



Ethernet スタータキット RA

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 RA 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

目次

注意事項	1
安全上のご注意	2
1. 本キットのサンプルプログラムに関して	4
1.1. RA マイコン向けサンプルプログラム	4
1.2. PC 向けサンプルプログラム	4
2. RAxxx_MAGIC_PACKET プロジェクト	6
2.1. プロジェクトツリー	11
2.2. FSP の設定	13
2.2.1. クロック設定	13
2.2.2. スタックの追加	17
2.2.3. スタックの設定	18
2.2.4. 端子設定	27
2.2.5. コード生成	28
2.3. ユーザ側で作成したプログラムコード	29
2.4. ユーザ作成関数	41
2.5. メイン関数での処理	42
3. RAxxx_MAGIC_PACKET2 プロジェクト	43
3.1. Ether の初期化	46
3.2. Ethernet フレームの送信	47
3.3. Ethernet フレームの受信	48
3.4. MagicPacket の送信	53
3.5. MagicPacket の受信	55
3.5.1. マイコンの MagicPacket 受信機能を使用	55
3.5.2. 受信データをユーザプログラムで処理	57
3.6. メイン関数での処理	62
3.7. 割り込み関数での処理	64
3.8. ユーザ作成関数	65
4. RAxxx_ETHER_NOAPI プロジェクト	66
4.1. メイン関数での処理	69
4.2. Ether の初期化	71
4.3. ディスクリプタの使用	73
4.4. Ethernet フレームの送信	80
4.5. Ethernet フレームの受信	81
4.6. 割り込み関数での処理	82
4.7. 補足	87
4.8. ユーザ作成関数	89
4.9. memcpy ライブラリ関数に関して[参考]	93
4.9.1. Cortex-M4(RA6M2)の場合	93

4.9.2. Cortex-M33(RA6M4)の場合	96
4.9.3. Cortex-M85(RA8M1)の場合	97
4.9.4. RX(RXv2)の場合[参考]	99
4.9.5. ライブラリ関数を使わない場合	100
5. RAxxx_PING プロジェクト	104
5.1. メイン関数での処理	106
5.2. 割り込み関数での処理	108
5.3. ping とは	111
5.4. ARP リクエスト	111
5.5. ARP リプライ	115
5.6. ping リクエスト	117
5.7. ping リプライ	123
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	125
5.9. ユーザ作成関数	130
6. 構造体定義	132
6.1. Ethernet ヘッダ	132
6.2. IP ヘッダ	133
6.3. ARP メッセージ	134
6.4. ping メッセージ	135
取扱説明書改定記録	136
お問合せ窓口	136

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可能性がある事が想定される

絵記号の意味

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

警告



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

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・異臭にお気づ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、長時間直射日光が当たる場所、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. RA 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、e2studio のワークスペースを zip ファイルにエクスポートしたものです。
e2studio のワークスペースにインポートを行って下さい。

e2studio(2026-07)以降

FSP6.6.0 以降

が必要です。

FSP は、RA 向けの FSP を e2studio にセットアップするか、RA 向けの FSP と e2studio がセットになっているもの (FSP Windows Platform Installer)

をインストールしてください。(後者の方が簡単です。ルネサスエレクトロニクス製の開発環境に慣れていない方は、FSP+e2studio のインストーラからのインストールを推奨します。その場合は、ARM 向けの toolchain (コンパイラ (LLVM) など) も一緒にインストールされます。)

(通常版の e2studio に RA 向けの FSP をセットアップする場合は、ARM 向けの toolchain のインストールも必要です。)

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 VisualStudio 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. Rxxxx_MAGIC_PACKET プロジェクト

RA マイコン向けのプロジェクトは、CD 内の

SOURCE¥Rxxxx_yyy¥Rxxxx_MAGIC_PACKET.zip

に格納されています。

使用しているマイコンボードが、HSBRA6M2F144(144 ピンのマイコンが搭載されたボード)の場合は、SOURCE¥RA6M2_144¥RA6M2_MAGIC_PACKET.zip のアーカイブファイルを e2studio のワークスペースにインポートしてください。

xxx: RA のリリース名 (RA6M2, RA8E1 など)

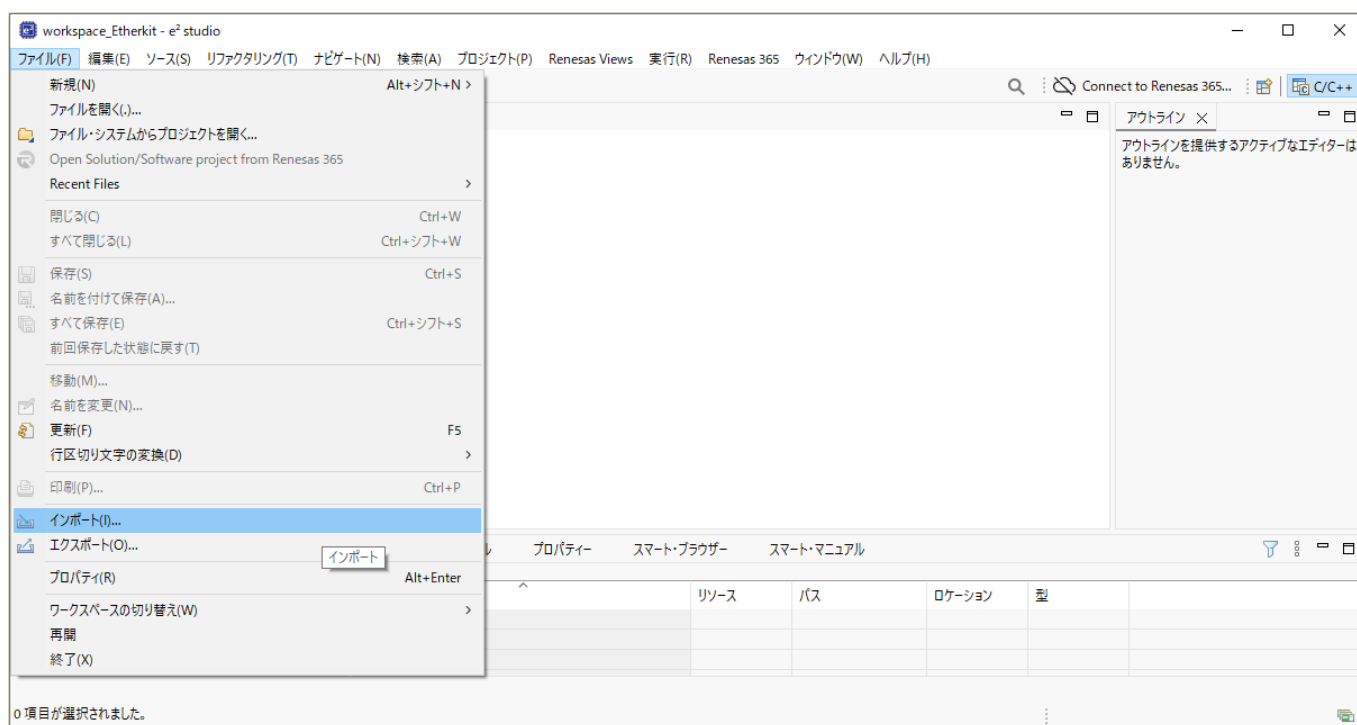
yyy: マイコンのピン数 (176, 144 など)

(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。)

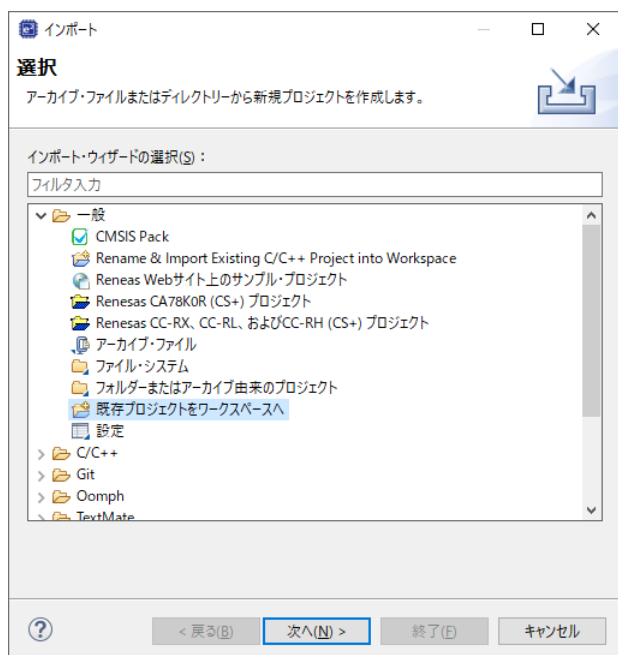
マイコンボード型番	フォルダ名(Rxxxx_yyy)	プロジェクト名
HSBRA6M2F144	RA6M2_144	RA6M2_MAGIC_PACKET
HSBRA6M3F176	RA6M3_176	RA6M3_MAGIC_PACKET
HSBRA6M4F144	RA6M4_144	RA6M4_MAGIC_PACKET
HSBRA6M5F176	RA6M5_176	RA6M5_MAGIC_PACKET
HSBRA8E1F144	RA8E1_144	RA8E1_MAGIC_PACKET
HSBRA8M1F144	RA8M1_144	RA8M1_MAGIC_PACKET
HSBRA8M1F176	RA8M1_176	RA8M1_MAGIC_PACKET
HSBRA8T1F176	RA8T1_176	RA8T1_MAGIC_PACKET

※インポートするファイル名はプロジェクト名+.zip です

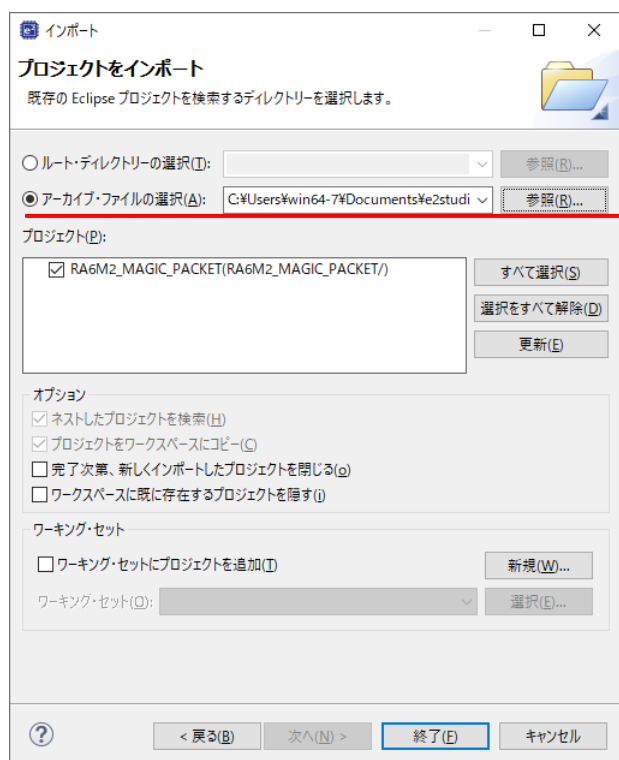
プロジェクトのインポートに関して



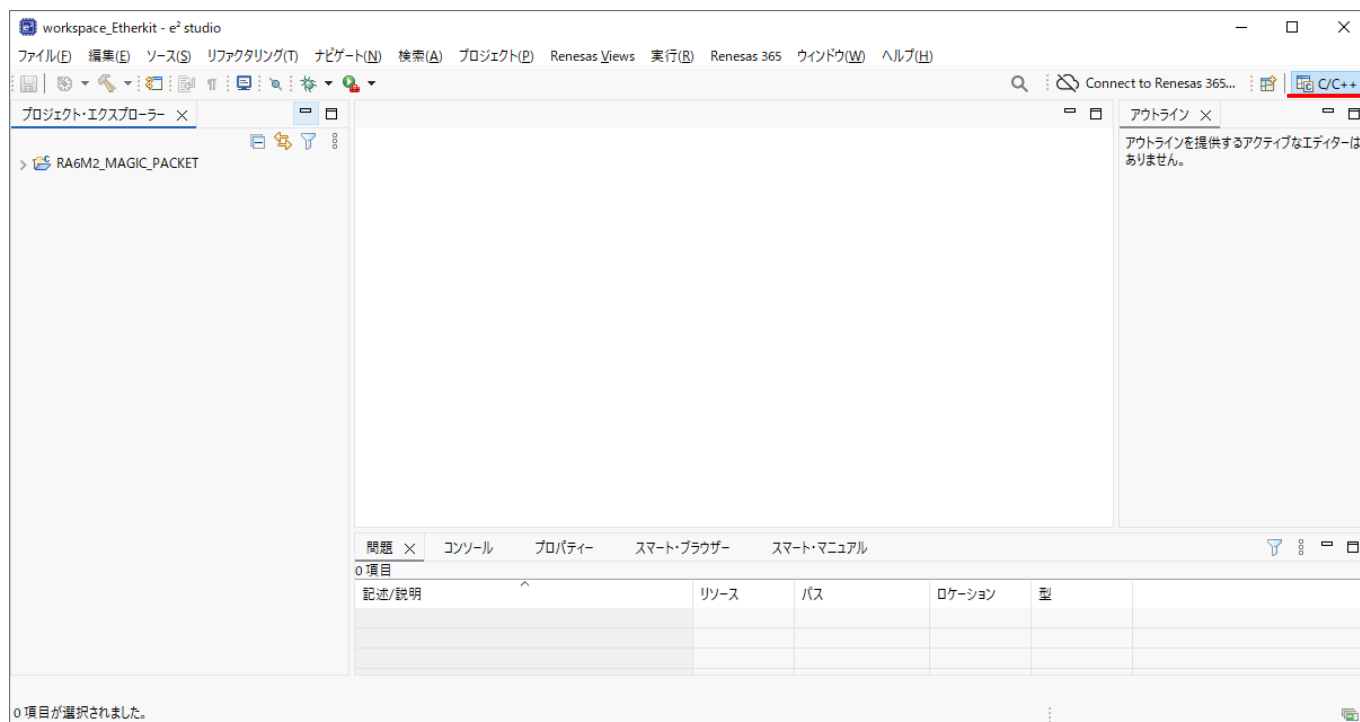
ファイルインポート



既存プロジェクトをワークスペースへ
「次へ」



アーカイブファイルの選択で、CD 内の
SOURCE¥[マイコンボード毎のフォルダ]¥RAXxx_MAGIC_PACKET.zip
(インポートしたいプロジェクトの zip ファイルを選択)
を選択して「終了」を押します。



プロジェクトエクスプローラー上でインポートしたプロジェクトが見えていればインポートは成功しています。(プロジェクトエクスプローラーが見えていない場合は、右上で C/C++ ペインを選択)

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.
RA6M2 Ethernet 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:
  PUSH-SW : 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。

起動時にプッシュスイッチが押されていると、送信先と本ボードの MAC アドレスが逆に設定されます。

プッシュスイッチ

→Ethernet スタータキット RX/RA 取扱説明書(1)

(ETHERNET_STARTER_KIT_RXRA_1_REV_x_x_x_x_s.pdf)

の

4.3. MagicPacket(WOL)の送受信

の項に、ボード毎のプッシュスイッチに関して記載されています。RA のボードでは、ボード上にプッシュスイッチが 2 つあり、片方がリセット、もう一方が MagicPacket 送信などに使用するプッシュスイッチです。

ここでは、起動時に MagicPacket 送信などに使用するプッシュスイッチを押した状態とした場合(プッシュスイッチを押したままで電源投入、またはプッシュスイッチを押したままで、リセットスイッチを押して離す)の挙動に関して説明します。

プッシュスイッチを押したまま、ボードを起動すると、

```
board -> alternate setting.
PUSH-SW press detected. Release OK.
```

端末には上記表示が出ます。この表示が出た場合は、ボードは 2 台目の設定が適用されます。同一ネットワーク上に 2 台のボードを接続する場合、片方のボードを 2 台目の設定としてください。

	通常起動	プッシュスイッチを押したまま起動 (2 台目の設定)
MAC アドレス	00-0D-76-00-40-01	00-0D-76-00-40-02
MagicPacket 送信先	00-0D-76-00-40-02	00-0D-76-00-40-01

2 台目の設定とした場合、ボード自身の MAC アドレスと MagicPacket 送信先の MAC アドレスが逆に設定されます。(なお、起動後に S(Swap)コマンドを入力すると、プッシュスイッチを押したままボードを起動した場合と同じ設定になります。)

ボード起動後に、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-SW -> send MagicPacket
Ether0: LINK-UP
```

→送信モード、ボードのプッシュスイッチ(リセットスイッチとは別なプッシュスイッチ)を押すと MagicPacket 送信

r コマンドで動作開始

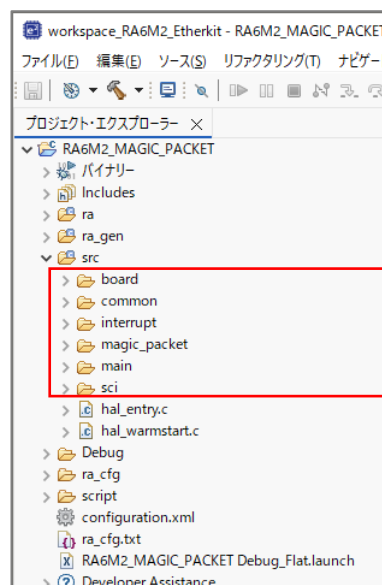
```
>command=r

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。



ユーザ側で追加

ra

ra_gen

ra_cfg

は、FSP により生成されるフォルダです。これらのフォルダ以下のファイルは、FSP が生成、変更しますので、基本的にはユーザ側で書き換える事はありません。

src

ソースファイルを格納するフォルダです。

このフォルダに、ソース(yyy.c, yyy.h)をコピーしておくと、ビルド対象になります。

hal_entry.c

→ユーザ作成プログラムのエントリポイントです。ユーザ側で作成したメイン処理(main_ether())関数を呼び出すだけの内容としています

hal_warmstart.c

→本キットでは、このファイルを書き換える事はしていません

Debug

は、マイコンボードに書き込むバイナリ(srec ファイル)が出力されます。

script

は、リンカスクリプトが格納されるフォルダです。ユーザ側でリンカスクリプトを追加する場合は、この下にファイルを置いておきます。(4 章の RAXxx_ETHER_NOAPI プロジェクトでは、ユーザ作成のリンカスクリプトを使用しています)

Configuration.xml

は、FSP の設定を行うファイルです。FSP では、マイコンのクロックの設定や、周辺機能(ether や UART、タイマなど)や端子設定、割り込みの設定など、マイコンを動かす上で必要な各種設定を行います。)

プロジェクトを作成した段階では、上記の赤枠の部分はありません。赤枠部は、こちらで追加したプログラムソースとなります。

以下、src 内の赤枠部のフォルダ(ユーザ側で追加したプログラムソース)の説明を行います。

board 以下

→ボード固有設定ヘッダ

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_ether.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

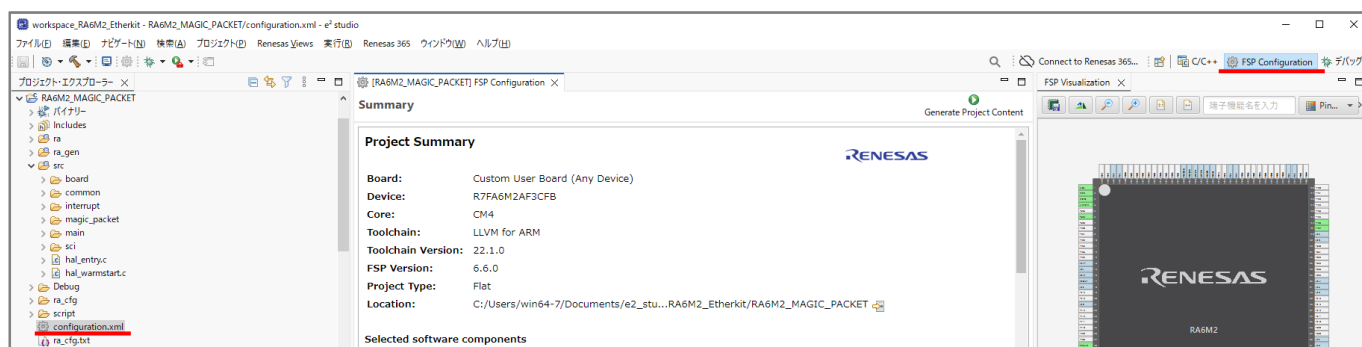
sci 以下

→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

board, common, interrupt, magic_packet, main. sci 以下に含まれるのが、ユーザ側で作成したコード。ra, ra_gen, ra_cfg 以下は、FSP 上で 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. FSP の設定

FSP(FlexibleSoftwarePackage)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。

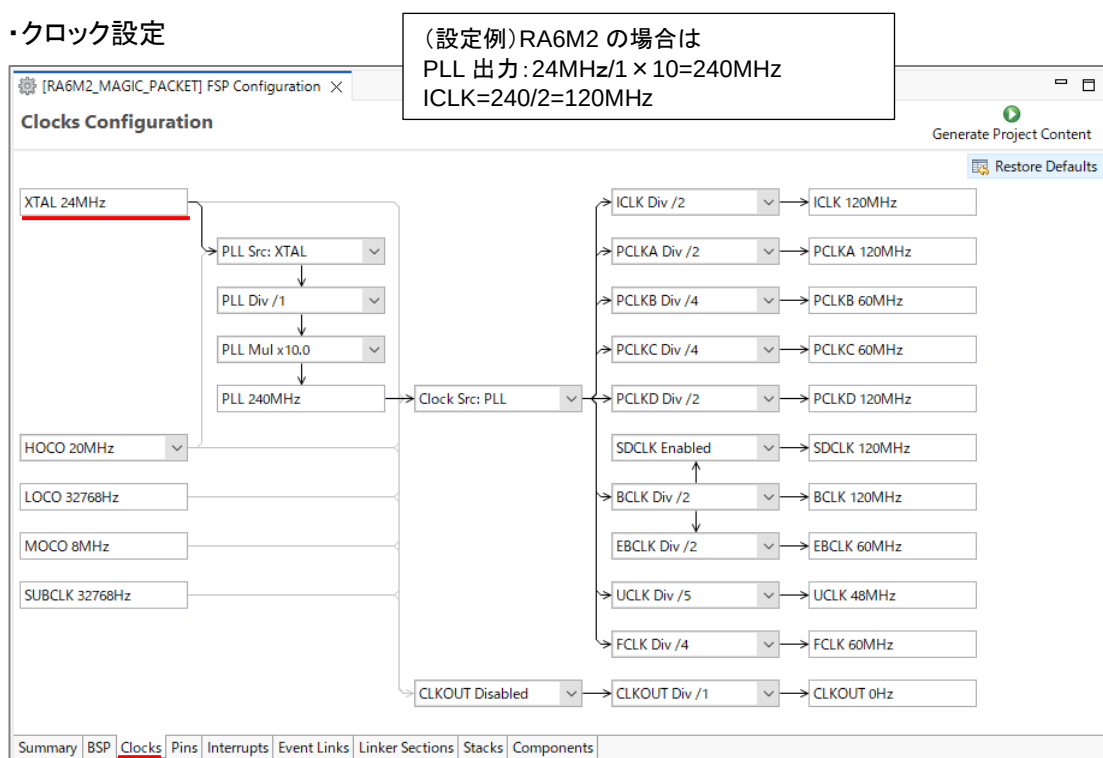


Configuration.xml をダブルクリックで開く。

右上のパーспекティブの選択で「FSP Configuration」を選択。

2.2.1. クロック設定

・クロック設定

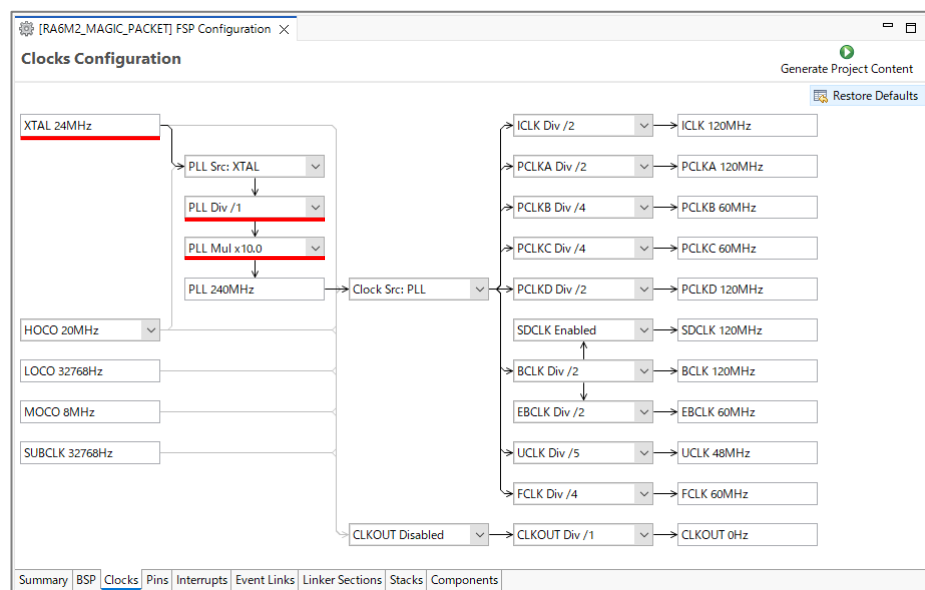


Clocks タブを選択して、クロックの設定を行います。本キットに対応しているボードは全て、24MHz の XTAL を搭載しているので、XTAL は 24,000,000(24M)を入力してください。

PLL を XTAL 入力として、基本的にはマイコンの最大動作周波数となる様に設定してください。

Ether の機能を使う上で、EDMAC は PCLKA で動作しますので、CPU クロックと PCLKA ができるだけ高い周波数となる様に設定してください。

—RA6M2 の場合の設定例—



※赤線部がデフォルト値から変更必要な箇所です

XTAL : 24,000,000(24M)

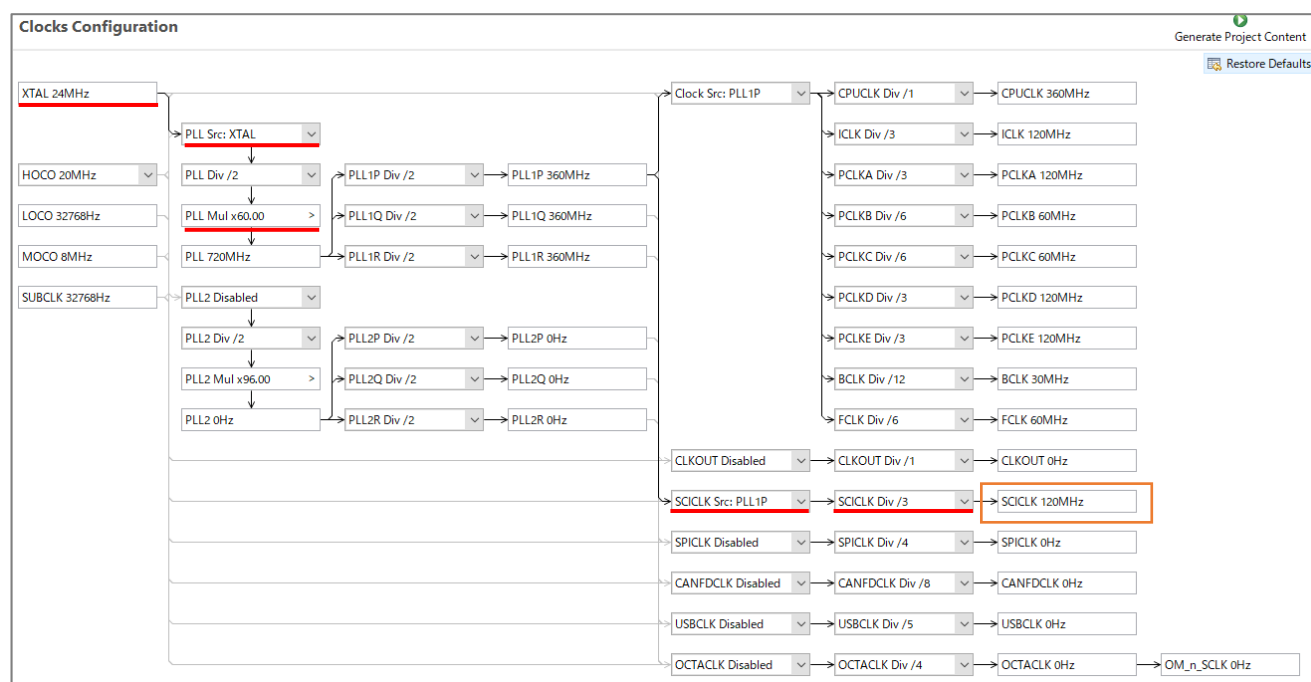
PLL Div : /1

PLL Mul : x10

—RA6M3, RA6M4, RA6M5 の場合の設定例—

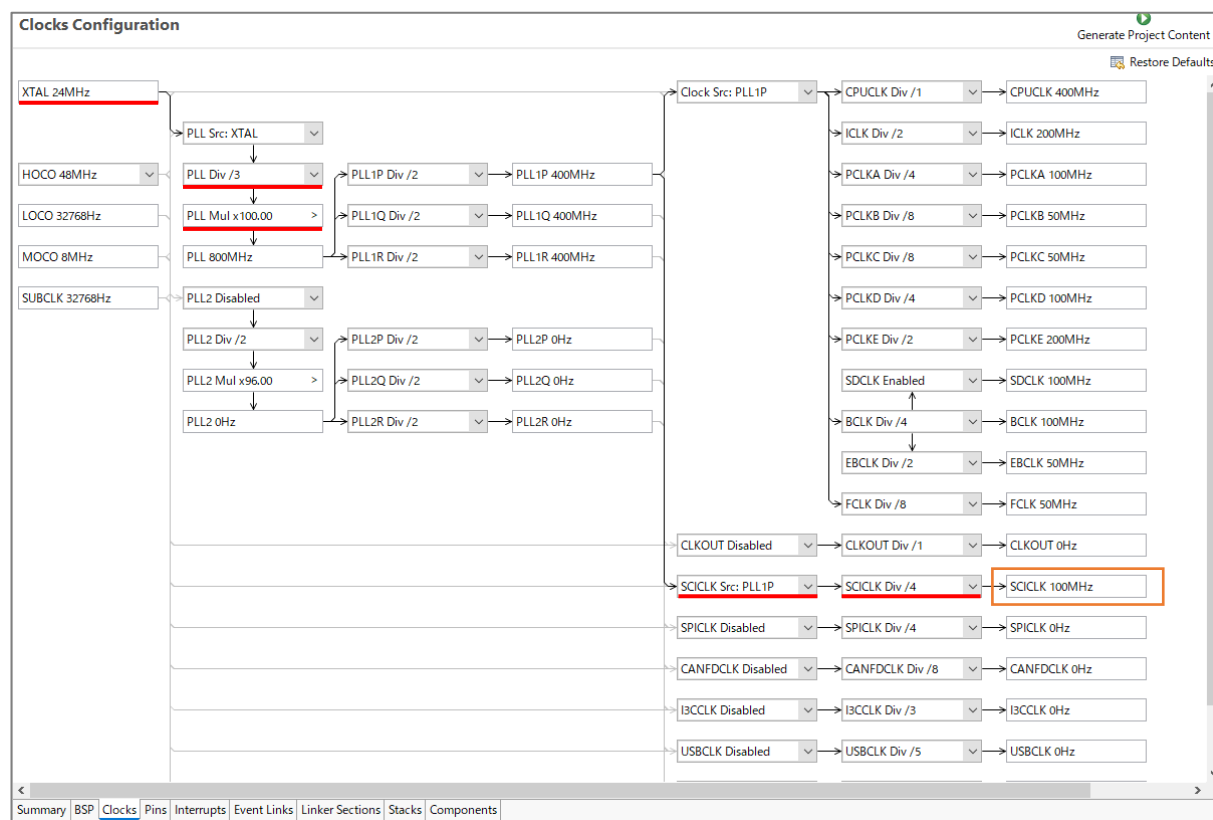
デフォルト値から変更不要です。

—RA8E1 の場合の設定例—

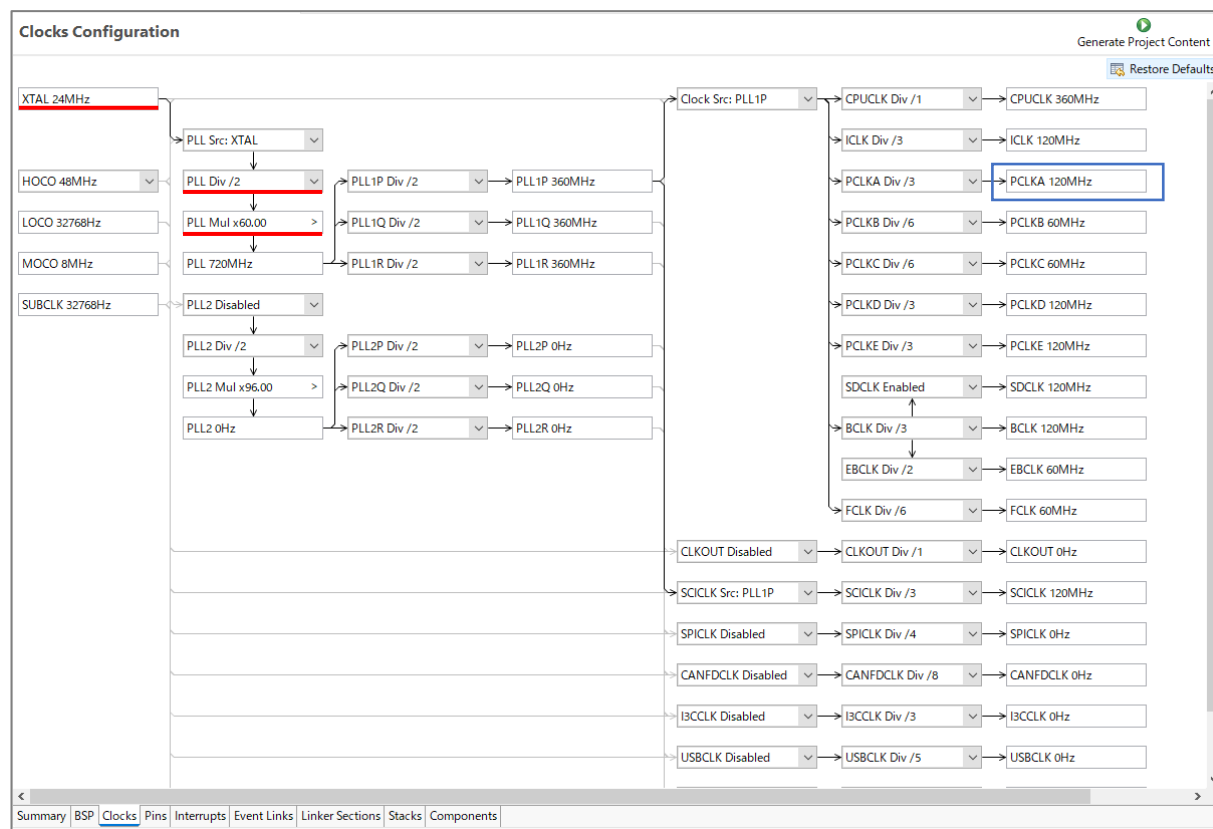


プログラムで SCI の機能を使用しているので SCICLK を供給する様にしてください。(周波数は任意)

—RA8M1/RA8T1 の場合の設定例(1)—



—RA8M1/RA8T1 の場合の設定例(2) PCLKA 最大—



Ether の処理では、CPU での処理、EDMAC での処理、メモリの読み書きの速度が重要になります。

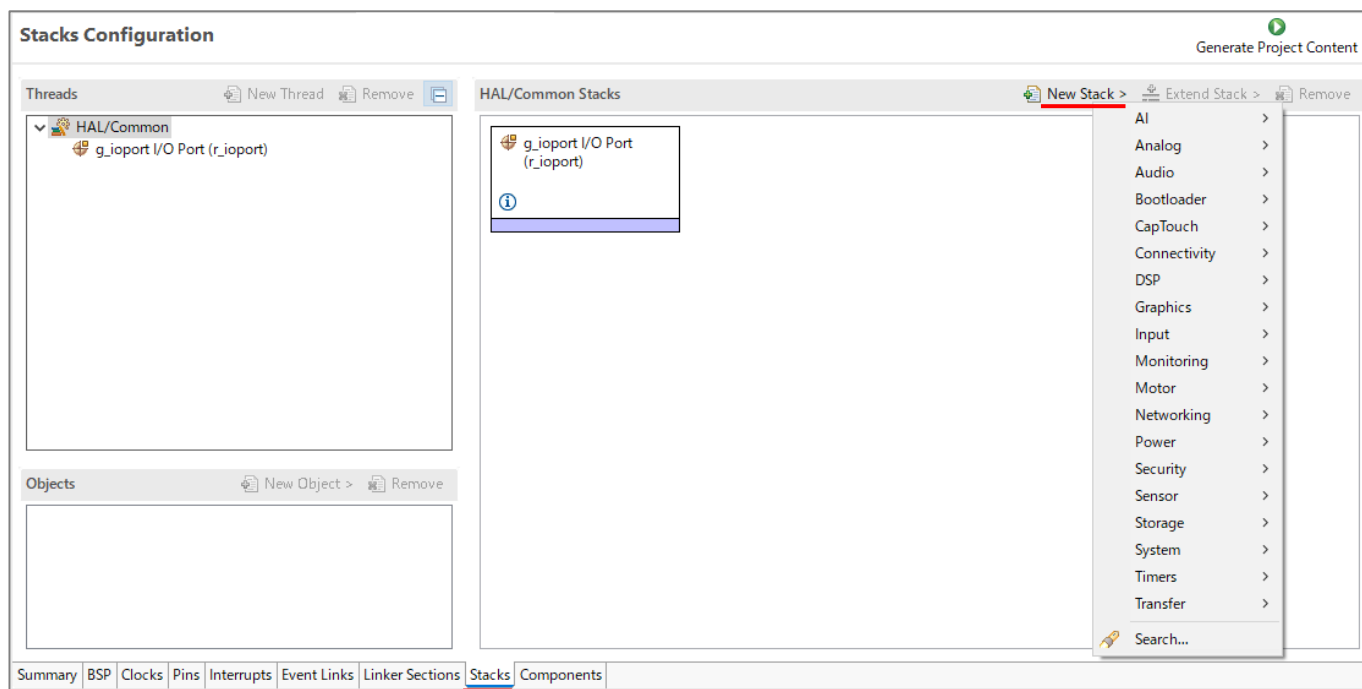
	CPU	EDMAC (PCLKA)	SRAM (ICLK)
設定例(1)	400MHz	100MHz	200MHz, 1 ウェイト (10ns)
設定例(2)	360MHz	120MHz	120MHz (8.3ns)

動作を比較すると、(1)は CPU クロックは高速ですが、EDMAC 動作と SRAM アクセスにおいては(2)の方が高速です。

SRAM を読み書きする簡単なプログラムで速度を比較すると、(2)は(1)に対し SRAM 自体のアクセス速度は 20% 遅いはずですが、全体の処理時間としては(1)の方が高速に動作しました。

本プロジェクトでは、設定例(1)の CPU を最大動作周波数で動かす設定とします。

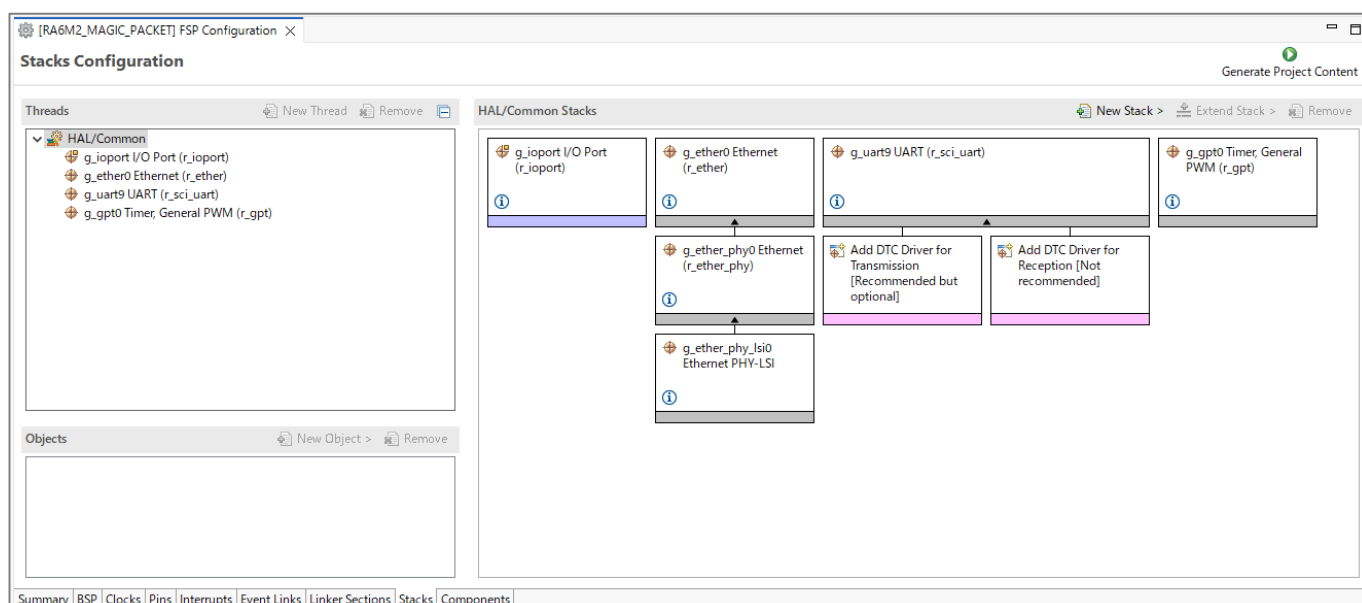
2.2.2. スタックの追加



Stacks タブ、New Stack から使用する周辺機能を追加します。RAxxx_MAGIC_PACKET プロジェクトでは、
Networking->Ethernet(r_ether)
Connectivity->UART(r_sci_uart) (*1)
Timers->Timer, General PWM(r_gpt)

上記、3 種類の Stack を追加します。

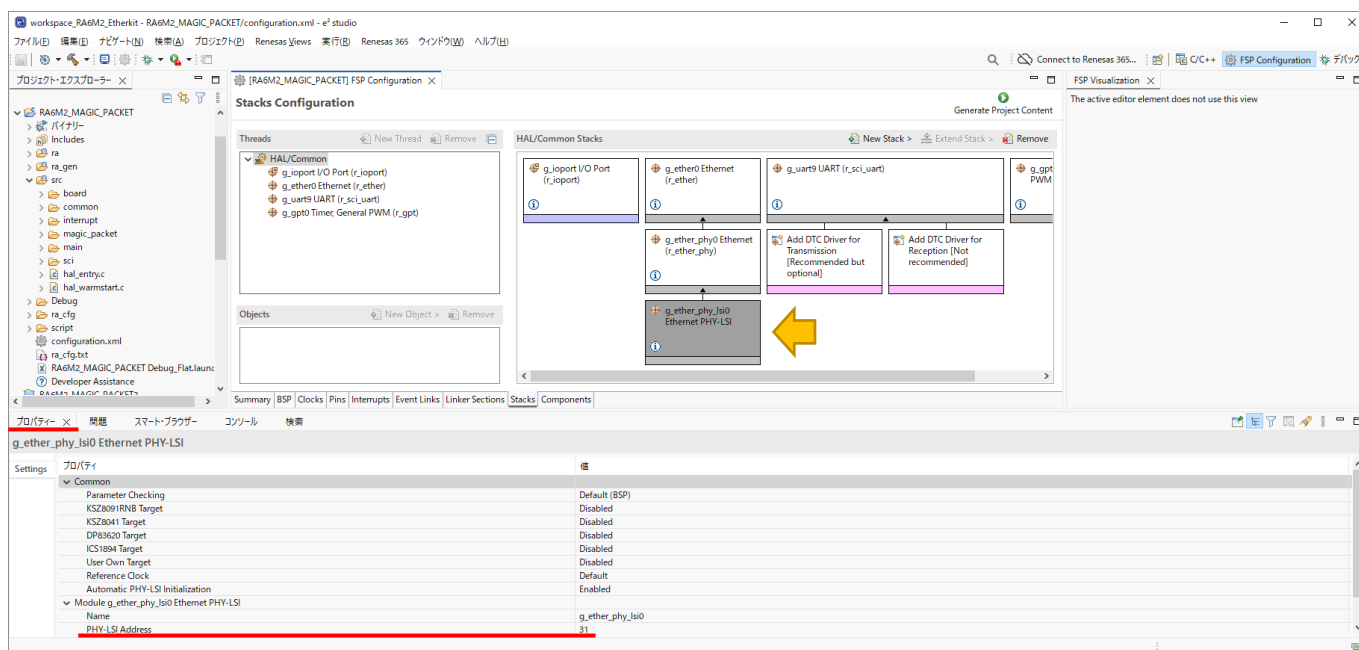
(*1)マイコン種が RA8E1, RA8M1, RA8T1 の時は r_sci_b_uart



追加後は、上記のようになります。

2.2.3. スタックの設定

・Ethernet PHY-LSI



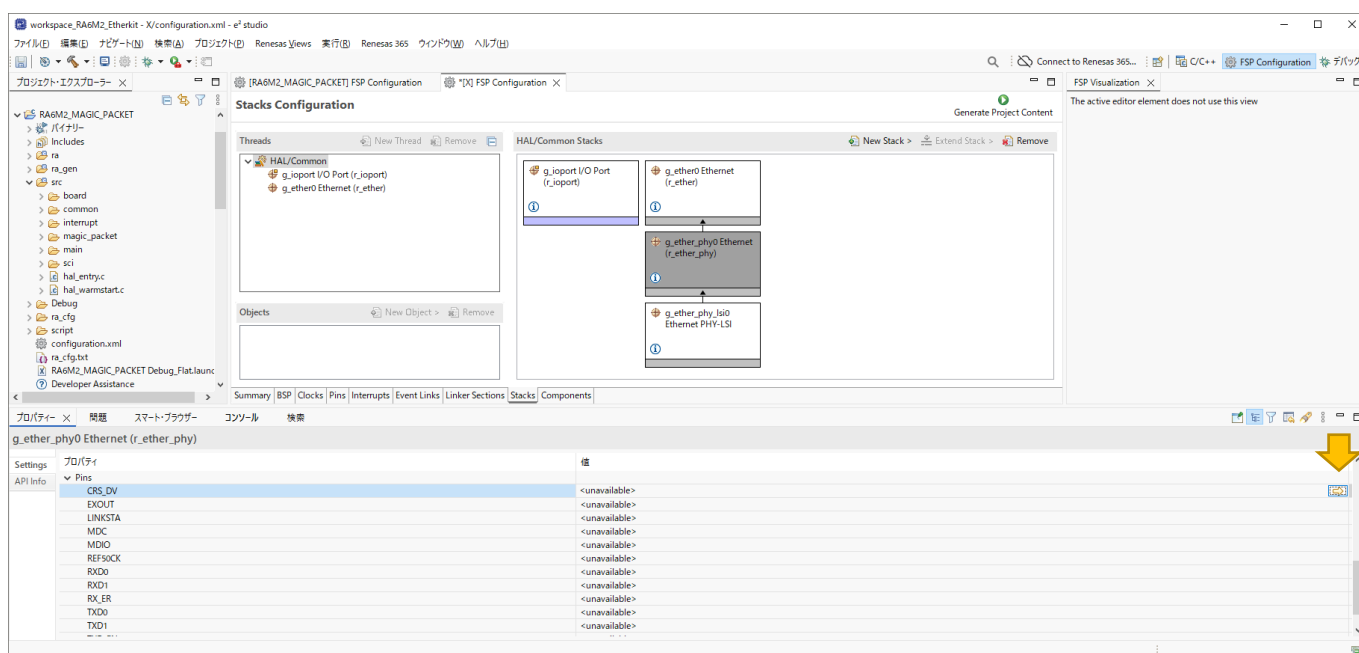
「プロパティ」タブを選択。

PHY-LSI Address を 31 に設定してください。

デフォルトから変更必要な項目	デフォルト値		変更後の値
Module g_ether_phy_lsi0 Ethernet PHY-LSI			
PHY-LSI Address	0	→	31

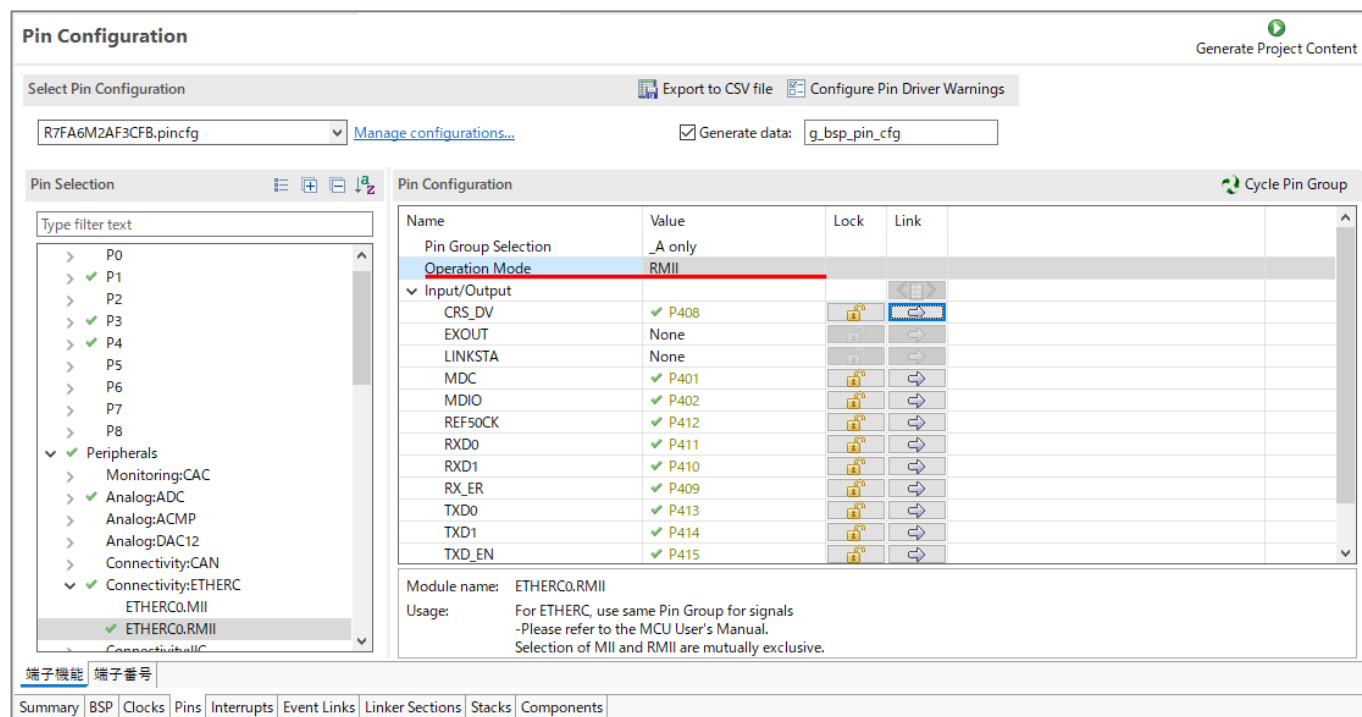
Ethernet オプションボード側で PHY アドレスが 31(=0b11111)に設定されているため、上記値を入力してください。

・r_ether_phy



r_ether_phy にも、同じ設定項目がありますので、31 に設定してください。

次に、Pins の設定 (端子設定) を行います。Pins の端子名の右矢印のアイコンを押すと、端子設定画面に飛びます。

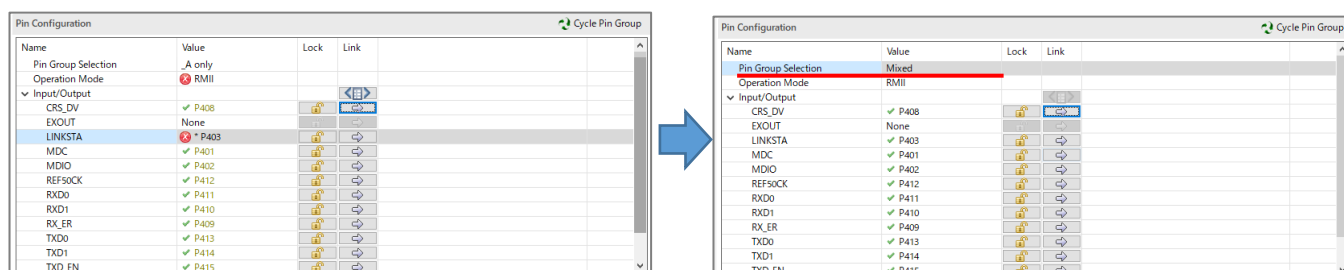


Operation Mode を RMII に設定します。

	HSBRA6M2F144 HSBRA6M4F144	HSBRA6M3F176 HSBRA6M5F176	HSBRA8E1F144 HSBRA8M1F144 (*4)	HSBRA8M1F176 HSBRA8T1F176 (*6)	備考
CRS_DV	P408	P705	P705	P705	
EXOUT	None	None	None	None	
LINKSTA	P403 (*2)	P403	P403	P403	
MDC	P401	P401 (*3)	P401 (*5)	P401	
MDIO	P402	P402 (*3)	P402 (*5)	P402	
REF50CLK	P412	P701	P701	P701	
RXD0	P411	P702	P702	P702	
RXD1	P410	P703	P703	P703	
RX_ER	P409	P704	P704	P704	
TXD0	P413	P700	P700	P700	
TXD1	P414	P406	P406	P406	
TXD_EN	P415	P405	P405	P405	
WOL	P400 (*1)(*2)	P400 (*1)	P400 (*1)	None (*7)	

(*1)RAXxx_MAGIC_PACKET, RAXxx_MAGIC_PACKET2, RAXxx_ETHER_NOAPI のプロジェクトで使用
(他のプロジェクトでは未使用)

端子は、上記を選んでください。



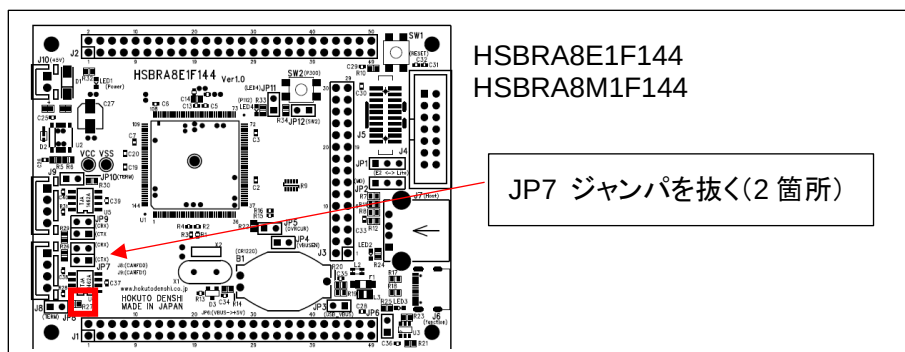
上記表の端子を選択して、×印が付いた場合は、Pin Group Selection を Mixed に変更してください。

(*2)最初に A_only を選んで Mixed に変更、赤字の部分の端子変更すると簡単に設定可能

(*3)最初に B_only を選んで Mixed に変更、赤字の部分の端子変更すると簡単に設定可能

(*4)B_only を選んで、LINKSTA と WOL を追加すると簡単に設定可能

(*5)CAN0 の端子と兼用になっていますので、Ether 機能使用時はボードの JP7-A, JP7-B は抜いてください



(*6)B_only を選んで、LINKSTA を追加すると簡単に設定可能

(*7)プッシュスイッチで P400 を使用しているので WOL 端子は未設定とする

・r_ether

デフォルトから変更が必要な項目のみ示します。

デフォルトから変更必要な項目	デフォルト値		変更後の値	備考
Common				
Link Signal Change Flag	Unused	→	Used	
Module g_ether0 Ethernet(r_ether)				
General				
MAC Address	00:01:02:03:04:05	→	00:0d:76:00:40:01	設定値は任意
Buffers				
Number of TX Buffer	1	→	8	設定値は任意
Number of RX Buffer	1	→	8	設定値は任意
Interrupts				
Interrupt priority	Priority 12		Priority 1	設定値は任意
Callback	NULL		ether0_callback	

Link Signal Change Flag は、PHY-LSI から出力される LINKSTA 信号を使う設定です。PHY-LSI 側で LINK-UP (Ether のケーブルを挿した)、LINK-DOWN (Ether のケーブルを抜いた) 際に、PHY-LSI はマイコンの LINKSTA 端子に対して信号を送りますので、マイコン側はその信号を検知して LINK 状態を判断します。

MAC Address は、本キットのデフォルト値である 00-0d-76-00-40-01 に設定していますが、設定値は任意です。また、後でユーザプログラムで書き換えもできますので、デフォルト値のままでも問題はありません。

Number of TX Buffer, Number of RX Buffer は Ether の送信バッファと受信バッファをいくつ用意するかの設定です。FSP の設定では 8 が最大値なので、ここでは最大値としています。バッファ 1 つあたり、RAM を 1536 バイト消費します。(TX:8 個、RX:8 個の設定でも、消費する RAM は 24kB 程度なので最大値を設定)

Interrupt priority は、割り込み優先度の設定です。RA では 0 が最高の優先度で、15 が最低の優先度です。Ether の処理を優先的に処理したいので、デフォルトの 12 から優先度を上げています。設定値は任意です。

Callback は Ether の割り込みが掛ったときに呼ばれる関数名を設定します。ユーザプログラム内で、ether0_callback() という関数を用意しているので、その関数名を設定します。(ユーザ側で作成している関数名と合わせる必要があります。)

g_ether0 Ethernet (r_ether)		
Settings	プロパティ	値
API Info	▼ Common	
	Parameter Checking	Default (BSP)
	ET0_LINKSTA Pin Status Flag	Fall -> Rise
	Link Signal Change Flag	Used
	Read Buffer Size Checking	Disabled
	▼ Module g_ether0 Ethernet (r_ether)	
	▼ General	
	Name	g_ether0
	Channel	0
	MAC address	00:0d:76:00:40:01
	Zero-copy Mode	Disable
	Flow control functionality	Disable
	> Filters	
	▼ Buffers	
	Number of TX buffer	8
	Number of RX buffer	8
	Allocate RX buffer	Enable
	Buffer size	1514
	Padding size	Disable
	Padding offset	0
	▼ Interrupts	
	> EESR event	
	> ECSR event	
	Interrupt priority	Priority 1
	Callback	ether0_callback

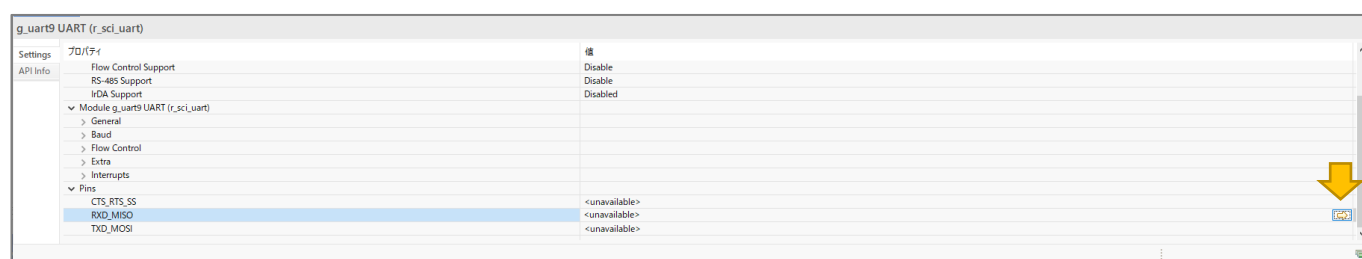
設定後の画面は上記の様になります。

・r_sci_uart

デフォルトから変更必要な項目	デフォルト値		変更後の値	備考
Module g_uart9 UART(r_sci_uart)				
General				
Name	g_uart0	→	g_uart9	設定値は任意
Channel	0	→	9	
Interrupts				
Callback	NULL	→	uart9_callback	

Channel は SCI9 を使いますので、9 に設定してください。Name は識別名なので、どのような名称を設定しても問題はありせん。本プロジェクトでは、SCI9 を使っているので、g_uart9 という名称に変えています。ここの名称と、プログラム内で SCI9 を使用する際の引数の値は合わせる必要があります。(この名称を g_uart9 以外の値とすると、プログラムコードの書き換えは必要です。)

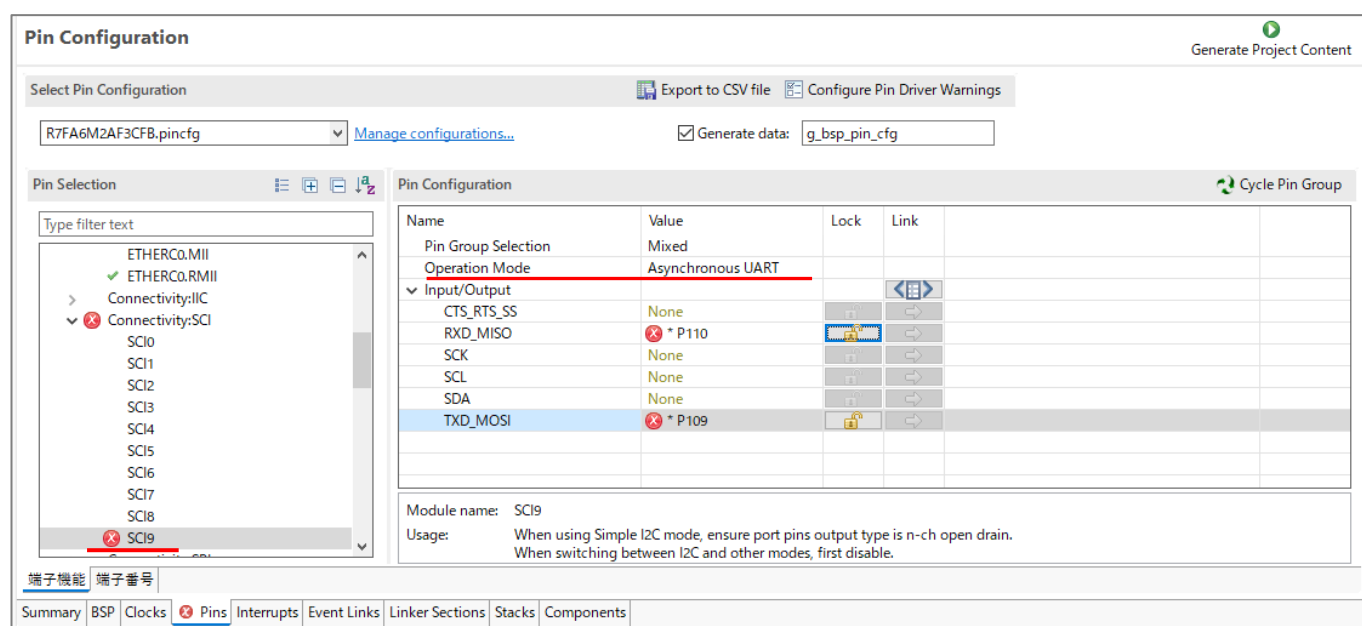
Pins 設定



「r_sci_uart の Pins の右矢印から飛ぶ」

「Pins タブの Connectivity:SCI SCI9 を選択」

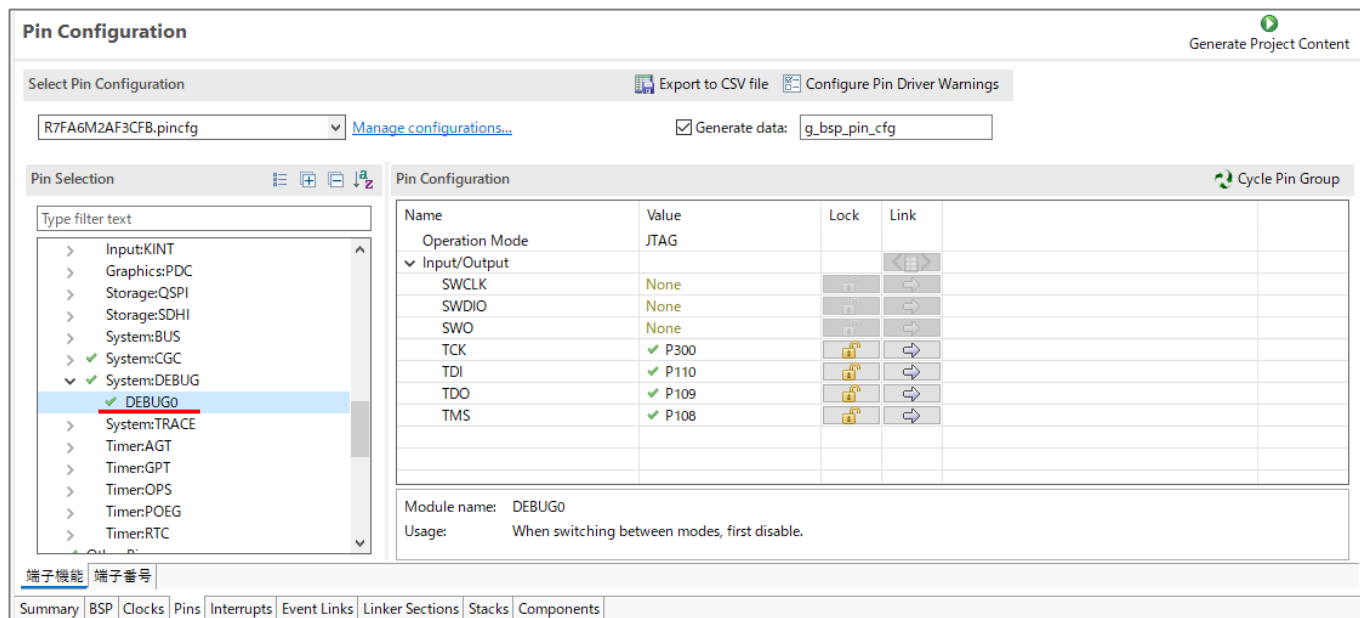
のどちらでも、同じ設定項目の画面となります。



Operation Mode を Asynchronous UART に設定してください。

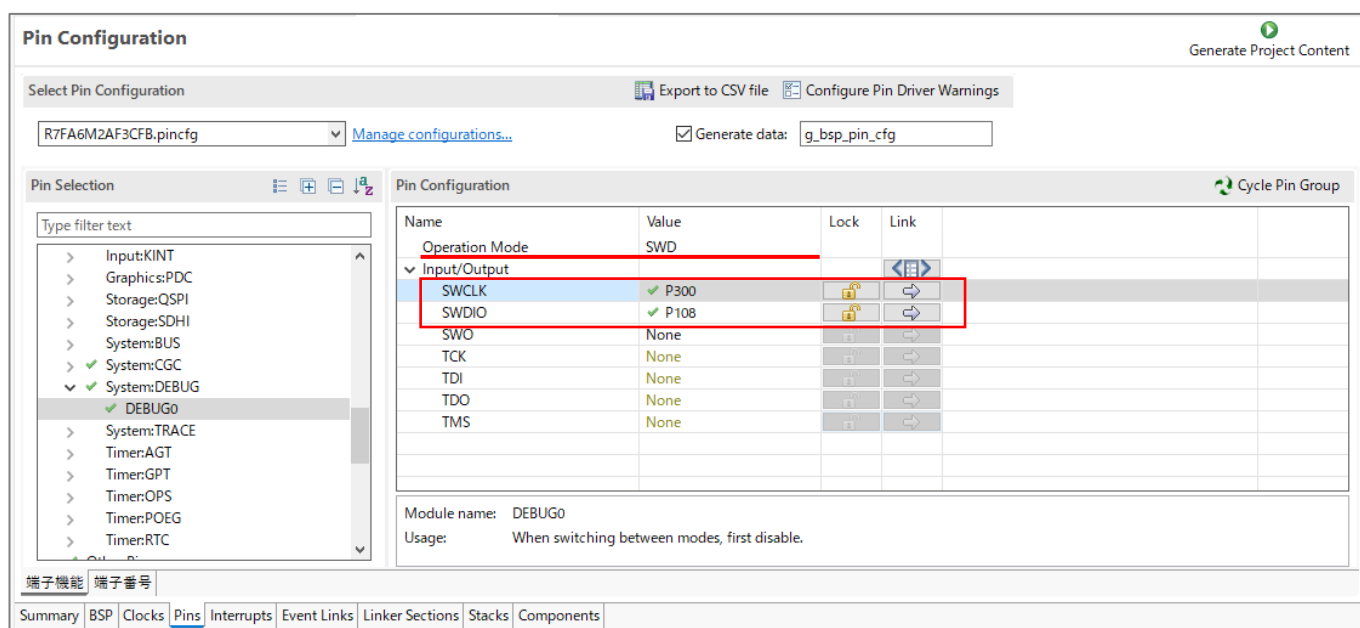
	HSBRA6M2F144 HSBRA6M4F144	HSBRA6M3F176 HSBRA6M5F176	HSBRA8E1F144 HSBRA8M1F144	HSBRA8M1F176 HSBRA8T1F176	備考
RXD	P110	P110	P208	P208	
TXD	P109	P109	P209	P209	

TXD と RXD は上記端子を選択してください。この時、端子には赤の×印が付きます。これは、デバッグで使用する端子と競合しているためです。



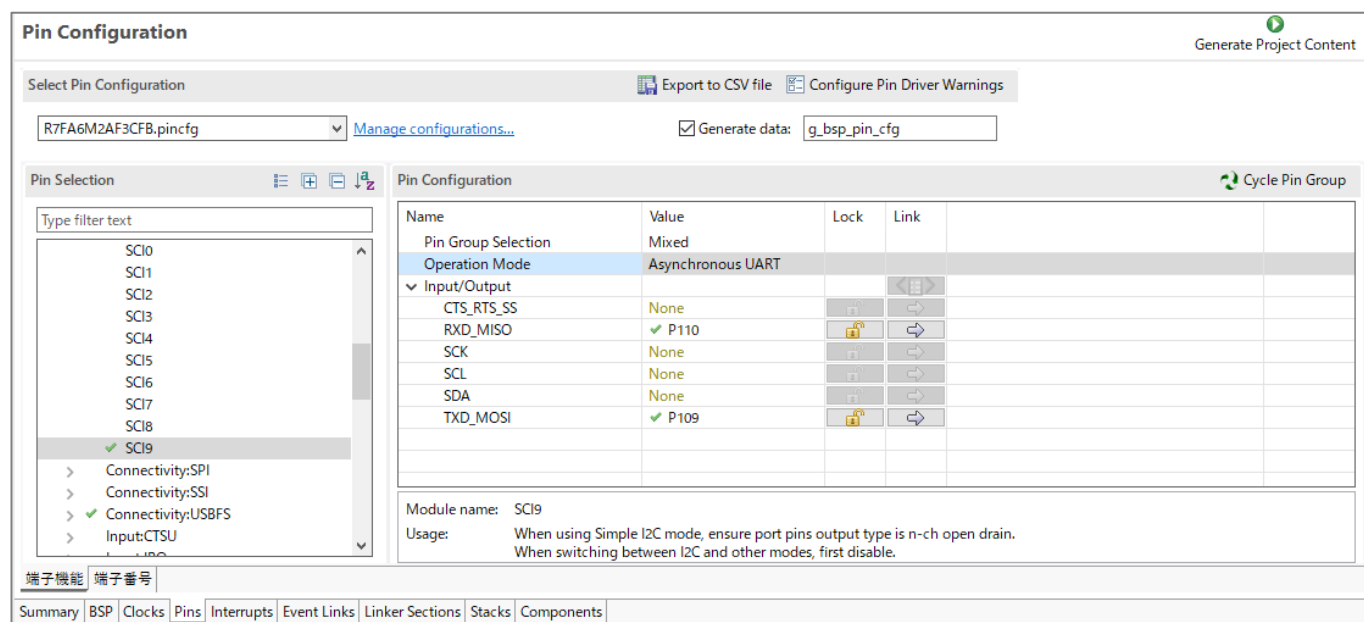
The screenshot shows the 'Pin Configuration' window. On the left, under 'Pin Selection', the tree view shows 'System:DEBUG' expanded, with 'DEBUG0' selected and highlighted in red. The main table shows the configuration for 'DEBUG0'. The 'Operation Mode' is set to 'JTAG'. The 'Input/Output' section shows 'SWCLK' and 'SWDIO' both set to 'None'. The 'Lock' column shows 'None' for SWCLK and 'P300' for SWDIO. The 'Link' column shows 'None' for SWCLK and 'P108' for SWDIO. The 'Module name' is 'DEBUG0' and the 'Usage' is 'When switching between modes, first disable.'.

Pins タブの System:DEBUG DEBUG0 を選択して、デバッグで使用する端子設定を開きます。



The screenshot shows the 'Pin Configuration' window with the 'Pins' tab selected. The 'Operation Mode' is now set to 'SWD'. The 'Input/Output' section shows 'SWCLK' and 'SWDIO' both set to 'P300'. The 'Lock' column shows 'None' for SWCLK and 'P108' for SWDIO. The 'Link' column shows 'None' for SWCLK and 'P108' for SWDIO. The 'Module name' is 'DEBUG0' and the 'Usage' is 'When switching between modes, first disable.'.

Operation Mode を SWD に変更します。SWCLK と SWDIO の端子を選択します(選択できる端子は 1 つしかありません)。※RA5(HSBRA6M5F176)ではデフォルトで SWDIO の方が選択されています



DEBUG で使用する端子設定を変更すると、SCI9 での端子設定のエラーは解消されます。

・r_gpt

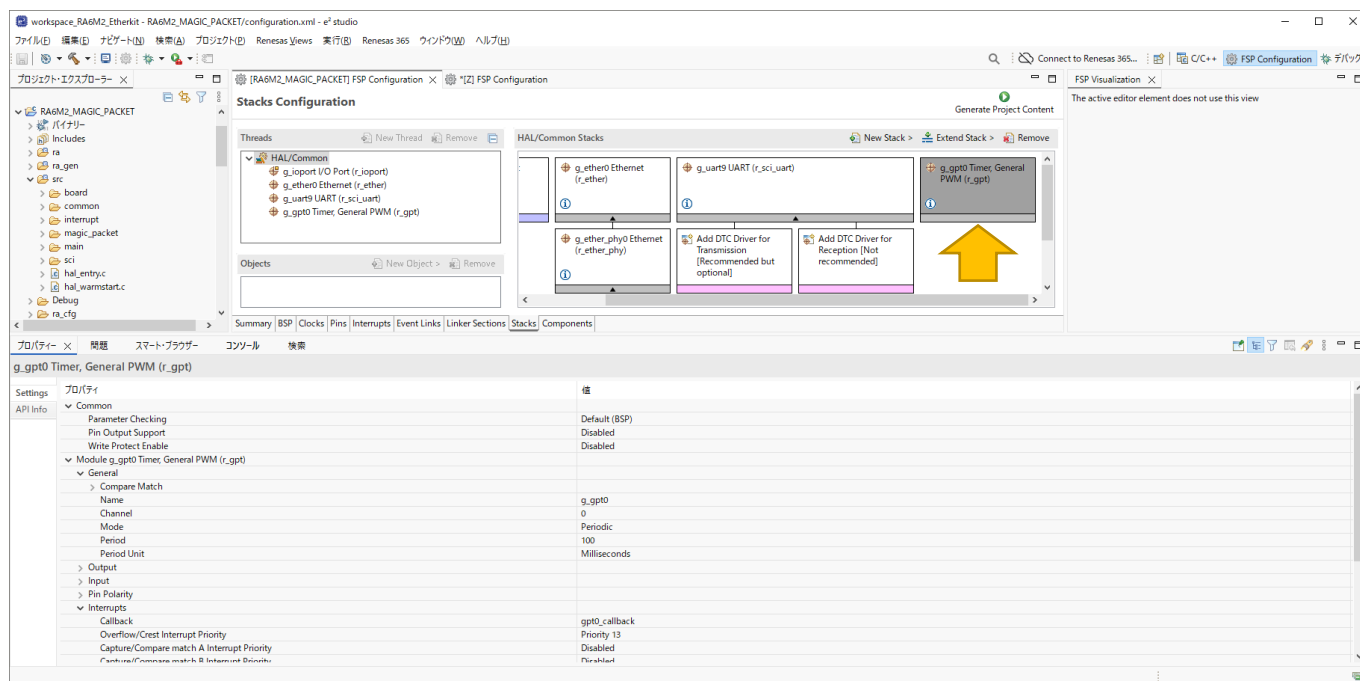
デフォルトから変更必要な項目	デフォルト値		変更後の値	備考
Module g_gpt0 Timer, General PWM(r_gpt)				
General				
Name	g_timer0	→	g_gpt0	設定値は任意
Period	0x10000	→	100	
Period Unit	Raw Counts	→	Milliseconds	
Interrupts				
Callback	NULL	→	gpt0_callback	
Overflow/Crest Interrupt Priority	Disabled	→	Priority 13	設定値は任意

Name は識別名なので、値は任意で構いませんが、本プロジェクトでは使用するタイマ種 GPT のチャンネル 0 を使用する場合は、g_gpt0 という名称に設定しています。(値を変えた場合はプログラム内で g_gpt0 を参照している部分を設定した値に書き換える必要があります。)

Period と Period Unit はタイマの周期で、本プロジェクトでは 100ms に設定したいので、100, Milliseconds を設定しています。

Callback は、ユーザ側で作成する関数名に合わせて、gpt0_callback としています。

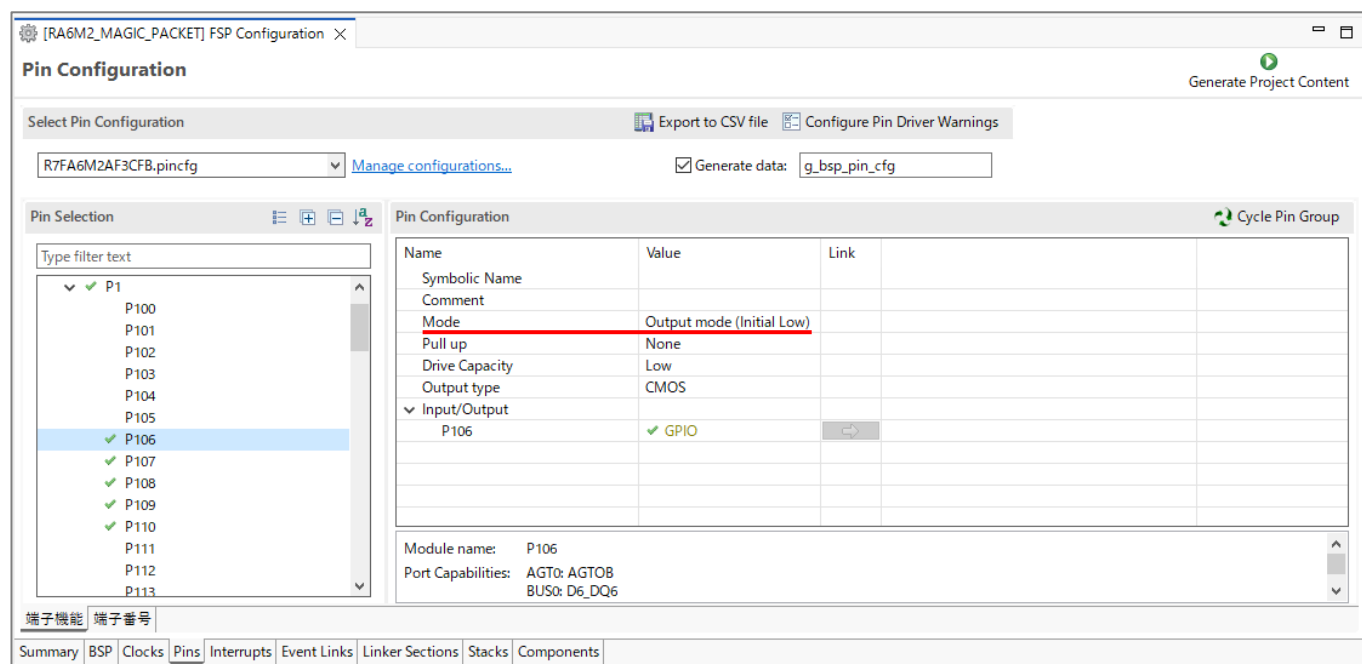
Overflow/Crest Interrupt Priority は、デフォルトでは Disabled と割り込みが掛らない設定なので、何らかの優先度を割り当てます。100ms のタイマは、スイッチのチャタリング防止に使用していますので、ここでは優先度 13(優先度低め)に設定します。



r_gpt の設定後の画面は上記の様になります。

FSP で、
Stack の追加
Stack のプロパティの設定
を行う事により、マイコンに内蔵されている、ether や UART、タイマ等の機能を容易に使用できるようになります。

2.2.4. 端子設定



Pins タブを選択。

Ether や UART、DEBUG で使用する端子は既に設定済みです。ここでは、LED とスイッチ、Ether の PHY-LSI のリセット信号の設定を行います。

	HSBRA6M2F144 HSBRA6M4F144	HSBRA6M3F176 HSBRA6M5F176	HSBRA8E1F144 HSBRA8M1F144	HSBRA8M1F176 HSBRA8T1F176	Mode 設定値
LED	P106	P106	P112	P408 (*1)	Output Mode(Initial Low)
プッシュスイッチ	P107	P107	P300	P400 (*2)	Input Mode
PHY-LSI RESET	P405	P707	P410	P706	Output Mode(Initial Low)

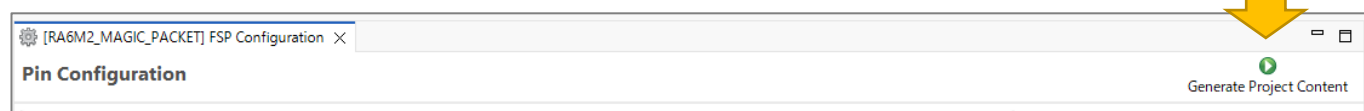
本プロジェクトで使用する、LED と PHY-LSI のリセット端子を出力モード(初期値 L)、プッシュスイッチの端子を入力モードに設定します。

(*1)ボード上の評価 LED(LED6)は P401 ですが、P401/MDC は Ether で使用するので、本プログラムでは代わりに LED2 (USB-FS の LED)を使用しています (USB-FS の Host 端子に給電する事で LED2 を点灯させます)

(*2)Ether の WOL 端子と被っているので、本プログラムでは P400 をスイッチの入力として使用します (RA8M1-176pin, RA8T1 では、プログラムで Ether ボード上の WOL-LED の点灯は制御されません)

2.2.5. コード生成

FSP Configuration の一通りの設定を終えたら



Generate Project Content を押してください。これにより、設定項目が自動生成のソースコードに反映されます。
(今後、設定項目を変更した場合は、最後にこのボタンを押す様にしてください。)

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
board	board.h	ボード固有設定
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_ether.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ

FSP 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。ユーザ側で作成したプログラムコードは、プロジェクトフォルダの src 以下にコピーすると自動的にビルド対象となります。

・board.h

本ファイル内では、ボードの固有設定、PHY チップのリセット端子やスイッチ、LED の端子が定義されています。

```
//マイコン名
#define MCU_TYPE          "RA6M2"

//PHY リセット端子
#define PHY_RESET          R_PORT4->PODR_b.PODR5

//ボードのプッシュスイッチ
#define PUSH_SW            R_PORT1->PIDR_b.PIDR7

//ボードの DIP スイッチ
//#define DIP_SW            //未定義(DIP スイッチなし)

//ボードの LED
#define LED                R_PORT1->PODR_b.PODR6
#define LED_ON              (1)
#define LED_OFF             (0)
```

・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 = 0;
```

MAC アドレスや使用する Ether のチャンネル等を定義しています。(Ether のチャンネル番号は画面表示に使用)
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 ether0_callback(ether_callback_args_t * p_args);

void gpt0_callback(timer_callback_args_t *p_args);
```

上記の関数が定義されています。

関数名	内容
ether0_callback	Ether のコールバック関数(詳細は後述)
gpt0_callback	GPT0 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 : 0xffx6
    //yy yy yy yy yy yy : MagicPacket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

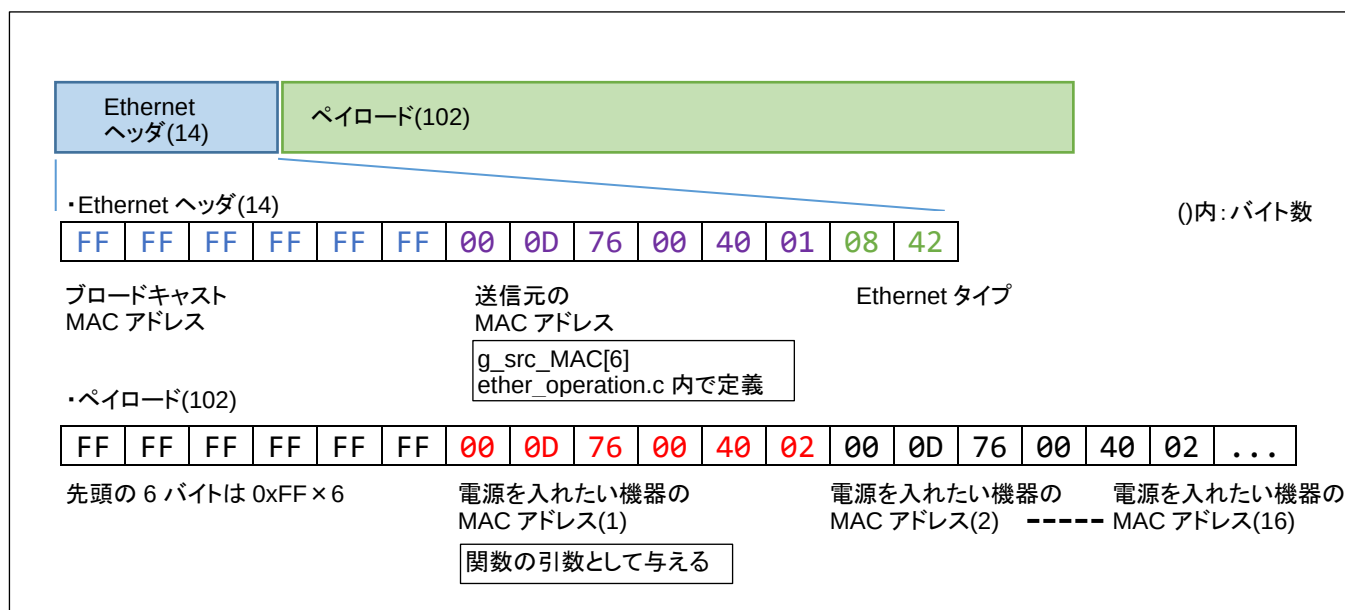
    for (i=0; i<16; i++)
    {
        for (j=0; j<6; j++) send_data[14 + 6 + i * 6 + j] = MAC[j]; //MACアドレスを16個繰り返す
    }

    //パケット送信
    ret = R_ETHER_Write(&g_ether0_ctrl, (void *)send_data, sizeof(send_data));

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

    if (ret != FSP_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。)

FSP では、src 以下の

hal_entry.c

内の hal_entry() が、ユーザ作成プログラムのエントリポイントになっています。(ここから、ユーザ作成コードに飛ばす。)

hal_entry.c

```
extern void main_ether(void);

void hal_entry(void)
{
    /* TODO: add your own code here */
    main_ether();
}
```

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

main_ether.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.c

main_ether() ※抜粋

```
sci_write_str("\nCopyright (C) 2026 HokutoDenshi. All Rights Reserved.\n");
sci_write_str(MCU_TYPE);
sci_write_str(" Ethernet 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(" PUSH-SW : 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");
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.
RA6M2 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:
 PUSH-SW : 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;

//Ethernet アドレス設定 (FSP で設定した値を上書き)
for (int i=0; i<6; i++)
{
    g_ether0_cfg.p_mac_address[i] = g_src_MAC[i];
}

//Ether のオープン
ret = R_ETHER_Open(&g_ether0_ctrl, &g_ether0_cfg);

if (ret != FSP_SUCCESS)
{
    sci_write_str("Ethernet interface open error.\n");
    while (1);
}
```

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、RA マイコンと RJ45 コネクタの間に接続されている Ethernet-PHY (RX 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、PHY_RESET(RA6M2 のボードでは P405) を H にしているだけです)

R_ETHER_Open()

は、FSP で追加した、r_ether の初期化の API 関数です。

main_ether() (続き)

```
//メインループ
while (1)
{
    //LINK-UP 待ち
    if (R_ETHER_LinkProcess(&g_ether0_ctrl) == FSP_SUCCESS)
    {
        if (g_operation_mode != OPERATION_RECEIVE)
        {
            //WOL 受信専用モードの時はデータ送信不可, 送信モードの時は SW が押された場合は本ボードから
            MagicPacket を送出
            if ((PUSH_SW == 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()が正常終了した場合、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、ボードのプッシュスイッチが押された場合は 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 GPT0 の割り込み内で、10 カウント経過後に、プッシュスイッチの読み取りが再度有効化されます。)

○割り込み関数内での処理(ether_interrupt.c)

ether_callback()

```
void ether0_callback(ether_callback_args_t * p_args)
{
    fsp_err_t ret;
    uint32_t dummy_s;
    static uint8_t dummy_d[1514];

    switch (p_args->event)
    {
        case ETHER_EVENT_WAKEON_LAN:
            //ECSR の割り込みで MPD を有効にした場合
            //R_ETHER_LinkProcess()でのステータス変更
            //により、この部分に飛んでくる
            //※FSP の ether の割り込みで MPD を有効にした場合、2 回この部分が実行される
            sci_write_str("Ether");
            sci_write_uint8((unsigned char)g_ether_channel);
            sci_write_str(": MagicPacket received.¥n");
            LED = LED_ON; //ボードの LED を点灯させる
            R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS); //WOL-LED を目視できるようウェイト(ウェイト
//が無い場合は 1.5ms 間消灯)
            break;

        case ETHER_EVENT_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_WakeOnLANEnable(&g_ether0_ctrl);
                if (ret == FSP_SUCCESS)
                {
                    sci_write_str("-> MagicPacket waiting ...¥n");
                }
                else
                {
                    sci_write_str("NORMAL -> WOL(MagicPacket) mode change error.¥n");
                }
            }
            break;

        case ETHER_EVENT_LINK_OFF:
            sci_write_str("Ether");
            sci_write_uint8((unsigned char)g_ether_channel);
            sci_write_str(": LINK-DOWN¥n");
            break;

        case ETHER_EVENT_RX_COMPLETE:
            while (1) //本プログラムでは受信データがある場合でもバッファ読み出しを行うのみ
            {
                ret = R_ETHER_Read(&g_ether0_ctrl, (void *)dummy_d, &dummy_s);
                if (ret == FSP_SUCCESS)
                {
                    //読み出したデータは本プログラムでは使用しない
                }
                else if (ret == FSP_ERR_ETHER_ERROR_NO_DATA)
                {
                    break;
                }
                else
                {
                    sci_write_str("Error: Receive fail.¥n");
                    break;
                }
            }
            //sci_write_str("data received.¥n");
            break;
    }
}
```

ether_callback() ※続き

```

    case ETHER_EVENT_RX_MESSAGE_LOST:
        sci_write_str("RX data lost.%n");
        break;

    case ETHER_EVENT_TX_COMPLETE:
        //sci_write_str("data send.%n");
        break;

    case ETHER_EVENT_TX_BUFFER_EMPTY:
        sci_write_str("TX data lost.%n");
        break;

    case ETHER_EVENT_TX_ABORTED:
        sci_write_str("send aborted.%n");
        break;

    case ETHER_EVENT_ERR_GLOBAL :
        //sci_write_str("error detected.%n");
        break;

    default:
        break;
}
}

```

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)
- 等に呼び出されます。

p_args->event の値を見る事で、どのような要因で割り込みが入ったのかを判断する事ができます。

MagicPacketを受信した際には、ボードの LED 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) するプログラムコードはありません。

(なお、RAXxx_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 MagicPacket 受信機能に任せています。)

(データを受信した際は、

case ETHER_EVENT_RX_COMPLETE

に飛んできますが、本プロジェクトではこの部分で受信データの処理は行っていません。以降のプロジェクトでは、この部分で受信したデータを処理しています。)

gpt0_callback()

```
void gpt0_callback(timer_callback_args_t *p_args)
{
    //100ms 割り込み関数

    extern unsigned short g_sw_read_counter;

    if (TIMER_EVENT_CYCLE_END == p_args->event)
    {
        //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、GPT0 の 100ms 毎に呼び出されます。プッシュスイッチを押した際に一定期間(10 カウント)プッシュスイッチ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_uint16_hex	16bit 変数を 16 進数 4 桁で表示
sci_write_uint32_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、FSP 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、FSP で設定した値となります。

以上が、RAxxx_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と思います。

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

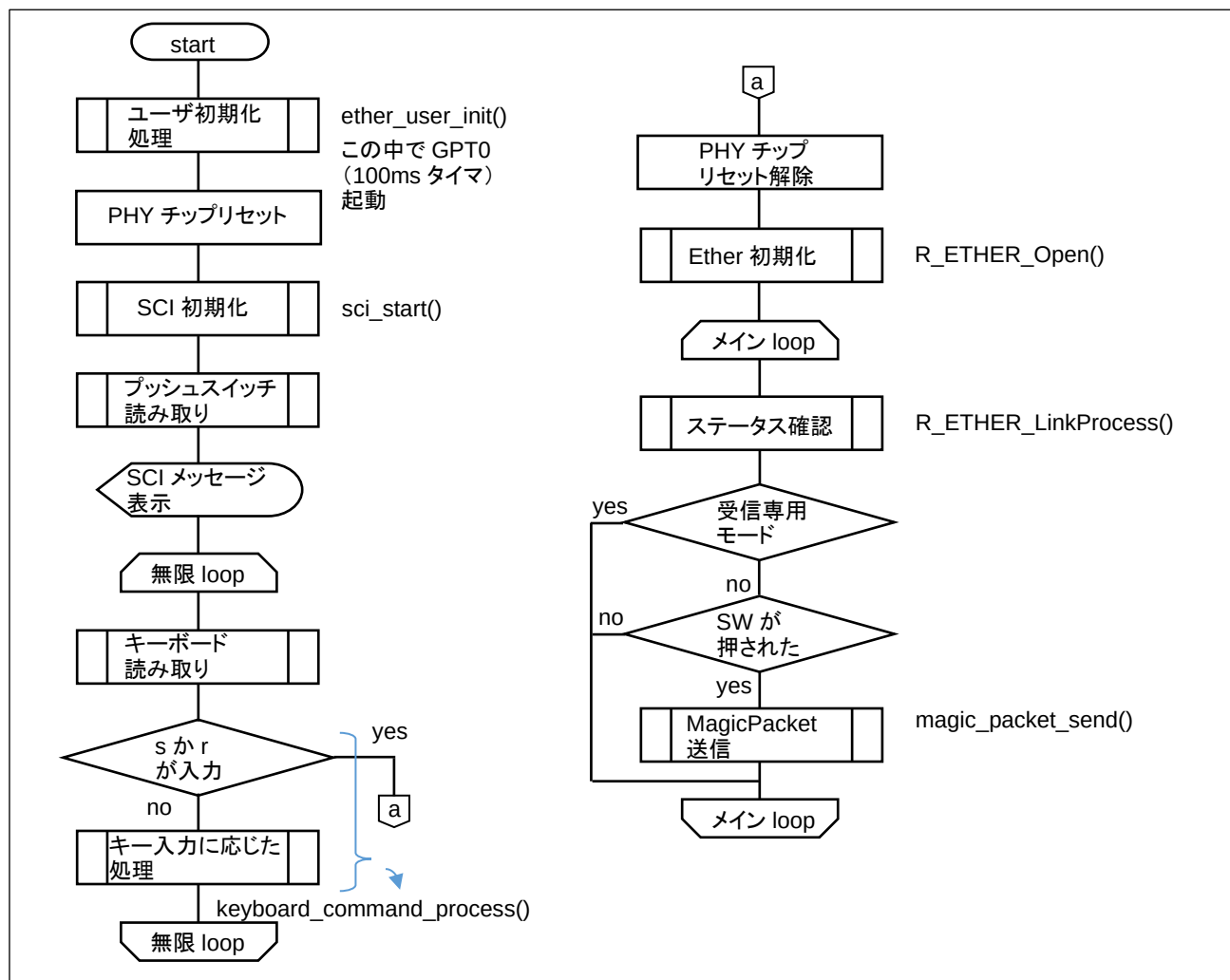
戻り値: なし

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

・main_ether() (main_ether.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. RAXxx_MAGIC_PACKET2 プロジェクト

CD 内の

SOURCE¥RAXxx_yyy¥RAXxx_MAGIC_PACKET2.zip

(使用しているマイコンボードに応じたファイル)をワークスペースにインポートしてください。

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。

・FSP で追加するスタック

コンポーネント	備考
r_ether	Ethernet
r_sci_uart	SCI(UART)通信 (マイコン種によっては r_sci_b_uart)
r_gpt	タイマ(100ms 毎)

(→RAXxx_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
board	board.h	ボード固有設定
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
interrupt	ether_operation.h	上記ヘッダ
	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_ether.c	メイン関数
sci	sci.c	UART 処理関数
	sci.h	上記ヘッダ

(→RAXxx_MAGIC_PACKET プロジェクトと同じ)

使用しているスタックとソースコードの構成は、RAXxx_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.
RA6M2 Ether MagicPacket(WOL) sample program2.

COMMAND:
  m : target MAC address set
  i : target broadcast IP address set
  l : 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.
  U : send packet type is IP/UDP.(default)
---
  s : Operation start with send/receive mode
  r : Operation start with receive only mode
---

USAGE:
  PUSH-SW : 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 : l command for change
---
First m, i, l, p, S, W, U 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	(l コマンド(大文字 l)で変更)

です。(起動時にプッシュスイッチが押されていると、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, l コマンドでアドレスの設定を行ってください。

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

U 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-SW -> send MagicPacket  
this board MAC address = 00-0D-76-00-40-01  
Ether0: LINK-UP
```

- 送信・受信モード、ボードのプッシュスイッチを押すと 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に変更可能ですが、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。

・main_ether.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;
```

・board.h

```
//PHY リセット端子  
#define PHY_RESET      R_PORT4->PODR_b.PODR5
```

PHY-LSI のリセット端子には、RA6M2 のボードでは P405 が接続されており、P405=L で PHY-LSI をリセットします。

P405 を出力端子に設定しているのは、FSP の端子設定です。

PHY_RESET=0 の後で、100us 以上経過してから、PHY_RESET=1 を実行する事で 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。

・main_ether.c

```
//Ether のオープン  
ret = R_ETHER_Open(&g_ether0_ctrl, &g_ether0_cfg);
```

r_ether スタックで提供されている API 関数です。

Ethernet モジュールの初期化・動作開始を行います。

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

    (中略, send_dataの中身の設定)

    //パケット送信
    ret = R_ETHER_Write(&g_ether0_ctrl, (void *)send_data, sizeof(send_data));

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

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

magic_packet_send_wol()関数は
RAXxx_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(&g_ether0_ctrl, (void *)send_data, 112);

の様に、

- 1 番目の引数は使用する Ether のコントロールブロック (FSP で設定した名称)
- 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、

FSP_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)

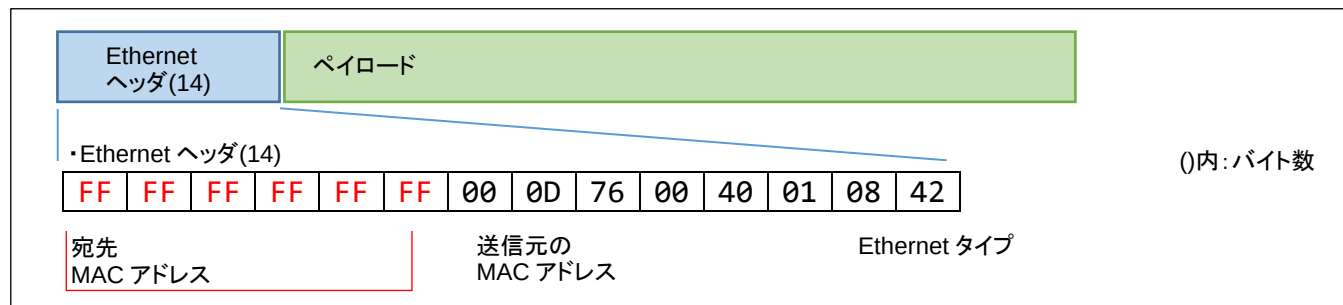
取扱説明書では、MagicPacket や ping、UDP、TCP 通信で、どのような Ethernet フレームのデータとなるか詳しく解説していますので、送るべき Ethernet フレームの構造 (中身) が判っていれば送信操作は難しくはありません。

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 ether0_callback(ether_callback_args_t * p_args)
{
    fsp_err_t ret;
    uint32_t receive_size = 0;

    switch (p_args->event)
    {
        case ETHER_EVENT_WAKEON_LAN:
            sci_write_str("Ether");
            sci_write_uint8((unsigned char)g_ether_channel);
            sci_write_str(": MagicPacket received.\n");
            LED = LED_ON; //ボードの LED を点灯させる
            R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS); //WOL-LED を目視できるようウェイト(ウェイトが無い場合は 1.5ms 間消灯)
            break;

        case ETHER_EVENT_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_WakeOnLANEnable(&g_ether0_ctrl);
                if (ret == FSP_SUCCESS)
                {
                    sci_write_str("-> MagicPacket waiting ...\n");
                }
                else
                {
                    sci_write_str("NORMAL -> WOL(MagicPacket) mode change error.\n");
                }
            }
            break;
    }
}
```

•ether_interrupt.c（一部省略，続き）

```

case ETHER_EVENT_LINK_OFF:
    sci_write_str("Ether");
    sci_write_uint8((unsigned char)g_ether_channel);
    sci_write_str(": LINK-DOWN¥n");
    break;

case ETHER_EVENT_RX_COMPLETE:
    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，2つ目以降のデータは空読みとなる
    {
        //Zero-copy モードを使わない
        ret = R_ETHER_Read(&g_ether0_ctrl, (void *)recv_tmp_buf, &receive_size);
        if (ret == FSP_SUCCESS)
        {
            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, recv_tmp_buf, receive_size); //受信データを
                g_ether_recv_buf にコピー(2 回コピーが生じる)
                g_ether_recv_flag = true; //受信フラグを立てる
            }
            else if (ret == FSP_ERR_ETHER_ERROR_NO_DATA)
            {
                break;
            }
            else
            {
                sci_write_str("Error: Receive fail.¥n");
                break;
            }
        }
    }

#ifdef ETHER_DEBUG_LEVEL_0
    sci_write_str("data received, ");
    sci_write_uint32(receive_size);
    sci_write_str(" bytes¥n");
#endif
    //sci_write_str("data received.¥n");
    break;

```

Ether の割り込みは 2 種類あります。

(1)ETHERC0 のレジスタ(ETHERC0.ECSR)が変化した場合の割り込み

→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が。

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。

(2)EDMAC0 のレジスタ(EDMAC0.EESR)が変化した場合の割り込み

→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

割り込みコールバック関数(ether0_callback())は、(1)(2)の両方を兼ねています。((1)(2)のどちらの事象が発生した場合でも、ether0_callback に飛んてくる。case 文で割り込み発生原因に応じた処理を実行する事が可能。)

データの受信時は、p_args->event が ETHER_EVENT_RX_COMPLETE になりますので、この case 文のところに受信時の処理を記載します。

```
ret = R_ETHER_Read(&g_ether0_ctrl, (void *)recv_tmp_buf, &receive_size);
```

で、recv_tmp_buf に受信データを格納。receive_size には、受信データバイト数が返ります。

```
memcpy(g_ether_recv_buf, recv_tmp_buf, receive_size)
```

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 receive_size です。

ー参考 Zero-copy モードに関してー

g_ether0 Ethernet (r_ether)		
Settings	プロパティ	値
API Info	Link Signal Change Flag	Used
	Read Buffer Size Checking	Disabled
	▼ Module g_ether0 Ethernet (r_ether)	
	▼ General	
	Name	g_ether0
	Channel	0
	MAC address	00:0d:76:00:40:01
	<u>Zero-copy Mode</u>	<u>Disable</u>
	Flow control functionality	Disable

r_ether スタックの設定で、Zero-copy Mode の設定があり、現状 Disable(デフォルト)に設定しています。

・Disable の場合 (Zero-copy モードを使わない場合)

```
static uint8_t recv_tmp_buf[ETHER_MAX_PACKET_SIZE]; //ETHER_MAX_PACKET_SIZE=1514
ret = R_ETHER_Read(&g_ether0_ctrl, (void *)recv_tmp_buf, &receive_size); //コピー(1)
if (ret == FSP_SUCCESS)
{
    if (g_ether_recv_flag == false)
    {
        memcpy(g_ether_recv_buf, recv_tmp_buf, receive_size); //コピー(2)
        g_ether_recv_flag = true;
    }
}
```

本プロジェクトでは上記の様な処理としています。この場合、R_ETHER_Read()で、FSP で確保した受信バッファから、recv_tmp_buf にコピーが行われます。その後、recv_tmp_buf から g_ether_recv_buf(ユーザが確保した受信バッファ)に再度コピーが行われます。(1)(2)で 2 度のデータコピーが行われる事となり、効率が良くありません。

2 回コピーしているのは、読み出しが成功した場合のみ、g_ether_recv_buf を更新したいという理由です。一度 recv_tmp_buf を経由する事なく、直接 g_ether_recv_buf にコピーする形でも問題はないと思います。

受信バッファが未読み出しの場合 (g_ether_recv_flag が立っている場合) 2 回目のデータコピーは行われませんが、データの空読みを行う場合でも、1 回はデータコピーを行う必要があります。

・Enable の場合 (Zero-copy モードを使用する場合)

```
static uint8_t *buffer_address;
ret = R_ETHER_Read(&g_ether0_ctrl, (void *)&buffer_address, &receive_size);
if (ret == FSP_SUCCESS)
{
    if (g_ether_recv_flag == false)
    {
        memcpy(g_ether_recv_buf, buffer_address, receive_size); //コピー(1)
        g_ether_recv_flag = true;
    }
    R_ETHER_BufferRelease(&g_ether0_ctrl); //受信バッファを開放する(zero copy モード)
}
```

Zero-copy モードを使う場合、R_ETHER_Read() ではデータの先頭アドレスを返すだけですので、データのコピーは行われません。データのコピーは memcpy で 1 回です。また、受信バッファが未読み出しの場合は 1 回もコピーが生じません。

基本的には、Zero-copy モードの方が効率が良いので Zero-copy モードを使いたいのですが、問題があり Zero-copy モードは未使用としています。

Zero-copy モードを有効にすると、受信バッファだけではなく、送信バッファ側も Zero-copy モードとなります。

送信時、Zero-copy モードが有効の場合、

```
R_ETHER_Write(&g_ether0_ctrl, (void *)send_data, sizeof(send_data));
```

send_data のデータはコピーされずに、EDMAC (Ether 専用の DMA コントローラ) に、send_data のアドレスが渡されます。

この場合面倒なのが、以下の 3 点です。

(1) send_data のアドレスのアライメント

→EDMAC で扱うバッファは、16 バイト (設定に拠る) のアライメントに配置される (16 で割り切れる) 必要がある

(2)send_data が NON-Secure 領域に配置される必要がある

→TrustZone 対応マイコン(RA6M4 や RA8M1 等)に限られますが、EDMAC は Secure 領域のデータを取り扱う事ができませんので、send_buf 変数が特定の領域(アドレス)に配置される様な設定が必要

(3)send_data のデータを保持する期間の管理が必要

→Ether の送信完了割り込み等で送信完了を確認してから、send_data に次のデータを書く等の判断、管理する必要が生じる

(複数の送信バッファを使用する場合、どの送信バッファが現在利用可能かをユーザのプログラムで管理する必要があり非常に手間が掛かる。)

これらの送信バッファに関する事柄は、4 章のプロジェクト(RAxxx_ETHER_NOAPI)にも関係してきます。4 章のプロジェクトでは、Ether に関する FSP は未使用ですのでユーザのコードや設定でケアしています。

FSP のスタックの設定で、送信側は Zero-copy モードを使わない。受信側は Zero-copy モードを使用する設定ができれば良いのですが、現状の FSP6.6.0 ではその様な設定はできないので、受信・送信の両方で Zero-copy モードを使わない設定としています。

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()の戻り値が FSP_SUCCESS 以外になるまで読み出しを行っています。

※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 ether0_callback()内で、受信データを g_ether_rcv_buf にコピー(及び g_ether_rcv_flag を立てる)。メイン関数のメインループ内で、g_ether_rcv_buf のデータを処理しています(処理が終わると g_ether_rcv_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.  
RA6M2 Ether MagicPacket(WOL) sample program2.  
  
COMMAND:  
m : target MAC address set  
i : target broadcast IP address set  
l : 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.  
U : send packet type is IP/UDP.(default)  
---  
s : Operation start with send/receive mode  
r : Operation start with receive only mode  
---
```

何もしなければ、IP/UDP 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、RAxxx_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 U 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 : MagicPaket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_ether0_ctrl, (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 RAxxx_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。

送信で使用する API 関数は、2 章で使用しているのと同じ R_ETHER_Write()です。本プロジェクトでは、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.
RA6M2 Ether MagicPacket(WOL) sample program2.

COMMAND:

m : target MAC address set
i : target broadcast IP address set
l : 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.
U : send packet type is IP/UDP.(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、RAXxx_MAGIC_PACKET の受信動作と変わりません。

・ether_interrupt.c

```
void ether0_callback(ether_callback_args_t * p_args)
{
    fsp_err_t ret;
    uint32_t receive_size = 0;

    switch (p_args->event)
    {
        case ETHER_EVENT_WAKEON_LAN:
            sci_write_str("Ether");
            sci_write_uint8((unsigned char)g_ether_channel);
            sci_write_str(": MagicPacket received.¥n");
            LED = LED_ON; //ボードの LED を点灯させる
            R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS); //WOL-LED を目視できるようウェイト
            break;

        case ETHER_EVENT_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_WakeOnLANEnable(&g_ether0_ctrl1);
                if (ret == FSP_SUCCESS)
                {
                    sci_write_str("-> MagicPacket waiting ...¥n");
                }
                else
                {
                    sci_write_str("NORMAL -> WOL(MagicPacket) mode change error.¥n");
                }
            }
            break;

        case ETHER_EVENT_LINK_OFF:
            sci_write_str("Ether");
            sci_write_uint8((unsigned char)g_ether_channel);
            sci_write_str(": LINK-DOWN¥n");
            break;
    }
}
```

ether0_callback()関数では、p_args->event をモニタする事で割り込み要因が判ります。

p_args->event == ETHER_EVENT_LINK_ON の時は、リンクアップです。Ether ボードの 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_args->event == ETHER_EVENT_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_WakeOnLANEnable(&g_ether0_ctrl);
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_args->event == ETHER_EVENT_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 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_WakeOnLANEnable()、MagicPacket 受信待ちとなります。

※MagicPacket 受信時に一瞬 WOL の LED が消灯します

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_operation.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。プッシュスイッチを 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-SW -> 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(&g_ether0_ctrl, (void *)recv_tmp_buf, &receive_size);
if (ret == FSP_SUCCESS)
{
    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, recv_tmp_buf, receive_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
    }
}
```

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_rcv_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_ether.c(抜粋)

```
//メインループ
while (1)
{
    //LINK-UP 待ち
    if (R_ETHER_LinkProcess(&g_ether0_ctrl) == FSP_SUCCESS)
    {
        if (g_operation_mode != OPERATION_RECEIVE)
        {
            //プッシュスイッチが押された場合は本ボードから MagicPacket を送出
            if ((PUSH_SW == 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
            }
        }

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

送信処理

受信処理

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

・main_ether.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)純粹な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 : 0xffx6 ★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 : MagicPaketの0xffx6 ★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_ether.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 IP/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(116 バイト)を受信

MagicPacket(1) (WOL) received.

PC から送信された MagicPacket(144 バイト)を受信

MagicPacket(2) (UDP) received.

※[参考]本プログラムでは、受信バイト数の情報を、receive_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。

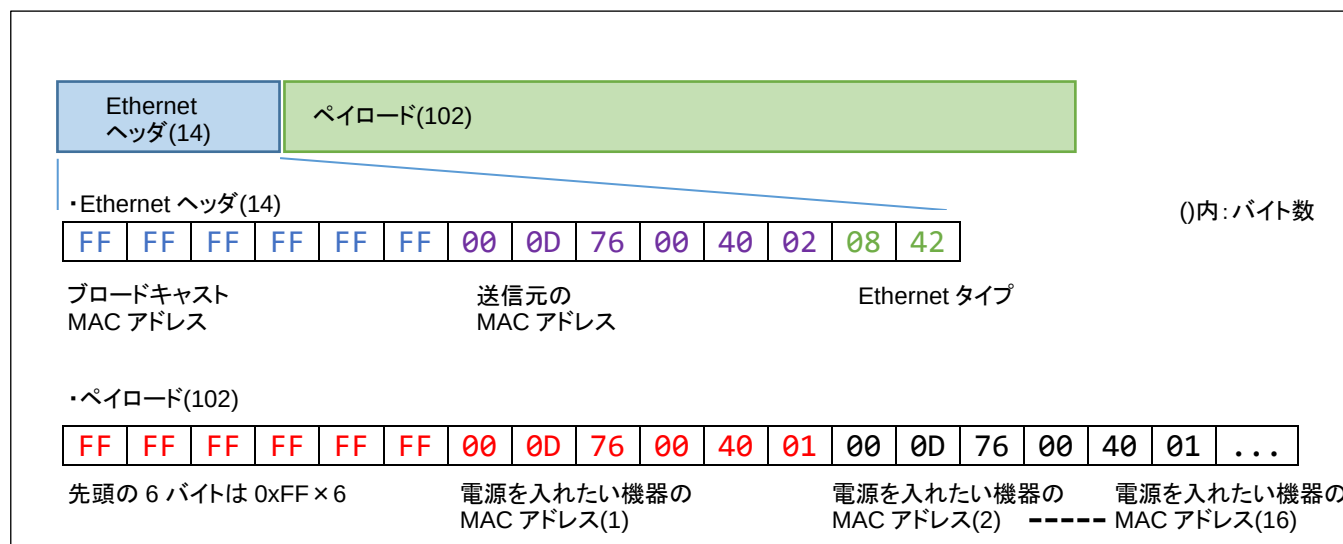
また、Ethernet の最小フレーム長のデータ(60 バイト)を受信した場合でも、アクセスしているデータは古いデータではなく、その際に受信したデータです。

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_rcv_buf)の中身を見てみると、

式	型	値
g_ether_rcv_buf	unsigned...	0x1ffe1294 <g_ether_rcv_buf>
g_ether_rcv_buf[0]	unsigned...	0xff(16 進)
g_ether_rcv_buf[1]	unsigned...	0xff(16 進)
g_ether_rcv_buf[2]	unsigned...	0xff(16 進)
g_ether_rcv_buf[3]	unsigned...	0xff(16 進)
g_ether_rcv_buf[4]	unsigned...	0xff(16 進)
g_ether_rcv_buf[5]	unsigned...	0xff(16 進)
g_ether_rcv_buf[6]	unsigned...	0x0(16 進)
g_ether_rcv_buf[7]	unsigned...	0xd(16 進)
g_ether_rcv_buf[8]	unsigned...	0x76(16 進)
g_ether_rcv_buf[9]	unsigned...	0x0(16 進)
g_ether_rcv_buf[10]	unsigned...	0x40(16 進)
g_ether_rcv_buf[11]	unsigned...	0x2(16 進)
g_ether_rcv_buf[12]	unsigned...	0x8(16 進)
g_ether_rcv_buf[13]	unsigned...	0x42(16 進)
g_ether_rcv_buf[14]	unsigned...	0xff(16 進)
g_ether_rcv_buf[15]	unsigned...	0xff(16 進)
g_ether_rcv_buf[16]	unsigned...	0xff(16 進)
g_ether_rcv_buf[17]	unsigned...	0xff(16 進)
g_ether_rcv_buf[18]	unsigned...	0xff(16 進)
g_ether_rcv_buf[19]	unsigned...	0xff(16 進)
g_ether_rcv_buf[20]	unsigned...	0x0(16 進)
g_ether_rcv_buf[21]	unsigned...	0xd(16 進)
g_ether_rcv_buf[22]	unsigned...	0x76(16 進)
g_ether_rcv_buf[23]	unsigned...	0x0(16 進)
g_ether_rcv_buf[24]	unsigned...	0x40(16 進)
g_ether_rcv_buf[25]	unsigned...	0x1(16 進)
g_ether_rcv_buf[26]	unsigned...	0x0(16 進)
g_ether_rcv_buf[27]	unsigned...	0xd(16 進)
g_ether_rcv_buf[28]	unsigned...	0x76(16 進)

のようになっており、



上記の様なデータが、g_ether_rcv_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_ether.c 内の main_ether() では、

- ・ユーザ初期化処理 (GPT タイマの起動など)
 - ・PHY チップのリセット
 - ・SCI 初期化
 - ・プッシュスイッチのチェック
- プッシュスイッチが ON の場合は、MAC アドレス=00-0D-76-00-40-02 に設定
- ・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・動作開始
 - ・メインループ

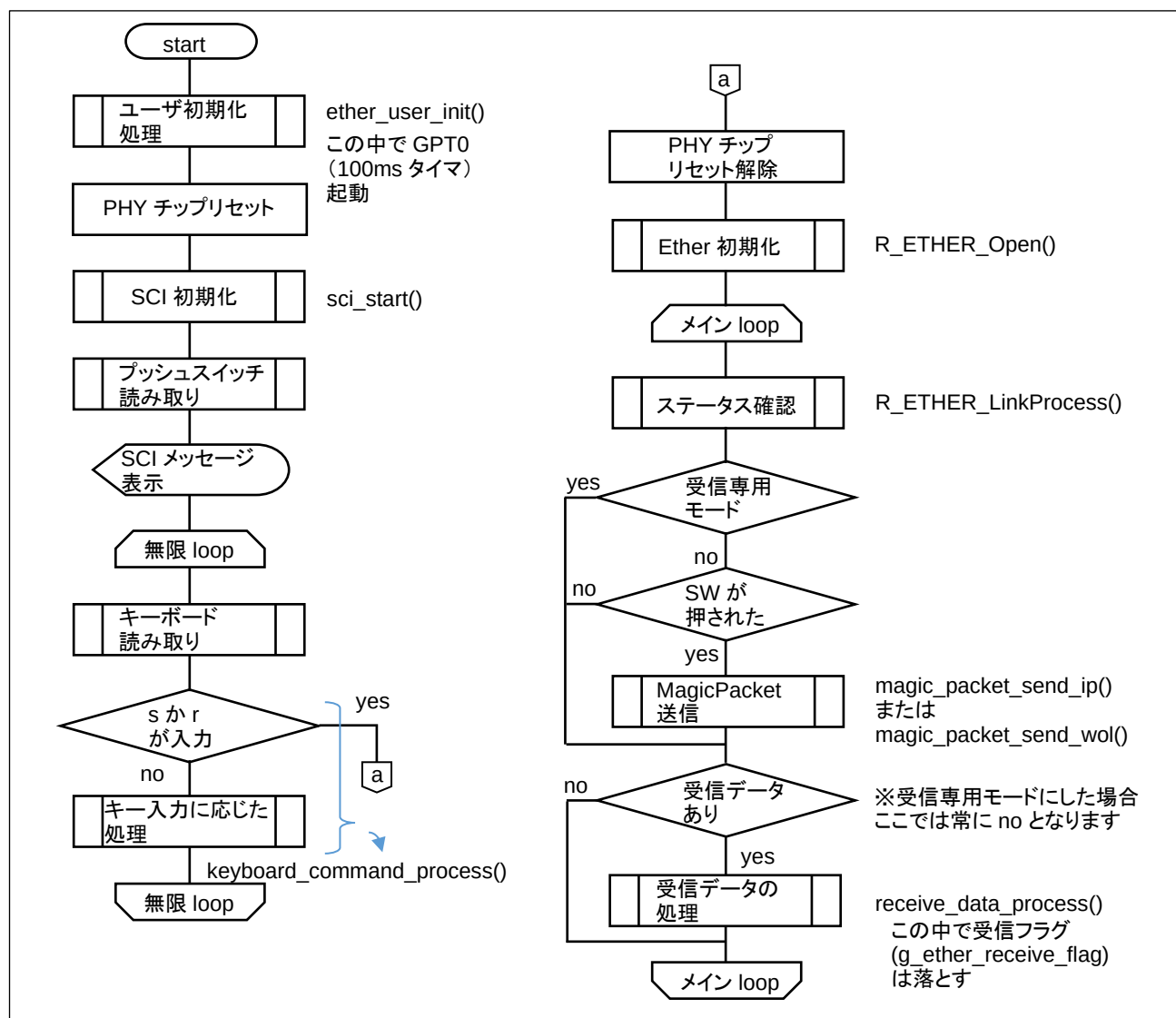
の処理を行っています。

メインループでは、

- ・API 関数のステータスチェック
- ・プッシュスイッチ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

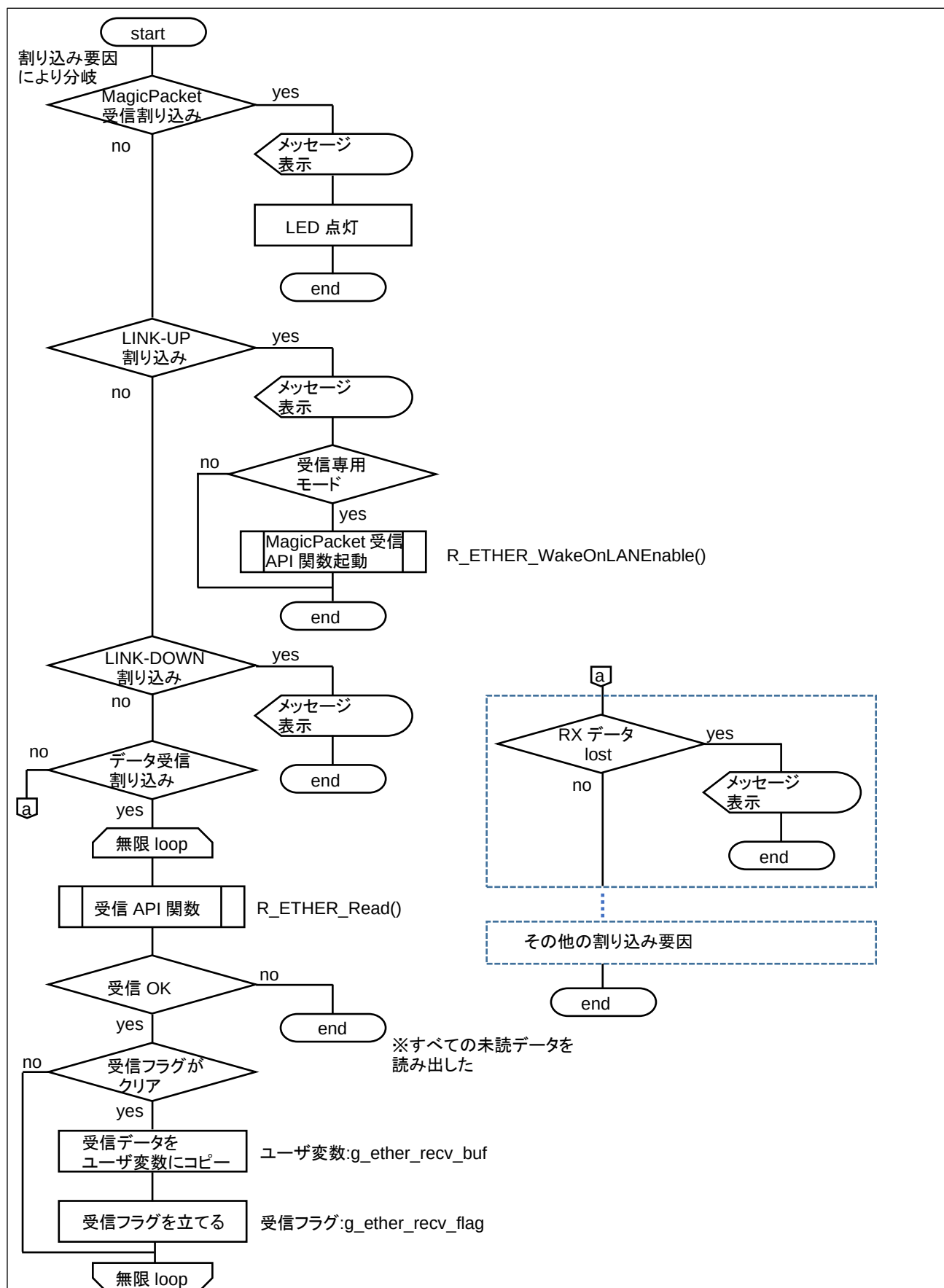
となっています。

・main_ether() (main_ether.c 内)



3.7. 割り込み関数での処理

・ether0_callback() (ether_interrupt.c 内)



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: `RAxxx_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. RAXXX_ETHER_NOAPI プロジェクト

CD 内の

SOURCE¥RAXXX_yyy¥RAXXX_ETHER_NOAPI.zip

(使用しているマイコンボードに応じたファイル)をワークスペースにインポートしてください。

本プロジェクトは、RAXXX_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 で定義されている関数)を使用していません。

r_ether スタックを使用せずに、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 スタック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、RAXXX_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 スタック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。

・FSP で追加するスタック

コンポーネント	備考
~~r_ether~~	~~Ethernet~~ ※本プロジェクトでは追加しない
r_sci_uart	SCI(UART)通信 (マイコン種によっては r_sci_b_uart)
r_gpt	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
board	board.h	ボード固有設定
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_ether.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.
RA6M2 Ether MagicPacket(WOL) sample program no API version.

COMMAND:
  m : target MAC address set
  i : target broadcast IP address set
  l : 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.
  U : send packet type is IP.(default)
---
  s : Operation start with send/receive mode
  r : Operation start with receive only mode
---

USAGE:
  PUSH-SW : 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 : l command for change
---
First m, i, l, p, S, W, U 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 (I コマンド(大文字 i)で変更)
です。(起動時に DIP-SW の PSEL/P35 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, I コマンドでアドレスの設定を行ってください。
また、W, U コマンドで送信する MagicPacket のタイプを変更可能です。

U 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-SW -> 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、ボードのプッシュスイッチを押すと 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、RAxxx_MAGIC_PACKET2 と同等です。

4.1. メイン関数での処理

main_ether.c 内の main_ether()では、

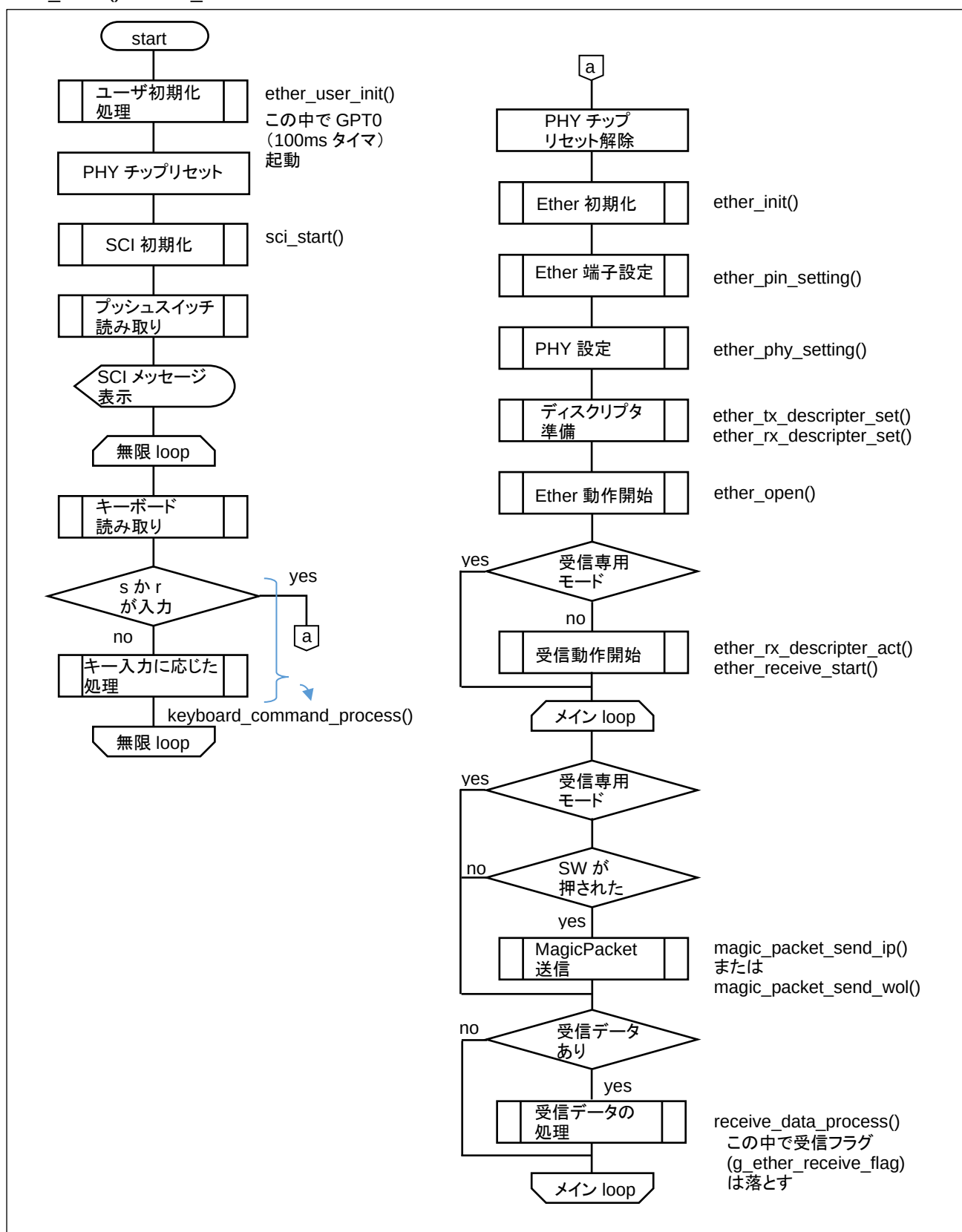
- ・ユーザ初期化処理 (GPT タイマの起動など)
 - ・PHY チップのリセット
 - ・SCI 初期化
 - ・プッシュスイッチのチェック
- プッシュスイッチが ON の場合は、MAC アドレス=00-0D-76-00-40-02 に設定
- ・起動時のメッセージ表示
 - ・キーボードからの入力待ち
 - ・PHY チップのリセット解除
 - ・Ether の初期化
 - ・端子設定
 - ・PHY 設定
 - ・ディスクリプタ準備
 - ・Ether の動作開始
 - ・受信動作開始
 - ・メインループ

の処理を行っています。

メインループでは、

- ・プッシュスイッチ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。

・main_ether() (main_ether.c 内)



4.2. Ether の初期化

•ether_init()

ETHERC0 のモジュールストップ解除

EDMAC0 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

•ether_pin_setting()

何もしない。(RX のプログラムとソース共通化のために関数を残していますが、中味は空です。)

•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_descriptor_set()

•ether_rx_descriptor_set()

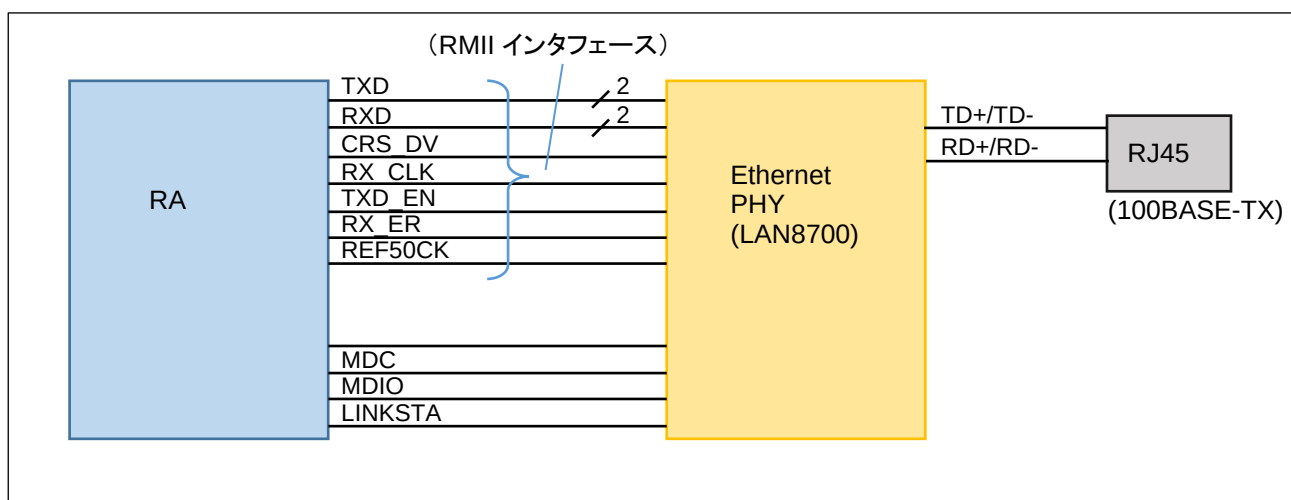
送信と受信用のディスクリプタ設定

•ether_open()

割り込みの有効化

ETHERC0 の送受信有効化

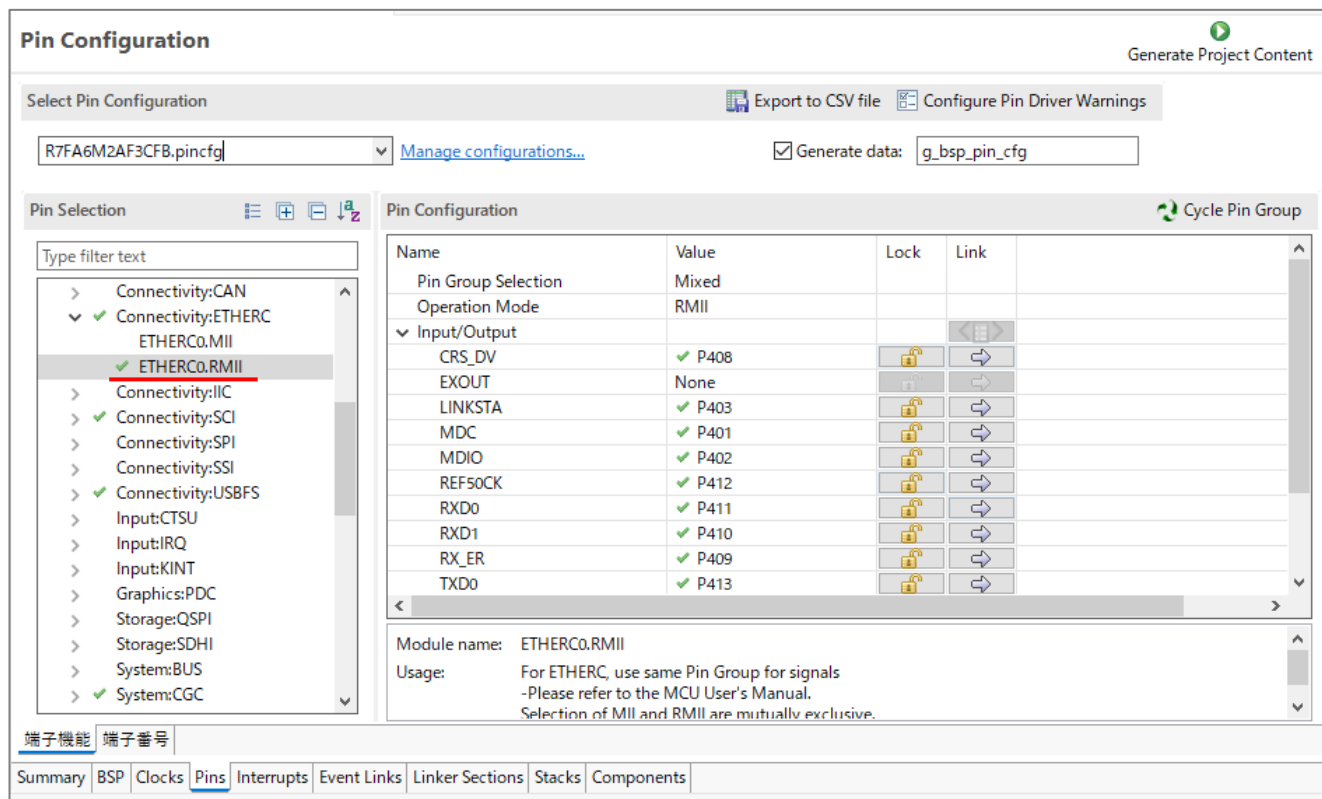
Ether の初期化はいくつかの関数で構成されています。この中で、Ether に固有な設定として PHY チップの設定とディスクリプタの設定があります。



PHY チップは、RA 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。)

・FSP での端子設定



Pin Configuration

Select Pin Configuration Export to CSV file Configure Pin Driver Warnings

R7FA6M2AF3CFB.pincfg Manage configurations... ☒ Generate data: g_bsp_pin_cfg

Pin Selection

Type filter text

- > Connectivity:CAN
- > ✓ Connectivity:ETHERC
 - ETHERC0:MII
 - ✓ **ETHERC0:RMII**
- > Connectivity:IIC
- > ✓ Connectivity:SCI
- > Connectivity:SPI
- > Connectivity:SS1
- > ✓ Connectivity:USBFS
- > Input:CTSU
- > Input:IRQ
- > Input:KINT
- > Graphics:PDC
- > Storage:QSPI
- > Storage:SDHI
- > System:BUS
- > ✓ System:CGC

Pin Configuration Cycle Pin Group

Name	Value	Lock	Link
Pin Group Selection	Mixed		
Operation Mode	RMII		
Input/Output			
CRS_DV	✓ P408		
EXOUT	None		
LINKSTA	✓ P403		
MDC	✓ P401		
MDIO	✓ P402		
REF50CK	✓ P412		
RXD0	✓ P411		
RXD1	✓ P410		
RX_ER	✓ P409		
TXD0	✓ P413		

Module name: ETHERC0:RMII

Usage: For ETHERC, use same Pin Group for signals
-Please refer to the MCU User's Manual.
Selection of MII and RMII are mutually exclusive.

端子機能 端子番号

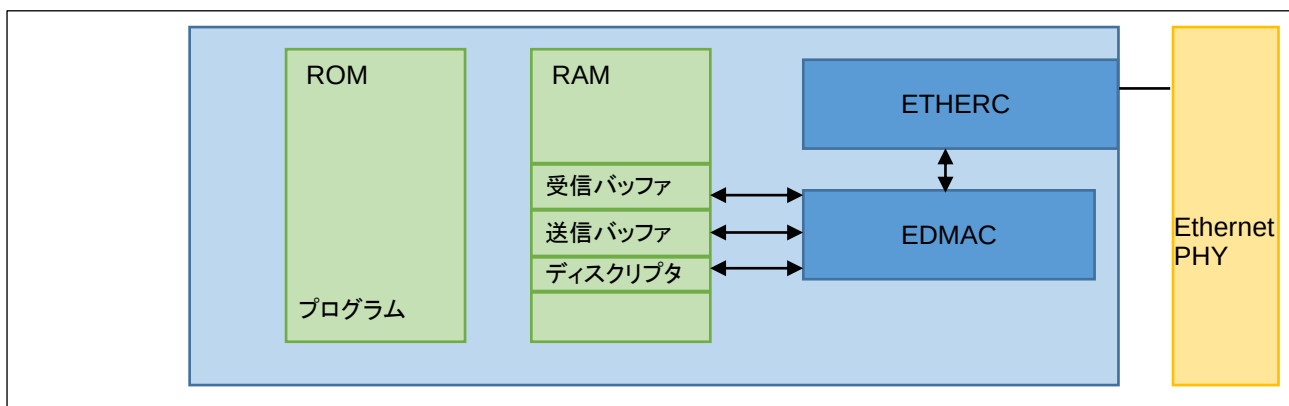
Summary BSP Clocks **Pins** Interrupts Event Links Linker Sections Stacks Components

Pins タブの、Connectivity:ETHERC – ETHERC0:RMII のところに、EthernetPHY LSI と接続する信号を設定してください。

2,3 章のプロジェクトでは、r_ether のスタックの端子設定から Pins タブの端子設定に飛んでいましたが、r_ether のスタックを追加しなくても、直接 Pins タブを選択して Ether の端子設定を行う事が可能です。

(ここで設定した内容は、マイコンの PFS(周辺機能選択)レジスタ等に反映されます。)

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 上に配置された送信バッファからデータを読み取って Ether で送信、受信したデータを RAM 上の受信バッファ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。)

送信バッファと、受信バッファ、ディスクリプタは、本プロジェクトでは以下の様に設定しています。

- ・ether_noapi.c (非 TrustZone マイコン)

```
#define MEMORY_ALIGNMENT    (32)           //32 バイト境界

//Ether のバッファメモリ 受信バッファ、送信バッファを 2kB×各 1 確保
unsigned char g_ether_rx_buffer[DESCRIPTORS][2048] BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);
unsigned char g_ether_tx_buffer[DESCRIPTORS][2048] BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);

//Ether の RX ディスクリプタ 受信ディスクリプタ 1 つ確保
unsigned char g_ether_rx_descriptor[DESCRIPTORS][16] BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);

//Ether の TX ディスクリプタ 送信ディスクリプタ 1 つ確保
unsigned char g_ether_tx_descriptor[DESCRIPTORS][16] BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);
```

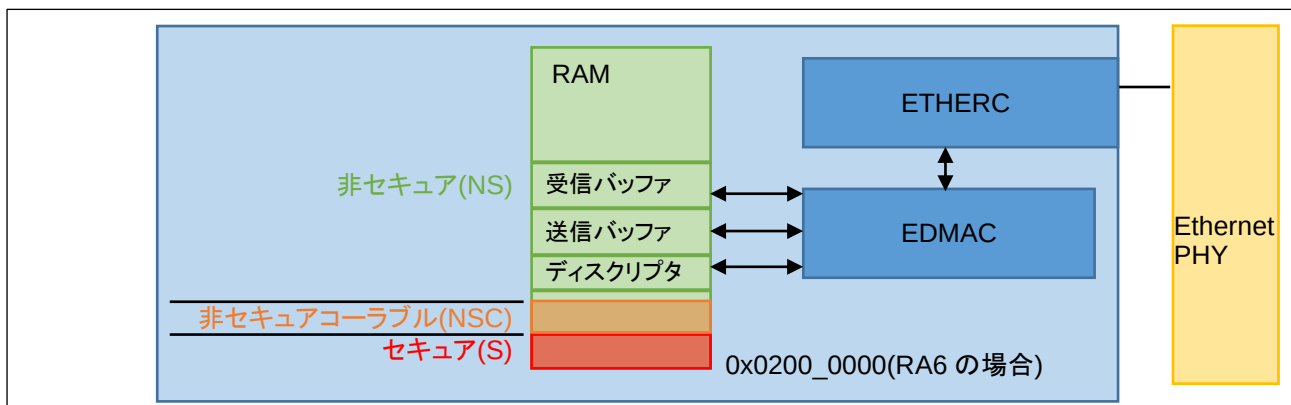
本サンプルプログラムでは、送受信バッファは 2048 × ディスクリプタ数[bytes]としています。

(送受信バッファのサイズは、1514 以上で 32 で割り切れる値で値であれば良いです。デバッグ時に見やすいという理由で、2048 に設定しています。)

(本サンプルプログラムでは、DESCRIPTORS=1 です)

32 バイトアライメント(32 で割り切れるアドレスに配置)は、明示的に指定しています。

・TrustZone マイコンでのアドレス



TrustZone 対応マイコンでは、RAM が 3 つの領域に区切られます。セキュア、非セキュアコーラブル、非セキュアの 3 領域です。RAM 以外に周辺機能にもセキュリティ区分があります。セキュアに設定された周辺機能からは、セキュア領域の読み書きが行えますが、EDMAC は常に非セキュアの周辺機能となります。

よって、EDMAC によりアクセスされる受信バッファ、送信バッファ、ディスクリプタは非セキュア領域に配置する必要があります。

・ether_noapi.c (TrustZone マイコン)

```
#define MEMORY_ALIGNMENT    (32)           //32 バイト境界

//メモリの特定のアドレスに割り付ける場合(リンカスクリプトに追加要)

//Ether のバッファメモリ 受信バッファ、送信バッファを 2kB×各 1 確保
unsigned char __attribute__(( section(".ether_buffer"))) g_ether_rx_buffer[DESCRIPTERS][2048]
BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);
unsigned char __attribute__(( section(".ether_buffer"))) g_ether_tx_buffer[DESCRIPTERS][2048]
BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);

//Ether の RX ディスクリプタ 受信ディスクリプタ 1 つ確保
unsigned char __attribute__(( section(".ether_buffer"))) g_ether_rx_descriptor[DESCRIPTERS][16]
BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);

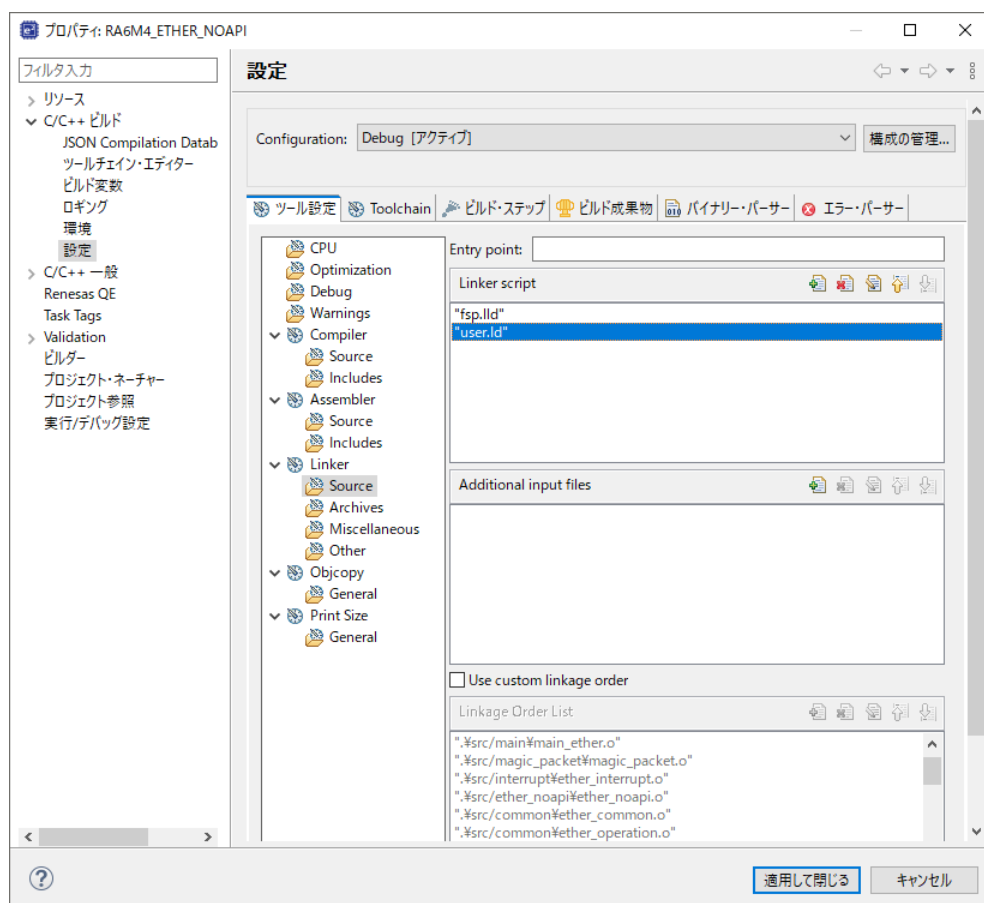
//Ether の TX ディスクリプタ 送信ディスクリプタ 1 つ確保
unsigned char __attribute__(( section(".ether_buffer"))) g_ether_tx_descriptor[DESCRIPTERS][16]
BSP_ALIGN_VARIABLE(MEMORY_ALIGNMENT);
```

TrustZone 対応マイコンでは、受信バッファ、送信バッファ、ディスクリプタを特定のセクション(.ether_buffer)に割り付ける設定としています。

・user.ld (RA6M4 の場合)

```
SECTIONS
{
    .ether_buffer(0x20038000):
    {
        KEEP(*(.ether_buffer))
    }>RAM
}
```

上記の様なリンカスクリプトを作成して、



Linker – Source に追加する事で、変数の配置アドレスを固定化しています。

RAM の非セキュアの開始アドレスに関しては、
Debug¥プロジェクト名.rpd
ファイルを見る事で確認が可能です。

• Debug¥RA6M4_ETHER_NOAPI.rpd (抜粋)

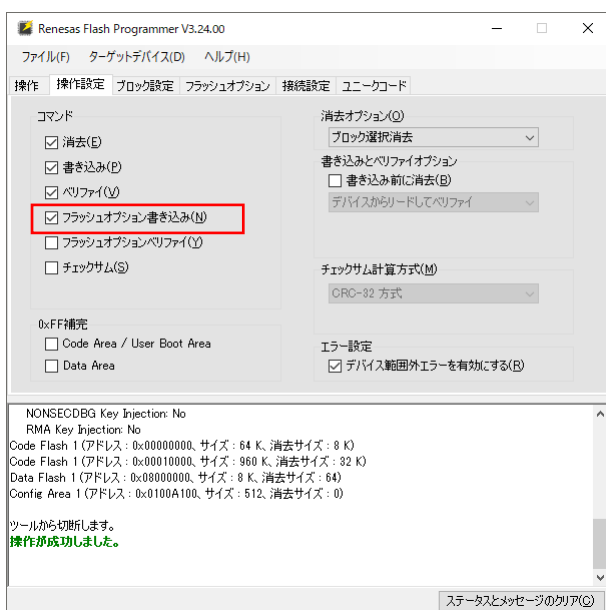
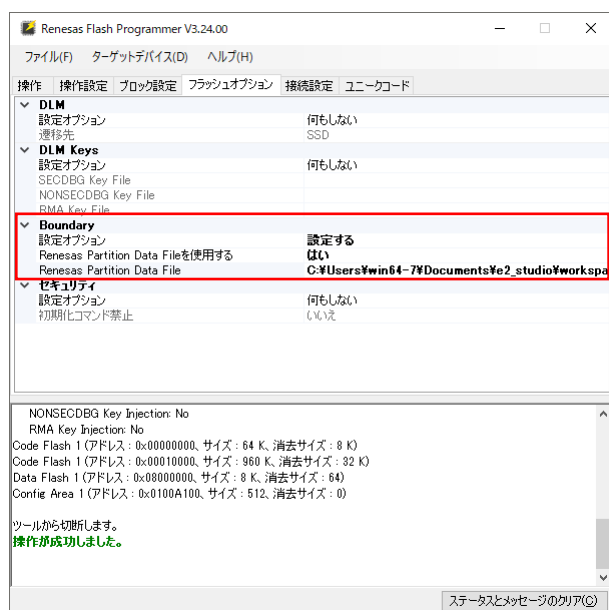
```
RAM_S_START=0x20000000
RAM_S_SIZE=0x1c00
RAM_C_START=0x20001c00
RAM_C_SIZE=0x400
```

上記の場合は、

非セキュア領域(NS)	終了アドレス	0x2003_FFFF	サイズ(残り)
	開始アドレス	0x2000_2000	
非セキュアコーラブル領域(NSC)	終了アドレス	0x2000_1FFF	サイズ=0x400
	開始アドレス	0x2000_1C00	
セキュア領域(S)	終了アドレス	0x2000_1BFF	サイズ=0x1C00
	開始アドレス	0x2000_0000	

のようになります。この場合、受信バッファ、送信バッファ、ディスクリプタは、0x2000_2000～0x2003_FFFF のどこかに配置すれば良いという事になります。(RA6M4 では、0x2003_8000～の領域に配置しています。)

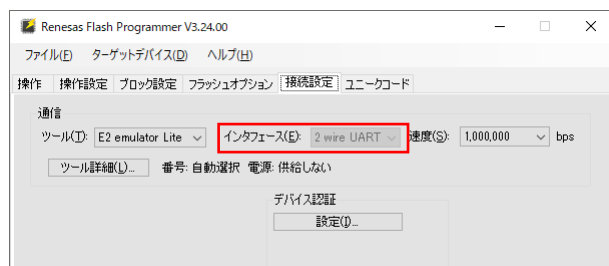
※TrustZone 対応マイコンの場合



プログラム書き込みの際、セキュア領域区分の情報がチップに書き込まれる様にしてください。

(RenesasFlashProgrammer での書き込み時は、デフォルトではセキュア領域区分の情報は書き込む設定にはなっていない。)

(RenesasFlashProgrammer で、エミュレータを使用してセキュア領域区分の情報を書き込む場合は、接続形態が 2 wire UART である必要があります。)



デバッグ接続での書き込み(ダウンロード)時は、セキュア領域区分が自動的に設定されますが、ボードの MD ジャンパが 1-2 ショート(MD と SWCLK を接続する)に設定されている必要があります。(どのジャンパが MD のジャンパかは、ボードのマニュアルを参照してください。)

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_descriptor_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_DESCRIPTOR_t *txd;

    txd = (TX_DESCRIPTOR_t *)&g_ether_tx_descriptor[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_descriptor_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_DESCRIPTOR_t *rxd;

    rxd = (RX_DESCRIPTOR_t *)&g_ether_rx_descriptor[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_descriptor_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_DESCRIPTOR_t *txd;

    txd = (TX_DESCRIPTOR_t *)&g_ether_tx_descriptor[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_descriptor_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_DESCRIPTOR_t *rxd;

    rxd = (RX_DESCRIPTOR_t *)&g_ether_rx_descriptor[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。

ーディスクリプタのビットフィールド設定ー

本サンプルプログラムでは、ディスクリプタサイズは 16 バイトに設定しています。

ディスクリプタに配置されるデータは、プログラム上エンディアンを選べるようになっていますが、ether_noapi.c 内で

```
R_ETHERC_EDMAC->EDMR_b.DE = 1; /* リトルエンディアン */
```

としています。

ディスクリプタの構造体定義で、ビットフィールドの部分は LSB からの並びです。

```
typedef struct TX_DESCRIPTOR
{
    //送信ディスクリプタ

    union {
        unsigned long LONG;
        struct {
            unsigned long TFS:26; // [b25-0] ライトバック対象    LSB
            unsigned long TWBI:1; // [b26]
            unsigned long TFE:1; // [b27] ライトバック対象    .
            unsigned long TFP:2; // [b29-28]
            unsigned long TDLE:1; // [b30]
            unsigned long TACT:1; // [b31] ライトバック対象    .
            // MSB の並び
        } BIT;
    } TD0;

    union {
        unsigned long LONG;
        struct {
            unsigned short dummy1; // [b15-0]    LSB
            unsigned short TBL; // [b31-16]    MSB の並び
        } BIT;
    } TD1;

    union {
        unsigned long LONG;
        struct {
            unsigned long TBA;
        } BIT;
    } TD2;

    unsigned long padding1;
} TX_DESCRIPTOR_t;
```

LLVM や GCC の環境で何もオプション指定等を行わないと、上記の書き方で期待した順番でデータが並ぶようになりますが、環境によってはビットの格納順が期待通りにならない事も考えられます。

ビットの並びは、エンディアンの設定とも絡んでくるので、ディスクリプタに書いた内容とプログラムの動きが対応していないと思われる場合は、ディスクリプタのデータが格納されているメモリ内のビットの並びが EDMAC のエンディアン設定と合っているか確認してください。

4.4. Ethernet フレームの送信

・ether_noapi.c

```
int ether_write(unsigned short descriptor_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
    // descriptor_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 (descriptor_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過している
        return -2;
    }

    //ディスクリプタ初期化
    ether_tx_descriptor_set(descriptor_num);

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

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

    //EDMAC にディスクリプタを読み込ませて送信
    R_ETHERC_EDMAC->EDTRR_b.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_ether.c

```
//通常の受信動作
ether_rx_descriptor_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_descriptor_act()は前出、引数 0 はディスクリプタの番号

・ether_noapi.c

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

    //戻り値
    // なし

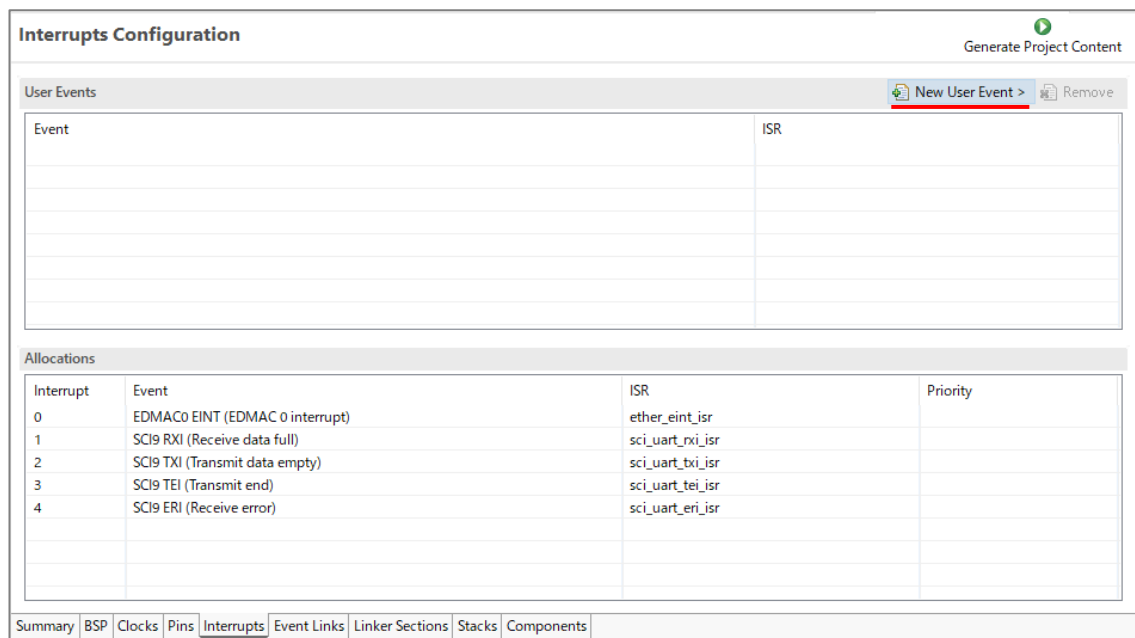
    //引数
    // なし

    R_ETHERC_EDMAC->EDRRR_b.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

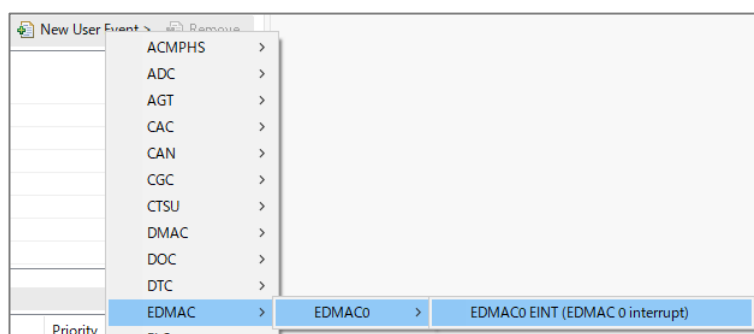
本プロジェクトでは、FSP での Ether スタック(r_ether)を使用していないため、Ether の割り込みが掛った際ユーザ関数に飛ばす設定が必要になります。



The **Interrupts Configuration** window is shown with the **Allocations** tab selected. It displays a table of interrupt allocations.

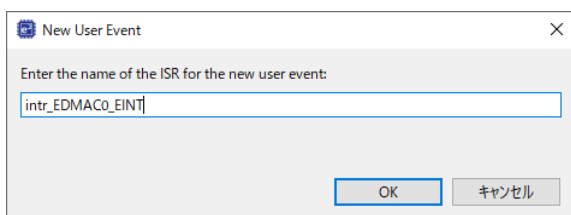
Interrupt	Event	ISR	Priority
0	EDMAC0 EINT (EDMAC 0 interrupt)	ether_eint_isr	
1	SCI9 RXI (Receive data full)	sci_uart_rx_isr	
2	SCI9 TXI (Transmit data empty)	sci_uart_tx_isr	
3	SCI9 TEI (Transmit end)	sci_uart_te_isr	
4	SCI9 ERI (Receive error)	sci_uart_eri_isr	

FSP の設定で、Interrupts タブの New User Event から、EDMAC の割り込みを追加します。



The **New User Event** dialog box is shown with the **EDMAC** category selected. The **EDMAC0 EINT (EDMAC 0 interrupt)** option is highlighted.

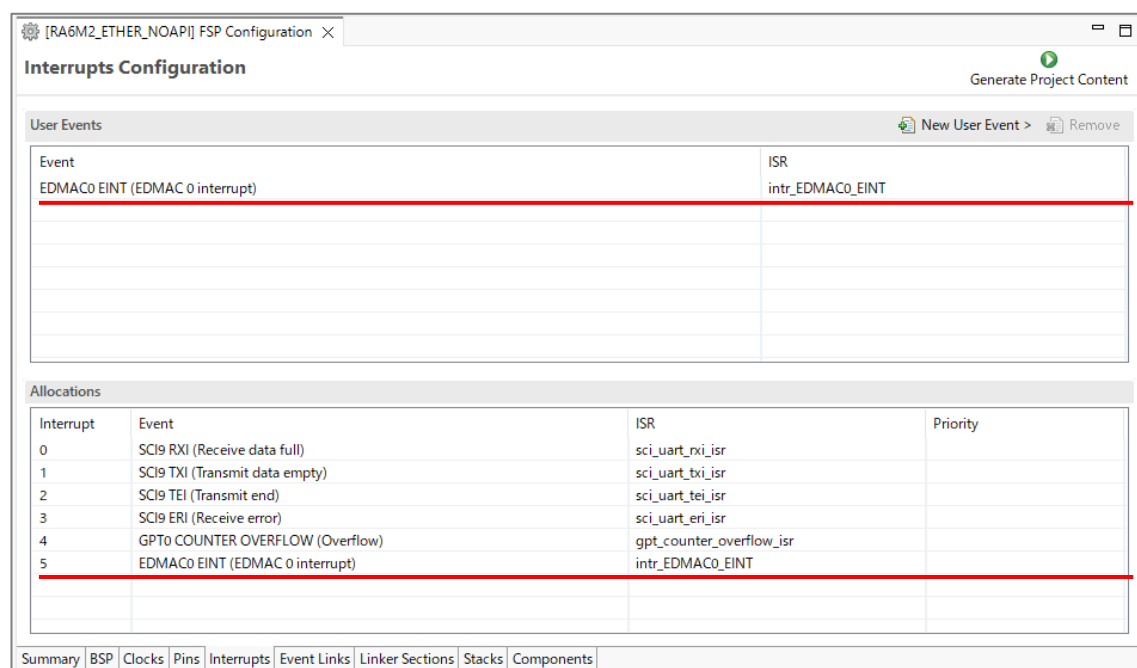
EDMAC – EDMAC0 – EDMAC0 EINT
を追加します。



The **New User Event** dialog box is shown with the name `intr_EDMAC0_EINT` entered in the text field.

`intr_EDMAC0_EINT` を入力します。

ここで入力した文字列が、そのまま関数名になります。



追加後は、User Events, Allocations の欄に追加されます。ここで入力する関数名はユーザ作成コード内の関数名と一致していれば良いです。(本プロジェクトでは、割り込みの名称 EDMAC0_EINT に intr_ を付与する形としています。)

• ether_noapi.c 内 ether_init() 抜粋

```
//割り込み設定
NVIC_SetPriority((uint32_t)VECTOR_NUMBER_EDMAC0_EINT, 1); /* 割り込み優先度 1 */
R_ICU->IELSR_b[(uint32_t)VECTOR_NUMBER_EDMAC0_EINT].IR = 0U; /* 割り込みフラグクリア */
```

ここでは、割り込み優先度の設定を行っています。(最優先の 0 に次ぐ 1 を設定)

• ether_noapi.c

```
void intr_EDMAC0_EINT(void)
{
    //ETHER0 割り込み関数

    //ETHER 割り込み
    if (R_ETHERC0->ECSR != 0)
    {
        ether_callback();
    }

    //EDMAC 割り込み
    if (R_ETHERC_EDMAC->EESR != 0)
    {
        edmac_callback();
    }

    //割り込みフラグクリア
    R_ICU->IELSR_b[(uint32_t)VECTOR_NUMBER_EDMAC0_EINT].IR = 0U;
}
```

割り込み関数の中身は、ETHERC0 モジュールの割り込みと EDMAC0 モジュールの割り込みで別々の関数に飛ばす様にしています。

・ether_interrupt.c

```
void ether_callback(void)
{
    //ETHERC 割り込み
    if ((R_ETHERC0->ECSR & ETHER_ECSR_ICD) != 0)
    {
        R_ETHERC0->ECSR_b.ICD = 1; //フラグクリア
        sci_write_str("Ether0: ICD flag detected.\n");
    }

    if ((R_ETHERC0->ECSR & ETHER_ECSR_MPD) != 0)
    {
        R_ETHERC0->ECSR_b.MPD = 1; //フラグクリア
        sci_write_str("Ether0: MagicPacket received.\n");
        LED = LED_ON; //ボードの LED を点灯させる
        sci_write_str("\n-- MagicPacket receive operation is stopped.\n");
        sci_write_str(" (This program is not reboot ETHERC Interface.)\n");
    }

    if ((R_ETHERC0->ECSR & ETHER_ECSR_LCHNG) != 0)
    {
        R_ETHERC0->ECSR_b.LCHNG = 1; //フラグクリア
        if (R_ETHERC0->PSR_b.LMON == 0)
        {
            sci_write_str("Ether0: 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
                R_ETHERC0->ECMR_b.MPDE = 1;
                sci_write_str("-> MagicPacket waiting ...\n");
            }
        }
        else
        {
            sci_write_str("Ether0: LINK-DOWN\n");
        }
    }

    if ((R_ETHERC0->ECSR & ETHER_ECSR_PSRT0) != 0)
    {
        R_ETHERC0->ECSR_b.PSRT0 = 1; //フラグクリア
        sci_write_str("Ether0: PSRT0 flag detected.\n");
    }

    if ((R_ETHERC0->ECSR & ETHER_ECSR_BFR) != 0)
    {
        R_ETHERC0->ECSR_b.BFR = 1; //フラグクリア
        sci_write_str("Ether0: BFR flag detected.\n");
    }
}
```

ETHERC0 側の処理は LINK-UP, LINK-DOWN の処理と、マイコンの機能を使用した MagicPacket の受信処理です。

ether_interrupt.c

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

    unsigned short size;
    RX_DESCRIPTOR_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 (R_ETHERC_EDMAC->EESR_b.TC != 0)
    {
        R_ETHERC_EDMAC->EESR_b.TC = 1;    //フラグクリア
    }

    //送信ディスクリプタ枯渇
    if (R_ETHERC_EDMAC->EESR_b.TDE != 0)
    {
        R_ETHERC_EDMAC->EESR_b.TDE = 1;    //フラグクリア
    }

    //フレーム受信
    if (R_ETHERC_EDMAC->EESR_b.FR != 0)
    {
        R_ETHERC_EDMAC->EESR_b.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

        //受信バッファフルの状態では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
        if ((g_ether_rcv_buf_write_index - g_ether_rcv_buf_read_index) != ETHER_BUFFER_SIZE)
        {
            rxd = (RX_DESCRIPTOR_t *)&g_ether_rx_descriptor[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_rcv_buf[g_ether_rcv_buf_write_index & (ETHER_BUFFER_SIZE - 1)][0],
                &g_ether_rx_buffer[0][0], size);    //受信データを g_ether_rcv_buf にコピー
            g_ether_rcv_buf_write_index++;    //書き込み済みインデックスを進める
        }

        ether_rx_descriptor_set(0);    //受信ディスクリプタ設定
        ether_rx_descriptor_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
    }

    //受信ディスクリプタ枯渇
    if (R_ETHERC_EDMAC->EESR_b.RDE != 0)
    {
        R_ETHERC_EDMAC->EESR_b.RDE = 1;    //フラグクリア

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

    //受信フレームカウンタオーバーフロー
    if (R_ETHERC_EDMAC->EESR_b.RFCOF != 0)
    {
        R_ETHERC_EDMAC->EESR_b.RFCOF = 1;    //フラグクリア

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

    //受信 FIFO オーバフロー
    if (R_ETHERC_EDMAC->EESR_b.RFOF != 0)
    {
        R_ETHERC_EDMAC->EESR_b.RFOF = 1;    //フラグクリア
    }
}
```

EDMAC0 側は、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_rcv_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。)

EDMAC が取り扱う受信バッファ(g_ether_rx_buffer)を、ユーザが用意した受信バッファ g_ether_rcv_buf にコピーするという形は、RAXXX_MAGIC_PACKET2 と同じですが、本プロジェクトでは g_ether_rcv_buf を複数個持たせています。これは、データを連続で受信した場合の対策です。

•ether_common.h

```
//受信バッファサイズ
#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
#define ETHER_BUFFER_SIZE 2 //バッファを2つ設ける(2のべき乗の値に限る) ※最適化OFF(-O0)の場合は2だとデータを取りこぼす(かも)
```

デフォルトでは、ユーザ側で用意する受信バッファ数は ETHER_BUFFER_SIZE = 2 です。

割り込み関数内で g_ether_rcv_buf[0], g_ether_rcv_buf[1](現状バッファは 2 つ)に格納されたデータはメイン関数内で読み出されて処理されます。メイン関数内での読み出しの処理より、データ受信の間隔が短い場合は処理の取りこぼしが起こりますが、一時的に連続でデータを受信する場合などは、バッファ数を増やす事で処理の取りこぼしを防ぐことが出来るかもしれません。

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-SW -> 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 2 つ用意していますが、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てくる)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ー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ーフロー割り込み」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ーフローは、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。

※ROE interrupt と RFCOF interrupt は 1 以上の時に表示されます(表示されない場合は、どちらの割り込みも生じていないという事です)

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。

※データの取りこぼしに関しては、手元の環境では RA6M2@120MHz の場合

GCC 取りこぼしなし

LLVM 10,000 データに対して 1 回程度(0.01%)程度取りこぼす

という感じでした。

(RA6M4/RA6M5(200MHz), RA8(360MHz/400MHz)では、LLVM 環境でも取りこぼしは観測されませんでした。)

(ちなみに、RX では受信パッファを 1 つしか使用しなくても取りこぼしはなし。)

GCC, LLVM の toolchain と RX でのデータコピーに関する話は、4.9 節を参照してください。

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:

・なにもしません

※RX との互換性のために残している関数で端子設定は FSP 側の設定で完結しているため、本関数は空です

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_descriptor_set

概要: 送信ディスクリプタ設定

宣言: void ether_tx_descriptor_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_descriptor_set

概要: 受信ディスクリプタ設定

宣言: void ether_rx_descriptor_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_descriptor_act

概要: 送信ディスクリプタ有効化

宣言: void ether_tx_descriptor_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_descriptor_act

概要: 受信ディスクリプタ有効化

宣言: void ether_rx_descriptor_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 descriptor_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 descriptor_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

4.9. memcpy ライブラリ関数に関して[参考]

Ether のデータを取り扱う際に、EDMAC でのデータの格納先からユーザ側で用意したバッファにデータをコピーする処理を行っています。この際に本キットでは、memcpy 関数を使用しています。

buf から buf2 に 1500 バイトのデータコピーを行う際、一般的にはループで処理するより、ライブラリ関数を使用した方が高速に処理できるためです。(…と思ったのですが。4.9.5 では、ループで処理した場合も記載しています。)

ー C 言語での処理例 (ループ処理) ー

```
for (int i=0; i<1500; i++)
{
    buf2[i] = buf[i];
}
```

ー ライブラリ関数を使った場合 ー

```
memcpy(buf2, buf, 1500);
```

memcpy 関数で 1500 バイトのデータコピーを行った場合の処理速度を実測してみました。

4.9.1. Cortex-M4(RA6M2)の場合

FSP6.6.0 では、デフォルトの toolchain が LLVM。追加で GCC を選択すると、GCC も使用可能です。(GCC も FSP6.6.0 でサポートされている toolchain です。)

コンパイルオプション	LLVM	GCC
-Os(サイズ優先)	113us (*1)	75us
-O2(通常最適化)	100us	75us (*1)
-O3(とにかく高速)	100us	75us
-O0(最適化なし)	100us	75us

※RA6M2@120MHz での実行時間、太字は FSP6.6.0 のデフォルトオプション

(*1)RA の FSP6.6.0 の環境では、LLVM では-Os(サイズ優先)の最適化がデフォルトです。GCC では-O2 がデフォルトです。

なお、ライブラリ関数は(通常は)バイナリで提供されているので、基本的には実行時間が最適化オプションに左右されないはずですが。

LLVM と GCC でどのような処理となっているか軽く中身を見えます。

C でのソースコードとしては、

```
R_PORT1->PODR_b.PODR6 = 1;
memcpy(buf2, buf, 1500);
R_PORT1->PODR_b.PODR6 = 0;
```

の様なものです。(memcpy の前後にポートを叩いているのは、オシロで時間を測るためです)

toolchain が LLVM の場合の実行内容です。

－LLVM -Os(サイズ最適化)－

```

30          R_PORT1-&gtPODR_b.PODR6 = 1;
0000046c: orr.w  r0, r0, #64      @ 0x40
00000470: strh   r0, [r4, #0]
31      memcpy(buf2, buf, 1500);
00000472: movw   r0, #2584        @ 0xa18
00000476: movt   r0, #8190        @ 0x1ffe
0000047a: movt   r1, #8190        @ 0x1ffe
0000047e: movw   r2, #1500        @ 0x5dc
00000482: bl     0xb16 <memcpy>
32          R_PORT1-&gtPODR_b.PODR6 = 0;
00000486: ldrh   r0, [r4, #0]
00000488: bic.w  r0, r0, #64      @ 0x40
0000048c: strh   r0, [r4, #0]
0000048e: nop

```

－LLVM -O2(通常最適化)－

```

30          R_PORT1-&gtPODR_b.PODR6 = 1;
0000046c: orr.w  r0, r0, #64      @ 0x40
00000470: strh   r0, [r4, #0]
31      memcpy(buf2, buf, 1500);
00000472: movw   r0, #2584        @ 0xa18
00000476: movt   r0, #8190        @ 0x1ffe
0000047a: movt   r1, #8190        @ 0x1ffe
0000047e: movw   r2, #1500        @ 0x5dc
00000482: bl     0xb94 <memcpy>
32          R_PORT1-&gtPODR_b.PODR6 = 0;
00000486: ldrh   r0, [r4, #0]
00000488: bic.w  r0, r0, #64      @ 0x40
0000048c: strh   r0, [r4, #0]
0000048e: nop

```

－LLVM memcpy の中身－

```

memcpy:
00000b94: push   {r4, lr}
00000b96: movs   r3, #0
00000b98: cmp    r2, r3
00000b9a: it     eq
00000b9c: popeq  {r4, pc}
00000b9e: ldrb   r4, [r1, r3]
00000ba0: strb   r4, [r0, r3]
00000ba2: adds   r3, #1
00000ba4: b.n    0xb98 <memcpy+4>

```

最適化オプションに拘わらず、memcpy を呼んでいる部分の処理に違いはなく、memcpy の中身も同一です。

唯一違いがあるのが、memcpy のバイナリの配置アドレスです。-Os の場合は、0xb16 番地に配置されており、-O2 の場合は、0xb94 番地です。0xb94(=2964, 2964/4=741)は 4 で割り切れるアドレスですが、0xb16(=2838, 2838/4=709.5)は 4 では割り切れないアドレスです。(多分ですが)この違いにより、-O2(100us)に対し、-Os(113us)と 10%以上の速度低下があります。

—GCC -O2(通常最適化)—

```

30      R_PORT1->PODR_b.PODR6 = 1;
000004f2: orr.w   r3, r3, #64      @ 0x40
000004f6: strh    r3, [r4, #32]
31      memcpy(buf2, buf, 1500);
000004f8: movw    r2, #1500        @ 0x5dc
000004fc: bl      0xbdc <memcpy>
32      R_PORT1->PODR_b.PODR6 = 0;
00000500: ldrh    r3, [r4, #32]
00000502: bfi     r3, r5, #6, #1
00000506: strh    r3, [r4, #32]

```

—GCC memcpy の中身—

```

memcpy:
00000bdc: add     r2, r1
00000bde: cmp     r1, r2
00000be0: add.w   r3, r0, #4294967295
00000be4: bne.n   0xbe8 <memcpy+12>
00000be6: bx      lr
00000be8: push    {r4, lr}
00000bea: ldrb.w  r4, [r1], #1
00000bee: strb.w  r4, [r3, #1]!
00000bf2: cmp     r1, r2
00000bf4: bne.n   0xbea <memcpy+14>
00000bf6: pop     {r4, pc}

```

GCC での memcpy の部分は上記です。

GCC 環境では、-Os オプションで最適化を行った際、memcpy()の前後に nop 命令を入れて配置アドレスをずらすとしましたが、(試した限りでは)memcpy は 4 で割り切れるアドレスに配置され、実行時間のブレはありませんでした。

memcpy 関数の中身に関してですが、

行	LLVM	GCC
1	push {r4, lr}	add r2, r1
2	movs r3, #0	cmp r1, r2
3	cmp r2, r3	add.w r3, r0, #4294967295
4	it eq	bne.n 0xbe8<memcpy+12>
5	popcq {r4, pc}	bx lr
6	ldrb r4, [r1, r3]	push {r4, lr}
7	strb r4, [r0, r3]	ldrb.w r4, [r1], #1
8	adds r3, #1	strb.w r4, [r3, #1]!
9	b.n 0xb98<memcpy+4>	cmp r1, r2
10		bne.n 0xbea<memcpy+14>
11		pop {r4, pc}

LLVM の方が短く簡潔に見えます。処理内容も素直です。それに対し、GCC の方は、ちょっとトリッキーな感じがします。

※プログラム内の b.n 0xb98 等は相対ジャンプ命令なので、固有价值ではない

でも実行結果は、LLVM の 100us(4 バイトアライメントアドレス配置の場合)に対し、GCC では 75us と 25%も高速です。GCC の memcpy の中身は ARM の命令セットを熟知しているエンジニアがゴリゴリに書いた感じがします。

FSP6.6.0 からは、GCC はオプション(FSP6.5.0 までは、FSP6.5.0+e2studio のインストーラに GCC が含まれていたが、FSP6.6.0 からは別途ダウンロードする形に変更)です。今後も、RA の FSP では LLVM がメインの toolchain になるのではないかと思います。

この結果を見る限りでは、現時点では LLVM に対する GCC の優位性もあるのではないかと思います。toolchain としては、LLVM、GCC、サードパーティーの toolchain を使用する選択肢もあります。

(本キットでは、FSP6.6.0 のデフォルトである LLVM を使用していますが、RA では複数の toolchain が選べます。)

4.9.2. Cortex-M33(RA6M4)の場合

コンパイルオプション	LLVM	GCC
-Os(サイズ優先)	68us (*1)	68us
-O2(通常最適化)	68us (*2)	68us
-O3(とにかく高速)	68us	68us
-O0(最適化なし)	68us	68us

※RA6M4@200MHz での実行時間、太字は FSP6.6.0 のデフォルトオプション

—LLVM memcpy の中身—

```

memcpy:
0000077e: push    {r4, lr}
00000780: movs    r3, #0
00000782: cmp     r2, r3
00000784: beq.n   0x78e <memcpy+16>
00000786: ldrb    r4, [r1, r3]
00000788: strb    r4, [r0, r3]
0000078a: adds    r3, #1
0000078c: b.n     0x782 <memcpy+4>
0000078e: pop     {r4, pc}

```

—GCC memcpy の中身—

```

memcpy:
00000854: add     r2, r1
00000856: subs    r3, r0, #1
00000858: cmp     r1, r2
0000085a: bne.n   0x85e <memcpy+10>
0000085c: bx      lr
0000085e: push    {r4, lr}
00000860: ldrb.w  r4, [r1], #1
00000864: cmp     r1, r2
00000866: strb.w  r4, [r3, #1]!
0000086a: bne.n   0x860 <memcpy+12>
0000086c: pop     {r4, pc}
0000086e: movs    r0, r0

```

(*1)4 で割り切れないアドレスに memcpy が配置

(*2)4 で割り切れるアドレスに memcpy が配置

→実行速度は変わらない

Cortex-M33 の場合は、LLVM と GCC で実行時間はほぼ同じでした。LLVM では、-Os の場合は memcpy が 4 で割り切れないアドレスに配置されましたが、実行時間は変わりませんでした。

(インストラクションキャッシュの構成や動作により、Cortex-M33 では命令配置アドレスの 4 バイトアライメントが実行速度におけるポイントではなくなっているものと思われます。)

memcpy の中身は、Cortex-M4 と Cortex-M33 で同一ではありませんが似たような感じです。

Cortex-M4 では GCC の方が memcpy を高速に処理していましたが、Cortex-M33 では LLVM と GCC の差がなくなっています。

4.9.3. Cortex-M85(RA8M1)の場合

memcpy の実行速度に関しては

Cortex-M4 GCC < LLVM

Cortex-M33 GCC = LLVM

といった結果でしたが、Cortex-M85 だと条件によっては逆転しました。

コンパイルオプション	LLVM	GCC
-Os(サイズ優先)	68us (*1)	60us (*3)
-O2(通常最適化)	15.4us (*2)	60us (*3)
-O3(とにかく高速)	15.4us (*2)	60us (*3)
-O0(最適化なし)	15.4us (*2)	60us (*3)

※RA8M1@400MHz での実行時間、太字は FSP6.6.0 のデフォルトオプション

—(*1)LLVM -Os—

—(*2)LLVM -Os 以外—

—(*3)GCC—

```

memcpy:
020008fe: push    {r4, lr}
02000900: movs    r3, #0
02000902: cbz     r2, 0x200090e <memcpy+16>
02000904: ldrb    r4, [r1, r3]
02000906: subs    r2, #1
02000908: strb    r4, [r0, r3]
0200090a: adds    r3, #1
0200090c: b.n     0x2000902 <memcpy+4>
0200090e: pop     {r4, pc}

```

```

memcpy:
02000b94: mov     r12, r0
02000b96: orr.w   r3, r1, r0
02000b9a: ands.w  r3, r3, #3
02000b9e: bne.n   0x2000c34 <memcpy+160>
02000ba0: subs    r2, #64 @ 0x40
02000ba2: bcc.n   0x2000bec <memcpy+88>
02000ba4: ldr     r3, [r1, #0]
02000ba6: str     r3, [r0, #0]
02000ba8: ldr     r3, [r1, #4]
02000baa: str     r3, [r0, #4]
02000bac: ldr     r3, [r1, #8]
02000bae: str     r3, [r0, #8]
02000bb0: ldr     r3, [r1, #12]
02000bb2: str     r3, [r0, #12]
02000bb4: ldr     r3, [r1, #16]
02000bb6: str     r3, [r0, #16]
02000bb8: ldr     r3, [r1, #20]
02000bba: str     r3, [r0, #20]
02000bbc: ldr     r3, [r1, #24]
02000bbe: str     r3, [r0, #24]
02000bc0: ldr     r3, [r1, #28]
02000bc2: str     r3, [r0, #28]
02000bc4: ldr     r3, [r1, #32]
02000bc6: str     r3, [r0, #32]
02000bc8: ldr     r3, [r1, #36] @ 0x24
02000bca: str     r3, [r0, #36] @ 0x24
02000bcc: ldr     r3, [r1, #40] @ 0x28
02000bce: str     r3, [r0, #40] @ 0x28
02000bd0: ldr     r3, [r1, #44] @ 0x2c
02000bd2: str     r3, [r0, #44] @ 0x2c
02000bd4: ldr     r3, [r1, #48] @ 0x30
02000bd6: str     r3, [r0, #48] @ 0x30
02000bd8: ldr     r3, [r1, #52] @ 0x34
02000bda: str     r3, [r0, #52] @ 0x34
02000bdc: ldr     r3, [r1, #56] @ 0x38
02000bde: str     r3, [r0, #56] @ 0x38
02000be0: ldr     r3, [r1, #60] @ 0x3c
(後略)

```

```

memcpy:
02000fbc: add     r2, r1
02000fbe: subs    r3, r0, #1
02000fc0: cmp     r1, r2
02000fc2: bne.n   0x2000fc6 <memcpy+10>
02000fc4: bx      lr
02000fc6: push    {r4, lr}
02000fc8: ldrb.w  r4, [r1, #1]
02000fcc: cmp     r1, r2
02000fce: strb.w  r4, [r3, #1]!
02000fd2: bne.n   0x2000fc8 <memcpy+12>
02000fd4: pop     {r4, pc}
02000fd6: movs    r0, r0

```

LLVM の -Os の時のコードは、シンプルにコピーするコード。GCC の場合は、従来通りちょっとトリッキーなコードです。(*1)と(*3)の比較では若干 GCC の方が高速。LLVM の -Os オプションの実行時間は、RA6M4@200MHz とほぼ同一です。

LLVM の -Os 以外(-O0 の最適化 OFF 時も)は、メモリコピーをひたすら書き下すコードになっています。(さすがに、1500 バイトのコピーを全てフラットにしている訳ではなく、64 バイトコピーしてのループにはなっている。)

LLVM(-Os)と GCC のコードではバイト単位でのコピー命令ですが、LLVM の -Os 以外では、4 バイト単位でのコピー命令を使っています(4 バイト×16 回=64 バイトが 1 ループ)。

GCC では、最適化オプションに限らず既定の(=バイナリの)memcpy を呼び出す形です。LLVM は、最適化オプションにより、呼び出されるコードが異なっています。

LLVM の-Os 以外では、4 バイト幅でコピーしていく事で、GCC の約 1/4 の実行時間となっています。

Cortex-M85 では、FSP のデフォルトオプション(太字同士の比較)では GCC の方が高速ですが、速度重視のコンパイルオプションを選択すると、圧倒的に LLVM の方が高速になります。

※RA8M1@400MHz 動作の場合 ICLK=200MHz で動作させていますが、SRAM(ICLK 同期)は 1 ウェイトとなるので、100MHz(10ns)動作となり、単純に SRAM のアクセス速度という観点では RA6M4@200MHz と RA8M1@400MHz は変わりません

Arm コアの世代としては、古い順で Cortex-M4, Cortex-M33, Cortex-M85 です。

RA6M2, RA6M3	Cortex-M4	Armv7E-M
RA6M4, RA6M5	Cortex-M33	Armv8-M
RA8	Cortex-M85	Armv8.1-M

memcpy の実行時間という観点では、古い世代では GCC の方が高速で、新しい世代では LLVM の方が高速という傾向があるのかとも思います。

(本キットでは、どの世代の MPU でも toolchain としては LLVM を使っています。)

(Cortex-M4 時代では有効であったトリッキーなコードが後の世代では優位性がなくなっています。CPU アーキテクチャの進歩により、命令セットを駆使してアセンブラでコードを書くメリットが消えてゆくのでしょうか。)

(LLVM では、ライブラリ関数を呼ぶ=決まったバイナリをリンクする訳ではなく、最適化オプションに依存する。)

4.9.4. RX(RXv2)の場合[参考]

[参考 RX における memcpy]

ffe00722	fbee2ac008		PORTA.PODR.BIT.B1 = 1;
ffe00727	f0e1		MOV.L #08002AH, R14
ffe00729	fb3adc05		BSET #1, [R14]
ffe0072d	fb2204140000		MOV.L #05DCH, R3
ffe00733	fb12041a0000		memcpy(buf2, buf, 1500);
ffe00739	7f8f		MOV.L #00001404H, R2
ffe0073b	fbee2ac008		MOV.L #00001A04H, R1
ffe00740	f0e9		SMOVF
			PORTA.PODR.BIT.B1 = 0;
			MOV.L #08002AH, R14
			BCLR #1, [R14]

RX で memcpy を実行した場合ですが、memcpy のライブラリ関数にジャンプしている部分がありません。

RX(CC-RX)	命令の内容
MOV.L #05DCH, R3	R3 レジスタに 1500(コピーするバイト数)をセット
MOV.L #00001404H, R2	R2 レジスタにコピー元の buf の先頭アドレスをセット
MOV.L #00001A04H, R1	R1 レジスタにコピー先の buf2 の先頭アドレスをセット
SMOVF	順方向ストリング転送命令

1500 バイトのデータコピーが、SMOVF という 1 命令で完結します。RA(ARM)の場合は(LLVM 側の処理例)、

- ・1500 回をカウントするループ変数の比較
- ・データをメモリからレジスタに転送
- ・レジスタからメモリにデータを転送
- ・ループ変数の加算
- ・先頭に戻る

上記を 1500 ループする形です(7 命令×1500 で 10,500 命令を実行します)。これらの処理がたった 1 つの命令で行えます。(SMOVF の 1 命令だからといって、数クロックで処理が終わる訳ではありませんが、ソフトではなくハードで処理してしまうのは強いです。)

実行時間としては、

コンパイルオプション	
-optimize=2 -size(サイズ最適化)[デフォルト]	7.5us
-optimize=2 -speed(速度最適化)	7.5us
-optimize=0(最適化なし)	7.5us

※RX65N@120MHz での実行時間

CPU 命令の処理なので実行時間は最適化オプションには依存しません。

クロック周波数が同一の RA マイコン(RA6M2@120MHz)の GCC(75us)より 10 倍高速です。400MHz 動作の RA8M1 での最大限の最適化(15.4us)より 2 倍以上高速です。

memcpy の処理に関しては、専用命令を持つ RX が圧勝です。

4.9.5. ライブラリ関数を使わない場合

ちなみに、C 言語で memcpy に相当する処理を書いて実行した場合の実行速度に関しても測定してみました。

```
void memcpyu(unsigned char *dst, unsigned char *src, int size)
{
    int i;
    for (i=0; i<size; i++)
    {
        *dst++ = *src++;
    }
}
```

上記のコードで、1500 バイトコピーを行った場合です。

・Cortex-M4

コンパイルオプション	LLVM(22.1.0)	GCC(13.2.1.arm-13-7)
-Os(FSP での LLVM のデフォルト)	113 (*1) (*2)	113 (*3)
-O2(FSP での GCC のデフォルト)	100 (*1) (*2)	75 (*1)
-O3(速度重視)	100 (*1) (*2)	75 (*1)
-O0(最適化オフ)	312	250

単位[us], RA6M2@120MHz での実行時間

(*1)C 言語で書いたソースですが、ライブラリ関数である memcpy を呼ぶコードに最適化されました
(GCC での-O2, -O3、LLVM での-O0 以外...上表の 5/8 の条件で memcpy への置き換え)

GCC, LLVM 共にユーザが書き下したコードの処理が memcpy と同じである事を見抜いてライブラリ関数を呼ぶ最適化がなされる事に驚きました。

(*2)LLVM で同じライブラリ関数呼び出しで 100us と 113us に分かれるのは、memcpy の配置番地が 4 で割られるか否かの相違です(前出と同じ理由)

(*3)LLVM の memcpy ライブラリ関数のコードとほぼ同一のコードが生成されました

GCC での-Os が LLVM の memcpy とほぼ同じコードです。(LLVM の memcpy は GCC のコンパイラが吐き出す様なシンプルな処理となっていると言えるかと思います。)

・Cortex-M33

コンパイルオプション	LLVM(22.1.0)	GCC(13.2.1.arm-13-7)
-Os(FSP での LLVM のデフォルト)	68 (*1)	83
-O2(FSP での GCC のデフォルト)	68 (*1)	60 (*1)
-O3(速度重視)	68 (*1)	60 (*1)
-O0(最適化オフ)	334	330

単位[us], RA6M4@200MHz での実行時間

(*1)結果的に memcpy が呼ばれたので memcpy を使った場合と同じ実行時間です

・Cortex-M85

コンパイルオプション	LLVM(22.1.0)	GCC(13.2.1.arm-13-7)
-Os(FSP での LLVM のデフォルト)	68 (*1)	68
-O2(FSP での GCC のデフォルト)	15.4 (*1)	60 (*1)
-O3(速度重視)	15.4 (*1)	60 (*1)
-O0(最適化オフ)	309	322

単位[us], RA8M1@400MHz での実行時間

(*1)結果的に memcpy が呼ばれたので memcpy を使った場合と同じ実行時間です

RX で同様の実行時間を測定すると、以下の様になりました。

コンパイルオプション	CC-RX(3.08)
-optimize=2 -size (通常最適化, サイズ優先 [CS+のデフォルト])	62.5 (*1)
-optimize=2 -speed (通常最適化, 速度優先)	43.5 (*2)
-optimize=Max -size (最適化最大, サイズ優先)	62.5
-optimize=Max -speed (最適化最大, 速度優先)	31.4 (*3)
-optimize=0 (最適化オフ)	288

単位[us], RX65N@120MHz での実行時間

(*1)通常最適化、サイズ優先の場合の生成コード

```

_memcpy
void memcpy(unsigned char *dst, unsigned char *src, int size)
MOV.L #0H,R14
{
    int i;

    for (i=0; i<size; i++)
    CMP
    BGE.B
    {
        *dst++ = *src++;
        MOVU.B [R2+],R15
        ADD #1H,R14
        MOV.B R15,[R1+]
        BRA.B _memcpy+2H
    }
}
RTS

```

(*2)通常最適化、速度優先の場合の生成コード

```

_memcpy
void memcpy(unsigned char *dst, unsigned char *src, int size)
CMP #0H,R3
PUSH.L R6
{
    int i;

    for (i=0; i<size; i++)
    ADD #0600H,R14,R1
    MOV.L #02EEH,R5
    ADD #2H,R1
    SUB #1H,R5
    BNE.B _main+31H
    BLE.B _memcpy+30H
    CMP #1H,R3
    BLE.B _memcpy+32H
    MOV.L R3,R14
    BCLR #0,R14
    SHAR #1,R3,R15
    MOV.L R1,R4
    MOV.L R2,R5
    ADD #2H,R4
    SUB #1H,R15
    BNE.B _memcpy+15H
    CMP R3,R14
    BEQ.B _memcpy+30H
    RTSD #04H,R6-R6
    SUB #1H,R3
    BNE.B _memcpy+32H
    {
        *dst++ = *src++;
        MOVU.B [R14+],R4
        MOVU.B [R14+],R2
        MOV.B R4,[R1]
        MOVU.B R2,01H[R1]
        MOVU.B [R5+],R6
        MOV.B R6,[R4]
        MOVU.B [R5+],R6
        MOVU.B R6,01H[R4]
        MOVU.B [R14,R2],R15
        MOV.B R15,[R14,R1]
        MOVU.B [R2+],R14
        MOVU.B R14,[R1+]
    }
}
RTSD #04H,R6-R6

```

(*3)最適化最大、速度優先の場合の生成コード

```

_memcpy
void memcpy(unsigned char *dst, unsigned char *src, int size)
CMP #0H,R3
PUSH.L R6
{
    int i;

    for (i=0; i<size; i++)
    ADD #0600H,R14,R1
    MOV.L #0FAH,R15
    ADD #8H,R1
    SUB #1H,R15
    BNE.B _main+43H
    BLE.B _memcpy+54H
    CMP #7H,R3
    MOV.L #0H,R14
    BLE.B _memcpy+44H
    MOV.L R3,R14
    AND #08H,R14
    SHAR #3,R3,R5
    MOV.L R1,R4
    MOV.L R2,R15
    ADD #8H,R4
    SUB #1H,R5
    BNE.B _memcpy+18H
    CMP R3,R14
    BEQ.B _memcpy+54H
    SUB R14,R3
    ADD R14,R1
    ADD R2,R14
    SUB #1H,R3
    BNE.B _memcpy+50H
    {
        *dst++ = *src++;
        MOVU.B [R14+],R5
        MOVU.B [R14+],R2
        MOVU.B [R14+],R3
        MOVU.B R2,01H[R1]
        MOVU.B [R14+],R2
        MOVU.B R3,02H[R1]
        MOVU.B R2,03H[R1]
        MOVU.B [R14+],R3
        MOVU.B [R14+],R2
        MOVU.B R5,[R1]
        MOVU.B R3,04H[R1]
        MOVU.B R2,05H[R1]
        MOVU.B [R15+],R6
        MOVU.B R6,[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,01H[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,02H[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,03H[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,04H[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,05H[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,06H[R4]
        MOVU.B [R15+],R6
        MOVU.B R6,07H[R4]
        MOVU.B [R14+],R15
        MOVU.B R15,[R1+]
    }
}
RTSD #04H,R6-R6

```

(*1)に対して(*3)はやたらプログラムコードが長くなっていますが、速度はほぼ半分になっています。

RX では、memcpy(=SMOVF 命令)に置き換える最適化は行われませんでした。(*1)のシンプルなコードでも (ARM の命令コードを駆使した)GCC のライブラリ関数より高速です。

(RX では、最適化 Max でも 4 バイト単位の転送命令は使っていません。4 バイト単位の転送命令に置き換えた場合は、8.2us となったので SMOVF 命令を使わなくても、RA8M1@400MHz よりも高速に処理できるポテンシャルは持っていると思います。)

(そもそも、SMOVF 命令を使った場合は、RA8M1@400MHz より 2 倍以上高速ですが…。)

(ARM と RX で得意な処理、不得意な処理はあると思いますが、メモリ間のコピーに関しては専用命令を使わなくても RX の方が得意な様に思えます。)

5. RAXxx_PING プロジェクト

CD 内の

SOURCE¥RAXxx_yyy¥RAXxx_PING.zip

(使用しているマイコンボードに応じたファイル)をワークスペースにインポートしてください。

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、Ether のスタックを使用する形です。

・FSP で追加するスタック

コンポーネント	備考
r_ether	Ethernet
r_sci_uart	SCI(UART)通信 (マイコン種によっては r_sci_b_uart)
r_gpt	タイマ(100ms 毎)

(→RAXxx_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
board	board.h	ボード固有設定
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_ether.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.
RA6M2 Ether ping sample program.

COMMAND:
i : target IP address set
l : 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:
PUSH-SW : 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 : l 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, l, 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 (l 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。(起動時にプッシュスイッチが 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-SW -> send ping to target
Ether0: LINK-UP
```

プッシュスイッチを押すと、設定した IP アドレスに対して ping を打ちます。
 (2 回以上プッシュスイッチを押してください、2 回目以降で ping を送信)

5.1. メイン関数での処理

main_ether.c 内の main_ether() では、

- ・ユーザの初期化処理
- ・PHY チップのリセット
- ・SCI 初期化
- ・プッシュスイッチのチェック
- プッシュスイッチが 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
- ・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
- ・メインループ

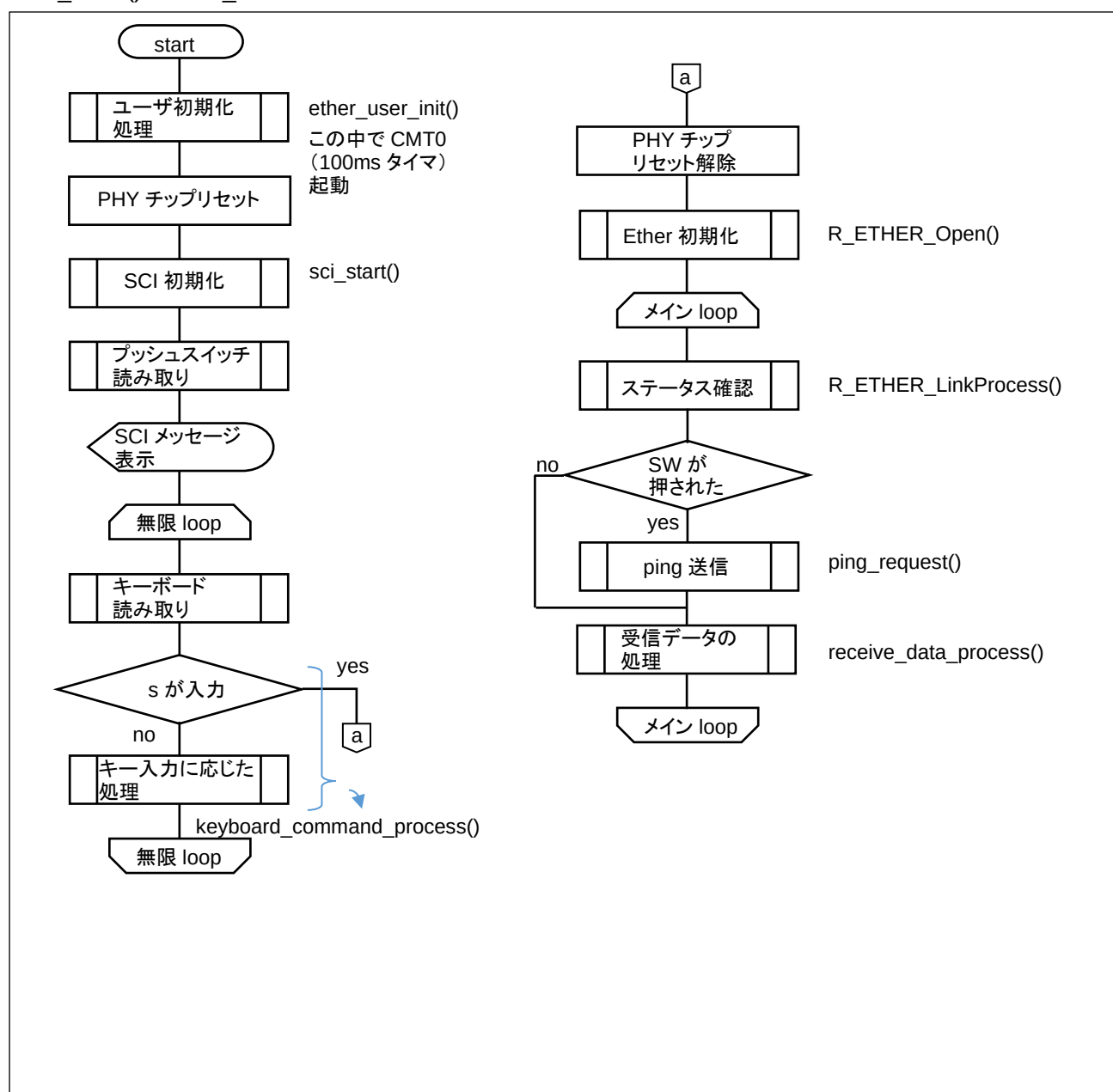
の処理を行っています。

メインループでは、

- ・API 関数のステータスチェック
- ・プッシュスイッチ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。

・main_ether() (main_ether.c 内)



プログラムの全体的な流れは RAxxx_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、

プッシュスイッチで 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, ether0_callback()内

```
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
{
    //Zero-copy モードを使わない

    ret = R_ETHER_Read(&g_ether0_ctrl, (void *)recv_tmp_buf, &receive_size);    //一旦ローカルの変数に
    コピーして受信
    if (ret == FSP_SUCCESS)
    {
        buffer_address = recv_tmp_buf;

        ether_rcv_data_enqueue(&buffer_address, (unsigned short)receive_size); //ユーザ側で用意した受
        信バッファにコピー(2回コピーが生じる)
    }
    else if (ret == FSP_ERR_ETHER_ERROR_NO_DATA)
    {
        break;
    }
    else
    {
        sci_write_str("Error: Receive fail.%n");
        break;
    }
}
```

受信の処理を、RAXxx_ETHER_NOAPI から多少変えています。RAXxx_ETHER_NOAPI では、ユーザ側で用意した変数の書き込み、読み出しを管理していましたが、これらの処理を ether_data_enqueue()関数で処理する様にしています。

行っている処理はほぼ同じで、単に関数に追い出しているだけです。

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_rcv_data_enqueue(unsigned char **src_address, unsigned short 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

    //補足
    //  ETHER_RECV_BUF_OVERRIDE 未定義時はバッファ溢れの際に引数として与えられたデータは格納せず
    ETHER_RET_FIFO_OVERFLOW を返す
    //  ETHER_RECV_BUF_OVERRIDE 定義時はバッファ溢れの際古いデータを上書きして引数として与えられたデータは格納,
    ETHER_RET_SUCESS を返す

    unsigned long index;
    bool primask;

    if ((g_ether_rcv_buf_write_index - g_ether_rcv_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
        primask = __get_PRIMASK() ? 0 : 1;
        __disable_irq();
        g_ether_rcv_buf_read_index++;
        if (primask) __enable_irq();

        g_ether_rcv_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_rcv_buf_write_index & (ETHER_RECV_BUF_SIZE - 1);
    memcpy(g_ether_rcv_buf[index], *src_address, size);

    //受信サイズを保存しておく
    g_ether_rcv_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
    primask = __get_PRIMASK() ? 0 : 1;
    __disable_irq();
    g_ether_rcv_buf_write_index++; //write_index に書き込みを行うのはこの部分のみ
    if (primask) __enable_irq();

    return ETHER_SUCESS;
}
```

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_rcv_buf[ETHER_RECV_BUF_SIZE][ETHER_MAX_PACKET_SIZE] 受信データを保存
g_ether_rcv_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

RAXxx_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。)

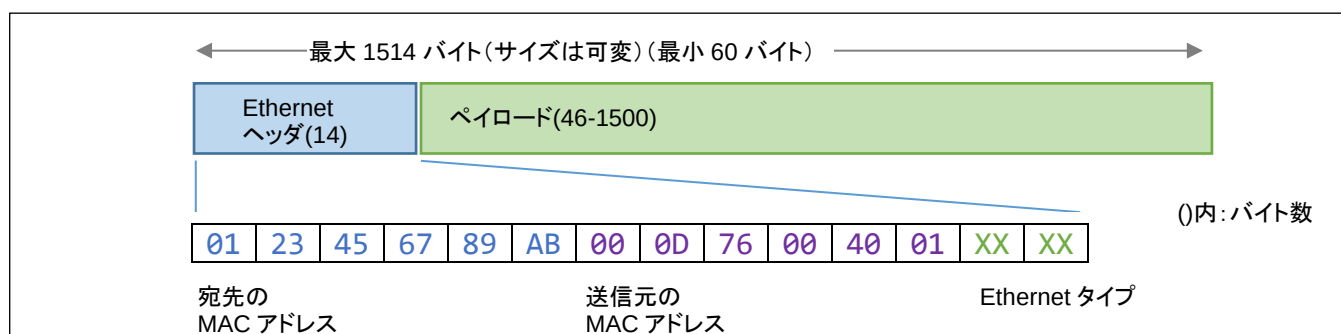
MagicPacket を扱うプロジェクトでは、MAC アドレスでの通信を取り扱っていましたが、本プロジェクトでは IP アドレスを使用した通信を行います。

(※RAXxx_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を説明します。)

5.4. ARP リクエスト

PC のコマンドプロンプトから ping を実行する場合は、
ping 192.168.0.81
の様に輸入しますが、ここで ping の宛先は、「IP アドレス」になります。

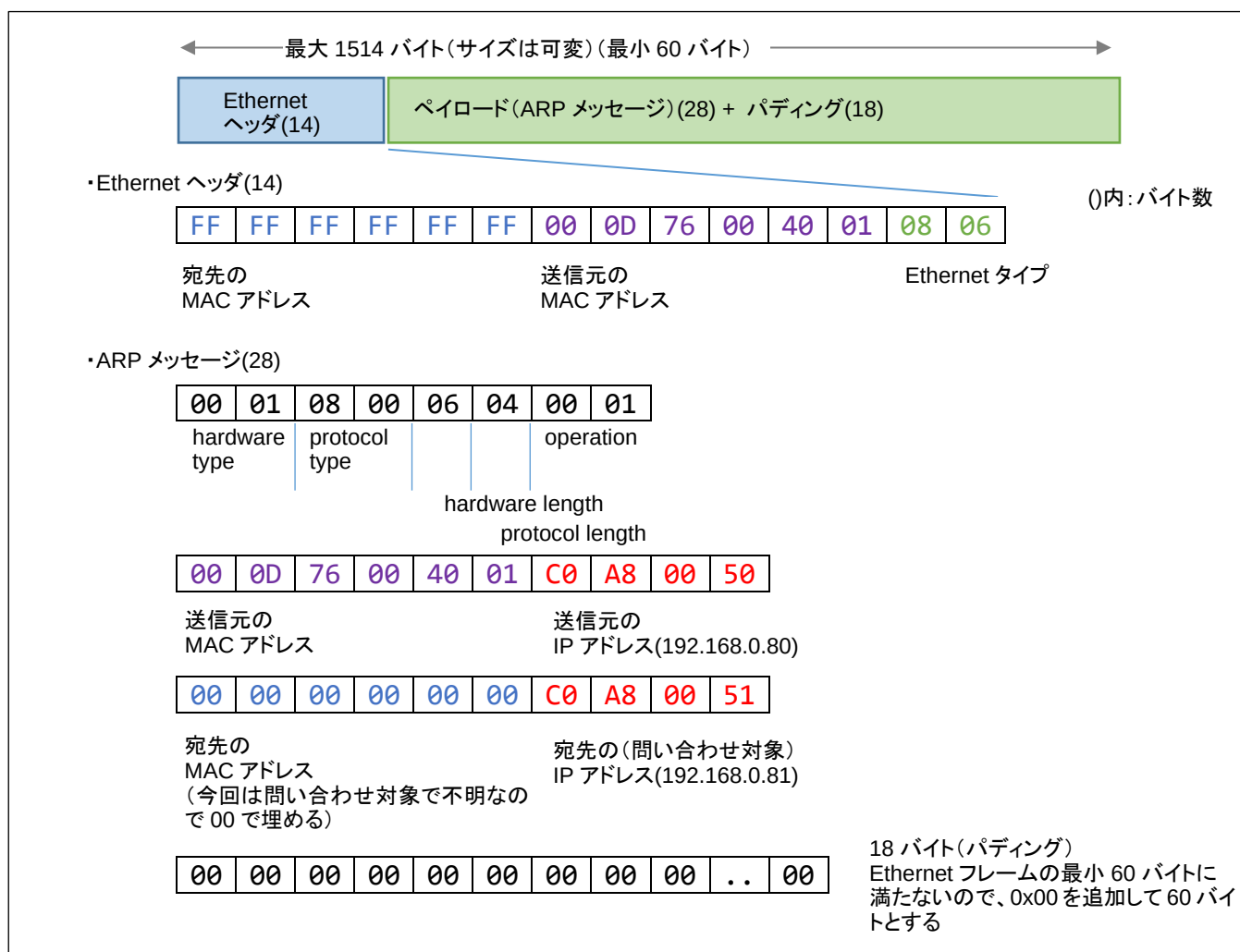
Ethernet フレームの構造としては、



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。

ーARP リクエストのパケットー



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-SW -> send ping to target
Ether0: LINK-UP
```

初回プッシュスイッチを押した際、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

    fsp_err_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]; //(<u>Ethernet</u> 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]; //(<u>ARP</u> 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_ether0_ctrl, (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;
```

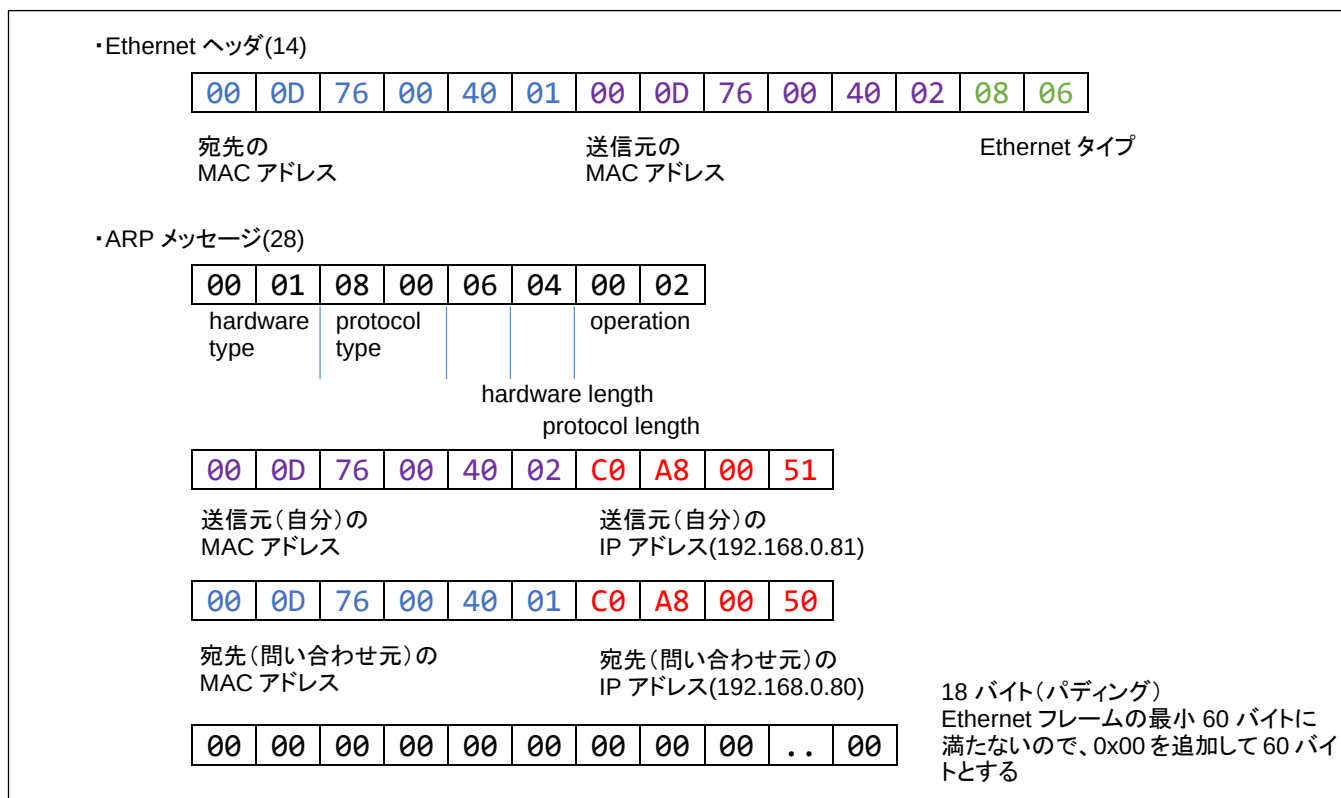
構造体のメンバに値を代入した後で、バイトストリームに変換して送信。

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。

ーARP リプライのパケットー



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

    fsp_err_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

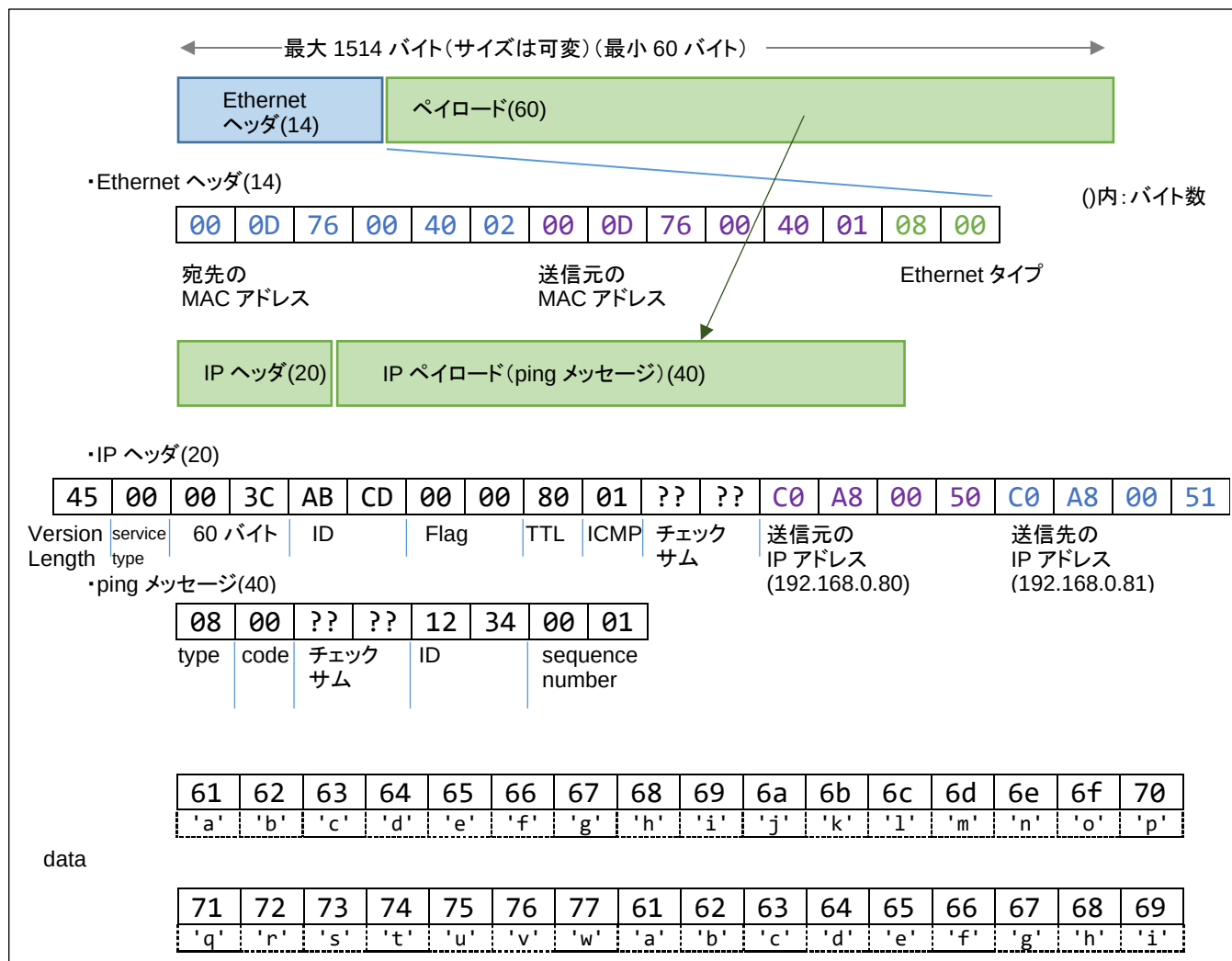
    //API 関数を使いデータを送信
    ret = R_ETHER_Write(&g_ether0_ctrl, (void *)send_data, sizeof(send_data));
    (後略)
```

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。

ーping リクエストのパケットー



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 回目のプッシュスイッチで ARP リクエスト送信。2 回目以降のプッシュスイッチで 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;

    ETHERNET_HEADER ethernet_packet;
    IP_HEADER ip_packet;
    PING_MESSAGE ping_packet;

    fsp_err_t ret;

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

    static unsigned short seq_no = 1;

    sci_write_str("%n");

    //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;
    }

    //Ethernet 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_ether_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;
```

本サンプルプログラムでは、IP ヘッダの ID は 0xABCD に固定しています
(通常はランダムな値とします)

0x80=128, ルータを 128 回通過するとパケットは捨てられます

・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(ID)
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:abcdefghijklmnopqrstuvwabcedefghi (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
ip_packet.ip_checksum = ip_header_sum(&ip_packet);

//ping メッセージのチェックサム計算
ping_packet.ping_checksum = ping_message_sum(&ping_packet);

//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_ether0_ctrl, (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((3)で送るデータに(2)で受信した内容が含まれる)

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。ping リクエストを送信するのは、ARP リプライを受信した後です。)

•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、 $20+4 \times n$ (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、あたためて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、あたためて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

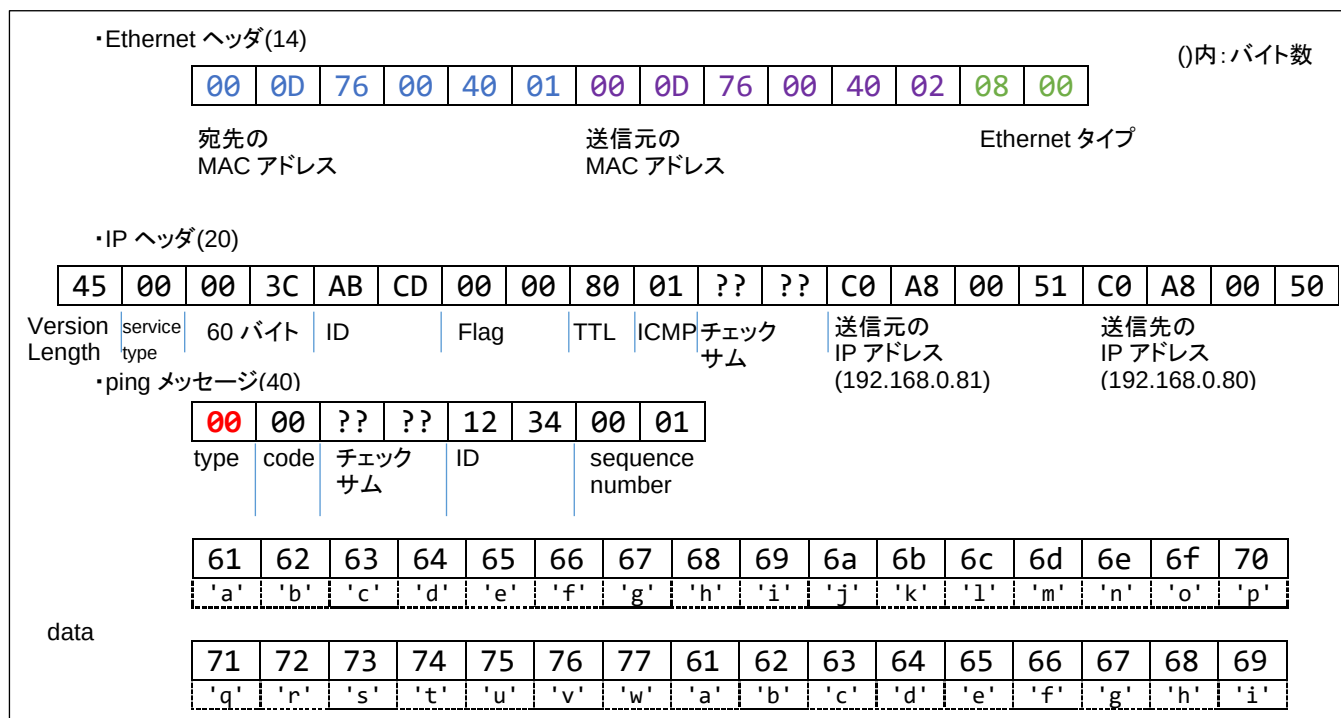
```
//送信IPヘッダのチェックサム計算
sum = ip_header_sum(&ip_packet); //IPヘッダの1の補数和
sum = (~sum) & 0xffff; //NOT演算反転
ip_packet.ip_checksum = (unsigned short)sum;
```

→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。

ーping リプライのパケットー



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;

    ETHERNET_HEADER ethernet_packet;
    IP_HEADER ip_packet;
    PING_MESSAGE ping_packet;

    fsp_err_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
    ip_packet.ip_checksum = ip_header_sum(&ip_packet);

    //ping メッセージのチェックサム計算
    ping_packet.ping_checksum = ping_message_sum(&ping_packet);

    //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_ether0_ctrl, (void *)send_data, sizeof(send_data));
}

(後略)
```

ping_packet.ping_id
ping_packet.ping_sequence_number
は、引数として与えられたデータ(ping リプライ
受信データ)をそのまま送信する形です

ping_reply()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、割り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_ether.c, main_ether()内

```
while (1)
{
    //LINK-UP 待ち
    if (R_ETHER_LinkProcess(&g_ether0_ctrl) == FSP_SUCCESS)
    {
        //プッシュスイッチが押された場合は本ボードから ping を送出
        if ((PUSH_SW == 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
- ・プッシュスイッチが押された場合は ping リクエストの送信
- ・受信データの処理(receive_data_process())

です。

・main_ether.c

```
void receive_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_ether_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_ether.c, receice_data_process() (続き)

```
switch (ethernet_packet.eth_ether_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				operation	
				hardware length			
				protocol length			

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_ether.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);

一致    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
    {
        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	

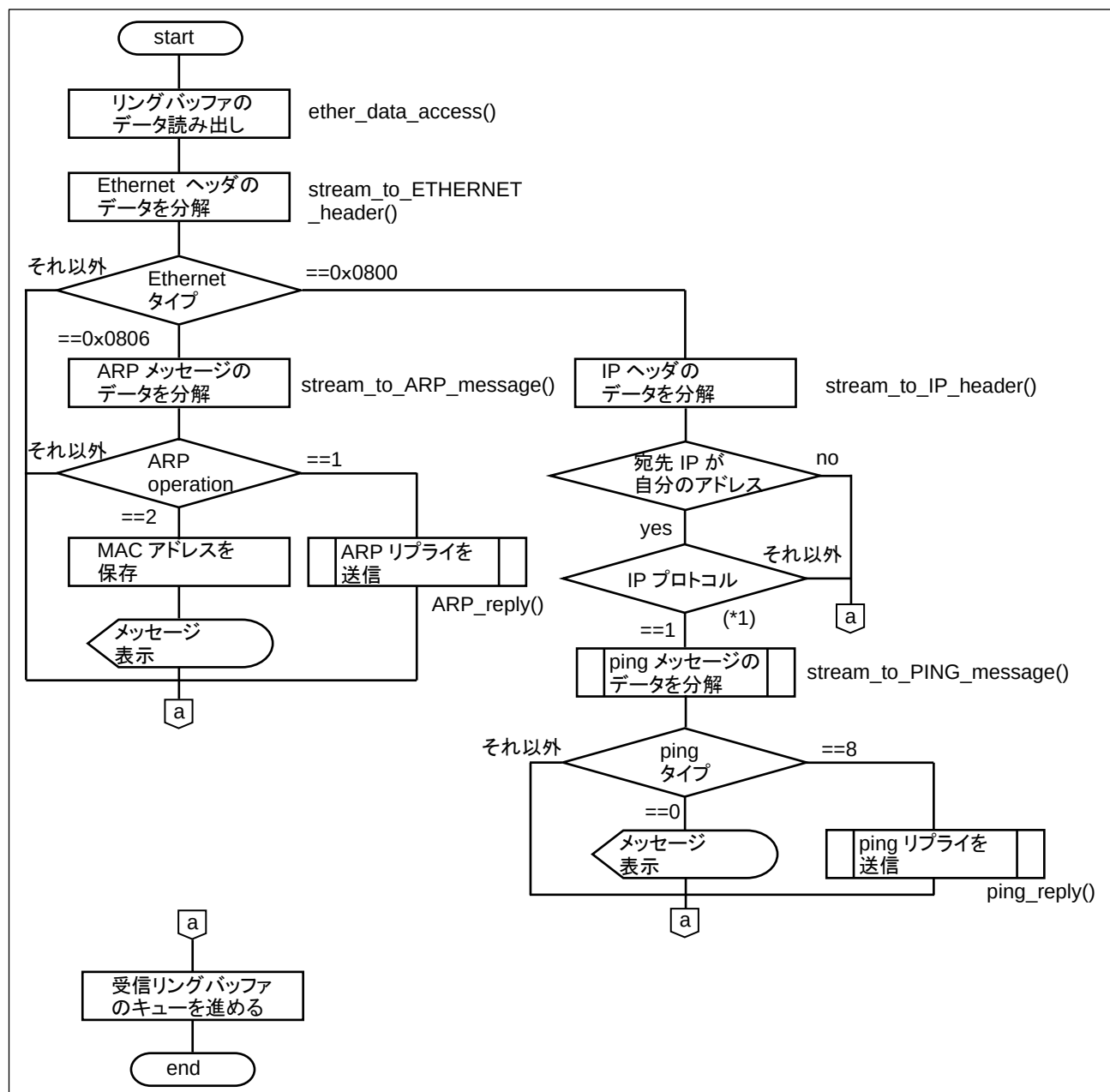
ping メッセージ内の type により、ping リプライか ping リクエストかを判断して処理を分けています。

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。

• receive_data_process() (main_ether.c 内)



(*1)本プログラムでは、IP プロトкол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кол (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固定で 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)

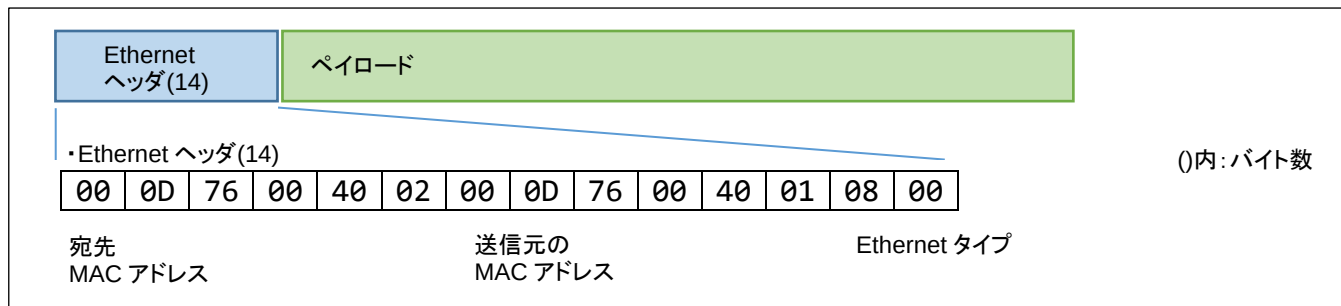
戻り値: なし

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 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)

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);
```

の样に行えます。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;
```

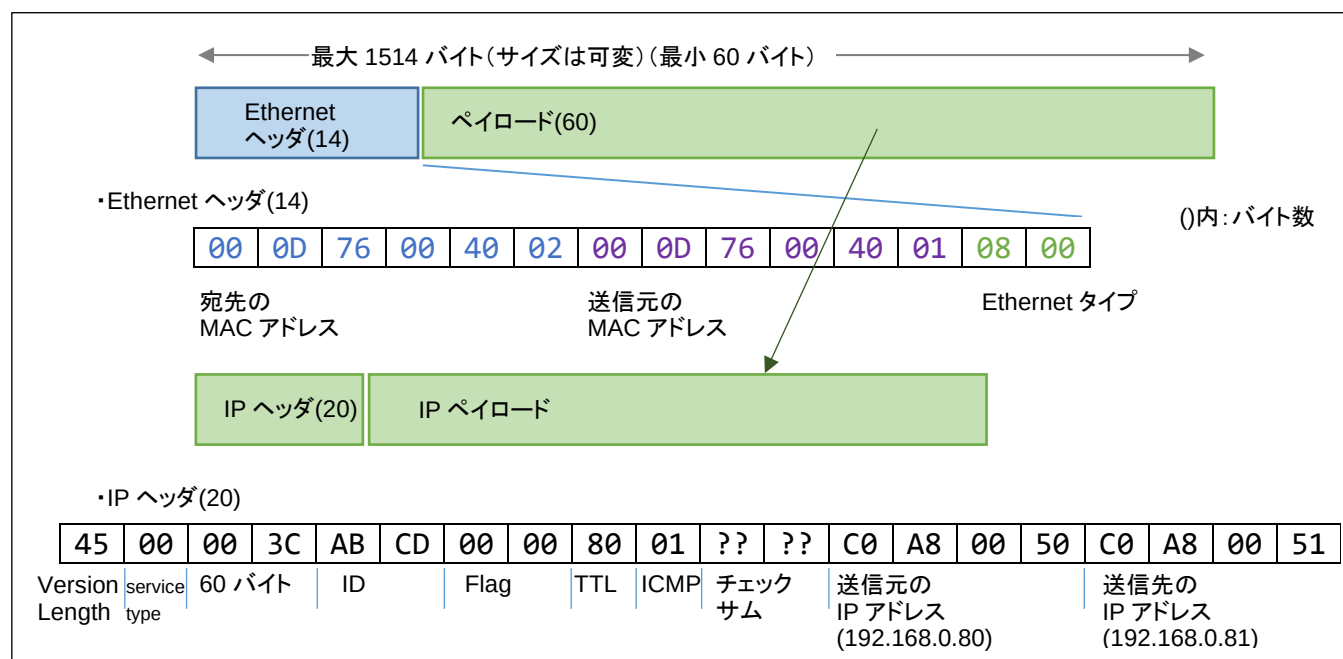
```
unsigned char send_data[60];

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。

Ethernet 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)

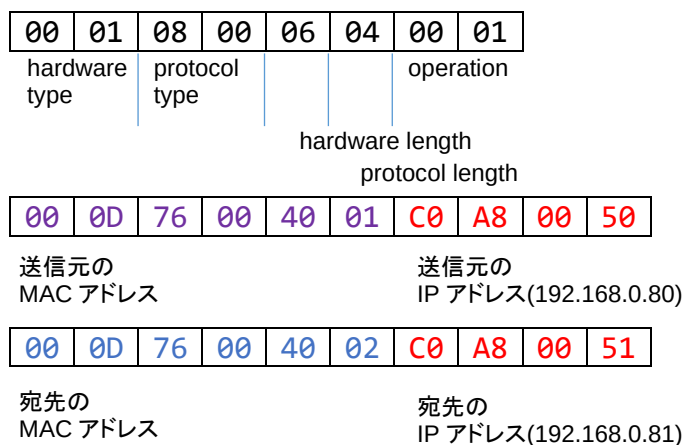
送信先アドレス ip_dst_ip[4] (unsigned char)

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	2026.9.29	—	初版発行, RX 向けのマニュアルを RA 向けに改訂

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>

商標等の表記について

- ・ 全ての商標及び登録商標はそれぞれの所有者に帰属します。
- ・ パーソナルコンピュータを PC と称します。

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

Ethernet スタータキット RA ソフトウェア編 取扱説明書(1)

株式会社 **北斗電子**

©2026 北斗電子 Printed in Japan 2026 年 9 月 29 日改訂 REV.1.0.0.0 (260929)
