User's Manual

RX62Gグループ
Peripheral Driver Generator
リファレンスマニュアル

ルネサス エレクトロニクス
www.renesas.com

Rev.1.02　2014.5

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して、お客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
2. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
3. 本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害に関し、当社は、何らの責任を負うものではありません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、
各品質水準は、以下に示す用途に製品が使用されることを意図しております。

    標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、
    家電、工作機械、パーソナル機器、産業用ロボット等

    高品質水準：　輸送機器（自動車、電車、船舶等）、交通用信号機器、
    防災・防犯装置、各種安全装置等

    当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。たとえ、意図しない用途に当社製品を使用したことによりお客様または第三者に損害が生じても、当社は一切その責任を負いません。なお、ご不明点がある場合は、当社営業にお問い合わせください。
6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 本資料に記載されている当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。また、当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途に使用しないでください。当社製品または技術を輸出する場合は、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。

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

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

はじめに

本書は Peripheral Driver Generator を用いた RX62G 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

はじめに 3
目次 4
1. 概要 10
1.1 サポート範囲 10
1.2 関連ツール 10
2. プロジェクトの作成 11
3. 周辺機能の設定 12
3.1 設定画面 12
3.2 端子機能 13
3.2.1 端子機能シート 13
3.2.2 周辺機能別使用端子シート 14
4. チュートリアル 16
4.1 コンペアマッチタイマ(CMT)の割り込みでLEDを点滅 17
4.2 マルチファンクションタイマパルスユニット3(MTU3)のPWM波でLEDを点滅 28
4.3 10ビットA/Dコンバータ（ADA）の連続スキャン 34
4.4 IRQによるDTC転送のトリガ 41
4.5 SCIb チャネル0とチャネル2で調歩同期通信 47
5. 生成関数仕様 54
5.1 クロック発生回路 60
5.1.1 R_PG_Clock_Set 60
5.1.2 R_PG_Clock_GetMainClockStatus 61
5.2 電圧検出回路（LVD） 62
5.2.1 R_PG_LVD_Set 62
5.2.2 R_PG_LVD_GetLVDDetectionFlag 63
5.3 消費電力低減機能 64
5.3.1 R_PG_LPC_Set 64
5.3.2 R_PG_LPC_Sleep 65
5.3.3 R_PG_LPC_AllModuleClockStop 66
5.3.4 R_PG_LPC_SoftwareStandby 67
5.3.5 R_PG_LPC_DeepSoftwareStandby 68
5.3.6 R_PG_LPC_IOPortRelease 69
5.3.7 R_PG_LPC_GetPowerOnResetFlag 70
5.3.8 R_PG_LPC_GetLVDDetectionFlag 71
5.3.9 R_PG_LPC_GetDeepSoftwareStandbyResetFlag 72
5.3.10 R_PD_LPC_GetDeepSoftwareStandbyCancelFlag 73
5.3.11 R_PG_LPC_GetStatus 74
5.3.12 R_PG_LPC_WriteBackup 75
5.3.13 R_PG_LPC_ReadBackup 76
5.4 割り込みコントローラ（ICU） 77
5.4.1 R_PG_ExtInterrupt_Set_*<割り込み種別>* 77

5.4.2 R_PG_ExtInterrupt_Disable_<割り込み種別> 79
5.4.3 R_PG_ExtInterrupt_GetRequestFlag_<割り込み種別> 80
5.4.4 R_PG_ExtInterrupt_ClearRequestFlag_<割り込み種別> 81
5.4.5 R_PG_SoftwareInterrupt_Set 82
5.4.6 R_PG_SoftwareInterrupt_Generate 83
5.4.7 R_PG_FastInterrupt_Set 84
5.4.8 R_PG_Exception_Set 85
5.5 バス 86
5.5.1 R_PG_ExtBus_SetBus 86
5.5.2 R_PG_ExtBus_GetErrorStatus 87
5.5.3 R_PG_ExtBus_ClearErrorFlags 88
5.6 データトランスファコントローラ (DTC) 89
5.6.1 R_PG_DTC_Set 89
5.6.2 R_PG_DTC_Set_<転送開始要因> 90
5.6.3 R_PG_DTC_Activate 91
5.6.4 R_PG_DTC_SuspendTransfer 92
5.6.5 R_PG_DTC_GetTransmitStatus 93
5.6.6 R_PG_DTC_StopModule 94
5.7 I/Oポート 95
5.7.1 R_PG_IO_PORT_Set_P<ポート番号> 95
5.7.2 R_PG_IO_PORT_Set_P<ポート番号><端子番号> 96
5.7.3 R_PG_IO_PORT_Read_P<ポート番号> 97
5.7.4 R_PG_IO_PORT_Read_P<ポート番号><端子番号> 98
5.7.5 R_PG_IO_PORT_Write_P<ポート番号> 99
5.7.6 R_PG_IO_PORT_Write_P<ポート番号><端子番号> 100
5.8 マルチファンクションタイマパルスユニット3 (MTU3) 101
5.8.1 R_PG_Timer_Set_MTU_U<ユニット番号>_<チャネル> 101
5.8.2 R_PG_Timer_StartCount_MTU_U<ユニット番号>_C<チャネル番号>(_<相>) 103
5.8.3 R_PG_Timer_SynchronouslyStartCount_MTU_U<ユニット番号> 104
5.8.4 R_PG_Timer_HaltCount_MTU_U<ユニット番号>_C<チャネル番号>(_<相>) 105
5.8.5 R_PG_Timer_GetCounterValue_MTU_U<ユニット番号>_C<チャネル番号> 106
5.8.6 R_PG_Timer_SetCounterValue_MTU_U<ユニット番号>_C<チャネル番号>(_<相>) 107
5.8.7 R_PG_Timer_GetRequestFlag_MTU_U<ユニット番号>_C<チャネル番号> 108
5.8.8 R_PG_Timer_StopModule_MTU_U<ユニット番号> 110
5.8.9 R_PG_Timer_GetTGR_MTU_U<ユニット番号>_C<チャネル番号> 111
5.8.10 R_PG_Timer_SetTGR_<ジェネラルレジスタ>_MTU_U<ユニット番号>_C<チャネル番号> 113
5.8.11 R_PG_Timer_SetBuffer_AD_U<ユニット番号>_C<チャネル番号> 114
5.8.12 R_PG_Timer_SetBuffer_CycleData_MTU_U<ユニット番号>_<チャネル> 115
5.8.13 R_PG_Timer_SetOutputPhaseSwitch_MTU_U<ユニット番号>_<チャネル> 116
5.8.14 R_PG_Timer_ControlOutputPin_MTU_U<ユニット番号>_<チャネル> 117
5.8.15 R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U<ユニット番号>_<チャネル> 118
5.8.16 R_PG_Timer_ControlBufferTransfer_MTU_U<ユニット番号>_<チャネル> 119
5.9 ポートアウトプットイネーブル3 (POE3) 120
5.9.1 R_PG_POE_Set 120
5.9.2 R_PG_POE_SetHiZ_<タイマチャネル> 121

5.9.3 R_PG_POE_GetRequestFlagHiZ_<タイマチャネル>....122
5.9.4 R_PG_POE_GetShortFlag_<タイマチャネル>....123
5.9.5 R_PG_POE_ClearFlag_<タイマチャネル>....124
5.10 汎用PWM タイマ（GPT）....125
5.10.1 R_PG_Timer_Set_GPT_U<ユニット番号>....125
5.10.2 R_PG_Timer_Set_GPT_U<ユニット番号>_C<チャネル番号>....126
5.10.3 R_PG_Timer_StartCount_GPT_U<ユニット番号>_C<チャネル番号>....127
5.10.4 R_PG_Timer_SynchronouslyStartCount_GPT_U<ユニット番号>....128
5.10.5 R_PG_Timer_HaltCount_GPT_U<ユニット番号>_C<チャネル番号>....129
5.10.6 R_PG_Timer_SynchronouslyHaltCount_GPT_U<ユニット番号>....130
5.10.7 R_PG_Timer_SetGTCCR_<GTCCR>_GPT_U<ユニット番号>_C<チャネル番号>....131
5.10.8 R_PG_Timer_GetGTCCR_GPT_U<ユニット番号>_C<チャネル番号>....132
5.10.9 R_PG_Timer_SetCounterValue_GPT_U<ユニット番号>_C<チャネル番号>....133
5.10.10 R_PG_Timer_GetCounterValue_GPT_U<ユニット番号>_C<チャネル番号>....134
5.10.11 R_PG_Timer_SynchronouslyClearCounter_GPT_U<ユニット番号>....135
5.10.12 R_PG_Timer_SetCycle_GPT_U<ユニット番号>_C<チャネル番号>....136
5.10.13 R_PG_Timer_SetBuffer_Cycle_GPT_U<ユニット番号>_C<チャネル番号>....137
5.10.14 R_PG_Timer_SetDoubleBuffer_Cycle_GPT_U<ユニット番号>_C<チャネル番号>....138
5.10.15 R_PG_Timer_SetAD_GPT_U<ユニット番号>_C<チャネル番号>....139
5.10.16 R_PG_Timer_SetBuffer_AD_GPT_U<ユニット番号>_C<チャネル番号>....140
5.10.17 R_PG_Timer_SetDoubleBuffer_AD_GPT_U<ユニット番号>_C<チャネル番号>....141
5.10.18 R_PG_Timer_SetBuffer_GTDV<U/D>_GPT_U<ユニット番号>_C<チャネル番号>....142
5.10.19 R_PG_Timer_GetRequestFlag_GPT_U<ユニット番号>_C<チャネル番号>....143
5.10.20 R_PG_Timer_GetRequestFlag_GPT_U<ユニット番号>....144
5.10.21 R_PG_Timer_GetCounterStatus_GPT_U<ユニット番号>_C<チャネル番号>....145
5.10.22 R_PG_Timer_BufferEnable_GPT_U<ユニット番号>_C<チャネル番号>....146
5.10.23 R_PG_Timer_BufferDisable_GPT_U<ユニット番号>_C<チャネル番号>....147
5.10.24 R_PG_Timer_Buffer_Force_GPT_U <ユニット番号>_C<チャネル番号>....148
5.10.25 R_PG_Timer_CountDirection_Down_GPT_U<ユニット番号>_C<チャネル番号>....149
5.10.26 R_PG_Timer_CountDirection_Up_GPT_U<ユニット番号>_C<チャネル番号>....150
5.10.27 R_PG_Timer_SoftwareNegate_GPT_U<ユニット番号>_C<チャネル番号>....151
5.10.28 R_PG_Timer_StartCount_LOCO_GPT_U<ユニット番号>....152
5.10.29 R_PG_Timer_HaltCount_LOCO_GPT_U<ユニット番号>....153
5.10.30 R_PG_Timer_ClearCounter_LOCO_GPT_U<ユニット番号>....154
5.10.31 R_PG_Timer_InitialiseCountResultValue_LOCO_GPT_U<ユニット番号>....155
5.10.32 R_PG_Timer_GetCounterValue_LOCO_GPT_U<ユニット番号>....156
5.10.33 R_PG_Timer_GetCounterAverageValue_LOCO_GPT_U<ユニット番号>....157
5.10.34 R_PG_Timer_GetCountResultValue_LOCO_GPT_U<ユニット番号>....158
5.10.35 R_PG_Timer_SetPermissibleDeviation_LOCO_GPT_U<ユニット番号>....159
5.10.36 R_PG_Timer_AdjustEdgeDelay_GPT_U<ユニット番号>_C<チャネル番号>....160
5.10.37 R_PG_Timer_EnableEdgeDelay_GPT_U<ユニット番号>....161
5.10.38 R_PG_Timer_DisableEdgeDelay_GPT_U<ユニット番号>....162
5.10.39 R_PG_Timer_StopModule_GPT_U<ユニット番号>....162
5.11 コンペアマッチタイマ（CMT）....164
5.11.1 R_PG_Timer_Start_CMT_U<ユニット番号>_C<チャネル番号>....164

5.11.2 R_PG_Timer_HaltCount_CMT_U<ユニット番号>_C<チャネル番号> 165
5.11.3 R_PG_Timer_ResumeCount_CMT_U<ユニット番号>_C<チャネル番号> 166
5.11.4 R_PG_Timer_GetCounterValue_CMT_U<ユニット番号>_C<チャネル番号> 167
5.11.5 R_PG_Timer_SetCounterValue_CMT_U<ユニット番号>_C<チャネル番号> 168
5.11.6 R_PG_Timer_StopModule_CMT_U<ユニット番号> 169
5.12 ウォッチドッグタイマ（WDT） 170
5.12.1 R_PG_Timer_Start_WDT 170
5.12.2 R_PG_Timer_HaltCount_WDT 171
5.12.3 R_PG_Timer_ResetCounter_WDT 172
5.12.4 R_PG_Timer_ClearOverflowFlag_WDT 173
5.13 独立ウォッチドッグタイマ（IWDT） 174
5.13.1 R_PG_Timer_Set_IWDT 174
5.13.2 R_PG_Timer_RefreshCounter_IWDT 175
5.13.3 R_PG_Timer_GetCounterValue_IWDT 176
5.13.4 R_PG_Timer_ClearUnderflowFlag_IWDT 177
5.14 シリアルコミュニケーションインタフェース（SCIb） 178
5.14.1 R_PG_SCI_Set_C<チャネル番号> 178
5.14.2 R_PG_SCI_StartSending_C<チャネル番号> 179
5.14.3 R_PG_SCI_SendAllData_C<チャネル番号> 181
5.14.4 R_PG_SCI_GetSentDataCount_C<チャネル番号> 182
5.14.5 R_PG_SCI_StartReceiving_C<チャネル番号> 183
5.14.6 R_PG_SCI_ReceiveAllData_C<チャネル番号> 185
5.14.7 R_PG_SCI_StopCommunication_C<チャネル番号> 186
5.14.8 R_PG_SCI_GetReceivedDataCount_C<チャネル番号> 187
5.14.9 R_PG_SCI_GetReceptionErrorFlag_C<チャネル番号> 188
5.14.10 R_PG_SCI_GetTransmitStatus_C<チャネル番号> 189
5.14.11 R_PG_SCI_SendTargetStationID_C<チャネル番号> 190
5.14.12 R_PG_SCI_ReceiveStationID_C<チャネル番号> 191
5.14.13 R_PG_SCI_StopModule_C<チャネル番号> 192
5.14.14 R_PG_SCI_ControlClockOutput_C<チャネル番号> 193
5.15 CRC演算器（CRC） 194
5.15.1 R_PG_CRC_Set 194
5.15.2 R_PG_CRC_InputData 195
5.15.3 R_PG_CRC_GetResult 196
5.15.4 R_PG_CRC_StopModule 197
5.16 I2Cバスインタフェース（RIIC） 198
5.16.1 R_PG_I2C_Set_C<チャネル番号> 198
5.16.2 R_PG_I2C_MasterReceive_C<チャネル番号> 199
5.16.3 R_PG_I2C_MasterReceiveLast_C<チャネル番号> 201
5.16.4 R_PG_I2C_MasterSend_C<チャネル番号> 203
5.16.5 R_PG_I2C_MasterSendWithoutStop_C<チャネル番号> 205
5.16.6 R_PG_I2C_GenerateStopCondition_C<チャネル番号> 207
5.16.7 R_PG_I2C_GetBusState_C<チャネル番号> 208
5.16.8 R_PG_I2C_SlaveMonitor_C<チャネル番号> 209
5.16.9 R_PG_I2C_SlaveSend_C<チャネル番号> 211

5.16.10 R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉....212
5.16.11 R_PG_I2C_GetTR_C〈チャネル番号〉....213
5.16.12 R_PG_I2C_GetEvent_C〈チャネル番号〉....214
5.16.13 R_PG_I2C_GetReceivedDataCount_C〈チャネル番号〉....215
5.16.14 R_PG_I2C_GetSentDataCount_C〈チャネル番号〉....216
5.16.15 R_PG_I2C_Reset_C〈チャネル番号〉....217
5.16.16 R_PG_I2C_StopModule_C〈チャネル番号〉....218
5.17 シリアルペリフェラルインタフェース（RSPI）....219
5.17.1 R_PG_RSPI_Set_C〈チャネル番号〉....219
5.17.2 R_PG_RSPI_SetCommand_C〈チャネル番号〉....220
5.17.3 R_PG_RSPI_StartTransfer_C〈チャネル番号〉....221
5.17.4 R_PG_RSPI_TransferAllData_C〈チャネル番号〉....223
5.17.5 R_PG_RSPI_GetStatus_C〈チャネル番号〉....225
5.17.6 R_PG_RSPI_GetError_C〈チャネル番号〉....226
5.17.7 R_PG_RSPI_GetCommandStatus_C〈チャネル番号〉....227
5.17.8 R_PG_RSPI_StopModule_C〈チャネル番号〉....228
5.17.9 R_PG_RSPI_LoopBack〈ループバックモード〉_C〈チャネル番号〉....229
5.18 LINモジュール（LIN）....230
5.18.1 R_PG_LIN_Set_LIN0....230
5.18.2 R_PG_LIN_Transmit_LIN0....231
5.18.3 R_PG_LIN_Receive_LIN0....233
5.18.4 R_PG_LIN_ReadData_LIN0....235
5.18.5 R_PG_LIN_EnterResetMode_LIN0....236
5.18.6 R_PG_LIN_EnterOperationMode_LIN0....237
5.18.7 R_PG_LIN_EnterWakeUpMode_LIN0....238
5.18.8 R_PG_LIN_WakeUpTransmit_LIN0....239
5.18.9 R_PG_LIN_WakeUpReceive_LIN0....240
5.18.10 R_PG_LIN_GetCheckSum_LIN0....241
5.18.11 R_PG_LIN_EnterSelfTestMode_LIN0....242
5.18.12 R_PG_LIN_WriteCheckSum_LIN0....244
5.18.13 R_PG_LIN_GetMode_LIN0....246
5.18.14 R_PG_LIN_GetStatus_LIN0....247
5.18.15 R_PG_LIN_GetErrorStatus_LIN0....248
5.18.16 R_PG_LIN_StopModule_LIN0....250
5.19 12ビットA/Dコンバータ（S12ADA）....251
5.19.1 R_PG_ADC_12_Set_S12ADA〈ユニット番号〉....251
5.19.2 R_PG_ADC_12_Set....252
5.19.3 R_PG_ADC_12_StartConversionSW_S12ADA〈ユニット番号〉....253
5.19.4 R_PG_ADC_12_StopConversion_S12ADA〈ユニット番号〉....254
5.19.5 R_PG_ADC_12_GetResult_S12ADA〈ユニット番号〉....255
5.19.6 R_PG_ADC_12_GetResult_SelfDiag_S12AD〈ユニット番号〉....257
5.19.7 R_PG_ADC_12_StopModule_S12ADA〈ユニット番号〉....259
5.20 10ビットA/Dコンバータ（ADA）....260
5.20.1 R_PG_ADC_10_Set_AD〈ユニット番号〉....260
5.20.2 R_PG_ADC_10_StartConversionSW_AD〈ユニット番号〉....261

5.20.3 R_PG_ADC_10_StopConversion_AD〈*ユニット番号*〉....262
5.20.4 R_PG_ADC_10_GetResult_AD〈*ユニット番号*〉....263
5.20.5 R_PG_ADC_10_SetSelfDiag_VREF_〈*電圧値*〉AD〈*ユニット番号*〉....264
5.20.6 R_PG_ADC_10_StopModule_AD〈*ユニット番号*〉....265
5.21 通知関数に関する注意事項....266
5.21.1 割り込みとプロセッサモード....266
5.21.2 割り込みとDSP命令....266
6. 生成ファイルのIDEへの登録とビルド....267
付録 1. 割当先を変更できる端子機能....269

1. 概要

1.1 サポート範囲

Peripheral Driver Generator がサポートする RX62G グループの製品型名、周辺機能、エンディアンは以下の通りです。

(1) 製品型名

型名	パッケージ	型名	パッケージ
R5F562GAADFH	PLQP0112JA-A	R5F562GADDFH	PLQP0112JA-A
R5F562GAADFP	PLQP0100KB-A	R5F562GADDFP	PLQP0100KB-A
R5F562G7ADFH	PLQP0112JA-A	R5F562G7DDFH	PLQP0112JA-A
R5F562G7ADFP	PLQP0100KB-A	R5F562G7DDFP	PLQP0100KB-A

(2) 周辺機能

電圧検出回路 (LVD)	コンペアマッチタイマ (CMT)
クロック発生回路	ウォッチドッグタイマ (WDT)
消費電力低減機能	独立ウォッチドッグタイマ (IWDT)
割り込みコントローラ (ICU), 例外処理	シリアルコミュニケーションインタフェース(SCIb)
バス ※不正アドレスアクセス検出	CRC 演算器 (CRC)
データトランスファコントローラ (DTC)	I2C バスインタフェース (RIIC)
I/O ポート	シリアルペリフェラルインタフェース (RSPI)
マルチファンクションタイマパルスユニット3 (MTU3)	LINモジュール (LIN)
ポートアウトプットイネーブル3 (POE3)	12 ビットA/D コンバータ (S12ADA)
汎用PWMタイマ (GPTa)	10 ビットA/D コンバータ(ADA)

CANモジュールはサポートしていません。
メモリプロテクションユニット(MPU)はサポートしていません。
12 ビットA/D コンバータ (S12ADA)のコンパレータはサポートしていません。

(3) エンディアン

ビッグエンディアン、 リトルエンディアン

1.2 関連ツール

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

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

・RX62G/RX62Tグループ Renesas Peripheral Driver Library V.1.10
(Peripheral Driver Generatorに同梱されています。)

2. プロジェクトの作成

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

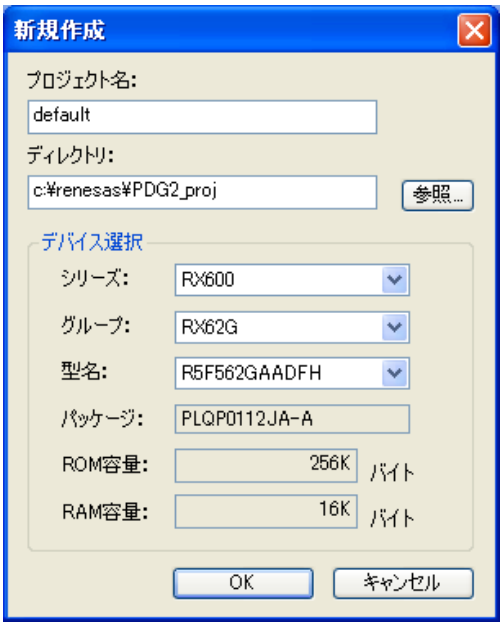


図 2.1 新規作成ダイアログボックス

RX62G 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［RX62G］を選択してください。使用する製品の型名を選択すると、その製品のパッケージ、ROM 容量、RAM 容量が表示されます。

[OK]をクリックすると新規プロジェクトを作成して開きます。

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

図 2.2 新規プロジェクトのエラー表示

ここでは使用するクロック周波数を設定してください。

3. 周辺機能の設定

3.1 設定画面

図 3.1 に周辺モジュール設定ウィンドウの表示例を示します。

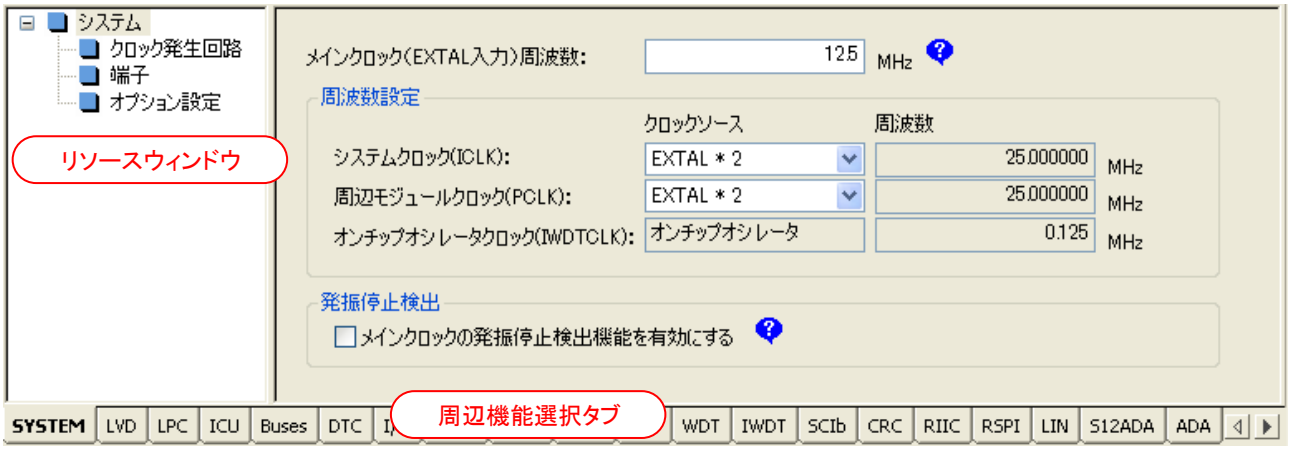


図3.1 周辺機能設定ウィンドウの表示例

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

表 3.1 周辺機能選択タブおよびリソースウィンドウの項目と周辺機能の対応

タブ	リソースウィンドウ	対応する周辺機能
SYSTEM	クロック発生回路	クロック発生回路
	端子	端子機能
	オプション設定	エンディアン設定
LVD	電圧検出回路（LVD）	電圧検出回路 LVD1および2
LPC	消費電力低減機能	消費電力低減機能
ICU	割り込み	割り込みコントローラ（ICU） （高速割り込み，ソフトウェア割り込み，外部割込み(NMI, IRQ0～7)）
	例外	例外処理
Buses	バスエラー監視	バスエラー監視（不正アドレスアクセス検出）
DTC	DTC	データトランスファコントローラ（DTC）
I/O	ポート1～9,A,B,D,E,G	I/Oポート ポート1～9,A,B,D,E,G
MTU3	MTU0～MTU7	マルチファンクションタイマパルスユニット3（MTU3）チャネル0～7
POE3	POE3	ポートアウトプットイネーブル3（POE3）
GPTa	GPT0 ～ GPT3	汎用PWMタイマ（GPT）チャネル0～3
CMT	ユニット0（CMT0, CMT1）	コンペアマッチタイマ（CMT）ユニット0（チャネル0, 1）
	ユニット1（CMT2, CMT3）	コンペアマッチタイマ（CMT）ユニット1（チャネル2, 3）
WDT	ウォッチドッグタイマ（WDT）	ウォッチドッグタイマ（WDT）
IWDT	独立ウォッチドッグタイマ（IWDT）	独立ウォッチドッグタイマ（IWDT）
SCIb	SCI0～SCI2	シリアルコミュニケーションインタフェース（SCIb）チャネル0～2
CRC	CRC演算器（CRC）	CRC 演算（CRC）
RIIC	RIIC0	I2Cバスインタフェース（RIIC）チャネル0
RSPI	RSPI0	シリアルペリフェラルインタフェース（RSPI）チャネル0
LIN	LIN0	LINモジュール（LIN）チャネル0
S12ADA	S12AD0, S12AD1	12 ビットA/D コンバータ（S12ADA）ユニット0, 1
ADA	AD0	10ビットA/Dコンバータ(ADA) ユニット0

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。

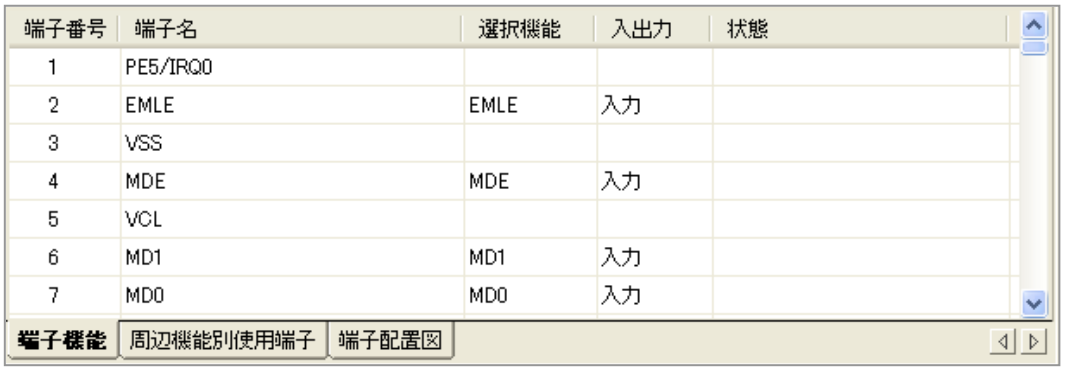


図3.3 端子機能ウィンドウ 端子機能シート

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（端子に割り当てられる全機能）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 112 ピンのパッケージにおいて、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 86 番の端子(P60/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
86	P60/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 P60 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
⚠86	P60/AN0	AN0/P60		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

- RX62G 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 112 ピンのパッケージにおいて、周辺機能の設定で外部割込み IRQ0 を設定すると、IRQ0 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
IRQ0	外部割り込み入力	P10/MTCLKD/IRQ0	110	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 P10 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
⚠ IRQ0	外部割り込み入力	P10/MTCLKD/IRQ0	110	入力	他の機能と競合しています。

図3.8 端子競合時の表示例

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

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るには、ドロップダウンボタンをクリックし、表示された選択肢から割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

IRQ0 の割当先を PE5/IRQ0 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「付録 1 割当先を変更できる端子機能」に示します。

4. チュートリアル

本章では、Peripheral Driver Generator と High-performance Embedded Workshop を使用して RX62G 用 Renesas Starter Kit のボードを動作させる以下のチュートリアルプログラムの作成方法を示しながら、Peripheral Driver Generator の使用手順を紹介します。

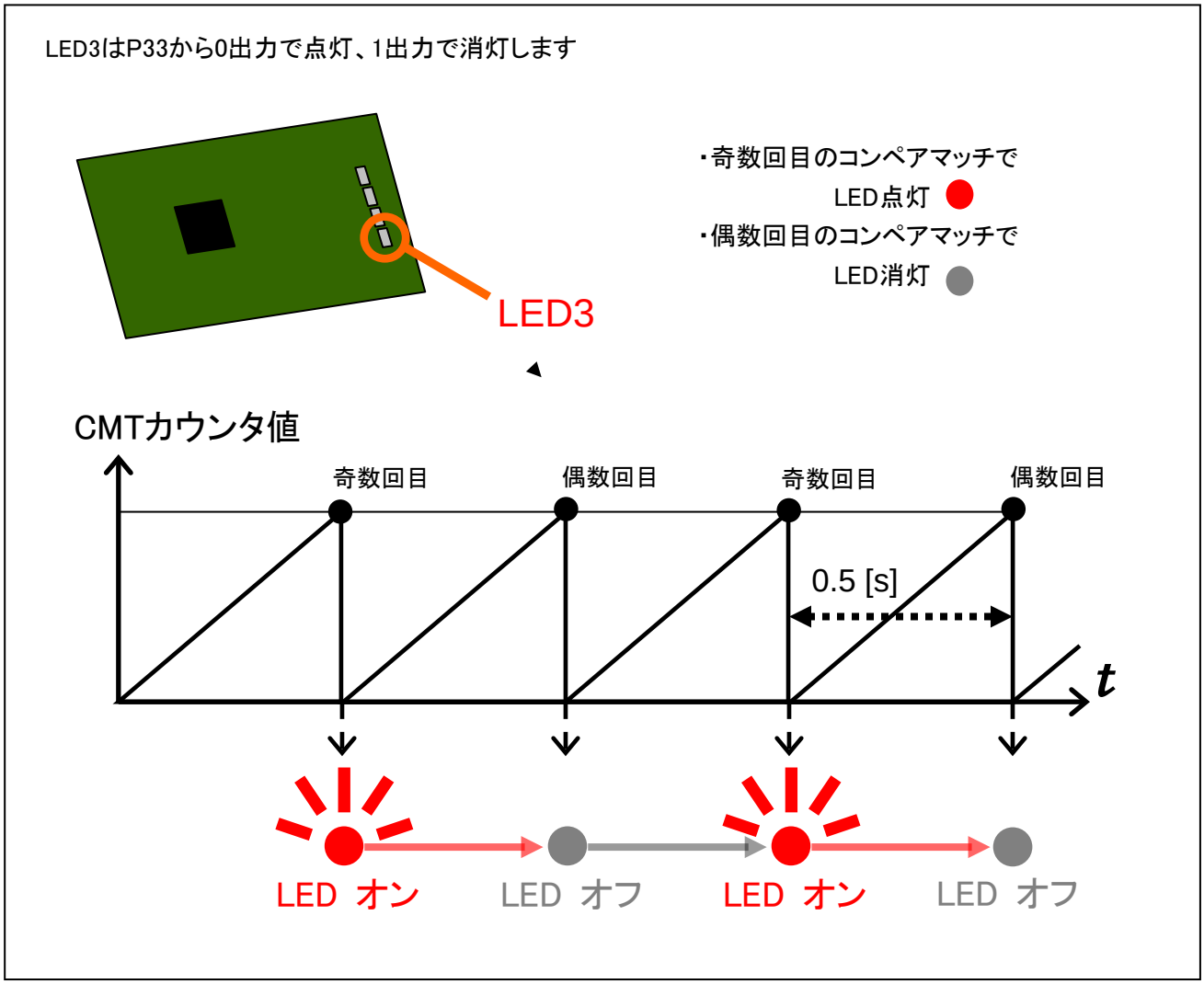
- コンペアマッチタイマ(CMT)の割り込みで LED を点滅
- マルチファンクションタイマパルスユニット 3(MTU3)の PWM 波で LED を点滅
- 10 ビット A/D コンバータ（ADA）の連続スキャン
- IRQ による DTC 転送のトリガ
- SCIb チャネル 0 とチャネル 2 で調歩同期通信

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

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

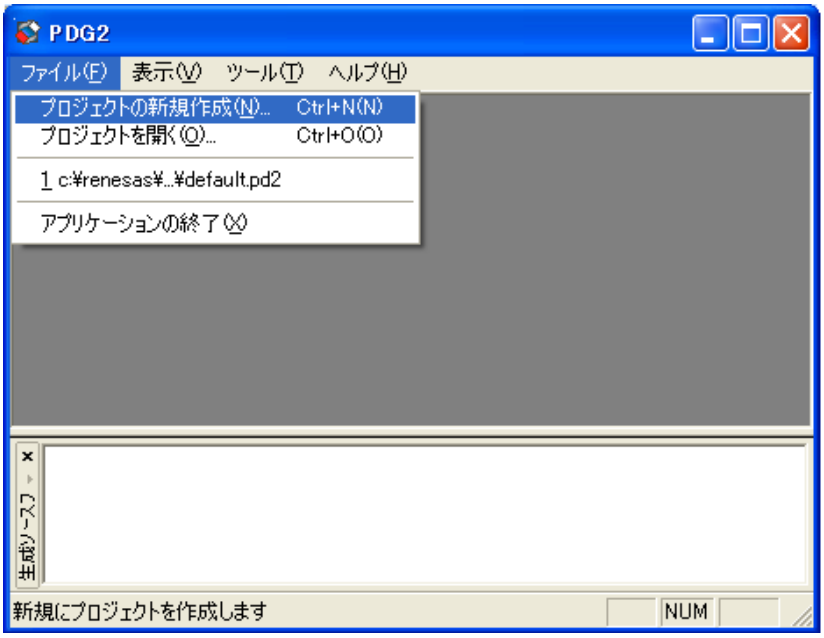
4.1 コンペアマッチタイマ(CMT)の割り込みでLEDを点滅

RX62G RSK ボード上の LED3 は P33 に接続されています。 このチュートリアルではコンペアマッチタイマ(CMT)と I/O ポートを設定し、LED を次のように点滅させます。

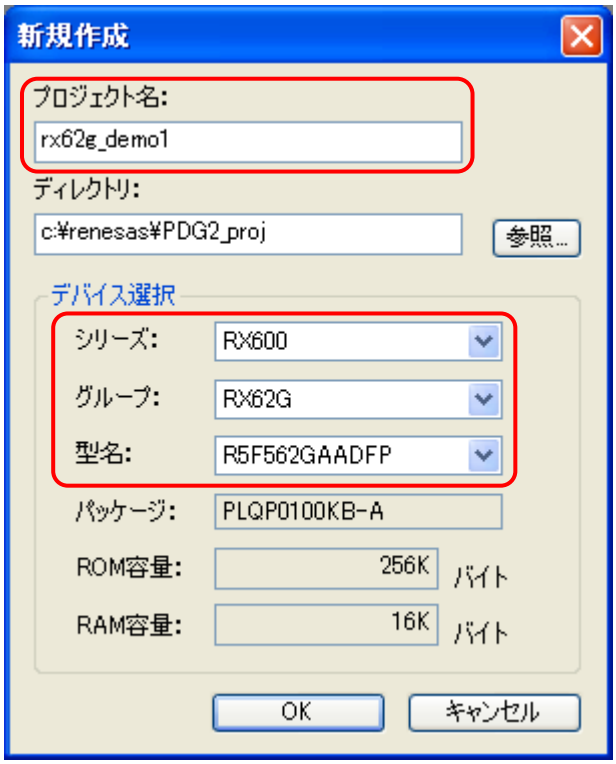


(1) Peripheral Driver Generator プロジェクトの作成 **PDG**

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

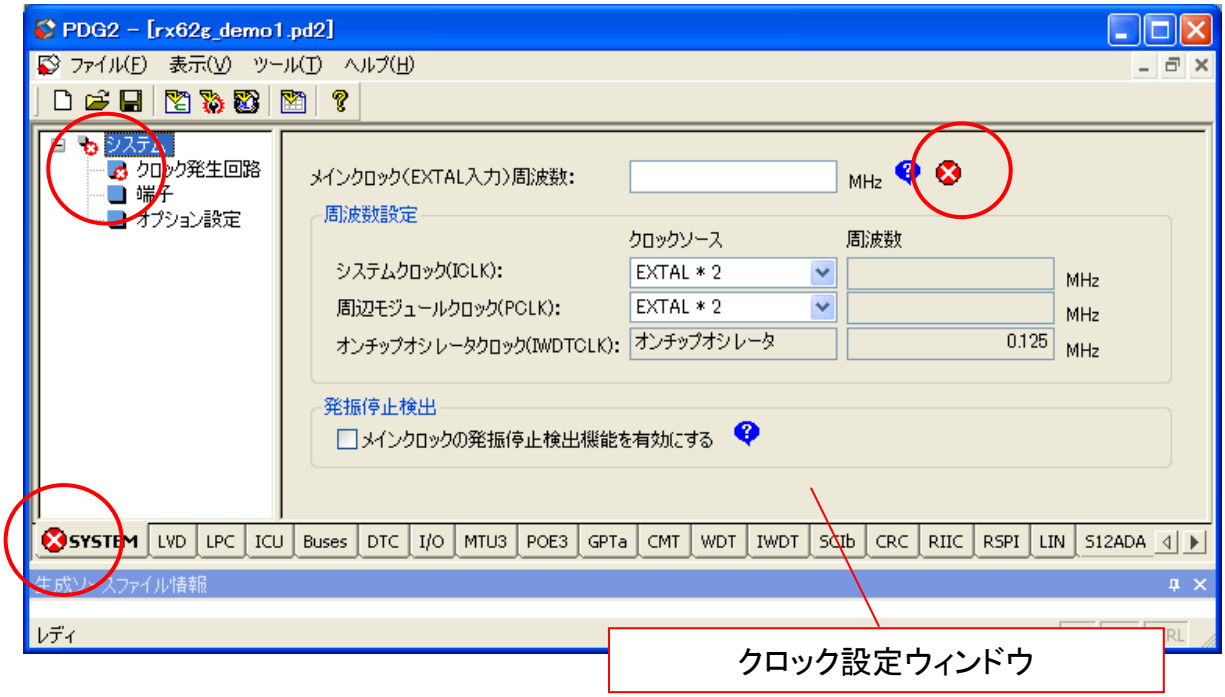


3. プロジェクト名に“rx62g_demo1”を指定してください。
   CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。
   シリーズ : RX600
   グループ : RX62G
   型名 : R5F562GAADFP

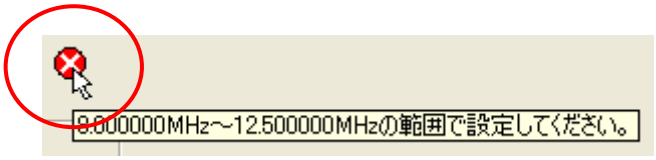


(2) 初期状態 **PDG**

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



・エラーアイコンの上にマウスポインタを置くと、エラーの内容が表示されます。



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

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

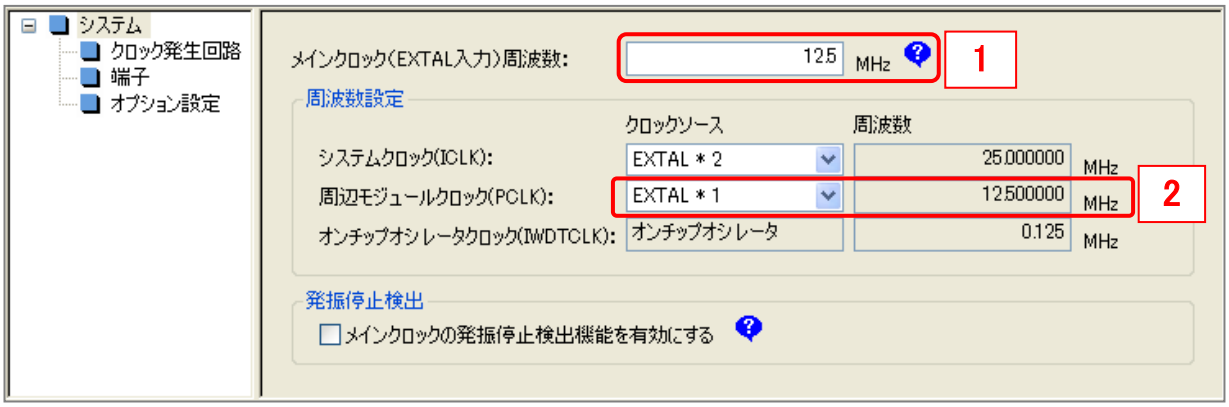
警告
設定は可能ですが、誤っている可能性があります。
ソースファイルの生成は可能です。

インフォメーション
複雑な設定箇所の付加情報です。

設定ウィンドウ上のアイコンのみツールチップを表示できます。

(3) クロックの設定 **PDG**

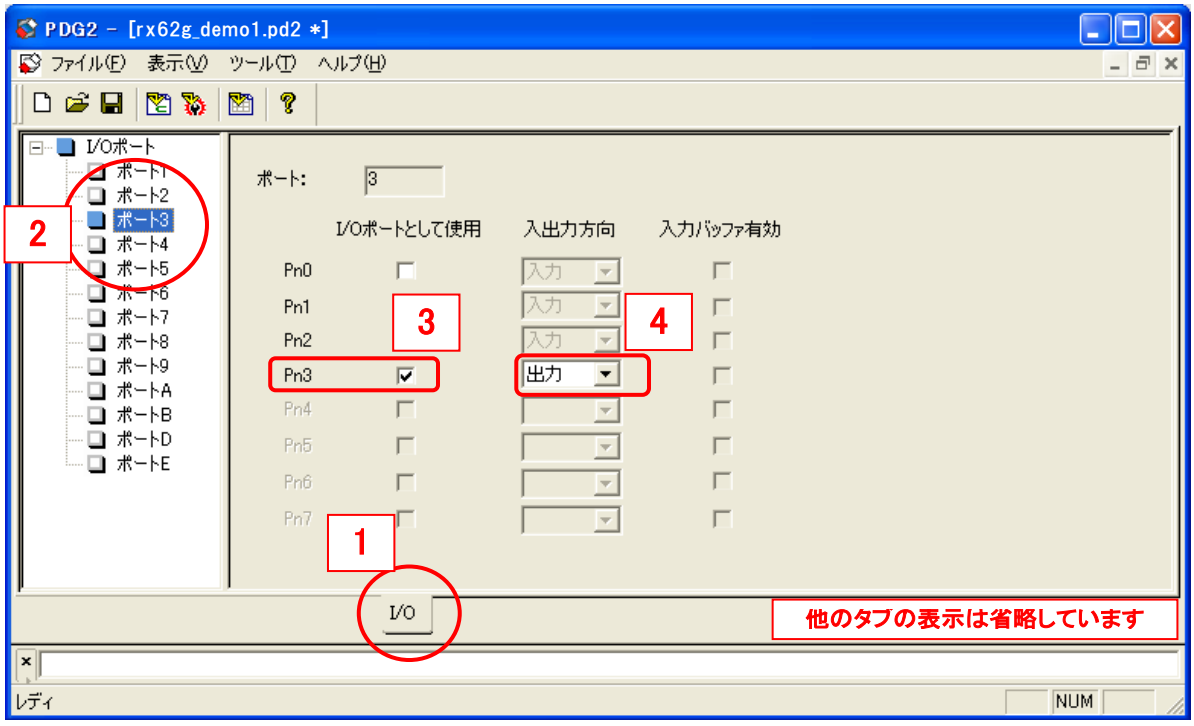
1. 最初にメインクロック(EXTAL 入力)周波数を設定してください。
   RSK ボードの外部入力周波数は 12.5MHz です。“12.5”と入力してください。
2. 周辺モジュールクロック(PCLK)は 12.5MHzで使用します。
   PCLK の倍率に“EXTAL *1”を選択し、PCLK 周波数を 12.5MHzに設定してください。



(4) I/O ポートの設定 **PDG**

LED3 が接続されている P33 を出力ポートに設定します。

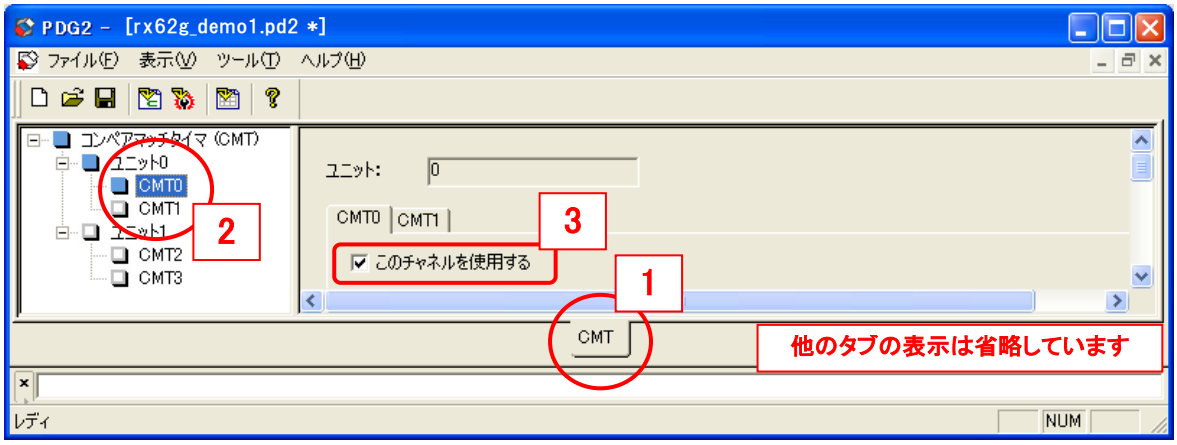
1. [I/O] タブを選択してください
2. [ポート3] を選択してください
3. [Pn3] をチェックしてください
4. [出力] を選択してください



(5) CMT の設定-1 PDG

このチュートリアルでは CMT (コンペアマッチタイマ) のユニット 0 の CMT0 を使用します。

1. [CMT] タブを選択してください。
2. [CMT0] を選択してください。
3. [このチャネルを使用する] を選択してください。

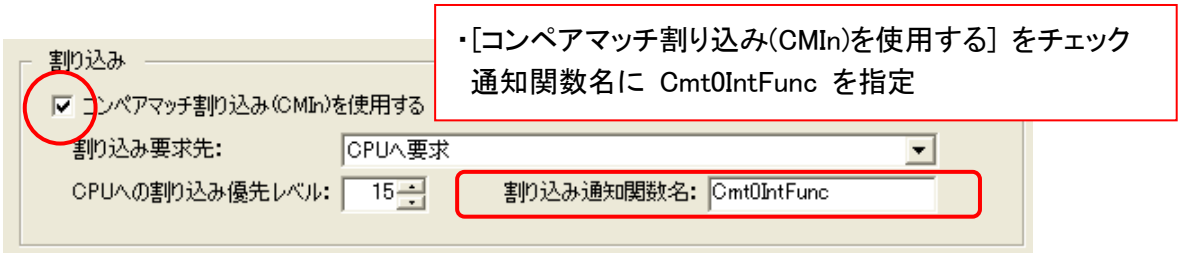


(6) CMT の設定-2 PDG

CMT の他の項目を以下の通り設定してください。

(7) CMT の設定-3 PDG

割り込み通知関数を設定します。
この関数は割り込みが発生すると呼ばれます。



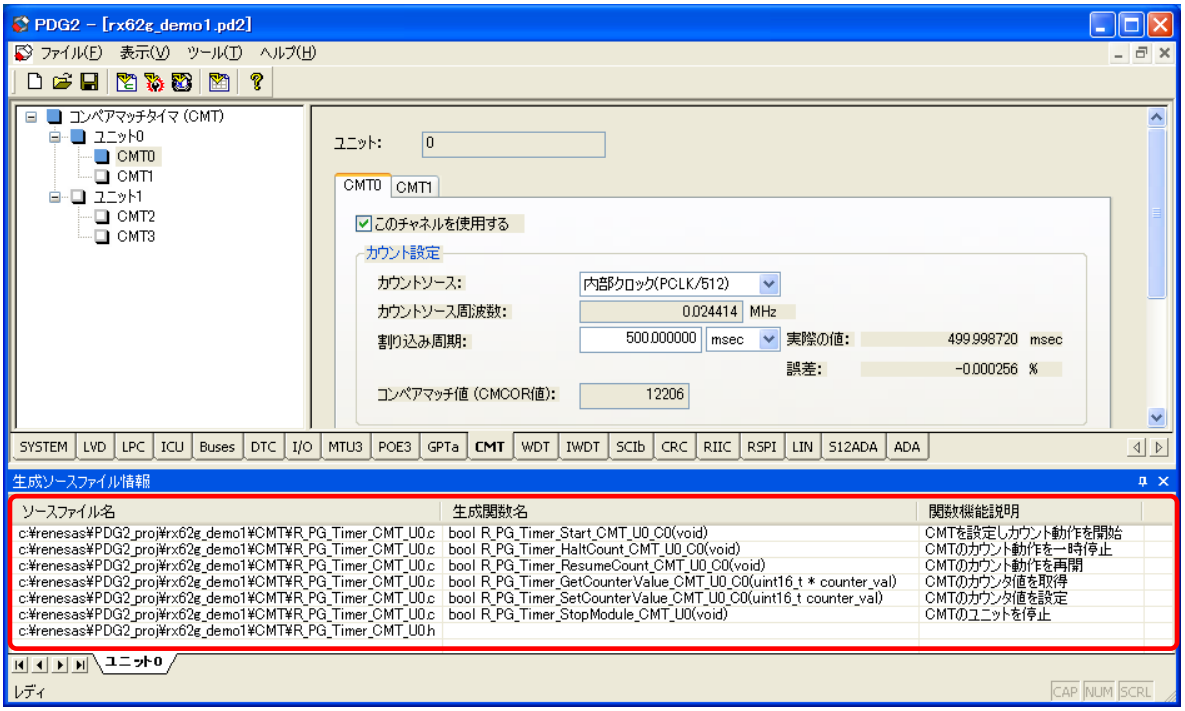
(8) ソースファイルの生成 PDG

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。

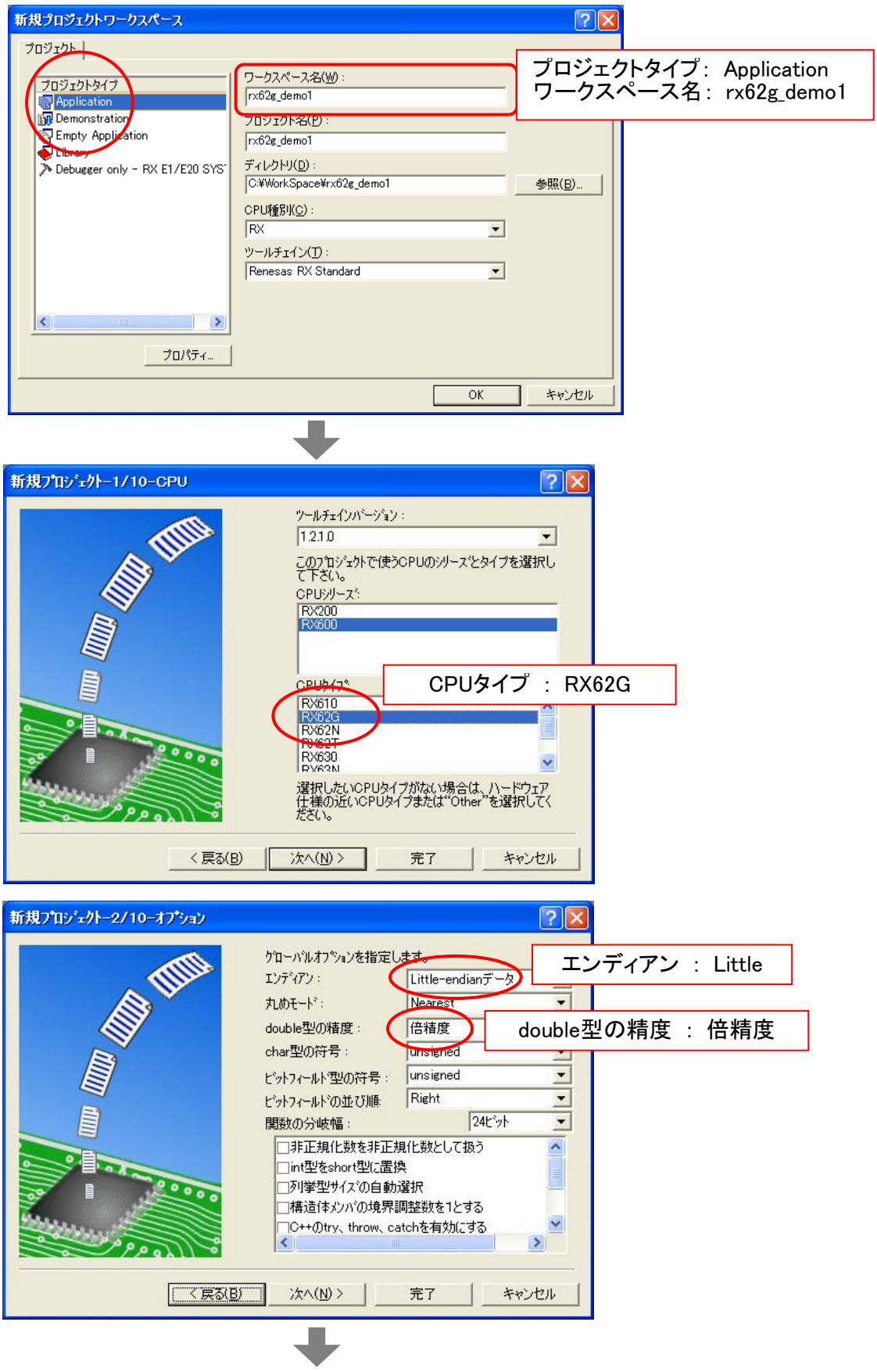
3. 登録の完了を示すダイアログボックスが表示されます。 [OK]をクリックしてください。

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。

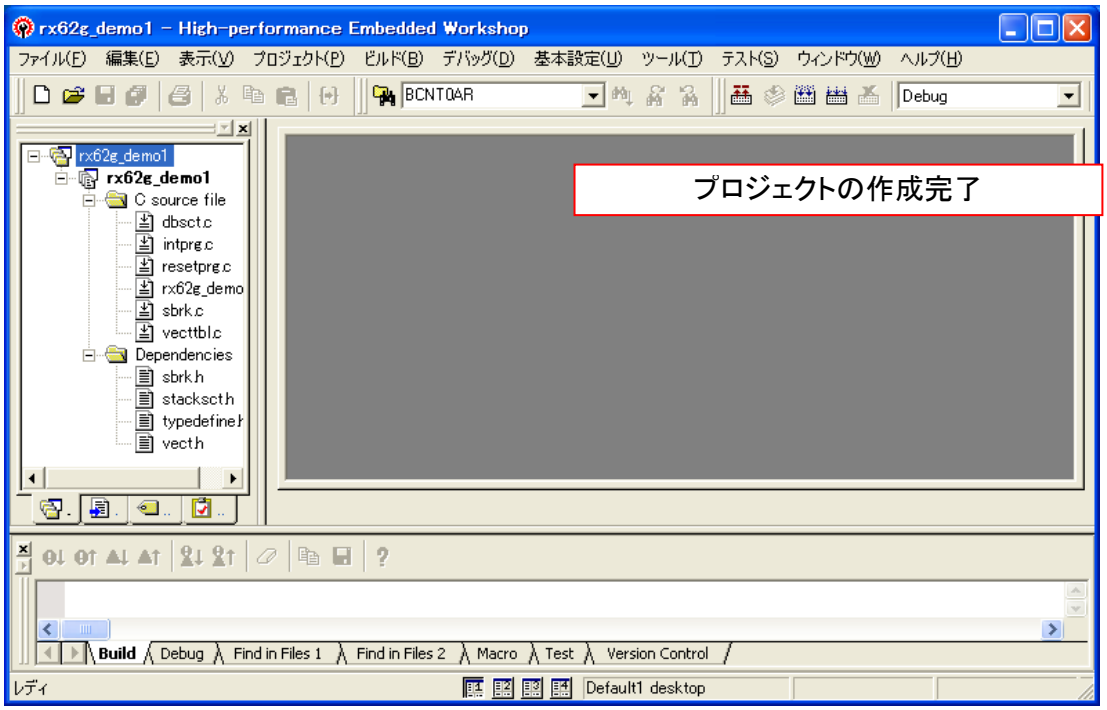


(9) High-performance Embedded Workshop プロジェクトの準備 HEW

High-performance Embedded Workshop を起動し、RX62G 用の新規ワークスペースを作成します。



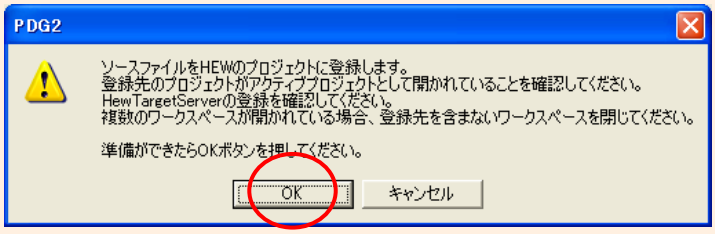

新規プロジェクト-8/10-デバッガ
ターゲットエミュレータを指定
ターゲット :
RX E1/E20 SYSTEM
RX600 Simulator
ターゲットタイプ :
RX600
ターゲットCPU :
RX62G
< 戻る(B)
次へ(N) >
完了
キャンセル


rx62g_demo1 - High-performance Embedded Workshop
ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) 基本設定(U) ツール(T) テスト(S) ウィンドウ(W) ヘルプ(H)
BCNT0AR
Debug
プロジェクトの作成完了
rx62g_demo1
C source file
dbsct.c
intprg.c
resetprg.c
sbrk.c
vecttbl.c
Dependencies
sbrk.h
stacksct.h
typedefine.h
vect.h
Build
Debug
Find in Files 1
Find in Files 2
Macro
Test
Version Control
レディ
Default1 desktop

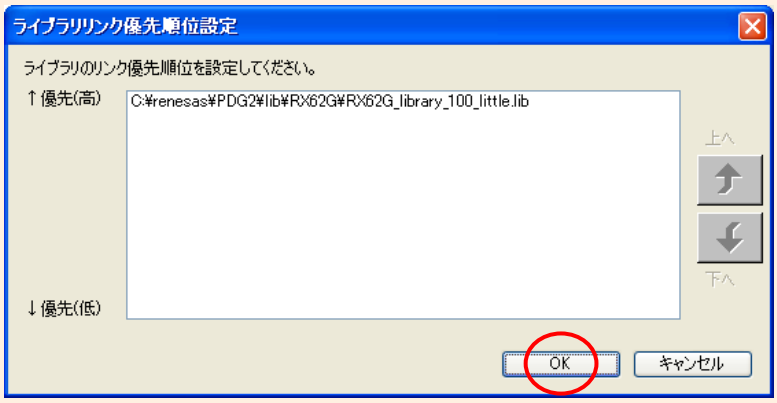
(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録

PDG

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

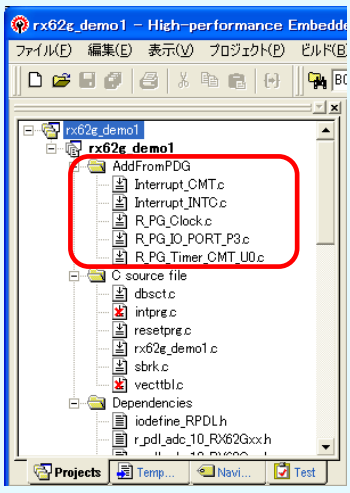


3. Renesas Peripheral Driver Library とのリンク設定のためのダイアログが開きます。
   複数の lib ファイルとリンクする場合、このダイアログ上でリンク順を設定できます。



HEW

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



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

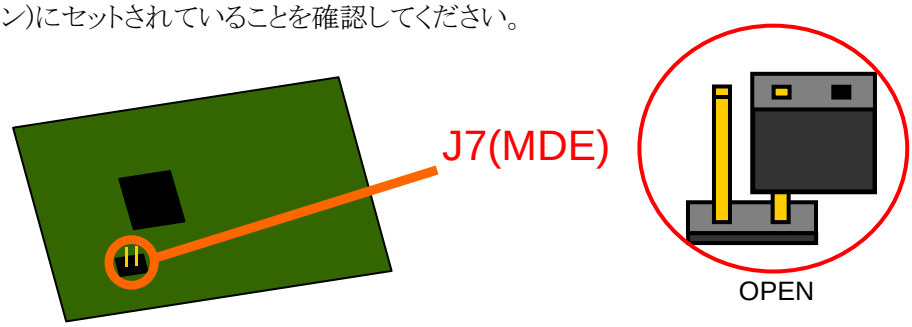
(11) プログラムの作成 **HEW**

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

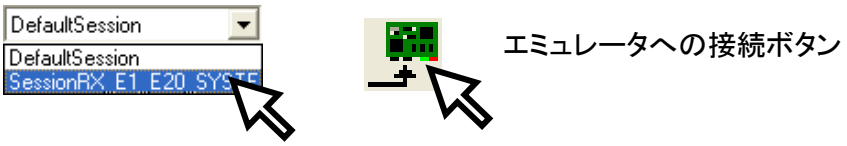
```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx62g_demo1.h"

bool led=false;

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_Write_P33(1); //初期出力値
    R_PG_IO_PORT_Set_P33();

    //CMT0を設定しカウントを開始
    R_PG_Timer_Start_CMT_U0_C0();

    while(1);
}
//コンペアマッチ割り込みの通知関数
void CmtOIntFunc(void)
{
    if( led ){
        //LED消灯
        R_PG_IO_PORT_Write_P33(1);
        led = false;
    }
    else{
        //LED点灯
        R_PG_IO_PORT_Write_P33(0);
        led = true;
    }
}
```

(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J7(MDE)がOPEN(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。



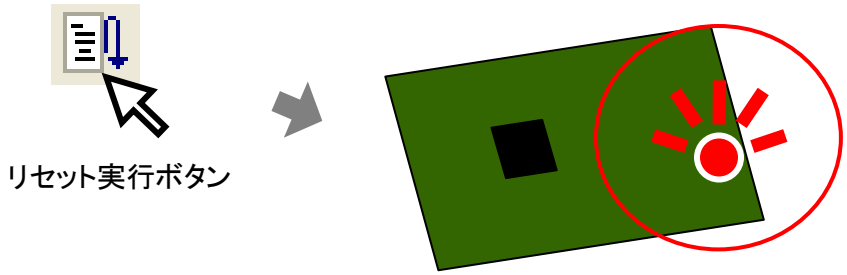
2. エミュレータに接続してください。



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

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。

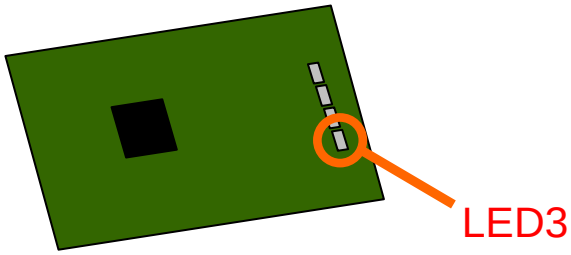


4.2 マルチファンクションタイマパルスユニット 3(MTU3)のPWM波でLEDを点滅

RX62G RSK ボードでは P33 端子に LED3 が接続されています。P33 はマルチファンクションタイマパルスユニット 3(MTU3)の PWM 波形出力端子(MTIOC3A) としても使用することができます。このチュートリアルではマルチファンクションタイマパルスユニット 3(MTU3)を PWM モード 1 で動作させ、その出力パルスで LED を点滅させます。

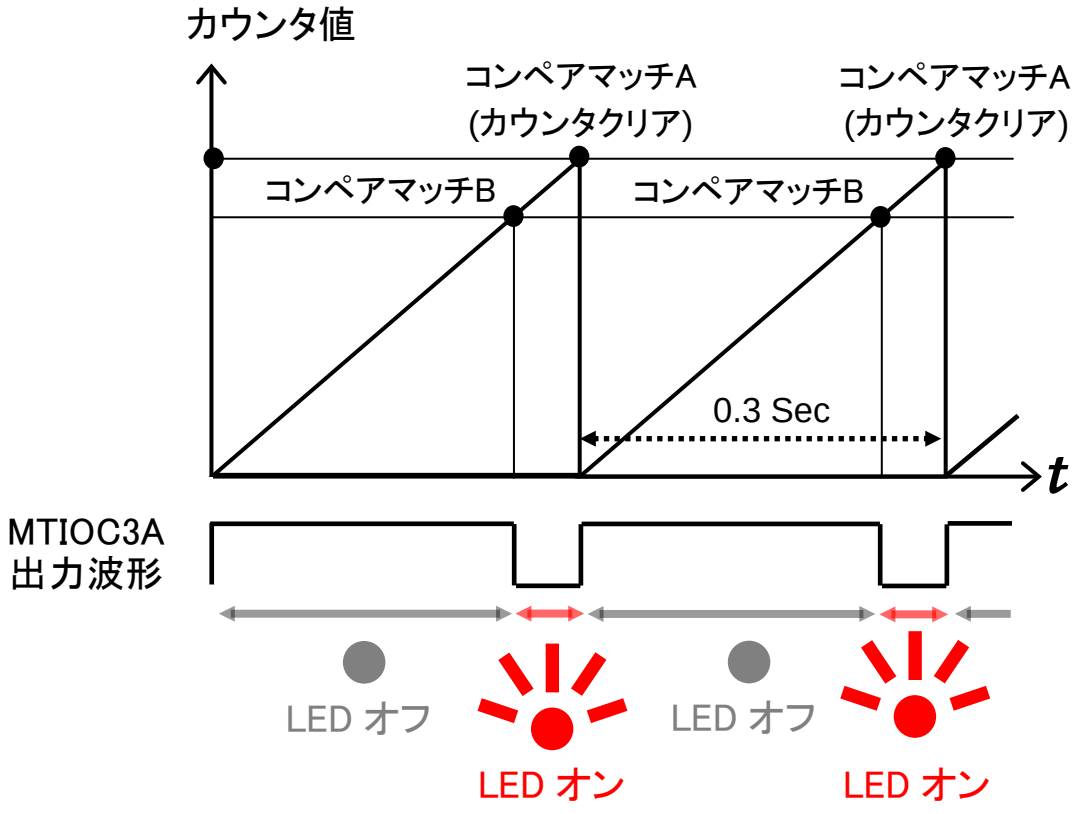
使用するRSKボード上にP33(MTIOC3A)の有効/無効を切り替えるスイッチがある場合は有効にしてください。

LED3はP33から0出力で点灯、1出力で消灯します



MTU3のチャネル3(MTU3_3)をPWMモード1で動作させます。PWMモード1は、コンペアマッチAおよびBでMTIOC3Aの出力レベルを制御するモードです。

設定するタイマの動作
- ・コンペアマッチBで0出力 → LED点灯
- ・コンペアマッチAで1出力 → LED消灯
- ・コンペアマッチAでカウンタクリア （カウンタクリア周期は0.3sec）

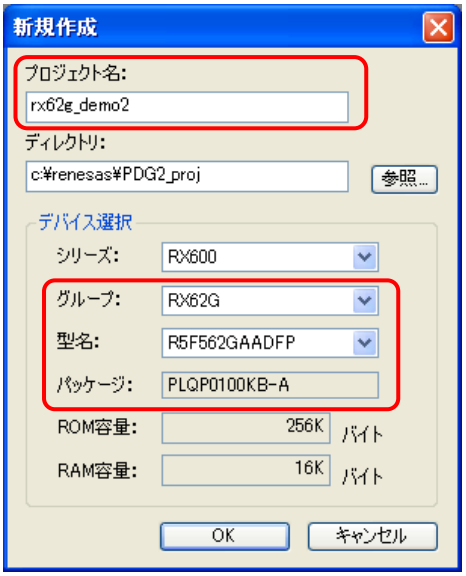


(1) Peripheral Driver Generator プロジェクトの作成 **PDG**

プロジェクト名に“rx62g_demo2”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1(1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

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

シリーズ : RX600
グループ : RX62G
型名 : R5F562GAADFP



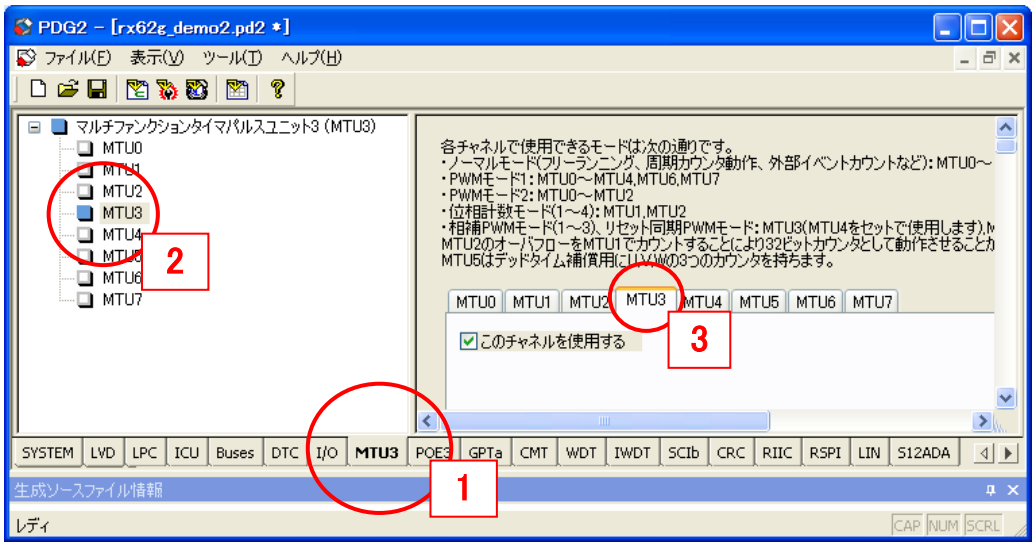
(2) クロックの設定 **PDG**

1. プロジェクトを作成するとクロック設定ウィンドウが開きます。設定画面上の や などのアイコンについては、「4.1(2) 初期状態」を参照してください。
2. 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。
3. 周辺モジュールクロック(PCLK)は 12.5MHzで使用します。
   PCLK の倍率に“EXTAL *1”を選択し、PCLK 周波数を 12.5MHzに設定してください。

(3) MTU3 の設定-1 PDG

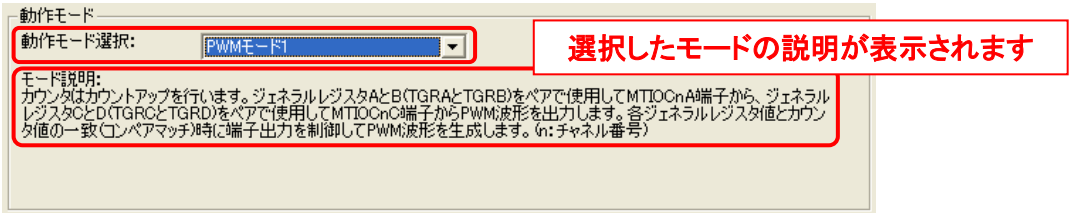
MTU3 チャネル 3(MTU3_3)を設定します。

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



(4) MTU3 の設定-2 PDG

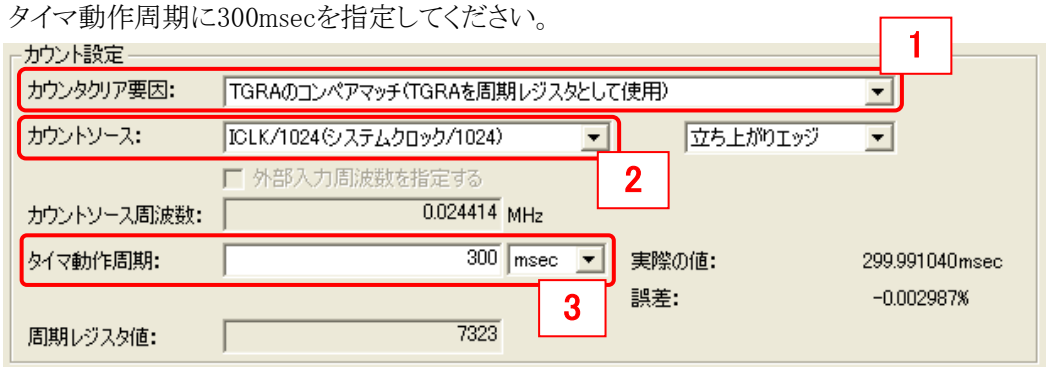
動作モードに[PWM モード 1]を指定してください。



(5) MTU3 の設定-3 PDG

以下の通りカウンタの動作を設定してください。

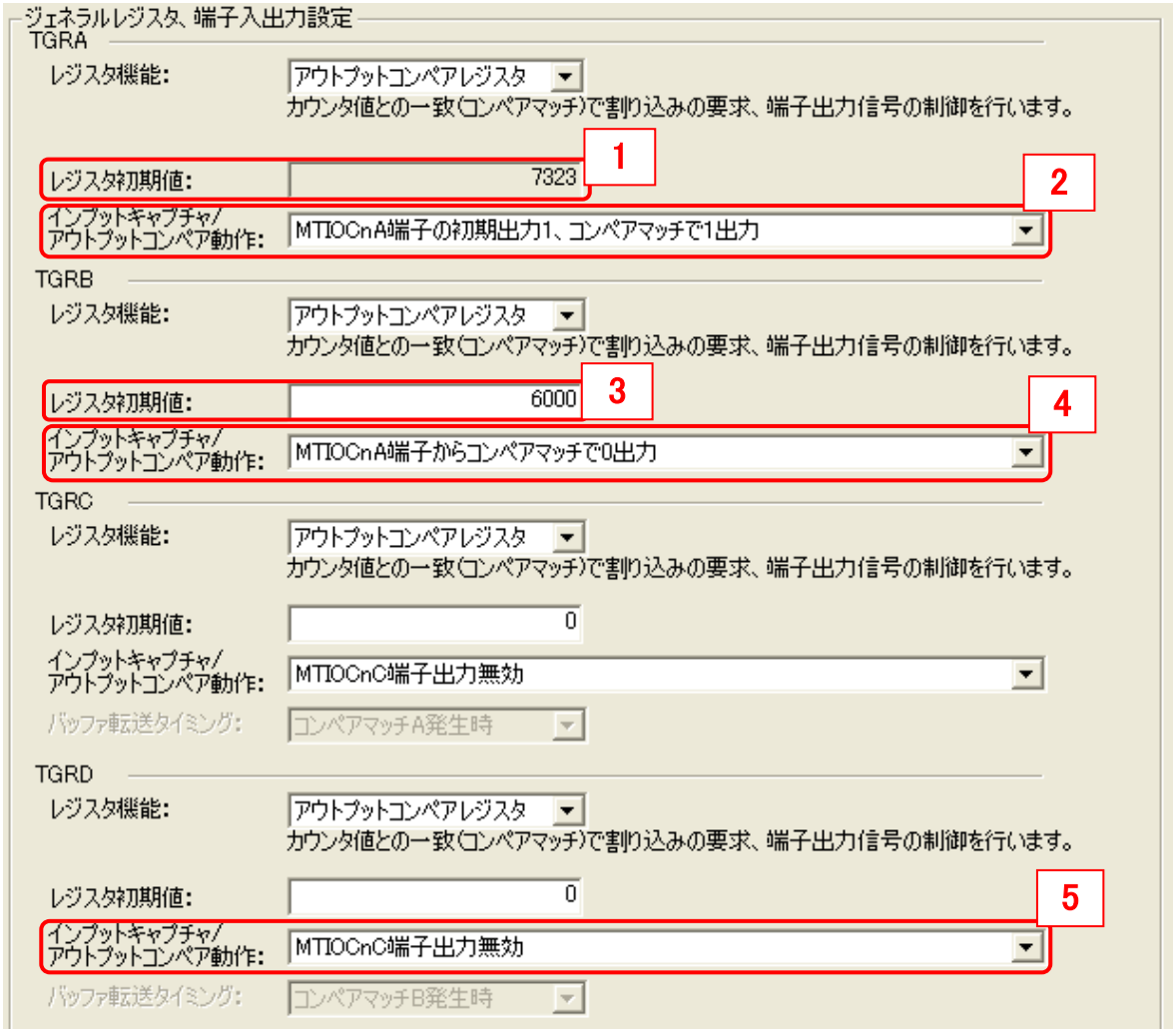
1. カウンタクリア要因に[TGRAのコンペアマッチ] を選択してください。
2. カウントソースに[ICLK/1024(システムクロック/1024)] を選択してください。
3. タイマ動作周期に300msecを指定してください。



(6) MTU3 の設定-4 PDG

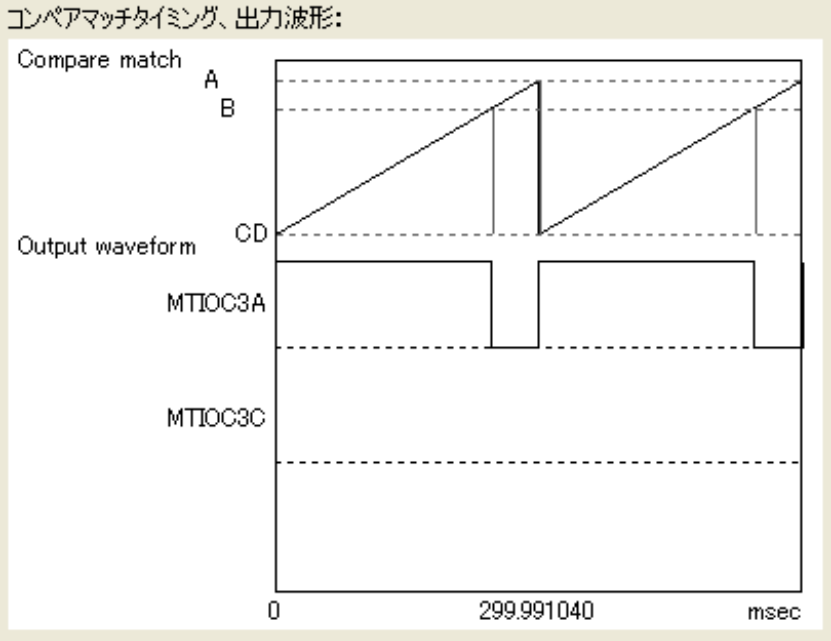
以下の通りジェネラルレジスタを設定してください。

1. カウント設定においてカウンタクリア要因にコンペアマッチAを指定したので、TGRAの値はカウントソース周波数と入力したタイマ動作周期を元に算出されます。
2. TGRAのアウトプットコンペア動作に[MTIOCnA端子の初期出力1、コンペアマッチで1出力]を選択してください。
3. TGRBのレジスタ初期値に6000を設定してください。
4. TGRBのアウトプットコンペア動作に[MTIOCnA端子からコンペアマッチで0出力]を選択してください。
5. TGRCとTGRDをペアで使用すると、MTIOCnC端子からPWM出力することが可能です。ここでは使用しませんので、TGRDのアウトプットコンペア動作には[MTIOCnC端子出力無効]を選択してください。



(7) MTU3 の設定-5 PDG

設定内容に応じて、コンペアマッチのタイミングと出力波形が図示されます。



(8) ソースファイルの生成 PDG

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1(8) ソースファイルの生成」を参照してください。

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

High-performance Embedded Workshop を起動し、RX62G 用のワークスペースを作成してください。作成方法については「4.1(9) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

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

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(11) プログラムの作成 HEW

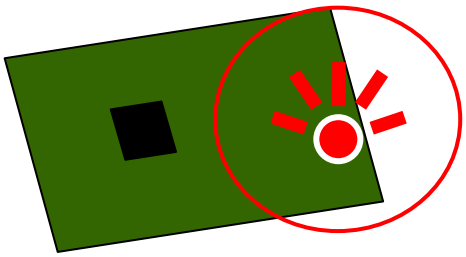
High-performance Embedded Workshop 上で main 関数の部分を変更し、以下のプログラムを作成してください。

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx62g_demo2.h"

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

    //MTU3チャネル3の設定
    R_PG_Timer_Set_MTU_U0_C3();

    //MTU3チャネル3のカウント開始
    R_PG_Timer_StartCount_MTU_U0_C3();

    while(1);
}
```

(12) エミュレータの接続、プログラムのビルド、実行 HEW

作成したプログラムをビルドし、実行してください。 LED が点滅します。

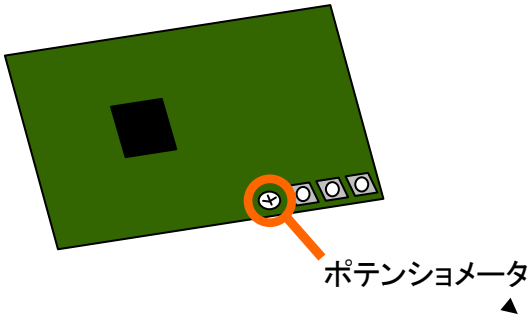
エミュレータの接続、プログラムのビルド、実行の方法については「4.1(12) エミュレータの接続、プログラムのビルド、実行」 を参照してください。



4.3 10 ビットA/Dコンバータ（ADA）の連続スキャン

RX62G 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 High-performance Embedded Workshop 上でリアルタイムに確認します。



使用する RSK ボード上に AN0 の有効/無効を切り替えるスイッチがある場合は有効にしてください。

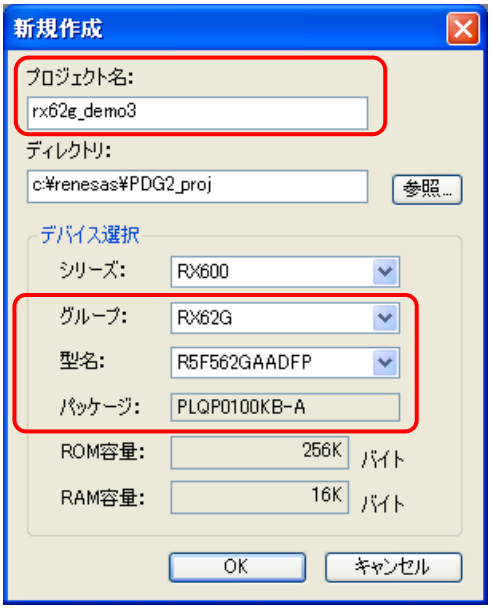
(1) Peripheral Driver Generator プロジェクトの作成

PDG

プロジェクト名に“rx62g_demo3”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1(1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

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

シリーズ ：RX600
グループ ：RX62G
型名 ：R5F562GAADFP

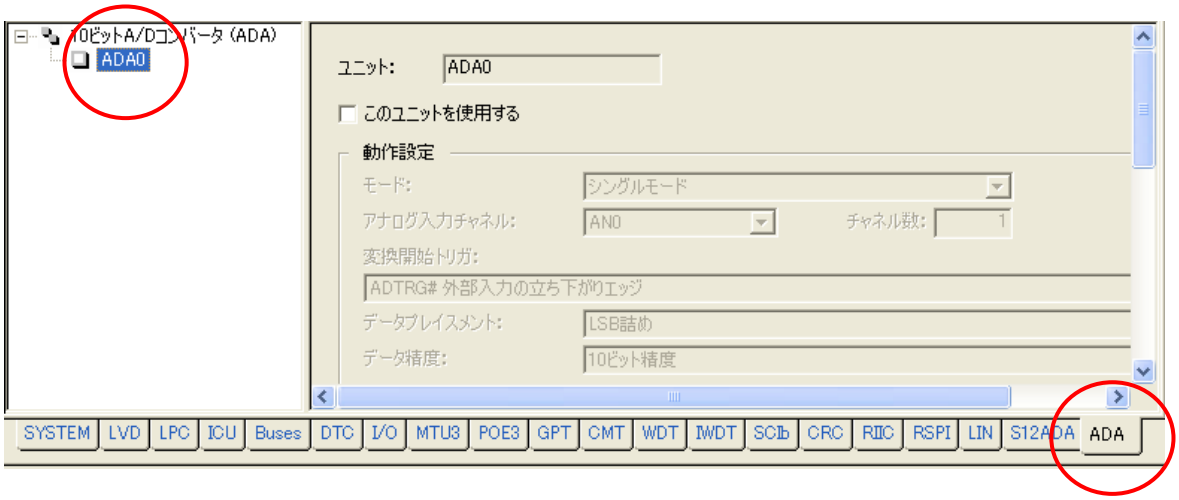


(2) クロックの設定 **PDG**

1. プロジェクトを作成するとクロック設定ウィンドウが開きます。設定画面上の や などのアイコンについては、「4.1(2) 初期状態」を参照してください。
2. 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。
3. 周辺モジュールクロック(PCLK)は 12.5MHzで使用します。
   PCLK の倍率に“EXTAL *1”を選択し、PCLK 周波数を 12.5MHzに設定してください。

(3) A/D 変換器の設定-1 **PDG**

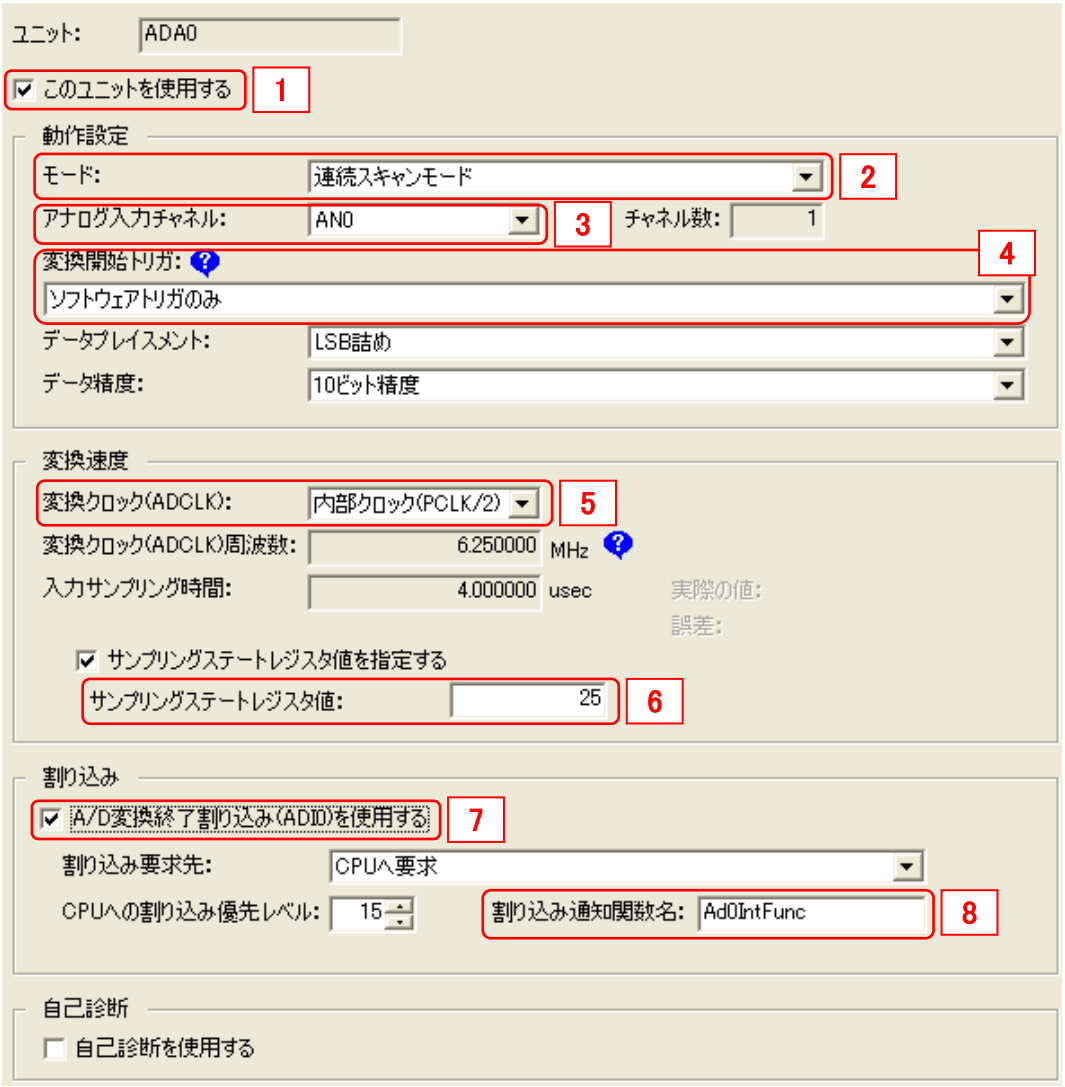
ADA タブを選択し、ツリー表示上で ADA0 を選択してください。



(4) A/D 変換器の設定-2 PDG

ADA0 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 ： [連続スキャンモード]
3. アナログ入力チャネル ： [AN0]
4. 変換開始トリガ ： [ソフトウェアトリガのみ]
5. 変換クロック ： [内部クロック(PCLK/2)]
6. サンプリングステートレジスタ値 ： 25 (初期値)
7. [A/D変換終了割り込み(ADI0)を使用する]をチェック
8. A/D変換終了割り込み通知関数名 ： Ad0IntFunc



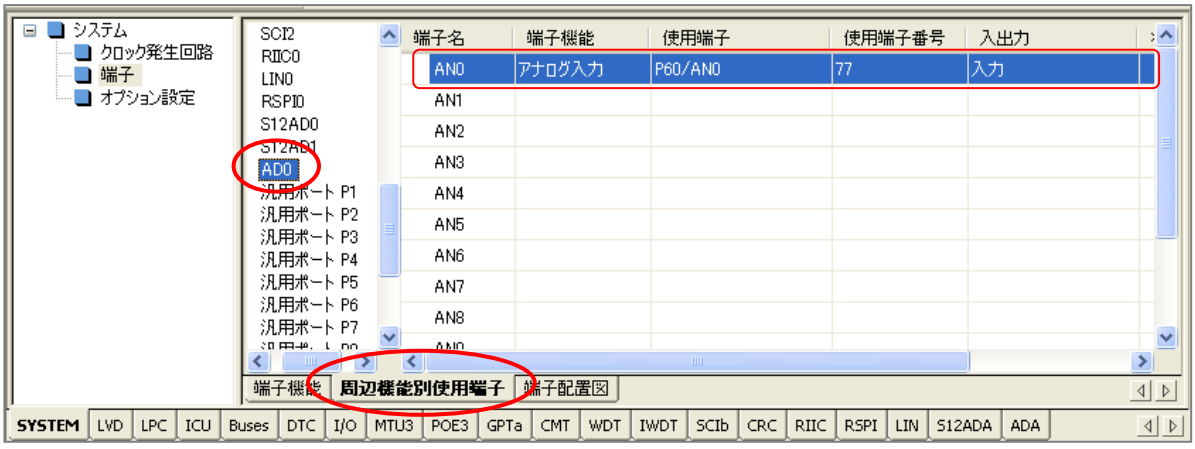
(5) 端子使用状況の確認 PDG

・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. ADA0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 77 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。



(6) ソースファイルの生成 PDG

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1(8) ソースファイルの生成」を参照してください。

(7) HHigh-performance Embedded Workshop ロジェクトの準備 HEW

High-performance Embedded Workshop を起動し、RX62G 用のワークスペースを作成してください。作成方法については「4.1(9) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

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

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(9) プログラムの作成 HEW

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

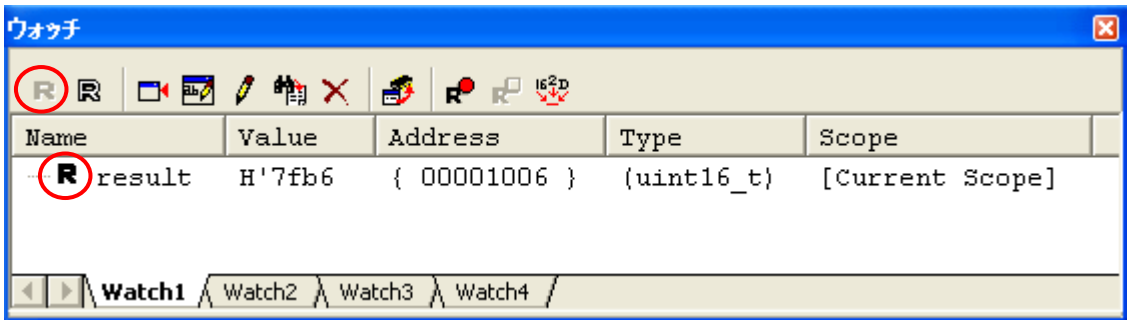
```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx62g_demo3.h"

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();

    //AN0のA/D変換開始
    R_PG_ADC_10_StartConversionSW_AD0();

    while(1);
}

//変換結果格納先変数
uint16_t result;

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

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

作成したプログラムをビルドし、ダウンロードしてください。
エミュレータの接続、プログラムのビルド方法については「4.1(12) エミュレータの接続、プログラムのビルド、実行」 を参照してください。

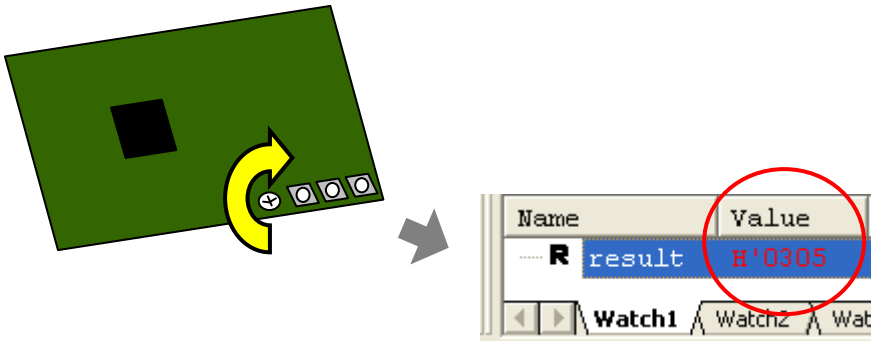
(11) A/D 変換結果格納変数のウォッチウィンドウ登録 **HEW**

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



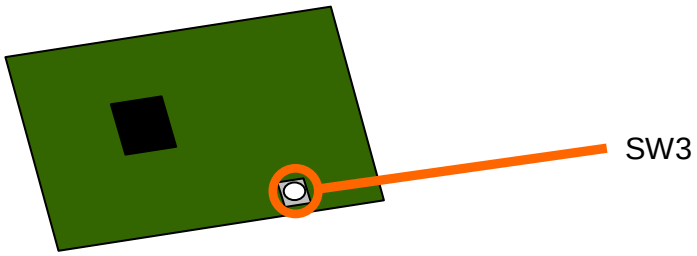
(12) プログラムの実行と A/D 変換結果の確認 **HEW**

プログラムを実行し、実行中にポテンショメータを回してアナログ入力電圧を変動させてください。
ウォッチウィンドウ上の ″result″ の値が変化します。



4.4 IRQによるDTC転送のトリガ

RX62G RSK ボードではスイッチ 3 (SW3) が IRQ3 外部割込み入力端子に接続されています。
このチュートリアルでは IRQ3 をトリガとした DTC 転送を行います。



使用する RSK ボード上に IRQ3 の有効/無効を切り替えるスイッチがある場合は有効にしてください。

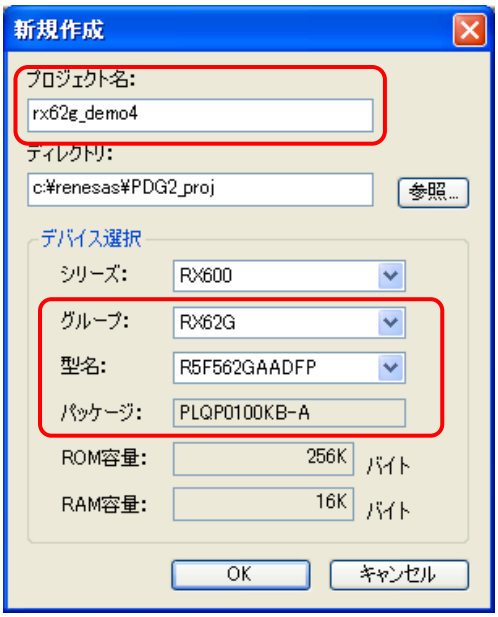
(1) Peripheral Driver Generator プロジェクトの作成

PDG

プロジェクト名に“rx62g_demo4”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1(1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

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

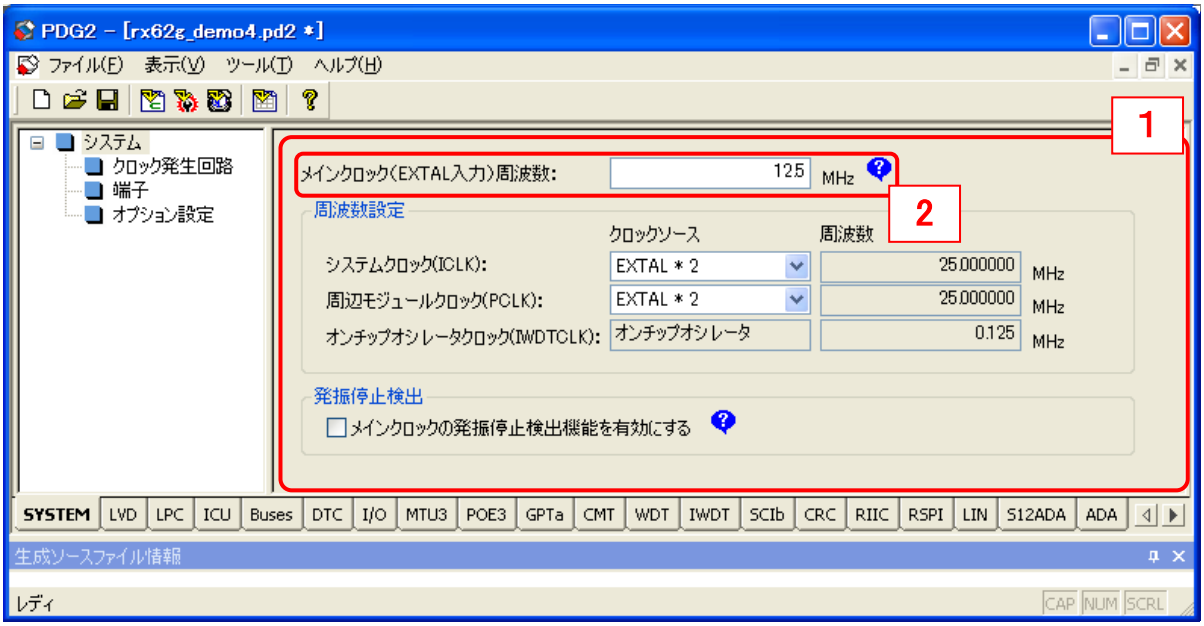
シリーズ : RX600
グループ : RX62G
型名 : R5F562GAADFP



*本チュートリアルは型名 R5F562GAxxxx にのみ対応しています。

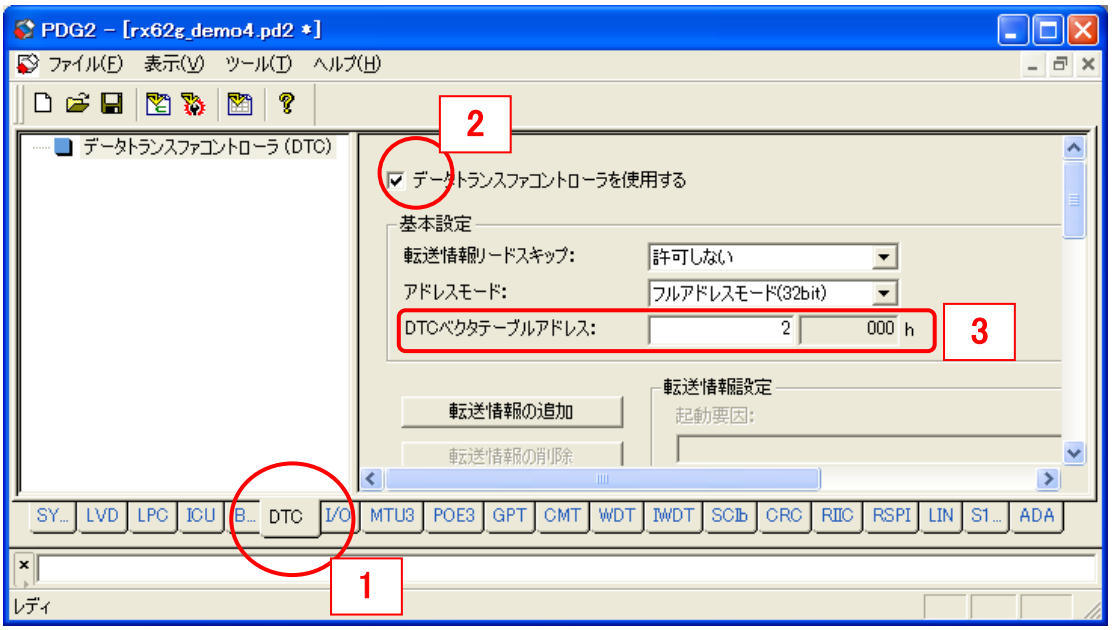
(2) クロックの設定 **PDG**

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



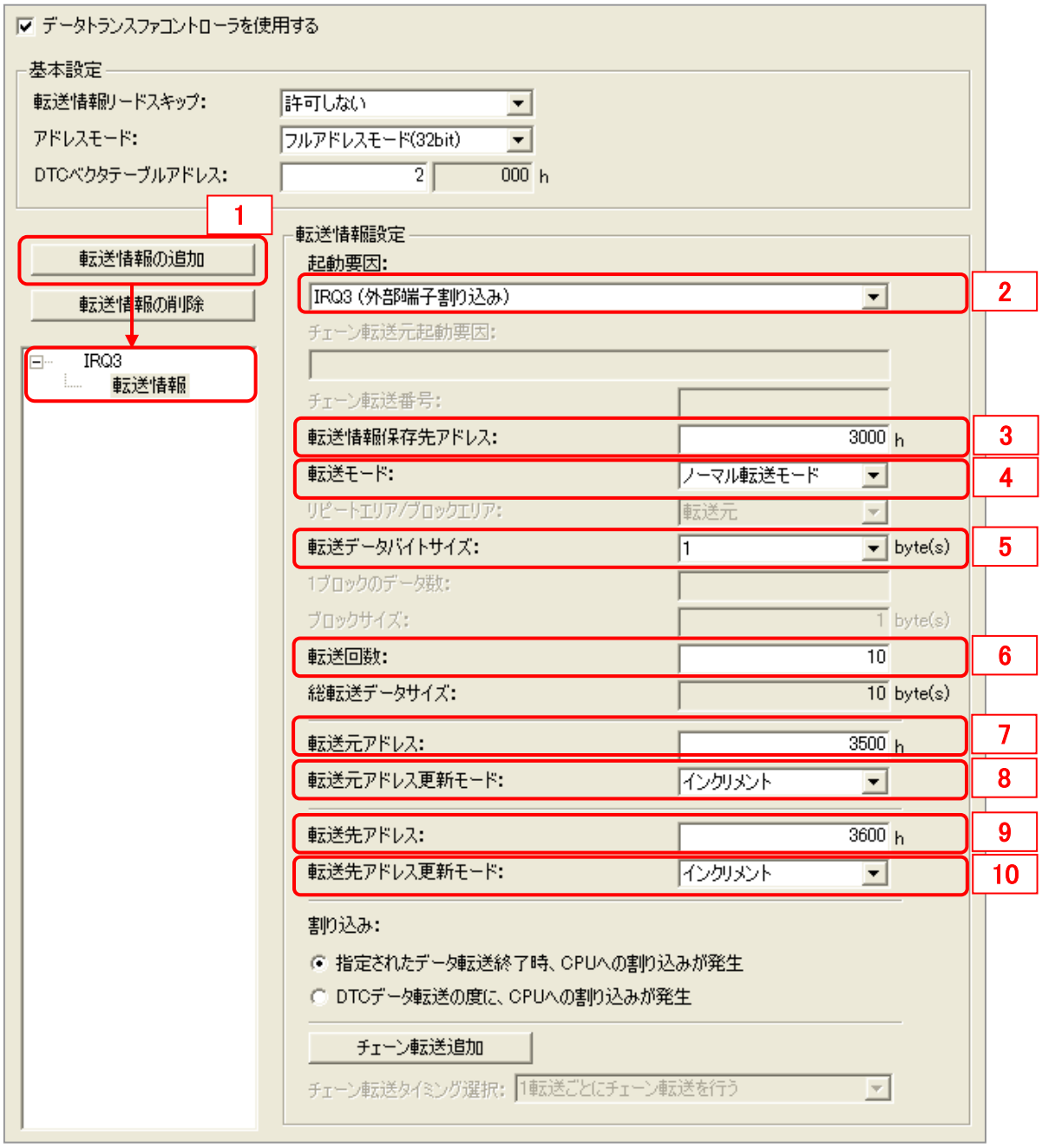
(3) DTC の設定-1 **PDG**

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



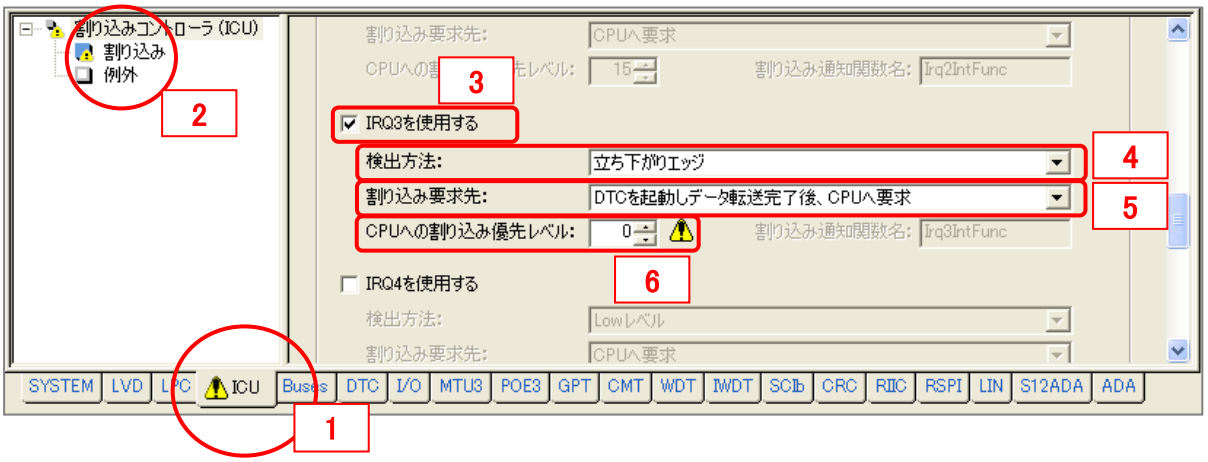
(4) DTC の設定-2 PDG

1. [転送情報の追加]ボタンをクリックすると、転送情報が追加されます。
2. 起動要因に[IRQ3 (外部端子割り込み)]を指定してください。
3. 転送情報保存先アドレスに 3000 を指定してください。
4. 転送モードに[ノーマル転送モード]を指定してください。
5. 転送データバイトサイズに 1 を指定してください。
6. 転送回数に 10 を指定してください。
7. 転送元アドレスに 3500 を指定してください。
8. 転送元アドレス更新モードに[インクリメント]を指定してください。
9. 転送先アドレスに 3600 を指定してください。
10. 転送先アドレス更新モードに[インクリメント]を指定してください。



(5) IRQ の設定 PDG

1. ICU タブを選択してください。
2. ツリー表示上で[割り込み]を選択してください。
3. [IRQ3 を使用する]をチェックしてください。
4. 検出方法に[立ち下がりエッジ]を指定してください。
5. 割り込み要求先に[DTC を起動しデータ転送完了後、CPU へ要求]を指定してください。
6. IRQ3 の CPU 割り込みは使用しません。割り込み優先レベルに 0 を指定してください。



(6) ソースファイルの生成 PDG

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1(8) ソースファイルの生成」を参照してください。

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

High-performance Embedded Workshop を起動し、RX62G 用のワークスペースを作成してください。作成方法については「4.1(9) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

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

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(9) プログラムの作成 **HEW**

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

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx62g_demo4.h"

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

//DTC転送情報保存先(IRQ3)
#pragma address dtc_transfer_data_IRQ3 = 0x00003000
uint32_t dtc_transfer_data_IRQ3 [2];

//転送元
#pragma address dtc_src_data = 0x00003500
uint8_t dtc_src_data [10] = "ABCDEFGHIJ";

//転送先
#pragma address dtc_dest_data = 0x00003600
uint8_t dtc_dest_data [10];

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

    //クロックの設定
    R_PG_Clock_Set();

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

    //DTCの設定(IRQ3をトリガとする転送の設定)
    R_PG_DTC_Set_IRQ3();

    //IRQ3の設定
    R_PG_ExtInterrupt_Set_IRQ3();

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

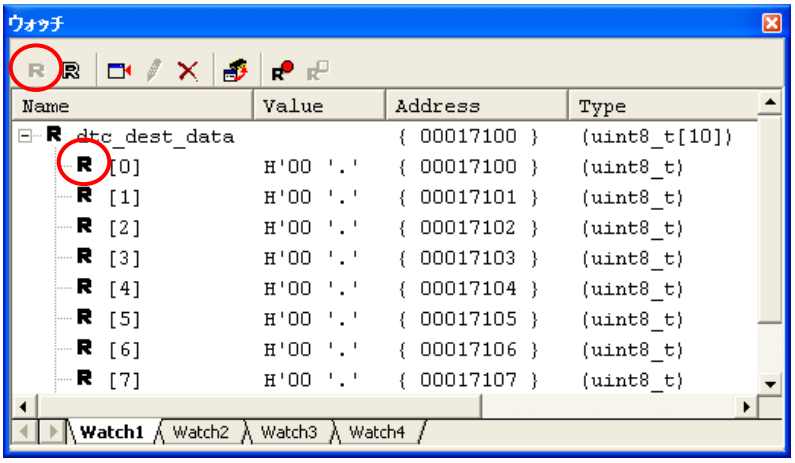
(10) エミュレータの接続、プログラムのビルド、ダウンロード **HEW**

作成したプログラムをビルドし、ダウンロードしてください。

エミュレータの接続、プログラムのビルド方法については「4.1(12) エミュレータの接続、プログラムのビルド、実行」 を参照してください。

(11) 転送先変数のウォッチウィンドウ登録 **HEW**

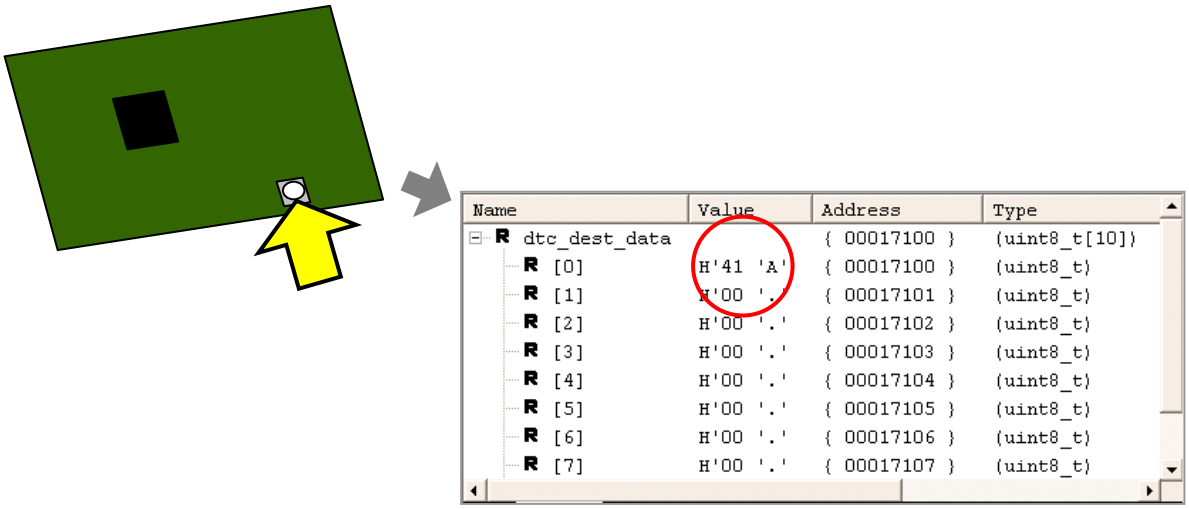
High-performance Embedded Workshop のウォッチウィンドウを開き、転送先変数 ″dtc_dest_data″ を登録してください。″dtc_dest_data″を展開しリアルタイム更新に設定すると、実行中に値の変化を確認することができます。



(12) プログラムの実行と転送結果の確認 **HEW**

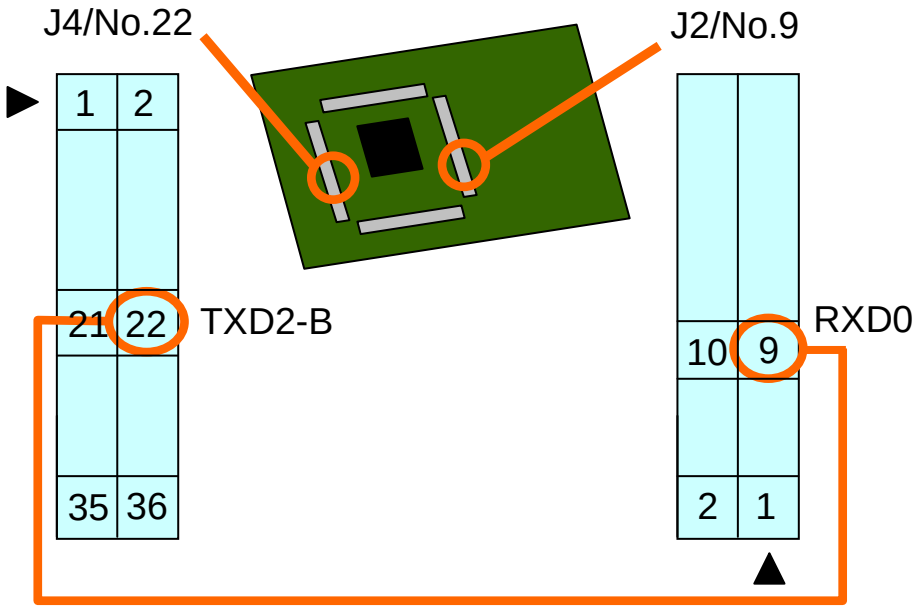
プログラを実行し、実行中 SW3 を押して IRQ3 割り込みを発生させてください。

ボタンを押すたびにデータが転送されます。



4.5 SCIb チャネル 0 とチャネル 2 で調歩同期通信

このチュートリアルでは、シリアルのチャネル 2 からチャネル 0 に調歩同期モードでデータを送信します。RSK ボード上でチャネル 2 の送信端子(TXD2-B)とチャネル 0 の受信端子(RXD0)を図の様に接続してください。TXD2-B は RSK ボードの J4/No.22、RXD0 は J2/No.9 です。



使用する RSK ボード上に TXD2-B、RXD0 の有効/無効を切り替えるスイッチがある場合は有効にしてください。

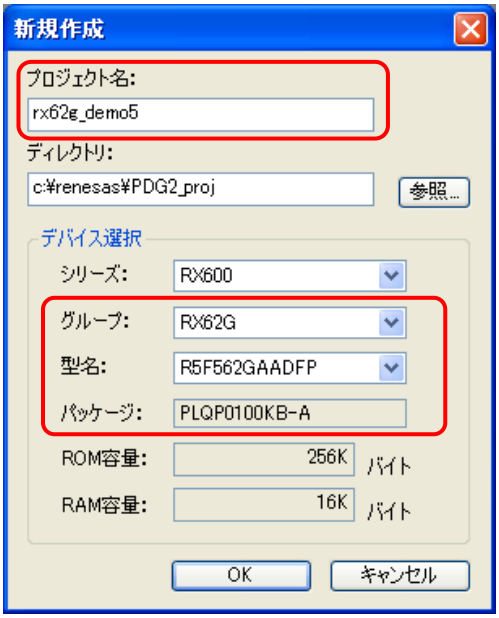
PDG

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

プロジェクト名に“rx62g_demo5”を指定し、Peripheral Driver Generator の新規プロジェクトを作成してください。(プロジェクト作成方法の詳細については「4.1(1) Peripheral Driver Generator プロジェクトの作成」を参照してください。)

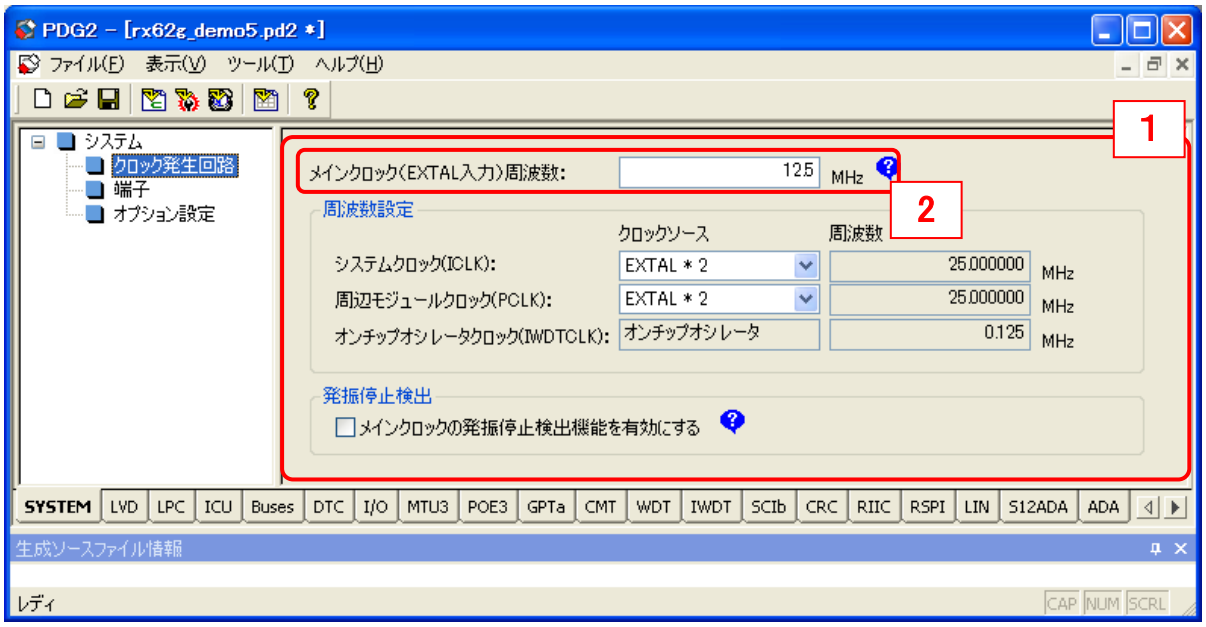
CPU 種別は以下の通り設定してください。但し使用する RSK ボードに他の型名のチップが搭載されている場合は、ボードに合わせて設定してください。

シリーズ ：RX600
グループ ：RX62G
型名 ：R5F562GAADFP



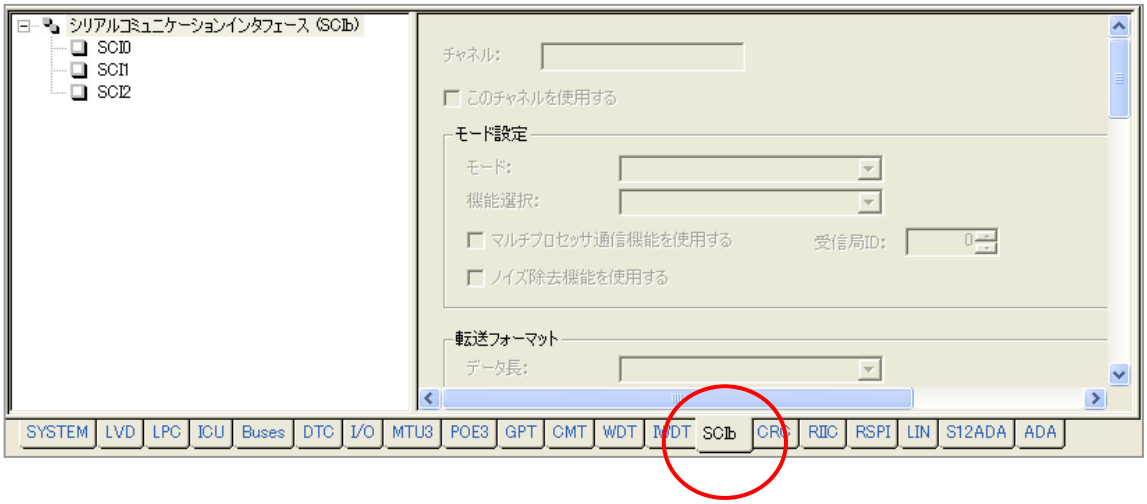
(2) クロックの設定 **PDG**

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



(3) SCIb の設定 **PDG**

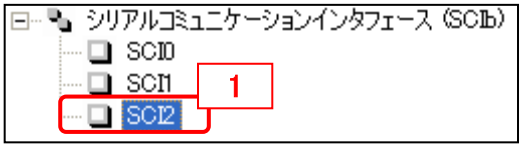
SCIb タブを選択し、SCIb の設定ウィンドウを開いてください。



(4) SCI2(送信側)の設定 PDG

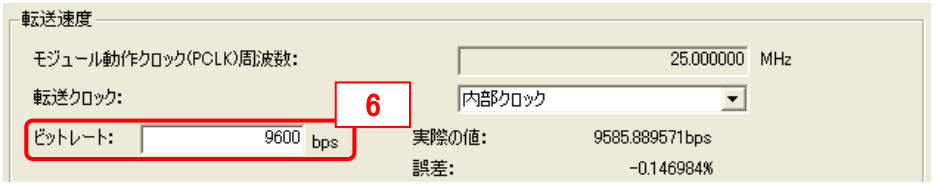
SCI2 を以下の通り設定してください。

1. ツリー表示上で SCI2 を選択してください。

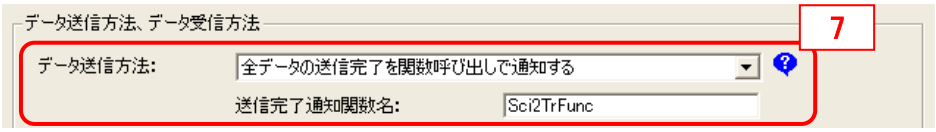


2. [このチャネルを使用する]をチェックしてください。
3. モードに[調歩同期式モード]を選択してください。
4. 機能選択に[送信]を指定してください。
5. 転送フォーマットは初期設定のままとしてください。

6. 転送速度設定のビットレートに 9600bps を設定してください。



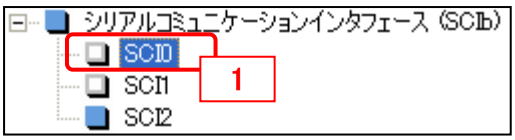
7. データ送信方法に[全データの送信完了を関数呼び出しで通知する]を指定し、送信完了通知関数名を初期設定の”Sci2TrFunc”としてください。



(5) SCI0(受信側)の設定 PDG

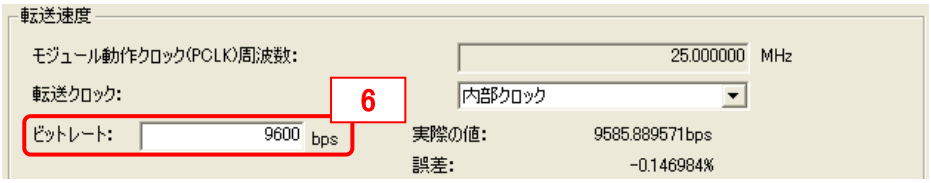
SCI0 を以下の通り設定してください。

1. ツリー表示上で SCI0 を選択してください。

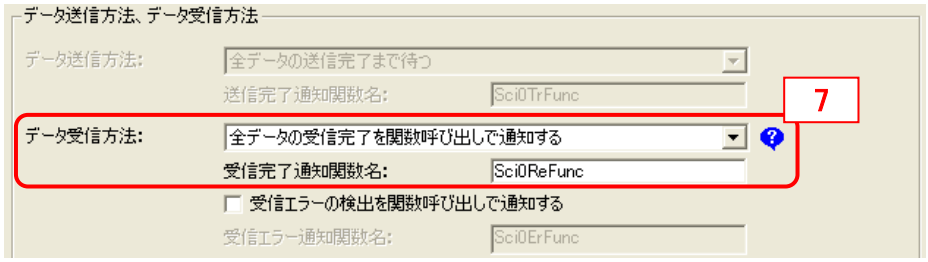


2. [このチャネルを使用する]をチェックしてください。
3. モードに[調歩同期式モード]を選択してください。
4. 機能選択に[受信]を指定してください。
5. 転送フォーマットは初期設定のままとしてください。

6. 転送速度設定のビットレートに 9600bps を設定してください。



7. データ受信方法に[全データの受信完了を関数呼び出しで通知する]を指定し、受信完了通知関数名を初期設定の”Sci0ReFunc”としてください。



(6) 使用する端子の設定 PDG

TXD2 は TXD2-A (PB5) と TXD2-B (P81) から選択することができます。以下の方法で使用する端子を選択してください。

1. SYSTEM タブを選択してください。
2. ツリー表示上で[端子]を選択してください。
3. [周辺機能別使用端子]タブを選択してください。
4. 周辺機能の一覧から SCI2 を選択してください。
5. TXD2 の行で[使用端子]カラムにマウスポインタを置くと、端子選択のドロップダウンボタンが表示されます。ドロップダウンリストから[P81/MTIC5V/TXD2]を選択してください。

(7) ソースファイルの生成 PDG

ツールバー上の  をクリックしてソースファイルを生成してください。 ソースファイル生成の詳細については「4.1(8) ソースファイルの生成」を参照してください。

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

High-performance Embedded Workshop を起動し、RX62G 用のワークスペースを作成してください。作成方法については「4.1(9) High-performance Embedded Workshop プロジェクトの準備」を参照してください。

PDG

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

ツールバー上の  をクリックして Peripheral Driver Generator が生成したソースファイルを High-performance Embedded Workshop のプロジェクトに登録してください。ソースファイル生成の詳細については「4.1(10) Peripheral Driver Generator 生成ファイルの High-performance Embedded Workshop への登録」を参照してください。

(10) プログラムの作成

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

```
//Include "R_PG_<プロジェクト名>.h"
#include "R_PG_rx62g_demo5.h"
//SCI2送信データ
uint8_t tr_data[10] = "ABCDEFGHIJ";
//SCI0受信データ
uint8_t re_data[10] = "----------";
void main(void)
{
    //クロックの設定
    R_PG_Clock_Set();
    //SCI2の設定
    R_PG_SCI_Set_C2();
    //SCI0の設定
    R_PG_SCI_Set_C0();
    //SCI0受信開始 (受信データ数:10)
    R_PG_SCI_StartReceiving_C0( re_data, 10 );
    //SCI2送信開始 (送信データ数:10)
    R_PG_SCI_StartSending_C2( tr_data, 10 );
    while(1);
}
//SCI2送信完了通知関数
void Sci2TrFunc(void)
{
    //SCI2通信終了
    R_PG_SCI_StopCommunication_C2();
}
//SCI0受信完了通知関数
void Sci0ReFunc(void)
{
    //SCI0通信終了
    R_PG_SCI_StopCommunication_C0();
}
```

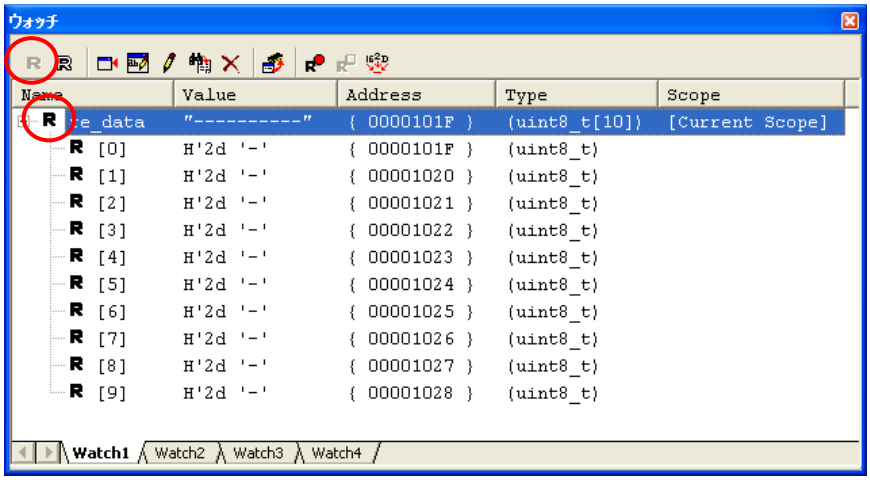
(11) エミュレータの接続、プログラムのビルド、ダウンロード **HEW**

作成したプログラムをビルドし、ダウンロードしてください。

エミュレータの接続、プログラムのビルド方法については「4.1(12) エミュレータの接続、プログラムのビルド、実行」 を参照してください。

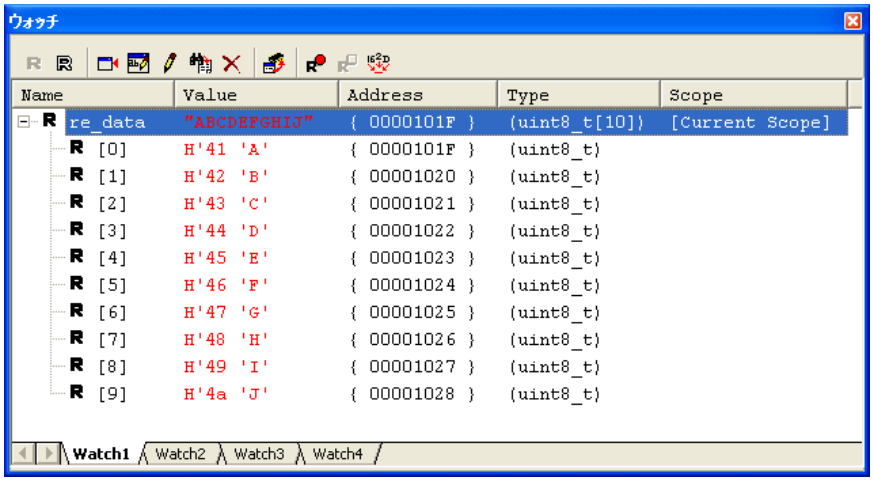
(12) 受信データ格納変数のウォッチウィンドウ登録 **HEW**

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



(13) プログラムの実行と転送結果の確認 **HEW**

プログラムを実行し、変数の値を確認してください。



5. 生成関数仕様

RX62G の生成関数を表 5.1 に示します。

表 5.1 RX62G の生成関数

クロック発生回路

生成関数	機能
R_PG_Clock_Set	クロックの設定
R_PG_Clock_GetMainClockStatus	メインクロックの状態を取得

電圧検出回路(LVD)

生成関数	機能
R_PG_LVD_Set	電圧検出回路の設定
R_PG_LVD_GetLVDDetectionFlag	LVD検知フラグの取得

消費電力低減機能

生成関数	機能
R_PG_LPC_Set	消費電力低減機能の設定
R_PG_LPC_Sleep	スリープモードへ移行
R_PG_LPC_AllModuleClockStop	全モジュールクロックスタンバイモードへ移行
R_PG_LPC_SoftwareStandby	ソフトウェアスタンバイモードへ移行
R_PG_LPC_DeepSoftwareStandby	ディープソフトウェアスタンバイモードへ移行
R_PG_LPC_IOPortRelease	I/Oポート出力保持を解除
R_PG_LPC_GetPowerOnResetFlag	パワーオンリセットフラグの取得
R_PG_LPC_GetLVDDetectionFlag	LVD検出の取得
R_PG_LPC_GetDeepSoftwareStandbyResetFlag	ディープソフトウェアスタンバイリセットフラグの取得
R_PD_LPC_GetDeepSoftwareStandbyCancelFlag	ディープソフトウェアスタンバイ解除要求フラグの取得
R_PG_LPC_GetStatus	消費電力低減機能の状態を取得
R_PG_LPC_WriteBackup	バックアップレジスタへの書き込み
R_PG_LPC_ReadBackup	バックアップレジスタからの読み出し

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_SoftwareInterrupt_Set	ソフトウェア割り込みの設定
R_PG_SoftwareInterrupt_Generate	ソフトウェア割り込みの生成
R_PG_FastInterrupt_Set	高速割り込みの設定
R_PG_Exception_Set	例外ハンドラの設定

バス

生成関数	機能
R_PG_ExtBus_SetBus	バスエラー監視の設定
R_PG_ExtBus_GetErrorStatus	バスエラー検出状態の取得
R_PG_ExtBus_ClearErrorFlags	バスエラーステータスレジスタのクリア

データトランスファコントローラ (DTC)

生成関数	機能
R_PG_DTC_Set	DTCの設定
R_PG_DTC_Set_<転送開始トリガ>	DTC転送情報の設定
R_PG_DTC_Activate	DTCをトリガ入力待ち状態にする
R_PG_DTC_SuspendTransfer	DTC転送の停止
R_PG_DTC_GetTransmitStatus	DTC転送状態の取得
R_PG_DTC_StopModule	DTCの停止

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

マルチファンクションタイマパルスユニット3 (MTU3)

生成関数	機能
R_PG_Timer_Set_MTU_U<*ユニット番号*>_<*チャネル*>	MTUの設定
R_PG_Timer_StartCount_MTU_U<*ユニット番号*>_C<*チャネル番号*>(<*相*>)	カウント動作の開始
R_PG_Timer_SynchronouslyStartCount_MTU_U<*ユニット番号*>	複数チャネルのカウント動作を同時に開始
R_PG_Timer_HaltCount_MTU_U<*ユニット番号*>_C<*チャネル番号*>(<*相*>)	カウント動作の停止
R_PG_Timer_GetCounterValue_MTU_U<*ユニット番号*>_C<*チャネル番号*>	カウンタ値の取得
R_PG_Timer_SetCounterValue_MTU_U<*ユニット番号*>_C<*チャネル番号*>(<*相*>)	カウンタ値の設定
R_PG_Timer_GetRequestFlag_MTU_U<*ユニット番号*>_C<*チャネル番号*>	割り込み要求フラグの取得とクリア
R_PG_Timer_StopModule_MTU_U<*ユニット番号*>	MTUのユニットを停止
R_PG_Timer_GetTGR_MTU_U<*ユニット番号*>_C<*チャネル番号*>	ジェネラルレジスタ値の取得
R_PG_Timer_SetTGR_<*レジスタ*>_MTU_U<*ユニット番号*>_C<*チャネル番号*>	ジェネラルレジスタ値の設定
R_PG_Timer_SetBuffer_AD_MTU_U<*ユニット番号*>_C<*チャネル番号*>	A/D変換要求周期設定バッファレジスタの設定
R_PG_Timer_SetBuffer_CycleData_MTU_U<*ユニット番号*>_<*チャネル*>	周期バッファレジスタ値の設定
R_PG_Timer_SetOutputPhaseSwitch_MTU_U<*ユニット番号*>_<*チャネル*>	PWM出力レベルの切り替え
R_PG_Timer_ControlOutputPin_MTU_U<*ユニット番号*>_<*チャネル*>	PWM出力の有効化/無効化
R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U<*ユニット番号*>_<*チャネル*>	PWM出力レベルをバッファレジスタに設定
R_PG_Timer_ControlBufferTransfer_MTU_U<*ユニット番号*>_<*チャネル*>	バッファレジスタからテンポラリレジスタへのバッファ転送の有効化、無効化

ポートアウトプットイネーブル3 (POE3)

生成関数	機能
R_PG_POE_Set	POEの設定
R_PG_POE_SetHiZ_<*タイマチャネル*>	MTU端子をハイインピーダンスに設定

R_PG_POE_GetRequestFlagHiZ_*<タイマチャネル>*	ハイインピーダンス要求フラグの取得
R_PG_POE_GetShortFlag_*<タイマチャネル>*	MTU端子の出力短絡フラグの取得
R_PG_POE_ClearFlag_*<タイマチャネル>*	ハイインピーダンス要求/出力短絡フラグのクリア

汎用PWM タイマ（GPT）

生成関数	機能
R_PG_Timer_Set_GPT_U*<ユニット番号>*	GPTの設定
R_PG_Timer_Set_GPT_U*<ユニット番号>*_C*<チャネル番号>*	GPTチャネルの設定
R_PG_Timer_StartCount_GPT_U*<ユニット番号>*_C*<チャネル番号>*	GPTのカウント動作開始
R_PG_Timer_SynchronouslyStartCount_GPT_U*<ユニット番号>*	GPTの複数チャネルのカウント動作を同時に開始
R_PG_Timer_HaltCount_GPT_U*<ユニット番号>*_C*<チャネル番号>*	GPTのカウント動作を停止
R_PG_Timer_SynchronouslyHaltCount_GPT_U*<ユニット番号>*	GPTの複数チャネルのカウント動作を同時に停止
R_PG_Timer_SetGTCCR_*<GTCCR>*_GPT_U*<ユニット番号>*_C*<チャネル番号>*	コンペアキャプチャレジスタ（ GTCCRn n:A～F ）値の設定
R_PG_Timer_GetGTCCR_GPT_U*<ユニット番号>*_C*<チャネル番号>*	コンペアキャプチャレジスタ(GTCCRA～F)値の取得
R_PG_Timer_SetCounterValue_GPT_U*<ユニット番号>*_C*<チャネル番号>*	GPTのカウンタ値を設定
R_PG_Timer_GetCounterValue_GPT_U*<ユニット番号>*_C*<チャネル番号>*	GPTのカウンタ値の取得
R_PG_Timer_SynchronouslyClearCounter_GPT_U*<ユニット番号>*	複数チャネルのカウンタを同時にクリア
R_PG_Timer_SetCycle_GPT_U*<ユニット番号>*_C*<チャネル番号>*	タイマ周期設定レジスタ(GTPR)値の設定
R_PG_Timer_SetBuffer_Cycle_GPT_U*<ユニット番号>*_C*<チャネル番号>*	タイマ周期設定バッファレジスタ(GTPBR)値の設定
R_PG_Timer_SetDoubleBuffer_Cycle_GPT_U*<ユニット番号>*_C*<チャネル番号>*	タイマ周期設定ダブルバッファレジスタ(GTPDBR)値の設定
R_PG_Timer_SetAD_GPT_U*<ユニット番号>*_C*<チャネル番号>*	A/D変換開始要求タイミングレジスタA,B (GTADTRA, GTADTRB) 値の設定
R_PG_Timer_SetBuffer_AD_GPT_U*<ユニット番号>*_C*<チャネル番号>*	A/D変換開始要求タイミングバッファレジスタA,B (GTADTBRA, GTADTBRB) 値の設定
R_PG_Timer_SetDoubleBuffer_AD_GPT_U*<ユニット番号>*_C*<チャネル番号>*	A/D変換開始要求タイミングダブルバッファレジスタA,B (GTADTDBRA, GTADTDBRB) 値の設定
R_PG_Timer_SetBuffer_GTDVU_GPT_*<U/V>*<ユニット番号>_C*<チャネル番号>*	タイマデッドタイムバッファレジスタU,D (GTDBU, GTDBD) 値の設定
R_PG_Timer_GetRequestFlag_GPT_U*<ユニット番号>*_C*<チャネル番号>*	割り込み要求フラグの取得とクリア
R_PG_Timer_GetRequestFlag_GPT_U*<ユニット番号>*	LOCOカウント機能および外部トリガの割り込み要求フラグの取得とクリア
R_PG_Timer_GetCounterStatus_GPT_U*<ユニット番号>*_C*<チャネル番号>*	カウンタの状態の取得
R_PG_Timer_BufferEnable_GPT_U*<ユニット番号>*_C*<チャネル番号>*	バッファ動作の有効化
R_PG_Timer_BufferDisable_GPT_U*<ユニット番号>*_C*<チャネル番号>*	バッファ動作の無効化
R_PG_Timer_Buffer_Force_GPT_U*<ユニット番号>*_C*<チャネル番号>*	バッファ強制転送の実行
R_PG_Timer_CountDirection_Down_GPT_U*<ユニット番号>*_C*<チャネル番号>*	カウント方向のダウンカウントへの切り替え
R_PG_Timer_CountDirection_Up_GPT_U*<ユニット番号>*_C*<チャネル番号>*	カウント方向のアップカウントへの切り替え
R_PG_Timer_SoftwareNegate_GPT_U*<ユニット番号>*_C*<チャネル番号>*	GTIOCnAおよびGTIOCnB端子出力のソフトウェアネゲート制御
R_PG_Timer_StartCount_LOCO_GPT_U*<ユニット番号>*	LOCOのカウントを開始
R_PG_Timer_HaltCount_LOCO_GPT_U*<ユニット番号>*	LOCOのカウントを停止
R_PG_Timer_ClearCounter_LOCO_GPT_U*<ユニット番号>*	LOCOカウント値レジスタのクリア
R_PG_Timer_InitialiseCountResultValue_LOCO_GPT_U*<ユニット番号>*	LOCOカウント結果レジスタの初期化
R_PG_Timer_GetCounterValue_LOCO_GPT_U*<ユニット番号>*	LOCOカウント値レジスタの取得
R_PG_Timer_GetCounterAverageValue_LOCO_GPT_U*<ユニット番号>*	LOCOのカウント結果の平均値を取得

R_PG_Timer_GetCountResultValue_LOCO_GPT_U〈*ユニット番号*〉	LOCOのカウント結果の取得
R_PG_Timer_SetPermissibleDeviation_LOCO_GPT_U〈*ユニット番号*〉	LOCOのカウント上限／下限許容偏差値の設定
R_PG_Timer_StopModule_GPT_U〈*ユニット番号*〉	GPT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_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTを設定しカウント動作を開始
R_PG_Timer_HaltCount_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTのカウント動作を一時停止
R_PG_Timer_ResumeCount_CMT_U〈*ユニット番号*〉_C〈*チャネル番号*〉	CMTのカウント動作を再開
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

ウォッチドッグタイマ（WDT）

生成関数	機能
R_PG_Timer_Start_WDT	WDTを設定しカウント動作を開始
R_PG_Timer_HaltCount_WDT	カウント動作の停止
R_PG_Timer_ResetCounter_WDT	カウンタのリセット
R_PG_Timer_ClearOverflowFlag_WDT	オーバフローフラグの取得とクリア

独立ウォッチドッグタイマ（IWDT）

生成関数	機能
R_PG_Timer_Set_IWDT	IWDTの設定
R_PG_Timer_RefreshCounter_IWDT	カウンタのリフレッシュ
R_PG_Timer_GetCounterValue_IWDT	カウンタ値の取得
R_PG_Timer_ClearUnderflowFlag_IWDT	アンダフローフラグの取得とクリア

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_SendTargetStationID_C〈*チャネル番号*〉	データ送信先IDの送信
R_PG_SCI_ReceiveStationID_C〈*チャネル番号*〉	自局IDと一致するIDコードの受信
R_PG_SCI_StopModule_C〈*チャネル番号*〉	シリアルI/Oチャネルの停止
R_PG_SCI_ControlClockOutput_C〈*チャネル番号*〉	SCKn端子出力切り替え

CRC演算器（CRC）

生成関数	機能
R_PG_CRC_Set	CRC演算器の設定
R_PG_CRC_InputData	CRC演算器にデータを入力
R_PG_CRC_GetResult	演算結果の取得

R_PG_CRC_StopModule	CRC演算器の停止

I2Cバスインタフェース（RIIC）

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

シリアルペリフェラルインタフェース（RSPI）

生成関数	機能
R_PG_RSPI_Set_C〈*チャネル番号*〉	RSPIチャネルの設定
R_PG_RSPI_SetCommand_C〈*チャネル番号*〉	コマンドの設定
R_PG_RSPI_StartTransfer_C〈*チャネル番号*〉	データの転送開始
R_PG_RSPI_TransferAllData_C〈*チャネル番号*〉	全データの転送
R_PG_RSPI_GetStatus_C〈*チャネル番号*〉	転送状態の取得
R_PG_RSPI_GetError_C〈*チャネル番号*〉	エラー検出状態の取得
R_PG_RSPI_GetCommandStatus_C〈*チャネル番号*〉	コマンドステータスの取得
R_PG_RSPI_StopModule_C〈*チャネル番号*〉	RSPIチャネルの停止
R_PG_RSPI_LoopBack〈*ループバックモード*〉_C〈*チャネル番号*〉	ループバックモードの設定

LINモジュール（LIN）

生成関数	機能
R_PG_LIN_Set_LIN0	LINモジュールの設定
R_PG_LIN_Transmit_LIN0	データの送信
R_PG_LIN_Receive_LIN0	データの受信
R_PG_LIN_ReadData_LIN0	データの読み出し
R_PG_LIN_EnterResetMode_LIN0	LINリセットモードへ移行
R_PG_LIN_EnterOperationMode_LIN0	LIN動作モードへの移行
R_PG_LIN_EnterWakeUpMode_LIN0	LINウェイクアップモードへの移行
R_PG_LIN_WakeUpTransmit_LIN0	ウェイクアップ信号の送信
R_PG_LIN_WakeUpReceive_LIN0	ウェイクアップ信号の受信
R_PG_LIN_GetCheckSum_LIN0	チェックサム値の取得
R_PG_LIN_EnterSelfTestMode_LIN0	LINセルフテストモードへの移行

R_PG_LIN_WriteCheckSum_LIN0	チェックサム値の書き込み
R_PG_LIN_GetMode_LIN0	モードの取得
R_PG_LIN_GetStatus_LIN0	LINモジュールの状態の取得
R_PG_LIN_GetErrorStatus_LIN0	エラー検出状態の取得
R_PG_LIN_StopModule_LIN0	LINモジュールの停止

12ビットA/Dコンバータ（S12AD）

生成関数	機能
R_PG_ADC_12_Set_S12ADA〈*ユニット番号*〉	12ビットA/Dコンバータの設定
R_PG_ADC_12_Set	ゲインアンプの設定
R_PG_ADC_12_StartConversionSW_S12ADA〈*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_12_StopConversion_S12ADA〈*ユニット番号*〉	A/D変換の中断
R_PG_ADC_12_GetResult_S12ADA〈*ユニット番号*〉	A/D変換結果の取得
R_PG_ADC_12_GetResult_SelfDiag_S12ADA〈*ユニット番号*〉	A/D変換結果の取得（自己診断）
R_PG_ADC_12_StopModule_S12ADA〈*ユニット番号*〉	12ビットA/Dコンバータの停止

10ビットA/Dコンバータ

生成関数	機能
R_PG_ADC_10_Set_AD〈*ユニット番号*〉	10ビットA/Dコンバータの設定
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_SetSelfDiag_VREF_〈*電圧値*〉_AD〈*ユニット番号*〉	A/D自己診断機能の設定
R_PG_ADC_10_StopModule_AD〈*ユニット番号*〉	10ビットA/Dコンバータの停止

5.1 クロック発生回路

5.1.1 R_PG_Clock_Set

定義　bool R_PG_Clock_Set(void)

概要　クロックの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Clock.c

使用RPDL関数　R_CGC_Set

詳細

- 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)および周辺モジュールクロック(PCLK)の逓倍率を設定します。
- メインクロック発信停止検出機能を設定します。

使用例

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

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

5.1.2 R_PG_Clock_GetMainClockStatus

定義 bool R_PG_Clock_GetMainClockStatus (bool * stop)

概要 メインクロック発振停止検出フラグの取得

生成条件 メインクロック発振停止検出機能が有効

引数

bool * stop	メインクロック発振停止検出フラグの格納先

戻り値

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

出力先ファイル R_PG_Clock.c

使用RPDL関数 R_CGC_GetStatus

詳細

- メインクロック発振停止検出フラグを取得します
- メインクロック発振停止検出時にNMIを発生させるには、GUI上のNMI設定で発振停止検出割り込みを有効にしてください。NMIはR_PG_ExtInterrupt_Set_NMIにより設定できます。

使用例

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

bool main_stop;

void func(void)
{
    // メインクロック発振停止検出フラグの取得
    R_PG_Clock_GetMainClockStatus(&main_stop);
}
```

5.2 電圧検出回路（LVD）

5.2.1 R_PG_LVD_Set

定義　bool R_PG_LVD_Set (void)

概要　電圧検出回路の設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_LVD.c

使用RPDL関数　R_LVD_Control

詳細

- 低電圧検出時の動作(内部リセットまたは割り込み)を設定します。
- 1回の呼び出しでLVD1とLVD2を設定することができます。
- 低電圧検出時の動作に割り込みを選択した場合はNMIを設定する必要があります。低電圧検出時にNMIを発生させるには、GUI上のNMI設定で電源電圧降下検出割り込みを有効にしてください。NMIはR_PG_ExtInterrupt_Set_NMIにより設定できます。
- 低電圧検出フラグ(LVD1およびLVD2)はR_PG_LVD_GetLVDDetectionFlagにより取得できます。

使用例

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

void func(void)
{
    //電圧検出回路の設定
    R_PG_LVD_Set (void);
}
```

5.2.2 R_PG_LVD_GetLVDDetectionFlag

定義 bool R_PG_LVD_GetLVDDetectionFlag (bool * lvd1, bool * lvd2)

概要 LVD検出フラグの取得

引数

bool * lvd1	LVD1検出フラグの格納先
bool * lvd2	LVD2検出フラグの格納先

戻り値

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

出力先ファイル R_PG_LVD.c

使用RPDL関数 R_LPC_GetStatus

詳細

- LVD検出取得します。
- 取得しないフラグには0を指定してください。

使用例

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

bool lvd1;
bool lvd2;

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

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

5.3 消費電力低減機能

5.3.1 R_PG_LPC_Set

定義 bool R_PG_LPC_Set (void)

概要 消費電力低減機能の設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Create

詳細

- 消費電力低減機能を設定します。

使用例

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

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

5.3.2 R_PG_LPC_Sleep

定義 bool R_PG_LPC_Sleep (void)

概要 スリープモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- スリープモードへ移行します

使用例

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

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

5.3.3 R_PG_LPC_AllModuleClockStop

定義 bool R_PG_LPC_AllModuleClockStop (void)

概要 全モジュールクロックスタンバイモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

- 全モジュールクロックスタンバイモードへ移行します。

使用例

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

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

5.3.4 R_PG_LPC_SoftwareStandby

定義 bool R_PG_LPC_SoftwareStandby(void)

概要 ソフトウェアスタンバイモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

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

使用例

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

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

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

5.3.5 R_PG_LPC_DeepSoftwareStandby

定義 bool R_PG_LPC_DeepSoftwareStandby(void)

概要 ディープソフトウェアスタンバイモードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

詳細

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

使用例

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

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

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

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

5.3.6 R_PG_LPC_IOPortRelease

定義 bool R_PG_LPC_IOPortRelease (void)

概要 I/Oポート出力保持を解除

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

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_Control

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

使用例

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

5.3.7 R_PG_LPC_GetPowerOnResetFlag

定義 bool R_PG_LPC_GetPowerOnResetFlag (bool * reset)

概要 パワーオンリセットフラグの取得

引数

bool * reset	パワーオンリセットフラグの格納先

戻り値

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

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- パワーオンリセットフラグを取得します
- 本関数を呼び出すとRSTSR.LVD1F(LVD1検知フラグ)、RSTSR.LVD2F(LVD2検知フラグ)、RSTSR.DPSRSTF(ディープソフトウェアスタンバイリセットフラグ)およびDPSIFR(ディープソフトウェアスタンバイ解除要求フラグ)がクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。
- RSTSR.PORF(パワーオンリセットフラグ)は端子リセットでのみクリアされます。

使用例

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

bool reset;

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

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

5.3.8 R_PG_LPC_GetLVDDetectionFlag

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

概要 LVD検知フラグの取得

引数

bool * lvd1	LVD1検出フラグの格納先
bool * lvd2	LVD2検出フラグの格納先

戻り値

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

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- LVD検出フラグを取得します。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
- 取得しないフラグには0を指定してください。
- 本関数を呼び出すとRSTSR.LVD1F(LVD1検出フラグ)、RSTSR.LVD2F(LVD2検出フラグ)、RSTSR.DPSRSTF(ディープソフトウェアスタンバイリセットフラグ)およびDPSIFR(ディープソフトウェアスタンバイ解除要求フラグ)がクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

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

bool lvd1;
bool lvd2;

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

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

5.3.9 R_PG_LPC_GetDeepSoftwareStandbyResetFlag

定義 bool R_PG_LPC_GetDeepSoftwareStandbyResetFlag(bool *reset)

概要 ディープソフトウェアスタンバイリセットフラグの取得

引数

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

戻り値

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

出力先ファイル R_PG_LPC.c

使用RPDL関数 R_LPC_GetStatus

詳細

- ディープソフトウェアスタンバイリセットフラグを取得します
- 本関数を呼び出すとRSTSR.LVD1F(LVD1検知フラグ)、RSTSR.LVD2F(LVD2検知フラグ)、RSTSR.DPSRSTF(ディープソフトウェアスタンバイリセットフラグ)およびDPSIFR(ディープソフトウェアスタンバイ解除要求フラグ)がクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

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

bool reset;

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

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

5.3.10 R_PD_LPC_GetDeepSoftwareStandbyCancelFlag

定義

bool R_PD_LPC_GetDeepSoftwareStandbyCancelFlag
(bool *irq0, bool *irq1, bool *lvd, bool *nmi)

概要

ディープソフトウェアスタンバイ解除要求フラグの取得

引数

bool *irq0	IRQ0ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *irq1	IRQ1ディープソフトウェアスタンバイ解除要求フラグの格納先
bool *lvd	LVDディープソフトウェアスタンバイ解除要求フラグの格納先
bool *nmi	NMIディープソフトウェアスタンバイ解除要求フラグの格納先

戻り値

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

出力先ファイル

R_PG_LPC.c

使用RPDL関数

R_LPC_GetStatus

詳細

- ディープソフトウェアスタンバイ解除要求フラグを取得します。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
- 取得しないフラグには0を指定してください。
- 本関数を呼び出すとRSTSR.LVD1F(LVD1検出フラグ)、RSTSR.LVD2F(LVD2検出フラグ)、RSTSR.DPSRSTF(ディープソフトウェアスタンバイリセットフラグ)およびDPSIFR(ディープソフトウェアスタンバイ解除要求フラグ)がクリアされます。これらのフラグを同時に取得する必要がある場合は本関数の代わりにR_PG_LPC_GetStatusを使用してください。

使用例

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

      if( irq0 ){
         // IRQ0-Aからのディープソフトウェアスタンバイ解除要求検出時処理
      }
      if( nmi ){
         //NMIからのディープソフトウェアスタンバイ解除要求検出時処理
      }
}
```

5.3.11 R_PG_LPC_GetStatus

定義 bool R_PG_LPC_GetStatus(uint16_t *data)

概要 消費電力低減機能の状態を取得

引数

uint16_t *data	ステータス情報の格納先

戻り値

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

出力先ファイル R_PG_LPC.h

使用RPDL関数 R_LPC_GetStatus

詳細

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

b15	b14-b11	b10	b9	b8
リセットステータス(RSTSR) (0: 未検出; 1:検出)				
ディープソフトウェアリセット	0	LVD2	LVD1	パワーオンリセット

b7	b6	b5	b4	b3	b2	b1	b0
ディープソフトウェアスタンバイ解除要求検出(DPSIFR) (0: 未検出; 1:検出)							
NMI	0	0	LVD	0	0	IRQ1-A	IRQ0-A

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

使用例

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

5.3.12 R_PG_LPC_WriteBackup

定義 bool R_PG_LPC_WriteBackup (uint8_t * data, uint8_t count)

概要 ディープスタンバイバックアップレジスタへの書き込み

引数

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

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LPC.h

使用RPDL関数 R_LPC_WriteBackup

詳細

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

使用例

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

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

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

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

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

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

5.3.13 R_PG_LPC_ReadBackup

定義 bool R_PG_LPC_ReadBackup (uint8_t * data, uint8_t count)

概要 ディープスタンバイバックアップレジスタからの読み出し

引数

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

戻り値

true	読み出しに成功した場合
false	読み出しに失敗した場合

出力先ファイル R_PG_LPC.h

使用RPDL関数 R_LPC_ReadBackup

詳細

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

使用例

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

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

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

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

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

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

5.4 割り込みコントローラ (ICU)

5.4.1 R_PG_ExtInterrupt_Set_*〈割り込み種別〉*

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

概要　外部割り込みの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数　R_INTC_CreateExtInterrupt

詳細

- 外部割り込み(IRQ0～IRQ7、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/C)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「5.21 通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み優先レベルを0に設定した場合、外部割り込み要求信号が入力されてもCPU割り込みは発生しません。割り込み要求フラグは
  R_PG_ExtInterrupt_GetRequestFlag_*〈割り込み種別〉* により取得することができ、
  R_PG_ExtInterrupt_ClearRequestFlag_*〈割り込み種別〉* によりクリアすることができます。

使用例1　割り込み通知関数名にIrq0IntFuncを指定する場合

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

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

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

使用例2 割り込み優先レベルを0に設定した場合

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

void func(void)
{
	bool flag;

	//IRQ0を設定する
	R_PG_ExtInterrupt_Set_IRQ0();

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

	func_irq0();	//IRQ0の処理

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

5.4.2 R_PG_ExtInterrupt_Disable_〈*割り込み種別*〉

定義　bool R_PG_ExtInterrupt_Disable_〈*割り込み種別*〉(void)
〈*割り込み種別*〉 ： IRQ0～IRQ7

概要　外部割り込みの設定解除

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数　R_INTC_ControlExtInterrupt

詳細
- 外部割り込み(IRQ0～IRQ7) を無効にします。
- 外部割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Irq0IntFuncを指定する場合

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

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

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

      func_irq0();       //IRQ0の処理
}
```

5.4.3 R_PG_ExtInterrupt_GetRequestFlag_〈割り込み種別〉

定義　bool R_PG_ExtInterrupt_GetRequestFlag_〈割り込み種別〉(bool * flag)
〈割り込み種別〉 ： IRQ0～IRQ7、またはNMI

概要　外部割り込み要求フラグの取得

引数

bool * flag	割り込み要求フラグの格納先

戻り値

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

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

使用RPDL関数　R_INTC_GetExtInterruptStatus

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

使用例　割り込み優先レベルを0に設定した場合

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

void func(void)
{
    bool flag;

    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();

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

    func_irq0();      //IRQ0の処理

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

5.4.4 R_PG_ExtInterrupt_ClearRequestFlag_〈割り込み種別〉

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

概要　外部割り込み要求フラグのクリア

引数　なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

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

使用RPDL関数　R_INTC_ControlExtInterrupt

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

使用例　割り込み優先レベルを0に設定した場合

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

void func(void)
{
    bool flag;

    //IRQ0を設定する
    R_PG_ExtInterrupt_Set_IRQ0();

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

    func_irq0();      //IRQ0の処理

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

5.4.5 R_PG_SoftwareInterrupt_Set

定義 bool R_PG_SoftwareInterrupt_Set(void)

概要 ソフトウェア割り込みの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_SoftwareInterrupt.c

使用RPDL関数 R_INTC_CreateSoftwareInterrupt

詳細

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

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

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

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

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

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

5.4.6 R_PG_SoftwareInterrupt_Generate

定義 bool R_PG_SoftwareInterrupt_Generate(void)

概要 ソフトウェア割り込みの生成

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_SoftwareInterrupt.c

使用RPDL関数 R_INTC_Write

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

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

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

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

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

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

5.4.7 R_PG_FastInterrupt_Set

定義 bool R_PG_FastInterrupt_Set (void)

概要 高速割り込みの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_FastInterrupt.c

使用RPDL関数 R_INTC_CreateFastInterrupt

詳細

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

使用例 GUI上でIRQ0を高速割り込みに指定した場合

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

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

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

5.4.8 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「5.21 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_<プロジェクト名>.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 (){
    func_fp_excep();    //浮動小数点例外発生時の処理
}
```

5.5 バス

5.5.1 R_PG_ExtBus_SetBus

定義　bool R_PG_ExtBus_SetBus(void)

概要　バス端子とバスエラー監視の設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ExtBus.c

使用RPDL関数　R_BSC_Create

詳細

- バスエラー監視を設定します。
- 本関数内でバスエラー割り込みを設定します。GUI上で[バスエラー割り込みを通知関数呼び出しで通知する]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「5.21 通知関数に関する注意事項」の内容に注意してください。
- バスエラーの検出状態はR_PG_ExtBus_ClearErrorFlagsにより取得することができます。

使用例　バスエラー割り込み通知関数名にBusErrFuncを指定した場合

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

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

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

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

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

5.5.2 R_PG_ExtBus_GetErrorStatus

定義

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

概要

バスエラー検出状態の取得

引数

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

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_ExtBus.c

使用RPDL関数

R_BSC_GetStatus

詳細

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

使用例

R_PG_ExtBus_SetBusの使用例を参照してください

5.5.3 R_PG_ExtBus_ClearErrorFlags

定義 bool R_PG_ExtBus_ClearErrorFlags(void)

概要 バスエラーステータスレジスタのクリア

引数 なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_ExtBus.c

使用RPDL関数 R_BSC_Control

詳細
- バスエラーステータスレジスタ（不正アドレスアクセスフラグ、バスマスタIDコード、アクセス先アドレスの値）をクリアします。
- バスエラー割り込み要求フラグ(IR)は本関数内でクリアされます。

使用例 R_PG_ExtBus_SetBusの使用例を参照してください

5.6 データトランスファコントローラ（DTC）

5.6.1 R_PG_DTC_Set

定義 bool R_PG_DTC_Set (void)

概要 DTCの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Dtc.c

使用RPDL関数 R_DTC_Set

詳細

- 転送情報リードスキップ、アドレスモードおよびDTCベクタテーブルのベースアドレスを設定します。

使用例

GUI上で以下の通り設定した場合

- DTCベクタテーブルのアドレスを15000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定

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

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table [256];

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

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

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

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

5.6.2 R_PG_DTC_Set_〈*転送開始要因*〉

定義

bool R_PG_DTC_Set_〈転送開始要因〉(void)

〈転送開始要因〉：

SWINT, CMT0～3, SPRI0, SPTI0, IRQ0～7, ADI0, S12ADI0～1, CMPI, TGIA0～D7, TCIV4 および 7, TGIU5～W5, GTCIA0～C3, GTCIE0～E3, GTCIV0～V3, LOCOI, RXI0～2, TXI0～2, ICRXI0, ICTXI0

概要

DTC転送情報の設定

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Dtc.c

使用RPDL関数

R_DTC_Create

詳細

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

使用例

GUI上で以下の通り設定した場合

- DTCベクタテーブルのアドレスを15000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定
- 転送開始要因をIRQ1に指定したDTC転送を設定

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

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table [256];

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

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

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

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

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

5.6.3 R_PG_DTC_Activate

定義 bool R_PG_DTC_Activate (void)

概要 DTCを転送開始トリガの入力待ち状態に設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Dtc.c

使用RPDL関数 R_DTC_Control

詳細

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

使用例

GUI上で以下の通り設定した場合

- DTCベクタテーブルのアドレスを15000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定
- 割り込みの発生条件に[指定されたデータ転送終了時、CPU割込みが発生]を指定
- チェイン転送無効
- IRQ0の割り込み通知関数名に Irq0IntFunc を指定

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

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table [256];

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

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

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

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

5.6.4 R_PG_DTC_SuspendTransfer

定義 bool R_PG_DTC_SuspendTransfer (void)

概要 DTC転送の停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_Dtc.c

使用RPDL関数 R_DTC_Control

詳細

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

使用例

GUI上で以下の通り設定した場合

- DTCベクタテーブルのアドレスを15000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定

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

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table [256];

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

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

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

//DTC転送の中断
void func2(void)
{
    R_PG_DTC_SuspendTransfer();
}

//DTC転送の再開
void func3(void)
{
    R_PG_DTC_Activate();
}
```

5.6.5 R_PG_DTC_GetTransmitStatus

定義 bool R_PG_DTC_GetTransmitStatus (uint8_t * vector, bool * active)

概要 DTC転送状態の取得

引数

uint8_t * vector	転送動作中の場合、現在の転送の起動要因のベクタ番号 (*activeが1の場合に有効化値が格納されます)
bool * active	現在の転送状態 (0:転送動作なし 1:転送動作中)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Dtc.c

使用RPDL関数 R_DTC_GetStatus

詳細

- DTCアクティブフラグとDTCアクティブベクタ番号を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。
  取得しない項目には0を指定してください。

使用例

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

uint8_t vector;
bool active;

void func(void)
{
    //DTC転送状態の取得
    R_PG_DTC_GetTransmitStatus ( &vector, &active);

    if(active){
        switch( vector ){
            case 64:
                //ベクタ番号64の割込みによる転送中の処理
                break;
            case 65:
                //ベクタ番号65の割込みによる転送中の処理
                break;
            default:
        }
    }
}
```

5.6.6 R_PG_DTC_StopModule

定義 bool R_PG_DTC_StopModule (void)

概要 DTCの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_Dtc.c

使用RPDL関数 R_DTC_Destroy

詳細
- DTCを停止し、モジュールストップ状態に移行します。
- あらかじめ各周辺機能の関数によりDTCのトリガ要因として使用した割り込みを無効にしてください。

使用例 GUI上で以下の通り設定した場合
- DTCベクタテーブルのアドレスを15000hに設定
- 転送開始要因をIRQ0に指定したDTC転送を設定
- 転送開始要因をIRQ1に指定したDTC転送を設定

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

//DTCベクタテーブル
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table [256];

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

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

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

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

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

//DTCの停止
void func2(void)
{
    //IRQ0,IRQ1の停止
    R_PG_ExtInterrupt_Disable_IRQ0();
    R_PG_ExtInterrupt_Disable_IRQ1();
    //DTCの停止
    R_PG_DTC_StopModule();
}
```

5.7 I/Oポート

5.7.1 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*〉： 1～9, A, B, D, E, G

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*〉： 1～9, A, B, D, E, G

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_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //P1を設定する
    R_PG_IO_PORT_Set_P1();
}
```

5.7.2 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*〉 : 1～9, A, B, D, E, G
〈*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*〉 : 1～9, A, B, D, E, G

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、入力バッファの設定を行います。
- 1端子のみ設定します。

使用例

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

void func(void)
{
    //P10を設定する
    R_PG_IO_PORT_Set_P10();

    //P11を設定する
    R_PG_IO_PORT_Set_P11();
}
```

5.7.3 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*〉： 1～9, A, B, D, E, G

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	読み出しに成功した場合
false	読み出しに失敗した場合

出力先ファイル　R_PG_IO_PORT_P〈*ポート番号*〉.c
〈*ポート番号*〉： 1～9, A, B, D, E, G

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_〈プロジェクト名〉.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;

    //P1端子状態を取得する
    R_PG_IO_PORT_Read_P1( &data );
}
```

5.7.4 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*〉 : 1～9, A, B, D, E, G

〈*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	読み出しに成功した場合
false	読み出しに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

(〈*ポート番号*〉 : 1～9, A, B, D, E, G)

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_〈プロジェクト名〉.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_p10, data_p11;

    //P10端子状態を取得する
    R_PG_IO_PORT_Read_P10( & data_p10);

    //P11端子状態を取得する
    R_PG_IO_PORT_Read_P11( & data_p11);
}
```

5.7.5 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*〉： 1～9, A, B, D, E, G

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	書き込みに成功した場合
false	書き込みに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉： 1～9, A, B, D, E, G

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_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //P1を設定する
    R_PG_IO_PORT_Set_P1();

    //P1から0x03を出力する
    R_PG_IO_PORT_Set_P1( 0x03 );
}
```

5.7.6 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*〉 : 1～9, A, B, D, E, G

〈*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	書き込みに成功した場合
false	書き込みに失敗した場合

出力先ファイル

R_PG_IO_PORT_P〈*ポート番号*〉.c

〈*ポート番号*〉 : 1～9, A, B, D, E, G

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_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    //P10を設定する
    R_PG_IO_PORT_Set_P10();

    //P11を設定する
    R_PG_IO_PORT_Set_P11();

    //P10からLを出力する
    R_PG_IO_PORT_Write_P10( 0x00 );

    //P11からHを出力する
    R_PG_IO_PORT_Write_P11( 0x01 );
}
```

5.8 マルチファンクションタイマパルスユニット 3 (MTU3)

5.8.1 R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉

定義 bool R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉(void)
〈ユニット番号〉: 0
〈チャネル〉: C0～C7,C3_C4,C6_C7

概要 MTUの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数 R_MTU3_Set, R_MTU3_Create

詳細

- MTUのモジュールストップ状態を解除して初期設定します。
- 本関数内でMTUの割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。 void 〈割り込み通知関数名〉(void)
  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。
- GUI上で割り込み優先レベルを0に設定した場合、CPU割り込みは発生しません。割り込み要求フラグは R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉により取得することができます。
- 外部入力カウントクロック、外部リセット信号、インプットキャプチャ、パルス出力を使用する場合、本関数内で使用する端子の入出力方向と入力バッファを設定します。
- カウント動作を開始するには本関数を呼び出した後に R_PG_Timer_StartCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉) または R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉 を呼び出してください。
- 相補PWMモードおよびリセット同期PWMモードでは、ペアで使用する2チャネルを設定します。チャネル3の設定ではチャネル3,4が、チャネル6の設定ではチャネル6,7が設定されます。
- 相補PWMモードおよびリセット同期PWMモードでは、初期状態でPWM出力が無効です。端子出力を有効にするには、カウントを開始する前に R_PG_Timer_ControlOutputPin_MTU_U〈ユニット番号〉_〈チャネル〉 を呼び出してください。

使用例 1

GUI上で以下の通り設定した場合

- MTUチャネル6を通常モードで設定
- コンペアマッチA割り込み通知関数名にMtu6IcCmAIntFuncを指定

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

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C6();       // MTU6の設定
    R_PG_Timer_StartCount_MTU_U0_C6();      //  カウント動作開始
}

void Mtu6IcCmAIntFunc(void)
{
    //コンペアマッチA割り込み発生時処理
}
```

使用例 2

GUI上で以下の通り設定した場合

- MTUチャネル3,4を相補PWMモードで設定

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

void func(void)
{
    //MTU3,4を相補PWMモードで設定
    R_PG_Timer_Set_MTU_U0_C3_C4();

    //PWM出力端子1の正相、逆相出力を有効化
    R_PG_Timer_ControlOutputPin_MTU_U0_C3_C4(
        1, //p1： 有効
        1, //n1： 有効
        0, //p2： 無効
        0, //n2： 無効
        0, //p3： 無効
        0 //n3： 無効
    );

    //MTU3,4のカウント動作開始
    R_PG_Timer_SynchronouslyStartCount_MTU_U0(
        0, //ch0
        0, //ch1
        0, //ch2
        1, //ch3
        1, //ch4
        0, //ch6
        0 //ch7
    );
}
```

5.8.2 R_PG_Timer_StartCount_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*(_*〈相〉*)

定義

bool R_PG_Timer_StartCount_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

bool R_PG_Timer_StartCount_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*_*〈相〉*(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 5
〈相〉: U, V, W

概要

MTUのカウント動作開始

引数

なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数

R_MTU3_ControlChannel

詳細

- MTUのカウント動作を開始します。
- あらかじめR_PG_Timer_Set_MTU_U*〈ユニット番号〉*_*〈チャネル〉* によりMTUを初期設定してください。
- 相補PWMモードおよびリセット同期PWMモードでは、R_PG_Timer_SynchronouslyStartCount_MTU_U*〈ユニット番号〉* によりペアで使用する2チャネルのカウント動作を同時に開始してください。
- R_PG_Timer_StartCount_MTU_U0_C5 はU,V,W相のカウンタを同時に開始させます。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

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

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   //MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   //カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_HaltCount_MTU_U0_C1();   //カウント動作停止

    //コンペアマッチA割り込み発生時処理

    R_PG_Timer_StartCount_MTU_U0_C1();   //カウント動作再開
}
```

5.8.3 R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉

定義

bool R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉
(bool ch0, bool ch1, bool ch2, bool ch3, bool ch4, bool ch6, bool ch7)
〈ユニット番号〉: 0

概要

MTUの複数チャネルのカウント動作を同時に開始

引数

bool ch0	チャネル0のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch1	チャネル1のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch2	チャネル2のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch3	チャネル3のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch4	チャネル4のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch6	チャネル6のカウント動作 (0:カウント開始しない 1:カウント開始)
bool ch7	チャネル7のカウント動作 (0:カウント開始しない 1:カウント開始)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_MTU3_ControlUnit

詳細

- MTUの複数チャネルのカウント動作を同時に開始します。
- あらかじめR_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉 によりMTUを初期設定してください。
- 相補PWMモードおよびリセット同期PWMモードでは、本関数によりペアで使用する2チャネルのカウント動作を同時に開始してください。

使用例

R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉 の使用例2を参照してください。

5.8.4 R_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉)

定義

bool R_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7
bool R_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉(void)
〈ユニット番号〉: 0
〈チャネル番号〉: 5
〈相〉: U, V, W

概要

MTUのカウント動作を一時停止

引数

なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数

R_MTU3_ControlChannel

詳細

- MTUのカウント動作を一時停止します。
- カウント動作を再開するには
  R_PG_Timer_StartCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉) または
  R_PG_Timer_SynchronouslyStartCount_MTU_U〈ユニット番号〉 を呼び出してください。
- R_PG_Timer_HaltCount_MTU_U0_C5 はU,V,W相のカウンタを同時に停止させます。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル1を設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

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

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();      // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();    // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_HaltCount_MTU_U0_C1();     //カウント動作停止

    //コンペアマッチA割り込み発生時処理

    R_PG_Timer_StartCount_MTU_U0_C1();     //カウント動作再開
}
```

5.8.5 R_PG_Timer_GetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * counter_val)
〈ユニット番号〉: 0
〈チャネル番号〉: 0～4, 6, 7

bool R_PG_Timer_GetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * counter_u_val, uint16_t * counter_v_val, uint16_t * counter_w_val)
〈ユニット番号〉: 0
〈チャネル番号〉: 5

概要

MTUのカウンタ値を取得

引数

MTU0～MTU4, MTU6, MTU7

uint16_t * counter_val	カウンタ値の格納先

MTU5

uint16_t * counter_u_val	カウンタU値の格納先
uint16_t * counter_v_val	カウンタV値の格納先
uint16_t * counter_w_val	カウンタW値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数

R_MTU3_ReadChannel

詳細

- MTUのカウンタ値を取得します。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル0を設定
- TGRAをインプットキャプチャレジスタに設定し、インプットキャプチャA割り込みを有効に設定
- インプットキャプチャA割り込み通知関数名にMtu0IcCmAIntFuncを指定

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

uint16_t counter_val;

void func(void)
{
      R_PG_Timer_Set_MTU_U0_C0();       // MTU0の設定
      R_PG_Timer_StartCount_MTU_U0_C0();       //  カウント動作開始
}

void Mtu0IcCmAIntFunc(void)
{
      //MTUのカウンタ値を取得
      R_PG_Timer_GetCounterValue_MTU_U0_C0( & counter_val );
}
```

5.8.6 R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉(_〈相〉)

定義

bool R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t counter_val)
〈ユニット番号〉: 0 〈チャネル番号〉: 0～4, 6, 7

bool R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉
(uint16_t counter_val)
〈ユニット番号〉: 0 〈チャネル番号〉: 5 〈相〉: U, V, W

bool R_PG_Timer_SetCounterValue_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t counter_u_val, uint16_t counter_v_val, uint16_t counter_w_val)
〈ユニット番号〉: 0 〈チャネル番号〉: 5

概要

MTUのカウンタ値を設定

引数

MTU0～MTU7

uint16_t counter_val	カウンタに設定する値

MTU5

uint16_t counter_u_val	カウンタUに設定する値
uint16_t counter_v_val	カウンタVに設定する値
uint16_t counter_w_val	カウンタWに設定する値

戻り値

true	カウンタ値の設定に成功した場合
false	カウンタ値の設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数

R_MTU3_ControlChannel

詳細

- MTUのカウンタ値を設定します。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
  コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

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

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();  // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_SetCounterValue_MTU_U0_C1( 0 );   //カウンタの0クリア
}
```

5.8.7 R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(bool* cm_ic_a, bool* cm_ic_b, bool* cm_ic_c, bool* cm_ic_d,
bool* cm_e, bool* cm_f, bool* ov, bool* un);
〈ユニット番号〉: 0
〈チャネル番号〉: 0～4, 6, 7

bool R_PG_Timer_GetRequestFlag_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(bool* cm_ic_u, bool* cm_ic_v, bool* cm_ic_w);
〈ユニット番号〉: 0
〈チャネル番号〉: 5

概要

MTUの割り込み要求フラグの取得とクリア

引数

bool* cm_ic_a	コンペアマッチ/インプットキャプチャAフラグの格納先
bool* cm_ic_b	コンペアマッチ/インプットキャプチャBフラグの格納先
bool* cm_ic_c	コンペアマッチ/インプットキャプチャCフラグの格納先
bool* cm_ic_d	コンペアマッチ/インプットキャプチャDフラグの格納先
bool* cm_e	コンペアマッチEフラグの格納先
bool* cm_f	コンペアマッチFフラグの格納先
bool* ov	オーバフローフラグの格納先
bool* un	アンダフローフラグの格納先
bool* cm_ic_u	コンペアマッチ/インプットキャプチャUフラグの格納先
bool* cm_ic_v	コンペアマッチ/インプットキャプチャVフラグの格納先
bool* cm_ic_w	コンペアマッチ/インプットキャプチャWフラグの格納先

各チャネルで有効なフラグは以下です。

MTU0	cm_ic_a～cm_ic_d, cm_e, cm_f, ov
MTU1, 2	cm_ic_a, cm_ic_b, ov, un
MTU3, 4, 6, 7	cm_ic_a～cm_ic_d, ov
MTU5	cm_ic_u, cm_ic_v, and cm_ic_w
MTU3, 6 (相補PWMモードおよびリセット同期PWMモード)	cm_ic_a, cm_ic_b
MTU4, 7 (相補PWMモードおよびリセット同期PWMモード)	cm_ic_a, cm_ic_b, un

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数

R_MTU3_ReadChannel

詳細

- MTUの割り込み要求フラグを取得します。
- 本関数内で全フラグがクリアされます。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
- コンペアマッチA割り込みの優先レベルを0に設定

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

bool cma_flag;

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   //  カウント動作開始

    //コンペアマッチAの発生を待つ
    do{
        R_PG_Timer_GetRequestFlag_MTU_U0_C1(
            & cma_flag, //a
            0, //b
            0, //c
            0, //d
            0, //e
            0, //f
            0, //e
            0, //ov
            0 //un
        );
    } while( !cma_flag );

    //コンペアマッチA発生時処理

}
```

5.8.8 R_PG_Timer_StopModule_MTU_U〈ユニット番号〉

定義 bool R_PG_Timer_StopModule_MTU_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 MTUのユニットを停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

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

使用RPDL関数 R_MTU3_Destroy

詳細
- MTUを停止し、モジュールストップ状態に移行します。複数のチャネルが動作している場合、本関数を呼び出すと全チャネルが停止します。1チャネルの動作だけを停止させる場合はR_PG_Timer_HaltCount_MTU_U〈ユニット番号〉_C〈チャネル番号〉_〈相〉を呼び出してください。

使用例 GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
- コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定

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

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    // MTUユニット0の停止
    R_PG_Timer_StopModule_MTU_U0();
}
```

5.8.9 R_PG_Timer_GetTGR_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetTGR_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t* tgr_a_val, uint16_t* tgr_b_val, uint16_t* tgr_c_val,
uint16_t* tgr_d_val, uint16_t* tgr_e_val, uint16_t* tgr_f_val);
〈ユニット番号〉: 0
〈チャネル番号〉: 0～4, 6, 7

bool R_PG_Timer_GetTGR_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * tgr_u_val, uint16_t * tgr_v_val, uint16_t * tgr_w_val);
〈ユニット番号〉: 0
〈チャネル番号〉: 5

概要

ジェネラルレジスタの値の取得

引数

uint16_t* tgr_a_val	ジェネラルレジスタA値の格納先
uint16_t* tgr_b_val	ジェネラルレジスタB値の格納先
uint16_t* tgr_c_val	ジェネラルレジスタC値の格納先
uint16_t* tgr_d_val	ジェネラルレジスタD値の格納先
uint16_t* tgr_e_val	ジェネラルレジスタE値の格納先
uint16_t* tgr_f_val	ジェネラルレジスタF値の格納先
uint16_t* tgr_u_val	ジェネラルレジスタU値の格納先
uint16_t* tgr_v_val	ジェネラルレジスタV値の格納先
uint16_t* tgr_w_val	ジェネラルレジスタW値の格納先

各チャネルで有効な引数は以下です。

MTU0	tgr_a_val ～ tgr_f_val
MTU1, 2	tgr_a_val, tgr_b_val
MTU3, 4, 6, 7	tgr_a_val ～ tgr_d_val
MTU5	tgr_u_val ～ tgr_w_val
MTU3, 6 (相補PWMモード)	tgr_a_val ～ tgr_e_val
MTU4, 7 (相補PWMモード)	tgr_a_val ～ tgr_f_val
MTU3, 4, 6, 7 (リセット同期PWMモード)	tgr_a_val ～ tgr_d_val

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数

R_MTU3_ReadChannel

詳細

- ジェネラルレジスタの値を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。

使用例

GUI上で以下の通り設定した場合

- MTUチャネル0を設定
- TGRAをインプットキャプチャレジスタに設定し、インプットキャプチャA割り込みを有効に設定
- インプットキャプチャA割り込み通知関数名にMtu0IcCmAIntFuncを指定

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

uint16_t tgr_a_val;

void func(void)
{
	R_PG_Timer_Set_MTU_U0_C0();	// MTU0の設定
	R_PG_Timer_StartCount_MTU_U0_C0();	// カウント動作開始
}

void Mtu0IcCmAIntFunc(void)
{
	//TGRAの値を取得
	R_PG_Timer_GetTGR_MTU_U0_C0(
		& tgr_a_val, //a
		0, //b
		0, //c
		0, //d
		0, //e
		0 //f
	);
}
```

5.8.10 R_PG_Timer_SetTGR_〈ジェネラルレジスタ〉_MTU_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_SetTGR_〈ジェネラルレジスタ〉_MTU_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t value);

〈ジェネラルレジスタ〉:

MTU1, 2	: A または B
MTU3, 4, 6, 7	: A, B, C または D
MTU5	: U, V または W
MTU3, 4, 6, 7 (相補PWMモード)	: A, B, C, D, E(*1), または F(*1)
MTU3, 4, 6, 7 (リセット同期PWMモード)	: A, B, C(*2), または D(*3)

(*1 ダブルバッファ有効時のみ)
(*2 TGRCをバッファレジスタとして使用する場合のみ)
(*3 TGRDをバッファレジスタとして使用する場合のみ)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

概要 ジェネラルレジスタの値の設定

引数

uint16_t value	ジェネラルレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～7

使用RPDL関数 R_MTU3_ControlChannel

詳細
- ジェネラルレジスタの値を設定します。

使用例

GUI上で以下の通り設定した場合
- MTUチャネル1を設定
- TGRAをアウトプットコンペアレジスタに設定し、コンペアマッチA割り込みを有効に設定
  コンペアマッチA割り込み通知関数名にMtu1IcCmAIntFuncを指定
-

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

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C1();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C1();   // カウント動作開始
}

void Mtu1IcCmAIntFunc(void)
{
    R_PG_Timer_SetTGR_A_MTU_U0_C1( 1000 );   //TGRAの設定
}
```

5.8.11 R_PG_Timer_SetBuffer_AD_U*〈ユニット番号〉*_C*〈チャネル番号〉*

定義

bool R_PG_Timer_SetBuffer_AD_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*
(uint16_t tadcobr_a_val, uint16_t tadcobr_b_val);
〈ユニット番号〉: 0
〈チャネル番号〉: 4 or 7

概要

A/D変換要求周期設定バッファレジスタの設定

生成条件

A/D変換要求周期レジスタ値のバッファ転送が有効

引数

uint16_t tadcobr_a_val	A/D変換要求周期設定バッファレジスタAに設定する値
uint16_t tadcobr_b_val	A/D変換要求周期設定バッファレジスタBに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U*〈ユニット番号〉*_C*〈チャネル番号〉*.c
〈ユニット番号〉: 0
〈チャネル番号〉: 3(*), 4, 6(*), 7 (*相補PWMモードおよびリセット同期PWMモード)

使用RPDL関数

R_MTU3_ControlChannel

詳細

- A/D変換要求周期設定バッファレジスタAおよびB(TADCOBRA、TADCOBRB)を設定します。

使用例

GUI上で以下の通り設定した場合

- A/D変換要求周期レジスタ値のバッファ転送を有効に設定

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

void func1(void)
{
    R_PG_Timer_Set_MTU_U0_C4();   // MTU1の設定
    R_PG_Timer_StartCount_MTU_U0_C4();   // カウント動作開始
}

void func2(void)
{
    // A/D変換要求周期設定バッファレジスタの設定
    R_PG_Timer_SetBuffer_AD_MTU_U0_C4( 0x10, 0x20 );
}
```

5.8.12 R_PG_Timer_SetBuffer_CycleData_MTU_U〈*ユニット番号*〉_〈*チャネル*〉

定義

bool R_PG_Timer_SetBuffer_CycleData_MTU_U〈*ユニット番号*〉_〈*チャネル*〉
(uint16_t tcbr_val);
〈*ユニット番号*〉: 0
〈*チャネル*〉: C3_C4, C6_C7

概要

周期バッファレジスタ値の設定

生成条件

MTUチャネルを相補PWMモードに設定

引数

uint16_t tcbr_val	周期バッファレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 3, 6

使用RPDL関数

R_MTU3_ControlChannel

詳細

- タイマ周期バッファレジスタ(TCBRA (チャネル3,4) または TCBRB (チャネル6,7))を設定します。

使用例

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

void func(void)
{
      R_PG_Timer_SetBuffer_CycleData_MTU_U0_C3_C4(0x1000);
}
```

5.8.13 R_PG_Timer_SetOutputPhaseSwitch_MTU_U〈ユニット番号〉_〈チャネル〉

定義

bool R_PG_Timer_SetOutputPhaseSwitch_MTU_U〈ユニット番号〉_〈チャネル〉
(uint8_t output_level);
〈ユニット番号〉: 0
〈チャネル〉: C3_C4

概要

PWM出力レベルの切り替え

生成条件

- MTUチャネルを相補PWMモードまたはリセット同期PWMモードに設定
- DCブラシレスモータ制御を有効に設定し、出力制御方法にソフトウェアを指定

引数

uint8_t output_level	出力設定 (0～7)

各値での出力は以下の通りです

値	MTIOC3B U相	MTIOC4A V相	MTIOC4B W相	MTIOC3D U相	MTIOC4C V相	MTIOC4D W相
0	OFF	OFF	OFF	OFF	OFF	OFF
1	ON	OFF	OFF	OFF	OFF	ON
2	OFF	ON	OFF	ON	OFF	OFF
3	OFF	ON	OFF	OFF	OFF	ON
4	OFF	OFF	ON	OFF	ON	OFF
5	ON	OFF	OFF	OFF	ON	OFF
6	OFF	OFF	ON	ON	OFF	OFF
7	OFF	OFF	OFF	OFF	OFF	OFF

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 3

使用RPDL関数

R_MTU3_ControlUnit

詳細

- DBブラシレスモータ制御時のPWM出力レベルを切り替えます

使用例

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

void func(void)
{
    R_PG_Timer_SetOutputPhaseSwitch_MTU_U0_C3_C4 (0x7);
}
```

5.8.14 R_PG_Timer_ControlOutputPin_MTU_U〈ユニット番号〉_〈チャネル〉

定義

bool R_PG_Timer_ControlOutputPin_MTU_U〈ユニット番号〉_〈チャネル〉
(bool p1_enable, bool n1_enable, bool p2_enable, bool n2_enable,
bool p3_enable, bool n3_enable)
〈ユニット番号〉: 0
〈チャネル〉: C3_C4, C6_C7

概要

PWM出力の有効化/無効化

生成条件

MTUチャネルを相補PWMモードまたはリセット同期PWMモードに設定

引数

bool p1_enable	U相 正相（MTIOCmB）出力（0:出力無効 1:出力有効）
bool n1_enable	U相 逆相（MTIOCmD）出力（0:出力無効 1:出力有効）
bool p2_enable	V相 正相（MTIOCnA）出力（0:出力無効 1:出力有効）
bool n2_enable	V相 逆相（MTIOCnC）出力（0:出力無効 1:出力有効）
bool p3_enable	W相 正相（MTIOCnB）出力（0:出力無効 1:出力有効）
bool n3_enable	W相 逆相（MTIOCnD）出力（0:出力無効 1:出力有効）

m : 3, 6 n : 4, 7

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 3, 6

使用RPDL関数

R_MTU3_ControlUnit

詳細

- 相補PWMモード、リセット同期PWMモードの6相のPWM出力を有効化、無効化します。
- 相補PWMモードおよびリセット同期PWMモードでは、初期状態でPWM出力が無効です。端子出力を有効にするには、カウントを開始する前に本関数を呼び出してください。

使用例

R_PG_Timer_Set_MTU_U〈ユニット番号〉_〈チャネル〉の使用例2を参照してください。

5.8.15 R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U〈*ユニット番号*〉_〈*チャネル*〉

定義

bool R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U〈*ユニット番号*〉_〈*チャネル*〉
(bool p1_high, bool n1_high, bool p2_high, bool n2_high,
bool p3_high, bool n3_high)
〈*ユニット番号*〉: 0
〈*チャネル*〉: C3_C4, C6_C7

概要

PWM出力レベルをバッファレジスタに設定

生成条件

MTUチャネルを相補PWMモードまたはリセット同期PWMモードに設定

引数

bool p1_high	U相 正相（MTIOCmB）出力
bool n1_high	U相 逆相（MTIOCmD）出力
bool p2_high	V相 正相（MTIOCnA）出力
bool n2_high	V相 逆相（MTIOCnC）出力
bool p3_high	W相 正相（MTIOCnB）出力
bool n3_high	W相 逆相（MTIOCnD）出力

m : 3, 6 n : 4, 7

各値での出力レベルは以下の通りです。

値	種別	正相	逆相
0	アクティブレベル	Low	Low
	初期出力	Low	Low
	アップカウント時コンペアマッチ	Low	High
	ダウンカウント時コンペアマッチ	High	Low
1	アクティブレベル	High	High
	初期出力	High	High
	アップカウント時コンペアマッチ	High	Low
	ダウンカウント時コンペアマッチ	Low	High

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 3, 6

使用RPDL関数

R_MTU3_ControlUnit

詳細

- PWM出力レベル設定をタイマアウトプットレベルバッファレジスタ（TOLBRA(チャネル3,4), TOLBRB(チャネル6,7)）に設定します。

使用例

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

void func(void)
{
    R_PG_Timer_SetBuffer_PWMOutputLevel_MTU_U0_C3_C4( 0, 0, 0, 0, 0, 0 );
}
```

5.8.16 R_PG_Timer_ControlBufferTransfer_MTU_U〈ユニット番号〉_〈チャネル〉

定義

bool R_PG_Timer_ControlBufferTransfer_MTU_U〈*ユニット番号*〉_〈*チャネル*〉
(bool enable)
〈*ユニット番号*〉: 0
〈*チャネル*〉: C3_C4, C6_C7

概要

バッファレジスタからテンポラリレジスタへのバッファ転送の有効化、無効化

生成条件

- MTUチャネルを相補PWMモードに設定
- 割り込み間引きモードに割り込み間引き機能1を選択

引数

bool enable	バッファ転送設定 (0:有効 1:無効)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_MTU_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 3, 6

使用RPDL関数

R_MTU3_ControlUnit

詳細

- 相補PWMモードで使用するバッファレジスタからテンポラリレジスタへのバッファ転送を有効化、無効化します

使用例

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

void func(void)
{
    R_PG_Timer_ControlBufferTransfer_MTU_U0_C3_C4( 1 );
}
```

5.9 ポートアウトプットイネーブル 3 (POE3)

5.9.1 R_PG_POE_Set

定義 bool R_PG_POE_Set (void)

概要 POEの設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_POE.c

使用RPDL関数 R_POE_Set, R_POE_Create

詳細

- GUI上で選択されたMTU0, 3, 4, 6, 7, GPT0, 1, 2, 3の出力端子の制御と、ハイインピーダンス要求信号に使用する入力端子、アウトプットイネーブル割り込みを設定します。
- MTUおよびGPTの端子出力は、MTUおよびGPTのGUIおよび関数により設定してください。MTUで出力端子に設定していない端子は、POEで設定しないでください。
- GUI上で割り込み通知関数名を指定した場合、CPUへの割り込みが発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void *〈割り込み通知関数名〉*(void)
  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例 GUI上で以下の通り設定した場合

- アウトプットイネーブル割り込み2(OEI2)を有効に設定し、割り込み通知関数名にPoeOei2IntFuncを指定

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

void func(void)
{
    R_PG_POE_Set();     // POEの設定
}

void PoeOei2IntFunc (void)
{
    //アウトプットイネーブル割り込み処理
}
```

5.9.2 R_PG_POE_SetHiZ_*<タイマチャネル>*

定義 bool R_PG_POE_SetHiZ_*<タイマチャネル>*(void)

<タイマチャネル> : MTU0, MTU3_4, MTU6_7, GPT0_1, GPT2_3

概要 タイマ出力端子をハイインピーダンスに設定

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_POE.c

使用RPDL関数 R_POE_Control

詳細

- GUI上でハイインピーダンス制御対象に指定されたMTU0, 3, 4, 6, 7, GPT0, 1, 2, 3の出力端子をハイインピーダンス状態にします。

使用例 GUI上で以下の通り設定した場合

- MTU0の端子出力を設定（MTUの設定GUI上）
- MTU0の出力端子をPOEのハイインピーダンス制御対象に指定

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

void func1(void)
{
    R_PG_Timer_Set_MTU_U0_C0();    //MTU0の設定
    R_PG_POE_Set();      // POEの設定
    R_PG_Timer_StartCount_MTU_U0_C0();      //MTU0のカウント動作開始
}

void func2(void)
{
    R_PG_POE_SetHiZ_MTU0();      //MTU0の出力端子をHiZに設定
}
```

5.9.3 R_PG_POE_GetRequestFlagHiZ_<タイマチャネル>

定義

bool R_PG_POE_GetRequestFlagHiZ_MTU3_4 (bool* poe0)
bool R_PG_POE_GetRequestFlagHiZ_MTU6_7 (bool* poe4)
bool R_PG_POE_GetRequestFlagHiZ_MTU0 (bool* poe8)
bool R_PG_POE_GetRequestFlagHiZ_GPT0_1 (bool* poe10)
bool R_PG_POE_GetRequestFlagHiZ_GPT2_3 (bool* poe11)

概要

ハイインピーダンス要求フラグの取得

引数

bool* poe0	POE0#端子のハイインピーダンス要求フラグの格納先
bool* poe4	POE4#端子のハイインピーダンス要求フラグの格納先
bool* poe8	POE8#端子のハイインピーダンス要求フラグの格納先
bool* poe10	POE10#端子のハイインピーダンス要求フラグの格納先
bool* poe11	POE11#端子のハイインピーダンス要求フラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_POE.c

使用RPDL関数

R_POE_GetStatus

詳細

- POEn#(n:0～11)端子へのハイインピーダンス要求信号入力フラグ(POEnF n:0～11)を取得します。
- 取得するフラグに対応する引数に格納先アドレスを指定してください。取得しないフラグに対応する引数には0を指定してください。
- GUI上でハイインピーダンス要求条件に指定していないPOE端子のフラグには有効な値が格納されません。

使用例

GUI上で以下の通り設定した場合

- MTU3,4の端子出力を設定 (MTUの設定GUI上)
- MTU3,4の出力端子をPOEのハイインピーダンス制御対象に指定
- ハイインピーダンス要求条件にPOE0を指定

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

bool poe0;

void func(void)
{
    R_PG_Timer_Set_MTU_U0_C3();    //MTUの設定
    R_PG_POE_Set();    // POEの設定
    R_PG_Timer_StartCount_MTU_U0_C3();    //MTUのカウント動作開始

    //ハイインピーダンス要求入力を待つ
    do{
        R_PG_POE_GetRequestFlagHiZ_MTU3_4( &poe0 );
    }while( ! poe0 );

    //ハイインピーダンス要求入力時処理

    R_PG_POE_ClearFlag_MTU3_4();    //ハイインピーダンス要求フラグのクリア
}
```

5.9.4 R_PG_POE_GetShortFlag_<タイマチャネル>

定義 bool R_PG_POE_GetShortFlag_MTU3_4 (bool * detected)
bool R_PG_POE_GetShortFlag_MTU6_7 (bool * detected)

概要 MTU端子の出力短絡フラグの取得

引数

bool* detected	出力短絡フラグ(MTU3,4:OSF1またはMTU6,7:OSF2)の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_POE.c

使用RPDL関数 R_POE_GetStatus

詳細
- MTU3,4またはMTU6,7の相補PWM出力短絡フラグ(MTU3,4:OSF1またはMTU6,7:OSF2)を取得します。

使用例 GUI上で以下の通り設定した場合
- アウトプットイネーブル割り込み1(OEI1)を有効に設定
- アウトプットイネーブル割り込み1の通知関数名にPoeOei1IntFuncを指定

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

void func(void)
{
    R_PG_POE_Set();    // POEの設定
}

void PoeOei1IntFunc(void)
{
    bool detected;

    //出力短絡フラグの取得
    R_PG_POE_GetShortFlag_MTU3_4 (&detected);

    if( detected ){
        //MTU3,4の出力短絡検出時処理

        R_PG_POE_ClearFlag_MTU3_4();    //出力短絡フラグ(OSF1)のクリア
    }
}
```

5.9.5 R_PG_POE_ClearFlag_〈タイマチャネル〉

定義 bool R_PG_POE_ClearFlag_〈タイマチャネル〉(void)

〈タイマチャネル〉: MTU0, MTU3_4, MTU6_7, GPT0_1, GPT2_3

概要 ハイインピーダンス要求フラグと出力短絡フラグのクリア

引数 なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル R_PG_POE.c

使用RPDL関数 R_POE_Control

詳細

- ハイインピーダンス要求フラグと出力短絡フラグをクリアします。
- タイマの各チャネルに対応した関数でクリアされるフラグは次の通りです。

タイマチャネル	クリア対象
MTU3, 4	POE0要求フラグ(POE0F)、MTU3,4出力短絡フラグ(OSF1)
MTU6, 7	POE4要求フラグ(POE4F)、MTU6,7出力短絡フラグ(OSF2)
MTU0	POE8要求フラグ(POE8F)
GPT0, 1	POE10要求フラグ(POE10F)
GPT2, 3	POE11要求フラグ(POE11F)

使用例 GUI上で以下の通り設定した場合

- アウトプットイネーブル割り込み1(OEI1)を有効に設定
- アウトプットイネーブル割り込み1の通知関数名にPoeOei1IntFuncを指定

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

void func(void)
{
    R_PG_POE_Set();    // POEの設定
}

void PoeOei1IntFunc(void)
{
    bool detected;

    //出力短絡フラグの取得
    R_PG_POE_GetShortFlag_MTU3_4 (&detected);

    if( detected ){
        //MTU3,4の出力短絡検出時処理

        R_PG_POE_ClearFlag_MTU3_4();    //出力短絡フラグのクリア
    }
}
```

5.10 汎用PWM タイマ（GPT）

5.10.1 R_PG_Timer_Set_GPT_U〈ユニット番号〉

定義　bool R_PG_Timer_Set_GPT_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要　GPTユニットの設定（各チャネルで共通の設定）

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数　R_GPT_Set, R_GPT_ControlUnit

詳細

- GPTのモジュールストップ状態を解除して、使用するタイマ入出力端子を設定します。また、LOCOカウント機能を使用する場合は本関数内で設定されます。
  R_PG_Timer_Set_GPT_U〈ユニット番号〉C〈チャネル番号〉 を呼び出す前に本関数を呼び出してください。
- LOCLカウントを開始するには本関数を呼び出した後
  R_PG_Timer_StartCount_LOCO_GPT_U〈ユニット番号〉 を呼び出してください。

使用例

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

void func(void)
{
    R_PG_Timer_Set_GPT_U0();                // GPTのモジュールストップ状態を解除
    R_PG_Timer_Set_GPT_U0_C0();                    // GPT0の設定
    R_PG_Timer_SetGTCCR_A_GPT_U0_C0( 0x6000 );  // GTCCRAの設定
    R_PG_Timer_SetGTCCR_C_GPT_U0_C0( 0x4000 );  // GTCCRCの設定
    R_PG_Timer_StartCount_GPT_U0_C0();             //カウント動作開始
}
```

5.10.2 R_PG_Timer_Set_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_Set_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉(void)
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要 GPT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_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_GPT_Create, R_GPT_ControlChannel

詳細
- GPTチャネルを初期設定します。
- 本関数を呼び出す前に R_PG_Timer_Set_GPT_U〈*ユニット番号*〉 によりGPTのモジュールストップ状態を解除してください。
- コンペアキャプチャレジスタ(GTCCRA～GTCCRF)は本関数で設定されません。コンペアキャプチャレジスタを設定するには R_PG_Timer_SetGTCCR_n_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉(n：A to F) を使用してください。のこぎり波ワンショットパルスモードおよび三角波PWMモード3では、バッファ強制転送によりコンペアキャプチャレジスタA,Bを設定します。バッファ強制転送は R_PG_Timer_Buffer_Force_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉 により実行することができます。
- カウント動作を開始するには、コンペアキャプチャレジスタの設定後、R_PG_Timer_StartCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉 または R_PG_Timer_SynchronouslyStartCount_GPT_U〈*ユニット番号*〉 を呼び出してください。
- 本関数内でGPT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「5.21 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 R_PG_Timer_Set_GPT_U〈*ユニット番号*〉 の使用例を参照してください。

5.10.3 R_PG_Timer_StartCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義 bool R_PG_Timer_StartCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉(void)
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要 GPTのカウント動作開始

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_GPT_ControlChannel

詳細
- GPTのカウント動作を開始します。
- あらかじめ R_PG_Timer_Set_GPT_U〈*ユニット番号*〉 および R_PG_Timer_Set_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉 によりGPTを初期設定してください。
- 複数のチャネルのカウント動作を同時に開始するには、R_PG_Timer_SynchronouslyStartCount_GPT_U〈*ユニット番号*〉を使用してください。

使用例 R_PG_Timer_Set_GPT_U〈*ユニット番号*〉 の使用例を参照してください。

5.10.4 R_PG_Timer_SynchronouslyStartCount_GPT_U〈ユニット番号〉

定義

bool R_PG_Timer_SynchronouslyStartCount_GPT_U〈ユニット番号〉
(bool gpt0, bool gpt1, bool gpt2, bool gpt3)
〈ユニット番号〉: 0

概要

GPTの複数チャネルのカウント動作を同時に開始

引数

bool gpt0	チャネル0のカウント動作 (0:カウント開始しない 1:カウント開始)
bool gpt1	チャネル1のカウント動作 (0:カウント開始しない 1:カウント開始)
bool gpt2	チャネル2のカウント動作 (0:カウント開始しない 1:カウント開始)
bool gpt3	チャネル3のカウント動作 (0:カウント開始しない 1:カウント開始)

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_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_GPT_ControlUnit

詳細

- GPTの複数チャネルのカウント動作を同時に開始します。
- あらかじめ R_PG_Timer_Set_GPT_U〈ユニット番号〉 および R_PG_Timer_Set_GPT_U〈ユニット番号〉_C〈チャネル番号〉 によりGPTを初期設定してください。

使用例

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

void func1(void)
{
    R_PG_Timer_Set_GPT_U0();   // GPTのモジュールストップ状態を解除
    R_PG_Timer_Set_GPT_U0_C0();   // GPT0の設定
    R_PG_Timer_Set_GPT_U0_C2();   // GPT2の設定

    R_PG_Timer_SetGTCCR_A_GPT_U0_C0( 0x0000 );   // GPT0.GTCCRAの設定
    R_PG_Timer_SetGTCCR_C_GPT_U0_C0( 0x00ff );   // GPT0.GTCCRCの設定

    R_PG_Timer_SetGTCCR_A_GPT_U0_C2( 0x0000 );   // GPT2.GTCCRAの設定
    R_PG_Timer_SetGTCCR_C_GPT_U0_C2( 0x00ff );   // GPT2.GTCCRCの設定
}
void func2(void)
{
    // GPT0、2のカウント動作を開始
    R_PG_Timer_SynchronouslyStartCount_GPT_U0( 1, 0, 1, 0 );
}

void func3(void)
{
    // GPT0、2のカウント動作を停止
    R_PG_Timer_SynchronouslyHaltCount_GPT_U0( 1, 0, 1, 0 );
}
```

5.10.5 R_PG_Timer_HaltCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

bool R_PG_Timer_HaltCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉(void)
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要

GPTのカウント動作を停止

引数

なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- GPTのカウント動作を一時停止します。
- カウント動作を再開するには
  R_PG_Timer_StartCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉 または
  R_PG_Timer_SynchronouslyStartCount_GPT_U〈*ユニット番号*〉 を呼び出してください。

使用例

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

void func(void)
{
    // GPT0のカウントを停止
    R_PG_Timer_HaltCount_GPT_U0_C0();

    // カウンタの設定
    R_PG_Timer_SetCounterValue_GPT_U0_C0( 0xff );

    // GPT0のカウントを再開
    R_PG_Timer_StartCount_GPT_U0_C0();
}
```

5.10.6 R_PG_Timer_SynchronouslyHaltCount_GPT_U〈ユニット番号〉

定義

bool R_PG_Timer_SynchronouslyHaltCount_GPT_U〈ユニット番号〉
（ bool gpt0， bool gpt1， bool gpt2， bool gpt3 ）
〈ユニット番号〉: 0

概要

GPTの複数チャネルのカウント動作を同時に停止

引数

bool gpt0	チャネル0のカウント動作（0:カウント停止しない 1:カウント停止）
bool gpt1	チャネル1のカウント動作（0:カウント停止しない 1:カウント停止）
bool gpt2	チャネル2のカウント動作（0:カウント停止しない 1:カウント停止）
bool gpt3	チャネル3のカウント動作（0:カウント停止しない 1:カウント停止）

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_GPT_ControlUnit

詳細

- MTUの複数チャネルのカウント動作を同時に停止します。
- カウント動作を再開するには
  R_PG_Timer_StartCount_GPT_U〈ユニット番号〉_C〈チャネル番号〉 または
  R_PG_Timer_SynchronouslyStartCount_GPT_U〈ユニット番号〉 を呼び出してください。

使用例

R_PG_Timer_SynchronouslyStartCount_GPT_U〈ユニット番号〉 の使用例を参照してください。

5.10.7 R_PG_Timer_SetGTCCR_〈*GTCCR*〉_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

bool R_PG_Timer_SetGTCCR_〈*GTCCR*〉_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉
(uint16_t gtccr_val)
〈*GTCCR*〉: A～F
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要

コンペアキャプチャレジスタ (GTCCRn n:A～F) 値の設定

引数

uint16_t gtccr_val	コンペアキャプチャレジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- コンペアキャプチャレジスタ(GTCCRn n:A～F)の値を設定します。
- R_PG_Timer_Set_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉 ではコンペアキャプチャレジスタは設定されません。初期設定時にコンペアキャプチャレジスタを設定する場合は、カウントを開始する前に本関数によりコンペアキャプチャレジスタを設定してください。

使用例

R_PG_Timer_Set_GPT_U〈*ユニット番号*〉 の使用例を参照してください。

5.10.8 R_PG_Timer_GetGTCCR_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

bool R_PG_Timer_GetGTCCR_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉
(uint16_t * gtccr_a_val, uint16_t * gtccr_b_val, uint16_t * gtccr_c_val,
uint16_t * gtccr_d_val, uint16_t * gtccr_e_val, uint16_t * gtccr_f_val)

〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要

コンペアキャプチャレジスタ（ GTCCRA～F ）値の取得

引数

uint16_t * gtccr_a_val	コンペアキャプチャレジスタA値の格納先
uint16_t * gtccr_b_val	コンペアキャプチャレジスタB値の格納先
uint16_t * gtccr_c_val	コンペアキャプチャレジスタC値の格納先
uint16_t * gtccr_d_val	コンペアキャプチャレジスタD値の格納先
uint16_t * gtccr_e_val	コンペアキャプチャレジスタE値の格納先
uint16_t * gtccr_f_val	コンペアキャプチャレジスタF値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数

R_GPT_ReadChannel

詳細

- コンペアキャプチャレジスタ(GTCCRA～F)の値を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。

使用例

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

uint16_t  gtccr_a_val, gtccr_c_val;

void func(void)
{
    // GTCCRA, GTCCRB の値を取得する
    R_PG_Timer_GetGTCCR_GPT_U0_C0(
        &gtccr_a_val,   //GTCCRA
        0,              //GTCCRB (取得しない)
        &gtccr_c_val,   //GTCCRC
        0,              //GTCCRD (取得しない)
        0,              //GTCCRE (取得しない)
        0               //GTCCRF (取得しない)
    );
}
```

5.10.9 R_PG_Timer_SetCounterValue_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SetCounterValue_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉
(uint16_t counter_val)

〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要

GPTのカウンタ値を設定

引数

uint16_t counter_val	カウンタに設定する値

戻り値

true	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_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- カウンタ値を設定します。
- カウンタ値はカウント停止中のみ変更可能です。

使用例

R_PG_Timer_HaltCount_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉 の使用例を参照してください。

5.10.10 R_PG_Timer_GetCounterValue_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetCounterValue_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t * counter_val)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

GPTのカウンタ値の取得

引数

uint16_t * counter_val	カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ReadChannel

詳細

- カウンタ値を取得します。

使用例

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

uint16_t  counter_val;

void func(void)
{
      //カウンタ値の取得
      R_PG_Timer_GetCounterValue_GPT_U0_C0( & counter_val );
}
```

5.10.11 R_PG_Timer_SynchronouslyClearCounter_GPT_U〈*ユニット番号*〉

定義

bool R_PG_Timer_SynchronouslyClearCounter_GPT_U〈*ユニット番号*〉
(bool gpt0, bool gpt1, bool gpt2, bool gpt3)
〈*ユニット番号*〉: 0

概要

複数チャネルのカウンタを同時にクリア

引数

bool gpt0	GPT0のカウンタクリア動作 (0:カウンタクリアしない 1:カウンタクリア)
bool gpt1	GPT1のカウンタクリア動作 (0:カウンタクリアしない 1:カウンタクリア)
bool gpt2	GPT2のカウンタクリア動作 (0:カウンタクリアしない 1:カウンタクリア)
bool gpt3	GPT3のカウンタクリア動作 (0:カウンタクリアしない 1:カウンタクリア)

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉.c
〈*ユニット番号*〉: 0

使用RPDL関数

R_GPT_ControlUnit

詳細

- GPTの複数チャネルのカウンタを同時にクリアします。

使用例

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

uint16_t  counter_val;

void func(void)
{
      // GPT0, GPT2 のカウンタをクリア
      R_PG_Timer_SynchronouslyClearCounter_GPT_U0( 1, 0, 1, 0 );
}
```

5.10.12 R_PG_Timer_SetCycle_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義 bool R_PG_Timer_SetCycle_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉(uint16_t gtpr_val)

〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要 タイマ周期設定レジスタ(GTPR)値の設定

引数

uint16_t gtpr_val	タイマ周期設定レジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数 R_GPT_ControlChannel

詳細

- タイマ周期設定レジスタ(GTPR)の値を設定します。

使用例

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

void func(void)
{
      //タイマ周期設定レジスタの設定
      R_PG_Timer_SetCycle_GPT_U0_C0( 0x6000 );
}
```

5.10.13 R_PG_Timer_SetBuffer_Cycle_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SetBuffer_Cycle_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t gtpbr_val)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

タイマ周期設定バッファレジスタ(GTPBR)値の設定

生成条件

周期レジスタ(GTPBR)のバッファ動作を設定

引数

uint16_t gtpbr_val	タイマ周期設定バッファレジスタ(GTPBR)に設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- タイマ周期設定バッファレジスタ(GTPBR)の値を設定します。

使用例

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

void func(void)
{
    //タイマ周期設定バッファレジスタの設定
    R_PG_Timer_SetBuffer_Cycle_GPT_U0_C0( 0x5000 );
}
```

5.10.14 R_PG_Timer_SetDoubleBuffer_Cycle_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SetDoubleBuffer_Cycle_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t gtpdbr_val)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

タイマ周期設定ダブルバッファレジスタ(GTPDBR)値の設定

生成条件

周期レジスタ(GTPR)のダブルバッファ動作を設定

引数

uint16_t gtpdbr_val	タイマ周期設定ダブルバッファレジスタ(GTPDBR)に設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- タイマ周期設定ダブルバッファレジスタ(GTPDBR)の値を設定します。

使用例

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

void func(void)
{
    //タイマ周期設定ダブルバッファレジスタの設定
    R_PG_Timer_SetDoubleBuffer_Cycle_GPT_U0_C0( 0x4000 );
}
```

5.10.15 R_PG_Timer_SetAD_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SetAD_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t gtadtra_val, uint16_t gtadtrb_val)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

A/D変換開始要求タイミングレジスタA,B (GTADTRA, GTADTRB) 値の設定

生成条件

A/D変換開始要求を使用する

引数

uint16_t gtadtra_val	GTADTRAに設定する値
uint16_t gtadtrb_val	GTADTRBに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- A/D変換開始要求タイミングレジスタA,B (GTADTRA, GTADTRB) の値を設定します。

使用例

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

void func(void)
{
    // A/D変換開始要求タイミングレジスタの設定
    R_PG_Timer_SetAD_GPT_U0_C0(
        0x3000,   // A/D変換開始要求タイミングレジスタA (GTADTRA)
        0x2000    // A/D変換開始要求タイミングレジスタB (GTADTRB)
    );
}
```

5.10.16 R_PG_Timer_SetBuffer_AD_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SetBuffer_AD_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t gtadtbra_val, uint16_t gtadtbrb_val)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

A/D変換開始要求タイミングバッファレジスタA,B (GTADTBRA, GTADTBRB) 値の設定

生成条件

A/D変換開始要求タイミングレジスタのバッファ転送を使用する

引数

uint16_t gtadtbra_val	GTADTBRAに設定する値
uint16_t gtadtbrb_val	GTADTBRBに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- A/D変換開始要求タイミングバッファレジスタA,B (GTADTBRA, GTADTBRB) の値を設定します。

使用例

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

void func(void)
{
    // A/D変換開始要求タイミングバッファレジスタの設定
    R_PG_Timer_SetBuffer_AD_GPT_U0_C0(
        0x6000,   // A/D変換開始要求タイミングバッファレジスタA (GTADTBRA)
        0x3000    // A/D変換開始要求タイミングバッファレジスタB (GTADTBRB)
    );
}
```

5.10.17 R_PG_Timer_SetDoubleBuffer_AD_GPT_U*<ユニット番号>*_C*<チャネル番号>*

定義

bool R_PG_Timer_SetDoubleBuffer_AD_GPT_U*<ユニット番号>*_C*<チャネル番号>*
(uint16_t gtadtdbra_val, uint16_t gtadtdbrb_val)

<ユニット番号>: 0
<チャネル番号>: 0～3

概要

A/D変換開始要求タイミングダブルバッファレジスタA,B (GTADTDBRA, GTADTDBRB) 値の設定

生成条件

A/D変換開始要求タイミングレジスタのダブルバッファ転送を使用する

引数

uint16_t gtadtdbra_val	GTADTDBRAに設定する値
uint16_t gtadtdbrb_val	GTADTDBRBに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U*<ユニット番号>*_C*<チャネル番号>*.c
<ユニット番号>: 0
<チャネル番号>: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- A/D変換開始要求タイミングダブルバッファレジスタA,B (GTADTDBRA, GTADTDBRB) の値を設定します。

使用例

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

void func(void)
{
    // A/D変換開始要求タイミングダブルバッファレジスタの設定
    R_PG_Timer_SetDoubleBuffer_AD_GPT_U0_C0(
        0x8000,   // GTADTDBRA
        0x4000    // GTADTDBRB
    );
}
```

5.10.18 R_PG_Timer_SetBuffer_GTDV〈*U/D*〉_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

bool R_PG_Timer_SetBuffer_GTDVU_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉
（ uint16_t gtdbu_val ）

bool R_PG_Timer_SetBuffer_GTDVD_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉
（ uint16_t gtdbd_val ）

〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要

タイマデッドタイムバッファレジスタU,D (GTDBU, GTDBD) 値の設定

生成条件

デッドタイム自動設定が有効

引数

uint16_t gtdbu_val	GTDVUに設定する値
uint16_t gtdbd_val	GTDVDに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- タイマデッドタイム値レジスタU,D (GTDVU, GTDVD) のバッファレジスタであるタイマデッドタイムバッファレジスタU,D (GTDBU, GTDBD) の値を設定します。

使用例

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

void func(void)
{
    // タイマデッドタイム値レジスタUの設定
    R_PG_Timer_SetBuffer_GTDVU_GPT_U0_C0( 0x500 );
    // タイマデッドタイム値レジスタDの設定
    R_PG_Timer_SetBuffer_GTDVD_GPT_U0_C0( 0x300 );
}
```

5.10.19 R_PG_Timer_GetRequestFlag_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetRequestFlag_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(bool * cm_ic_a, bool * cm_ic_b, bool * cm_c, bool * cm_d,
bool * cm_e, bool * cm_f, bool * ov, bool * un, bool * dt_error)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

割り込み要求フラグの取得とクリア

引数

bool * cm_ic_a	コンペアマッチ/インプットキャプチャAフラグの格納先
bool * cm_ic_b	コンペアマッチ/インプットキャプチャBフラグの格納先
bool * cm_c	コンペアマッチ/インプットキャプチャCフラグの格納先
bool * cm_d	コンペアマッチ/インプットキャプチャDフラグの格納先
bool * cm_e	コンペアマッチ/インプットキャプチャEフラグの格納先
bool * cm_f	コンペアマッチ/インプットキャプチャFフラグの格納先
bool * ov	オーバフローフラグの格納先
bool * un	アンダフローフラグの格納先
bool * dt_error	デッドタイムエラーフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ReadChannel

詳細

- GPTの割り込み要求フラグを取得します。
- 本関数内で全フラグがクリアされます。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

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

bool cm_ic_a, ov;

void func(void)
{
    //コンペアマッチ/インプットキャプチャAフラグとオーバフローフラグの取得
    R_PG_Timer_GetRequestFlag_GPT_U0_C0(
        &cm_ic_a,    // コンペアマッチ/インプットキャプチャAフラグ
        0,           // コンペアマッチ/インプットキャプチャBフラグ（取得しない）
        0,           // コンペアマッチ/インプットキャプチャCフラグ（取得しない）
        0,           // コンペアマッチ/インプットキャプチャDフラグ（取得しない）
        0,           // コンペアマッチ/インプットキャプチャEフラグ（取得しない）
        0,           // コンペアマッチ/インプットキャプチャFフラグ（取得しない）
        &ov,         // オーバフローフラグ
        0,           //アンダフローフラグ（取得しない）
        0            // デッドタイムエラーフラグ（取得しない）
    )
}
```

5.10.20 R_PG_Timer_GetRequestFlag_GPT_U〈ユニット番号〉

定義

bool R_PG_Timer_GetRequestFlag_GPT_U〈ユニット番号〉
(bool * loco_rising, bool * loco_deviation, bool * loco_ov,
bool * ext_rising, bool * ext_falling)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

LOCOカウント機能および外部トリガの割り込み要求フラグの取得とクリア

引数

bool * loco_rising	LOCO分周クロック立ち上がり割り込み要求フラグの格納先
bool * loco_deviation	LOCOカウント値偏差超え割り込み要求フラグの格納先
bool * loco_ov	LCNTオーバフロー割り込み要求フラグの格納先
bool * ext_rising	外部トリガ立ち上がり入力割り込み要求フラグの格納先
bool * ext_falling	外部トリガ立ち下がり入力割り込み要求フラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ReadUnit

詳細

- LOCOカウント機能および外部トリガの割り込み要求フラグを取得しクリアします。
- 本関数内で全フラグがクリアされます。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

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

bool loco_deviation, ext_rising;

void func(void)
{
    // LOCOカウント値偏差超え割り込み要求フラグと
    //外部トリガ立ち上がり入力割り込み要求フラグの取得

    R_PG_Timer_GetRequestFlag_GPT_U0 (
        0,                  //LOCO分周クロック立ち上がり割り込みフラグ(取得しない)
        &loco_deviation, //LOCOカウント値偏差超え割り込みフラグ
        0,                  //LCNTオーバフロー割り込みフラグ(取得しない)
        &ext_rising,       //外部トリガ立ち上がり入力割り込みフラグ
        0                   //外部トリガ立ち下がり入力割り込みフラグ(取得しない)
    );
}
```

5.10.21 R_PG_Timer_GetCounterStatus_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_GetCounterStatus_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(bool * active, bool * up)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

カウンタの状態の取得

引数

bool * active	カウンタスタートビットの格納先 (0： カウント停止 1： カウント動作)
bool * up	カウント方向フラグの格納先 (0： ダウンカウント 1： アップカウント)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ReadChannel

詳細

- カウンタスタートビットとカウント方向フラグを取得します。
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

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

bool up;

void func(void)
{
    //カウント方向の取得
    R_PG_Timer_GetCounterStatus_GPT_U0_C0 (
      0,                    // カウンタスタートビット（取得しない）
      & up                  // カウント方向フラグ
    );
}
```

5.10.22 R_PG_Timer_BufferEnable_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_BufferEnable_GPT_U〈ユニット番号〉_C〈チャネル番号〉
（ bool gtccr, bool gtpr, bool gtadtr, bool gtdv ）

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

バッファ動作の有効化

引数

bool gtccr	コンペアキャプチャレジスタGTCCRA, GTCCRC, GTCCRDおよび、GTCCRB, GTCCRE, GTCCRFレジスタのバッファ転送設定 （ 0:バッファ転送を有効にしない 1:バッファ転送を有効にする ）
bool gtpr	周期設定レジスタ(GTPR), 周期設定バッファレジスタ(GTPBR), 周期設定ダブルバッファレジスタ(GTPDBR)のバッファ転送設定 （ 0:バッファ転送を有効にしない 1:バッファ転送を有効にする ）
bool gtadtr	A/D変換開始要求タイミングレジスタ(GTADTRA), A/D変換開始要求タイミングバッファレジスタ(GTADTBRA), A/D変換開始要求タイミングダブルバッファレジスタ(GTADTDBRA) のバッファ転送設定 （ 0:バッファ転送を有効にしない 1:バッファ転送を有効にする ）
bool gtdv	タイマデッドタイム値レジスタU,D (GTDVU, GTDVD) とタイマデッドタイムバッファレジスタU,D (GTDBU, GTDBD) のバッファ転送設定 （ 0:バッファ転送を有効にしない 1:バッファ転送を有効にする ）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- バッファ動作を有効化します。

使用例

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

void func(void)
{
    // GTCCRA, C, D と GTCCRB, E, F のバッファ動作を有効にする
    R_PG_Timer_BufferEnable_GPT_U0_C0 (
        1,  // GTCCRA, C, D と GTCCRB, E, F のバッファ動作を有効にする
        0,  // GTPR のバッファ動作を有効にしない
        0,  // GTADTRA のバッファ動作を有効にしない
        0   // GTDVU と GTDVD のバッファ動作を有効にしない
    );
}
```

5.10.23 R_PG_Timer_BufferDisable_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_BufferDisable_GPT_U〈ユニット番号〉_C〈チャネル番号〉
（ bool gtccr, bool gtpr, bool gtadtr, bool gtdv ）

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

バッファ動作の無効化

引数

bool gtccr	コンペアキャプチャレジスタGTCCRA, GTCCRC, GTCCRDおよび、GTCCRB, GTCCRE, GTCCRFレジスタのバッファ転送設定 （ 0:バッファ転送を無効にしない 1:バッファ転送を無効にする ）
bool gtpr	周期設定レジスタ(GTPR), 周期設定バッファレジスタ(GTPBR), 周期設定ダブルバッファレジスタ(GTPDBR)のバッファ転送設定 （ 0:バッファ転送を無効にしない 1:バッファ転送を無効にする ）
bool gtadtr	A/D変換開始要求タイミングレジスタ(GTADTRA), A/D変換開始要求タイミングバッファレジスタ(GTADTBRA), A/D変換開始要求タイミングダブルバッファレジスタ(GTADTDBRA) のバッファ転送設定 （ 0:バッファ転送を無効にしない 1:バッファ転送を無効にする ）
bool gtdv	タイマデッドタイム値レジスタU,D (GTDVU, GTDVD) とタイマデッドタイムバッファレジスタU,D (GTDBU, GTDBD) のバッファ転送設定 （ 0:バッファ転送を無効にしない 1:バッファ転送を無効にする ）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- バッファ動作を無効化します。

使用例

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

void func(void)
{
    // GTCCRA, C, D と GTCCRB, E, F のバッファ動作を無効にする
    R_PG_Timer_BufferDisable_GPT_U0_C0 (
        1,  // GTCCRA, C, D と GTCCRB, E, F のバッファ動作を無効にする
        0,  // GTPR のバッファ動作を無効にしない
        0,  // GTADTRA のバッファ動作を無効にしない
        0   // GTDVU と GTDVD のバッファ動作を無効にしない
    );
}
```

5.10.24 R_PG_Timer_Buffer_Force_GPT_U *〈ユニット番号〉*_C*〈チャネル番号〉*

定義　bool R_PG_Timer_Buffer_Force_GPT_U*〈ユニット番号〉*_C*〈チャネル番号〉*(void)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要　バッファ強制転送の実行

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_GPT_U*〈ユニット番号〉*_C*〈チャネル番号〉*.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数　R_GPT_ControlChannel

詳細
- GTCCRAおよびGTCCRBのバッファ強制転送を実行します。

使用例

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

void func(void)
{
    R_PG_Timer_Set_GPT_U0();                    // GPTのモジュールストップ状態を解除
    R_PG_Timer_Set_GPT_U0_C0();                 // GPT0の設定
    R_PG_Timer_SetGTCCR_C_GPT_U0_C0( 0x6000 );  // GTCCRCの設定
    R_PG_Timer_SetGTCCR_D_GPT_U0_C0( 0x3000 );  // GTCCRDの設定
    R_PG_Timer_SetGTCCR_E_GPT_U0_C0( 0x8000 );  // GTCCREの設定
    R_PG_Timer_SetGTCCR_F_GPT_U0_C0( 0x4000 );  // GTCCRFの設定
    R_PG_Timer_Buffer_Force_GPT_U0_C0();        //バッファ強制転送の実行
    R_PG_Timer_StartCount_GPT_U0_C0();          //カウント動作開始
}
```

5.10.25 R_PG_Timer_CountDirection_Down_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_CountDirection_Down_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(bool force)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

カウント方向のダウンカウントへの切り替え

引数

bool force	カウント方向強制設定 (0:強制設定しない 1:強制設定する)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- カウント方向をダウンカウントに切り替えます。

使用例

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

void func(void)
{
    // カウント方向をダウンカウントに設定 ( 強制設定しない )
    R_PG_Timer_CountDirection_Down_GPT_U0_C0( 0 );
}
```

5.10.26 R_PG_Timer_CountDirection_Up_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉

定義

bool R_PG_Timer_CountDirection_Up_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉
(bool force)

〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

概要

カウント方向のアップカウントへの切り替え

引数

bool force	カウント方向強制設定 (0:強制設定しない 1:強制設定する)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉_C〈*チャネル番号*〉.c
〈*ユニット番号*〉: 0
〈*チャネル番号*〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- カウント方向をアップカウントに切り替えます。

使用例

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

void func(void)
{
    // カウント方向をアップカウントに設定 ( 強制設定 )
    R_PG_Timer_CountDirection_Up_GPT_U0_C0( 1 );
}
```

5.10.27 R_PG_Timer_SoftwareNegate_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_SoftwareNegate_GPT_U〈ユニット番号〉_C〈チャネル番号〉
(bool on)

〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

概要

GTIOCnAおよびGTIOCnB端子出力のソフトウェアネゲート制御 (n:チャネル番号)

生成条件

GTIOCnAまたはGTIOCnB端子のネゲート制御を有効にし、ネゲート要因にソフトウェア制御を選択

引数

bool on	ネゲート要因の出力値 (1:ON 0:OFF)

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_C〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉: 0～3

使用RPDL関数

R_GPT_ControlChannel

詳細

- GTIOCnAおよびGTIOCnB端子出力をネゲート制御します。
  (n:チャネル番号)

使用例

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

void func(void)
{
    // ネゲート要因の値を1に設定
    R_PG_Timer_CountDirection_Up_GPT_U0_C0( 1 );
}
```

5.10.28 R_PG_Timer_StartCount_LOCO_GPT_U〈ユニット番号〉

定義 bool R_PG_Timer_StartCount_LOCO_GPT_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 LOCOのカウントを開始

生成条件 LOCOカウント機能が有効

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_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数 R_GPT_ControlUnit

詳細
- LOCOのカウントを開始します。

使用例

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

void func(void)
{
    // IWDTの設定と動作開始
    R_PG_Timer_Set_IWDT();
    R_PG_Timer_RefreshCounter_IWDT();

    // GPTのモジュールストップ状態を解除し、LOCOカウント機能を設定
    R_PG_Timer_Set_GPT_U0();

    // LOCOのカウント開始
    R_PG_Timer_StartCount_LOCO_GPT_U0();
}
```

5.10.29 R_PG_Timer_HaltCount_LOCO_GPT_U〈ユニット番号〉

定義 bool R_PG_Timer_HaltCount_LOCO_GPT_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 LOCOのカウントを停止

生成条件 LOCOカウント機能が有効

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

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

使用RPDL関数 R_GPT_ControlUnit

詳細
- LOCOのカウントを停止します。

使用例

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

void func(void)
{
    // LOCOのカウント停止
    R_PG_Timer_HaltCount_LOCO_GPT_U0();
}
```

5.10.30 R_PG_Timer_ClearCounter_LOCO_GPT_U〈ユニット番号〉

定義 bool R_PG_Timer_ClearCounter_LOCO_GPT_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 LOCOカウント値レジスタのクリア

生成条件 LOCOカウント機能が有効

引数 なし

戻り値

true	クリアに成功した場合
false	クリアに失敗した場合

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

使用RPDL関数 R_GPT_ControlUnit

詳細
- LOCOカウント値レジスタをクリアします。

使用例

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

void func(void)
{
    // LOCOカウント値レジスタのクリア
    R_PG_Timer_ClearCounter_LOCO_GPT_U0();
}
```

5.10.31 R_PG_Timer_InitialiseCountResultValue_LOCO_GPT_U〈ユニット番号〉

定義 bool R_PG_Timer_InitialiseCountResultValue_LOCO_GPT_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 LOCOカウント結果レジスタの初期化

生成条件 LOCOカウント機能が有効

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_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数 R_GPT_ControlUnit

詳細
- LOCOカウント結果レジスタLCNT00の値で、LCN01～LCN15をクリアします。

使用例

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

void func(void)
{
    // LOCOカウント結果レジスタの初期化
    R_PG_Timer_InitialiseCountResultValue_LOCO_GPT_U0();
}
```

5.10.32 R_PG_Timer_GetCounterValue_LOCO_GPT_U〈ユニット番号〉

定義

bool R_PG_Timer_GetCounterValue_LOCO_GPT_U〈ユニット番号〉
(uint16_t * loco_counter_val)
〈ユニット番号〉: 0

概要

LOCOカウント値レジスタの取得

生成条件

LOCOカウント機能が有効

引数

uint16_t * loco_counter_val	LOCOカウント値レジスタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_GPT_ReadUnit

詳細

- LOCOカウント値レジスタの値を取得します。

使用例

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

uint16_t loco_counter_val;

void func(void)
{
    // LOCOカウント値レジスタの取得
    R_PG_Timer_InitialiseCountResultValue_LOCO_GPT_U0( &loco_counter_val );
}
```

5.10.33 R_PG_Timer_GetCounterAverageValue_LOCO_GPT_U〈*ユニット番号*〉

定義

bool R_PG_Timer_GetCounterAverageValue_LOCO_GPT_U〈*ユニット番号*〉
(uint16_t * loco_counter_ave_val)
〈*ユニット番号*〉: 0

概要

LOCOのカウント結果の平均値を取得

生成条件

LOCOカウント機能が有効

引数

uint16_t * loco_counter_ave_val	LOCOのカウント結果の平均値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉.c
〈*ユニット番号*〉: 0

使用RPDL関数

R_GPT_ReadUnit

詳細

- LOCOカウント結果平均レジスタ値を取得します。

使用例

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

uint16_t    loco_counter_ave_val;

void func(void)
{
      // LOCOのカウント結果の平均値を取得
      R_PG_Timer_GetCounterAverageValue_LOCO_GPT_U0( & loco_counter_ave_val );
}
```

5.10.34 R_PG_Timer_GetCountResultValue_LOCO_GPT_U〈ユニット番号〉

定義

bool R_PG_Timer_GetCountResultValue_LOCO_GPT_U〈ユニット番号〉
(uint16_t * loco_count_result_val)
〈ユニット番号〉: 0

概要

LOCOのカウント結果の取得

生成条件

LOCOカウント機能が有効

引数

uint16_t * loco_count_result_val	LOCOのカウント結果の格納先の先頭アドレス (32バイトの領域を確保してください)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_GPT_ReadUnit

詳細

- LOCOカウント結果レジスタ（LCNT00～LCNT15）の値を取得します。

使用例

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

uint16_t    loco_count_result_val[16];

void func(void)
{
      // LOCOのカウント結果の取得
      R_PG_Timer_GetCountResultValue_LOCO_GPT_U0( loco_count_result_val );
}
```

5.10.35 R_PG_Timer_SetPermissibleDeviation_LOCO_GPT_U〈ユニット番号〉

定義

bool R_PG_Timer_SetPermissibleDeviation_LOCO_GPT_U〈ユニット番号〉
(uint16_t maximum_val, uint16_t minimum_val)
〈ユニット番号〉: 0

概要

LOCOのカウント上限／下限許容偏差値の設定

生成条件

LOCOカウント機能が有効かつ、LOCOカウント値偏差超え割り込みが有効

引数

uint16_t maximum_val	LOCOカウント上限許容偏差値レジスタに設定する値
uint16_t minimum_val	LOCOカウント下限許容偏差値レジスタに設定する値

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉.c
〈ユニット番号〉: 0

使用RPDL関数

R_GPT_ControlUnit

詳細

- LOCOのカウント上限／下限許容偏差値を設定します。

使用例

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

void func(void)
{
    // LOCOのカウント上限／下限許容偏差値の設定
    R_PG_Timer_SetPermissibleDeviation_LOCO_GPT_U0(
        0x10   //上限許容偏差値
        0x10   //下限許容偏差値
    );
}
```

5.10.36 R_PG_Timer_AdjustEdgeDelay_GPT_U〈ユニット番号〉_C〈チャネル番号〉

定義

bool R_PG_Timer_AdjustEdgeDelay_GPT_U〈ユニット番号〉_〈チャネル番号〉
(uint8_t GTIOCA_Rising_Delay, uint8_t GTIOCA_Falling_Delay,
uint8_t GTIOCB_Rising_Delay, uint8_t GTIOCB_Falling_Delay)
〈ユニット番号〉: 0
〈チャネル番号〉0～3

概要

遅延時間設定値変更

引数

uint8_t GTIOCA_Rising_Delay	GTIOCA 立ち上がり出力遅延レジスタ(GTDLYRA)に設定する値 (1～31:遅延値 0:遅延なし)
uint8_t GTIOCA_Falling_Delay	GTIOCA 立ち下がり出力遅延レジスタ(GTDLYFA)に設定する値 (1～31:遅延値 0:遅延なし)
uint8_t GTIOCB_Rising_Delay	GTIOCB 立ち上がり出力遅延レジスタ(GTDLYRB)に設定する値 (1～31:遅延値 0:遅延なし)
uint8_t GTIOCB_Falling_Delay	GTIOCB 立ち下がり出力遅延レジスタ(GTDLYFB)に設定する値 (1～31:遅延値 0:遅延なし)

戻り値

True	変更に成功した場合
False	変更に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈ユニット番号〉_〈チャネル番号〉.c
〈ユニット番号〉: 0
〈チャネル番号〉:0～3

使用RPDL関数

R_GPT_EdgeDelay_Control

詳細

- 引数で指定した遅延時間を設定します。
  遅延生成機能を有効にする場合は、R_PG_Timer_EnableEdgeDelay_GPT_U〈ユニット番号〉を呼び出してください。
  使用時はハードウェアマニュアルの「PWM遅延生成回路の遅延値設定に関する注意事項」を確認してください。

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)
{
    // GPTの遅延時間設定値変更
    R_PG_Timer_AdjustEdgeDelay_GPT_U0_C0(5, 10, 5, 10);
    // GPTの遅延生成機能を有効にする。
    R_PG_Timer_EnableEdgeDelay_GPT_U0(1, 0, 0, 0);
}
```

5.10.37 R_PG_Timer_EnableEdgeDelay_GPT_U〈*ユニット番号*〉

定義

bool R_PG_Timer_EnableEdgeDelay_GPT_U〈*ユニット番号*〉
(bool C0_Enable, bool C1_Enable, bool C2_Enable, bool C3_Enable)
〈*ユニット番号*〉: 0

概要

遅延生成機能を有効にする

引数

bool C0_Enable	GPT0の遅延生成機能設定（1:有効にする　0:変更しない）
bool C1_Enable	GPT1の遅延生成機能設定（1:有効にする　0:変更しない）
bool C2_Enable	GPT2の遅延生成機能設定（1:有効にする　0:変更しない）
bool C3_Enable	GPT3の遅延生成機能設定（1:有効にする　0:変更しない）

戻り値

true	設定に成功した場合
false	設定に失敗した場合

出力先ファイル

R_PG_Timer_GPT_U〈*ユニット番号*〉.c
〈*ユニット番号*〉: 0

使用RPDL関数

R_GPT_EdgeDelay_Create

詳細

- 引数で指定したGPTのチャネルの遅延生成機能を有効にします。
  遅延生成機能を無効にする場合は、R_PG_Timer_DisableEdgeDelay_GPT_U〈*ユニット番号*〉を呼び出してください。

使用例

```
//この関数を使用するには"R_PG_〈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)
{
    // GPTの遅延時間設定値変更
    R_PG_Timer_AdjustEdgeDelay_GPT_U0_C0(5, 10, 5, 10);
    // GPTの遅延生成機能を有効にする。
    R_PG_Timer_EnableEdgeDelay_GPT_U0(1, 0, 0, 0);
}
```

5.10.38 R_PG_Timer_DisableEdgeDelay_GPT_U〈ユニット番号〉

定義 bool R_PG_Timer_DisableEdgeDelay_GPT_U〈ユニット番号〉
(bool C0_Disable, bool C1_Disable, bool C2_Disable, bool C3_Disable)
〈ユニット番号〉: 0

概要 遅延生成機能を無効にする

引数

bool C0_Disable	GPT0の遅延生成機能設定（1:無効にする　0:変更しない）
bool C1_Disable	GPT1の遅延生成機能設定（1:無効にする　0:変更しない）
bool C2_Disable	GPT2の遅延生成機能設定（1:無効にする　0:変更しない）
bool C3_Disable	GPT3の遅延生成機能設定（1:無効にする　0:変更しない）

戻り値

true	設定に成功した場合
false	設定に失敗した場合

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

使用RPDL関数 R_GPT_EdgeDelay_Create

詳細
- 引数で指定したGPTのチャネルの遅延生成機能を無効にします。

使用例

```
//この関数を使用するには"R_PG_〈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)
{
    // GPTの遅延生成機能を無効にする。
    R_PG_Timer_DisableEdgeDelay_GPT_U0(1, 0, 0, 0);
}
```

5.10.39 R_PG_Timer_StopModule_GPT_U〈ユニット番号〉

定義 bool R_PG_Timer_StopModule_GPT_U〈ユニット番号〉(void)
〈ユニット番号〉: 0

概要 GPTを停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

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

使用RPDL関数 R_GPT_Destroy

詳細
- GPTを停止し、モジュールストップ状態に移行します。複数のチャネルが動作している場合、本関数を呼び出すと全チャネルが停止します。1チャネルの動作だけを停止させる場合は R_PG_Timer_HaltCount_GPT_U〈ユニット番号〉_C〈チャネル番号〉 または R_PG_Timer_SynchronouslyHaltCount_GPT_U〈ユニット番号〉 を呼び出してください。

使用例

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

```
#include "R_PG_default.h"

void func(void)
{
    // GPTを停止
    R_PG_Timer_StopModule_GPT_U0();
}
```

5.11 コンペアマッチタイマ（CMT）

5.11.1 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を設定しカウント動作を開始

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_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、カウント動作を開始します。
- 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「5.21 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_〈プロジェクト名〉.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
}
```

5.11.2 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のカウント動作を一時停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数　R_CMT_Control

詳細
- CMTのカウント動作を一時停止します。カウント動作を再開するには R_PG_Timer_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_〈プロジェクト名〉.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のカウント動作を一時停止
    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のカウント動作を再開
    R_PG_Timer_ResumeCount_CMT_U0_C0();
}
```

5.11.3 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のカウント動作を再開

引数　なし

戻り値

true	カウント動作の再開が正しく行われた場合
false	カウント動作の再開に失敗した場合

出力先ファイル　R_PG_Timer_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のカウント動作を再開します。

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_〈プロジェクト名〉.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のカウント動作を一時停止
	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のカウント動作を再開
	R_PG_Timer_ResumeCount_CMT_U0_C0();
}
```

5.11.4 R_PG_Timer_GetCounterValue_CMT_U〈ユニット番号〉C〈チャネル番号〉

定義 bool R_PG_Timer_GetCounterValue_CMT_U〈ユニット番号〉C〈チャネル番号〉
(uint16_t * counter_val)
〈ユニット番号〉： 0, 1
〈チャネル番号〉： 0～3

概要 CMTのカウンタ値を取得

引数

uint16_t * counter_val	カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数 R_CMT_Read

詳細
- CMTのカウンタ値を取得します。

使用例 GUI上でCMT0を設定

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

uint16_t counter_val;

void func1(void)
{
	//CMT0を設定する
	R_PG_Timer_Start_CMT_U0_C0();
}

void func2(void)
{
	//CMT0のカウンタ値を取得
	R_PG_Timer_GetCounterValue_CMT_U0_C0( &counter_val );
}
```

5.11.5 R_PG_Timer_SetCounterValue_CMT_U〈ユニット番号〉_C〈チャネル番号〉

定義 bool R_PG_Timer_SetCounterValue_CMT_U〈ユニット番号〉_C〈チャネル番号〉
(uint16_t counter_val)
〈ユニット番号〉 : 0, 1
〈チャネル番号〉 : 0～3

概要 CMTのカウンタ値を設定

引数

uint16_t counter_val	カウンタに設定する値

戻り値

true	カウンタ値の設定に成功した場合
false	カウンタ値の設定に失敗した場合

出力先ファイル R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉 : 0, 1

使用RPDL関数 R_CMT_Control

詳細
- CMTのカウンタ値を設定します。

使用例 GUI上でCMT0を設定

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

void 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;
}
```

5.11.6 R_PG_Timer_StopModule_CMT_U〈ユニット番号〉

定義　bool R_PG_Timer_StopModule_CMT_U〈ユニット番号〉(void)
〈ユニット番号〉： 0, 1

概要　CMTのユニットを停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_Timer_CMT_U〈ユニット番号〉.c
〈ユニット番号〉： 0, 1

使用RPDL関数　R_CMT_Destroy

詳細

- CMTのユニットを停止し、モジュールストップ状態に移行します。ユニット単位で停止させます。ユニット0のCMT0とCMT1（ユニット1はCMT2とCMT3）が両方動作している場合、本関数を呼び出すとユニット内の2チャネルが停止します。片方のチャネルの動作だけを停止させる場合は、
  R_PG_Timer_HaltCount_CMT_U〈ユニット番号〉_C〈チャネル番号〉
  を使用してください。

使用例　GUI上で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_〈プロジェクト名〉.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();
}
```

5.12 ウォッチドッグタイマ（WDT）

5.12.1 R_PG_Timer_Start_WDT

定義　bool R_PG_Timer_Start_WDT (void)

概要　WDTを設定しカウント動作を開始

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_WDT.c

使用RPDL関数　R_WDT_Create

詳細

- WDTを初期設定し、カウント動作を開始します。
- GUI上で動作モードにインターバルタイマモードを指定した場合、本関数内でインターバルタイマ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「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例　GUI上で以下の通り設定した場合

- 動作モードをインターバルタイマモードに設定
- WdtIntFuncをインターバルタイマ割り込み通知関数名に指定

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

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Start_WDT();   //WDTの設定とカウント動作の開始
}

void WdtIntFunc(void)
{
    //WDTのカウンタオーバフロー時処理
}
```

5.12.2 R_PG_Timer_HaltCount_WDT

定義 bool R_PG_Timer_HaltCount_WDT (void)

概要 WDTのカウント動作を停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_Timer_WDT.c

使用RPDL関数 R_WDT_Control

詳細
- WDTのカウント動作を停止します。
- カウント動作を再開するにはR_PG_Timer_Start_WDT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
- 動作モードをインターバルタイマモードに設定
- WdtIntFuncをインターバルタイマ割り込み通知関数名に指定

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

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Start_WDT();   //WDTの設定とカウント動作の開始
}

void WdtIntFunc(void)
{
    R_PG_Timer_HaltCount_WDT();    //WDTのカウント動作停止

    //WDTのカウンタオーバフロー時処理

    R_PG_Timer_Start_WDT();   //WDTの設定とカウント動作の再開
}
```

5.12.3 R_PG_Timer_ResetCounter_WDT

定義 bool R_PG_Timer_ResetCounter_WDT(void)

概要 カウンタのリセット

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_Timer_WDT.c

使用RPDL関数 R_WDT_Control

詳細

- WDTのカウンタをリセットします

使用例

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

void func1(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Start_WDT();   //WDTの設定とカウント動作の開始
}

void func2(void)
{
    R_PG_Timer_ResetCounter_WDT();    //WDTのカウンタをリセット
}
```

5.12.4 R_PG_Timer_ClearOverflowFlag_WDT

定義 bool R_PG_Timer_ClearOverflowFlag_WDT (bool* ov)

概要 オーバフローフラグの取得とクリア

引数

bool* ov	オーバフローフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_WDT.c

使用RPDL関数 R_WDT_Read

詳細
- オーバフローフラグの取得し、クリアします。
- フラグを取得しない場合は引数に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
- 動作モードをインターバルタイマモードに設定
- インターバルタイマ割り込み優先レベルを0に設定

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

bool ov;

void func(void)
{
    R_PG_Clock_Set();   //クロックの設定
    R_PG_Timer_Start_WDT();   //WDTの設定とカウント動作の開始

    do{
        R_PG_Timer_ClearOverflowFlag_WDT( &ov ); //オーバフローフラグの取得
    }while( !ov );

    //オーバフロー発生時処理
}
```

5.13 独立ウォッチドッグタイマ（IWDT）

5.13.1 R_PG_Timer_Set_IWDT

定義　bool R_PG_Timer_Set_IWDT (void)

概要　IWDTの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_Timer_IWDT.c

使用RPDL関数　R_IWDT_Set

詳細

- IWDTを設定します。
- カウント動作はカウンタのリフレッシュにより開始します。本関数を呼び出した後、R_PG_Timer_RefreshCounter_IWDTを呼び出すことによりカウント動作が開始します。

使用例

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

uint8_t output_val;

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

    //IWDTの設定
    R_PG_Timer_Set_WDT();

    //カウンタのリフレッシュによりカウント動作開始
    R_PG_Timer_RefreshCounter_IWDT();
}

void func2(void)
{
    R_PG_Timer_RefreshCounter_IWDT();   //カウンタのリフレッシュ
}
```

5.13.2 R_PG_Timer_RefreshCounter_IWDT

定義 bool R_PG_Timer_RefreshCounter_IWDT (void)

概要 カウンタのリフレッシュ

引数 なし

戻り値

true	リフレッシュに成功した場合
false	リフレッシュに失敗した場合

出力先ファイル R_PG_Timer_IWDT.c

使用RPDL関数 R_IWDT_Control

詳細
- IWDTのカウンタをリフレッシュします。
- カウント動作を開始するにはR_PG_Timer_Set_IWDTによりIWDTを設定した後、本関数を呼び出してください。
- カウント動作開始後、本関数によりアンダフロー発生までにカウンタをリフレッシュしてください。

使用例 R_PG_Timer_Set_IWDTの使用例を参照してください。

5.13.3 R_PG_Timer_GetCounterValue_IWDT

定義 bool R_PG_Timer_GetCounterValue_IWDT(uint16_t * counter_val)

概要 カウンタ値の取得

引数

uint16_t * counter_val	カウンタ値の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_IWDT.c

使用RPDL関数 R_IWDT_Read

詳細

- IWDTのカウンタ値を取得します。
- 本関数内でアンダフローフラグはクリアされます。

使用例

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

uint16_t counter_val;

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

    //IWDTの設定
    R_PG_Timer_Set_WDT();

    //カウンタのリフレッシュによりカウント動作開始
    R_PG_Timer_RefreshCounter_IWDT();
}

void func2(void)
{
    R_PG_Timer_GetCounterValue_IWDT( &counter_val );

    if( counter_val < 0x1000){
        //カウンタのリフレッシュ
        R_PG_Timer_RefreshCounter_IWDT();   //カウンタのリフレッシュ
    }
}
```

5.13.4 R_PG_Timer_ClearUnderflowFlag_IWDT

定義 bool R_PG_Timer_ClearUnderflowFlag_IWDT(bool * un)

概要 アンダフローフラグの取得とクリア

引数

bool * un	アンダフローフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_Timer_IWDT.c

使用RPDL関数 R_IWDT_Read

詳細

- アンダフローフラグを取得し、クリアします。
- フラグを取得しない場合は引数に0を指定してください。

使用例

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

bool un;

void func(void)
{
	R_PG_Timer_ClearUnderflowFlag_IWDT ( &un );
	if(un){
		//アンダフロー発生によるリセット後処理
	}
}
```

5.14 シリアルコミュニケーションインタフェース（SCIb）

5.14.1 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*〉: 0～2

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*〉: 0～2

使用RPDL関数　R_SCI_Create, R_SCI_Set

詳細

- SCIチャネルのモジュールストップ状態を解除して初期設定し、使用する端子の入出力方向、入力バッファを設定します。
- 本関数を使用する場合、あらかじめR_PG_Clock_Setによりクロックを設定してください。
- 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「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例　SCI0をGUI上で設定

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

void func(void)
{
    R_PG_Clock_Set();      //クロックの設定
    R_PG_SCI_Set_C0();     //SCI0を設定
}
```

5.14.2 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*〉: 0～2

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*〉: 0～2

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「5.21 通知関数に関する注意事項」の内容に注意してください。
- R_PG_SCI_GetSentDataCount_C〈*チャネル番号*〉により送信済みデータ数を取得することができます。R_PG_SCI_StopCommunication_C〈*チャネル番号*〉により、最終バイトの送信完了を待たずに送信を中断することができます。
- 65536バイトのデータが送信されると、0番目のデータに戻ります。

使用例　GUI上でSCI0の送信終了通知関数名にSci0TrFuncを指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
uint8_t data[255];
void func(void)
{
    R_PG_Clock_Set();           //クロックの設定
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_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();
}
```

5.14.3 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*〉: 0～2

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*〉: 0～2

使用RPDL関数

R_SCI_Send

詳細

- シリアルデータを送信します。
- 本関数はGUI上でデータ送信方法に“全データの送信完了を関数呼び出しで通知する”以外が選択されている場合に出力されます。
- 指定した数のデータ送信完了まで関数内でウェイトします。
- 65536バイトのデータが送信されると、0番目のデータに戻ります。

使用例

GUI上でSCI0のデータ送信方法に“全データの送信完了まで待つ”を選択

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

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_SendAllData_C0(data, 255);          //255バイトのデータを送信する
    R_PG_SCI_StopModule_C0();          //SCI0を停止
}
```

5.14.4 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*〉: 0～2

概要 シリアルデータの送信数取得

生成条件 GUI上でSCIチャネルの送信機能を設定し、データ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択

引数

uint16_t * count	現在の送信処理で送信されたデータ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～2

使用RPDL関数 R_SCI_GetStatus

詳細
- GUI上でデータ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数により送信済みデータ数を取得することができます。

使用例 GUI上でSCI0の送信機能を設定
送信終了通知関数名にSci0TrFuncを指定

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

uint16_t data[255];

void func(void)
{
    R_PG_Clock_Set();           //クロックの設定
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_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_StopCommunication_C0();   //送信を中断
    }
}
```

5.14.5 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*〉: 0～2

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*〉: 0～2

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「5.21 通知関数に関する注意事項」の内容に注意してください。
- R_PG_SCI_GetReceivedDataCount_C 〈*チャネル番号*〉により受信済みデータ数を取得することができます。R_PG_SCI_StopCommunication_C〈*チャネル番号*〉 により、最終バイトの受信完了を待たずに受信を中断することができます。
- 最大受信データ数は65535です。

使用例

・GUI上でSCI0の受信機能を設定

・受信終了通知関数名にSci0ReFuncを指定

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

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();           //クロックの設定
    R_PG_SCI_Set_C0();          //SCI0を設定
    R_PG_SCI_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();
}
```

5.14.6 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*〉: 0～2

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*〉: 0～2

使用RPDL関数

R_SCI_Receive

詳細

- シリアルデータを受信します。
- 本関数はGUI上でデータ受信方法に “全データの受信完了を関数呼び出しで通知する” 以外が選択されている場合に出力されます。
- 本関数は指定した数のデータ受信完了までウェイトします。
- 最大受信データ数は65535です。

使用例

GUI上でSCI0の受信機能を設定

データ受信方法に“全データの受信完了まで待つ”を選択

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

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_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
}
```

5.14.7 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〉: 0～2

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〉: 0～2

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_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
uint8_t data[255];
void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_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
    }
}
```

5.14.8 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*〉: 0～2

概要　シリアルデータの受信数取得

生成条件　GUI上でSCIチャネルの受信機能が設定され、データ受信方法に“全データの受信完了を関数呼び出しで通知する”

引数

uint16_t * count	現在の受信処理で受信したデータ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～2

使用RPDL関数　R_SCI_GetStatus

詳細
- GUI上でデータ受信方法に“全データの受信完了を関数呼び出しで通知する”が選択されている場合、本関数により受信済みデータ数を取得することができます。

使用例　GUI上でSCI0の受信機能を設定
受信終了通知関数名にSci0ReFuncを指定

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

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_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
    }
}
```

5.14.9 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〉: 0～2

概要 シリアル受信エラーフラグの取得

生成条件 GUI上でSCIチャネルの受信機能を設定

引数

bool * parity	パリティエラーフラグ格納先
bool * framing	フレーミングエラーフラグ格納先
bool * overrun	オーバランエラーフラグ格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_SCI_C〈チャネル番号〉.c
〈チャネル番号〉: 0～2

使用RPDL関数 R_SCI_GetStatus

詳細
- 受信エラーフラグを取得します。
- 取得しないフラグは0を設定してください。
- 検出したエラーのフラグには1が設定されます。

使用例 GUI上でSCI0の受信機能を設定
受信終了通知関数名にSci0ReFuncを指定

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

uint8_t data[255];

//受信エラーフラグ
bool parity;
bool framing;
bool overrun;

void func(void)
{
	R_PG_Clock_Set();		//クロックの設定
	R_PG_SCI_Set_C0();		//SCI0を設定
	R_PG_SCI_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 );
}
```

5.14.10R_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*〉: 0～2

概要　シリアルデータ送信状態の取得

生成条件　GUI上でSCIチャネルの送信機能を設定

引数

bool * complete	送信終了フラグ格納先 (0: 送信中　1: 送信終了)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～2

使用RPDL関数　R_SCI_GetStatus

詳細
- シリアルデータの送信状態を取得します。

使用例

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

bool complete;

void func(void)
{
	//送信状態の取得
	R_PG_SCI_GetTransmitStatus_C0( &complete );
}
```

5.14.11 R_PG_SCI_SendTargetStationID_C〈*チャネル番号*〉

定義

bool R_PG_SCI_SendTargetStationID_C〈*チャネル番号*〉(uint8_t id)

〈*チャネル番号*〉: 0～2

概要

データ送信先IDの送信

生成条件

- GUI上でSCIチャネルの送信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定

引数

uint8_t id	送信するIDコード (0～255)

戻り値

True	送信に成功した場合
False	送信に失敗した場合

出力先ファイル

R_PG_SCI_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0～2

使用RPDL関数

R_SCI_Send

詳細

- マルチプロセッサモードのID送信サイクルを生成し、データ送信先の受信局IDコードを出力します。
- 本関数はID送信サイクル終了までウェイトします。

使用例

GUI上で以下の通り設定した場合

- SCI2チャネルの送信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定
- データ送信方法に“全データの送信完了まで待つ”を選択

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
uint8_t data[10] = "ABCDEFGHIJ";

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C2();             //SCI2を設定
    R_PG_SCI_SendTargetStationID_C2( 5 );      //IDコードの送信 (ID:5)
    R_PG_SCI_SendAllData_C2( data, 10 );       //データの送信
}
```

5.14.12 R_PG_SCI_ReceiveStationID_C〈*チャネル番号*〉

定義　bool R_PG_SCI_ReceiveStationID_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉: 0～2

概要　自局IDと一致するIDコードの受信

生成条件
- GUI上でSCIチャネルの受信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定

引数　なし

戻り値

true	受信に成功した場合
false	受信に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～2

使用RPDL関数　R_SCI_Receive

詳細
- 本関数は自局のIDと一致するIDコードを受信するまでウェイトします。

使用例　GUI上で以下の通り設定した場合
- SCI0チャネルの受信機能を設定
- 調歩同期式通信方式でマルチプロセッサ通信機能を有効に設定
- データ受信方法に“全データの受信完了まで待つ”を選択

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

uint8_t data[10];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0の設定
    R_PG_SCI_ReceiveStationID_C0();    //IDの受信を待つ
    R_PG_SCI_StartReceiving_C0( data, 10 );    //受信開始
}
```

5.14.13R_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〉: 0～2

概要　シリアルI/Oチャネルの停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_SCI_C〈チャネル番号〉.c
〈チャネル番号〉: 0～2

使用RPDL関数　R_SCI_Destroy

詳細
- SCIのチャネルを停止し、モジュールストップ状態に移行します。

使用例
GUI上で以下の通り設定した場合
- GUI上でSCI0の受信機能を設定
- データ受信方法に“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_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

uint8_t data[255];

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_SCI_Set_C0();             //SCI0を設定
    R_PG_SCI_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
}
```

5.14.14 R_PG_SCI_ControlClockOutput_C〈*チャネル番号*〉

定義　bool R_PG_SCI_ControlClockOutput_C〈*チャネル番号*〉(bool output_enable)
〈*チャネル番号*〉: 0～2

概要　SCKn端子出力の切り替え

生成条件
- スマートカードインターフェース通信方式でGSMモードを有効に設定
- SCKns端子機能をLowレベル出力固定または、Highレベル出力固定に設定

引数

bool output_enable	SCKn端子の出力（1:クロック出力 0:出力固定）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_SCI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0～2

使用RPDL関数　R_SCI_Control

詳細
- SCKn端子の出力を、クロック出力もしくは、出力固定に切り替えます。

使用例　GUI上で以下の通り設定した場合
- GUI上でSCI0の通信モードをスマートカードインターフェースモードに設定
- スマートカードインターフェースモード設定のGSMモードを有効に設定
- 端子機能のSCKn端子機能をLowレベル出力固定に設定

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

void func(void)
{
    R_PG_Clock_Set();                          //クロックの設定
    R_PG_SCI_Set_C0();                         //SCI0を設定
    R_PG_SCI_ControlClockOutput_C0( true ); //SCK0端子出力をクロック出力に設定
}
```

5.15 CRC演算器（CRC）

5.15.1 R_PG_CRC_Set

定義　bool R_PG_CRC_Set(void)

概要　CRC演算器の設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_CRC.c

使用RPDL関数　R_CRC_Create

詳細

- CRC演算器のモジュールストップ状態を解除して初期設定します。

使用例

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

uint16_t result;

void func(void)
{
    R_PG_CRC_Set();   //CRC演算器の設定
    R_PG_CRC_InputData(0xf0);    //ペイロードデータ入力
    R_PG_CRC_InputData(0x8f);    //前半チェックサム入力
    R_PG_CRC_InputData(0x7f);    //後半チェックサム入力
    R_PG_CRC_GetResult (&result);   //演算結果取得
    R_PG_CRC_StopModule();   //CRC演算器停止
}
```

5.15.2 R_PG_CRC_InputData

定義 bool R_PG_CRC_InputData (uint8_t data)

概要 データの入力

引数

uint8_t data	入力するデータ

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Write

詳細
- CRCデータ入力レジスタにデータを設定します。

使用例 R_PG_CRC_Setの使用例を参照してください。

5.15.3 R_PG_CRC_GetResult

定義 bool R_PG_CRC_GetResult (uint16_t * result)

概要 演算結果の取得

引数

uint16_t * result	演算結果の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Read

詳細

- 演算結果を取得します。

使用例 R_PG_CRC_Setの使用例を参照してください。

5.15.4 R_PG_CRC_StopModule

定義 bool R_PG_CRC_StopModule(void)

概要 CRC演算器の停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_CRC.c

使用RPDL関数 R_CRC_Destroy

詳細
- CRC演算器を停止し、モジュールストップ状態に移行します。

使用例 R_PG_CRC_Setの使用例を参照してください。

5.16 I2Cバスインタフェース（RIIC）

5.16.1 R_PG_I2C_Set_C〈*チャネル番号*〉

定義　bool R_PG_I2C_Set_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉: 0

概要　I2Cバスインタフェースチャネルの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_I2C_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0

使用RPDL関数　R_IIC_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。
  本関数を使用する場合、あらかじめR_PG_Clock_Setによりクロックを設定してください。
-

使用例　GUI上でRIIC0を設定

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

void func(void)
{
    R_PG_Clock_Set();       //クロックの設定
    R_PG_I2C_Set_C0();      //RIIC0を設定
}
```

5.16.2 R_PG_I2C_MasterReceive_C〈チャネル番号〉

定義 bool R_PG_I2C_MasterReceive_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0

概要 マスタのデータ受信

生成条件 マスタ機能を使用

引数

bool addr_10bit	スレーブアドレス幅（0:7ビット, 1:10ビット）
uint16_t slave	スレーブアドレス
uint8_t* data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_MasterReceive

詳細

- スレーブからデータを読み出します。指定した数のデータを受信するとSTOP条件を生成し転送を終了します。
- GUI上でマスタ受信方法に“全データの受信完了まで待つ”が選択されている場合、本関数は転送終了までウェイトします。GUI上でマスタ受信方法に“全データの受信完了を関数呼び出しで通知する”が選択されている場合、本関数はすぐにリターンし、転送終了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。
- 通信の最初にSTART条件が生成されます。前回の転送でSTOP条件が生成されていない場合は反復START条件が生成されます。
- スレーブアドレスは、7ビットアドレスの場合は指定した値の7～1ビットが出力されます。10ビットアドレスの場合は10～1ビットが出力されます。
- R_PG_I2C_GetReceivedDataCount_C〈チャネル番号〉により受信済みデータ数を取得することができます。
- 10ビットアドレスを使用する場合、GUI上のマスタ受信方法は“全データの受信完了を関数で通知する”以外を選択してください。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ受信方法に“全データの受信完了まで待つ”を選択

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

//受信データの格納先
uint8_t iic_data[10];

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      // スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,      //受信データの格納先アドレス
        10      //受信データ数
    );

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.16.3 R_PG_I2C_MasterReceiveLast_C〈*チャネル番号*〉

定義

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

概要

マスタのデータ受信終了

生成条件

- マスタ機能を使用
- GUI上でマスタ受信方法にDTCによる転送を選択

引数

uint8_t* data	受信したデータの格納先のアドレス

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_I2C_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0

使用RPDL関数

R_IIC_MasterReceiveLast

詳細

- 本関数はGUI上で[マスタ受信方法]に[受信データを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。
- DTCの転送終了時に受信を終了する場合は、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DTC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DTCで転送する]を指定。
- DTCの設定で以下の設定を作成。
  転送開始要因 ： RXI0(RIIC0受信データフル割り込み)
  転送データバイトサイズ ： 1byte
  転送回数 ： 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
  転送元アドレス更新モード ： 固定

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

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //DTCを設定し、ICRXI0をトリガとする転送を設定
    R_PG_DTC_Set();
    R_PG_DTC_Set_ICRXI0();

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

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        PDL_NO_PTR, //受信データ格納先 (DTC転送の場合はPDL_NO_PTR)
        0       //受信データ数 (DTC転送の場合は0)
    );
}

void func2()
{
    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 );
}
```

5.16.4 R_PG_I2C_MasterSend_C〈チャネル番号〉

定義 bool R_PG_I2C_MasterSend_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0

概要 マスタのデータ送信

生成条件 マスタ機能を使用

引数

bool addr_10bit	スレーブアドレス幅（0:7ビット, 1:10ビット）
uint16_t slave	スレーブアドレス
uint8_t* data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_MasterSend

詳細

- スレーブにデータを送信します。指定した数のデータを送信するとSTOP条件を生成し転送を終了します。
- GUI上でマスタ送信方法に“全データの送信完了まで待つ”が選択されている場合、本関数は転送終了または他のイベント検出までウェイトします。検出したイベントはR_PG_I2C_GetEvent_C〈チャネル番号〉により取得できます。GUI上でマスタ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数はすぐにリターンし、転送終了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については「5.21 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10～1ビットが出力されます。
- R_PG_I2C_GetSentDataCount_C〈チャネル番号〉により送信済みデータ数を取得することができます。
- 10ビットアドレスを使用する場合、GUI上のマスタ送信方法は“全データの送信完了を関数で通知する”以外に設定してください。

使用例　　GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了まで待つ”を選択

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

//送信データの格納先
uint8_t iic_data[10];

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSend_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,      //送信データの格納先アドレス
        10      //送信データ数
    );

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.16.5 R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉

定義

R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉
(bool addr_10bit, uint16_t slave, uint8_t* data, uint16_t count)
〈チャネル番号〉: 0

概要

マスタのデータ送信（STOP条件無し）

生成条件

マスタ機能を使用

引数

bool addr_10bit	スレーブアドレス幅（0:7ビット, 1:10ビット）
uint16_t slave	スレーブアドレス
uint8_t* data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数

R_IIC_MasterSend

詳細

- スレーブにデータを送信します。転送が終了してもSTOP条件を生成しません。本関数によるデータの送信後再び転送を開始した場合は反復START条件が生成されます。STOP条件を生成するにはR_PG_I2C_GenerateStopCondition_C〈チャネル番号〉を呼び出してください。
- GUI上でマスタ送信方法に“全データの送信完了まで待つ”が選択されている場合、本関数は転送終了または他のイベント検出までウェイトします。検出したイベントはR_PG_I2C_GetEvent_C〈チャネル番号〉により取得できます。GUI上でマスタ送信方法に“全データの送信完了を関数呼び出しで通知する”が選択されている場合、本関数はすぐにリターンし、転送終了時に指定した名前の関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については「5.21 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10～1ビットが出力されます。
- R_PG_I2C_GetSentDataCount_C〈チャネル番号〉により送信済みデータ数を取得することができます。
- 10ビットアドレスを使用する場合、GUI上のマスタ送信方法は“全データの送信完了を関数で通知する”以外に設定してください。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択
- マスタ送信の通知関数名に IIC0MasterTrFunc を指定

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

//送信データの格納先
uint8_t iic_data[10];

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSendWithoutStop_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,      //送信データの格納先アドレス
        10      //送信データ数
    );
}

void IIC0MasterTrFunc(void)
{
    //STOP条件を生成
    R_PG_I2C_GenerateStopCondition_C0();

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.16.6 R_PG_I2C_GenerateStopCondition_C〈チャネル番号〉

定義 R_PG_I2C_GenerateStopCondition_C〈チャネル番号〉(void)
〈チャネル番号〉: 0

概要 マスタのSTOP条件生成

生成条件 マスタ機能を使用

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_Control

詳細
- R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉 により転送を開始した場合、STOP条件を生成することができます。

使用例 R_PG_I2C_MasterSendWithoutStop_C〈チャネル番号〉の使用例を参照してください。

5.16.7 R_PG_I2C_GetBusState_C〈*チャネル番号*〉

定義 R_PG_I2C_GetBusState_C〈*チャネル番号*〉(bool *busy)

〈*チャネル番号*〉: 0

概要 バス状態の取得

生成条件 マスタ機能を使用

引数

bool *busy	バスビジー検出フラグの格納先 バスビジーフラグ 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）

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細

- バスビジー検出フラグを取得します。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
//送信データの格納先
uint8_t iic_data[10];
//バスビジー検出フラグの格納先
bool busy;
void func(void)
{
	//クロックの設定
	R_PG_Clock_Set();
	//RIIC0を設定
	R_PG_I2C_Set_C0();
	//バスフリー状態を待つ
	do{
		R_PG_I2C_GetBusState_C0( & busy );
	} while( busy );
	//マスタ送信
	R_PG_I2C_MasterSend_C0(
		0,	//スレーブアドレスフォーマット
		6,	//スレーブアドレス
		iic_data,	//送信データの格納先アドレス
		10	//送信データ数
	);
}
```

5.16.8 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉

定義 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉(uint8_t *data, uint16_t count)
〈チャネル番号〉: 0

概要 スレーブのバス監視

生成条件 スレーブ機能を使用

引数

uint8_t *data	受信したデータの格納先の先頭のアドレス
uint16_t count	受信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_SlaveMonitor

詳細

- マスタからのアクセスを監視します。
- GUI上でスレーブモニタ方法に“全データの受信完了、スレーブリード要求、ストップ条件検出を関数呼び出しで通知する”が選択されている場合、マスタからの読み出し要求またはマスタからの受信後にSTOP条件を検出すると、指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void 〈通知関数名〉(void)
  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。
  GUI上でスレーブモニタ方法に“全データの受信完了、スレーブリード要求、ストップ条件検出まで待つ”が選択されている場合、本関数はマスタからの読み出し要求またはマスタからの受信後にSTOP条件を検出するまでウェイトします。
- マスタからデータが送信された場合は指定した領域に受信データが格納されます。受信データ量が格納領域を上回らないよう、受信データ数設定してください。
  指定したデータ数を上回るデータがマスタから送信された場合はNACKを生成します。
- R_PG_I2C_GetTR_C〈チャネル番号〉 により送信/受信モードを取得することができます。マスタから送信(読み出し)が要求された場合、R_PG_I2C_SlaveSend_C〈チャネル番号〉 によりデータを送信できます。
- 検出したスレーブアドレスを取得するには R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉 を使用してください。START条件、STOP条件等の検出イベントを取得するには R_PG_I2C_GetEvent_C〈チャネル番号〉 を使用してください。
- 10ビットアドレスを使用する場合、GUI上のスレーブモニタ方法は “全データの受信完了、スレーブリード要求、ストップ条件検出を関数呼び出しで通知する” 以外に設定してください。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をスレーブとして使用
  スレーブモニタの通知関数名に IIC0SlaveFunc を指定

```
//この関数を使用するには"R_PG_<プロジェクト名>.h"をインクルードしてください
#include "R_PG_default.h"
//受信データの格納先
uint8_t iic_data_re[10];
//送信データの格納先(スレーブアドレス0)
uint8_t iic_data_tr_0[10];
//送信データの格納先(スレーブアドレス1)
uint8_t iic_data_tr_1[10];
void func(void)
{
    //クロックの設定
    R_PG_Clock_Set();
    //RIIC0を設定
    R_PG_I2C_Set_C0();
    //スレーブモニタ
    R_PG_I2C_SlaveMonitor_C0(
        iic_data_re,    //受信データの格納先アドレス
        10      //受信データ数
    );
}
void IIC0SlaveFunc(void)
{
    bool transmit, start, stop;
    bool addr0, addr1;
    //イベントを取得する
    R_PG_I2C_GetEvent_C0(0, &stop, &start, 0, 0);
    //送受信モードを取得する
    R_PG_I2C_GetTR_C0(&transmit);
    //検出アドレスを取得する
    R_PG_I2C_GetDetectedAddress_C0(&addr0, &addr1, 0, 0, 0, 0);
    if(start && transmit && address0){
        //マスタにデータを送信
        R_PG_I2C_SlaveSend_C(
            iic_data_tr_0,
            10
        );
    }
    else if(start && read && address1){
        //マスタにデータを送信
        R_PG_I2C_SlaveSend_C(
            iic_data_tr_1,
            10
        );
    }
}
```

5.16.9 R_PG_I2C_SlaveSend_C〈チャネル番号〉

定義 R_PG_I2C_SlaveSend_C〈チャネル番号〉(uint8_t *data, uint16_t count)
〈チャネル番号〉: 0

概要 スレーブのデータ送信

生成条件 スレーブ機能を使用

引数

uint8_t *data	送信するデータの格納先の先頭のアドレス
uint16_t count	送信するデータ数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_SlaveSend

詳細
- マスタにデータを送信します。
- マスタが送信データ数を上回るデータを要求する場合、先頭のアドレスに戻って送信します。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉 の使用例を参照してください。

5.16.10 R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉

定義 R_PG_I2C_GetDetectedAddress_C〈チャネル番号〉
(bool *addr0, bool *addr1, bool *addr2, bool *general, bool *device, bool *host)
〈チャネル番号〉: 0

概要 検出したスレーブアドレスの取得

生成条件 スレーブ機能を使用

引数

bool *addr0	スレーブアドレス0検出フラグ格納先
bool *addr1	スレーブアドレス1検出フラグ格納先
bool *addr2	スレーブアドレス2検出フラグ格納先
bool *general	ジェネラルコールアドレス検出フラグ格納先
bool *device	デバイスID検出フラグ格納先
bool *host	ホストアドレス検出フラグ格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細
- 検出したアドレスを取得します。
- 取得しないフラグは0を設定してください。
- 検出したアドレスのフラグには1が設定されます。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉 の使用例を参照してください。

5.16.11 R_PG_I2C_GetTR_C〈チャネル番号〉

定義 R_PG_I2C_GetTR_C〈チャネル番号〉(bool * transmit)

〈チャネル番号〉: 0

概要 送信/受信モードの取得

生成条件 スレーブ機能を使用

引数

bool * transmit	送信/受信モードフラグの格納先 送信/受信モードフラグ 0:受信モード 1:送信モード

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細

- 送信/受信モードを取得します。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉 の使用例を参照してください。

5.16.12 R_PG_I2C_GetEvent_C〈チャネル番号〉

定義 R_PG_I2C_GetEvent_C〈チャネル番号〉
(bool *nack, bool *stop, bool *start, bool *lost, bool *timeout)
〈チャネル番号〉: 0

概要 検出イベントの取得

引数

bool *nack	NACK検出フラグ格納先
bool *stop	STOP条件検出フラグ格納先
bool *start	START条件検出フラグ格納先
bool *lost	アービトレーションロスト検出フラグ格納先
bool *timeout	タイムアウト検出フラグ格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数 R_IIC_GetStatus

詳細
- 検出したイベントを取得します。
- 取得しないフラグは0を設定してください。
- 検出したイベントのフラグには1が設定されます。

使用例 R_PG_I2C_SlaveMonitor_C〈チャネル番号〉 の使用例を参照してください。

5.16.13R_PG_I2C_GetReceivedDataCount_C〈チャネル番号〉

定義　bool R_PG_I2C_GetReceivedDataCount_C〈チャネル番号〉(uint16_t *count)
〈チャネル番号〉: 0

概要　受信済みデータ数の取得

引数

uint16_t *count	受信データ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_I2C_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数　R_IIC_GetStatus

詳細
- 現在の転送で受信したデータ数を取得します。

使用例　GUI上で以下の通り設定した場合
- RIIC0 をマスタとして使用
- マスタ受信方法に“全データの受信完了を関数呼び出しで通知する”を選択

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

//受信データの格納先
uint8_t iic_data[256];

//受信データ数の格納先
uint16_t count;

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,       //受信データの格納先アドレス
        256     //受信データ数
    );

    //64バイト受信するまで待つ
    do{
        R_PG_I2C_GetReceivedDataCount_C0( &count );
    } while( count < 64 );
}
```

5.16.14 R_PG_I2C_GetSentDataCount_C〈*チャネル番号*〉

定義　bool R_PG_I2C_GetSentDataCount_C〈*チャネル番号*〉(uint16_t *count)

〈*チャネル番号*〉: 0

概要　送信済みデータ数の取得

引数

uint16_t *count	送信データ数の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル　R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数　R_IIC_GetStatus

詳細
- 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_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

//送信データの格納先
uint8_t iic_data[256];

//送信データ数の格納先
uint16_t count;

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ送信
    R_PG_I2C_MasterSend_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,      //受信データの格納先アドレス
        256      //受信データ数
    );

    //64バイト送信するまで待つ
    do{
        R_PG_I2C_GetSentDataCount_C0( &count );
    } while( count < 64 );
}
```

5.16.15 R_PG_I2C_Reset_C〈*チャネル番号*〉

定義　R_PG_I2C_Reset_C〈*チャネル番号*〉(void)

〈*チャネル番号*〉: 0

概要　バスのリセット

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_I2C_C〈*チャネル番号*〉.c

〈*チャネル番号*〉: 0

使用RPDL関数　R_IIC_Control

詳細
- モジュールをリセットします。
- 設定は維持されます。

使用例　GUI上で以下の通り設定した場合
- RIIC0 をマスタとして使用
- マスタ送信方法に“全データの送信完了を関数呼び出しで通知する”を選択
- マスタ送信の通知関数名に IIC0MasterTrFunc を指定

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

//送信データの格納先
uint8_t iic_data[10];

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

	//RIIC0を設定
	R_PG_I2C_Set_C0();

	//マスタ送信
	R_PG_I2C_MasterSend_C0(
		0,	//スレーブアドレスフォーマット
		6,	//スレーブアドレス
		iic_data,	//送信データの格納先アドレス
		10	//送信データ数
	);
}

void IIC0MasterTrFunc(void)
{
	if( error ){
		R_PG_I2C_Reset_C0();
	}
}
```

5.16.16 R_PG_I2C_StopModule_C〈チャネル番号〉

定義 bool R_PG_I2C_StopModule_C〈チャネル番号〉(void)

〈チャネル番号〉: 0

概要 I2Cバスインタフェースチャネルの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル R_PG_I2C_C〈チャネル番号〉.c

〈チャネル番号〉: 0

使用RPDL関数 R_IIC_Destroy

詳細

- I2Cバスインタフェースチャネルを停止し、モジュールストップ状態に移行します。

使用例 GUI上で以下の通り設定した場合

- RIIC0 をマスタとして使用
- マスタ受信方法に“全データの受信完了まで待つ”を選択

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

//受信データの格納先
uint8_t iic_data[10];

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

    //RIIC0を設定
    R_PG_I2C_Set_C0();

    //マスタ受信
    R_PG_I2C_MasterReceive_C0(
        0,      //スレーブアドレスフォーマット
        6,      //スレーブアドレス
        iic_data,       //受信データの格納先アドレス
        10      //受信データ数
    );

    //RIIC0を停止
    R_PG_I2C_StopModule_C0();
}
```

5.17 シリアルペリフェラルインタフェース（RSPI）

5.17.1 R_PG_RSPI_Set_C〈*チャネル番号*〉

定義　bool R_PG_RSPI_Set_C〈*チャネル番号*〉(void)
〈*チャネル番号*〉: 0

概要　RSPIチャネルの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数　R_SPI_Create

詳細
- シリアルペリフェラルインタフェースチャネルのモジュールストップ状態を解除して初期設定し、使用する端子を設定します。
- 本関数を使用する場合、あらかじめR_PG_Clock_Setによりクロックを設定してください。
- 本関数でコマンドは設定されません。コマンドを設定するには R_PG_RSPI_SetCommand_C〈*チャネル番号*〉を呼び出してください。

使用例

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

void func(void)
{
    R_PG_Clock_Set();       //クロック発生回路の設定
    R_PG_RSPI_Set_C0();     //RSPI0の設定
    R_PG_RSPI_SetCommand_C0();      //コマンドの設定
}
```

5.17.2 R_PG_RSPI_SetCommand_C〈チャネル番号〉

定義　bool R_PG_RSPI_SetCommand_C〈チャネル番号〉(void)
〈チャネル番号〉: 0

概要　コマンドの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数　R_SPI_Command

詳細
- RPSIコマンドレジスタを設定します。
- GUI上で設定した最大8コマンドを全て設定します。

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

5.17.3 R_PG_RSPI_StartTransfer_C〈*チャネル番号*〉

定義

送信および受信機能(全二重同期式シリアル通信機能)選択時

bool R_PG_RSPI_StartTransfer_C〈*チャネル番号*〉
(uint32_t * tx_start, uint32_t * rx_start, uint16_t sequence_loop_count)
〈*チャネル番号*〉: 0

送信機能のみ選択時

bool R_PG_RSPI_StartTransfer_C〈*チャネル番号*〉
(uint32_t * tx_start, uint16_t sequence_loop_count)
〈*チャネル番号*〉: 0

概要

データの転送開始

生成条件

転送方法に“転送完了、エラー検出を関数呼び出しで通知する”を選択

引数

uint32_t * tx_start	送信するデータの先頭のアドレス
uint32_t * rx_start	受信したデータの格納先の先頭のアドレス
uint16_t sequence_loop_count	コマンドシーケンスの繰り返し回数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_RSPI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0

使用RPDL関数

R_SPI_Transfer

詳細

- データの転送を開始します。
- 本関数はGUI上で転送方法に“転送完了、エラー検出を関数呼び出しで通知する”が選択されている場合に出力されます。
- 本関数はすぐにリターンし、エラー検出時または指定した回数のコマンドシーケンス完了時に、指定した名前の通知関数が呼ばれます。通知関数は次の定義で作成してください。
  void 〈*通知関数名*〉(void)
  通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例

GUI上で以下の通り設定した場合

- RSPIをSPI動作マスタモードで設定
- 転送方法に“転送完了、エラー検出を関数呼び出しで通知する”を指定
- 通知関数名にrsi0_int_funcを指定
- コマンド数:1 フレーム数:4
- コマンド0のビット長:8

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

uint32_t  tx_data[4] = { 0x11, 0x22, 0x33, 0x44 };
uint32_t  rx_data[4] = { 0x00, 0x00, 0x00, 0x00 };
bool   over_run,   mode_fault,   parity_error;

void func(void)
{
      R_PG_Clock_Set();       //クロック発生回路の設定
      R_PG_RSPI_Set_C0();       //RSPI0の設定
      R_PG_RSPI_SetCommand_C0();       //コマンドの設定
      R_PG_RSPI_StartTransfer_C0( tx_data, rx_data, 1 ); //8bit*4フレーム転送
}

void rsi0_int_func (void)
{
      R_PG_RSPI_GetError_C0 ( &over_run, &mode_fault, &parity_error );   //エラー取得
      if( over_run || mode_fault || parity_error ){
            //エラー検出時処理
      }
      R_PG_RSPI_StopModule_C0();
}
```

5.17.4 R_PG_RSPI_TransferAllData_C〈*チャネル番号*〉

定義

送信および受信機能(全二重同期式シリアル通信機能)選択時

bool R_PG_RSPI_TransferAllData_C〈*チャネル番号*〉
(uint32_t * tx_start, uint32_t * rx_start, uint16_t sequence_loop_count)
〈*チャネル番号*〉: 0

送信機能のみ選択時

bool R_PG_RSPI_TransferAllData_C〈*チャネル番号*〉
(uint32_t * tx_start, uint16_t sequence_loop_count)
〈*チャネル番号*〉: 0

転送方法にDTCによる転送を選択した場合

bool R_PG_RSPI_TransferAllData_C〈*チャネル番号*〉
(uint16_t sequence_loop_count)
〈*チャネル番号*〉: 0

概要

全データの転送

生成条件

転送方法に“転送完了、エラー検出を関数呼び出しで通知する”以外を選択

引数

uint32_t * tx_start	送信するデータの先頭のアドレス
uint32_t * rx_start	受信したデータの格納先の先頭のアドレス
uint16_t sequence_loop_count	コマンドシーケンスの繰り返し回数

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル

R_PG_RSPI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0

使用RPDL関数

R_SPI_Transfer

詳細

- 全データを転送します。
- 本関数はGUI上で転送方法に“転送完了、エラー検出を関数呼び出しで通知する”以外が選択されている場合に出力されます。
- 本関数はエラー検出または指定した回数のコマンドシーケンス完了までウェイトします。

使用例

GUI上で以下の通り設定した場合

- RSPIをSPI動作マスタモードで設定
- 転送方法に“転送完了まで待つ”を指定
- 通知関数名にrsi0_int_funcを指定
- コマンド数:1 フレーム数:4
- コマンド0のビット長:8

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

uint32_t  tx_data[4] = { 0x11, 0x22, 0x33, 0x44 };
uint32_t  rx_data[4] = { 0x00, 0x00, 0x00, 0x00 };
bool   over_run,   mode_fault,   parity_error;

void func(void)
{
      R_PG_Clock_Set();       //クロック発生回路の設定
      R_PG_RSPI_Set_C0();       //RSPI0の設定
      R_PG_RSPI_SetCommand_C0();       //コマンドの設定
      R_PG_RSPI_TransferAllData_C0( tx_data, rx_data, 1 ); //8bit*4フレーム転送

      R_PG_RSPI_GetError_C0 ( &over_run, &mode_fault, &parity_error );    //エラー取得
      if( over_run || mode_fault || parity_error ){
            //エラー検出時処理
      }
      R_PG_RSPI_StopModule_C0();
}
```

5.17.5 R_PG_RSPI_GetStatus_C〈*チャネル番号*〉

定義

bool R_PG_RSPI_GetStatus_C〈*チャネル番号*〉
(bool * idle, bool * receive_full, bool * transmit_empty)
〈*チャネル番号*〉: 0

概要

転送状態の取得

引数

bool * idle	アイドルフラグの格納先 (0:アイドル状態 1:転送状態)
bool * receive_full	受信バッファフルフラグの格納先 (0:受信バッファにデータなし 1:受信バッファにデータあり)
bool * transmit_empty	送信バッファエンプティフラグの格納先 (0:送信バッファにデータあり 1:送信バッファにデータなし)

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_RSPI_C〈*チャネル番号*〉.c
〈*チャネル番号*〉: 0

使用RPDL関数

R_SPI_GetStatus

詳細

- データの転送状態を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。
- 本関数内でエラーフラグ(オーバランエラーフラグ、モードフォルトエラーフラグ、パリティエラーフラグ)はクリアされます。エラーフラグを取得する場合は本関数を呼び出す前にR_PG_RSPI_GetError_C〈*チャネル番号*〉を呼び出してください。

使用例

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

bool idle;

void func(void)
{
    do{
        //アイドルフラグの取得
        R_PG_RSPI_GetStatus_C0( & idle, 0, 0 );
    }while( idle );
}
```

5.17.6 R_PG_RSPI_GetError_C〈チャネル番号〉

定義

bool R_PG_RSPI_GetError_C〈チャネル番号〉
（bool * over_run, bool * mode_fault, bool * parity_error）
〈チャネル番号〉: 0

概要

エラー検出状態の取得

引数

bool * over_run	オーバランエラーフラグの格納先
bool * mode_fault	モードフォルトエラーフラグの格納先
bool * parity_error	パリティエラーフラグの格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_RSPI_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数

R_SPI_GetStatus

詳細

- エラーフラグを取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。
- 本関数内でエラーフラグはクリアされます。

使用例

R_PG_RSPI_StartTransfer_C〈チャネル番号〉 、 R_PG_RSPI_TransferAllData_C〈チャネル番号〉 およびR_PG_RSPI_GetCommandStatus_C〈チャネル番号〉 の使用例を参照してください。

5.17.7 R_PG_RSPI_GetCommandStatus_C〈チャネル番号〉

定義

bool R_PG_RSPI_GetCommandStatus_C〈チャネル番号〉
(uint8_t * current_command, uint8_t * error_command)
〈チャネル番号〉: 0

概要

コマンドステータスの取得

生成条件

RSPIチャネルをマスタモードに設定した場合

引数

uint8_t * current_command	現在のコマンドポインタ(0～7)の格納先
uint8_t * error_command	エラー検出時のコマンドポインタ(0～7)の格納先

戻り値

true	取得に成功した場合
false	取得に失敗した場合

出力先ファイル

R_PG_RSPI_C〈チャネル番号〉.c
〈チャネル番号〉: 0

使用RPDL関数

R_SPI_GetStatus

詳細

- 現在のコマンドポインタ(0～7)と、エラー検出時のコマンドポインタ(0～7)を取得します。
- 取得する項目に対応する引数に、値の格納先アドレスを指定してください。取得しない項目には0を指定してください。
- 本関数内でエラーフラグ(オーバランエラーフラグ、モードフォルトエラーフラグ、パリティエラーフラグ)はクリアされます。エラーフラグを取得する場合は本関数を呼び出す前にR_PG_RSPI_GetError_C〈チャネル番号〉を呼び出してください。

使用例

GUI上で以下の通り設定した場合

- GUI上でRSPIをSPI動作マスタモードで設定

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

bool  over_run,  mode_fault,  parity_error;
uint8_t error_command;

void func(void)
{
      R_PG_RSPI_GetError_C0 ( &over_run, &mode_fault, &parity_error );   //エラー取得
      if( over_run || mode_fault || parity_error ){
            R_PG_RSPI_GetCommandStatus_C0( &error_command );

            //エラー検出時処理

      }
}
```

5.17.8 R_PG_RSPI_StopModule_C〈チャネル番号〉

定義 bool R_PG_RSPI_StopModule_C〈チャネル番号〉(void)
〈チャネル番号〉: 0

概要 RSPIチャネルの停止

引数 なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

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

使用RPDL関数 R_SPI_Destroy

詳細
- RSPIチャネルを停止し、モジュールストップ状態に移行します。

使用例 R_PG_RSPI_StartTransfer_C〈チャネル番号〉およびR_PG_RSPI_TransferAllData_C〈チャネル番号〉の使用例を参照してください。

5.17.9 R_PG_RSPI_LoopBack〈*ループバックモード*〉_C〈*チャネル番号*〉

定義 bool R_PG_RSPI_LoopBack〈*ループバックモード*〉_C〈*チャネル番号*〉(void)
〈*ループバックモード*〉: Direct, Reversed, Disable
〈*チャネル番号*〉: 0

概要 ループバックモードの設定

生成条件 ループバックモードが設定されている場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数 R_SPI_Control

詳細
- 端子をループバックモードに設定または無効化します。
- R_PG_RSPI_LoopBackDirect_C〈*チャネル番号*〉を呼び出すとシフトレジスタの入力経路と出力経路を接続します。(送信データ＝受信データ)
- R_PG_RSPI_LoopBackReversed_C〈*チャネル番号*〉を呼び出すとシフトレジスタの入力経路と出力経路の反転を接続します。(送信データの反転＝受信データ)
- R_PG_RSPI_LoopBackDisable_C〈*チャネル番号*〉を呼び出すとループバックモードを無効にします。

使用例

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

void func(void)
{
    R_PG_RSPI_LoopBackDirect_C0();  //ループバックモードの設定
}
```

5.18 LINモジュール (LIN)

5.18.1 R_PG_LIN_Set_LIN0

定義 bool R_PG_LIN_Set_LIN0 (void)

概要 LINモジュールの設定

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_LIN_LIN0.c

使用RPDL関数 R_LIN_Create

詳細

- LINモジュールのモジュールストップ状態を解除して初期設定し、使用する端子を設定します。
  LINモジュールをLIN動作モードに設定します。
- 本関数内でLIN割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、
- CPUへの割り込み要求が発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。
  void <割り込み通知関数名> (void)
  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例

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

void func(void)
{
    //LINモジュールの初期設定 (LINモジュールはLIN動作モードに設定されます)
    R_PG_LIN_Set_LIN0();

    //ウェイクアップモードへの移行
    R_PG_LIN_EnterWakeUpMode_LIN0();

    //ウェイクアップの送信
    R_PG_LIN_WakeUpTransmit_LIN0();
}
```

5.18.2 R_PG_LIN_Transmit_LIN0

定義 bool R_PG_LIN_Transmit_LIN0
(uint8_t id, uint8_t * send_data, uint8_t data_count, bool checksum_enhanced)

概要 データの送信

引数

uint8_t id	ヘッダで送信するID
uint8_t * send_data	レスポンスフィールドで送信するデータの格納先の先頭のアドレス
uint8_t data_count	送信するデータ数 (0～8)
bool checksum_enhanced	チェックサム方式 (0:クラシック 1:エンハンス)

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_LIN_LIN0.c

使用RPDL関数 R_LIN_Transfer

詳細

- ヘッダとレスポンスを送信します。
- チェックサムは自動で計算されレスポンスに付加されます。
- 本関数を呼び出す前に _PG_LIN_Set_LIN0 によりLINモジュールを設定してください。
- 本関数はLIN動作モードで呼び出してください。他の動作モードからLIN動作モードに移行するには R_PG_LIN_EnterOperationMode_LIN0 を呼び出してください。
- ヘッダとフレームの送信状態は R_PG_LIN_GetStatus_LIN0 により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

- LIN割り込み通知関数名に Lin0IntFunc を指定

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

uint8_t data_t[8];     //送信データ格納先
bool frame_wakeup_transmission;   //フレーム、ウェイクアップ送信完了フラグ格納先
bool error;   //エラーフラグ格納先

void func(void)
{
      //送信データ初期化
      InitData();

      //ヘッダ、レスポンスの送信 (ID:3 データ数:8 チェックサム方式:クラシック)
      R_PG_LIN_Transmit_LIN0( 3, data_t, 8, 0 );
}

void Lin0IntFunc(void)
{
      //フレーム、ウェイクアップ送信状態、エラー検出状態の取得
      R_PG_LIN_GetStatus_LIN0(
            & frame_wakeup_transmission,
            0,
            & error,
            0,
            0,
      );

      if( error ){
            //エラー検出
      }
      else if( frame_wakeup_transmission ){
            //フレーム、ウェイクアップ送信完了
      }
}

void InitData(void)
{
      t data_t[0] = 0x12;
      t data_t[1] = 0x34;
      t data_t[2] = 0x56;
      t data_t[3] = 0x78;
      t data_t[4] = 0x9a;
      t data_t[5] = 0xbc;
      t data_t[6] = 0xde;
      t data_t[7] = 0xf0;
}
```

5.18.3 R_PG_LIN_Receive_LIN0

定義 bool R_PG_LIN_Receive_LIN0 (uint8_t id, uint8_t data_count, bool checksum_enhanced)

概要 データの受信

引数

uint8_t id	ヘッダで送信するID
uint8_t data_count	受信するデータ数（0～8）
bool checksum_enhanced	チェックサム方式 （1:エンハンス 0:クラシック）

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LIN_LIN0.c

使用RPDL関数 R_LIN_Transfer

詳細

- ヘッダを送信し、レスポンスを受信します。
- 本関数を呼び出す前に _PG_LIN_Set_LIN0 によりLINモジュールを設定してください。
- 本関数はLIN動作モードで呼び出してください。他の動作モードからLIN動作モードに移行するには R_PG_LIN_EnterOperationMode_LIN0 を呼び出してください。
- ヘッダの送信状態、レスポンスの受信状態は R_PG_LIN_GetStatus_LIN0 により取得することができます。
- 受信したデータは R_PG_LIN_ReadData_LIN0 により読み出すことができます。

使用例 GUI上で以下の通り設定した場合

- LIN割り込み通知関数名に Lin0IntFunc を指定

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

uint8_t data_r[8];     //受信データ格納先
bool frame_wakeup_reception;   //フレーム、ウェイクアップ受信完了フラグ格納先
bool error;   //エラーフラグ格納先
uint8_t check_sum    //チェックサム値格納先

void func(void)
{
      //ヘッダの送信、レスポンスの受信 (ID:3 データ数:8 チェックサム方式:クラシック)
      R_PG_LIN_Receive_LIN0( 3, 8, 0 );
}

void Lin0IntFunc(void)
{
      //フレーム、ウェイクアップ受信状態、エラー検出状態の取得
      R_PG_LIN_GetStatus_LIN0(
            0,
            & frame_wakeup_reception,
            & error,
            0,
            0,
      );

      if( error ){
            //エラー検出
      }
      else if( frame_wakeup_reception ){
            //フレーム、ウェイクアップ受信完了

            //受信データの読み出し
            R_PG_LIN_ReadData_LIN0( data_r, 8 );

            //チェックサム値の読み出し
            R_PG_LIN_GetCheckSum_LIN0( & check_sum );
      }
}
```

5.18.4 R_PG_LIN_ReadData_LIN0

定義　bool R_PG_LIN_ReadData_LIN0(uint8_t * receive_data, uint8_t data_count)

概要　データの読み出し

引数

uint8_t * receive_data	取得するデータの格納先の先頭アドレス
uint8_t data_count	取得するデータ数

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_LIN_LIN0.c

使用RPDL関数　R_LIN_Read

詳細
- データバッファレジスタを読み出します。
- R_PG_LIN_Receive_LIN0 により送信したヘッダに対するレスポンスは本関数により読み出すことができます。
- receive_data で指定するデータの格納先には、data_countで指定するデータ数に応じた領域を確保してください。
- チェックサムバッファレジスタは R_PG_LIN_GetCheckSum_LIN0 により読み出すことができます。

使用例　R_PG_LIN_Receive_LIN0 の使用例を参照してください。

5.18.5 R_PG_LIN_EnterResetMode_LIN0

定義　bool R_PG_LIN_EnterResetMode_LIN0 (void)

概要　LINリセットモードへの移行

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_LIN_LIN0.c

使用RPDL関数　R_LIN_Control

詳細

- LINモジュールをLINリセットモードに移行させます。
- LINセルフテストモードからLINウェイクアップモードまたはLIN動作モードに移行させる場合は、本関数により一旦LINリセットモードに移行させてください。

使用例

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

void func(void)
{
    //LINモジュールの初期設定 (LINモジュールはLIN動作モードに設定されます)
    R_PG_LIN_Set_LIN0();

    //LINリセットモードへの移行
    R_PG_LIN_EnterResetMode_LIN0();
}
```

5.18.6 R_PG_LIN_EnterOperationMode_LIN0

定義 bool R_PG_LIN_EnterOperationMode_LIN0 (void)

概要 LIN動作モードへの移行

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル R_PG_LIN_LIN0.c

使用RPDL関数 R_LIN_Control

詳細
- LINモジュールをLIN動作モードに移行させます。
- LINセルフテストモードからLIN動作モードに移行させる場合は、
  R_PG_LIN_EnterResetMode_LIN0 により一旦LINリセットモードに移行させてください。

使用例

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

void func(void)
{
    //LINリセットモードへの移行
    R_PG_LIN_EnterResetMode_LIN0();

    //LIN動作モードへの移行
    R_PG_LIN_EnterOperationMode_LIN0();
}
```

5.18.7 R_PG_LIN_EnterWakeUpMode_LIN0

定義　bool R_PG_LIN_EnterWakeUpMode_LIN0 (void)

概要　LINウェイクアップモードへの移行

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_LIN_LIN0.c

使用RPDL関数　R_LIN_Control

詳細

- LINモジュールをLINウェイクアップモードに移行させます。
- LINセルフテストモードからLINウェイクアップモードに移行させる場合は、
  R_PG_LIN_EnterResetMode_LIN0 により一旦LINリセットモードに移行させてください。

使用例

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

void func(void)
{
    //LINモジュールの初期設定 (LINモジュールはLIN動作モードに設定されます)
    R_PG_LIN_Set_LIN0();

    //LINウェイクアップモードへの移行
    R_PG_LIN_EnterWakeUpMode_LIN0();

    //LINウェイクアップの送信
    R_PG_LIN_WakeUpTransmit_LIN0();
}
```

5.18.8 R_PG_LIN_WakeUpTransmit_LIN0

定義　bool R_PG_LIN_WakeUpTransmit_LIN0 (void)

概要　ウェイクアップ信号の送信

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_LIN_LIN0.c

使用RPDL関数　R_LIN_Control

詳細

- ウェイクアップ信号を送信します。
- 本関数はLINウェイクアップモードで呼び出してください。
- ウェイクアップ信号の送信状態は R_PG_LIN_GetStatus_LIN0 により取得することができます。

使用例　R_PG_LIN_EnterWakeUpMode_LIN0 の使用例を参照してください。

5.18.9 R_PG_LIN_WakeUpReceive_LIN0

定義　bool R_PG_LIN_WakeUpReceive_LIN0 (void)

概要　ウェイクアップ信号の受信

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_LIN_LIN0.c

使用RPDL関数　R_LIN_Control

詳細

- ウェイクアップ信号を受信します。
- 本関数はLINウェイクアップモードで呼び出してください。
- ウェイクアップ信号の受信状態は R_PG_LIN_GetStatus_LIN0 により取得することができます。

使用例

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

bool frame_wakeup_reception;  //フレーム、ウェイクアップ受信完了フラグ格納先

void func(void)
{
    //LINモジュールの初期設定 (LINモジュールはLIN動作モードに設定されます)
    R_PG_LIN_Set_LIN0();

    //LINウェイクアップモードへの移行
    R_PG_LIN_EnterWakeUpMode_LIN0();

    //LINウェイクアップの受信
    R_PG_LIN_WakeUpReceive_LIN0();

    //ウェイクアップ信号の受信完了を待つ
    do{
        R_PG_LIN_GetStatus_LIN0(
            0,
            & frame_wakeup_reception,
            0,
            0,
            0,
        );
    }while( ! frame_wakeup_reception );
}
```

5.18.10R_PG_LIN_GetCheckSum_LIN0

定義 bool R_PG_LIN_GetCheckSum_LIN0(uint8_t * check_sum)

概要 チェックサム値の取得

引数

uint8_t * check_sum	チェックサムの格納先

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_LIN_LIN0.c

使用RPDL関数 R_LIN_GetStatus

詳細
- チェックサムバッファレジスタを読み出します。

使用例 R_PG_LIN_Receive_LIN0 の使用例を参照してください。

5.18.11 R_PG_LIN_EnterSelfTestMode_LIN0

定義　bool R_PG_LIN_EnterSelfTestMode_LIN0 (void)

概要　LINセルフテストモードへの移行

生成条件　LINセルフテストモードを使用する場合

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_LIN_LIN0.c

使用RPDL関数　R_LIN_Control

詳細

- LINモジュールをLINセルフテストモードに移行させます。
- LINウェイクアップモードまたはLIN動作モードからLINセルフテストモードに移行させる場合は、本関数内で一旦LINリセットモードに移行します。

使用例 GUI上で以下の通り設定した場合

- LIN割り込み通知関数名に Lin0IntFunc を指定

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

uint8_t data_t[8];     //送信データ格納先
uint8_t data_r[8];     //受信データ格納先
bool frame_wakeup_transmission;   //フレーム、ウェイクアップ送信完了フラグ格納先
bool error;   //エラーフラグ格納先

void func(void)
{
    //LINモジュールの初期設定 (LINモジュールはLIN動作モードに設定されます)
    R_PG_LIN_Set_LIN0();

    //LINセルフテストモードへの移行
    R_PG_LIN_EnterSelfTestMode_LIN0();

    //送信データ初期化
    InitData();

    //送信テスト開始
    R_PG_LIN_Transmit_LIN0( 3, data_t, 8, 0 );
}

void Lin0IntFunc(void)
{
    //フレーム、ウェイクアップ送信状態、エラー検出状態の取得
    R_PG_LIN_GetStatus_LIN0(
        & frame_wakeup_transmission,
        0,
        & error,
        0,
        0,
    );

    if( error ){
        //エラー検出
    }
    else if( frame_wakeup_transmission ){
        //送信完了

        //データバッファレジスタの読み出し
        R_PG_LIN_ReadData_LIN0( data_r, 8 );
    }
}

void InitData(void)
{
    t data_t[0] = 0x12;
    t data_t[1] = 0x34;
    t data_t[2] = 0x56;
    t data_t[3] = 0x78;
    t data_t[4] = 0x9a;
    t data_t[5] = 0xbc;
    t data_t[6] = 0xde;
    t data_t[7] = 0xf0;
}
```

5.18.12 R_PG_LIN_WriteCheckSum_LIN0

定義　bool R_PG_LIN_WriteCheckSum_LIN0 (uint8_t check_sum)

概要　チェックサム値の書き込み

生成条件　LINセルフテストモードを使用する場合

引数

uint8_t check_sum	書き込むチェックサム値

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_LIN_LIN0.c

使用RPDL関数　R_LIN_Control

詳細

- チェックサムバッファレジスタに値を書き込みます。
- LINセルフテストモードでの受信で、チェックサムを格納する場合は本関数を呼び出してください。

使用例　GUI上で以下の通り設定した場合

・ LIN割り込み通知関数名に Lin0IntFunc を指定

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

uint8_t data_r[8];    //受信データ格納先
bool frame_wakeup_reception;   //フレーム、ウェイクアップ受信完了フラグ格納先
bool error;   //エラーフラグ格納先

void func(void)
{
    //LINモジュールの初期設定 (LINモジュールはLIN動作モードに設定されます)
    R_PG_LIN_Set_LIN0();

    //LINセルフテストモードへの移行
    R_PG_LIN_EnterSelfTestMode_LIN0();

    //データバッファレジスタの設定
    SetData();

    //受信テスト開始
    R_PG_LIN_Receive_LIN0( 3, 8, 0 );
}

void Lin0IntFunc(void)
{
    //フレーム、ウェイクアップ受信状態、エラー検出状態の取得
    R_PG_LIN_GetStatus_LIN0(
        0,
        & frame_wakeup_reception,
        & error,
        0,
        0,
    );

    if( error ){
        //エラー検出
    }
    else if( frame_wakeup_reception ){
        //受信完了

        //データバッファレジスタの読み出し
        R_PG_LIN_ReadData_LIN0( data_r, 8 );
    }
}

void SetData(void)
{
    //データバッファレジスタへの書き込み
    *((uint8_t*)(0x94018))=data_t[0]; //LDB1
    *((uint8_t*)(0x94019))=data_t[1]; //LDB2
    *((uint8_t*)(0x9401A))=data_t[2]; //LDB3
    *((uint8_t*)(0x9401B))=data_t[3]; //LDB4
    *((uint8_t*)(0x9401C))=data_t[4]; //LDB5
    *((uint8_t*)(0x9401D))=data_t[5]; //LDB6
    *((uint8_t*)(0x9401E))=data_t[6]; //LDB7
    *((uint8_t*)(0x9401F))=data_t[7]; //LDB8

    //チェックサムの書き込み
    R_PG_LIN_WriteCheckSum_LIN0 ( 0xc3 )
}
```

5.18.13 R_PG_LIN_GetMode_LIN0

定義

bool R_PG_LIN_GetMode_LIN0 (uint8_t * mode)

概要

モードの取得

引数

uint8_t * mode	モードを示す値の格納先 各モードで格納される値 LINリセットモード 0x00 LINウェイクアップモード 0x01 LIN動作モード 0x03 LINセルフテストモード 0x04

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_LIN_LIN0.c

使用RPDL関数

R_LIN_GetStatus

詳細

- 現在のLINモジュールの動作モードを取得します

使用例

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

uint8_t mode    //モードを示す値の格納先

void func(void)
{
    //現在の動作モードの取得
    R_PG_LIN_GetMode_LIN0( &mode )

    switch( mode ){
        case 0x00:
            //LINリセットモード
            break;
        case 0x01:
            //LINウェイクアップモード
            break;
        case 0x03:
            //LIN動作モード
            break;
        case 0x04:
            //LINセルフテストモード
            break;
        default;
            break;
    }
}
```

5.18.14 R_PG_LIN_GetStatus_LIN0

定義

bool R_PG_LIN_GetStatus_LIN0
（bool * frame_wakeup_transmission, bool * frame_wakeup_reception, bool * error,
bool * data1_reception, bool * header_transmission ）

概要

LINモジュールの状態の取得

引数

bool * frame_wakeup_transmission	フレーム／ウェイクアップ送信完了フラグの格納先
bool * frame_wakeup_reception	フレーム／ウェイクアップ受信完了フラグの格納先
bool * error	エラー検出フラグの格納先
bool * data1_reception	データ1受信完了フラグの格納先
bool * header_transmission	ヘッダ送信完了フラグの格納先

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_LIN_LIN0.c

使用RPDL関数

R_LIN_GetStatus

詳細

- LINモジュールの状態を取得します
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

R_PG_LIN_Transmit_LIN0、R_PG_LIN_Receive_LIN0、R_PG_LIN_WakeUpReceive_LIN0、
R_PG_LIN_EnterSelfTestMode_LIN0、R_PG_LIN_WriteCheckSum_LIN0 の使用例を参照してください。

5.18.15 R_PG_LIN_GetErrorStatus_LIN0

定義

```
bool R_PG_LIN_GetErrorStatus_LIN0
（bool * bit_error,      bool * bus_error,      bool * frame_timeout,
  bool * framing,        bool * check_sum_error                      ）
```

概要

エラー検出状態の取得

引数

bool * bit_error	ビットエラーフラグの格納先
bool * bus_error	フィジカルバスエラーフラグの格納先
bool * frame_timeout	フレームタイムアウトエラーフラグの格納先
bool * framing	フレーミングエラーフラグの格納先
bool * check_sum_error	チェックサムエラーフラグの格納先

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_LIN_LIN0.c

使用RPDL関数

R_LIN_GetStatus

詳細

- LINモジュールのエラー検出状態を取得します
- 取得するフラグに対応する引数に、フラグ値の格納先アドレスを指定してください。
  取得しないフラグには0を指定してください。

使用例

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

bool error;             //エラーフラグ格納先

bool bit_error;         //ビットエラーフラグ格納先
bool bus_error;         //フィジカルバスエラーフラグ格納先
bool frame_timeout;     //フレームタイムアウト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 check_sum_error;   //チェックサムエラーフラグ格納先

void func(void)
{
    //エラー検出状態の取得
    R_PG_LIN_GetStatus_LIN0(
        0,
        0,
        & error,
        0,
        0,
    );

    if( error ){
        //エラー検出

        //エラー検出状態の取得
        R_PG_LIN_GetErrorStatus_LIN0(
            & bit_error,
            & bus_error,
            & frame_timeout,
            & framing,
            & check_sum_error       );

        if( bit_error ){
            //ビットエラー検出
        }
        if( bus_error ){
            //フィジカルバスエラー検出
        }
        if( frame_timeout ){
            //フレームタイムアウトエラー検出
        }
        if( framing ){
            //フレーミングエラー検出
        }
        if( check_sum_error ){
            //チェックサムエラー検出
        }
    }
}
```

5.18.16 R_PG_LIN_StopModule_LIN0

定義　bool R_PG_LIN_StopModule_LIN0 (void)

概要　LINモジュールの停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_LIN_LIN0.c

使用RPDL関数　R_LIN_Destroy

詳細

- LINモジュールを停止し、モジュールストップ状態に移行します。

使用例

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

void func(void)
{
    //LINモジュールの停止
    R_PG_LIN_StopModule_LIN0();
}
```

5.19 12 ビットA/Dコンバータ（S12ADA）

5.19.1 R_PG_ADC_12_Set_S12ADA〈*ユニット番号*〉

定義　bool R_PG_ADC_12_Set_S12ADA〈*ユニット番号*〉(void)　〈*ユニット番号*〉： 0～1

概要　12ビットA/Dコンバータの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ADC_12_S12ADA〈*ユニット番号*〉.c　〈*ユニット番号*〉： 0, 1

使用RPDL関数　R_ADC_12_CreateUnit, R_ADC_12_Set

詳細

- 12ビットA/Dコンバータのモジュールストップ状態を解除して初期設定し、変換開始トリガ入力待ち状態にします。変換開始トリガにソフトウェアを選択した場合は、R_PG_ADC_12_StartConversionSW_S12ADA〈*チャネル番号*〉により変換を開始します。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。
- 本関数内で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、入力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を指定した場合、CPUへの割り込み要求が発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。

  void 〈割り込み通知関数名〉(void)

  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例

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

void func(void)
{
    R_PG_Clock_Set();                   //クロックの設定
    R_PG_ADC_12_Set_S12ADA0();          //12ビットA/Dコンバータを設定
}
```

5.19.2 R_PG_ADC_12_Set

定義　bool R_PG_ADC_12_Set(void)

概要　ゲインアンプの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ADC_12.c

使用RPDL関数　R_ADC_12_CreateChannel

詳細
- 本関数内でプログラマブルゲインアンプのゲインを設定します。

使用例　GUI上で以下の通り設定した場合
- プログラマブルゲインアンプを使用する設定にした場合

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

void func(void)
{
    R_PG_Clock_Set();                //クロックの設定
    R_PG_ADC_12_Set_S12ADA0();       //12ビットA/Dコンバータを設定

    //ゲインアンプの設定
    R_PG_ADC_12_Set();

    //ソフトウェアトリガによりA/D変換開始
    R_PG_ADC_12_StartConversionSW_S12ADA0();
}
```

5.19.3 R_PG_ADC_12_StartConversionSW_S12ADA〈*ユニット番号*〉

定義　bool R_PG_ADC_12_StartConversionSW_S12ADA〈*ユニット番号*〉(void)
〈*ユニット番号*〉 ： 0～1

概要　A/D変換の開始（ソフトウェアトリガ）

生成条件　変換開始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	設定が正しく行われた場合
false	設定に失敗した場合

出力先ファイル　R_PG_ADC_12_S12ADA〈*ユニット番号*〉.c　〈*ユニット番号*〉 ： 0, 1

使用RPDL関数　R_ADC_12_Control

詳細
- 起動要因にソフトウェアトリガを選択したA/D変換器のA/D変換を開始します。

使用例　GUI上で以下の通り設定した場合
- 起動要因をソフトウェアトリガに指定して設定

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

void func(void)
{
    R_PG_Clock_Set();                   //クロックの設定
    R_PG_ADC_12_Set_S12ADA0();          //12ビットA/Dコンバータを設定

    //ソフトウェアトリガによりA/D変換開始
    R_PG_ADC_12_StartConversionSW_S12ADA0();
}
```

5.19.4 R_PG_ADC_12_StopConversion_S12ADA〈ユニット番号〉

定義　bool R_PG_ADC_12_StopConversion_S12ADA〈ユニット番号〉(void)
〈ユニット番号〉： 0～1

概要　A/D変換の停止

引数　なし

戻り値

true	変換停止に成功した場合
false	変換停止に失敗した場合

出力先ファイル　R_PG_ADC_12_S12ADA〈ユニット番号〉.c　〈ユニット番号〉： 0, 1

使用RPDL関数　R_ADC_12_Control

詳細
- 本関数により連続スキャンモードのA/D変換を停止することができます。連続スキャンモード以外のモードではA/D変換完了後に本関数を呼び出す必要はありません。
- 本関数でA/D変換を停止させた後、A/D変換開始トリガを入力すると連続スキャンを再開します。連続スキャンを終了するにはR_PG_ADC_12_StopModule_S12ADA〈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上で以下の通り設定した場合
- 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_<プロジェクト名>.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();              //クロックの設定
    R_PG_ADC_12_Set_S12ADA0();     //12ビットA/Dコンバータを設定
}

void func2(void)
{
    //連続スキャンを停止
    R_PG_ADC_12_StopConversion_S12ADA0();

    //A/D変換結果の取得
    R_PG_ADC_12_GetResult_S12ADA0(&data);

    //A/D変換ユニットの停止
    R_PG_ADC_12_StopModule_S12ADA0();
}
```

5.19.5 R_PG_ADC_12_GetResult_S12ADA〈*ユニット番号*〉

定義　bool R_PG_ADC_12_GetResult_S12ADA〈*ユニット番号*〉(uint16_t * result)

〈*ユニット番号*〉： 0～1

概要　A/D変換結果の取得

引数

uint16_t * result	A/D変換結果の格納先の先頭アドレス

戻り値

true	結果の取得に成功した場合
false	結果の取得に失敗した場合

出力先ファイル　R_PG_ADC_12_S12ADA〈*ユニット番号*〉.c　〈*ユニット番号*〉： 0, 1

使用RPDL関数　R_ADC_12_Read

詳細

- A/D変換結果を格納するのに必要な領域は、(1 ～ 5) * 2バイトとなります。必要な領域数は、入力チャネルとA/D変換開始トリガに依存します。
- GUI上で、A/D変換開始トリガとして入力チャネルANn00(n = 0,1)用にダブルトリガを選択した場合は、指定した配列の先頭2バイトにAトリガによる入力チャネルANn00(n = 0,1)のA/D変換結果が格納され、終端2バイトにBトリガによる入力チャネルANn00(n = 0,1)のA/D変換結果が格納されます。

  【ダブルトリガ】(以下6パターン) (Aトリガ/Bトリガ)

  ① (TRG4AN/TRG4BN)
  ② (TRG7AN/TRG7BN)
  ③ (GTADTRA0N/GTADTRB0N)
  ④ (GTADTRA1N/GTADTRB1N)
  ⑤ (GTADTRA2N/GTADTRB2N)
  ⑥ (GTADTRA3N/GTADTRB3N)

  【例】 チャネルANn00～ANn03を変換し、ANn00の変換開始要因にダブルとリガを使用する場合

  result[0] :AトリガによるANn00(n = 0,1)のA/D変換結果 ADDR0A
  result[1] :ANn01(n = 0,1)のA/D変換結果 ADDR1
  result[2] :ANn02(n = 0,1)のA/D変換結果 ADDR2
  result[3] :ANn03(n = 0,1)のA/D変換結果 ADDR3
  result[4] :BトリガによるANn00(n = 0,1)のA/D変換結果 ADDR0B
- GUI上でMTU3またはGPTのトリガを選択した場合は、ダブルトリガ以外の場合はADDR0Bを使用しませんが、配列の終端にADDR0Bの値が格納されます。
- GUI上で割り込み通知関数名を指定していない場合、本関数を呼び出した時点でA/D変換中であったときは、結果を読み出す前に変換が終了するまで本関数内で待ちます。

使用例 GUI上で以下の通り設定した場合

- 2チャネルスキャンモードで設定
  グループ0: AN000 (変換開始トリガ: TRG4AN/TRG4BN)
  グループ1: AN001 - AN003 (変換開始トリガ: TRG7BN)

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

uint16_t result[5];        //A/D変換結果の格納先

void func(void)
{
      R_PG_Clock_Set();                //クロックの設定
      R_PG_ADC_12_Set_S12ADA0();       //12ビットA/Dコンバータを設定
}

//AD変換終了割り込み通知関数
void S12ad0IntFunc(void)
{
      //A/D変換結果の取得
      R_PG_ADC_12_GetResult_S12ADA0( result );
}
```

5.19.6 R_PG_ADC_12_GetResult_SelfDiag_S12AD〈*ユニット番号*〉

定義　bool R_PG_ADC_12_GetResult_SelfDiag_S12AD〈*ユニット番号*〉(uint16_t * result)
〈*ユニット番号*〉： 0～1

概要　A/Dコンバータの自己診断でA/D変換した結果の取得

引数

uint16_t * result	A/D変換結果の格納先

戻り値

true	結果の取得に成功した場合
false	結果の取得に失敗した場合

出力先ファイル　R_PG_ADC_12_S12ADA〈*ユニット番号*〉.c　〈*ユニット番号*〉： 0, 1

使用RPDL関数　R_ADC_12_Read

詳細
- 本関数内で、自己診断のA/D変換結果を取得します。
- 自己診断機能を使用する場合、自己診断はスキャンごとの最初に1回実施され、A/Dコンバータ内部で生成する3つの電圧値のうち1つをA/D変換します。
- 取得したA/D変換結果には自己診断ステータス情報(*1)が含まれます。
  データフォーマットは以下のようになります。

[GUI上でデータプレイスメントに右詰めを選択した場合]
b15-b14 :自己診断ステータス情報(*1)
b11-b0 :自己診断のA/D変換結果
[GUI上でデータプレイスメントに左詰めを選択した場合]
b15-b4 :自己診断のA/D変換結果
b1-b0 :自己診断ステータス情報(*1)

*1:自己診断ステータス情報
b'00 :一度も自己診断を実施していない
b'01 :0[V]の電圧値の自己診断を実施したことを示す
b'10 :VREFH0×1/2の電圧値の自己診断を実施したことを示す
b'11 :VREFH0の電圧値の自己診断を実施したことを示す

使用例　　GUI上で以下の通り設定した場合

- シングルスキャンモードを選択
- アナログ入力端子にAN000とAN003を指定
- 起動要因にソフトウェアトリガを選択
- データプレイスメントに右詰めを選択
- 自己診断機能を有効に設定
- A/D変換終了割り込み通知関数名に S12ad0IntFunc を設定

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

uint16_t result_selfdiag;   //自己診断A/D変換結果の格納先
uint16_t adrd_ad;           //12ビットA/D変換値の格納先
uint16_t adrd_diagst;       //自己診断ステータス情報の格納先
uint16_t result[16];        //AN000,AN008のA/D変換結果の格納先
uint16_t result_an000;      //AN000のA/D変換結果の格納先
uint16_t result_an003;      //AN003のA/D変換結果の格納先

void func(void)
{
    R_PG_Clock_Set();                   //クロックの設定
    R_PG_ADC_12_Set_S12AD0();   //12ビットA/Dコンバータ(S12AD0)を設定

    //ソフトウェアトリガによりAD変換開始
    R_PG_ADC_12_StartConversionSW_S12AD0();
}

//A/D変換終了割り込み通知関数
void S12ad0AIntFunc(void)
{
    //自己診断A/D変換結果の取得
    R_PG_ADC_12_GetResult_SelfDiag_S12AD0( &result_selfdiag );

    adrd_ad = (result_selfdiag & 0x0fff);
    adrd_diagst = (result_selfdiag >> 14);

    //AN000,AN003のA/D変換結果の取得
    R_PG_ADC_12_GetResult_S12AD0( result );

    result_an000 = result[0];
    result_an003 = result[3];
}
```

5.19.7 R_PG_ADC_12_StopModule_S12ADA〈*ユニット番号*〉

定義　bool R_PG_ADC_12_StopModule_S12ADA〈*ユニット番号*〉(void) 〈*ユニット番号*〉 : 0～1

概要　12ビットA/Dコンバータの停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

出力先ファイル　R_PG_ADC_12_S12ADA〈*ユニット番号*〉.c　〈*ユニット番号*〉 : 0, 1

使用RPDL関数　R_ADC_12_Destroy

詳細
- 12ビットA/Dコンバータ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。(消費電力低減機能)
- 12ビットA/Dコンバータを2ユニット使用している場合は、一方のユニットに対して本関数を呼び出してもモジュールストップ状態には移行しません。その後もう一方のユニットに対して本関数を呼び出した時に両ユニットがモジュールストップ状態に移行します。

使用例　R_PG_ADC_12_StopConversion_S12ADA〈*ユニット番号*〉の使用例を参照してください。

5.20 10 ビットA/Dコンバータ（ADA）

5.20.1 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

概要　10ビットA/Dコンバータの設定

引数　なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

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。
- 本関数を呼び出す前にR_PG_Clock_Setによりクロックを設定してください。
- 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、入力バッファを無効にします。
- ソフトウェアトリガによりA/D変換を開始する場合は、本関数を呼び出した後に R_PG_ADC_10_StartConversionSW_AD〈*ユニット番号*〉 を呼び出してください。
- 本関数内でA/D変換終了割り込みを設定します。GUI上で割り込み通知関数名を指定した場合、CPUへの割り込み要求が発生すると指定した名前の関数が呼び出されます。通知関数は次の定義で作成してください。

  void 〈割り込み通知関数名〉(void)

  割り込み通知関数については「5.21 通知関数に関する注意事項」の内容に注意してください。

使用例　GUI上で以下の通り設定した場合

- 起動要因にハードウェアトリガを指定
- 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_〈プロジェクト名〉.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();           //クロックの設定
    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)
{
    R_PG_ADC_10_GetResult_AD0(&data);      //A/D変換結果の取得
}
```

5.20.2 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

概要　A/D変換の開始（ソフトウェアトリガ）

引数　なし

戻り値

true	変換開始に成功した場合
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

使用RPDL関数　R_ADC_10_Control

詳細
- ソフトウェアトリガをかける場合は、本関数を呼び出してください。
- ハードウェアトリガを選択している場合でも、本関数を呼び出すことによりA/D変換を開始させることが可能です。

使用例　GUI上で以下の通り設定した場合
- 起動要因にソフトウェアトリガを指定

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

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_ADC_10_Set_AD0();         //AD0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_AD0();
}
```

5.20.3 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

概要　A/D変換の停止

引数　なし

戻り値

true	変換停止に成功した場合
false	変換停止に失敗した場合

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

使用RPDL関数　R_ADC_10_Control

詳細
- 本関数により連続スキャンモードのA/D変換を停止することができます。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変換を停止させた後、A/D変換開始トリガを入力すると連続スキャンを再開します。連続スキャンを終了するにはR_PG_ADC_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上で以下の通り設定した場合
- 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_<プロジェクト名>.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();              //クロックの設定
    R_PG_ADC_10_Set_AD0();         //AD0を設定
}

void func2(void)
{
    //連続スキャンを停止
    R_PG_ADC_10_StopConversion_AD0();

    //A/D変換結果の取得
    R_PG_ADC_10_GetResult_AD0(&data);

    //A/D変換ユニットの停止
    R_PG_ADC_10_StopModule_AD0();
}
```

5.20.4 R_PG_ADC_10_GetResult_AD〈*ユニット番号*〉

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

〈*ユニット番号*〉 : 0

概要　A/D変換結果の取得

引数

uint16_t * result	A/D変換結果の格納先

戻り値

true	結果の取得に成功した場合
false	結果の取得に失敗した場合

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

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、本関数を呼び出した時点でA/D変換中であったときは、結果を読み出す前に変換が終了するまで本関数内で待ちます。

使用例　GUI上で以下の通り設定した場合

- アナログ入力端子にAN0～AN3の4チャネルを指定
- 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_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"

void func(void)
{
    R_PG_Clock_Set();              //クロックの設定
    R_PG_ADC_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 result[4];   //使用チャネル数分のA/D変換結果の格納先
    uint16_t result_an0;  //AN0のA/D変換結果の格納先
    uint16_t result_an1;  //AN1のA/D変換結果の格納先
    uint16_t result_an2;  //AN2のA/D変換結果の格納先
    uint16_t result_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_AD0( result );

    result_an0 = result[0];
    result_an1 = result[1];
    result_an2 = result[2];
    result_an3 = result[3];
}
```

5.20.5 R_PG_ADC_10_SetSelfDiag_VREF_〈電圧値〉_AD〈ユニット番号〉

定義 bool R_PG_ADC_10_SetSelfDiag_VREF_〈電圧値〉_AD〈ユニット番号〉(void)

〈電圧値〉 : 0, 0_5, 1 (0:Vref*0, 0_5:Vref/2, 1:Vref) 〈ユニット番号〉 : 0

概要 A/D自己診断機能の設定

生成条件 自己診断機能を使用する場合

引数 なし

戻り値

true	設定が正しく行われた場合
false	設定に失敗した場合

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

使用RPDL関数 R_ADC_10_Create

詳細

- 自己診断機能を設定します。
- 本関数内でA/D変換モードはシングルモードに、変換開始トリガはソフトウェアトリガに設定されます。
- A/Dコンバータを再設定するにはR_PG_ADC_10_Set_AD〈ユニット番号〉を呼び出してください。
- 自己診断を開始するには R_PG_ADC_10_StartConversionSW_AD〈ユニット番号〉 を、自己診断結果を取得するには R_PG_ADC_10_GetResult_AD〈ユニット番号〉 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

- 自己診断機能を使用する設定を指定

```
//この関数を使用するには"R_PG_〈プロジェクト名〉.h"をインクルードしてください
#include "R_PG_default.h"
uint16_t SelfDiagnostic_0()
{
	uint16_t result;
	R_PG_ADC_10_SetSelfDiag_VREF_0_AD0();
	R_PG_ADC_10_StartConversionSW_AD0();
	R_PG_ADC_10_GetResult_AD0 (&result);
	return result;
}
uint16_t SelfDiagnostic_0_5()
{
	uint16_t result;
	R_PG_ADC_10_SetSelfDiag_VREF_0_5_AD0();
	R_PG_ADC_10_StartConversionSW_AD0();
	R_PG_ADC_10_GetResult_AD0 (&result);
	return result;
}
uint16_t SelfDiagnostic_1()
{
	uint16_t result;
	R_PG_ADC_10_SetSelfDiag_VREF_1_AD0();
	R_PG_ADC_10_StartConversionSW_AD0();
	R_PG_ADC_10_GetResult_AD0 (&result);
	return result;
}
```

5.20.6 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

概要　10ビットA/Dコンバータの停止

引数　なし

戻り値

true	停止に成功した場合
false	停止に失敗した場合

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

使用RPDL関数　R_ADC_10_Destroy

詳細
- 10ビットA/Dコンバータ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。(消費電力低減機能)

使用例　R_PG_ADC_10_StopConversion_AD〈ユニット番号〉の使用例を参照してください。

5.21 通知関数に関する注意事項

5.21.1 割り込みとプロセッサモード

RX CPU は、スーパバイザモード、およびユーザモードの 2 つのプロセッサモードをサポートします。Peripheral Driver Generator の出力関数および Renesas Peripheral Driver Library の関数はユーザモードで実行されますが、各通知関数は Renesas Peripheral Driver Library の割り込みハンドラから呼び出されるため、スーパバイザモードで動作します。スーパバイザモードでは特権命令(RTFI、RTE、WAIT)を使用できますが、通知関数と通知関数から呼び出される他の関数では以下の点に注意してください。

- RTFI および RTE 命令は Renesas Peripheral Driver Library の割り込みハンドラで実行するため、ユーザプログラムでこれらを実行する必要はありません。
- Peripheral Driver Generator の出力関数および Renesas Peripheral Driver Library の関数では消費電力低減のために wait()命令を呼び出しています。ユーザプログラムから wait()を呼び出さないでください。

プロセッサモードについての詳細は RX ファミリ ソフトウェアマニュアルを参照してください。

5.21.2 割り込みとDSP命令

アキュムレータ(ACC)は以下の命令で変更されます。

- DSP 機能命令(MACHI、MACLO、MULHI、MULLO、MVTACHI、MVTACLO、および RACW)
- 乗算命令、積和演算命令(EMUL、EMULU、FMUL、MUL、および RMPA)

Renesas Peripheral Driver Library の割り込みハンドラでは ACC の値をスタックに退避しません。各通知関数は Renesas Peripheral Driver Library の割り込みハンドラから呼び出されるため、通知関数内でこれらの命令を使用する場合は ACC の値を退避し、通知関数が終了する前に再設定してください。

6. 生成ファイルのIDEへの登録とビルド

Peripheral Driver Generator で生成したファイルの IDE(High-performance Embedded Workshop／CubeSuite+／e2 studio)への登録とビルドについては以下の点に注意してください。

(1) Peripheral Driver Generator が生成するソースファイルにはスタートアッププログラムは含まれません。IDE のプロジェクト作成時にプロジェクトタイプとして Application を指定してスタートアッププログラムを作成してください。

(2) Peripheral Driver Generator が IDE に登録するソースファイルには割り込みハンドラとベクタテーブルが含まれます。IDE で生成したスタートアッププログラムに含まれる割り込みハンドラ、ベクタテーブルとの重複を避けるため、Peripheral Driver Generator から IDE にソースファイルを登録する際、intprg.c と vecttbl.c (e2 studio の場合は interrupt_handlers.c と vector_table.c) はビルドの対象から除外されます。

(3) Peripheral Driver Generator が IDE に登録する割り込みハンドラを含むソースファイル Interrupt_<周辺機能名>.c は、Peripheral Driver Generator のソース生成時に上書きされます。

(4) Renesas Peripheral Driver Library ライブラリファイルは、デフォルトのオプションで作成しています。(ただし、double 型の精度は倍精度に設定して作成しています) お客様のプロジェクトでデフォルト以外のオプションを指定する場合は、お客様の責任で Renesas Peripheral Driver Library のソースファイルを利用してください。

(5) Renesas Peripheral Driver Library は double 型の精度を倍精度に設定して作成されています。そのため、Peripheral Driver Generator が生成したソースを含むプログラムをビルドするには、以下のように IDE のビルダ設定で double 型の精度を指定してください。(e2 studio ではソース登録と同時に自動で変更します)

CubeSuite+

1. プロジェクトツリーの［CC-RX(ビルド・ツール)］をダブルクリックし、[CC-RXのプロパティ］を表示してください。
2. [CPU]カテゴリ内の［double型、およびlong double型の精度］に［倍精度として扱う］を設定してください。

High-performance Embedded Workshop

1. メインメニューから［ビルド］->［RX Standard Toolchain］を選択し、[RX Standard Toolchain］ダイアログボックスを開いてください。
2. ［CPU］タブを選択してください。
3. ［詳細］ボタンをクリックし、[CPU詳細］ダイアログボックスを開いてください。
4. ［double型の精度］に［倍精度］を設定してください。

(6) Renesas Peripheral Driver Library は FIXEDVECT セクションの開始アドレスを、0xFFFFFFD0 にして作成しています。そのため PDG2 が生成したソースを含むプログラムをビルドするには、以下のようにビルダの設定で FIXEDVECT セクションのアドレスを変更してください。(CubeSuite+ および High-performance Embedded Workshop では変更不要)

e2 studio

1. プロジェクトエクスプローラでプロジェクトを選んでください。
2. メニューから[ファイル]->[プロパティ]を選択し[プロパティ]を表示してください。
3. プロパティの[C/C++ビルド]の[設定]を選んでください。
4. 構成:で[全ての構成]を選んでください。
5. [Linker]の[セクション]を選択し、「セクション・ビューアー」を表示してください。
6. [セクション・ビューアー]で、FIXEDVECTセクションのアドレスを0xFFFFFFD0に設定してください。

付録 1. 割当先を変更できる端子機能

表 a-1.1 112-pin LQFP （上段が初期設定です）

周辺機能	端子機能	割り当て先	Pin No.
ICU (外部割込み)	IRQ0	P10/MTCLKD/IRQ0	110
		PE5/IRQ0	1
		PG0/IRQ0/TRSYNC	59
	IRQ1	P11/MTCLKC/IRQ1	109
		PE4/MTCLKC/IRQ1/POE10#	8
		PG1/IRQ1/TRDATA0	58
	IRQ2	PE3/MTCLKD/IRQ2/POE11#	9
		PG2/IRQ2/TRDATA1	57
MTU3	MTCLKA	P33/MTIOC3A/MTCLKA/SSL3	67
	*1	P21/ADTRG1#/MTCLKA/IRQ6	76
	MTCLKB	P32/MTIOC3C/MTCLKB/SSL2	68
	*1	P20/ADTRG0#/MTCLKB/IRQ7	77
	MTCLKC	P31/MTIOC0A/MTCLKC/SSL1	70
		P11/MTCLKC/IRQ1	109
	*1	PE4/MTCLKC/IRQ1/POE10#	8
	MTCLKD	P30/MTIOC0B/MTCLKD/SSL0	72
		P10/MTCLKD/IRQ0	110
	*1	PE3/MTCLKD/IRQ2/POE11#	9
MTU3_0	MTIOC0A	PB3/MTIOC0A/SCK0	35
		P31/MTIOC0A/MTCLKC/SSL1	70
	MTIOC0B	PB2/MTIOC0B/TXD0/SDA	36
		P30/MTIOC0B/MTCLKD/SSL0	72
POE	POE10#	PE2/NMI/POE10#	15
		PE4/MTCLKC/IRQ1/POE10#	8
GPT0	GTIOC0A	P71/MTIOC3B/GTIOC0A	65
	*2	PD7/GTIOC0A/CTX/SSL1	18
	GTIOC0B	P74/MTIOC3D/GTIOC0B	62
	*2	PD6/GTIOC0B/SSL0	19
GPT1	GTIOC1A	P72/MTIOC4A/GTIOC1A	64
	*2	PD5/GTIOC1A/RXD1	20
	GTIOC1B	P75/MTIOC4C/GTIOC1B	61
	*2	PD4/GTIOC1B/SCK1	21
GPT2	GTIOC2A	P73/MTIOC4B/GTIOC2A	63
	*2	PD3/GTIOC2A/TXD1	22
	GTIOC2B	P76/MTIOC4D/GTIOC2B	60
	*2	PD2/GTIOC2B/MOSI	23
SCI2	TXD2	PB5/CTX/TXD2	31
	*3	P81/MTIC5V/TXD2	106
	RXD2	PB6/CRX/RXD2	30
	*3	P80/MTIC5W/RXD2	107
	SCK2	PB7/SCK2	29
	*3	P82/MTIC5U/SCK2	105
RSPI0	RSPCK	P24/RSPCK	73
		PA4/ADTRG0#/MTIOC1B/RSPCK	40
	*4	PD0/GTIOC3B/RSPCK	25
	MOSI	P23/CTX/LTX/MOSI	74
		PB0/MTIOC0D/MOSI	38

	*4	PD2/GTIOC2B/MOSI	23
	MISO	P22/ADTRG#/CRX/LRX/MISO	75
		PA5/ADTRG1#/MTIOC1A/MISO	39
	*4	PD1/GTIOC3A/MISO	24
	SSL0	P30/MTIOC0B/MTCLKD/SSL0	72
		PA3/MTIOC2A/SSL0	41
	*4	PD6/GTIOC0B/SSL0	19
	SSL1	P31/MTIOC0A/MTCLKC/SSL1	70
		PA2/MTIOC2B/SSL1	42
	*4	PD7/GTIOC0A/CTX/SSL1	18
	SSL2	P32/MTIOC3C/MTCLKB/SSL2	68
		PA1/MTIOC6A/SSL2	43
	*4	PE0/CRX/SSL2	17
	SSL3	P33/MTIOC3A/MTCLKA/SSL3	67
		PA0/MTIOC6C/SSL3	44
	*4	PE1/SSL3	16
S12ADA0	ADTRG0#	PA4/ADTRG0#/MTIOC1B/RSPCK	40
		P20/ADTRG0#/MTCLKB/IRQ7	77
S12ADA1	ADTRG1#	PA5/ADTRG1#/MTIOC1A/MISO	39
		P21/ADTRG1#/MTCLKA/IRQ6	76

*1 ～ 4 設定は連動して変更されます。

表 a-1.2 100-pin LQFP （上段が初期設定です）

周辺機能	端子機能	割り当て先	Pin No.
ICU (外部割込み)	IRQ0	P10/MTCLKD/IRQ0	100
		PE5/IRQ0	1
	IRQ1	P11/MTCLKC/IRQ1	99
		PE4/MTCLKC/IRQ1/POE10#	8
MTU3	MTCLKA	P33/MTIOC3A/MTCLKA/SSL3	58
	*1	P21/ADTRG1#/MTCLKA/IRQ6	67
	MTCLKB	P32/MTIOC3C/MTCLKB/SSL2	59
	*1	P20/ADTRG0#/MTCLKB/IRQ7	68
	MTCLKC	P31/MTIOC0A/MTCLKC/SSL1	61
		P11/MTCLKC/IRQ1	99
	*1	PE4/MTCLKC/IRQ1/POE10#	8
	MTCLKD	P30/MTIOC0B/MTCLKD/SSL0	63
		P10/MTCLKD/IRQ0	100
	*1	PE3/MTCLKD/IRQ2/POE11#	9
MTU3_0	MTIOC0A	PB3/MTIOC0A/SCK0	32
		P31/MTIOC0A/MTCLKC/SSL1	61
	MTIOC0B	PB2/MTIOC0B/TXD0/SDA	33
		P30/MTIOC0B/MTCLKD/SSL0	63
POE	POE10#	PE2/NMI/POE10#	15
		PE4/MTCLKC/IRQ1/POE10#	8
GPT0	GTIOC0A	P71/MTIOC3B/GTIOC0A	56
	*2	PD7/GTIOC0A/CTX/SSL1/TRST#	18
	GTIOC0B	P74/MTIOC3D/GTIOC0B	53
	*2	PD6/GTIOC0B/SSL0/TMS	19
GPT1	GTIOC1A	P72/MTIOC4A/GTIOC1A	55
	*2	PD5/GTIOC1A/RXD1/TDI	20

	GTIOC1B	P75/MTIOC4C/GTIOC1B	52
	*2	PD4/GTIOC1B/SCK1/TCK	21
GPT2	GTIOC2A	P73/MTIOC4B/GTIOC2A	54
	*2	PD3/GTIOC2A/TXD1/TDO	22
	GTIOC2B	P76/MTIOC4D/GTIOC2B	51
	*2	PD2/GTIOC2B/MOSI/TRCLK	23
SCI2	TXD2	PB5/CTX/TXD2/TRSYNC	28
	*3	P81/MTIC5V/TXD2	97
	RXD2	PB6/CRX/RXD2/TRDATA0	27
	*3	P80/MTIC5W/RXD2	98
	SCK2	PB7/SCK2/TRDATA1	26
	*3	P82/MTIC5U/SCK2	96
RSPI0	RSPCK	P24/RSPCK	64
		PA4/ADTRG0#/MTIOC1B/RSPCK	37
	*4	PD0/GTIOC3B/RSPCK/TRDATA2	25
	MOSI	P23/CTX/LTX/MOSI	65
		PB0/MTIOC0D/MOSI	35
	*4	PD2/GTIOC2B/MOSI/TRCLK	23
	MISO	P22/ADTRG#/CRX/LRX/MISO	66
		PA5/ADTRG1#/MTIOC1A/MISO	36
	*4	PD1/GTIOC3A/MISO/TRDATA3	24
	SSL0	P30/MTIOC0B/MTCLKD/SSL0	63
		PA3/MTIOC2A/SSL0	38
	*4	PD6/GTIOC0B/SSL0/TMS	19
	SSL1	P31/MTIOC0A/MTCLKC/SSL1	61
		PA2/MTIOC2B/SSL1	39
	*4	PD7/GTIOC0A/CTX/SSL1/TRST#	18
	SSL2	P32/MTIOC3C/MTCLKB/SSL2	59
		PA1/MTIOC6A/SSL2	40
	*4	PE0/CRX/SSL2	17
	SSL3	P33/MTIOC3A/MTCLKA/SSL3	58
		PA0/MTIOC6C/SSL3	41
	*4	PE1/SSL3	16
S12ADA0	ADTRG0#	PA4/ADTRG0#/MTIOC1B/RSPCK	37
		P20/ADTRG0#/MTCLKB/IRQ7	68
S12ADA1	ADTRG1#	PA5/ADTRG1#/MTIOC1A/MISO	36
		P21/ADTRG1#/MTCLKA/IRQ6	67

*1 ～ 4 設定は連動して変更されます。

RX62Gグループ
Peripheral Driver Generator
リファレンスマニュアル

発行年月日　2014年5月16日　Rev.1.02

発行　ルネサス エレクトロニクス株式会社
〒211-8668 神奈川県川崎市中原区下沼部1753

編集　株式会社ルネサス ソリューションズ
ツールビジネス本部 ツール開発第四部

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

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

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

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

RX62G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

R20UT2275JJ0102