RX65N は Ethernet 1ch なので、チャネル 0 しかありません。

・ether_main.c

```
//端子設定
R_ETHER_PinSet_ETHERC0_MII();
```

Ether で使用する端子設定を行う関数です。ポートの PFS 設定です。



Ether で使用する端子は、スマート・コンフィグレータの「端子」タブで選択します。

ET0_COL に割り当てるのは PC7、などマイコンと PHY-LSI の接続により端子割り当てが必要です。ここで、正しい端子を割り当てないと、Ether の通信ができません。

	HSBRX65N176 HSBRX65N144A HSBRX65N100A
ET0_COL	PC7
ET0_CRS	PB7
ET0_ERXD0	PB1
ET0_ERXD1	PB0
ET0_ERXD2	PC1
ET0_ERXD3	PC0
ET0_ETXD0	PB5
ET0_ETXD1	PB6
ET0_ETXD2	PC5
ET0_ETXD3	PC6
ET0_LINKSTA	PA5
ET0_MDC	PA4
ET0_MDIO	PA3
ET0_RX_CLK	PB2
ET0_RX_DV	PC2
ET0_RX_ER	PB3
ET0_TX_CLK	PC4
ET0_TX_EN	PB4
ET0_TX_ER	PC3

スマート・コンフィグレータでは、複数の端子から選べますので上記を選んでください。

・ether_main.c

```
//Etherのオープン
ret = R_ETHER_Open_ZC2(g_ether_channel, g_src_MAC, pause_enable);
```

Ether の動作開始を行う API 関数です。

g_src_MAC は、マイコンボードの MAC アドレスです（ether_operation.c 内で定義。起動後、送受信動作前にキーボードから変更可能。MAC アドレスは、この関数呼び出し前までに決めておかなければならない。）

本関数実行後に、PHY-LSI との通信処理が行われ、LAN ケーブルがつながっていると、通信可能な状態となります。

3.2. Ethernet フレームの送信

・magic_packet.c

```
void magic_packet_send_wol(unsigned char *MAC)
{
    //MagicPacket(112bytes)を送信する関数

    unsigned char send_data[14 + 102];
    ether_return_t ret;

(中略, send_dataの中身の設定)

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

    sci_write_str("MagicPacket data send (");
    sci_print_MAC(MAC);
    sci_write_str(")");

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

magic_packet_send_wol()関数は
RX65N_MAGIC_PACKET プロジェクトの
magic_packet_send()関数の関数名を変えたものです
（中身は同じです）

R_ETHER_Write(0, (void *)send_data, 112);

の様に、

1 番目の引数は Ether のチャンネル（RX65N では常に 0）

2 番目の引数は送信データの先頭アドレス

3 番目の引数は送信バイト数

を指定して呼び出します。

戻り値が、

ETHER_SUCCESS(=0)

の場合は、API 関数は正しく処理が行えたという事となります。

プリアンブルと、FCS（誤り検出）のデータは、ユーザ側で用意する必要はありません。（自動的に付加されます）

取扱説明書では、MagicPacket や ping、UDP, TCP 通信で、どのような Ethernet フレームのデータとなるか詳しく解説していますので、送るべき Ethernet フレームの構造が判っていれば送信操作は難しくはありません。

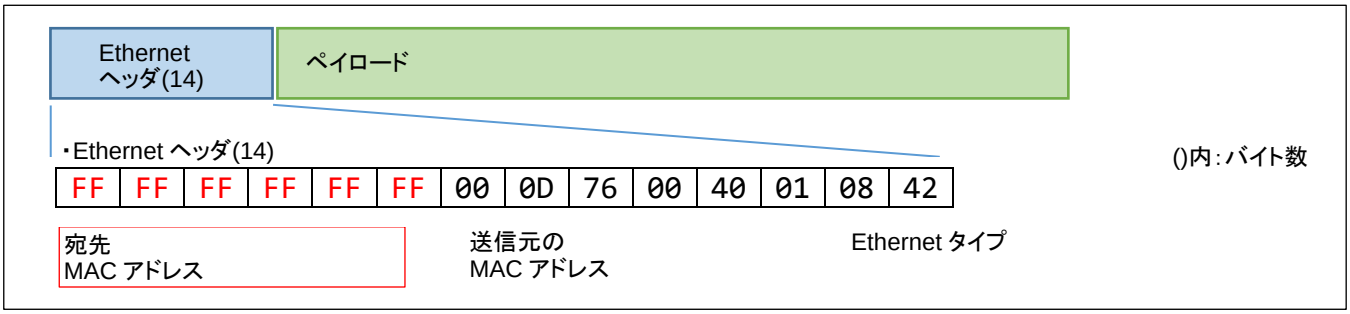
3.3. Ethernet フレームの受信

マイコンがプロミスキャスモードに設定されていない場合は、Ethernet ヘッダ内の宛先 MAC アドレスが

・初期化（R_ETHER_Open_ZC2()呼び出し）時に指定した MAC アドレスと一致

・ブロードキャストアドレス

以外のデータは受信しません。



※プロミスキャスモードとは、全ての Ethernet フレームを受信する一種のデバッグモードです

よって、プログラムで Ethernet ヘッダの中を見て自分宛のデータかどうかをチェックする処理は、基本的には不要です。（そもそもマイコンの Ethernet モジュールが自分宛のデータしか受信しない様になっている）

・ether_interrupt.c （一部省略）

```
void ether_int_handler(void *arg)
{
    ether_cb_arg_t *p_arg;

    uint32_t ret, ret2;

    unsigned char *buffer_address;

    p_arg = (ether_cb_arg_t *)arg;

    //受信処理
    if ((p_arg->status_eesr & ETHER_EESR_FR) != 0)
    {
        while (1)//複数の受信データがある場合は全てのデータを処理する
        {
            ret = R_ETHER_Read_ZC2(g_ether_channel, (void **)&buffer_address);
            if (ETHER_NO_DATA < ret)
            {
                if (g_ether_recv_flag == FALSE)//受信フラグが立っている状態で次のデータを受信した場合はバッファへのコピーは行わない
                {
                    memcpy(g_ether_recv_buf, buffer_address, ret);//受信データをg_ether_recv_bufにコピー
                    g_ether_recv_flag = TRUE;//受信フラグを立てる
                }
                ret2 = R_ETHER_Read_ZC2_BufRelease(g_ether_channel);//受信バッファを開放する

                if (ret2 != ETHER_SUCCESS)
                {
                    sci_write_str("Error: BufRelease fail.¥n");
                }
            }
            else
            {
                break;//受信データなし
            }
        }
    }
}
```

・ether_common.c

```
//受信データと受信フラグ
unsigned char g_ether_recv_buf[ETHER_MAX_PACKET_SIZE];
volatile int g_ether_recv_flag = FALSE;
```

```
#define ETHER_MAX_PACKET_SIZE 1514
//イーサネットのフレーム最大サイズ
(1518)から FCS の 4 バイトを引いた値
```

ether_int_handler()は、様々な割り込み要因で呼ばれる関数ですので、p_arg->status_eesr の ETHER_EESR_FR(=0x00040000)フラグが立っている場合、データを受信した事で割り込みが掛っているので、受信データの処理を行います。

ret = R_ETHER_Read_ZC2(0, (void **)&buffer_address);
で受信バッファの先頭アドレスを拾ってきます。ret には、受信データバイト数が返ります。
※ZC は ZeroCopy の略の様です。データのコピー自体は行わず、データの先頭アドレスを返します。

memcpy(g_ether_recv_buf, buffer_address, ret);
で、ユーザ側で用意したバッファ(g_ether_recv_buf)に、受信データの実体をコピーします。コピーするバイト数は ret です。

－参考－

memcpy はライブラリ関数ですが比較的高速にメモリ間のコピーを行う事ができます。

```
                memcpy(g_ether_recv_buf, buffer_address, ret);     //受信データをg_ether_recv_bufにコピ
MOV.L     [R0],R2
MOV.L     #0000143AH,R1
SMOVF
```

memcpy の部分は、CPU の命令としては、3 命令で処理されています。(この命令の前に、R3 レジスタの設定が済んでいるので、正確には 4 命令必要ですが。)
あるアドレスの複数バイト(ここでは、60 バイト)のデータを別なアドレスにコピーする場合、C 言語では

```
i=60;
while (i !=0)
{
  *a++ = *b++;
  i--;
}
```

の様な処理となります。データをコピーする部分(転送命令)。転送元データのアドレスと転送先データのアドレスをインクリメントする部分。ループ変数の減算と、ループの終了判断(条件分岐)。CPU の命令に分解した場合、複数の命令になりそうな感じですが、これらの処理を SMOVF(順方向ストリング転送命令)というたった 1 命令で処理しています。
1 命令だからといって 1 クロックで終わるわけではありませんが、データのコピーにループを回すよりかは、memcpy に任せた方が高速で処理できると思われます。

ret2 = R_ETHER_Read_ZC2_BufRelease(0);
受信バッファを開放する処理です。データを API 関数が用意した受信バッファから、ユーザ側で用意したバッファにコピーしたので、API 関数側のバッファを開放します。

なお、割り込み関数に飛んできた際、複数のデータを受信している事も考えられるので、while ループを回して R_ETHER_Read_ZC2()の戻り値が ETHER_NO_DATA(=0)になるまで読み出しとバッファの開放を行っています。

※本プロジェクトのプログラムでは、単純化のためユーザ側のバッファを 1 つしか用意していないので、複数のデータを受信した場合は 2 つ目以降のデータは捨てています。今後のプロジェクトではユーザ側バッファを複数用意して、複数のデータにも対応しています。

割り込み関数 ether_int_handler()内で、受信データを g_ether_recv_buf にコピー（及び g_ether_recv_flag を立てる）。メイン関数のメインループ内で、g_ether_recv_buf のデータを処理しています（処理が終わると g_ether_recv_flag を落とす）。

3.4. MagicPacket の送信

本プログラムでは、2 種類の MagicPacket の送信を行います。

起動後に、

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether MagicPacket(WOL) sample program2.

COMMAND:
   m : target MAC address set
   i : target broadcast IP address set
   j : this board IP address set
   p : print MAC/IP address
   S : swap [this board MAC address] <-> [target MAC address]
   W : send packet type is WOL.
   I : send packet type is IP.(default)
---
   s : Operation start with send/receive mode
   r : Operation start with receive only mode
---
```

何もしなければ、IP 形式での MagicPacket を送ります。動作開始前に W コマンドを入力すれば、RX65N_MAGIC_PACKET プロジェクトと同じ 112 バイトのデータを送ります。

MagicPacket 送信関数は、magic_packet.c 内で定義されており、magic_paket_send_wol()関数は、2 章で説明した magic_packet_send()関数と同じものです。W コマンド入力時はこちらの関数が使用されます。

デフォルト、または I コマンドを入力した際は、IP/UDP ベースの MagicPacket 送信関数である magic_packet_ip()関数が実行されます。

・magic_packet.c

```
void magic_packet_send_ip(unsigned char *MAC, unsigned char *IP)
{
    //MagicPacket(144bytes)を送信する関数

    //戻り値
    //  なし

    //引数
    //  *MAC : MagicPacketに含めるMACアドレス

    //MagicPacketの構造

    //(2)PCから送信されるMagicPacket（IPパケットのUDPとして送信される）
    //--Ethernetヘッダ(14bytes)
    //ff ff ff ff ff ff : 宛先MACアドレス（ブロードキャスト）
    //xx xx xx xx xx xx : 送信元MACアドレス（本ボードのMACアドレス）
    //08 00             : Ethernetタイプ(IP)
    //--IPヘッダ(20bytes)
    //45 00             :IPv4
    //00 82             :パケットサイズ130bytes
    //12 34             :Id
    //00 00             :Flag
    //80                :TTL
    //11                :UDP(17)
    //XX XX             :チェックサム
    //c0 a8 00 50       :送信元IPアドレス(192.168.0.80)（本ボードのIPアドレス）
    //c0 a8 00 ff       :宛先IPアドレス(192.168.0.255)（ブロードキャストアドレス，サブネットマスクが
255.255.255.0 の時は xxx.xxx.xxx.255）
    //--UDPヘッダ(8bytes)
    //00 09             :送信元ポート(9)
    //00 09             :宛先ポート(9)
    //00 6e             :パケット長110bytes
    //XX XX             :チェックサム
    //--UDPデータ(102bytes)
    //ff ff ff ff ff ff :MagicPakcetの0xff×6
    //yy yy yy yy yy yy : MagicPaket対象（WakeOnLanで電源を入れる機器）のMACアドレス
    //yy yy yy yy yy yy : 以下6バイトのMACアドレスが合計16個
    //（以下省略）

unsigned char send_data[14 + 20 + 8 + 102];//Ethernetヘッダ(14)，IPヘッダ(14)，UDPヘッダ(8)，UDPデー
タ(102)
ether_return_t ret;

//send_dataの中身の設定（記載を省略）

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

magic_packet_ip()関数では、上記コメントに記載されている様な Ethernet フレームを送信します。（こちらのデータは、PC のアプリで受信可能です。）

※IP や UDP を使った通信に関しては、後の章で説明します。ここでは、単純に RX65N_MAGIC_PACKET プロジェクトに対して Ethernet フレームのデータを変えているという認識で良いです。

送信で使用している API 関数は、R_ETHER_Write()で、送信するデータの中身が変わっているだけで処理としては変わりません。

3.5. MagicPacket の受信

```
Copyright (C) 2026 HokutoDenshi. All Rights Reserved.
RX65N Ether MagicPacket(WOL) sample program2.

COMMAND:
  m : target MAC address set
  i : target broadcast IP address set
  j : this board IP address set
  p : print MAC/IP address
  S : swap [this board MAC address] <-> [target MAC address]
  W : send packet type is WOL.
  I : send packet type is IP.(default)
---
  s : Operation start with send/receive mode
  r : Operation start with receive only mode
---
```

3.5.1. マイコンの MagicPacket 受信機能を使用

r コマンドで動作を開始した場合です。この場合は、RX65_MAGIC_PACKET の受信動作と変わりません。

・ether_interrupt.c

```
void ether_callback(void *arg)
{
    ether_cb_arg_t *p_arg;
    p_arg = (ether_cb_arg_t *)arg;
    ether_return_t ret;

    if (p_arg->event_id == ETHER_CB_EVENT_ID_WAKEON_LAN)
    {
        sci_write_str("Ether");
        sci_write_uint8((unsigned char)g_ether_channel);
        sci_write_str(": MagicPacket received.¥n");
        LED = 1;//ボードのLEDを点灯させる
    }
    else if (p_arg->event_id == ETHER_CB_EVENT_ID_LINK_ON)
    {
        sci_write_str("Ether");
        sci_write_uint8((unsigned char)g_ether_channel);
        sci_write_str(": LINK-UP¥n");

        if (g_operation_mode == OPERATION_RECEIVE)
        {
            //LINK-UPしたらWOL(MagicPacket受信モード）に切り替える
            ret = R_ETHER_WakeOnLAN(g_ether_channel);
            if (ret == ETHER_SUCCESS)
            {
                sci_write_str("-> MagicPacket waiting ...¥n");
            }
            else
            {
                sci_write_str("NORMAL -> WOL(MagicPacket) mode change error.¥n");
            }
        }
    }
    else if (p_arg->event_id == ETHER_CB_EVENT_ID_LINK_OFF)
    {
        sci_write_str("Ether");
        sci_write_uint8((unsigned char)g_ether_channel);
        sci_write_str(": LINK-DOWN¥n");
    }
}
```

ether_callback()関数では、p_arg->event_id をモニタする事で割り込み要因が判ります。
p_arg->event_id == ETHER_CB_EVENT_ID_LINK_ON の時は、リンクアップです。HSBRX65Nxxx のボードの RJ45 コネクタの横の LINK LED が点灯した場合、LAN ケーブルがハブなどに接続されており物理的な接続が行えたという事です。

キーボードから r コマンドを入力。

```
>command=r

Operation start with MagicPacket receive only mode.
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-UP
-> MagicPacket waiting ...
```

端末にも、Ether0: LINK-UP の表示が出ます。

p_arg->event_id == ETHER_CB_EVENT_ID_LINK_OFF の時は、逆に LAN ケーブルが抜けた等で、リンクダウンを検出した場合です。

```
Ether0: receive abort detected
Ether0: LINK-DOWN
```

r コマンドで動作を開始すると、リンクアップのタイミングで
R_ETHER_WakeOnLAN(0);
を実行します。

この API 関数は、マイコンを WakeOnLAN（=MagicPacket 受信待ち）とするモードです。受信した MagicPacket（WOL パケット）内に、自局の MAC アドレスが含まれていた場合は、
p_arg->event_id == ETHER_CB_EVENT_ID_WAKEON_LAN
で、割り込みが飛んできます。

その際は、

```
Ether0: MagicPacket received.
Ether0: LINK-UP
-> MagicPacket waiting ...
```

Ether0: MagicPacket received の表示が出て、ボードの LED(D5)が点灯となります。
また、一旦接続をリセットして再度リンクアップ、R_ETHER_WakeOnLAN()、MagicPacket 受信待ちとなります。

MagicPacket の受信はマイコンの機能として行われ、ユーザプログラムで受信データが自局宛の MagicPacket か否かを判断している訳はないという点がポイントです。
（自局 MAC アドレス：ether_common.c 内で定義されている、00-0D-76-00-40-01。起動後キーボードからの m コマンドと S コマンドで変更可能。DIP-SW 3 を ON にして起動した場合は、00-0D-76-00-40-02 に設定されます。）

3.5.2. 受信データをユーザプログラムで処理

s コマンドで動作を開始した場合です。この場合は受信データの中身を見て、ユーザプログラムで MagicPacket かどうかの判定を行います。

キーボードから s コマンドを入力。

```
>command=s

Operation start with MagicPacket send/receive mode.
PUSH SW3 -> send MagicPacket
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-UP
```

・ether_interrupt.c（3.3 節に示したコードの抜粋）

```
            ret = R_ETHER_Read_ZC2(g_ether_channel, (void **)&buffer_address);
            if (ETHER_NO_DATA < ret)
            {
                if (g_ether_recv_flag == FALSE)//受信フラグが立っている状態で次のデータを受信した場合はバッファへのコピーは行わない
                {
                    memcpy(g_ether_recv_buf, buffer_address, ret);//受信データをg_ether_recv_bufにコピー
                    g_ether_recv_flag = TRUE;//受信フラグを立てる
```

受信割り込みのタイミングで、
g_ether_recv_buf に受信データをコピー
g_ether_recv_flag（フラグ変数）を 1(=TRUE)にセット
を行います。

メイン関数内の無限ループにて、g_ether_recv_flag が立っている場合は、受信データの処理を行っています。

・main.c（抜粋）

```
//メインループ
while (1)
{
    R_ETHER_LinkProcess(g_ether_channel);          定期的に呼ぶ必要がある API 関数

    if (g_operation_mode != OPERATION_RECEIVE)
    {
        //SW3が押された場合は本ボードからMagicPacketを送出
        if ((SW3 == 0) && sw_valid())
        {
            switch (g_send_packet_type)
            {
                case PACKET_TYPE_ETHERNET_IPv4:
                magic_packet_send_ip(g_dst_MAC, g_dst_IP);//IP/UDPパケットとして送信
                break;

                case PACKET_TYPE_ETHERNET_WOL:
                magic_packet_send_wol(g_dst_MAC);//WOLパケットとして送信
                break;
        }

        sw_disable();//スイッチを一定期間無効化する
        }                                          送信処理
    }

    //受信データの処理                            受信処理
    if (g_ether_recv_flag)
    {
        reveive_data_process();
    }
}
```

receive_data_process()が受信データを処理する関数です。

・main.c

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

    //戻り値
    //  なし

    //引数
    //  なし

    //補足
    //  本ボードのMACアドレスが含まれるMagicPacketを受信した際LEDを点灯させる

    //MagicPacketの構造

    //(1)純粋なWOL(MagicPacket)構造
    //--Ethernetヘッダ(14bytes)
    //ff ff ff ff ff ff : 宛先MACアドレス（ブロードキャスト）
    //xx xx xx xx xx xx : 送信元MACアドレス
    //08 42             : Ethernetタイプ(WOL)
    //--データ(102bytes)
    //ff ff ff ff ff ff : 0xff×6  ★チェック対象
    //yy yy yy yy yy yy : MagicPaket対象（WakeOnLanで電源を入れる機器）のMACアドレス  ★チェック対象
    //yy yy yy yy yy yy : 以下6バイトのMACアドレスが合計16個
    //（以下省略）

    //(2)PCから送信されるMagicPacket（IPパケットのUDPとして送信される）
    //--Ethernetヘッダ(14bytes)
    //ff ff ff ff ff ff : 宛先MACアドレス（ブロードキャスト）
    //xx xx xx xx xx xx : 送信元MACアドレス
    //08 00             : Ethernetタイプ(IP)
    //--IPヘッダ(20bytes)
    //45 00             :IPv4
    //00 82             :パケットサイズ130bytes
    //?? ??             :Id
    //00 00             :Flag
    //80                :TTL
    //11                :UDP(17)
    //XX XX             :チェックサム
    //c0 a8 00 01       :送信元IPアドレス(192.168.0.1)
    //c0 a8 00 ff       :宛先IPアドレス(192.168.0.255)
    //--UDPヘッダ(8bytes)
    //00 09             :送信元ポート(9)
    //00 09             :宛先ポート(9)
    //00 6e             :パケット長110bytes
    //XX XX             :チェックサム
    //--UDPデータ(102bytes)
    //ff ff ff ff ff ff :MagicPakcetの0xff×6  ★チェック対象
    //yy yy yy yy yy yy : MagicPaket対象（WakeOnLanで電源を入れる機器）のMACアドレス  ★チェック対象
    //yy yy yy yy yy yy : 以下6バイトのMACアドレスが合計16個
    //（以下省略）
```

・main.c（続き, receive_data_process()）

```
    unsigned char ff_mac[6] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

    //★の部分のデータが 0xff×6 かつ 本ボードのMACアドレスの場合MagicPacketを受信したものとして処理
    //パケットの他の部分はチェック対象外
    if ((MAC_compare(&g_ether_recv_buf[14], ff_mac) == 0) &&
       (MAC_compare(&g_ether_recv_buf[14+6], g_src_MAC) == 0))
    {
        //(1)のタイプのMagicPacketを受信
        sci_write_str("MagicPacket(1) received.¥n");
        LED = 1;//ボードのLEDを点灯させる
    }
    else if  ((MAC_compare(&g_ether_recv_buf[14+20+8], ff_mac) == 0) &&
              (MAC_compare(&g_ether_recv_buf[14+20+8+6], g_src_MAC) == 0))
    {
        //(2)のタイプのMagicPacketを受信
        sci_write_str("MagicPacket(2) received.¥n");
        LED = 1;//ボードのLEDを点灯させる
    }

    //受信データの処理が終わったのでフラグを落とす
    g_ether_recv_flag = FALSE;
}
```

本プログラムでは、

(1)純粋な MagicPacket（116 バイト）

(2)PC から UDP 形式で送った MagicPacket（144 バイト）

の、どちらの MagicPacket も受信します。

・MagicPacket 内に、0xFF が 6 つ

・0xFF×6 の次の 6 バイトが自分の MAC アドレスと同じ

場合に、自分宛の MagicPacket だと認識します。（MagicPacket 内には、MAC アドレスが 16 回繰り返されますが、最初の 1 回が自分の MAC アドレスと一致していれば自分宛の MagicPacket と認識します。）

マイコンボードから送信された MagicPacket を受信

```
MagicPacket(1) received.
```

PC から送信された MagicPacket を受信

```
MagicPacket(2) received.
```

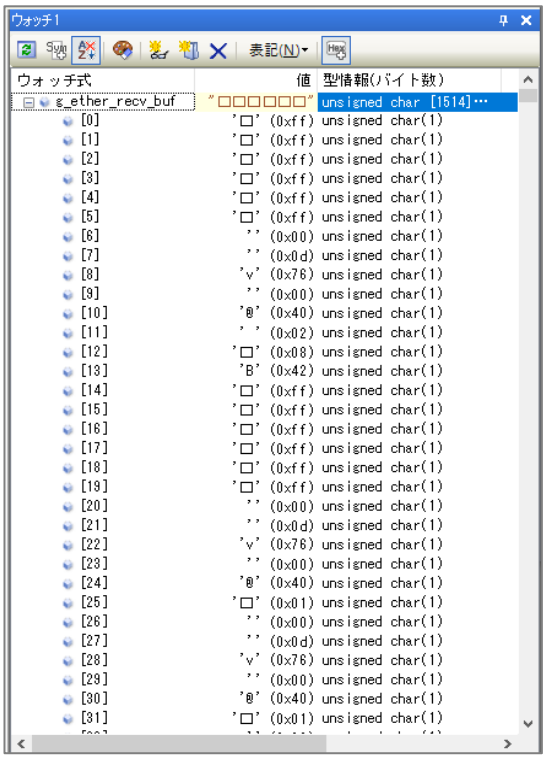
※[参考]本プログラムでは、受信バイト数の情報を、reveive_data_process()に渡していません。

g_ether_recv_buf は、g_ether_recv_buf[1514]の定義ですので、&g_ether_recv_buf[14+20+8+6]（+6 まで）を読み出しても、配列外アクセスにはなりません。

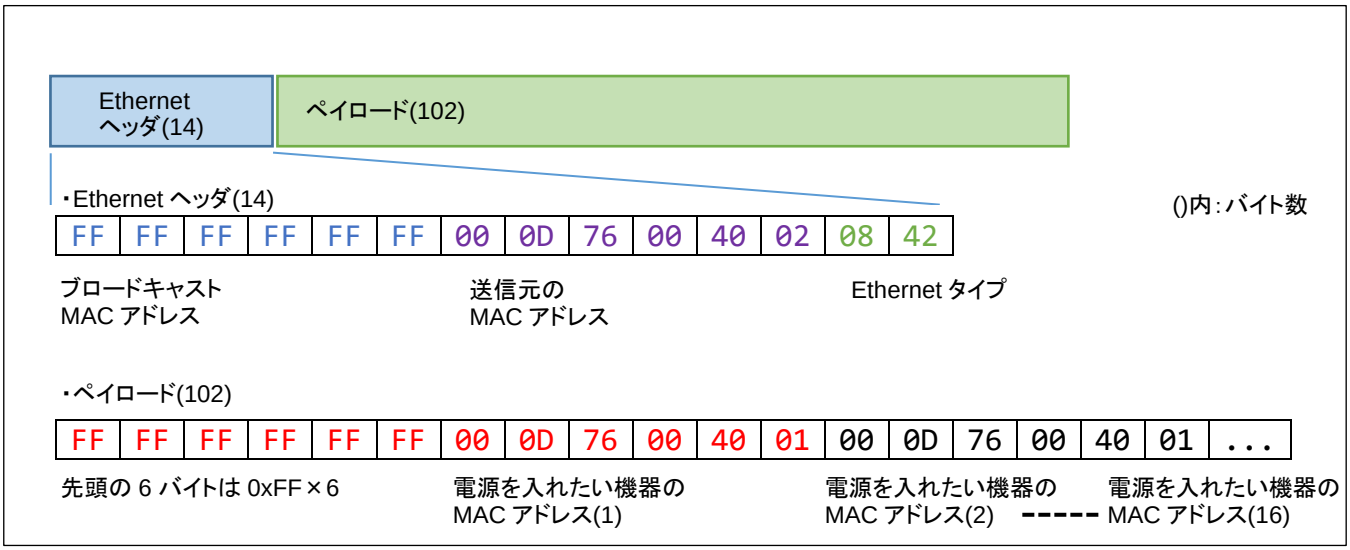
また、Ethernet の最小フレーム長のデータ（60 バイト）を受信した場合でも、アクセスしているデータは古いデータではなく、その際に受信したデータです。

本来は、受信データの処理では、受信したデータ長の範囲でデータを処理すべきですが、本プログラムではデータ長の管理をさぼっています。

(1)のタイプの MagicPacket を受信した際に、受信データをコピーした変数(g_ether_recv_buf)の中身を見てみると、



の様になっており、



上記の様なデータが、g_ether_recv_buf には格納されている事が確認できます。
(プログラムでは、赤色の部分が判定基準。「データの先頭から 14~19 バイトが 0xFF」かつ「20~26 バイト目が自分自身の MAC アドレス」である事。…本プログラムでは、それ以外のデータは判定の基準ではありません。)
(なお、Ether ヘッダ内の宛先(先頭の 6 バイト)が、自分宛(00-0D-76-00-40-01)、またはブロードキャストアドレスでない場合は、Ethernet フレーム自体をマイコン側の機能として受信しないので、receive_data_process()が呼ばれている時点で、自分宛(ブロードキャストを含む)の Ethernet フレームです。)

3.6. 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、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
・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

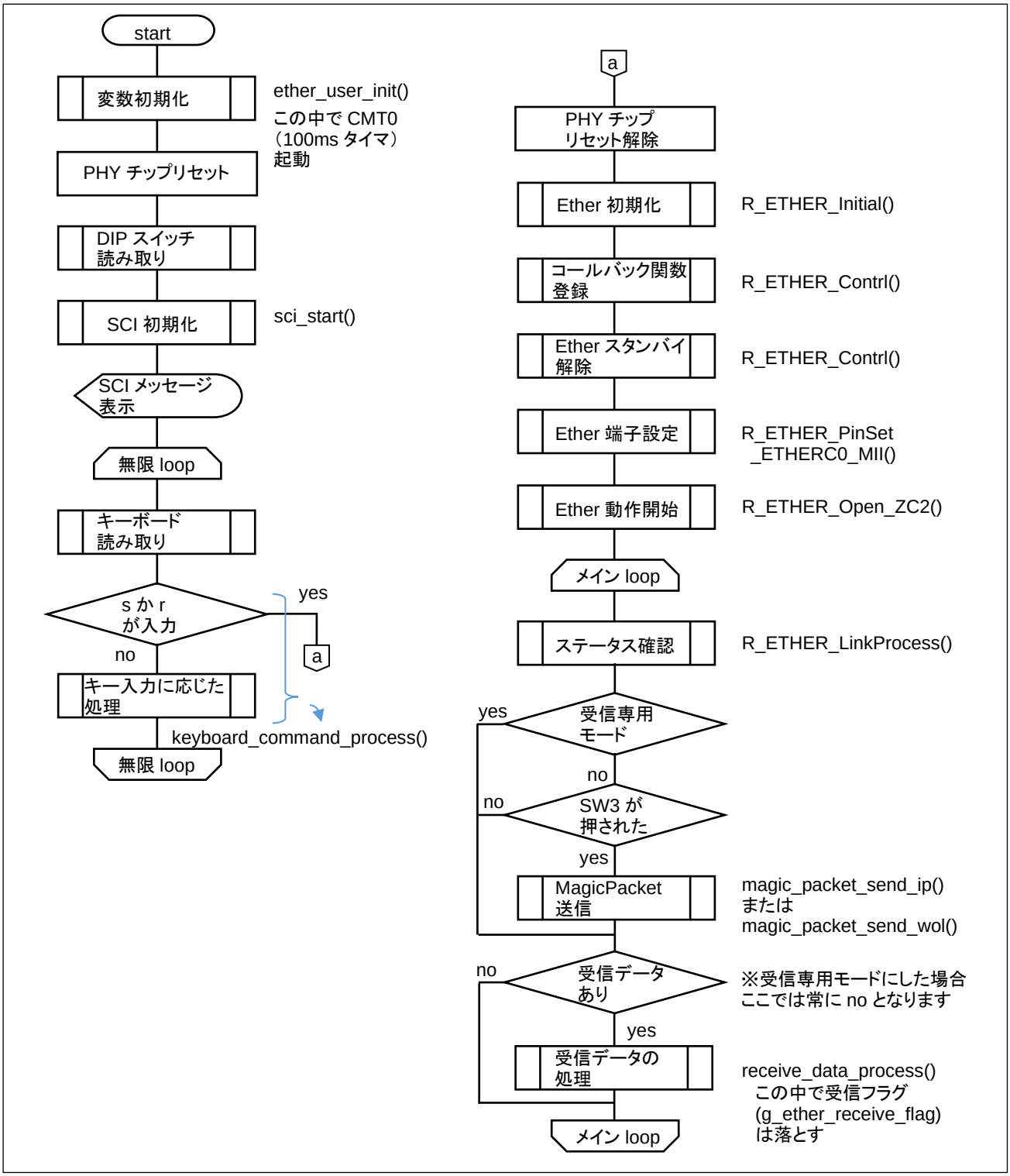
の処理を行っています。

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 MagicPacket 送信（送信/受信モードで起動された場合）
・受信データの処理

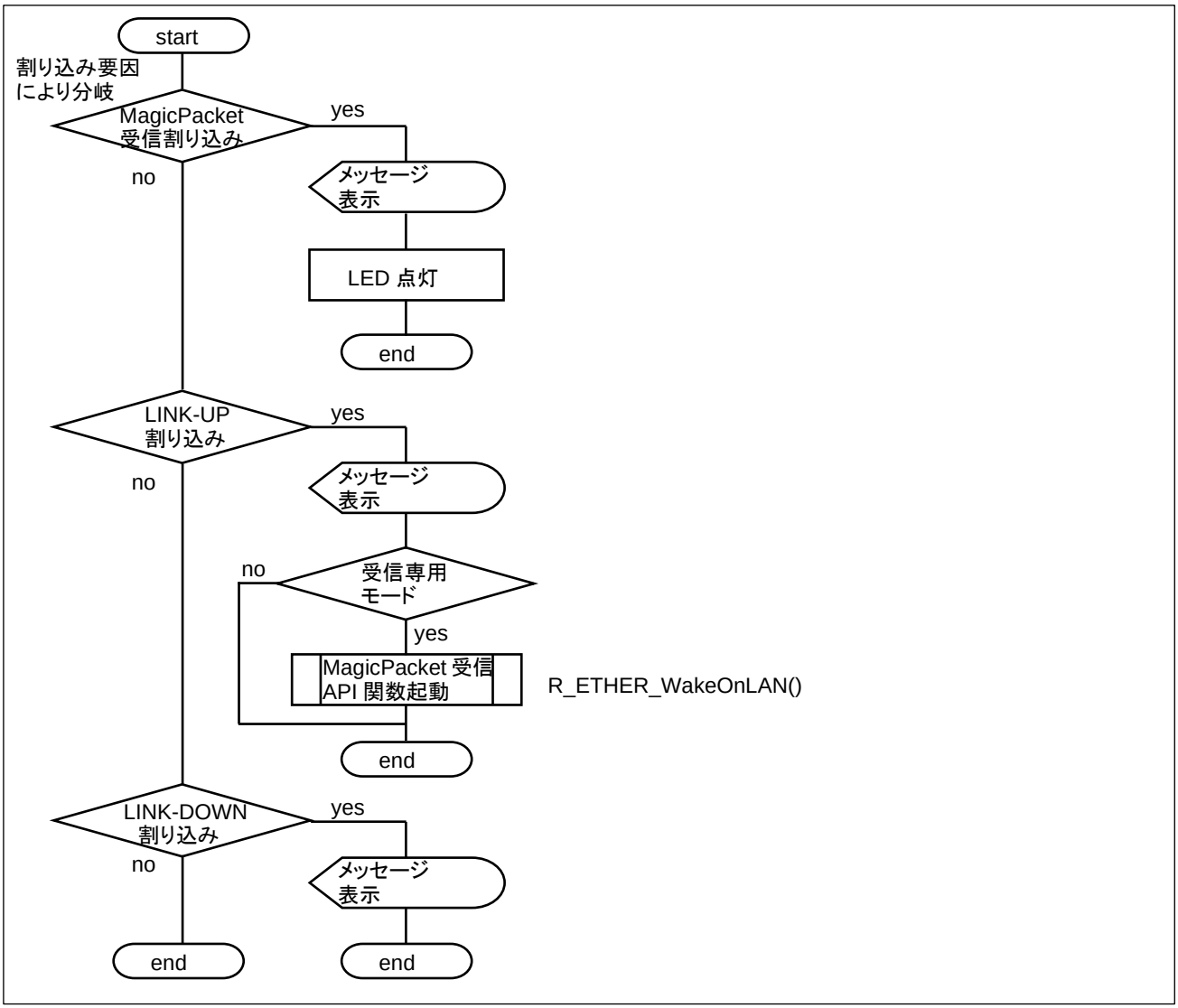
となっています。

・ether_main() （main.c 内）



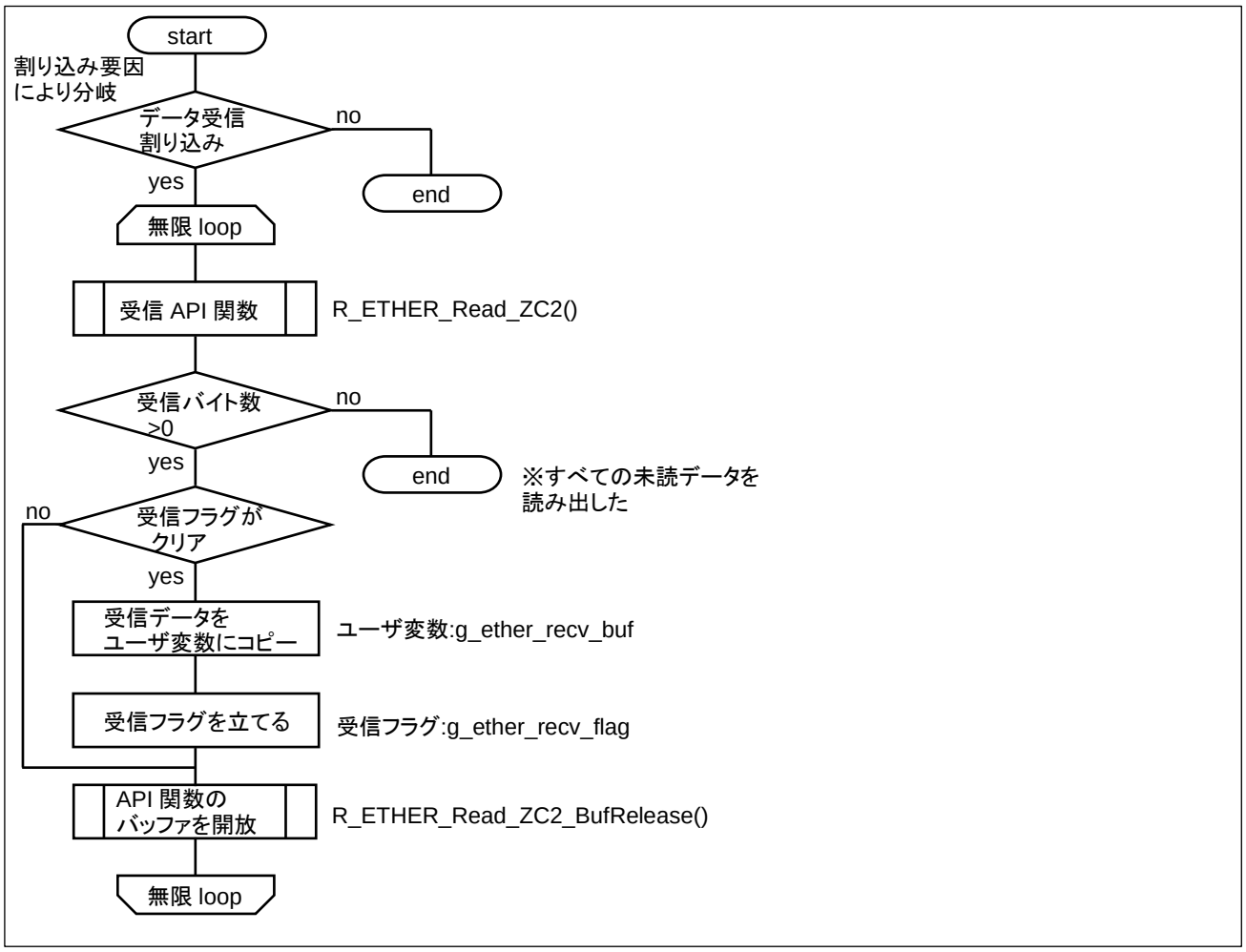
3.7. 割り込み関数での処理

・ether_callback() （ether_interrupt.c 内）



ETHERC0 のレジスタ(ETHERC0.ECSR)が変化した際に呼ばれるコールバック関数。

・ether_int_handler() （ether_interrupt.c 内）



EDMAC0 のレジスタ(EDMAC0.EESR)が変化した際に呼ばれるコールバック関数。

3.8. ユーザ作成関数

magic_packet_send_wol

概要:MagicPacket(WOL パケット)の送信
宣言:void magic_packet_send_wol(unsigned char *MAC)
説明:
・MagicPacket の送信
を行います
引数:
　unsigned char *MAC: MagicPacket 送信対象の MAC アドレス
戻り値:なし
補足:　RX65N_MAGIC_PACKET プロジェクトの magic_packet_send()関数と同じ

magic_packet_send_ip

概要:IP/UDP を使用した MagicPacket(WOL パケット)の送信
宣言:void magic_packet_send_wol(unsigned char *MAC, unsigned char *IP)
説明:
・MagicPacket の送信
を行います
引数:
　unsigned char *MAC: MagicPacket 送信対象の MAC アドレス
　unsigned char *IP: MagicPacket 送信先のネットワークアドレス(ブロードキャストアドレス)
戻り値:なし
補足:　IP/UDP の形式での MagicPacket を送信します

4. RX65N_ETHER_NOAPI プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_ETHER_NOAPI
SOURCE¥176PIN¥RX65N_ETHER_NOAPI

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

本プロジェクトは、RX65N_MAGIC_PACKET2 プロジェクトと同じ動作を行うものです。
但し、API 関数（r_ether_rx 内で定義されている関数）を使用していません。

r_ether_rx コンポーネントを使用せずに、Ether の部分を全て自前で作るという実験的なプロジェクトです。

r_ether_rx コンポーネントを追加して使用した場合、R_ETHER_で始まる API 関数の中身はブラックボックス化する、良く判らないけど、動いているからヨシとなると思います。この場合、トラブルが無く動作していれば困りませんが、いざトラブルが起こった場合、どのように原因を突き止めれば良いか、お手上げ状態となります。

API 関数が裏でやっている事を理解した上で、既存の API 関数を使用する。そのために、一旦 API 関数を使わないと、どのような処理が必要になるのかという事を学ぶためのプロジェクトです。

プログラムの動作としては、RX65N_MAGIC_PACKET2 と変わりませんので、ネットワークの動作としては新しい要素は出てきません。

既存の API 関数を使わずにプログラムを作る、という修行みたいなプロジェクトになっていますので、ネットワークの勉強をする目的であれば、本プロジェクトはスキップして次の章へ進んでも問題ありません。

r_ether_rx コンポーネントを使わずにプログラムを作成すると、どのような処理が必要になるかを本章で示します。

・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~~ ※本プロジェクトでは追加しない
Config_CMT0	コンペアマッチタイマ（100ms 毎）

・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
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
ether_noapi	**ether_noapi.c**	**Ether の処理**
	ether_noapi.h	**上記ヘッダ**
interrupt	ether_interrupt.c	割り込み関数
	ether_interrupt.h	上記ヘッダ
magic_packet	magic_packet.c	MagicPacket(WOL)を処理する関数
	magic_packet.h	上記ヘッダ
main	main.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ

ether_noapi.c 内に、本プロジェクトで追加しているコードが入っています。

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 MagicPacket(WOL) sample program no API version.

COMMAND:
   m : target MAC address set
   i : target broadcast IP address set
   j : this board IP address set
   p : print MAC/IP address
   S : swap [this board MAC address] <-> [target MAC address]
   W : send packet type is WOL.
   I : send packet type is IP.(default)
---
   s : Operation start with send/receive mode
   r : Operation start with receive only mode
---

USAGE:
   SW3 : MagicPacket send
   LED : MagicPacket received

-default setting address-
---
target MAC address is         -> 00-0D-76-00-40-02 : m command for change
this board MAC address is -> 00-0D-76-00-40-01 : S command for swap
target IP address is          -> 192.168.0.255 : i command for change
this board IP address is    -> 192.168.0.80 : j command for change
---
First m, i, j, p, S, W, I command -> initial setting

Please input 's' or 'r' for operation start!
>
```

デフォルトは、

送信先 MAC アドレス　00-0D-76-00-40-02　　（m コマンドで変更）

本ボードの MAC アドレス　00-0D-76-00-40-01　（S コマンドで送信先と本ボードの MAC アドレスの交換）

送信先の IP アドレス　192.168.0.255　　（i コマンドで変更）

本ボードの IP アドレス　192.168.0.80　　（j コマンドで変更）
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 MAC アドレスが逆に設定されます。）

必要に応じて、m, S, i, j コマンドでアドレスの設定を行ってください。
また、W, l コマンドで送信する MagicPacket のタイプを変更可能です。

l コマンド（起動時のデフォルト）IP アドレスに対して MagicPacket の送信
W コマンド　MAC アドレスに対して MagicPacket の送信

上記が、初期設定のコマンドです。初期設定後、s か r コマンドで動作開始です。

s コマンドで動作開始

```
>command=s

Operation start with MagicPacket send/receive mode.
PUSH SW3 -> send MagicPacket
target MAC address         = 00-0D-76-00-40-02
target IP address          = 192.168.0.255
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-DOWN
Ether0: LINK-UP
```

→送信・受信モード、ボードの SW3 を押すと MagicPacket 送信
→表示された（この場合は 00-0D-76-00-40-01）に対しての MagicPacket を受信
（ユーザプログラムが受信判定）

r コマンドで動作開始

```
>command=r

Operation start with MagicPacket receive only mode.
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-DOWN
Ether0: LINK-UP
-> MagicPacket waiting ...
```

→受信専用モード、MagicPacket が外部から送信されるのを待機
（マイコンの Ether モジュールが受信判定）

※コマンドや動作は、RX65N_MAGIC_PACKET2 と同等です。

4.1. メイン関数での処理

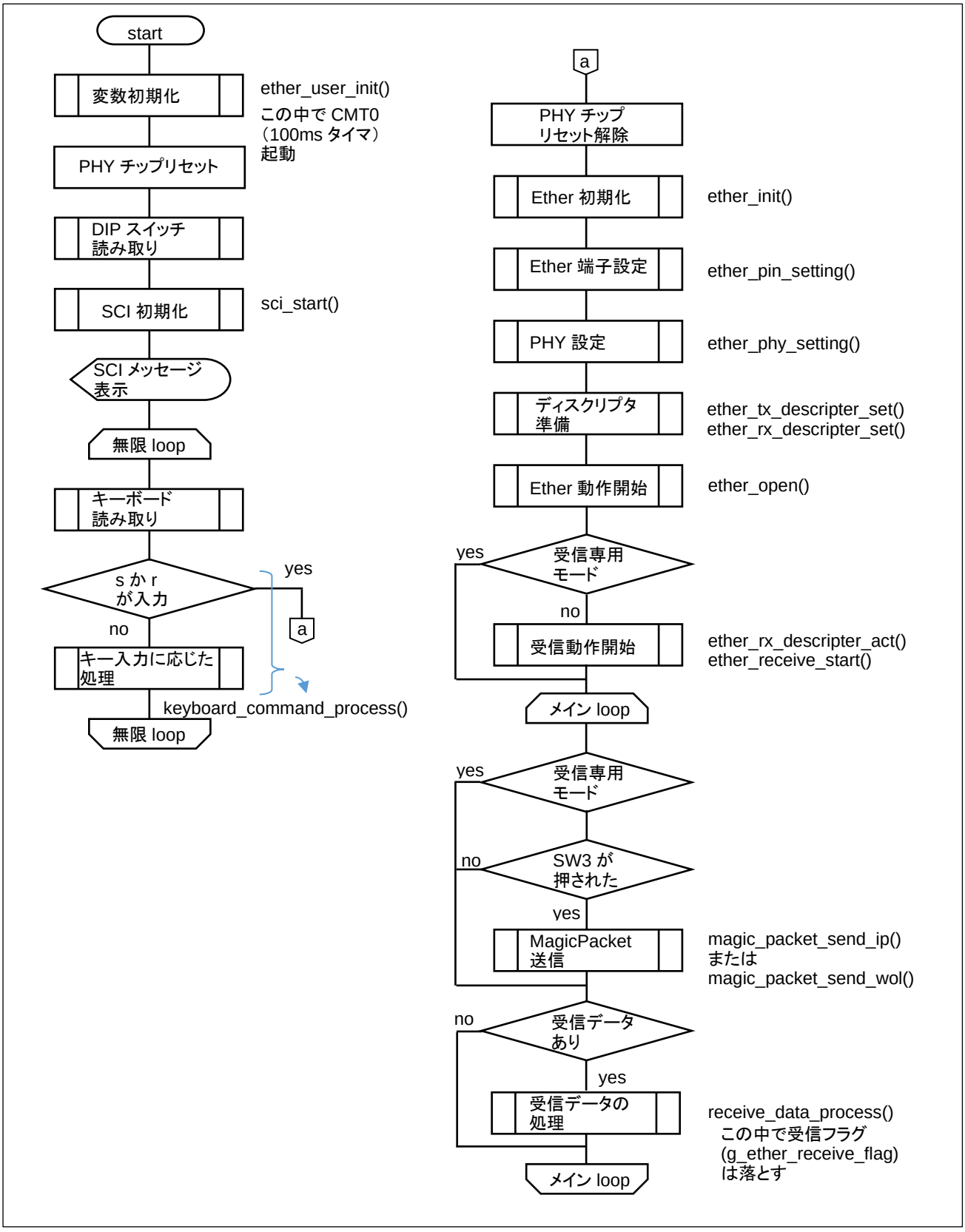
main.c 内の ether_main()では、

・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、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
・PHY チップのリセット解除
・Ether の初期化
・端子設定
・PHY 設定
・ディスクリプタ準備
・Ether の動作開始
・受信動作開始
・メインループ

の処理を行っています。

メインループでは、
・SW3 が押されている場合は MagicPacket 送信（送信/受信モードで起動された場合）
・受信データの処理
となっています。

・ether_main() （main.c 内）



4.2. Ether の初期化

・ether_init()
　ETHERC0 のモジュールストップ解除
　EDMAC のソフトウェアリセット
　ETHERC0 のレジスタ設定
　EDMAC0 のレジスタ設定
　ETHERC0/EDMAC0 の割り込み設定（グループ AL1 の割り込み設定）

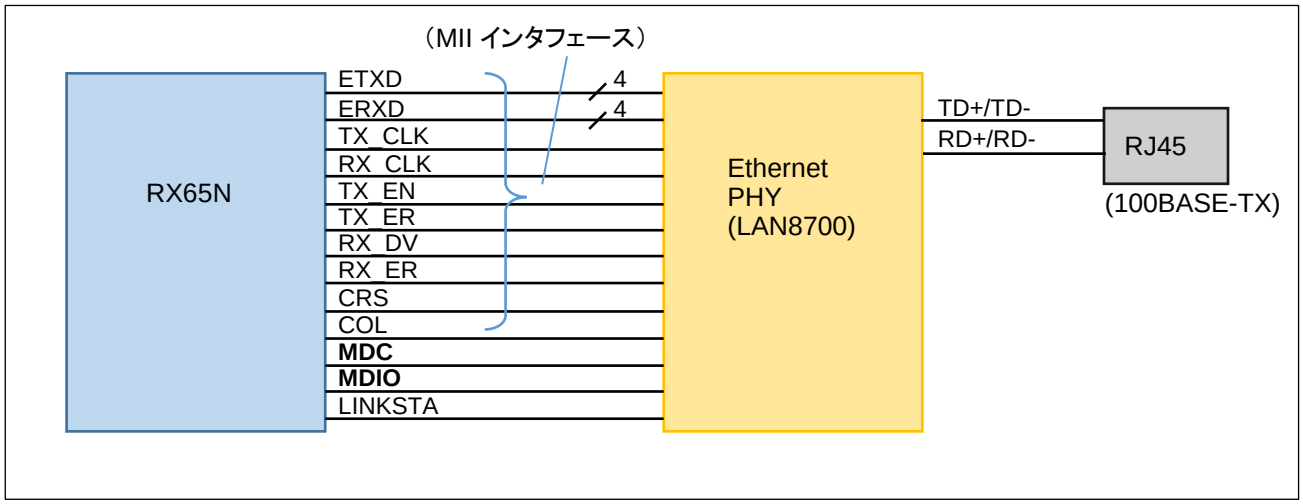
・ether_pin_setting()
　ETHERC0 で使用する各種端子の PFS 設定, PMR 設定

・ether_phy_setting()
　PHY チップのレジスタ設定（レジスタにリセット値を設定）
→MDC（クロック出力）, MDIO（データ入出力）の 2 端子で PHY チップ(LAN8700)と通信

・ether_tx_desctipter_set()
・ether_rx_desctipter_set()
　送信と受信用のディスクリプタ設定

・ether_open()
　割り込み（グループ AL1）の有効化
　ETHERC0 の送受信有効化

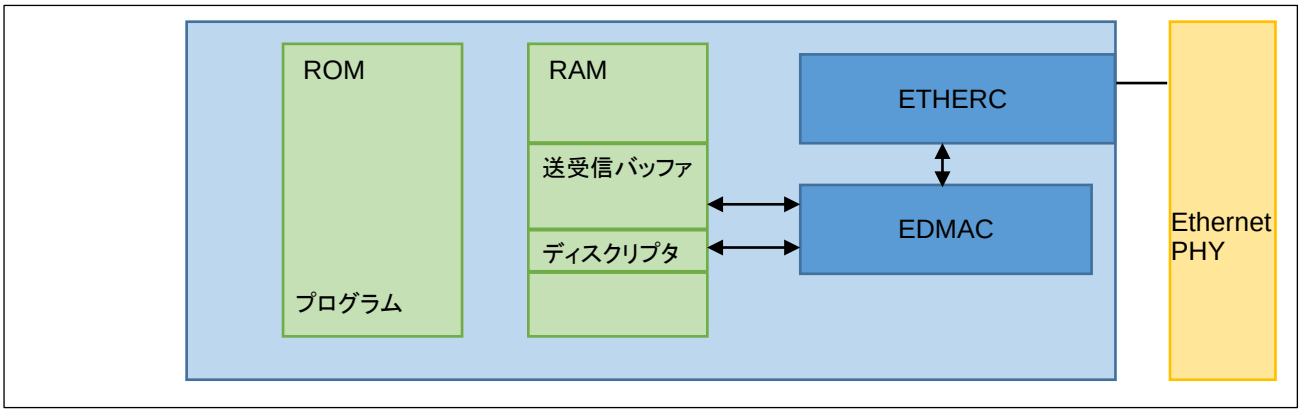
　Ether の初期化はいくつかの関数で構成されています。この中で、Ether に固有な設定として PHY チップの設定とディスクリプタの設定があります。



　PHY チップは、RX65N の I/O ポート（L=0V, H=3.3V のデジタル信号）を、Ethernet(100BASE-TX)の TD+/TD-, RD+/RD-の信号に変換を行うチップです。

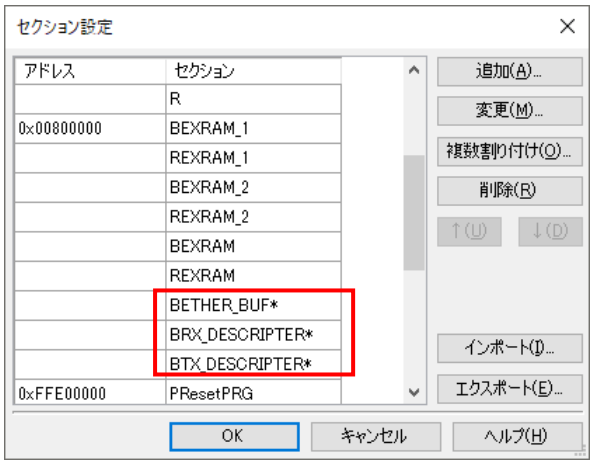
PHY チップはレジスタを持っており、通信速度(100BASE, 10BASE)や接続の規格(Full/Half-duplex)などを、レジスタ書き換えで設定します。マイコンから PHY チップへのアクセス（レジスタの読み書き）は、MDC と MDIO の 2 本の信号線で通信を行います。（MDC：クロック端子を L/H 変化させて、そのタイミングで MDIO にデータを出力する事でレジスタの書き換え。MDIO のレベルを読む事で、レジスタの読み取り。）

4.3. ディスクリプタの使用



ディスクリプタは、RAM 上に配置されたデータです。EDMAC（Ether 専用 DMA コントローラ）は、ディスクリプタに書かれている内容に従って、RAM からデータを読み取って Ether で送信、受信したデータを RAM に保存するといった動作を行います。ディスクリプタは、動作の指示書のようなものです。

Ether で送受信を行う場合は、
・RAM 上に送受信バッファを確保する
・送受信バッファのアドレスや送受信待機、続くデータがあるかなどの情報を含んだディスクリプタ（データ）を作成
・ディスクリプタのアドレスを EDMAC のレジスタに設定
の様なちょっと回りくどい操作が必要となります。
（ディスクリプタの配置アドレスは、設定したディスクリプタ長に応じて、16 バイトか 32 バイトか 64 バイトの境界（割り切れるアドレス）に配置する必要があります。）（受信バッファは、32 バイト境界に配置する必要があります。）



本プロジェクトでは、0x0080_0000（拡張 RAM 部）赤枠部に送受信バッファとディスクリプタを配置しています。ディスクリプタと送受信バッファは配置アドレスによっては動作しない点に注意が必要です。

ディスクリプタの中身は、本プロジェクトでは以下の様に設定しています。

・ether_noapi.c

```
void ether_tx_descripter_set(unsigned short n)
{
    //送信ディスクリプタ設定関数

    //戻り値
    //  なし

    //引数
    //  n : ディスクリプタ番号(0~)

    TX_DESCRIPTER_t *txd;

    txd = (TX_DESCRIPTER_t *)&g_ether_tx_descripter[n];

    txd->TD0.BIT.TFS = 0;//★作成時0
    txd->TD0.BIT.TWBI = 1;//ライトバック完了割り込み許可
    txd->TD0.BIT.TFE = 0;//★エラーはクリア
    txd->TD0.BIT.TFP = 0x3;//フレーム分割なし
    txd->TD0.BIT.TDLE = 1;//本ディスクリプタが最終（現状1つのディスクリプタしか使っていない）
    txd->TD0.BIT.TACT = 0;//★本ディスクリプタは有効ではない（送信データをセットした後で有効化）

    txd->TD1.BIT.TBL = 0;//送信データサイズ（送信時に設定）
    txd->TD1.BIT.dummy1 = 0;//未使用領域

    txd->TD2.BIT.TBA = (unsigned long)&g_ether_tx_buffer[n];//送信バッファアドレス

    txd->padding1 = 0;//パディング（クリア）

    //★はライトバック対象
}

void ether_rx_descripter_set(unsigned short n)
{
    //受信ディスクリプタ設定関数

    //戻り値
    //  なし

    //引数
    //  n : ディスクリプタ番号(0~)

    RX_DESCRIPTER_t *rxd;

    rxd = (RX_DESCRIPTER_t *)&g_ether_rx_descripter[n];

    rxd->RD0.BIT.RFS = 0;//★作成時0
    rxd->RD0.BIT.RFE = 0;//★エラーはクリア
    rxd->RD0.BIT.RFP = 0;//★フレーム分割
    rxd->RD0.BIT.RDLE = 1;//本ディスクリプタが最終（現状1つのディスクリプタしか使っていない）
    rxd->RD0.BIT.RACT = 0;//★本ディスクリプタは有効ではない（受信待機時に有効化）

    rxd->RD1.BIT.RFL = 0;//★受信データサイズ（ライトバックで設定）
    rxd->RD1.BIT.RBL = 2048;//受信バッファのサイズ

    rxd->RD2.BIT.RBA = (unsigned long)&g_ether_rx_buffer[n];//受信バッファアドレス

    rxd->padding1 = 0;//パディング（クリア）

    //★はライトバック対象
}
```

ディスクリプタの作成時は上記の内容としています。★付きの部分は EDMAC が処理した後で書き換える部分です。

ディスクリプタ作成後に、データを送信する場合は、送信ディスクリプタに、送信バイト数を設定後に、TACT を 1 にすれば送信動作。データを受信する場合は、送信ディスクリプタの RACT を 1 にする事で受信できます。

・ether_noapi.c

```
void ether_tx_descripter_act(unsigned short n, unsigned short size)
{
    //送信ディスクリプタ有効化関数

    //戻り値
    //  なし

    //引数
    //  n : ディスクリプタ番号(0~)
    //  size : 送信データサイズ

    TX_DESCRIPTER_t *txd;

    txd = (TX_DESCRIPTER_t *)&g_ether_tx_descripter[n];

    txd->TD1.BIT.TBL = size;//送信データサイズ
    txd->TD0.BIT.TACT = 1;//★本ディスクリプタを有効化

    //★はライトバック対象
}

void ether_rx_descripter_act(unsigned short n)
{
    //受信ディスクリプタ設定関数

    //戻り値
    //  なし

    //引数
    //  n : ディスクリプタ番号(0~)

    RX_DESCRIPTER_t *rxd;

    rxd = (RX_DESCRIPTER_t *)&g_ether_rx_descripter[n];

    rxd->RD0.BIT.RACT = 1;//★本ディスクリプタを有効化

    //★はライトバック対象
}
```

Ether の場合は、ディスクリプタ(送受信に必要な項目を記載したデータ)を経由して、送受信動作を行うという動作となります。

4.4. Ethernet フレームの送信

・ether_noapi.c

```
int ether_write(unsigned short descripter_num, unsigned char *data, unsigned short size)
{
    //Ether送信関数

    //戻り値
    //  0 : 送信OK（送信完了ではなく送信ディスクリプタをアクティブに設定し、EDMACに動作を引き継いだ）
    //  -2 : 引数NG

    //引数
    //  descripter_num : 使用するディスクリプタ番号(0~)
    //  *data : 送信データ先頭アドレス
    //  size : 送信データサイズ

    if (descripter_num >= DESCRIPTERS)
    {
        //ディスクリプタ番号が確保したディスクリプタ数を超えている
        return -2;
    }

    //ディスクリプタ初期化
    ether_tx_descripter_set(descripter_num);

    //送信バッファにデータをコピー
    memcpy(g_ether_tx_buffer[descripter_num], data, size);

    //ディスクリプタを有効化
    ether_tx_descripter_act(descripter_num, size);

    //EDMACにディスクリプタを読み込ませて送信
    EDMAC0.EDTRR.BIT.TR = 1;

    return 0;
}
```

本プロジェクトでは上記関数で送信を行っています。g_ether_tx_buffer は、EDMAC が読み取る送信バッファです。（送信バッファのアドレスをディスクリプタに記載し、ディスクリプタを EDMAC に渡す。）

4.5. Ethernet フレームの受信

・main.c

```
//通常の受信動作
ether_rx_descripter_act(0);//受信ディスクリプタのRACTフラグを立てる
ether_receive_start();//受信動作開始
```

※ether_rx_descripter_act()は前出

・ether_noapi.c

```
void ether_receive_start(void)
{
    //Etherの受信動作開始関数

    //戻り値
    //  なし

    //引数
    //  なし

    EDMAC0.EDRRR.BIT.RR = 1;/* EDMAC 受信ディスクリプタを読み込ませて受信動作開始 */
}
```

受信に関しては、受信ディスクリプタを有効化した後で EDMAC の受信を有効化する形です。

4.6. 割り込み関数での処理

Ether で使用する、ETHERC0, EDMAC0 の割り込みは、どちらもグループ AL1 割り込みとなります。

・ether_noapi.c　内 ether_init()

```
//割り込み設定 グループAL1
IR(ICU,GROUPAL1) = 0;/* 割り込みフラグクリア (ICU.IPR[113].BIT.IR) */
IPR(ICU,GROUPAL1) = 14;/* 割り込み優先度 (ICU.IPR[113].BIT.IPR) */

err = R_BSP_InterruptWrite(BSP_INT_SRC_AL1_EDMAC0_EINT0, (bsp_int_cb_t)Intr_ETHER0);
/* 割り込みコールバック関数の登録 */
```

スマート・コンフィグレータ環境でグループ割り込みを取り扱う場合、スマート・コンフィグレータの関数でコールバック関数を定義する方法が（スマート・コンフィグレータが生成したコードを変更したりする必要がないので）スマートかと思います。ここでは、グループ AL1 割り込みが生じた際、ユーザ関数として定義した、Intr_ETHER0()が呼ばれる様に設定しています。

・ether_noapi.c

```
void Intr_ETHER0(void)
{
    //ETHER0割り込み関数

    //ETHER割り込み
    if (ETHERC0.ECSR.LONG != 0)
    {
        ether_callback();
    }

    //EDMAC割り込み
    if (EDMAC0.EESR.LONG != 0)
    {
        ether_int_handler();
    }
}
```

グループ AL1 の割り込みは、ETHERC0 起因の場合は ether_callback()を、EDMAC 起因の場合は ether_int_handler()を呼びます。これらの関数の名称は、RX65N_MAGIC_PACKET/RX65N_MAGIC_PACKET2 プロジェクトと同じです。

・ether_interrupt.c

```
void ether_int_handler(void)
{
    //EDMAC割り込み

    unsigned short size;
    RX_DESCRIPTER_t *rxd;

    extern volatile unsigned long g_ether_frame_receive_num;
    extern volatile unsigned long g_ether_rx_rde_count;
    extern volatile unsigned long g_ether_rx_rfcof_count;

    //送信完了
    if (EDMAC0.EESR.BIT.TC != 0)
    {
        EDMAC0.EESR.BIT.TC = 1;//フラグクリア
    }

    //フレーム受信
    if (EDMAC0.EESR.BIT.FR != 0)
    {
        EDMAC0.EESR.BIT.FR = 1;//フラグクリア

        g_ether_frame_receive_num++;//割り込みに来た回数をカウント

        if (g_ether_recv_flag == FALSE)//受信フラグが立っている状態で次のデータを受信した場合はバッファへのコピ
ーは行わない
        {
            rxd = (RX_DESCRIPTER_t *)&g_ether_rx_descripter[0];//本プログラムではディスクリプタ番号0のみ使用
            size = rxd->RD1.BIT.RFL;//受信データサイズ
            memcpy(g_ether_recv_buf, &g_ether_rx_buffer[0], size);//受信データをg_ether_recv_bufにコピー
            g_ether_recv_flag = TRUE;//受信フラグを立てる
    }

    ether_rx_descripter_set(0);//受信ディスクリプタ設定
    ether_rx_descripter_act(0);//受信ディスクリプタのRACTフラグを立てる
    ether_receive_start();//受信動作開始
}
```

割り込み要因が、データ受信の場合は、前のプロジェクト同様 g_ether_recv_buf にデータをコピーする処理です。本プログラムでは、受信ディスクリプタを 1 つのみ定義していますので、受信データが入ってくるのは受信バッファの 0 番(g_ether_rx_buffer[0])に固定しています。
（複数の受信ディスクリプタを使用する場合は、ディスクリプタの中身を見てデータを受信している受信バッファのデータを読み出す処理が必要になってきます。本プログラムは、API を使わず、ディスクリプタをユーザが操作する一番単純なサンプルプログラムとしています。）

受信データがあった場合は、メイン関数内で自分宛の MagicPacket かどうかを判定して、端末表示や LED の点灯を行う形で、この部分は MAGIC_PACKET2 プロジェクトと同じです。

4.7. 補足

本プロジェクトでは、自前で用意したディスクリプタを使いデータの送受信を行っています。

起動後に、s コマンドで動作を開始すると、30 秒毎に赤字のデータが表示されます。

```
>command=s

Operation start with MagicPacket send/receive mode.
PUSH SW3 -> send MagicPacket
target MAC address         = 00-0D-76-00-40-02
target IP address          = 192.168.0.255
this board MAC address = 00-0D-76-00-40-01
Ether0: LINK-DOWN
Ether0: LINK-UP
read_data / receive_data = 68 / 68
read_data / receive_data = 190 / 190
read_data / receive_data = 278 / 278
read_data / receive_data = 380 / 380
read_data / receive_data = 499 / 499
```

read_data / receive_data = 68 / 68　の場合は、

メイン関数内のメインループで 68 個のデータを処理(read_data)

受信割り込みに飛んできた回数 68 回（68 個のデータを受信）(receive_data)

という意味です。

この数が合わない場合は、受信したデータをメイン関数内で処理する前に次の受信割り込みが来たという事を意味します。（このプロジェクトでは、受信バッファを 1 つしか用意していないが、受信バッファの数を増やす必要がある、メイン関数内でのデータ処理を軽くする必要がある、などです。）

※本プロジェクトで受信対象となる（受信割り込みに飛んでくる）のは、宛先 MAC アドレスが「ブロードキャストアドレス」の場合か「自分宛」の場合です。それ以外のデータは、そもそも受信対象外です。

また、

```
RDE interrupt     = 10
RFCOF interrupt = 10
```

上記の様に

RDE interrupt : 受信ディスクリプタ枯渇割り込み回数

RFCOF interrupt : 受信フレームカウンタオーバフロー割り込み回数

「受信ディスクリプタ枯渇割り込み」や「受信フレームカウンタオーバフロー割り込み」が生じた際は、その回数が表示されます。

受信ディスクリプタ枯渇は、本プロジェクトでは受信ディスクリプタを 1 つしか使っていないので、連続してデータを受信した場合発生する可能性があります。（受信割り込みでバッファからデータをコピーして、次の受信待機にする前に、次のデータが来た場合など）。受信ディスクリプタ枯渇が頻発する場合は、単純に受信ディスクリプタ数を増やすという対策が有効かと考えます。

受信フレームカウンタオーバフローは、マイコンが持っている Ether の受信 FIFO が溢れた場合に発生します。

ディスクリプタを自前で制御した場合、Ether で受信するデータに対して、処理が間に合っているかどうかの判断のため、本プロジェクトでは表示する様にしています。

本プロジェクトは実験的な要素が強いプロジェクトです。送信用と受信用にディスクリプタ 1 つだけ用意し、API 関数を使わずに送受信ができる事を示しました。

API 関数を使わずに、複数のディスクリプタを操作しながらデータの送受信を行うのは、結構手間が掛かります。実際のプログラミングでは、ディスクリプタの操作が隠蔽化された API 関数を使うのが現実的ではないかと思います。Ether モジュールがどのような形で EDMAC を制御して、データの送受信を行っているのか。本プロジェクトでは、大まかな動作原理を理解できれば良いと思います。

4.8. ユーザ作成関数

ether_init

概要:Ethernet モジュールの初期化
宣言:void ether_init(void)
説明:
・ETHERC のレジスタ設定
・EDMAC のレジスタ設定
・割り込み設定、割り込みコールバック関数の登録
を行います
引数:なし
戻り値:なし

ether_open

概要:Ethernet の動作開始
宣言:void ether_open(void)
説明:
・割り込み有効化
・ETHERC の送受信機能の有効化
を行います
引数:なし
戻り値:なし

ether_receive_start

概要:受信ディスクリプタの有効化
宣言:void ether_receive_start(void)
説明:
・EDMAC に受信ディスクリプタを読み込ませて受信動作の開始
を行います
引数:なし
戻り値:なし

ether_pin_setting

概要:Ethernet 関連の端子設定
宣言:void ether_pin_setting(void)

説明:
・Ethernet に使用する端子（PFS, PMR 設定）
を行います
引数:なし
戻り値:なし

ether_phy_setting

概要:PHY-LSI の初期設定
宣言:void ether_phy_setting(void)

説明:
・ボード上の PHY-LSI(LAN8700)の初期化
を行います
引数:なし
戻り値:なし

ether_tx_descripter_set

概要:送信ディスクリプタ設定
宣言:void ether_tx_descripter_set(unsigned short n)
説明:
・送信ディスクリプタの初期設定
を行います
引数:
　unsigned short n: ディスクリプタ番号
戻り値:なし

ether_rx_descripter_set

概要:受信ディスクリプタ設定
宣言:void ether_rx_descripter_set(unsigned short n)
説明:
・受信ディスクリプタの初期設定
を行います
引数:
　unsigned short n: ディスクリプタ番号
戻り値:なし

ether_tx_descripter_act

概要:送信ディスクリプタ有効化
宣言:void ether_tx_descripter_set(unsigned short n)
説明:
・送信ディスクリプタの有効化
を行います
引数:
　unsigned short n: ディスクリプタ番号
戻り値:なし
補足:送信バッファに送信データを格納後、本関数を呼び出すとデータが送信されます
　　ユーザが直接実行しても問題ありませんが、基本的には ether_write()関数内で呼び出される関数です

ether_rx_descripter_act

概要:受信ディスクリプタ有効化
宣言:void ether_rx_descripter_set(unsigned short n)
説明:
・受信ディスクリプタの有効化
を行います
引数:
　unsigned short n: ディスクリプタ番号
戻り値:なし
補足:本関数を呼び出すとデータの受信動作が開始されます

ether_write

概要:Ethernet フレーム送信関数
宣言:int ether_write(unsigned short descripter_num, unsigned char *data, unsigned short size)
説明:
・Ethernet フレームの送信
を行います
引数:
　unsigned short descripter_num: ディスクリプタ番号
　unsigned char *data: 送信データ
　unsigned short size: 送信バイト数
戻り値:
　0: 正常終了
　-2: 引数エラー

補足:EDMAC に送信の指示を出した場合、正常終了となります
　　正常終了＝送信完了ではありません

phy_read

概要:PHY レジスタ読み出し
宣言:unsigned short phy_read(unsigned short phy_addr, unsigned short reg_addr)
説明:
・PHY-LSI のレジスタ読み出し
を行います
引数:
　unsigned short phy_addr: PHY アドレス（本ボードでは 31）
　unsigned short reg_addr: レジスタアドレス
戻り値:
　レジスタ値

phy_write

概要:PHY レジスタ書き込み
宣言:void phy_write(unsigned short phy_addr, unsigned short reg_addr, unsigned short data)
説明:
・PHY-LSI のレジスタ書き込み
を行います
引数:
　unsigned short phy_addr: PHY アドレス（本ボードでは 31）
　unsigned short reg_addr: レジスタアドレス
　unsigned short data: レジスタに書き込む値
戻り値:なし

5. RX65N_PING プロジェクト

CD 内の

SOURCE¥144_100PIN¥RX65N_PING
SOURCE¥176PIN¥RX65N_PING

（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 ping 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 ping に対して応答する
事を行います。

本プロジェクトでは、API 関数を使用する形です。

・スマート・コンフィグレータで追加するコンポーネント

コンポーネント	備考
Config_PORT	SW と LED のポート設定
Config_SCI1	SCI(UART)通信
r_ether_rx	Ethernet
Config_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
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

arp, ethernet, ip, ping のフォルダが新規です。

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 ping sample program.

COMMAND:
   i : target IP address set
   j : this board IP address set
   m : this board MAC address set
   p : print IP/MAC address
   S : IP address swap/MAC address increment
---
   s : Operation start
---

USAGE:
   SW3 : ping send
   LED : ping received

-default setting address-
---
target IP address is          -> 192.168.0.81 : i command for change, S command for target<->source swap
this board IP address is    -> 192.168.0.80 : j command for change, S command for source<->target swap
this board MAC address is -> 00-0D-76-00-40-01 : m command for change, S command for increment MAC
address
---
First m, i, j, p, S command -> initial setting

Please input 's' for operation start!
>
```

デフォルトは、

ping 先 IP アドレス　192.168.0.81　　（i コマンドで変更）

本ボードの IP アドレス　192.168.0.80　　（j コマンドで変更）

本ボードの MAC アドレス　00-0D-76-00-40-01　（m コマンドで変更）

S コマンドで、ping 先と本ボードの IP アドレスの交換，送信先と本ボードの MAC アドレスの交換です。（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。）

s コマンドで動作開始です。

```
>command=s

Operation start !
PUSH SW3 -> send ping to target
Ether0: LINK-UP
```

SW3 を押すと、設定した IP アドレスに対して ping を打ちます。

（2 回以上 SW3 を押してください、2 回目以降で ping を送信）

5.1. 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
・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

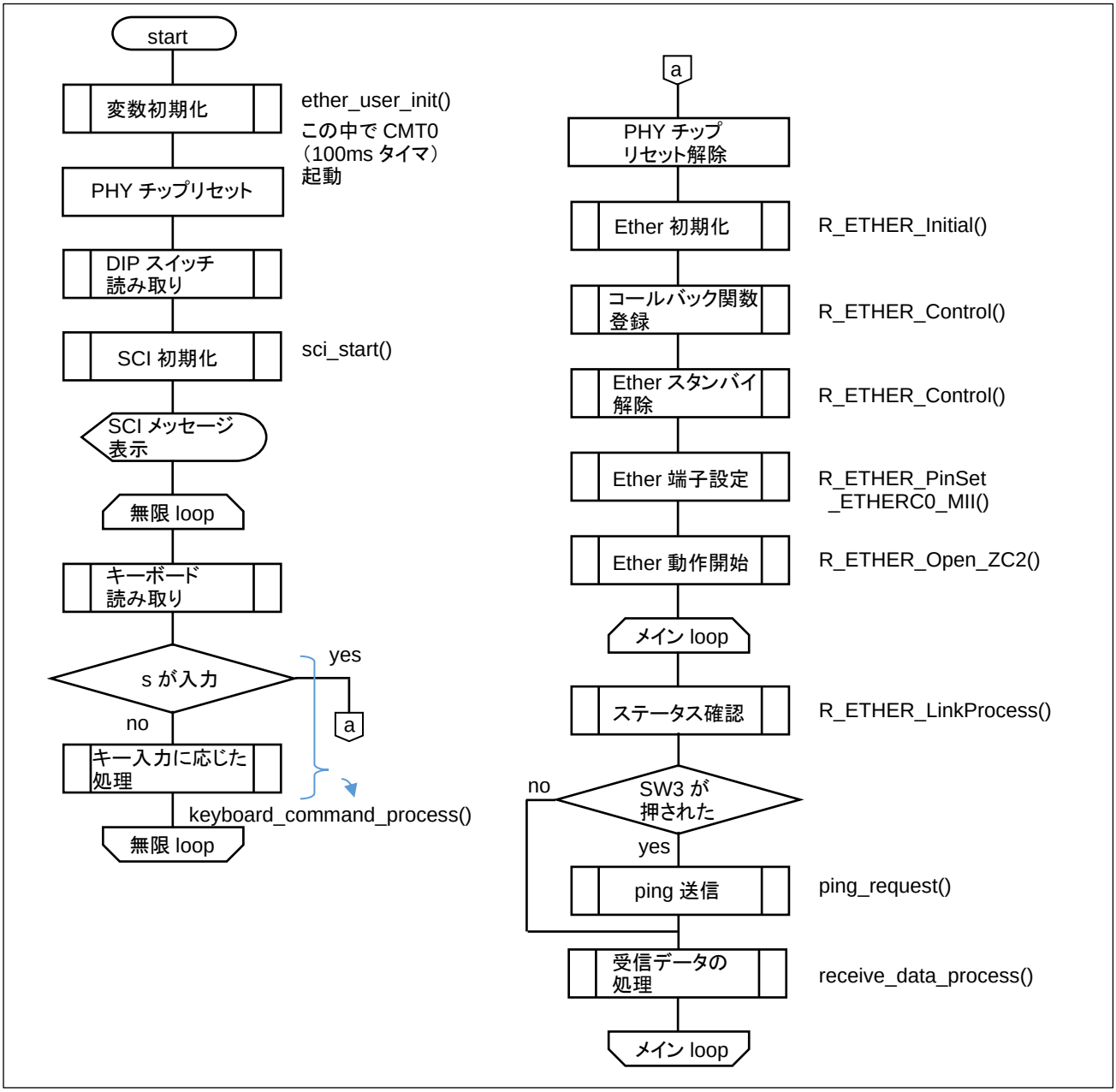
の処理を行っています。

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

となっています。

・ether_main() （main.c 内）



RX65N_MAGIC_PACKET2 プロジェクトとそれ程は変わりません。

本プロジェクトでは、s コマンドで動作開始後、

SW3 で ping を送信

受信データの中に自分宛の ping 要求が含まれていた場合、ping データを送信

という動作となります。

5.2. 割り込み関数での処理

・ether_interrupt.c, ether_int_handler()内

```
//受信処理
if ((p_arg->status_eesr & ETHER_EESR_FR) != 0)
{
    while (1)//複数の受信データがある場合は全てのデータを受信キューにコピーする
    {
        ret = R_ETHER_Read_ZC2(g_ether_channel, (void **)&buffer_address);
        if (ETHER_NO_DATA < ret)
        {
            ether_data_enqueue(&buffer_address, ret);
            ret2 = R_ETHER_Read_ZC2_BufRelease(g_ether_channel);

            if (ret2 != ETHER_SUCCESS)
            {
                sci_write_str("Error: BufRelease fail.¥n");
            }
        }
        else
        {
            break;//受信データなし
        }
    }
}
```

受信の処理を、RX65N_MAGIC_PACKET2 から変えています。RX65N_MAGIC_PACKET2 では、

・受信バッファへのコピー

・受信フラグを立てる

というものでしたが、これらの処理を ether_data_enqueue()関数で処理する様にしています。

単に関数に追い出した訳ではなく、複数のデータを保持できる様に拡張しています。

ether_data_enqueue()が、受信データをキュー（リングバッファ）に格納する関数です。

・ether_common.c

```
int ether_data_enqueue(unsigned char **src_address, unsigned long size)
{
    //受信データをリングバッファにコピーする関数

    //戻り値
    //  ETHER_SUCESS(0) : 正常終了
    //  ETHER_RET_FIFO_OVERFLOW(-13) : リングバッファフル

    //引数
    //  **src_address : 受信データの先頭アドレス
    //  size : 受信データのサイズ

    unsigned long index;
    unsigned long psw_i;

    if ((g_ether_recv_buf_write_index - g_ether_recv_buf_read_index) == ETHER_RECV_BUF_SIZE)
    {

#ifdef ETHER_RECV_BUF_OVERRIDE

        //キューが溢れた場合古いデータを捨てる（一番古いデータを読み出し済みとする）

        //read_index変数更新の際に一時的に割り込み禁止とする
        psw_i = __get_psw() & 0x00010000UL;
        clrpsw_i();
        g_ether_recv_buf_read_index++;
        if (psw_i) setpsw_i();

        g_ether_recv_buf_override = TRUE;//上書きフラグをセット

#else
        //キューが溢れた場合はデータを格納しない

        return ETHER_RET_FIFO_OVERFLOW;
#endif
    }

    //リングバッファにデータを格納
    index = g_ether_recv_buf_write_index & (ETHER_RECV_BUF_SIZE - 1);
    memcpy(g_ether_recv_buf[index], *src_address, size);

    //受信サイズを保存しておく
    g_ether_recv_size[index] = size;

    //write_indexを進める
    //write_index変数更新の際に一時的に割り込み禁止とする
    psw_i = __get_psw() & 0x00010000UL;
    clrpsw_i();
    g_ether_recv_buf_write_index++;//write_indexに書き込みを行うのはこの部分のみ
    if (psw_i) setpsw_i();

    return ETHER_SUCCESS;
}
```

デフォルトはこちら側を選択
（キューが溢れた場合古いデータを捨てる）
（最新のデータはキューに保存される）

g_ether_recv_buf[ETHER_RECV_BUF_SIZE][ETHER_MAX_PACKET_SIZE]　受信データを保存

g_ether_recv_size[ETHER_RECV_BUF_SIZE]　受信データバイト数を保存

・ether_common.h

```
//受信リングバッファサイズ
#define ETHER_RECV_BUF_SIZE  16//1514×バッファ数のメモリを消費, 2のべき乗である必要がある
#define ETHER_MAX_PACKET_SIZE  1514//イーサネットのフレーム最大サイズ(1518)からCRCの4バイトを引いた値
```

データ保存用のバッファは、リングバッファとしており、複数のデータを保存できる様にしています。デフォルトでは、ether_common.h 内で、16(=ETHER_RECV_BUF_SIZE)個のバッファを確保しています。
※バッファの数は 2 のべき乗の値としてください

RX65N_MAGIC_PACKET2 プロジェクトでは、何バイト受信したかの情報は保存していませんでしたが、本プロジェクトでは、別変数に受信バイト数を保存しています。

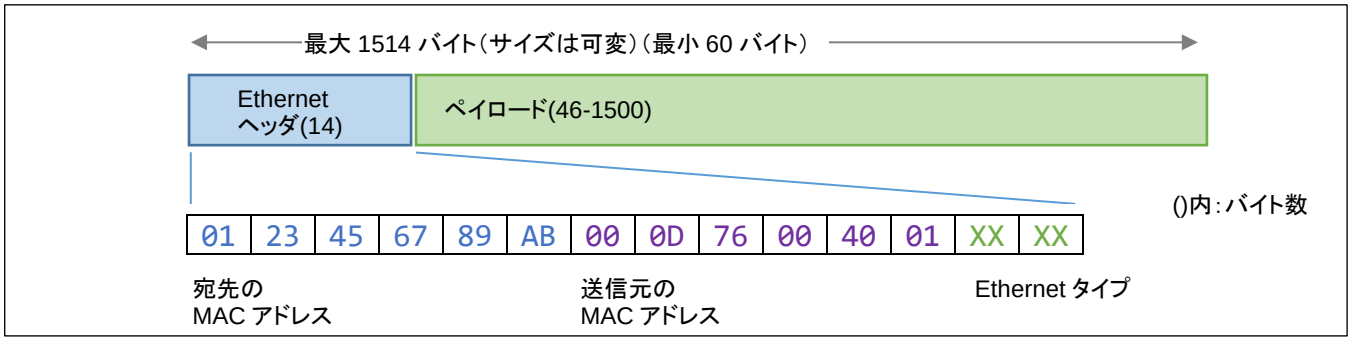
5.3. ping とは

ping は、調べる IP アドレスの機器が応答するかどうかを確認する手段です。
（ping にあえて応答しない機器もあるので、ping に対する応答がない＝その機器がダウンしているという訳でもありませんが。）
今までのプロジェクトでは、MAC アドレスでの通信を取り扱っていましたが、本プロジェクトでは IP アドレスも取り扱います。
（※RX65N_MAGIC_PACKET2 では、IP/UDP の通信を行っていますが、その際は IPアドレスを取り扱う方法に関して述べていませんでしたので、実質的に本章で初めて IP アドレスを取り扱う事となります。）

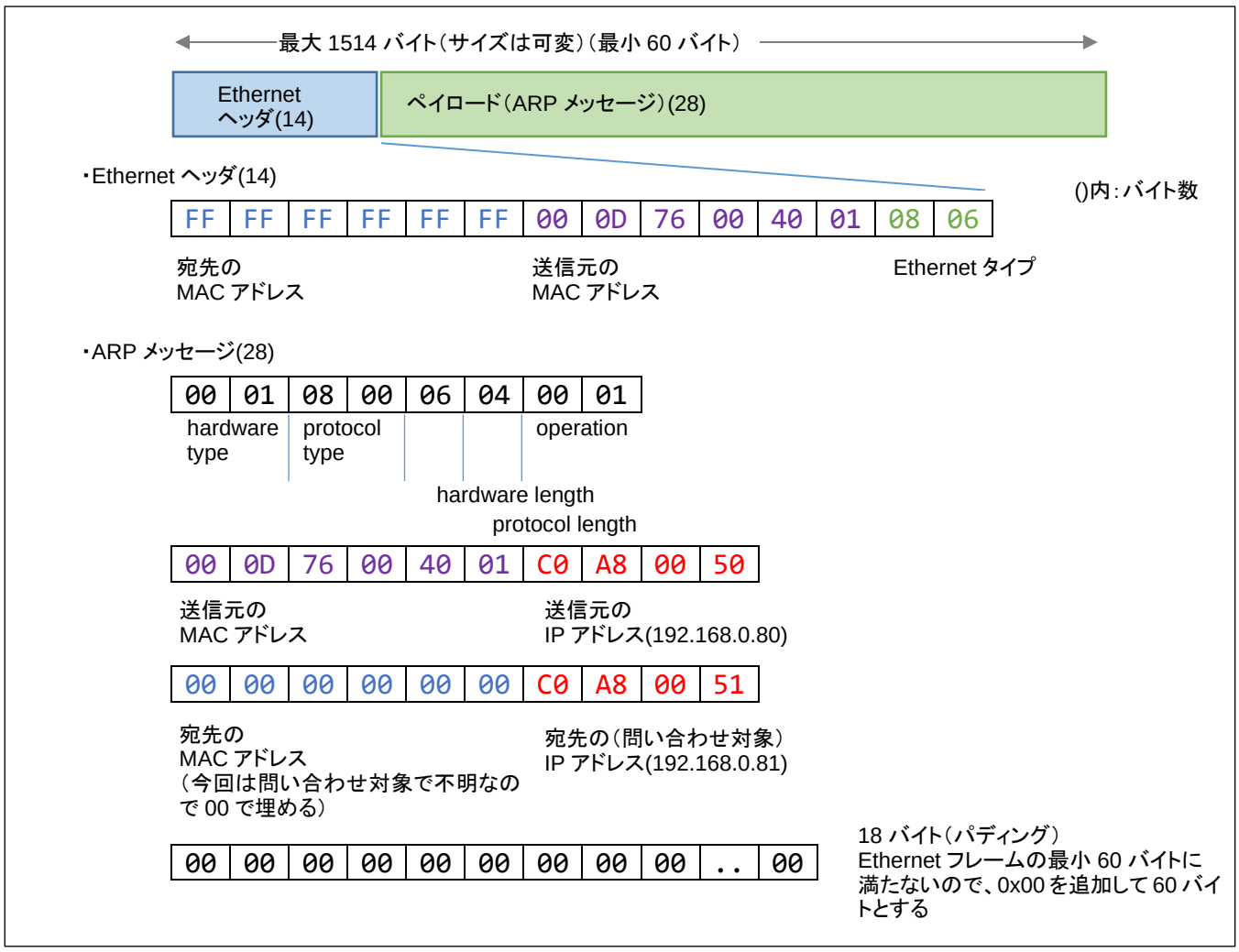
5.4. ARP リクエスト

PC のコマンドプロンプトから ping を実行する場合、
ping 192.168.0.81
の様に実行します。ping の宛先は、IP アドレスになります。

Ethernet フレームの構造としては、



最初に宛先 MAC アドレスが入ります。よって、ping を送る場合でも、送り先の MAC アドレスの情報が必要になるという事です。（Ethernet フレームのペイロードに何かを格納して送る場合、必ず宛先の MAC アドレスが判っている必要がある。）
今は、ping を送る IPアドレスは決まっているが、その IP アドレスを使用している機器の MAC アドレスが判らない状態だとします。その場合に使用されるのが、ARP リクエストという通信です。192.168.0.81 を使っている機器の MAC アドレスを問い合わせるプロトコルです。



ARP リクエストは、上記の様なパケットとなっているので、単純に 60 バイトのデータを作って、API 関数(R_ETHER_Write)で送信すれば良いです。

起動後に、s コマンドで動作を開始します。

```
>command=s

Operation start !
PUSH SW3 -> send ping to target
Ether0: LINK-UP
```

初回 SW3 を押した際、ARP リクエストを送信します。

```
ping 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
```

赤字の行は、ARP リプライを返す機器が存在する場合に表示されます。

・arp.c

```
void ARP_request(unsigned char *dst_IP)
{
    //ARPリクエスト送信関数

    //戻り値
    //  なし

    //引数
    //  *dst_IP  : 問い合わせ対象のIPアドレス

    ETHERNET_HEADER ethernet_header;
    ARP_MESSAGE arp_message;

    unsigned char send_data[60] = {0};//Ethernetパケットの最小サイズ
    ether_return_t ret;
    unsigned short i;

    for (i=0; i<6; i++)
    {
        ethernet_header.eth_dst_MAC[i] = 0xff;//送信先MACアドレス:ブロードキャストアドレス
    }

    for (i=0; i<6; i++)
    {
        ethernet_header.eth_src_MAC[i] = g_src_MAC[i];//（Etnernetパケット内の）送信元MACアドレス:本ボードの
MACアドレス
    }

    ethernet_header.eth_ethernet_type = 0x0806;//上位プロトコルの種類(ARP)

    arp_message.arp_hardware_type = 0x0001;//ネットワークの物理媒体の種類（イーサネット）
    arp_message.arp_protocol_type = 0x0800;//上位プロトコルの種類(IP)
    arp_message.arp_hardware_length = 6;//ネットワークの物理媒体のアドレス長（MACアドレス）
    arp_message.arp_protocol_length = 4;//上位プロトコルのアドレス長(IPV4)
    arp_message.arp_operation = 1;//ARPの動作（ARPリクエスト）

    for (i=0; i<6; i++)
    {
        arp_message.arp_src_MAC[i] = g_src_MAC[i];//（ARPパケット内の送信元MACアドレス:本ボードのMACアドレス
    }

    for (i=0; i<4; i++)
    {
        arp_message.arp_src_IP[i] = g_src_IP[i];//送信元IPアドレス:本ボードのIPアドレス
    }

    for (i=0; i<6; i++)
    {
        arp_message.arp_dst_MAC[i] = 0x00;//宛先MACアドレス
    }

    for (i=0; i<4; i++)
    {
        arp_message.arp_dst_IP[i] = dst_IP[i];//宛先IPアドレス
    }

    //ARPパケット = Ethernetヘッダ(14bytes) + ARPメッセージ(28bytes)…60バイトに満たないので残りは0x00を送信
    ETHERNET_header_to_stream(&ethernet_header, send_data);//Ethernetヘッダの14バイトをsend_dataに設定
    ARP_message_to_stream(&arp_message, &send_data[14]);//ARPメッセージの28バイトをsend_data[14]～に設定

    //API関数を使いデータを送信
    ret = R_ETHER_Write(g_ether_channel, (void *)send_data, sizeof(send_data));

（後略）
```

本関数で行っている事は、前ページのデータを作って API 関数で送信するという単純なものです。

ここでは、14 バイトの Ethernet ヘッダと 28 バイトの APP メッセージは、構造体を作ってデータを埋める様にしています。最終的には送信データはバイトストリームになるのですが、構造体のメンバでデータにアクセスすると何バイト目が何のデータなのかが判り易いというメリットがあるためです。

・ether_common.h

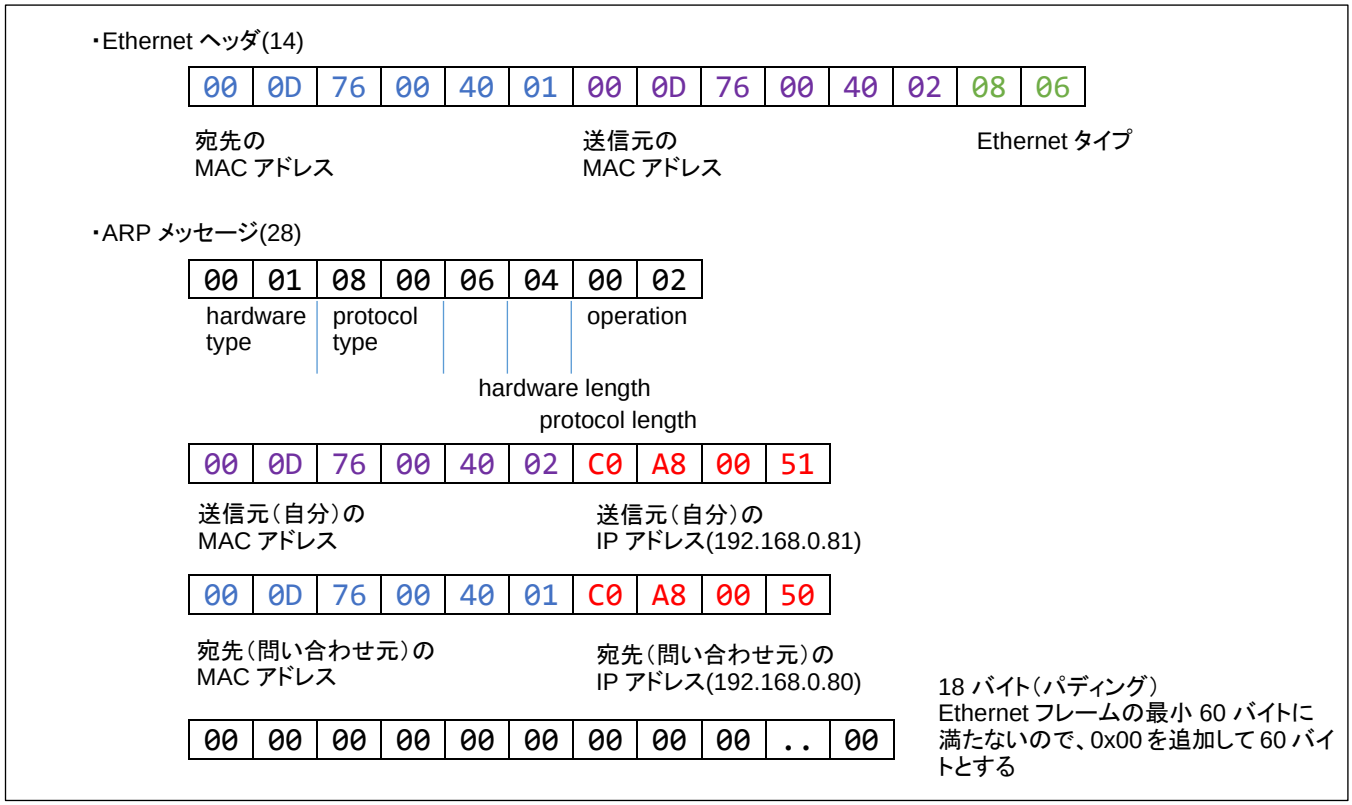
```
typedef struct
{
    //イーサネットヘッダ(14bytes)
    unsigned char eth_dst_MAC[6];
    unsigned char eth_src_MAC[6];
    unsigned short eth_ethernet_type;
}ETHERNET_HEADER;

typedef struct
{
    //ARPメッセージ(28bytes)
    unsigned short arp_hardware_type;
    unsigned short arp_protocol_type;
    unsigned char arp_hardware_length;
    unsigned char arp_protocol_length;
    unsigned short arp_operation;
    unsigned char arp_src_MAC[6];
    unsigned char arp_src_IP[4];
    unsigned char arp_dst_MAC[6];
    unsigned char arp_dst_IP[4];
}ARP_MESSAGE;
```

構造体のメンバに値を代入した後で、バイトストリームに変換して送信。

また、受信したデータを構造体のメンバに分解してアクセスする目的で、構造体を利用しています。

5.5. ARP リプライ

例えば、192.168.0.81 に対して ARP リクエストを行った場合、192.168.0.81 の MAC アドレスを教えてくれるのは、192.168.0.81 の機器です。192.168.0.81 の機器は、ARP リクエスト（宛先 MAC アドレスは、ブロードキャスト(FF-FF-FF-FF-FF-FF)）を受け取ると、ARP リプライという形でネットワークに自分自身の MAC アドレスの情報を送信します。



ARP リプライの場合、送信元、送信先の MAC アドレス、IP アドレスが変わるのと、ARP の operation が 02（リプライ）に変わります。

外部から ARP リクエストを受信した場合に、ARP リプライを返す動作となります。

```
ARP reply send (this board MAC address = 00-0D-76-00-40-02) to 00-0D-76-00-40-01
```

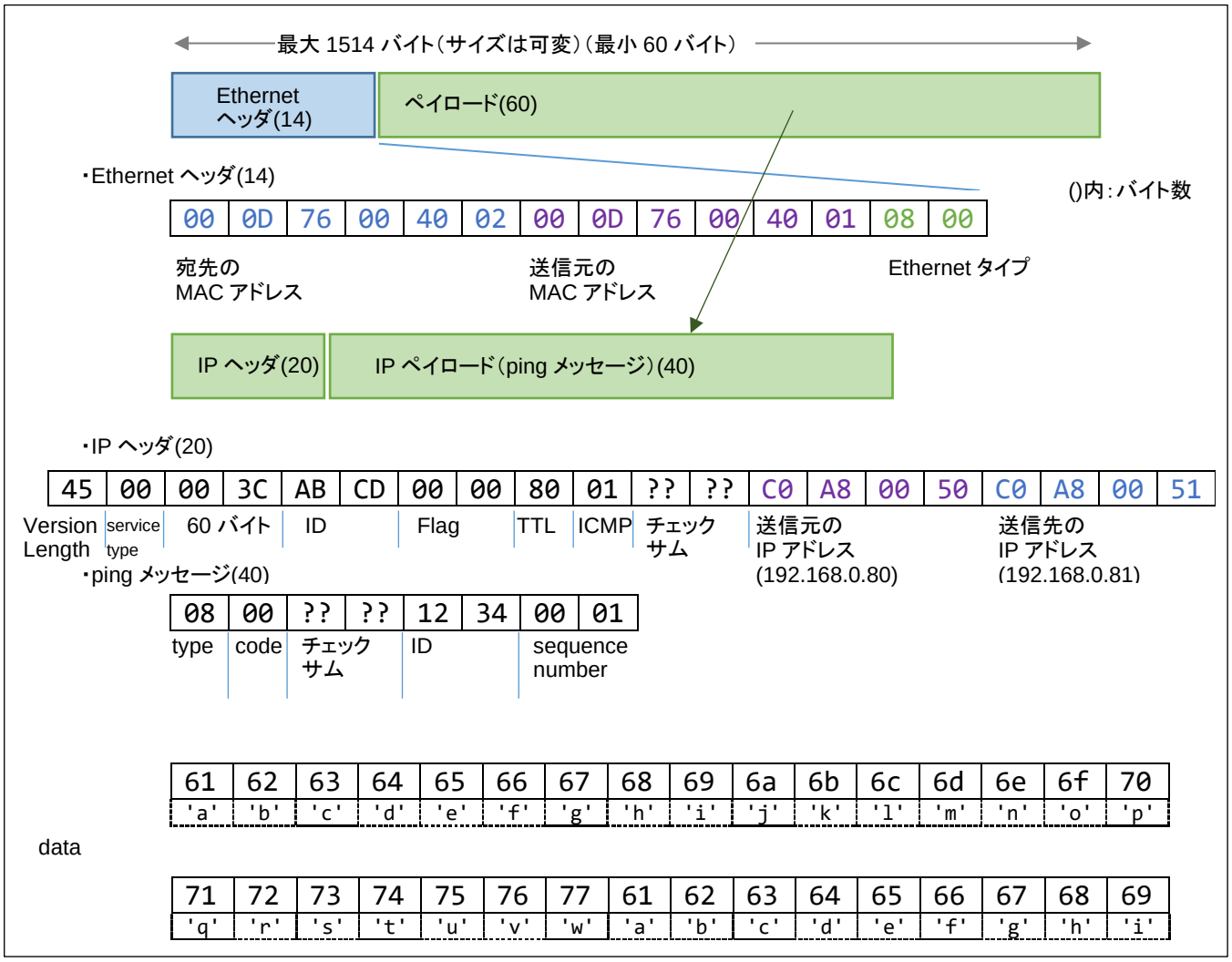
・arp.c

```
void ARP_reply(unsigned char *packet)
{
    //ARP応答関数

    //戻り値
    //  なし

    //引数
    //  *packet : 受信したARPパケット

    ETHERNET_HEADER ethernet_header;
    ARP_MESSAGE arp_message;

    unsigned char send_data[60] = {0};//Ethernetパケットの最小サイズ

    ether_return_t ret;

    unsigned short i;

    //受信したARPパケットをEthernetヘッダとARPメッセージ構造体に分解して格納
    stream_to_ETHERNET_header(packet, &ethernet_header);
    stream_to_ARP_message(&packet[14], &arp_message);

    //宛先MACアドレスは送信元の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_header.eth_dst_MAC[i] = ethernet_header.eth_src_MAC[i];
        ethernet_header.eth_src_MAC[i] = g_src_MAC[i];
    }

    arp_message.arp_operation = 0x0002;//ARPの動作（ARPリプライ）

    //宛先MACアドレスは送信元の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++)
    {
        arp_message.arp_dst_MAC[i] = arp_message.arp_src_MAC[i];
        arp_message.arp_src_MAC[i] = g_src_MAC[i];
    }

    //宛先IPアドレスは送信元のIPアドレスを設定，送信元は本ボードのIPアドレスを指定
    for (i=0; i<4; i++)
    {
        arp_message.arp_dst_IP[i] = arp_message.arp_src_IP[i];
        arp_message.arp_src_IP[i] = g_src_IP[i];
    }

    //ARPパケット = Ethernetヘッダ(14bytes) + ARPメッセージ(28bytes)…60バイトに満たないので残りは0x00を送信
    ETHERNET_header_to_stream(&ethernet_header, send_data);//Ethernetヘッダの14バイトをsend_dataに設定
    ARP_message_to_stream(&arp_message, &send_data[14]);//ARPメッセージの28バイトをsend_data[14]～に設定


    ret = R_ETHER_Write(g_ether_channel, (void *)send_data, sizeof(send_data));
（後略）
```

ARP リプライ関数では、受信したデータを引数として受け取り、送信元のアドレスに自分自身のアドレスを設定。送信元のアドレスに受信したデータ内のアドレスを設定する形としています。所定のデータを作って API 関数で送信するという、基本的なところは変わっていません。（ARP のプロトコルを理解して、何バイト目にどんなデータを送るべきなのかが判っていれば、プログラムを作成できます。）

5.6. ping リクエスト

マイコンボードから他のボードや PC に対して、ping を打つ場合以下の様なデータ（ping リクエスト）を送信します。



ping リクエストの送信時も、所定のデータを作って API 関数で送信するという基本的な流れは変わりません。

ping の場合は、IP ベースでの通信となるので、Ethernet のペイロードが、IP ヘッダ＋IP ペイロードという形で構成されます。ping 以外でも IP を使った通信は、同じようなデータ構成となります。

（ping リクエストは、IP ベースの通信の簡単なプロトコルです）

起動後、s コマンドで動作開始後、1 回目の SW3 で ARP リクエスト送信。2 回目以降の SW3 で ping リクエストの送信を行います。

```
ping request send to 192.168.0.81
ping reply received from 192.168.0.81
```

2 行目（赤字）は、相手から ping リプライが返ってきた場合に表示されます。

・ping.c

```
void ping_request(unsigned char *dst_IP)
{
    unsigned short i;

    unsigned long sum;

    ETHERNET_HEADER ethernet_packet;
    IP_HEADER ip_packet;
    PING_MESSAGE ping_packet;

    ether_return_t ret;

    unsigned char send_data[14+20+40] = {0};

    static unsigned short seq_no = 1;

    //pingの宛先の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("ping 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
        //（次回本関数が呼ばれた場合でIP-MAC対応が判明している場合はpingを送る）

        return;
    }

    //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 = 60;

    //パケットの識別子(ID)
    ip_packet.ip_id = 0xABCD;

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

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

・ping.c（続き）

```
    //上位プロトコルの種類(ICMP)
    ip_packet.ip_protocol = 0x01;

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

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

    //pingメッセージの設定//

    //メッセージの種類（pingリクエスト）
    ping_packet.ping_type = 0x08;

    //コード
    ping_packet.ping_code = 0x00;

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

    //パケットの識別子
    ping_packet.ping_id = 0x1234;

    //シーケンス番号
    ping_packet.ping_sequence_number = seq_no++;//1から始まり送信毎にインクリメント

    //データ（ここでは32バイトとする）
    //送信データ:abcdefghijklmnopqrstuvwabcdefghi（Windowsのpingが送るデータに合わせているだけで、この並びで送らなければならないという訳ではない）
    for (i=0; i<23; i++) ping_packet.ping_data[i] = (unsigned char)('a' + i);
    for (i=0; i<9; i++) ping_packet.ping_data[23 + i] = (unsigned char)('a' + i);

    //送信IPヘッダのチェックサム計算
    sum = ip_header_sum(&ip_packet);//IPヘッダの1の補数和
    sum = (~sum) & 0xffff;//NOT演算反転
    ip_packet.ip_checksum = (unsigned short)sum;

    //pingメッセージのチェックサム計算
    sum = ping_message_sum(&ping_packet);//pingメッセージの1の補数和
    sum = (~sum) & 0xffff;//NOT演算反転
    ping_packet.ping_checksum = (unsigned short)sum;

    //pingパケット = ETHERNETヘッダ(14) + IPヘッダ(20) + pingメッセージ(40)
    ETHERNET_header_to_stream(&ethernet_packet, send_data);
    IP_header_to_stream(&ip_packet, &send_data[14]);
    PING_message_to_stream(&ping_packet, &send_data[14+20]);
    ret = R_ETHER_Write(g_ether_channel, (void *)send_data, sizeof(send_data));

（後略）
```

ping_request()関数を呼び出した際、初回は送信先の MAC アドレスが判明していないので、MAC アドレスの問い合わせ(ARP_request)を実行します。その後、関数は抜けてしまいます。

→初回の ping_request()では、ARP リクエストのみ実行し、実際に ping は送りません

→2 回目以降、ping_request()を実行した際は、ping リクエストのパケットを送信します

（上記の仕様としています。ping リクエストを送信する際は、

(1)ARP リクエストを送信

(2)ARP リプライを受信（送信先の MAC アドレスの情報はこの段階で判明）

(3)ping リクエストの送信

のフローで処理を行う必要があります。「送信」「送信」であれば、順に実行すればよいですが、間に受信が入るので 1 回目は(1)のみ実行。2 回目以降は、(3)を実行する関数として構成しています。）
（※ARP リプライを受信しない場合は、2 回目以降も(1)を実行して終了します。）

・ether_common.h

```
typedef struct
{
    //IPヘッダ(20bytes)
    unsigned char ip_version_length;
    unsigned char ip_service_type;
    unsigned short ip_total_length;
    unsigned short ip_id;
    unsigned short ip_flags_fragment_offset;
    unsigned char ip_time_to_live;
    unsigned char ip_protocol;
    unsigned short ip_checksum;
    unsigned char ip_src_IP[4];
    unsigned char ip_dst_IP[4];
}IP_HEADER;

typedef struct
{
    //pingメッセージ(40bytes)
    unsigned char ping_type;
    unsigned char ping_code;
    unsigned short ping_checksum;
    unsigned short ping_id;
    unsigned short ping_sequence_number;
    unsigned char ping_data[32];
}PING_MESSAGE;
```

IP ヘッダと ping メッセージのデータの構造体は上記の様になっています。

IP ヘッダは最低 20 バイト、オプションを含めると、20+4×n（20 より大きい 4 の倍数）バイトとなりますが、本プログラムではオプションは含めず、20 バイトのデータとして取り扱います。

IP ヘッダや ping メッセージには、チェックサムというフィールドがあり、データ化け検出のため、所定の計算を行ったデータを埋め込む必要があります。（詳細な計算アルゴリズムは取扱説明書の方に記載があります。）

・ip.c

```
unsigned long ip_header_sum(IP_HEADER *ip_header)
{
    unsigned long sum = 0;

    //2バイト単位で合計値を計算
    //計算結果が0xffffより大きい場合は最上位の桁を消去して、あたらめて1を加算する

    sum += ((unsigned long)ip_header->ip_version_length) << 8;
    sum += (unsigned long)ip_header->ip_service_type;
    sum = one_complement_sum(sum);

    sum += (unsigned long)ip_header->ip_total_length;
    sum = one_complement_sum(sum);

    sum += (unsigned long)ip_header->ip_id;
    sum = one_complement_sum(sum);

    sum += (unsigned long)ip_header->ip_flags_fragment_offset;
    sum = one_complement_sum(sum);

    sum += ((unsigned long)ip_header->ip_time_to_live) << 8;
    sum += (unsigned long)ip_header->ip_protocol;
    sum = one_complement_sum(sum);

    sum += ((unsigned long)ip_header->ip_src_IP[0]) << 8;
    sum += (unsigned long)ip_header->ip_src_IP[1];
    sum = one_complement_sum(sum);

    sum += ((unsigned long)ip_header->ip_src_IP[2]) << 8;
    sum += (unsigned long)ip_header->ip_src_IP[3];
    sum = one_complement_sum(sum);

    sum += ((unsigned long)ip_header->ip_dst_IP[0]) << 8;
    sum += (unsigned long)ip_header->ip_dst_IP[1];
    sum = one_complement_sum(sum);

    sum += ((unsigned long)ip_header->ip_dst_IP[2]) << 8;
    sum += (unsigned long)ip_header->ip_dst_IP[3];
    sum = one_complement_sum(sum);

    return sum;
}
```

・ether_common.c

```
unsigned long one_complement_sum(unsigned long sum)
{
    //計算結果が0xffffより大きい場合は最上位の桁を消去して、あたらめて1を加算する
    //（1の補数和）

    return (sum & 0xffff) + (sum >> 16);
}
```

IP ヘッダに埋め込むチェックサムの計算は、上記の様に行っています。

チェックサムの計算は、2 バイト単位で加算して、1 の補数和を求める形です。

・IP ヘッダ

チェックサム対象のデータを 2 バイト毎に区切る

4500	003C	ABCD	0000	8001	0000	C0A8	0050	C0A8	0051

（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）

・数値を単純に加算していく

4500	+	003C	=	453C

453C	+	ABCD	=	F109

F109	+	0000	=	F109

F109	+	8001	=	1710A

・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(0x1710A-0x10000+0x1)

→one_complement_sum()

710B	+	0000	=	710B

710B	+	C0A8	=	131B3

31B4	+	0050	=	3204

3204	+	C0A8	=	F2AC

F2AC	+	0051	=	F2FD

→ip_header_sum ではここまでを計算

・最後にビットごとの反転を取る

~F2FC	=	0D02

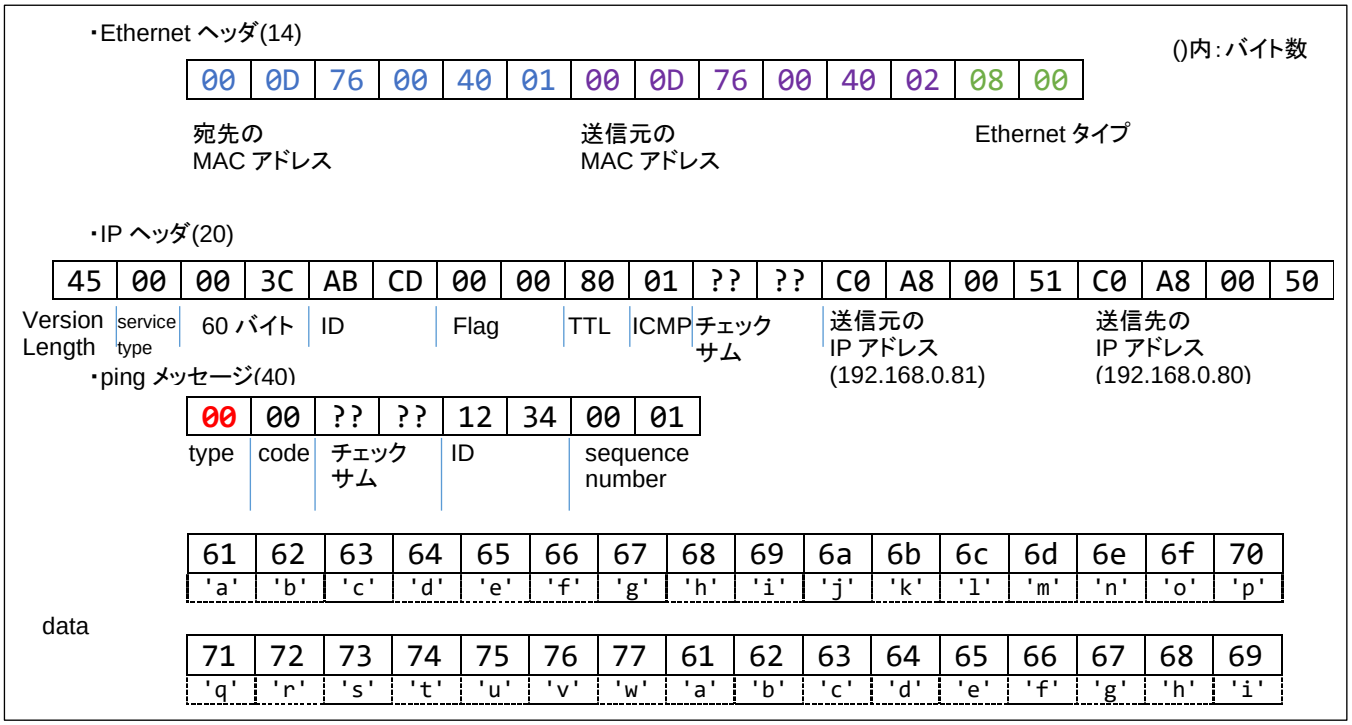
・ping.c 内

```
    //送信IPヘッダのチェックサム計算
    sum = ip_header_sum(&ip_packet);//IPヘッダの1の補数和
    sum = (~sum) & 0xffff;//NOT演算反転
    ip_packet.ip_checksum = (unsigned short)sum;
```

→最終的にはビットごとの反転(NOT)を取った値を、送信データに含めます。

5.7. ping リプライ

マイコンボードに対し、外部から ping リクエストが来た場合の応答が ping リプライです。



宛先のアドレスは、ping リクエストを送ってきた機器のアドレスを入れます。送信元のアドレスは、自分のアドレスを入れます。ping メッセージの、type は ping リプライの場合は変える必要があります。ping メッセージ内の ID, sequence number は、ping リクエストで送ってきたデータを埋め込んだ形で送信しています。

ping リクエストを受け取って、ping リプライを返した場合、以下の様な表示が出ます。

```
ping reply send to 192.168.0.80
```

・ping.c

```
void ping_reply(unsigned char *packet)
{
    unsigned short i;

    unsigned long sum;

    ETHERNET_HEADER ethernet_packet;
    IP_HEADER ip_packet;
    PING_MESSAGE ping_packet;

    ether_return_t ret;

    unsigned char send_data[14+20+40] = {0};

    stream_to_ETHERNET_header(packet, &ethernet_packet);
    stream_to_IP_header(&packet[14], &ip_packet);
    stream_to_PING_message(&packet[14+20], &ping_packet);

    //宛先MACアドレスは送信元の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] = ethernet_packet.eth_src_MAC[i];
        ethernet_packet.eth_src_MAC[i] = g_src_MAC[i];
    }

    //IPヘッダのパケットの識別子(ID)
    ip_packet.ip_id = 0xABCD;

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

    ping_packet.ping_type = 0;//メッセージの種類（pingリプライ）

    //送信IPヘッダのチェックサム計算
    sum = ip_header_sum(&ip_packet);//IPヘッダの1の補数和
    sum = (~sum) & 0xffff;//NOT演算反転
    ip_packet.ip_checksum = (unsigned short)sum;

    //pingメッセージのチェックサム計算
    sum = ping_message_sum(&ping_packet);//pingメッセージの1の補数和
    sum = (~sum) & 0xffff;//NOT演算反転
    ping_packet.ping_checksum = (unsigned short)sum;

    //pingパケット = ETHERNETヘッダ(14) + IPヘッダ(20) + pingメッセージ(40)
    ETHERNET_header_to_stream(&ethernet_packet, send_data);
    IP_header_to_stream(&ip_packet, &send_data[14]);
    PING_message_to_stream(&ping_packet, &send_data[14+20]);
    ret = R_ETHER_Write(g_ether_channel, (void *)send_data, sizeof(send_data));

（後略）
```

ping_replay()関数は、引数として ping リクエストで受信したパケットを受け取り、そのパケット内のデータを利用しながら送信データを生成します。

5.8. メイン関数内のメインループでの受信データの処理

受信データは、割込みでリングバッファに格納。メイン関数内のメインループ内で処理しています。

・main.c, ether_main()内

```
while (1)
{
    R_ETHER_LinkProcess(g_ether_channel);

    //SW3が押された場合は本ボードからpingを送出
    if ((SW3 == 0) && sw_valid())
    {
        ping_request(g_dst_IP);
        sw_disable();//スイッチを一定期間無効化する
    }

    //受信データの処理
    reveive_data_process();
}
```

コードとしては上記で、

・API 関数のステータス取得

・SW3 が押された場合は ping リクエストの送信

・受信データの処理(receieve_data_process())

です。

・main.c

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

    //戻り値
    //  なし

    //引数
    //  なし

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

    unsigned short i;

    ret = ether_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

    switch (ethernet_packet.eth_ethernet_type)//Ethernetタイプで処理を分ける
    {
        case 0x0806://ARPパケットの時
```

receive_data_process では、割り込み関数でデータを受信した際にリングバッファにデータを格納したデータを取り出して処理します。

データを取り出すのは、ether_data_access()関数です。

・ether_common.c

```
int ether_data_access(unsigned char **addr, unsigned short *override_flag)
{
    //リングバッファからデータを読み出す関数

    //戻り値
    //  ETHER_NO_DATA(0) : 未読み出しのデータなし
    //  > 1 : バッファに格納されているデータサイズ

    //引数
    //  **addr : データの先頭アドレスを格納して返す
    //  *override_flag : 上書きフラグを格納して返す（0:上書きなし，1:上書きあり（未読み出しで捨てられたデータあり））

    //補足
    //  本関数を実行してもリングバッファのキューは移動しません
    //  （複数回呼び出してもether_data_dequeue実行前は同じデータを返す）
    //  データ上書きフラグはdeque時にクリアされます

    //引数
    //なし

    unsigned long index;

    if (g_ether_recv_buf_write_index == g_ether_recv_buf_read_index)
    {
        return  ETHER_NO_DATA;
    }

    index = g_ether_recv_buf_read_index & (ETHER_RECV_BUF_SIZE - 1);

    *addr = &g_ether_recv_buf[index][0];
    *override_flag = g_ether_recv_buf_override;

    return (int)g_ether_recv_size[index];
}
```

リングバッファでデータを取り扱う際、

(1)データを保存　ether_data_enqueue()　受信割り込み関数内で実行

(2)データにアクセス　ether_data_access()

(3)キューを進める　ether_data_dequeue()

の 3 つの関数を使用しています。

リングバッファは、FIFO(FirstInFirstOut)のバッファとして働きます。リングバッファ（キュー）に保存されたデータは、ether_data_access()でアクセスできます。ether_data_access()は、リングバッファに保存されているデータの内、一番古いデータの先頭アドレスを返します。この関数は、データのコピーは行いません。ether_data_access()を複数回呼び出しても同じデータを返します。

1 つの受信データの処理が終わったら、ether_data_dequeue()でキューを進める（一番古いデータを読み出し済みにする）処理を行います。

・main.c, receice_data_process()（続き）

```
    switch (ethernet_packet.eth_ethernet_type)//Ethernetタイプで処理を分ける
    {
        case 0x0806://ARPパケットの時
        {
            ARP_MESSAGE arp_packet;

            addr2 = addr;
            addr2 += 14;//ARPパケットのオフセット（14バイト目以降がARPパケット）

            stream_to_ARP_message(addr2, &arp_packet);//ARPのメッセージを分解

            if (IP_compare(arp_packet.arp_dst_IP, g_src_IP) == 0)//ARPパケット内の宛先IPアドレスが本ボ
ードのIPアドレスに一致
            {
                switch (arp_packet.arp_operation)
                {
                    case 2://ARPリプライの時（こちらが送った問い合わせに相手が反応を返した）

                        for (i=0; i<6; i++)
                        {
                            g_dst_MAC[i] = arp_packet.arp_src_MAC[i];//パケットに記載されている送信元MAC
アドレスを変数に格納する
                        }

                        sci_write_str("ARP reply received from ");
                        sci_print_IP(arp_packet.arp_src_IP);
                        sci_write_str(", MAC address = ");
                        sci_print_MAC(g_dst_MAC);
                        sci_write_str("¥n");
                        break;

                    case 1://ARPリクエストの時

                        ARP_reply(addr);//ARPリプライ（本ボードのMACアドレスを相手に通知する）
                        break;
                }
            }
        }
        break;
```

・Ethernet ヘッダ(14)

00	0D	76	00	40	01	00	0D	76	00	40	02	08	06

宛先の MAC アドレス　　送信元の MAC アドレス　　Ethernet タイプ

Ethernet フレーム内の、Ethernet タイプの値で処理を分けています。この値が、0x0806 の時は ARP リクエストか ARP リプライです。

・ARP メッセージ(28)

00	01	08	00	06	04	00	02

hardware type　protocol type　hardware length　protocol length　operation

ARP の時は、operation の値を見て、ARP リクエストの時は、マイコンボードの MAC アドレスを APR リプライで返します。ARP リプライの場合は MAC アドレスを変数(g_dst_MAC)に保存します。

・main.c, receice_data_process()（続き）

```
        case 0x0800: //IPパケットの時
        {
            IP_HEADER ip_packet;

            addr2 = addr;
            addr2 += 14;//ARPパケットのオフセット（14バイト目以降がIPパケット）

            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)
                {
                    case 1://ICMPパケットの時
                    {
                        PING_MESSAGE ping_packet;

                        addr3 = addr;
                        addr3 += (14 + 20);//pingメッセージパケットのオフセット（ARPパケットが14バイト，
IPヘッダが20バイト）

                        stream_to_PING_message(addr3, &ping_packet);

                        switch (ping_packet.ping_type)
                        {
                            case 0://pingリプライ時（こちらが送ったpingに対する応答）
                                sci_write_str("ping reply received from ");
                                sci_print_IP(ip_packet.ip_src_IP);
                                sci_write_str("¥n");
                                break;

                            case 8://pingリクエストの時（誰かが本ボードのIPアドレスに対してpingを打ってき
たので応答する）
                                ping_reply(addr);//pinのリプライを返す
                                break;
                        }
                    }
                    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_data_dequeue();
}
```

Ethernet タイプが、0x0800 の時は、ping の処理を行っています。

・ping メッセージ(40)

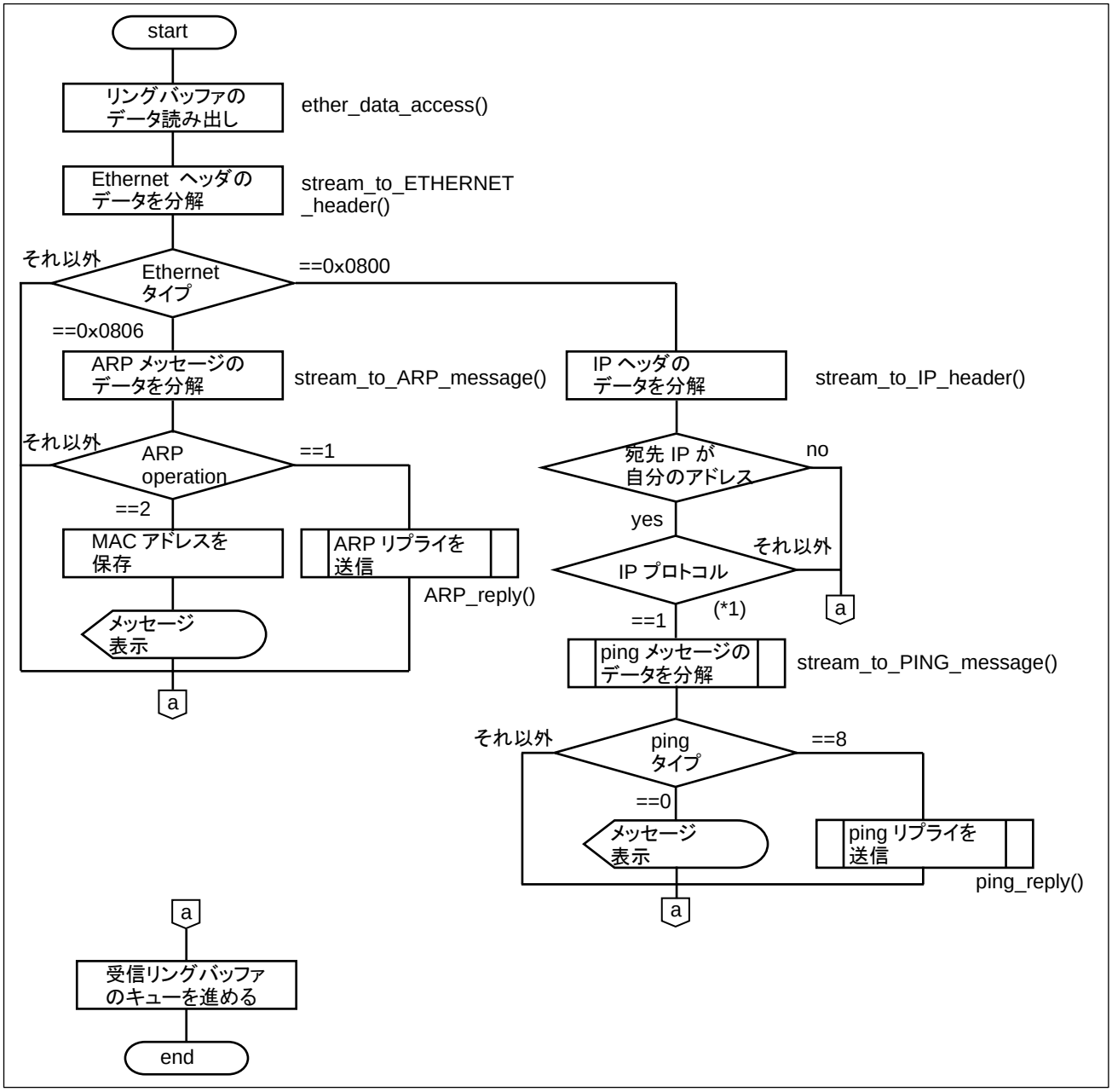
00	00	??	??	12	34	00	01
type	code	チェックサム		ID		sequence number	

ping メッセージ内の type により、ping リプライか ping リクエストかを判断して処理を分けています。

receice_data_process()の最後に、ether_data_dequeue()でキューを進める（データを読み出し済みとして、次のデータを処理する）様にしています。

・Ethernet ヘッダを見て Ethernet タイプにより ARP のデータか IP のデータかを判断
・ARP のデータの場合は ARP リクエストか ARP リプライかを判断
・IP のデータの場合は ICMP(ping)かどうかを判断
・ping の場合は ping リクエストか ping リプライかを判断
といった流れで、Ethernet ヘッダや IP ヘッダ、ARP データ、ping データの特定のフィールドの値を見て処理を行っていく形です。

・receice_data_process() （main.c 内）



(*1)本プログラムでは、IP プロトコルが 1 の時に ping データとして処理していますが、今後別なプロトコル（UDP や TCP）を取り扱う際は、この部分での分岐（case 文）を増やせば良いという形になります。

5.9. ユーザ作成関数

ARP_request

概要:ARP リクエスト
宣言:void ARP_request(unsigned char *dst_IP)
説明:
・ARP リクエスト（IP アドレスに対応する MAC アドレスを調べる要求を送出）
を行います
引数:
　unsigned char *dst_IP: 問い合わせを行う IP アドレス
戻り値:なし
補足:IP アドレスは
　　unsigned char dst_IP[4] = {192, 168, 0, 81};　//IP address = 192.168.0.81
　　の様に、unsigned char の 4 バイトのデータを与えます

ARP_reply

概要:ARP リプライ
宣言:void ARP_reply(unsigned char *packet)
説明:
・ARP リプライ（本ボードに対しての ARP リクエストに応答）
を行います
引数:
　unsigned char *packet: ARP リクエストのパケット（データストリーム）
戻り値:なし
補足:基本的には ARP パケット内の IP アドレスが本ボードの IP アドレスに一致した場合に、本関数が呼ばれる事を
　　想定しています
　　（ARP パケット内の IP アドレスが本ボードの IP アドレスに一致しているかどうかは本関数内では確認して
　　いません）

ping_request

概要:ping リクエスト
宣言:void ping_request(unsigned char *dst_IP)
説明:
・引数で指定された IP アドレスに対して ping を打つ動作
を行います

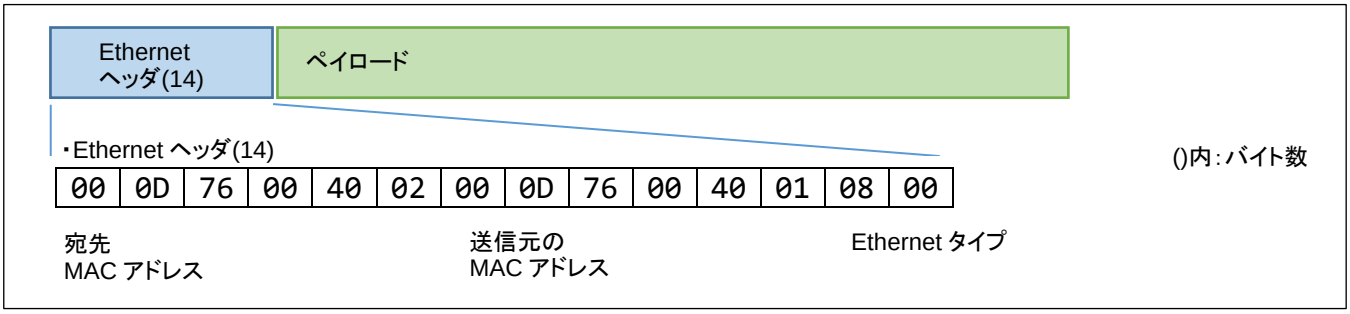
引数:
　unsigned char *dst_IP: 問い合わせを行う IP アドレス
戻り値:なし
補足:ping で送信するデータは 32 バイトです

ping_reply

概要:ping リプライ
宣言:ping_reply(unsigned char *packet)
説明:
・本ボードに対しての ping に対して応答
を行います
引数:
　unsigned char *packet: ping リクエストのパケット（データストリーム）
戻り値:なし
補足:基本的には ping パケット内の IP アドレスが本ボードの IP アドレスに一致した場合に、本関数が呼ばれる事を想定しています
（ping ケット内の IP アドレスが本ボードの IP アドレスに一致しているかどうかは本関数内では確認していません）

6. 構造体定義

6.1. Ethernet ヘッダ



・ether_common.h

```
typedef struct
{
    //イーサネットヘッダ(14bytes)
    unsigned char eth_dst_MAC[6];
    unsigned char eth_src_MAC[6];
    unsigned short eth_ethernet_type;
}ETHERNET_HEADER;
```

Ethernet ヘッダは、14 バイトで

宛先 MAC アドレス　eth_dst_MAC[6] (unsigned char)

送信元 MAC アドレス　eth_src_MAC[6] (unsigned char)

Ethernet タイプ　eth_ethernet_type (unsigned short)

で定義しています。

受信したデータから Ethernet ヘッダの特定のデータを取り出す場合は、

```
ETHERNET_HEADER ethernet_packet;
stream_to_ETHERNET_header(addr, &ethernet_packet);
```

の様に行えます。addr は、受信データの先頭アドレス。

ethernet_packet.eth_ethernet_type

で、受信データ内の Ethernet タイプのデータを取り出せます。

送信データを作る際には、

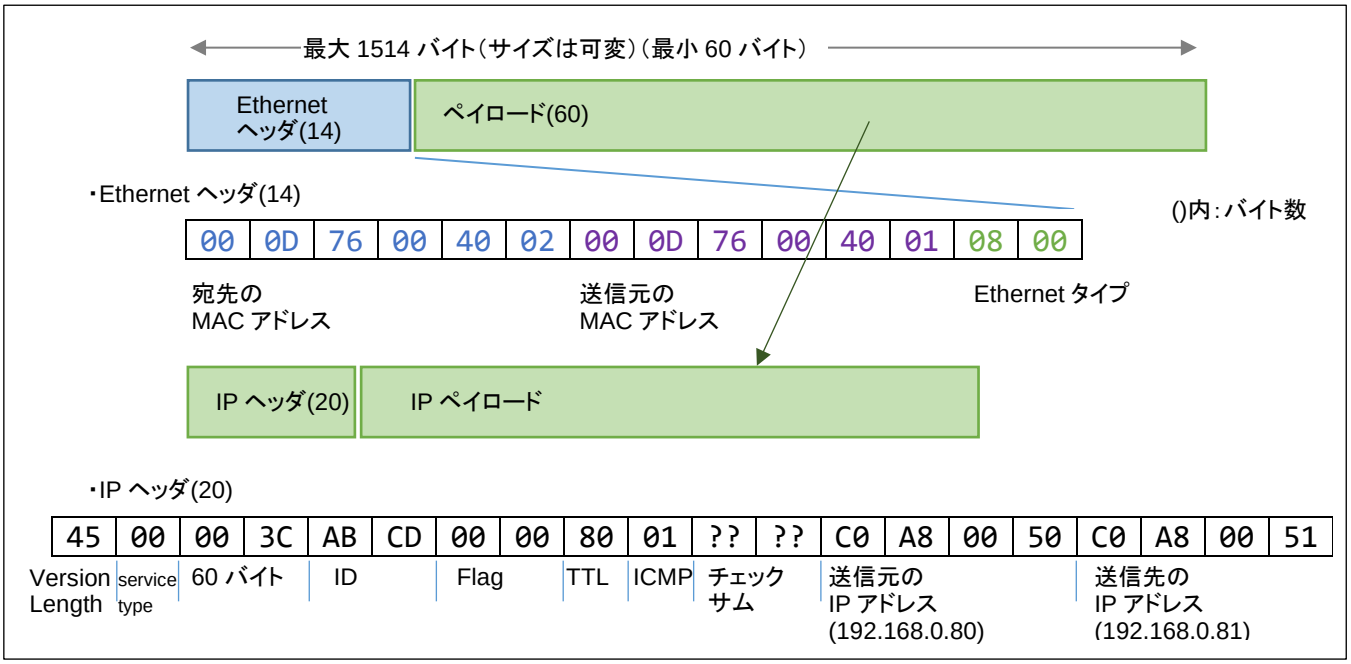
```
ETHERNET_HEADER ethernet_packet;
ethernet_packet.eth_dst_MAC[0] = 0x00;
ethernet_packet.eth_dst_MAC[1] = 0x0D;
ethernet_packet.eth_dst_MAC[2] = 0x76;
ethernet_packet.eth_dst_MAC[3] = 0x00;
ethernet_packet.eth_dst_MAC[4] = 0x40;
ethernet_packet.eth_dst_MAC[5] = 0x02;
ethernet_packet.eth_src_MAC[0] = 0x00;
 （中略）
ethernet_packet.eth_ethernet_type = 0x0800;
```

```
unsigned char send_data[60];

ETHERNET_header_to_stream(&ethernet_packet, send_data);
```

の様に、構造体の各メンバの値を設定した後で、ストリームデータに変換する形です。

Ethenet ヘッダの何バイト目が何のデータかを覚えておいて処理するよりかは、構造体のメンバとして処理する方が楽になるかと思います。

6.2. IP ヘッダ



・ether_common.h

```
typedef struct
{
    //IPヘッダ(20bytes)
    unsigned char ip_version_length;
    unsigned char ip_service_type;
    unsigned short ip_total_length;
    unsigned short ip_id;
    unsigned short ip_flags_fragment_offset;
    unsigned char ip_time_to_live;
    unsigned char ip_protocol;
    unsigned short ip_checksum;
    unsigned char ip_src_IP[4];
    unsigned char ip_dst_IP[4];
}IP_HEADER;
```

IP ヘッダは、20 バイトで

IP バージョンとヘッダ長　ip_version_length (unsigned char)

サービスタイプ　ip_service_type (unsigned char)

データ長　ip_total_length (unsigned short)

ID　ip_id (unsigned short)

フラグ/分割オフセット　ip_flags_fragment_offset (unsigned short)

TTL　ip_time_to_live (unsigned char)

プロトコル　ip_protocol (unsigned char)

チェックサム　ip_checksum (unsigned short)

送信元アドレス　ip_src_IP[4] (unsigned char)

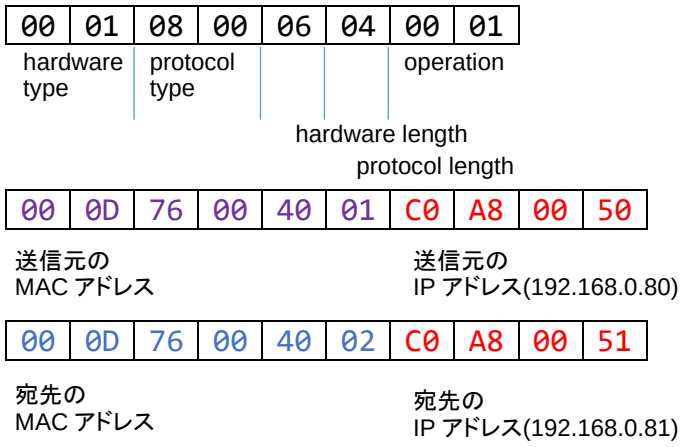
送信先アドレス　ip_dst_ip[4] (unsigned char)

で定義しています。

IP ヘッダは、オプションを含めて、20 バイトより大きくする事もありますが、本キットでは 20 バイトのデータとして取り扱っています。

6.3. ARP メッセージ

・ARP メッセージ(28)



・ether_common.h

```
typedef struct
{
    //ARPメッセージ(28bytes)
    unsigned short arp_hardware_type;
    unsigned short arp_protocol_type;
    unsigned char arp_hardware_length;
    unsigned char arp_protocol_length;
    unsigned short arp_operation;
    unsigned char arp_src_MAC[6];
    unsigned char arp_src_IP[4];
    unsigned char arp_dst_MAC[6];
    unsigned char arp_dst_IP[4];
}ARP_MESSAGE;
```

ARP メッセージは上記構造体で取り扱っています。

6.4. ping メッセージ

・ping メッセージ(40)

08	00	??	??	12	34	00	01
type	code	チェックサム		ID		sequence number	

data

61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f	70
'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'	'p'

71	72	73	74	75	76	77	61	62	63	64	65	66	67	68	69
'q'	'r'	's'	't'	'u'	'v'	'w'	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'

・ether_common.h

```
typedef struct
{
    //pingメッセージ(40bytes)
    unsigned char ping_type;
    unsigned char ping_code;
    unsigned short ping_checksum;
    unsigned short ping_id;
    unsigned short ping_sequence_number;
    unsigned char ping_data[32];
}PING_MESSAGE;
```

ping メッセージは上記構造体で取り扱っています。

ping データは 32 バイトに決まっている訳ではありませんが、本キットでは ping は 32 バイトデータとしています。

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 スターターキット RX65N
ソフトウェア編　取扱説明書(1)

株式会社北斗電子
 2026 年 6 月 8 日改訂 REV.1.0.0.0 (260608)