User's Manual

RX210グループ
Peripheral Driver Generator
リファレンスマニュアル

ルネサス エレクトロニクス
www.renesas.com

Rev.1.04　2014.5

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。

はじめに

本書はPeripheral Driver Generator用いたRX210グループの周辺I/Oドライバ作成の作成方法について説明します。マイクロコントローラ機種に依存しないPeripheral Driver Generatorの基本操作方法については、Peripheral Driver Generatorユーザーズマニュアルを参照してください。

目次

1. 概要

1.1 サポート範囲

Peripheral Driver Generator がサポートする RX210 グループの製品型名、周辺機能、エンディアンは以下の通りです。

(1) 製品型名

型名	パッケージ	型名	パッケージ
R5F52108AxFP	PLQP0100KB	R5F52106BxFM	PLQP0064KB
R5F52108AxFN	PLQP0080KB	R5F52106BxFL	PLQP0048KB
R5F52108AxFF	PLQP0080JA	R5F52106BxLJ	PTLG0100JA
R5F52108AxFM	PLQP0064KB	R5F52106BxLA	PTLG0100KA
R5F52108AxFK	PLQP0064GA	R5F52106BxFF	PLQP0080JA
R5F52108AxLJ	PTLG0100JA	R5F52106BxFK	PLQP0064GA
R5F52107AxFP	PLQP0100KB	R5F52106BxLH	PTLG0064JA
R5F52107AxFN	PLQP0080KB	R5F52106BxBM	SWBG0069LA
R5F52107AxFF	PLQP0080JA	R5F52105BxFB	PLQP0144KA
R5F52107AxFM	PLQP0064KB	R5F52105BxLK	PTLG0145KA
R5F52107AxFK	PLQP0064GA	R5F52105BxFP	PLQP0100KB
R5F52107AxLJ	PTLG0100JA	R5F52105BxFN	PLQP0080KB
R5F52106AxFP	PLQP0100KB	R5F52105BxFM	PLQP0064KB
R5F52106AxFN	PLQP0080KB	R5F52105BxFL	PLQP0048KB
R5F52106AxFF	PLQP0080JA	R5F52105BxLJ	PTLG0100JA
R5F52106AxFM	PLQP0064KB	R5F52105BxLA	PTLG0100KA
R5F52106AxFK	PLQP0064GA	R5F52105BxFF	PLQP0080JA
R5F52106AxLJ	PTLG0100JA	R5F52105BxFK	PLQP0064GA
R5F52105AxFP	PLQP0100KB	R5F52105BxLH	PTLG0064JA
R5F52105AxFN	PLQP0080KB	R5F52105BxBM	SWBG0069LA
R5F52105AxFF	PLQP0080JA	R5F52104BxFM	PLQP0064KB
R5F52105AxFM	PLQP0064KB	R5F52104BxFL	PLQP0048KB
R5F52105AxFK	PLQP0064GA	R5F52104BxFF	PLQP0080JA
R5F52105AxLJ	PTLG0100JA	R5F52104BxLH	PTLG0064JA
R5F5210BBxFB	PLQP0144KA	R5F52103BxFM	PLQP0064KB
R5F5210BBxLK	PTLG0145KA	R5F52103BxFL	PLQP0048KB
R5F5210BBxFP	PLQP0100KB	R5F52103BxFF	PLQP0080JA
R5F5210BBxLJ	PTLG0100JA	R5F52103BxLH	PTLG0064JA
R5F5210ABxFB	PLQP0144KA	R5F52108CxFP	PLQP0100KB
R5F5210ABxLK	PTLG0145KA	R5F52108CxFN	PLQP0080KB
R5F5210ABxFP	PLQP0100KB	R5F52108CxFM	PLQP0064KB
R5F5210ABxLJ	PTLG0100JA	R5F52108CxLJ	PTLG0100JA
R5F52108BxFB	PLQP0144KA	R5F52108CxFF	PLQP0080JA
R5F52108BxLK	PTLG0145KA	R5F52108CxFK	PLQP0064GA
R5F52107BxFB	PLQP0144KA	R5F52107CxFP	PLQP0100KB
R5F52107BxLK	PTLG0145KA	R5F52107CxFN	PLQP0080KB
R5F52106BxFB	PLQP0144KA	R5F52107CxFM	PLQP0064KB
R5F52106BxLK	PTLG0145KA	R5F52107CxLJ	PTLG0100JA
R5F52106BxFP	PLQP0100KB	R5F52107CxFF	PLQP0080JA
R5F52106BxFN	PLQP0080KB	R5F52107CxFK	PLQP0064GA

(2) 周辺機能

電圧検出回路 (LVDAa)	8 ビットタイマ(TMR)
クロック発生回路	コンペアマッチタイマ (CMT)
クロック周波数精度測定回路(CAC)	リアルタイムクロック (RTCb)
消費電力低減機能	ウォッチドッグタイマ (WDTA)
レジスタライトプロテクション機能	独立ウォッチドッグタイマ (IWDTa)
割り込みコントローラ (ICUb), 例外処理	シリアルコミュニケーションインタフェース(SCIc, SCId)
バス	I^2C バスインタフェース (RIIC)
DMAコントローラ (DMACA)	シリアルペリフェラルインタフェース (RSPI)
データトランスファコントローラ (DTCa)	CRC 演算器 (CRC)
イベントリンクコントローラ(ELC)	12 ビットA/D コンバータ (S12ADb)
I/O ポート	D/A コンバータ(DA)
マルチファンクションピンコントローラ(MPC)	コンパレータA(CMPA)
マルチファンクションタイマパルスユニット2 (MTU2a)	コンパレータA(CMPB)
ポートアウトプットイネーブル2 (POE2a)	データ演算回路(DOC)
16ビットタイマパルスユニット (TPUa)	

(3) エンディアン

リトル
ビッグ

1.2 関連ツール

本バージョンの Peripheral Driver Generator で RX210 グループを使用する際に必要な関連ツールは以下の通りです。

・RXファミリ用C/C++コンパイラパッケージ V.1.02 Release 01

・RX210グループ Renesas Peripheral Driver Library V.2.10 (Peripheral Driver Generatorに同梱されています。)

2. プロジェクトの作成

プロジェクトを新規に作成するにはメニューから［ファイル］->［プロジェクトの新規作成］を選択してください。[新規作成]ダイアログボックスが開きます。

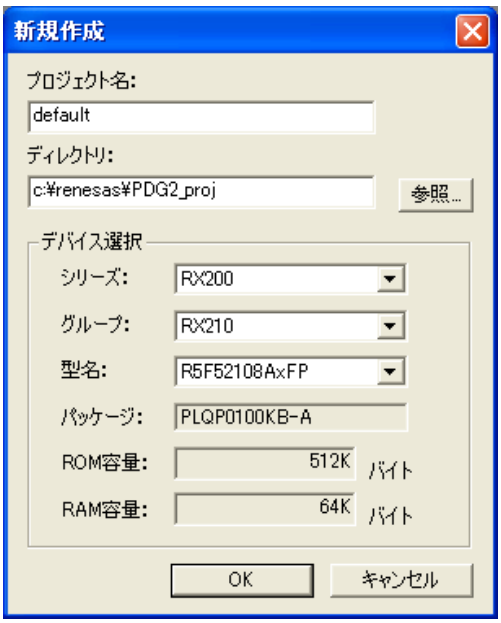


図 2.1 新規作成ダイアログボックス

RX210 グループのプロジェクトを作成するにはシリーズに［RX200］を、グループに［RX210］を選択してください。使用する製品の型名を選択すると、その製品のパッケージ、ROM 容量、RAM 容量が表示されます。

[OK]をクリックすると新規プロジェクトを作成して開きます。

新規プロジェクトの作成直後は EXTAL 入力周波数が設定されていないためエラーが表示されます。エラーの表示についてはユーザーズマニュアルを参照してください。

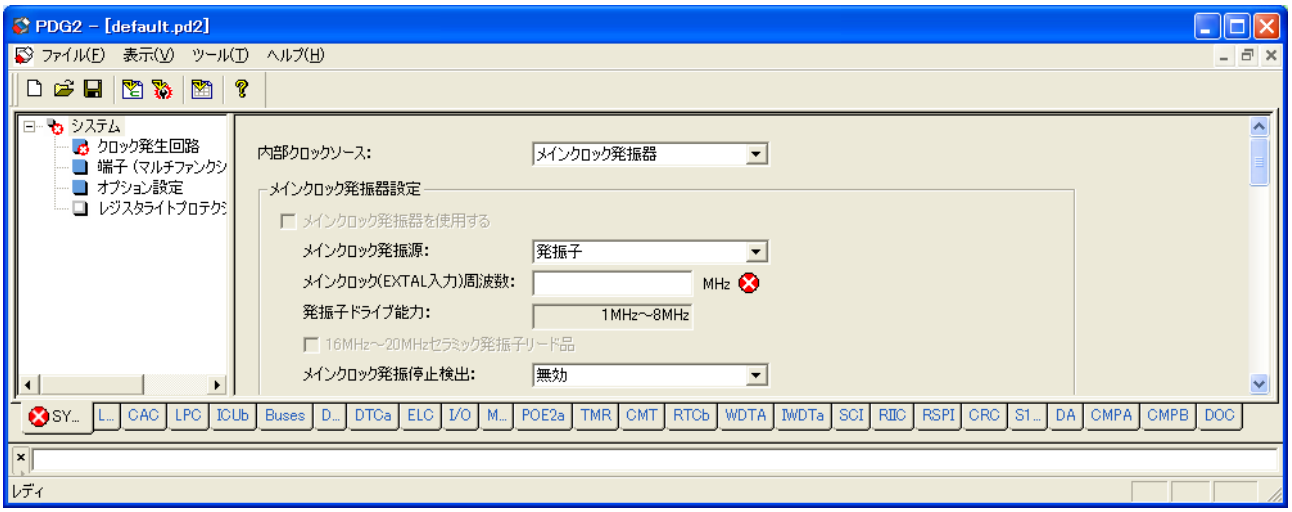


図 2.2 新規プロジェクトのエラー表示

ここでは使用するクロック周波数を設定してください。

3. 周辺機能の設定

3.1 設定画面

図 3.1 に周辺モジュール設定ウィンドウの表示例を示します。

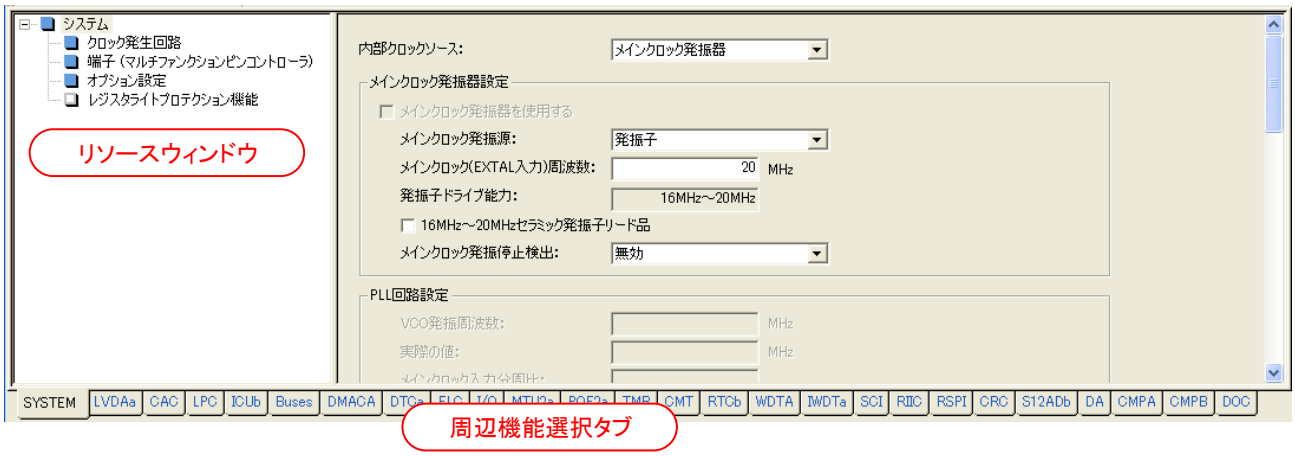


図3.1 周辺機能設定ウィンドウの表示例

周辺機能選択タブおよびリソースウィンドウに表示される項目と、周辺機能の対応を表 3.1 に示します。

表 3.1 周辺機能選択タブおよびリソースウィンドウの項目と周辺機能の対応

タブ	リソースウィンドウ	対応する周辺機能
SYSTEM	クロック発生回路	クロック発生回路
	端子(マルチファンクションピンコントローラ)	端子機能(マルチファンクションピンコントローラ(MPC))
	オプション設定	エンディアン設定
	レジスタライトプロテクション機能	レジスタライトプロテクション機能
LVDAa	電圧監視0～2	電圧監視0～2
CAC	クロック周波数精度測定回路(CAC)	クロック周波数精度測定回路(CAC)
LPC	消費電力低減機能	消費電力低減機能
ICUb	割り込み	割り込みコントローラ（ICUb） （高速割り込み，ソフトウェア割り込み、外部割込み(NMI, IRQ0～7)）
	例外	例外処理
Buses	CS0 ～ CS3	CS領域（CS0～CS3）
	共通設定	バスプライオリティ、バスエラー監視
DMACA	DMACA0 ～ DMACA3	DMAコントローラ（DMACA）チャネル0～3
DTCa	データトランスファコントローラ(DTCa)	データトランスファコントローラ（DTCa）
ELC	イベントリンク設定	イベントリンクコントローラ(ELC) イベントリンク設定
	ポートグループ、シングルポート設定	イベントリンクコントローラ(ELC) ポートグループ、シングルポート設定
I/O	ポート0 ～ ポートJ	I/Oポート ポート0～J
MTU2a	ユニット0（MTU0～MTU5）	マルチファンクションタイマパルスユニット2（MTU2a）チャネル0～5
POE2a	ポートアウトプットイネーブル2(POE2a)	ポートアウトプットイネーブル2（POE2a）
TMR	ユニット0（TMR0、TMR1）	8ビットタイマ（TMR）ユニット0（チャネル0、1）
	ユニット1（TMR2、TMR3）	8ビットタイマ（TMR）ユニット1（チャネル2、3）

CMT	ユニット0 (CMT0、CMT1)	コンペアマッチタイマ (CMT) ユニット0 (チャネル0、1)
	ユニット1 (CMT2、CMT3)	コンペアマッチタイマ (CMT) ユニット1 (チャネル2、3)
RTCb	リアルタイムクロック(RTCb)	リアルタイムクロック (RTCb)
WDTA	ウォッチドッグタイマ (WDTA)	ウォッチドッグタイマ (WDTA)
IWDTa	独立ウォッチドッグタイマ (IWDTa)	独立ウォッチドッグタイマ (IWDTa)
SCI	SCI0, 1, 5, 6, 8, 9, 12	シリアルコミュニケーションインタフェース SCIc(SCI0,1,5,6,8,9), SCId(SCI12)
RIIC	RIIC0	I^2Cバスインタフェース (RIIC)
RSPI	RSPI0	シリアルペリフェラルインタフェース (RSPI)
CRC	CRC演算(CRC)	CRC演算 (CRC)
S12ADb	S12AD0	12 ビットA/D コンバータ (S12ADb)
DA	DA0、DA1	D/Aコンバータ チャネル0、1
CMPA	コンパレータA(CMPA)	コンパレータA1,A2
CMPB	コンパレータA(CMPB)	コンパレータB0,B1
DOC	データ演算回路(DOC)	データ演算回路(DOC)

周辺機能の設定手順については、ユーザーズマニュアルを参照してください。 端子機能の設定については「3.2 端子機能」を参照してください。

3.2 端子機能（マルチファンクションピンコントローラ）の設定

RX210 グループはマルチファンクションピンコントローラ(MPC)により各端子の端子機能を選択します。Peripheral Driver Generator ではマルチファンクションピンコントローラ(MPC)の設定を端子機能ウィンドウから行うことができます。

周辺機能選択タブから[SYSTEM]を選択し、リソースウィンドウで[端子(マルチファンクションピンコントローラ)]を選択すると、端子機能ウィンドウが開きます。

図 3.2 端子機能ウィンドウの表示方法

端子機能ウィンドウは[端子機能]シートと、[周辺機能別使用端子]シートで構成されます。これらのシートの設定内容は連動し、どちらのシートからも端子機能を設定することができます。

3.2.1 端子機能シート

(1) 構成

端子機能シートはマイクロコントローラの全端子を番号順に表示し、各端子に割り当てられている端子機能を表示します。 割り当てる機能を複数から選択できるポートでは、本シートで割り当てる機能を選択することができます。

図3.3 端子機能ウィンドウ 端子機能シート

各カラムの表示内容を表 3.2 に示します。

表 3.2　端子機能シートの表示内容

カラム	内容
端子番号	端子の番号が表示されます。
端子名	端子名（端子に割り当てられる全機能）が表示されます。
選択機能	現在割り当てられている端子機能が表示されます。
入出力	現在割り当てられている端子機能の入出力方向が表示されます。
状態	警告またはエラーが発生している場合はその内容が表示されます。

(2)　初期状態

初期状態（周辺機能が何も設定されていない状態）では、ポートの[選択機能]カラムに、機能が割り当てられていないことを示す”NotAssigned”が表示されます。(図 3.4)

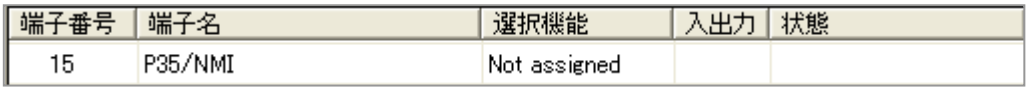
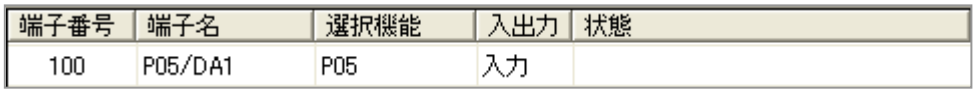


図 3.4　初期状態の端子機能シートの表示（100 ピン LQFP 版）

注意

・RX210 グループは、初期状態でのポートの端子機能は汎用入力ポートに設定されています。端子機能シートでは初期状態（周辺機能が何も設定されていない状態）のポートの選択機能が”NotAssigned”となっていますが、実際には汎用入力ポートとして動作します。I/O ポートの設定ウィンドウで端子を汎用入力ポートとして設定すると、[選択機能]に汎用ポート名が表示されます。(図 3.5)

(a) 初期状態



(b)　I/O ポート設定で汎用入力ポート P05 を設定後

図 3.5　初期状態と汎用ポート設定後のポート P05 の表示（100 ピン LQFP 版）

(3)　端子機能の選択

割り当てる機能を複数の中から選択できるポートは、マウスポインタを[選択機能]カラム上に置くと、ドロップダウンボタンが表示され、クリックすると端子機能の選択肢が表示されます。(図 3.6)

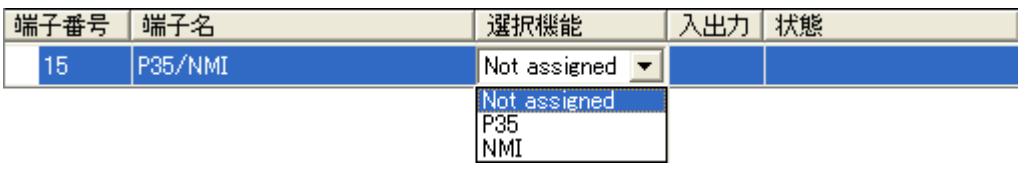


図 3.6　端子機能の選択肢

初期状態（周辺機能が何も設定されていない状態）では、端子機能を “Not assigned” から別の機能に変更すると、[<端子機能名>は周辺機能の設定で使用するように設定されていません] の警告が表示されます。例えば割り込みコントローラ(ICUA)が未設定の状態で、P35/NMI の端子機能を”Not assigned” から NMI に変更すると、図 3.7 のように表示されます。

端子番号	端子名	選択機能	入出力	状態
⚠ 15	P35/NMI	NMI		NMI は周辺機能の設定で、使用するように設定されていません。

図 3.7 初期状態で選択機能を変更した場合の警告表示

ここで、割り込みコントローラ(ICUA)設定ウィンドウから NMI を設定すると、警告表示が消え、選択機能の NMI が表示されます。

端子番号	端子名	選択機能	入出力	状態
15	P35/NMI	NMI	入力	

図 3.8 NMI を選択した端子の表示

注意

・図 3.7 の警告が表示されている場合、ソースファイルを生成することは可能ですが、この端子を NMI として使用することはできません。詳細については「3.2.4 端子設定に関する警告とエラー」を参照してください。

(4) 端子機能の配置を決めてから周辺機能を設定する場合

端子機能シートで端子機能の配置を指定してから周辺機能を設定すると、指定した場所に端子機能が割り当てられます。

例えば IRQ5 は P15,PA4,PD5,PE5 (80 ピン版、64 ピン版は P15,PA4, PE5) のいずれかの端子に割り当てることが可能です。IRQ5 を PE5に割り当てる場合、端子機能シートで PE5 の端子機能に IRQ5 を指定します。(図 3.9)

端子番号	端子名	選択機能	入出力	状態
⚠ 73	PE5/D13[A13/...	IRQ5		IRQ5 は周辺機能の設定で、使用するように設定されていません。

図 3.9 IRQ5 を選択した PE5 の表示 (ICUA 未設定)

割り込みコントローラ(ICUA)設定ウィンドウから IRQ5 を設定すると、IRQ5 は PE5 に割り当てられます。(図 3.10)

端子番号	端子名	選択機能	入出力	状態
73	PE5/D13[A13/...	IRQ5	入力	

図 3.10 IRQ5 を選択した PE5 の表示 (ICUA 設定後)

3.2.2 周辺機能別使用端子シート

周辺機能別使用端子シートは周辺機能ごとに端子の使用状況を表示します。左側の周辺機能一覧から選択した周辺機能に関連する端子機能と、それぞれの割当先が表示されます。割当先を複数のポートから選択することができる端子機能は、本シートで割当先を変更することができます。

図 3.11 端子機能ウィンドウ 周辺機能別使用端子シート

各カラムの表示内容を表 3.3 に示します。

表 3.3 周辺機能別使用端子シートの表示内容

カラム	内容
端子名	左側の周辺機能一覧で選択した周辺機能の端子機能名が表示されます。
端子機能	選択されている端子機能の内容が表示されます。
使用端子	割り当て先の端子名（端子に割り当ている全機能）が表示されます。
使用端子番号	割り当て先の端子番号が表示されます。
入出力	端子の入出力状態が表示されます。
状態	警告またはエラーが発生している場合はその内容が表示されます。

(1) 初期状態

初期状態（周辺機能が何も設定されていない状態）では、[端子機能]や[使用端子]カラムは空欄です。(図 3.12)

端子名	端子機能	使用端子	使用端子番号	入出力	状態
IRQ0					

図 3.12 周辺機能別使用端子シートの初期表示

(2) 端子機能の割り当て

端子の入出力に関連する周辺機能を設定すると、周辺機能で使用する端子機能がポートに割り当てられ、設定の結果がウィンドウに表示されます。例えば周辺機能の設定で外部割込み IRQ0 を設定すると、IRQ0 端子は P30 に割り当てられ図 3.13 に示すように表示されます。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
IRQ0	外部割り込み入力	P30/MTIOC4B/TMRI3/POE.	20	入力	

図 3.13 周辺機能が設定された端子機能の表示

注意

・初期状態(周辺機能および、端子機能シートでの端子機能の割り当てが設定されていない場合) から周辺機能を設定すると、「付録 1 割当先を変更できる端子機能」の「初期状態の割当先」に記載されているポートに端子機能が割り当てられます。周辺機能を設定する前に端子機能シートで端子機能の割り当て先を選択した場合は、選択したポートに割り当てられます。

この状態で I/O ポート設定ウィンドウから同じ端子を使用する汎用入出力ポート P30 を設定すると、図 3.14 に示すように端子機能の競合が警告されます。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
⚠ IRQ0	外部割り込み入力	P30/MTIOC4B/TMRI3/POE.	20	入力	他の機能と競合しています。

図 3.14 1 つの端子に複数の機能が割り当てられた場合の警告表示

注意

・一つの端子に複数の端子機能が割り当てられている場合(図 3.13 の状態)でもソースファイルの生成は可能です。この場合、複数の機能を同時に使用することはできませんが、端子機能を切り替えて使用することが可能です。詳細については「3.2.4 端子設定に関する警告とエラー」を参照してください。

IRQ0 は割当先を変更することができます。割当先を変更できる端子機能は、使用端子のセルにマウスポインタを置くと、割当先端子の選択肢を開くためのドロップダウンボタンが表示されます。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
⚠ IRQ0	外部割り込み入力	P30/MTIOC4B/TMRI3/P▾	20	入力	他の機能と競合しています。

図 3.15 ドロップダウンボタンの表示

端子機能の割当先を変更するには、ドロップダウンボタンをクリックし、表示された選択肢から割り当て先の端子を指定してください。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
⚠ IRQ0	外部割り込み入力	P30/MTIOC4B/TMRI3/P▾	20	入力	他の機能と競合しています。
		P30/MTIOC4B/TMRI3/POE8#/RXD1/SMISO1/SSCL1/IRQ0-DS/RTCIC0			
		PD0/D0[A0/D0]/IRQ0			
		PH1/TMO0/IRQ0			

図 3.16 割り当て先の選択肢

IRQ0 の割当先を PH1 に変更し、変更後の割当先端子が他の機能で使用されていなければ、競合状態を解決することができます。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
IRQ0	外部割り込み入力	PH1/TMO0/IRQ0	37	入力	

図 3.17 端子機能の割り当て先変更後の表示

割当先を変更できる端子機能を「付録 1 割当先を変更できる端子機能」に示します。

注意

・周辺機能が設定されていない状態(図 3.12 の状態)では、本シートから端子機能の割当先を変更することはできません。

3.2.3 端子配置図シート

端子機能シートはマイクロコントローラのパッケージ図で各端子の状態を表示します。割り当てる機能を複数から選択できるポートでは、本シートで割り当てる機能を選択することができます。

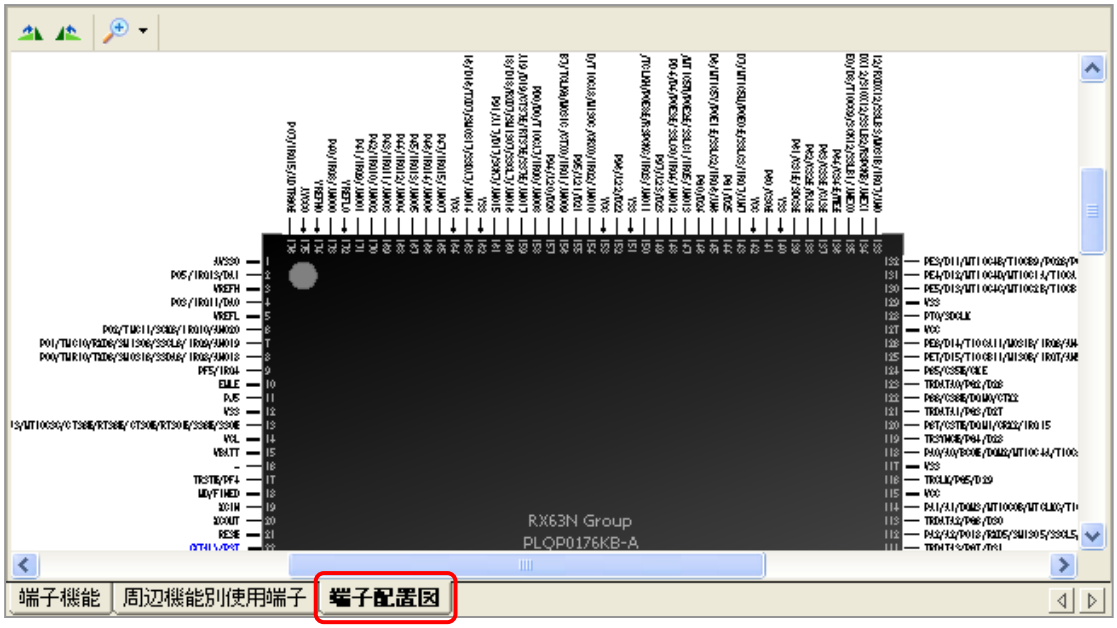


図 3.18 端子機能ウィンドウ 端子配置図シート

注意

TFLGAパッケージまたはLFBGAパッケージの製品を選択した場合、実際のパッケージとは異なり、LQFP パッケージの図が表示されます。このとき、図 3.19 に示すメッセージが表示されます。

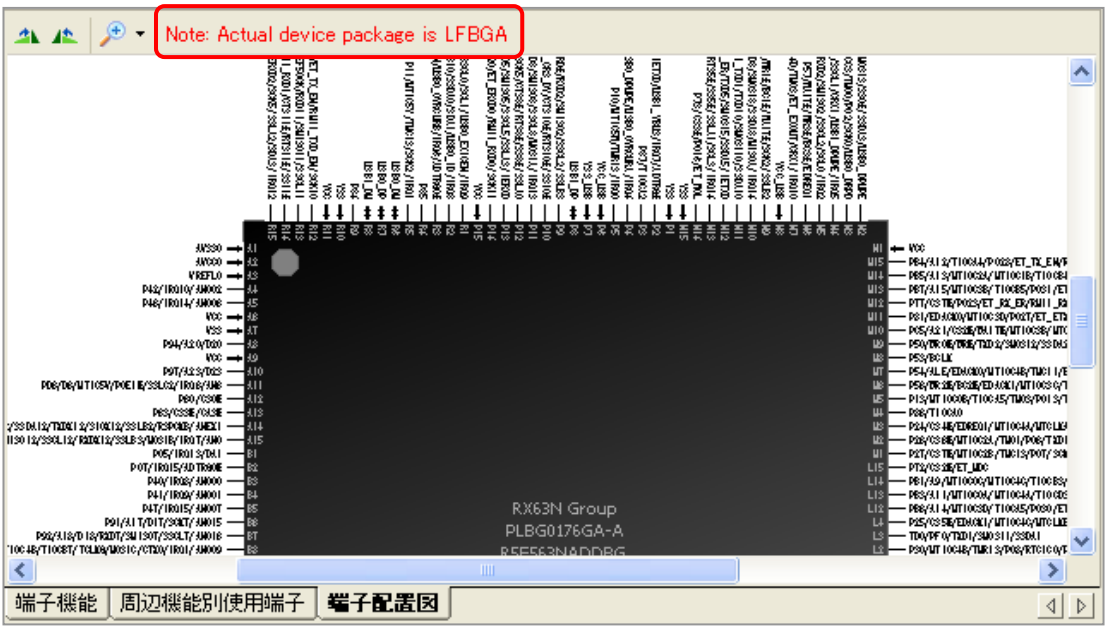


図 3.19 TFLGA パッケージおよび LFBGA パッケージ選択時の表示

(1) 機能

端子配置図シートには以下の機能があります。

・ 回転

回転ボタン( )により、右回りまたは左回りに表示を回転させることができます。

・ 拡大/縮小

拡大ボタン( )により、表示を拡大することができます。また、ドロップダウンリストから表示サイズを選択することができます。

(2) 端子機能の選択

割り当てる機能を複数の中から選択できるポートは、マウスポインタを端子上に置き、マウスの右ボタンをクリックすると、機能の選択肢が表示されます。(図 3.20)

図 3.20 端子機能の選択肢

本シートで機能を選択すると、他のシートに変更内容が反映されます。各シート間の連動については、「3.2.4 端子機能設定の連動」を参照してください。

(3) 端子状態の表示

端子の設定状態は以下のように表示されます。

・ 選択機能

本シートまたは他のシートから端子機能が設定された場合、図 3.21 のように選択されている機能が括弧で囲まれます。

図 3.21 選択機能の表示 (MTIOC2B 選択時)

・ 入出力状態

設定されている端子機能により、図 3.22 に示すように、各端子の入出力方向が表示されます。

a. 端子機能未設定　b. 出力　c. 入力　d. 入出力

図 3.22 入出力の表示

注意

1 つの端子に複数の機能が割り当てられ、警告が発生している場合は、入出力方向は表示されません。

・ 設定状態

各端子は設定状態に応じて、図 3.23 のように表示されます。

a. 機能が設定されていない場合（表示色:黒）

b. 機能が設定され、エラーまたは警告が発生していない場合（表示色:青）

c. 機能が設定され、警告が発生している場合（表示色:茶）

d. 機能が設定され、エラーが発生している場合（表示色:赤）

図 3.23 設定状態の表示

エラーと警告の内容は、端子機能シートの対応する端子を参照してください。端子設定に関するエラーと警告については「3.2.5 端子設定に関する警告とエラー」を参照してください。

3.2.4 ウィンドウ間の設定の連動

端子機能シートと周辺機能別使用端子シートは設定の変更が相互に連動します。端子機能シートで端子機能の割り当てを変更すると、周辺機能別使用端子シートの設定が変更されます。同様に、周辺機能別使用端子シートで端子機能の割当先を変更すると、端子機能シートの設定が変更されます。(図 3.24)

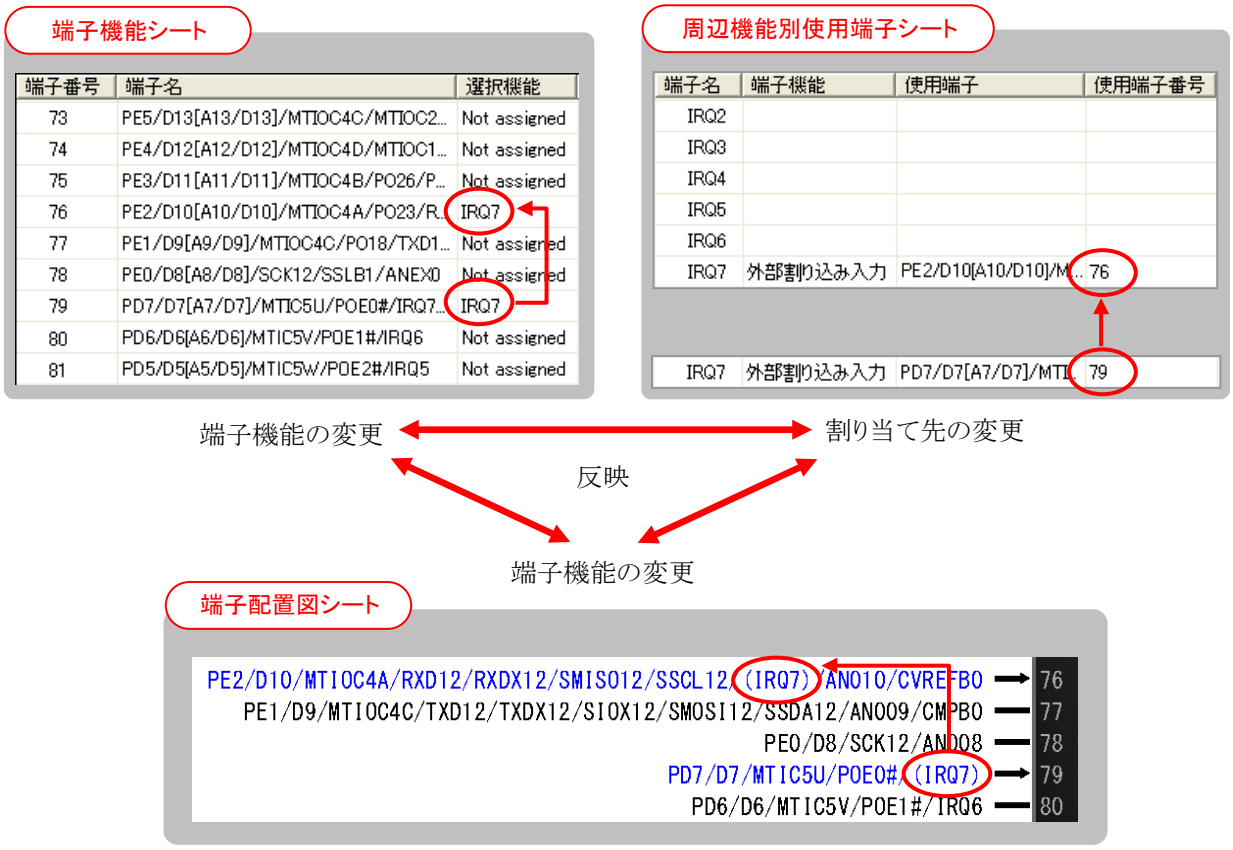


図 3.24 端子機能シート、端子配置図シート、周辺機能別使用端子シートの連動

各周辺機能の設定状態は、端子機能シートと周辺機能別使用端子シートに反映されます。例えば割り込みコントローラ(ICUb)の設定ウィンドウで IRQn の設定を行うと、周辺機能別使用端子シートで IRQn が使用された状態となり、割当先の状態が端子機能シートと周辺機能別使用端子シートに表示されます。

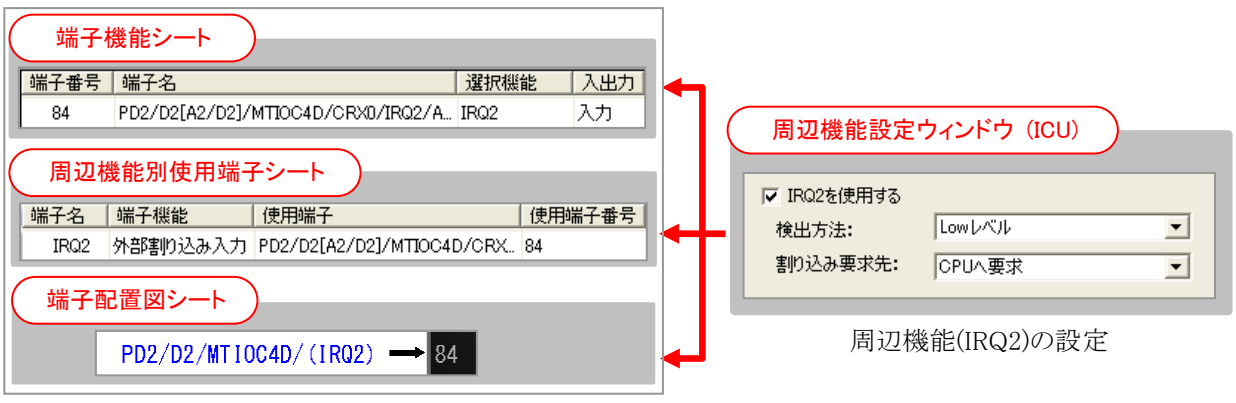


図 3.25 周辺機能の設定と端子機能の割り当て

割り込みコントローラ(ICUb)の設定ウィンドウで IRQnの設定を解除すると、端子機能シートと周辺機能別使用端子シートで IRQnの割り当てが解除されます。

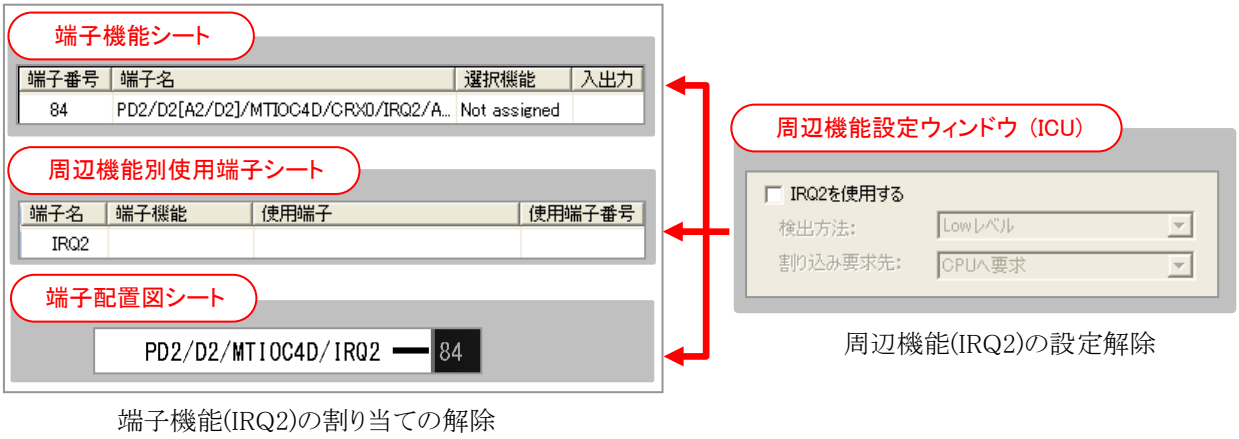


図 3.26 周辺機能の設定解除と端子機能の割り当て解除

一方、端子機能シートおよび周辺機能別使用端子シートの設定変更は、周辺機能の設定に反映されません。割り込みコントローラ(ICUb)の設定ウィンドウで IRQnを設定した状態で、端子機能シート（または端子配置図シート）から IRQnの割当先を"Not assigned"に変更しても、割り込みコントローラ(ICUb)の設定ウィンドウでは IRQnの設定は解除されません。この場合、IRQnはどこにも割り当てられていないため、エラーが表示されます。エラーについては「3.2.5 端子設定に関する警告とエラー」を参照してください。

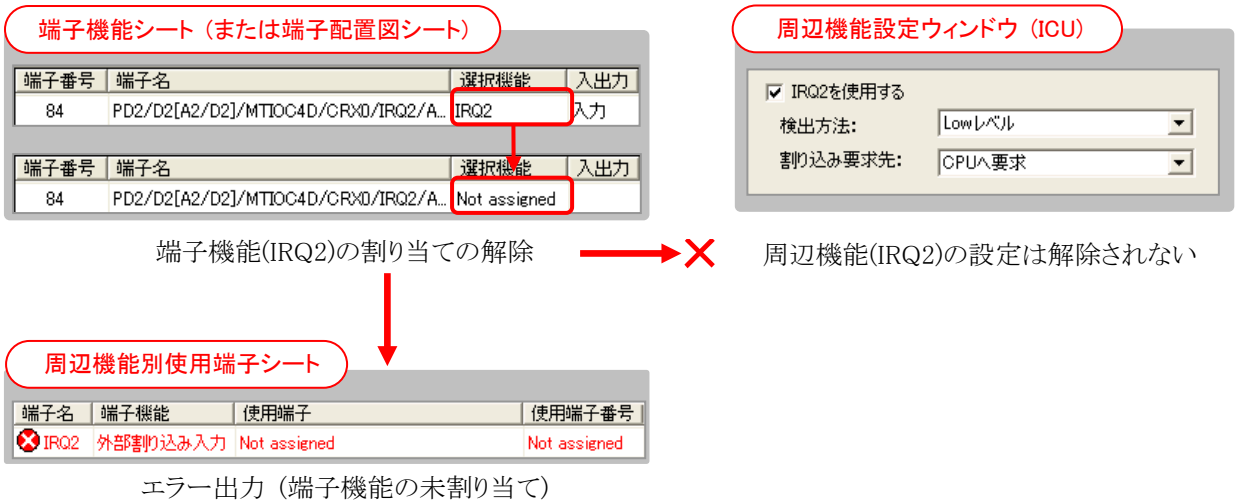


図 3.27 端子機能の割り当て解除とエラー表示

3.2.5 端子設定に関する警告とエラー

設定状態によっては端子機能シートおよび周辺機能別使用端子シートにエラーや警告が表示されます。エラーと警告の分類を表 3.4 に示します。

表 3.4 エラーおよび警告の分類

設定状態	分類	メッセージ
同一機能の複数端子への割り当て	エラー	“*<端子番号>*で同一の機能が選択されています。” （端子機能シート） “同一の機能を複数の端子に割り当てないでください。” （周辺機能別使用端子シート）
端子機能の未割り当て	エラー	“Not assigned” （周辺機能別使用端子シート）
1つの端子への複数機能の割り当て	警告	“複数の機能で競合しています。” （端子機能シート） “他の機能と競合しています。” （周辺機能別使用端子シート）
デバッガ用端子との競合	警告	“オンチップエミュレータ用端子と競合しています。” （端子機能シート） “オンチップエミュレータ用端子と周辺機能が競合しています。“ （周辺機能別使用端子シート）
周辺機能の未設定	警告	“*<端子機能>*は周辺機能の設定で、使用するように設定されていません。” （端子機能シート）

各エラーの内容を以下に示します。

(1) 同一機能の複数端子への割り当て

1 つの端子機能が複数の端子に割り当てられている場合はエラーとなり、ソースファイルを生成することはできません。端子機能シートで、その機能として使用しない端子の機能を別の機能または”Not assigned”に変更するか、周辺機能別使用端子シートで端子機能の割当先を選択しなおしてください。

端子番号	端子名	選択機能	入出力	状態
20	P30/MTIOC4B/TMRI3/POE8.	IRQ0	入力	20/86 で同一の端子機能が使用されています。
86	PD0/D0[A0/D0]/IRQ0	IRQ0	入力	20/86 で同一の端子機能が使用されています。

(a) 端子機能シート

端子名	端子機能	使用端子	使用端子番号	入出力	状態
IRQ0	外部割り込み入力	Conflicted	20/86	入力	同一の機能を複数の端子に割り当てないでください。

(b) 周辺機能別使用端子シート

図 3.28 エラー表示例（同一機能の複数端子への割り当て）

(2) 端子機能の未割り当て

設定された周辺機能が使用する端子機能が、どの端子にも割り当てられていない場合、エラーとなりソースファイルを生成することができません。端子機能シートで、割り当て先の端子の端子機能に、その機能を選択するか、周辺機能別使用端子シートで端子機能の割当先を指定してください。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
IRQ0	外部割り込み入力	Not assigned	Not assigned	入力	Not assigned.

周辺機能別使用端子シート

図 3.29 エラー表示例（端子機能の未割り当て）

(3) 1 つの端子への複数機能の割り当て

一つの端子に複数の端子機能が割り当てられている場合、警告が表示されますがソースファイルの生成は可能です。この場合、複数の機能を同時に使用することはできませんが、端子機能を切り替えて使用することが可能です。端子機能は各周辺機能の初期設定関数で設定されるため、初期設定を行った機能に切り替わります。ただし RTCOUT と RTCIC2 を同一の端子に割り当てることはできません。

端子番号	端子名	選択機能	入出力	状態
⚠ 20	P30/MTIOC4B/TMRI3/POE8.	P30/IRQ0		複数の機能で競合しています。

(a) 端子機能シート

端子名	端子機能	使用端子	使用端子番号	入出力	状態
⚠ IRQ0	外部割り込み入力	P30/MTIOC4B.	20	入力	他の機能と競合しています。

端子名	端子機能	使用端子	使用端子番号	入出力	状態
⚠ P30	汎用入力ポート	P30/MTIOC4B.	20	入力	他の機能と競合しています。

(b) 周辺機能別使用端子シート

図 3.30 警告表示例（1 つの端子への複数機能の割り当て）

(4) デバッガ用端子との競合

オンチップエミュレータが使用する端子に周辺機能の端子機能が割り当てられた場合、警告が表示されますがソースファイルの生成は可能です。オンチップエミュレータを使用する場合、同じ端子に割り当てられた端子機能を使用することができない場合があります。

端子番号	端子名	選択機能	入出力	状態
⚠ 21	P27/CS3#/MTIOC2B/TMCI3.	TMCI3		オンチップエミュレータ用端子と競合しています。

(a) 端子機能シート

端子名	端子機能	使用端子	使用端子番号	入出力	状態
⚠ TMCI3	カウントソースクロック入力	P27/CS3#..	21	入力	オンチップエミュレータ用端子と周辺機能が競合しています。

(b) 周辺機能別使用端子シート

図 3.31 警告表示例（デバッガ用端子との競合）

(5) 周辺機能の未設定

周辺機能を設定しない状態で、端子機能シート上で端子機能の割り当てのみを行った場合は警告が表示されます。この場合、ソースファイルの生成は可能ですが、その端子を指定した機能で使用することはできません。 端子機能を変更するためのレジスタの設定は、端子を使用する各周辺機能の初期設定関数の中で行われるため、 端子機能を有効にするには、その機能を使用する周辺機能を設定し、初期化関数を呼び出してください。

端子番号	端子名	選択機能	入出力	状態
⚠ 20	P30/MTIOC4B/...	IRQ0		IRQ0 は周辺機能の設定で、使用するように設定されていません

端子機能シート

図 3.32 警告表示例（周辺機能の未設定）

3.3 エンディアンの設定

周辺機能選択タブから[SYSTEM]を選択し、リソースウィンドウで[オプション設定]を選択すると、エンディアン設定ウィンドウが開きます。

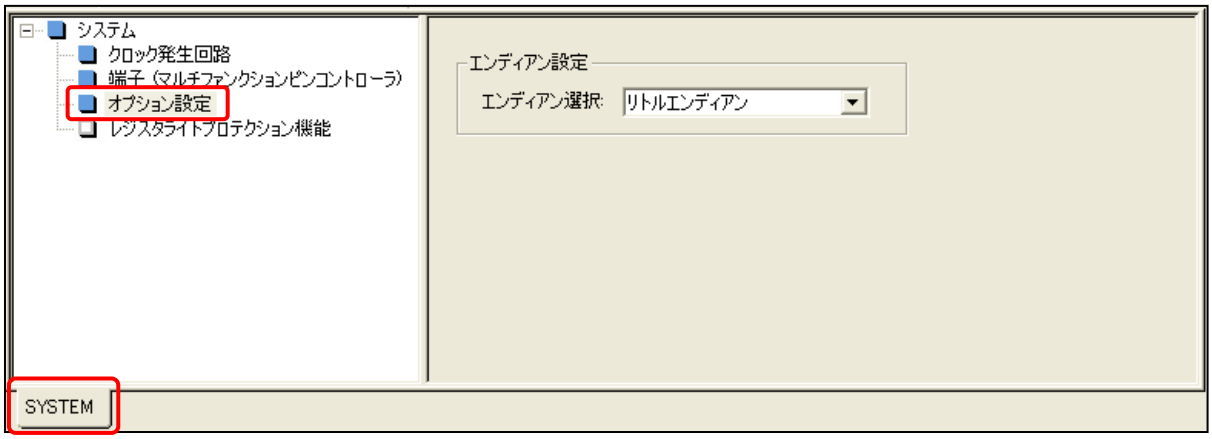


図 3.33 エンディアンの設定方法

ここでは使用するエンディアンを選択してください。

本設定はリンクする Renesas Peripheral Driver Library のライブラリファイルの選択(xxx_little.lib または xxx_big.lib)にのみ使用され、出力されるソースコードには影響しません。

4. チュートリアル

本章では、Peripheral Driver Generator と HEW を使用して RX210 用 Renesas Starter Kit のボードを動作させる以下のチュートリアルプログラムの作成方法を示しながら、Peripheral Driver Generator の使用手順を紹介します。

- 8 ビットタイマ(TMR)の割り込みで LED を点滅
- マルチファンクションタイマパルスユニット 2 (MTU2a)の PWM 波で LED を点滅
- 12 ビット A/D コンバータ（S12ADb）の連続スキャン
- ICUb による DTCa 転送のトリガ
- SCIc チャネル 0 とチャネル 5 で調歩同期通信

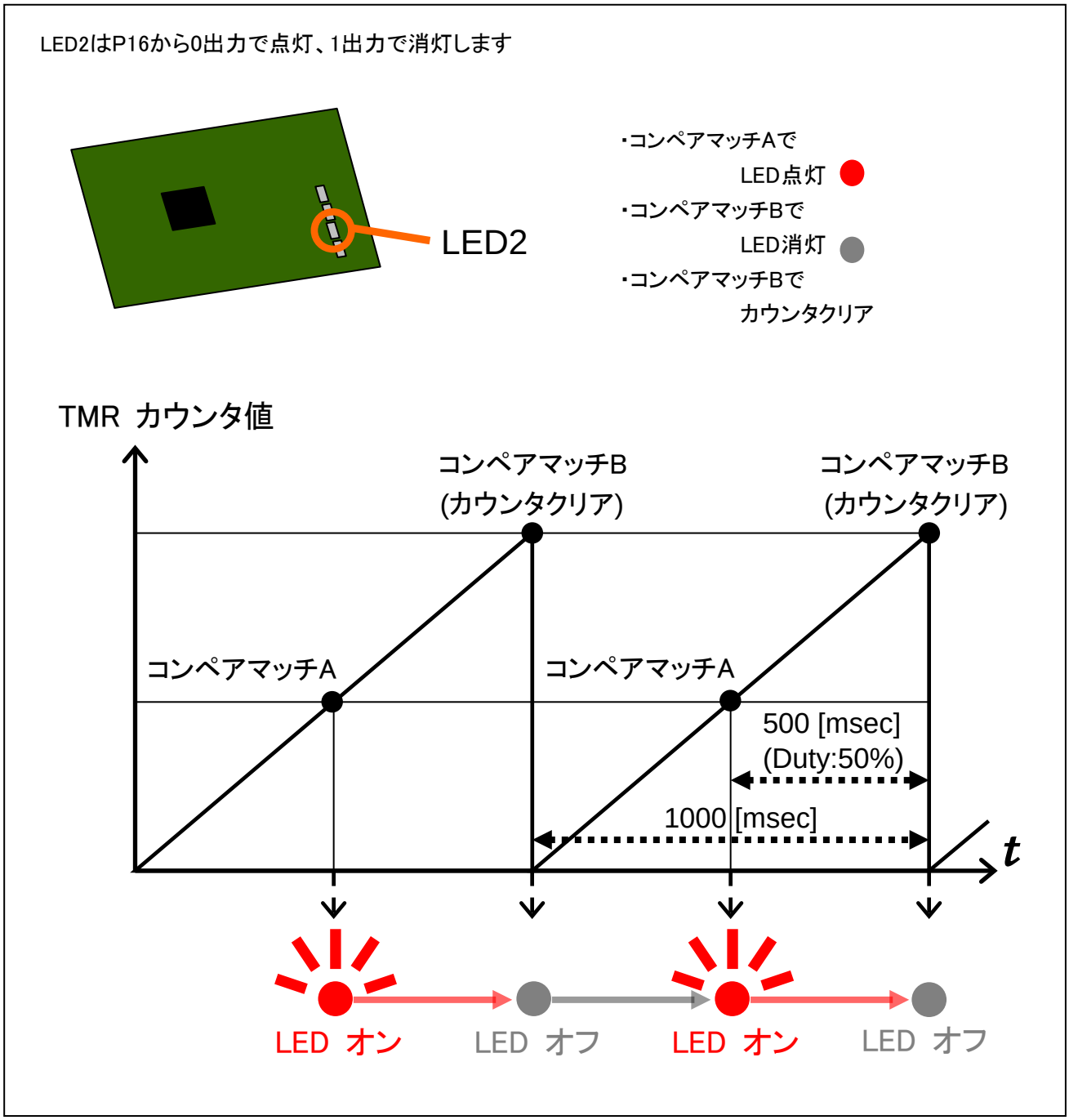
説明の中にある以下の表示はそれぞれ Peripheral Driver Generator、High-performance Embedded Workshop 上での操作をあらわします。

PDG	: Peripheral Driver Generator上の操作をあらわします
HEW	: High-performance Embedded Workshop上の操作をあらわします

4.1 8 ビットタイマ(TMR)の割り込みでLEDを点滅

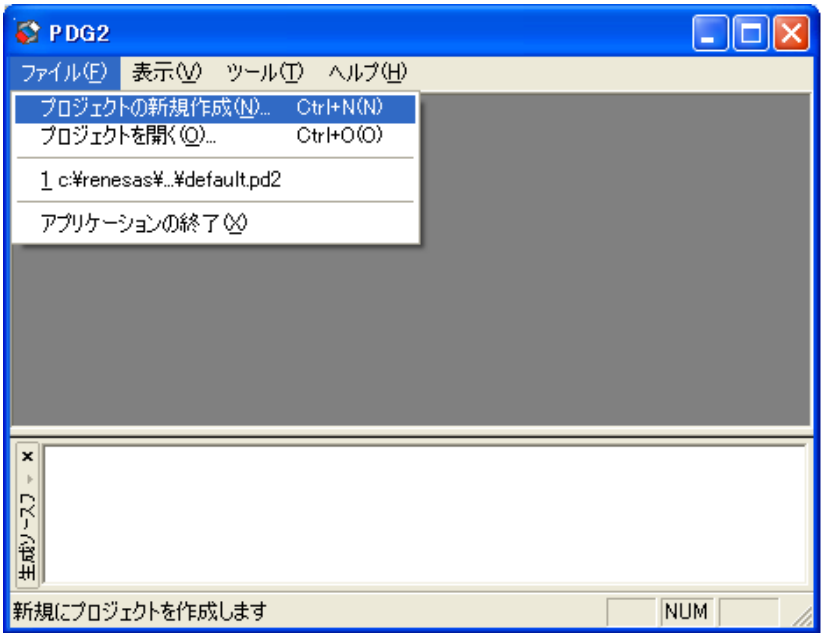
RSK ボード上の LED2 は P16 に接続されています。このチュートリアルでは 8 ビットタイマ(TMR)と I/O ポートを設定し、LED を次のように点滅させます。

使用する RSK ボード上に P16 の有効/無効を切り替えるスイッチがある場合は有効にしてください。



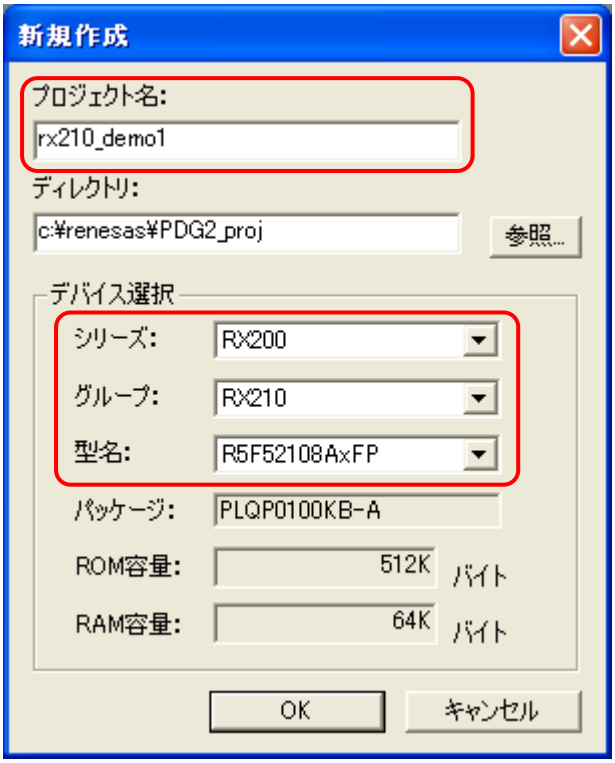
(1) Peripheral Driver Generator プロジェクトの作成 PDG

1. Peripheral Driver Generator を起動してください。
2. メニューから［ファイル］->［プロジェクトの新規作成］を選択してください。



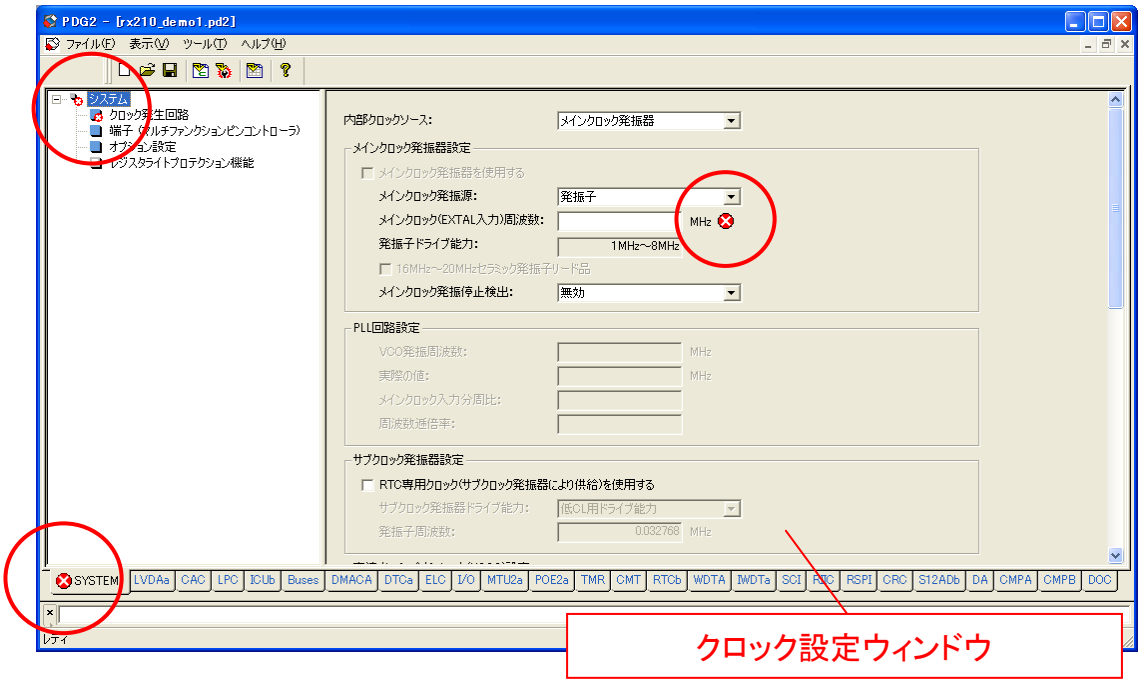
3. プロジェクト名に“rx210_demo1”を指定してください。
CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。

シリーズ ：RX200
グループ ：RX210
型名 ：R5F52108AxFP

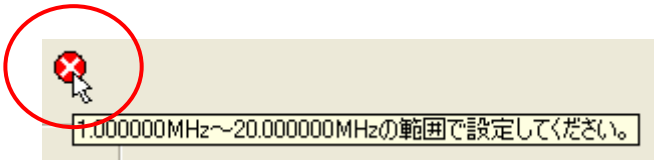


(2) 初期状態 **PDG**

・プロジェクトの作成直後はクロック設定ウィンドウが開き、エラーアイコンが表示されます。



・エラーアイコンの上にマウスポインタを置くと、エラーの内容が表示されます。



Peripheral Driver Generator には3 種類のアイコンがあります。

エラー
設定は許可されません。
設定にエラーがある場合、ソースファイルの生成はできません。

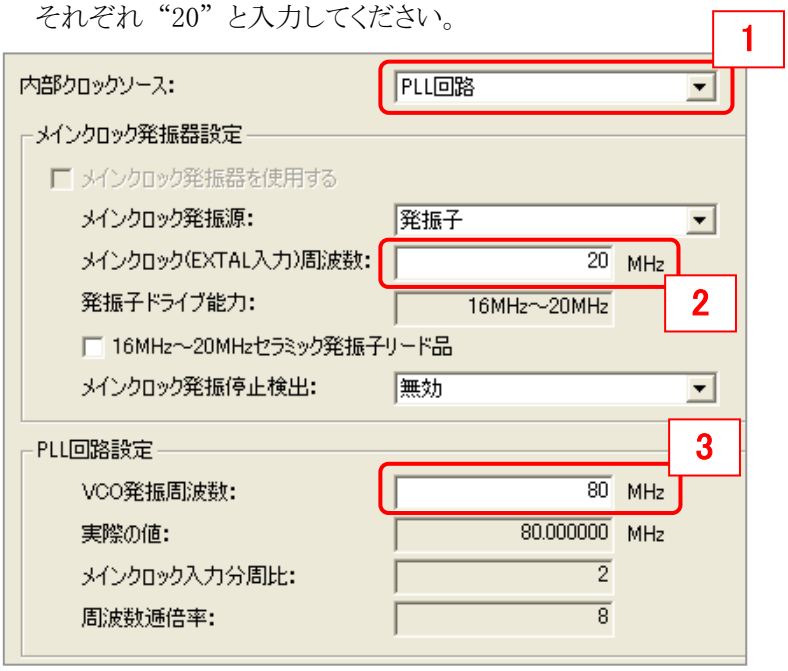
警告
設定は可能ですが、誤っている可能性があります。
ソースファイルの生成は可能です。

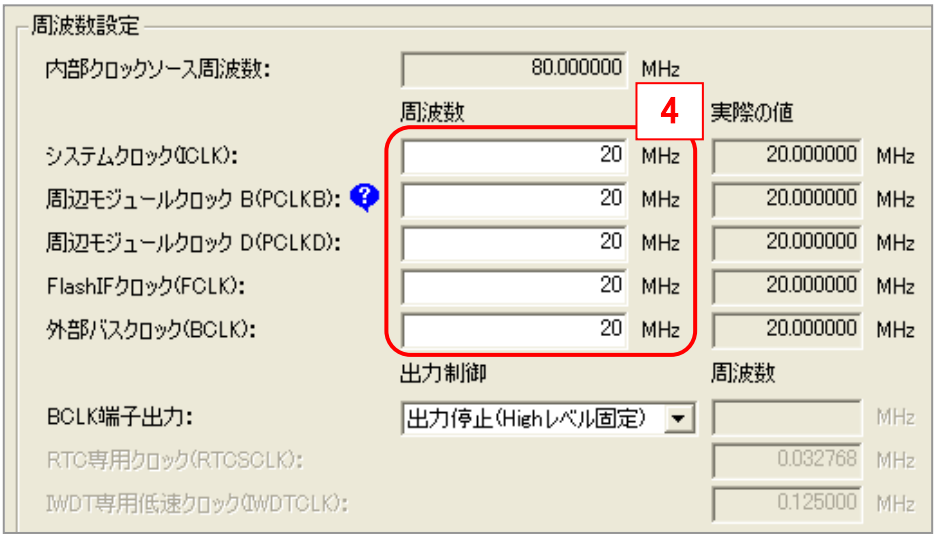
インフォメーション
複雑な設定箇所の付加情報です。

設定ウィンドウ上のアイコンのみツールチップを表示できます。

(3) クロックの設定 PDG

1. 内部クロックソースとして PLL 回路を選択してください。
2. メインクロック(EXTAL 入力)周波数を設定してください。
   RSK ボードの外部入力周波数は 20MHz です。“20”と入力してください。
3. VCO 発振周波数を設定してください。
   このチュートリアルでは 80MHz を設定します。“80”と入力してください。
4. システムクロック(ICLK)、周辺モジュールクロック B(PCLKB)、周辺モジュールクロック D(PCLKD)、FlashIF クロック(FCLK)、外部バスクロック(BCLK)はそれぞれ 20MHz で使用します。
   それぞれ“20”と入力してください。





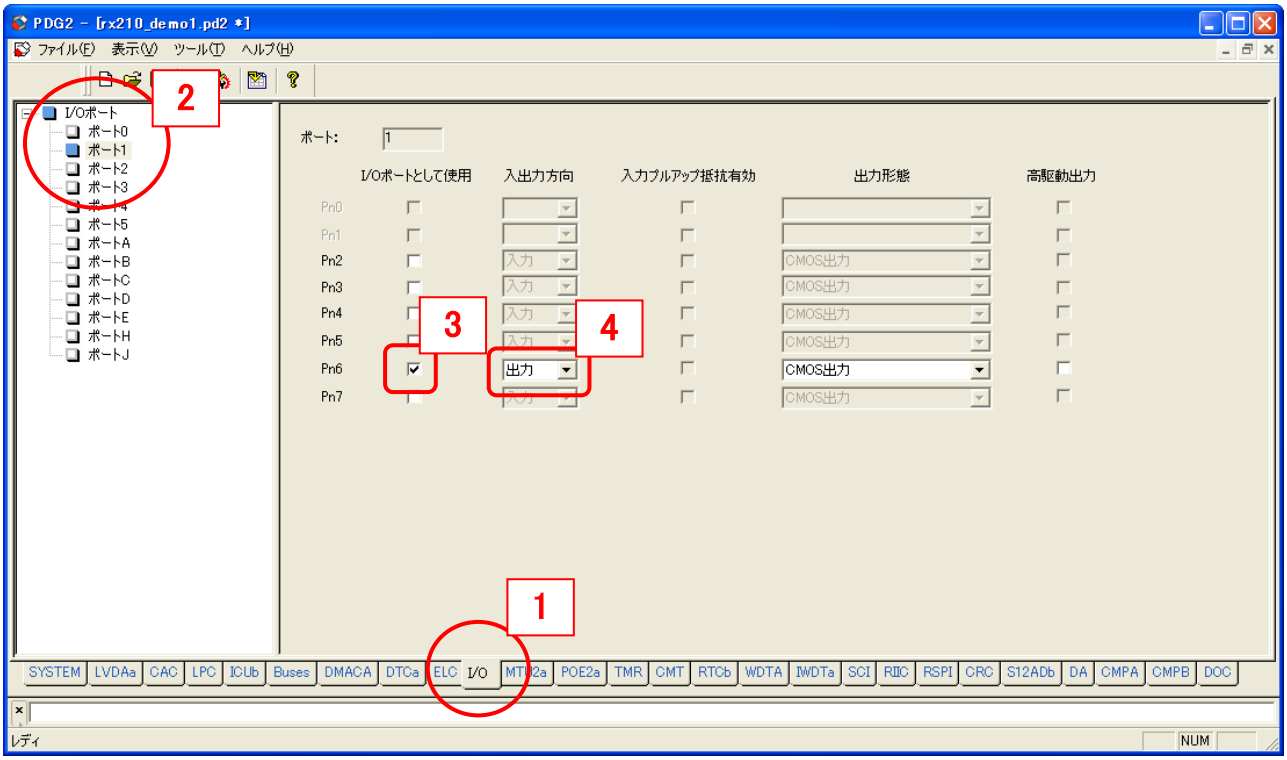
(4) エンディアンの設定 PDG

エンディアンの設定については、「3.3 エンディアンの設定」を参照してください。

(5) I/O ポートの設定 PDG

LED2 が接続されている P16 を出力ポートに設定します。

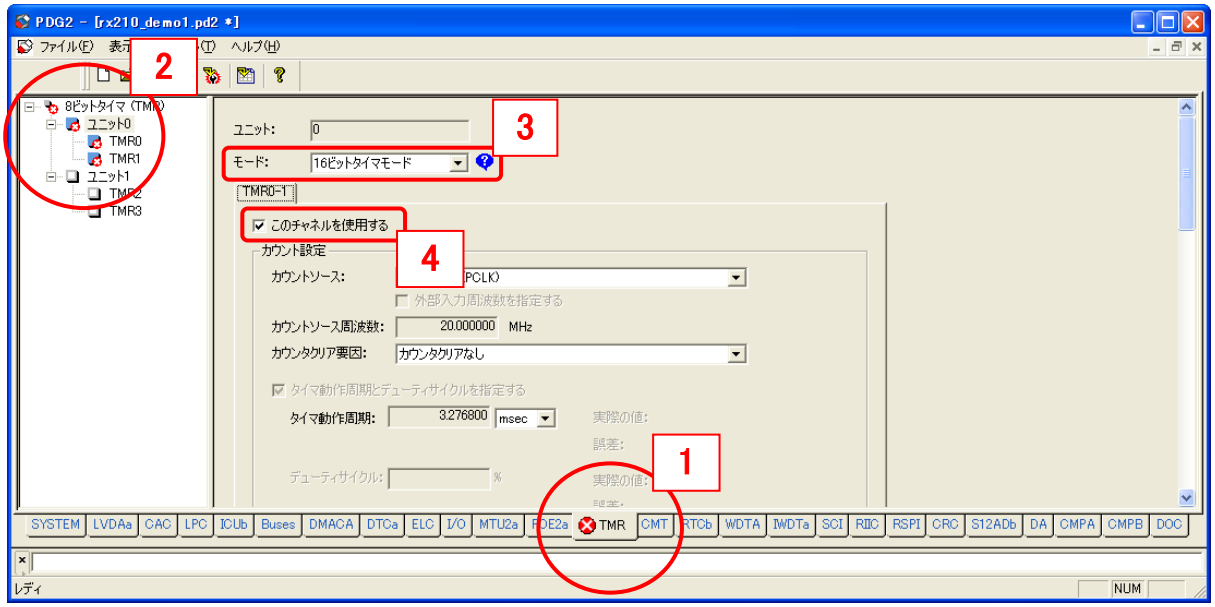
1. [I/O] タブを選択してください
2. [ポート1] を選択してください
3. [Pn6] をチェックしてください
4. [出力] を選択してください



(6) TMR の設定-1 PDG

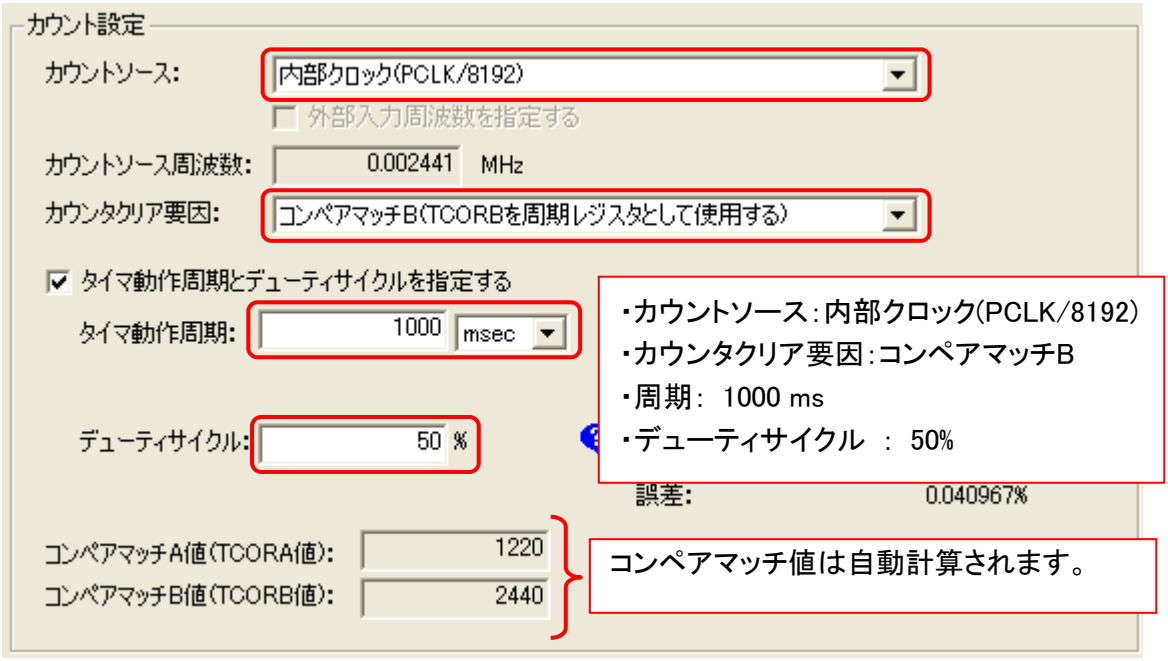
このチュートリアルでは TMR (8 ビットタイマ) のユニット 0 を 16 ビットタイマモード (2 つの 8 ビットタイマをカスケード接続するモード)で使用します。

1. [TMR] タブを選択してください。
2. [ユニット0] を選択してください。
3. [16ビットタイマモード] を選択してください。
4. [このチャネルを使用する] を選択してください。



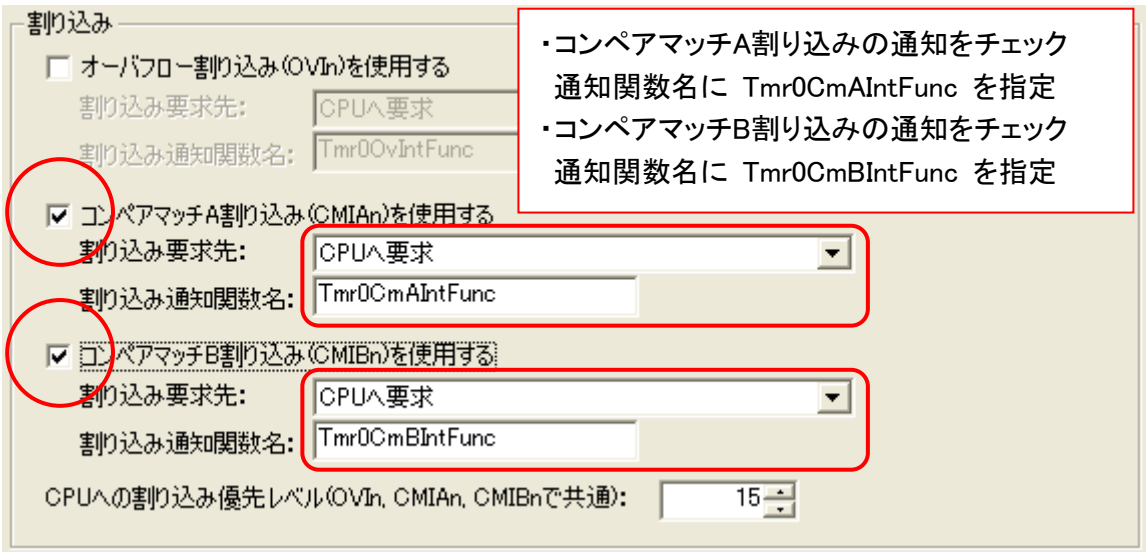
(7) TMR の設定-2 PDG

TMR の他の項目を以下の通り設定してください。



(8) TMR の設定-3 PDG

割り込み通知関数を設定します。
これらの関数は割り込みが発生すると呼ばれます。



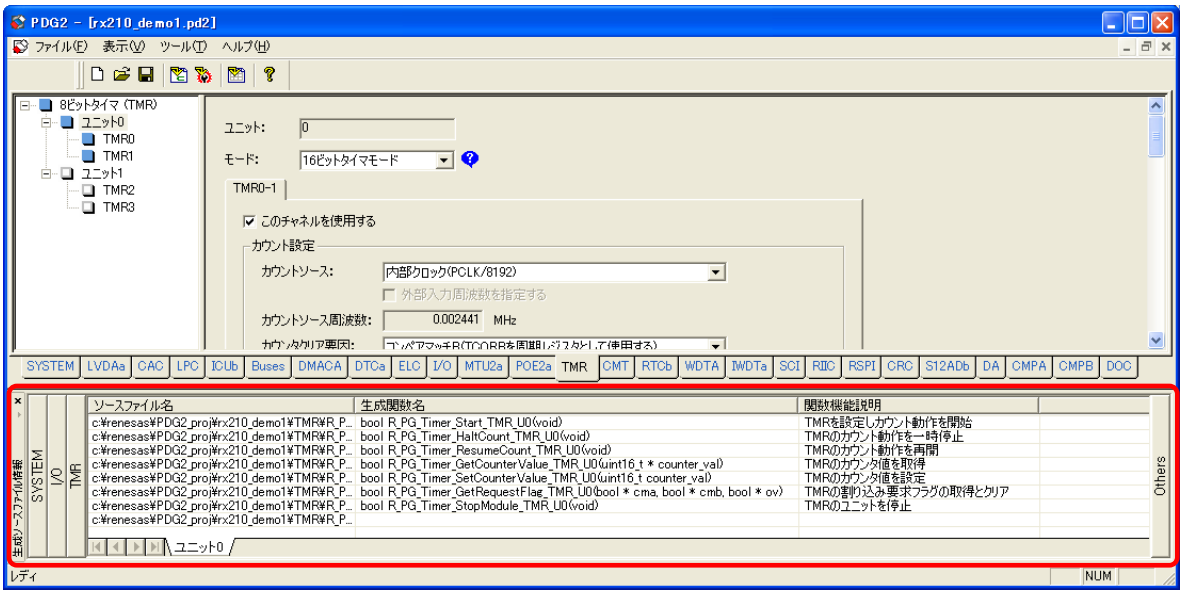
(9) ソースファイルの生成 **PDG**

1. ツールバー上の  をクリックするとソースファイルが生成されます。

2. プロジェクトの保存を確認するダイアログボックスが表示されます。[はい]をクリックしてください。

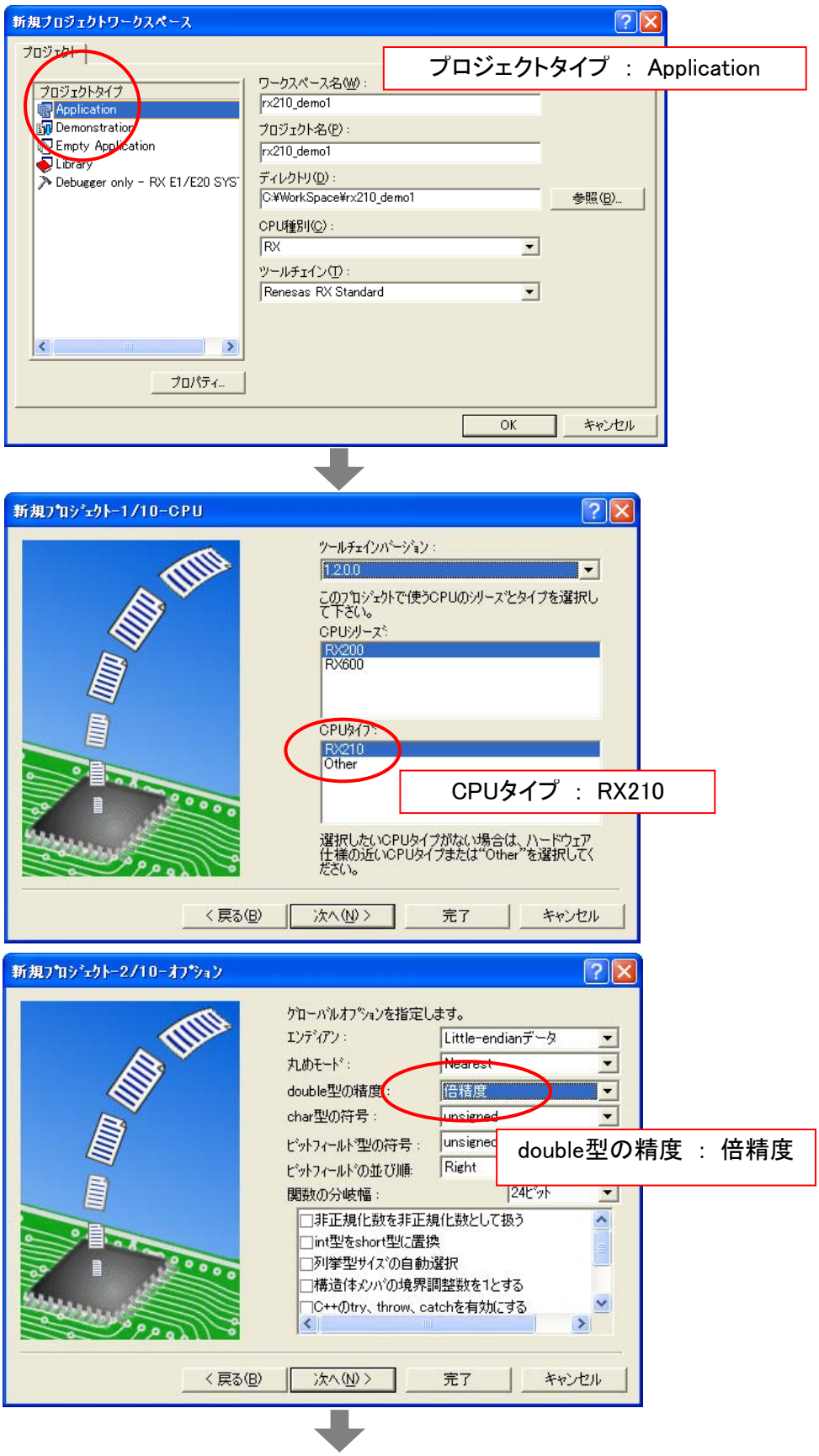
3. 登録の完了を示すダイアログボックスが表示されます。 [OK]をクリックしてください。

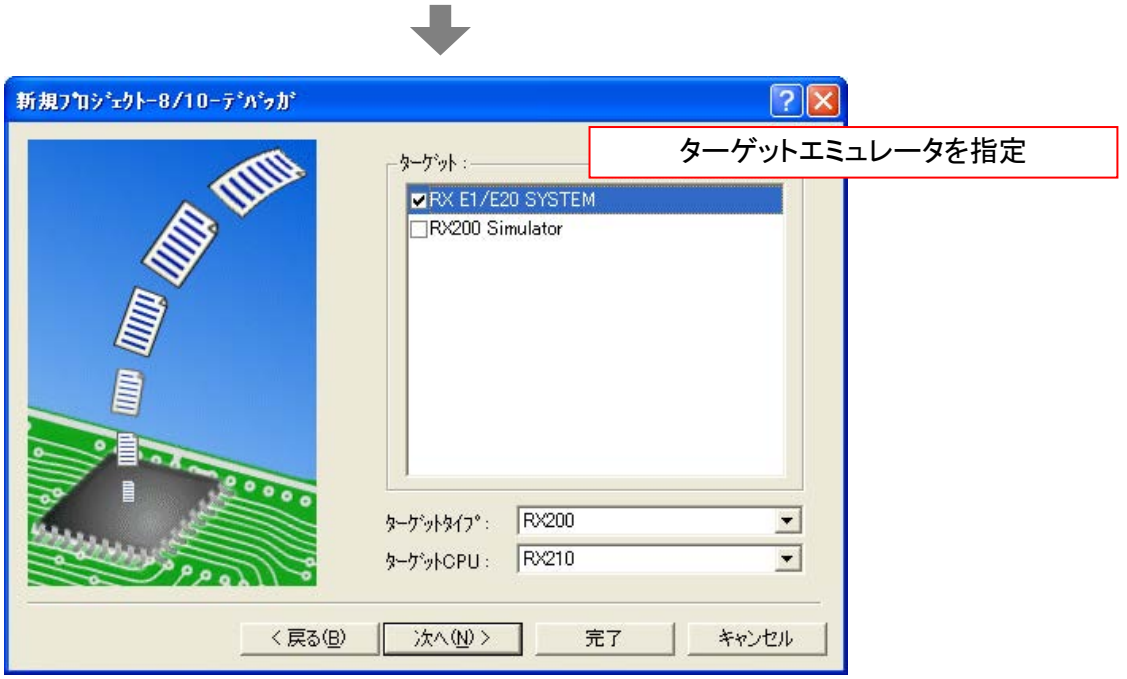
4. 生成された関数が下部のウィンドウに表示されます。
   関数をダブルクリックするとソースファイルが開きます。

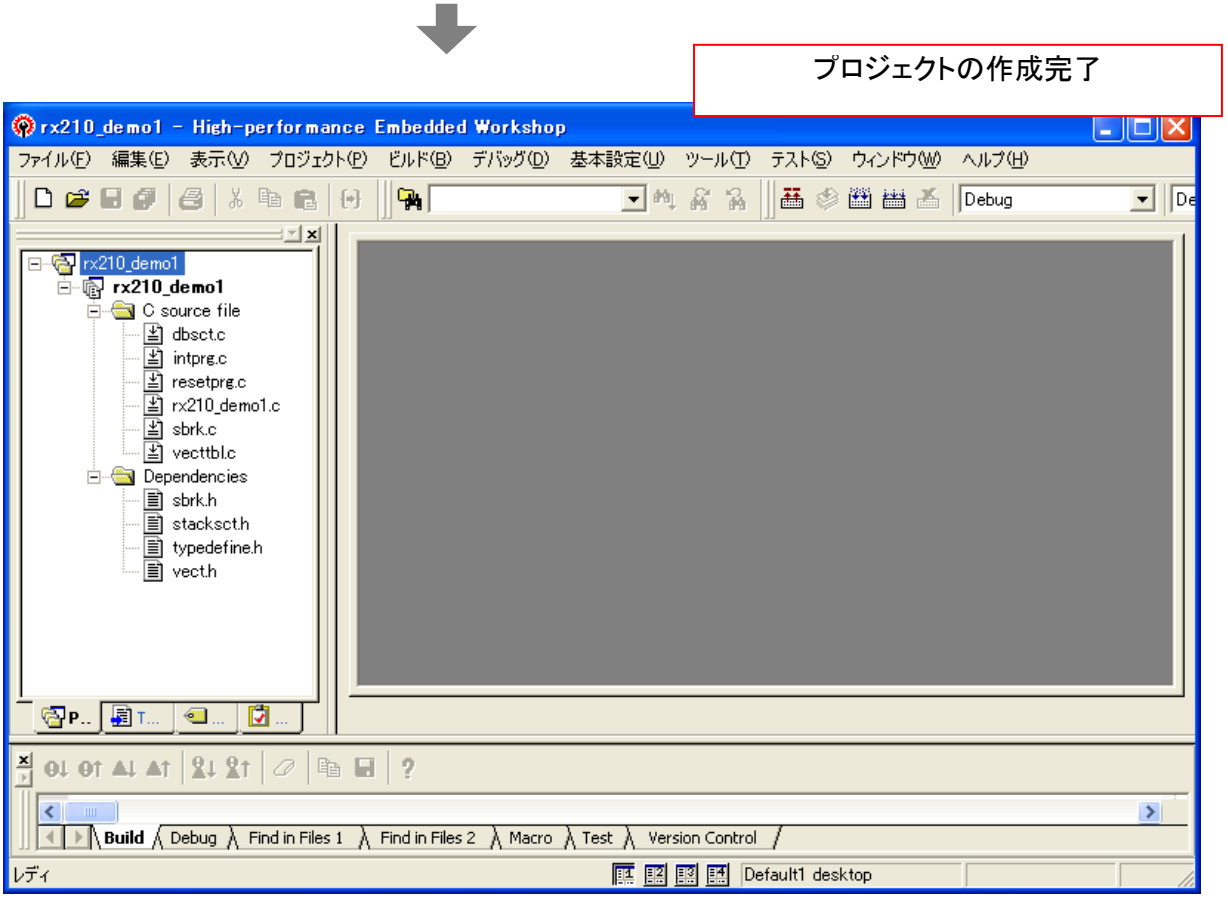


(10) High-performance Embedded Workshop プロジェクトの準備 HEW

High-performance Embedded Workshop を起動し、RX210 用の新規ワークスペースを作成します。



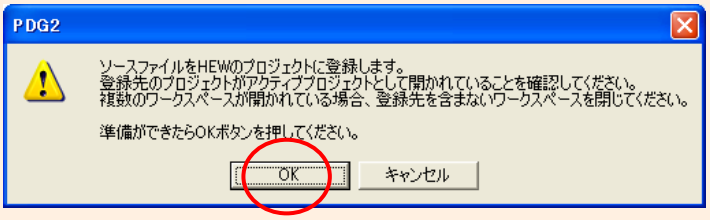

新規ﾌﾟﾛｼﾞｪｸﾄ-8/10-ﾃﾞﾊﾞｯｶﾞ
ターゲットエミュレータを指定
ﾀｰｹﾞｯﾄ :
RX E1/E20 SYSTEM
RX200 Simulator
ﾀｰｹﾞｯﾄﾀｲﾌﾟ :
RX200
ﾀｰｹﾞｯﾄCPU :
RX210
< 戻る(B)
次へ(N) >
完了
キャンセル


プロジェクトの作成完了
rx210_demo1 - High-performance Embedded Workshop
ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) 基本設定(U) ツール(T) テスト(S) ウィンドウ(W) ヘルプ(H)
Debug
rx210_demo1
rx210_demo1
C source file
dbsct.c
intprg.c
resetprg.c
rx210_demo1.c
sbrk.c
vecttbl.c
Dependencies
sbrk.h
stacksct.h
typedefine.h
vect.h
Build
Debug
Find in Files 1
Find in Files 2
Macro
Test
Version Control
レディ
Default1 desktop

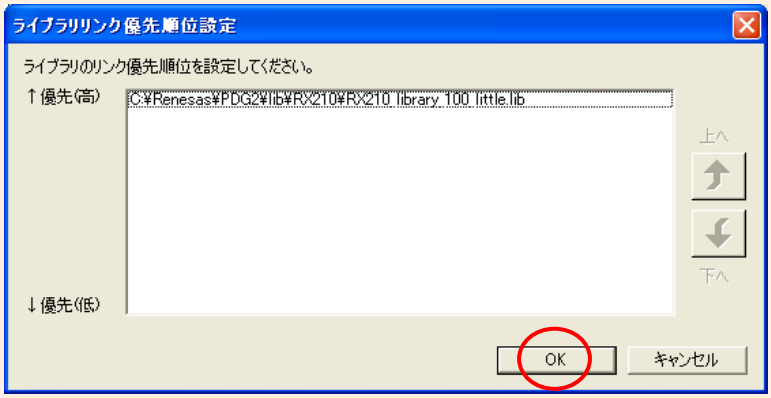
(11) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録

PDG

1. ファイルを High-performance Embedded Workshop に追加するには
   Peripheral Driver Generator のツールバー上の  をクリックします。
2. 確認のダイアログボックスで[OK]をクリックしてください。

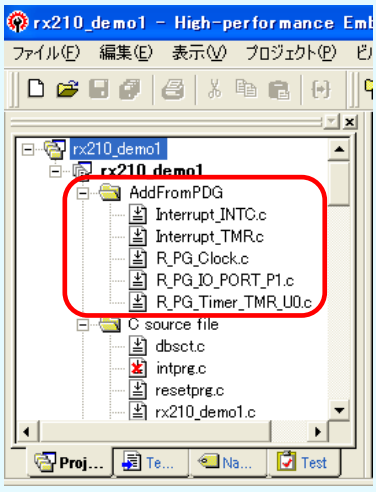


3. Renesas Peripheral Driver Library とのリンク設定のためのダイアログが開きます。
   複数の lib ファイルとリンクする場合、このダイアログ上でリンク順を設定できます。



HEW

4. High-performance Embedded Workshop のプロジェクトにファイルが追加されます。
   追加されたファイルは
   AddFromPDG
   フォルダに格納されます。



ソースファイルはHEW Target Server経由で追加されます。追加を実行する前にHEW Target Serverが設定されていることを確認してください。詳細についてはPeripheral Driver Generatorのユーザーズマニュアルを参照してください。

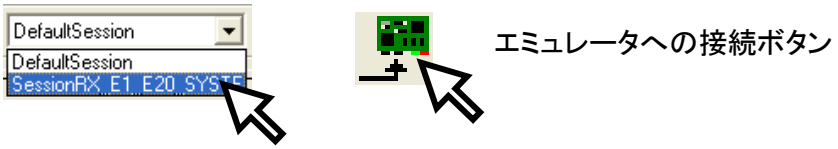
(12) プログラムの作成 **HEW**

High-performance Embedded Workshop 上で main 関数部分を変更し、以下のプログラムを作成してください。

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx210_demo1.h"

void main(void)
{
    //存在しないポートの設定
//  R_PG_IO_PORT_SetPortNotAvailable();

    //クロックの設定(発振安定時間ウェイト)
    R_PG_Clock_WaitSet(0.01);

    //ポートP16の設定
    R_PG_IO_PORT_Write_P16(1);
    R_PG_IO_PORT_Set_P1();

    //TMRユニット0を設定しカウントを開始
    R_PG_Timer_Start_TMR_U0();

    while(1);
}

//コンペアマッチA割り込みの通知関数
void TmrOCmAIntFunc(void)
{
    //LED点灯
    R_PG_IO_PORT_Write_P16(0);
}

//コンペアマッチB割り込みの通知関数
void TmrOCmBIntFunc(void)
{
    //LED消灯
    R_PG_IO_PORT_Write_P16(1);
}
```

(13) エミュレータの接続、プログラムのビルド、実行 **HEW**

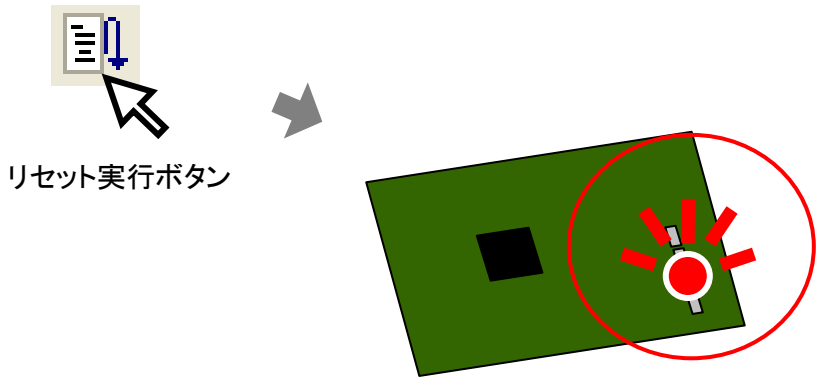
1. エミュレータに接続してください。



2. Renesas Peripheral Driver Libraryのライブラリとインクルードディレクトリはソースの登録時に設定されているため、[ビルド]ボタンをクリックするだけでビルドすることができます。

3. プログラムをダウンロードしてください。

4. プログラムを実行し、RSKボード上のLEDを確認してください。

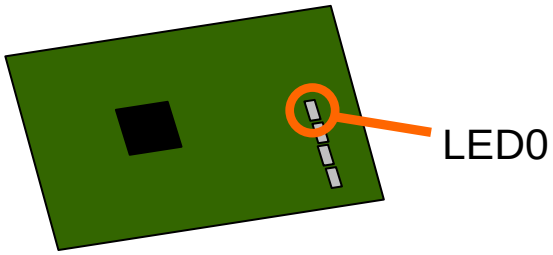


4.2 マルチファンクションタイマパルスユニット 2 (MTU2a)のPWM波でLEDを点滅

RX210 RSK ボードでは P14 端子に LED0 が接続されています。P14 はマルチファンクションタイマパルスユニット 2 (MTU2a)の PWM 波形出力端子(MTIOC3A)としても使用することができます。このチュートリアルではマルチファンクションタイマパルスユニット 2 (MTU2a)を PWM モード 1 で動作させ、その出力パルスで LED を点滅させます。

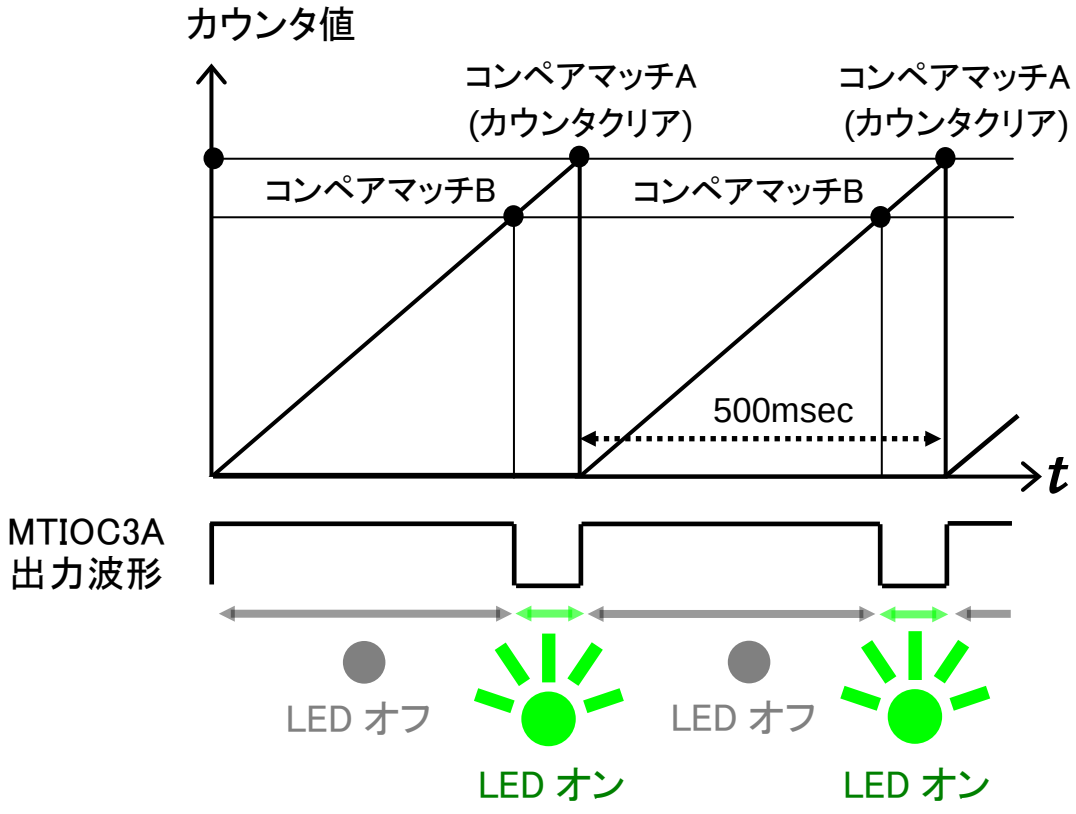
使用する RSK ボード上に P14(MTIOC3A)の有効/無効を切り替えるスイッチがある場合は有効にしてください。

LED0はP14から0出力で点灯、1出力で消灯します



MTU2aのチャネル3(MTU3)をPWMモード1で動作させます。PWMモード1は、コンペアマッチAおよびBでMTIOC3Aの出力レベルを制御するモードです。

設定するタイマの動作
- ・コンペアマッチBで0出力 → LED点灯
- ・コンペアマッチAで1出力 → LED消灯
- ・コンペアマッチAでカウンタクリア （カウンタクリア周期は500msec）

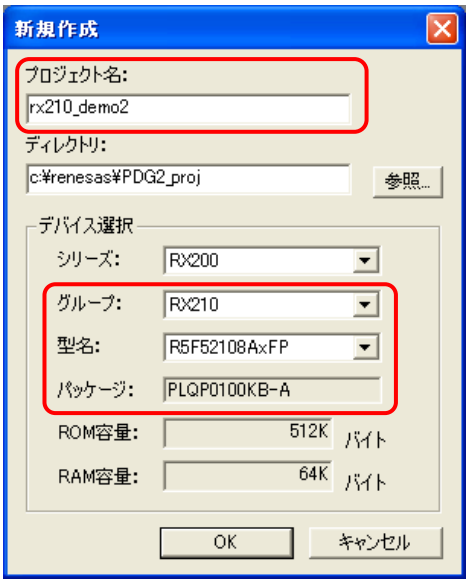


(1) Peripheral Driver Generator プロジェクトの作成 **PDG**

プロジェクト名に“rx210_demo2”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1 (1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。

シリーズ ：RX200
グループ ：RX210
型名 ：R5F52108AxFP



(2) クロックの設定 **PDG**

1. プロジェクトを作成するとクロック設定ウィンドウが開きます。設定画面上の や などのアイコンについては、「4.1 (2)初期状態」を参照してください。
2. クロックの設定については、「4.1 (3)クロックの設定」を参照してください。

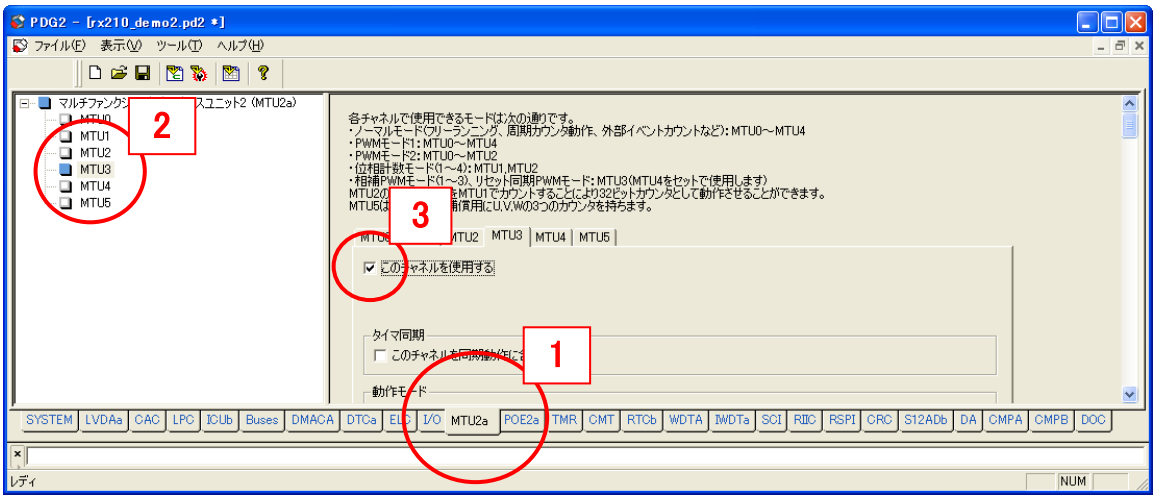
(3) エンディアンの設定 **PDG**

エンディアンの設定については、「3.3 エンディアンの設定」を参照してください。

(4) MTU2a の設定-1 **PDG**

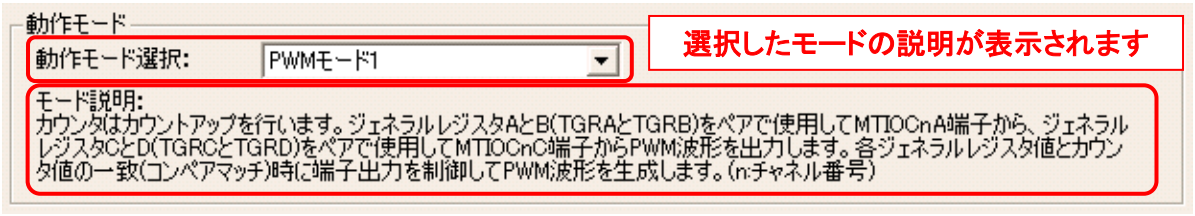
MTU2a チャネル 3(MTU3)を設定します。

1. [MTU2a] タブを選択してください。
2. [MTU3] を選択してください。
3. [このチャネルを使用する] をチェックしてください。



(5) MTU2a の設定-2 **PDG**

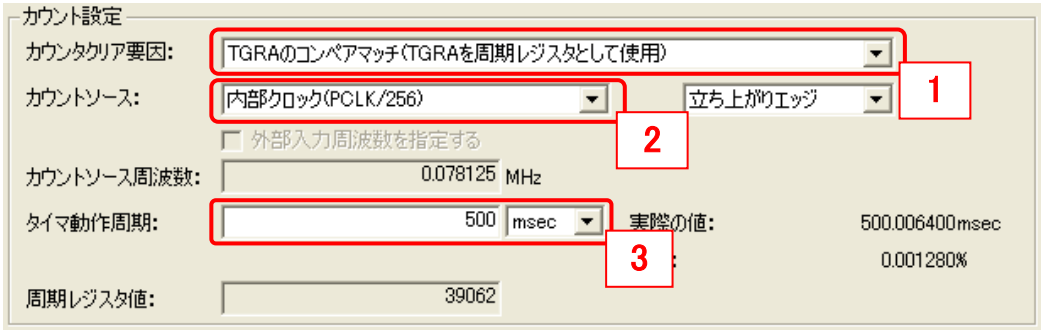
動作モードに[PWM モード 1]を指定してください。



(6) MTU2a の設定-3 **PDG**

以下の通りカウンタの動作を設定してください。

1. カウンタクリア要因に[TGRAレジスタのコンペアマッチ] を選択してください。
2. カウントソースに[内部クロック(PCLK/256)] を選択してください。
3. タイマ動作周期に500msecを指定してください。



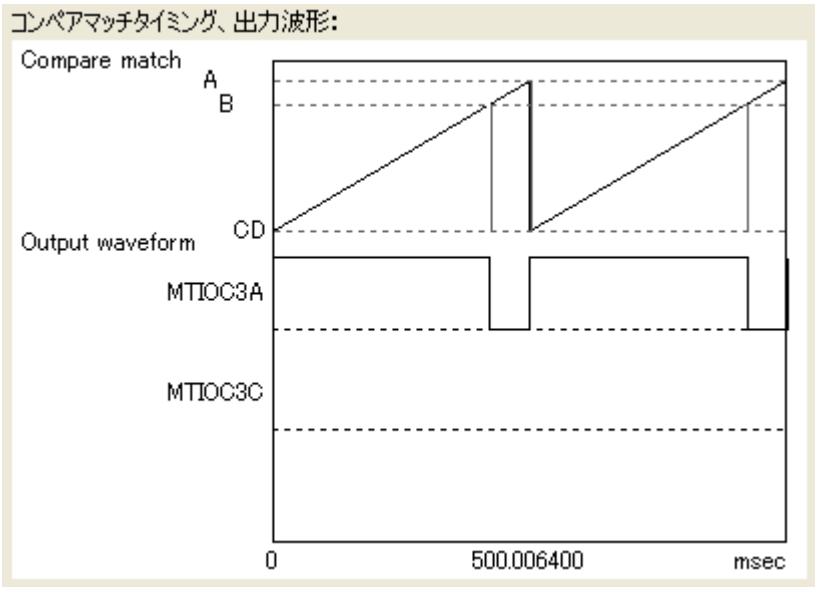
(7) MTU2a の設定-4 **PDG**

以下の通りジェネラルレジスタを設定してください。

1. カウント設定においてカウンタクリア要因にコンペアマッチAを指定したので、TGRAの値はカウントソース周波数と入力したタイマ動作周期を元に算出されます。
2. TGRAのアウトプットコンペア動作に[MTIOCnA端子の初期出力1、コンペアマッチで1出力]を選択してください。
3. TGRBのレジスタ初期値に33000を設定してください。
4. TGRBのアウトプットコンペア動作に[MTIOCnA端子からコンペアマッチで0出力]を選択してください。
5. TGRCとTGRDをペアで使用すると、MTIOCnC端子からPWM出力することが可能です。ここでは使用しませんので、TGRDのアウトプットコンペア動作には[MTIOCnC端子出力無効]を選択してください。

(8) MTU2a の設定-5 PDG

設定内容に応じて、コンペアマッチのタイミングと出力波形が図示されます。



(9) ソースファイルの生成 PDG

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1 (9)ソースファイルの生成」を参照してください。

(10) High-performance Embedded Workshop プロジェクトの準備 HEW

High-performance Embedded Workshop を起動し、RX210 用のワークスペースを作成してください。作成方法については「4.1 (10) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

(11) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1 (11) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(12) プログラムの作成 HEW

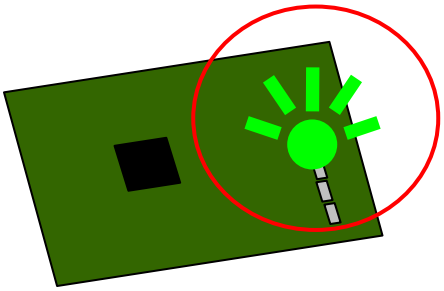
High-performance Embedded Workshop 上で main 関数の部分を変更し、以下のプログラムを作成してください。

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx210_demo2.h"

void main(void)
{
    //存在しないポートの設定
//  R_PG_IO_PORT_SetPortNotAvailable();

    //クロックの設定(発振安定時間ウェイト)
    R_PG_Clock_WaitSet(0.01);

    //MTU2aチャネル3の設定
    R_PG_Timer_Set_MTU_U0_C3();

    //MTU2aチャネル3のカウント開始
    R_PG_Timer_StartCount_MTU_U0_C3();

    while(1);
}
```

(13) エミュレータの接続、プログラムのビルド、実行 HEW

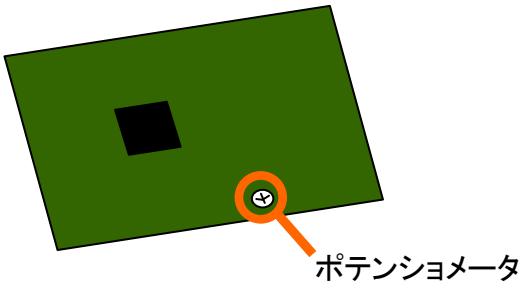
作成したプログラムをビルドし、実行してください。 LED が点滅します。

エミュレータの接続、プログラムのビルド、実行の方法については「4.1 (13) エミュレータの接続、プログラムのビルド、実行」 を参照してください。



4.3 12 ビットA/Dコンバータ（S12ADb）の連続スキャン

RX210 RSK ボードではポテンショメータが AN000 アナログ入力端子に接続されています。
このチュートリアルでは AN000 の A/D 変換を連続スキャンし、A/D 変換結果を High-performance Embedded Workshop 上でリアルタイム
に確認します。



使用する RSK ボード上に AN000 の有効/無効を切り替えるスイッチがある場合は有効にしてください。

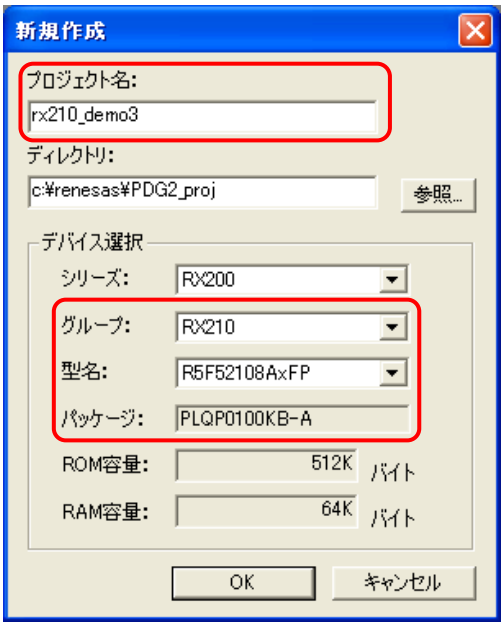
PDG

(1) PPeripheral Driver Generator ロジェクトの作成

プロジェクト名に“rx210_demo3”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1 (1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。

シリーズ ：RX200
グループ ：RX210
型名 ：R5F52108AxFP



(2) クロックの設定 **PDG**

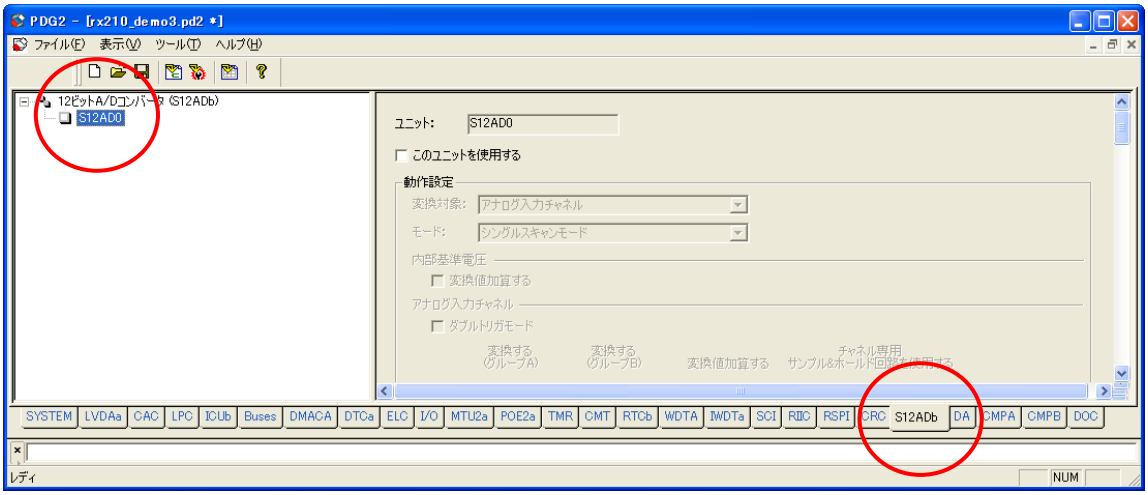
1. プロジェクトを作成するとクロック設定ウィンドウが開きます。設定画面上の や などのアイコンについては、「4.1 (2)初期状態」を参照してください。
2. クロックの設定については、「4.1 (3)クロックの設定」を参照してください。

(3) エンディアンの設定 **PDG**

エンディアンの設定については、「3.3 エンディアンの設定」を参照してください。

(4) A/D 変換器の設定-1 **PDG**

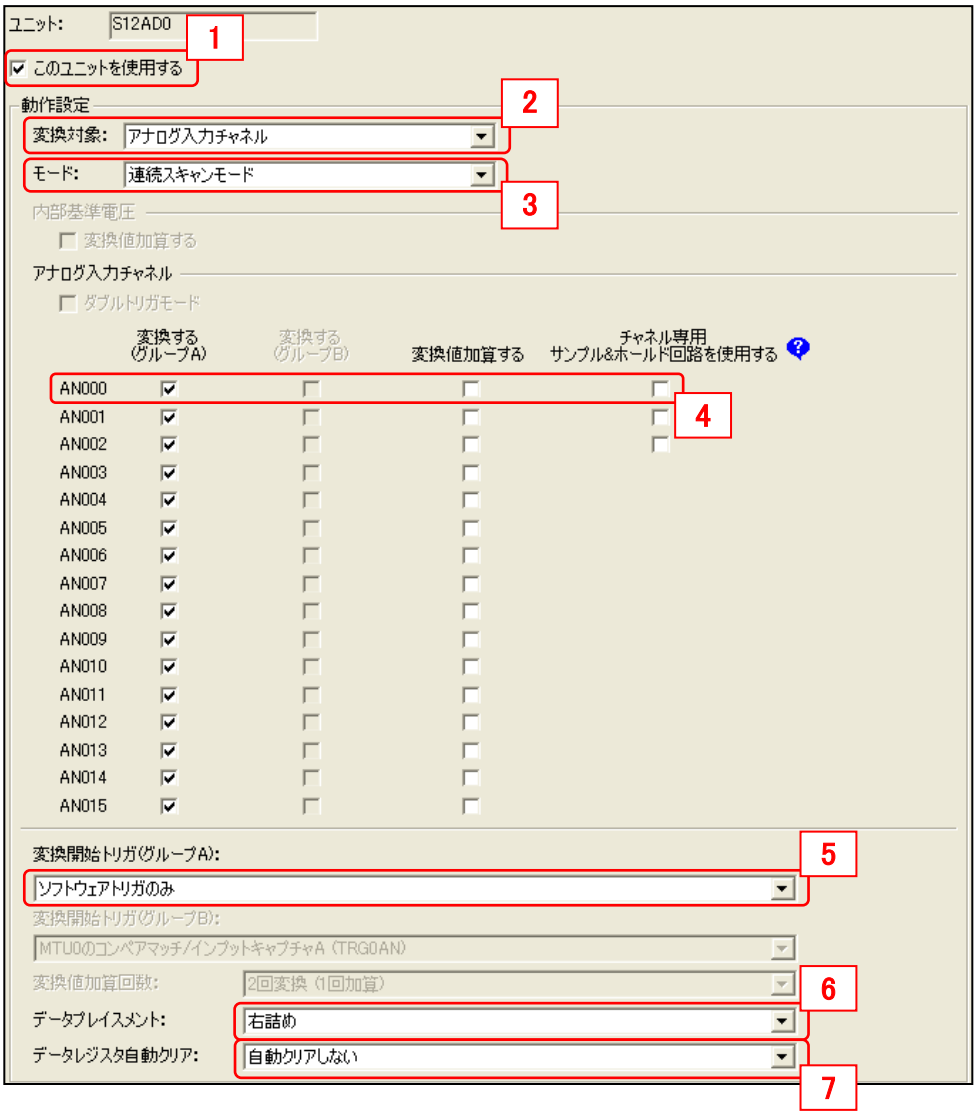
S12ADb タブを選択し、ツリー表示上で S12AD0 を選択してください。



(5) A/D 変換器の設定-2 PDG

S12AD0 を以下の通り設定してください。

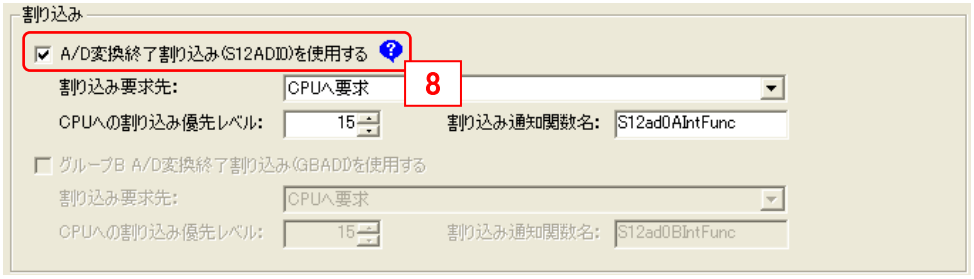
1. [このユニットを使用する]をチェック
2. 変換対象 ： [アナログ入力チャネル]
3. モード ： [連続スキャンモード]
4. AN000 ： [変換する(グループA)]をチェック
5. 変換開始トリガ(グループA) ： [ソフトウェアトリガのみ]
6. データプレイスメント ： 右詰め
7. データレジスタ自動クリア ： 自動クリアしない



(6) A/D 変換器の設定-3 PDG

S12AD0 を以下の通り設定してください。

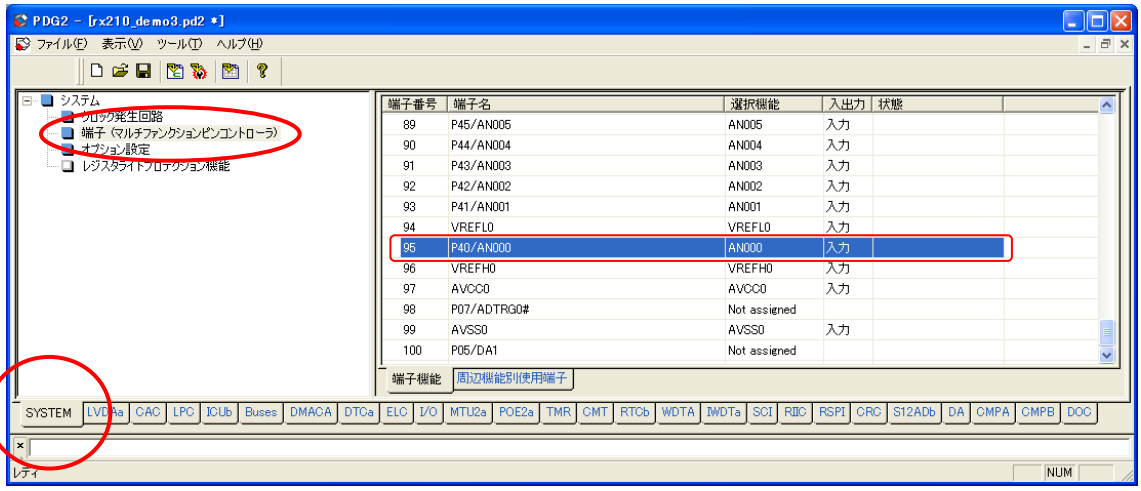
8. [A/D変換終了割り込み(S12ADI0)を使用する]をチェック



(7) 端子使用状況の確認 PDG

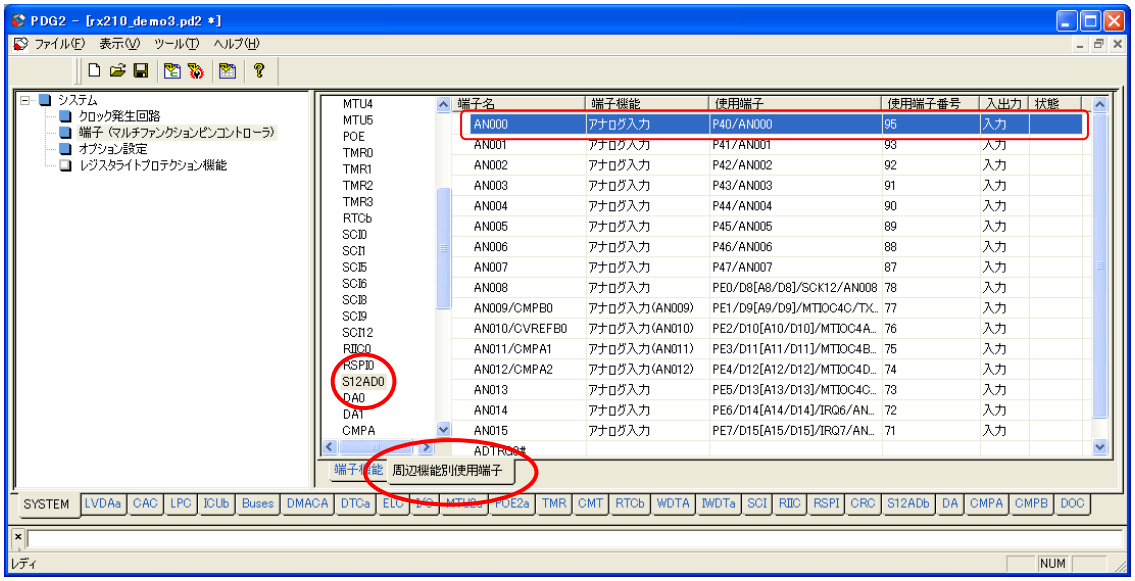
・端子機能ウィンドウで端子の使用状況を確認することができます。

1. S12ADを設定後、[SYSTEM]タブを選択し、
   ツリー表示上で[端子(マルチファンクションピンコントローラ)]を選択してください。
2. [端子機能]ウィンドウ上で 95 ピンが AN000 として使用されていることを確認してください。



・周辺機能ごとの端子の使用状況は周辺機能別使用端子ウィンドウで確認することができます。

[周辺機能別使用端子]タブをクリックし、周辺機能の一覧からS12AD0を選択してAN000端子の使用状況を確認してください。



(8) ソースファイルの生成 **PDG**

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1 (9)ソースファイルの生成」を参照してください。

(9) High-performance Embedded Workshop プロジェクトの準備 **HEW**

High-performance Embedded Workshop を起動し、RX210 用のワークスペースを作成してください。作成方法については「4.1 (10) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1 (11) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(11) プログラムの作成 **HEW**

High-performance Embedded Workshop 上で main 関数の部分を変更し、以下のプログラムを作成してください。

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx210_demo3.h"

void main(void)
{
    //存在しないポートの設定
//  R_PG_IO_PORT_SetPortNotAvailable();
    //クロックの設定(発振安定時間ウェイト)
    R_PG_Clock_WaitSet(0.01);
    //A/D変換器の設定
    R_PG_ADC_12_Set_S12AD0();
    //A/D変換器の開始(ソフトウェアトリガ)
    R_PG_ADC_12_StartConversionSW_S12AD0();
    while(1);
}

//変換結果格納先変数
uint16_t result;
//A/D変換終了割り込み通知関数
void S12ad0AIntFunc(void)
{
    //A/D変換結果の取得
    R_PG_ADC_12_GetResult_S12AD0(&result);
}
```

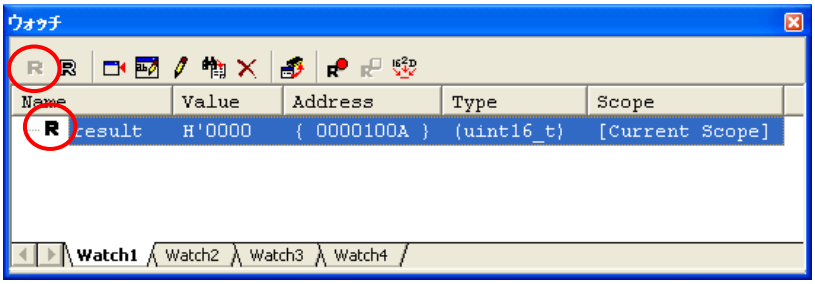
(12) エミュレータの接続、プログラムのビルド、ダウンロード **HEW**

作成したプログラムをビルドし、ダウンロードしてください。

エミュレータの接続、プログラムのビルド方法については「4.1 (13) エミュレータの接続、プログラムのビルド、実行」 を参照してください。

(13) A/D 変換結果格納変数のウォッチウィンドウ登録 **HEW**

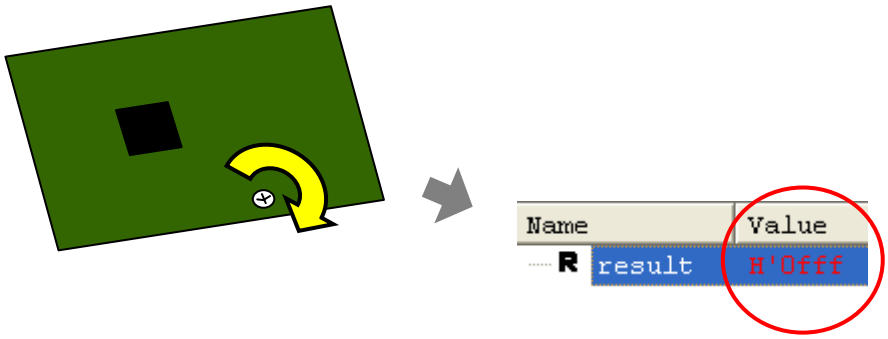
High-performance Embedded Workshop のウォッチウィンドウを開き、変数 ″result″ を登録してください。″result″をリアルタイム更新に設定すると、実行中に値の変化を確認することができます。



(14) プログラムの実行と A/D 変換結果の確認 **HEW**

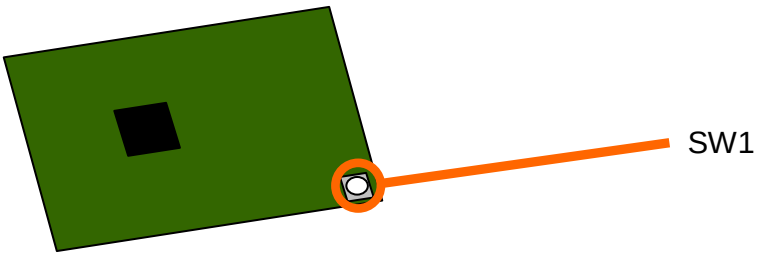
プログラムを実行し、実行中ポテンショメータを回してアナログ入力電圧を変動させてください。

ウォッチウィンドウ上の ″result″ の値が変化します。



4.4 ICUbによるDTCa転送のトリガ

RX210 RSK ボードではスイッチ 1 (SW1) が IRQ1 外部割込み入力端子に接続されています。
このチュートリアルでは IRQ1 をトリガとした DTCa 転送を行います。



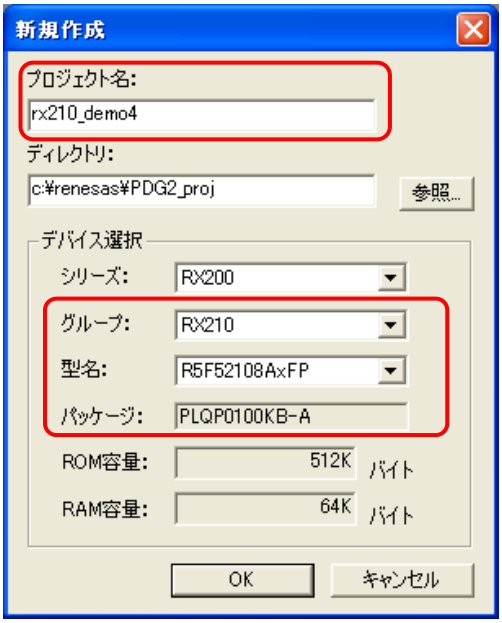
使用する RSK ボード上に IRQ1 の有効/無効を切り替えるスイッチがある場合は有効にしてください。

(1) Peripheral Driver Generator プロジェクトの作成 PDG

プロジェクト名に“rx210_demo4”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1 (1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。

シリーズ ：RX200
グループ ：RX210
型名 ：R5F52108AxFP



(2) クロックの設定 PDG

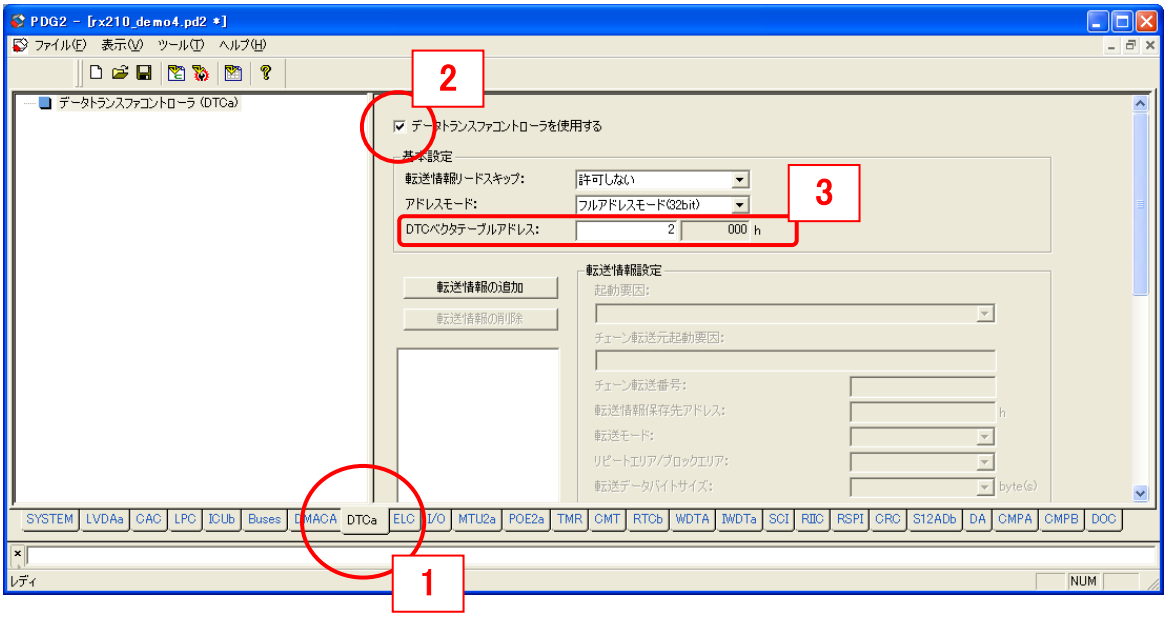
1. プロジェクトを作成するとクロック設定ウィンドウが開きます。設定画面上の や などのアイコンについては、「4.1 (2)初期状態」を参照してください。
2. クロックの設定については、「4.1 (3)クロックの設定」を参照してください。

(3) エンディアンの設定 PDG

エンディアンの設定については、「3.3 エンディアンの設定」を参照してください。

(4) DTCa の設定-1 PDG

1. DTCa タブを選択し、DTCa の設定ウィンドウを開いてください。
2. [データトランスファコントローラを使用する]をチェックしてください。
3. DTC ベクタテーブルアドレスは 2000h に配置します。2 と入力してください。

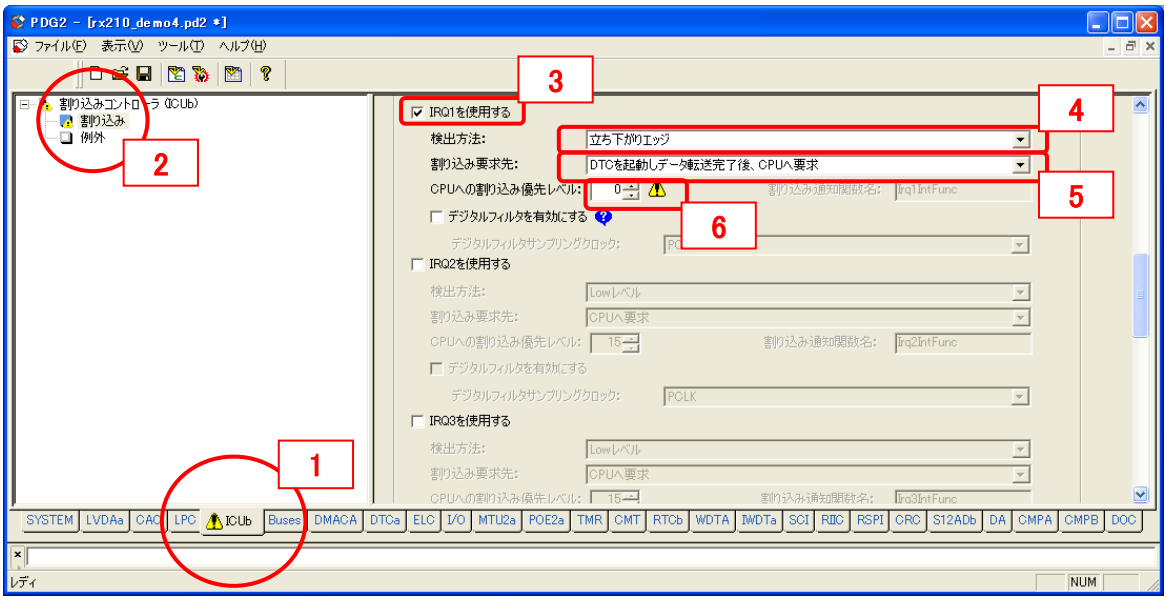


(5) DTCa の設定-2 **PDG**

1. [転送情報の追加]ボタンをクリックすると、転送情報が追加されます。
2. 起動要因に[IRQ1 (外部端子割り込み)]を指定してください。
3. 転送情報保存先アドレスに 2400 を指定してください。
4. 転送モードに[ノーマル転送モード]を指定してください。
5. 転送データバイトサイズに 1 を指定してください。
6. 転送回数に 10 を指定してください。
7. 転送元アドレスに 2410 を指定してください。
8. 転送元アドレス更新モードに[インクリメント]を指定してください。
9. 転送先アドレスに 2420 を指定してください。
10. 転送先アドレス更新モードに[インクリメント]を指定してください。

(6) ICUb の設定 PDG

1. ICUb タブを選択してください。
2. ツリー表示上で[割り込み]を選択してください。
3. [IRQ1 を使用する]をチェックしてください。
4. 検出方法に[立ち下がりエッジ]を指定してください。
5. 割り込み要求先に[DTC を起動しデータ転送完了後、CPU へ要求]を指定してください。
6. IRQ の CPU 割り込みは使用しません。割り込み優先レベルに 0 を指定してください。



(7) ソースファイルの生成 PDG

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1 (9)ソースファイルの生成」を参照してください。

(8) High-performance Embedded Workshop プロジェクトの準備 HEW

High-performance Embedded Workshop を起動し、RX210 用のワークスペースを作成してください。作成方法については「4.1 (10) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

(9) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1 (11) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(10) プログラムの作成 **HEW**

High-performance Embedded Workshop 上で main 関数の部分を変更し、以下のプログラムを作成してください。

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx210_demo4.h"

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00002000
uint32_t dtc_vector_table [256];

//DTC転送情報保存先(IRQ1)
#pragma address dtc_transfer_data_IRQ1 = 0x00002400
uint32_t dtc_transfer_data_IRQ1 [4];

//転送元
#pragma address dtc_src_data = 0x00002410
uint8_t dtc_src_data [10] = "ABCDEFGHIJ";

//転送先
#pragma address dtc_dest_data = 0x00002420
uint8_t dtc_dest_data [10];

void main(void)
{
    //転送先初期化
    int i;
    for(i=0; i<10; i++ ){
        dtc_dest_data[i] = 0;
    }

    //存在しないポートの設定
//  R_PG_IO_PORT_SetPortNotAvailable();

    R_PG_Clock_WaitSet(0.01); //クロックの設定(発振安定時間ウェイト)

    //DTCの設定(ベクタテーブルアドレスなど)
    R_PG_DTC_Set();

    //DTCの設定(IRQ1をトリガとする転送の設定)
    R_PG_DTC_Set_IRQ1();

    R_PG_ExtInterrupt_Set_IRQ1();  //IRQ1の設定

    R_PG_DTC_Activate();  //DTC転送開始
    while(1);
}
```

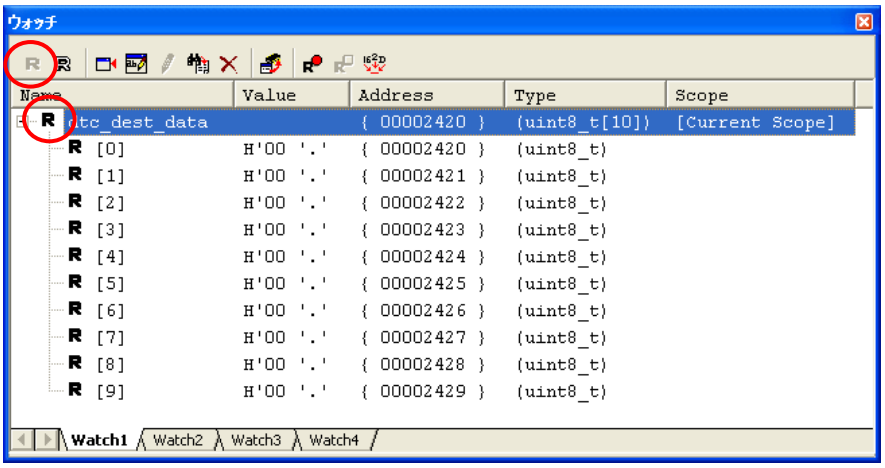
(11) エミュレータの接続、プログラムのビルド、ダウンロード **HEW**

作成したプログラムをビルドし、ダウンロードしてください。

エミュレータの接続、プログラムのビルド方法については「4.1 (13) エミュレータの接続、プログラムのビルド、実行」 を参照してください。

(12) 転送先変数のウォッチウィンドウ登録 **HEW**

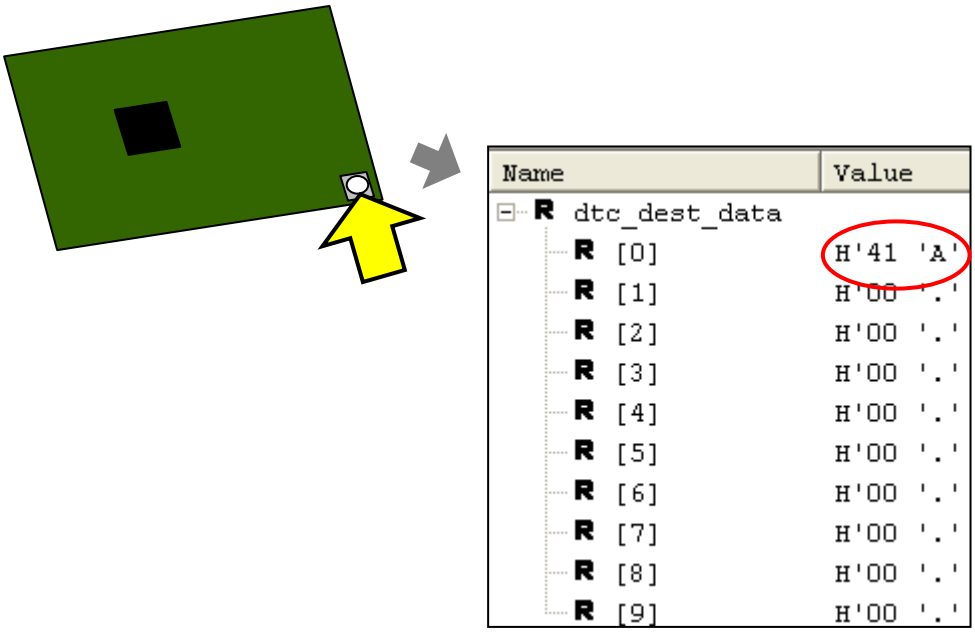
High-performance Embedded Workshop のウォッチウィンドウを開き、転送先変数 "dtc_dest_data" を登録してください。"dtc_dest_data"を展開しリアルタイム更新に設定すると、実行中に値の変化を確認することができます。



(13) プログラムの実行と転送結果の確認 **HEW**

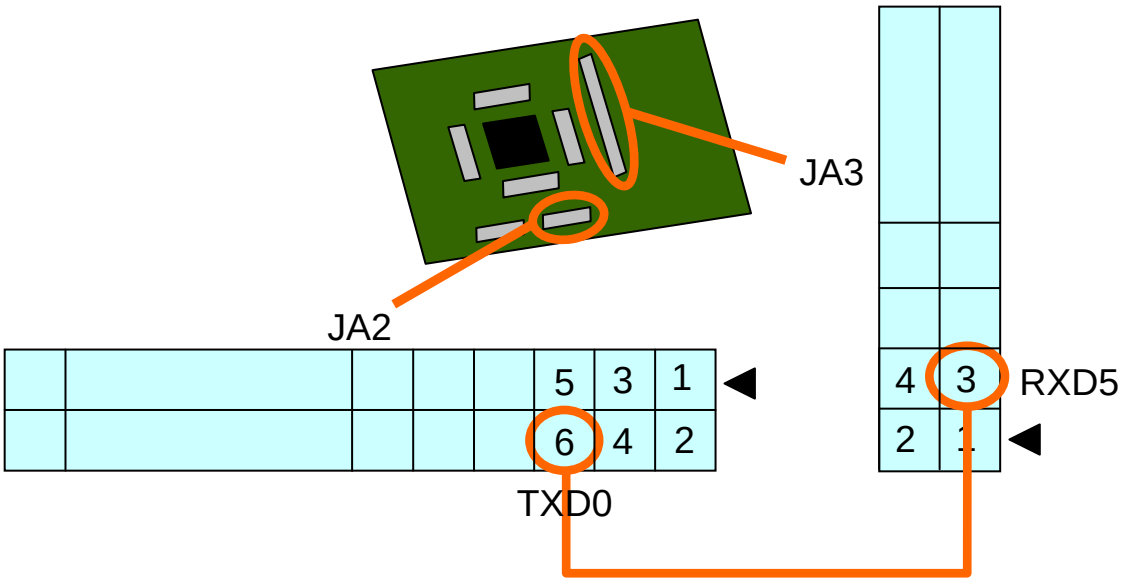
プログラムを実行し、実行中 SW1 を押して IRQ1 割り込みを発生させてください。

ボタンを押すたびにデータが転送されます。



4.5 SCIc チャネル 0 とチャネル 5 で調歩同期通信

このチュートリアルでは、シリアルのチャネル 0 からチャネル 5 に調歩同期モードでデータを送信します。RSK ボード上でチャネル 0 の送信端子(TXD0)とチャネル 5 の受信端子(RXD5)を図の様に接続してください。TXD0 は RSK ボードの JA2/No.6、RXD5 は JA3/No.3 です。



使用する RSK ボード上に TXD0、RXD5 の有効/無効を切り替えるスイッチがある場合は有効にしてください。

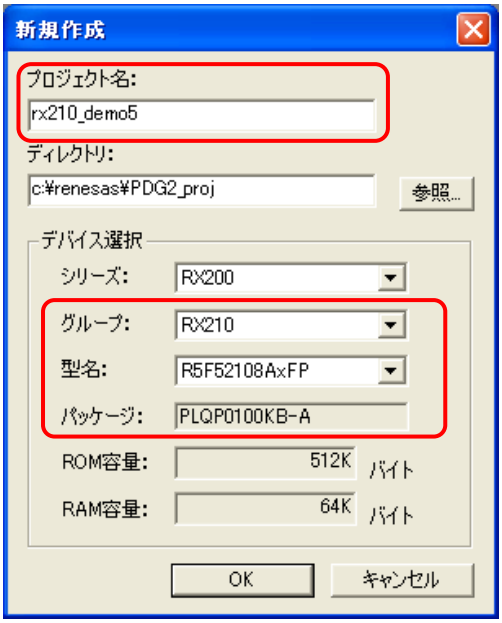
(1) Peripheral Driver Generator プロジェクトの作成

PDG

プロジェクト名に“rx210_demo5”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1 (1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。

シリーズ ：RX200
グループ ：RX210
型名 ：R5F52108AxFP



(2) クロックの設定 **PDG**

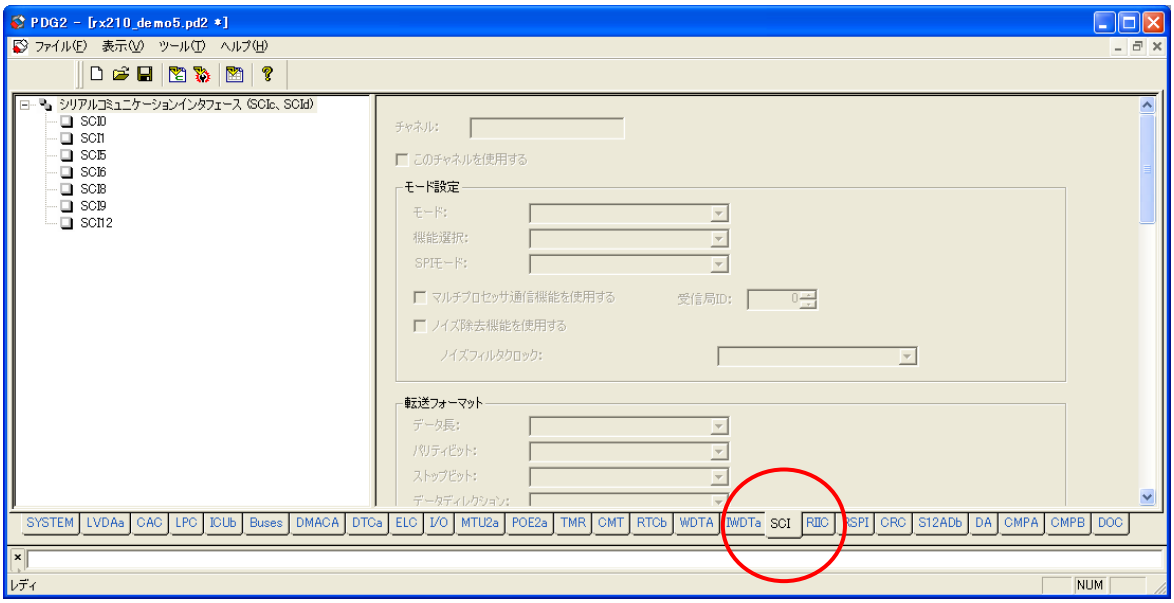
1. プロジェクトを作成するとクロック設定ウィンドウが開きます。設定画面上の や などのアイコンについては、「4.1 (2)初期状態」を参照してください。
2. クロックの設定については、「4.1 (3)クロックの設定」を参照してください。

(3) エンディアンの設定 **PDG**

エンディアンの設定については、「3.3 エンディアンの設定」を参照してください。

(4) SCIc の設定 **PDG**

SCI タブを選択し、SCIc の設定ウィンドウを開いてください。



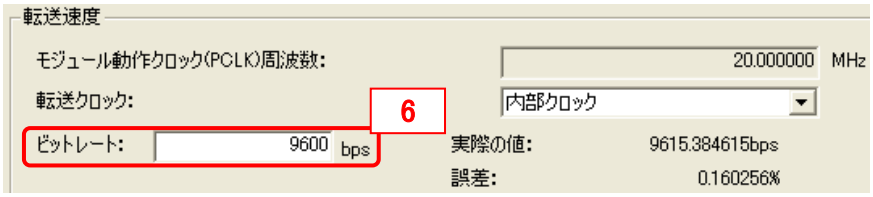
(5) SCI0(送信側)の設定 PDG

SCI0 を以下の通り設定してください。

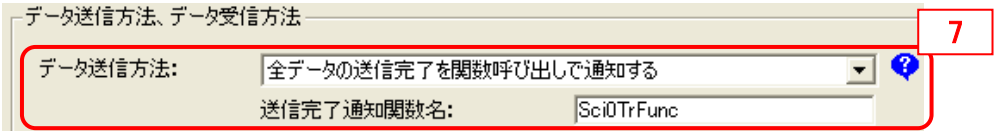
1. ツリー表示上で SCI0 を選択してください。

2. [このチャネルを使用する]をチェックしてください。
3. モードに[調歩同期式モード]を選択してください。
4. 機能選択に[送信]を指定してください。
5. 転送フォーマットは初期設定のままとしてください。

6. 転送速度設定のビットレートに 9600bps を設定してください。



7. データ送信方法に[全データの送信完了を関数呼び出しで通知する]を指定し、送信完了通知関数名を初期設定の”Sci0TrFunc”としてください。

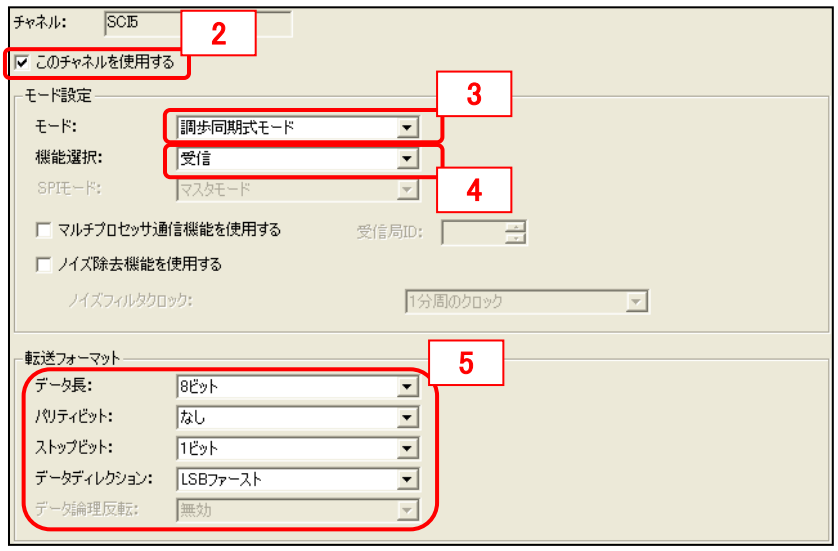


(6) SCI5(受信側)の設定 PDG

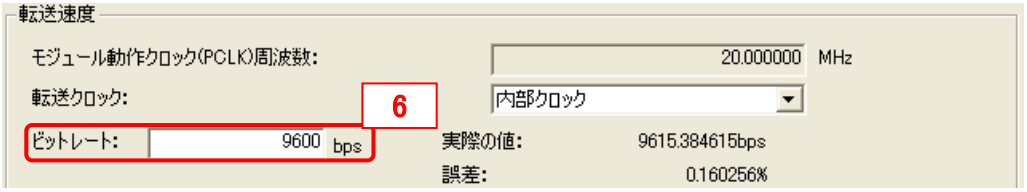
SCI5 を以下の通り設定してください。

1. ツリー表示上で SCI5 を選択してください。

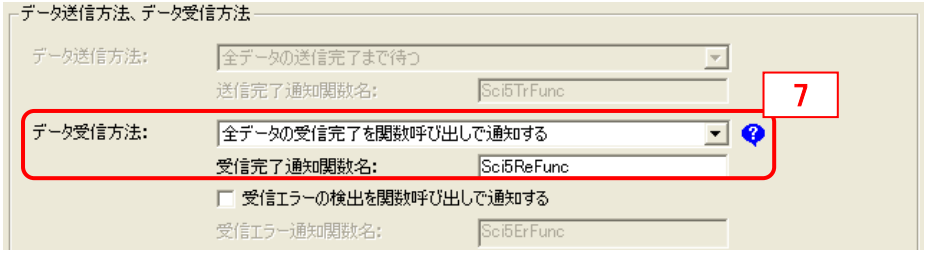
2. [このチャネルを使用する]をチェックしてください。
3. モードに[調歩同期式モード]を選択してください。
4. 機能選択に[受信]を指定してください。
5. 転送フォーマットは初期設定のままとしてください。



6. 転送速度設定のビットレートに 9600bps を設定してください。



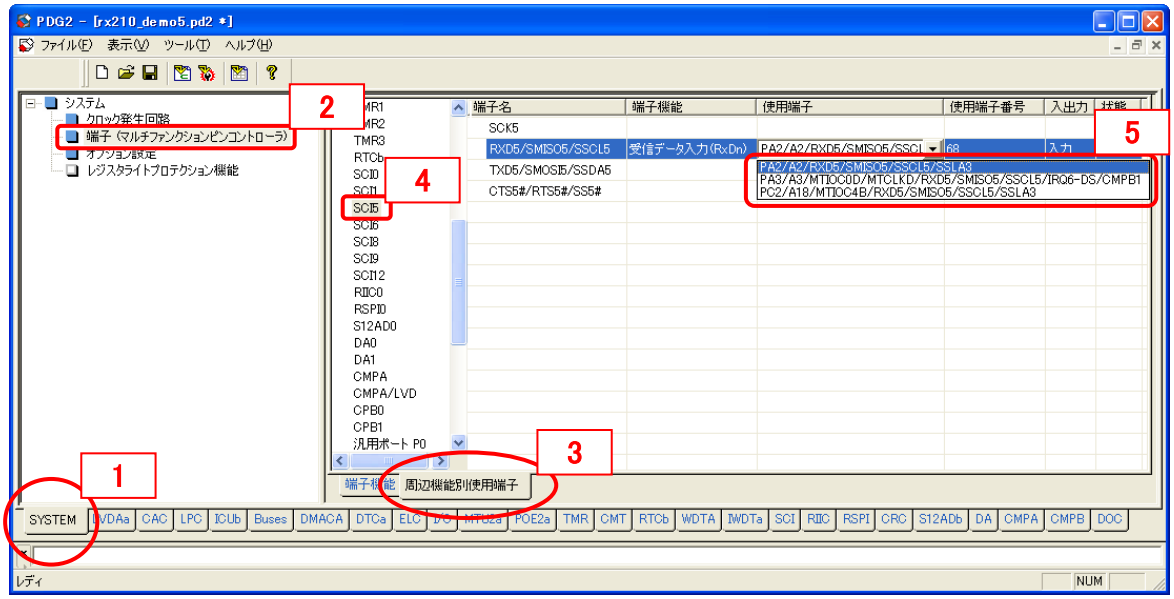
7. データ受信方法に[全データの受信完了を関数呼び出しで通知する]を指定し、受信完了通知関数名を初期設定の”Sci5ReFunc”としてください。



(7) 使用する端子の設定 **PDG**

RXD5 は RXD5 (PA2) と RXD5 (PA3) と RXD5 (PC2) から選択することができます。以下の方法で使用する端子を選択してください。

1. SYSTEM タブを選択してください。
2. ツリー表示上で[端子]を選択してください。
3. [周辺機能別使用端子]タブを選択してください。
4. 周辺機能の一覧から SCI5 を選択してください。
5. RXD5 の行で[使用端子カラム]カラムにマウスポインタを置くと、端子選択のドロップダウンボタンが表示されます。ドロップダウンリストから[PA2/A2/RXD5/SMISO5/SSCL5/SSLA3]を選択してください。



(8) ソースファイルの生成 **PDG**

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1 (9)ソースファイルの生成」を参照してください。

(9) High-performance Embedded Workshop プロジェクトの準備 **HEW**

High-performance Embedded Workshop を起動し、RX210 用のワークスペースを作成してください。作成方法については「4.1 (10) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1 (11) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(11) プログラムの作成 HEW

High-performance Embedded Workshop 上で main 関数の部分を変更し、以下のプログラムを作成してください。

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx210_demo5.h"
//SCI0送信データ
uint8_t tr_data[10] = "ABCDEFGHIJ";
//SCI5受信データ
uint8_t re_data[10] = "----------";
void main(void)
{
    //存在しないポートの設定
//  R_PG_IO_PORT_SetPortNotAvailable();
    //クロックの設定(発振安定時間ウェイト)
    R_PG_Clock_WaitSet(0.01);
    //SCI0の設定
    R_PG_SCI_Set_C0();
    //SCI5の設定
    R_PG_SCI_Set_C5();
    //SCI5受信開始 (受信データ数:10)
    R_PG_SCI_StartReceiving_C5( re_data, 10 );
    //SCI0送信開始 (送信データ数:10)
    R_PG_SCI_StartSending_C0( tr_data, 10 );
    while(1);
}
//SCI0送信完了通知関数
void Sci0TrFunc(void)
{
    //SCI0通信終了
    R_PG_SCI_StopCommunication_C0();
}
//SCI5受信完了通知関数
void Sci5ReFunc(void)
{
    //SCI5通信終了
    R_PG_SCI_StopCommunication_C5();
}
```

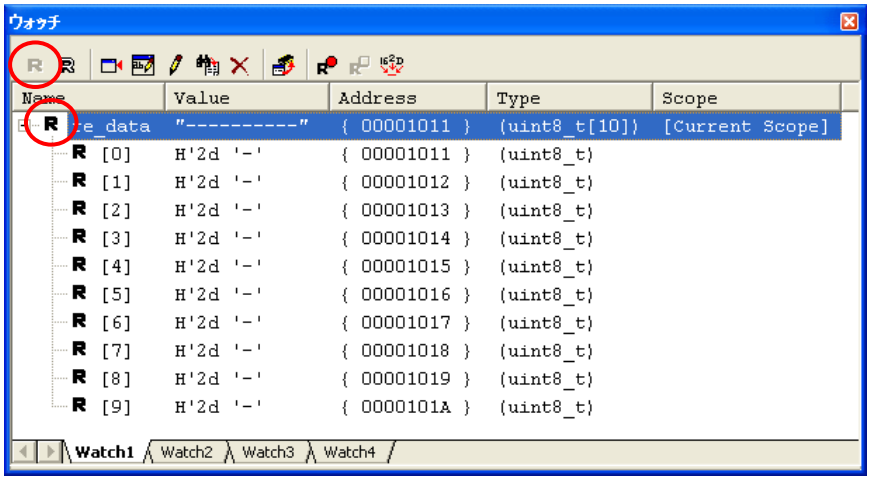
(12) エミュレータの接続、プログラムのビルド、ダウンロード **HEW**

作成したプログラムをビルドし、ダウンロードしてください。

エミュレータの接続、プログラムのビルド方法については「4.1 (13) エミュレータの接続、プログラムのビルド、実行」 を参照してください。

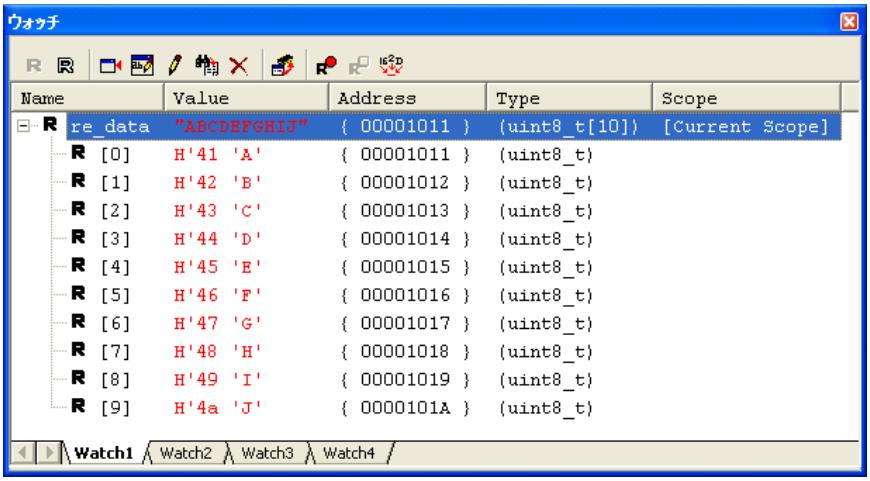
(13) 受信データ格納変数のウォッチウィンドウ登録 **HEW**

High-performance Embedded Workshop のウォッチウィンドウを開き、転送先変数 ″re_data″ を登録してください。″re_data″を展開しリアルタイム更新に設定すると、実行中に値の変化を確認することができます。



(14) プログラムの実行と転送結果の確認 **HEW**

プログラムを実行し、変数の値を確認してください。



5. 生成関数仕様

RX210 の生成関数を表 5.1 に示します。

表 5.1 RX210 の生成関数

クロック発生回路

生成関数	機能
R_PG_Clock_Set	クロックの設定
R_PG_Clock_WaitSet	クロックの設定(発振安定時間ウェイト)
R_PG_Clock_Start_MAIN	メインクロックの発振開始
R_PG_Clock_Stop_MAIN	メインクロックの発振停止
R_PG_Clock_Start_SUB	サブクロックの発振開始
R_PG_Clock_Stop_SUB	サブクロックの発振停止
R_PG_Clock_Start_LOCO	低速オンチップオシレータ (LOCO)の発振開始
R_PG_Clock_Stop_LOCO	低速オンチップオシレータ (LOCO)の発振停止
R_PG_Clock_Start_HOCO	高速オンチップオシレータ (HOCO)の発振開始
R_PG_Clock_Stop_HOCO	高速オンチップオシレータ (HOCO)の発振停止
R_PG_Clock_PowerON_HOCO	高速オンチップオシレータ (HOCO)の電源をONにする
R_PG_Clock_PowerON_HOCO	高速オンチップオシレータ (HOCO)の電源をOFFにする
R_PG_Clock_Start_PLL	PLL回路の動作開始
R_PG_Clock_Stop_PLL	PLL回路の動作停止
R_PG_Clock_Enable_BCLK_PinOutput	BCLK端子出力の有効化
R_PG_Clock_Disable_BCLK_PinOutput	BCLK端子出力の無効化
R_PG_Clock_Enable_MAIN_StopDetection	メインクロック発振停止検出機能の有効化
R_PG_Clock_Disable_MAIN_StopDetection	メインクロック発振停止検出機能の無効化
R_PG_Clock_GetFlag_MAIN_StopDetection	メインクロック発振停止検出フラグの取得
R_PG_Clock_ClearFlag_MAIN_StopDetection	メインクロック発振停止検出フラグのクリア
R_PG_Clock_GetSelectedClockSource	現在の内部クロックソースの取得
R_PG_Clock_GetClocksStatus	クロック発振状態の取得
R_PG_Clock_GetHOCOPowerStatus	高速オンチップオシレータ(HOCO)の電源状態取得

電圧検出回路(LVDAa)

生成関数	機能
R_PG_LVD_Set	電圧検出回路の設定(電圧監視1, 2一括設定)
R_PG_LVD_GetStatus	電圧検出回路のステータスフラグを取得
R_PG_LVD_ClearDetectionFlag_LVD	電圧監視n電圧変化検出フラグのクリア(n : 1, 2)
R_PG_LVD_Disable_LVD*<電圧検出回路番号>*	電圧監視nの無効化(n : 1, 2)

クロック周波数精度測定回路 (CAC)

生成関数	機能
R_PG_CAC_Set	クロック周波数精度測定回路の設定と測定の開始
R_PG_CAC_ClearFlag_FrequencyError	周波数エラーフラグのクリア
R_PG_CAC_ClearFlag_MeasurementEnd	測定終了フラグのクリア
R_PG_CAC_ClearFlag_Overflow	オーバフローフラグのクリア

R_PG_CAC_StartMeasurement	クロック周波数測定の開始
R_PG_CAC_StopMeasurement	クロック周波数測定の停止
R_PG_CAC_GetStatusFlags	CACステータスフラグの取得
R_PG_CAC_GetCounterBufferRegister	カウンタバッファレジスタ(CACNTBR)値の取得
R_PG_CAC_StopModule	クロック周波数精度測定回路の停止

消費電力低減機能

生成関数	機能
R_PG_LPC_Set	消費電力低減機能の設定
R_PG_LPC_Sleep	スリープモードへの移行
R_PG_LPC_AllModuleClockStop	全モジュールクロックスタンバイモードへの移行
R_PG_LPC_SoftwareStandby	ソフトウェアスタンバイモードへの移行
R_PG_LPC_DeepSoftwareStandby	ディープソフトウェアスタンバイモードへの移行
R_PG_LPC_IOPortRelease	I/Oポート出力保持を解除
R_PG_LPC_ChangeOperatingPowerControl	動作電力制御モードを変更
R_PG_LPC_ChangeSleepModeReturnClock	スリープモード復帰クロックソースを変更
R_PG_LPC_GetPowerOnResetFlag	パワーオンリセットフラグの取得
R_PG_LPC_GetLVDDetectionFlag	LVD検知フラグの取得
R_PG_LPC_GetDeepSoftwareStandbyResetFlag	ディープソフトウェアスタンバイリセットフラグの取得
R_PD_LPC_GetDeepSoftwareStandbyCancelFlag	ディープソフトウェアスタンバイ解除要求フラグの取得
R_PG_LPC_GetOperatingPowerControlFlag	動作電力制御モード遷移状態フラグの取得
R_PG_LPC_GetStatus	消費電力低減機能の状態を取得
R_PG_LPC_WriteBackup	ディープスタンバイバックアップレジスタへの書き込み
R_PG_LPC_ReadBackup	ディープスタンバイバックアップレジスタからの読み出し

レジスタライトプロテクション機能

生成関数	機能
R_PG_RWP_RegisterWriteCgc	クロック発生回路関連レジスタへの書き込みを許可/禁止
R_PG_RWP_RegisterWriteModeLpcReset	動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込みを許可/禁止
R_PG_RWP_RegisterWriteVrcr	VRCRへの書き込みを許可/禁止
R_PG_RWP_RegisterWriteLvd	LVD関連レジスタへの書き込みを許可/禁止
R_PG_RWP_RegisterWriteMpc	端子機能選択レジスタへの書き込みを許可/禁止
R_PG_RWP_GetStatusCgc	クロック発生回路関連レジスタへの書き込み状態の取得
R_PG_RWP_GetStatusModeLpcReset	動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込み状態の取得
R_PG_RWP_GetStatusVrcr	VRCRへの書き込み状態の取得
R_PG_RWP_GetStatusLvd	LVD関連レジスタへの書き込み状態の取得
R_PG_RWP_GetStatusMpc	端子機能選択レジスタへの書き込み状態の取得

割り込みコントローラ (ICUb)

生成関数	機能
R_PG_ExtInterrupt_Set_*<割り込み種別>*	外部割り込みの設定
R_PG_ExtInterrupt_Disable_*<割り込み種別>*	外部割り込みの設定解除
R_PG_ExtInterrupt_GetRequestFlag_*<割り込み種別>*	外部割り込み要求フラグの取得

R_PG_ExtInterrupt_ClearRequestFlag_*〈割り込み種別〉*	外部割り込み要求フラグのクリア
R_PG_ExtInterrupt_EnableFilter_*〈割り込み種別〉*	デジタルフィルタの再有効化
R_PG_ExtInterrupt_DisableFilter_*〈割り込み種別〉*	デジタルフィルタの無効化
R_PG_SoftwareInterrupt_Set	ソフトウェア割り込みの設定
R_PG_SoftwareInterrupt_Generate	ソフトウェア割り込みの生成
R_PG_FastInterrupt_Set	高速割り込みの設定
R_PG_Exception_Set	例外ハンドラの設定

バス

生成関数	機能
R_PG_ExtBus_PresetBus	バスプライオリティの設定
R_PG_ExtBus_SetBus	バス端子とバスエラー監視の設定
R_PG_ExtBus_SetArea_CS*〈CS領域の番号〉*	CS領域の設定
R_PG_ExtBus_SetEnable	外部バスの有効化
R_PG_ExtBus_GetErrorStatus	バスエラー検出状態の取得
R_PG_ExtBus_ClearErrorFlags	バスエラーステータスレジスタのクリア
R_PG_ExtBus_DisableArea_CS*〈CS領域の番号〉*	CS領域の設定解除
R_PG_ExtBus_SetDisable	外部バスの無効化

DMAコントローラ（DMACA）

生成関数	機能
R_PG_DMAC_Set_C*〈チャネル番号〉*	DMACの設定
R_PG_DMAC_Activate_C*〈チャネル番号〉*	DMACを転送開始トリガの入力待ち状態に設定
R_PG_DMAC_StartTransfer_C*〈チャネル番号〉*	DMA一転送の開始(ソフトウェアトリガ)
R_PG_DMAC_StartContinuousTransfer_C*〈チャネル番号〉*	DMA連続転送の開始(ソフトウェアトリガ)
R_PG_DMAC_StopContinuousTransfer_C*〈チャネル番号〉*	ソフトウェアトリガにより開始したDMA連続転送の停止
R_PG_DMAC_Suspend_C*〈チャネル番号〉*	データ転送の中断
R_PG_DMAC_GetTransferCount_C*〈チャネル番号〉*	転送カウンタ値の取得
R_PG_DMAC_SetTransferCount_C*〈チャネル番号〉*	転送カウンタ値の設定
R_PG_DMAC_GetRepeatBlockSizeCount_C*〈チャネル番号〉*	リピート/ブロックサイズカウンタ値の取得
R_PG_DMAC_SetRepeatBlockSizeCount_C*〈チャネル番号〉*	リピート/ブロックサイズカウンタ値の設定
R_PG_DMAC_ClearInterruptFlag_C*〈チャネル番号〉*	割り込み要求フラグの取得とクリア
R_PG_DMAC_GetTransferEndFlag_C*〈チャネル番号〉*	転送終了フラグの取得
R_PG_DMAC_ClearTransferEndFlag_C*〈チャネル番号〉*	転送終了フラグのクリア
R_PG_DMAC_GetTransferEscapeEndFlag_C*〈チャネル番号〉*	転送エスケープ終了フラグの取得
R_PG_DMAC_ClearTransferEscapeEndFlag_C*〈チャネル番号〉*	転送エスケープ終了フラグのクリア
R_PG_DMAC_SetSrcAddress_C*〈チャネル番号〉*	転送元アドレスの設定
R_PG_DMAC_SetDestAddress_C*〈チャネル番号〉*	転送先アドレスの設定
R_PG_DMAC_SetAddressOffset_C*〈チャネル番号〉*	アドレスオフセット値の設定
R_PG_DMAC_SetExtendedRepeatSrc_C*〈チャネル番号〉*	転送元拡張リピートエリアの設定
R_PG_DMAC_SetExtendedRepeatDest_C*〈チャネル番号〉*	転送先拡張リピートエリアの設定
R_PG_DMAC_StopModule_C*〈チャネル番号〉*	DMACチャネルの停止

データトランスファコントローラ（DTCa）

生成関数	機能
R_PG_DTC_Set	DTCの設定
R_PG_DTC_Set_*〈転送開始要因〉*	DTC転送情報の設定

R_PG_DTC_Activate	DTCを転送開始トリガの入力待ち状態に設定
R_PG_DTC_SuspendTransfer	DTC転送の停止
R_PG_DTC_GetTransmitStatus	DTC転送状態の取得
R_PG_DTC_StopModule	DTCの停止

イベントリンクコントローラ (ELC)

生成関数	機能
R_PG_ELC_Set	ELCの設定
R_PG_ELC_SetLink_*<周辺機能>*	イベントリンクの設定
R_PG_ELC_DisableLink_*<周辺機能>*	イベントリンク動作の停止
R_PG_ELC_Set_PortGroup*<ポートグループ番号>*	ポートグループの設定
R_PG_ELC_Set_SinglePort*<シングルポート番号>*	シングルポートの設定
R_PG_ELC_AllEventLinkEnable	全イベントリンクの有効化
R_PG_ELC_AllEventLinkDisable	全イベントリンクの無効化
R_PG_ELC_Generate_SoftwareEvent	ソフトウェアイベントの生成
R_PG_ELC_GetPortBufferValue_Group*<ポートグループ番号>*	ポートバッファレジスタ値の取得
R_PG_ELC_SetPortBufferValue_Group*<ポートグループ番号>*	ポートバッファレジスタ値の設定
R_PG_ELC_StopModule	ELCの停止

I/Oポート

生成関数	機能
R_PG_IO_PORT_Set_P*<ポート番号>*	I/Oポートの設定
R_PG_IO_PORT_Set_P*<ポート番号><端子番号>*	I/Oポート(1端子)の設定
R_PG_IO_PORT_Read_P*<ポート番号>*	ポート入力レジスタの読み出し
R_PG_IO_PORT_Read_P*<ポート番号><端子番号>*	ポート入力レジスタからのビット読み出し
R_PG_IO_PORT_Write_P*<ポート番号>*	ポート出力データレジスタへの書き込み
R_PG_IO_PORT_Write_P*<ポート番号><端子番号>*	ポート出力データレジスタへのビット書き込み
R_PG_IO_PORT_SetPortNotAvailable	存在しない端子の処理

マルチファンクションタイマパルスユニット2 (MTU2a)

生成関数	機能
R_PG_Timer_Set_MTU_U*<ユニット番号>*_*<チャネル>*	MTUの設定
R_PG_Timer_StartCount_MTU_U*<ユニット番号>*_C*<チャネル番号>*	MTUのカウント動作開始
R_PG_Timer_SynchronouslyStartCount_MTU_U*<ユニット番号>*	MTUの複数チャネルのカウント動作を同時に開始
R_PG_Timer_HaltCount_MTU_U*<ユニット番号>*_C*<チャネル番号>*	MTUのカウント動作を一時停止
R_PG_Timer_GetCounterValue_MTU_U*<ユニット番号>*_C*<チャネル番号>*	MTUのカウンタ値を取得
R_PG_Timer_SetCounterValue_MTU_U*<ユニット番号>*_C*<チャネル番号>*	MTUのカウンタ値を設定
R_PG_Timer_GetRequestFlag_MTU_U*<ユニット番号>*_C*<チャネル番号>*	MTUの割り込み要求フラグの取得とクリア
R_PG_Timer_StopModule_MTU_U*<ユニット番号>*	MTUのユニットを停止
R_PG_Timer_GetTGR_MTU_U*<ユニット番号>*_C*<チャネル番号>*	ジェネラルレジスタの値の取得
R_PG_Timer_SetTGR_*<ジェネラルレジスタ>*_MTU_U*<ユニット番号>*_C*<チャネル番号>*	ジェネラルレジスタの値の設定
R_PG_Timer_SetBuffer_AD_MTU_U*<ユニット番号>*_C*<チャネル番号>*	A/D変換要求周期設定バッファレジスタの設定
R_PG_Timer_SetBuffer_CycleData_MTU_U*<ユニット番号>*_*<チャネル>*	周期バッファレジスタ値の設定

R_PG_Timer_SetOutputPhaseSwitch_MTU_U〈*ユニット番号*〉_〈*チャネル*〉	PWM出力レベルの切り替え
R_PG_Timer_ControlOutputPin_MTU_U〈*ユニット番号*〉_〈*チャネル*〉	PWM出力の有効化/無効化
R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U〈*ユニット番号*〉_〈*チャネル*〉	PWM出力レベルをバッファレジスタに設定
R_PG_Timer_ControlBufferTransfer_MTU_U〈*ユニット番号*〉_〈*チャネル*〉	バッファレジスタからテンポラリレジスタへのバッファ転送の有効化、無効化

ポートアウトプットイネーブル2 (POE2a)

生成関数	機能
R_PG_POE_Set	POEの設定
R_PG_POE_SetHiZ_〈*タイマチャネル*〉	タイマ出力端子をハイインピーダンスに設定
R_PG_POE_GetRequestFlagHiZ_〈*タイマチャネル/フラグ*〉	ハイインピーダンス要求フラグの取得
R_PG_POE_GetShortFlag_〈*タイマチャネル*〉	MTU端子の出力短絡フラグの取得
R_PG_POE_ClearFlag_〈*タイマチャネル/フラグ*〉	ハイインピーダンス要求フラグと出力短絡フラグのクリア

16ビットタイマパルスユニット (TPUa)

生成関数	機能
R_PG_Timer_Set_TPU_U〈*ユニット番号*〉	TPUをユニット単位で設定
R_PG_Timer_Start_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	TPUを設定しカウントを開始
R_PG_Timer_SynchronouslyStartCount_TPU_U〈*ユニット番号*〉	TPUの複数チャネルのカウント動作を同時に開始
R_PG_Timer_HaltCount_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	TPUのカウントを一時停止
R_PG_Timer_ResumeCount_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	TPUのカウントを再開
R_PG_Timer_GetCounterValue_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	TPUのカウンタ値を取得
R_PG_Timer_SetCounterValue_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	TPUのカウンタ値を設定
R_PG_Timer_GetTGR_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	ジェネラルレジスタの値の取得
R_PG_Timer_SetTGR_〈*ジェネラルレジスタ*〉_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	ジェネラルレジスタの値の設定
R_PG_Timer_GetRequestFlag_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉	TPUの割り込み要求フラグの取得とクリア
R_PG_Timer_StopModule_TPU_U〈*ユニット番号*〉	TPUを停止

8ビットタイマ (TMR)

生成関数	機能
R_PG_Timer_Start_TMR_U〈*ユニット番号*〉(_C〈*チャネル番号*〉)	TMRを設定しカウント動作を開始
R_PG_Timer_HaltCount_TMR_U〈*ユニット番号*〉(_C〈*チャネル番号*〉)	TMRのカウント動作を一時停止
R_PG_Timer_ResumeCount_TMR_U〈*ユニット番号*〉(_C〈*チャネル番号*〉)	TMRのカウント動作を再開
R_PG_Timer_GetCounterValue_TMR_U〈*ユニット番号*〉(_C〈*チャネル番号*〉)	TMRのカウンタ値を取得
R_PG_Timer_SetCounterValue_TMR_U〈*ユニット番号*〉(_C〈*チャネル番号*〉)	TMRのカウンタ値を設定
R_PG_Timer_GetRequestFlag_TMR_U〈*ユニット番号*〉(_C〈*チャネル番号*〉)	TMRの割り込み要求フラグの取得とクリア
R_PG_Timer_HaltCountElc_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉	ELCにより開始したTMRのカウント動作を一時停止
R_PG_Timer_GetCountStateElc_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉	ELCによるタイマカウント状態を取得
R_PG_Timer_StopModule_TMR_U〈*ユニット番号*〉	TMRのユニットを停止

コンペアマッチタイマ (CMT)

生成関数	機能

R_PG_Timer_Set_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTの設定
R_PG_Timer_StartCount_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTのカウント動作を開始/再開
R_PG_Timer_HaltCount_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTのカウント動作を一時停止
R_PG_Timer_GetCounterValue_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTのカウンタ値を取得
R_PG_Timer_SetCounterValue_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTのカウンタ値を設定
R_PG_Timer_StopModule _CMT_U〈*ユニット番号*〉	CMTのユニットを停止

リアルタイムクロック（RTCb）

生成関数	機能
R_PG_RTC_Start	RTCを設定しカウント動作を開始
R_PG_RTC_WarmStart	ウォームスタート時RTCを再設定しカウント動作を開始
R_PG_RTC_Stop	RTCのカウント動作を一時停止
R_PG_RTC_Restart	RTCのカウント動作を再開
R_PG_RTC_SetCurrentTime	現在時刻の設定
R_PG_RTC_GetStatus	RTCの状態を取得
R_PG_RTC_Adjust30sec	30秒調整を行う
R_PG_RTC_ManualErrorAdjust	時計誤差を補正
R_PG_RTC_Set24HourMode	RTCを24時間モードに設定
R_PG_RTC_Set12HourMode	RTCを12時間モードに設定
R_PG_RTC_AutoErrorAdjust_Enable	時計誤差自動補正有効化
R_PG_RTC_AutoErrorAdjust_Disable	時計誤差自動補正無効化
R_PG_RTC_Alarm_Control	アラームの有効化/無効化
R_PG_RTC_SetAlarmTime	アラーム時刻の設定
R_PG_RTC_SetPeriodicInterrupt	周期割り込みの周期設定
R_PG_RTC_ClockOut_Enable	クロック出力の有効化
R_PG_RTC_ClockOut_Disable	クロック出力の無効化
R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉Enable	時間キャプチャ有効化
R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉Disable	時間キャプチャ無効化
R_PG_RTC_GetCaptureTime〈*時間キャプチャイベント入力端子番号*〉	キャプチャ時間取得

ウォッチドッグタイマ（WDTA）

生成関数	機能
R_PG_Timer_Start_WDT	WDTを設定しカウント動作を開始
R_PG_Timer_RefreshCounter_WDT	カウンタのリフレッシュ
R_PG_Timer_GetStatus_WDT	WDTのステータスフラグとカウント値を取得

独立ウォッチドッグタイマ（IWDTa）

生成関数	機能
R_PG_Timer_Start_IWDT	IWDTの設定と開始
R_PG_Timer_RefreshCounter_IWDT	カウンタのリフレッシュ
R_PG_Timer_GetStatus_IWDT	IWDTのステータスフラグとカウント値を取得

シリアルコミュニケーションインタフェース（SCIc, SCId）

生成関数	機能
R_PG_SCI_Set_C〈*チャネル番号*〉	シリアルI/Oチャネルの設定
R_PG_SCI_SendTargetStationID_C〈*チャネル番号*〉	データ送信先IDの送信
R_PG_SCI_StartSending_C〈*チャネル番号*〉	シリアルデータの送信開始
R_PG_SCI_SendAllData_C〈*チャネル番号*〉	シリアルデータを全て送信

R_PG_SCI_I2CMode_Send_C〈*チャネル番号*〉	簡易I²Cモードのデータ送信
R_PG_SCI_I2CMode_SendWithoutStop_C〈*チャネル番号*〉	簡易I²Cモードのデータ送信(STOP条件無し)
R_PG_SCI_I2CMode_GenerateStopCondition_C〈*チャネル番号*〉	STOP条件の生成
R_PG_SCI_I2CMode_Receive_C〈*チャネル番号*〉	簡易I²Cモードのデータ受信
R_PG_SCI_I2CMode_RestartReceive_C〈*チャネル番号*〉	簡易I²Cモードのデータ受信(RE-START条件)
R_PG_SCI_I2CMode_ReceiveLast_C〈*チャネル番号*〉	簡易I²Cモードの受信完了
R_PG_SCI_I2CMode_GetEvent_C〈*チャネル番号*〉	簡易I²Cモードの検出イベントの取得
R_PG_SCI_SPIMode_Transfer_C〈*チャネル番号*〉	簡易SPIモードのデータ転送
R_PG_SCI_SPIMode_GetErrorFlag_C〈*チャネル番号*〉	簡易SPIモードのシリアル受信エラーフラグの取得
R_PG_SCI_GetSentDataCount_C〈*チャネル番号*〉	シリアルデータの送信数取得
R_PG_SCI_ReceiveStationID_C〈*チャネル番号*〉	自局IDと一致するIDコードの受信
R_PG_SCI_StartReceiving_C〈*チャネル番号*〉	シリアルデータの受信開始
R_PG_SCI_ReceiveAllData_C〈*チャネル番号*〉	シリアルデータを全て受信
R_PG_SCI_ControlClockOutput_C〈*チャネル番号*〉	SCKn端子出力を切り替え(n : 0, 1, 5, 6, 8, 9, 12)
R_PG_SCI_StopCommunication_C〈*チャネル番号*〉	シリアルデータの送受信停止
R_PG_SCI_GetReceivedDataCount_C〈*チャネル番号*〉	シリアルデータの受信数取得
R_PG_SCI_GetReceptionErrorFlag_C〈*チャネル番号*〉	シリアル受信エラーフラグの取得
R_PG_SCI_ClearReceptionErrorFlag_C〈*チャネル番号*〉	シリアル受信エラーフラグのクリア
R_PG_SCI_GetTransmitStatus_C〈*チャネル番号*〉	シリアルデータ送信状態の取得
R_PG_SCI_StopModule_C〈*チャネル番号*〉	シリアルI/Oチャネルの停止

I²Cバスインタフェース（RIIC）

生成関数	機能
R_PG_I2C_Set_C〈*チャネル番号*〉	I²Cバスインタフェースチャネルの設定
R_PG_I2C_MasterReceive_C〈*チャネル番号*〉	マスタのデータ受信
R_PG_I2C_MasterReceiveLast_C〈*チャネル番号*〉	マスタのデータ受信終了
R_PG_I2C_MasterSend_C〈*チャネル番号*〉	マスタのデータ送信
R_PG_I2C_MasterSendWithoutStop_C〈*チャネル番号*〉	マスタのデータ送信(STOP条件無し)
R_PG_I2C_GenerateStopCondition_C〈*チャネル番号*〉	マスタのSTOP条件生成
R_PG_I2C_GetBusState_C〈*チャネル番号*〉	バス状態の取得
R_PG_I2C_SlaveMonitor_C〈*チャネル番号*〉	スレーブのバス監視
R_PG_I2C_SlaveSend_C〈*チャネル番号*〉	スレーブのデータ送信
R_PG_I2C_GetDetectedAddress_C〈*チャネル番号*〉	検出したスレーブアドレスの取得
R_PG_I2C_GetTR_C〈*チャネル番号*〉	送信/受信モードの取得
R_PG_I2C_GetEvent_C〈*チャネル番号*〉	検出イベントの取得
R_PG_I2C_GetReceivedDataCount_C〈*チャネル番号*〉	受信済みデータ数の取得
R_PG_I2C_GetSentDataCount_C〈*チャネル番号*〉	送信済みデータ数の取得
R_PG_I2C_Reset_C〈*チャネル番号*〉	バスのリセット
R_PG_I2C_StopModule_C〈*チャネル番号*〉	I²Cバスインタフェースチャネルの停止

シリアルペリフェラルインタフェース（RSPI）

生成関数	機能
R_PG_RSPI_Set_C〈*チャネル番号*〉	RSPIチャネルの設定
R_PG_RSPI_SetCommand_C〈*チャネル番号*〉	コマンドの設定
R_PG_RSPI_StartTransfer_C〈*チャネル番号*〉	データの転送開始
R_PG_RSPI_TransferAllData_C〈*チャネル番号*〉	全データの転送
R_PG_RSPI_GetStatus_C〈*チャネル番号*〉	転送状態の取得

R_PG_RSPI_GetError_C〈*チャネル番号*〉	エラー検出状態の取得
R_PG_RSPI_GetCommandStatus_C〈*チャネル番号*〉	コマンドステータスの取得
R_PG_RSPI_LoopBack〈*ループバックモード*〉_C〈*チャネル番号*〉	ループバックモードの設定
R_PG_RSPI_StopModule_C〈*チャネル番号*〉	RSPIチャネルの停止

CRC演算器（CRC）

生成関数	機能
R_PG_CRC_Set	CRC演算器の設定
R_PG_CRC_InputData	データの入力
R_PG_CRC_GetResult	演算結果の取得
R_PG_CRC_StopModule	CRC演算器の停止

12ビットA/Dコンバータ（S12ADb）

生成関数	機能
R_PG_ADC_12_Set_S12AD0	12ビットA/Dコンバータの設定
R_PG_ADC_12_StartConversionSW_S12AD0	A/D変換の開始（ソフトウェアトリガ）
R_PG_ADC_12_StopConversion_S12AD0	A/D変換の中断
R_PG_ADC_12_GetResult_S12AD0	アナログ入力、温度センサ出力または内部基準電圧をA/D変換した結果の取得
R_PG_ADC_12_GetResult_DblTrigger_S12AD0	ダブルトリガ(2回目)によるA/D変換結果の取得
R_PG_ADC_12_GetResult_SelfDiag_S12AD0	A/Dコンバータの自己診断でA/D変換した結果の取得
R_PG_ADC_12_StopModule_S12AD0	12ビットA/Dコンバータの停止

D/A コンバータ（DA）

生成関数	機能
R_PG_DAC_Set_C〈*チャネル番号*〉	D/Aコンバータのチャネルを設定
R_PG_DAC_SetWithInitialValue_C〈*チャネル番号*〉	初期値を指定してD/Aコンバータのチャネルを設定
R_PG_DAC_StopOutput_C〈*チャネル番号*〉	アナログ出力の停止
R_PG_DAC_ControlOutput_C〈*チャネル番号*〉	D/A変換値の設定
R_PG_DAC_Set_C0_C1	D/Aコンバータのチャネルを設定（DA0, DA1一括設定）

温度センサ（TEMPSa）

生成関数	機能
R_PG_TS_Set	温度センサの設定
R_PG_TS_StartPGA	温度センサPGAの動作開始
R_PG_TS_StopPGA	温度センサPGAの動作停止
R_PG_TS_StopModule	温度センサの停止

コンパレータA（CMPA）

生成関数	機能
R_PG_CPA_Set_CP〈*コンパレータ回路番号*〉	コンパレータnの設定(n : A1, A2)
R_PG_CPA_Disable_CP〈*コンパレータ回路番号*〉	コンパレータn回路の無効化(n : A1, A2)
R_PG_CPA_GetStatus	コンパレータAのステータスフラグを取得

コンパレータB（CMPB）

生成関数	機能
R_PG_CPB_Set_CPB〈*チャネル番号*〉	コンパレータBのチャネルを設定
R_PG_CPB_GetStatusFlag_CPB〈*チャネル番号*〉	モニタフラグ値を取得

R_PG_CPB_StopModule_CPB〈*チャネル番号*〉	コンパレータBのチャネルを停止

データ演算回路（DOC）

生成関数	機能
R_PG_DOC_Set	データ演算回路の設定
R_PG_DOC_GetStatusFlag	データ演算回路のステータスフラグの取得
R_PG_DOC_GetResult	データ演算結果の取得
R_PG_DOC_InputData	演算対象データの設定
R_PG_DOC_UpdateData	演算データの更新
R_PG_DOC_StopModule	データ演算回路を停止

5.1 クロック発生回路

5.1.1 R_PG_Clock_Set

定義 bool R_PG_Clock_Set(void)

概要 クロックの設定

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Set

詳細

- 各クロックスースを設定し、発振を開始します。
- 内部クロックソースをGUIで指定したクロックに切り替えます。
- システムクロック(ICLK)、周辺モジュールクロックB(PCLKB)、周辺モジュールクロックD(PCLKD)、FlaxhIFクロック(FCLK)、外部バスクロック(BCLK) の周波数を設定します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //クロック発生回路を設定し、クロックの発振を開始
      R_PG_Clock_Set();
}
```

5.1.2 R_PG_Clock_WaitSet

定義 bool R_PG_Clock_WaitSet(double wait_time)

概要 クロックの設定(発振安定時間ウェイト)

引数

double wait_time	発振安定待機時間(秒)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Set, R_CGC_Control

詳細

- 各クロックスースを設定し、発振を開始します。
- 内部クロックソースをGUIで指定したクロックに切り替えます。
- クロックソースを切り替える前に引数で指定された時間のウェイトを挿入します。ウェイトを挿入しない場合は、R_PG_Clock_Setを使用してください。
- 実際のウェイト時間が指定した値と異なる場合があります。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
    //クロック発生回路を設定し、0.5秒ウェイト後にクロックソース切り替え
    R_PG_Clock_WaitSet(0.5);
}
```

5.1.3 R_PG_Clock_Start_MAIN

定義 bool R_PG_Clock_Start_MAIN(void)

概要 メインクロックの発振開始

生成条件 GUI上でメインクロックまたはPLL回路を使用する設定をした場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- メインクロックの発振を開始します。
- GUI上でメインクロックを設定した場合、R_PG_Clock_Setによりメインクロックが設定され、発振を開始します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //メインクロックの発振開始
      R_PG_Clock_Start_MAIN();
}
```

5.1.4 R_PG_Clock_Stop_MAIN

定義 bool R_PG_Clock_Stop_MAIN(void)

概要 メインクロックの発振停止

生成条件 GUI上でメインクロックまたはPLL回路を使用する設定をした場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- メインクロックの発振を停止します。
- メインクロックまたはPLLクロックを内部クロックソースとして使用している場合は、メインクロックを停止することはできません。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //メインクロックの発振停止
      R_PG_Clock_Stop_MAIN();
}
```

5.1.5 R_PG_Clock_Start_SUB

定義 bool R_PG_Clock_Start_SUB(void)

概要 サブクロックの発振開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- サブクロックの発振を開始します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //サブクロックの発振開始
      R_PG_Clock_Start_SUB();
}
```

5.1.6 R_PG_Clock_Stop_SUB

定義 bool R_PG_Clock_Stop_SUB(void)

概要 サブクロックの発振停止

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- サブクロックの発振を停止します。
- サブクロックを内部クロックソースとして使用している場合は、サブクロックを停止することはできません。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //サブクロックの発振停止
      R_PG_Clock_Stop_SUB();
}
```

5.1.7 R_PG_Clock_Start_LOCO

定義 bool R_PG_Clock_Start_LOCO(void)

概要 低速オンチップオシレータ (LOCO)の発振開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- 低速オンチップオシレータ (LOCO)の発振を開始します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //低速オンチップオシレータ (LOCO)の発振開始
    R_PG_Clock_Start_LOCO();
}
```

5.1.8 R_PG_Clock_Stop_LOCO

定義 bool R_PG_Clock_Stop_LOCO(void)

概要 低速オンチップオシレータ (LOCO)の発振停止

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- 低速オンチップオシレータ (LOCO)の発振を停止します。
- 低速オンチップオシレータ (LOCO)を内部クロックソースとして使用している場合は、LOCOを停止することはできません。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
    //低速オンチップオシレータ (LOCO)の発振停止
    R_PG_Clock_Stop_LOCO();
}
```

5.1.9 R_PG_Clock_Start_HOCO

定義 bool R_PG_Clock_Start_HOCO(void)

概要 高速オンチップオシレータ (HOCO)の発振開始

生成条件 GUI上で高速オンチップオシレータ (HOCO)を使用する設定をした場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- 高速オンチップオシレータ (HOCO)の発振を開始します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //高速オンチップオシレータ (HOCO)の発振開始
    R_PG_Clock_Start_HOCO();
}
```

5.1.10 R_PG_Clock_Stop_HOCO

定義 bool R_PG_Clock_Stop_HOCO(void)

概要 高速オンチップオシレータ (HOCO)の発振停止

生成条件 GUI上で高速オンチップオシレータ (HOCO)を使用する設定をした場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- 高速オンチップオシレータ (HOCO)の発振を停止します。
- 高速オンチップオシレータ (HOCO)を内部クロックソースとして使用している場合は、HOCOを停止することはできません。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
    //高速オンチップオシレータ (HOCO)の発振停止
    R_PG_Clock_Stop_HOCO();
}
```

5.1.11 R_PG_Clock_PowerON_HOCO

定義 bool R_PG_Clock_PowerON_HOCO(void)

概要 高速オンチップオシレータ (HOCO)の電源をONにする

生成条件 GUI上で高速オンチップオシレータ (HOCO)を使用する設定をした場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- 高速オンチップオシレータ (HOCO)の電源をONにします。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //高速オンチップオシレータ (HOCO)の電源ON
    R_PG_Clock_PowerON_HOCO();
}
```

5.1.12 R_PG_Clock_PowerOFF_HOCO

定義 bool R_PG_Clock_PowerON_HOCO(void)

概要 高速オンチップオシレータ (HOCO)の電源をOFFにする

生成条件 GUI上で高速オンチップオシレータ (HOCO)を使用する設定をした場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- 高速オンチップオシレータ (HOCO)の電源をOFFにします。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //高速オンチップオシレータ (HOCO)の電源OFF
    R_PG_Clock_PowerOFF_HOCO();
}
```

5.1.13 R_PG_Clock_Start_PLL

定義 bool R_PG_Clock_Start_PLL(void)

概要 PLL回路の動作開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- PLL回路の動作を開始します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //PLL回路の動作開始
      R_PG_Clock_Start_PLL();
}
```

5.1.14 R_PG_Clock_Stop_PLL

定義 bool R_PG_Clock_Stop_PLL(void)

概要 PLL回路の動作停止

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- PLL回路の動作を停止します。
- PLL回路を内部クロックソースとして使用している場合は、PLL回路を停止することはできません。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //PLL回路の動作停止
      R_PG_Clock_Stop_PLL();
}
```

5.1.15 R_PG_Clock_Enable_BCLK_PinOutput

定義 bool R_PG_Clock_Enable_BCLK_PinOutput(void)

概要 BCLK端子出力の有効化

生成条件 GUI上でBCLK端子出力を設定した場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- BCLK端子からのクロック出力を有効にします。
- BCLK端子のクロックは、外部バス有効時に出力されます。
- GUI上でBCLK端子からの出力を有効に設定した場合、R_PG_Clock_SetによりBCLK端子出力が有効になります。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //BCLK端子出力の有効化
      R_PG_Clock_Enable_BCLK_PinOutput();
}
```

5.1.16 R_PG_Clock_Disable_BCLK_PinOutput

定義 bool R_PG_Clock_Disable_BCLK_PinOutput(void)

概要 BCLK端子出力の無効化

生成条件 GUI上でBCLK端子出力を設定した場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- BCLK端子からのクロック出力を無効にします。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
    //BCLK端子出力の無効化
    R_PG_Clock_Disable_BCLK_PinOutput();
}
```

5.1.17 R_PG_Clock_Enable_MAIN_StopDetection

定義 bool R_PG_Clock_Enable_MAIN_StopDetection(void)

概要 メインクロック発振停止検出機能の有効化

生成条件 GUI上でメインクロック発振停止検出機能を設定した場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- メインクロック発振停止検出機能を有効にします。
- GUI上でメインクロック発振停止検出機能を有効に設定した場合は、R_PG_Clock_Setでメインクロック発振停止検出機能が設定され、有効になります。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //メインクロック発振停止検出機能の有効化
      R_PG_Clock_Enable_MAIN_StopDetection();
}
```

5.1.18 R_PG_Clock_Disable_MAIN_StopDetection

定義 bool R_PG_Clock_Disable_MAIN_StopDetection(void)

概要 メインクロック発振停止検出機能の無効化

生成条件 GUI上でメインクロック発振停止検出機能を設定した場合

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細

- メインクロック発振停止検出機能を無効にします。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
    //メインクロック発振停止検出機能の無効化
    R_PG_Clock_Disable_MAIN_StopDetection();
}
```

5.1.19 R_PG_Clock_GetFlag_MAIN_StopDetection

定義 bool R_PG_Clock_GetFlag_MAIN_StopDetection (bool* stop)

概要 メインクロック発振停止検出フラグの取得

生成条件 GUI上でメインクロック発振停止検出機能を設定した場合

bool* stop	メインクロック発振停止検出フラグの格納先

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_GetStatus

詳細
- メインクロック発振停止検出フラグを取得します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

bool stop;

void func(void)
{
      //メインクロック発振停止フラグの取得
      R_PG_Clock_GetFlag_MAIN_StopDetection( &stop );
}
```

5.1.20 R_PG_Clock_ClearFlag_MAIN_StopDetection

定義 bool R_PG_Clock_ClearFlag_MAIN_StopDetection (void)

概要 メインクロック発振停止検出フラグのクリア

生成条件 GUI上でメインクロック発振停止検出機能を設定した場合

引数

なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_Control

詳細
- メインクロック発振停止検出フラグをクリアします。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
    //メインクロック発振停止フラグのクリア
    R_PG_Clock_ClearFlag_MAIN_StopDetection();
}
```

5.1.21 R_PG_Clock_GetSelectedClockSource

定義 bool R_PG_Clock_GetSelectedClockSource (uint8_t* clock)

概要 現在の内部クロックソースの取得

uint8_t* clock	内部クロックソースに対応する値の格納先 格納される値に対応するクロックソース 0:低速オンチップオシレータ 1:高速オンチップオシレータ 2:メインクロック 3:サブクロック 4:PLL回路

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_GetStatus

詳細
- 現在選択されている内部クロックソースを取得します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

uint8_t clock;

void func(void)
{
      //現在選択の内部クロックソースの取得
      R_PG_Clock_GetSelectedClockSource( &clock );
}
```

5.1.22 R_PG_Clock_GetClocksStatus

定義

bool R_PG_Clock_GetClocksStatus
(bool* pll, bool* main, bool* sub, bool* loco, bool* iwdt, bool* hoco)

概要

クロック発振状態の取得

bool* pll	PLL停止ビットの値の格納先 (0:動作 1:停止)
bool* main	メインクロック発振停止ビットの格納先 (0:動作 1:停止)
bool* sub	サブクロック発振停止ビットの格納先 (0:動作 1:停止)
bool* loco	低速オンチップオシレータ停止ビットの格納先 (0:動作 1:停止)
bool* iwdt	IWDT専用低速オンチップオシレータ停止ビットの格納先 (0:動作 1:停止)
bool* hoco	高速オンチップオシレータ停止ビットの格納先 (0:動作 1:停止)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Clock.c

使用RPDL関数

R_CGC_GetStatus

詳細

- 各クロックの発振状態を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目に対応する引数には0を指定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool loco;

void func(void)
{
    //低速オンチップオシレータの発振状態を取得
    R_PG_Clock_GetClocksStatus ( 0,  0,  0,  &loco,  0,  0 );
}
```

5.1.23 R_PG_Clock_GetHOCOPowerStatus

定義 bool R_PG_Clock_GetHOCOPowerStatus (bool* power)

概要 高速オンチップオシレータ(HOCO)の電源状態取得

bool* power	高速オンチップオシレータ電源制御ビットの値の格納先 (0:ON 1:OFF)

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_GetStatus

詳細
- 高速オンチップオシレータ(HOCO)の電源状態を取得します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

bool power;

void func(void)
{
	//HOCOの電源状態取得
	R_PG_Clock_GetHOCOPowerStatus ( & power );
}
```

5.2 電圧検出回路 (LVDAa)

5.2.1 R_PG_LVD_Set

定義 bool R_PG_LVD_Set (void)

概要 電圧検出回路の設定(電圧監視1，電圧監視2一括設定)

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LVD.c

使用RPDL関数 R_LVD_Create

詳細

- 低電圧検出時の動作(内部リセットまたは割り込み)を設定します。
- 1回の呼び出しで電圧監視1と電圧監視2を設定することができます。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();  //クロックの設定

    //電圧検出回路の設定(電圧監視1，電圧監視2一括設定)
    R_PG_LVD_Set();
}
```

5.2.2 R_PG_LVD_GetStatus

定義 bool R_PG_LVD_GetStatus
(bool * lvd1_detect, bool * lvd1_monitor, bool * lvd2_detect, bool * lvd2_monitor)

概要 電圧検出回路のステータスフラグを取得

引数

bool * lvd1_detect	電圧監視1電圧変化検出フラグの格納先
bool * lvd1_monitor	電圧監視1信号モニタフラグの格納先
bool * lvd2_detect	電圧監視2電圧変化検出フラグの格納先
bool * lvd2_monitor	電圧監視2信号モニタフラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LVD.c

使用RPDL関数 R_LVD_GetStatus

詳細
- 電圧検出回路のステータスフラグを取得します。
- 取得しないフラグには0を指定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool lvd1_det, lvd2_det;
bool lvd1_mon, lvd2_mon;

void func(void)
{
    //電圧検出回路のステータスフラグを取得
    R_PG_LVD_GetStatus( &lvd1_det, &lvd1_mon, &lvd2_det, &lvd2_mon );

    if( lvd1_det ){
        //電圧監視1電圧変化検出時処理
    }
    if( lvd2_det ){
        //電圧監視2電圧変化検出時処理
    }
}
```

5.2.3 R_PG_LVD_ClearDetectionFlag_LVD〈*電圧検出回路番号*〉

定義　bool R_PG_LVD_ClearDetectionFlag_LVD〈*電圧検出回路番号*〉(void)
〈*電圧検出回路番号*〉： 1, 2

概要　電圧監視n電圧変化検出フラグのクリア　　n ： 1, 2

引数　なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル　R_PG_LVD.c

使用RPDL関数　R_LVD_Control

詳細
- 電圧監視n電圧変化検出フラグをクリアします。　　n ： 1, 2

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //電圧監視1電圧変化検出フラグのクリア
    R_PG_LVD_ClearDetectionFlag_LVD1();
}
```

5.2.4 R_PG_LVD_Disable_LVD〈*電圧検出回路番号*〉

定義　bool R_PG_LVD_Disable_LVD〈*電圧検出回路番号*〉(void)

〈*電圧検出回路番号*〉： 1, 2

概要　電圧監視nの無効化　　n ： 1, 2

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_LVD.c

使用RPDL関数　R_LVD_Control

詳細

- 電圧監視nを無効化します。　　n ： 1, 2

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //電圧監視1の無効化
    R_PG_LVD_Disable_LVD1();
}
```

5.3 クロック周波数精度測定回路（CAC）

5.3.1 R_PG_CAC_Set

定義 bool R_PG_CAC_Set(void)

概要 クロック周波数精度測定回路の設定と測定の開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Create

詳細

- クロック周波数精度測定回路(CAC)を設定し、測定を開始します。
- 本関数を使用する前に、R_CGC_Setによりクロックを設定してください。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
      //クロックの設定
      R_PG_Clock_Set();

      //CACの設定と測定の開始
      R_PG_LVD_Set (void);
}
```

5.3.2 R_PG_CAC_ClearFlag_FrequencyError

定義 bool R_PG_CAC_ClearFlag_FrequencyError(void)

概要 周波数エラーフラグのクリア

生成条件 GUI上で周波数エラー割り込み(FERRF)が設定された場合

引数

なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Control

詳細
- 周波数エラーフラグをクリアします。

使用例 GUI上で以下の通り設定した場合
- 周波数エラー割り込み(FERRF)を設定
- 周波数エラー割り込み(FERRF)通知関数名に CacErrIntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void CacErrIntFunc(void)
{
    //周波数エラー割り込み発生時処理
    func2();

    //周波数エラーフラグのクリア
    R_PG_CAC_ClearFlag_FrequencyErro();
}

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //CACの設定と測定の開始
    R_PG_LVD_Set (void);
}
```

5.3.3 R_PG_CAC_ClearFlag_MeasurementEnd

定義 bool R_PG_CAC_ClearFlag_MeasurementEnd(void)

概要 測定終了フラグのクリア

生成条件 GUI上で測定終了割り込み(MENDF)が設定された場合

引数

なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Control

詳細
- 測定終了フラグをクリアします。

使用例 GUI上で以下の通り設定した場合
- 測定終了割り込み(MENDF)を設定
- 測定終了割り込み(MENDF)通知関数名に CacEndIntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void CacEndIntFunc(void)
{
    //測定終了割り込み発生時処理
    func2();

    //測定終了フラグのクリア
    R_PG_CAC_ClearFlag_MeasurementEnd();
}

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //CACの設定と測定の開始
    R_PG_LVD_Set (void);
}
```

5.3.4 R_PG_CAC_ClearFlag_Overflow

定義 bool R_PG_CAC_ClearFlag_Overflow(void)

概要 オーバフローフラグのクリア

生成条件 GUI上でオーバフロー割り込み(OVFF)が設定された場合

引数

なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Control

詳細
- オーバフローフラグをクリアします。

使用例 GUI上で以下の通り設定した場合
- オーバフロー割り込み(OVFF)を設定
- オーバフロー割り込み(OVFF)通知関数名に CacOvIntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void CacOvIntFunc(void)
{
    //オーバフロー割り込み発生時処理
    func2();

    //オーバフローフラグのクリア
    R_PG_CAC_ClearFlag_Overflow();
}

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //CACの設定と測定の開始
    R_PG_LVD_Set (void);
}
```

5.3.5 R_PG_CAC_StartMeasurement

定義 bool R_PG_CAC_StartMeasurement(void)

概要 クロック周波数測定の開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Control

詳細

- R_PG_CAC_StopMeasurementにより停止した測定を再開します。.

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func1(void)
{
	//クロック周波数測定の停止
	R_PG_CAC_StopMeasurement();
}

void func2(void)
{
	//クロック周波数測定の再開
	R_PG_CAC_StartMeasurement();
}
```

5.3.6 R_PG_CAC_StopMeasurement

定義 bool R_PG_CAC_StopMeasurement(void)

概要 クロック周波数測定の停止

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Control

詳細
- 測定を停止します。.

使用例

R_PG_CAC_StartMeasurementの使用例を参照してください。

5.3.7 R_PG_CAC_GetStatusFlags

定義 bool R_PG_CAC_GetStatusFlags(bool *err, bool *end, bool *ov)

概要 CACステータスフラグの取得

bool *err	周波数エラーフラグ格納先
bool *end	測定終了フラグ格納先
bool *ov	オーバフローフラグ格納先

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_GetStatus

詳細
- 周波数エラーフラグ、測定終了フラグ、オーバフローフラグを取得します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

bool    g_err;
bool    g_end;
bool    g_ov;

void func(void)
{
      //CACステータスフラグの取得
      bool R_PG_CAC_GetStatusFlags(&g_err, &g_end, &g_ov);
}
```

5.3.8 R_PG_CAC_GetCounterBufferRegister

定義 bool R_PG_CAC_GetCounterBufferRegister(uint16_t *cacntbr_val)

概要 カウンタバッファレジスタ(CACNTBR)値の取得

uint16_t *cacntbr_val	カウンタバッファレジスタ(CACNTBR)値格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_GetStatus

詳細
- カウンタバッファレジスタ(CACNTBR)値を取得します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

uint16_t cacntbr_val;

void func(void)
{
    //カウンタバッファレジスタ値の取得
    R_PG_CAC_GetCounterBufferRegister( &cacntbr_val );
}
```

5.3.9 R_PG_CAC_StopModule

定義 bool R_PG_CAC_StopModule(void)

概要 クロック周波数精度測定回路の停止

引数

なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_CAC.c

使用RPDL関数 R_CAC_Destroy

詳細

- クロック周波数精度測定回路(CAC)を停止します。

使用例

```
//この関数を使用するには”R_PG_<プロジェクト名>.h”をインクルードしてください
#include “R_PG_default.h”

void func(void)
{
	//CACの停止
	R_PG_CAC_StopModule();
}
```

5.4 消費電力低減機能

5.4.1 R_PG_LPC_Set

定義 bool R_PG_LPC_Set (void)

概要 消費電力低減機能の設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Create

詳細

- 消費電力低減機能を設定します。
- GUI上でクロックの発振安定待ち時間を設定した場合は、発振安定待ち時間を設定したクロックが停止した状態で本関数を呼び出してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //サブクロックの停止
    R_PG_Clock_Stop_SUB();
    //消費電力低減機能の設定
    R_PG_LPC_Set (void);
    //サブクロックの発振開始
    R_PG_Clock_Start_SUB();
    //クロック発生回路を設定し、2秒ウェイト後にクロックソース切り替え
    R_PG_Clock_WaitSet(2);
}
```

5.4.2 R_PG_LPC_Sleep

定義 bool R_PG_LPC_Sleep (void)

概要 スリープモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- スリープモードへ移行します

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //スリープモードへの移行
    R_PG_LPC_Sleep(void);
}
```

5.4.3 R_PG_LPC_AllModuleClockStop

定義 bool R_PG_LPC_AllModuleClockStop (void)

概要 全モジュールクロックスタンバイモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- 全モジュールクロックスタンバイモードへ移行します。
- 全モジュールクロックスタンバイモードへ移行する前に、全モジュールクロックスタンバイモード中の動作を許可するTMRユニットを設定します。
- 初期設定では全モジュールクロックスタンバイモード中にTMRは停止します。全モジュールクロックスタンバイモード中にTMRを動作させるには、動作させるTMRのユニットをGUI上で選択してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //全モジュールクロックスタンバイモードへの移行
    R_PG_LPC_AllModuleClockStop (void);
}
```

5.4.4 R_PG_LPC_SoftwareStandby

定義 bool R_PG_LPC_SoftwareStandby(void)

概要 ソフトウェアスタンバイモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- ソフトウェアスタンバイモードへ移行します。
- 本関数を呼ぶ前にR_PG_LPC_Setを呼び出して、ソフトウェアスタンバイモード中の動作を設定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //消費電力低減機能の設定
    R_PG_LPC_Set (void);

    //ソフトウェアスタンバイモードへの移行
    R_PG_LPC_SoftwareStandby (void);
}
```

5.4.5 R_PG_LPC_DeepSoftwareStandby

定義 bool R_PG_LPC_DeepSoftwareStandby(void)

概要 ディープソフトウェアスタンバイモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- ディープソフトウェアスタンバイモードへ移行します。
- 本関数を呼ぶ前にR_PG_LPC_Setを呼び出して、ディープソフトウェアスタンバイモード中の動作と解除要因を設定してください。
- ディープソフトウェア解除フラグはディープソフトウェアモード以外の場合も解除要求が発生すると”1”になります。本関数ではディープソフトウェアスタンバイモードに移行する前にディープソフトウェア解除フラグはクリアされません。
  R_PD_LPC_GetDeepSoftwareStandbyCancelFlagによりディープソフトウェア解除フラグをクリアしてからディープソフトウェアスタンバイモードに移行してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //消費電力低減機能の設定
    R_PG_LPC_Set (void);

    //ディープソフトウェアスタンバイ解除フラグのクリア
    R_PD_LPC_GetDeepSoftwareStandbyCancelFlag(0,0,0,0,0,0,0,0);

    //ディープソフトウェアスタンバイモードへの移行
    R_PG_LPC_DeepSoftwareStandby (void);
}
```

5.4.6 R_PG_LPC_IOPortRelease

定義 bool R_PG_LPC_IOPortRelease (void)

概要 I/Oポート出力保持を解除

生成条件 GUI上で[I/Oポート状態保持]に［ディープソフトウェアスタンバイ解除後のIOKEEPビットへの"0"書き込みで保持を解除]を選択した場合に出力されます。

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- ディープソフトウェアスタンバイ解除後のI/Oポートの出力保持状態を解除します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //I/Oポートの出力保持状態を解除
    R_PG_LPC_IOPortRelease(void);
}
```

5.4.7 R_PG_LPC_ChangeOperatingPowerControl

定義 bool R_PG_LPC_ChangeOperatingPowerControl(uint8_t mode)

概要 動作電力制御モードを変更

引数

uint8_t mode	動作電力制御モード 0:高速動作モード 1:中速動作モード1A 2:中速動作モード1B 3:低速動作モード1 4:低速動作モード2 5:中速動作モード2A 6:中速動作モード2B 中速動作モード2Aおよび中速動作モード2Bは、チップバージョンBの製品(型名:R5F5210xBxxx)のみ選択可能です。

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- 動作電力制御モードを変更します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //動作電力制御モードを中速モードAに変更
    R_PG_LPC_ChangeOperatingPowerControl( 1 );
}
```

5.4.8 R_PG_LPC_ChangeSleepModeReturnClock

定義 bool R_PG_LPC_ChangeSleepModeReturnClock(uint8_t return_clock)

概要 スリープモード復帰クロックソースを変更

引数

uint8_t return_clock	スリープモード復帰クロックソース 0:切り替え無効 1:HOCO 2:メインクロック

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- スリープモード復帰クロックソースを変更します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //スリープモード復帰クロックソースをHOCOに変更
    R_PG_LPC_ChangeSleepModeReturnClock( 1 );
}
```

5.4.9 R_PG_LPC_GetPowerOnResetFlag

定義 bool R_PG_LPC_GetPowerOnResetFlag (bool * reset)

概要 パワーオンリセットフラグの取得

引数

bool * reset	パワーオンリセットフラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- パワーオンリセットフラグを取得します
- 本関数を呼び出すとリセット検出フラグおよびディープソフトウェアスタンバイ解除要求フラグがクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。
- RSTSR.PORF(パワーオンリセットフラグ)は端子リセットでのみクリアされます。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool reset;

void func(void)
{
    //パワーオンリセットフラグの取得
    R_PG_LPC_GetPowerOnResetFlag( &reset );

    if( reset ){
        //パワーオンリセット検出時処理
    }
}
```

5.4.10 R_PG_LPC_GetLVDDetectionFlag

定義 bool R_PG_LPC_GetLVDDetectionFlag (bool * lvd0, bool * lvd1, bool * lvd2)

概要 LVD検知フラグの取得

引数

bool * lvd0	LVD0検知フラグの格納先
bool * lvd1	LVD1検知フラグの格納先
bool * lvd2	LVD2検知フラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- LVD検知フラグを取得します。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
- 取得しないフラグには0を指定してください。
- 本関数を呼び出すとリセット検出フラグおよびディープソフトウェアスタンバイ解除要求フラグがクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool lvd1;
bool lvd2;

void func(void)
{
    //LVD1、LVD2検出フラグの取得
    R_PG_LPC_GetLVDDetectionFlag ( 0, &lvd1, &lvd2 );

    if( lvd1 ){
        //LVD1検出時処理
    }
    if( lvd2 ){
        //LVD2検出時処理
    }
}
```

5.4.11 R_PG_LPC_GetDeepSoftwareStandbyResetFlag

定義 bool R_PG_LPC_GetDeepSoftwareStandbyResetFlag(bool *reset)

概要 ディープソフトウェアスタンバイリセットフラグの取得

引数

bool *reset	ディープソフトウェアスタンバイリセットフラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- ディープソフトウェアスタンバイリセットフラグを取得します
- 本関数を呼び出すとリセット検出フラグおよびディープソフトウェアスタンバイ解除要求フラグがクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool reset;

void func(void)
{
    //ディープソフトウェアスタンバイリセットフラグの取得
    R_PG_LPC_GetDeepSoftwareStandbyResetFlag ( &reset);

    if( reset ){
        //ディープソフトウェアスタンバイリセット検出時の処理
    }
}
```

5.4.12 R_PD_LPC_GetDeepSoftwareStandbyCancelFlag

定義

```
bool R_PD_LPC_GetDeepSoftwareStandbyCancelFlag
(bool * irq0,       bool * irq1,       bool * irq2,       bool * irq3,
 bool * irq4,       bool * irq5,       bool * irq6,       bool * irq7,
 bool * lvd1,       bool * lvd2,       bool * rtc_period, bool * rtc_alarm,
 bool * nmi,        bool * sda,        bool * scl )
```

概要 ディープソフトウェアスタンバイ解除要求フラグの取得

引数

bool *irq0	IRQ0ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq1	IRQ1ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq2	IRQ2ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq3	IRQ3ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq4	IRQ4ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq5	IRQ5ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq6	IRQ6ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq7	IRQ7ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *lvd1	LVD1ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *lvd2	LVD2ディープソフトウェアスタンバイ解除要求フラグの格納先
bool * rtc_period	RTC周期割り込みディープソフトウェアスタンバイ解除要求フラグの格納先
bool * rtc_alarm	RTCアラーム割り込みディープソフトウェアスタンバイ解除要求フラグの格納先
bool *nmi	NMIディープソフトウェアスタンバイ解除要求フラグの格納先
bool * sda	SDA-DSディープソフトウェアスタンバイ解除要求フラグの格納先
bool * scl	SCL-DSディープソフトウェアスタンバイ解除要求フラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- ディープソフトウェアスタンバイ解除要求フラグを取得します。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
- 取得しないフラグには0を指定してください。
- 本関数を呼び出すとリセット検出フラグおよびディープソフトウェアスタンバイ解除要求フラグがクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
bool irq0;
bool nmi;
void func(void)
{
    // ディープソフトウェアスタンバイ解除要求フラグを取得 (IQR0-A および NMI)
    R_PD_LPC_GetDeepSoftwareStandbyCancelFlag (
        &irq0,  0,  0,  0,
        0,      0,  0,  0,
        0,      0,  0,  0,
        &nmi,   0,  0       );
    if( irq0 ){
      // IRQ0-Aからのディープソフトウェアスタンバイ解除要求検出時処理
    }
    if( nmi ){
      //NMIからのディープソフトウェアスタンバイ解除要求検出時処理
    }
}
```

5.4.13 R_PG_LPC_GetOperatingPowerControlFlag

定義 bool R_PG_LPC_GetOperatingPowerControlFlag(bool * during_transition)

概要 動作電力制御モード遷移状態フラグの取得

引数

bool * during_transition	動作電力制御モード遷移状態フラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- 動作電力制御モード遷移状態フラグを取得します。
- 本関数を呼び出すとリセット検出フラグおよびディープソフトウェアスタンバイ解除要求フラグがクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool during_transition;

void func(void)
{
    //動作電力制御モード遷移状態フラグの取得
    R_PD_LPC_GetDeepSoftwareStandbyCancelFlag ( &during_transition );
}
```

5.4.14 R_PG_LPC_GetStatus

定義 bool R_PG_LPC_GetStatus(uint32_t *data)

概要 消費電力低減機能の状態を取得

引数

uint32_t *data	ステータス情報の格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_LPC.h

使用RPDL関数 R_LPC_GetStatus

詳細

- リセットステータスとディープソフトウェアスタンバイ解除要求フラグを取得します。
- 本関数を呼び出すと、RPDLの関数 R_LPC_GetStatus が直接呼び出されます。
- 取得した情報は以下の形式で格納されます。

b31-b25	b24
0	動作電力制御モード遷移状態 0:遷移完了 1:遷移中

b23	b22-b20	b19	b18	b17	b16
リセットステータス(RSTSR) (0: 未検出; 1:検出)					
ディープソフトウェアリセット	0	LVD2	LVD1	LVD0	パワーオンリセット

b15	b14	b13	b12	b11	b10	b9	b8
ディープソフトウェアスタンバイ解除要求検出(DPSIFR) (0: 未検出; 1:検出)							
0	IIC (SCL)	IIC (SDA)	NMI	RTC アラーム	RTC 周期	LVD2	LVD1

b7	b6	b5	b4	b3	b2	b1	b0
ディープソフトウェアスタンバイ解除要求検出(DPSIFR) (0: 未検出; 1:検出)							
IRQ7 -DS	IRQ6 -DS	IRQ5 -DS	IRQ4 -DS	IRQ3 -DS	IRQ2 -DS	IRQ1 -DS	IRQ0 -DS

- 本関数を呼び出すとRSTSR.LVD1F(LVD1検知フラグ)、RSTSR.LVD2F(LVD2検知フラグ)、RSTSR.DPSRSTF(ディープソフトウェアスタンバイリセットフラグ)およびDPSIFR(ディープソフトウェアスタンバイ解除要求フラグ)がクリアされます。
- RSTSR.PORF(パワーオンリセットフラグ)は端子リセットでのみクリアされます。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
uint16_t data;
void func(void)
{
    //消費電力低減機能の状態を取得
    R_PG_LPC_GetStatus( &data );
    //ディープソフトウェアリセットを検出したか
    if( (data >> 15) & 0x1 ){
        if( (data >> 7) &0x1){
            //NMIによるディープソフトウェアスタンバイ解除時処理
        }
        else if( data &0x1){
            //IRQ0-Aによるディープソフトウェアスタンバイ解除時処理
        }
    }
}
```

5.4.15 R_PG_LPC_WriteBackup

定義 bool R_PG_LPC_WriteBackup (uint8_t * data, uint8_t count)

概要 ディープスタンバイバックアップレジスタへの書き込み

引数

uint8_t * data	ディープスタンバイバックアップレジスタに書き込むデータ
uint8_t count	書き込むデータのバイト数 (1～32)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.h

使用RPDL関数 R_LPC_WriteBackup

詳細

- ディープスタンバイバックアップレジスタにデータを書き込みます。
- 本関数を呼び出すと、RPDLの関数 R_LPC_WriteBackup が直接呼び出されます。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t w_data[]="ABCDEFG";
uint8_t r_data[]="--------";

void func1(void)
{
    //消費電力低減機能の設定
    R_PG_LPC_Set (void);

    //ディープスタンバイバックアップレジスタへの書き込み
    R_PG_LPC_WriteBackup( w_data, 7 );

    //ディープソフトウェアスタンバイモードへの移行
    R_PG_LPC_DeepSoftwareStandby (void);
}

void func2(void)
{
    //ディープスタンバイバックアップレジスタからの読み出し
    R_PG_LPC_ReadBackup( r_data, 7 );
}
```

5.4.16 R_PG_LPC_ReadBackup

定義 bool R_PG_LPC_ReadBackup (uint8_t * data, uint8_t count)

概要 ディープスタンバイバックアップレジスタからの読み出し

引数

uint8_t * data	読み出したデータの保存先
uint8_t count	読み出すデータのバイト数 (1～32)

戻り値

true	読み出しに成功した場合
false	読み出しに失敗した場合

出力先ファイル R_PG_LPC.h

使用RPDL関数 R_LPC_ReadBackup

詳細

- ディープスタンバイバックアップレジスタからデータを読み出します。
- 本関数を呼び出すと、RPDLの関数 R_LPC_ReadBackup が直接呼び出されます。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t w_data[]="ABCDEFG";
uint8_t r_data[]="--------";

void func1(void)
{
    //消費電力低減機能の設定
    R_PG_LPC_Set (void);

    //ディープスタンバイバックアップレジスタへの書き込み
    R_PG_LPC_WriteBackup( w_data, 7 );

    //ディープソフトウェアスタンバイモードへの移行
    R_PG_LPC_DeepSoftwareStandby (void);
}

void func2(void)
{
    //ディープスタンバイバックアップレジスタからの読み出し
    R_PG_LPC_ReadBackup( r_data, 7 );
}
```

5.5 レジスタライトプロテクション機能

5.5.1 R_PG_RWP_RegisterWriteCgc

定義 bool R_PG_RWP_RegisterWriteCgc (bool enable)

概要 クロック発生回路関連レジスタへの書き込みを許可/禁止

引数

bool enable	レジスタへの書き込み設定（1:許可　0:禁止）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_Control

詳細

- クロック発生回路関連レジスタへの書き込みを許可/禁止します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool  cgc;
bool  mode_lpc_reset;
bool  lvd;
bool  b0wi,pfswe;

void func1(void)
{
    //クロック発生回路関連レジスタへの書き込みを許可
    R_PG_RWP_RegisterWriteCgc( 1 );

    //動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタ
    //への書き込みを許可
    R_PG_RWP_RegisterWriteModeLpcReset( 1 );

    /電圧レギュレータ制御レジスタ（VRCR)への書き込みを許可
    R_PG_RWP_RegisterWriteVrcr( 1 );

    //LVD関連レジスタへの書き込みを許可
    R_PG_RWP_RegisterWriteLvd( 1 );

    //端子機能選択レジスタへの書き込みを許可
    R_PG_RWP_RegisterWriteMpc( 1 );
}

void func2(void)
{
    //クロック発生回路関連レジスタへの書き込みを禁止
    R_PG_RWP_RegisterWriteCgc( 0 );

    //動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタ
    //への書き込みを禁止
    R_PG_RWP_RegisterWriteModeLpcReset( 0 );

    //電圧レギュレータ制御レジスタ（VRCR)への書き込みを禁止
    R_PG_RWP_RegisterWriteVrcr( 0 );

    //LVD関連レジスタへの書き込みを禁止
```

```
    R_PG_RWP_RegisterWriteLvd( 0 );

    //端子機能選択レジスタへの書き込みを禁止
    R_PG_RWP_RegisterWriteMpc( 0 );
}

void func3(void)
{
    //クロック発生回路関連レジスタへの書き込み状態の取得
    R_PG_RWP_GetStatusCgc(&cgc);

    //動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタ
    //への書き込み状態の取得
    R_PG_RWP_GetStatusModeLpcReset(&mode_lpc_reset);

    //電圧レギュレータ制御レジスタ（VRCR)への書き込み状態の取得
    R_PG_RWP_GetStatusVrcr(&vrcr);

    //LVD関連レジスタへの書き込み状態の取得
    R_PG_RWP_GetStatusLvd(&lvd);

    //端子機能選択レジスタへの書き込み状態の取得
    R_PG_RWP_GetStatusMpc(&b0wi, &pfswe);
}
```

5.5.2 R_PG_RWP_RegisterWriteModeLpcReset

定義 bool R_PG_RWP_RegisterWriteModeLpcReset (bool enable)

概要 動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込みを許可/禁止

引数

bool enable	レジスタへの書き込み設定（1:許可　0:禁止）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_Control

詳細
- 動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込みを許可/禁止します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.1 R_PG_RWP_RegisterWriteVrcr

定義 bool R_PG_RWP_RegisterWriteVrcr (bool enable)

概要 電圧レギュレータ制御レジスタ(VRCR)への書き込みを許可/禁止

引数

bool enable	レジスタへの書き込み設定（1:許可　0:禁止）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_Control

詳細
- 電圧レギュレータ制御レジスタ（VRCR)への書き込みを許可/禁止します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.2 R_PG_RWP_RegisterWriteLvd

定義 bool R_PG_RWP_RegisterWriteLvd (bool enable)

概要 LVD関連レジスタへの書き込みを許可/禁止

引数

bool enable	レジスタへの書き込み設定（1:許可　0:禁止）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_Control

詳細
- LVD関連レジスタへの書き込みを許可/禁止します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.3 R_PG_RWP_RegisterWriteMpc

定義 bool R_PG_RWP_RegisterWriteMpc (bool enable)

概要 端子機能選択レジスタへの書き込みを許可/禁止

引数

bool enable	レジスタへの書き込み設定（1:許可　0:禁止）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_Control

詳細
- 端子機能選択レジスタへの書き込みを許可/禁止します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.4 R_PG_RWP_GetStatusCgc

定義 bool R_PG_RWP_GetStatusCgc (bool * cgc)

概要 クロック発生回路関連レジスタへの書き込み状態の取得

引数

bool * cgc	クロック発生回路関連レジスタへのレジスタへの書き込み状態 (1:許可 0:禁止)

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_GetStatus

詳細
- クロック発生回路関連レジスタへの書き込み状態を取得します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.5 R_PG_RWP_GetStatusModeLpcReset

定義 bool R_PG_RWP_GetStatusModeLpcReset (bool * mode_lpc_reset)

概要 動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込み状態の取得

引数

bool * mode_lpc_reset	動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込み状態（1:許可　0:禁止）

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_GetStatus

詳細
- 動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込み状態を取得します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.1 R_PG_RWP_GetStatusVrcr

定義 bool R_PG_RWP_GetStatusLvd (bool * vrcr)

概要 電圧レギュレータ制御レジスタ(VRCR)への書き込み状態の取得

引数

bool * vrcr	VRCRへの書き込み状態（1:許可　0:禁止）

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_GetStatus

詳細
- 電圧レギュレータ制御レジスタ(VRCR)への書き込み状態を取得します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.2 R_PG_RWP_GetStatusLvd

定義 bool R_PG_RWP_GetStatusLvd (bool * lvd)

概要 LVD関連レジスタへの書き込み状態の取得

引数

bool * lvd	LVD関連レジスタへの書き込み状態（1:許可　0:禁止）

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_GetStatus

詳細
- LVD関連レジスタへの書き込み状態を取得します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.5.3 R_PG_RWP_GetStatusMpc

定義 bool R_PG_RWP_GetStatusMpc (bool * b0wi, bool * pfswe)

概要 端子機能選択レジスタへの書き込み状態の取得

引数

bool * b0wi	PWPRレジスタPFSWEビットへの書き込み状態（1:禁止　0:許可）
bool * pfswe	PFSレジスタへの書き込み状態（1:許可　0:禁止）

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_RWP.c

使用RPDL関数 R_RWP_GetStatus

詳細
- 端子機能選択レジスタへの書き込み状態を取得します。

使用例 R_PG_RWP_RegisterWriteCgcの使用例を参照してください。

5.6 割り込みコントローラ (ICUb)

5.6.1 R_PG_ExtInterrupt_Set_〈割り込み種別〉

定義　bool R_PG_ExtInterrupt_Set_〈割り込み種別〉(void)
〈割り込み種別〉 : IRQ0～IRQ7、またはNMI

概要　外部割り込みの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ExtInterrupt_〈割り込み種別〉.c
〈割り込み種別〉 : IRQ0～IRQ7、またはNMI

使用RPDL関数　R_INTC_SetExtInterrupt, R_INTC_CreateExtInterrupt

詳細

- 使用する外部割り込み端子を有効にするために、MPCのレジスタを設定します。また端子を入力として使用するために、I/Oポートのレジスタを設定します。IRQnは[周辺機能別使用端子]ウィンドウ上の選択に従い、使用端子の設定を行います。
- GUI上で割り込み通知関数名が指定されている場合、CPUへ割り込みが発生すると指定された名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void 〈割り込み通知関数名〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み優先レベルを0に設定した場合、外部割り込み要求信号が入力されてもCPU割り込みは発生しません。割り込み要求フラグは R_PG_ExtInterrupt_GetRequestFlag_〈割り込み種別〉 により取得することができ、R_PG_ExtInterrupt_ClearRequestFlag_〈割り込み種別〉 によりクリアすることができます。
- GUI上で[デジタルフィルタを有効にする]を指定した場合、本関数を呼び出すとデジタルフィルタが有効になります。

使用例1　割り込み通知関数名にIrq0IntFuncを指定する場合

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
}

//IRQ0通知関数
void Irq0IntFunc(void)
{
    func_irq0();    //IRQ0の処理
}
```

使用例2 割り込み優先レベルを0に設定した場合

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
	bool flag;

	//IRQ0を設定する
	R_PG_ExtInterrupt_Set_IRQ0();

	do{
		//IRQ0の割り込み要求フラグを取得する
		R_PG_ExtInterrupt_GetRequestFlag_IRQ0( &flag );
	}while( ! flag );

	func_irq0();	//IRQ0の処理

	//IRQ0の割り込み要求フラグをクリアする
	R_PG_ExtInterrupt_ClearRequestFlag_IRQ0();
}
```

5.6.2 R_PG_ExtInterrupt_Disable_*〈割り込み種別〉*

定義 bool R_PG_ExtInterrupt_Disable_*〈割り込み種別〉*(void)

〈割り込み種別〉 ： IRQ0～IRQ7

概要 外部割り込みの設定解除

引数 なし

戻り値

true	設定解除が正しく行われた場合
false	設定解除に失敗した場合

出力先ファイル R_PG_ExtInterrupt_*〈割り込み種別〉*.c

〈割り込み種別〉 ： IRQ0～IRQ7

使用RPDL関数 R_INTC_ControlExtInterrupt

詳細

- 外部割り込み(IRQ0～IRQ7) を無効にします。
- 外部割り込みに使用した端子の設定(MPC及びI/Oポートのレジスタ設定)は保持されます。
- 割り込み端子を無効にした時、割り込み要求フラグは自動的にクリアされます。
- 割り込み端子が無効になる前に有効な割り込みが発生すると、GUI上で割り込み通知関数名が指定されている場合、指定された名前の関数が呼び出されます。

使用例 割り込み通知関数名にIrq0IntFuncを指定する場合

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
}

//外部割り込み(IRQ0)通知関数
void Irq0IntFunc (void)
{
    //IRQ0を無効にする
    R_PG_ExtInterrupt_Disable_IRQ0();

    func_irq0();    //IRQ0の処理
}
```

5.6.3 R_PG_ExtInterrupt_GetRequestFlag_*〈割り込み種別〉*

定義　bool R_PG_ExtInterrupt_GetRequestFlag_*〈割り込み種別〉*(bool * flag)

〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

概要　外部割り込み要求フラグの取得

引数

bool * flag	割り込み要求フラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル　R_PG_ExtInterrupt_*〈割り込み種別〉*.c

〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

使用RPDL関数　R_INTC_GetExtInterruptStatus

詳細

- 外部割り込み(IRQ0～IRQ7、またはNMI) の割り込み要求フラグを取得します。
  割り込み要求がある場合、flagで指定した格納先にtrueが入ります。

使用例　R_PG_ExtInterrupt_Set_*〈割り込み種別〉*の使用例2を参照してください。

5.6.4 R_PG_ExtInterrupt_ClearRequestFlag_*〈割り込み種別〉*

定義 bool R_PG_ExtInterrupt_ClearRequestFlag_*〈割り込み種別〉*(void)

〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

概要 外部割り込み要求フラグのクリア

引数 なし

戻り値

true	フラグのクリアに成功した場合
false	フラグのクリアに失敗した場合

出力先ファイル R_PG_ExtInterrupt_*〈割り込み種別〉*.c

〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

使用RPDL関数 R_INTC_ControlExtInterrupt

詳細
- 外部割り込み(IRQ0～IRQ7、またはNMI) の割り込み要求フラグをクリアします。
- 割り込みがLowレベル検出の場合、要求フラグは割り込み入力端子へのHighレベル入力でクリアされます。Lowレベル検出の場合は本関数により外部割込み要求フラグをクリアできません。

使用例 R_PG_ExtInterrupt_Set_*〈割り込み種別〉*の使用例2を参照してください。

5.6.5 R_PG_ExtInterrupt_EnableFilter_*<割り込み種別>*

定義 bool R_PG_ExtInterrupt_EnableFilter_*<割り込み種別>*(uint32_t div)

<割り込み種別> ： IRQ0～IRQ7、またはNMI

概要 デジタルフィルタの再有効化

生成条件 GUI上で[デジタルフィルタを有効にする]を指定した場合

引数

uint32_t div	周辺モジュールクロック分周値 1: デジタルフィルタサンプリングクロック = PCLK 8: デジタルフィルタサンプリングクロック = PCLK/8 32: デジタルフィルタサンプリングクロック = PCLK/32 64: デジタルフィルタサンプリングクロック = PCLK/64

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ExtInterrupt_*<割り込み種別>*.c

<割り込み種別> ： IRQ0～IRQ7、またはNMI

使用RPDL関数 R_INTC_ControlExtInterrupt

詳細

- R_PG_ExtInterrupt_DisableFilter_*<割り込み種別>*にて無効化されたデジタルフィルタを有効にし、デジタルフィルタサンプリングクロックの再設定を行います。

使用例 GUI上で[IRQ0を使用する]を指定した場合
([デジタルフィルタを有効にする]を指定)

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定

    //IRQ0を設定(デジタルフィルタ有効)
    R_PG_ExtInterrupt_Set_IRQ0();
}

void func2(void)
{
    //デジタルフィルタの無効化
    R_PG_ExtInterrupt_DisableFilter_IRQ0();

    //デジタルフィルタの再有効化
    R_PG_ExtInterrupt_EnableFilter_IRQ0( 1 );
}
```

5.6.6 R_PG_ExtInterrupt_DisableFilter_*〈割り込み種別〉*

定義 bool R_PG_ExtInterrupt_DisableFilter_*〈割り込み種別〉*(void)

〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

概要 デジタルフィルタの無効化

生成条件 GUI上で[デジタルフィルタを有効にする]を指定した場合

引数 なし

戻り値

true	無効化が正しく行われた場合
false	無効化に失敗した場合

出力先ファイル R_PG_ExtInterrupt_*〈割り込み種別〉*.c

〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

使用RPDL関数 R_INTC_ControlExtInterrupt

詳細
- デジタルフィルタを無効化します。
- ソフトウェアスタンバイモードに移行する際は、デジタルフィルタを無効化してください。ソフトウェアスタンバイモードからの復帰後に再度デジタルフィルタを使用する場合は、R_PG_ExtInterrupt_EnableFilter_*〈割り込み種別〉*を呼び出してください。

使用例 R_PG_ExtInterrupt_EnableFilter_*〈割り込み種別〉*の使用例を参照してください。

5.6.7 R_PG_SoftwareInterrupt_Set

定義 bool R_PG_SoftwareInterrupt_Set(void)

概要 ソフトウェア割り込みの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_SoftwareInterrupt.c

使用RPDL関数 R_INTC_CreateSoftwareInterrupt

詳細
- ソフトウェア割り込みを設定します。
- 本関数の呼び出しではソフトウェア割り込みは発生しません。ソフトウェア割り込みを発生させるには本関数の呼出し後にR_PG_SoftwareInterrupt_Generateを呼び出してください。

使用例 GUI上でソフトウェア割り込み通知関数名に SwIntFunc を指定した場合

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void SwIntFunc(void);

void func(void)
{
    //ソフトウェア割り込みを設定する
    R_PG_SoftwareInterrupt_Set();

    //ソフトウェア割り込みを発生させる
    R_PG_SoftwareInterrupt_Generate();
}

void SwIntFunc(void)
{
    //ソフトウェア割り込みの処理
}
```

5.6.8 R_PG_SoftwareInterrupt_Generate

定義 bool R_PG_SoftwareInterrupt_Generate(void)

概要 ソフトウェア割り込みの生成

引数 なし

戻り値

true	生成が正しく行われた場合
false	生成に失敗した場合

出力先ファイル R_PG_SoftwareInterrupt.c

使用RPDL関数 R_INTC_Write

詳細
- ソフトウェア割り込みを発生させます。
- 本関数を呼び出す前にR_PG_SoftwareInterrupt_Setを呼び出してソフトウェア割り込みを設定してください。

使用例 R_PG_SoftwareInterrupt_Setの使用例を参照してください。

5.6.9 R_PG_FastInterrupt_Set

定義 bool R_PG_FastInterrupt_Set (void)

概要 高速割り込みの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_FastInterrupt.c

使用RPDL関数 R_INTC_CreateFastInterrupt

詳細

- GUI上で指定した割り込み要因を高速割り込みに設定します。指定する割り込み要因の設定と有効化は行いません。高速割り込みに設定する割り込み要因の設定と有効化は、周辺機能の関数により行ってください。
- 本関数では高速割り込みベクタレジスタ(FINTV)を設定するために無条件トラップ(BRK命令)を使用しています。割り込みが無効の状態(プロセッサステータスワードの割り込み許可ビット(I)が0の場合)には、本関数はロックします。
- GUI上で高速割込みに指定した割り込みのハンドラは、#pragma interruptでfintを指定してコンパイルすることにより高速割り込みとして処理されます。

使用例 GUI上でIRQ0を高速割り込みに指定した場合

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //IRQ0を高速割り込みに設定する
    R_PG_FastInterrupt_Set();

    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
}
```

5.6.10 R_PG_Exception_Set

定義　bool R_PG_Exception_Set (void)

概要　例外ハンドラの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Exception.c

使用RPDL関数　R_INTC_CreateExceptionHandlers

詳細

- 例外通知関数を設定します。GUI上で例外通知関数名が指定されている場合、本関数の呼び出し後に例外が発生すると、指定された名前の関数が呼び出されます。
  例外通知関数は次の定義で作成してください。
  void *〈例外通知関数名〉*(void)
  例外通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。

使用例　GUI上で次の例外通知関数を設定した場合
特権命令例外 ： PrivInstExcFunc
未定義命令例外 ： UndefInstExcFunc

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //例外ハンドラの設定
    R_PG_Exception_Set();
}

void PrivInstExcFunc(){
    func_pi_excep();    //特権命令例外発生時の処理
}

void UndefInstExcFunc (){
    func_ui_excep();    //未定義命令例外発生時の処理
}
```

5.7 バス

5.7.1 R_PG_ExtBus_PresetBus

定義　bool R_PG_ExtBus_PresetBus(void)

概要　バスプライオリティの設定

生成条件　GUI上でバスプライオリティを設定した場合

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ExtBus.c

使用RPDL関数　R_BSC_Set

詳細

- バスプライオリティを設定します。
- バスプライオリティを設定する場合は、R_PG_ExtBus_SetBusを呼び出す前に本関数を呼んでください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_ExtBus_PresetBus();   //バスプライオリティの設定
    R_PG_ExtBus_SetBus();   //バス端子とバスエラー監視の設定
}
```

5.7.2 R_PG_ExtBus_SetBus

定義 bool R_PG_ExtBus_SetBus(void)

概要 バス端子とバスエラー監視の設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ExtBus.c

使用RPDL関数 R_BSC_Create

詳細

- バス端子とバスエラー監視を設定します。
- 本関数内でバスエラー割り込みを設定します。GUI上でバスエラー割り込みが有効に設定された場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void *〈割り込み通知関数名〉*(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- バスエラーの検出状態はR_PG_ExtBus_GetErrorStatusにより取得することができます。
- 外部バスクロック(BCLK)はR_PG_Clock_Setにより設定してください。
- バスプライオリティを設定する場合は、本関数を呼び出す前にR_PG_ExtBus_PresetBusを呼んでください。

使用例 バスエラー割り込み通知関数名にBusErrFuncを指定した場合

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_ExtBus_SetBus();   //バス端子とバスエラー監視の設定
}

//バスエラー通知関数
void BusErrFunc(void)
{
    bool addr_err;
    uint8_t master;
    uint16_t err_addr;

    //バスエラー検出状態の取得
    R_PG_ExtBus_GetErrorStatus( &addr_err,  0,  &master,  &err_addr );
    if( addr_err ){
        //不正アドレスアクセスエラー検出時処理
    }

    //バスエラーステータスレジスタのクリア
    R_PG_ExtBus_ClearErrorFlags();
}
```

5.7.3 R_PG_ExtBus_SetArea_CS〈*CS領域の番号*〉

定義 bool R_PG_ExtBus_SetArea_CS〈*CS領域の番号*〉(void)

〈*CS領域の番号*〉 : 0～3

概要 CS領域の設定

生成条件 GUI上で外部空間を設定した場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ExtBus_CS〈*CS領域の番号*〉.c

〈*CS領域の番号*〉 : 0～3

使用RPDL関数 R_BSC_CreateArea

詳細

- CS領域を設定します。
- 本関数を呼び出す前にR_PG_ExtBus_SetBusを呼び出して、使用する端子とバスエラー監視を設定してください。

使用例 CS1およびCS2を設定する場合

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //バス端子とバスエラー監視の設定
    R_PG_ExtBus_SetBus();

    //CS1の設定
    R_PG_ExtBus_SetArea_CS1();

    //CS2の設定
    R_PG_ExtBus_SetArea_CS2();

    //外部バスの有効化
    R_PG_ExtBus_SetEnable();
}
```

5.7.4 R_PG_ExtBus_SetEnable

定義 bool R_PG_ExtBus_SetEnable(void)

概要 外部バスの有効化

生成条件 GUI上で外部空間を設定した場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ExtBus.c

使用RPDL関数 R_BSC_Control

詳細
- 外部バスを有効にします。
- 本関数を呼び出す前に R_PG_ExtBus_SetBus と R_PG_ExtBus_SetArea_CS*<CS領域の番号>* を呼び出して、使用する端子とバスエラー監視、CS領域を設定してください。

使用例 R_PG_ExtBus_SetArea_CS*<CS領域の番号>*の使用例を参照してください。

5.7.5 R_PG_ExtBus_GetErrorStatus

定義

bool R_PG_ExtBus_GetErrorStatus
(bool * addr_err, bool * time_err, uint8_t * master, uint16_t * err_addr)

概要

バスエラー検出状態の取得

生成条件

GUI上でバスエラー監視を設定した場合

引数

bool * addr_err	不正アドレスアクセスフラグの格納先
bool * time_err	タイムアウトフラグの格納先
uint8_t * master	バスエラーを発生させたバスマスタのIDコードの格納先 バスマスタに対応するIDコード: 0:CPU 3:DMAC/DTC
uint16_t * err_addr	バスエラーを起こしたアドレスの上位13ビットの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_ExtBus.c

使用RPDL関数

R_BSC_GetStatus

詳細

- バスエラーステータスレジスタからバスエラー検出状態を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。

使用例

バスエラー割り込み通知関数名にBusErrFuncを指定した場合

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_ExtBus_SetBus();   //バス端子とバスエラー監視の設定
}

//バスエラー通知関数
void BusErrFunc(void)
{
    bool addr_err;
    uint8_t master;
    uint16_t err_addr;

    //バスエラー検出状態の取得
    R_PG_ExtBus_GetErrorStatus( &addr_err, 0, &master, &err_addr );
    if( addr_err ){
        //不正アドレスアクセスエラー検出時処理
    }

    //バスエラーステータスレジスタのクリア
    R_PG_ExtBus_ClearErrorFlags();
}
```

5.7.6 R_PG_ExtBus_ClearErrorFlags

定義 bool R_PG_ExtBus_ClearErrorFlags(void)

概要 バスエラーステータスレジスタのクリア

生成条件 GUI上でバスエラー監視を設定した場合

引数 なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_ExtBus.c

使用RPDL関数 R_BSC_Control

詳細

- バスエラーステータスレジスタ（不正アドレスアクセスフラグ、タイムアウトフラグ、バスマスタIDコード、アクセス先アドレスの値）をクリアします。

使用例 バスエラー割り込み通知関数名にBusErrFuncを指定した場合

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
	//バス端子とバスエラー監視の設定
	R_PG_ExtBus_SetBus();
}

//バスエラー通知関数
void BusErrFunc(void)
{
	bool addr_err;
	uint8_t master;
	uint16_t err_addr;

	//バスエラー検出状態の取得
	R_PG_ExtBus_GetErrorStatus( &addr_err, 0, &master, &err_addr );
	if( addr_err ){
		//不正アドレスアクセスエラー検出時処理
	}

	//バスエラーステータスレジスタのクリア
	R_PG_ExtBus_ClearErrorFlags();
}
```

5.7.7 R_PG_ExtBus_DisableArea_CS〈*CS領域の番号*〉

定義 bool R_PG_ExtBus_DisableArea_CS〈*CS領域の番号*〉(void)

〈*CS領域の番号*〉： 0～3

概要 CS領域の設定解除

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ExtBus_CS〈*CS領域の番号*〉.c

〈*CS領域の番号*〉： 0～3

使用RPDL関数 R_BSC_Destroy

詳細

- CS領域の設定を解除します。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //バス端子とバスエラー監視の設定
    R_PG_ExtBus_SetBus();

    //CS0の設定
    R_PG_ExtBus_SetArea_CS0();
}

void func2(void)
{
    //CS0の設定解除
    R_PG_ExtBus_DisableArea_CS0();
}
```

5.7.8 R_PG_ExtBus_SetDisable

定義 bool R_PG_ExtBus_SetDisable(void)

概要 外部バスの無効化

生成条件 GUI上で外部空間を設定した場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ExtBus.c

使用RPDL関数 R_BSC_Control

詳細

- 外部バスを無効にします。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //外部バスの無効化
    R_PG_ExtBus_SetDisable(void);
}
```

5.8 DMAコントローラ（DMACA）

5.8.1 R_PG_DMAC_Set_C〈*チャネル番号*〉

定義　bool R_PG_DMAC_Set_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉： 0～3

概要　DMACの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_DMAC_C〈*チャネル番号*〉.c
〈*チャネル番号*〉： 0～3

使用RPDL関数　R_DMAC_Create

詳細

- DMACのモジュールストップ状態を解除して初期設定します。
- 転送開始要因に割り込みを選択した場合は、本関数を呼び出した後R_PG_DMAC_Activate_C〈*チャネル番号*〉を呼び出すことにより割り込みの入力待ち状態になります。転送開始要因にソフトウェアトリガを選択した場合は、本関数を呼び出した後R_PG_DMAC_StartTransfer_C〈*チャネル番号*〉またはR_PG_DMAC_StartContinuousTransfer_C〈*チャネル番号*〉を呼び出すことにより転送を開始します。
- GUI上で割り込み通知関数名を指定した場合、本関数内でDMA割り込みを設定します。CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。　void 〈*割り込み通知関数名*〉(void)
割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- シリアルの送信データをDMA転送する場合は、GUI上で以下の設定をしてください。

DMAC設定

転送開始要因	： TXI0 (SCI0 送信データエンプティ割り込み)
転送終了時処理	： 起動要因の割り込みフラグをクリアする
転送先開始アドレス	： トランスミットデータレジスタ(TDR)のアドレス *転送先開始アドレスはプログラムからも設定することができます。使用例2,3を参照してください。
転送先アドレス更新モード	： 固定
1データのビット長	： 1バイト

SCIc設定

データ送信方法	： DMAC により送信データを転送する

関数の使用方法については使用例2を参照してください。

- シリアルの受信データをDMA転送する場合は、GUI上で以下の設定をしてください。

DMAC設定

転送開始要因	： RXI0 (SCI0 受信データフル割り込み)
転送終了時処理	： 起動要因の割り込みフラグをクリアする
転送元開始アドレス	： レシーブデータレジスタ(RDR)のアドレス *転送元開始アドレスはプログラムからも設定することができます。使用例2,3を参照してください。
転送元アドレス更新モード	： 固定
1データのビット長	： 1バイト

SCIc設定

データ受信方法	： DMACにより受信データを転送する

関数の使用方法については使用例3を参照してください。

使用例1 IRQ0割り込みにより転送を開始する場合

- GUI上でDMAC0の転送開始要因をIRQ0割り込みに設定
- GUI上でDMA0割り込み通知関数名に Dmac0IntFunc を指定
- GUI上でIRQ0の割り込み要求先をDMACに設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_DMAC_Set_C0();  //DMAC0を設定する

    R_PG_ExtInterrupt_Set_IRQ0();  //IRQ0を設定する

    R_PG_DMAC_Activate_C0();  //DMAC0を転送開始トリガ入力待ち状態にする
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    R_PG_DMAC_StopModule_C0();  //DMACを停止
}
```

使用例2 DMA転送によりシリアル送信データを転送する場合

- GUI上でDMA0割り込み通知関数名に Dmac0IntFunc を指定
- SCI0の送信データエンプティ割り込みにより転送開始

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

volatile bool sci_dma_transfer_complete;  //DMA転送終了フラグ
uint8_t tr[]="ABCDEFG";  //送信データ

void func(void)
{
    //DMA転送終了フラグの初期化
    sci_dma_transfer_complete = false;

    R_PG_Clock_Set();  //クロックの設定

    //SCI0を設定する
    R_PG_SCI_Set_C0();

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMA転送元、転送先アドレス、転送カウンタを設定する
    R_PG_DMAC_SetSrcAddress_C0( tr );
    R_PG_DMAC_SetDestAddress_C0((void*)&(SCI0.TDR));
    R_PG_DMAC_SetTransferCount_C0( 8 );

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //SCI0の送信を有効にする（TXI割り込みが発生し、DMA転送が開始）
    R_PG_SCI_SendAllData_C0(
        PDL_NO_PTR,
        PDL_NO_DATA
    );

    //DMA転送終了を待つ
    while (sci_dma_transfer_complete == false);
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
```

```
    //シリアル送信終了フラグ
    bool sci_transfer_complete;
    sci_transfer_complete = false;

    //シリアル送信の終了を待つ
    do{
        R_PG_SCI_GetTransmitStatus_C0( &sci_transfer_complete );
    } while( ! sci_transfer_complete );

    //シリアル通信を停止
    R_PG_SCI_StopCommunication_C0();

   //DMACを停止
    R_PG_DMAC_StopModule_C0();

    sci_dma_transfer_complete = true;
}
```

使用例3　　DMA転送によりシリアル受信データを転送する場合

- GUI上でDMA0割り込み通知関数名に Dmac0IntFunc を指定
- SCI0の受信データフル割り込みにより転送開始

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

volatile bool sci_dma_transfer_complete;   //DMA転送終了フラグ
uint8_t re[]="--------";   //受信データ格納先

void func(void)
{
    //DMA転送終了フラグの初期化
    sci_dma_transfer_complete = false;

    //SCI0を設定する
    R_PG_SCI_Set_C0();

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMA転送元、転送先アドレス、転送カウンタを設定する
    R_PG_DMAC_SetSrcAddress_C0((void*)&(SCI0.RDR) );
    R_PG_DMAC_SetDestAddress_C0( re );
    R_PG_DMAC_SetTransferCount_C0( 8 );

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //SCI0の受信を開始する
    R_PG_SCI_ReceiveAllData_C0(
        PDL_NO_PTR,
        PDL_NO_DATA
    );
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //シリアル通信を停止
    R_PG_SCI_StopCommunication_C0();

    //DMACを停止
    R_PG_DMAC_StopModule_C0();
}
```

5.8.2 R_PG_DMAC_Activate_C〈*チャネル番号*〉

定義　bool R_PG_DMAC_Activate_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉： 0～3

概要　DMACを転送開始トリガの入力待ち状態に設定

生成条件　転送開始要因に割り込みを選択

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_DMAC_C〈*チャネル番号*〉.c
〈*チャネル番号*〉： 0～3

使用RPDL関数　R_DMAC_Control

詳細
- 転送開始要因を割り込みに設定したDMACのチャネルをトリガ入力待ち状態に設定します。
- 本関数は転送開始要因に割り込みを指定した場合に生成されます。
- あらかじめR_PG_DMAC_Set_C〈*チャネル番号*〉によりDMACのチャネルを設定してください。

使用例　GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0の転送開始要因をIRQ0割り込みに設定した場合
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMACを停止
    R_PG_DMAC_StopModule_C0();
}
```

5.8.3 R_PG_DMAC_StartTransfer_C〈*チャネル番号*〉

定義　bool R_PG_DMAC_StartTransfer_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉： 0～3

概要　DMA一転送の開始(ソフトウェアトリガ)

生成条件　転送開始要因にソフトウェアトリガを選択

引数　なし

戻り値

true	転送開始が正しく行われた場合
false	転送開始に失敗した場合

出力先ファイル　R_PG_DMAC_C〈*チャネル番号*〉.c

〈*チャネル番号*〉： 0～3

使用RPDL関数　R_DMAC_Control

詳細
- 転送開始要因をソフトウェアトリガに設定したチャネルのDMA転送を開始します。
- データ転送が開始されると、DMA転送要求は自動的にクリアされます。

使用例　GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0の転送開始要因をソフトウェアトリガに設定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

volatile bool transferred;

void func(void)
{
	transferred = false;

	//DMAC0を設定する
	R_PG_DMAC_Set_C0();

	while( transferred == false ){
		//DMAC0の転送を開始する
		R_PG_DMAC_StartTransfer_C0();
	}

	//DMACを停止
	R_PG_DMAC_StopModule_C0();
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
	transferred = true;
}
```

5.8.4 R_PG_DMAC_StartContinuousTransfer_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_StartContinuousTransfer_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉： 0～3

概要 DMA連続転送の開始(ソフトウェアトリガ)

生成条件 転送開始要因にソフトウェアトリガを選択

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C〈*チャネル番号*〉.c

〈*チャネル番号*〉： 0～3

使用RPDL関数 R_DMAC_Control

詳細
- 転送開始要因をソフトウェアトリガに設定したチャネルのDMA転送を開始します。
- 転送終了後に再度DMA転送要求が発生するため、連続したDMA転送が可能です。

使用例 GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0の転送開始要因をソフトウェアトリガに設定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0の転送を開始する
    R_PG_DMAC_StartContinuousTransfer_C0();
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMACを停止
    R_PG_DMAC_StopModule_C0();
}
```

5.8.5 R_PG_DMAC_StopContinuousTransfer_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_StopContinuousTransfer_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉： 0～3

概要 ソフトウェアトリガにより開始したDMA連続転送の停止

生成条件 転送開始要因にソフトウェアトリガを選択

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C〈*チャネル番号*〉.c

〈*チャネル番号*〉： 0～3

使用RPDL関数 R_DMAC_Control

詳細

- DMAソフトウェア起動レジスタ(DMREQ)のDMAソフトウェア起動ビット(SWREQ)およびDMAソフトウェア起動ビット自動クリア選択ビット(CLRS)を0に設定することにより、ソフトウェアトリガにより開始したDMA連続転送を停止します。

使用例 GUI上で以下の通り設定した場合

- ノーマル転送モードでDMAC0の転送開始要因をソフトウェアトリガに設定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0の転送を開始する
    R_PG_DMAC_StartContinuousTransfer_C0();
}

void func2(void)
{
    //ソフトウェアによるDMA転送要求のクリア
    R_PG_DMAC_StopContinuousTransfer_C0();
}
```

5.8.6 R_PG_DMAC_Suspend_C〈*チャネル番号*〉

定義　bool R_PG_DMAC_Suspend_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉： 0～3

概要　データ転送の中断

引数　なし

戻り値

true	中断に成功した場合
false	中断に失敗した場合

出力先ファイル　R_PG_DMAC_C〈*チャネル番号*〉.c
〈*チャネル番号*〉： 0～3

使用RPDL関数　R_DMAC_Control

詳細
- DMA転送を中断(禁止)します。
- 転送開始要因に割り込みを選択した場合、転送を再開するには転送開始要因の割り込み要求フラグをクリアし、R_PG_DMAC_Activate_C〈*チャネル番号*〉により割り込み入力待ち状態に設定してください。

使用例　GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0の転送開始要因をIRQ0割り込みに設定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定
- IRQ1の割り込み通知関数名にIrq1IntFuncを指定
- IRQ2の割り込み通知関数名にIrq2IntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_DMAC_Set_C0();      //DMAC0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();      //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ1();      //IRQ1を設定する
    R_PG_ExtInterrupt_Set_IRQ2();      //IRQ2を設定する
    R_PG_DMAC_Activate_C0();      //DMAC0を転送開始トリガ入力待ち状態にする
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    R_PG_DMAC_StopModule_C0();      //DMAC0を停止
}

//IRQ1割り込みでDMA転送停止
void Irq1IntFunc (void)
{
    R_PG_DMAC_Suspend_C0();      //DMAC0の転送を中断
}

//IRQ2割り込みでDMA転送再開
void Irq2IntFunc (void)
{
    R_PG_ExtInterrupt_ClearRequestFlag_IRQ0();      //トリガの要求フラグクリア
    R_PG_DMAC_Activate_C0();      //DMA転送有効化
}
```

5.8.7 R_PG_DMAC_GetTransferCount_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_GetTransferCount_C〈*チャネル番号*〉(uint16_t * count)

〈*チャネル番号*〉 : 0～3

概要 転送カウンタ値の取得

引数

uint16_t * count	転送カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_DMAC_C〈*チャネル番号*〉.c

〈*チャネル番号*〉 : 0～3

使用RPDL関数 R_DMAC_GetStatus

詳細

- 現在の転送カウンタの値を取得します。
- DMA割り込み要求フラグ(IRフラグ)は本関数内でクリアされます。DMA割り込み要求フラグを取得する必要がある場合は、本関数を呼び出す前に R_PG_DMAC_ClearInterruptFlag_C〈*チャネル番号*〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合

- DMAC0の転送開始要因に割り込みを選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    uint16_t count;

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //転送カウンタ値が10未満になるのを待つ
    do{
        R_PG_DMAC_GetTransferCount_C0( & count );
    } while( count >= 10 );

    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
}
```

5.8.8 R_PG_DMAC_SetTransferCount_C〈*チャネル番号*〉

定義

bool R_PG_DMAC_SetTransferCount_C〈*チャネル番号*〉(uint16_t count)

〈 *チャネル番号* 〉: 0～3

概要

転送カウンタ値の設定

引数

uint16_t count	転送カウンタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_DMAC_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～3

使用RPDL関数

R_DMAC_Control

詳細

- 転送カウンタの値を設定します。
- 有効な値はノーマル転送モードでは0～65535 (0:フリーランニングモード) 、リピート転送モードおよびブロック転送モードでは0～1023 (0:1024回)です。

使用例

GUI上で以下の通り設定した場合

- DMAC0の転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetSrcAddress_C0( src_address );   //転送元アドレス
    R_PG_DMAC_SetDestAddress_C0( dest_address );   //転送先アドレス
    R_PG_DMAC_SetTransferCount_C0( tr_count );   //転送カウンタ値
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.9 R_PG_DMAC_GetRepeatBlockSizeCount_C〈チャネル番号〉

定義 bool R_PG_DMAC_GetRepeatBlockSizeCount_C〈チャネル番号〉(uint16_t * count)
〈 チャネル番号 〉: 0～3

概要 リピート/ブロックサイズカウンタ値の取得

生成条件 転送モードにリピート転送モードまたはブロック転送モードを選択

引数

uint16_t * count	カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_DMAC_C 〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_GetStatus

詳細
- リピート/ブロックサイズカウンタの現在の値を取得します。
- DMA割り込み要求フラグ(IRフラグ)は本関数内でクリアされます。DMA割り込み要求フラグを取得する必要がある場合は、本関数を呼び出す前に R_PG_DMAC_ClearInterruptFlag_C〈チャネル番号〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- リピート転送モードでDMAC0を設定
- 転送開始要因は割り込み

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    uint16_t count;

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //リピートサイズカウンタ値が10未満になるのを待つ
    do{
        R_PG_DMAC_GetRepeatBlockSizeCount_C0( & count );
    } while( count >= 10 );

    //転送を中断
    R_PG_DMAC_Suspend_C0();
}
```

5.8.10 R_PG_DMAC_SetRepeatBlockSizeCount_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_SetRepeatBlockSizeCount_C〈*チャネル番号*〉(uint16_t count)
〈 *チャネル番号* 〉: 0～3

概要 リピート/ブロックサイズカウンタ値の設定

生成条件 転送モードにリピート転送モードまたはブロック転送モードを選択

引数

uint16_t count	リピート/ブロックサイズカウンタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C 〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- リピート/ブロックサイズカウンタの現在の値を設定します。
- 有効な値はリピート転送モードでは0～1023 (0:1024回)、ブロック転送モードでは1～1023です。

使用例 GUI上で以下の通り設定した場合
- リピート転送モードでDMAC0を設定
- 転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetTransferCount_C0( tr_count );   //転送カウンタ値
    R_PG_DMAC_SetRepeatBlockSizeCount_C0( repeat_count );   //リピートサイズカウンタ値
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.11 R_PG_DMAC_ClearInterruptFlag_C〈チャネル番号〉

定義 bool R_PG_DMAC_ClearInterruptFlag_C〈チャネル番号〉(bool* int_request)
〈チャネル番号〉: 0～3

概要 割り込み要求フラグの取得とクリア

生成条件 DMA割り込み有効時

引数

bool* int_request	割り込み要求フラグの格納先

戻り値

true	取得とクリアに成功した場合
false	取得とクリアに失敗した場合

出力先ファイル R_PG_DMAC_C 〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_GetStatus

詳細
- DMA割り込み要求フラグ(IRフラグ)を取得し、クリアします。

使用例 GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0を設定
- 転送開始要因は割り込み
- DMA割り込み有効
- DMA割り込み優先レベルは0

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
	bool int_request;

	//DMAC0を設定する
	R_PG_DMAC_Set_C0();

	//DMAC0を転送開始トリガ入力待ち状態にする
	R_PG_DMAC_Activate_C0();

	//IRフラグが1になるのを待つ
	do{
		R_PG_DMAC_ClearInterruptFlag_C0( & int_request );
	} while( int_request == false );
}
```

5.8.12 R_PG_DMAC_GetTransferEndFlag_C〈チャネル番号〉

定義 bool R_PG_DMAC_GetTransferEndFlag_C〈チャネル番号〉(bool* end)
〈チャネル番号〉: 0～3

概要 転送終了フラグの取得

引数

bool* end	転送終了フラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_DMAC_C 〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_GetStatus

詳細
- 転送終了フラグを取得します。
- DMA割り込み要求フラグ(IRフラグ)は本関数内でクリアされます。DMA割り込み要求フラグを取得する必要がある場合は、本関数を呼び出す前にR_PG_DMAC_ClearInterruptFlag_C〈チャネル番号〉を呼び出してください。
- 転送終了フラグは本関数内でクリアされません。転送終了フラグをクリアする必要がある場合はR_PG_DMAC_ClearTransferEndFlag_C〈チャネル番号〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0を設定
- 転送開始要因は割り込み
- DMA割り込み無効

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    bool end;

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //転送終了フラグが1になるのを待つ
    do{
        R_PG_DMAC_GetTransferEndFlag_C0( & end );
    } while( end == false );

    //転送終了フラグをクリアする
    R_PG_DMAC_ClearTransferEndFlag_C0();
}
```

5.8.13 R_PG_DMAC_ClearTransferEndFlag_C〈チャネル番号〉

定義 bool R_PG_DMAC_ClearTransferEndFlag_C〈チャネル番号〉(void)
〈チャネル番号〉: 0～3

概要 転送終了フラグのクリア

引数 なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_DMAC_C 〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- 転送終了フラグをクリアします。
- 転送終了フラグを取得するにはR_PG_DMAC_GetTransferEndFlag_C〈チャネル番号〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- ノーマル転送モードでDMAC0を設定
- 転送開始要因は割り込み
- DMA割り込み無効

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
	bool end;

	//DMAC0を設定する
	R_PG_DMAC_Set_C0();

	//DMAC0を転送開始トリガ入力待ち状態にする
	R_PG_DMAC_Activate_C0();

	//転送終了フラグが1になるのを待つ
	do{
		R_PG_DMAC_GetTransferEndFlag_C0( & end );
	} while( end == false );

	//転送終了フラグをクリアする
	R_PG_DMAC_ClearTransferEndFlag_C0();
}
```

5.8.14 R_PG_DMAC_GetTransferEscapeEndFlag_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_GetTransferEscapeEndFlag_C〈*チャネル番号*〉(bool* end)
〈*チャネル番号*〉: 0～3

概要 転送エスケープ終了フラグの取得

生成条件 割り込み出力要因に[リピート/ブロックサイズの転送終了]、[転送元アドレスの拡張リピートエリアオーバフロー]または[転送先アドレスの拡張リピートエリアオーバフロー]が選択された場合

引数

bool* end	転送エスケープ終了フラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_DMAC_C 〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_DMAC_GetStatus

詳細
- 転送エスケープ終了フラグを取得します。
- EXDMA割り込み要求フラグ(IRフラグ)は本関数内でクリアされます。DMA割り込み要求フラグを取得する必要がある場合は、本関数を呼び出す前にR_PG_DMAC_ClearInterruptFlag_C〈*チャネル番号*〉を呼び出してください。
- 転送エスケープ終了フラグは本関数内でクリアされません。転送エスケープ終了フラグをクリアする必要がある場合はR_PG_DMAC_ClearTransferEscapeEndFlag_C〈*チャネル番号*〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- リピート転送モードでDMAC0を設定
- 転送開始要因は割り込み
- 割り込み出力要因に[リピート/ブロックサイズの転送終了]を指定
- DMA割り込み優先レベルは0

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    bool end;

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //転送エスケープ終了フラグが1になるのを待つ
    do{
        R_PG_DMAC_GetTransferEscapeEndFlag_C0( & end );
    } while( end == false );

    //転送エスケープ終了フラグをクリアする
    R_PG_DMAC_ClearTransferEscapeEndFlag_C0();
}
```

5.8.15 R_PG_DMAC_ClearTransferEscapeEndFlag_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_ClearTransferEscapeEndFlag_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉: 0～3

概要 転送エスケープ終了フラグのクリア

生成条件 割り込み出力要因に[リピート/ブロックサイズの転送終了]、[転送元アドレスの拡張リピートエリアオーバフロー]または[転送先アドレスの拡張リピートエリアオーバフロー]が選択された場合

引数 なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_DMAC_C 〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- 転送エスケープ終了フラグをクリアします。
- 転送エスケープ終了フラグを取得するにはPG_DMAC_GetTransferEscapeEndFlag_C〈*チャネル番号*〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- リピート転送モードでDMAC0を設定
- 転送開始要因は割り込み
- 割り込み出力要因に[リピート/ブロックサイズの転送終了]を指定
- DMA割り込み優先レベルは0

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
	bool end;

	//DMAC0を設定する
	R_PG_DMAC_Set_C0();

	//DMAC0を転送開始トリガ入力待ち状態にする
	R_PG_DMAC_Activate_C0();

	//転送エスケープ終了フラグが1になるのを待つ
	do{
		R_PG_DMAC_GetTransferEscapeEndFlag_C0( & end );
	} while( end == false );

	//転送エスケープ終了フラグをクリアする
	R_PG_DMAC_ClearTransferEscapeEndFlag_C0();
}
```

5.8.16 R_PG_DMAC_SetSrcAddress_C〈チャネル番号〉

定義 bool R_PG_DMAC_SetSrcAddress_C〈チャネル番号〉(void * src_addr)
〈チャネル番号〉: 0～3

概要 転送元アドレスの設定

引数

void * src_addr	設定する転送元アドレス

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- 転送元アドレスを設定します

使用例 GUI上で以下の通り設定した場合
- DMAC0の転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetSrcAddress_C0( src_address );   //転送元アドレス
    R_PG_DMAC_SetDestAddress_C0( dest_address );   //転送先アドレス
    R_PG_DMAC_SetTransferCount_C0( tr_count );   //転送カウンタ値
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.17 R_PG_DMAC_SetDestAddress_C〈チャネル番号〉

定義

bool R_PG_DMAC_SetDestAddress_C〈チャネル番号〉(void * dest_addr)

〈チャネル番号〉: 0～3

概要

転送先アドレスの設定

引数

void * dest_addr	設定する転送先アドレス

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_DMAC_C〈チャネル番号〉.c

〈チャネル番号〉: 0～3

使用RPDL関数

R_DMAC_Control

詳細

- 転送先アドレスを設定します

使用例

GUI上で以下の通り設定した場合

- DMAC0の転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetSrcAddress_C0( src_address );   //転送元アドレス
    R_PG_DMAC_SetDestAddress_C0( dest_address );   //転送先アドレス
    R_PG_DMAC_SetTransferCount_C0( tr_count );   //転送カウンタ値
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.18 R_PG_DMAC_SetAddressOffset_C〈*チャネル番号*〉

定義 bool R_PG_DMAC_SetAddressOffset_C〈*チャネル番号*〉(int32_t offset)
〈*チャネル番号*〉: 0～3

概要 アドレスオフセット値の設定

生成条件 転送元アドレス更新モードまたは転送先アドレス更新オードにオフセット加算が選択された場合

引数

int32_t offset	設定するオフセット値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- アドレスオフセット加算値を設定します
- 有効な値は +FFFFFFh ～ -1000000h です。

使用例 GUI上で以下の通り設定した場合
- DMAC0の転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定
- 転送元アドレス更新モードまたは転送先アドレス更新オードにオフセット加算を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetSrcAddress_C0( src_address );   //転送元アドレス
    R_PG_DMAC_SetDestAddress_C0( dest_address );   //転送先アドレス
    R_PG_DMAC_SetTransferCount_C0( tr_count );   //転送カウンタ値
    R_PG_DMAC_SetAddressOffset_C0( offset );   //アドレスオフセット
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.19 R_PG_DMAC_SetExtendedRepeatSrc_C〈チャネル番号〉

定義 bool R_PG_DMAC_SetExtendedRepeatSrc_C〈チャネル番号〉(uint32_t area)
〈チャネル番号〉: 0～3

概要 転送元拡張リピートエリアの設定

生成条件 転送元が拡張リピートエリアに指定された場合

引数

uint32_t area	設定する転送元拡張リピートエリア値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- 転送元拡張リピートエリアの範囲を設定します
- 有効な値は2^{1}～2^{27}の2のべき乗です

使用例 GUI上で以下の通り設定した場合
- DMAC0の転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定
- 転送元および転送先を拡張リピートエリアに指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetSrcAddress_C0(src_address);   //転送元アドレス
    R_PG_DMAC_SetDestAddress_C0(dest_address);   //転送先アドレス
    R_PG_DMAC_SetTransferCount_C0(tr_count);   //転送カウンタ値
    R_PG_DMAC_SetExtendedRepeatSrc_C0(src_repeat); //転送元拡張リピートエリアサイズ
    R_PG_DMAC_SetExtendedRepeatDest_C0(dest_repeat); //転送先拡張リピートエリアサイズ
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.20 R_PG_DMAC_SetExtendedRepeatDest_C〈チャネル番号〉

定義 bool R_PG_DMAC_SetExtendedRepeatDest_C〈チャネル番号〉(uint32_t area)
〈チャネル番号〉: 0～3

概要 転送先拡張リピートエリアの設定

生成条件 転送先が拡張リピートエリアに指定された場合

引数

uint32_t area	設定する転送先拡張リピートエリア値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DMAC_C〈チャネル番号〉.c
〈チャネル番号〉: 0～3

使用RPDL関数 R_DMAC_Control

詳細
- 転送先拡張リピートエリアの範囲を設定します
- 有効な値は2^{1}～2^{27}の2のべき乗です

使用例 GUI上で以下の通り設定した場合
- DMAC0の転送開始要因にIRQ0を指定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定
- 転送元および転送先を拡張リピートエリアに指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();
    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
    // DMAC0の設定を変更
    R_PG_DMAC_SetSrcAddress_C0(src_address);   //転送元アドレス
    R_PG_DMAC_SetDestAddress_C0(dest_address);   //転送先アドレス
    R_PG_DMAC_SetTransferCount_C0(tr_count);   //転送カウンタ値
    R_PG_DMAC_SetExtendedRepeatSrc_C0( src_repeat );//転送元拡張リピートエリアサイズ
    R_PG_DMAC_SetExtendedRepeatDest_C0(dest_repeat);//転送先拡張リピートエリアサイズ
    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();
}
```

5.8.21 R_PG_DMAC_StopModule_C〈*チャネル番号*〉

定義　bool R_PG_DMAC_StopModule_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉: 0～3

概要　DMACチャネルの停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_DMAC_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～3

使用RPDL関数　R_DMAC_Destroy

詳細
- DMACのチャネルを停止します。
- DMACの全チャネルとDTCが停止している場合、DMACおよびDTCはモジュールストップ状態に移行します。
- 他の周辺機能が転送開始要因として使用されている場合は、本関数を呼ぶ前に転送開始要因を無効にしてください。

使用例　GUI上で以下の通り設定した場合
- DMAC0の転送開始要因をソフトウェアトリガに設定
- DMA0割り込み通知関数名に Dmac0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //DMAC0の転送を開始する
    R_PG_DMAC_StartTransfer_C0();
}

//DMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0を停止
    R_PG_DMAC_StopModule_C0();
}
```

5.9 データトランスファコントローラ（DTCa）

5.9.1 R_PG_DTC_Set

定義 bool R_PG_DTC_Set (void)

概要 DTCの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DTC.c

使用RPDL関数 R_DTC_Set

詳細

- モジュールストップ状態を解除し、転送情報リードスキップ、アドレスモードおよびDTCベクタテーブルのベースアドレスを設定します。
- DTCの他の関数を使用する前に本関数を呼び出してください。

使用例

GUI上で以下の通り設定した場合

- DTCベクタテーブルのアドレスを2000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00002000
uint32_t dtc_vector_table [256];

//DTCの初期設定
void func(void)
{
    //データトランスファコントローラの基本設定
    R_PG_DTC_Set();

    //転送開始要因をIRQ0に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ0();

    //DTCを転送開始トリガ入力待ち状態にする
    R_PG_DTC_Activate();

    //IRQ0の設定
    R_PG_ExtInterrupt_Set_IRQ0();
}
```

5.9.2 R_PG_DTC_Set_〈*転送開始要因*〉

定義 bool R_PG_DTC_Set_〈*転送開始要因*〉(void)

〈*転送開始要因*〉 :

SWINT	ソフトウェア割り込み
CMI0～3	CMT0～3 コンペアマッチ割り込み
SPRI0	RSPI0 受信バッファフル割り込み
SPTI0	RSPI0 送信バッファエンプティ割り込み
CMPB0/1	CMPB コンパレータB0/1割り込み
IRQ0～7	外部端子割り込み
S12ADI0	S12ADb A/Dスキャン終了割り込み
GBADI	S12ADb グループBスキャン終了割り込み
ELSR18I	ELC 割り込み
ELSR19I	ELC 割り込み
TGIA0～D0	MTU0 インプットキャプチャ/コンペアマッチA～D割り込み
TGIA1/B1	MTU1 インプットキャプチャ/コンペアマッチA/B割り込み
TGIA2/B2	MTU2 インプットキャプチャ/コンペアマッチA/B割り込み
TGIA3～D3	MTU3 インプットキャプチャ/コンペアマッチA～D割り込み
TGIA4～D4	MTU4 インプットキャプチャ/コンペアマッチA～D割り込み
TCIV4	MTU4 オーバフロー/アンダフロー割り込み
TGIU5～W5	MTU5 インプットキャプチャ/コンペアマッチU～W割り込み
TGI0A～D	TPU0 インプットキャプチャ/コンペアマッチA～D割り込み
TGI1A,B	TPU1 インプットキャプチャ/コンペアマッチA/B割り込み
TGI2A,B	TPU2 インプットキャプチャ/コンペアマッチA/B割り込み
TGI3A～D	TPU3 インプットキャプチャ/コンペアマッチA～D割り込み
TGI4A,B	TPU4 インプットキャプチャ/コンペアマッチA/B割り込み
TGI5A,B	TPU5 インプットキャプチャ/コンペアマッチA/B割り込み
CMIA0/B0	TMR0 コンペアマッチA/B割り込み
CMIA1/B1	TMR1 コンペアマッチA/B割り込み
CMIA2/B2	TMR2 コンペアマッチA/B割り込み
CMIA3/B3	TMR3 コンペアマッチA/B割り込み
DMAC0I～3I	DMACA0～3 割り込み
RXI0～12	SCI0～12 受信データフル割り込み
TXI0～12	SCI0～12 送信データエンプティ割り込み
ICRXI0	RIIC0 受信データフル割り込み
ICTXI0	RIIC0 送信データエンプティ割り込み

概要 DTC転送情報の設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DTC.c

使用RPDL関数 R_DTC_Create

詳細

- 起動要因によりトリガされる転送情報を指定されたアドレスに保存し、転送情報のアドレスをDTCベクタテーブルに設定します。
- 起動要因によりトリガされるチェイン転送の情報も保存されます。
- 指定されたアドレスに既に転送情報が保存されている場合は上書きされます。
- 本関数では起動要因として使用する割り込みの設定を行いません。起動要因として使用する割り込みは各周辺機能の関数で設定してください。
  起動要因として使用する割り込みは、割り込み要求先をDTCに指定してください。
- データ転送に関わる周辺モジュールの設定をする前に、本関数を呼び出してください。

使用例

GUI上で以下の通り設定した場合

- DTCベクタテーブルのアドレスを2000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定
- 転送開始要因をIRQ1に指定したDTC転送を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00002000
uint32_t dtc_vector_table [256];

//DTCの初期設定
void func(void)
{
    //データトランスファコントローラの基本設定
    R_PG_DTC_Set();

    //転送開始要因をIRQ0に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ0();

    //転送開始要因をIRQ1に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ1();

    //DTCを転送開始トリガ入力待ち状態にする
    R_PG_DTC_Activate();

    //IRQ0,IRQ1の設定
    R_PG_ExtInterrupt_Set_IRQ0();
    R_PG_ExtInterrupt_Set_IRQ1();
}
```

5.9.3 R_PG_DTC_Activate

定義 bool R_PG_DTC_Activate (void)

概要 DTCを転送開始トリガの入力待ち状態に設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DTC.c

使用RPDL関数 R_DTC_Control

詳細
- DTCを転送開始トリガの入力待ち状態に設定します。
- あらかじめR_PG_DTC_SetによりDTCを設定し、R_PG_DTC_Set<*転送開始要因*>により転送情報を保存してください。

使用例 GUI上で以下の通り設定した場合
- DTCベクタテーブルのアドレスを2000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定
- 割り込みの発生条件に[指定されたデータ転送終了時、CPU割込みが発生]を指定
- チェイン転送無効
- IRQ0の割り込み通知関数名に Irq0IntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00002000
uint32_t dtc_vector_table [256];

//DTCの初期設定
void func(void)
{
    //データトランスファコントローラの基本設定
    R_PG_DTC_Set();

    //転送開始要因をIRQ0に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ0();

    //DTCを転送開始トリガ入力待ち状態にする
    R_PG_DTC_Activate();
}

//IRQ0の割り込み通知関数（指定した回数のDTC転送終了時に割り込み発生）
void Irq0IntFunc(void)
{
    //IRQ0の停止
    //(指定した回数の転送終了後もトリガ入力で転送が継続し、
    //転送カウンタはインクリメントします。転送を終了するには
    //起動要因の割り込みを無効にしてください。)
    R_PG_ExtInterrupt_Disable_IRQ0();
}
```

5.9.4 R_PG_DTC_SuspendTransfer

定義 bool R_PG_DTC_SuspendTransfer (void)

概要 DTC転送の停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_DTC.c

使用RPDL関数 R_DTC_Control

詳細
- DTC転送を停止します。
- 転送動作中に停止した場合、受付済みの転送要求は処理が終わるまで動作します。
- DTC転送を有効にするにはR_DTC_Activateを呼び出してください。

使用例

GUI上で以下の通り設定した場合
- DTCベクタテーブルのアドレスを2000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00002000
uint32_t dtc_vector_table [256];

//DTCの初期設定
void func1(void)
{
    //データトランスファコントローラの基本設定
    R_PG_DTC_Set();

    //転送開始要因をIRQ0に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ0();

    //DTCを転送開始トリガ入力待ち状態にする
    R_PG_DTC_Activate();
}

//DTC転送の中断
void func2(void)
{
    R_PG_DTC_SuspendTransfer();
}

//DTC転送の再開
void func3(void)
{
    R_PG_DTC_Activate();
}
```

5.9.5 R_PG_DTC_GetTransmitStatus

定義 bool R_PG_DTC_GetTransmitStatus (uint8_t * vector, bool * active)

概要 DTC転送状態の取得

引数

uint8_t * vector	転送動作中の場合、現在の転送の起動要因のベクタ番号 (*activeが1の場合に有効化値が格納されます)
bool * active	現在の転送状態 (0:転送動作なし 1:転送動作中)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_DTC.c

使用RPDL関数 R_DTC_GetStatus

詳細

- DTCアクティブフラグとDTCアクティブベクタ番号を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。
  取得しない項目には0を指定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t vector;
bool active;

void func(void)
{
    //DTC転送状態の取得
    R_PG_DTC_GetTransmitStatus ( &vector, &active);

    if(active){
        switch( vector ){
            case 64:
                //ベクタ番号64の割込みによる転送中の処理
                break;
            case 65:
                //ベクタ番号65の割込みによる転送中の処理
                break;
            default:
        }
    }
}
```

5.9.6 R_PG_DTC_StopModule

定義 bool R_PG_DTC_StopModule (void)

概要 DTCの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_DTC.c

使用RPDL関数 R_DTC_Destroy

詳細
- DTCを停止し、モジュールストップ状態に移行します。
- あらかじめ各周辺機能の関数によりDTCのトリガ要因として使用した割り込みを無効にしてください。
- 本関数はDMACもモジュールストップ状態に移行します。

使用例 GUI上で以下の通り設定した場合
- DTCベクタテーブルのアドレスを2000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定
- 転送開始要因をIRQ1に指定したDTC転送を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00002000
uint32_t dtc_vector_table [256];

//DTCの初期設定
void func1(void)
{
    //データトランスファコントローラの基本設定
    R_PG_DTC_Set();

    //転送開始要因をIRQ0に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ0();

    //転送開始要因をIRQ1に指定したDTC転送の設定
    R_PG_DTC_Set_IRQ1();

    //DTCを転送開始トリガ入力待ち状態にする
    R_PG_DTC_Activate();

    //IRQ0,IRQ1の設定
    R_PG_ExtInterrupt_Set_IRQ0();
    R_PG_ExtInterrupt_Set_IRQ1();
}

//DTCの停止
void func2(void)
{
    //IRQ0,IRQ1の停止
    R_PG_ExtInterrupt_Disable_IRQ0();
    R_PG_ExtInterrupt_Disable_IRQ1();
    //DTCの停止
    R_PG_DTC_StopModule();
}
```

5.10 イベントリンクコントローラ（ELC）

5.10.1 R_PG_ELC_Set

定義　bool R_PG_ELC_Set (void)

概要　ELCの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ELC.c

使用RPDL関数　R_ELC_Create

詳細

- モジュールストップ状態を解除します。
- GUI上で割り込み1または割り込み2のイベントリンクを設定し、割り込み通知関数名を指定した場合、CPUへの割り込み要求が発生すると指定した名前の関数が呼び出されます。割り込み通知関数は次の定義で作成してください。

  void <割り込み通知関数名> (void)

  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- 他のELCの関数を使用する前に本関数を呼び出してください。

使用例　GUI上で以下の通り設定した場合

- 割り込み1のイベントリンクを設定
- 割り込み通知関数名にElc1IntFuncを指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //イベントリンクコントローラ(ELC)を設定
    R_PG_ELC_Set();

    //イベントリンクの設定
    R_PG_ELC_SetLink_Interrupt1();

    //全イベントリンクの有効化
    R_PG_ELC_AllEventLinkEnable();
}

//割り込み通知関数
void Elc1IntFunc(void)
{
    //割り込み発生時処理

    //イベントリンク動作の停止
    R_PG_ELC_DisableLink_Interrupt1();
}
```

5.10.2 R_PG_ELC_SetLink_〈*周辺機能*〉

定義 bool R_PG_ELC_SetLink_〈*周辺機能*〉(void)

〈*周辺機能*〉

MTU1～4	マルチファンクションタイマパルスユニット2(MTU2a) チャネル1～4
CMT1	コンペアマッチタイマ(CMT) チャネル1
TMR0/2	8ビットタイマ(TMR) チャネル0/2
ADC12	12ビットA/Dコンバータ(S12ADb)
DA0	D/Aコンバータ チャネル0
Interrupt1/2	割り込み1/2
Output_Group1/2	出力ポートグループ1/2
Input_Group1/2	入力ポートグループ1/2
SinglePort0～3	シングルポート0～3
ClockSource	クロックソース切り替え
POE2	ポートアウトプットイネーブル2(POE2a)

概要 イベントリンクの設定

生成条件 イベント受信モジュールが設定された場合に、各モジュールに対応する関数を生成

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Control

詳細
- イベントとイベントを受けるモジュールとのリンクを設定します。
- イベントを受けた時のモジュールの動作を設定します。
- イベントのリンクを有効にするにはR_PG_ELC_AllEventLinkEnableを呼び出してください。
- CMT1の[イベント入力時動作]にイベントカウンタを選択した場合は、本関数を呼び出した後、イベントのリンクを有効にする前にR_PG_Timer_StartCount_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉を呼び出してください。
- TMR0/TMR2の[イベント入力時動作]にカウントスタート/イベントカウンタを選択した場合は、本関数を呼び出した後、イベントのリンクを有効にする前にR_PG_Timer_Start_TMR_U〈*ユニット番号*〉C〈*チャネル番号*〉を呼び出してください。

使用例 R_PG_ELC_Setの使用例を参照してください。

5.10.3 R_PG_ELC_DisableLink_〈*周辺機能*〉

定義 bool R_PG_ELC_DisableLink_〈*周辺機能*〉(void)

〈*周辺機能*〉

MTU1～4	マルチファンクションタイマパルスユニット2(MTU2a) チャネル1～4
CMT1	コンペアマッチタイマ(CMT) チャネル1
TMR0/2	8ビットタイマ(TMR) チャネル0/2
ADC12	12ビットA/Dコンバータ(S12ADb)
DA0	D/Aコンバータ チャネル0
Interrupt1/2	割り込み1/2
Output_Group1/2	出力ポートグループ1/2
Input_Group1/2	入力ポートグループ1/2
SinglePort0～3	シングルポート0～3
ClockSource	クロックソース切り替え
POE2	ポートアウトプットイネーブル2(POE2a)

概要 イベントリンク動作の停止

生成条件 イベント受信モジュールが設定された場合に、各モジュールに対応する関数を生成

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Control

詳細
- 設定済みのリンクの動作を停止します。

使用例 R_PG_ELC_Setの使用例を参照してください。

5.10.4 R_PG_ELC_Set_PortGroup〈*ポートグループ番号*〉

定義　bool R_PG_ELC_Set_PortGroup〈*ポートグループ番号*〉(void)
〈*ポートグループ番号*〉： 1, 2

概要　ポートグループの設定

生成条件　出力ポートグループまたは入力ポートグループの[ポートグループに含める]のいずれかのビットにチェックあり

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ELC.c

使用RPDL関数　R_ELC_Control

詳細
- ポートグループ1(ポートB)またはポートグループ2(ポートE)について、GUI上で選択されたビットでポートグループを構成します。
- ポートグループのイベント発生条件を設定します。

使用例　GUI上で以下の通り設定した場合
- I/Oポート設定にて、PB0～PB3の入出力方向を入力に設定
- 入力ポートグループ1に対するイベント信号に以下を選択
  [ソフトウェアイベント信号]
- 入力ポートグループ1のイベント入力時動作に以下を選択
  [外部端子の信号値をPDBFnレジスタに転送]
- 入力ポートグループ1は以下のビットをポートグループに含める
  [PB0～PB3]

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t pdbf1_val;   //ポートバッファレジスタ1の値の格納先

void func1(void)
{
    R_PG_IO_PORT_Set_PB0();   //I/Oポート(PB0)の設定
    R_PG_IO_PORT_Set_PB1();   //I/Oポート(PB1)の設定
    R_PG_IO_PORT_Set_PB2();   //I/Oポート(PB2)の設定
    R_PG_IO_PORT_Set_PB3();   //I/Oポート(PB3)の設定

    R_PG_ELC_Set();   //イベントリンクコントローラ(ELC)を設定
    R_PG_ELC_SetLink_Input_Group1();   //イベントリンクの設定
    R_PG_ELC_Set_PortGroup1();   //ポートグループの設定
    R_PG_ELC_AllEventLinkEnable();   //全イベントリンクの有効化
    R_PG_ELC_Generate_SoftwareEvent();   //ソフトウェアイベントの生成
}

void func2(void)
{
    R_PG_ELC_GetPortBufferValue_Group1(&pdbf1_val); //ポートバッファレジスタ値取得
    R_PG_ELC_StopModule();   //ELCの停止
}
```

5.10.5 R_PG_ELC_Set_SinglePort〈*シングルポート番号*〉

定義　bool R_PG_ELC_Set_SinglePort〈*シングルポート番号*〉(void)

〈*シングルポート番号*〉 : 0～3

概要　シングルポートの設定

生成条件　ポート設定のシングルポートの設定が以下全ての条件を満たす場合

1. ポート設定にていずれかのポートが選択されている
2. イベント発生条件が選択可能

※ポート端子が出力用の場合には本関数は生成されません。

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ELC.c

使用RPDL関数　R_ELC_Control

詳細

- GUI上で選択されたビットをシングルポートに設定します。
- シングルポートのイベント発生条件を設定します。
- 本関数はポート端子が入力用の場合にのみ使用します。

使用例　GUI上で以下の通り設定した場合

- I/Oポート設定にて、PB0の入出力方向を入力に設定
- 割り込み1に対するイベント信号に以下を選択
  [シングル入力ポート0・入力エッジ検出信号]
- 割り込み1のイベント入力時動作に[CPUへ要求]を選択
- シングルポート0のポート設定に[PB0]を選択
- シングルポート0のイベント発生条件に[立ち下がりエッジ検出]を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
uint8_t elc1int_count=0;   //割り込み発生回数

void func(void)
{
      R_PG_IO_PORT_Set_PB0();   //I/Oポート(端子毎)の設定
      R_PG_ELC_Set();   //イベントリンクコントローラ(ELC)を設定
      R_PG_ELC_SetLink_Interrupt1();   //イベントリンクの設定
      R_PG_ELC_Set_SinglePort0();   //シングルポートの設定
      R_PG_ELC_AllEventLinkEnable();   //全イベントリンクの有効化
}

//割り込み通知関数
void Elc1IntFunc(void)
{
      elc1int_count++;
}
```

5.10.6 R_PG_ELC_AllEventLinkEnable

定義 bool R_PG_ELC_AllEventLinkEnable (void)

概要 全イベントリンクの有効化

引数 なし

戻り値

true	有効化が正しく行われた場合
false	有効化に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Control

詳細
- 設定された全てのイベントのリンクを有効にします。

使用例 R_PG_ELC_Setの使用例を参照してください。

5.10.7 R_PG_ELC_AllEventLinkDisable

定義 bool R_PG_ELC_AllEventLinkDisable (void)

概要 全イベントリンクの無効化

引数 なし

戻り値

true	無効化が正しく行われた場合
false	無効化に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Control

詳細
- 全てのイベントのリンクを無効にします。

使用例 R_PG_ELC_Set_SinglePort〈*シングルポート番号*〉の使用例を参照してください。

5.10.8 R_PG_ELC_Generate_SoftwareEvent

定義 bool R_PG_ELC_Generate_SoftwareEvent (void)

概要 ソフトウェアイベントの生成

生成条件 イベント信号にソフトウェアイベント信号が選択されている場合

引数 なし

戻り値

true	生成が正しく行われた場合
false	生成に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Control

詳細
- ソフトウェアイベントを生成します。

使用例 R_PG_ELC_Setの使用例を参照してください。

5.10.9 R_PG_ELC_GetPortBufferValue_Group〈*ポートグループ番号*〉

定義 bool R_PG_ELC_GetPortBufferValue_Group〈*ポートグループ番号*〉(uint8_t * reg_val)
〈*ポートグループ番号*〉： 1, 2

概要 ポートバッファレジスタ値の取得

生成条件 ポート設定にて、[出力ポートグループn または 入力ポートグループn]の[ポートグループに含める]に1つ以上チェックがある場合
n ： 1, 2

引数

uint8_t * reg_val	ポートバッファレジスタ値の格納先

戻り値

true	取得が正しく行われた場合
false	取得に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Read

詳細
- ポートバッファレジスタ値を取得します。
  〈*ポートグループ番号*〉： 1
  PDBF1(ポートバッファレジスタ1)の値を取得
  〈*ポートグループ番号*〉： 2
  PDBF2(ポートバッファレジスタ2)の値を取得

使用例 R_PG_ELC_Set_PortGroup〈*ポートグループ番号*〉の使用例を参照してください。

5.10.10 R_PG_ELC_SetPortBufferValue_Group〈*ポートグループ番号*〉

定義　bool R_PG_ELC_SetPortBufferValue_Group〈*ポートグループ番号*〉(uint8_t reg_val)
〈*ポートグループ番号*〉： 1, 2

概要　ポートバッファレジスタ値の設定

生成条件　ポート設定にて、[出力ポートグループn または 入力ポートグループn]の[ポートグループに含める]に1つ以上チェックがある場合
n ： 1, 2

引数

uint8_t reg_val	ポートバッファレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ELC.c

使用RPDL関数　R_ELC_Write

詳細
- ポートバッファレジスタ値を設定します。
  〈*ポートグループ番号*〉： 1
  PDBF1(ポートバッファレジスタ1)の値を設定
  〈*ポートグループ番号*〉： 2
  PDBF2(ポートバッファレジスタ2)の値を設定

使用例　GUI上で以下の通り設定した場合

- I/Oポート設定にて、PB4～PB7の入出力方向を出力に設定
- 出力ポートグループ1に対するイベント信号に以下を選択
  [ソフトウェアイベント信号]
- 出力ポートグループ1のイベント入力時動作に以下を選択
  [バッファ値を出力]
- 出力ポートグループ1は以下のビットをポートグループに含める
  [PB4～PB7]

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    uint8_t pdbf1_val = 0xf0;   //ポートバッファレジスタ1設定値

    R_PG_IO_PORT_Set_PB4();   //I/Oポート(PB4)の設定
    R_PG_IO_PORT_Set_PB5();   //I/Oポート(PB5)の設定
    R_PG_IO_PORT_Set_PB6();   //I/Oポート(PB6)の設定
    R_PG_IO_PORT_Set_PB7();   //I/Oポート(PB7)の設定

    R_PG_ELC_Set();   //イベントリンクコントローラ(ELC)を設定
    R_PG_ELC_SetLink_Output_Group1();   //イベントリンクの設定
    R_PG_ELC_Set_PortGroup1();   //ポートグループの設定
    R_PG_ELC_SetPortBufferValue_Group1(pdbf1_val); //ポートバッファレジスタ値設定
    R_PG_ELC_AllEventLinkEnable();   //全イベントリンクの有効化
    R_PG_ELC_Generate_SoftwareEvent();   //ソフトウェアイベントの生成
}
```

5.10.11 R_PG_ELC_StopModule

定義 bool R_PG_ELC_StopModule (void)

概要 ELCの停止

引数 なし

戻り値

true	停止が正しく行われた場合
false	停止に失敗した場合

出力先ファイル R_PG_ELC.c

使用RPDL関数 R_ELC_Destroy

詳細
- 全てのイベントリンクを無効にし、モジュールストップ状態にします。

使用例 R_PG_ELC_Set_PortGroup<*ポートグループ番号*>の使用例を参照してください。

5.11 I/Oポート

5.11.1 R_PG_IO_PORT_Set_P〈*ポート番号*〉

定義　bool R_PG_IO_PORT_Set_P〈*ポート番号*〉(void)
〈*ポート番号*〉 ： 0～5、A～E、H、J

概要　I/Oポートの設定

生成条件　GUI上で、ポート内で1つ以上の端子について、[I/Oポートとして使用]にチェックがある場合。
ただし、ポート3にてP35にのみチェックがある場合、R_PG_IO_PORT_Set_P3は生成されません。

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_IO_PORT_P〈*ポート番号*〉.c
〈*ポート番号*〉 ： 0～5、A～E、H、J

使用RPDL関数　R_IO_PORT_Set

詳細
- GUI上で[I/Oポートとして使用]にチェックされた端子の入出力方向、入力プルアップ抵抗の有効/無効、出力形態、駆動能力の設定を行います。
- ポート内の[I/Oポートとして使用]がチェックされた全端子を一括して設定します。
- 出力ポートとして設定(高駆動出力設定)している場合、ディープソフトウェアスタンバイになると全ビットが通常出力になります。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //存在しない端子の処理
    R_PG_IO_PORT_SetPortNotAvailable();

    //P0を設定する
    R_PG_IO_PORT_Set_P0();
}
```

5.11.2 R_PG_IO_PORT_Set_P〈*ポート番号*〉〈*端子番号*〉

定義

bool R_PG_IO_PORT_Set_P〈*ポート番号*〉〈*端子番号*〉(void)

〈*ポート番号*〉 : 0～5、A～E、H、J

〈*端子番号*〉 : 0～7

概要

I/Oポート(1端子)の設定

生成条件

GUI上で、「I/Oポートとして使用」にチェックがある場合。

ただしR_PG_IO_PORT_Set_P35は生成されません。

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉 : 0～5、A～E、H、J

使用RPDL関数

R_IO_PORT_Set

詳細

- GUI上で[I/Oポートとして使用]にチェックされた端子の入出力方向、入力プルアップ抵抗の有効/無効、出力形態、駆動能力の設定を行います。
- 1端子のみ設定します。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //P03を設定する
    R_PG_IO_PORT_Set_P03();

    //P05を設定する
    R_PG_IO_PORT_Set_P05();
}
```

5.11.3 R_PG_IO_PORT_Read_P〈*ポート番号*〉

定義

bool R_PG_IO_PORT_Read_P〈*ポート番号*〉(uint8_t * data)

〈*ポート番号*〉 : 0～5、A～E、H、J

概要

ポート入力レジスタの読み出し

生成条件

GUI上で、ポート内で1つ以上の端子について、[I/Oポートとして使用]にチェックがある場合。

引数

uint8_t * data	読み出した端子状態の格納先

戻り値

true	読み出しに成功した場合
false	読み出しに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉 : 0～5、A～E、H、J

使用RPDL関数

R_IO_PORT_Read

詳細

- ポート入力レジスタを読み出し、端子の状態を取得します。(ポート単位)

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data;

void func(void)
{
    //P0端子状態を取得する
    R_PG_IO_PORT_Read_P0( &data );
}
```

5.11.4 R_PG_IO_PORT_Read_P〈*ポート番号*〉〈*端子番号*〉

定義

bool R_PG_IO_PORT_Read_P〈*ポート番号*〉〈*端子番号*〉(uint8_t * data)

〈*ポート番号*〉 : 0～5、A～E、H、J

〈*端子番号*〉 : 0～7

概要

ポート入力レジスタからのビット読み出し

生成条件

GUI上で、ポート内で1つ以上の端子について、[I/Oポートとして使用]にチェックがある場合に、ポート内に存在する全端子に対する関数が生成されます。

引数

uint8_t * data	読み出した端子状態の格納先

戻り値

true	読み出しに成功した場合
false	読み出しに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉 : 0～5、A～E、H、J

使用RPDL関数

R_IO_PORT_Read

詳細

- ポート入力レジスタを読み出し、1端子の状態を取得します。
- 値は*dataの下位1ビットに格納されます。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_p03, data_p05;

void func(void)
{
    //P03端子状態を取得する
    R_PG_IO_PORT_Read_P03( & data_p03);

    //P05端子状態を取得する
    R_PG_IO_PORT_Read_P05( & data_p05);
}
```

5.11.5 R_PG_IO_PORT_Write_P〈*ポート番号*〉

定義

bool R_PG_IO_PORT_Write_P〈*ポート番号*〉(uint8_t data)

〈*ポート番号*〉： 0～5、A～E、H、J

概要

ポート出力データレジスタへの書き込み

生成条件

GUI上で、ポート内で1つ以上の端子について、[I/Oポートとして使用]にチェックがある場合。

ただし、ポート3にてP35にのみチェックがある場合、R_PG_IO_PORT_Write_P3は生成されません。

引数

uint8_t data	書き込む値

戻り値

true	書き込みに成功した場合
false	書き込みに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉： 0～5、A～E、H、J

使用RPDL関数

R_IO_PORT_Write

詳細

- ポート出力データレジスタに値を書き込みます。レジスタに書き込んだ値が出力ポートから出力されます。
- R_PG_IO_PORT_Write_P3を呼び出す場合、引数のb5には”0”を指定してください。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //P0を設定する
    R_PG_IO_PORT_Set_P0();

    //P0から0x28を出力する
    R_PG_IO_PORT_Write_P0( 0x28 );
}
```

5.11.6 R_PG_IO_PORT_Write_P〈*ポート番号*〉〈*端子番号*〉

定義

bool R_PG_IO_PORT_Write_P〈*ポート番号*〉〈*端子番号*〉(uint8_t data)

〈*ポート番号*〉 : 0～5、A～E、H、J

〈*端子番号*〉 : 0～7

概要

ポート出力データレジスタへのビット書き込み

生成条件

GUI上で、「I/Oポートとして使用」にチェックがある場合。

ただしR_PG_IO_PORT_Write_P35は生成されません。

引数

uint8_t data	書き込む値

戻り値

true	書き込みに成功した場合
false	書き込みに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉 : 0～5、A～E、H、J

使用RPDL関数

R_IO_PORT_Write

詳細

- ポート出力データレジスタに値を書き込みます。レジスタに書き込んだ値が出力ポートから出力されます。値はdataの下位1ビットに格納してください。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //P03を設定する
    R_PG_IO_PORT_Set_P03();

    //P05を設定する
    R_PG_IO_PORT_Set_P05();

    //P03からLを出力する
    R_PG_IO_PORT_Write_P03( 0x00 );

    //P05からHを出力する
    R_PG_IO_PORT_Write_P05( 0x01 );
}
```

5.11.7 R_PG_IO_PORT_SetPortNotAvailable

定義	bool R_PG_IO_PORT_SetPortNotAvailable (void)
概要	存在しないポートの設定
引数	なし

戻り値

true	設定が正しく行われた場合

出力先ファイル	R_PG_IO_PORT.c
使用RPDL関数	R_IO_PORT_NotAvailable
詳細	• 少ピンパッケージに存在しない全てのポートをCMOSのLowレベル出力に設定します。 • 100ピン以外のパッケージを使用する場合は、最初に必ず本関数を呼び出してください。
使用例	R_PG_IO_PORT_Set_P*<ポート番号>*の使用例を参照してください。

5.12 マルチファンクションタイマパルスユニット 2 (MTU2a)

5.12.1 R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉

定義　bool R_PG_Timer_Set_MTU_U〈*ユニット番号*〉_〈*チャネル*〉(void)
〈*ユニット番号*〉: 0
〈*チャネル*〉: C0～C5
C3_C4 (相補PWM、リセット同期PWMモード時)

概要　MTUの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～5

使用RPDL関数　R_MTU2_Set, R_MTU2_Create

詳細

- MTUのモジュールストップ状態を解除して初期設定します。
- 本関数内でMTUの割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。　void 〈*割り込み通知関数名*〉(void)
  割り込み通知関数については「通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み優先レベルを0に設定した場合、CPU割り込みは発生しません。割り込み要求フラグは R_PG_Timer_GetRequestFlag_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉により取得することができます。
- 外部入力カウントクロック、外部リセット信号、インプットキャプチャ、パルス出力を使用する場合、本関数内で使用する端子を設定します。
- カウント動作を開始するには本関数を呼び出した後に R_PG_Timer_StartCount_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉(_〈*相*〉) または R_PG_Timer_SynchronouslyStartCount_MTU_U〈*ユニット番号*〉 を呼び出してください。
- 相補PWMモードおよびリセット同期PWMモードでは、ペアで使用する2チャネルを設定します。R_PG_Timer_Set_MTU_U0_C3_C4によりチャネル3,4が設定されます。
- 相補PWMモードおよびリセット同期PWMモードでは、初期状態でPWM出力が無効です。端子出力を有効にするには、カウントを開始する前に R_PG_Timer_ControlOutputPin_MTU_U〈*ユニット番号*〉_〈*チャネル*〉 を呼び出してください。

使用例 1

GUI上で以下の通り設定した場合

- MTUチャネル1を通常モードで設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();      // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();      // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    //コンペアマッチA割り込み発生時処理
}
```

使用例 2

GUI上で以下の通り設定した場合

- MTUチャネル3,4を相補PWMモードで設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //MTU3,4を相補PWMモードで設定
    R_PG_Timer_Set_MTU_U0_C3_C4();

    //PWM出力端子1の正相、逆相出力を有効化
    R_PG_Timer_ControlOutputPin_MTU_U0_C3_C4(
        1, //p1： 有効
        1, //n1： 有効
        0, //p2： 無効
        0, //n2： 無効
        0, //p3： 無効
        0 //n3： 無効
    );

    //MTU3,4のカウント動作開始
    R_PG_Timer_SynchronouslyStartCount_MTU_U0(
        0, //ch0
        0, //ch1
        0, //ch2
        1, //ch3
        1 //ch4
    );
}
```

5.12.2 R_PG_Timer_StartCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉)

定義

bool R_PG_Timer_StartCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

bool R_PG_Timer_StartCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 5
〈相〉: U, V, W

概要

MTUのカウント動作開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

使用RPDL関数

R_MTU2_ControlChannel

詳細

- MTUのカウント動作を開始します。
- あらかじめR_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉 によりMTUを初期設定してください。
- 相補PWMモードおよびリセット同期PWMモードでは、R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉 によりペアで使用する2チャネルのカウント動作を同時に開始してください。
- R_PG_Timer_StartCount_MTU_U0_C5 はU,V,W相のカウンタを同時に開始させます。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   //MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   //カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_HaltCount_MTU_U0_C1();   //カウント動作停止

    //コンペアマッチA割り込み発生時処理

    R_PG_Timer_StartCount_MTU_U0_C1();   //カウント動作再開
}
```

5.12.3 R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉

定義

bool R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉
(bool ch0, bool ch1, bool ch2, bool ch3, bool ch4)
〈ユニット番号〉: 0

概要

MTUの複数チャネルのカウント動作を同時に開始

引数

bool ch0	チャネル0のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch1	チャネル1のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch2	チャネル2のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch3	チャネル3のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch4	チャネル4のカウント動作 (0:カウント開始しない 1:カウント開始)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_MTU2_ControlUnit

詳細

- MTUの複数チャネルのカウント動作を同時に開始します。
- あらかじめR_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉 によりMTUを初期設定してください。
- 相補PWMモードおよびリセット同期PWMモードでは、本関数によりペアで使用する2チャネルのカウント動作を同時に開始してください。

使用例

R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉 の使用例2を参照してください。

5.12.4 R_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉)

定義

bool R_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5
bool R_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 5
〈相〉: U, V, W

概要

MTUのカウント動作を一時停止

引数

なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

使用RPDL関数

R_MTU2_ControlChannel

詳細

- MTUのカウント動作を一時停止します。
- カウント動作を再開するには
  R_PG_Timer_StartCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉) または
  R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉 を呼び出してください。
- R_PG_Timer_HaltCount_MTU_U0_C5 はU,V,W相のカウンタを同時に停止させます。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル1を設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();      // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();      // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_HaltCount_MTU_U0_C1();      //カウント動作停止

    //コンペアマッチA割り込み発生時処理

    R_PG_Timer_StartCount_MTU_U0_C1();      //カウント動作再開
}
```

5.12.5 R_PG_Timer_GetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * counter_val)
〈ユニット番号〉: 0
〈チャネル番号〉: 0～4

bool R_PG_Timer_GetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * counter_u_val, uint16_t * counter_v_val, uint16_t * counter_w_val)
〈ユニット番号〉: 0
〈チャネル番号〉: 5

概要

MTUのカウンタ値を取得

引数

MTU0～MTU4

uint16_t * counter_val	カウンタ値の格納先

MTU5

uint16_t * counter_u_val	カウンタU値の格納先
uint16_t * counter_v_val	カウンタV値の格納先
uint16_t * counter_w_val	カウンタW値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

使用RPDL関数

R_MTU2_ReadChannel

詳細

- MTUのカウンタ値を取得します。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル0を設定
- TGRAをインプットキャプチャレジスタに設定し、インプットキャプチャA割り込みを有効に設定
- インプットキャプチャA割り込み通知関数名にMtu0IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter_val;

void func(void)
{
	R_PG_Timer_Set_MTU_U0_C0();	// MTU0の設定
	R_PG_Timer_StartCount_MTU_U0_C0();	// カウント動作開始
}

void Mtu0IcCmAIntFunc(void)
{
	//MTUのカウンタ値を取得
	R_PG_Timer_GetCounterValue_MTU_U0_C0( & counter_val );
}
```

5.12.6 R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉)

定義

bool R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t counter_val)
〈ユニット番号〉: 0 〈チャネル番号〉: 0～4

bool R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉
(uint16_t counter_val)
〈ユニット番号〉: 0 〈チャネル番号〉: 5 〈相〉: U, V, W

bool R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t counter_u_val, uint16_t counter_v_val, uint16_t counter_w_val)
〈ユニット番号〉: 0 〈チャネル番号〉: 5

概要

MTUのカウンタ値を設定

引数

MTU0～MTU7

uint16_t counter_val	カウンタに設定する値

MTU5

uint16_t counter_u_val	カウンタUに設定する値
uint16_t counter_v_val	カウンタVに設定する値
uint16_t counter_w_val	カウンタWに設定する値

戻り値

true	カウンタ値の設定に成功した場合
false	カウンタ値の設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

使用RPDL関数

R_MTU2_ControlChannel

詳細

- MTUのカウンタ値を設定します。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
  コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();  // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_SetCounterValue_MTU_U0_C1( 0 );   //カウンタの0クリア
}
```

5.12.7 R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(bool* cm_ic_a, bool* cm_ic_b, bool* cm_ic_c, bool* cm_ic_d,
bool* cm_e, bool* cm_f, bool* ov, bool* un);
〈ユニット番号〉: 0
〈チャネル番号〉: 0～4

bool R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(bool* cm_ic_u, bool* cm_ic_v, bool* cm_ic_w);
〈ユニット番号〉: 0
〈チャネル番号〉: 5

概要

MTUの割り込み要求フラグの取得とクリア

引数

bool* cm_ic_a	コンペアマッチ/インプットキャプチャAフラグの格納先
bool* cm_ic_b	コンペアマッチ/インプットキャプチャBフラグの格納先
bool* cm_ic_c	コンペアマッチ/インプットキャプチャCフラグの格納先
bool* cm_ic_d	コンペアマッチ/インプットキャプチャDフラグの格納先
bool* cm_e	コンペアマッチEフラグの格納先
bool* cm_f	コンペアマッチFフラグの格納先
bool* ov	オーバフローフラグの格納先
bool* un	アンダフローフラグの格納先
bool* cm_ic_u	コンペアマッチ/インプットキャプチャUフラグの格納先
bool* cm_ic_v	コンペアマッチ/インプットキャプチャVフラグの格納先
bool* cm_ic_w	コンペアマッチ/インプットキャプチャWフラグの格納先

各チャネルで有効なフラグは以下です。

MTU0	cm_ic_a～cm_ic_d, cm_e, cm_f, ov
MTU1, 2	cm_ic_a, cm_ic_b, ov, un
MTU3, 4	cm_ic_a～cm_ic_d, ov
MTU5	cm_ic_u, cm_ic_v, and cm_ic_w
MTU3 (相補PWMモードおよびリセット同期PWMモード)	cm_ic_a～cm_ic_d
MTU4 (相補PWMモードおよびリセット同期PWMモード)	cm_ic_a～cm_ic_d, un

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

使用RPDL関数

R_MTU2_ReadChannel

詳細

- MTUの割り込み要求フラグを取得します。
- 本関数内で全フラグがクリアされます。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
- コンペアマッチA割り込みの優先レベルを0に設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool cma_flag;

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   //  カウント動作開始

    //コンペアマッチAの発生を待つ
    do{
        R_PG_Timer_GetRequestFlag_MTU_U0_C1(
            & cma_flag, //a
            0, //b
            0, //c
            0, //d
            0, //e
            0, //f
            0, //e
            0, //ov
            0 //un
        );
    } while( !cma_flag );

    //コンペアマッチA発生時処理

}
```

5.12.8 R_PG_Timer_StopModule_MTU_U〈ユニット番号〉

定義 bool R_PG_Timer_StopModule_MTU_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 MTUのユニットを停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_Timer_MTU_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数 R_MTU2_Destroy

詳細
- MTUを停止し、モジュールストップ状態に移行します。複数のチャネルが動作している場合、本関数を呼び出すと全チャネルが停止します。1チャネルの動作だけを停止させる場合はR_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    // MTUユニット0の停止
    R_PG_Timer_StopModule_MTU_U0();
}
```

5.12.9 R_PG_Timer_GetTGR_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetTGR_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t* tgr_a_val, uint16_t* tgr_b_val, uint16_t* tgr_c_val,
uint16_t* tgr_d_val, uint16_t* tgr_e_val, uint16_t* tgr_f_val);
〈ユニット番号〉: 0
〈チャネル番号〉: 0～4

bool R_PG_Timer_GetTGR_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * tgr_u_val, uint16_t * tgr_v_val, uint16_t * tgr_w_val);
〈ユニット番号〉: 0
〈チャネル番号〉: 5

概要

ジェネラルレジスタの値の取得

引数

uint16_t* tgr_a_val	ジェネラルレジスタA値の格納先
uint16_t* tgr_b_val	ジェネラルレジスタB値の格納先
uint16_t* tgr_c_val	ジェネラルレジスタC値の格納先
uint16_t* tgr_d_val	ジェネラルレジスタD値の格納先
uint16_t* tgr_e_val	ジェネラルレジスタE値の格納先
uint16_t* tgr_f_val	ジェネラルレジスタF値の格納先
uint16_t* tgr_u_val	ジェネラルレジスタU値の格納先
uint16_t* tgr_v_val	ジェネラルレジスタV値の格納先
uint16_t* tgr_w_val	ジェネラルレジスタW値の格納先

各チャネルで有効な引数は以下です。

MTU0	tgr_a_val ～ tgr_f_val
MTU1, 2	tgr_a_val, tgr_b_val
MTU3, 4	tgr_a_val ～ tgr_d_val
MTU5	tgr_u_val ～ tgr_w_val

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～5

使用RPDL関数

R_MTU2_ReadChannel

詳細

- ジェネラルレジスタの値を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル0を設定
- TGRAをインプットキャプチャレジスタに設定し、インプットキャプチャA割り込みを有効に設定
- インプットキャプチャA割り込み通知関数名にMtu0IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t tgr_a_val;

void func(void)
{
	R_PG_Timer_Set_MTU_U0_C0();	// MTU0の設定
	R_PG_Timer_StartCount_MTU_U0_C0();	// カウント動作開始
}

void Mtu0IcCmAIntFunc(void)
{
	//TGRAの値を取得
	R_PG_Timer_GetTGR_MTU_U0_C0(
		& tgr_a_val, //a
		0, //b
		0, //c
		0, //d
		0, //e
		0 //f
	);
}
```

5.12.10 R_PG_Timer_SetTGR_*〈ジェネラルレジスタ〉*_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*

定義 bool R_PG_Timer_SetTGR_*〈ジェネラルレジスタ〉*_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*
(uint16_t value);

〈ジェネラルレジスタ〉:

MTU1, 2	: A または B
MTU3, 4	: A, B, C または D
MTU5	: U, V または W

〈ユニット番号〉: 0

〈チャネル番号〉: 0～5

概要 ジェネラルレジスタの値の設定

引数

uint16_t value	ジェネラルレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*.c

〈ユニット番号〉: 0

〈チャネル番号〉: 0～5

使用RPDL関数 R_MTU2_ControlChannel

詳細

- ジェネラルレジスタの値を設定します。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_SetTGR_A_MTU_U0_C1( 1000 );   //TGRAの設定
}
```

5.12.11 R_PG_Timer_SetBuffer_AD_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SetBuffer_AD_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t tadcobr_a_val, uint16_t tadcobr_b_val);
〈ユニット番号〉: 0
〈チャネル番号〉: 4

概要

A/D変換要求周期設定バッファレジスタの設定

生成条件

A/D変換要求周期レジスタ値のバッファ転送が有効

引数

uint16_t tadcobr_a_val	A/D変換要求周期設定バッファレジスタAに設定する値
uint16_t tadcobr_b_val	A/D変換要求周期設定バッファレジスタBに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 3(*), 4 (*相補PWMモードおよびリセット同期PWMモード)

使用RPDL関数

R_MTU2_ControlChannel

詳細

- A/D変換要求周期設定バッファレジスタAおよびB(TADCOBRA、TADCOBRB)を設定します。

使用例

GUI上で以下の通り設定した場合

- A/D変換要求周期レジスタ値のバッファ転送を有効に設定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Timer_Set_MTU_U0_C4();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C4();   // カウント動作開始
}

void func2(void)
{
    // A/D変換要求周期設定バッファレジスタの設定
    R_PG_Timer_SetBuffer_AD_MTU_U0_C4( 0x10, 0x20 );
}
```

5.12.12 R_PG_Timer_SetBuffer_CycleData_MTU_U〈*ユニット番号*〉_〈*チャネル*〉

定義

bool R_PG_Timer_SetBuffer_CycleData_MTU_U〈*ユニット番号*〉_〈*チャネル*〉
(uint16_t tcbr_val);
〈*ユニット番号*〉: 0
〈*チャネル*〉: C3_C4

概要

周期バッファレジスタ値の設定

生成条件

MTUチャネルを相補PWMモードに設定

引数

uint16_t tcbr_val	周期バッファレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 3

使用RPDL関数

R_MTU2_ControlChannel

詳細

- タイマ周期バッファレジスタ(TCBR)を設定します。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
      R_PG_Timer_SetBuffer_CycleData_MTU_U0_C3_C4(0x1000);
}
```

5.12.13 R_PG_Timer_SetOutputPhaseSwitch_MTU_U〈ユニット番号〉_〈チャネル〉

定義

bool R_PG_Timer_SetOutputPhaseSwitch_MTU_U〈ユニット番号〉_〈チャネル〉
(uint8_t output_level);
〈ユニット番号〉: 0
〈チャネル〉: C3_C4

概要

PWM出力レベルの切り替え

生成条件

- MTUチャネルを相補PWMモードまたはリセット同期PWMモードに設定
- DCブラシレスモータ制御を有効に設定し、出力制御方法にソフトウェアを指定

引数

uint8_t output_level	出力設定 (0～5)

各値での出力は以下の通りです

値	MTIOC3B U相	MTIOC4A V相	MTIOC4B W相	MTIOC3D U相	MTIOC4C V相	MTIOC4D W相
0	OFF	OFF	OFF	OFF	OFF	OFF
1	ON	OFF	OFF	OFF	OFF	ON
2	OFF	ON	OFF	ON	OFF	OFF
3	OFF	ON	OFF	OFF	OFF	ON
4	OFF	OFF	ON	OFF	ON	OFF
5	ON	OFF	OFF	OFF	ON	OFF
6	OFF	OFF	ON	ON	OFF	OFF
7	OFF	OFF	OFF	OFF	OFF	OFF

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 3

使用RPDL関数

R_MTU2_ControlUnit

詳細

- DBブラシレスモータ制御時のPWM出力レベルを切り替えます

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_SetOutputPhaseSwitch_MTU_U0_C3_C4(0x7);
}
```

5.12.14 R_PG_Timer_ControlOutputPin_MTU_U〈ユニット番号〉_〈チャネル〉

定義

bool R_PG_Timer_ControlOutputPin_MTU_U〈ユニット番号〉_〈チャネル〉
(bool p1_enable, bool n1_enable, bool p2_enable, bool n2_enable,
bool p3_enable, bool n3_enable)
〈ユニット番号〉: 0
〈チャネル〉: C3_C4

概要

PWM出力の有効化/無効化

生成条件

MTUチャネルを相補PWMモードまたはリセット同期PWMモードに設定

引数

bool p1_enable	U相 正相（MTIOCmB）出力（0:出力無効 1:出力有効）
bool n1_enable	U相 逆相（MTIOCmD）出力（0:出力無効 1:出力有効）
bool p2_enable	V相 正相（MTIOCnA）出力（0:出力無効 1:出力有効）
bool n2_enable	V相 逆相（MTIOCnC）出力（0:出力無効 1:出力有効）
bool p3_enable	W相 正相（MTIOCnB）出力（0:出力無効 1:出力有効）
bool n3_enable	W相 逆相（MTIOCnD）出力（0:出力無効 1:出力有効）

m：3 n：4

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 3

使用RPDL関数

R_MTU2_ControlUnit

詳細

- 相補PWMモード、リセット同期PWMモードの6相のPWM出力を有効化、無効化します。
- 相補PWMモードおよびリセット同期PWMモードでは、初期状態でPWM出力が無効です。端子出力を有効にするには、カウントを開始する前に本関数を呼び出してください。

使用例

R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉の使用例2を参照してください。

5.12.15 R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U〈*ユニット番号*〉_〈*チャネル*〉

定義

bool R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U〈*ユニット番号*〉_〈*チャネル*〉
(bool p1_high, bool n1_high, bool p2_high, bool n2_high,
bool p3_high, bool n3_high)
〈*ユニット番号*〉: 0
〈*チャネル*〉: C3_C4

概要

PWM出力レベルをバッファレジスタに設定

生成条件

- MTUチャネルを相補PWMモードまたはリセット同期PWMモードに設定
- PWM出力レベル設定のバッファ転送を有効に設定

引数

bool p1_high	U相 正相（MTIOCmB）出力
bool n1_high	U相 逆相（MTIOCmD）出力
bool p2_high	V相 正相（MTIOCnA）出力
bool n2_high	V相 逆相（MTIOCnC）出力
bool p3_high	W相 正相（MTIOCnB）出力
bool n3_high	W相 逆相（MTIOCnD）出力

m：3 n：4

各値での出力レベルは以下の通りです。

値	種別	正相	逆相
0	アクティブレベル	Low	Low
	初期出力	Low	Low
	アップカウント時コンペアマッチ	Low	High
	ダウンカウント時コンペアマッチ	High	Low
1	アクティブレベル	High	High
	初期出力	High	High
	アップカウント時コンペアマッチ	High	Low
	ダウンカウント時コンペアマッチ	Low	High

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 3

使用RPDL関数

R_MTU2_ControlUnit

詳細

- PWM出力レベル設定をタイマアウトプットレベルバッファレジスタ（TOLBR）に設定します。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U0_C3_C4( 0, 0, 0, 0, 0, 0 );
}
```

5.12.16 R_PG_Timer_ControlBufferTransfer_MTU_U〈ユニット番号〉_〈チャネル〉

定義

bool R_PG_Timer_ControlBufferTransfer_MTU_U〈*ユニット番号*〉_〈*チャネル*〉
(bool enable)
〈*ユニット番号*〉: 0
〈*チャネル*〉: C3_C4

概要

バッファレジスタからテンポラリレジスタへのバッファ転送の有効化、無効化

生成条件

- MTUチャネルを相補PWMモードに設定
- 割り込み間引き機能を設定

引数

bool enable	バッファ転送設定 (0:有効 1:無効)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 3

使用RPDL関数

R_MTU2_ControlUnit

詳細

- 相補PWMモードで使用するバッファレジスタからテンポラリレジスタへのバッファ転送を有効化、無効化します

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_ControlBufferTransfer_MTU_U0_C3_C4( 1 );
}
```

5.13 ポートアウトプットイネーブル 2 (POE2a)

5.13.1 R_PG_POE_Set

定義　bool R_PG_POE_Set (void)

概要　POEの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_POE.c

使用RPDL関数　R_POE_Set, R_POE_Create

詳細

- GUI上で選択されたMTU0, 3, 4の出力端子の制御と、ハイインピーダンス要求信号に使用する入力端子、アウトプットイネーブル割り込みを設定します。
- MTUの端子出力は、MTUのGUIおよび関数により設定してください。MTUで出力端子に設定していない端子は、POEで設定しないでください。
- GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void *〈割り込み通知関数名〉*(void)
  割り込み通知関数については「通知関数に関する注意事項」の内容に注意してください。

使用例　GUI上で以下の通り設定した場合

- アウトプットイネーブル割り込み2(OEI2)を有効に設定し、割り込み通知関数名にPoeOei2IntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_POE_Set();     // POEの設定
}

void PoeOei2IntFunc (void)
{
    //アウトプットイネーブル割り込み処理
}
```

5.13.2 R_PG_POE_SetHiZ_*<タイマチャネル>*

定義 bool R_PG_POE_SetHiZ_*<タイマチャネル>*(void)

<タイマチャネル> : MTU3_4, MTU0

概要 タイマ出力端子をハイインピーダンスに設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_POE.c

使用RPDL関数 R_POE_Control

詳細

- GUI上でハイインピーダンス制御対象に指定されたMTU0, 3, 4の出力端子をハイインピーダンス状態にします。

使用例

GUI上で以下の通り設定した場合

- MTU0の端子出力を設定（MTUの設定GUI上）
- MTU0の出力端子をPOEのハイインピーダンス制御対象に指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Timer_Set_MTU_U0_C0();    //MTU0の設定
    R_PG_POE_Set();      // POEの設定
    R_PG_Timer_StartCount_MTU_U0_C0();      //MTU0のカウント動作開始
}

void func2(void)
{
    R_PG_POE_SetHiZ_MTU0();      //MTU0の出力端子をHiZに設定
}
```

5.13.3 R_PG_POE_GetRequestFlagHiZ_<タイマチャネル/フラグ>

定義

bool R_PG_POE_GetRequestFlagHiZ_MTU3_4
(bool * poe0, bool * poe1, bool * poe2, bool * poe3)

bool R_PG_POE_GetRequestFlagHiZ_MTU0 (bool * poe8)

bool R_PG_POE_GetRequestFlagHiZ_OSTSTF (bool * oststf)

概要

ハイインピーダンス要求フラグの取得

引数

bool* poe0	POE0#端子のハイインピーダンス要求フラグの格納先
bool* poe1	POE1#端子のハイインピーダンス要求フラグの格納先
bool* poe2	POE2#端子のハイインピーダンス要求フラグの格納先
bool* poe3	POE3#端子のハイインピーダンス要求フラグの格納先
bool* poe8	POE8#端子のハイインピーダンス要求フラグの格納先
bool* oststf	OSTSTハイインピーダンス要求フラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_POE.c

使用RPDL関数

R_POE_GetStatus

詳細

- POEn#端子へのハイインピーダンス要求信号入力フラグ(POEnF)を取得します。(n:0～3,8)
- 取得するフラグに対応する引数に格納先アドレスを指定してください。取得しないフラグに対応する引数には0を指定してください。
- GUI上でハイインピーダンス要求条件に指定していないPOE端子のフラグには有効な値が格納されません。

使用例

GUI上で以下の通り設定した場合

- MTU3,4の端子出力を設定 (MTUの設定GUI上)
- MTU3,4の出力端子をPOEのハイインピーダンス制御対象に指定
- ハイインピーダンス要求条件にPOE0を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool poe0;

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C3();    //MTUの設定
    R_PG_POE_Set();    // POEの設定
    R_PG_Timer_StartCount_MTU_U0_C3();    //MTUのカウント動作開始

    //ハイインピーダンス要求入力を待つ
    do{
        R_PG_POE_GetRequestFlagHiZ_MTU3_4( &poe0, 0, 0, 0 );
    }while( ! poe0 );

    //ハイインピーダンス要求入力時処理

    R_PG_POE_ClearFlag_MTU3_4();    //ハイインピーダンス要求フラグのクリア
}
```

5.13.4 R_PG_POE_GetShortFlag_〈タイマチャネル〉

定義 bool R_PG_POE_GetShortFlag_〈タイマチャネル〉(bool * detected)

〈タイマチャネル〉: MTU3_4

概要 MTU端子の出力短絡フラグの取得

引数

bool* detected	出力短絡フラグ(OSF1)の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_POE.c

使用RPDL関数 R_POE_GetStatus

詳細

- MTU3,4の相補PWM出力短絡フラグ(OSF1)を取得します。

使用例

GUI上で以下の通り設定した場合

- アウトプットイネーブル割り込み1(OEI1)を有効に設定
- アウトプットイネーブル割り込み1の通知関数名にPoeOei1IntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_POE_Set();    // POEの設定
}

void PoeOei1IntFunc(void)
{
    bool detected;

    //出力短絡フラグの取得
    R_PG_POE_GetShortFlag_MTU3_4 (&detected);

    if( detected ){
        //MTU3,4の出力短絡検出時処理

        R_PG_POE_ClearFlag_MTU3_4();    //出力短絡フラグ(OSF1)のクリア
    }
}
```

5.13.1 R_PG_POE_ClearFlag_〈タイマチャネル/フラグ〉

定義　bool R_PG_POE_ClearFlag_〈タイマチャネル/フラグ〉(void)

〈タイマチャネル/フラグ〉: MTU3_4, MTU0, OSTSTF

概要　ハイインピーダンス要求フラグと出力短絡フラグのクリア

引数　なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル　R_PG_POE.c

使用RPDL関数　R_POE_Control

詳細

- ハイインピーダンス要求フラグと出力短絡フラグをクリアします。
- タイマの各チャネル、フラグに対応した関数でクリアされるフラグは次の通りです。

タイマチャネル/フラグ	クリア対象
MTU3, 4	POEn要求フラグ(POEnF) (n:0～3) MTU3,4出力短絡フラグ(OSF1)
MTU0	POE8要求フラグ(POE8F)
OSTSTF	OSTSTハイインピーダンスフラグ

使用例　R_PG_POE_GetShortFlag_〈タイマチャネル〉 の使用例を参照してください。

5.14 16 ビットタイマパルスユニット（TPUa）

5.14.1 R_PG_Timer_Set_TPU_U〈ユニット番号〉

定義　bool R_PG_Timer_Set_TPU_U〈ユニット番号〉(void)
〈ユニット番号〉： 0

概要　TPUの複数チャネルを設定

引数　なし

戻り値

True	設定が正しく行われた場合
False	設定に失敗した場合

出力先ファイル　R_PG_Timer_TPU_U〈ユニット番号〉.c
〈ユニット番号〉： 0

使用RPDL関数　R_TPU_Create

詳細
- TPUのモジュールストップ状態を解除して同一ユニットの複数のチャネルを初期設定します。
- 本関数内でTPUの割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。　void 〈割り込み通知関数名〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み通知関数名を指定しない場合、割り込みが発生しても割り込みハンドラは呼び出されません。要求フラグの状態は R_PG_Timer_GetRequestFlag_TPU_U〈ユニット番号〉_C〈チャネル番号〉 により取得することができます。
- 外部入力カウントクロック、外部リセット信号、パルス出力を使用する場合、本関数内で使用する端子の入出力方向と入力バッファを設定します。

使用例　GUI上で以下の通り設定した場合
- TPU ユニット0 を設定
- チャネル0のコンペアマッチA割り込み通知関数名に Tpu0IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TPU ユニット0を設定
    R_PG_Timer_Start_TPU_U0();
}

void Tpu0IcCmAIntFunc(void)
{
    func_cmA();     //コンペアマッチA割り込み発生時の処理
}
```

5.14.2 R_PG_Timer_Start_TPU_U〈ユニット番号〉_C〈チャネル番号〉

定義　bool R_PG_Timer_Start_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉(void)
〈*ユニット番号*〉： 0
〈*チャネル番号*〉： 0～5

概要　TPUを設定しカウントを開始

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉： 0
〈*チャネル番号*〉： 0～5

使用RPDL関数　R_TPU_Create

詳細
- TPUのモジュールストップ状態を解除して初期設定し、カウントを開始します。
- 本関数内でTPUの割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。　void 〈*割り込み通知関数名*〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み通知関数名を指定しない場合、割り込みが発生しても割り込みハンドラは呼び出されません。要求フラグの状態は R_PG_Timer_GetRequestFlag_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉 により取得することができます。
- 外部入力カウントクロック、外部リセット信号、パルス出力を使用する場合、本関数内で使用する端子の入出力方向と入力バッファを設定します。

使用例　GUI上で以下の通り設定した場合
- TPU チャネル1 を設定
- コンペアマッチA割り込み通知関数名に Tpu1IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TPU1を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C1();
}

void Tpu1IcCmAIntFunc(void)
{
    func_cmA();    //コンペアマッチA割り込み発生時の処理
}
```

5.14.1 R_PG_Timer_SynchronouslyStartCount_TPU_U〈*ユニット番号*〉

定義 bool R_PG_Timer_SynchronouslyStartCount_TPU_U〈ユニット番号〉
(bool ch0, bool ch1, bool ch2, bool ch3, bool ch4,bool ch5)
〈*ユニット番号*〉 : 0

概要 TPUの複数チャネルのカウント動作を同時に開始

引数 ユニット0

bool ch0	チャネル0のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch1	チャネル1のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch2	チャネル2のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch3	チャネル3のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch4	チャネル4のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch5	チャネル5のカウント動作 (0:カウント開始しない 1:カウント開始)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_TPU_U〈*ユニット番号*〉.c
〈*ユニット番号*〉 : 0

使用RPDL関数 R_TPU_ControlUnit

詳細
- TPUの複数チャネルのカウント動作を同時に開始します。
- あらかじめR_PG_Timer_SetTPU_U〈*ユニット番号*〉によりTPUを初期設定してください。

使用例 GUI上で以下の通り設定した場合
- TPU ユニット0 チャネル0, 1 のカウント動作を開始

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TPU0、1のカウントを開始
    R_PG_Timer_SynchronouslyStartCount_TPU_U0( 1, 1, 0, 0, 0, 0 );
}
```

5.14.2 R_PG_Timer_HaltCount_TPU_U〈ユニット番号〉_C〈チャネル番号〉

定義　bool R_PG_Timer_HaltCount_TPU_U〈ユニット番号〉_C〈チャネル番号〉(void)
〈ユニット番号〉： 0
〈チャネル番号〉： 0～5

概要　TPUのカウントを一時停止

引数　なし

戻り値

true	停止が正しく行われた場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_TPU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉： 0
〈チャネル番号〉： 0～5

使用RPDL関数　R_TPU_ControlChannel

詳細
- TPUのカウントを一時停止します。カウントを再開するには
  R_PG_Timer_ResumeCount_TPU_U〈ユニット番号〉_C〈チャネル番号〉
  を呼び出してください。

使用例　GUI上で以下の通り設定した場合
- TPU チャネル1 を設定
- コンペアマッチA割り込み通知関数名に Tpu1IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TPU1を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C1();
}

void Tpu1IcCmAIntFunc(void)
{
    //TPU1のカウントを一時停止
    R_PG_Timer_HaltCount_TPU_U0_C1();

    func_cmA();     //コンペアマッチA割り込み発生時の処理

    //TPU1のカウントを再開
    R_PG_Timer_ResumeCount_TPU_U0_C1();
}
```

5.14.3 R_PG_Timer_ResumeCount_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義　bool R_PG_Timer_ResumeCount_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉(void)
〈*ユニット番号*〉： 0
〈*チャネル番号*〉： 0～5

概要　TPUのカウントを再開

引数　なし

戻り値

true	カウントの再開が正しく行われた場合
false	カウントの再開に失敗した場合

出力先ファイル　R_PG_Timer_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉： 0
〈*チャネル番号*〉： 0～5

使用RPDL関数　R_TPU_ControlChannel

詳細
- R_PG_Timer_HaltCount_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉により停止したTPUのカウントを再開します。

使用例　GUI上で以下の通り設定した場合
- TPU チャネル1 を設定
- コンペアマッチA割り込み通知関数名に Tpu1IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TPU1を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C1();
}

void Tpu1IcCmAIntFunc(void)
{
    //TPU1のカウントを一時停止
    R_PG_Timer_HaltCount_TPU_U0_C1();

    func_cmA();     //コンペアマッチA割り込み発生時の処理

    //TPU1のカウントを再開
    R_PG_Timer_ResumeCount_TPU_U0_C1();
}
```

5.14.4 R_PG_Timer_GetCounterValue_TPU_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_GetCounterValue_TPU_U〈ユニット番号〉_C〈チャネル番号〉(uint16_t * data)

〈ユニット番号〉 : 0

〈チャネル番号〉 : 0～5

概要 TPUのカウンタ値を取得

引数

uint16_t * data	取得したカウンタ値の格納先

戻り値

true	カウンタ値の取得が正しく行われた場合
false	カウンタ値の取得に失敗した場合

出力先ファイル R_PG_Timer_TPU_U〈ユニット番号〉_C〈チャネル番号〉.c

〈ユニット番号〉 : 0

〈チャネル番号〉 : 0～5

使用RPDL関数 R_TPU_Read

詳細

- TPUのカウンタ値を取得します。

使用例 GUI上で以下の通り設定した場合

- TPU チャネル0 を設定
- TGRAをインプットキャプチャレジスタに設定し、インプットキャプチャ割り込みを設定
- インプットキャプチャA割り込み通知関数名に Tpu0IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter;

void func(void)
{
    //TPU0を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C0();
}

void Tpu0IcCmAIntFunc(void)
{
    //TPU0のカウンタ値を取得
    R_PG_Timer_GetCounterValue_TPU_U0_C0( &counter );
}
```

5.14.5 R_PG_Timer_SetCounterValue_TPU_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_SetCounterValue_TPU_U〈ユニット番号〉_C〈チャネル番号〉(uint16_t data)

〈ユニット番号〉 : 0

〈チャネル番号〉 : 0～5

概要 TPUのカウンタ値を設定

引数

uint16_t data	カウンタに設定する値

戻り値

true	カウンタ値の設定が正しく行われた場合
false	カウンタ値の設定に失敗した場合

出力先ファイル R_PG_Timer_TPU_U〈ユニット番号〉_C〈チャネル番号〉.c

〈ユニット番号〉 : 0

〈チャネル番号〉 : 0～5

使用RPDL関数 R_TPU_ControlChannel

詳細

- TPUのカウンタ値を設定します。

使用例 GUI上で以下の通り設定した場合

- TPU チャネル1 を設定
- TGRAをアウトプットコンペアレジスタに設定し、アウトプットコンペア割り込みを設定
  コンペアマッチA割り込み通知関数名に Tpu1IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter;

void func(void)
{
    //TPU1を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C1();
}

void Tpu1IcCmAIntFunc(void)
{
    //TPU1のカウンタ値を設定
    R_PG_Timer_SetCounterValue_TPU_U0_C1( counter );
}
```

5.14.6 R_PG_Timer_GetTGR_TPU_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_GetTGR_TPU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * tgr_a_val, uint16_t * tgr_b_val, uint16_t * tgr_c_val, uint16_t * tgr_d_val)
〈ユニット番号〉 : 0
〈チャネル番号〉 : 0～5

概要 ジェネラルレジスタの値の取得

引数

uint16_t * tgr_a_val	ジェネラルレジスタA値の格納先
uint16_t * tgr_b_val	ジェネラルレジスタB値の格納先
uint16_t * tgr_c_val	ジェネラルレジスタC値の格納先
uint16_t * tgr_d_val	ジェネラルレジスタD値の格納先

各チャネルで有効な引数は以下です。

TPU0, 3, 6, 9	tgr_a_val ～ tgr_d_val
TPU1, 2, 4, 5, 7, 8, 10, 11	tgr_a_val, tgr_b_val

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_TPU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉 : 0
〈チャネル番号〉 : 0～5

使用RPDL関数 R_TPU_Read

詳細
- ジェネラルレジスタの値を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。

使用例 GUI上で以下の通り設定した場合
- TPUチャネル0を設定
- TGRAをインプットキャプチャレジスタに設定し、インプットキャプチャA割り込みを有効に設定
- インプットキャプチャA割り込み通知関数名にTpu0IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t tgr_a_val;

void func(void)
{
    R_PG_Timer_Start_TPU_U0_C0();   //TPUを設定しカウントを開始
}

void Tpu0IcCmAIntFunc(void)
{
    //TGRAの値を取得
    R_PG_Timer_GetTGR_TPU_U0_C0(&tgr_a_val, 0, 0, 0);
}
```

5.14.7 R_PG_Timer_SetTGR_〈ジェネラルレジスタ〉_TPU_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_SetTGR_〈ジェネラルレジスタ〉_TPU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t value);

〈ジェネラルレジスタ〉:

TPU0, 3	: A, B, C, D
TPU1, 2, 4, 5	: A または B

〈ユニット番号〉: 0,
〈チャネル番号〉: 0～5

概要 ジェネラルレジスタの値の設定

引数

uint16_t value	ジェネラルレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_TPU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉 : 0,
〈チャネル番号〉 : 0～5

使用RPDL関数 R_TPU_ControlChannel

詳細
- ジェネラルレジスタの値を設定します。

使用例 GUI上で以下の通り設定した場合
- TPUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
- コンペアマッチA割り込み通知関数名にTpu1IcCmAIntFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Timer_Start_TPU_U0_C1();  //TPUを設定しカウントを開始
}

void Tpu1IcCmAIntFunc(void)
{
    R_PG_Timer_SetTGR_A_TPU_U0_C1( 1000 );  //TGRAの設定
}
```

5.14.8 R_PG_Timer_GetRequestFlag_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

```
bool R_PG_Timer_GetRequestFlag_TPU_U〈ユニット番号〉_C〈チャネル番号〉(
    bool* a,
    bool* b,
    bool* c,
    bool* d,
    bool* ov,
    bool* un
);
```

〈*ユニット番号*〉 : 0

〈*チャネル番号*〉 : 0～5

概要

TPUの割り込み要求フラグの取得とクリア

引数

bool* a	コンペアマッチ/インプットキャプチャAフラグの格納先
bool* b	コンペアマッチ/インプットキャプチャBフラグの格納先
bool* c	コンペアマッチ/インプットキャプチャCフラグの格納先
bool* d	コンペアマッチ/インプットキャプチャDフラグの格納先
bool* ov	オーバフローフラグの格納先
bool* un	アンダフローフラグの格納先

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル

R_PG_Timer_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c

〈*ユニット番号*〉 : 0

〈*チャネル番号*〉 : 0～5

使用RPDL関数

R_TPU_Read

詳細

- TPUの割り込み要求フラグを取得します。
- 本関数内で全フラグをクリアします。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。
- コンペアマッチ/インプットキャプチャC、Dはチャネル0、3でのみ取得可能です。それ以外のチャネルでは0を指定してください。

使用例 GUI上で以下の通り設定した場合

- TPU チャネル1 を設定
- TGRAをアウトプットコンペアレジスタに設定し、アウトプットコンペア割り込みを設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool cma_flag;

void func(void)
{
    //TPU1を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C1();

    //コンペアマッチAの検出を待つ
    do{
        R_PG_Timer_GetRequestFlag_TPU_U0_C1(
            & cma_flag,
            0,
            0,
            0,
            0,
            0
        );
    } while( !cma_flag );

    func_cmA();     //コンペアマッチA割り込み発生時の処理

    R_PG_Timer_StopModule_TPU_U0();     //TPUを停止
}
```

5.14.9 R_PG_Timer_StopModule_TPU_U〈*ユニット番号*〉

定義　bool R_PG_Timer_StopModule_TPU_U〈*ユニット番号*〉(void)
〈*ユニット番号*〉 : 0

概要　TPUを停止

引数　なし

戻り値

true	停止が正しく行われた場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_TPU_U〈*ユニット番号*〉.c
〈*ユニット番号*〉 : 0

使用RPDL関数　R_TPU_Destroy

詳細
- TPUを停止し、モジュールストップ状態に移行します。本関数を呼び出すと全チャネルが停止します。チャネルごとにカウントを停止させる場合は、
R_PG_Timer_HaltCount_TPU_U〈*ユニット番号*〉_C〈*チャネル番号*〉
を使用してください。

使用例　GUI上で以下の通り設定した場合
- TPU チャネル1 を設定
- TGRAをアウトプットコンペアレジスタに設定し、アウトプットコンペア割り込みを設定
- コンペアマッチA割り込み通知関数名に Tpu1IcCmAIntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter;

void func(void)
{
    //TPU1を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U0_C1();
}

void Tpu1IcCmAIntFunc(void)
{
    //TPUを停止
    R_PG_Timer_StopModule_TPU_U0( counter );
}
```

5.15 8 ビットタイマ（TMR）

5.15.1 R_PG_Timer_Start_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉)

定義　bool R_PG_Timer_Start_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉) (void)
〈*ユニット番号*〉 ： 0, 1
〈*チャネル番号*〉 ： 0～3
((_C〈*チャネル番号*〉) は8ビットモード時に付加します)

概要　TMRを設定しカウント動作を開始

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_TMR_U〈*ユニット番号*〉.c
〈*ユニット番号*〉 ： 0, 1

使用RPDL関数　R_TMR_Set
R_TMR_CreateChannel (8ビットモード時)
R_TMR_CreateUnit (16ビットモード時)

詳細
- TMRのモジュールストップ状態を解除して初期設定し、カウント動作を開始します。8ビットモード時はチャネルごとに、16ビットモード(ユニット内の2チャネルをカスケード接続)時はユニットごとに設定します。
- 本関数内でTMRの割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。　void 〈*割り込み通知関数名*〉(void)
割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み優先レベルを0に設定した場合、CPU割り込みは発生しません。割り込み要求フラグは R_PG_Timer_GetRequestFlag_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉) により取得することができます。
- 外部入力カウントクロック、外部リセット信号、パルス出力を使用する場合、本関数内で使用する端子の設定を行います。

使用例　16ビットタイマモードでTMRのユニット1を設定
GUI上で次の割り込み通知関数を設定した場合
オーバフロー割り込み ： TmrOf2IntFunc
コンペアマッチA割り込み ： TmrCma2IntFunc
コンペアマッチB割り込み ： TmrCmb2IntFunc

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TMRユニット1を16ビットモードで設定する
    R_PG_Timer_Start_TMR_U1();
}
```

```
void TmrOf2IntFunc(void)
{
    func_of();      //オーバフロー割り込み発生時の処理
}

void TmrCma2IntFunc(void)
{
    func_cmA();     //コンペアマッチA割り込み発生時の処理
}

void TmrCmb2IntFunc(void)
{
    func_cmB();     //コンペアマッチB割り込み発生時の処理
}
```

GUI上でTMR0を8ビットタイマモードに設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    bool cma_flag;

    //TMR0を8ビットモードで設定し、カウント動作を開始する
    R_PG_Timer_Start_TMR_U0_C0();

    while(1){
        //コンペアマッチA割り込み要求フラグを取得する
        R_PG_Timer_GetRequestFlag_TMR_U0_C0( &cma_flag, 0, 0 );

        if( cma_flag ){
            func_cmA0();    //コンペアマッチA割り込みの処理
        }
    }
}
```

5.15.2 R_PG_Timer_HaltCount_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義　bool R_PG_Timer_HaltCount_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉(void)
〈*ユニット番号*〉： 0, 1
〈*チャネル番号*〉： 0～3
(_C〈*チャネル番号*〉は8ビットモード時に付加します)

概要　TMRのカウント動作を一時停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_TMR_U〈*ユニット番号*〉.c
〈*ユニット番号*〉： 0, 1

使用RPDL関数　R_TMR_ControlChannel (8ビットモード時)
R_TMR_ControlUnit (16ビットモード時)

詳細
- TMRのカウント動作を一時停止します。カウント動作を再開するには R_PG_Timer_ResumeCount_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉を呼び出してください。

使用例　GUI上でTMR0を8ビットタイマモードに設定
GUI上でコンペアマッチA割り込み関数名に TmrCma0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TMR0を8ビットモードで設定する
    R_PG_Timer_Start_TMR_U0_C0();
}

void TmrCma0IntFunc(void)
{
    //TMR0のカウント動作を一時停止
    R_PG_Timer_HaltCount_TMR_U0_C0();

    func_cmA();     //コンペアマッチA割り込み発生時の処理

    //TMR0のカウント動作を再開
    R_PG_Timer_ResumeCount_TMR_U0_C0();
}
```

5.15.3 R_PG_Timer_ResumeCount_TMR_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_ResumeCount_TMR_U〈ユニット番号〉_C〈チャネル番号〉(void)

〈ユニット番号〉 : 0, 1

〈チャネル番号〉 : 0～3

((_C〈チャネル番号〉) は8ビットモード時に付加します)

概要 TMRのカウント動作を再開

引数 なし

戻り値

true	カウント動作の再開が正しく行われた場合
false	カウント動作の再開に失敗した場合

出力先ファイル R_PG_Timer_TMR_U〈ユニット番号〉.c

〈ユニット番号〉 : 0, 1

使用RPDL関数 R_TMR_ControlChannel (8ビットモード時)

R_TMR_ControlUnit (16ビットモード時)

詳細

- R_PG_Timer_HaltCount_TMR_U〈ユニット番号〉_C〈チャネル番号〉により停止したTMRのカウント動作を再開します。

使用例 R_PG_Timer_HaltCount_TMR_U〈ユニット番号〉_C〈チャネル番号〉の使用例を参照してください。

5.15.4 R_PG_Timer_GetCounterValue_TMR_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

・8ビットモード時

bool R_PG_Timer_GetCounterValue_TMR_U〈*ユニット番号*〉C〈*チャネル番号*〉
(uint8_t * counter_val)

〈*ユニット番号*〉 : 0, 1

〈*チャネル番号*〉 : 0～3

・16ビットモード時

bool R_PG_Timer_GetCounterValue_TMR_U〈*ユニット番号*〉(uint16_t * counter_val)

〈*ユニット番号*〉 : 0, 1

概要　TMRのカウンタ値を取得

引数

uint8_t * counter_val uint16_t * counter_val	カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_Timer_TMR_U〈*ユニット番号*〉.c

〈*ユニット番号*〉 : 0, 1

使用RPDL関数　R_TMR_ReadChannel (8ビットモード時)

R_TMR_ReadUnit (16ビットモード時)

詳細

- TMRのカウンタ値を取得します。
  8ビットタイマモード時は指定したチャネルの8ビットカウンタ値が、16ビットモード時は次のように各チャネルのカウンタ値が格納されます。

ユニット	b15 - b8	b7 - b0
0	TMR0カウンタ	TMR1カウンタ
1	TMR2カウンタ	TMR3カウンタ

※16ビットモード時はTMR0(TMR2)が上位ビットとして動作します。

使用例　GUI上でTMR0を8ビットタイマモードに設定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t counter_val;

void func1(void)
{
    //TMR0を8ビットモードで設定する
    R_PG_Timer_Start_TMR_U0_C0();
}

void func2(void)
{
    //TMR0のカウンタ値を取得
    R_PG_Timer_GetCounterValue_TMR_U0_C0( &counter_val );
}
```

5.15.5 R_PG_Timer_SetCounterValue_TMR_U〈ユニット番号〉_C〈チャネル番号〉

定義

・8ビットモード時

bool R_PG_Timer_SetCounterValue_TMR_U〈ユニット番号〉_C〈チャネル番号〉
(uint8_t counter_val)

〈ユニット番号〉 : 0, 1

〈チャネル番号〉 : 0～3

・16ビットモード時

bool R_PG_Timer_SetCounterValue_TMR_U〈ユニット番号〉
(uint16_t counter_val)

〈ユニット番号〉 : 0, 1

概要

TMRのカウンタ値を設定

引数

uint8_t counter_val (8ビットモード時) uint16_t counter_val (16ビットモード時)	カウンタに設定する値

戻り値

true	カウンタ値の設定に成功した場合
false	カウンタ値の設定に失敗した場合

出力先ファイル

R_PG_Timer_TMR_U〈ユニット番号〉.c

〈ユニット番号〉 : 0, 1

使用RPDL関数

R_TMR_ControlChannel (8ビットモード時)

R_TMR_ControlUnit (16ビットモード時)

詳細

・ TMRのカウンタ値を設定します。

8ビットタイマモード時は指定したチャネルの8ビットカウンタ値が、16ビットモード時は次のように各チャネルのカウンタ値が格納されます。

ユニット	b15 - b8	b7 - b0
0	TMR0カウンタ	TMR1カウンタ
1	TMR2カウンタ	TMR3カウンタ

※16ビットモード時はTMR0(TMR2)が上位ビットとして動作します。

使用例

GUI上でTMR0を8ビットタイマモードに設定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //TMR0を8ビットモードで設定する
    R_PG_Timer_Start_TMR_U0_C0();
}

void func2(void)
{
    //TMR0のカウンタ値を設定
    R_PG_Timer_SetCounterValue_TMR_U0_C0( 0 );
}
```

5.15.6 R_PG_Timer_GetRequestFlag_TMR_U〈ユニット番号〉_C〈チャネル番号〉)

定義 bool R_PG_Timer_GetRequestFlag_TMR_U〈ユニット番号〉_C〈チャネル番号〉)
(bool* cma, bool* cmb, bool* ov);
〈ユニット番号〉 : 0, 1
〈チャネル番号〉 : 0～3
((_C〈チャネル番号〉) は8ビットモード時に付加します)

概要 TMRの割り込み要求フラグの取得とクリア

引数

bool* cma	コンペアマッチAフラグの格納先
bool* cmb	コンペアマッチBフラグの格納先
bool* ov	オーバフローフラグの格納先

戻り値

true	フラグの取得に成功した場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_Timer_TMR_U〈ユニット番号〉.c
〈ユニット番号〉 : 0, 1

使用RPDL関数 R_TMR_ReadChannel(8ビットモード時)
R_TMR_ReadUnit(16ビットモード時)

詳細
- TMRの割り込み要求フラグを取得します。
- 本関数内で全フラグがクリアされます。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
- 取得しないフラグには0を指定してください。

使用例 GUI上でTMR0を8ビットタイマモードに設定
GUI上でコンペアマッチA割り込み関数名に TmrCma0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

bool cma_flag;

void func(void)
{
    //TMR0を8ビットモードで設定する
    R_PG_Timer_Start_TMR_U0_C0();

    //コンペアマッチAの検出を待つ
    do{
        R_PG_Timer_GetRequestFlag_TMR_U0_C0(
            & cma_flag,
            0,
            0
        );
    } while( !cma_flag );

    func_cmA();      //コンペアマッチA割り込み発生時の処理
}
```

5.15.7 R_PG_Timer_HaltCountElc_TMR_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_HaltCountElc_TMR_U〈ユニット番号〉_C〈チャネル番号〉(void)

〈ユニット番号〉 : 0, 1

〈チャネル番号〉 : 0, 2

概要 ELCにより開始したTMRのカウント動作を一時停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_Timer_TMR_U〈ユニット番号〉.c

〈ユニット番号〉 : 0, 1

使用RPDL関数 R_TMR_ControlChannel

詳細

- ELCからのイベント信号受信により開始したTMRのカウント動作を停止します。
  再度イベント信号を受信することにより、カウント動作は再開します。

使用例 GUI上で以下の通り設定した場合

【TMR】

- ユニット0 : 8ビットタイマモード

【ELC】

- TMR0のイベント入力時動作 : カウントスタート

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //イベントリンクコントローラ(ELC)を設定
    R_PG_ELC_Set();

    //イベントリンクの設定（TMRチャネル0）
    R_PG_ELC_SetLink_TMR0();

    //TMRを設定
    R_PG_Timer_Start_TMR_U0_C0();

    //全イベントリンクの有効化
    R_PG_ELC_AllEventLinkEnable();
}

void func2(void)
{
    //ELCにより開始したTMRのカウントを一時停止
    R_PG_Timer_HaltCountElc_TMR_U0_C0();
}
```

5.15.8 R_PG_Timer_GetCountStateElc_TMR_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetCountStateElc_TMR_U〈ユニット番号〉_C〈チャネル番号〉
(bool * count_state)
〈ユニット番号〉 : 0, 1
〈チャネル番号〉 : 0, 2

概要

ELCによるタイマカウント状態を取得

引数

bool * count_state	タイマカウント状態の格納先 0: ELCによるカウント停止状態 1: ELCによるカウント開始状態

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_TMR_U〈ユニット番号〉.c
〈ユニット番号〉 : 0, 1

使用RPDL関数

R_TMR_ReadChannel

詳細

- ELCによるタイマカウント状態を取得します。

使用例

GUI上で以下の通り設定した場合

【TMR】

- ユニット0 : 8ビットタイマモード

【ELC】

- TMR0のイベント入力時動作 : カウントスタート

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

bool count_state=0;

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //イベントリンクコントローラ(ELC)を設定
    R_PG_ELC_Set();

    //イベントリンクの設定（TMRチャネル0)
    R_PG_ELC_SetLink_TMR0();

    //TMRを設定
    R_PG_Timer_Start_TMR_U0_C0();

    //全イベントリンクの有効化
    R_PG_ELC_AllEventLinkEnable();
}

void func2(void)
{
    //ELCによるタイマカウント状態を取得
    R_PG_Timer_GetCountStateElc_TMR_U0_C0(&count_state);
}
```

5.15.9 R_PG_Timer_StopModule_TMR_U〈ユニット番号〉

定義　bool R_PG_Timer_StopModule_TMR_U〈ユニット番号〉(void)

〈ユニット番号〉： 0, 1

概要　TMRのユニットを停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_TMR_U〈ユニット番号〉.c

〈ユニット番号〉： 0, 1

使用RPDL関数　R_TMR_Destroy

詳細

- TMRのユニットを停止し、モジュールストップ状態に移行します。ユニット単位で停止させます。ユニット0のTMR0とTMR1（ユニット1はTMR2とTMR3）が両方動作している場合、本関数を呼び出すとユニット内の2チャネルが停止します。片方のチャネルの動作だけを停止させる場合は、
  R_PG_Timer_HaltCount_TMR_U〈ユニット番号〉_C〈チャネル番号〉
  を使用してください。

使用例　GUI上でTMR0を8ビットタイマモードに設定
GUI上でコンペアマッチA割り込み関数名に TmrCma0IntFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //TMR0を8ビットモードで設定する
    R_PG_Timer_Start_TMR_U0_C0();
}

void TmrCma0IntFunc(void)
{
    func_cmA();     //コンペアマッチA割り込み発生時の処理

    //TMRユニット0を停止
    R_PG_Timer_StopModule_TMR_U0();
}
```

5.16 コンペアマッチタイマ（CMT）

5.16.1 R_PG_Timer_Set_CMT_U〈ユニット番号〉C〈チャネル番号〉

定義　bool R_PG_Timer_Set_CMT_U〈ユニット番号〉C〈チャネル番号〉(void)
〈ユニット番号〉： 0, 1
〈チャネル番号〉： 0～3

概要　CMTの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数　R_CMT_Create

詳細
- CMTのモジュールストップ状態を解除して、初期設定をします。
- R_PG_Timer_StartCount_CMT_U〈ユニット番号〉C〈チャネル番号〉によりカウント動作を開始します。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。
- 本関数内でCMTの割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。　void 〈割り込み通知関数名〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。

使用例　GUI上で以下の通り設定した場合

- CMT0を使用
- コンペアマッチ割り込みを使用
  割り込み要求先：CPUへ要求
  割り込み通知関数名：Cmt0IntFunc

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Set_CMT_U0_C0();   //CMT0を設定する
    R_PG_Timer_StartCount_CMT_U0_C0();   //CMT0のカウント動作を開始
}

//コンペアマッチ割り込み通知関数
void Cmt0IntFunc(void)
{
    R_PG_Timer_HaltCount_CMT_U0_C0();   //CMT0のカウント動作を一時停止
    func_cmt0();   //コンペアマッチ割り込み発生時の処理
    R_PG_Timer_StartCount_CMT_U0_C0();   //CMT0のカウント動作を再開
}
```

5.16.2 R_PG_Timer_StartCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉

定義　bool R_PG_Timer_StartCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉(void)
〈ユニット番号〉： 0, 1
〈チャネル番号〉： 0～3

概要　CMTのカウント動作を開始/再開

引数　なし

戻り値

true	カウント動作の開始/再開が正しく行われた場合
false	カウント動作の開始/再開に失敗した場合

出力先ファイル　R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数　R_CMT_Control

詳細
- CMTのカウント動作を開始します。
- R_PG_Timer_HaltCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉により一時停止したCMTのカウント動作を再開します。

使用例　R_PG_Timer_Set_CMT_U〈ユニット番号〉_C〈チャネル番号〉の使用例を参照してください。

5.16.3 R_PG_Timer_HaltCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_HaltCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉(void)
〈ユニット番号〉： 0, 1
〈チャネル番号〉： 0～3

概要 CMTのカウント動作を一時停止

引数 なし

戻り値

true	一時停止に成功した場合
false	一時停止に失敗した場合

出力先ファイル R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数 R_CMT_Control

詳細
- CMTのカウント動作を一時停止します。カウント動作を再開するには
R_PG_Timer_StartCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉
を呼び出してください。

使用例 R_PG_Timer_Set_CMT_U〈ユニット番号〉_C〈チャネル番号〉の使用例を参照してください。

5.16.4 R_PG_Timer_GetCounterValue_CMT_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_GetCounterValue_CMT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * counter_val)
〈ユニット番号〉 : 0, 1
〈チャネル番号〉 : 0～3

概要 CMTのカウンタ値を取得

引数

uint16_t * counter_val	カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉 : 0, 1

使用RPDL関数 R_CMT_Read

詳細
- CMTのカウンタ値を取得します。

使用例 GUI上で以下の通り設定した場合
- CMT0を使用

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter_val;

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Set_CMT_U0_C0();   //CMT0を設定する
    R_PG_Timer_StartCount_CMT_U0_C0();   //CMT0のカウント動作を開始
}

void func2(void)
{
    //CMT0のカウンタ値を取得
    R_PG_Timer_GetCounterValue_CMT_U0_C0( &counter_val );
}
```

5.16.5 R_PG_Timer_SetCounterValue_CMT_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_SetCounterValue_CMT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t counter_val)
〈ユニット番号〉 : 0, 1
〈チャネル番号〉 : 0～3

概要 CMTのカウンタ値を設定

引数

uint16_t counter_val	カウンタに設定する値

戻り値

true	カウンタ値の設定に成功した場合
false	カウンタ値の設定に失敗した場合

出力先ファイル R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉 : 0, 1

使用RPDL関数 R_CMT_Control

詳細
- CMTのカウンタ値を設定します。

使用例 GUI上で以下の通り設定した場合
- CMT0を使用

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Set_CMT_U0_C0();   //CMT0を設定する
    R_PG_Timer_StartCount_CMT_U0_C0();   //CMT0のカウント動作を開始
}

void func2(void)
{
    //CMT0のカウンタ値を設定
    R_PG_Timer_SetCounterValue_CMT_U0_C0( 0 );
}
```

5.16.6 R_PG_Timer_StopModule_CMT_U〈ユニット番号〉

定義　bool R_PG_Timer_StopModule_CMT_U〈ユニット番号〉(void)
〈ユニット番号〉： 0, 1

概要　CMTのユニットを停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数　R_CMT_Destroy

詳細
- CMTのユニットを停止し、モジュールストップ状態に移行します。ユニット単位で停止させます。ユニット0のCMT0とCMT1（ユニット1はCMT2とCMT3）が両方動作している場合、本関数を呼び出すとユニット内の2チャネルが停止します。片方のチャネルの動作だけを停止させる場合は、
R_PG_Timer_HaltCount_CMT_U〈ユニット番号〉C〈チャネル番号〉
を使用してください。

使用例　GUI上で以下の通り設定した場合
- CMT0を使用
- コンペアマッチ割り込みを使用
  割り込み要求先：CPUへ要求
  割り込み通知関数名：Cmt0IntFunc

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Set_CMT_U0_C0();   //CMT0を設定する
    R_PG_Timer_StartCount_CMT_U0_C0();   //CMT0のカウント動作を開始
}

//コンペアマッチ割り込み通知関数
void Cmt0IntFunc(void)
{
    func_cmt0();   //コンペアマッチ割り込み発生時の処理

    //CMTユニット0を停止
    R_PG_Timer_StopModule_CMT_U0();
}
```

5.17 リアルタイムクロック（RTCb）

5.17.1 R_PG_RTC_Start

定義　bool R_PG_RTC_Start (void)

概要　RTCを設定しカウント動作を開始

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_RTC.c

使用RPDL関数　R_RTC_Create

詳細

- アラーム割り込みと周期割り込み、RTCOUT端子からの1Hzクロック出力を設定し、カウント動作を開始します。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。
- 現在時刻は設定されません。アラーム割り込みを使用する場合、現在時刻は本関数を呼び出した後にR_PG_RTC_SetCurrentTimeにより設定してください。
- GUI上でアラーム日時を設定した場合はアラームレジスタが設定されます。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //ウォームスタートの判定
    If( SYSTEM.RSTSR1.BIT.CWSF == 0 ) {
        R_PG_RTC_Start(); //RTCの設定とカウント動作の開始
        SYSTEM.RSTSR1.BIT.CWSF = 1; //ウォームスタートフラグの設定
    }
    else {
        R_PG_RTC_WarmStart(); //RTCの再設定とカウント動作の開始
    }
}
```

5.17.1 R_PG_RTC_WarmStart

定義 bool R_PG_RTC_WarmStart (void)

概要 ウォームスタート時のRTCの再設定と開始

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_CreateWarm

詳細
- アラーム割り込みと周期割り込みを設定し、カウント動作を開始します。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。

使用例 R_PG_RTC_Startの使用例を参照してください。

5.17.2 R_PG_RTC_Stop

定義 bool R_PG_RTC_Stop (void)

概要 RTCのカウント動作を一時停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細
- RTCのカウント動作を停止します。
- カウント動作を再開するにはR_PG_RTC_Restartを呼び出してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定とカウント動作の開始
}

void func2(void)
{
    R_PG_RTC_Stop ();   //カウント動作の停止
}

void func3(void)
{
    R_PG_RTC_Restart();   //カウント動作の再開
}
```

5.17.3 R_PG_RTC_Restart

定義 bool R_PG_RTC_Restart (void)

概要 RTCのカウント動作を再開

引数 なし

戻り値

true	再開に成功した場合
false	再開に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細
- R_PG_RTC_Stopにより停止したRTCのカウント動作を再開します。

使用例 R_PG_RTC_Stopの使用例を参照してください。

5.17.4 R_PG_RTC_SetCurrentTime

定義

```
bool R_PG_RTC_SetCurrentTime
 (uint8_t seconds,    uint8_t minutes,    bool pm,    uint8_t hours,
  uint8_t day,        uint8_t month,      uint16_t year   )
```

概要

現在時刻の設定

引数

uint8_t seconds	秒（有効な値:0x00～0x59 (BCDコード))
uint8_t minutes	分（有効な値:0x00～0x59 (BCDコード))
bool pm	AM/PM 0 :午前 1 :午後
uint8_t hours	時間（有効な値:0x00～0x23 (BCDコード)(24時間モード時) 0x01～0x12 (BCDコード)(12時間モード時))
uint8_t day	日（有効な値:0x01～monthで指定される月の日数（BCDコード))
uint8_t month	月（有効な値:0x01～0x12 (BCDコード))
uint16_t year	年（有効な値: 0x0000～0x9999 (BCDコード))

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- 現在時刻を設定します。
- 曜日カウンタの値は指定された年月日から算出され、曜日カウンタに設定されます。
- カウント動作中に本関数を呼び出した場合、現在時刻のカウンタ設定中はカウント動作が停止し、設定後にカウントを再開します。
- アラーム割り込みで使用しない項目にも有効な範囲の値を設定してください。
- 24時間モード時はpmに0を設定してください。

使用例

GUI上で以下の通り設定した場合

- アラーム割り込みを設定
- RtcAlmIntFuncをアラーム割り込み通知関数名に指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定とカウント動作の開始

    R_PG_RTC_SetCurrentTime( //現在時刻の設定(2000年11月22日03時44分55秒)
        0x55, //55秒
        0x44, //44分
        0, //AM
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );

    R_PG_RTC_SetAlarmTime( //アラーム時刻の設定(2000年11月22日03時45分00秒)
        0x00, //00秒
        0x45, //45分
```

```
            0, //AM
            0x03, //03時
            0xff, //曜日(0xff : 指定した日時から自動計算する)
            0x22, //22日
            0x11, //11月
            0x2000 //2000年
        );

        R_PG_RTC_AlarmControl( //年、月、日、曜日、時、分、秒アラーム有効化
            1, //秒アラーム有効
            1, //分アラーム有効
            1, //時アラーム有効
            1, //曜日アラーム有効
            1, //日アラーム有効
            1, //月アラーム有効
            1 //年アラーム有効
        );
}

void RtcAlmIntFunc(void)
{
        //アラーム割り込み処理
}
```

5.17.5 R_PG_RTC_GetStatus

定義

```
bool R_PG_RTC_GetStatus
( bool * hour_mode24,   uint8_t * seconds,      uint8_t * minutes,   bool * pm,
  uint8_t * hours,      uint8_t * day_of_week,  uint8_t * day,       uint8_t * month,
  uint16_t * year,      bool * carry,           bool * alarm,        bool * period,
  bool * adjustment,    bool * reset,           bool * running   )
```

概要 RTCの状態を取得

引数

bool * hour_mode24	24時間モードの格納先 (0:12時間モード 1:24時間モード)
uint8_t * seconds	現在の秒カウンタ値の格納先
uint8_t * minutes	現在の分カウンタ値の格納先
bool * pm	AM/PMの格納先
uint8_t * hours	現在の時カウンタ値の格納先
uint8_t * day_of_week	現在の曜日カウンタ値の格納先
uint8_t * day	現在の日カウンタ値の格納先
uint8_t * month	現在の月カウンタ値の格納先
uint16_t * year	現在の年カウンタ値の格納先
bool * carry	桁上げ割り込みフラグの格納先
bool * alarm	アラーム割り込みフラグの格納先
bool * period	周期割り込みフラグの格納先
bool * adjustment	30秒調整ビットの格納先 (0:通常動作 1:調整中)
bool * reset	リセットビットの格納先 (0:通常動作 1:リセット中)
bool * running	スタートビットの格納先 (0:クロック停止 1:クロック動作中)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Read

詳細

- RTCの状態を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。
- 割り込みフラグは本関数内でクリアされます。
- 桁上げ割り込みフラグが1の場合、状態の取得中に現在時刻が変更されているため、再読み出しが必要です。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定とカウント動作の開始
}

void func2(void)
{
    do{
        //現在時刻と桁上げ割り込みフラグの取得
```

```
                R_PG_RTC_GetStatus(
                &hour_mode24,//24時間モード
                &seconds,   //秒
                &minutes,   //分
                &pm,        //AM/PM
                &hours,     //時
                0,          //曜日
                0,          //日
                0,          //月
                0,          //年
                &carry,     //桁上げ割り込みフラグ
                0,          //アラーム割り込みフラグ
                0,          //周期割り込みフラグ
                0,          //30秒調整ビット
                0,          //リセットビット
                0           //スタートビット
            );
    } while( carry );
}
```

5.17.6 R_PG_RTC_Adjust30sec

定義 bool R_PG_RTC_Adjust30sec (void)

概要 30秒調整を行う

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- 30秒調整（30秒未満は00秒に切り捨て、30秒以降は1分に桁上げ）を実行します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
	R_PG_Clock_Set(); //クロックの設定
	R_PG_RTC_Start(); //RTCの設定とカウント動作の開始
}

void func2(void)
{
	R_PG_RTC_Adjust30sec(); //30秒調整の実行
}
```

5.17.7 R_PG_RTC_ManualErrorAdjust

定義 bool R_PG_RTC_ManualErrorAdjust (int8_t cycle)

概要 時計誤差を補正

引数

int8_t cycle	時計誤差補正値(サブクロックのクロックサイクル数) -63 ～ -1 :時計を遅らせる 0 ～ 63 :時計を進める

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- サブクロックの発振精度による時計の誤差(遅れる/進む)を補正します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

int8_t cycle=-1;

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //RTCの設定と開始
    R_PG_RTC_Start();
}

void RtcPrdIntFunc(void)
{
    //時計誤差を補正
    R_PG_RTC_ManualErrorAdjust(cycle);
}
```

5.17.8 R_PG_RTC_Set24HourMode

定義 bool R_PG_RTC_Set24HourMode (void)

概要 RTCを24時間モードに設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- RTCを24時間モードに設定/変更します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //RTCの設定と開始
    R_PG_RTC_Start();

    //現在日時設定(2000年11月22日03時44分55秒)
    R_PG_RTC_SetCurrentTime(
        0x55, //55秒
        0x44, //44分
        0, //AM
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );

    //RTCを24時間モードに変更
    R_PG_RTC_Set24HourMode();
}
```

5.17.9 R_PG_RTC_Set12HourMode

定義 bool R_PG_RTC_Set12HourMode (void)

概要 RTCを12時間モードに設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- RTCを12時間モードに設定/変更します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //RTCの設定と開始
    R_PG_RTC_Start();

    //現在日時設定(2000年11月22日03時44分55秒)
    R_PG_RTC_SetCurrentTime(
        0x55, //55秒
        0x44, //44分
        0, //24時間モード
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );

    //RTCを12時間モードに変更
    R_PG_RTC_Set12HourMode();
}
```

5.17.10 R_PG_RTC_AutoErrorAdjust_Enable

定義　bool R_PG_RTC_AutoErrorAdjust_Enable (int8_t cycle)

概要　時計誤差自動補正有効化

生成条件　時計誤差補正機能が設定されている場合

引数

int8_t cycle	時計誤差補正値(サブクロックのクロックサイクル数) -63 ～ -1 :時計を遅らせる 0 ～ 63 :時計を進める

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_RTC.c

使用RPDL関数　R_RTC_Control

詳細

- 自動補正機能を有効にします。
- GUI上で選択した補正周期ごとに、サブクロックの発振精度による時計の誤差(遅れる/進む)を自動補正します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

int8_t cycle=-60;

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //RTCの設定と開始
    R_PG_RTC_Start();

    //時計誤差自動補正有効化
    R_PG_RTC_AutoErrorAdjust_Enable(cycle);

    //現在日時設定(2000年11月22日03時44分55秒)
    R_PG_RTC_SetCurrentTime(
        0x55, //55秒
        0x44, //44分
        0, //AM
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );
}
```

5.17.11 R_PG_RTC_AutoErrorAdjust_Disable

定義　bool R_PG_RTC_AutoErrorAdjust_Disable (void)

概要　時計誤差自動補正無効化

生成条件　時計誤差補正機能が設定されている場合

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_RTC.c

使用RPDL関数　R_RTC_Control

詳細

- 自動補正機能を無効にします。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

int8_t cycle=-60;

void func1(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //RTCの設定と開始
    R_PG_RTC_Start();

    //時計誤差自動補正有効化
    R_PG_RTC_AutoErrorAdjust_Enable(cycle);

    //現在日時設定(2000年11月22日03時44分55秒)
    R_PG_RTC_SetCurrentTime(
        0x55, //55秒
        0x44, //44分
        0, //AM
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );
}

void func2(void)
{
    //時計誤差自動補正無効化
    R_PG_RTC_AutoErrorAdjust_Disable();
}
```

5.17.12 R_PG_RTC_AlarmControl

定義 bool R_PG_RTC_AlarmControl
（ bool sec_enable, bool min_enable, bool hour_enable, bool day_of_week_enable,
bool day_enable, bool month_enable, bool year_enable ）

概要 アラームの有効化/無効化

生成条件 アラーム割り込みが設定されている場合

引数

bool sec_enable	秒アラーム設定（1:有効 0:無効）
bool min_enable	分アラーム設定（1:有効 0:無効）
bool hour_enable	時アラーム設定（1:有効 0:無効）
bool day_of_week_enable	曜日アラーム設定（1:有効 0:無効）
bool day_enable	日アラーム設定（1:有効 0:無効）
bool month_enable	月アラーム設定（1:有効 0:無効）
bool year_enable	年アラーム設定（1:有効 0:無効）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細
- 秒、分、時、曜日、日、月、年アラームの有効/無効を設定します。

使用例 R_PG_RTC_SetCurrentTimeの使用例を参照してください。

5.17.13 R_PG_RTC_SetAlarmTime

定義

bool R_PG_RTC_SetAlarmTime
（ uint8_t seconds, uint8_t minutes, bool pm, uint8_t hours,
uint8_t day_of_week, uint8_t day, uint8_t month, uint16_t year ）

概要

アラーム時刻の設定

生成条件

アラーム割り込みが設定されている場合

引数

uint8_t seconds	秒（有効な値:0x00～0x59 (BCDコード))
uint8_t minutes	分（有効な値:0x00～0x59 (BCDコード))
bool pm	AM/PM 0 :午前 1 :午後
uint8_t hours	時間（有効な値:0x00～0x23 (BCDコード)(24時間モード時) 0x01～0x12 (BCDコード)(12時間モード時))
uint8_t day_of_week	曜日（有効な値:0x00(日曜)～0x06(土曜)) 0xffを設定するとday,month,yearで指定した値から算出されます
uint8_t day	日（有効な値:0x01～monthで指定される月の日数（BCDコード))
uint8_t month	月（有効な値:0x01～0x12 (BCDコード))
uint16_t year	年（有効な値: 0x0000～0x9999 (BCDコード))

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_RTC.c

使用RPDL関数

R_RTC_Control

詳細

- アラーム時刻を設定します。
- アラームで使用しない項目にも有効な範囲の値を設定してください。
- 24時間モード時はpmに0を設定してください。

使用例

R_PG_RTC_SetCurrentTimeの使用例を参照してください。

5.17.14 R_PG_RTC_SetPeriodicInterrupt

定義 bool R_PG_RTC_SetPeriodicInterrupt (float frequency)

概要 周期割り込みの周期設定

生成条件 周期割り込みが設定されている場合

引数

float frequency	割り込みの周波数(Hz)(有効な値: 0.5, 1, 2, 4, 16, 64, 256)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細
- 周期割り込みの発生周期を変更します。

使用例

GUI上で以下の通り設定した場合
- 周期割り込みを設定
- RtcPrdIntFuncを周期割り込み通知関数名に指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定とカウント動作の開始
}

void RtcAlmIntFunc(void)
{
    //周期割り込みの処理

    R_PG_RTC_SetPeriodicInterrupt( 4 );   //周期割り込みの発生周期を1/4秒に設定
}
```

5.17.15 R_PG_RTC_ClockOut_Enable

定義　bool R_PG_RTC_ClockOut_Enable (void)

概要　クロック出力の有効化

生成条件　RTCOUTからの1Hzクロック出力が有効な場合

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_RTC.c

使用RPDL関数　R_RTC_Control

詳細

- RTCOUTからの1Hzクロック出力を開始します。

使用例

GUI上で以下の通り設定した場合

- RTCOUTからの1Hzクロック出力を有効に設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定、カウント動作とクロック出力の開始
}

void func2(void)
{
    R_PG_RTC_ClockOut_Disable();   //クロック出力の停止
}

void func3(void)
{
    R_PG_RTC_ClockOut_Enable();   //クロック出力の再開
}
```

5.17.16 R_PG_RTC_ClockOut_Disable

定義 bool R_PG_RTC_ClockOut_Disable (void)

概要 クロック出力の無効化

生成条件 RTCOUTからの1Hzクロック出力が有効な場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細
- RTCOUTからの1Hzクロック出力を停止します。

使用例 GUI上で以下の通り設定した場合
- RTCOUTからの1Hzクロック出力を有効に設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定、カウント動作とクロック出力の開始
}

void func2(void)
{
    R_PG_RTC_ClockOut_Disable();  //クロック出力の停止
}

void func3(void)
{
    R_PG_RTC_ClockOut_Enable();  //クロック出力の再開
}
```

5.17.17 R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉_Enable

定義 bool R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉_Enable (void)

〈*時間キャプチャイベント入力端子番号*〉: 0～2

概要 時間キャプチャ有効化

生成条件 時間キャプチャが設定されている場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- 時間キャプチャ機能を有効にします。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t  seconds,minutes,hours,day,month;
bool  pm,event;

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定

    //RTCの設定と開始
    R_PG_RTC_Start();

    //現在日時設定(2000年11月22日03時44分55秒)
    R_PG_RTC_SetCurrentTime(
        0x55, //55秒
        0x44, //44分
        0, //AM
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );

    //時間キャプチャ有効化
    R_PG_RTC_TimeCapture0_Enable();

    do{
        //キャプチャ時間取得
        R_PG_RTC_GetCaptureTime0(
            &seconds,
            &minutes,
            &pm,
            &hours,
            &day,
            &month,
            &event
        );
    }while(event == 0);
}
```

5.17.18 R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉_Disable

定義 bool R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉_Disable (void)

〈*時間キャプチャイベント入力端子番号*〉: 0～2

概要 時間キャプチャ無効化

生成条件 時間キャプチャが設定されている場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Control

詳細

- 時間キャプチャ機能を無効にします。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t  seconds,minutes,hours,day,month;
bool  pm,event;

void func(void)
{
    R_PG_Clock_Set(); //クロックの設定
    R_PG_RTC_Start(); //RTCの設定と開始
    R_PG_RTC_SetCurrentTime( //現在日時設定(2000年11月22日03時44分55秒)
        0x55, //55秒
        0x44, //44分
        0, //AM
        0x03, //03時
        0x22, //22日
        0x11, //11月
        0x2000 //2000年
    );
    R_PG_RTC_TimeCapture0_Enable(); //時間キャプチャ有効化

    do{
        R_PG_RTC_GetCaptureTime0( //キャプチャ時間取得
            &seconds,
            &minutes,
            &pm,
            &hours,
            &day,
            &month,
            &event
        );
    }while(event == 0);

    //時間キャプチャ無効化
    R_PG_RTC_TimeCapture0_Disable();
}
```

5.17.19 R_PG_RTC_GetCaptureTime〈*時間キャプチャイベント入力端子番号*〉

定義 bool R_PG_RTC_GetCaptureTime〈*時間キャプチャイベント入力端子番号*〉
（ uint8_t * seconds， uint8_t * minutes， bool * pm， uint8_t * hours，
uint8_t * day， uint8_t * month， bool * event ）
〈*時間キャプチャイベント入力端子番号*〉: 0～2

概要 キャプチャ時間取得

生成条件 時間キャプチャが設定されている場合

引数

uint8_t * seconds	現在の秒カウンタ値の格納先
uint8_t * minutes	現在の分カウンタ値の格納先
bool * pm	午後の格納先
uint8_t * hours	現在の時カウンタ値の格納先
uint8_t * day	現在の日カウンタ値の格納先
uint8_t * month	現在の月カウンタ値の格納先
bool * event	時間キャプチャステータスビットの格納先 （ 0:イベント検出なし 1:イベント検出あり ）

戻り値

true	取得が正しく行われた場合
false	取得に失敗した場合

出力先ファイル R_PG_RTC.c

使用RPDL関数 R_RTC_Read

詳細
- キャプチャ時間を取得します。

使用例 R_PG_RTC_TimeCapture〈*時間キャプチャイベント入力端子番号*〉_Enableの使用例を参照してください。

5.18 ウォッチドッグタイマ（WDTA）

5.18.1 R_PG_Timer_Start_WDT

定義　bool R_PG_Timer_Start_WDT (void)

概要　WDTを設定しカウント動作を開始

生成条件　レジスタスタートモードが選択されている場合
(オートスタートモードが選択されている場合は本関数は出力されず、 R_PG_MCU_OFS.c にオプション機能選択レジスタを設定するマクロが出力されます)

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_WDT.c

使用RPDL関数　R_WDT_Set

詳細
- WDTを初期設定し、カウント動作を開始します。

使用例　GUI上で以下の通り設定した場合
- スタートモード:レジスタスタートモード

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();    //クロックの設定
    R_PG_Timer_Start_WDT();    //WDTの設定とカウント動作の開始
}
```

5.18.2 R_PG_Timer_RefreshCounter_WDT

定義 bool R_PG_Timer_RefreshCounter_WDT(void)

概要 カウンタのリフレッシュ

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_WDT.c

使用RPDL関数 R_WDT_Control

詳細

- WDTのカウンタをリフレッシュします

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Start_WDT();   //WDTの設定とカウント動作の開始
}

void func2(void)
{
    R_PG_Timer_RefreshCounter_WDT();   //WDTのカウンタをリフレッシュ
}
```

5.18.3 R_PG_Timer_GetStatus_WDT

定義 bool R_PG_Timer_GetStatus_WDT(uint16_t * counter_val, bool * undf, bool * ref_err)

概要 WDTのステータスフラグとカウント値を取得

引数

uint16_t * counter_val	カウンタ値の格納先
bool * undf	アンダフローフラグの格納先
bool * ref_err	リフレッシュエラーフラグの格納先

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_WDT.c

使用RPDL関数 R_WDT_Read

詳細

- WDTのステータスフラグとカウント値を取得します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter_val;
bool undf;
bool ref_err;

void func(void)
{
    //WDTのステータスフラグとカウント値を取得
    R_PG_Timer_GetStatus_WDT(&counter_val, &undf, &ref_err);
}
```

5.19 独立ウォッチドッグタイマ (IWDTa)

5.19.1 R_PG_Timer_Start_IWDT

定義 bool R_PG_Timer_Start_IWDT (void)

概要 IWDTの設定と開始

生成条件 レジスタスタートモードが選択されている場合
(オートスタートモードが選択されている場合は本関数は出力されず、 R_PG_MCU_OFS.c にオプション機能選択レジスタを設定するマクロが出力されます)

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_IWDT.c

使用RPDL関数 R_IWDT_Set

詳細
- IWDTを設定し、カウント動作を開始します。
- 本関数を呼び出す前に R_PG_Clock_Set によりクロックを設定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //IWDTの設定と開始
    R_PG_Timer_Start_IWDT();
}
```

5.19.2 R_PG_Timer_RefreshCounter_IWDT

定義 bool R_PG_Timer_RefreshCounter_IWDT (void)

概要 カウンタのリフレッシュ

引数 なし

戻り値

true	リフレッシュに成功した場合
false	リフレッシュに失敗した場合

出力先ファイル R_PG_Timer_IWDT.c

使用RPDL関数 R_IWDT_Control

詳細

- IWDTのカウンタをリフレッシュします。
- カウント動作開始後、本関数によりアンダフロー発生までにカウンタをリフレッシュしてください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //IWDTの設定と開始
    R_PG_Timer_Start_IWDT();
}

void func2(void)
{
    //IWDTのカウンタをリフレッシュ
    R_PG_Timer_RefreshCounter_IWDT();
}
```

5.19.3 R_PG_Timer_GetStatus_IWDT

定義 bool R_PG_Timer_GetStatus_IWDT (uint16_t * counter_val, bool * undf, bool * ref_err)

概要 IWDTのステータスフラグとカウント値を取得

引数

uint16_t * counter_val	カウンタ値の格納先
bool * undf	アンダフローフラグの格納先
bool * ref_err	リフレッシュエラーフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_IWDT.c

使用RPDL関数 R_IWDT_Read

詳細

- IWDTのステータスフラグとカウント値を取得します。
- 本関数内でアンダフローフラグはクリアされます。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t counter_val;
bool undf;
bool ref_err;

void func(void)
{
    //IWDTのステータスフラグとカウント値を取得
    R_PG_Timer_GetStatus_IWDT(&counter_val, &undf, &ref_err);
}
```

5.20 シリアルコミュニケーションインタフェース（SCIc、SCId）

5.20.1 R_PG_SCI_Set_C〈*チャネル番号*〉

定義　bool R_PG_SCI_Set_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0～12

概要　シリアルI/Oチャネルの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Create, R_SCI_Set

詳細

- SCIチャネルのモジュールストップ状態を解除して初期設定します。
- 本関数を使用する場合、あらかじめR_PG_Clock_Setによりクロックを設定してください。
- GUI上で通知関数名を指定した場合、対応するイベントが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。

  void 〈*割り込み通知関数名*〉(void)

  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。

使用例　SCI0をGUI上で設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();       //クロックの設定
    R_PG_SCI_Set_C0();      //SCI0を設定
}
```

5.20.2 R_PG_SCI_SendTargetStationID_C〈*チャネル番号*〉

定義 bool R_PG_SCI_SendTargetStationID_C〈*チャネル番号*〉(uint8_t id)

〈*チャネル番号*〉: 0～12

概要 データ送信先IDの送信

生成条件
- GUI上でSCIチャネルの送信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定

引数

uint8_t id	送信するIDコード (0～255)

戻り値

true	送信に成功した場合
false	送信に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_Send

詳細
- マルチプロセッサモードのID送信サイクルを生成し、データ送信先の受信局IDコードを出力します。
- 本関数はID送信サイクル終了までウェイトします。

使用例 GUI上で以下の通り設定した場合
- SCI0チャネルの送信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定
- データ送信方法に“全データの送信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[] = "ABCDEFGHIJ";

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_SendTargetStationID_C0( 5 );      //IDコードの送信 (ID:5)
    R_PG_SCI_SendAllData_C0( data, 10 );       //データの送信
}
```

5.20.3 R_PG_SCI_StartSending_C〈*チャネル番号*〉

定義　bool R_PG_SCI_StartSending_C〈*チャネル番号*〉(uint8_t * data, uint16_t count)
〈*チャネル番号*〉: 0～12

概要　シリアルデータの送信開始

生成条件
- GUI上でSCIチャネルの送信機能を設定
- データ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択

引数

uint8_t * data	送信するデータの先頭のアドレス
uint16_t count	送信するデータ数 0を指定した場合はNULLのデータまで送信します。

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Send

詳細
- シリアルデータを送信します。
- 本関数はGUI上でデータ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合に出力されます。
- 本関数はすぐにリターンし、指定した数のデータ送信完了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈*通知関数名*〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- R_PG_SCI_GetSentDataCount_C〈*チャネル番号*〉により送信済みデータ数を取得することができます。R_PG_SCI_StopCommunication_C〈*チャネル番号*〉 により、最終バイトの送信完了を待たずに送信を中断することができます。
- 65536バイトのデータが送信されると、0番目のデータに戻ります。

使用例　GUI上でSCI0の送信終了通知関数名にSci0TrFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
uint8_t data[255];
void func(void)
{
    R_PG_Clock_Set();           //クロックの設定
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_StartSending_C0(data, 255);        //255バイトのデータを送信する
}
//全データが送信されると呼び出される送信終了通知関数
void Sci0TrFunc(void)
{
    //SCI0を停止
    R_PG_SCI_StopModule_C0();
}
```

5.20.4 R_PG_SCI_SendAllData_C〈*チャネル番号*〉

定義 bool R_PG_SCI_SendAllData_C〈*チャネル番号*〉(uint8_t * data, uint16_t count)

〈*チャネル番号*〉: 0～12

概要 シリアルデータを全て送信

生成条件
- GUI上でSCIチャネルの送信機能を設定
- データ送信方法に“全データの送信完了を関数呼び出しで通知する”以外を選択

引数

uint8_t * data	送信するデータの先頭のアドレス
uint16_t count	送信するデータ数 0を指定した場合はNULLのデータまで送信します。

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_Send

詳細
- シリアルデータを送信します。
- 本関数はGUI上でデータ送信方法に“全データの送信完了を関数呼び出しで通知する”以外が選択されている場合に出力されます。
- 指定した数のデータ送信完了まで関数内でウェイトします。
- 65536バイトのデータが送信されると、0番目のデータに戻ります。

使用例 GUI上でSCI0のデータ送信方法に“全データの送信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_SendAllData_C0(data, 255);          //255バイトのデータを送信する
    R_PG_SCI_StopModule_C0();          //SCI0を停止
}
```

5.20.5 R_PG_SCI_I2CMode_Send_C〈*チャネル番号*〉

定義 bool R_PG_SCI_I2CMode_Send_C〈*チャネル番号*〉
(bool addr_10bit, uint16_t slave, uint8_t * data, uint16_t count)
〈*チャネル番号*〉: 0～12

概要 簡易I²Cモードのデータ送信

生成条件
- モードに“簡易I²Cモード”を選択

引数

bool addr_10bit	スレーブアドレスフォーマット（1:10ビット　0:7ビット）
uint16_t slave	スレーブアドレス
uint8_t * data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	データ送信方法に[全データの送信完了まで待つ]を選択している時に動作が正常に完了した場合 (データ送信方法に[全データの送信完了まで待つ]以外を選択している時は必ずtrueが戻ります)
false	データ送信方法に[全データの送信完了まで待つ]を選択している時にエラーを検出した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_IIC_Write

詳細
- 簡易I²Cモードでデータを送信します。

使用例
GUI上で以下の通り設定した場合
[SCI0]
- モード：簡易I²Cモード
- データ送信方法：全データの送信完了を関数呼び出しで通知する

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_tr[]="ABCDEFGHIJ";
uint16_t  tr_count;

void func(void)
{
      R_PG_Clock_Set();   //クロックの設定

      R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

      //簡易I²Cモードのデータ送信
      R_PG_SCI_I2CMode_Send_C0(0, 0x0006, data_tr, 10);
}

void Sci0TrFunc(void)
{
      R_PG_SCI_GetSentDataCount_C0(&tr_count);   //シリアルデータの送信数取得
}
```

5.20.6 R_PG_SCI_I2CMode_SendWithoutStop_C〈チャネル番号〉

定義 bool R_PG_SCI_I2CMode_SendWithoutStop_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t * data, uint16_t count)
〈チャネル番号〉: 0～12

概要 簡易I²Cモードのデータ送信(STOP条件無し)

生成条件
- モードに“簡易I²Cモード”を選択

引数

bool addr_10bit	スレーブアドレスフォーマット (1:10ビット 0:7ビット)
uint16_t slave	スレーブアドレス
uint8_t * data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	データ送信方法に[全データの送信完了まで待つ]を選択している時に動作が正常に完了した場合 (データ送信方法に[全データの送信完了まで待つ]以外を選択している時は必ずtrueが戻ります)
false	データ送信方法に[全データの送信完了まで待つ]を選択している時にエラーを検出した場合

出力先ファイル R_PG_SCI_C〈チャネル番号〉.c
〈チャネル番号〉: 0～12

使用RPDL関数 R_SCI_IIC_Write

詳細
- 簡易I²Cモードでデータを送信します。(STOP条件無し)

使用例
GUI上で以下の通り設定した場合
[SCI0]
- モード:簡易I²Cモード
- データ送信方法:全データの送信完了を関数呼び出しで通知する

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_tr[10];
uint8_t data_re[10];

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

    //簡易I²Cモードのデータ送信(STOP条件無し)
    R_PG_SCI_I2CMode_SendWithoutStop_C0(0, 0x0006, data_tr, 10);
}

void Sci0TrFunc(void)
{
    //簡易I²Cモードのデータ受信(RE-START条件)
    R_PG_SCI_I2CMode_RestartReceive_C0(0, 0x0006, data_re, 10);
}
```

5.20.7 R_PG_SCI_I2CMode_GenerateStopCondition_C〈チャネル番号〉

定義 bool R_PG_SCI_I2CMode_GenerateStopCondition_C〈チャネル番号〉(void)

〈チャネル番号〉: 0～12

概要 STOP条件の生成

生成条件
- モードに“簡易I²Cモード”を選択

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_SCI_C〈チャネル番号〉.c

〈チャネル番号〉: 0～12

使用RPDL関数 R_SCI_Control

詳細
- STOP条件を生成します。

使用例 GUI上で以下の通り設定した場合

[SCI0]
- モード:簡易I²Cモード
- データ送信方法:DMACにより送信データを転送する

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_tr[]="ABCDEFGHIJ";

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    //DMACの設定
    R_PG_DMAC_Set_C0();

    //転送元アドレスの設定
    R_PG_DMAC_SetSrcAddress_C0(data_tr);

    //DMACをデータ転送開始トリガの入力待ち状態に設定
    R_PG_DMAC_Activate_C0();

    //シリアルI/Oチャネルの設定
    R_PG_SCI_Set_C0();

    //簡易I2Cモードのデータ送信
    R_PG_SCI_I2CMode_Send_C0(0, 0x0006, data_tr, 10);
}

void Dmac0IntFunc(void)
{
    //STOP条件の生成
    R_PG_SCI_I2CMode_GenerateStopCondition_C0();
}
```

5.20.8 R_PG_SCI_I2CMode_Receive_C〈*チャネル番号*〉

定義 bool R_PG_SCI_I2CMode_Receive_C〈*チャネル番号*〉
(bool addr_10bit, uint16_t slave, uint8_t * data, uint16_t count)
〈*チャネル番号*〉: 0～12

概要 簡易I²Cモードのデータ受信

生成条件
- モードに“簡易I²Cモード”を選択

引数

bool addr_10bit	スレーブアドレスフォーマット (1:10ビット 0:7ビット)
uint16_t slave	スレーブアドレス
uint8_t * data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	データ受信方法に[全データの受信完了まで待つ]を選択している時に動作が正常に完了した場合 (データ受信方法に[全データの受信完了まで待つ]以外を選択している時は必ずtrueが戻ります)
false	データ受信方法に[全データの受信完了まで待つ]を選択している時にエラーを検出した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_IIC_Read

詳細
- 簡易I²Cモードでデータを受信します。

使用例
GUI上で以下の通り設定した場合
[SCI0]
- モード:簡易I²Cモード
  機能:送信および受信

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_re[10];

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

    //簡易I²Cモードのデータ受信
    R_PG_SCI_I2CMode_Receive_C0(0, 0x0006, data_re, 10);
}
```

5.20.9 R_PG_SCI_I2CMode_RestartReceive_C〈*チャネル番号*〉

定義 bool R_PG_SCI_I2CMode_RestartReceive_C〈*チャネル番号*〉
(bool addr_10bit, uint16_t slave, uint8_t * data, uint16_t count)
〈*チャネル番号*〉: 0～12

概要 簡易I²Cモードのデータ受信(RE-START条件)

生成条件
- モードに“簡易I²Cモード”を選択

引数

bool addr_10bit	スレーブアドレスフォーマット（1:10ビット　0:7ビット）
uint16_t slave	スレーブアドレス
uint8_t * data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	データ受信方法に[全データの受信完了まで待つ]を選択している時に動作が正常に完了した場合 (データ受信方法に[全データの受信完了まで待つ]以外を選択している時は必ずtrueが戻ります)
false	データ受信方法に[全データの受信完了まで待つ]を選択している時にエラーを検出した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c 〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_IIC_Read

詳細
- 簡易I²Cモードでデータを受信します。(RE-START条件)

使用例
GUI上で以下の通り設定した場合
[SCI0]
- モード:簡易I²Cモード

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_re[10];

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

    //簡易I²Cモードのデータ送信(STOP条件無し)
    R_PG_SCI_I2CMode_SendWithoutStop_C0(
        1,                  //10bitアドレスフォーマット
        0x0006,             //スレーブアドレス
        PDL_NO_PTR,         //送信するデータの格納先の先頭のアドレス
        PDL_NO_DATA         //送信するデータ数
    );

    //簡易I²Cモードのデータ受信(RE-START条件)
    R_PG_SCI_I2CMode_RestartReceive_C0(
        0,                  //7bitアドレスフォーマット
        0x00f0,             //スレーブアドレス
        data_re,            //受信したデータの格納先の先頭のアドレス
        10                  //受信するデータ数
    );
}
```

5.20.10 R_PG_SCI_I2CMode_ReceiveLast_C〈*チャネル番号*〉

定義 bool R_PG_SCI_I2CMode_ReceiveLast_C〈*チャネル番号*〉(uint8_t * data)

〈*チャネル番号*〉: 0～12

概要 簡易I²Cモードの受信完了

生成条件
- モードに“簡易I²Cモード”を選択
- データ受信方法に“DMACにより受信データを転送する”または“DTCにより受信データを転送する”を選択

引数

uint8_t * data	受信したデータの格納先の先頭のアドレス

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_IIC_ReadLastByte

詳細
- 簡易I²CモードにてDMACまたはDTCにより受信データを転送する場合、転送完了後に本関数を呼び出すことにより受信を終了します。
- DMA割り込み通知関数または受信完了通知関数から本関数を呼び出してください。

使用例 GUI上で以下の通り設定した場合

[SCI0]
- モード:簡易I²Cモード
- データ受信方法:DMACにより受信データを転送する

[DMAC0]
- 転送開始要因:RXI0 (SCI0受信データフル割り込み)
- 転送モード:ノーマル転送モード
- 1データのビット長:1byte
- 転送回数:4
- 転送元開始アドレス:8a005h
- DMA割り込み(DMACIn)を使用する

[DMAC1]
- 転送開始要因:TXI0 (SCI0 送信データエンプティ割り込み)
- 転送モード:ノーマル転送モード
- 1データのビット長:1byte
- 転送回数:3
- 転送元アドレス更新モード:固定
- 転送先開始アドレス:8a003h

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_re[5];
uint8_t dummy_data=0xFF;

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定
```

```
    R_PG_DMAC_Set_C0();  //DMACの設定
    R_PG_DMAC_Set_C1();  //DMACの設定
    R_PG_DMAC_SetDestAddress_C0(data_re);  //転送先アドレスの設定
    R_PG_DMAC_SetSrcAddress_C1(&dummy_data);  //転送元アドレスの設定
    R_PG_DMAC_Activate_C0();  //DMACをデータ転送開始トリガ入力待ち状態に設定
    R_PG_DMAC_Activate_C1();  //DMACをデータ転送開始トリガ入力待ち状態に設定

    //簡易I2Cモードのデータ受信
    R_PG_SCI_I2CMode_Receive_C0(0, 0x0006, PDL_NO_PTR, 0);
}

void Dmac0IntFunc(void)
{
    //簡易I2Cモードの受信完了
    R_PG_SCI_I2CMode_ReceiveLast_C0(&data_re[4]);
}
```

5.20.11 R_PG_SCI_I2CMode_GetEvent_C〈*チャネル番号*〉

定義　bool R_PG_SCI_I2CMode_GetEvent_C〈*チャネル番号*〉(bool * nack)

〈*チャネル番号*〉: 0～12

概要　簡易I²Cモードの検出イベントの取得

生成条件　モードに“簡易I²Cモード”を選択

引数

bool * nack	NACK検出フラグ格納先 0:ACK受信 1:NACK受信

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_GetStatus

詳細

- 簡易I²CモードのACK受信データフラグを取得します。

使用例

[SCI0]

モード:簡易I²Cモード

データ送信方法:全データの送信完了を関数呼び出しで通知する

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_tr[]="ABCDEFGHIJ";
bool nack;

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

    //簡易I²Cモードのデータ送信
    R_PG_SCI_I2CMode_Send_C0(0, 0x0006, data_tr, 10);
}

void Sci0TrFunc(void)
{
    //簡易I²Cモードの検出イベントの取得
    R_PG_SCI_I2CMode_GetEvent_C0(&nack);
}
```

5.20.12 R_PG_SCI_SPIMode_Transfer_C〈*チャネル番号*〉

定義

bool R_PG_SCI_SPIMode_Transfer_C〈*チャネル番号*〉
(uint8_t * tx_start, uint8_t * rx_start, uint16_t count)
〈*チャネル番号*〉: 0～12

概要

簡易SPIモードのデータ転送

生成条件

- モードに“簡易SPIモード”を選択

引数

uint8_t * tx_start	送信するデータの先頭のアドレス
uint8_t * rx_start	受信したデータの格納先の先頭のアドレス
uint16_t count	転送するデータ数

戻り値

true	データ送信(受信)方法に[全データの送信(受信)完了まで待つ]を選択している時に動作が正常に完了した場合 (データ送信(受信)方法に[全データの送信(受信)完了まで待つ]以外を選択している時は必ずtrueが戻ります)
false	データ送信(受信)方法に[全データの送信(受信)完了まで待つ]を選択している時にエラーを検出した場合

出力先ファイル

R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～12

使用RPDL関数

R_SCI_SPI_Transfer

詳細

- 簡易SPIモードでデータを転送します。

使用例

GUI上で以下の通り設定した場合
[SCI0]
- モード:簡易SPIモード
- 機能:送信および受信

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_tr[10];
uint8_t data_re[10];

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定
}

void func2(void)
{
    //簡易SPIモードのデータ転送
    R_PG_SCI_SPIMode_Transfer_C0(data_tr, data_re, 10);
}
```

5.20.13 R_PG_SCI_SPIMode_GetErrorFlag_C〈チャネル番号〉

定義 bool R_PG_SCI_SPIMode_GetErrorFlag_C〈チャネル番号〉(bool * overrun)

〈チャネル番号〉: 0～12

概要 簡易SPIモードのシリアル受信エラーフラグの取得

生成条件 モードに“簡易SPIモード”を選択

引数

bool * overrun	オーバランエラーフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_SCI_C〈チャネル番号〉.c

〈チャネル番号〉: 0～12

使用RPDL関数 R_SCI_GetStatus

詳細
- 簡易SPIモードのシリアル受信エラーフラグを取得します。
- 取得しないフラグは0を設定してください。
- 検出したエラーのフラグには1が設定されます。

使用例

[SCI0]

モード:簡易SPIモード

機能:送信および受信

受信エラーの検出を関数呼び出しで通知する

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t tx_data[4];
uint8_t rx_data[4];
bool overrun;

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定
    R_PG_SCI_SPIMode_Transfer_C0(tx_data, rx_data, 4); //簡易SPIモードのデータ転送
}

void Sci0ErFunc(void)
{
    //簡易SPIモードのシリアル受信エラーフラグの取得
    R_PG_SCI_SPIMode_GetErrorFlag_C0(&overrun);
}
```

5.20.14 R_PG_SCI_GetSentDataCount_C〈*チャネル番号*〉

定義 bool R_PG_SCI_GetSentDataCount_C〈*チャネル番号*〉(uint16_t * count)

〈*チャネル番号*〉: 0～12

概要 シリアルデータの送信数取得

生成条件 GUI上でSCIチャネルの送信機能を設定し、データ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択

引数

uint16_t * count	現在の送信処理で送信されたデータ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_GetStatus

詳細

- GUI上でデータ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数により送信済みデータ数を取得することができます。

使用例 GUI上でSCI0の送信機能を設定

送信終了通知関数名にSci0TrFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_StartSending_C0(data, 255);          //255バイトのデータを送信する
}

//全データが送信されると呼び出される送信終了通知関数
void Sci0TrFunc(void)
{
    R_PG_SCI_StopModule_C0();      //SCI0を停止
}

//送信済みデータ数をチェックし、送信を中断する関数
void func_terminate_SCI(void)
{
    uint8_t count;
    R_PG_SCI_GetSentDataCount_C0(&count);   //送信済みデータ数を取得

    if( count > 32 ){
        R_PG_SCI_StopCommunication_C0();    //送信を中断
    }
}
```

5.20.15 R_PG_SCI_ReceiveStationID_C〈*チャネル番号*〉

定義　bool R_PG_SCI_ReceiveStationID_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0～12

概要　自局IDと一致するIDコードの受信

生成条件
- GUI上でSCIチャネルの受信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定

引数　なし

戻り値

true	受信に成功した場合
false	受信に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Receive

詳細
- 本関数は自局のIDと一致するIDコードを受信するまでウェイトします。

使用例　GUI上で以下の通り設定した場合
- SCI0チャネルの受信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定
- データ受信方法に“全データの受信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[10];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0の設定
    R_PG_SCI_ReceiveStationID_C0();    //IDの受信を待つ
    R_PG_SCI_ReceiveAllData_C0( data, 10 );    //受信開始
}
```

5.20.16 R_PG_SCI_StartReceiving_C〈*チャネル番号*〉

定義　bool R_PG_SCI_StartReceiving_C〈*チャネル番号*〉(uint8_t * data, uint16_t count)

〈*チャネル番号*〉: 0～12

概要　シリアルデータの受信開始

生成条件
- GUI上でSCIチャネルの受信機能を設定
- データ受信方法に“全データの受信完了を関数呼び出しで通知する”を選択

引数

uint8_t * data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Receive

詳細
- シリアルデータを受信します。
- 本関数はGUI上でデータ受信方法に“全データの受信完了を関数呼び出しで通知する”が選択されている場合に生成されます。
- 本関数はすぐにリターンし、指定した数のデータ受信完了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。

  void 〈*通知関数名*〉(void)

  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- R_PG_SCI_GetReceivedDataCount_C 〈*チャネル番号*〉により受信済みデータ数を取得することができます。R_PG_SCI_StopCommunication_C〈*チャネル番号*〉 により、最終バイトの受信完了を待たずに受信を中断することができます。
- 最大受信データ数は65535です。

使用例
・GUI上でSCI0の受信機能を設定
・受信終了通知関数名にSci0ReFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_StartReceiving_C0(data, 255);          //255バイトのデータを受信する
}

//全データを受信すると呼び出される受信終了通知関数
void Sci0ReFunc(void)
{
    //SCI0を停止
    R_PG_SCI_StopModule_C0();
}
```

5.20.17 R_PG_SCI_ReceiveAllData_C〈*チャネル番号*〉

定義　bool R_PG_SCI_ReceiveAllData_C〈*チャネル番号*〉(uint8_t * data, uint16_t count)

〈*チャネル番号*〉: 0～12

概要　シリアルデータを全て受信

生成条件
- GUI上でSCIチャネルの受信機能を設定
- データ受信方法に“全データの受信完了を関数呼び出しで通知する”以外を選択

引数

uint8_t * data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Receive

詳細
- シリアルデータを受信します。
- 本関数はGUI上でデータ受信方法に “全データの受信完了を関数呼び出しで通知する” 以外が選択されている場合に出力されます。
- 本関数は指定した数のデータ受信完了までウェイトします。
- 最大受信データ数は65535です。

使用例

GUI上でSCI0の受信機能を設定

データ受信方法に“全データの受信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();                   //クロックの設定
    R_PG_SCI_Set_C0();                  //SCI0を設定
    R_PG_SCI_ReceiveAllData_C0(data, 255);          //255バイトのデータを受信する
    R_PG_SCI_StopModule_C0();           //SCI0を停止
}
```

5.20.18 R_PG_SCI_ControlClockOutput_C〈*チャネル番号*〉

定義　bool R_PG_SCI_ControlClockOutput_C〈*チャネル番号*〉(bool output_enable)

〈*チャネル番号*〉: 0～12

概要　SCKn端子出力を切り替え　n: 0～12

生成条件
- モードに“スマートカードインタフェースモード”を選択
- GSMモードを有効に設定
- SCKn端子機能に“Lowレベル出力固定”または“Highレベル出力固定”を選択

引数

bool output_enable	SCKn端子の出力 (1:クロック出力 0:出力固定)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Control

詳細
- SCKn端子からのクロック出力を制御します。

使用例　GUI上で以下の通り設定した場合

[SCI0]
- モード:スマートカードインタフェースモード
- GSMモード:有効
- SCKn端子機能:Highレベル出力固定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

    //SCKn端子出力を切り替え
    R_PG_SCI_ControlClockOutput_C0( 1 );
}
```

5.20.19 R_PG_SCI_StopCommunication_C〈チャネル番号〉

定義　R_PG_SCI_StopCommunication_C〈チャネル番号〉(void)

〈チャネル番号〉: 0～12

概要　シリアルデータの送受信停止

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈チャネル番号〉.c

〈チャネル番号〉: 0～12

使用RPDL関数　R_SCI_Control

詳細

- シリアルの送受信を停止します。
- GUI上でデータ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数によりR_PG_SCI_StartSending_C〈チャネル番号〉で指定した全データの送信完了を待たずに送信を中断することができます。
- GUI上でデータ受信方法に“全データの受信完了を関数呼び出しで通知する”が選択されている場合、本関数によりR_PG_SCI_StartReceiving_C〈チャネル番号〉で指定した全データの受信完了を待たずに受信を中断することができます。

使用例　R_PG_SCI_GetSentDataCount_C〈チャネル番号〉の使用例を参照してください。

5.20.20 R_PG_SCI_GetReceivedDataCount_C〈*チャネル番号*〉

定義 bool R_PG_SCI_GetReceivedDataCount_C〈*チャネル番号*〉(uint16_t * count)

〈*チャネル番号*〉: 0～12

概要 シリアルデータの受信数取得

生成条件 GUI上でSCIチャネルの受信機能が設定され、データ受信方法に“全データの受信完了を関数呼び出しで通知する”

引数

uint16_t * count	現在の受信処理で受信したデータ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_GetStatus

詳細

- GUI上でデータ受信方法に“全データの受信完了を関数呼び出しで通知する”が選択されている場合、本関数により受信済みデータ数を取得することができます。

使用例 GUI上でSCI0の受信機能を設定

受信終了通知関数名にSci0ReFuncを指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_StartReceiving_C0(data, 255);          //255バイトのデータを受信する
}

//全データを受信すると呼び出される受信終了通知関数
void Sci0ReFunc(void)
{
    R_PG_SCI_StopModule_C0();      //SCI0を停止
}

//受信済みデータ数をチェックし、受信を中断する関数
void func_terminate_SCI(void)
{
    uint16_t count;
    R_PG_SCI_GetReceivedDataCount_C0(&count);   //受信済みデータ数を取得

    if( count > 32 ){
        R_PG_SCI_StopCommunication_C0();      //受信を中断
    }
}
```

5.20.21 R_PG_SCI_GetReceptionErrorFlag_C〈*チャネル番号*〉

定義 bool R_PG_SCI_GetReceptionErrorFlag_C〈*チャネル番号*〉
(bool * parity, bool * framing, bool * overrun)
〈*チャネル番号*〉: 0～12

概要 シリアル受信エラーフラグの取得

生成条件 GUI上でSCIチャネルの受信機能を設定

引数

bool * parity	パリティエラーフラグ格納先
bool * framing	フレーミングエラーフラグ格納先
bool * overrun	オーバランエラーフラグ格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～12

使用RPDL関数 R_SCI_GetStatus

詳細
- 受信エラーフラグを取得します。
- 取得しないフラグは0を設定してください。
- 検出したエラーのフラグには1が設定されます。

使用例 GUI上でSCI0の受信機能を設定
受信終了通知関数名にSci0ReFuncを指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

//受信エラーフラグ
bool parity;
bool framing;
bool overrun;

void func(void)
{
	R_PG_Clock_Set();		//クロックの設定
	R_PG_SCI_Set_C0();		//SCI0を設定
	R_PG_SCI_StartReceiving_C0(data, 1);		//1バイトのデータを受信する
}

//全データを受信すると呼び出される受信終了通知関数
void Sci0ReFunc(void)
{
	//受信エラーを取得
	R_PG_SCI_GetReceptionErrorFlag_C0( &parity, &framing, & overrun );
}
```

5.20.22 R_PG_SCI_ClearReceptionErrorFlag_C〈*チャネル番号*〉

定義　bool R_PG_SCI_ClearReceptionErrorFlag_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0～12

概要　シリアル受信エラーフラグのクリア

生成条件
- モードに“調歩同期式モード”、“クロック同期式モード”または“スマートカードインタフェースモード”を選択
- 機能に“受信”または“送信および受信”を選択

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Control

詳細
- シリアル受信エラーフラグをクリアします。

使用例　GUI上で以下の通り設定した場合

[SCI0]
- モード:調歩同期式モード
- 機能:受信
- データ受信方法:全データの受信完了を関数呼び出しで通知する

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data_re[10];
bool parity,framing,overrun;

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    R_PG_SCI_Set_C0();   //シリアルI/Oチャネルの設定

    //シリアルデータの受信開始
    R_PG_SCI_StartReceiving_C0(data_re, 10);
}

void Sci0ReFunc(void)
{
    //シリアル受信エラーフラグの取得
    R_PG_SCI_GetReceptionErrorFlag_C0(&parity, &framing, &overrun);

    //シリアル受信エラーフラグのクリア
    R_PG_SCI_ClearReceptionErrorFlag_C0();
}
```

5.20.23 R_PG_SCI_GetTransmitStatus_C〈*チャネル番号*〉

定義　bool R_PG_SCI_GetTransmitStatus_C〈*チャネル番号*〉(bool * complete)

〈*チャネル番号*〉: 0～12

概要　シリアルデータ送信状態の取得

生成条件　GUI上でSCIチャネルの送信機能を設定

引数

bool * complete	送信終了フラグ格納先 (0: 送信中　1: 送信終了)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_GetStatus

詳細

- シリアルデータの送信状態を取得します。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

bool complete;

void func(void)
{
    //送信状態の取得
    R_PG_SCI_GetTransmitStatus_C0( &complete );
}
```

5.20.24 R_PG_SCI_StopModule_C〈*チャネル番号*〉

定義　bool R_PG_SCI_StopModule_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0～12

概要　シリアルI/Oチャネルの停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～12

使用RPDL関数　R_SCI_Destroy

詳細

- SCIのチャネルを停止し、モジュールストップ状態に移行します。

使用例

GUI上で以下の通り設定した場合

- GUI上でSCI0の受信機能を設定
- データ受信方法に“全データの受信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_ReceiveAllData_C0(data, 255);          //255バイトのデータを受信する
    R_PG_SCI_StopModule_C0();          //SCI0を停止
}
```

5.21 I²Cバスインタフェース（RIIC）

5.21.1 R_PG_I2C_Set_C〈チャネル番号〉

定義　bool R_PG_I2C_Set_C〈チャネル番号〉(void)

〈チャネル番号〉: 0

概要　I²Cバスインタフェースチャネルの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数　R_IIC_Set, R_IIC_Create

詳細

- I²Cバスインタフェースチャネルのモジュールストップ状態を解除して初期設定します。
- 本関数を使用する場合、あらかじめR_PG_Clock_Setによりクロックを設定してください。

使用例　GUI上でRIIC0を設定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();      //クロックの設定
    R_PG_I2C_Set_C0();     //RIIC0を設定
}
```

5.21.2 R_PG_I2C_MasterReceive_C〈チャネル番号〉

定義 bool R_PG_I2C_MasterReceive_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0

概要 マスタのデータ受信

生成条件 マスタ機能を使用

引数

bool addr_10bit	スレーブアドレスフォーマット（1:10ビット　0:7ビット）
uint16_t slave	スレーブアドレス
uint8_t* data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_MasterReceive

詳細
- スレーブからデータを読み出します。指定した数のデータを受信するとSTOP条件を生成し転送を終了します。
- GUI上でマスタ受信方法に“全データの受信完了まで待つ”が選択されている場合、本関数は転送終了までウェイトします。GUI上でマスタ受信方法に“全データの受信完了を関数呼び出しで通知する”が選択されている場合、本関数はすぐにリターンし、転送終了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- 通信の最初にSTART条件が生成されます。前回の転送でSTOP条件が生成されていない場合は反復START条件が生成されます。
- スレーブアドレスは、7ビットアドレスの場合は指定した値の7～1ビットが出力されます。10ビットアドレスの場合は10～1ビットが出力されます。
- R_PG_I2C_GetReceivedDataCount_C〈チャネル番号〉により受信済みデータ数を取得することができます。
- 10ビットアドレスを使用する場合、GUI上のマスタ受信方法は“全データの受信完了を関数で通知する”以外を選択してください。

使用例 GUI上で以下の通り設定した場合
- RIIC0 をマスタとして使用
- マスタ受信方法に“全データの受信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

//受信データの格納先
uint8_t iic_data[10];

void func(void)
{
```

```
    //クロックの設定
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,       //受信データの格納先アドレス
        10      //受信データ数
    );

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.21.3 R_PG_I2C_MasterReceiveLast_C〈チャネル番号〉

定義 bool R_PG_I2C_MasterReceiveLast_C〈チャネル番号〉(uint8_t* data)

〈チャネル番号〉: 0

概要 マスタのデータ受信終了

生成条件
- マスタ機能を使用
- GUI上でマスタ受信方法にDMACまたはDTCによる転送を選択

引数

uint8_t* data	受信したデータの格納先のアドレス

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_IIC_MasterReceiveLast

詳細
- 本関数はGUI上で[マスタ受信方法]に[受信データをDMACで転送する]または[受信データをDTCで転送する] が選択されている場合に出力されます。
- マスタのデータ受信において、受信したデータをDMACまたはDTCで転送する場合、本関数を呼び出すことにより、NACKとストップ条件を発行して受信を終了します。
- DMACまたはDTCの転送終了時に受信を終了する場合は、DMACまたはDTCの転送終了割り込み通知関数から本関数を呼び出してください。
- 本関数内で、受信データレジスタから受信データを1バイト追加取得します。
- 受信中に検出したイベントや受信データ数は、R_PG_I2C_GetEvent_CnおよびR_PG_I2C_GetReceivedDataCount_Cnで取得することができます。

使用例 GUI上で以下の通り設定し、マスタが受信したデータをDMACで転送する場合
- RIIC0の設定でマスタ受信方法に[受信データをDMACで転送する]を指定。
- DMAC0の設定で以下通り設定。

  転送開始要因 ： ICRXI0(RIIC0受信データフル割り込み)

  転送方式 ： 単一オペランド転送

  単位データサイズ ： 1byte

  1オペランドのデータ数 ：1

  転送データサイズ ： RIIC0が受信するデータ数

  転送元スタートアドレス ： RIIC0受信データレジスタのアドレス

  転送先スタートアドレス ： RIIC0受信データの転送先開始アドレス

  DMAC0転送終了割り込み通知関数名 ： Dmac0IntFunc

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void Dmac0IntFunc(){
    uint8_t data; //追加データの格納先

    //NACK, STOP条件を発行し転送終了
    R_PG_I2C_MasterReceiveLast( &data );
}

void func(void)
{
    //クロックの設定
```

```
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //DMAC0を設定
    R_PG_DMAC_Set_C0();

    //DMAC0を転送開始トリガ入力待ち状態にする
    R_PG_DMAC_Activate_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        PDL_NO_PTR, //受信データ格納先（DMAC転送の場合はPDL_NO_PTR）
        0       //受信データ数（DMAC転送の場合は0）
    );
}
```

5.21.4 R_PG_I2C_MasterSend_C〈チャネル番号〉

定義 bool R_PG_I2C_MasterSend_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0

概要 マスタのデータ送信

生成条件 マスタ機能を使用

引数

bool addr_10bit	スレーブアドレスフォーマット（1:10ビット　0:7ビット）
uint16_t slave	スレーブアドレス
uint8_t* data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_MasterSend

詳細

- スレーブにデータを送信します。指定した数のデータを送信するとSTOP条件を生成し転送を終了します。
- GUI上でマスタ送信方法に“全データの送信完了まで待つ”が選択されている場合、本関数は転送終了または他のイベント検出までウェイトします。検出したイベントはR_PG_I2C_GetEvent_C〈チャネル番号〉により取得できます。GUI上でマスタ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数はすぐにリターンし、転送終了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- 通信の最初にSTART条件が生成されます。前回の転送でSTOP条件が生成されていない場合は反復START条件が生成されます。
- スレーブアドレスは、7ビットアドレスの場合は指定した値の7～1ビットが出力されます。10ビットアドレスの場合は10～1ビットが出力されます。
- R_PG_I2C_GetSentDataCount_C〈チャネル番号〉により送信済みデータ数を取得することができます。
- 10ビットアドレスを使用する場合、GUI上のマスタ送信方法は“全データの送信完了を関数で通知する”以外に設定してください。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
	//クロックの設定
	R_PG_Clock_Set();

	//RIIC0を設定
	R_PG_I2C_Set_C0();

	//マスタ送信
	R_PG_I2C_MasterSend_C0(
		0,	//スレーブアドレスフォーマット
		6,	//スレーブアドレス
		iic_data,	//送信データの格納先アドレス
		10	//送信データ数
	);

	//RIIC0を停止
	R_PG_I2C_StopModule_C0();
}
```

5.21.5 R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉

定義 R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0

概要 マスタのデータ送信（STOP条件無し）

生成条件 マスタ機能を使用

引数

bool addr_10bit	スレーブアドレスフォーマット（1:10ビット　0:7ビット）
uint16_t slave	スレーブアドレス
uint8_t* data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_MasterSend

詳細

- スレーブにデータを送信します。転送が終了してもSTOP条件を生成しません。本関数によるデータの送信後再び転送を開始した場合は反復START条件が生成されます。STOP条件を生成するにはR_PG_I2C_GenerateStopCondition_C〈チャネル番号〉を呼び出してください。
- GUI上でマスタ送信方法に“全データの送信完了まで待つ”が選択されている場合、本関数は転送終了または他のイベント検出までウェイトします。検出したイベントはR_PG_I2C_GetEvent_C〈チャネル番号〉により取得できます。GUI上でマスタ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数はすぐにリターンし、転送終了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- 通信の最初にSTART条件が生成されます。前回の転送でSTOP条件が生成されていない場合は反復START条件が生成されます。
- スレーブアドレスは、7ビットアドレスの場合は指定した値の7～1ビットが出力されます。10ビットアドレスの場合は10～1ビットが出力されます。
- R_PG_I2C_GetSentDataCount_C〈チャネル番号〉により送信済みデータ数を取得することができます。
- 10ビットアドレスを使用する場合、GUI上のマスタ送信方法は“全データの送信完了を関数で通知する”以外に設定してください。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択
- マスタ送信の通知関数名に IIC0MasterTrFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
	//クロックの設定
	R_PG_Clock_Set();

	//RIIC0を設定
	R_PG_I2C_Set_C0();

	//マスタ送信
	R_PG_I2C_MasterSendWithoutStop_C0(
		0,	//スレーブアドレスフォーマット
		6,	//スレーブアドレス
		iic_data,	//送信データの格納先アドレス
		10	//送信データ数
	);
}

void IIC0MasterTrFunc(void)
{
	//STOP条件を生成
	R_PG_I2C_GenerateStopCondition_C0();

	//RIIC0を停止
	R_PG_I2C_StopModule_C0();
}
```

5.21.6 R_PG_I2C_GenerateStopCondition_C〈*チャネル番号*〉

定義 R_PG_I2C_GenerateStopCondition_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0

概要 マスタのSTOP条件生成

生成条件 マスタ機能を使用

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_Control

詳細

- R_PG_I2C_MasterSendWithoutStop_C〈*チャネル番号*〉 により転送を開始した場合、STOP条件を生成することができます。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に “全データの送信完了を関数呼び出しで通知する” を選択
- マスタ送信の通知関数名に IIC0MasterTrFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSendWithoutStop_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,       //送信データの格納先アドレス
        10      //送信データ数
    );
}

void IIC0MasterTrFunc(void)
{
    //STOP条件を生成
    R_PG_I2C_GenerateStopCondition_C0();

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.21.7 R_PG_I2C_GetBusState_C〈*チャネル番号*〉

定義 R_PG_I2C_GetBusState_C〈*チャネル番号*〉(bool *busy)

〈*チャネル番号*〉: 0

概要 バス状態の取得

生成条件 マスタ機能を使用

引数

bool *busy	バスビジー検出フラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細

- バスビジー検出フラグを取得します。

バスビジー検出フラグ

0	バスが開放状態（バスフリー状態）
1	バスが占有状態（バスビジー状態またはバスフリーの期間中）

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
//送信データの格納先
uint8_t iic_data[10];
//バスビジー検出フラグの格納先
bool busy;
void func(void)
{
	//クロックの設定
	R_PG_Clock_Set();
	//RIIC0を設定
	R_PG_I2C_Set_C0();
	//バスフリー状態を待つ
	do{
		R_PG_I2C_GetBusState_C0( & busy );
	} while( busy );
	//マスタ送信
	R_PG_I2C_MasterSend_C0(
		0,	//スレーブアドレスフォーマット
		6,	//スレーブアドレス
		iic_data,	//送信データの格納先アドレス
		10	//送信データ数
	);
}
```

5.21.8 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉

定義 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉(uint8_t *data, uint16_t count)

〈チャネル番号〉: 0

概要 スレーブのバス監視

生成条件 スレーブ機能を使用

引数

uint8_t *data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_IIC_SlaveMonitor

詳細

- マスタからのアクセスを監視します。
- GUI上でスレーブモニタ方法に“全データの受信完了、スレーブリード要求、ストップ条件検出を関数呼び出しで通知する”が選択されている場合、マスタからの読み出し要求またはマスタからの受信後にSTOP条件を検出すると、指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。

  void 〈通知関数名〉(void)

  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- GUI上でスレーブモニタ方法に“全データの受信完了、スレーブリード要求、ストップ条件検出まで待つ”が選択されている場合、本関数はマスタからの読み出し要求またはマスタからの受信後にSTOP条件を検出するまでウェイトします。
- マスタからデータが送信された場合は指定した領域に受信データが格納されます。受信データ量が格納領域を上回らないよう、受信データ数設定してください。
  指定したデータ数を上回るデータがマスタから送信された場合はNACKを生成します。
- R_PG_I2C_GetTR_C〈チャネル番号〉 により送信/受信モードを取得することができます。マスタから送信(読み出し)が要求された場合、R_PG_I2C_SlaveSend_C〈チャネル番号〉 によりデータを送信できます。
- 検出したスレーブアドレスを取得するには R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉 を使用してください。START条件、STOP条件等の検出イベントを取得するには R_PG_I2C_GetEvent_C〈チャネル番号〉 を使用してください。
- 10ビットアドレスを使用する場合、GUI上のスレーブモニタ方法は “全データの受信完了、スレーブリード要求、ストップ条件検出を関数呼び出しで通知する” 以外に設定してください。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をスレーブとして使用
  スレーブモニタの通知関数名に IIC0SlaveFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
//受信データの格納先
```

```
uint8_t iic_data_re[10];
//送信データの格納先(スレーブアドレス0)
uint8_t iic_data_tr_0[10];
//送信データの格納先(スレーブアドレス1)
uint8_t iic_data_tr_1[10];
void func(void)
{
	//クロックの設定
	R_PG_Clock_Set();
	//RIIC0を設定
	R_PG_I2C_Set_C0();
	//スレーブモニタ
	R_PG_I2C_SlaveMonitor_C0(
		iic_data_re,	//受信データの格納先アドレス
		10	//受信データ数
	);
}
void IIC0SlaveFunc(void)
{
	bool transmit, start, stop;
	bool addr0, addr1;
	//イベントを取得する
	R_PG_I2C_GetEvent_C0(0, &stop, &start, 0, 0);
	//送受信モードを取得する
	R_PG_I2C_GetTR_C0(&transmit);
	//検出アドレスを取得する
	R_PG_I2C_GetDetectedAddress_C0(&addr0, &addr1, 0, 0, 0, 0);
	if(start && transmit && address0){
		R_PG_I2C_SlaveSend_C(
			iic_data_tr_0,
			10
		);
	}
	else if(start && read && address1){
		R_PG_I2C_SlaveSend_C(
			iic_data_tr_1,
			10
		);
	}
}
```

5.21.9 R_PG_I2C_SlaveSend_C〈チャネル番号〉

定義 R_PG_I2C_SlaveSend_C〈チャネル番号〉(uint8_t *data, uint16_t count)
〈チャネル番号〉: 0

概要 スレーブのデータ送信

生成条件 スレーブ機能を使用

引数

uint8_t *data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_SlaveSend

詳細
- マスタにデータを送信します。
- マスタが送信データ数を上回るデータを要求する場合、先頭のアドレスに戻って送信します。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉の使用例を参照してください。

5.21.10 R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉

定義 R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉
(bool *addr0, bool *addr1, bool *addr2, bool *general, bool *device, bool *host)
〈チャネル番号〉: 0

概要 検出したスレーブアドレスの取得

生成条件 スレーブ機能を使用

引数

bool *addr0	スレーブアドレス0検出フラグ格納先
bool *addr1	スレーブアドレス1検出フラグ格納先
bool *addr2	スレーブアドレス2検出フラグ格納先
bool *general	ジェネラルコールアドレス検出フラグ格納先
bool *device	デバイスID検出フラグ格納先
bool *host	ホストアドレス検出フラグ格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細
- 検出したアドレスを取得します。
- 取得しないフラグは0を設定してください。
- 検出したアドレスのフラグには1が設定されます。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉 の使用例を参照してください。

5.21.11 R_PG_I2C_GetTR_C〈チャネル番号〉

定義 R_PG_I2C_GetTR_C〈チャネル番号〉(bool * transmit)
〈チャネル番号〉: 0

概要 送信/受信モードの取得

生成条件 スレーブ機能を使用

引数

bool * transmit	送信/受信モードフラグの格納先 送信/受信モードフラグ 0:受信モード 1:送信モード

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細
- 送信/受信モードを取得します。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉 の使用例を参照してください。

5.21.12 R_PG_I2C_GetEvent_C〈*チャネル番号*〉

定義 R_PG_I2C_GetEvent_C〈*チャネル番号*〉
(bool *nack, bool *stop, bool *start, bool *lost, bool *timeout)
〈*チャネル番号*〉: 0

概要 検出イベントの取得

引数

bool *nack	NACK検出フラグ格納先
bool *stop	STOP条件検出フラグ格納先
bool *start	START条件検出フラグ格納先
bool *lost	アービトレーションロスト検出フラグ格納先
bool *timeout	タイムアウト検出フラグ格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細
- 検出したイベントを取得します。
- 取得しないフラグは0を設定してください。
- 検出したイベントのフラグには1が設定されます。

使用例 R_PG_I2C_SlaveMonitor_C〈*チャネル番号*〉 の使用例を参照してください。

5.21.13 R_PG_I2C_GetReceivedDataCount_C〈*チャネル番号*〉

定義 bool R_PG_I2C_GetReceivedDataCount_C〈*チャネル番号*〉(uint16_t *count)

〈*チャネル番号*〉: 0

概要 受信済みデータ数の取得

引数

uint16_t *count	受信データ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細

- 現在の転送で受信したデータ数を取得します。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ受信方法に“全データの受信完了を関数呼び出しで通知する”を選択

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//受信データの格納先
uint8_t iic_data[256];

//受信データ数の格納先
uint16_t count;

void func(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,       //受信データの格納先アドレス
        256     //受信データ数
    );

    //64バイト受信するまで待つ
    do{
        R_PG_I2C_GetReceivedDataCount_C0( &count );
    } while( count < 64 );
}
```

5.21.14 R_PG_I2C_GetSentDataCount_C〈*チャネル番号*〉

定義 bool R_PG_I2C_GetSentDataCount_C〈*チャネル番号*〉(uint16_t *count)

〈*チャネル番号*〉: 0

概要 送信済みデータ数の取得

引数

uint16_t *count	送信データ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細

- I²Cバス送信データレジスタに書き込んだデータ数を取得します。
- 送信関数に指定したデータ数分送信が完了している場合には 0 が取得されます。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

//送信データの格納先
uint8_t iic_data[256];

//送信データ数の格納先
uint16_t count;

void func(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSend_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,      //受信データの格納先アドレス
        256      //受信データ数
    );

    //64バイト送信するまで待つ
    do{
        R_PG_I2C_GetSentDataCount_C0( &count );
    } while( count < 64 );
}
```

5.21.15 R_PG_I2C_Reset_C〈チャネル番号〉

定義 R_PG_I2C_Reset_C〈チャネル番号〉(void)

〈チャネル番号〉: 0

概要 バスのリセット

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_IIC_Control

詳細

- モジュールをリセットします。
- 設定は維持されます。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択
- マスタ送信の通知関数名に IIC0MasterTrFunc を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
	//クロックの設定
	R_PG_Clock_Set();

	//RIIC0を設定
	R_PG_I2C_Set_C0();

	//マスタ送信
	R_PG_I2C_MasterSend_C0(
		0,	//スレーブアドレスフォーマット
		6,	//スレーブアドレス
		iic_data,	//送信データの格納先アドレス
		10	//送信データ数
	);
}

void IIC0MasterTrFunc(void)
{
	if( error ){
		R_PG_I2C_Reset_C0();
	}
}
```

5.21.16 R_PG_I2C_StopModule_C〈*チャネル番号*〉

定義 bool R_PG_I2C_StopModule_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0

概要 I²Cバスインタフェースチャネルの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_Destroy

詳細

- I²Cバスインタフェースチャネルを停止し、モジュールストップ状態に移行します。

使用例

GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ受信方法に“全データの受信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

//受信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //クロックの設定
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,      //受信データの格納先アドレス
        10      //受信データ数
    );

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.22 シリアルペリフェラルインタフェース（RSPI）

5.22.1 R_PG_RSPI_Set_C〈*チャネル番号*〉

定義　bool R_PG_RSPI_Set_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0

概要　RSPIチャネルの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_RSPI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数　R_SPI_Create

詳細

- シリアルペリフェラルインタフェースチャネルのモジュールストップ状態を解除して初期設定し、使用する端子を設定します。
- 本関数を使用する場合、あらかじめR_PG_Clock_Setによりクロックを設定してください。
- 本関数でコマンドは設定されません。コマンドを設定するには R_PG_RSPI_SetCommand_C〈*チャネル番号*〉を呼び出してください。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
      R_PG_Clock_Set();       //クロック発生回路の設定
      R_PG_RSPI_Set_C0();       //RSPI0の設定
      R_PG_RSPI_SetCommand_C0();       //コマンドの設定
}
```

5.22.2 R_PG_RSPI_SetCommand_C〈チャネル番号〉

定義 bool R_PG_RSPI_SetCommand_C〈チャネル番号〉(void)

〈チャネル番号〉: 0

概要 コマンドの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_SPI_Command

詳細
- RPSIコマンドレジスタを設定します。
- GUI上で設定した最大8コマンドを全て設定します。

使用例 R_PG_RSPI_Set_C〈チャネル番号〉の使用例を参照してください。

5.22.3 R_PG_RSPI_StartTransfer_C〈チャネル番号〉

定義

送信および受信機能(全二重同期式シリアル通信機能)選択時

bool R_PG_RSPI_StartTransfer_C〈チャネル番号〉
(uint32_t * tx_start, uint32_t * rx_start, uint16_t sequence_loop_count)
〈チャネル番号〉: 0

送信機能のみ選択時

bool R_PG_RSPI_StartTransfer_C〈チャネル番号〉
(uint32_t * tx_start, uint16_t sequence_loop_count)
〈チャネル番号〉: 0

概要

データの転送開始

生成条件

転送方法に“転送完了、エラー検出を関数呼び出しで通知する”を選択

引数

uint32_t * tx_start	送信するデータの先頭のアドレス
uint32_t * rx_start	受信したデータの格納先の先頭のアドレス
uint16_t sequence_loop_count	コマンドシーケンスの繰り返し回数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_RSPI_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数

R_SPI_Transfer

詳細

- データの転送を開始します。
- 本関数はGUI上で転送方法に“転送完了、エラー検出を関数呼び出しで通知する”が選択されている場合に出力されます。
- 本関数はすぐにリターンし、エラー検出時または指定した回数のコマンドシーケンス完了時に、指定した名前の通知関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。

使用例

GUI上で以下の通り設定した場合

- RSPIをSPI動作マスタモードで設定
- 転送方法に“転送完了、エラー検出を関数呼び出しで通知する”を指定
- 通知関数名にrsi0_int_funcを指定
- コマンド数:1 フレーム数:4
- コマンド0のビット長:8

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint32_t  tx_data[4] = { 0x11, 0x22, 0x33, 0x44 };
uint32_t  rx_data[4] = { 0x00, 0x00, 0x00, 0x00 };
bool   over_run,   mode_fault,   parity_error;

void func(void)
{
      R_PG_Clock_Set();       //クロック発生回路の設定
      R_PG_RSPI_Set_C0();       //RSPI0の設定
```

```
    R_PG_RSPI_SetCommand_C0();      //コマンドの設定
    R_PG_RSPI_StartTransfer_C0( tx_data, rx_data, 1 ); //8bit*4フレーム転送
}

void rsi0_int_func (void)
{
    R_PG_RSPI_GetError_C0 ( &over_run, &mode_fault, &parity_error );   //エラー取得
    if( over_run || mode_fault || parity_error ){
        //エラー検出時処理
    }
    R_PG_RSPI_StopModule_C0();
}
```

5.22.4 R_PG_RSPI_TransferAllData_C〈チャネル番号〉

定義

送信および受信機能(全二重同期式シリアル通信機能)選択時

bool R_PG_RSPI_TransferAllData_C〈チャネル番号〉
(uint32_t * tx_start, uint32_t * rx_start, uint16_t sequence_loop_count)
〈チャネル番号〉: 0

送信機能のみ選択時

bool R_PG_RSPI_TransferAllData_C〈チャネル番号〉
(uint32_t * tx_start, uint16_t sequence_loop_count)
〈チャネル番号〉: 0

転送方法にDTC/DMACによる転送を選択した場合

bool R_PG_RSPI_TransferAllData_C〈チャネル番号〉
(uint16_t sequence_loop_count)
〈チャネル番号〉: 0

概要 全データの転送

生成条件 転送方法に“転送完了、エラー検出を関数呼び出しで通知する”以外を選択

引数

uint32_t * tx_start	送信するデータの先頭のアドレス
uint32_t * rx_start	受信したデータの格納先の先頭のアドレス
uint16_t sequence_loop_count	コマンドシーケンスの繰り返し回数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_SPI_Transfer

詳細

- 全データを転送します。
- 本関数はGUI上で転送方法に“転送完了、エラー検出を関数呼び出しで通知する”以外が選択されている場合に出力されます。
- 本関数はエラー検出または指定した回数のコマンドシーケンス完了までウェイトします。

使用例

GUI上で以下の通り設定した場合

- RSPIをSPI動作マスタモードで設定
- 転送方法に“転送完了まで待つ”を指定
- 通知関数名にrsi0_int_funcを指定
- コマンド数:1 フレーム数:4
- コマンド0のビット長:8

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint32_t  tx_data[4] = { 0x11, 0x22, 0x33, 0x44 };
uint32_t  rx_data[4] = { 0x00, 0x00, 0x00, 0x00 };
bool   over_run,   mode_fault,   parity_error;

void func(void)
{
      R_PG_Clock_Set();       //クロック発生回路の設定
      R_PG_RSPI_Set_C0();       //RSPI0の設定
```

```
    R_PG_RSPI_SetCommand_C0();      //コマンドの設定
    R_PG_RSPI_TransferAllData_C0( tx_data, rx_data, 1 ); //8bit*4フレーム転送

    R_PG_RSPI_GetError_C0 ( &over_run, &mode_fault, &parity_error );   //エラー取得
    if( over_run || mode_fault || parity_error ){
        //エラー検出時処理
    }
    R_PG_RSPI_StopModule_C0();
}
```

5.22.5 R_PG_RSPI_GetStatus_C〈チャネル番号〉

定義 bool R_PG_RSPI_GetStatus_C〈チャネル番号〉(bool * idle)

〈チャネル番号〉: 0

概要 転送状態の取得

引数

bool * idle	アイドルフラグの格納先 (0:アイドル状態 1:転送状態)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_SPI_GetStatus

詳細

- データの転送状態を取得します。
- 本関数内でエラーフラグ(オーバランエラーフラグ、モードフォルトエラーフラグ、パリティエラーフラグ)はクリアされます。エラーフラグを取得する場合は本関数を呼び出す前にR_PG_RSPI_GetError_C〈チャネル番号〉を呼び出してください。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

bool idle;

void func(void)
{
	do{
		//アイドルフラグの取得
		R_PG_RSPI_GetStatus_C0( & idle );
	}while( idle );
}
```

5.22.6 R_PG_RSPI_GetError_C〈チャネル番号〉

定義 bool R_PG_RSPI_GetError_C〈チャネル番号〉
（bool * over_run, bool * mode_fault, bool * parity_error）
〈チャネル番号〉: 0

概要 エラー検出状態の取得

引数

bool * over_run	オーバランエラーフラグの格納先
bool * mode_fault	モードフォルトエラーフラグの格納先
bool * parity_error	パリティエラーフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_SPI_GetStatus

詳細
- エラーフラグを取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。
- 本関数内でエラーフラグはクリアされます。

使用例 R_PG_RSPI_StartTransfer_C〈チャネル番号〉 、 R_PG_RSPI_TransferAllData_C〈チャネル番号〉 およびR_PG_RSPI_GetCommandStatus_C〈チャネル番号〉 の使用例を参照してください。

5.22.7 R_PG_RSPI_GetCommandStatus_C〈チャネル番号〉

定義 bool R_PG_RSPI_GetCommandStatus_C〈チャネル番号〉
(uint8_t * current_command, uint8_t * error_command)
〈チャネル番号〉: 0

概要 コマンドステータスの取得

生成条件 RSPIチャネルをマスタモードに設定した場合

引数

uint8_t * current_command	現在のコマンドポインタ(0～7)の格納先
uint8_t * error_command	エラー検出時のコマンドポインタ(0～7)の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_SPI_GetStatus

詳細
- 現在のコマンドポインタ(0～7)と、エラー検出時のコマンドポインタ(0～7)を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。
- 本関数内でエラーフラグ(オーバランエラーフラグ、モードフォルトエラーフラグ、パリティエラーフラグ)はクリアされます。エラーフラグを取得する場合は本関数を呼び出す前にR_PG_RSPI_GetError_C〈チャネル番号〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- GUI上でRSPIをSPI動作マスタモードで設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool  over_run,  mode_fault,  parity_error;
uint8_t error_command;

void func(void)
{
    R_PG_RSPI_GetError_C0 ( &over_run, &mode_fault, &parity_error );   //エラー取得
    if( over_run || mode_fault || parity_error ){
        R_PG_RSPI_GetCommandStatus_C0( 0, &error_command );

        //エラー検出時処理
    }
}
```

5.22.8 R_PG_RSPI_LoopBack〈ループバックモード〉_C〈チャネル番号〉

定義 bool R_PG_RSPI_LoopBack〈ループバックモード〉_C〈チャネル番号〉(void)

〈ループバックモード〉: Direct, Reversed, Disable

〈チャネル番号〉: 0

概要 ループバックモードの設定

生成条件 ループバックモードが設定されている場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_SPI_Control

詳細

- 端子をループバックモードに設定または無効化します。
- R_PG_RSPI_LoopBackDirect_C〈チャネル番号〉 を呼び出すとシフトレジスタの入力経路と出力経路を接続します。(送信データ＝受信データ)
- R_PG_RSPI_LoopBackReversed_C〈チャネル番号〉 を呼び出すとシフトレジスタの入力経路と出力経路の反転を接続します。(送信データの反転＝受信データ)
- R_PG_RSPI_LoopBackDisable_C〈チャネル番号〉 を呼び出すとループバックモードを無効にします。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_RSPI_LoopBackDirect_C0();  //ループバックモードの設定
}
```

5.22.9 R_PG_RSPI_StopModule_C〈チャネル番号〉

定義 bool R_PG_RSPI_StopModule_C〈チャネル番号〉(void)

〈チャネル番号〉: 0

概要 RSPIチャネルの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_RSPI_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_SPI_Destroy

詳細

- RSPIチャネルを停止し、モジュールストップ状態に移行します。

使用例 R_PG_RSPI_StartTransfer_C〈チャネル番号〉およびR_PG_RSPI_TransferAllData_C〈チャネル番号〉の使用例を参照してください。

5.23 CRC演算器（CRC）

5.23.1 R_PG_CRC_Set

定義 bool R_PG_CRC_Set(void)

概要 CRC演算器の設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Create

詳細

- CRC演算器のモジュールストップ状態を解除して初期設定します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t data;

void func(void)
{
    R_PG_CRC_Set();   //CRC演算器の設定
    R_PG_CRC_InputData(0xf0);      //ペイロードデータ入力
    R_PG_CRC_InputData(0x8f);      //前半チェックサム入力
    R_PG_CRC_InputData(0xf7);      //後半チェックサム入力
    R_PG_CRC_GetResult (&data);   //演算結果取得
    R_PG_CRC_StopModule();   //CRC演算器停止
}
```

5.23.2 R_PG_CRC_InputData

定義 bool R_PG_CRC_InputData (uint8_t data)

概要 データの入力

引数

uint8_t data	入力するデータ

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Write

詳細
- CRCデータ入力レジスタにデータを設定します。

使用例 R_PG_CRC_Setの使用例を参照してください。

5.23.3 R_PG_CRC_GetResult

定義 bool R_PG_CRC_GetResult (uint16_t * result)

概要 演算結果の取得

引数

uint16_t * result	演算結果の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Read

詳細
- 演算結果を取得します。

使用例 R_PG_CRC_Setの使用例を参照してください。

5.23.4 R_PG_CRC_StopModule

定義 bool R_PG_CRC_StopModule(void)

概要 CRC演算器の停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Destroy

詳細
- CRC演算器を停止し、モジュールストップ状態に移行します。

使用例 R_PG_CRC_Setの使用例を参照してください。

5.24 12 ビットA/Dコンバータ（S12ADb）

5.24.1 R_PG_ADC_12_Set_S12AD0

定義　bool R_PG_ADC_12_Set_S12AD0 (void)

概要　12ビットA/Dコンバータの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ADC_12_S12AD0.c

使用RPDL関数　R_ADC_12_Set, R_ADC_12_CreateUnit, R_ADC_12_CreateChannel

詳細

- 12ビットA/Dコンバータのモジュールストップ状態を解除して初期設定し、変換開始トリガ入力待ち状態にします。変換開始トリガにソフトウェアを選択した場合は、R_PG_ADC_12_StartConversionSW_S12AD0により変換を開始します。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。
- 本関数内でA/D変換終了割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込み要求が発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。

  void *〈割り込み通知関数名〉*(void)

  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_ADC_12_Set_S12AD0();      //12ビットA/Dコンバータ(S12AD0)を設定
}
```

5.24.2 R_PG_ADC_12_StartConversionSW_S12AD0

定義 bool R_PG_ADC_12_StartConversionSW_S12AD0(void)

概要 A/D変換の開始 (ソフトウェアトリガ)

生成条件 シングルスキャンモード(ダブルトリガモードではない)、連続スキャンモードの場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_ADC_12_S12AD0.c

使用RPDL関数 R_ADC_12_Control

詳細
- 起動要因にソフトウェアトリガを選択したA/D変換器のA/D変換を開始します。

使用例

GUI上で以下の通り設定した場合
- 起動要因にソフトウェアトリガを選択

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();                   //クロックの設定
    R_PG_ADC_12_Set_S12AD0();       //12ビットA/Dコンバータ(S12AD0)を設定

    //ソフトウェアトリガによりAD変換開始
    R_PG_ADC_12_StartConversionSW_S12AD0();
}
```

5.24.3 R_PG_ADC_12_StopConversion_S12AD0

定義 bool R_PG_ADC_12_StopConversion_S12AD0(void)

概要 A/D変換の停止

引数 なし

戻り値

true	変換停止に成功した場合
false	変換停止に失敗した場合

出力先ファイル R_PG_ADC_12_S12AD0.c

使用RPDL関数 R_ADC_12_Control

詳細

- 本関数により連続スキャンモードのA/D変換を停止することができます。連続スキャンモード以外のモードではA/D変換完了後に本関数を呼び出す必要はありません。
- 本関数でA/D変換を停止させた後、A/D変換開始トリガを入力すると連続スキャンを再開します。連続スキャンを終了するにはR_PG_ADC_12_StopModule_S12AD0を呼び出し、A/D変換ユニットを停止状態にしてください。

使用例 GUI上で以下の通り設定した場合

- 動作モードを連続スキャンモードに設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set();               //クロックの設定
    R_PG_ADC_12_Set_S12AD0();       //12ビットA/Dコンバータ(S12AD0)を設定
}

void func2(void)
{
    //連続スキャンを停止
    R_PG_ADC_12_StopConversion_S12AD0();
}
```

5.24.4 R_PG_ADC_12_GetResult_S12AD0

定義　bool R_PG_ADC_12_GetResult_S12AD0(uint16_t * result)

概要　アナログ入力、温度センサ出力または内部基準電圧をA/D変換した結果の取得

引数

uint16_t * result	A/D変換結果の格納先

戻り値

true	結果の取得に成功した場合
false	結果の取得に失敗した場合

出力先ファイル　R_PG_ADC_12_S12AD0.c

使用RPDL関数　R_ADC_12_Read

詳細

- A/D変換結果の格納先は 2 * 16 バイト確保してください。
- GUI上で割り込み通知関数名を指定していない場合、本関数を呼び出した時点でA/D変換中であったときは、結果を読み出す前に変換が終了するまで本関数内で待ちます。

使用例　GUI上で以下の通り設定した場合

- グループスキャンモードを選択
  グループAのトリガ:TRG4AN，グループBのトリガ:TRG4BN
- アナログ入力端子は以下を選択
  グループA:AN000,AN015，グループB:AN003,AN006
- グループAのA/D変換終了割り込み通知関数名に S12ad0AIntFunc を設定
  グループBのA/D変換終了割り込み通知関数名に S12ad0BIntFunc を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();               //クロックの設定
    R_PG_ADC_12_Set_S12AD0();       //12ビットA/Dコンバータ(S12AD0)を設定
}

void S12ad0AIntFunc(void)   //グループAのA/D変換終了割り込み通知関数
{
    uint16_t result[16];       //AN000,AN015のA/D変換結果の格納先
    uint16_t result_an000;   //AN000のA/D変換結果の格納先
    uint16_t result_an015;   //AN015のA/D変換結果の格納先

    R_PG_ADC_12_GetResult_S12AD0( result );   //グループAのA/D変換結果の取得

    result_an000 = result[0];
    result_an015 = result[15];
}

void S12ad0BIntFunc(void)   //グループBのA/D変換終了割り込み通知関数
{
    uint16_t result[16];       //AN003,AN006のA/D変換結果の格納先
    uint16_t result_an003;   //AN003のA/D変換結果の格納先
    uint16_t result_an006;   //AN006のA/D変換結果の格納先

    R_PG_ADC_12_GetResult_S12AD0( result );   //グループBのA/D変換結果の取得

    result_an003 = result[3];
    result_an006 = result[6];
}
```

5.24.5 R_PG_ADC_12_GetResult_DblTrigger_S12AD0

定義 bool R_PG_ADC_12_GetResult_DblTrigger_S12AD0(uint16_t * result)

概要 ダブルトリガ(2回目)によるA/D変換結果の取得

生成条件 ダブルトリガモードの場合

引数

uint16_t * result	A/D変換結果の格納先

戻り値

true	結果の取得に成功した場合
false	結果の取得に失敗した場合

出力先ファイル R_PG_ADC_12_S12AD0.c

使用RPDL関数 R_ADC_12_Read

詳細

- ダブルトリガモード時の2回目のA/D変換結果を取得します。
- 取得するデータの数は、1チャネル分です。
- GUI上で割り込み通知関数名を指定していない場合、本関数を呼び出した時点でA/D変換中であったときは、結果を読み出す前に変換が終了するまで本関数内で待ちます。

使用例 GUI上で以下の通り設定した場合

- ダブルトリガモードを選択(トリガ:TRG4ABN)
- アナログ入力端子にAN003を指定
- A/D変換終了割り込み通知関数名に S12ad0AIntFunc を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();                //クロックの設定
    R_PG_ADC_12_Set_S12AD0();   //12ビットA/Dコンバータ(S12AD0)を設定
}

//A/D変換終了割り込み通知関数
void S12ad0AIntFunc(void)
{
    uint16_t result[16];          //AN003のA/D変換結果1の格納先
    uint16_t result_an003_2;   //AN003のA/D変換結果2の格納先

    //A/D変換結果1の取得
    R_PG_ADC_12_GetResult_S12AD0( result );

    //A/D変換結果2の取得
    R_PG_ADC_12_GetResult_DblTrigger_S12AD0( &result_an003_2 );
}
```

5.24.6 R_PG_ADC_12_GetResult_SelfDiag_S12AD0

定義 bool R_PG_ADC_12_GetResult_SelfDiag_S12AD0(uint16_t * result)

概要 A/Dコンバータの自己診断でA/D変換した結果の取得

引数

uint16_t * result	A/D変換結果の格納先

戻り値

true	結果の取得に成功した場合
false	結果の取得に失敗した場合

出力先ファイル R_PG_ADC_12_S12AD0.c

使用RPDL関数 R_ADC_12_Read

詳細

- 本関数内で、自己診断のA/D変換結果を取得します。
- 自己診断機能を使用する場合、自己診断はスキャンごとの最初に1回実施され、A/Dコンバータ内部で生成する3つの電圧値のうち1つをA/D変換します。
- 取得したA/D変換結果には自己診断ステータス情報(*1)が含まれます。
  データフォーマットは以下のようになります。

[GUI上でデータプレイスメントに右詰めを選択した場合]
b15-b14 :自己診断ステータス情報(*1)
b11-b0 :自己診断のA/D変換結果
[GUI上でデータプレイスメントに左詰めを選択した場合]
b15-b4 :自己診断のA/D変換結果
b1-b0 :自己診断ステータス情報(*1)

*1:自己診断ステータス情報
b'00 :一度も自己診断を実施していない
b'01 :0[V]の電圧値の自己診断を実施したことを示す
b'10 :VREFH0×1/2の電圧値の自己診断を実施したことを示す
b'11 :VREFH0の電圧値の自己診断を実施したことを示す

使用例 GUI上で以下の通り設定した場合

- シングルスキャンモードを選択
- アナログ入力端子にAN000とAN008を指定
- 起動要因にソフトウェアトリガを選択
- データプレイスメントに右詰めを選択
- 自己診断機能を有効に設定
- A/D変換終了割り込み通知関数名に S12ad0AIntFunc を設定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t result_selfdiag;   //自己診断A/D変換結果の格納先
uint16_t adrd_ad;           //12ビットA/D変換値の格納先
uint16_t adrd_diagst;       //自己診断ステータス情報の格納先
uint16_t result[16];        //AN000,AN008のA/D変換結果の格納先
uint16_t result_an000;      //AN000のA/D変換結果の格納先
uint16_t result_an008;      //AN008のA/D変換結果の格納先

void func(void)
{
    R_PG_Clock_Set();               //クロックの設定
```

```
    R_PG_ADC_12_Set_S12AD0();   //12ビットA/Dコンバータ(S12AD0)を設定

    //ソフトウェアトリガによりAD変換開始
    R_PG_ADC_12_StartConversionSW_S12AD0();
}

//A/D変換終了割り込み通知関数
void S12ad0AIntFunc(void)
{
    //自己診断A/D変換結果の取得
    R_PG_ADC_12_GetResult_SelfDiag_S12AD0( &result_selfdiag );

    adrd_ad = (result_selfdiag & 0x0fff);
    adrd_diagst = (result_selfdiag >> 14);

    //AN000,AN008のA/D変換結果の取得
    R_PG_ADC_12_GetResult_S12AD0( result );

    result_an000 = result[0];
    result_an008 = result[8];
}
```

5.24.7 R_PG_ADC_12_StopModule_S12AD0

定義 bool R_PG_ADC_12_StopModule_S12AD0(void)

概要 12ビットA/Dコンバータの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_ADC_12_S12AD0.c

使用RPDL関数 R_ADC_12_Destroy

詳細

- 12ビットA/Dコンバータを停止し、モジュールストップ状態に移行します。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t result[16];  //A/D変換結果の格納先

void func1(void)
{
    R_PG_Clock_Set();                //クロックの設定
    R_PG_ADC_12_Set_S12AD0();        //12ビットA/Dコンバータ(S12AD0)を設定
}

void func2(void)
{
    //連続スキャンを停止
    R_PG_ADC_12_StopConversion_S12AD0();

    //A/D変換結果の取得
    R_PG_ADC_12_GetResult_S12AD0( result );

    //12ビットA/Dコンバータ(S12AD0)を停止
    R_PG_ADC_12_StopModule_S12AD0();
}
```

5.25 D/A コンバータ (DA)

5.25.1 R_PG_DAC_Set_C〈*チャネル番号*〉

定義 bool R_PG_DAC_Set_C〈*チャネル番号*〉(void) 〈*チャネル番号*〉: 0, 1

概要 D/Aコンバータのチャネルを設定

生成条件 どちらか一方のチャネルのみ使用する場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DAC_C〈*チャネル番号*〉.c 〈*チャネル番号*〉: 0, 1

使用RPDL関数 R_DAC_10_Create

詳細

- D/Aコンバータのチャネルを設定します。
- 本関数を呼び出した場合、他方のチャネルは無効となります。
- D/Aコンバータのモジュールストップ状態が解除されます。
- アナログ出力端子からは、モジュールストップ状態解除後のD/Aデータレジスタの初期値(0)の変換結果が出力されます。初期値を指定して出力を開始する場合はR_DAC_SetWithInitialValue_C〈*チャネル番号*〉を使用してください。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //DA0端子を設定
    R_PG_DAC_Set_C0();
}

void func2( uint16_t output_val )
{
    //D/A変換値の変更
    R_PG_DAC_ControlOutput_C0( output_val );
}
```

5.25.2 R_PG_DAC_SetWithInitialValue_C〈*チャネル番号*〉

定義 bool R_PG_DAC_SetWithInitialValue_C〈*チャネル番号*〉(uint16_t initial_val)

〈*チャネル番号*〉: 0, 1

概要 初期値を指定してD/Aコンバータのチャネルを設定

引数

uint16_t initial_val	D/A変換値の初期値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DAC_C〈*チャネル番号*〉.c 〈*チャネル番号*〉: 0, 1

使用RPDL関数 R_DAC_10_Create

詳細

- D/A変換値の初期値を指定してD/Aコンバータのチャネルを設定し、出力を開始します。
- D/Aコンバータのモジュールストップ状態が解除されます。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(uint16_t initial_val)
{
    //DA0端子を設定し出力を開始
    R_PG_DAC_SetWithInitialValue_C0( initial_val );
}

void func2( uint16_t output_val )
{
    //D/A変換値の変更
    R_PG_DAC_ControlOutput_C0( output_val );
}
```

5.25.3 R_PG_DAC_StopOutput_C〈チャネル番号〉

定義 bool R_PG_DAC_StopOutput_C〈チャネル番号〉(void)

〈チャネル番号〉: 0, 1

概要 アナログ出力の停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_DAC_C〈チャネル番号〉.c 〈チャネル番号〉: 0, 1

使用RPDL関数 R_DAC_10_Destroy

詳細

- アナログ出力を停止します。
- 全チャネルの出力が停止する場合、D/Aコンバータはモジュールストップ状態に移行します。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(uint16_t initial_val)
{
    //DA0端子を設定し出力を開始
    R_PG_DAC_SetWithInitialValue_C0( initial_val );
}

void func2()
{
    //アナログ出力の停止
    R_PG_DAC_StopOutput_C0();
}
```

5.25.4 R_PG_DAC_ControlOutput_C〈チャネル番号〉

定義 bool R_PG_DAC_ControlOutput_C〈チャネル番号〉(uint16_t output_val)
〈チャネル番号〉: 0, 1

概要 D/A変換値の設定

引数

uint16_t output_val	D/Aデータレジスタに設定するD/A変換値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DAC_C〈チャネル番号〉.c 〈チャネル番号〉: 0, 1

使用RPDL関数 R_DAC_10_Write

詳細
- D/Aデータレジスタに D/A変換値を設定します。

使用例 R_PG_DAC_Set_C〈チャネル番号〉の使用例を参照してください。

5.25.5 R_PG_DAC_Set_C0_C1

定義 bool R_PG_DAC_Set_C0_C1 (void)

概要 D/Aコンバータのチャネルを設定 (DA0, DA1一括設定)

生成条件 DA0, DA1を両方使用する場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DAC.c

使用RPDL関数 R_DAC_10_Create

詳細

- D/Aコンバータのチャネルを一括設定します。
- D/Aコンバータのモジュールストップ状態が解除されます。
- アナログ出力端子からは、モジュールストップ状態解除後のD/Aデータレジスタの初期値(0)の変換結果が出力されます。初期値を指定して出力を開始する場合は R_DAC_SetWithInitialValue_C*<チャネル番号>*を使用してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    //DA0, DA1端子を一括設定
    R_PG_DAC_Set_C0_C1();
}

void func2( uint16_t output_val_c0, uint16_t output_val_c1 )
{
    //D/A変換値の変更
    R_PG_DAC_ControlOutput_C0( output_val_c0 );
    R_PG_DAC_ControlOutput_C1( output_val_c1 );
}
```

5.26 温度センサ（TEMPSa）

5.26.1 R_PG_TS_Set

定義　bool R_PG_TS_Set (void)

概要　温度センサの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_TS.c

使用RPDL関数　R_TS_Create

詳細

- 温度センサのモジュールストップ状態を解除して初期設定します。

使用例

```
//この関数を使用するには"R_PG_<PDGプロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void func1(void)
{
    //クロック発生回路を設定し、クロックの発振を開始
    R_PG_Clock_Set();
    //12ビットA/Dコンバータの設定
    R_PG_ADC_12_Set_S12AD0();
    //温度センサの設定
    R_PG_TS_Set();
    //温度センサPGA動作開始
    R_PG_TS_StartPGA();
}
void func2(void)
{
    //温度センサPGA動作停止
    R_PG_TS_StopPGA();
}
void func3(void)
{
    //温度センサの停止
    R_PG_TS_StopModule ();
}
```

5.26.2 R_PG_TS_StartPGA

定義 bool R_PG_TS_StartPGA (void)

概要 温度センサのPGA動作開始

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_TS.c

使用RPDL関数 R_TS_Control

詳細
- 温度センサのPGAの動作を開始し、温度センサから12ビットA/Dコンバータへの出力を開始します。

使用例 R_PG_TS_Setの使用例を参照してください。

5.26.3 R_PG_TS_StopPGA

定義 bool R_PG_TS_StopPGA (void)

概要 温度センサのPGA動作停止

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_TS.c

使用RPDL関数 R_TS_Control

詳細
- 温度センサのPGAの動作を停止し、温度センサから12ビットA/Dコンバータへの出力を停止します。

使用例 R_PG_TS_Setの使用例を参照してください。

5.26.4 R_PG_TS_StopModule

定義 bool R_PG_TS_StopModule (void)

概要 温度センサの停止

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_TS.c

使用RPDL関数 R_TS_Destroy

詳細
- 温度センサから12ビットA/Dコンバータへの出力を禁止します。
- 温度センサを停止し、モジュールストップ状態に移行します。

使用例 R_PG_TS_Setの使用例を参照してください。

5.27 コンパレータA (CMPA)

5.27.1 R_PG_CPA_Set_CP〈*コンパレータ回路番号*〉

定義 bool R_PG_CPA_Set_CP〈*コンパレータ回路番号*〉(void)
〈*コンパレータ回路番号*〉: A1, A2

概要 コンパレータnの設定 n: A1, A2

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CPn.c n: A1, A2

使用RPDL関数 R_CPA_Create

詳細
- コンパレータnの初期設定をします。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。

使用例

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    //コンパレータA1の設定
    R_PG_CPA_Set_CPA1();
}
```

5.27.2 R_PG_CPA_Disable_CP〈*コンパレータ回路番号*〉

定義 bool R_PG_CPA_Disable_CP〈*コンパレータ回路番号*〉(void)

〈*コンパレータ回路番号*〉: A1, A2

概要 コンパレータn回路の無効化 n: A1, A2

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CPn.c n: A1, A2

使用RPDL関数 R_CPA_Control

詳細

- コンパレータn回路を無効にします。

使用例

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定

    //コンパレータA1の設定
    R_PG_CPA_Set_CPA1();
}

void func2(void)
{
    //コンパレータA1回路の無効化
    R_PG_CPA_Disable_CPA1();
}
```

5.27.3 R_PG_CPA_GetStatus

定義 bool R_PG_CPA_GetStatus
（ bool * cpa1_detect, bool * cpa1_monitor, bool * cpa2_detect, bool * cpa2_monitor ）

概要 コンパレータAのステータスフラグを取得

引数

bool * cpa1_detect	コンパレータA1電圧変化検出フラグの格納先
bool * cpa1_monitor	コンパレータA1信号モニタフラグの格納先
bool * cpa2_detect	コンパレータA2電圧変化検出フラグの格納先
bool * cpa2_monitor	コンパレータA2信号モニタフラグの格納先

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_CPA.c

使用RPDL関数 R_CPA_GetStatus

詳細
- コンパレータAのステータスフラグを取得します。
- 取得しないフラグには0を指定してください。

使用例 GUI上で以下の通り設定した場合
- コンパレータA1を使用
- コンパレータAnモード： CMPAnがCVREFAを通過時にコンパレータAn割り込み
- コンパレータAn割り込み種類： マスカブル割り込み

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

bool cpa1_mon;

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定

    //コンパレータA1の設定
    R_PG_CPA_Set_CPA1();
}

void CMPA1IntFunc(void)
{
    //コンパレータAのステータスフラグを取得
    R_PG_CPA_GetStatus(0, &cpa1_mon, 0, 0);
}
```

5.28 コンパレータB (CMPB)

5.28.1 R_PG_CPB_Set_CPB〈*チャネル番号*〉

定義 bool R_PG_CPB_Set_CPB〈*チャネル番号*〉(void) 〈*チャネル番号*〉: 0, 1

概要 コンパレータBのチャネルを設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CPB〈*チャネル番号*〉.c 〈*チャネル番号*〉: 0, 1

使用RPDL関数 R_CPB_Create

詳細
- コンパレータBのモジュールストップ状態を解除して初期設定します。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。

使用例

GUI上で以下の通り設定した場合
- コンパレータB0を使用
- デジタルフィルタサンプリングクロックに内部クロック(PCLK)を選択
- 割り込み通知関数名に CMPB0IntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
bool status;
void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    //コンパレータB0の設定
    R_PG_CPB_Set_CPB0();
}
//割り込み通知関数
void CMPB0IntFunc (void)
{
    //コンパレータB0のモニタフラグ値を取得
    R_PG_CPB_GetStatusFlag_CPB0( &status );
    //コンパレータB0回路を無効にする
    R_PG_CPB_StopModule_CPB0();
}
```

5.28.2 R_PG_CPB_GetStatusFlag_CPB〈*チャネル番号*〉

定義 bool R_PG_CPB_GetStatusFlag_CPB〈*チャネル番号*〉(bool * status)
〈*チャネル番号*〉: 0, 1

概要 モニタフラグ値を取得

引数

bool * status	モニタフラグ値格納先 モニタフラグ値 0:アナログ入力電圧 < リファレンス入力電圧 1:アナログ入力電圧 > リファレンス入力電圧

戻り値

true	取得が正しく行われた場合
false	取得に失敗した場合

出力先ファイル R_PG_CPB〈*チャネル番号*〉.c 〈*チャネル番号*〉: 0, 1

使用RPDL関数 R_CPB_GetStatus

詳細
- コンパレータBnのモニタフラグ値を取得します。(n: 0, 1)

使用例 R_PG_CPB_Set_CPB〈*チャネル番号*〉の使用例を参照してください。

5.28.3 R_PG_CPB_StopModule_CPB〈*チャネル番号*〉

定義 bool R_PG_CPB_StopModule_CPB〈*チャネル番号*〉(void) 〈*チャネル番号*〉: 0, 1

概要 コンパレータBのチャネルを停止

引数 なし

戻り値

true	停止が正しく行われた場合
false	停止に失敗した場合

出力先ファイル R_PG_CPB〈*チャネル番号*〉.c 〈*チャネル番号*〉: 0, 1

使用RPDL関数 R_CPB_Destroy

詳細
- コンパレータBのチャネルの電源をOFFにします。
- コンパレータBをモジュールストップ状態に移行します。
  コンパレータBを2チャネル使用している場合は、一方のチャネルに対して本関数を呼び出してもモジュールストップ状態には移行しません。その後もう一方のチャネルに対して本関数を呼び出した時にコンパレータBがモジュールストップ状態に移行します。

使用例 R_PG_CPB_Set_CPB〈*チャネル番号*〉の使用例を参照してください。

5.29 データ演算回路（DOC）

5.29.1 R_PG_DOC_Set

定義　bool R_PG_DOC_Set (void)

概要　データ演算回路の設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_DOC.c

使用RPDL関数　R_DOC_Create

詳細

- 本関数はデータ演算回路(DOC)のモジュールストップ状態を解除して初期設定を行います。
- GUI上で割り込み通知関数名が指定されている場合、CPUへ割り込みが発生すると指定された名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void *<割り込み通知関数名>*(void)
  割り込み通知関数については本章末尾の「通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み通知関数名が指定されている場合、割り込み通知関数内でデータ演算回路フラグはクリアされます。

使用例　GUI上で以下の通り設定した場合

- 動作モードに[データ比較モード]を選択
- 検出条件に[データ比較の結果、一致を検出]を選択
- 割り込み通知関数名に DopcfIntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
uint16_t input_data[10]={1,0,0,1,0,0,0,0,0,1};
uint16_t comp_match_cnt=0;
void func(void)
{
    //データ演算回路の設定
    R_PG_DOC_Set();
    //演算対象データの設定
    R_PG_DOC_InputData(input_data, 10);
}
//データ演算回路割り込み通知関数
void DopcfIntFunc(void)
{
    comp_match_cnt++;
}
```

5.29.2 R_PG_DOC_GetStatusFlag

定義 bool R_PG_DOC_GetStatusFlag (bool * status)

概要 データ演算回路のステータスフラグの取得

引数

bool * status	ステータスフラグの格納先

戻り値

true	フラグの取得が正しく行われた場合
false	フラグの取得に失敗した場合

出力先ファイル R_PG_DOC.c

使用RPDL関数 R_DOC_Read

詳細

- データ演算回路のステータスフラグ(演算結果)を取得します。
- ステータスフラグは以下の場合に ”1” となります。
  - ・データ比較モードにて、データ比較の結果、一致または不一致を検出した場合
  - ・データ加算モードにて、データ加算の結果がFFFFhより大きくなった場合
  - ・データ減算モードにて、データ減算の結果が0000hより小さくなった場合
- 本関数内にて、フラグを取得した後ステータスフラグはクリアされます。

使用例 R_PG_DOC_StopModuleの使用例を参照してください。

5.29.3 R_PG_DOC_GetResult

定義 bool R_PG_DOC_GetResult (uint16_t * result)

概要 データ演算結果の取得

引数

uint16_t * result	演算結果の格納先

戻り値

true	結果の取得が正しく行われた場合
false	結果の取得に失敗した場合

出力先ファイル R_PG_DOC.c

使用RPDL関数 R_DOC_Read

詳細
- DOCデータセッティングレジスタ(DODSR)の値を取得します。
- 動作モードごとに、取得する値の内容が以下のように異なります。
  データ比較モード時 :比較の基準となるデータ
  データ加算モード時 :データ加算結果
  データ減算モード時 :データ減算結果

使用例 R_PG_DOC_StopModuleの使用例を参照してください。

5.29.4 R_PG_DOC_InputData

定義 bool R_PG_DOC_InputData (uint16_t * data, uint16_t count)

概要 演算対象データの設定

引数

uint16_t * data	入力データの格納先
uint16_t count	入力データ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DOC.c

使用RPDL関数 R_DOC_Write

詳細

- 演算対象データをDOCデータインプットレジスタ(DODIR)に設定します。
  データ比較モード時 :比較するデータを設定
  データ加算モード時 :加算するデータを設定
  データ減算モード時 :減算するデータを設定

使用例 R_PG_DOC_Setの使用例を参照してください。

5.29.5 R_PG_DOC_UpdateData

定義 bool R_PG_DOC_UpdateData (uint16_t data)

概要 演算データの更新

引数

uint16_t data	更新するデータ

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DOC.c

使用RPDL関数 R_DOC_Control

詳細

- 指定されたデータでDOCデータセッティングレジスタ(DODSR)の値を更新します。
  データ比較モード時 :比較の基準となるデータを更新
  データ加算モード時 :加算前の初期値を更新
  データ減算モード時 :減算前の初期値を更新

使用例 GUI上で以下の通り設定した場合

- 動作モードに[データ比較モード]を選択
- 検出条件に[データ比較の結果、一致を検出]を選択
- 比較基準値に ”0” を指定
- 割り込み通知関数名に DopcfIntFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t input_data[10]={1,0,0,1,0,0,0,0,0,1};
uint16_t comp_match_cnt=0;
uint16_t comp_match_0, comp_match_1;

void func(void)
{
    //データ演算回路の設定
    R_PG_DOC_Set();

    //演算対象データの設定
    R_PG_DOC_InputData(input_data, 10);

    comp_match_0 = comp_match_cnt;
    comp_match_cnt = 0;

    //演算データの更新
    R_PG_DOC_UpdateData(1);

    //演算対象データの設定
    R_PG_DOC_InputData(input_data, 10);

    comp_match_1 = comp_match_cnt;
}

//データ演算回路割り込み通知関数
void DopcfIntFunc(void)
{
    comp_match_cnt++;
}
```

5.29.6 R_PG_DOC_StopModule

定義 bool R_PG_DOC_StopModule (void)

概要 データ演算回路を停止

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_DOC.c

使用RPDL関数 R_DOC_Destroy

詳細
- 本関数はデータ演算回路(DOC)をモジュールストップ状態に移行します。

使用例

GUI上で以下の通り設定した場合
- 動作モードに[データ加算モード]を選択
- 加減演算結果初期値に ”0” を指定
- データ演算回路割り込み(DOPCF)を使用しない

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"

uint16_t result;
uint16_t data=0x0000;

void func(void)
{
    bool status;

    //データ演算回路の設定
    R_PG_DOC_Set();

    while(1){
        //演算対象データの設定
        R_PG_DOC_InputData(&data, 1);

        //データ演算回路のステータスフラグの取得
        R_PG_DOC_GetStatusFlag(&status);

        if(status == true){
            break;
        }

        //データ演算結果の取得
        R_PG_DOC_GetResult(&result);

        data++;
    }

    //データ演算回路を無効にする
    R_PG_DOC_StopModule();
}
```

5.30 通知関数に関する注意事項

5.30.1 割り込みとプロセッサモード

RX CPU は、スーパバイザモード、およびユーザモードの 2 つのプロセッサモードをサポートします。Peripheral Driver Generator の出力関数および Renesas Peripheral Driver Library の関数はユーザモードで実行されますが、各通知関数は Renesas Peripheral Driver Library の割り込みハンドラから呼び出されるため、スーパバイザモードで動作します。スーパバイザモードでは特権命令(RTFI、RTE、WAIT)を使用できますが、通知関数と通知関数から呼び出される他の関数では以下の点に注意してください。

・ RTFI および RTE 命令は Renesas Peripheral Driver Library の割り込みハンドラで実行するため、ユーザプログラムでこれらを実
行する必要はありません。

・ Peripheral Driver Generator の出力関数および Renesas Peripheral Driver Library の関数では消費電力低減のために wait()命令を呼び出しています。
ユーザプログラムから wait()を呼び出さないでください。

プロセッサモードについての詳細は RX ファミリ ソフトウェアマニュアルを参照してください。

5.30.2 割り込みとDSP命令

アキュムレータ(ACC)は以下の命令で変更されます。

・ DSP 機能命令(MACHI、MACLO、MULHI、MULLO、MVTACHI、MVTACLO、および RACW)
・ 乗算命令、積和演算命令 (EMUL、EMULU、FMUL、MUL、および RMPA)

Renesas Peripheral Driver Library の割り込みハンドラでは ACC の値をスタックに退避しません。各通知関数は Renesas Peripheral Driver Library の割り込みハンドラから呼び出されるため、通知関数内でこれらの命令を使用する場合は ACC の値を退避し、通知関数が終了する前に再設定してください。

6. 生成ファイルのIDEへの登録とビルド

Peripheral Driver Generator で生成したファイルの IDE(High-performance Embedded Workshop／CubeSuite+／e2 studio)への登録とビルドについては以下の点に注意してください。

(1) Peripheral Driver Generator が生成するソースファイルにはスタートアッププログラムは含まれません。IDE のプロジェクト作成時にプロジェクトタイプとして Application を指定してスタートアッププログラムを作成してください。

(2) Peripheral Driver Generator が IDE に登録するソースファイルには割り込みハンドラとベクタテーブルが含まれます。IDE で生成したスタートアッププログラムに含まれる割り込みハンドラ、ベクタテーブルとの重複を避けるため、Peripheral Driver Generator から IDE にソースファイルを登録する際、intprg.c と vecttbl.c(e2 studio の場合は、interrupt_bandlers.c と vector_table.c)はビルドの対象から除外されます。

(3) Peripheral Driver Generator が IDE に登録する割り込みハンドラを含むソースファイル Interrupt_<周辺機能名>.c は、Peripheral Driver Generator のソース生成時に上書きされます。

(4) Renesas Peripheral Driver Library ライブラリファイルは、デフォルトのオプションで作成しています。(ただし、double 型の精度は倍精度に設定して作成しています) お客様のプロジェクトでデフォルト以外のオプションを指定する場合は、お客様の責任で Renesas Peripheral Driver Library のソースファイルを利用してください。

(5) Renesas Peripheral Driver Library は double 型の精度を倍精度に設定して作成されています。そのため、Peripheral Driver Generator が生成したソースを含むプログラムをビルドするには、以下のように IDE のビルダ設定で double 型の精度を指定してください。(e2 studio ではソース登録と同時に自動で変更します)

CubeSuite+

1. プロジェクトツリーの［CC-RX(ビルド・ツール)］をダブルクリックし、[CC-RXのプロパティ］を表示してください。
2. [CPU]カテゴリ内の［double型、およびlong double型の精度］に［倍精度として扱う］を設定してください。

High-performance Embedded Workshop

1. メインメニューから［ビルド］->［RX Standard Toolchain］を選択し、[RX Standard Toolchain］ダイアログボックスを開いてください。
2. ［CPU］タブを選択してください。
3. ［詳細］ボタンをクリックし、[CPU詳細］ダイアログボックスを開いてください。
4. ［double型の精度］に［倍精度］を設定してください。

(6) Renesas Peripheral Driver Library は FIXEDVECT セクションの開始アドレスを、0xFFFFFFD0 にして作成しています。そのため Peripheral Driver Generator が生成したソースを含むプログラムをビルドするには、以下のようにビルダの設定で FIXEDVECT セクションのアドレスを変更してください。(CubeSuite+ および High-performance Embedded Workshop では変更不要)

e2 studio

1. プロジェクトエクスプローラでプロジェクトを選んでください。

2. メニューから[ファイル]->[プロパティ]を選択し[プロパティ]を表示してください。
3. プロパティの[C/C++ビルド]の[設定]を選んでください。
4. 構成:で[全ての構成]を選んでください。
5. [Linker]の[セクション]を選択し、「セクション・ビューアー」を表示してください。
6. [セクション・ビューアー]で、FIXEDVECTセクションのアドレスを0xFFFFFFD0に設定してください。

RX210グループ
Peripheral Driver Generator
リファレンスマニュアル

発行年月日　2014年5月16日　Rev.1.04

発行　ルネサス エレクトロニクス株式会社
〒211-8668 神奈川県川崎市中原区下沼部1753

編集　株式会社ルネサス ソリューションズ
ツールビジネス本部 ツール開発第四部

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/

RX210グループ

Peripheral Driver Generator

リファレンスマニュアル

ルネサス エレクトロニクス株式会社

R20UT1448JJ0104