RENESAS

R20UT3403JJ0100
Rev.1.00
2015.03.13

RH850 マルチコアプログラミングガイド 詳細編

目次

第 1 章　RH850 マルチコア向けプログラミングの留意点 3

第 2 章　libipcx 4

2.1　libipcx とは 4

2.2　libipcx が使用する RH850 の資源 4

2.3　libipcx の ID 4

第 3 章　排他制御 5

3.1　排他制御とは 5

3.1.1　割り込み処理との排他制御 7

3.1.2　他の PE との排他制御 7

3.2　ロックとは 8

3.3　ロックの使用方法 8

3.3.1　ID の初期化 8

3.3.2　ロックの獲得 8

3.3.3　ロックの解放 10

3.3.4　ロックの使用例 10

3.4　デッドロック 13

3.4.1　割り込み処理で発生するデッドロック 13

3.4.2　マルチコアで発生するデッドロック 14

3.5　ロックの数と資源の紐付けの方法 15

3.5.1　ジャイアント・ロック 15

3.5.2　PE ごとのロック 15

3.5.3　資源ごとのロック 15

第 4 章　同期 16

4.1　同期とは 16

4.2　順序関係がある同期の使用方法 16

4.2.1　ID の初期化 16

4.2.2　待ち状態への移行 16

4.2.3　同期 16

4.3　順序関係がない同期の使用方法 17

4.3.1　待ち状態への移行 17

4.3.2　同期 17

第 5 章　PE 間通信 18
5.1　PE 間通信とは 18
5.2　PE 間通信の使用方法 18
5.2.1　libipcx の PE 間通信用 API 18
5.2.2　P_call()関数、P_request()関数を使用するための準備 18
5.2.3　P_call()関数による PE 間の関数コール 18
5.2.4　P_request()関数による PE 間の関数コール 19

第1章　RH850マルチコア向けプログラミングの留意点

本章では RH850 のマルチコア向けプログラミングの留意点について説明します。マルチコアとは複数のプロセッサエレメント（以降 PE）を搭載したプロセッサで、シングルコアとは異なるプログラミングが必要になります。マルチコア向けのプログラミングを行う上で必要な項目として、排他制御、同期、PE 間通信の3つがあります。

排他制御とは、ある PE がメモリや IO レジスタなどの資源にアクセスしている間、他の PE や割り込み処理が同じ資源にアクセスできないようにする制御です。
同期とは、各 PE が待ち合わせを行い、待ち合わせを行った PE が同時に処理を開始（再開）する処理です。
PE 間通信とは、ある PE から他の PE に事象を伝達する処理です。

排他制御、同期、PE 間の通信の詳細については、3 章以降で説明します。
なお、ルネサス エレクトロニクスでは、排他制御、同期、PE 間通信をサポートするライブラリとして、libipcx を用意しています。libipcx については 2 章で説明します。また、3 章以降の使用方法では libipcx を使用しています。

第2章　libipcx

本章では libipcx について説明します。

2.1　libipcxとは

libipcx とは、ルネサス エレクトロニクス製の RH850 マルチコアをターゲットとしたサンプルライブラリです。このライブラリを使用することで、マルチコア向けプログラミングで必要となる 3 つの機能、排他制御、同期、PE 間通信を実現可能です。

2.2　libipcxが使用するRH850の資源

libipcx は RH850 の MEV レジスタと PE 間割り込み（IPIR）の一部を予約して使用します。
MEV レジスタはレジスタ名 MEV0～MEV 7、IPIR は IPIR_CH0,　IPIR_CH1 を使用します。MEV レジスタと IPIR の詳細は、RH850 のハードウェアマニュアルを参照してください。

2.3　libipcxのID

libipcx では予約した MEV レジスタの 0 と 1 の各ビットに1から 64 の ID を割り当てて管理しています。以降、ID は libipcx の ID を指します。
ID は後述の排他制御と同期の両方で使用します。排他制御と同期を同時に使用する場合、ID が重複しないように注意してください。

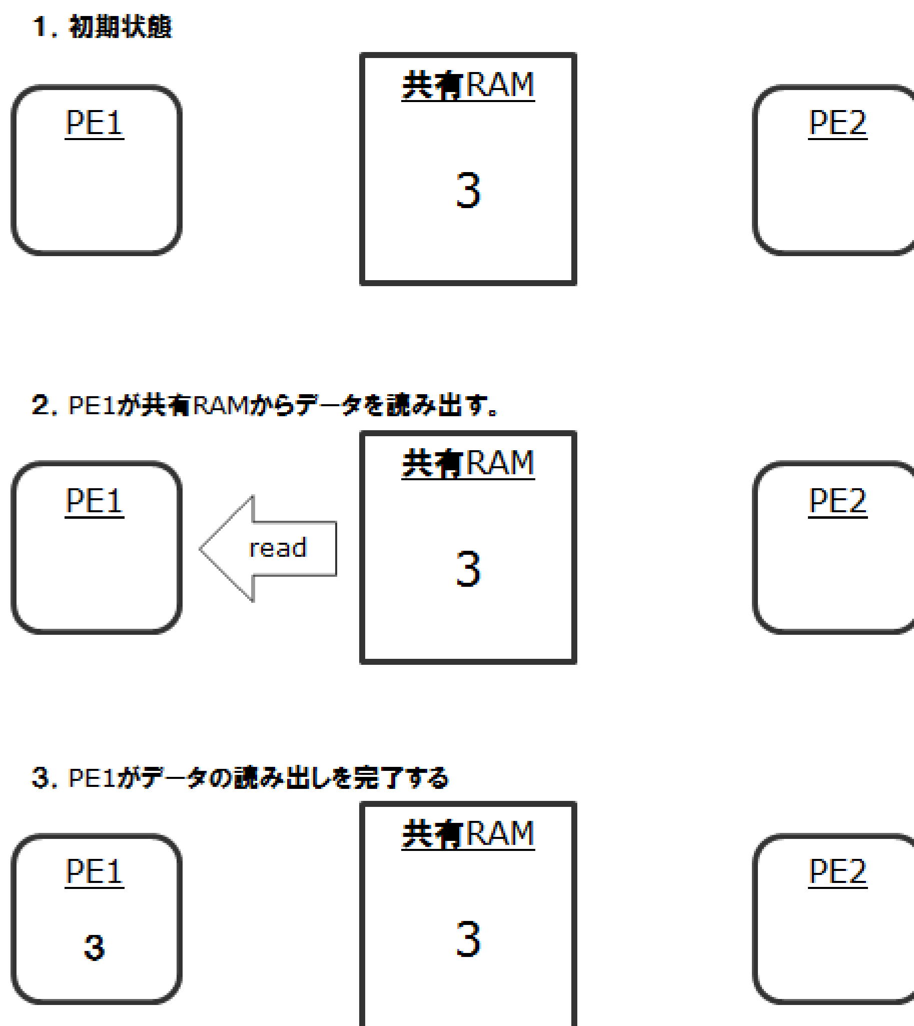
第3章　　排他制御

3.1　排他制御とは

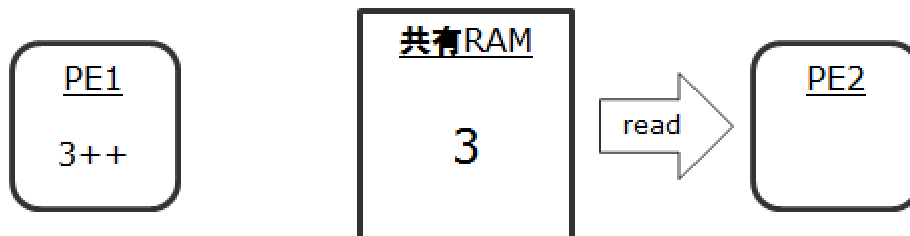
排他制御とは、ある PE がメモリや IO レジスタなどの資源にアクセスしている間、他の PE や割り込み処理が同じ資源にアクセスできないようにする制御です。本書では、資源とは RAM や IO レジスタを指します。また、アクセスとは、対象の領域から値を読み込み何らかの加工を行った後、対象の領域に書き込むまでの、Read Modify Write を指します。

ある資源にアクセス中、すなわち Read Modify Write 実行中に、割り込み処理や他の PE からのアクセスが発生すると、資源の一貫性を保てなくなります。これを避けるため、資源にアクセスしている間は、排他制御を行う必要があります。

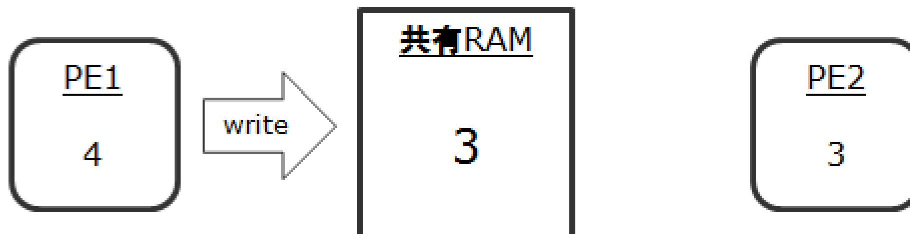
図1. マルチコアで資源の一貫性が保てなくなる例



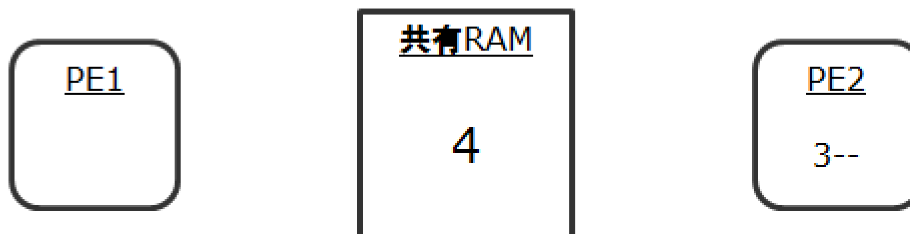
4. PE1が読み出した値に1加算する。PE2が共有RAMからデータを読み出す。



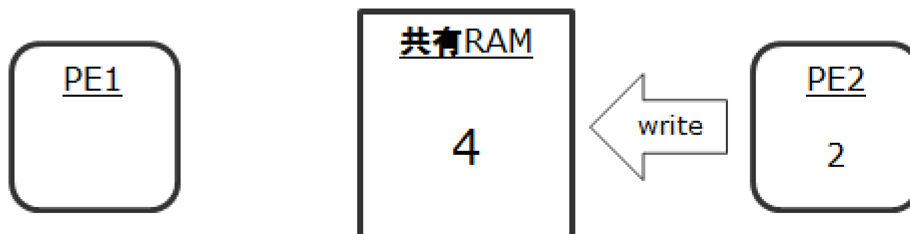
5. PE1が結果を共有RAMに書き込む。PE2がデータの読み出しを完了する。



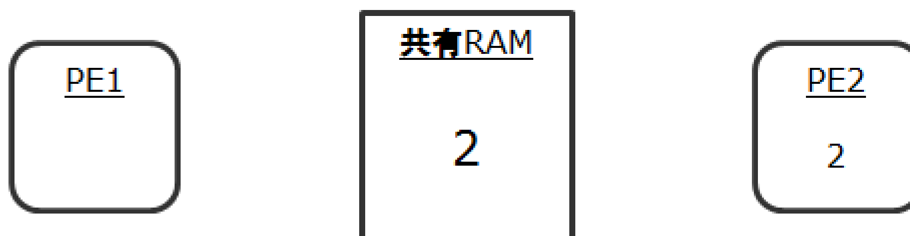
6. PE1が書き込みを完了する。PE2が読みだした値から1減算する。



7. PE2が結果を共有RAMに書き込む。



8. PE2が書き込みを完了する。



PE1の加算の結果"4"が、PE2の減算の結果"2"で上書きされてしまう。
PE2の減算はPE1の加算の結果を考慮しておらず、一貫性が失われてしまう。

3.1.1 割り込み処理との排他制御

通常の処理と割り込み処理で資源を共有している場合、通常の処理中で資源へのアクセス中に割り込みが発生し、割り込み処理内で同じ資源にアクセスが行われると、資源の一貫性が保てなくなります。これを避けるため、割り込み処理に対して排他制御を行う必要があります。

排他制御の実現方法として、割り込みを禁止にする方法が一般的です。資源へのアクセス時は、割り込みを禁止にすることで割り込みが発生しなくなるため、割り込み処理から資源にアクセスを行うことがなくなります。なお、割り込み処理に対する排他制御は、マルチコアに限った話ではなく、シングルコアにおいても同様に行う必要があります。

3.1.2 他のPEとの排他制御

複数の PE で資源を共有している場合、ある PE から資源にアクセスしている間に、他の PE から同じ資源にアクセスが行われると、資源の一貫性が保てなくなります。これを避けるため、他の PE に対して排他制御を行う必要があります。

排他制御の実現方法として、ロックを使用します。

3.2 ロックとは

ロックとは、排他制御を実現するための手法で、ロックの操作方法は獲得と解放の2つです。

ある PE がロックを獲得している間は、他の PE はロックを獲得することができません。この特性を利用して、共有 RAM や IO レジスタなどの資源にアクセスする場合には、アクセス前に必ずロックを獲得し、アクセス終了後にロックを解放するという手順を踏むことで、他の PE との排他制御を実現します。このとき、「この資源」を使用する場合には「このロック」を使用する、というようなロックと資源の紐付けをユーザが行う必要があります。

3.3 ロックの使用方法

ここではロックの使用方法について記述します。例として、libipcx の排他制御用 API を使用します。

3.3.1 IDの初期化

最初に libipcx の ID を初期化する必要があります。F_init()関数を実行して初期化を行います。引数には ID を指定します。1 から 64 のいずれかを指定した場合、指定された ID を初期化します。なお、0を指定した場合、1から 64 の全ての ID を初期化します。

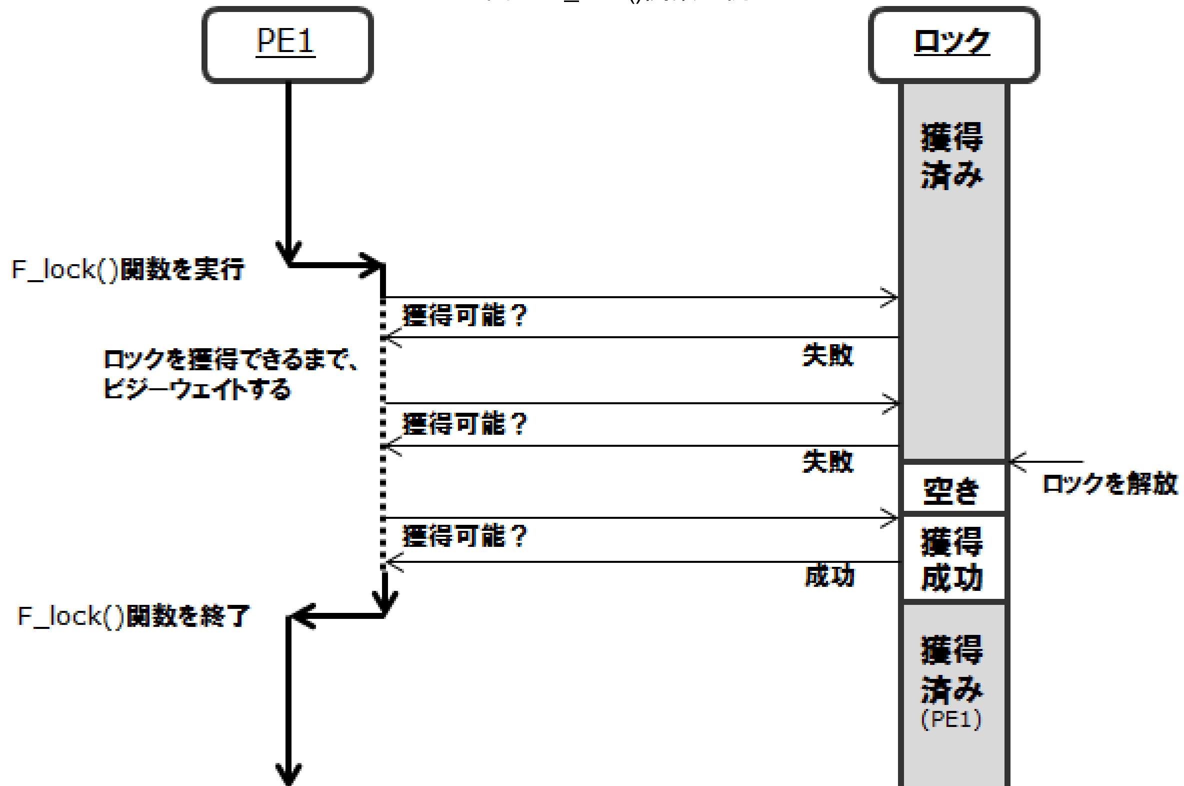
3.3.2 ロックの獲得

ロックの獲得は、F_lock()関数か S_lock()関数、F_try()関数か S_try()関数を実行することで行います。引数には ID を指定します

F_lock()関数か S_lock()が実行されると、libipcx は引数で指定された ID のロックが獲得できるかを確認し、獲得できた場合にのみ 0 を返却します。なお、指定された ID のロックがすでに獲得されている場合は、解放されるまで待ち続けます。この待ち続けている状態はビジーウェイトであるため、ロックを獲得できない時間が長くなるほど、待ち状態が続くことになります。

F_lock()関数と S_lock()関数の違いは、F_lock()関数では割り込み状態を変更しませんが、S_lock()関数では割り込みを禁止してからロックの獲得を行います。なお、ロックを獲得できない場合は、解放されるまで待ち続けるビジーウェイト状態になりますが、このとき定期的に少しの間割り込みを許可し、割り込みを受け付けることができます。ロック獲得時は割り込み禁止状態になります。

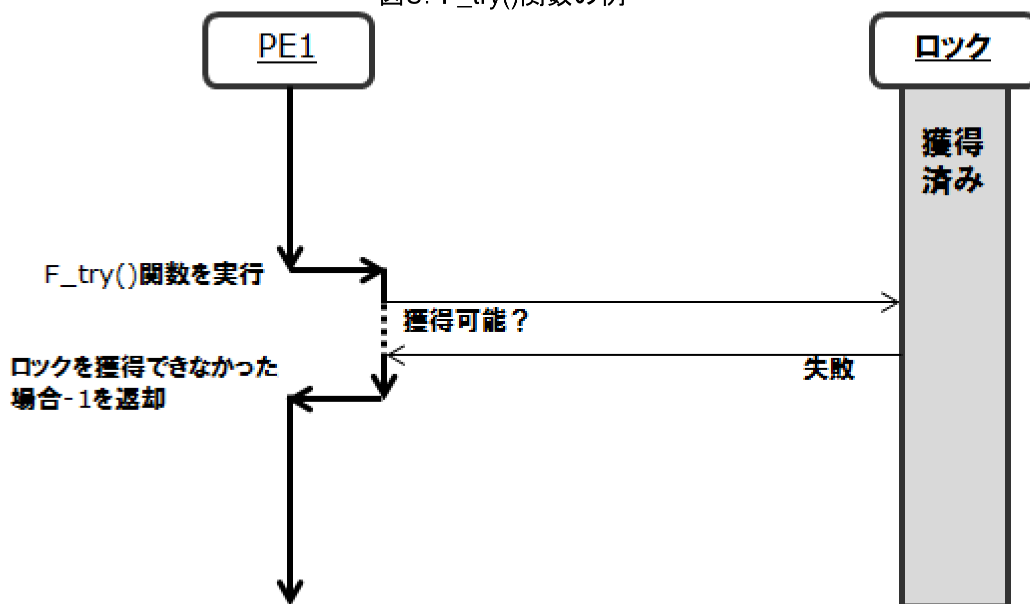
図2. F_lock()関数の例



F_try()関数かS_try()関数が実行されると、libipcxは引数で指定されたIDのロックが獲得できるかを確認し、獲得できた場合 0 を返却します。なお、ロックを獲得できない場合は、-1 を返却します。F_lock()と異なり、待ち状態には移行しません。

F_try()関数と S_try()関数の違いは、F_try()関数では割り込み状態を変更しませんが、S_try()関数では割り込みを禁止してからロックの獲得を行います。なお、ロックを獲得できない場合は、割り込みを許可してから-1 を返却します。

図3. F_try()関数の例



3.3.3 ロックの解放

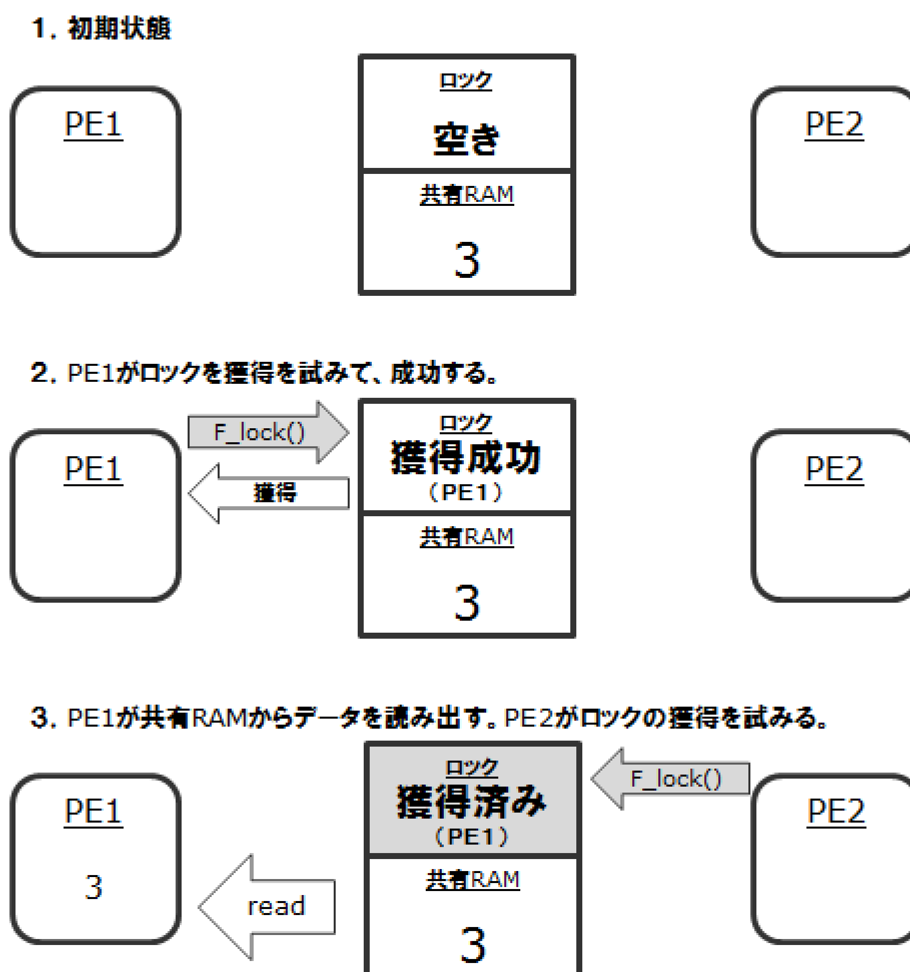
ロックの解放は、F_unlock()関数か S_unlock()を実行することで行います。引数には ID を指定します。F_unlock()関数か S_unlock()が実行されると、libipcx は引数で指定された ID のロックを解放します。戻り値は常に 0 を返却します。

F_lock()関数か F_try()関数でロックを行った場合は F_unlock()関数、S_lock()関数か S_try()関数でロックを行った場合は S_unlock()関数を必ず使用してください。

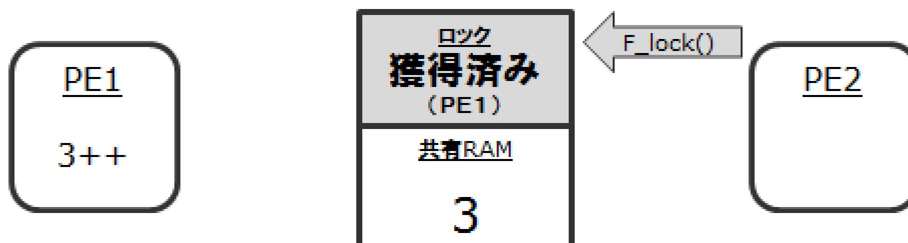
3.3.4 ロックの使用例

図4はロックを使用して排他制御を行った例です。ロックは共有RAMと紐付けられています。図1の例にロックを導入して排他制御を行うことで、資源の一貫性を保つことができるようになります。

図4. ロックの使用例

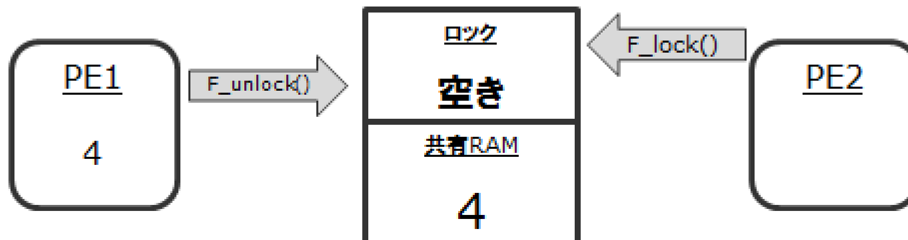


4. PE1が読みだした値に1加算する。

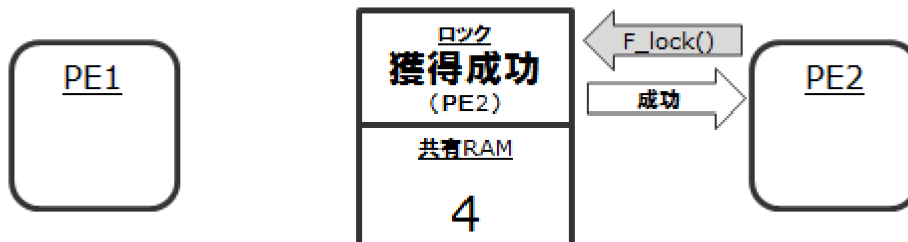


5. PE1が演算結果を共有RAMに書き込む。

6. PE1がロックを解放する。

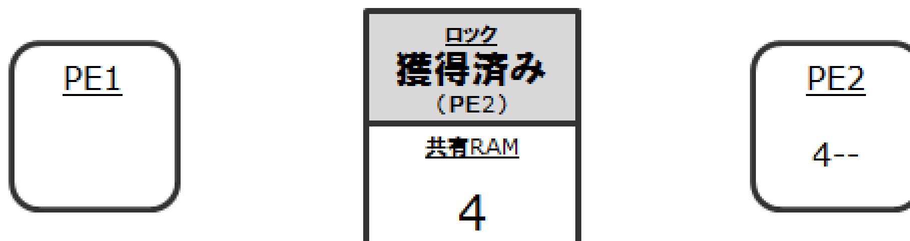


7. PE2がロックの獲得を試みて、成功する。

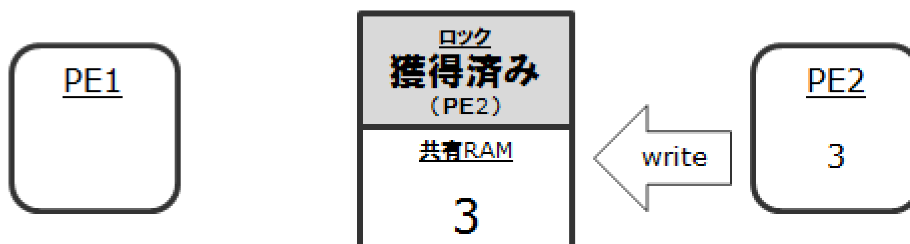


8. PE2が共有RAMからデータを読み込む。

9. PE2が読みだした値から1減算する。

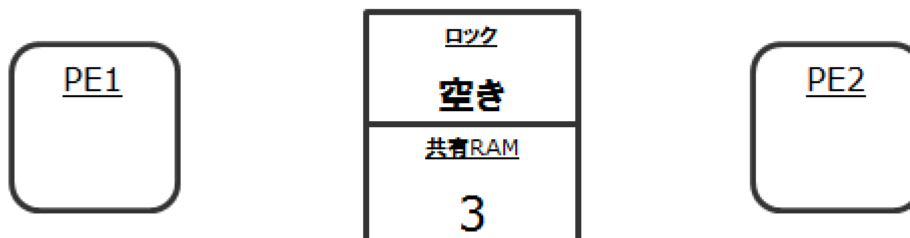


10. PE2が演算結果を共有RAMに書き込む



11. PE2がロックを解放する。

12. 処理が完了する。



3.4　デッドロック

デッドロックとは、獲得しようとしたロックが解放されず、ビジーウェイトから抜け出せない状態のことを言います。

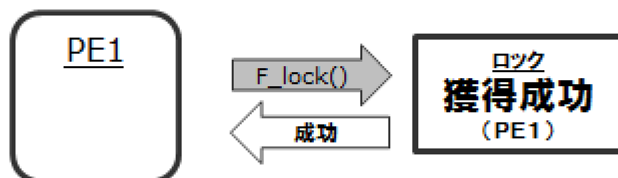
3.4.1　割り込み処理で発生するデッドロック

通常の処理で、ある ID のロックが獲得された状態で割り込みが発生し、割り込み処理中で同じ ID のロックを獲得しようとした場合、そのロックは解放されることが無いため、割り込み処理でロックを獲得できず、ビジーウェイトから抜け出すことができません。このような状態をデッドロックと呼びます。通常の処理と割り込み処理で同じ ID のロックを使用する場合、必ず割り込みを禁止にしてからロックの獲得を行い、ロックを解放したあとに割り込みを許可してください。

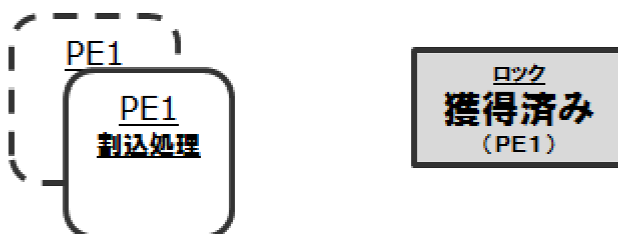
図5. 割り込み処理で発生するデッドロックの例

1．初期状態

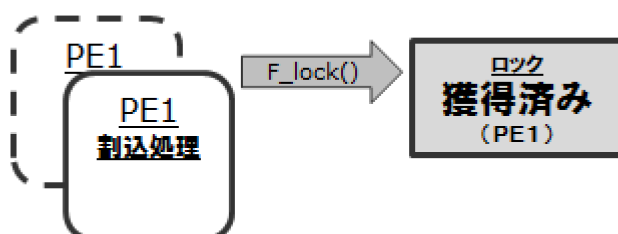
2．PE1がロックの獲得を試みて、成功する。



3．PE1に割り込みが発生し、割り込み状態になる。



4．PE1の割り込み処理からロックの獲得を試みて、デッドロック

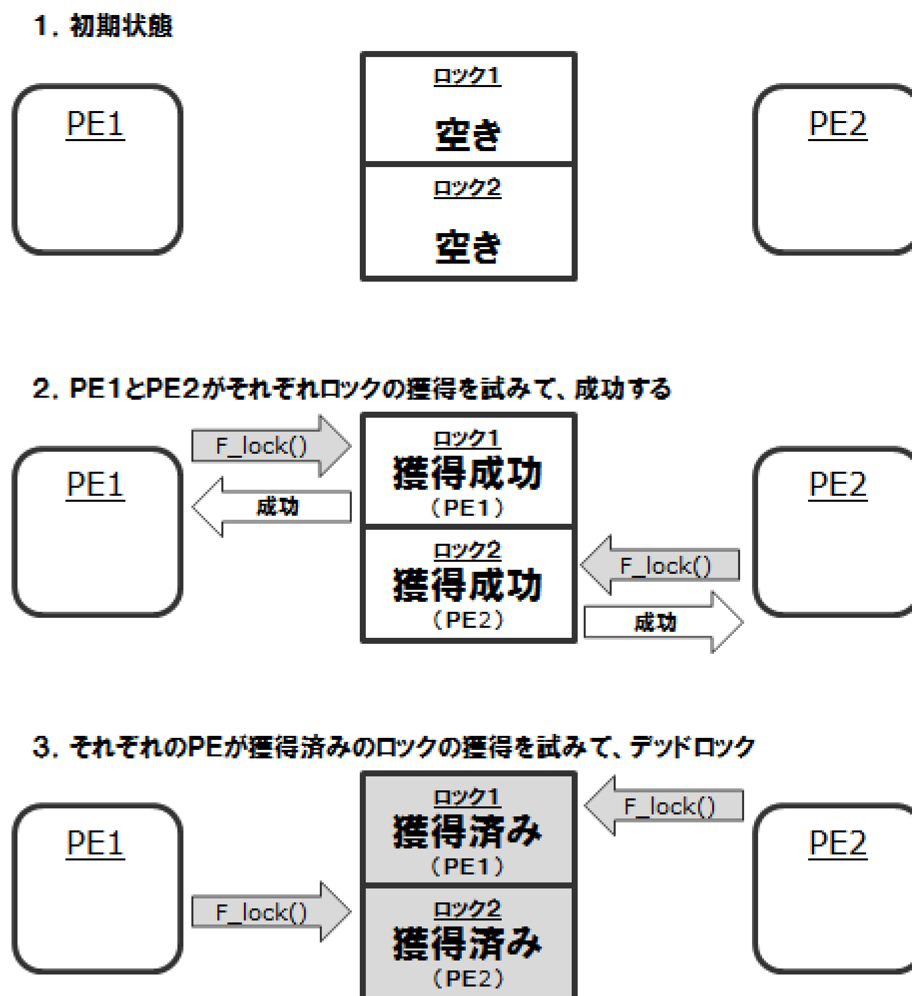


3.4.2 マルチコアで発生するデッドロック

マルチコアにおいては、ロックが複数ある場合にも、デッドロックが発生する可能性があります。

例として、2PE のマルチコアで、PE 毎のロック 1、2 を用意した場合において、PE1 がロック 1 を獲得し PE2 がロック 2 を獲得した状態で、PE1 がロック2を獲得しようとし、同時に PE2 がロック 1 を獲得しようとすると、デッドロックに陥ります。このように、あるロックを獲得している状態で別のロックを獲得しようとすると、デッドロックになる可能性があります。

図6. マルチコアでデッドロックが発生する例



複数のロックを使用する場合、デッドロックが発生しないように注意する必要があります。

複数のロックの獲得を行わないようにすることで、デッドロックを回避することができます。また、複数のロックの獲得を行う必要がある場合には、「必ずロック1を獲得した後にロック2を獲得する」ようにするなど、ロックの獲得順を管理してください。ロックの獲得順を管理することで、デッドロックを回避することができます。

RENESAS

3.5 ロックの数と資源の紐付けの方法

ロックを使用する場合、使用するロックの数を決める必要があります。ロックの数が少ないほど、ロックの管理は簡単になりますが、ロックを獲得できない待ち時間が長くなります。逆にロックの数が多くなるほどロックの管理は難しくなりますが、ロックを獲得できない待ち時間は短くなります。

また、「この資源」を使用する場合には「このロック」を使用する、というように、ロックと資源の紐付けを行う必要があります。ロックと資源の紐付けはユーザが行います。

3.5.1 ジャイアント・ロック

ジャイアント・ロックとは、マルチコアの全てのコアで1つのロックを用意し、1つのロックで全ての資源を管理する手法です。ロックが1つであるため、ロックの管理は簡単になります。しかし、いずれの資源にアクセスする場合でも、獲得できるロックは1つであるため、資源の数や PE の数が多くなるほど、ロックを獲得できない時間が長くなります。

3.5.2 PEごとのロック

PE ごとのロックとは、PE ごとにロックをに用意し、1つの PE 内の全ての資源を1つのロックで管理する手法です。

ジャイアント・ロックと比較して、ロックの数が増えるため、ロックを獲得できない時間を減らすことができるようになりますが、ロックの管理が複雑になります。なお、PE ごとのロックを用意する場合、ロックの獲得順によっては、デッドロックが発生する可能性があるため、ロックの管理に注意が必要です。

3.5.3 資源ごとのロック

資源ごとのロックとは、PE ごとのロックよりもロックの数を増やし、資源とロックを紐付けする手法です。ユーザはどの資源をどのロックに紐付けしたかを常に把握しておく必要があります。

複数のロックを用意して、各ロックに資源を振り分けて紐付けることで、ある PE があるロックを獲得している間でも、他のロックであれば獲得できるため、待ち時間なく資源にアクセス可能です。このようにロックを複数用意すれば、ロックを獲得できない待ち時間を減らすことができるようになります。ただし、ロックの数が増えるほどロックの管理が複雑になるため、資源ごとのロックを用意する場合は、ロックの管理に注意が必要です。

第4章　　同期

4.1　同期とは

同期とは、各 PE が待ち合わせを行い、待ち合わせを行った PE が同時に処理を開始（再開）する処理です。
RH850 では、各 PE が異なるアドレスから実行開始するものと、全ての PE が同じアドレスから実行開始するものがあります。後者の場合でも、初期化処理が完了した後に、通常は PE ごとの処理に分岐するため、大半は PE 毎にそれぞれ異なる処理を行っています。この異なる処理を行っている状態で、待ち合わせを行う場合に、同期を使用します。
同期には同期ポイントに到達する PE の順番が決まっている順序関係がある同期と、同期ポイントに到達する PE の順番が決まっていないバリア同期があります。

4.2　順序関係がある同期の使用方法

ここでは順序関係がある同期の使用方法について記述します。例として、libipcx の同期制御用 API を使用します。

4.2.1　IDの初期化

最初に libipcx の ID を初期化する必要があります。初期化については”3.3.1 ID の初期化”を参照してください。

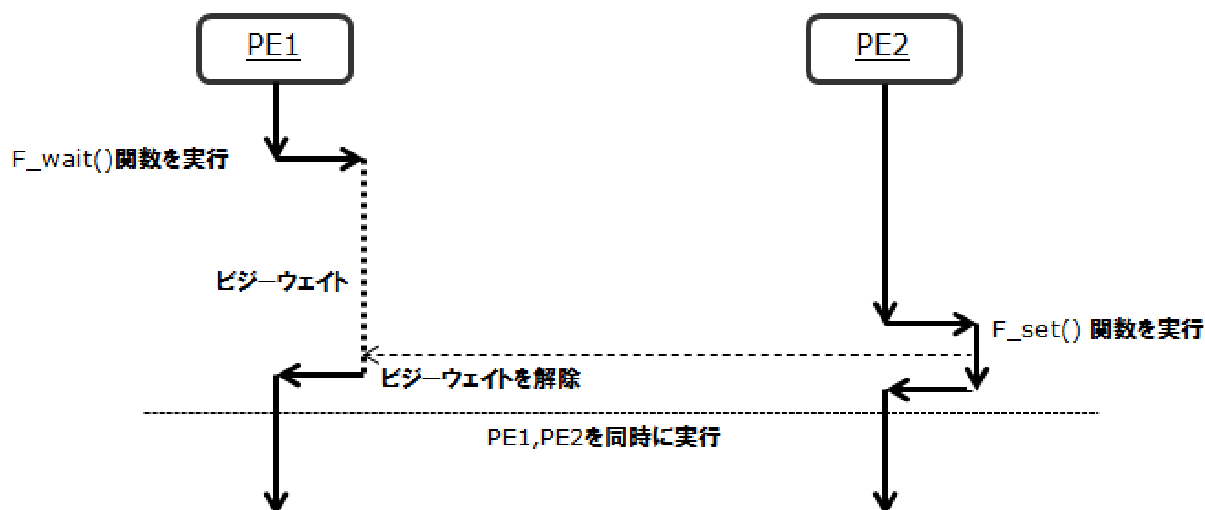
4.2.2　待ち状態への移行

順序関係がある同期では、先に同期ポイントに到達する PE を待ち状態に移行させる必要があります。待ち状態への移行は、F_wait()関数を実行することで行います。引数には ID を指定します。

4.2.3　同期

先に同期ポイントに到達している他の PE に通知を行い、中断していた処理を再開させ同期を行います。他の PE への通知は、F_set()関数を実行することで行います。引数には F_wait()関数で指定された ID と同じ ID を指定します。

図7. 同期の例



4.3 順序関係がない同期の使用方法

ここでは順序関係がない同期の使用方法について記述します。同期を行う複数の PE の処理が、どちらが先に同期ポイントに到達するかわからない場合には、バリア同期を使用します。例として、libipcx の同期制御用 API を使用します。

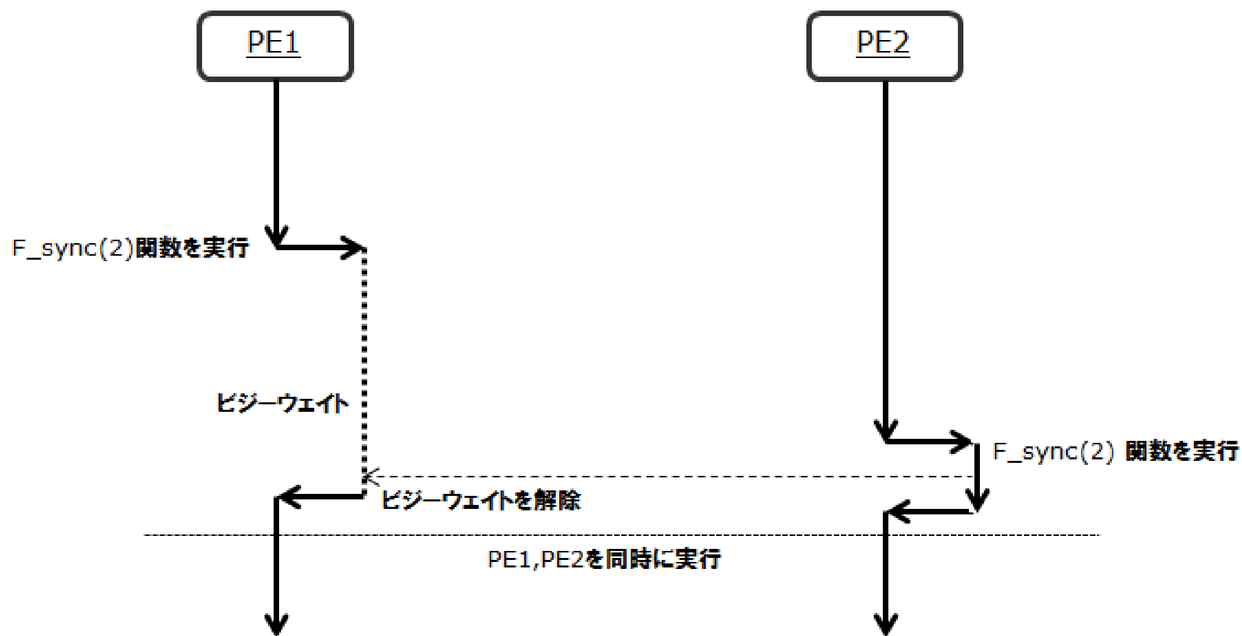
4.3.1 待ち状態への移行

順序関係がない同期では、同期ポイントに到達した PE から F_sync()関数を実行することで待ち状態に移行します。引数には同期を行う PE の総数を指定します。F_sync()関数の実行の順序は、どの PE からでも問題はありません。

4.3.2 同期

F_sync()関数の引数で指定された数の PE が F_sync()を実行すると、中断していた処理が再開され同期が行われます。
図 8 の例では PE1 が先に F_sync()関数を実行していますが、PE2 が先に F_sync()関数を実行しても同期を行うことができます。

図8. バリア同期の例



RENESAS

第5章　PE間通信

5.1 PE間通信とは

PE 間通信とは、ある PE から他の PE に事象を伝達する処理です。事象を伝えることで、他の PE に関数の実行を依頼することができます。

5.2 PE間通信の使用方法

ここでは PE 間通信の使用方法について記述します。例として libipcx の PE 間通信用 API を使用します。

5.2.1 libipcxのPE間通信用API

libipcx の PE 間通信用 API には、P_call()関数、P_vcall()関数、P_call_id()関数、P_vcall_id()関数、P_request()関数、P_vrequest()、P_request_id()関数、P_vrequest_id()関数があります。

なお、P_call()関数と P_vcall()関数の違いは P_call()関数は引数の上限が２つなのに対し、P_vcall()関数は引数の情報をリストとして渡します。P_request()関数と P_vrequest()関数についても同様です。

また、P_call()関数と P_call_id()関数の違いは P_call()関数は呼び出す関数を関数ポインタで指定するのに対し、P_call_id()関数では用意したテーブルを使用して id で関数を指定します。P_request()関数と P_request_id()関数についても同様です。

P_call_id()関数、P_request_id()関数、P_vcall()関数、P_vrequest()関数は、いずれも異なるのは引数のみで、機能自体は P_call()関数、P_request()と同じです。この例では P_call()関数、P_request()関数を使用します。

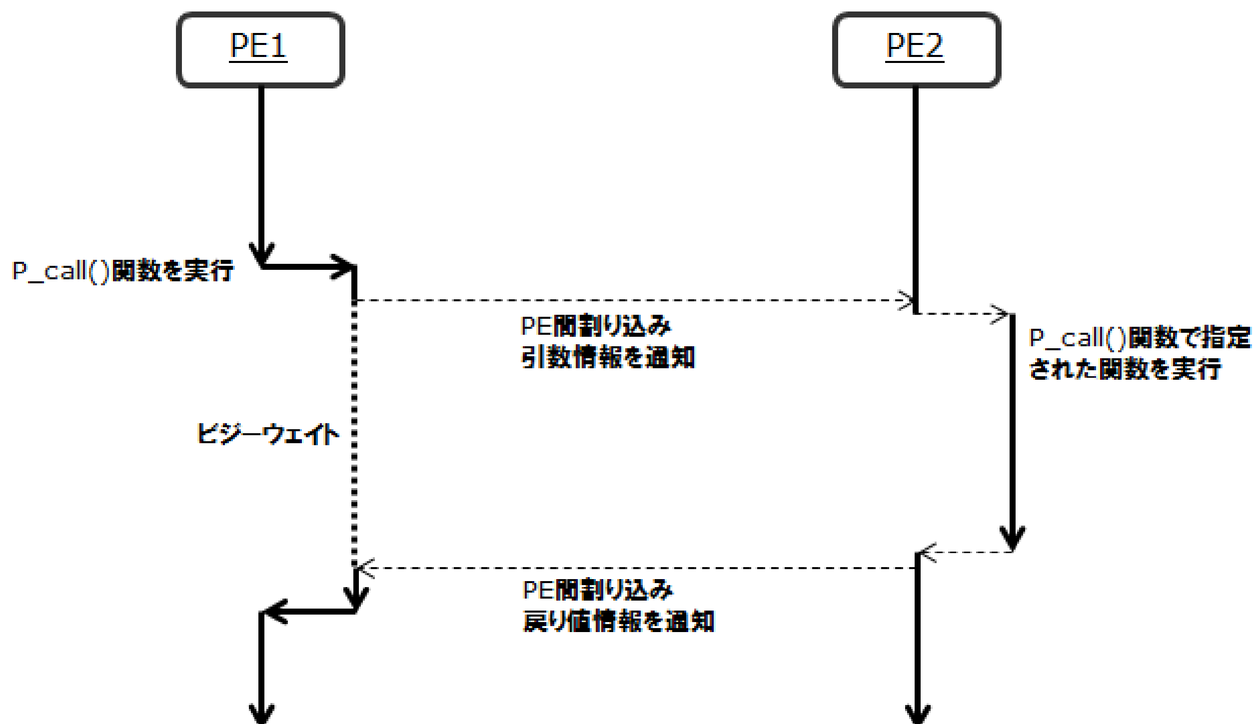
5.2.2 P_call()関数、P_request()関数を使用するための準備

P_call()関数、P_request()関数では、他の PE に関数ポインタ情報を通知して、通知した関数の実行を依頼します。そのため、PE 間の関数コール機能で使用する関数は PE 間で共有する必要があります。PE 間で関数を共有する方法についてはコンパイラにより異なりますので、使用するコンパイラのマニュアルを参照してください。

5.2.3 P_call()関数によるPE間の関数コール

P_call()関数の引数は、PE の ID、実行させる関数ポインタと、その関数に渡す第一引数、第二引数、の 4 つです。P_call()関数が実行されると、引数で指定された PE に割り込みを行い、割り込みを受けた PE が、引数で指定された関数ポインタの関数を実行します。実行した関数の戻り値を P_call()関数の戻り値として返却します。つまり、P_call()関数は、引数で指定した関数ポインタの関数が実行完了するまで待ち続けます。

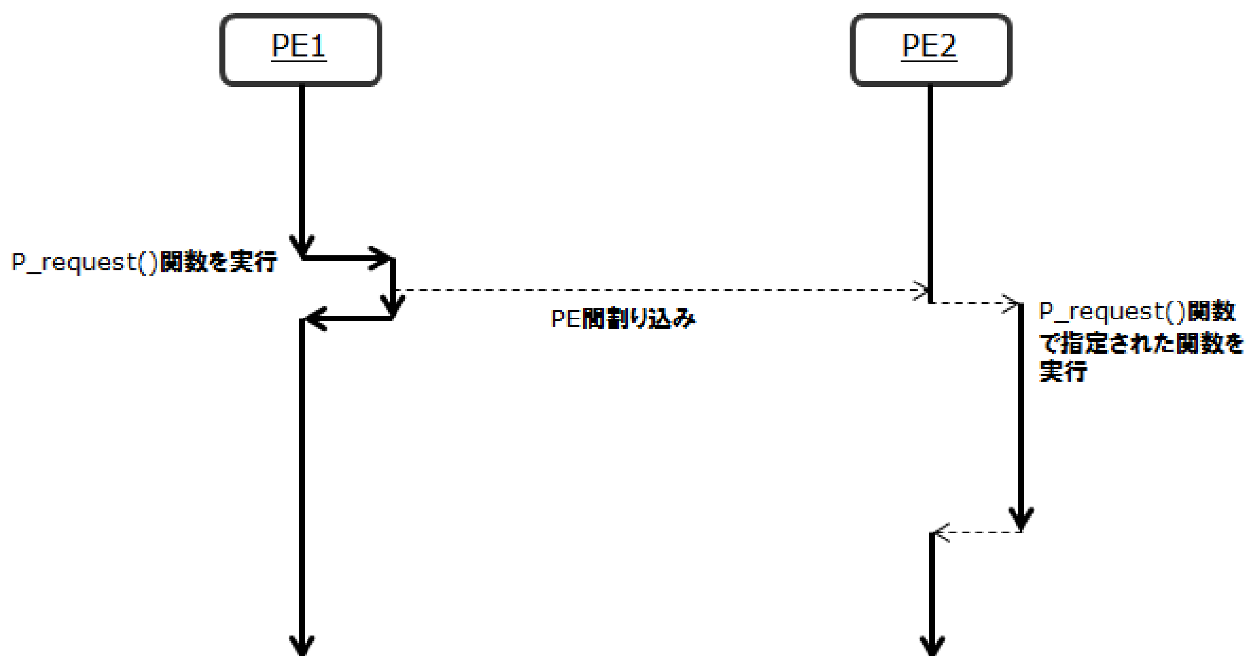
図9. P_call の使用例



5.2.4 P_request()関数によるPE間の関数コール

P_requet()関数の引数は、PE の ID、実行させる関数ポインタと、その関数に渡す第一引数、第二引数、の 4 つです。P_request()関数が実行されると、引数で指定された PE に割り込みを行い、割り込みを受けた PE が、引数で指定された関数ポインタの関数を実行します。なお、実行した関数の実行完了は待ちません。また、P_request()関数の戻り値もありません。

図10. P_request の使用例



以上

RENESAS

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して、お客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
2. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
3. 本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害に関し、当社は、何らの責任を負うものではありません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、
   各品質水準は、以下に示す用途に製品が使用されることを意図しております。
   標準水準：　コンピュータ、OA機器、通信機器、計測機器、AV機器、
   　　　　　　家電、工作機械、パーソナル機器、産業用ロボット等
   高品質水準：輸送機器（自動車、電車、船舶等）、交通用信号機器、
   　　　　　　防災・防犯装置、各種安全装置等
   当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。 たとえ、意図しない用途に当社製品を使用したことによりお客様または第三者に損害が生じても、当社は一切その責任を負いません。 なお、ご不明点がある場合は、当社営業にお問い合わせください。
6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 本資料に記載されている当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。また、当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途に使用しないでください。当社製品または技術を輸出する場合は、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。

注1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。
注2. 本資料において使用されている「当社製品」とは、注１において定義された当社の開発、製造製品をいいます。

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/contact/

© 2015 Renesas Electronics Corporation. All rights reserved.

Colophon 3.0