User's Manual

RX610グループ
Peripheral Driver Generator
リファレンスマニュアル

本資料に記載の全ての情報は本資料発行時点のものであり、ルネサス エレクトロニクスは、
予告なしに、本資料に記載した製品または仕様を変更することがあります。
ルネサス エレクトロニクスのホームページなどにより公開される最新情報をご確認ください。

ルネサス エレクトロニクス
www.renesas.com

Rev.1.01　2011.02

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。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。
3. 当社製品を改造、改変、複製等しないでください。
4. 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。
5. 輸出に際しては、「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。本資料に記載されている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の目的で使用しないでください。また、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に使用することができません。
6. 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。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されてい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生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   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

   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、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社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。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、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、社会的損害など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、機器またはシステムとしての出荷保証をお願いいたします。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、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、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。
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固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

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 RX610 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 RX610 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
R5F56108VNFPP	LQP0144KAA
R5F56107VNFPP	LQP0144KAA
R5F56106VNFPP	LQP0144KAA
R5F56104VNFPP	LQP0144KAA

(2) 周辺機能

- クロック発生回路
- 割り込みコントローラ (ICU)、例外処理
- I/O ポート
- DMA コントローラ (DMAC)
- 16 ビットタイマパルスユニット (TPU)
- 8 ビットタイマ (TMR)
- コンペアマッチタイマ (CMT)
- シリアルコミュニケーションインタフェース (SCI)
- I2C バスインタフェース (RIIC)
- A/D 変換器

(3) エンディアン

リトルエンディアン

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。

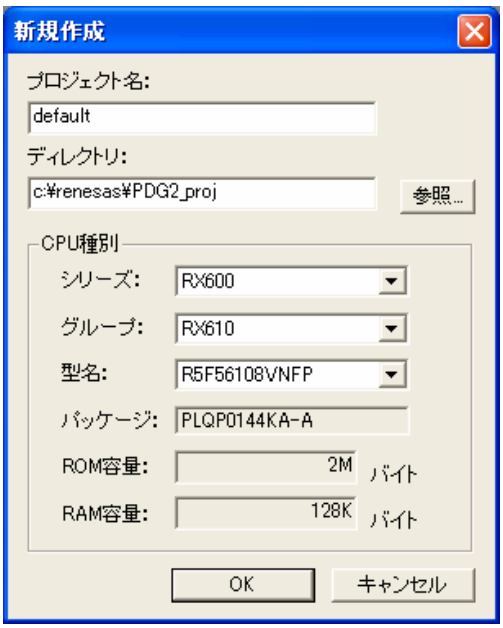


図 2.1 新規作成ダイアログボックス

RX610 グループのプロジェクトを作成するにはシリーズに［RX600］を、グループに［RX610］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。

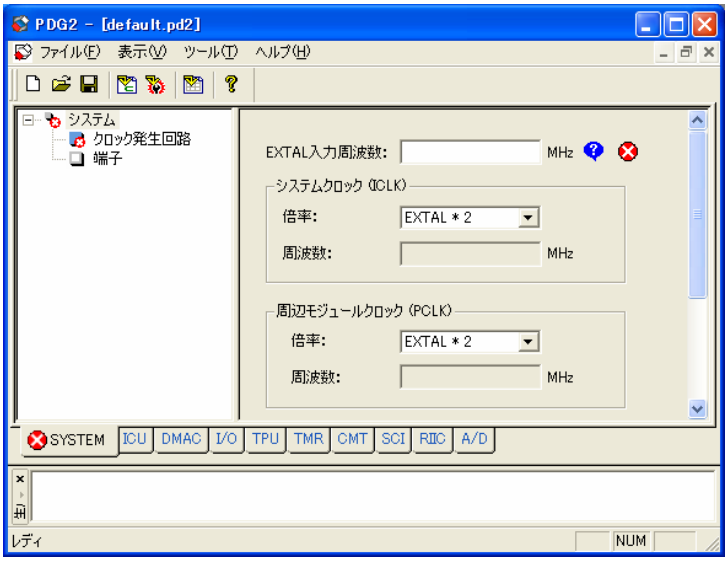


図 2.2 新規プロジェクトのエラー表示

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。

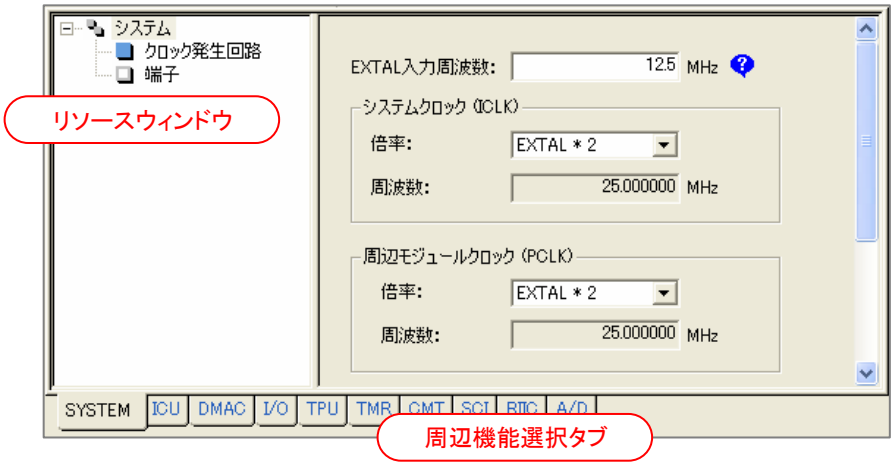


図3.1 周辺機能設定ウィンドウの表示例

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

周辺機能選択タブ	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	端子機能
ICU	割り込み	割り込みコントローラ（ICU）（高速割り込み, NMI, IRQ0～15）
	例外	例外処理
DMAC	DMAC0 ～ DMAC3	DMAコントローラ（DMAC）チャネル0～3
I/O	ポート0 ～ ポートE	I/Oポート ポート0～E
TPU	ユニット0（TPU0～TPU5）	16ビットタイマパルスユニット（TPU）ユニット0（チャネル0～5）
	ユニット1（TPU6～TPU11）	16ビットタイマパルスユニット（TPU）ユニット1（チャネル6～11）
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 and 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 and CMT3）	コンペアマッチタイマ（CMT）ユニット1（チャネル2、3）
SCI	SCI0 ～ SCI6	シリアルコミュニケーションインタフェース（SCI）チャネル0～6
RIIC	RIIC0、RIIC1	I2Cバスインタフェース（RIIC）チャネル0、1
A/D	AD0 ～ AD3	A/D 変換器 ユニット0～3

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

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]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。

3.2.1 端子機能シート

端子機能シートではマイクロコントローラの全端子を番号順に表示します。

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（端子に割り当ている全機能）が表示されます
選択機能	周辺機能の設定により選択されている端子機能が表示されます
入出力	周辺機能の設定により選択されている端子の入出力方向が表示されます
状態	設定状態が表示されます

端子の入出力に関連する周辺機能を設定すると、設定の結果がウィンドウに表示されます。例えば A/D 変換器 AD0 の設定ウィンドウで、アナログ入力端子 AN0 の入力を A/D 変換するよう設定した場合、AN0 が割り当てられている 141 番の端子(P40/IRQ8/AN0)の行は、図 3.4 に示す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
141	P40/IRQ8/AN0	AN0	入力	

図3.4 端子機能の表示例

この状態で I/O ポート P40 を設定すると、図 3.5 に示すように端子機能の競合が警告されます。

端子番号	端子名	選択機能	入出力	状態
141	P40/IRQ8/AN0	AN0/P40		複数の機能で競合しています

図3.5 端子機能競合時の表示

注意

- RX610 グループでは端子ごとに割り当てる機能を指定することはできません。端子の機能は周辺機能の設定により決まります。本ウィンドウで端子機能を変更することはできません。
- 端子機能によっては割当先の端子を切り替えることができます。端子機能の割当先は周辺機能別使用端子シートで変更することができます。
- 複数の出力機能が 1 つの端子で有効に設定された場合、出力優先度の高い機能の信号が出力されます。詳細についてはハードウェアマニュアルを参照してください。

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では周辺機能ごとに端子の使用状況が表示されます。左側の周辺機能一覧から選択した周辺機能の端子機能が表示されます。

図3.6 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
選択機能	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	設定状態が表示されます

端子の入出力に関連する周辺機能を設定すると、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 IRQ9 を設定すると、IRQ9 端子は、図 3.7 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
IRQ9	外部割込み入力	P01/IRQ9/TMCI2/RxD6	7	入力	

図3.7 使用端子の表示例

この状態で同じ端子を使用する I/O ポート P01 を設定すると、図 3.8 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
IRQ9	外部割込み入力	P01/IRQ9/TMCI2/RxD6	7	入力	他の機能と競合しています

図3.8 端子機能競合時の表示

IRQ9 は割当先を変更すことができます。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。

図3.9 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当先の端子を指定してください。

図3.10 端子機能の割当先変更

IRQ9 の割当先を P41/IRQ9/AN1 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。

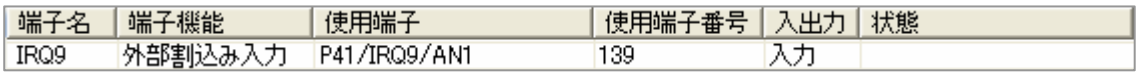


図3.11 端子機能の割当先変更後の表示

割当先を変更できる端子機能を表 3.4 に示します。

表 3.4 割当先を変更できる端子機能（RX610 144pin）（上段が初期設定です）

周辺機能	端子機能	割当先端子	端子番号
TPU ユニット0 （TPU0～TPU5 共通）	TCLKA *1	P32/IRQ2/PO10/TIOCC0/TCLKA	27
		P14/IRQ4/TCLKA/SDA1	43
	TCLKB *1	P33/IRQ3/PO11/TIOCC0/TIOCD0/TCLKB	26
		P15/IRQ5/TCLKB/SCK3/SCL1	42
	TCLKC *1	P35/PO13/TIOCA1/TIOCB1/TCLKC	40
		P16/IRQ6/TCLKC/RxD3/SDA0	50
	TCLKD *1	P37/PO15/TIOCA2/TIOCB2/TCLKD	38
		P17/IRQ7/TCLKD/TxD3/SCL0/ADTRG1#	48
TPU0	TIOCA0(IC) *2	P30/IRQ0/PO8/TIOCA0	29
		P31/IRQ1/PO9/TIOCA0/TIOCB0	28
	TIOCC0(IC) *2	P32/IRQ2/PO10/TIOCC0/TCLKA	27
		P33/IRQ3/PO11/TIOCC0/TIOCD0/TCLKB	26
TPU1	TIOCA1(IC) *2	P34/IRQ4/PO12/TIOCA1	25
		P35/PO13/TIOCA1/TIOCB1/TCLKC	50
TPU2	TIOCA2(IC) *2	P36/PO14/TIOCA2	49
		P37/PO15/TIOCA2/TIOCB2/TCLKD	48
TPU3	TIOCA3(IC) *2	P21/PO1/TIOCA3/TMCI0/RxD0	36
		P20/PO0/TIOCA3/TIOCB3/TMRI0/TxD0	37
	TIOCC3(IC) *2	P22/PO2/TIOCC3/TMO0/SCK0	35
		P23/PO3/TIOCC3/TIOCD3	34
TPU4	TIOCA4(IC) *2	P25/PO5/TIOCA4/TMCI1/RxD1	32
		P24/PO4/TIOCA4/TIOCB4/TMRI1	33
TPU5	TIOCA5(IC) *2	P26/PO6/TIOCA5/TMO1/TxD1	31
		P27/PO7/TIOCA5/TIOCB5/SCK1	30

周辺機能	端子機能	割当先端子	端子番号
TPU6	TIOCA6(IC) *2	PA0/A0/BC0#/PO16/TIOCA6	101
		PA1/A1/PO17/TIOCA6/TIOCB6	100
	TIOCC6(IC) *2	PA2/A2/PO18/TIOCC6/TCLKE	99
		PA3/A3/PO19/TIOCC6/TIOCD6/TCLKF	98
TPU7	TIOCA7(IC) *2	PA4/A4/PO20/TIOCA7	97
		PA5/A5/PO21/TIOCA7/TIOCB7/TCLKG	96
TPU8	TIOCA8(IC) *2	PA6/A6/PO22/TIOCA8	95
		PA7/A7/PO23/TIOCA8/TIOCB8/TCLKH	94
TPU9	TIOCA9(IC) *2	PB0/A8/PO24/TIOCA9	92
		PB1/A9PO25/TIOCA9/TIOCB9	85
	TIOCC9(IC) *2	PB2/A10/PO26/TIOCC9	84
		PB3/A11/PO27/TIOCC9/TIOCD9	83
TPU10	TIOCA10(IC) *2	PB4/A12/PO28/TIOCA10	82
		PB5/A13/PO29/TIOCA10/TIOCB10	81
TPU11	TIOCA11(IC) *2	PB6/A14/PO30/TIOCA11	80
		PB7/A15/PO31/TIOCA11/TIOCB11	79
ICU (外部割込み)	IRQ0	P30/IRQ0/PO8/TIOCA0	29
		P10/IRQ0	47
	IRQ1	P31/IRQ1/PO9/TIOCA0/TIOCB0	28
		P11/IRQ1/SCK2	46
	IRQ2	P32/IRQ2/PO10/TIOCC0/TCLKA	27
		P12/IRQ2/RxD2	45
	IRQ3	P33/IRQ3/PO11/TIOCC0/TIOCD0/TCLKB	26
		P13/IRQ3/TxD2/ADTRG0#	44
	IRQ4	P34/IRQ4/PO12/TIOCA1	25
		P14/IRQ4/TCLKA/SDA1	43
	IRQ5	PE5/IRQ5/D13	104
		P15/IRQ5/TCLKB/SCK3/SCL1	42
	IRQ6	PE6/IRQ6/D14	103
		P16/IRQ6/TCLKC/RxD3/SDA0	40
	IRQ7	PE7/IRQ7/D15	102
		P17/IRQ7/TCLKD/TxD3/SCL0/ADTRG1#	38
	IRQ8	P00/IRQ8/TMRI2/TxD6	8
		P40/IRQ8/AN0	141
	IRQ9	P01/IRQ9/TMCI2/RxD6	7
		P41/IRQ9/AN1	139
	IRQ10	P02/IRQ10/TMO2/SCK6/TRST#	6
		P42/IRQ10/AN2	138
	IRQ11	P03/IRQ11/TMRI3/SCK4/TMS	2
		P43/IRQ11/AN3	137
	IRQ12	P04/IRQ12/TMCI3/TxD4/TDI	1
		P44/IRQ12/AN4	136
	IRQ13	P05/IRQ13/TMO3/RxD4/TCK	144
		P45/IRQ13/AN5	135
	IRQ14	P76/IRQ14	67
		P46/IRQ14/AN6	134
	IRQ15	P65/IRQ15	9
		P47/IRQ15/AN7	133

*1 設定は連動して変更されます。

*2 インプットキャプチャ入力として使用した場合です。

4.生成関数仕様

RX610 の生成関数を表 4.1 に示します。

表 4.1 RX610 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

割り込みコントローラ（ICU）

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_FastInterrupt_Set	高速割り込みの設定
R_PG_Exception_Set	例外ハンドラの設定

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〉*	I/Oポートレジスタからの読み出し
R_PG_IO_PORT_Read_P*〈ポート番号〉〈端子番号〉*	I/Oポート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〉*	I/Oポート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〉*	I/Oポートデータレジスタへのビット書き込み

DMAコントローラ（DMAC）

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〉*	データ転送の開始(ソフトウェアトリガ)
R_PG_DMAC_Suspend_C*〈チャネル番号〉*	データ転送の中断
R_PG_DMAC_GetTransferredByteCount_C*〈チャネル番号〉*	転送済みデータ数の取得
R_PG_DMAC_ClearTransferEndFlag_C*〈チャネル番号〉*	データ転送終了フラグのクリア
R_PG_DMAC_SetReload_SrcAddress_C*〈チャネル番号〉*	データ転送元アドレスリロード値の設定
R_PG_DMAC_SetReload_DestAddress_C*〈チャネル番号〉*	データ転送先アドレスリロード値の設定
R_PG_DMAC_SetReload_ByteCount_C*〈チャネル番号〉*	データ転送サイズリロード値の設定
R_PG_DMAC_StopModule	DMAC全チャネルの停止

16ビットタイマパルスユニット（TPU）

生成関数	機能
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_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_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のユニットを停止

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_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_Start_TMR_U*〈ユニット番号〉*_C*〈チャネル番号〉*	CMTを設定しカウントを開始
R_PG_Timer_HaltCount_CMT_U*〈ユニット番号〉*_C*〈チャネル番号〉*	CMTのカウントを一時停止
R_PG_Timer_ResumeCount_CMT_U*〈ユニット番号〉*_C*〈チャネル番号〉*	CMTのカウントを再開
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

シリアルコミュニケーションインタフェース（SCI）

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_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_GetSentDataCount_C*〈チャネル番号〉*	シリアルデータの送信数取得
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_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_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

A/D変換器

生成関数	機能
R_PG_ADC_10_Set_AD〈*ユニット番号*〉	A/D変換器の設定
R_PG_ADC_10_StartConversionSW_AD〈*ユニット番号*〉	A/D変換の開始(ソフトウェアトリガ)
R_PG_ADC_10_StopConversion_AD〈*ユニット番号*〉	A/D変換の中断
R_PG_ADC_10_GetResult_AD〈*ユニット番号*〉	A/D変換結果の取得
R_PG_ADC_10_StopModule_AD〈*ユニット番号*〉	A/D変換器の停止

4.1 クロック発生回路

4.1.1 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

- クロック発生回路のレジスタを設定し、EXTALに対するシステムクロック(ICLK)、周辺モジュールクロック(PCLK)、外部バスクロック(BCLK)の逓倍率を設定します。

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 func(void)
{
    //クロック発生回路を設定する
    R_PG_Clock_Set();
}
```

4.2 割り込みコントローラ（ICU）

4.2.1 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～IRQ15、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～IRQ15、またはNMI

使用RPDL関数　R_INTC_CreateExtInterrupt

詳細

- 外部割り込み(IRQ0～IRQ15、またはNMI) を有効にし、使用する外部割り込み端子の入力方向と入力バッファの設定を行います。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(IRQn-A/B)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「4.11 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、外部割り込みが入力され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_ExtInterrupt_GetRequestFlag_〈*割り込み種別*〉 により取得することができます。

使用例1　割り込み通知関数名にIrq0ExtIntFuncを指定する場合

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

      While(1);
}

//IRQ0通知関数
void Irq0ExtIntFunc (void)
{
      func_irq0();      //IRQ0の処理
}
```

使用例2 割り込み通知関数名を指定しない場合

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

	While(1){
		bool flag;

		//IRQ0の割り込み要求フラグを取得する
		R_PG_ExtInterrupt_GetRequestFlag_IRQ0( &flag );
		if( 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();
	}
}
```

4.2.2 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～IRQ15

概要　外部割り込みの設定解除

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_ExtInterrupt_*〈割り込み種別〉*.c
〈割り込み種別〉 ： IRQ0～IRQ15

使用RPDL関数　R_INTC_ControlExtInterrupt

詳細
- 外部割り込み(IRQ0～IRQ15) を無効にします。
  外部割込みに使用した端子の設定(入出力方向、入力バッファ設定)は保持されます。

使用例　割り込み通知関数名にIrq0ExtIntFuncを指定する場合

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

	While(1);
}

//外部割り込み(IRQ0)通知関数
void Irq0ExtIntFunc (void)
{
	//IRQ0を無効にする
	R_PG_ExtInterrupt_Disable_IRQ0();

	func_irq0();	//IRQ0の処理
}
```

4.2.3 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～IRQ15、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_ExtInterrupt_〈割り込み種別〉.c

〈割り込み種別〉 : IRQ0～IRQ15、またはNMI

使用RPDL関数

R_INTC_GetExtInterruptStatus

詳細

- 外部割り込み(IRQ0～IRQ15、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

割り込み通知関数名を指定しない場合

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

    While(1){
        bool flag;

        //IRQ0の割り込み要求フラグを取得する
        R_PG_ExtInterrupt_GetRequestFlag_IRQ0( &flag );
        if( 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();
    }
}
```

4.2.4 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～IRQ15、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	クリアが正しく行われた場合
false	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～IRQ15、またはNMI

使用RPDL関数　R_INTC_ControlExtInterrupt

詳細
- 外部割り込み(IRQ0～IRQ15、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レベル検出で、入力信号がLowの場合はクリアできません。

使用例　割り込み通知関数名を指定しない場合

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

    While(1){
        bool flag;

        //IRQ0の割り込み要求フラグを取得する
        R_PG_ExtInterrupt_GetRequestFlag_IRQ0( &flag );
        if( 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();
    }
}
```

4.2.5 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_<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 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();
}
```

4.2.6 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「4.11 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
浮動小数点例外 : FpExcFunc

```
//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 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
}

void FpExcFunc (){
    funct_fp_excep();    //浮動小数点例外発生時の処理
}
```

4.3 I/Oポート

4.3.1 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*〉 ： 0～9、A～E

概要　I/O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
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*〉 ： 0～9、A～E

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、入力バッファ、プルアップ、オープンドレイン出力の設定を行います。
- [I/Oポートとして使用]がチェックされたポート内の全端子を一括して設定します。

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 func(void)
{
    //P0を設定する
    R_PG_IO_PORT_Set_P0();
}
```

4.3.2 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*〉 ： 0～9、A～E
〈*端子番号*〉 ： 0～7

概要　I/Oポート(1端子)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_IO_PORT_P〈*ポート番号*〉.c
〈*ポート番号*〉 ： 0～9、A～E

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、入力バッファ、プルアップ、オープンドレイン出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_〈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 func(void)
{
    //P00を設定する
    R_PG_IO_PORT_Set_P00();

    //P01を設定する
    R_PG_IO_PORT_Set_P01();

    //P02を設定する
    R_PG_IO_PORT_Set_P02();
}
```

4.3.3 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*〉： 0～9、A～E

概要　I/Oポートレジスタからの読み出し

引数

uint8_t * data	読み出した端子状態の格納先

戻り値

true	読み出しが正しく行われた場合
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*〉： 0～9、A～E

使用RPDL関数　R_IO_PORT_Read

詳細
- I/Oポートレジスタを読み出し、端子の状態を取得します。

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 func(void)
{
    uint8_t data;

    //P0端子状態を取得する
    R_PG_IO_PORT_Read_P0( &data );
}
```

4.3.4 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*〉 : 0～9、A～E

〈*端子番号*〉 : 0～7

概要

I/Oポートレジスタからのビット読み出し

引数

uint8_t * data	読み出した端子状態の格納先

戻り値

true	読み出しが正しく行われた場合
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*〉 : 0～9、A～E)

使用RPDL関数

R_IO_PORT_Read

詳細

- I/Oポート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_<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 func(void)
{
    uint8_t data_p00, data_p01, data_p02;

    //P00端子状態を取得する
    R_PG_IO_PORT_Read_P00( & data_p00);

    //P01端子状態を取得する
    R_PG_IO_PORT_Read_P01( & data_p01);

    //P02端子状態を取得する
    R_PG_IO_PORT_Read_P02( & data_p02);
}
```

4.3.5 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*〉 : 0～9、A～E

概要 I/Oポートデータレジスタへの書き込み

引数

uint8_t data	書き込む値

戻り値

true	書き込みが正しく行われた場合
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*〉 : 0～9、A～E

使用RPDL関数 R_IO_PORT_Write

詳細
- I/Oポート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。

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 func(void)
{
    //P0を設定する
    R_PG_IO_PORT_Set_P0();

    //P0から0x03を出力する
    R_PG_IO_PORT_Set_P0( 0x03 );
}
```

4.3.6 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*〉 : 0～9、A～E

〈*端子番号*〉 : 0～7

概要

I/Oポートデータレジスタへのビット書き込み

引数

uint8_t data	書き込む値

戻り値

true	書き込みが正しく行われた場合
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*〉 : 0～9、A～E

使用RPDL関数

R_IO_PORT_Write

詳細

- I/Oポート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_〈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 func(void)
{
    //P00を設定する
    R_PG_IO_PORT_Set_P00();

    //P01を設定する
    R_PG_IO_PORT_Set_P01();

    //P00からLを出力する
    R_PG_IO_PORT_Write_P00( 0x00 );

    //P01からHを出力する
    R_PG_IO_PORT_Write_P01( 0x01 );
}
```

4.4 DMACコントローラ（DMAC）

4.4.1 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*〉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「4.11 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

転送方式	： 単一オペランド転送
転送先開始アドレス	： シリアル送信データレジスタのアドレス
転送先アドレス加算方向	： 固定
単位データサイズ	： 1バイト
1オペランドのデータ数	： 1

SCI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

転送方式	： 単一オペランド転送
転送元開始アドレス	： シリアル受信データレジスタのアドレス
転送元アドレス加算方向	： 固定
単位データサイズ	： 1バイト
1オペランドのデータ数	： 1

SCI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_<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 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();
}

//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMACを停止
    R_PG_DMAC_StopModule();
}
```

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_<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"

//DMA転送終了フラグ
volatile uint8_t sci_dma_transfer_complete;

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

    //DMAC0を設定する
    R_PG_DMAC_Set_C0();

    //SCI0を設定する
    R_PG_SCI_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();

    //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);
}
```

```
//全データが転送されると呼び出される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_cmplete;
	sci_transfer_cmplete = false;

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

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

	//DMACを停止
	R_PG_DMAC_StopModule();

	sci_dma_transfer_complete = ture;
}
```

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_<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"

//DMA転送終了フラグ
volatile uint8_t sci_dma_transfer_complete;

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

	//DMAC0を設定する
	R_PG_DMAC_Set_C0();

	//SCI0を設定する
	R_PG_SCI_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();

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

//全データが転送されると呼び出される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

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

4.4.2 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
- 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_<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 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();
}

//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMACを停止
    R_PG_DMAC_StopModule();
}
```

4.4.3 R_PG_DMAC_StartTransfer_C〈*チャネル番号*〉

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

概要 データ転送の開始(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DMACチャネルの転送を開始します。
- 本関数は転送開始要因にソフトウェアトリガ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
- 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_〈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 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();
}

//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMACを停止
    R_PG_DMAC_StopModule();
}
```

4.4.4 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	中断が正しく行われた場合
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転送を中断します。

使用例　IRQ1の割り込みで転送を中断する場合
- 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Irq1ExtIntFuncを指定

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

    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();

    //IRQ1を設定する
    R_PG_ExtInterrupt_Set_IRQ1();

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

//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMAC0を停止
    R_PG_DMAC_StopModule();
}

//IRQ1割り込み通知関数
void Irq1ExtIntFunc (void)
{
    //DMAC0の転送を中断
    R_PG_DMAC_Suspend_C0();
}
```

4.4.5 R_PG_DMAC_GetTransferredByteCount_C〈*チャネル番号*〉

義 bool R_PG_DMAC_GetTransferredByteCount_C〈*チャネル番号*〉(uint32_t * data)

〈*チャネル番号*〉 : 0～3

概要 転送バイトカウントレジスタ値の取得

引数

uint32_t * data	カレント転送バイトカウント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_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

- カレント転送バイトカウントレジスタ値を取得します。

使用例 GUI上で以下の通り設定した場合

- 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_<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 func(void)
{
    uint32_t count;

    //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();

    //カレント転送バイトカウントレジスタ値が10以下になるまで待つ
    do{
        R_PG_DMAC_GetTransferredByteCount_C0( & count );
    } while( count > 10 );

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

4.4.6 R_PG_DMAC_ClearTransferEndFlag_C〈*チャネル番号*〉

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

概要 DMA転送終了フラグのクリア

引数 なし

戻り値

true	クリアが正しく行われた場合
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
- DMA転送終了フラグをクリアします。
- GUI上で割り込み通知関数を指定した場合、DMA転送終了フラグ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
- DMAC0の転送開始要因をソフトウェアトリガに設定
- 割り込みを使用しない

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

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

4.4.7 R_PG_DMAC_SetReload_SrcAddress_C〈チャネル番号〉

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

概要 データ転送元アドレスリロード値の設定

生成条件 転送元アドレスリロード機能を使用

引数

uint32_t data	データ転送元アドレスリロード値

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
- データ転送元アドレスリロード値を設定します。
- DMA割り込み通知関数の中から呼び出してください。

使用例 転送元アドレス、転送先アドレス、転送データサイズのリロード機能を使用する場合
- DMAC0の転送方式を連続オペランド転送に設定
- 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
- 転送元アドレス、転送先アドレス、転送データサイズのリロード機能を有効に設定

```
//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 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();
}
//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
	if( contimue ){	//リロードして転送を継続
		//転送元アドレスのリロード
		R_PG_DMAC_SetReload_SrcAddress_C0( src_address );
		//転送先アドレスのリロード
		R_PG_DMAC_SetReload_DestAddress_C0( dest_address );
		//転送データサイズのリロード
		R_PG_DMAC_SetReload_ByteCount_C0( byte_count );
	}
	else{	//DMAC0の転送を中断
		R_PG_DMAC_Suspend_C0();
	}
}
```

4.4.8 R_PG_DMAC_SetReload_DestAddress_C〈チャネル番号〉

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

概要　データ転送先アドレスリロード値の設定

生成条件　転送先アドレスリロード機能を使用

引数

uint32_t data	データ転送先アドレスリロード値

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
- データ転送先アドレスリロード値を設定します。
- DMA割り込み通知関数の中から呼び出してください。

使用例　転送元アドレス、転送先アドレス、転送データサイズのリロード機能を使用する場合
- DMAC0の転送方式を連続オペランド転送に設定
- 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
- 転送元アドレス、転送先アドレス、転送データサイズのリロード機能を有効に設定

```
//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 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();
}
//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
	if( contimue ){	//リロードして転送を継続
		//転送元アドレスのリロード
		R_PG_DMAC_SetReload_SrcAddress_C0( src_address );
		//転送先アドレスのリロード
		R_PG_DMAC_SetReload_DestAddress_C0( dest_address );
		//転送データサイズのリロード
		R_PG_DMAC_SetReload_ByteCount_C0( byte_count );
	}
	else{	//DMAC0の転送を中断
		R_PG_DMAC_Suspend_C0();
	}
}
```

4.4.9 R_PG_DMAC_SetReload_ByteCount_C〈*チャネル番号*〉

定義

bool R_PG_DMAC_SetReload_ByteCount_C〈*チャネル番号*〉(uint32_t data)

〈*チャネル番号*〉： 0～3

概要

転送データサイズリロード値の設定

生成条件

転送データサイズリロード機能を使用

引数

uint32_t data	転送データサイズリロード値

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

- 転送データサイズリロード値を設定します。
- DMA割り込み通知関数の中から呼び出してください。

使用例

転送元アドレス、転送先アドレス、転送データサイズのリロード機能を使用する場合

- DMAC0の転送方式を連続オペランド転送に設定
- 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
- 転送元アドレス、転送先アドレス、転送データサイズのリロード機能を有効に設定

```
//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 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();
}
//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
	if( contimue ){	//リロードして転送を継続
		//転送元アドレスのリロード
		R_PG_DMAC_SetReload_SrcAddress_C0( src_address );
		//転送先アドレスのリロード
		R_PG_DMAC_SetReload_DestAddress_C0( dest_address );
		//転送データサイズのリロード
		R_PG_DMAC_SetReload_ByteCount_C0( byte_count );
	}
	else{	//DMAC0の転送を中断
		R_PG_DMAC_Suspend_C0();
	}
}
```

4.4.10 R_PG_DMAC_StopModule

定義　bool R_PG_DMAC_StopModule (void)

概要　DMAC全チャネルの停止

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_DMAC.c

使用RPDL関数　R_DMAC_Destroy

詳細
- DMAC全チャネルを停止し、モジュールストップ状態に移行します。
- 1チャネルの転送を停止するには R_PG_DMAC_Suspend_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
- 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_<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 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();
}

//全データが転送されると呼び出されるDMA割り込み通知関数
void Dmac0IntFunc (void)
{
    //DMACを停止
    R_PG_DMAC_StopModule();
}
```

4.5 16 ビットタイマパルスユニット（TPU）

4.5.1 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*〉： 0,1
〈*チャネル番号*〉： 0～11

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*〉： 0,1
〈*チャネル番号*〉： 0～11

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「4.11 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1 チャネル6 を設定
- コンペアマッチA割り込み通知関数名に Tpu6IcCmAIntFunc を指定

```
//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 func(void)
{
    //TPU6を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U1_C6();
}

void Tpu6IcCmAIntFunc(void)
{
    func_cmA();    //コンペアマッチA割り込み発生時の処理
}
```

4.5.2 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〉： 0,1

〈チャネル番号〉： 0～11

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〉： 0,1

〈チャネル番号〉： 0～11

使用RPDL関数　R_TPU_Control

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 ユニット1 チャネル6 を設定
- コンペアマッチA割り込み通知関数名に Tpu6IcCmAIntFunc を指定

```
//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 func(void)
{
    //TPU6を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U1_C6();
}

void Tpu6IcCmAIntFunc(void)
{
    //TPU6のカウントを一時停止
    R_PG_Timer_HaltCount_TPU_U1_C6();

    func_cmA();     //コンペアマッチA割り込み発生時の処理

    //TMR0のカウントを再開
    R_PG_Timer_ResumeCount_TPU_U1_C6();
}
```

4.5.3 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〉 : 0,1
〈チャネル番号〉 : 0～11

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〉 : 0,1
〈チャネル番号〉 : 0～11

使用RPDL関数 R_TPU_Control

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 ユニット1 チャネル6 を設定
- コンペアマッチA割り込み通知関数名に Tpu6IcCmAIntFunc を指定

```
//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 func(void)
{
    //TPU6を設定し、カウントを開始
    R_PG_Timer_Start_TPU_U1_C6();
}

void Tpu6IcCmAIntFunc(void)
{
    //TPU6のカウントを一時停止
    R_PG_Timer_HaltCount_TPU_U1_C6();

    func_cmA();     //コンペアマッチA割り込み発生時の処理

    //TMR0のカウントを再開
    R_PG_Timer_ResumeCount_TPU_U1_C6();
}
```

4.5.4 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〉 : 0,1

〈チャネル番号〉 : 0～11

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〉 : 0,1

〈チャネル番号〉 : 0～11

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 ユニット0 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_〈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"

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 );
}
```

4.5.5 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〉 : 0,1

〈チャネル番号〉 : 0～11

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〉 : 0,1

〈チャネル番号〉 : 0～11

使用RPDL関数 R_TPU_Control

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 ユニット0 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_〈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"

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 );
}
```

4.5.6 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〉 : 0,1
〈チャネル番号〉 : 0～11

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〉 : 0,1
〈チャネル番号〉 : 0～11

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、6、9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 ユニット0 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_<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"

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 ユニット0を停止
}
```

4.5.7 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〉： 0,1

概要　TPUのユニットを停止

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〉： 0,1

使用RPDL関数　R_TPU_Destroy

詳細
- TPU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るため、本関数を呼び出すとユニット内の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 ユニット0 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_<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"

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 ユニット0を停止
	R_PG_Timer_StopModule_TPU_U0( counter );
}
```

4.6 8 ビットタイマ（TMR）

4.6.1 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_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「4.11 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_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入出力方向と入力バッファを設定します。

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 ： TmrCma2IntFunc

```
//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 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_U0();
}

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 TmrCma2IntFunc(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
GUI上で割り込みフラグのチェックにより割り込み要求の有無を確認

```
//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)
{
    bool cma_flag;

    //TMR0を8ビットモードで設定し、カウントを開始する
    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
        }
    }
}
```

4.6.2 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_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のカウントを一時停止します。カウントを再開するには
  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_<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 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のカウントを再開
    R_PG_Timer_ResumeCount_TMR_U0_C0();
}
```

4.6.3 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_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のカウントを再開します。

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_〈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 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のカウントを再開
    R_PG_Timer_ResumeCount_TMR_U0_C0();
}
```

4.6.4 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 * data)

〈*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 * data)

〈*ユニット番号*〉 : 0,1

概要

TMRのカウンタ値を取得

引数

uint8_t * data 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_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_〈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)
{
    //TMR0を8ビットモードで設定する
    R_PG_Timer_Start_TMR_U0_C0();
}

unt8_t func2(void)
{
    uint8_t data;

    //TMR0のカウンタ値を取得
    R_PG_Timer_GetCounterValue_TMR_U0_C0( &data );

    return data;
}
```

4.6.5 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 data)

〈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 data)

〈ユニット番号〉 : 0,1

概要

TMRのカウンタ値を設定

引数

uint8_t data (8ビットモード時) uint16_t data (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_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_〈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)
{
    //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 );

    return;
}
```

4.6.6 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

なし

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_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。

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_〈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"

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
}
```

4.6.7 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_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_<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 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();
}
```

4.7 コンペアマッチタイマ（CMT）

4.7.1 R_PG_Timer_Start_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義　bool R_PG_Timer_Start_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_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し、カウントを開始します。
- 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「4.11 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上でコンペアマッチ割り込み通知関数名に Cmt0IntFunc を設定した場合

```
//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 func(void)
{
    //CMT0を設定する
    R_PG_Timer_Start_CMT_U0_C0();
}

void Cmt0IntFunc(void)
{
    func_cmt0();    //コンペアマッチ割り込み発生時の処理
}
```

4.7.2 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_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のカウントを一時停止します。カウントを再開するには
  R_PG_Timer_ResumeCount_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　GUI上でコンペアマッチ割り込み関数名に Cmt0IntFunc を指定

```
//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 func(void)
{
	//CMT0を設定する
	R_PG_Timer_Start_CMT_U0_C0();
}

void Cmt0IntFunc(void)
{
	//CMT0のカウントを一時停止
	R_PG_Timer_HaltCount_CMT_U0_C0();

	func_cmt0();	//コンペアマッチ割り込み発生時の処理

	//CMT0のカウントを再開
	R_PG_Timer_ResumeCount_CMT_U0_C0();
}
```

4.7.3 R_PG_Timer_ResumeCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_ResumeCount_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_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
- R_PG_Timer_HaltCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉 により停止したCMTのカウントを再開します。

使用例 GUI上でコンペアマッチ割り込み関数名に Cmt0IntFunc を指定

```
//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 func(void)
{
	//CMT0を設定する
	R_PG_Timer_Start_CMT_U0_C0();
}

void Cmt0IntFunc(void)
{
	//CMT0のカウントを一時停止
	R_PG_Timer_HaltCount_CMT_U0_C0();

	func_cmt0();	//コンペアマッチ割り込み発生時の処理

	//CMT0のカウントを再開
	R_PG_Timer_ResumeCount_CMT_U0_C0();
}
```

4.7.4 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 * data)

〈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 * 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_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CMT0を設定

```
//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)
{
	//CMT0を設定する
	R_PG_Timer_Start_CMT_U0_C0();
}

unt8_t func2(void)
{
	Uint16_t data;

	//CMT0のカウンタ値を取得
	R_PG_Timer_GetCounterValue_CMT_U0_C0( &data );

	return data;
}
```

4.7.5 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 data)

〈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 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_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CMT0を設定

```
//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)
{
	//CMT0を設定する
	R_PG_Timer_Start_CMT_U0_C0();
}

void func2(void)
{
	//CMT0のカウンタ値を設定
	R_PG_Timer_SetCounterValue_CMT_U0_C0( 0 );

	return;
}
```

4.7.6 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_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上でコンペアマッチ割り込み関数名に Cmt0IntFunc を指定

```
//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 func(void)
{
    //CMT0を設定する
    R_PG_Timer_Start_CMT_U0_C0();
}

void Cmyt0IntFunc(void)
{
    func_cmt();     //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();
}
```

4.8 シリアルコミュニケーションインタフェース（SCI）

4.8.1 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 to 6

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 to 6

使用RPDL関数　R_SCI_Create

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、使用する端子の入出力方向、入力バッファを設定します。同時に、使用する端子の他の機能を無効にします。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「4.11 通知関数に関する注意事項」の内容に注意してください。
- 本関数によりTXD5端子からCS4またはCS7の外部バス出力を停止することはできません。SCI5チャネルを使用する場合、TXD5端子をCS4#_DまたはCS7#_D端子として使用することはできません。

使用例　SCI0をGUI上で設定

```
//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 func(void)
{
    R_PG_Clock_Set();       //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();      //SCI0を設定
}
```

4.8.2 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 to 6

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 to 6

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「4.11 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_<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"
uint8_t data[255];
void func(void)
{
    R_PG_Clock_Set();           //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_Send_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();
}
```

4.8.3 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 to 6

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 to 6

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。
- 送信データをDMACで転送する場合の使用方法は、R_PG_DMAC_Set_C〈*チャネル番号*〉の使用例2を参照してください。

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_<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"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();           //あらかじめクロック発生回路を設定する
    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
}
```

4.8.4 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 to 6

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、送信終了通知に“最終バイトの送信終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	データ数の取得が正しく行われた場合
false	データ数の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 to 6

使用RPDL関数

R_SCI_GetStatus

詳細

- GUI上で送信終了通知に“最終バイトの送信終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_<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"

uint16_t data[255];

void func(void)
{
    R_PG_Clock_Set();           //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_Send_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_StopSending_C0();  //送信を中断
    }
}
```

4.8.5 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 to 6

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 to 6

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「4.11 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_<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"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_Receive_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();
}
```

4.8.6 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 to 6

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 to 6

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。
- 受信データをDMACで転送する場合の使用方法は、R_PG_DMAC_Set_C〈*チャネル番号*〉 の使用例3を参照してください。

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

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"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_Receive_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
}
```

4.8.7 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 to 6

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 to 6

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 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_<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"
uint8_t data[255];
void func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_Receive_C0(data, 255);          //255バイトのデータを送信する
}
//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)
{
    uint8_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
    }
}
```

4.8.8 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 to 6

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、受信終了通知に“最終バイトの受信終了を関数呼び出しで通知する”を選択

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	データ数の取得が正しく行われた場合
false	データ数の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 to 6

使用RPDL関数 R_SCI_GetStatus

詳細
- GUI上で受信終了通知に“最終バイトの受信終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_<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"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();           //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_Receive_C0(data, 255);         //255バイトのデータを送信する
}

//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_StopReceiving_C0();    //受信を中断
    }
}
```

4.8.9 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 to 6

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	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_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0 to 6

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_<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"

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();		//あらかじめクロック発生回路を設定する
	R_PG_SCI_Set_C0();		//SCI0を設定
	R_PG_SCI_Receive_C0(data, 1);		//1バイトのデータを送信する
}

//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 );
}
```

4.8.10 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 to 6

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

戻り値

true	送信状態の取得が正しく行われた場合
false	送信状態の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 to 6

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。

送信終了フラグ

0	送信中
1	送信終了

使用例 R_PG_DMAC_Set_C〈チャネル番号〉 の使用例2を参照してください。

4.8.11 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 to 6

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_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0 to 6

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SCI0の受信機能を設定
受信終了通知に“最終バイトの受信終了まで受信関数内で待つ”を選択

```
//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"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_Receive_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
}
```

4.9 I2Cバスインタフェース（RIIC）

4.9.1 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*〉: 0,1

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*〉: 0,1

使用RPDL関数　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、使用する端子の入出力方向、入力バッファを設定します。同時に、使用する端子の他の機能を無効にします。
- 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_<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 func(void)
{
    R_PG_Clock_Set();      //あらかじめクロック発生回路を設定する
    R_PG_I2C_Set_C0();     //RIIC0を設定
}
```

4.9.2 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*〉(uint16_t slave, uint8_t* data, uint16_t count)
〈*チャネル番号*〉: 0,1

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

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*〉: 0,1

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「4.11 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8～1ビットが出力されます。10ビットアドレスの場合は10～9ビットと、8～0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。

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_<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"

//受信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        6,      //スレーブアドレス
        &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();
}
```

4.9.3 R_PG_I2C_MasterReceiveLast_C〈チャネル番号〉

bool R_PG_I2C_MasterReceiveLast_C〈チャネル番号〉
(uint8_t* data)
〈チャネル番号〉: 0,1

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 slave	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〉: 0,1

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 ： RXI0(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_<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 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)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //DMAC0を設定
    R_PG_DMAC_Set_C0();

    //DMAC0を設定
    R_PG_DMAC_Activate_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        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)
    );
}
```

4.9.4 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〉(uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0,1

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

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〉: 0,1

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「4.11 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8～1ビットが出力されます。10ビットアドレスの場合は10～9ビットと、8～0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。

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_<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"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSend_C0(
        6,      //スレーブアドレス
        &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();
}
```

4.9.5 R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉

定義

R_PG_I2C_MasterSendWithoutStop_C〈*チャネル番号*〉
(uint16_t slave, uint8_t* data, uint16_t count)
〈*チャネル番号*〉: 0,1

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

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*〉: 0,1

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「4.11 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8～1ビットが出力されます。10ビットアドレスの場合は10～9ビットと、8～0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。

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_<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"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSendWithoutStop_C0(
        6,      //スレーブアドレス
        &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();
}
```

4.9.6 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*〉: 0,1

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*〉: 0,1

使用RPDL関数 R_IIC_Control

詳細

- R_PG_I2C_MasterReceiveWithoutStop_C〈*チャネル番号*〉 または 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_<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"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSendWithoutStop_C0(
        6,      //スレーブアドレス
        &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();
}
```

4.9.7 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*〉: 0,1

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_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0,1

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_<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"
//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
uint8_t busy;
void func(void)
{
	//あらかじめクロック発生回路を設定する
	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(
		6,	//スレーブアドレス
		&data,	//送信データの格納先アドレス
		10	//送信データ数
	);
}
```

4.9.8 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〉: 0,1

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〉: 0,1

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「4.11 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。

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_<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"
//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)
{
	//あらかじめクロック発生回路を設定する
	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(
		&data,	//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
		);
	}
}
```

4.9.9 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〉: 0,1

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〉: 0,1

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。

4.9.10 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〉: 0,1

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_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0,1

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。

4.9.11 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〉: 0,1

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	送信モードフラグの格納先

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_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0,1

使用RPDL関数 R_IIC_GetStatus

詳細

- 送信/受信モードを取得します。

送信モードフラグ

0	受信モード
1	送信モード

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。

4.9.12 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〉: 0,1

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_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0,1

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。

4.9.13 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〉: 0,1

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_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0,1

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_<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"

//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)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        6,      //スレーブアドレス
        &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 );
}
```

4.9.14 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〉: 0,1

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_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0,1

使用RPDL関数 R_IIC_GetStatus

詳細

- 現在の転送で送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_<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"

//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)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSend_C0(
        6,      //スレーブアドレス
        &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 );
}
```

4.9.15 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*〉: 0,1

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*〉: 0,1

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_<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"

//送信データの格納先
uint8_t iic_data[10];

void func(void)
{
	//あらかじめクロック発生回路を設定する
	R_PG_Clock_Set();

	//RIIC0を設定
	R_PG_I2C_Set_C0();

	//マスタ送信
	R_PG_I2C_MasterSend_C0(
		6,	//スレーブアドレス
		&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();
	}
}
```

4.9.16 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*〉: 0,1

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_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0,1

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_<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"

//受信データの格納先
uint8_t iic_data[10];

void func(void)
{
    //あらかじめクロック発生回路を設定する
    R_PG_Clock_Set();

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        6,      //スレーブアドレス
        &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();
}
```

4.10 A/D変換器

4.10.1 R_PG_ADC_10_Set_AD〈*ユニット番号*〉

定義　bool R_PG_ADC_10_Set_AD〈*ユニット番号*〉(void)　〈*ユニット番号*〉： 0～4

概要　A/D変換器の設定

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_ADC_10_AD〈*ユニット番号*〉.c　〈*ユニット番号*〉： 0～4

使用RPDL関数　R_ADC_10_Create

詳細

- A/D変換器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_10_StartConversionSW_AD〈*チャネル番号*〉により変換を開始します。
- 本関数内ではサンプリング時間の設定のために、クロック周波数を使用します。クロック発生回路をリセット後の初期状態で使用する場合も、必ず本関数を呼び出すより先に、
- R_PG_Clock_Set を呼び出してクロックを設定してください。
  アナログ入力端子として使用する端子の入出力方向を入力に設定し、インプットバッファを無効にします。
- 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が指定した場合、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「4.11 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AD2を設定
GUI上でA/D変換終了割り込み通知関数名に Ad2IntFunc を設定

```
//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"

uint16_t data;   //A/D変換結果の格納先

void func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_ADC_10_Set_AD2();         //AD2を設定
}

//AD変換終了割り込み通知関数
void Ad2IntFunc(void)
{
    R_PG_ADC_10_GetResult_AD2(&data)      //A/D変換結果の取得
}
```

4.10.2 R_PG_ADC_10_StartConversionSW_AD〈*ユニット番号*〉

定義　bool R_PG_ADC_10_StartConversionSW_AD〈*ユニット番号*〉(void)
〈*ユニット番号*〉： 0～4

概要　A/D変換の開始 (ソフトウェアトリガ)

生成条件　A/D変換器が設定され、起動要因にソフトウェアトリガが指定された場合

引数　なし

戻り値

true	変換開始が正しく行われた場合
false	変換開始に失敗した場合

出力先ファイル　R_PG_ADC_10_AD〈*ユニット番号*〉.c
〈*ユニット番号*〉： 0～4

使用RPDL関数　R_ADC_10_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AD2のモードを連続スキャンモードで設定

```
//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 func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_ADC_10_Set_AD2();         //AD2を設定

    //ソフトウェアトリガによりAD変換開始
    R_PG_ADC_10_StartConversionSW_AD2();
}
```

4.10.3 R_PG_ADC_10_StopConversion_AD〈ユニット番号〉

定義　bool R_PG_ADC_10_StopConversion_AD〈ユニット番号〉(void)
〈ユニット番号〉： 0～4

概要　A/D変換の中断

引数　なし

戻り値

true	変換停止が正しく行われた場合
false	変換停止に失敗した場合

出力先ファイル　R_PG_ADC_10_AD〈ユニット番号〉.c
〈ユニット番号〉： 0～4

使用RPDL関数　R_ADC_10_Control

詳細

- 連続スキャンモードのA/D変換を停止します。シングルモードおよび1サイクルスキャンモードでは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変換を停止した後、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_10_StopModule_AD〈ユニット番号〉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AD2の起動要因をソフトウェアトリガ指定して設定

```
//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"

uint16_t data;   //A/D変換結果の格納先

void func1(void)
{
      R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
      R_PG_ADC_10_Set_AD2();         //AD2を設定
}

void func2(void)
{
      //連続スキャンを停止
      R_PG_ADC_10_StopConversion_AD2();

      //A/D変換結果の取得
      R_PG_ADC_10_GetResult_AD2(&data)
}
```

4.10.4 R_PG_ADC_10_GetResult_AD_AD〈*ユニット番号*〉

定義　bool R_PG_ADC_10_GetResult_AD〈*ユニット番号*〉(uint16_t * data)

〈*ユニット番号*〉： 0～4

概要　A/D変換結果の取得

引数

uint16_t * data	A/D変換結果の格納先

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_ADC_10_AD〈*ユニット番号*〉.c　〈*ユニット番号*〉： 0～4

使用RPDL関数　R_ADC_10_Read

詳細

- 取得するデータの数は、使用するA/D変換チャネルの数に依ります。使用するチャネルのA/D変換結果を格納するのに必要な領域を確保してください。
- GUI上で割り込み通知関数名を指定していない場合、本関を呼び出した時点でA/D変換が終了していないときは、結果を読み出す前にA/D変換が終了するまで関数内で待ちます。A/D変換開始トリガが入力されない場合、本関数から処理が戻りません。A/D変換器のレジスタがプログラムにより変更された場合、本関数はロックします。

使用例

GUI上でAD0を1サイクルスキャンモードで設定
AN0-AN3の4チャネルを使用
GUI上でA/D変換終了割り込み通知関数名に Ad0IntFunc を設定

```
//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 func(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_ADC_10_Set_AD0();         //AD0を設定
}

//AD変換終了割り込み通知関数
void Ad0IntFunc(void)
{
    uint16_t data[4];   //全チャネルのA/D変換結果の格納先
    uint16_t data_an0;  //AN0のA/D変換結果の格納先
    uint16_t data_an1;  //AN1のA/D変換結果の格納先
    uint16_t data_an2;  //AN2のA/D変換結果の格納先
    uint16_t data_an3;  //AN3のA/D変換結果の格納先

    //A/D変換結果の取得
    R_PG_ADC_10_GetResult_AD2(&data)

    data_an0 = data[0];
    data_an1 = data[1];
    data_an2 = data[2];
    data_an3 = data[3];
}
```

4.10.5 R_PG_ADC_10_StopModule_AD〈ユニット番号〉

定義　bool R_PG_ADC_10_StopModule_AD〈ユニット番号〉(void)
〈ユニット番号〉 : 0～4

概要　A/D変換器の停止

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_ADC_10_AD〈ユニット番号〉.c
〈ユニット番号〉 : 0～4

使用RPDL関数　R_ADC_10_Destroy

詳細
- A/D変換器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。

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"

uint16_t data;  //A/D変換結果の格納先

void func1(void)
{
    R_PG_Clock_Set();              //あらかじめクロック発生回路を設定する
    R_PG_ADC_10_Set_AD2();         //AD2を設定
}

void func2(void)
{
    //連続スキャンを停止
    R_PG_ADC_10_StopConversion_AD2();

    //AD変換結果の取得
    R_PG_ADC_10_GetResult_AD2(&data)

    //AD変換器を停止
    R_PG_ADC_10_StopModule_AD2();
}
```

4.11 通知関数に関する注意事項

4.11.1 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。PDG の出力関数および RPDL の関数はユーザモードで実行されますが、各通知関数は RPDL 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 RPDL 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。
- PDG の出力関数および RPDL 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。

4.11.2 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)

RPDL 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 RPDL 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。

5.生成ファイルのHEWへの登録とビルド

PDG で生成したファイルの HEW 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。

- PDG 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。HEW 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。
- PDG が HEW 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。HEW 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、PDG から HEW にソースファイルを登録する際、intprg.c と vecttbl.c はビルドの対象から除外されます。
- PDG が HEW 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、PDG のソース生成時に上書きされます。
- RPDL 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。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 RPDL ライブラリのソースファイルを利用してください。

6.チュートリアル

本章では、PDG と HEW を使用して RX610 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、PDG の使用手順を紹介します。

- TMR の割り込みで RSK の LED を点滅
- A/D 変換の連続スキャン
- TPU で PWM パルス出力
- I2C チャネル 0 とチャネル 1 で通信

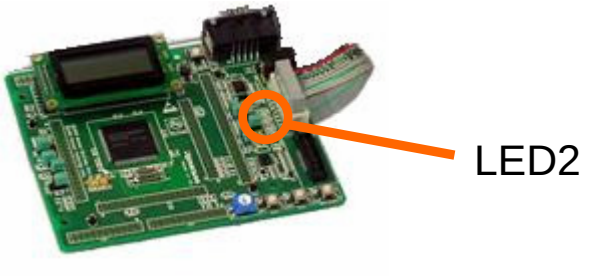
説明の中にある以下の表示はそれぞれ PDG、HEW 上での操作をあらわします。

PDG	: PDG上の操作をあらわします
HEW	: HEW上の操作をあらわします

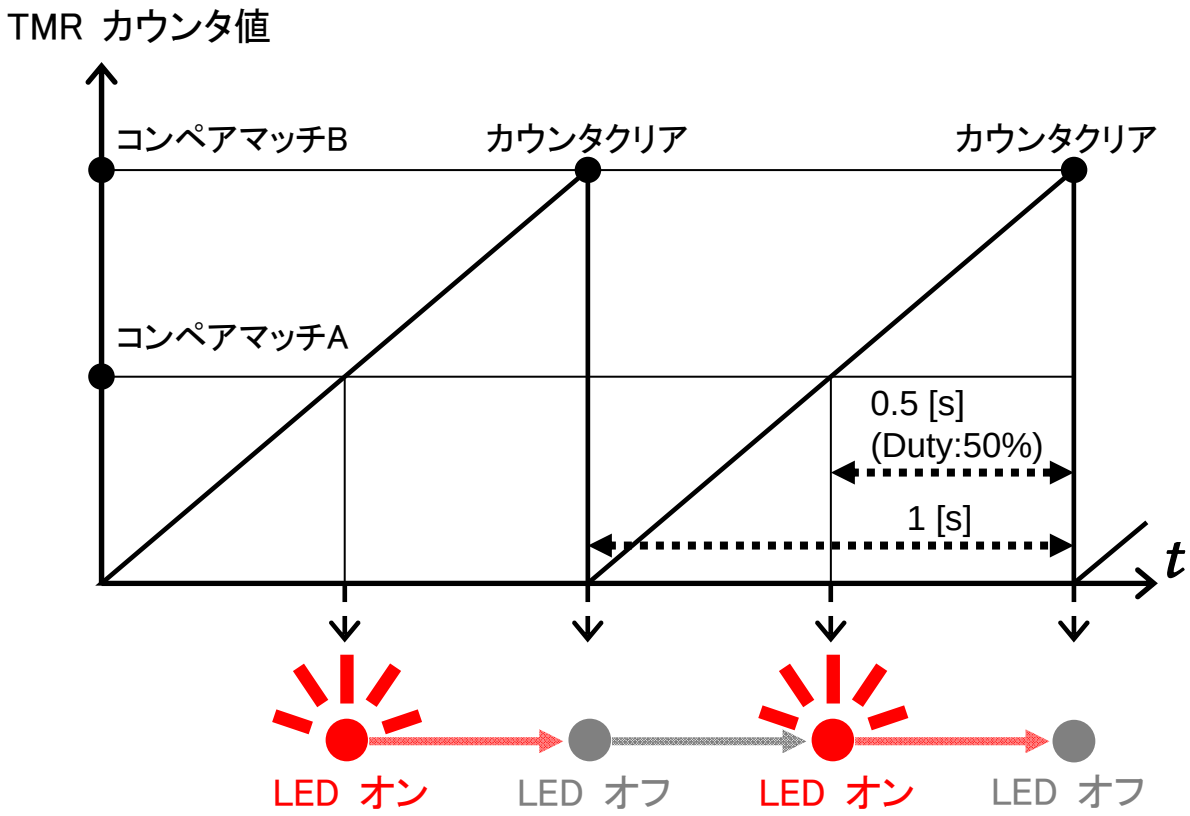
6.1 TMRの割り込みでRSKのLEDを点滅

RSK ボード上の LED2 は P33 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。

LED2はP33から0出力で点灯、1出力で消灯します

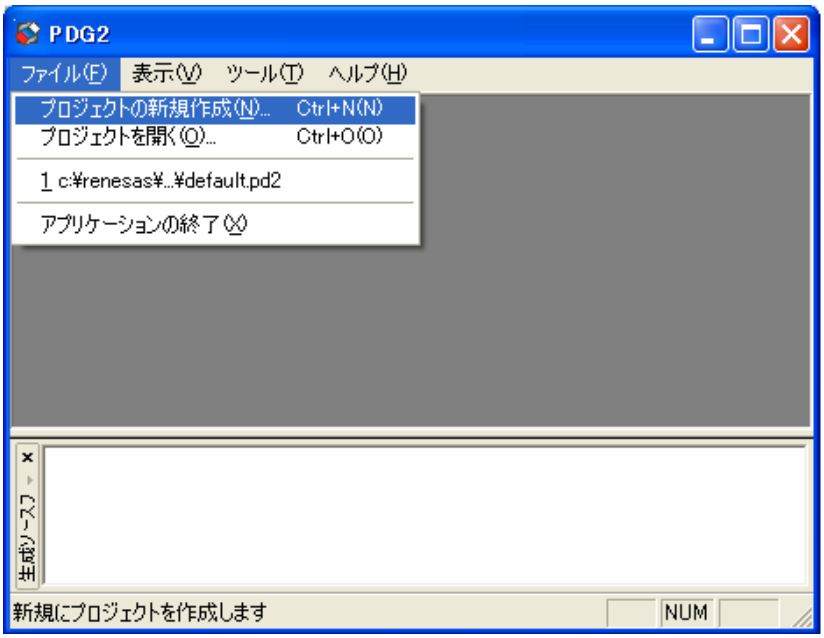


・コンペアマッチAで
LED点灯
・コンペアマッチBで
LED消灯
・コンペアマッチBで
カウンタクリア

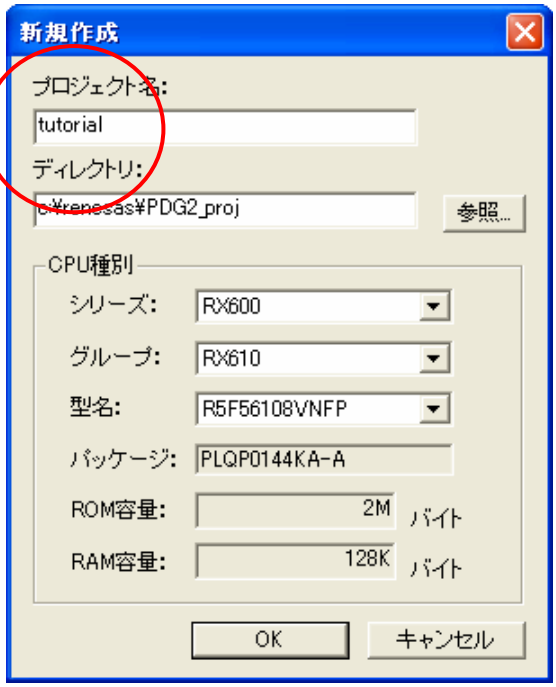


(1) PDG プロジェクトの作成 **PDG**

1. PDG 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。

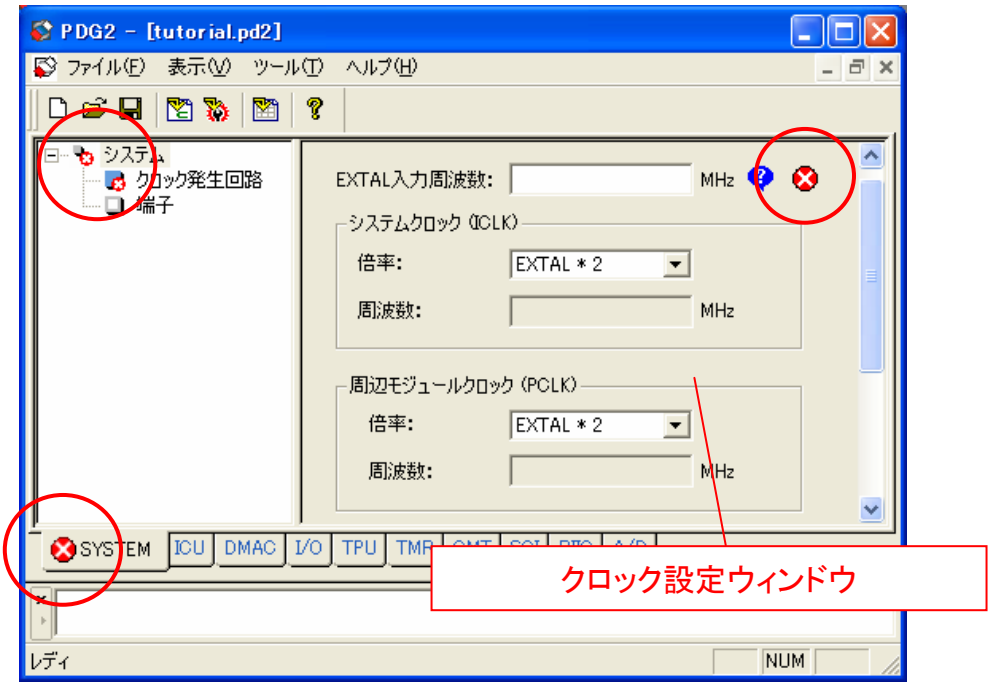


3. プロジェクト名に“tutorial”を指定してください。

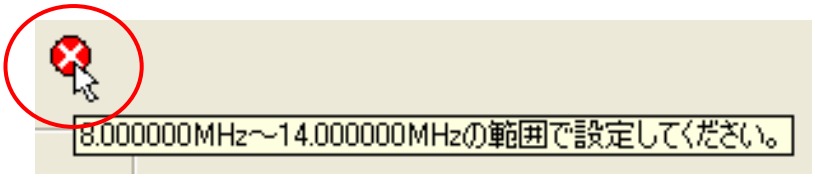


(2) 初期状態 **PDG**

・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。



PDG 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。

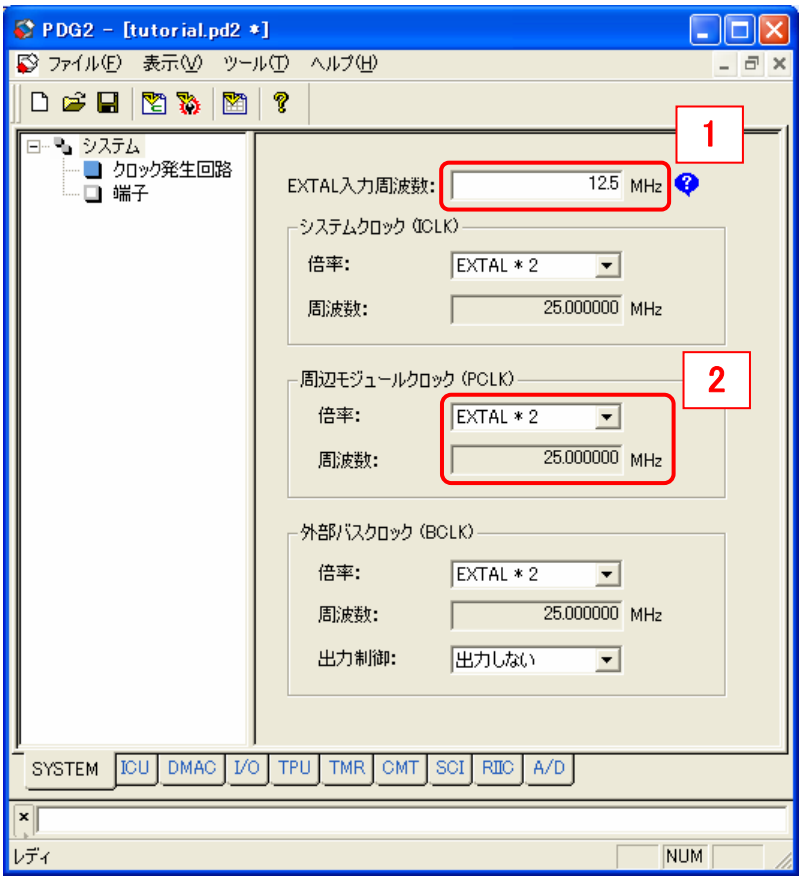
警告
設定は可能ですが、誤っている可能性があります。
ソースファイルの生成は可能です。

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**

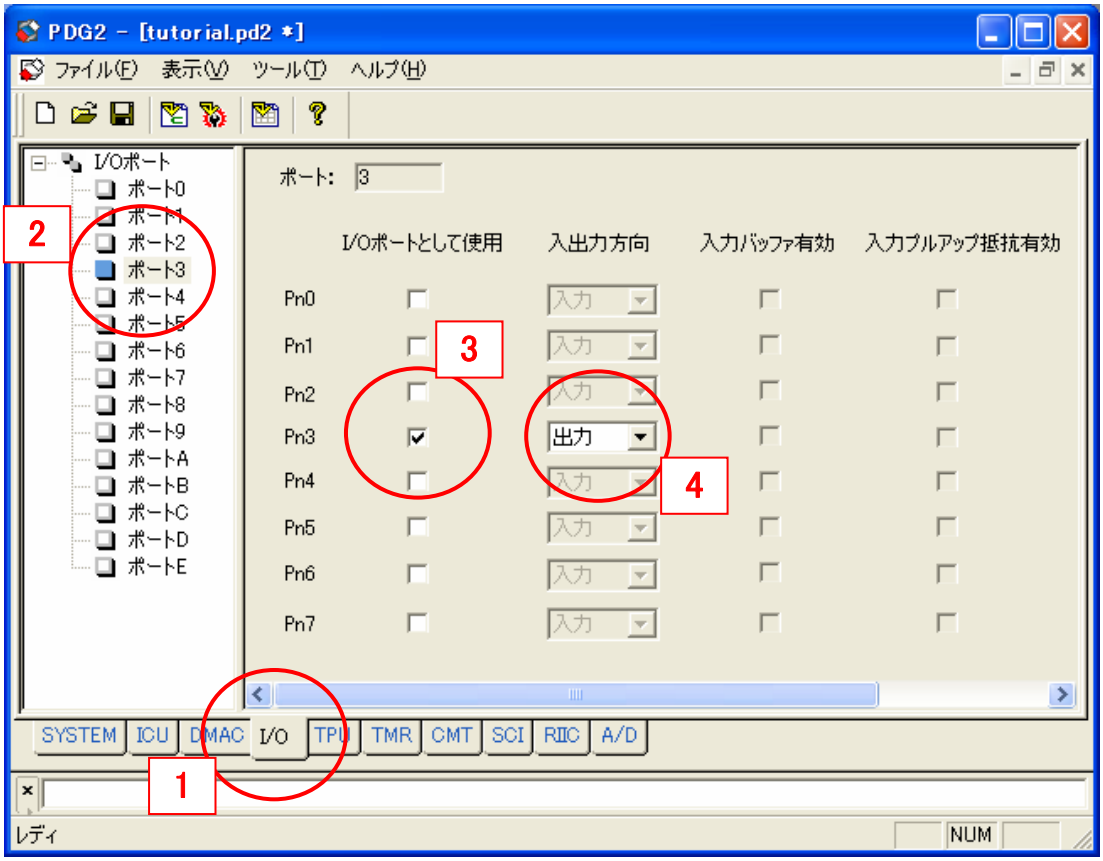
1. 最初に EXTAL 入力周波数を設定してください。
   RSK ボードの外部入力周波数は 12.5MHz です。“12.5”と入力してください。
2. PCLK は 25MHzで使用します。
   PCLK の倍率に“EXTAL *2”を選択し、PCLK 周波数を 25MHzに設定してください。



(4) I/O ポートの設定 PDG

LED2 が接続されている P33 を出力ポートに設定します。

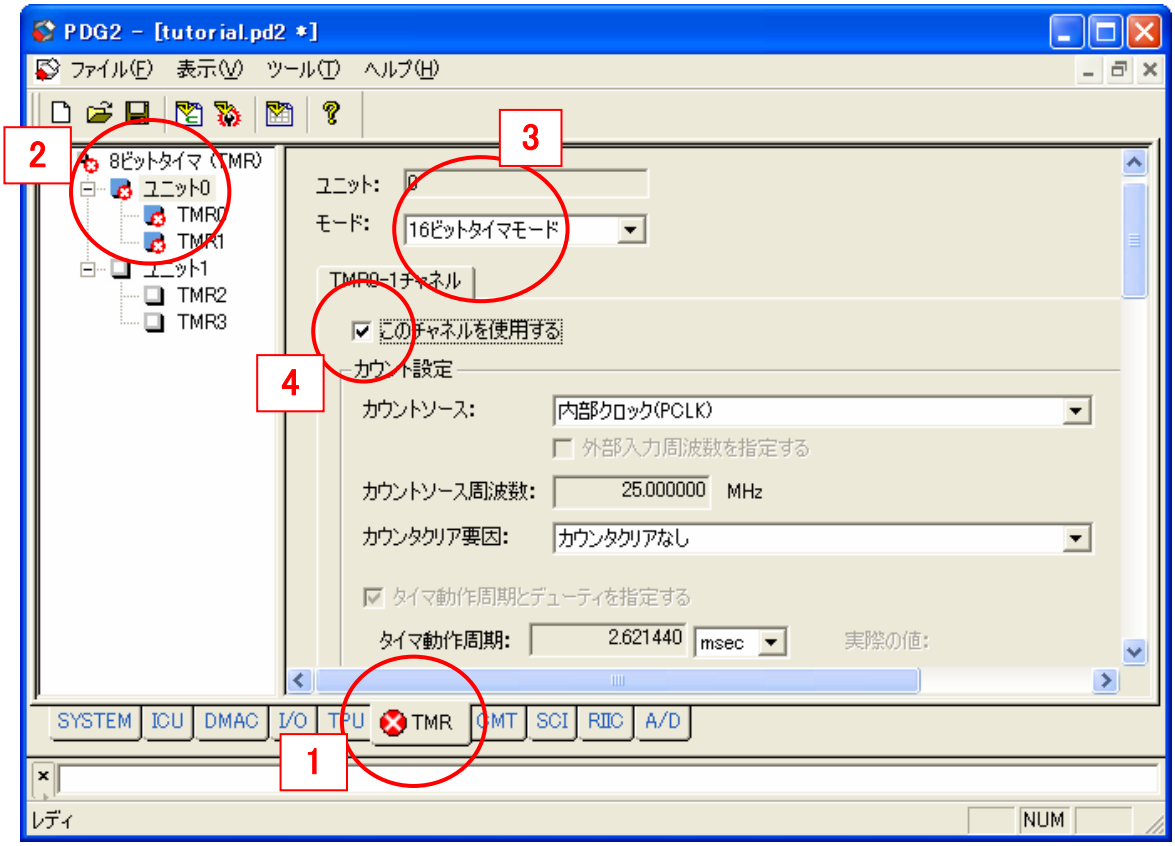
1. [I/O] タブを選択してください
2. [ポート3] を選択してください
3. [Pn3] をチェックしてください
4. [出力] を選択してください



(5) TMR の設定-1 PDG

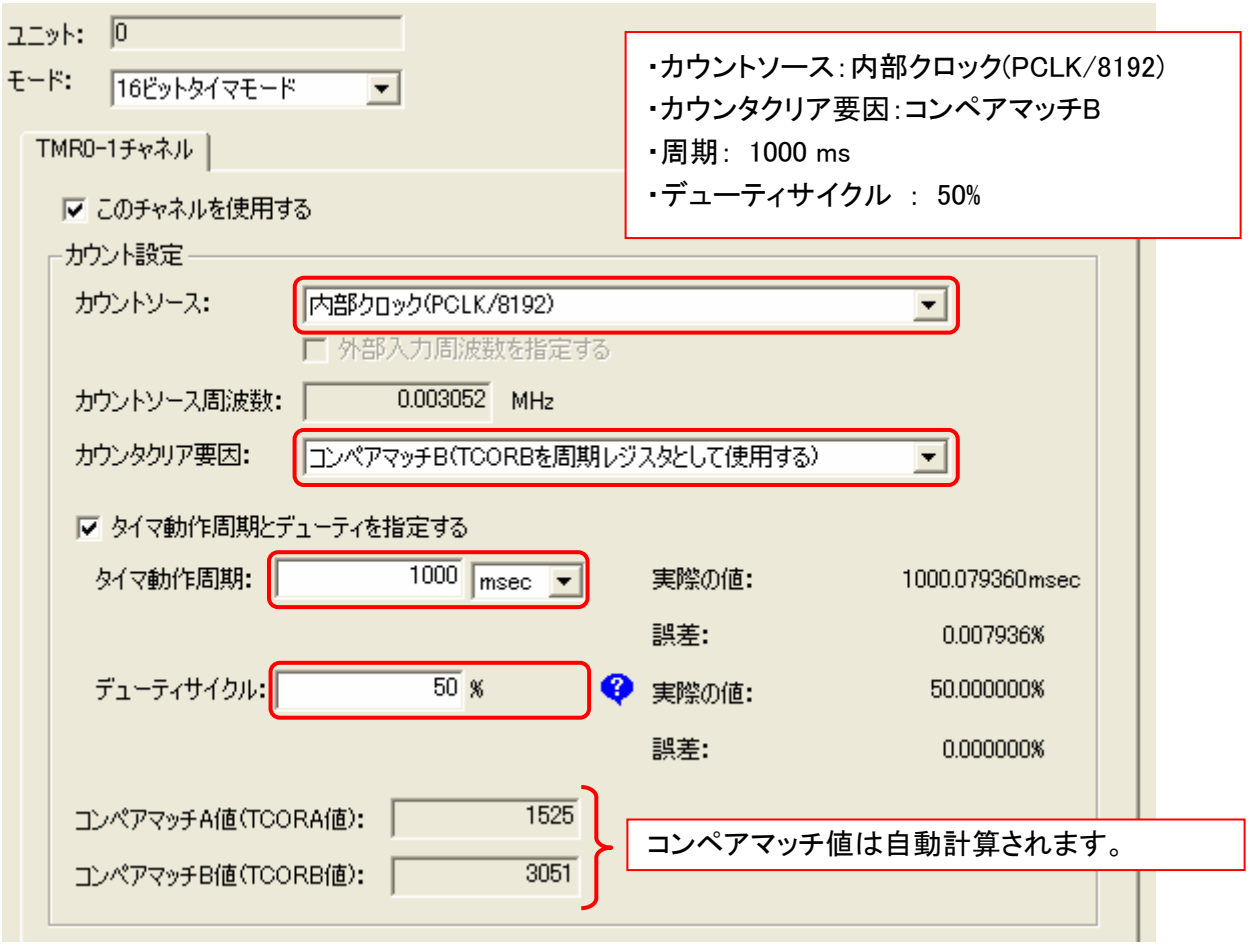
このチュートリアルでは TMR (8 ビットタイマ) のユニット 0 を 16 ビットモード (2 つの 8 ビットタイマをカスケード接続するモード)で使用します。

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。



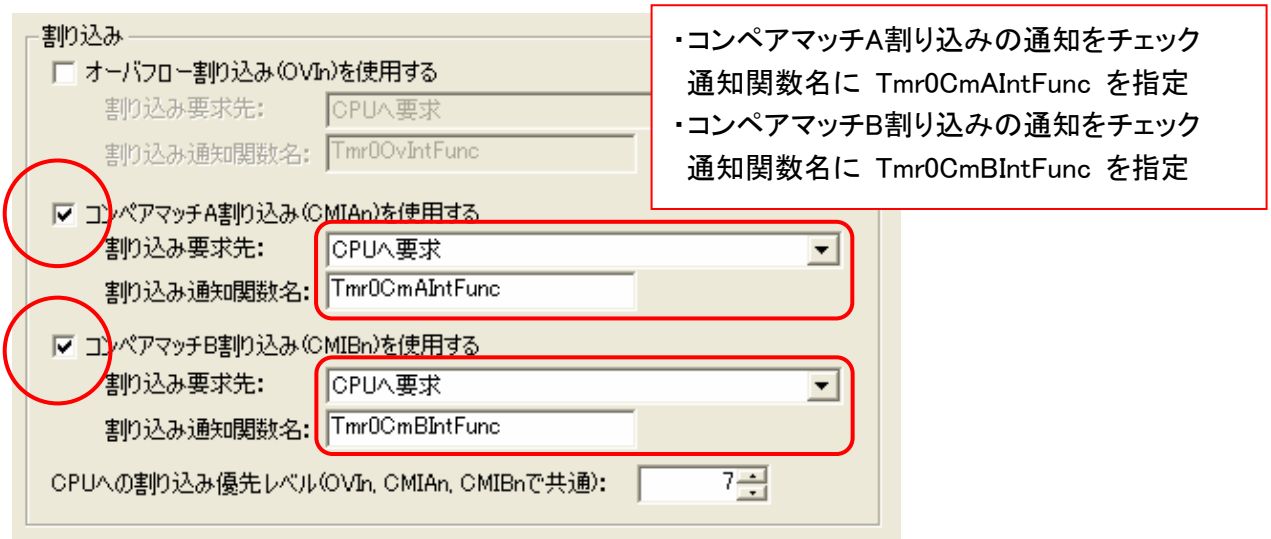
(6) TMR の設定-2 PDG

TMR の他の項目を以下の通り設定してください。



(7) TMR の設定-3 PDG

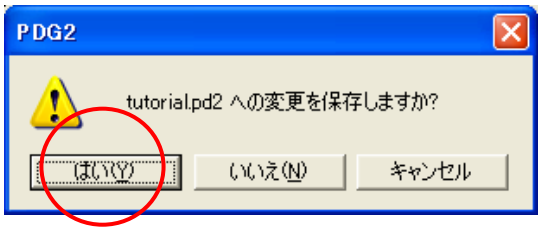
割り込み通知関数を設定します。
これらの関数は割り込みが発生すると呼ばれます。



(8) ソースファイルの生成 **PDG**

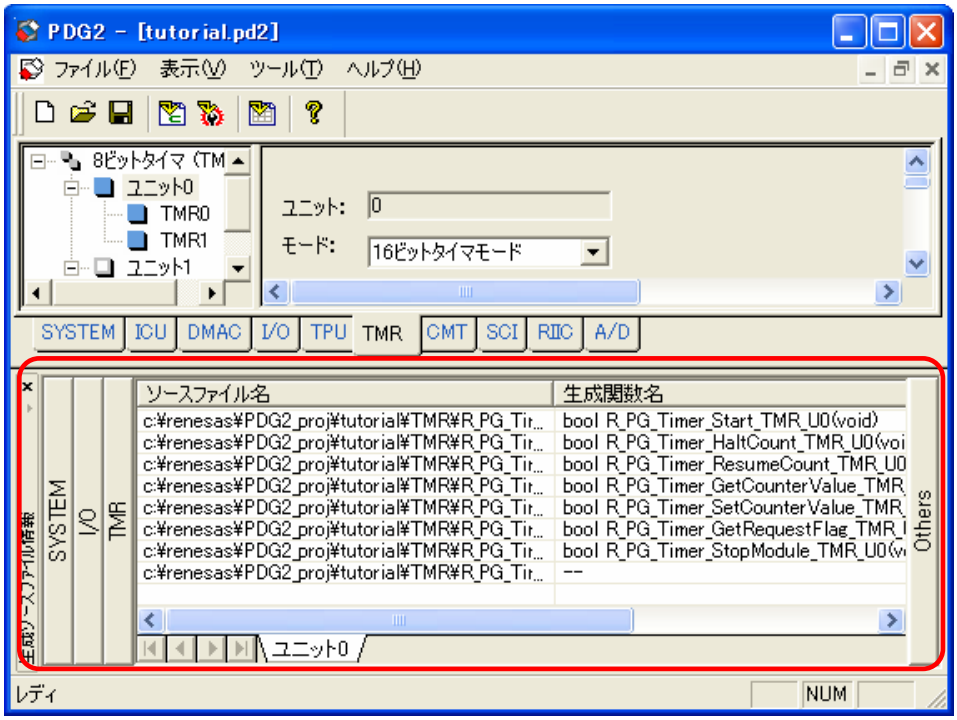
1. ツールバー上の  をクリックするとソースファイルが生成されます。

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。

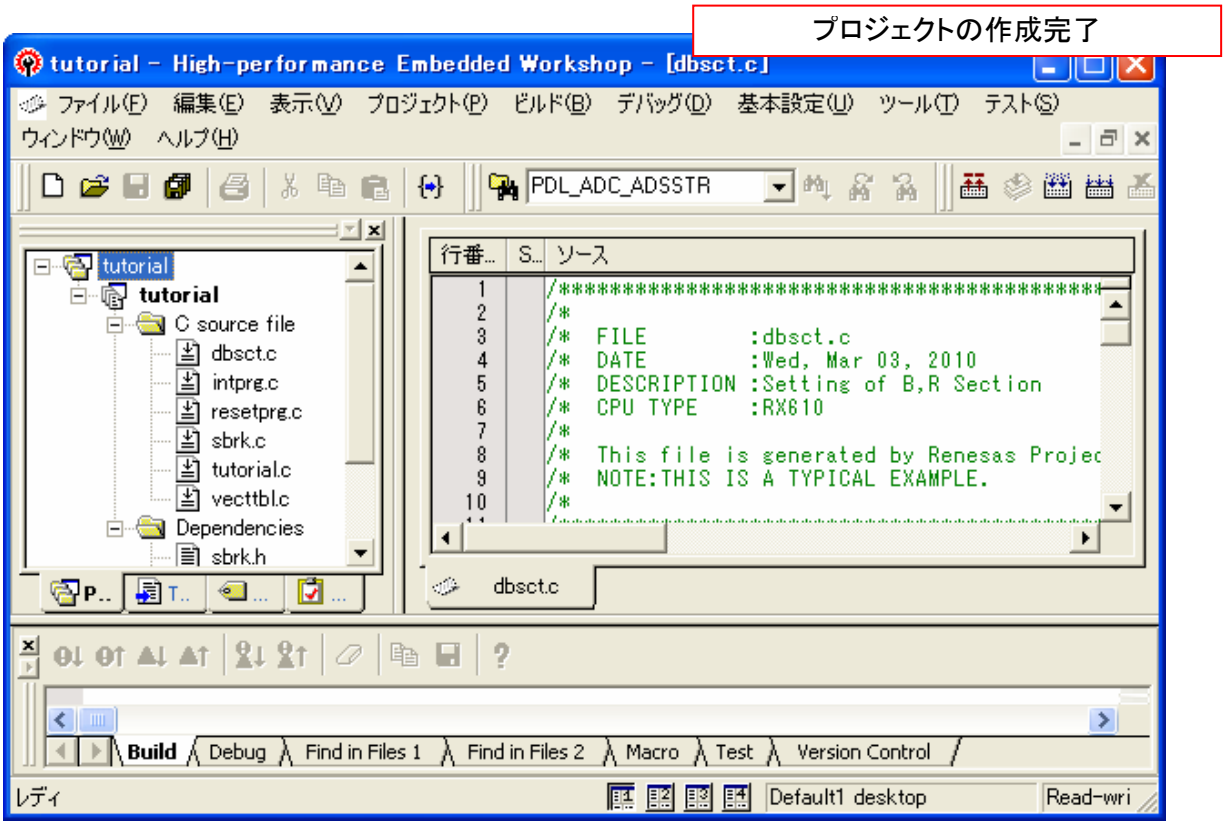
4. 生成された関数が下部のウィンドウに表示されます。
関数をダブルクリックするとソースファイルが開きます。



(9) HEW プロジェクトの準備 **HEW**

HEW を起動し、RX610 用の新規ワークスペースを作成します。


新規プロジェクト-8/10-デバッガ
ターゲットエミュレータを指定
ターゲット：
RX600 E1/E20 SYSTEM
RX600 Simulator
ターゲットタイプ：
RX600
ターゲットCPU：
All CPUs
< 戻る(B)
次へ(N) >
完了
キャンセル


プロジェクトの作成完了
tutorial - High-performance Embedded Workshop - [dbsct.c]
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)
PDL_ADC_ADSSTR
tutorial
C source file
dbsct.c
intprg.c
resetprg.c
sbrk.c
tutorial.c
vecttbl.c
Dependencies
sbrk.h
行番... S... ソース
/* FILE :dbsct.c
/* DATE :Wed, Mar 03, 2010
/* DESCRIPTION :Setting of B,R Section
/* CPU TYPE :RX610
/* This file is generated by Renesas Projec
/* NOTE:THIS IS A TYPICAL EXAMPLE.
dbsct.c
Build Debug Find in Files 1 Find in Files 2 Macro Test Version Control
レディ
Default1 desktop
Read-wri

(10) プログラムの作成 HEW

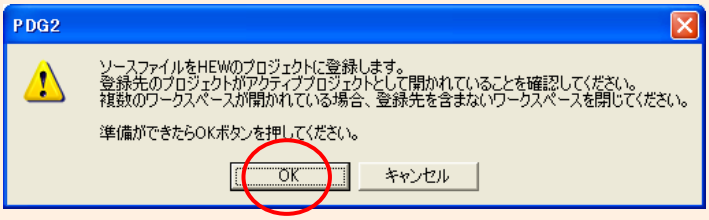
HEW 上で以下のプログラムを作成してください。

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

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

    //ポートP33の設定
    R_PG_IO_PORT_Set_P3();

    //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_P33(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_P33(1);
}
```

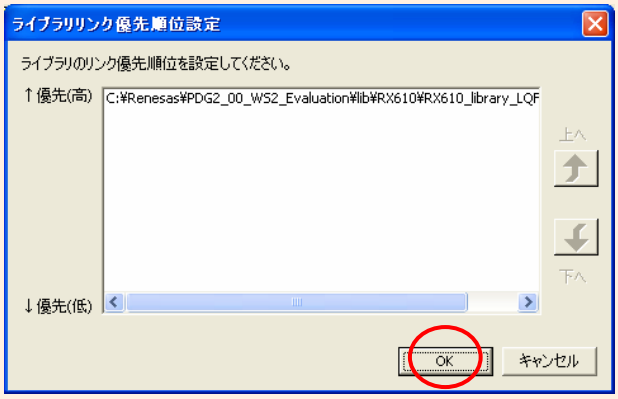
(11) PDG 生成ファイルの HEW への登録

PDG

1. ファイルを HEW に追加するには
   PDG 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。

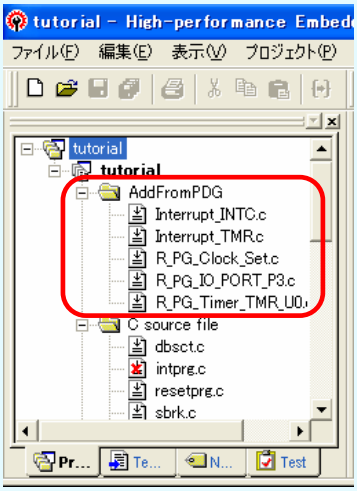


3. RPDL とのリンク設定のためのダイアログが開きます。
   複数の lib ファイルとリンクする場合、このダイアログ上でリンク順を設定できます。



HEW

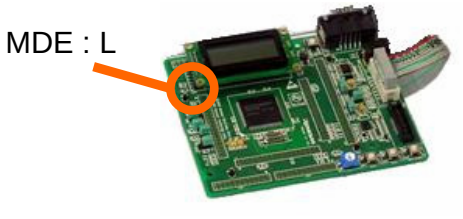
4. HEW のプロジェクトにファイルが追加されます。
   追加されたファイルは
   AddFromPDG
   フォルダに格納されます。



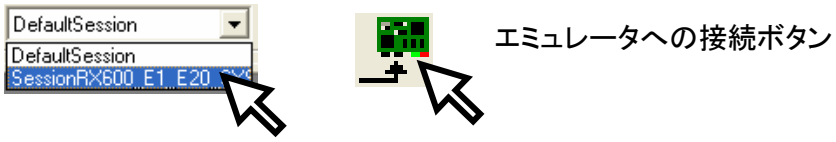
ソースファイルはHEW Target Server経由で追加されます。追加を実行する前にHEW Target Serverが設定されていることを確認してください。詳細についてはPDGのユーザーズマニュアルを参照してください。

(12) エミュレータの接続、プログラムのビルド、実行 **HEW**

1. エミュレータを接続する前に、RSKボード上のMDEがL(CPUはリトルエンディアン)にセットされていることを確認してください。



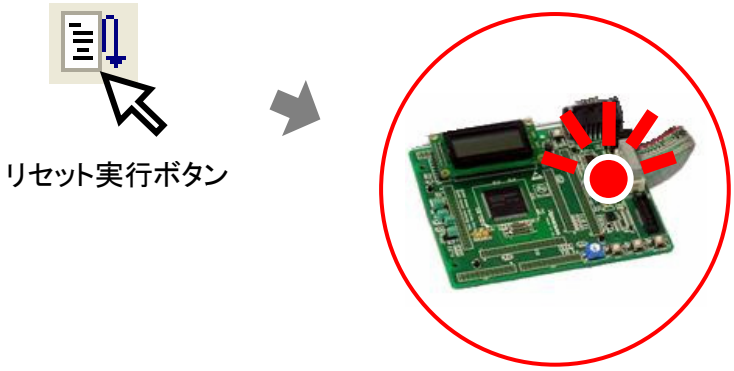
2. エミュレータに接続してください。



3. RPDL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。

4. プログラムをダウンロードしてください。

5. 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。



6.2 A/D変換の連続スキャン

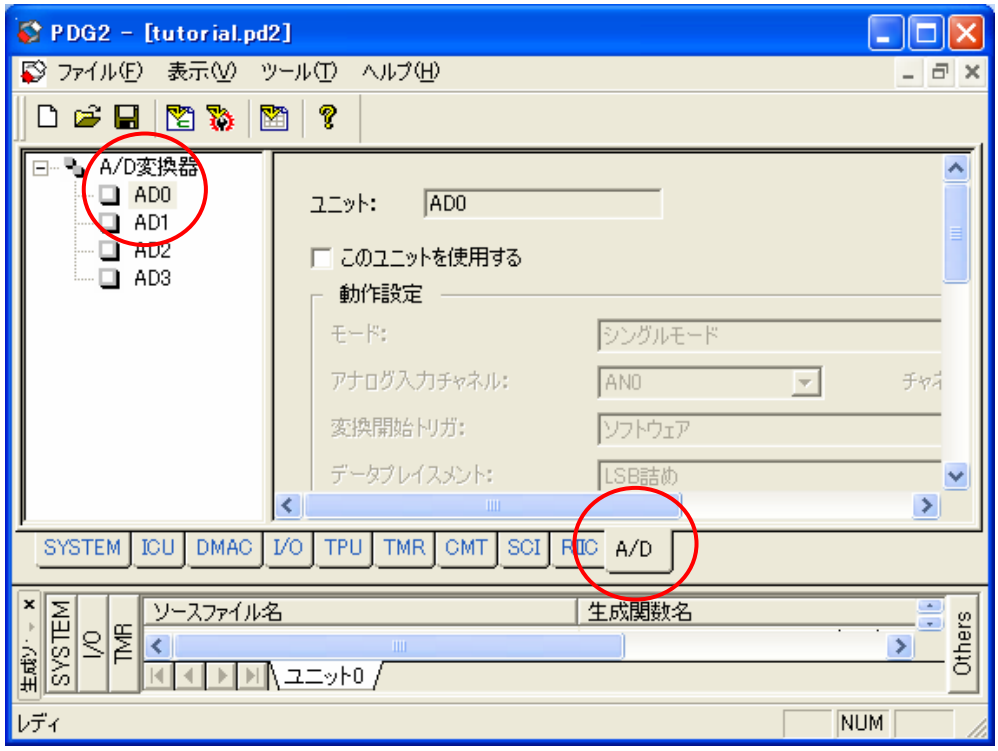
RX610 RSK ボードではポテンショメータが AN0 アナログ入力端子に接続されています。

このチュートリアルでは AD0 の A/D 変換を連続スキャンし、A/D 変換結果を HEW 上でリアルタイムに確認します。

(HEW と PDG のプロジェクトは「6.1 TMR の割り込みで RSK の LED を点滅」で作成したものを使用します。)

(1) A/D 変換器の設定-1 PDG

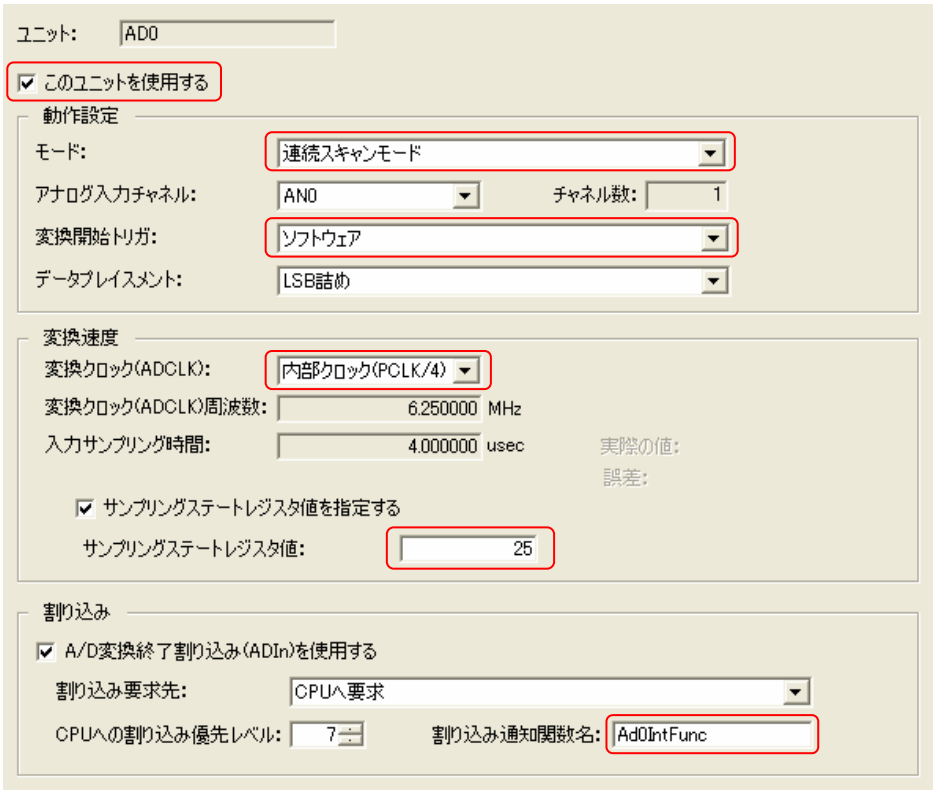
A/D タブを選択し、ツリー表示上で AD0 を選択してください。



(2) A/D 変換器の設定-2 PDG

AD0 を以下の通り設定してください。

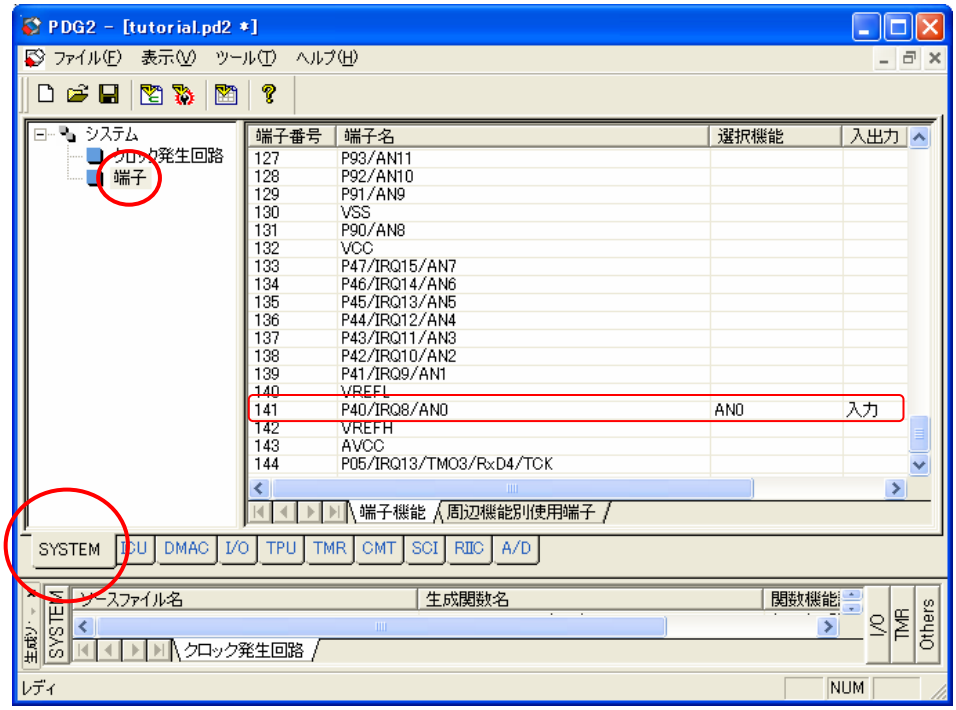
1. [このユニットを使用する]をチェック
2. モードは[連続スキャンモード]を選択
3. 変換開始トリガは[ソフトウェア]
4. 変換クロックは [内部クロック(PCLK/4)]
5. サンプリングステートレジスタ値は初期値(25)
6. A/D変換終了割り込み通知関数名に
   Ad0IntFunc を指定



(3) 端子使用状況の確認 PDG

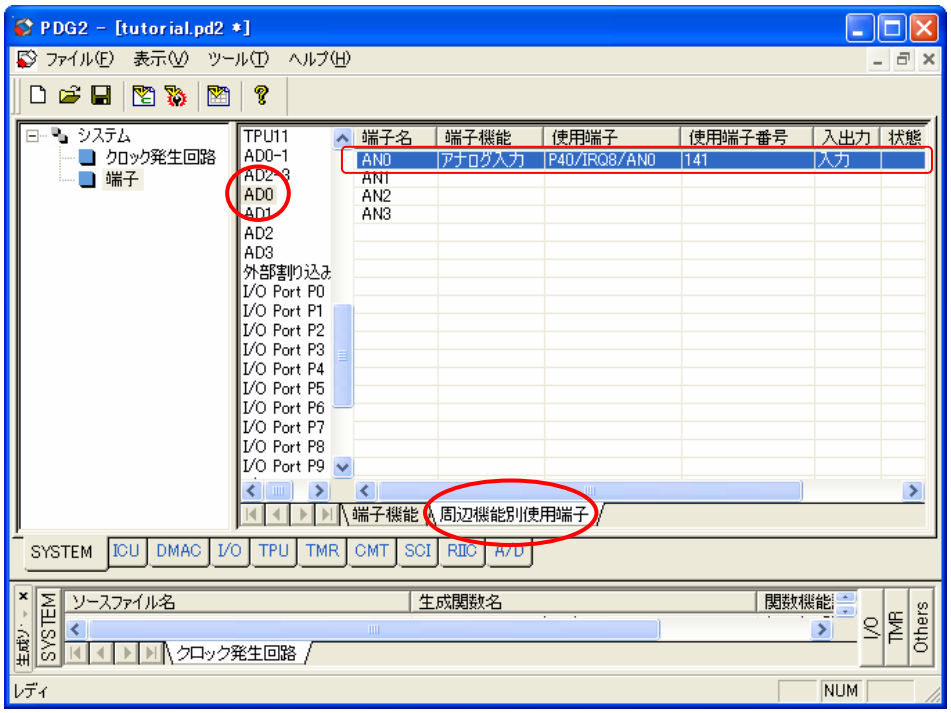
・端子機能ウィンドウで端子の使用状況を確認することができます。

1. AD0を設定後、[SYSTEM]タブを選択し、ツリー表示上で[端子]を選択してください。
2. [端子機能]ウィンドウ上で No.141 ピンが AN0 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AD0を選択してAN0端子の使用状況を確認してください。



(4) プログラムの作成 **HEW**

HEW 上で以下のプログラムを作成してください。

1. main関数を次のように変更してください。グレーの部分は「6.1 TMRの割り込みでRSKのLEDを点滅」で作成したものを使用してください。

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

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

    //A/Dコンバータ AD0の設定
    R_PG_ADC_10_Set_AD0();

    //ポートP33の設定
    R_PG_IO_PORT_Set_P3();

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

    //AD0のA/D変換開始
    R_PG_ADC_10_StartConversionSW_AD0();

    while(1);
}
```

2.次の通知関数を追加してください。

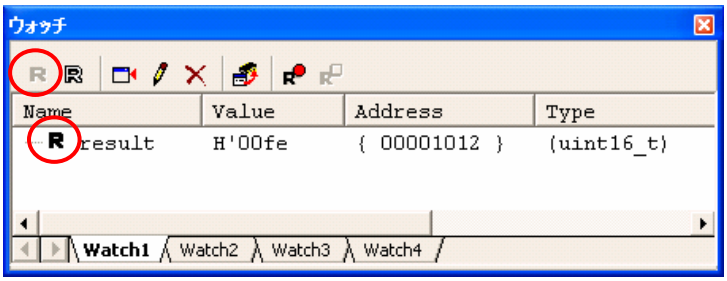
```
//変換結果格納先変数
uint16_t result;

//AD0変換終了割り込み通知関数
void Ad0IntFunc(void)
{
    //変換結果の取得
    R_PG_ADC_10_GetResult_AD0(&result);
}
```

(5) ソースファイルの生成と HEW への登録 **PDG**

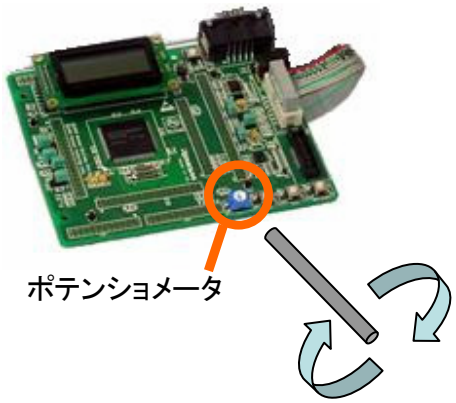
ソースファイルを生成し、HEW に登録してください。 (6.1 (8)(11)参照)

(6) プログラムのビルドと実行 **HEW**

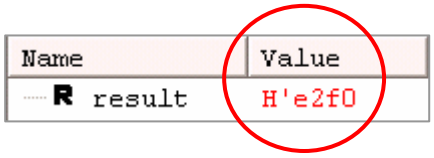
1. プログラムをビルドし、ダウンロードしてください。
2. ウォッチウィンドウを開き、変数 “result” を登録してください。
   “result”をリアルタイム更新に設定してください。



3. プログラムを実行してください。
4. 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。



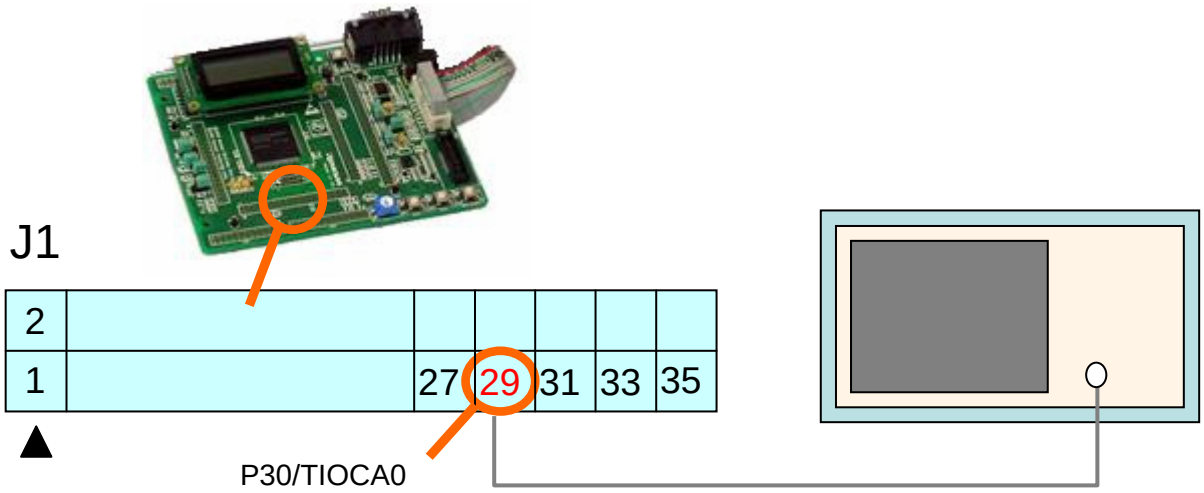
5. ウォッチウィンドウ上の “result” の値が変化します。



6.3 TPUでPWMパルス出力

このチュートリアルでは 16 ビットタイマパルスユニット(TPU) チャネル 0 を設定し、TIOCA0 端子からパルスを出力します。

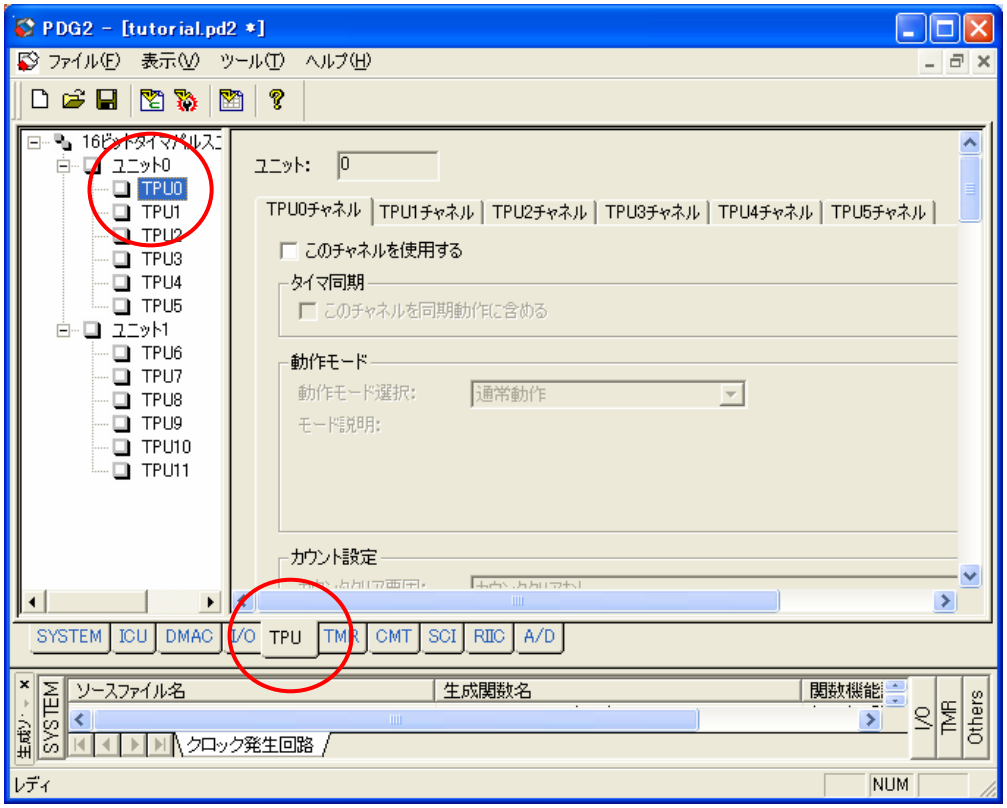
TIOCA0 端子は RSK ボードのスルーホール J1/No.29 です。オシロスコープ等を TIOCA0 に接続してください。



(HEW と PDG のプロジェクトは「6.2 A/D 変換の連続スキャン」までに作成したものを使用します。)

(1) TPU の設定-1 PDG

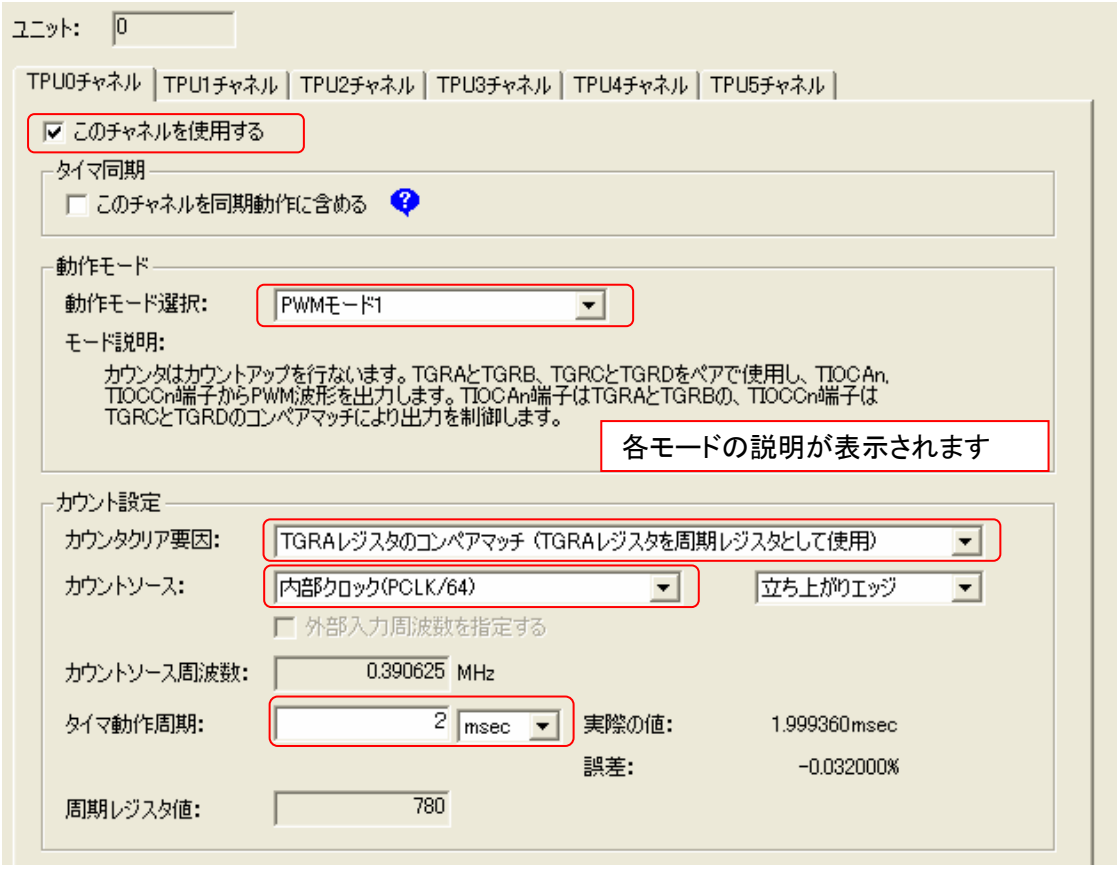
TPU タブを選択し、ツリー表示上で TPU0 を選択してください。



(2) TPU の設定-2 PDG

TPU0 を以下の通り設定してください。

1. [このチャネルを使用する]をチェック
2. 動作モードは[PWMモード1]を選択
3. カウンタクリア要因は[TGRAレジスタのコンペアマッチ]
4. カウントソースは [PCLK/64]
5. タイマ動作周期は2msに設定



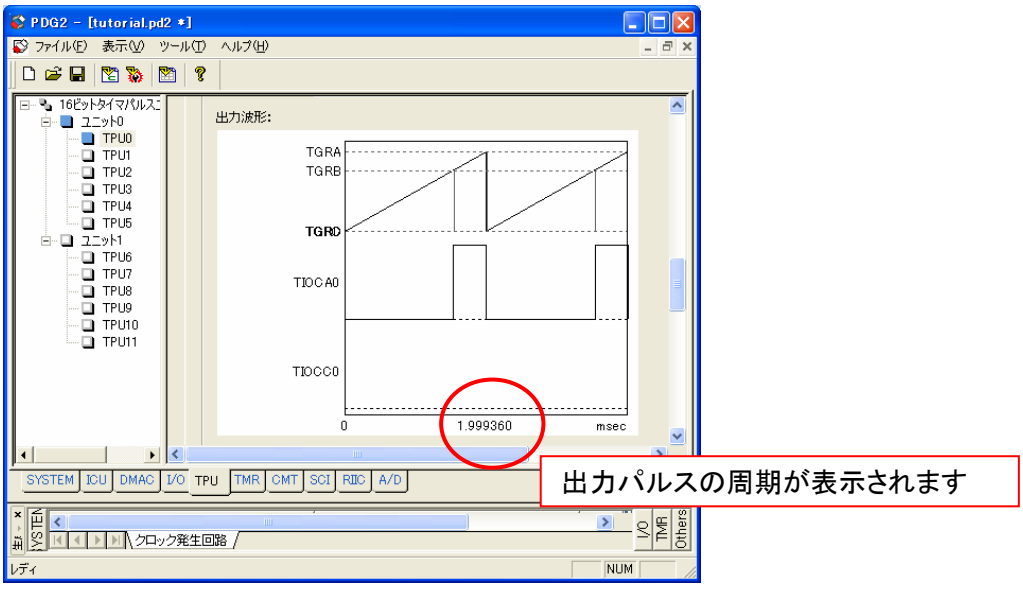
(3) TPU の設定-3 PDG

端子からの出力を以下の通り設定してください。

1. TGRAレジスタのコンペアマッチ(コンペアマッチA)で0出力
2. TGRB値(コンペアマッチB値)： 600
3. TGRBレジスタのコンペアマッチ(コンペアマッチB)で1出力
4. TGRC,TGRDレジスタのTIOCCn端子出力制御は無効

(4) 出力波形の確認 PDG

出力波形を TPU 設定ウィンドウ上で確認することができます。

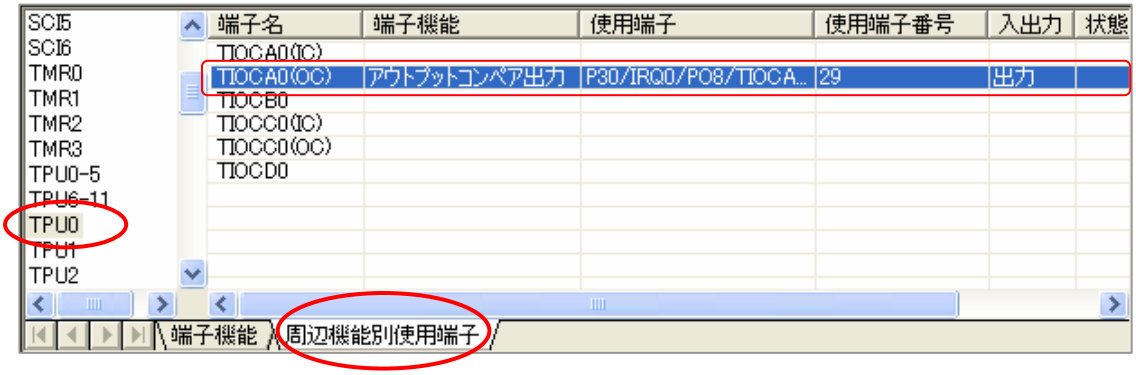


(5) 端子使用状況の確認 **PDG**

端子機能ウィンドウと周辺機能別使用端子ウィンドウで TIOCA0 端子の使用状況を確認してください。

端子機能ウィンドウ

周辺機能別使用端子ウィンドウ



(6) 端子一覧表の出力 PDG

・ツールバー上の  または[ツール]->[端子一覧表生成]メニューにより、端子機能ウィンドウの内容を CSV ファイルに出力することができます。

・出力先は“PDG プロジェクトのフォルダ/PIN”です。

端子機能ウィンドウ

端子番号	端子名	選択機能	入出力	状態
24	NMI			
25	P34/IRQ4/PO12/TIOCA1(IC)/TIOCA1(OC)			
26	P33/IRQ3/PO11/TIOCC0(IC)/TIOCD0/TC...	P33	出力	
27	P32/IRQ2/PO10/TIOCC0(IC)/TIOCC0(OC...			
28	P31/IRQ1/PO9/TIOCA0(IC)/TIOCB0			
29	P30/IRQ0/PO8/TIOCA0(IC)/TIOCA0(OC)	TIOCA0(OC)	出力	
30	P27/PO7/TIOCA5(IC)/TIOCB5/SCK1			
31	P26/PO6/TIOCA5(IC)/TIOCA5(OC)/TMO...			
32	P25/PO5/TIOCA4(IC)/TIOCA4(OC)/TMCI...			
33	P24/PO4/TIOCA4(IC)/TIOCB4/TMRI1			
34	P23/PO3/TIOCC3(IC)/TIOCD3			
35	P22/PO2/TIOCC3(IC)/TIOCC3(OC)/TMO...			

端子機能 周辺機能別使用端子

PinFUnction.csv

Pin No	Pin Name	Selected funct	Direction
1	P04/IRQ12/TMCI3/TxD4/TDI		
27	P32/IRQ2/PO10/TIOCC0(IC)/TIOCC0(O		
28	P31/IRQ1/PO9/TIOCA0(IC)/TIOCB0		
29	P30/IRQ0/PO8/TIOCA0(IC)/TIOCA0(OC	TIOCA0(OC)	Output
30	P27/PO7/TIOCA5(IC)/TIOCB5/SCK1		

周辺機能別使用端子ウィンドウ

	端子名	端子機能	使用端子	使用端子番号	入出力	状態
SCI5	TIOCA0(IC)					
SCI6	TIOCA0(OC)	アウトプットコンペア出力	P30/IRQ0/PO8/TIOCA...	29	出力	
TMR0	TIOCB0					
TMR1	TIOCC0(IC)					
TMR2	TIOCC0(OC)					
TMR3	TIOCD0					
TPU0-5						
TPU6-11						
TPU0						
TPU1						
TPU2						

端子機能 周辺機能別使用端子

PeripheralPinUsage.csv

Peripheral	Pin Name	Pin function	Assignment	Pin N	Direction
Clock	BCLK				
TPU6-11	TCLKH				
TPU0	TIOCA0(IC)				
TPU0	TIOCA0(OC)	Compare match signal output	P30/IRQ0/PO8/	29	Output
TPU0	TIOCB0				
TPU0	TIOCC0(IC)				

(7) プログラムの作成 HEW

HEW 上で以下のプログラムを作成してください。グレーの部分は「6.2 A/D 変換の連続スキャン」までに作成したものを使用してください。

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

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

    // AD0のA/D変換開始
    R_PG_ADC_10_Set_AD0();

    //ポートP33の設定
    R_PG_IO_PORT_Set_P3();

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

    // AD0のA/D変換開始
    R_PG_ADC_10_StartConversionSW_AD0 ();

    // TPU0を設定しカウントを開始
    R_PG_Timer_Start_TPU_U0_C0();

    while(1);
}
```

(8) ソースファイルの生成と HEW への登録 PDG

ソースファイルを生成し、HEW に登録してください。 (6.1 (8)(11)参照)

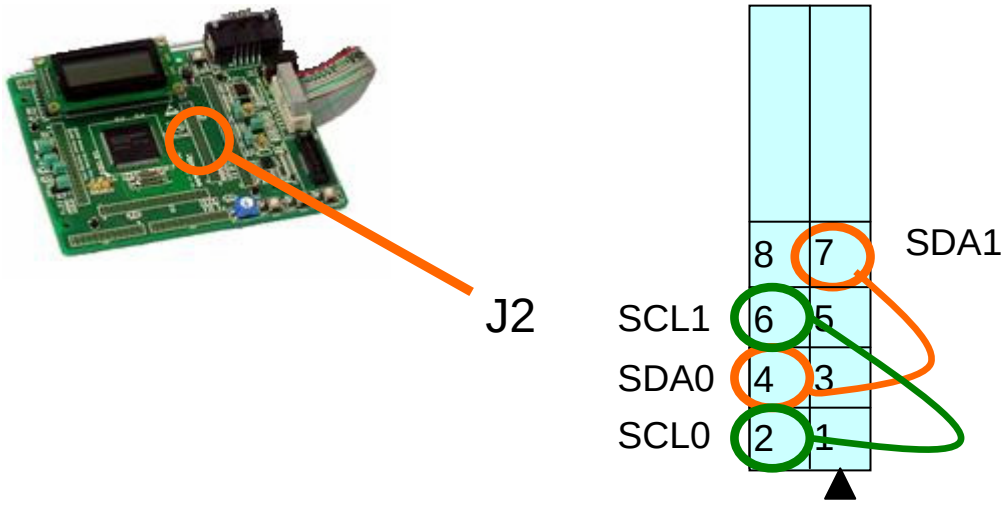
(9) プログラムのビルドと実行 HEW

1. プログラムをビルドしダウンロード、実行してください。
2. シロスコープ等でTIOCA0の出力パルス波形を確認してください。

6.4 I2C チャネル 0 とチャネル 1 で通信

RX610 には RIIC0 と RIIC1、2 つの I2C チャネルがあります。このチュートリアルでは、これらのチャネルを設定し、RIIC0 から RIIC1 にデータを転送します。RIIC0 をマスタ、RIIC1 をスレーブとして使用します。

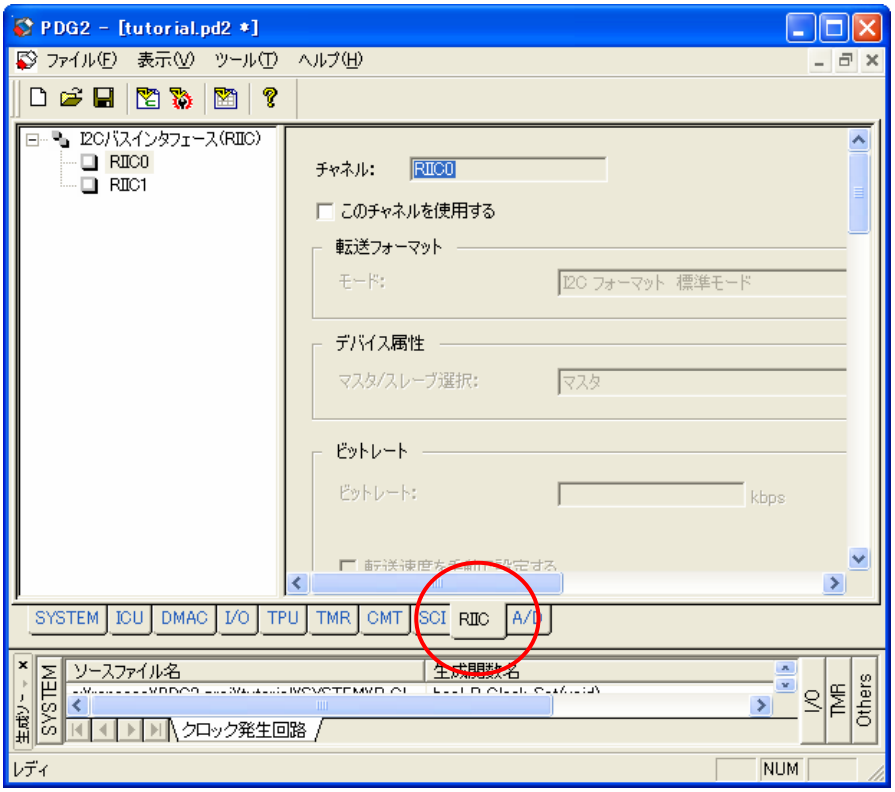
RSK ボード上の SCL0 と SCL1, SDA0 と SDA1 を接続してください。RIIC の端子は RSK ボードのスルーホール J2/No.2,4,6,7 です。



(HEW と PDG のプロジェクトは「6.3 TPU で PWM パルス出力」までに作成したものを使用します。)

(1) RIIC の設定 **PDG**

RIIC タブを選択してください。

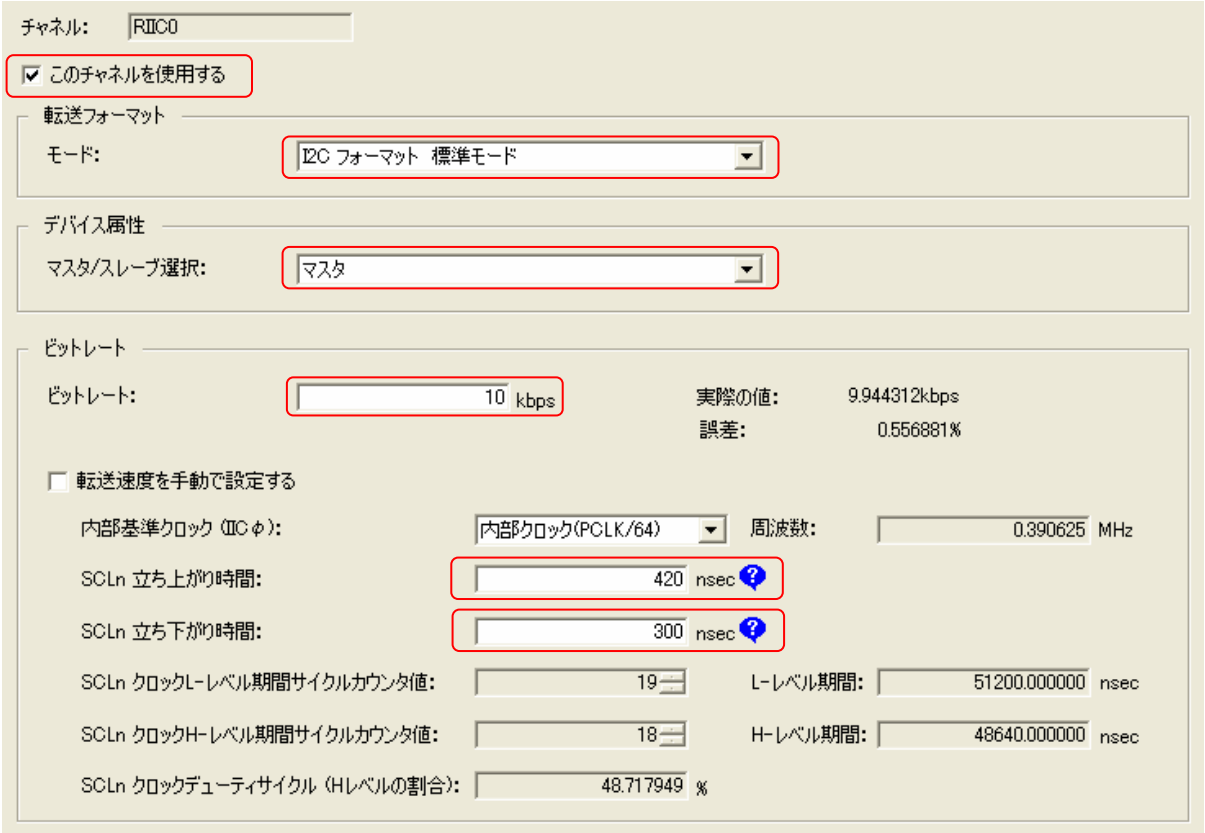


(2) RIIC0(マスタ)の設定 PDG

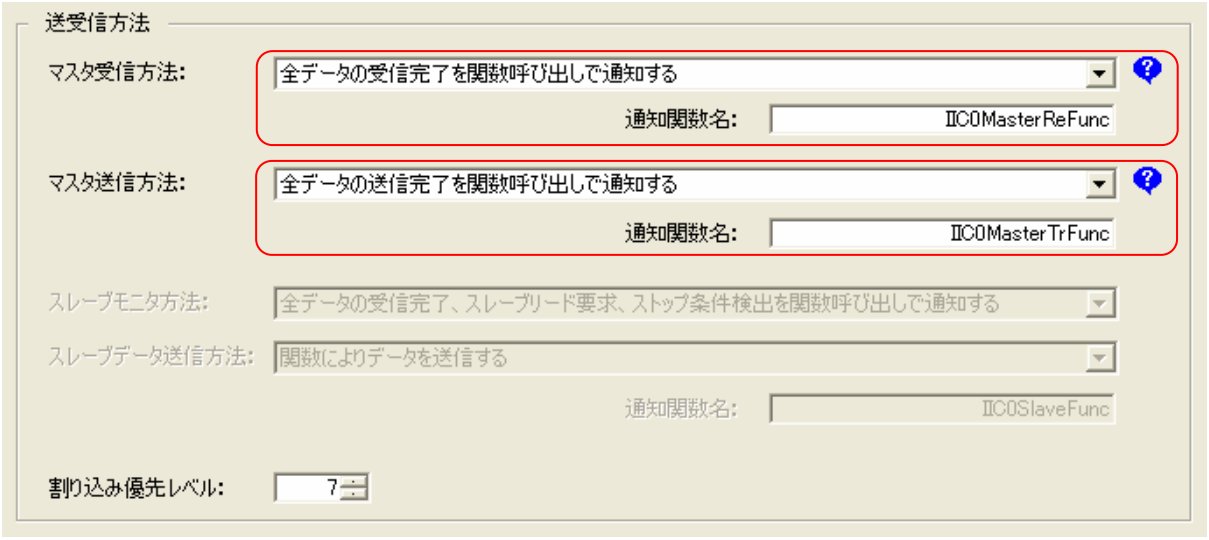
RIIC0 を以下の通り設定してください。

1. ツリー表示上でRIIC0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
3. モードは[I2Cフォーマット標準モード]を選択
4. マスタ/スレーブ選択は[マスタ]を選択
5. ビットレートは10kbpsに設定
6. SCLn立ち上がり時間/立ち下がり時間はHWシステムに依存します。
   RSKボードでは400ns、320nsを設定してください。



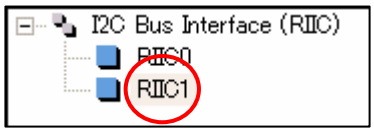
7 マスタ受信方法に[全データの受信完了を関数呼び出しで通知する]を選択
通知関数は IIC0MasterReFunc を指定
8 マスタ送信方法に[全データの送信完了を関数呼び出しで通知する]を選択
通知関数は IIC0MasterTrFunc を指定



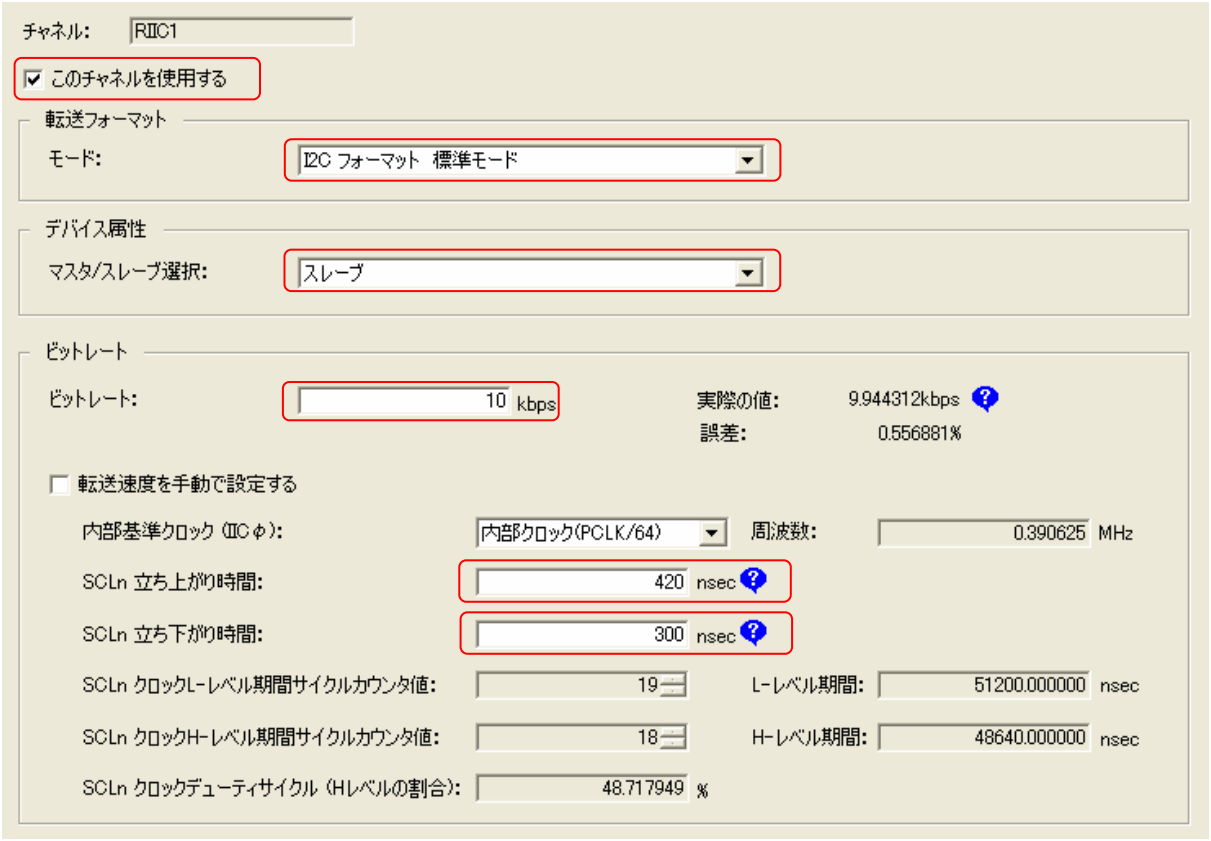
(3) RIIC1(スレーブ)の設定 PDG

RIIC1 を以下の通り設定してください。

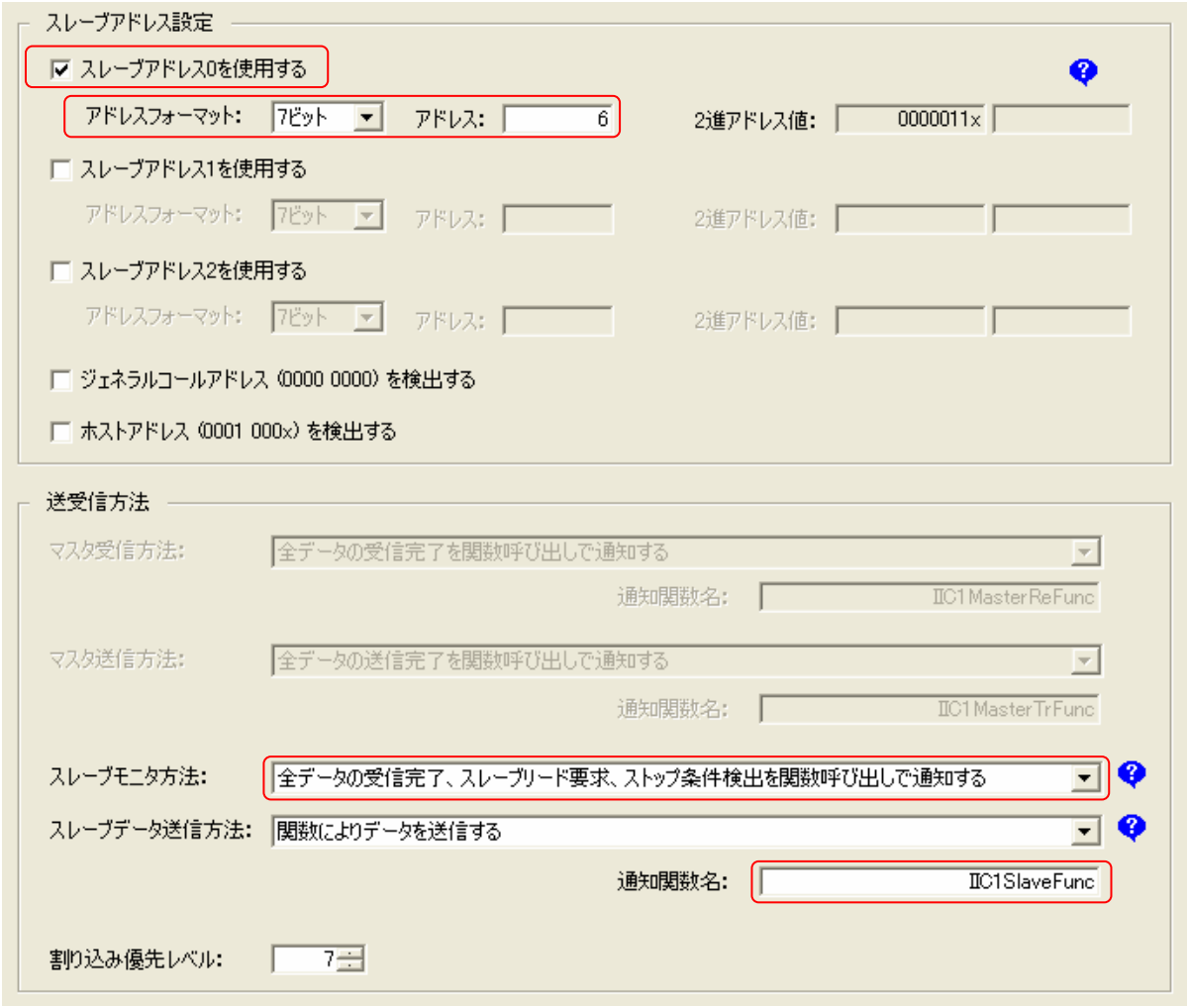
1. ツリー表示上でRIIC1を選択してください。



2. [このチャネルを使用する]をチェック
3. モードは[I2Cフォーマット標準モード]を選択
4. マスタ/スレーブ選択は[スレーブ]を選択
5. ビットレートはRIIC0と同様
6. SCLn立ち上がり時間/立ち下がり時間はRIIC0と同様



7. スレーブアドレス0を6(7bit)に設定
8. 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]を選択
   通知関数は IIC1SlaveFunc を指定



(4) プログラムの作成 **HEW**

HEW 上で以下のプログラムを作成してください。グレーの部分は「6.3 TPU で PWM パルス出力」までに作成したものを使用してください。

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

uint8_t tr[]="renesas";
uint8_t re[]="----------";

void main(void)
{
    //Set Clock
    R_PG_Clock_Set();

    // RIIC0 と RIIC1 を設定
    R_PG_I2C_Set_C0();
    R_PG_I2C_Set_C1();

    // RIIC0 スレーブモニタ開始（データ受信待ち）
    R_PG_I2C_SlaveMonitor_C1(
        re, //受信データの格納先
        8 //受信データ数
    );

    // RIIC0 マスタ送信開始
    R_PG_I2C_MasterSend_C0(
        6, // 送信先スレーブアドレス
        tr, // 送信データの格納場所
        8 // 送信データ数
    );
    while(1);
}
```

```
uint16_t tr_count;
uint16_t re_count;

// マスタ送信通知関数
void IIC0MasterTrFunc(void)
{
    R_PG_I2C_GetSentDataCount_C0(&tr_count);
}

// マスタ受信通知関数
void IIC0MasterReFunc(void)
{
}

// スレーブモニタ通知関数
void IIC1SlaveFunc (void)
{
    R_PG_I2C_GetReceivedDataCount_C1(& re_count);
}
```

(5) ソースファイルの生成と HEW への登録 PDG

ソースファイルを生成し、HEW に登録してください。 (6.1 (8)(11)参照)

(6) プログラムのビルドと実行 HEW

1. プログラムをビルドしダウンロード、実行してください。
2. 受信データ “re” の値をウォッチウィンドウで確認してください。

RX610グループ
Peripheral Driver Generator
リファレンスマニュアル

発行年月日　2011年2月16日　Rev.1.01

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 ツール開発第一部

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・電話番号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 エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）　(03)5201-5307

■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/inquiry

RX610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

R20UT0150JJ0101