User's Manual

32

Renesas Peripheral Driver Library

User's Manual

RX610 Group

— Release —

Renesas Electronics
www.renesas.com

Rev.1.00 Jul. 2010

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this manual, refer to the relevant sections of the manual. If the descriptions under General Precautions in the Handling of MPU/MCU Products and in the body of the manual differ from each other, the description in the body of the manual takes precedence.

1. Handling of Unused Pins

   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.

   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.

   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to one with a different part number, confirm that the change will not lead to problems.

   — The characteristics of MPU/MCU in the same group but having different part numbers may differ because of the differences in internal memory capacity and layout pattern. When changing to products of different part numbers, implement a system-evaluation test for each of the products.

Table of Contents

1. Introduction

The Renesas Peripheral Driver Library (RPDL) is a unified API for controlling the peripheral modules on the microcontrollers made by Renesas Electronics.

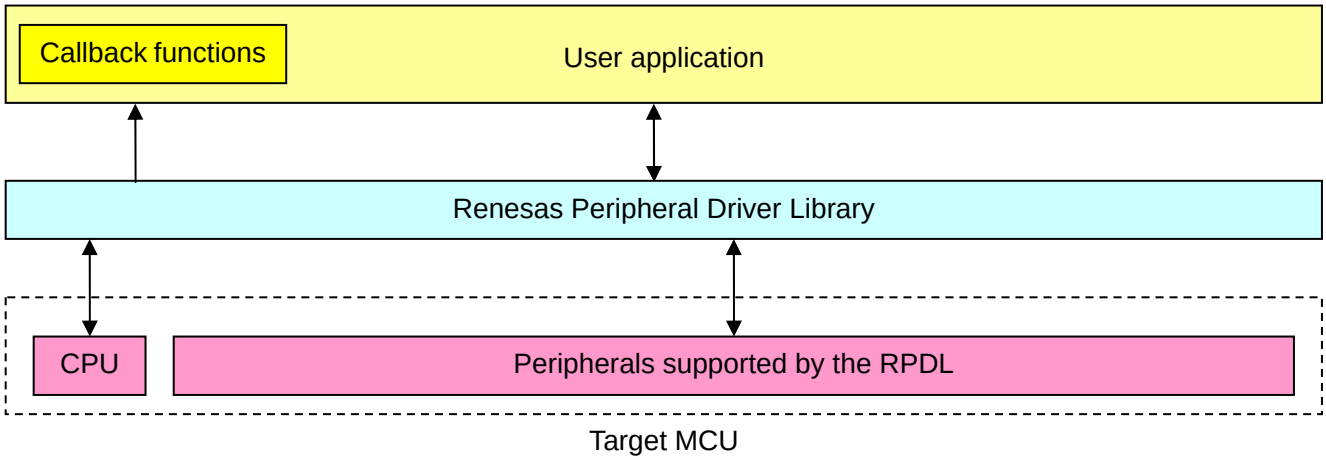


Figure 1-1: System configuration, with all peripherals supported by RPDL

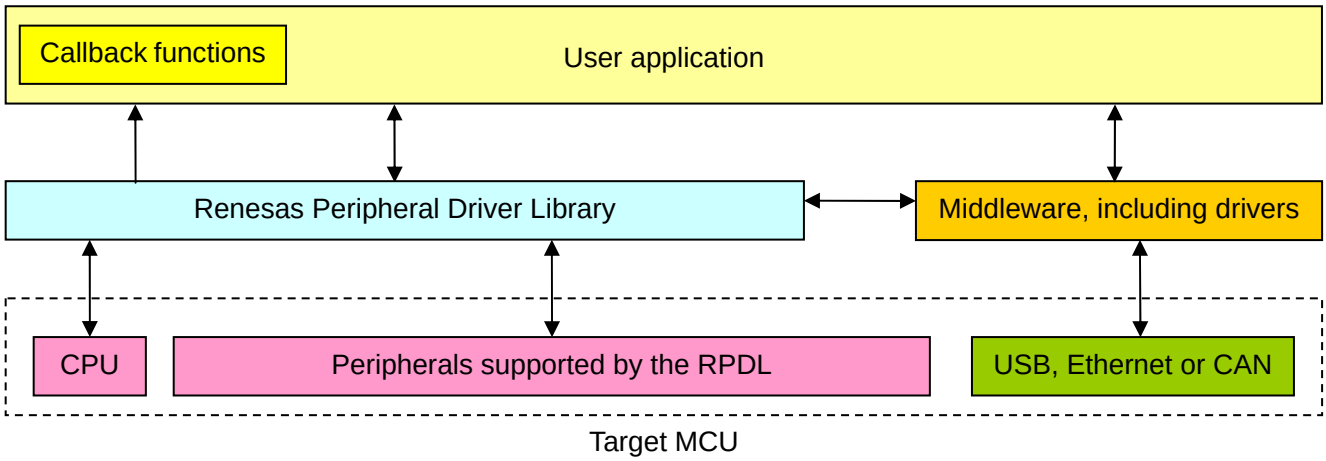


Figure 1-2: System configuration, with middleware taking direct control of some peripherals

The library is packaged as:

a) A binary file containing all of the peripheral driver functions,
b) Header files containing the information that the user needs to call any of the functions from their own application code and
c) Interrupt handlers supplied as source code.

For best use of this library, It is required that the user will have the following documents as a minimum:

i. The schematic
ii. The MCU hardware manual
iii. This RPDL API User's manual

The binary file is produced using the Renesas RX C compiler. It should be usable by another linker that conforms to the Renesas Application Binary Interface.

The coding standards and naming conventions are specified by Renesas.

The driver source code is tested for compliance with the MISRA-C:2004 guidelines.

1.1. Using the library within your project

The driver library can be used:

1. Via the PDG graphical utility

PDG can be downloaded from www.renesas.com/pdg.
The directions for use of the PDG utility are given in the PDG manual.

2. Or added to a project by the user and used stand-alone.

To add the driver library to your project's build environment, you need to

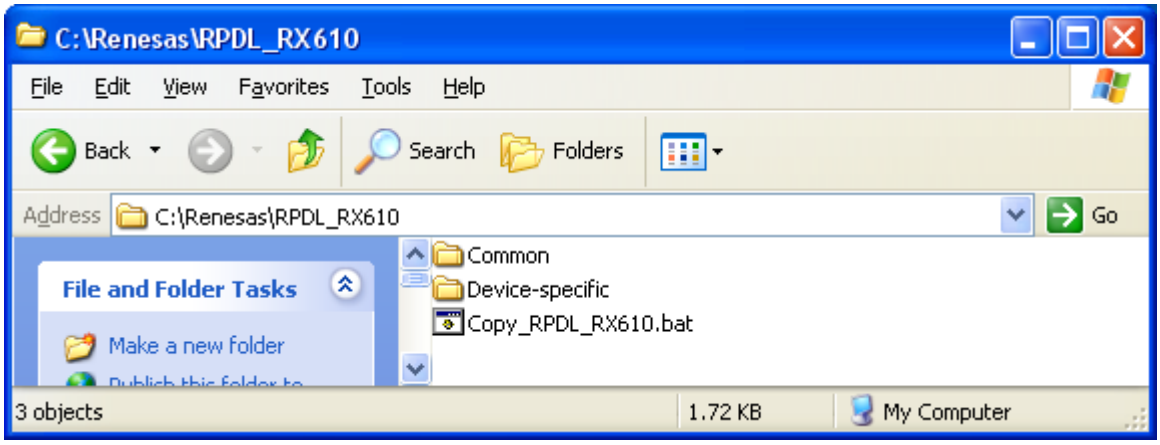
a) Unzip the RPDL distribution.
b) Copy the required source, header and library files into your project folder.
c) Include the required source files.
d) Add the driver library file to the linked files list.

1.1.1. Unzip the RPDL files

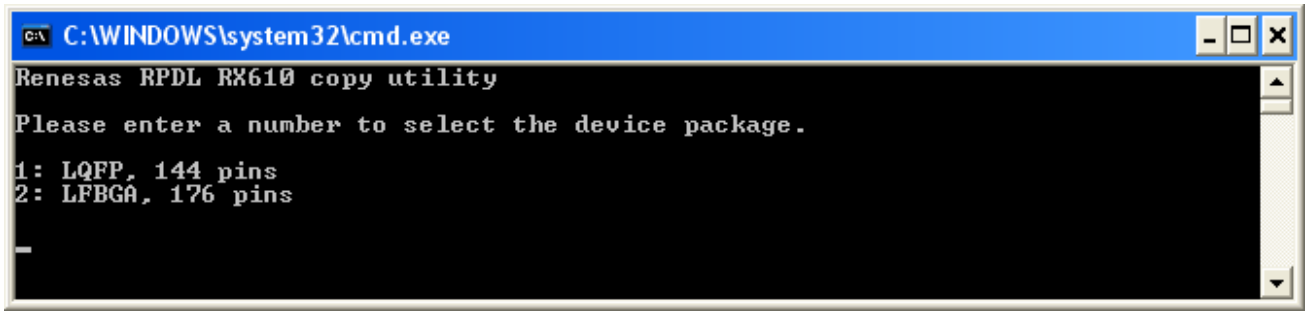
Double-click on the file RPDL_RX610.exe to unpack the files.
The default location is C:\Renesas\RPDL_RX610.

1.1.2. Copy the files into your project area

Navigate to where the RPDL files were unpacked.

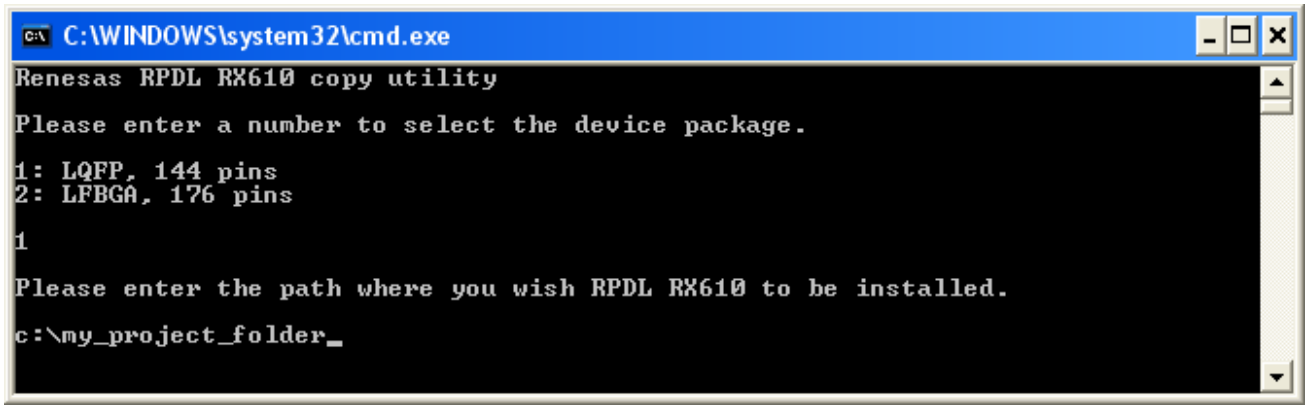


Double-click on "Copy_RPDL_RX610.bat" to start the copy process.

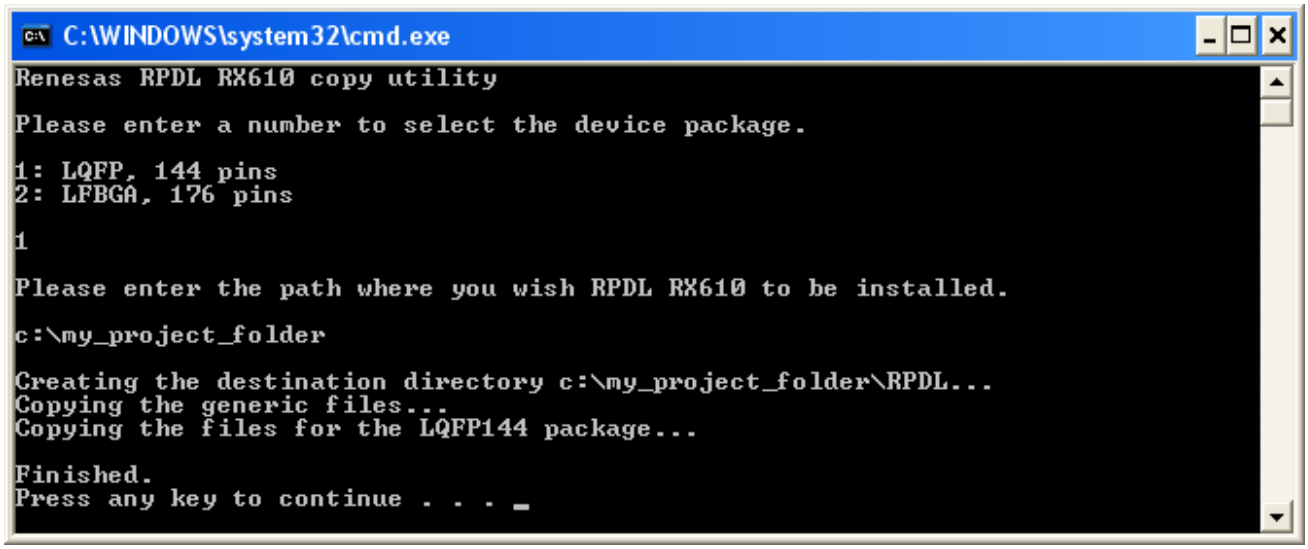


Select the device package option by pressing a number, and then press Enter.

Type the full path to the folder where you wish RPDL to be copied to, and then press Enter.



The utility will create a folder in the location that you specified and copy the files into the new folder.

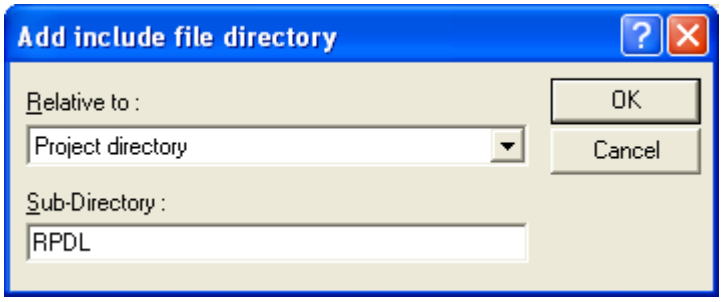


Press any key to close the window.

1.1.3. Include the new directory

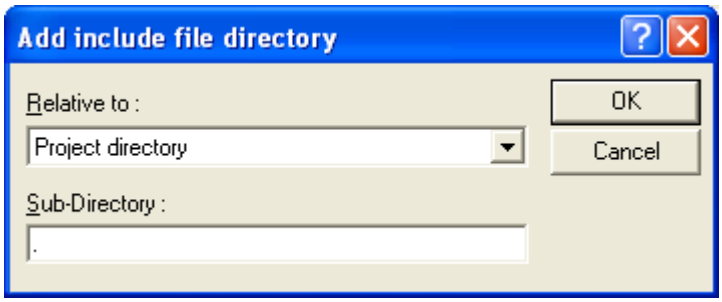
Use the key sequence Alt, B, R to open the “RX Standard Toolchain” window.
Select the C/C++ tab.
Use the key sequence S, I to show the included file directories.

Click on the “Add…” button.
In the “Add include file directory” window, enter the details as shown:



Click on “OK” to close the window.

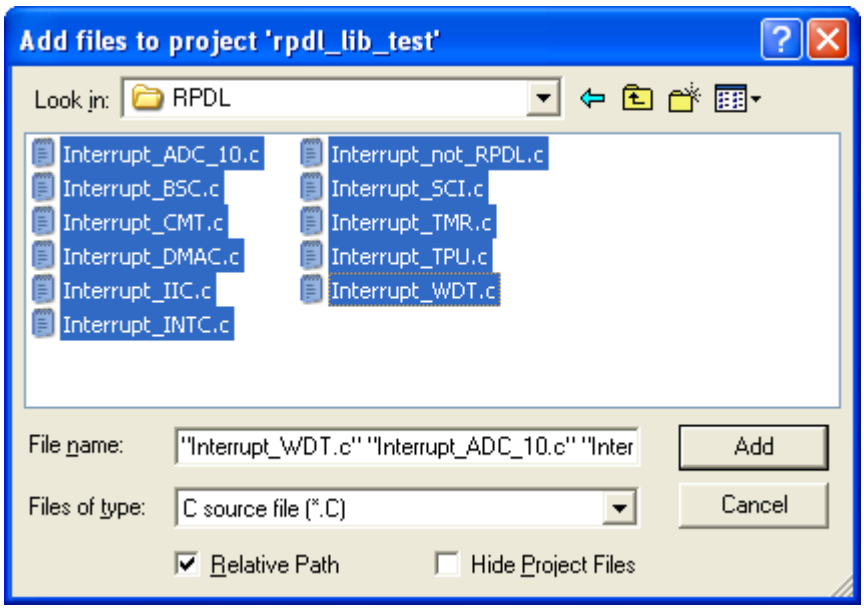
Click on the “Add…” button.
In the “Add include file directory” window, enter the details as shown:



Click on “OK” to close the window.
Click on “OK” to return to the main HEW window.

1.1.4. Include the new source files

Use the key sequence Alt, P, A to open the “Add files to project ‘<your project’” window.
Double click on the RPDL folder.
From the “Files of type” drop-down list, select “C source file (*.C)”.
Select all of the files, as shown below.



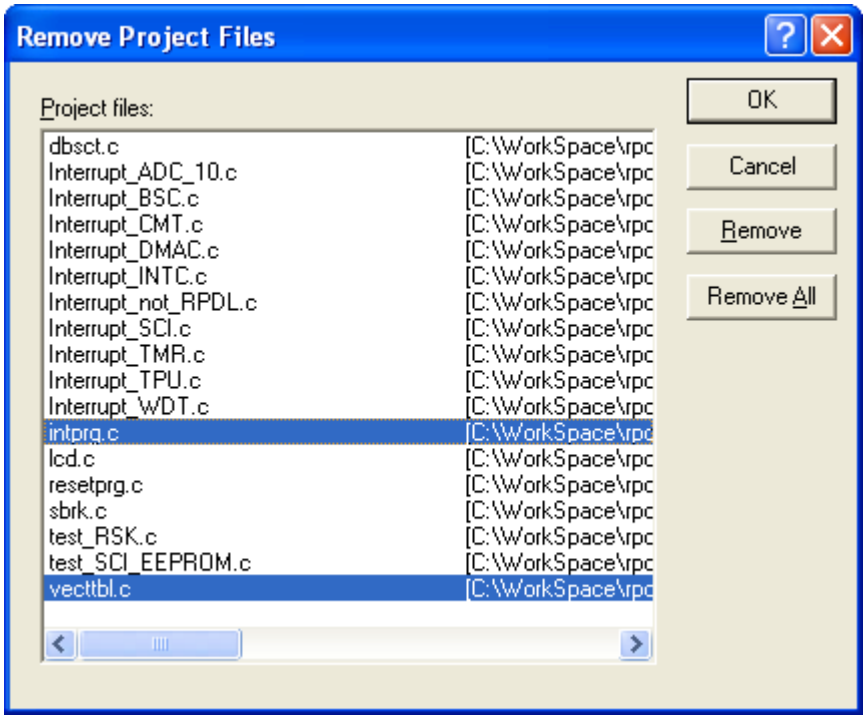
Click on “Add”.
Click on “OK” to return to the main HEW window.

1.1.5. Avoid conflicts with standard project files.

If the files ‘intprg.c’ or ‘vecttbl.c’ are included in the project, remove or exclude them.

1) Removal

Use the key sequence Alt, P, R to open the “Remove Project Files” window.
Select the files and click on Remove.



2) Exclusion

Select the two files and use the key sequence Alt, B, I to exclude them.

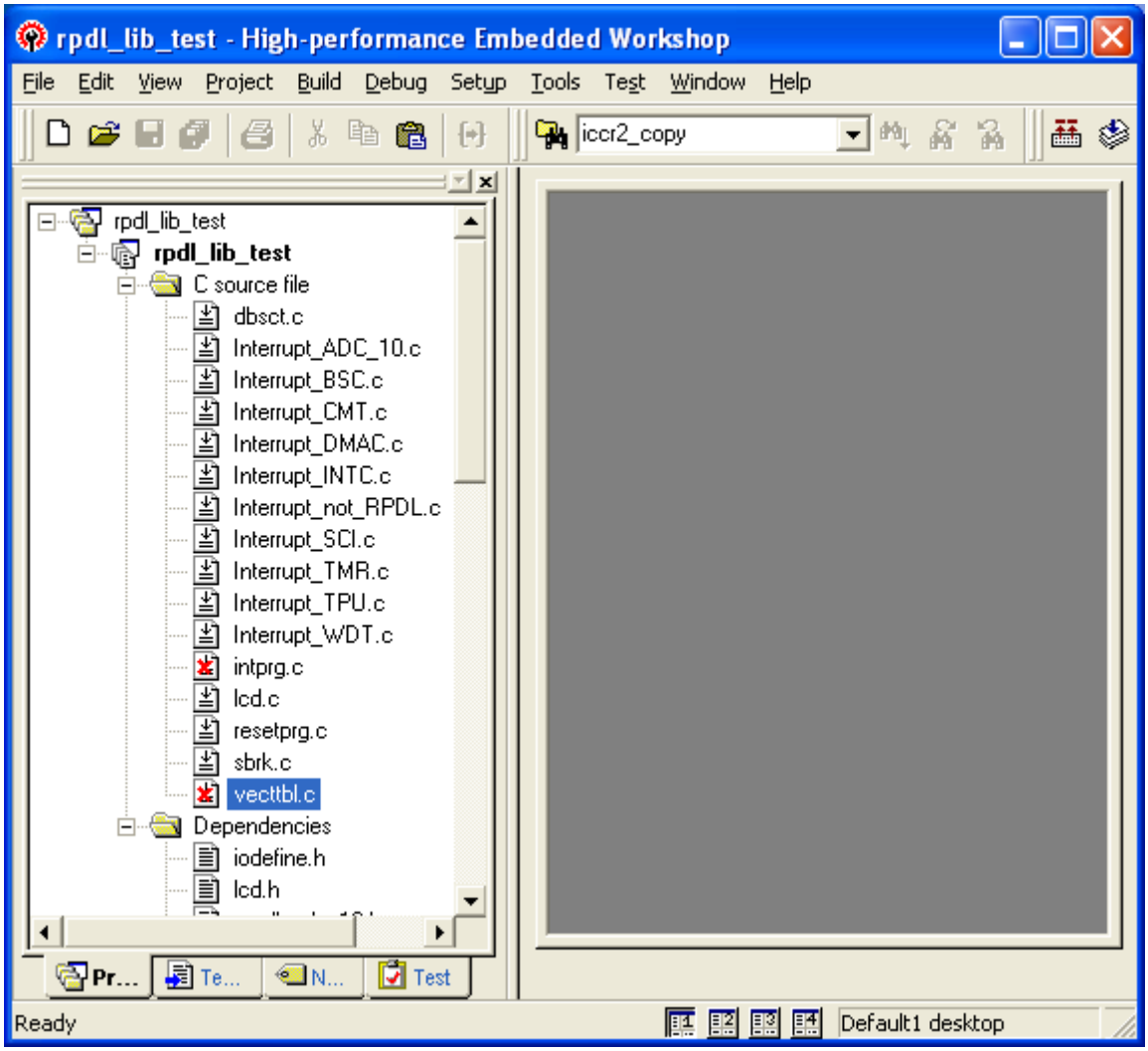
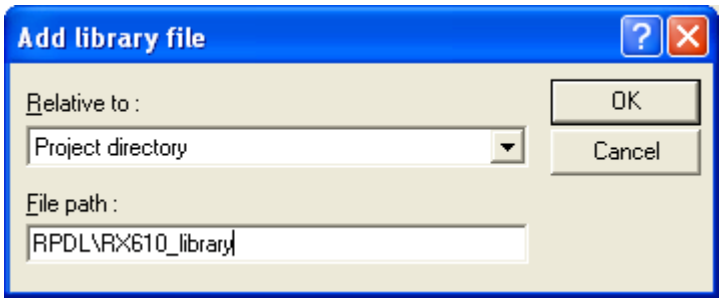


Figure 1-3: intprg.c and vecttbl.c have been excluded

1.1.6. Add the library file path

The library file is added to the list used by the linker application.

Use the key sequence Alt, B, R to open the “RX Standard Toolchain” window.
Select the Link/Library tab.
From the “Show entries for :” drop-down menu, select “Library files”.
Click on the “Add…” button.
In the “Add library file” window, enter the details as shown:



Click on “OK” to close the window.
Click on “OK” to return to the main HEW window.

1.1.7. Build the project

No further configuration should be required.
Simply build the project.

1.2. Document structure

The drivers are summarised in section 2 and explained in detail in section 4.

Section 5 provides usage examples.

Section 6 provides details which are specific to the RX CPU.

1.3. Acronyms and abbreviations

ADC	Analog to Digital Converter
API	Application Programming Interface
BCD	Binary-Coded Decimal
Bit	Binary digit
BSC	Bus State Controller
CMT	Compare Match Timer
CGC	Clock Generation Circuit
CMOS	Complementary MOS
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DMAC	DMA Controller
DSP	Digital Signal Processing
DTC	Data Transfer Controller
EEPROM	Electrically Eraseable and Programmable ROM
GSM	Global System for Mobile communications
HEW	High-performance Embedded Workbench
I²C	Inter-Integrated Circuit
INTC	Interrupt Controller
I/O	Input / Output
kB	Kilo Byte (1024 bytes)
LSB	Least-Significant Bit
LPC	Low Power Consumption
MCU	Microcontroller Unit
NMI	Non-Maskable Interrupt
NMOS	N-type MOS
MOS	Metal Oxide Semiconductor
MSB	Most-Significant Bit
PDG	Peripheral Driver Generator
PFC	Port Function Control
PPG	Programmable Pulse Generator
PWM	Pulse-Width Modulation
RAM	Random-Access Memory
ROM	Read-Only Memory
RPDL	Renesas Peripheral Driver Library
SCI	Serial Communications Interface
SMBus	System Management Bus
TPU	Timer Pulse Unit
WDT	Watchdog Timer

2. Driver

2.1. Overview

This library provides a set of peripheral function control programs (peripheral drivers) for Renesas microcontrollers and allows the peripheral driver to be built into a user program.

2.2. Control Functions summary

This library has the following control functions available as peripheral drivers.

(1) Clock Generation Circuit
These driver functions are used to configure the multiple internal clock signals.

(2) Interrupt
These driver functions are used for configuring the external interrupt pins, handling fixed interrupts and controlling the interrupt priority.

(3) I/O Port
These driver functions are used to configure the I/O pins and provide data read, write, compare and modify operations.

(4) Port Function
These driver functions are used for configuring the I/O pin optional functions.

(5) MCU Operation
These driver functions are used for configuring the MCU operation.

(6) Low Power Consumption
These driver functions are used for selecting lower power consumption.

(7) Bus Controller
These driver functions are used for configuring the external address bus, data bus and chip select pins and handling any bus errors.

(8) DMA Controller
These driver functions are used for configuring and controlling the transfer of data within the address space.

(9) Data Transfer Controller
These driver functions are used for configuring and controlling the transfer of data triggered by peripheral interrupts.

(10) Timer Pulse Unit
These driver functions are used for configuring and controlling the timers.

(11) Programmable Pulse Generator
These driver functions are used for configuring and controlling the pulse generator outputs.

(12) 8-bit Timer
These driver functions are used for configuring and controlling the timers.

(13) Compare Match Timer
These driver functions are used for configuring and controlling the timers.

(14) Watchdog Timer
These driver functions are used for configuring and controlling the timer.

(15) Serial Communication Interface
These driver functions are used to configure the serial channels and manage the transmission and / or

reception of data across them.

(16) CRC calculator
These driver functions are used for controlling the calculator.

(17) I²C Bus Interface
These driver functions are used for controlling the I²C bus channels.

(18) Analog to Digital Converter
These driver functions are used for configuring the ADC units, controlling the units and reading the conversion results.

(19) Digital to Analog converter
These driver functions are used for configuring the DAC module and setting the output voltages.

2.3. Clock Generation Circuit Driver

The driver functions support the control of the internal clock generator, providing the following operations.

1. Configuration of the multiple clock outputs for system, peripheral and external bus operation.

Note: Configuring the Clock Generation Circuit also provides information on clock frequencies that will be used by the integrated drivers for other peripherals.

2.4. Interrupt Control Driver

The driver functions support the use of the interrupt controller, providing the following operations.

1. Configuration an external interrupt pin for use.
2. Assigning an interrupt to be processed using the Fast Interrupt route.
3. Assigning handlers for the fixed exception interrupts.
4. Controlling an external interrupt input.
5. Reading the status of an external interrupt.
6. Reading an interrupt register.
7. Writing to an interrupt register.
8. Modifying an interrupt register.

2.5. I/O Port Driver

The driver functions support the use of the I/O port pins, providing the following operations.

1. Configuration for use.
2. Reading the pin or port configuration.
3. Modifying the pin or port configuration.
4. Reading a pin or 8-bit port value.
5. Writing to a pin or 8-bit port.
6. Comparing a pin or 8-bit port with a supplied value.
7. Modifying a pin or 8-bit port using a logical operation.
8. Waiting until a pin or 8-bit port matches a supplied value.

2.6. Port Function Control Driver

The driver functions support access to the Port Function Control (PFC) registers which select the mode of operation for some I/O pins.
The other driver functions modify the PFC registers automatically. For peripherals that are not supported by the driver library, these functions support:

1. Reading from a PFC register.
2. Writing to a PFC register.
3. Modifying a PFC register

2.7. MCU Operation Driver

The driver functions support access to the registers which select the mode of operation for the microcontroller. These functions support:

1. Controlling the on-chip ROM and RAM.
2. Reading the MCU status flags.

2.8. Low Power Consumption Driver

The driver functions support access to the registers which select the lower power modes of operation for the microcontroller. These functions support:

1. Configuring the state while in standby mode, and the activity that can be used to resume operation.
2. Selecting one of the low-power modes.
3. Writing data to the backup memory area.
4. Reading data from the backup memory area.
5. Determining the cause of the exit from the lowest power mode.

2.9. Bus Controller Driver

The driver functions support the control of the external bus, providing the following operations.

1. Configuration of the controller.
2. Configuration of the eight address space areas
3. Disabling an area that is not required.
4. Controlling the bus controller.
5. Reading the status of the controller.

2.10. DMA Controller Driver

The driver functions support the control of the Direct Memory Access (DMA) controller, providing the following operations.

1. Configuration for use, including
   - Access to all control bits.
   - Automatic interrupt control
2. Disabling DMA channels that are no longer required and enabling low-power mode.
3. Control of one or more channels.
4. Reading the status and operation registers of a channel.

2.11. Data Transfer Controller Driver

The driver functions support the control of the Data Transfer Controller, providing the following operations.

1. Setting the central options.
2. Configuration for use, including support for chain transfers.
3. Disabling the controller.
4. Starting or stopping the controller.
5. Reading the status flags and data transfer registers.

2.12. Timer Pulse Unit Driver

The driver functions support the use of the twelve 16-bit timers, providing the following operations.

1. Configuration for use, including
   - Access to all control bits.
   - Automatic interrupt control
   - Automatic I/O pin configuration
2. Disabling channels that are no longer required and enabling low-power mode.
3. Control of a timer.
4. Reading the status and registers of a timer.

Note: The Clock Generation Circuit must be configured before configuring any timer channel.

2.13. Programmable Pulse Generator Driver

The driver functions support the use of the pulse generator, providing the following operations.

1. Configuring the generator for use.
2. Disabling groups of outputs that are no longer required.
3. Control of the generator during run-time.

2.14. 8-bit Timer Driver

The driver functions support the use of the four 8-bit timers, providing the following operations.

1. Configuring a channel for use, using register values which have been determined elsewhere.
2. Configuring two channels as a 16-bit pair, using register values which have been determined elsewhere.
3. Configuration for as a periodic timer, including
   - Automatic clock setting using frequency or period as an input.
   - Automatic pulse width setting, using pulse width or duty cycle as an input.
   - Automatic interrupt control
   - I/O pin control
   - Automatic I/O pin configuration
4. Configuration for as a one-shot timer, including
   - Automatic clock setting, using pulse width as an input
   - Automatic interrupt control
   - CPU sleep option
   - I/O pin control
   - Automatic I/O pin configuration
   - Automatic support for using two channels as a single 16-bit timer.
5. Disabling channels that are no longer required and enabling low-power mode.
6. Control of a single timer channel.
7. Control of two timer channels when configured as one 16-bit channel.
8. Control of channels in periodic mode, enabling pulse-width modulation (PWM) output.
9. Reading the registers of a single timer channel.
10. Reading the registers of a 16-bit timer channel pair.

Note: The Clock Generation Circuit must be configured before configuring any timer channel.

2.15. Compare Match Timer Driver

The driver functions support the use of the four 16-bit timers, providing the following operations.

1. Configuration for use, including

   - Automatic clock setting using frequency or period as an input.
   - Manual clock setting using register values as inputs.
   - Automatic interrupt control

2. Configuration for use as a one-shot timer.

3. Disabling channels that are no longer required and enabling low-power mode.

4. Control of a timer, including constant register updates.

5. Control of a timer, including change of frequency.

Note: The Clock Generation Circuit must be configured before configuring any timer channel.

2.16. Watchdog Timer Driver

The driver functions support the use of the watchdog timer, providing the following operations.

4. Configuring the timer for use, including
    - Automatic clock setting using frequency or period as an input.
    - Internal timer mode
    - Watchdog timer mode
    - MCU reset generation while in watchdog timer mode
    - Automatic interrupt control
5. Control of the timer, including
    - Stopping the timer
    - Counter refresh to prevent overflow
6. Reading the timer status and counter register.

Note: The Clock Generation Circuit should be configured before configuring this timer.

2.17. Serial Communication Interface Driver

The driver functions support the use of the seven serial communication channels, providing the following operations.

1. Configuration for use, including
   - Automatic baud rate clock calculations
   - Automatic interrupt control
   - Automatic I/O pin configuration
2. Disabling channels that are no longer required and enabling low-power mode.
3. Transmitting data, with polling or interrupt mode automatically selected.
4. Receiving data, with polling or interrupt mode automatically selected.
5. Control the channel operation.
6. Reading the status flags.

Note: The Clock Generation Circuit must be configured before configuring any serial channel.

2.18. CRC Calculator Driver

The driver functions support the CRC calculator, providing the following operations.

1. Configuration for use, including
   - Polynomial selection.
   - Bit order selection.
   - Preparation for a new calculation.
2. Disabling the calculator and enabling low-power mode.
3. Writing data to be used for the calculation.
4. Reading the calculation result.

2.19. I²C Bus Interface Driver

The driver functions support the use of the two I²C modules, providing the following operations.

1. Configuration for use, including
   - Automatic clock setting using transfer rate as an input.
   - Automatic interrupt control
   - Automatic I/O pin configuration
2. Disabling modules that are no longer required and enabling low-power mode.
3. Transmitting data in Master mode.
4. Receiving data in Master mode.
5. Completing the reception of data that has utilised the DMAC or DTC.
6. Monitoring the bus and handling the reception of data in Slave mode.
7. Transmitting data in Slave mode.
8. Control of one or more units, including bus lock-up recovery support.
9. Reading the status of a module.

Note: The Clock Generation Circuit must be configured before configuring any I²C module.

2.20. 10-bit Analog to Digital Converter Driver

The driver functions support the use of the four ADC units, providing the following operations.

1. Configuration for use, including
   - Automatic clock setting using sampling time as an input.
   - Automatic interrupt control
   - Automatic I/O pin configuration
2. Disabling units that are no longer required and enabling low-power mode.
3. Control of one or more units, including
   - CPU sleep option
4. Reading the conversion results of one or more units, with support for polling or interrupts.

Note: The Clock Generation Circuit must be configured before configuring any ADC unit.

2.21. 10-bit Digital to Analog Converter Driver

The driver functions support the use of the DAC module, providing the following operations.

1. Configuring a channel for use, including
   - Independent or linker operation
   - Data alignment
2. Disabling channels that are no longer required and enabling low-power mode.
3. Writing data to a channel.

3. Types and definitions

3.1. Data types

This section describes the data types used in this library. For details about the setting values, refer to the section "4.2 Description of Each API".

The header files stdint.h and stdbool.h are included with the Renesas RX compiler.

Table 1: Data types

Type	Defined in	Description	Range
bool	stdbool.h	Boolean	0 (false) to 1 (true)
float	C	Floating point, 32 bits	-∞ to +∞
uint8_t	stdint.h	Unsigned, 8 bits	0 to 255
uint16_t		Unsigned, 16 bits	0 to 2^{15} - 1
uint32_t		Unsigned, 32 bits	0 to 2^{32} - 1

3.2. General definitions

3.2.1. PDL_NO_FUNC

Used as a parameter when there is no applicable function.

3.2.2. PDL_NO_PTR

Used as a parameter when there is no applicable data location.

3.2.3. PDL_NO_DATA

Used as a parameter when there is no applicable data value.

3.2.4. PDL_MCU_GROUP

The family supported by this build of the driver library. It is defined as RX610.

A usage example is:

```
#if PDL_MCU_GROUP != RX610
 #error "Wrong RPDL !"
#endif
```

3.2.5. PDL_VERSION

The version number of the RPDL library. The number is stored in BCD format (xx.xx). For example, 0100h is v1.00.

A usage example is:

```
const uint16_t rpdl_version_number = PDL_VERSION;
```

4. Library Reference

4.1. API List by Peripheral Function

Table 4.1 lists the Renesas Embedded APIs by peripheral function.

Table 4.1 Renesas Embedded API List

Category	Number	Name	Description
Clock Generation Circuit	1	R_CGC_Set	Configure the clock generation circuit.
Interrupt controller	1	R_INTC_CreateExtInterrupt	Configure an external interrupt pin.
	2	R_INTC_CreateFastInterrupt	Assign handlers for the fixed-vector interrupts.
	3	R_INTC_CreateExceptionHandlers	Enable faster interrupt processing for one interrupt.
	4	R_INTC_ControlExtInterrupt	External interrupt control.
	5	R_INTC_GetExtInterruptStatus	Read the external interrupt status.
	6	R_INTC_Read	Read an interrupt register.
	7	R_INTC_Write	Update an interrupt register.
	8	R_INTC_Modify	Modify an interrupt register.
I/O port	1	R_IO_PORT_Set	Configure an I/O port.
	2	R_IO_PORT_ReadControl	Read an I/O port's control registers.
	3	R_IO_PORT_ModifyControl	Modify an I/O port's control registers.
	4	R_IO_PORT_Read	Read data from an I/O port.
	5	R_IO_PORT_Write	Write data to an I/O port.
	6	R_IO_PORT_Compare	Check the pin states on an I/O port.
	7	R_IO_PORT_Modify	Modify the pin states on an I/O port.
	8	R_IO_PORT_Wait	Wait for a match on an I/O port.
Port Function Control	1	R_PFC_Read	Read a PFC register.
	2	R_PFC_Write	Write to a PFC register.
	3	R_PFC_Modify	Modify a PFC register.
MCU operation	1	R_MCU_Control	Control the operation of the MCU.
	2	R_MCU_GetStatus	Read the MCU status.
Low Power Consumption	1	R_LPC_Create	Configure the MCU low power conditions.
	2	R_LPC_Control	Select a low power consumption mode.
	3	R_LPC_WriteBackup	Write to the Backup registers.
	4	R_LPC_ReadBackup	Read from the Backup registers.
	5	R_LPC_GetStatus	Read the status flags.
Bus Controller	1	R_BSC_Create	Configure the external bus controller.
	2	R_BSC_CreateArea	Configure an external bus area.
	3	R_BSC_Destroy	Stop the Bus Controller.
	4	R_BSC_Control	Modify the External Bus Controller operation.
	5	R_BSC_GetStatus	Read the External Bus Controller status flags.
DMA Controller	1	R_DMAC_Create	Configure the DMA controller.
	2	R_DMAC_Destroy	Disable a DMA channel.
	3	R_DMAC_Control	Control the DMA controller.
	4	R_DMAC_GetStatus	Check the status of the DMA channel.
Data Transfer Controller	1	R_DTC_Set	Set the Data Transfer Controller options.
	2	R_DTC_Create	Configure the DTC for a transfer.
	3	R_DTC_Destroy	Shutdown the Data Transfer Controller.
	4	R_DTC_Control	Control the Data Transfer Controller.
	5	R_DTC_GetStatus	Check the status of the Data Transfer Controller.
Timer Pulse Unit	1	R_TPU_Create	Configure a Timer Pulse Unit channel.
	2	R_TPU_Destroy	Shut down a timer pulse unit.
	3	R_TPU_Control	Control a timer channel.
	4	R_TPU_Read	Read from timer channel registers.
Programmable Pulse Generator	1	R_PPG_Create	Configure a PPG group.
	2	R_PPG_Destroy	Disable a PPG unit.
	3	R_PPG_Control	Control a PPG group.

8-bit Timer	1	R_TMR_CreateChannel	Configure a TMR timer channel.
	2	R_TMR_CreateUnit	Configure a TMR timer unit.
	3	R_TMR_CreatePeriodic	Select periodic operation.
	4	R_TMR_CreateOneShot	Configure and use a one-shot timer.
	5	R_TMR_Destroy	Disable a TMR timer unit.
	6	R_TMR_ControlChannel	Write to timer channel registers.
	7	R_TMR_ControlUnit	Write to timer unit registers.
	8	R_TMR_ControlPeriodic	Control periodic operation.
	9	R_TMR_ReadChannel	Read from timer channel registers.
	10	R_TMR_ReadUnit	Read from timer unit registers.
Compare Match Timer	1	R_CMT_Create	Configure a CMT channel.
	2	R_CMT_CreateOneShot	Configure a CMT channel as a one-shot event.
	3	R_CMT_Destroy	Disable a CMT unit.
	4	R_CMT_Control	Control CMT operation.
	5	R_CMT_Read	Read CMT channel status and registers.
Watchdog Timer	1	R_WDT_Create	Configure the Watchdog timer.
	2	R_WDT_Control	Control the Watchdog operation.
	3	R_WDT_Read	Read the Watchdog timer status and registers.
Serial Communication Interface	1	R_SCI_Create	SCI channel setup.
	2	R_SCI_Destroy	Shut down a SCI channel.
	3	R_SCI_Send	Send a string of characters.
	4	R_SCI_Receive	Receive a string of characters.
	5	R_SCI_Control	Control the SCI channel.
	6	R_SCI_GetStatus	Check the status of a SCI channel.
CRC calculator	1	R_CRC_Create	Configure the CRC calculator.
	2	R_CRC_Destroy	Shut down the CRC calculator.
	3	R_CRC_Write	Write data into the CRC calculation register.
	4	R_CRC_Read	Read the CRC calculation result.
I²C bus interface	1	R_IIC_Create	I²C channel setup.
	2	R_IIC_Destroy	Disable an I²C channel.
	3	R_IIC_MasterSend	Write data to a slave device.
	4	R_IIC_MasterReceive	Read data from a slave device.
	5	R_IIC_MasterReceiveLast	Complete a DMAC or DTC-based read process.
	6	R_IIC_SlaveMonitor	Monitor the bus and receive data from a master.
	7	R_IIC_SlaveSend	Write data to a master device.
	8	R_IIC_Control	I²C channel control.
	9	R_IIC_GetStatus	Read the status for an I²C channel.
10-bit Analog to Digital converter	1	R_ADC_10_Create	Configure an ADC unit.
	2	R_ADC_10_Destroy	Shut down an ADC unit.
	3	R_ADC_10_Control	Start or stop an ADC unit.
	4	R_ADC_10_Read	Read the ADC conversion results.
10-bit Digital to Analog converter	1	R_DAC_10_Create	Configure the 10-bit DAC module.
	2	R_DAC_10_Destroy	Disable a DAC channel.
	3	R_DAC_10_Write	Write data to a DAC channel.

4.2. Description of Each API

This section describes each API and explains how to use them, showing a program example for each. The description of each API is divided into the following items.

Synopsis	Summarises processing by the API function.
Prototype	The function format and a brief explanation of the arguments.
Description	Explains how to use the API function and shows assignable parameters separating each argument with **[*argument*]**.
Return value	Describes the returned value of the API function.
Category	Indicates the category of the API function.
Reference	Indicates the API functions to be referred.
Remark	Describes notes to use the API function.
Program example	Represents how to use the API function by a program example. Two examples of return value checking are shown below.

```
/* RPDL definitions */
#include “r_pdl_pfc.h”
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    bool result;

    /* Write 0xFF to register PFC1 */
    result = (R_PFC_Write(
        1,
        0xFF
    );
    if (result == false)
    {
        /* Handle the error here */
    }

    /* Keep trying to send a string (if the channel is busy) */
    do
    {
        result = R_SCI_Send(
            2,
            “Renesas RX”,
            NULL,
            PDL_NO_FUNC
        );
    } while (result == false);
)
```

For clarity, the return value is not checked in the examples used in this manual.

4.2.1. Clock Generation Circuit

1) R_CGC_Set

Synopsis

Configure the clock generation circuit.

Prototype

```
bool R_CGC_Set(
    uint32_t data1,    // Input frequency
    uint32_t data2,    // System clock frequency
    uint32_t data3,    // Peripheral module clock frequency
    uint32_t data4,    // External bus clock frequency
    uint8_t data5      // Configuration options
);
```

Description

Set the clock output frequencies and options.

[data1]
The frequency of the main clock oscillator in Hertz.

[data2]
The desired frequency of the System clock (ICLK) in Hertz.

[data3]
The desired frequency of the Peripheral module clock (PCLK) in Hertz.

[data4]
The desired frequency of the External bus clock (BCLK) in Hertz.
Specify 0 if the value is not important.

[data5]
Configuration options. The default setting is shown in **bold**.

- BCLK pin output control

PDL_CGC_BCLK_ENABLE or **PDL_CGC_BCLK_DISABLE** or PDL_CGC_BCLK_HIGH	Output the external bus clock signal on pin BCLK, leave the BCLK pin as an input or fix the BCLK pin high.

Return value

True if all parameters are valid and exclusive; otherwise false.
For RX610, the following rules shall be checked:

- Main clock oscillator frequency: 8 to 14 MHz.
- f_{ICLK}: 8 to 100 MHz
- f_{PCLK}: 8 to 50 MHz
- f_{BCLK}: 8 to 25 MHz
- $f_{ICLK} \geq f_{PCLK}$ and $f_{ICLK} \geq f_{BCLK}$
- f_{ICLK}, f_{PCLK} and f_{BCLK} are achievable: (main clock oscillator x 8) ÷ 1, 2, 4 or 8

Functionality

Clock generation circuit

References

None.

Remarks

- This function must be called before configuring clock-dependent modules.
- This function modifies the BCLK pin for input or output.

Program example

```
/* RPDL definitions */
#include “r_pdl_cgc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure operation using a 12.5 MHz input clock */
    /* ICLK = 100 MHz, PCLK = 50 MHz, BCLK = 25 MHz */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_ENABLE
    );
}
```

4.2.2. Interrupt Control Unit

1) R_INTC_CreateExtInterrupt

Synopsis

Configure an external interrupt pin.

Prototype

```
bool R_INTC_CreateExtInterrupt(
    uint8_t data1,      // Pin selection
    uint16_t data2,     // Configuration
    void * func         // Callback function
    uint8_t data3,      // Interrupt priority level
);
```

Description

Sets the specified external interrupt.

[data1]
Choose the interrupt pin to be configured.

PDL_INTC_IRQn (n = 0 to 15) or PDL_INTC_NMI	IRQn (n = 0 to 15) interrupt pin or NMI interrupt pin

[data2]
Choose the pin settings. If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**.

- Detection sense selection (for the IRQ pins)

PDL_INTC_LOW or PDL_INTC_FALLING or PDL_INTC_RISING or PDL_INTC_BOTH	Select Low level, Falling edge, Rising edge or Falling and rising edge detection.

- Detection sense selection (for the NMI pin)

PDL_INTC_FALLING or PDL_INTC_RISING	Falling or rising edge detection.

- Alternate pin selection (for the IRQ pins)

PDL_INTC_A or PDL_INTC_B	Select the IRQn-A or IRQn-B pin to be used.

- DMAC / DTC trigger control. Not enabled if low-level detection is selected.

PDL_INTC_DMAC_DTC_TRIGGER_DISABLE or PDL_INTC_DMAC_TRIGGER_ENABLE or PDL_INTC_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC (valid for n = 0 to 3) or DTC (valid for n = 0 to 15) when a valid edge transition is detected on an IRQn pin.

[func]
The function to be called when a valid condition is detected.
Specify PDL_NO_FUNC if no IRQn interrupt is required.
A function must be specified for the NMI pin.

[data3]
The IRQn interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.
This value does not apply to the NMI pin and is ignored.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

External interrupt

Reference

R_INTC_ControlExtInterrupt, R_INTC_GetExtInterruptStatus

Remarks

- The selected interrupt pin is enabled automatically.
- Port Function Control registers PFCR8 or PFCR9 are modified to select the IRQn pin.
- The appropriate I/O port ICR and DDR registers are modified.
- Please see the notes on callback function use in §6.
- The NMI callback function should not return. It should stop operation or reset the system.
- If the NMI interrupt fails to initialise, this function will return false.

Program example

```
/* RPDL definitions */
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Declaration of callback function */
void CallBackFunc( void );

void func( void )
{
    /* Configure the IRQ0 interrupt on pin IRQ0-A */
    R_INTC_CreateExtInterrupt(
        PDL_INTC_IRQ0,
        PDL_INTC_FALLING | PDL_INTC_A,
        CallBackFunc,
        7
    );
}
```

2) R_INTC_CreateFastInterrupt

Synopsis

Enable faster interrupt processing for one interrupt.

Prototype

```
bool R_INTC_CreateFastInterrupt(
    uint8_t data      // The interrupt to be selected
);
```

Description (1/3)

[data]

Choose the interrupt vector to be processed using the fast interrupt process.

Name	Module	Interrupt cause
PDL_INTC_VECTOR_BUSERR	External bus	Error (illegal access or timeout)
PDL_INTC_VECTOR_FIFERR	Flash memory	Error
PDL_INTC_VECTOR_FRDYI		Ready
PDL_INTC_VECTOR_CMT0	Compare match timer	Compare match on channel n (n = 0 to 3)
PDL_INTC_VECTOR_CMT1		
PDL_INTC_VECTOR_CMT2		
PDL_INTC_VECTOR_CMT3		
PDL_INTC_VECTOR_IRQ0	External interrupt pin	Valid edge or level detected on pin IRQn (n = 0 to 15)
PDL_INTC_VECTOR_IRQ1		
PDL_INTC_VECTOR_IRQ2		
PDL_INTC_VECTOR_IRQ3		
PDL_INTC_VECTOR_IRQ4		
PDL_INTC_VECTOR_IRQ5		
PDL_INTC_VECTOR_IRQ6		
PDL_INTC_VECTOR_IRQ7		
PDL_INTC_VECTOR_IRQ8		
PDL_INTC_VECTOR_IRQ9		
PDL_INTC_VECTOR_IRQ10		
PDL_INTC_VECTOR_IRQ11		
PDL_INTC_VECTOR_IRQ12		
PDL_INTC_VECTOR_IRQ13		
PDL_INTC_VECTOR_IRQ14		
PDL_INTC_VECTOR_IRQ15		
PDL_INTC_VECTOR_WOVI	Watchdog timer	Overflow
PDL_INTC_VECTOR_ADI0	Analog to Digital converter	Conversion completed on unit n (n = 0 to 3)
PDL_INTC_VECTOR_ADI1		
PDL_INTC_VECTOR_ADI2		
PDL_INTC_VECTOR_ADI3		
PDL_INTC_VECTOR_TGI0A	Timer Pulse Unit, channel 0	Input capture or compare match A
PDL_INTC_VECTOR_TGI0B		Input capture or compare match B
PDL_INTC_VECTOR_TGI0C		Input capture or compare match C
PDL_INTC_VECTOR_TGI0D		Input capture or compare match D
PDL_INTC_VECTOR_TCI0V		Overflow
PDL_INTC_VECTOR_TGI1A	Timer Pulse Unit, channel 1	Input capture or compare match A
PDL_INTC_VECTOR_TGI1B		Input capture or compare match B
PDL_INTC_VECTOR_TCI1V		Overflow
PDL_INTC_VECTOR_TCI1U		Underflow
PDL_INTC_VECTOR_TGI2A	Timer Pulse Unit, channel 2	Input capture or compare match A
PDL_INTC_VECTOR_TGI2B		Input capture or compare match B
PDL_INTC_VECTOR_TCI2V		Overflow
PDL_INTC_VECTOR_TCI2U		Underflow
PDL_INTC_VECTOR_TGI3A	Timer Pulse Unit, channel 3	Input capture or compare match A
PDL_INTC_VECTOR_TGI3B		Input capture or compare match B
PDL_INTC_VECTOR_TGI3C		Input capture or compare match C
PDL_INTC_VECTOR_TGI3D		Input capture or compare match D
PDL_INTC_VECTOR_TCI3V		Overflow
PDL_INTC_VECTOR_TGI4A	Timer Pulse Unit, channel 4	Input capture or compare match A
PDL_INTC_VECTOR_TGI4B		Input capture or compare match B
PDL_INTC_VECTOR_TCI4V		Overflow
PDL_INTC_VECTOR_TCI4U		Underflow

Description (2/3)

PDL_INTC_VECTOR_TGI5A	Timer Pulse Unit, channel 5	Input capture or compare match A
PDL_INTC_VECTOR_TGI5B		Input capture or compare match B
PDL_INTC_VECTOR_TCI5V		Overflow
PDL_INTC_VECTOR_TCI5U		Underflow
PDL_INTC_VECTOR_TGI6A	Timer Pulse Unit, channel 6	Input capture or compare match A
PDL_INTC_VECTOR_TGI6B		Input capture or compare match B
PDL_INTC_VECTOR_TGI6C		Input capture or compare match C
PDL_INTC_VECTOR_TGI6D		Input capture or compare match D
PDL_INTC_VECTOR_TCI6V		Overflow
PDL_INTC_VECTOR_TGI7A	Timer Pulse Unit, channel 7	Input capture or compare match A
PDL_INTC_VECTOR_TGI7B		Input capture or compare match B
PDL_INTC_VECTOR_TCI7V		Overflow
PDL_INTC_VECTOR_TCI7U		Underflow
PDL_INTC_VECTOR_TGI8A	Timer Pulse Unit, channel 8	Input capture or compare match A
PDL_INTC_VECTOR_TGI8B		Input capture or compare match B
PDL_INTC_VECTOR_TCI8V		Overflow
PDL_INTC_VECTOR_TCI8U		Underflow
PDL_INTC_VECTOR_TGI9A	Timer Pulse Unit, channel 9	Input capture or compare match A
PDL_INTC_VECTOR_TGI9B		Input capture or compare match B
PDL_INTC_VECTOR_TGI9C		Input capture or compare match C
PDL_INTC_VECTOR_TGI9D		Input capture or compare match D
PDL_INTC_VECTOR_TCI9V		Overflow
PDL_INTC_VECTOR_TGI10A	Timer Pulse Unit, channel 10	Input capture or compare match A
PDL_INTC_VECTOR_TGI10B		Input capture or compare match B
PDL_INTC_VECTOR_TCI10V		Overflow
PDL_INTC_VECTOR_TCI10U		Underflow
PDL_INTC_VECTOR_TGI11A	Timer Pulse Unit, channel 11	Input capture or compare match A
PDL_INTC_VECTOR_TGI11B		Input capture or compare match B
PDL_INTC_VECTOR_TCI11V		Overflow
PDL_INTC_VECTOR_TCI11U		Underflow
PDL_INTC_VECTOR_CMIA0	8-bit timer TMR, channel 0	Compare match A
PDL_INTC_VECTOR_CMIB0		Compare match B
PDL_INTC_VECTOR_OVI0		Overflow
PDL_INTC_VECTOR_CMIA1	8-bit timer TMR, channel 1	Compare match A
PDL_INTC_VECTOR_CMIB1		Compare match B
PDL_INTC_VECTOR_OVI1		Overflow
PDL_INTC_VECTOR_CMIA2	8-bit timer TMR, channel 2	Compare match A
PDL_INTC_VECTOR_CMIB2		Compare match B
PDL_INTC_VECTOR_OVI2		Overflow
PDL_INTC_VECTOR_CMIA3	8-bit timer TMR, channel 3	Compare match A
PDL_INTC_VECTOR_CMIB3		Compare match B
PDL_INTC_VECTOR_OVI3		Overflow
PDL_INTC_VECTOR_DMTEND0	Direct memory access controller	Transfer complete on channel n (n = 0 to 3)
PDL_INTC_VECTOR_DMTEND1		
PDL_INTC_VECTOR_DMTEND2		
PDL_INTC_VECTOR_DMTEND3		
PDL_INTC_VECTOR_ERI0	SCI, channel 0	Error in data received
PDL_INTC_VECTOR_RXI0		Data received
PDL_INTC_VECTOR_TXI0		Start of next data transfer
PDL_INTC_VECTOR_TEI0		End of data transfer
PDL_INTC_VECTOR_ERI1	SCI, channel 1	Error in data received
PDL_INTC_VECTOR_RXI1		Data received
PDL_INTC_VECTOR_TXI1		Start of next data transfer
PDL_INTC_VECTOR_TEI1		End of data transfer
PDL_INTC_VECTOR_ERI2	SCI, channel 2	Error in data received
PDL_INTC_VECTOR_RXI2		Data received
PDL_INTC_VECTOR_TXI2		Start of next data transfer
PDL_INTC_VECTOR_TEI2		End of data transfer
PDL_INTC_VECTOR_ERI3	SCI, channel 3	Error in data received
PDL_INTC_VECTOR_RXI3		Data received
PDL_INTC_VECTOR_TXI3		Start of next data transfer
PDL_INTC_VECTOR_TEI3		End of data transfer

Description (3/3)

PDL_INTC_VECTOR_ERI4	SCI, channel 4	Error in data received
PDL_INTC_VECTOR_RXI4		Data received
PDL_INTC_VECTOR_TXI4		Start of next data transfer
PDL_INTC_VECTOR_TEI4		End of data transfer
PDL_INTC_VECTOR_ERI5	SCI, channel 5	Error in data received
PDL_INTC_VECTOR_RXI5		Data received
PDL_INTC_VECTOR_TXI5		Start of next data transfer
PDL_INTC_VECTOR_TEI5		End of data transfer
PDL_INTC_VECTOR_ERI6	SCI, channel 6	Error in data received
PDL_INTC_VECTOR_RXI6		Data received
PDL_INTC_VECTOR_TXI6		Start of next data transfer
PDL_INTC_VECTOR_TEI6		End of data transfer
PDL_INTC_VECTOR_ICEEI0	I²C bus interface, channel 0	Transfer error or event generation
PDL_INTC_VECTOR_ICRXI0		Data received
PDL_INTC_VECTOR_ICTXI0		Start of next data transfer
PDL_INTC_VECTOR_ICTEI0		End of data transfer
PDL_INTC_VECTOR_ICEEI1	I²C bus interface, channel 1	Transfer error or event generation
PDL_INTC_VECTOR_ICRXI1		Data received
PDL_INTC_VECTOR_ICTXI1		Start of next data transfer
PDL_INTC_VECTOR_ICTEI1		End of data transfer

Return value

True.

Category

Interrupt control

Reference

Remarks

- The fast interrupt processing is allocated to only one interrupt handler.
- Open the file r_pdl_user_definitions.h and edit the definition FAST_INTC_VECTOR to give it the same value as the interrupt vector used in parameter data1.
  For example:
  #define FAST_INTC_VECTOR PDL_INTC_VECTOR_ADI0
  This will direct the compiler to generate the instructions required for a fast interrupt vector.
- This function uses an interrupt routine to modify the FINTV register. If the user has disabled interrupts (cleared the 'I' bit in the PSW register) in their own code, this function will lock up.

Program example

```
/* RPDL definitions */
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Assign the fast interrupt to the handler for pin IRQ3 */
    R_INTC_CreateFastInterrupt(
        PDL_INTC_VECTOR_IRQ3
    );
}

/* Do not forget to edit r_pdl_user_definitions.h (see remark 2) */
```

3) R_INTC_CreateExceptionHandlers

Synopsis

Assign handlers for the fixed-vector interrupts.

Prototype

```
bool R_INTC_CreateExceptionHandlers(
    void * func1,     // Callback function
    void * func2,     // Callback function
    void * func3      // Callback function
);
```

Description

Register the user functions to be called by the fixed-vector interrupts.

[func1]
The function to be called when a privileged instruction is detected while in user mode.
Specify PDL_NO_FUNC if no callback function is required.

[func2]
The function to be called when an undefined instruction is detected.
Specify PDL_NO_FUNC if no callback function is required.

[func3]
The function to be called when a floating point exception is detected.
Specify PDL_NO_FUNC if no callback function is required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Interrupt control

Reference

Remarks

- Please see the notes on callback function use in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.

Program example

```
/* RPDL definitions */
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Declaration of callback function */
void CallBackFunc( void );

void func( void )
{
    /* Add a function to manage floating point errors */
    R_INTC_CreateExceptionHandlers(
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        FloatingPointFunc
    );
}
```

4) R_INTC_ControlExtInterrupt

Synopsis

External interrupt control.

Prototype

```
bool R_INTC_ControlExtInterrupt(
    uint8_t data1,     // Pin selection
    uint16_t data2     // Control
);
```

Description

Enables or disables the specified external interrupt.

[data1]
Choose the interrupt pin to be configured.

PDL_INTC_IRQn (n = 0 to 15) or PDL_INTC_NMI	IRQi (n = 0 to 15) interrupt pin or NMI interrupt pin

[data2]
Select the controls. If multiple selections are required, use "|" to separate each selection.

- Enable or disable the interrupt pin (for the IRQ pins)

PDL_INTC_ENABLE or PDL_INTC_DISABLE	Enable or disable the IRQn interrupt pin.

- Detection sense selection (for the IRQ pins)

PDL_INTC_LOW or	Low level detection
PDL_INTC_FALLING or	Falling edge detection
PDL_INTC_RISING or	Rising edge detection
PDL_INTC_BOTH	Falling and rising edge detection

- Interrupt request clearing

PDL_INTC_CLEAR_IR_FLAG	Clear the Interrupt Request flag. This is not required if: • A callback function has been specified. • The interrupt priority level is higher than 0. • The processor interrupt priority level is lower than the interrupt priority level. This operation will not work for a level-sensitive interrupt if the input signal is still low.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

External interrupt

Reference

R_INTC_CreateExtInterrupt, R_INTC_GetExtInterruptStatus

Remarks

- The NMI pin was enabled during R_INTC_CreateExtInterrupt and cannot be disabled (an MCU design feature).
- When disabling an IRQn pin, the Interrupt Request flag will be cleared automatically.
- A callback function may be called once more if a valid event occurs just before the interrupt pin is disabled.

Program example

```
/* RPDL definitions */
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Disable the IRQ1 interrupt pin and clear the flag */
    R_INTC_ControlExtInterrupt(
        PDL_INTC_IRQ1,
        PDL_INTC_DISABLE | PDL_INTC_CLEAR_IR_FLAG
    );
}
```

5) R_INTC_GetExtInterruptStatus

Synopsis

Read the external interrupt status.

Prototype

```
bool R_INTC_GetExtInterruptStatus(
    uint8_t data1,     // Pin selection
    uint8_t * data2    // A pointer to the buffer where the status data shall be stored.
);
```

Description

Acquire the status for the specified external interrupt.

[data1]
Choose the interrupt pin to be checked.

PDL_INTC_IRQn (n = 0 to 15) or PDL_INTC_NMI	IRQn (n = 0 to 15) interrupt pin or NMI interrupt pin

[data2]
The status flags shall be stored in the following format:

For an IRQ pin:

b7 – b4	b3 – b2	b1	b0
	Detection condition	Pin level	Detection status
0	00b: Low level 01b: Falling edge 10b: Rising edge 11b: Both edges	0: Low 1: High	0: Not detected 1: Detected

For the NMI pin:

b7 – b2	b1	b0
	Detection condition	Detection status
0	0: Falling edge 1: Rising edge	0: Not detected 1: Detected

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

External interrupt

Reference

R_INTC_CreateExtInterrupt, R_INTC_ControlExtInterrupt

Remarks

- I/O port register PFCR8 or PFCR9 is checked to determine which pin is used for IRQn.
- If this function is called from within a callback function, the detection status will be 0.

Program example

```
/* RPDL definitions */
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t irq_status;

    /* Read the IR flag and pin state for IRQ5 */
    R_INTC_GetExtInterruptStatus(
         PDL_INTC_IRQ5,
         &irq_status
    );
}
```

6) R_INTC_Read

Synopsis

Read an interrupt register.

Prototype

```
bool R_INTC_Read(
     uint8_t data1,     // Register selection
     uint8_t data2,     // Register number
     uint8_t * data3    // Data storage location
);
```

Description

Read an interrupt register and store the value.

[data1]

- The register to be read

PDL_INTC_REG_IPL or	Select the current CPU interrupt priority level or
PDL_INTC_REG_IR or	Interrupt Request register or
PDL_INTC_REG_ISELR or	Interrupt Request Destination Setting register or
PDL_INTC_REG_IER or	Interrupt Request Enable register or
PDL_INTC_REG_IPR	Interrupt Priority register

[data2]
The register number.
IPL: Ignored.
IR: Between 16 and 253 (10h to FFh).
ISELR: Between 28 and 252 (1Ch to FFh).
IER: Between 2 and 15 (02h to 1Fh).
IPR: Between 0 and 143 (00h to 8Fh).

[data3]
The location where the register's value shall be stored.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Interrupt control

Reference

R_INTC_Write, R_INTC_Modify

Remarks

- The Interrupt register tables are not contiguous. This function does not check for gaps within these tables.

Program example

```
/* RPDL definitions */
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t ipl;

    /* Read the IPL bits */
    R_INTC_Read(
         PDL_INTC_REG_IPL,
         0,
         &ipl
    );
}
```

7) R_INTC_Write

Synopsis

Update an interrupt register.

Prototype

```
bool R_INTC_Write(
      uint8_t data1,    // Register selection
      uint8_t data2,    // Register number
      uint8_t data3     // Register value
);
```

Description

Write the new value to an interrupt register.

[data1]

- The register to be updated.

PDL_INTC_REG_IPL or PDL_INTC_REG_IR or PDL_INTC_REG_ISELR or PDL_INTC_REG_IER or PDL_INTC_REG_IPR	Select the current CPU interrupt priority level or Interrupt Request register or Interrupt Request Destination Setting register or Interrupt Request Enable register or Interrupt Priority register

[data2]
The register number.
IPL: Ignored.
IR: Between 16 and 253 (10h to FFh).
ISELR: Between 28 and 252 (1Ch to FFh).
IER: Between 2 and 15 (02h to 1Fh).
IPR: Between 0 and 143 (00h to 8Fh).

[data3]
The value to be written to the register.

Return value

True if the parameter is within range; otherwise false.

Category

Interrupt control

Reference

R_INTC_Read, R_INTC_Modify

Remarks

- This function uses an interrupt routine to modify the IPL bits. If the user has disabled interrupts (cleared the 'I' bit in the PSW register) in their own code, this function will lock up.
- The Interrupt register tables are not contiguous. This function does not check for gaps within these tables.

Program example

```
/* RPDL definitions */
#include "r_pdl_intc.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func( void )
{
    /* Set the IPL to 6 */
    R_INTC_Write(
          PDL_INTC_REG_IPL,
          PDL_NO_DATA,
          6
    );
}
```

8) R_INTC_Modify

Synopsis

Modify an interrupt register.

Prototype

```
bool R_INTC_Modify(
     uint8_t data1,   // Register selection
     uint8_t data2,   // Number
     uint8_t data3,   // Modification value
     uint8_t data4    // Logical operation
);
```

Description

Write the new value to the IPL bits in the Processor Status Word register.

[data1]

- The register to be updated.

PDL_INTC_REG_IR or PDL_INTC_REG_ISELR or PDL_INTC_REG_IER or PDL_INTC_REG_IPR	Select the Interrupt Request register or Interrupt Request Destination Setting register or Interrupt Request Enable register or Interrupt Priority register

[data2]

The register number.
IR: Between 16 and 253 (10h to FFh).
ISELR: Between 28 and 252 (1Ch to FFh).
IER: Between 2 and 15 (02h to 1Fh).
IPR: Between 0 and 143 (00h to 8Fh).

[data3]

The value to be used by the logical operation.

[data4]

- The logical operation to be applied to the register contents.

PDL_INTC_AND or PDL_INTC_OR or PDL_INTC_XOR	Select between AND (&), OR (|) or Exclusive-OR (^).

Return value

True if the parameter is within range; otherwise false.

Category

Interrupt control

Reference

R_INTC_Read, R_INTC_Write

Remarks

- This function uses an interrupt routine to modify the IPL bits. If the user has disabled interrupts (cleared the 'I' bit in the PSW register) in their own code, this function will lock up.
- The Interrupt register tables are not contiguous. This function does not check for gaps within these tables.

Program example

```
/* RPDL definitions */
#include "r_pdl_intc.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func( void )
{
    /* Set bits 6 and 4 in IER09 to 1 */
    R_INTC_Modify(
         PDL_INTC_REG_IER,
         0x09,
         0x50,
         PDL_INTC_OR
    );
}
```

4.2.3. I/O Port

I/O Port functions may operate on a complete port, or on individual port pins. The available definitions are listed below.

I/O port definitions

PDL_IO_PORT_0	Port P0	PDL_IO_PORT_1	Port P1
PDL_IO_PORT_2	Port P2	PDL_IO_PORT_3	Port P3
PDL_IO_PORT_4	Port P4	PDL_IO_PORT_5	Port P5
PDL_IO_PORT_6	Port P6	PDL_IO_PORT_7	Port P7
PDL_IO_PORT_8	Port P8	PDL_IO_PORT_9	Port P9
PDL_IO_PORT_A	Port PA	PDL_IO_PORT_B	Port PB
PDL_IO_PORT_C	Port PC	PDL_IO_PORT_D	Port PD
PDL_IO_PORT_E	Port PE	PDL_IO_PORT_F	Port PF
PDL_IO_PORT_G	Port PG	PDL_IO_PORT_H	Port PH

Note: Ports PF to PH are available only on the 176-pin device.

I/O port pin definitions

PDL_IO_PORT_0_0	Port pin $P0_0$	PDL_IO_PORT_0_1	Port pin $P0_1$
PDL_IO_PORT_0_2	Port pin $P0_2$	PDL_IO_PORT_0_3	Port pin $P0_3$
PDL_IO_PORT_0_4	Port pin $P0_4$	PDL_IO_PORT_0_5	Port pin $P0_5$
PDL_IO_PORT_1_0	Port pin $P1_0$	PDL_IO_PORT_1_1	Port pin $P1_1$
PDL_IO_PORT_1_2	Port pin $P1_2$	PDL_IO_PORT_1_3	Port pin $P1_3$
PDL_IO_PORT_1_4	Port pin $P1_4$	PDL_IO_PORT_1_5	Port pin $P1_5$
PDL_IO_PORT_1_6	Port pin $P1_6$	PDL_IO_PORT_1_7	Port pin $P1_7$
PDL_IO_PORT_2_0	Port pin $P2_0$	PDL_IO_PORT_2_1	Port pin $P2_1$
PDL_IO_PORT_2_2	Port pin $P2_2$	PDL_IO_PORT_2_3	Port pin $P2_3$
PDL_IO_PORT_2_4	Port pin $P2_4$	PDL_IO_PORT_2_5	Port pin $P2_5$
PDL_IO_PORT_2_6	Port pin $P2_6$	PDL_IO_PORT_2_7	Port pin $P2_7$
PDL_IO_PORT_3_0	Port pin $P3_0$	PDL_IO_PORT_3_1	Port pin $P3_1$
PDL_IO_PORT_3_2	Port pin $P3_2$	PDL_IO_PORT_3_3	Port pin $P3_3$
PDL_IO_PORT_3_4	Port pin $P3_4$	PDL_IO_PORT_3_5	Port pin $P3_5$
PDL_IO_PORT_3_6	Port pin $P3_6$	PDL_IO_PORT_3_7	Port pin $P3_7$
PDL_IO_PORT_4_0	Port pin $P4_0$	PDL_IO_PORT_4_1	Port pin $P4_1$
PDL_IO_PORT_4_2	Port pin $P4_2$	PDL_IO_PORT_4_3	Port pin $P4_3$
PDL_IO_PORT_4_4	Port pin $P4_4$	PDL_IO_PORT_4_5	Port pin $P4_5$
PDL_IO_PORT_4_6	Port pin $P4_6$	PDL_IO_PORT_4_7	Port pin $P4_7$
PDL_IO_PORT_5_0	Port pin $P5_0$	PDL_IO_PORT_5_1	Port pin $P5_1$
PDL_IO_PORT_5_2	Port pin $P5_2$	PDL_IO_PORT_5_3	Port pin $P5_3$
PDL_IO_PORT_5_4	Port pin $P5_4$	PDL_IO_PORT_5_5	Port pin $P5_5$
PDL_IO_PORT_5_6	Port pin $P5_6$	PDL_IO_PORT_5_7	Port pin $P5_7$
PDL_IO_PORT_6_0	Port pin $P6_0$	PDL_IO_PORT_6_1	Port pin $P6_1$
PDL_IO_PORT_6_2	Port pin $P6_2$	PDL_IO_PORT_6_3	Port pin $P6_3$
PDL_IO_PORT_6_4	Port pin $P6_4$	PDL_IO_PORT_6_5	Port pin $P6_5$
PDL_IO_PORT_6_6	Port pin $P6_6$	PDL_IO_PORT_6_7	Port pin $P6_7$
PDL_IO_PORT_7_0	Port pin $P7_0$	PDL_IO_PORT_7_1	Port pin $P7_1$
PDL_IO_PORT_7_2	Port pin $P7_2$	PDL_IO_PORT_7_3	Port pin $P7_3$
PDL_IO_PORT_7_4	Port pin $P7_4$	PDL_IO_PORT_7_5	Port pin $P7_5$
PDL_IO_PORT_7_6	Port pin $P7_6$	PDL_IO_PORT_7_7	Port pin $P7_7$
PDL_IO_PORT_8_0	Port pin $P8_0$	PDL_IO_PORT_8_1	Port pin $P8_1$

PDL_IO_PORT_8_2	Port pin $P8_2$	PDL_IO_PORT_8_3	Port pin $P8_3$
PDL_IO_PORT_8_4	Port pin $P8_4$	PDL_IO_PORT_8_5	Port pin $P8_5$
PDL_IO_PORT_8_6	Port pin $P8_6$		
PDL_IO_PORT_9_0	Port pin $P9_0$	PDL_IO_PORT_9_1	Port pin $P9_1$
PDL_IO_PORT_9_2	Port pin $P9_2$	PDL_IO_PORT_9_3	Port pin $P9_3$
PDL_IO_PORT_9_4	Port pin $P9_4$	PDL_IO_PORT_9_5	Port pin $P9_5$
PDL_IO_PORT_9_6	Port pin $P9_6$	PDL_IO_PORT_9_7	Port pin $P9_7$
PDL_IO_PORT_A_0	Port pin PA_0	PDL_IO_PORT_A_1	Port pin PA_1
PDL_IO_PORT_A_2	Port pin PA_2	PDL_IO_PORT_A_3	Port pin PA_3
PDL_IO_PORT_A_4	Port pin PA_4	PDL_IO_PORT_A_5	Port pin PA_5
PDL_IO_PORT_A_6	Port pin PA_6	PDL_IO_PORT_A_7	Port pin PA_7
PDL_IO_PORT_B_0	Port pin PB_0	PDL_IO_PORT_B_1	Port pin PB_1
PDL_IO_PORT_B_2	Port pin PB_2	PDL_IO_PORT_B_3	Port pin PB_3
PDL_IO_PORT_B_4	Port pin PB_4	PDL_IO_PORT_B_5	Port pin PB_5
PDL_IO_PORT_B_6	Port pin PB_6	PDL_IO_PORT_B_7	Port pin PB_7
PDL_IO_PORT_C_0	Port pin PC_0	PDL_IO_PORT_C_1	Port pin PC_1
PDL_IO_PORT_C_2	Port pin PC_2	PDL_IO_PORT_C_3	Port pin PC_3
PDL_IO_PORT_C_4	Port pin PC_4	PDL_IO_PORT_C_5	Port pin PC_5
PDL_IO_PORT_C_6	Port pin PC_6	PDL_IO_PORT_C_7	Port pin PC_7
PDL_IO_PORT_D_0	Port pin PD_0	PDL_IO_PORT_D_1	Port pin PD_1
PDL_IO_PORT_D_2	Port pin PD_2	PDL_IO_PORT_D_3	Port pin PD_3
PDL_IO_PORT_D_4	Port pin PD_4	PDL_IO_PORT_D_5	Port pin PD_5
PDL_IO_PORT_D_6	Port pin PD_6	PDL_IO_PORT_D_7	Port pin PD_7
PDL_IO_PORT_E_0	Port pin PE_0	PDL_IO_PORT_E_1	Port pin PE_1
PDL_IO_PORT_E_2	Port pin PE_2	PDL_IO_PORT_E_3	Port pin PE_3
PDL_IO_PORT_E_4	Port pin PE_4	PDL_IO_PORT_E_5	Port pin PE_5
PDL_IO_PORT_E_6	Port pin PE_6	PDL_IO_PORT_E_7	Port pin PE_7
PDL_IO_PORT_F_0	Port pin PF_0	PDL_IO_PORT_F_1	Port pin PF_1
PDL_IO_PORT_F_2	Port pin PF_2	PDL_IO_PORT_F_3	Port pin PF_3
PDL_IO_PORT_F_4	Port pin PF_4	PDL_IO_PORT_F_5	Port pin PF_5
PDL_IO_PORT_F_6	Port pin PF_6		
PDL_IO_PORT_G_0	Port pin PG_0	PDL_IO_PORT_G_1	Port pin PG_1
PDL_IO_PORT_G_2	Port pin PG_2	PDL_IO_PORT_G_3	Port pin PG_3
PDL_IO_PORT_G_4	Port pin PG_4	PDL_IO_PORT_G_5	Port pin PG_5
PDL_IO_PORT_G_6	Port pin PG_6	PDL_IO_PORT_G_7	Port pin PG_7
PDL_IO_PORT_H_0	Port pin PH_0	PDL_IO_PORT_H_1	Port pin PH_1
PDL_IO_PORT_H_2	Port pin PH_2	PDL_IO_PORT_H_3	Port pin PH_3
PDL_IO_PORT_H_4	Port pin PH_4	PDL_IO_PORT_H_5	Port pin PH_5
PDL_IO_PORT_H_6	Port pin PH_6	PDL_IO_PORT_H_7	Port pin PH_7

Note: Port pins PF_X to PH_X are available only on the 176-pin device.

1) R_IO_PORT_Set

Synopsis

Configure an I/O port.

Prototype

```
bool R_IO_PORT_Set(
    uint16_t data1,    // Port pin selection
    uint8_t data2      // Configuration
);
```

Description

Set the operating conditions for I/O port pins.

[data1]
Select the port pins to be configured (from §4.2.3). Do not use any whole-port definitions.
Multiple pins on the same port may be specified, using "|" to separate each pin.

[data2]
Choose the pin settings. Use "|" to separate each selection.
If no selection is made, the control setting will be left unchanged.

- Direction control

PDL_IO_PORT_INPUT or PDL_IO_PORT_OUTPUT	Input or output.

- Input buffer control

PDL_IO_PORT_INPUT_BUFFER_ON or PDL_IO_PORT_INPUT_BUFFER_OFF	On or off.

- Input pull-up resistor control

PDL_IO_PORT_PULL_UP_ON or PDL_IO_PORT_PULL_UP_OFF	On or off. Valid for ports A to E.

- Output type control

PDL_IO_PORT_OPEN_DRAIN or PDL_IO_PORT_CMOS	NMOS open-drain or CMOS push-pull output. Valid for ports 2 and C.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

I/O port

References

R_IO_PORT_ModifyControl, R_IO_PORT_ReadControl

Remarks

- Ensure that the specified functions are valid for the selected port pin.
- The data direction and input buffer registers may be modified by other driver Create functions. Take care to not overwrite existing settings.

Program example

```
/* RPDL definitions */
#include "r_pdl_io_port.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Set up port P93 as an input port with the pull-resistor on */
    R_IO_PORT_Set(
        PDL_IO_PORT_9_3,
        PDL_IO_PORT_INPUT | PDL_IO_PORT_INPUT_BUFFER_ON | \
         PDL_IO_PORT_PULL_UP_ON
    );
}
```

2) R_IO_PORT_ReadControl

Synopsis

Read an I/O port's control register.

Prototype

```
bool R_IO_PORT_ReadControl(
    uint16_t data1,    // Port or port pin selection
    uint8_t data2,     // Control register selection
    uint8_t * data3    // Data storage location
);
```

Description

Read an I/O port pin control setting.

[data1]
Use either one of the following definition values (from §4.2.3).
- One port definition or
- One port pin definition.

[data2]
- Select the register to be read.

PDL_IO_PORT_DIRECTION or	Data direction register.
PDL_IO_PORT_INPUT_BUFFER or	Input buffer control register.
PDL_IO_PORT_PULL_UP or	Pull-up control register. Valid for ports A to E.
PDL_IO_PORT_TYPE	Open-drain control register. Valid for ports 2 and C.

[data3]
The address to where the register value shall be stored.
The value will be between 0x00 and 0xFF for a port; 0 or 1 for a pin.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

I/O port

References

R_IO_PORT_Set, R_IO_PORT_ModifyControl

Remarks

- Ensure that the specified register is valid for the selected port pin.

Program example

```
/* RPDL definitions */
#include "r_pdl_io_port.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    uint8_t direction;
    uint8_t output;

    /* Read the direction register for port PC */
    R_IO_PORT_ReadControl(
        PDL_IO_PORT_C,
        PDL_IO_PORT_DIRECTION
        &direction
    );

    /* Read the output type for pin P03 */
    R_IO_PORT_ReadControl(
        PDL_IO_PORT_0_3,
        PDL_IO_PORT_TYPE
        &output
    );
}
```

3) R_IO_PORT_ModifyControl

Synopsis

Modify an I/O port's control registers.

Prototype

```
bool R_IO_PORT_ModifyControl(
    uint16_t data1,    // Port or port pin selection
    uint8_t data2,     // Control register and logical operation selection
    uint8_t data3      // Modification value
);
```

Description

Modifying the operation of an I/O port or I/O port pin.

[data1]
Use either one of the following definition values (from §4.2.3).
- One port definition or
- One port pin definition.

[data2]
Select the register to be modified and the logical operation, using "|" to separate the selections.

- The control register to be modified.

PDL_IO_PORT_DIRECTION or	Data direction register.
PDL_IO_PORT_INPUT_BUFFER or	Input buffer control register.
PDL_IO_PORT_PULL_UP or	Pull-up MOS control register. Valid for ports A to E.
PDL_IO_PORT_TYPE	Open-drain control register. Valid for ports 2 or C.

- The logical operation to be applied to the control register.

PDL_IO_PORT_AND or PDL_IO_PORT_OR or PDL_IO_PORT_XOR	Select between AND (&), OR (|) or Exclusive-OR (^).

[data3]
The value to be used for the modification; Between 0x00 and 0xFF for a port, 0 or 1 for a pin.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

I/O port

References

R_IO_PORT_Set, R_IO_PORT_ReadControl

Remarks

- Ensure that the specified functions are valid for the selected port pin.
- The data direction and input buffer registers may be modified by other driver Create functions. Take care to not overwrite existing settings.

Program example

```
/* RPDL definitions */
#include “r_pdl_io_port.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Set the lower 4 bits on port P1 to output */
    R_IO_PORT_ModifyControl(
        PDL_IO_PORT_1,
        PDL_IO_PORT_DIRECTION | PDL_IO_PORT_OR,
        0x0F
    );

    /* Enable the pull-up on pin PA3 */
    R_IO_PORT_ModifyControl(
        PDL_IO_PORT_A_3,
        PDL_IO_PORT_PULL_UP | PDL_IO_PORT_OR,
        1
    );
}
```

4) R_IO_PORT_Read

Synopsis

Read data from an I/O port.

Prototype

```
bool R_IO_PORT_Read(
    uint16_t data1,    // Port or port pin selection
    uint8_t * data2    // Pointer to the variable in which the value shall be stored.
);
```

Description

Gets the value of an I/O port or I/O port pin.

[data1]
Use either one of the following definition values (from §4.2.3).
- One port definition or
- One port pin definition.

[data2]
The value will be between 0x00 and 0xFF for a port, 0 or 1 for a pin.

Return value

If the I/O port specification is incorrect, false is returned; otherwise, true is returned.

Functionality

I/O port

Reference

R_IO_PORT_Set

Remark

- If an invalid port or pin is specified, the operation of the function cannot be guaranteed.

Program example

```
/* RPDL definitions */
#include “r_pdl_io_port.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t data;

    /* Get the value of port pin P12 */
    R_IO_PORT_Read(
        PDL_IO_PORT_1_2,
        &data
    );

    /* Get the value of port 4 */
    R_IO_PORT_Read(
        PDL_IO_PORT_4,
        &data
    );
}
```

5) R_IO_PORT_Write

Synopsis

Write data to an I/O port.

Prototype

```
bool R_IO_PORT_Write(
    uint16_t data1,    // Port or port pin selection
    uint8_t data2      // The data to be written to the I/O port or port pin.
);
```

Description

Write data to an I/O port or I/O port pin.

[data1]
Use either one of the following definition values (from §4.2.3).
- One port definition or
- One port pin definition.

[data2]
The value must be between 0x00 and 0xFF for a port, 0 or 1 for a pin.

Return value

True if the parameters are valid; otherwise false.

Functionality

I/O port

References

R_IO_PORT_Set, R_IO_PORT_Read

Remarks

- If an invalid port or pin is specified, the operation of the function cannot be guaranteed.

Program example

```
/* RPDL definitions */
#include “r_pdl_io_port.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Set the output of port pin P05 */
    R_IO_PORT_Write(
        PDL_IO_PORT_0_5,
        0
    );

    /* Set the output of port 6 */
    R_IO_PORT_Write(
        PDL_IO_PORT_6,
        0x55
    );
}
```

6) R_IO_PORT_Compare

Synopsis

Check the pin states on an I/O port.

Prototype

```
bool R_IO_PORT_Compare(
    uint16_t data1,    // Input port or port pin selection
    uint8_t data2,     // Comparison value
    void * func        // Function pointer
);
```

Description

Read the input state of an I/O port or I/O port pin and call a function if a match occurs.

[data1]
Use either one of the following definition values (from §4.2.3):
- One port definition or
- One port pin definition.

[data2]
The value to be compared with; Between 0x00 and 0xFF for a port, 0 or 1 for a pin.

[func]
The function to be called if a match occurs.

Return value

True if the parameters are valid; otherwise false.

Functionality

I/O port

References

R_IO_PORT_Set

Remarks

- If an invalid port or pin is specified, the operation of the function cannot be guaranteed.

Program example

```
/* RPDL definitions */
#include “r_pdl_io_port.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void IoHandler1{}
void IoHandler2{}

void func( void )
{
    /* Call function IoHandler1 if port pin P05 is high */
    R_IO_PORT_Compare(
        PDL_IO_PORT_0_5,
        1,
        IoHandler1
    );

    /* Call function IoHandler2 if port 6 reads as 0x55 */
    R_IO_PORT_Compare(
        PDL_IO_PORT_6,
        0x55,
        IoHandler2
    );
}
```

7) R_IO_PORT_Modify

Synopsis

Modify the pin states on an I/O port.

Prototype

```
bool R_IO_PORT_Modify(
    uint16_t data1,    // Output port or port pin selection
    uint8_t data2,     // Logical operation
    uint8_t data3      // Modification value
);
```

Description

Read the output state of an I/O port or I/O port pin, modify the result and write it back to the port.

[data1]
Use either one of the following definition values (from §4.2.3).
- One port definition or
- One port pin definition.

[data2]
- The logical operation to be applied to the port or port pin.

PDL_IO_PORT_AND or PDL_IO_PORT_OR or PDL_IO_PORT_XOR	Select between AND (&), OR (|) or Exclusive-OR (^).

[data3]
The value to be used for the modification; Between 0x00 and 0xFF for a port, 0 or 1 for a pin.

Return value

True if the parameters are valid; otherwise false.

Functionality

I/O port

References

R_IO_PORT_Set, R_IO_PORT_Read, R_IO_PORT_Write

Remarks

- If an invalid port or pin is specified, the operation of the function cannot be guaranteed.

Program example

```
/* RPDL definitions */
#include “r_pdl_io_port.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Invert port pin P05 */
    R_IO_PORT_Modify(
        PDL_IO_PORT_0_5,
        PDL_IO_PORT_XOR,
        1
    );

    /* And the value port 6 with 0x55 */
    R_IO_PORT_Modify(
        PDL_IO_PORT_6,
        PDL_IO_PORT_AND,
        0x55
    );
}
```

8) R_IO_PORT_Wait

Synopsis

Wait for a match on an I/O port.

Prototype

```
bool R_IO_PORT_Wait(
    uint16_t data1,    // Output port or port pin selection
    uint8_t data2      // Comparison value
);
```

Description

Loop until an I/O port or I/O port pin matches the comparison value.

[data1]
Use either one of the following definition values (from §4.2.3).
- One port definition or
- One port pin definition.

[data2]
The value to be compared with; Between 0x00 and 0xFF for a port, 0 or 1 for a pin.

Return value

True if the parameters are valid; otherwise false.

Functionality

I/O port

References

R_IO_PORT_Set, R_IO_PORT_Read

Remarks

- If an invalid port or pin is specified, the operation of the function cannot be guaranteed.
- This function waits for the I/O port or port pin value to match the comparison data. If the I/O port's control registers are directly modified by the user, this function may lock up.

Program example

```
/* RPDL definitions */
#include "r_pdl_io_port.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func( void )
{
    /* Wait until pin P05 reads as 0 */
    R_IO_PORT_Wait(
        PDL_IO_PORT_0_5,
        0
    );

    /* Wait until port 6 reads as 0x55 */
    R_IO_PORT_Wait(
        PDL_IO_PORT_6,
        0x55
    );
}
```

4.2.4. Port Function Control

1) R_PFC_Read

Synopsis	Read a PFC register.
Prototype	**bool R_PFC_Read(** **uint8_t data1,** // PFC register selection **uint8_t * data2** // Pointer to the variable where the PFC register's value shall be stored. **);**
Description	Get the value of a PFC register. **[data1]** Any value from 0 to 9. **[data2]** The value read from the register.
Return value	True if a valid register is specified; otherwise false.
Functionality	PFC registers
References	R_PFC_Write
Remarks	• None.
Program example	

```
/* RPDL definitions */
#include “r_pdl_pfc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t data;

    /* Get the value of register PFC1 */
    R_PFC_Read(
        1,
        &data
    );
}
```

2) R_PFC_Write

Synopsis

Write to a PFC register.

Prototype

```
bool R_PFC_Write(
    uint8_t data1,    // PFC register selection
    uint8_t data2     // Data to be written to the PFC register
);
```

Description

Write the value to a PFC register.

[data1]
Any value from 0 to 9.

[data2]
The value to be written to the register.

Return value

True if a valid register is specified; otherwise false.

Functionality

PFC registers

References

R_PFC_Read, R_PFC_Modify

Remarks

- The PFC registers are modified by other driver functions. Take care to not overwrite existing settings.

Program example

```
/* RPDL definitions */
#include “r_pdl_pfc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Write data to register PFC1 */
    R_PFC_Write(
        1,
        0xFF
    );
}
```

3) R_PFC_Modify

Synopsis

Modify a PFC register.

Prototype

```
bool R_PFC_Modify(
    uint8_t data1,    // PFC register selection
    uint8_t data2,    // Logical operation
    uint8_t data3     // Modification value
);
```

Description

Write the value to a PFC register.

[data1]

Any value from 0 to 9.

[data2]

- The logical operation to be applied to the register contents.

PDL_PFC_AND or PDL_PFC_OR or PDL_PFC_XOR	Select between AND (&), OR (|) or Exclusive-OR (^).

[data3]

The value to be used for the modification.

Return value

True if a valid register is specified; otherwise false.

Functionality

PFC registers

References

R_PFC_Read

Remarks

- The PFC registers are modified by other driver functions. Take care to not overwrite existing settings.

Program example

```
/* RPDL definitions */
#include “r_pdl_pfc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Set bit 7 in PFC1 to 1 */
    R_PFC_Modify(
        1,
        PDL_PFC_OR,
        0x80
    );
}
```

4.2.5. MCU operation

1) R_MCU_Control

Synopsis

Control the operation of the MCU.

Prototype

```
bool R_MCU_Control(
    uint8_t data    // Control options
);
```

Description

Modify the MCU control registers.

[data]
Select the operation states. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- On-chip ROM control

PDL_MCU_ROM_ENABLE or PDL_MCU_ROM_DISABLE	Enable or disable the on-chip ROM.

- On-chip RAM control

PDL_MCU_RAM_ENABLE or PDL_MCU_RAM_DISABLE	Enable or disable the on-chip RAM.

Return value

True if a valid register is specified; otherwise false.

Functionality

MCU registers

References

R_MCU_GetStatus

Remarks

- None.

Program example

```
/* RPDL definitions */
#include "r_pdl_mcu.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func( void )
{
    /* Modify the MCU operation */
    R_MCU_Control(
        PDL_MCU_ROM_DISABLE
    );
}
```

2) R_MCU_GetStatus

Synopsis

Read the MCU status.

Prototype

```
bool R_MCU_GetStatus(
    uint16_t * data    // Pointer to the variable where the status value shall be stored.
);
```

Description

Read the status registers for the MCU.

[data]
The status flags shall be stored in the format below.

b15	b14	b13	b12	b11 – b10	b9	b8
Start-up states						
0	User boot mode 0: Other mode 1: Boot mode	0	Boot mode 0: Other mode 1: Boot mode	External bus 00: 16-bit 10: 8-bit	 0: Disabled 1: Enabled	On-chip ROM 0: Disabled 1: Enabled

b7	b6 – b2	b1	b0
Endian	0	Pin states	
0: Little 1: Big		MD1	MD0

Return value

True.

Functionality

MCU registers

References

R_MCU_Control

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_pdl_mcu.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint16_t status;

    /* Read the MCU status registers */
    R_MCU_GetStatus(
        &status
    );
}
```

4.2.6. Low Power Consumption

1) R_LPC_Create

Synopsis

Configure the MCU low power conditions.

Prototype

```
bool R_LPC_Create(
    uint32_t data1,    // Configuration options
    uint32_t data2     // Waiting times
);
```

Description (1/2)

Load the registers that control module or CPU operation.

[data1]
Select the required settings. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Software and Deep Software Standby mode output port control

PDL_LPC_EXT_BUS_ON or PDL_LPC_EXT_BUS_HI_Z	Leave the external bus address and control signals active, or set them to the high-impedance state.

- On-chip RAM power control

PDL_LPC_RAM_ON or **PDL_LPC_RAM_OFF**	Enable or disable power to the RAM (from 00000000h to 0000FFFFh) in deep software standby mode.

- I/O port retention control

PDL_LPC_IO_SAME or PDL_LPC_IO_DELAY	Select whether IO port retention is cancelled when deep software standby mode is ended, or when CPU operation has resumed.

- Deep software standby cancel control

PDL_LPC_CANCEL_IRQ0_DISABLE or PDL_LPC_CANCEL_IRQ0_FALLING or PDL_LPC_CANCEL_IRQ0_RISING	Prevent or allow an edge on the IRQ0-A pin to cancel deep software standby mode.
PDL_LPC_CANCEL_IRQ1_DISABLE or PDL_LPC_CANCEL_IRQ1_FALLING or PDL_LPC_CANCEL_IRQ1_RISING	Prevent or allow an edge on the IRQ1-A pin to cancel deep software standby mode.
PDL_LPC_CANCEL_IRQ2_DISABLE or PDL_LPC_CANCEL_IRQ2_FALLING or PDL_LPC_CANCEL_IRQ2_RISING	Prevent or allow an edge on the IRQ2-A pin to cancel deep software standby mode.
PDL_LPC_CANCEL_IRQ3_DISABLE or PDL_LPC_CANCEL_IRQ3_FALLING or PDL_LPC_CANCEL_IRQ3_RISING	Prevent or allow an edge on the IRQ3-A pin to cancel deep software standby mode.
PDL_LPC_CANCEL_NMI_DISABLE or PDL_LPC_CANCEL_NMI_FALLING or PDL_LPC_CANCEL_NMI_RISING	Prevent or allow an edge on the NMI pin to cancel deep software standby mode.

Description (2/2)

[data2]
Select the waiting times. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Software Standby waiting time

PDL_LPC_STANDBY_64 or PDL_LPC_STANDBY_512 or PDL_LPC_STANDBY_1024 or PDL_LPC_STANDBY_2048 or PDL_LPC_STANDBY_4096 or PDL_LPC_STANDBY_16384 or PDL_LPC_STANDBY_32768 or PDL_LPC_STANDBY_65536 or PDL_LPC_STANDBY_131072 or PDL_LPC_STANDBY_262144 or **PDL_LPC_STANDBY_524288**	Select the number of PCLK cycles that will elapse before the CPU resumes after exiting from software standby mode.

- Deep Software Standby waiting time

PDL_LPC_DEEP_STANDBY_64 or PDL_LPC_DEEP_STANDBY_512 or PDL_LPC_DEEP_STANDBY_1024 or PDL_LPC_DEEP_STANDBY_2048 or PDL_LPC_DEEP_STANDBY_4096 or PDL_LPC_DEEP_STANDBY_16384 or PDL_LPC_DEEP_STANDBY_32768 or PDL_LPC_DEEP_STANDBY_65536 or PDL_LPC_DEEP_STANDBY_131072 or PDL_LPC_DEEP_STANDBY_262144 or **PDL_LPC_DEEP_STANDBY_524288**	Select the number of PCLK cycles that will elapse before the CPU resumes after exiting from deep software standby mode.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

Low Power Consumption control registers

References

R_LPC_Control

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_pdl_lpc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Allow a falling edge on IRQ0-A to cancel deep software standby */
    R_LPC_Create(
        PDL_LPC_CANCEL_IRQ0_FALLING,
        PDL_LPC_STANDBY_64 | PDL_LPC_DEEP_STANDBY_1024
    );
}
```

2) R_LPC_Control

Synopsis

Select a low power consumption mode.

Prototype

```
bool R_LPC_Control(
    uint16_t data      // Mode selection
);
```

Description

Transition to one of the low power modes.

[data]
Mode selection.

- Mode selection

PDL_LPC_MODE_SLEEP or PDL_LPC_MODE_ALL_MODULE_CLOCK_STOP or PDL_LPC_MODE_SOFTWARE_STANDBY or PDL_LPC_MODE_DEEP_SOFTWARE_STANDBY	Select the mode to be entered.

- All-module clock stop cancellation modification

PDL_LPC_TMR_OFF or PDL_LPC_TMR_UNIT_0 or PDL_LPC_TMR_UNIT_1 or PDL_LPC_TMR_BOTH	Select whether the TMR units can be used to exit from All-module clock stop mode.

- I/O port retention cancellation

PDL_LPC_IO_RELEASE	Cancel the retention of I/O port pin states.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

Low Power Consumption control registers

References

R_LPC_Create

Remarks

- Sleep mode is utilised by some peripheral drivers to turn off the CPU when required.
- The peripheral Create functions bring modules out of the clock-stop state as required.
  The peripheral Destroy functions put modules into the clock-stop state as required.
  When All Module Clock-Stop mode is cancelled, the peripherals that were active when that mode was entered will be re-activated.

Program example

```
/* RPDL definitions */
#include “r_pdl_lpc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Enter deep software standby mode */
    R_LPC_Control(
        PDL_LPC_MODE_DEEP_SOFTWARE_STANDBY
    );
}

void func( void )
{
    /* Clear I/O port retention */
    R_LPC_Control(
        PDL_LPC_IO_RELEASE
    );
}
```

3) R_LPC_WriteBackup

Synopsis

Write to the Backup registers.

Prototype

```
bool R_LPC_WriteBackup(
    uint8_t * data1,    // Data pointer
    uint8_t data2       // Data count
);
```

Description

Write data into the backup registers.

[data1]
The data to be written to the backup area.

[data2]
The number of bytes to be written to the backup area. Valid from 1 to 32.

Return value

True if all parameters are valid; otherwise false.

Functionality

Low Power Consumption control registers

References

R_LPC_ReadBackup

Remarks

- The definition R_PDL_LPC_BACKUP_AREA_SIZE specifies the number of bytes that are available

Program example

```
/* RPDL definitions */
#include “r_pdl_lpc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t data_to_save[R_PDL_LPC_BACKUP_AREA_SIZE];

    /* Write data into the backup registers */
    R_LPC_WriteBackup(
        data_to_save,
        R_PDL_LPC_BACKUP_AREA_SIZE
    );
}
```

4) R_LPC_ReadBackup

Synopsis

Read from the Backup registers.

Prototype

```
bool R_LPC_ReadBackup(
    uint8_t * data1,    // Data pointer
    uint8_t data2       // Data count
);
```

Description

Read data from the backup registers.

[data1]
The storage area for the data read from the backup area.

[data2]
The number of bytes to be read from the backup area. Valid from 1 to 32.

Return value

True if all parameters are valid; otherwise false.

Functionality

Low Power Consumption control registers

References

R_LPC_WriteBackup

Remarks

- The definition R_PDL_LPC_BACKUP_AREA_SIZE specifies the number of bytes that are available

Program example

```
/* RPDL definitions */
#include “r_pdl_lpc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t data_to_restore[R_PDL_LPC_BACKUP_AREA_SIZE];

    /* Read data from the backup registers */
    R_LPC_ReadBackup(
        data_to_restore,
        R_PDL_LPC_BACKUP_AREA_SIZE
    );
}
```

5) R_LPC_GetStatus

Synopsis

Read the status flags.

Prototype

```
bool R_LPC_GetStatus(
    uint16_t * data    // Data pointer
);
```

Description

Read the registers that control module or CPU operation.

[data]
The status flags shall be stored in the format below.

b15	b14 – b8
0: No activity 1: An external interrupt has caused an exit from deep software standby mode, followed by an internal reset	0

b7	b6 – b4	b3	b2	b1	b0
Deep Software Standby cancel request detection					
0: No activity 1: The exit from deep software standby was caused by a valid edge on pins IRQn-A or NMI					
NMI	0	IRQ3-A	IRQ2-A	IRQ1-A	IRQ0-A

Return value

True.

Functionality

Low Power Consumption control registers

References

R_LPC_Create, R_LPC_Control

Remarks

- If a flag is set to 1, it shall be automatically cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_lpc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint16_t status_flags;

    /* Find out what caused the exit from deep software standby */
    R_LPC_GetStatus(
        &status_flags
    );
}
```

4.2.7. Bus Controller

1) R_BSC_Create

Synopsis

Configure the external bus controller.

Prototype

```
bool R_BSC_Create(
        uint32_t data1,     // Configuration
        uint32_t data2,     // Configuration
        uint8_t data3,      // Configuration
        void * func,        // Callback function
        uint8_t data4       // Interrupt priority level
);
```

Description (1/2)

Configure the I/O pins, error detection and register the callback function

Control the external bus controller.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

[data1]

- Chip select pin selection (only required for each external memory area that is enabled).

PDL_BSC_CS2_A or PDL_BSC_CS2_B	Select pin CS2#-A or CS2#-B.
PDL_BSC_CS3_A or PDL_BSC_CS3_B	Select pin CS3#-A or CS3#-B.
PDL_BSC_CS4_A or PDL_BSC_CS4_B or PDL_BSC_CS4_C or PDL_BSC_CS4_D	Select pin CS4#-A, CS4#-B, CS4#-C or CS4#-D.
PDL_BSC_CS5_A or PDL_BSC_CS5_B or PDL_BSC_CS5_C or PDL_BSC_CS5_D	Select pin CS5#-A, CS5#-B, CS5#-C or CS5#-D.
PDL_BSC_CS6_A or PDL_BSC_CS6_B or PDL_BSC_CS6_C or PDL_BSC_CS6_D	Select pin CS6#-A, CS6#-B, CS6#-C or CS6#-D.
PDL_BSC_CS7_A or PDL_BSC_CS7_B or PDL_BSC_CS7_C or PDL_BSC_CS7_D	Select pin CS7#-A, CS7#-B, CS7#-C or CS7#-D.

- Strobe pin control

PDL_BSC_WR1BC1_ENABLE or **PDL_BSC_WR1BC1_DISABLE**	Enable or disable the WR1# / BC1# output.

[data2]

- Address output control. The signals are **enabled** by default. Specify 0 for no change.

PDL_BSC_A0BC0_DISABLE	Disable the output of the A0 / BC0# signal. When disabled, the pin can be used as an input only.
PDL_BSC_A1_DISABLE	Disable the output of the A1 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A2_DISABLE	Disable the output of the A2 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A3_DISABLE	Disable the output of the A3 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A4_DISABLE	Disable the output of the A4 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A5_DISABLE	Disable the output of the A5 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A6_DISABLE	Disable the output of the A6 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A7_DISABLE	Disable the output of the A7 signal. When disabled, the pin can be used as an input only.
PDL_BSC_A8_DISABLE	Disable the output of the A8 signal.

Description (2/2)

PDL_BSC_A9_DISABLE	Disable the output of the A9 signal.
PDL_BSC_A10_DISABLE	Disable the output of the A10 signal.
PDL_BSC_A11_DISABLE	Disable the output of the A11 signal.
PDL_BSC_A12_DISABLE	Disable the output of the A12 signal.
PDL_BSC_A13_DISABLE	Disable the output of the A13 signal.
PDL_BSC_A14_DISABLE	Disable the output of the A14 signal.
PDL_BSC_A15_DISABLE	Disable the output of the A15 signal.
PDL_BSC_A16_DISABLE	Disable the output of the A16 signal.
PDL_BSC_A17_DISABLE	Disable the output of the A17 signal.
PDL_BSC_A18_DISABLE	Disable the output of the A18 signal.
PDL_BSC_A19_DISABLE	Disable the output of the A19 signal.
PDL_BSC_A20_DISABLE	Disable the output of the A20 signal.
PDL_BSC_A21_DISABLE	Disable the output of the A21 signal.
PDL_BSC_A22_DISABLE	Disable the output of the A22 signal.
PDL_BSC_A23_DISABLE	Disable the output of the A23 signal.

[data3]

- Error monitoring

PDL_BSC_ERROR_ILLEGAL_ADDRESS_ENABLE or **PDL_BSC_ERROR_ILLEGAL_ADDRESS_DISABLE**	Enable or disable illegal address access detection.
PDL_BSC_ERROR_TIME_OUT_ENABLE or **PDL_BSC_ERROR_TIME_OUT_DISABLE**	Enable or disable time-out detection.

[func]

The function to be called when a bus error occurs. Specify PDL_NO_FUNC if not required.

[data4]

The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority.
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Bus Controller

Reference

R_BSC_CreateArea

Remarks

- Multiple chip select signals can be output from one I/O pin.

Pin	CS0	CS1	CS2	CS3	CS4	CS5	CS6	CS7
P60	CS0#				CS4#-A	CS5#-B		
P61		CS1#	CS2#-B			CS5#-A	CS6#-B	CS7#-B
P62			CS2#-A				CS6#-A	
P63				CS3#-A				CS7#-A
P64					CS4#-B			
P70				CS3#-B				
P71					CS4#-C	CS5#-C	CS6#-C	CS7#-C
PC5						CS5#-D		
PC6							CS6#-D	
PC7					CS4#-D			CS7#-D

- Port Function Control registers PFCR1 to PFCR5 and the appropriate I/O port DDR and ICR registers are modified by this function.
- The external bus is enabled by this function.
- Call this function before using function R_BSC_CreateArea.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.

Program example

```
/* RPDL definitions */
#include “r_pdl_bsc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Bus error handler */
void BusErrorFunc(void){}

void func(void)
{
    /* Select CS2-B, all address signals, enable interrupts and register the
callback function */
    R_BSC_Create(
          PDL_BSC_CS2_B,
          0,
          PDL_BSC_ERROR_ILLEGAL_ADDRESS_ENABLE | \
           PDL_BSC_ERROR_TIME_OUT_ENABLE,
          BusErrorFunc,
          5
          );
}
```

2) R_BSC_CreateArea

Synopsis

Configure an external bus area.

Prototype

```
bool R_BSC_CreateArea(
        uint8_t data1,      // Area selection
        uint16_t data2,     // Configuration selection
        uint8_t data3,      // RRCV cycles
        uint8_t data4,      // WRCV cycles
        uint8_t data5,      // CSPRWAIT cycles
        uint8_t data6,      // CSPWWAIT cycles
        uint8_t data7,      // CSRWAIT cycles
        uint8_t data8,      // CSWWAIT cycles
        uint8_t data9,      // CSROFF cycles
        uint8_t data10,     // CSWOFF cycles
        uint8_t data11,     // WDOFF cycles
        uint8_t data12,     // RDON cycles
        uint8_t data13,     // WRON cycles
        uint8_t data14,     // WDON cycles
        uint8_t data15      // CSON cycles
);
```

Description (1/2)

Set up an external bus area.

[data1]
The address area n (where n = 0 to 7).

[data2]
Configure the operation of area CSn.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- External bus width

PDL_BSC_WIDTH_16 or PDL_BSC_WIDTH_8	Select 16- or 8-bit data bus width

- Endian mode

PDL_BSC_ENDIAN_SAME or PDL_BSC_ENDIAN_OPPOSITE	Set the bus endian mode to be the same or opposite to that of the CPU.

- Write access mode

PDL_BSC_WRITE_BYTE or PDL_BSC_WRITE_SINGLE	Select byte strobe or single write strobe mode.

- External wait control

PDL_BSC_WAIT_DISABLE or PDL_BSC_WAIT_ENABLE	Disable or enable external wait control (using the WAIT# pin).

- Page access control

PDL_BSC_PAGE_READ_DISABLE or PDL_BSC_PAGE_READ_NORMAL or PDL_BSC_PAGE_READ_CONTINUOUS	Disable or enable page read accesses using normal access compatible mode or continuous assertion mode.
PDL_BSC_PAGE_WRITE_DISABLE or PDL_BSC_PAGE_WRITE_ENABLE	Disable or enable page write accesses.

[data3]
The number of read recovery cycles (RRCV). Valid between 0 and 15.

[data4]
The number of write recovery cycles (WRCV). Valid between 0 and 15.

[data5]
The number of wait cycles used for second and subsequent accesses during a page read sequence (CSPRWAIT). Valid between 0 and 7.

Description (2/2)

[data6]
The number of wait cycles used for second and subsequent accesses during a page write sequence (CSPWWAIT). Valid between 0 and 7.

[data7]
The number of wait cycles for the first access during a normal or page read sequence (CSRWAIT). Valid between 0 and 31.

[data8]
The number of wait cycles for the first access during a normal or page write sequence (CSWWAIT). Valid between 0 and 31.

[data9]
The number of cycles that the CS signal is left asserted after the read strobe is negated (CSROFF). Valid between 0 and 7.

[data10]
The number of cycles that the CS signal is left asserted after the write strobe is negated (CSWOFF). Valid between 0 and 7.

[data11]
The number of cycles that the data output is left asserted after the write strobe is negated (WDOFF). Valid between 1 and 7.

[data12]
The number of cycles before the read strobe is asserted (RDON). Valid between 0 and 7.

[data13]
The number of cycles before the write strobe is asserted (WRON). Valid between 0 and 7.

[data14]
The number of cycles before the write data is output (WDON). Valid between 1 and 7.

[data15]
The number of cycles before the chip select is asserted (CSON). Valid between 0 and 7.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Bus Controller

Reference

R_BSC_Create, R_BSC_Destroy

Remarks

- Ensure that function R_BSC_Create is called once before using this function.
- The endian mode of the CPU is selected by the MDE pin (low = little endian; high = big endian).
- Port Function Control registers PFCR0 and PFCR5 are modified by this function.
- The cycle count parameters are not checked for validity. Use the hardware manual to check these values.

Program example

```
/* RPDL definitions */
#include “r_pdl_bsc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure CS2: 8-bit width, no wait cycles */
    R_BSC_CreateArea(2,
         PDL_BSC_WIDTH_8,
         0,
         0,
         0,
         0,
         0,
         0,
         0,
         0,
         1,
         0,
         0,
         1,
         0
    );
}
```

3) R_BSC_Destroy

Synopsis

Stop the External Bus Controller.

Prototype

```
bool R_BSC_Destroy(
        uint8_t data     // Area selection
);
```

Description

Disable an external bus area.

[data]
Select the external bus area CSn (where n = 0 to 7) to be disabled.

Return value

True.

Category

Bus Controller

Reference

R_BSC_CreateArea

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_pdl_bsc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Disable the CS4 area */
    R_BSC_Destroy(
          4
    );
}
```

4) R_BSC_Control

Synopsis

Modify the External Bus Controller operation.

Prototype

```
bool R_BSC_Control(
        uint8_t data     // Area selection
);
```

Description

Provide interrupt flag clearing

[data]

- Error clearing

PDL_BSC_ERROR_CLEAR	Clear the bus error signals. This will also clear the interrupt flag.

Return value

True if all parameters are valid; otherwise false.

Category

Bus Controller

Reference

R_BSC_Create, R_BSC_Destroy

Remarks

- This function can be called from the error handling function (see R_BSC_Create).

Program example

```
/* RPDL definitions */
#include “r_pdl_bsc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Clear the bus error signals */
    R_BSC_Control(
         PDL_BSC_ERROR_CLEAR
    );
}
```

5) R_BSC_GetStatus

Synopsis

Read the External Bus Controller status flags.

Prototype

```
bool R_BSC_GetStatus(
        uint8_t * data     // A pointer to the data storage location
);
```

Description

Read the interrupt status

[data2]
The status flags shall be stored in the following format:

b7 – b1	b0
0	0: Idle 1: Bus error condition detected

Return value

True.

Category

Bus Controller

Reference

R_BSC_Create

Remarks

- If the flag is set to 1, it shall be automatically cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_bsc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    uint8_t status;

    /* Read the flags */
    R_BSC_GetStatus(
         &status
    );
}
```

4.2.8. DMA Controller

1) R_DMAC_Create

Synopsis

Configure the DMA controller.

Prototype

```
bool R_DMAC_Create(
        uint8_t data1,      // Channel selection
        uint32_t data2,     // Configuration selection
        uint8_t data3,      // Trigger selection
        void * data4,       // Source start address
        void * data5,       // Destination start address
        uint32_t data6,     // Transfer byte count
        void * data7,       // Source reload address
        void * data8,       // Destination reload address
        uint32_t data9,     // Transfer byte count reload value
        void * func,        // Callback function
        uint8_t data10      // Interrupt priority level
);
```

Description (1/3)

Set up a DMA channel.

[data1]
The channel number n (where n = 0 to 3).

[data2]
Configure the operation of channel DMAn.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Transfer system selection

PDL_DMAC_SINGLE or PDL_DMAC_CONSECUTIVE or PDL_DMAC_NON_STOP	Single-operand, consecutive-operand or non-stop transfers.

- Address direction selection

PDL_DMAC_SOURCE_ADDRESS_FIXED or	Leave the address unchanged.
PDL_DMAC_SOURCE_ADDRESS_PLUS or	Increment the address.
PDL_DMAC_SOURCE_ADDRESS_MINUS or	Decrement the address.
PDL_DMAC_SOURCE_ADDRESS_ROTATE	Increment the address and return to the start address when a single-operand transfer is completed.
PDL_DMAC_DESTINATION_ADDRESS_FIXED or	Leave the address unchanged.
PDL_DMAC_DESTINATION_ADDRESS_PLUS or	Increment the address.
PDL_DMAC_DESTINATION_ADDRESS_MINUS or	Decrement the address.
PDL_DMAC_DESTINATION_ADDRESS_ROTATE	Increment the address and return to the start address when a single-operand transfer is completed.

- Transfer data size

PDL_DMAC_SIZE_8 or PDL_DMAC_SIZE_16 or PDL_DMAC_SIZE_32	Select 8, 16 or 32 bits for the data to be transferred.

- Operand transfer data count (applicable only for single-operand transfers)

PDL_DMAC_COUNT_1 or PDL_DMAC_COUNT_2 or PDL_DMAC_COUNT_4 or PDL_DMAC_COUNT_8 or PDL_DMAC_COUNT_16 or PDL_DMAC_COUNT_32 or PDL_DMAC_COUNT_64 or PDL_DMAC_COUNT_128	Select the data count to be transferred in a single-operand transfer.

Description (2/3)

- End-of-transfer reload control (not applicable for single-operand transfers)

PDL_DMAC_SOURCE_ADDRESS_RELOAD_DISABLE or PDL_DMAC_SOURCE_ADDRESS_RELOAD_ENABLE	Control reloading of the source address.
PDL_DMAC_DESTINATION_ADDRESS_RELOAD_DISABLE or PDL_DMAC_DESTINATION_ADDRESS_RELOAD_ENABLE	Control reloading of the destination address.
PDL_DMAC_COUNT_RELOAD_DISABLE or PDL_DMAC_COUNT_RELOAD_ENABLE	Control reloading of the byte count.

- DTC trigger control

PDL_DMAC_DTC_TRIGGER_OFF or PDL_DMAC_DTC_TRIGGER_ON	Disable or enable activation of the DTC upon completion of a DMA transfer.

[data3]
Configure the start trigger for channel DMAn.

- Start trigger

PDL_DMAC_REQUEST_SW or	Software trigger
PDL_DMAC_REQUEST_CMT0 or	CMT channel 0 interrupt
PDL_DMAC_REQUEST_CMT1 or	CMT channel 1 interrupt
PDL_DMAC_REQUEST_CMT2 or	CMT channel 2 interrupt
PDL_DMAC_REQUEST_CMT3 or	CMT channel 3 interrupt
PDL_DMAC_REQUEST_IRQ0 or	External pin IRQ0 interrupt
PDL_DMAC_REQUEST_IRQ1 or	External pin IRQ1 interrupt
PDL_DMAC_REQUEST_IRQ2 or	External pin IRQ2 interrupt
PDL_DMAC_REQUEST_IRQ3 or	External pin IRQ3 interrupt
PDL_DMAC_REQUEST_ADC0 or	ADC unit 0 interrupt
PDL_DMAC_REQUEST_ADC1 or	ADC unit 1 interrupt
PDL_DMAC_REQUEST_ADC2 or	ADC unit 2 interrupt
PDL_DMAC_REQUEST_ADC3 or	ADC unit 3 interrupt
PDL_DMAC_REQUEST_TPU0 or	TPU channel 0 interrupt
PDL_DMAC_REQUEST_TPU1 or	TPU channel 1 interrupt
PDL_DMAC_REQUEST_TPU2 or	TPU channel 2 interrupt
PDL_DMAC_REQUEST_TPU3 or	TPU channel 3 interrupt
PDL_DMAC_REQUEST_TPU4 or	TPU channel 4 interrupt
PDL_DMAC_REQUEST_TPU5 or	TPU channel 5 interrupt
PDL_DMAC_REQUEST_TPU6 or	TPU channel 6 interrupt
PDL_DMAC_REQUEST_TPU7 or	TPU channel 7 interrupt
PDL_DMAC_REQUEST_TPU8 or	TPU channel 8 interrupt
PDL_DMAC_REQUEST_TPU9 or	TPU channel 9 interrupt
PDL_DMAC_REQUEST_TPU10 or	TPU channel 10 interrupt
PDL_DMAC_REQUEST_TPU11 or	TPU channel 11 interrupt
PDL_DMAC_REQUEST_SCI0_RX or	SCI channel 0 data received interrupt
PDL_DMAC_REQUEST_SCI0_TX or	SCI channel 0 data transmitted interrupt
PDL_DMAC_REQUEST_SCI1_RX or	SCI channel 1 data received interrupt
PDL_DMAC_REQUEST_SCI1_TX or	SCI channel 1 data transmitted interrupt
PDL_DMAC_REQUEST_SCI2_RX or	SCI channel 2 data received interrupt
PDL_DMAC_REQUEST_SCI2_TX or	SCI channel 2 data transmitted interrupt
PDL_DMAC_REQUEST_SCI3_RX or	SCI channel 3 data received interrupt
PDL_DMAC_REQUEST_SCI3_TX or	SCI channel 3 data transmitted interrupt
PDL_DMAC_REQUEST_SCI4_RX or	SCI channel 4 data received interrupt
PDL_DMAC_REQUEST_SCI4_TX or	SCI channel 4 data transmitted interrupt
PDL_DMAC_REQUEST_SCI5_RX or	SCI channel 5 data received interrupt
PDL_DMAC_REQUEST_SCI5_TX or	SCI channel 5 data transmitted interrupt
PDL_DMAC_REQUEST_SCI6_RX or	SCI channel 6 data received interrupt
PDL_DMAC_REQUEST_SCI6_TX or	SCI channel 6 data transmitted interrupt
PDL_DMAC_REQUEST_IIC0_TX or	IIC channel 0 data transmitted interrupt
PDL_DMAC_REQUEST_IIC0_RX or	IIC channel 0 data received interrupt
PDL_DMAC_REQUEST_IIC1_TX or	IIC channel 1 data transmitted interrupt
PDL_DMAC_REQUEST_IIC1_RX	IIC channel 1 data received interrupt

[data4]
The source start address.

Description (3/3)

[data5]
The destination start address.

[data6]
The number of bytes to be transferred.

[data7]
The source reload address value. This value is ignored if the reload function is disabled.

[data8]
The destination reload address value. This value is ignored if the reload function is disabled.

[data9]
The number of bytes reload value. This value is ignored if the reload function is disabled.

[func]
The function to be called when a DMA transfer completes. Specify PDL_NO_FUNC if not required.

[data10]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

DMA controller

Reference

R_DMAC_Destroy, R_DMAC_Control, R_DMAC_GetStatus

Remarks

- If another peripheral will be used to trigger a DMA transfer, call this function before calling the Create function for the peripheral.
- If address increment or decrement is selected, the address changes according to the transfer data size.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.

Program example

```
/* RPDL definitions */
#include “r_pdl_dmac.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure DMA channel 2 */
    R_DMAC_Create(
         2,
         PDL_DMAC_SOURCE_ADDRESS_PLUS,
         PDL_DMAC_REQUEST_IRQ0,
         0x0000AA00,
         0x0000BB00,
         10,
         PDL_NO_PTR,
         PDL_NO_PTR,
         PDL_NO_DATA,
         PDL_NO_FUNC,
         0
    );
}
```

2) R_DMAC_Destroy

Synopsis	Shutdown the DMA controller.
Prototype	**bool R_DMAC_Destroy(** **void** // No parameter is required **);**
Description	Shutdown the DMAC module.
Return value	True if the shutdown succeeded; otherwise false.
Category	DMA controller
Reference	R_DMAC_Create, R_DMAC_Control
Remarks	• If all channels have been suspended, the DMAC module will be disabled. • If another peripheral is being used to trigger a DMA transfer, stop the triggers from that peripheral (using Control or Destroy for that peripheral) before calling this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_dmac.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Shutdown the DMA controller */
    R_DMAC_Destroy(
    );
}
```

3) R_DMAC_Control

Synopsis

Control the DMA controller.

Prototype

```
bool R_DMAC_Control (
        uint8_t data1,      // Channel number
        uint16_t data2,     // Control options
        uint8_t data3,      // Trigger selection
        void * data4,       // Source address
        void * data5,       // Destination address
        uint32_t data6,     // Transfer byte count
        void * data7,       // Source reload address
        void * data8,       // Destination reload address
        uint32_t data9      // Transfer byte reload count
);
```

Description (1/2)

Change the state of a DMA controller channel.

[data1]
Channel selection.
If multiple selections are required, use "|" to separate each selection.

PDL_DMAC_0 PDL_DMAC_1 PDL_DMAC_2 PDL_DMAC_3	The channel to be controlled.
or PDL_DMAC_ALL	

[data2]
Control the channel operation.
If multiple selections are required, use "|" to separate each selection.

- Enable / suspend control

PDL_DMAC_ENABLE	Enable / re-enable DMA transfers.
PDL_DMAC_SUSPEND	Suspend DMA transfers.

- Software trigger control

PDL_DMAC_START	Start a DMA transfer.

- Transfer end detection flag control

PDL_DMAC_CLEAR_DREQ	Clear the transfer request flag.

- Transfer end detection flag control

PDL_DMAC_CLEAR_TEDF	Clear the flag. This flag is cleared automatically if a callback function is enabled.

- Trigger control

PDL_DMAC_TRIGGER_RESET	If a peripheral is using a callback function and is also triggering the DMAC, the trigger will need to be reset after each interrupt. Specify this option and specify the trigger (used for this channel when R_DMAC_Create was called) in parameter data3.

Description (2/2)

- The registers to be modified, using the selected parameters.

PDL_DMAC_UPDATE_SOURCE	The Transfer Source Address register (DMCSA), using parameter data4.
PDL_DMAC_UPDATE_DESTINATION	The Transfer Destination Address register (DMCDA), using parameter data5.
PDL_DMAC_UPDATE_COUNT	The Byte Count register (DMCBC), using parameter data6.
PDL_DMAC_UPDATE_SOURCE_RELOAD	The Transfer Source Address reload register (DMRSA), using parameter data7.
PDL_DMAC_UPDATE_DESTINATION_RELOAD	The Transfer Destination Address reload register (DMRDA), using parameter data8.
PDL_DMAC_UPDATE_COUNT_RELOAD	The Byte Count reload register (DMRBC), using parameter data9.

[data3]
The DMAC start trigger. Specify PDL_NO_DATA if not required.

[data4]
The source address value. Specify PDL_NO_PTR if not required.

[data5]
The destination address value. Specify PDL_NO_PTR if not required.

[data6]
The number of bytes value. Specify PDL_NO_DATA if not required.

[data7]
The source address reload value. Specify PDL_NO_PTR if not required.

[data8]
The destination address reload value. Specify PDL_NO_PTR if not required.

[data9]
The number of bytes reload value. Specify PDL_NO_DATA if not required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

DMA controller

Reference

R_DMAC_Create

Remarks

- The Software trigger control is valid only if the Software trigger option has been selected.
- This function must be called in order to start the DMAC.
- The Suspend operation is executed at the start of this function.
  The Enable operation is executed at the end.
  Therefore, both options can be selected together with other control changes in one function call.

Program example

```
/* RPDL definitions */
#include “r_pdl_dmac.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Enable transfers on channel 2 */
    R_DMAC_Control(
        PDL_DMAC_2,
        PDL_DMAC_ENABLE,
        PDL_NO_DATA,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA
    );

    /* Suspend transfers on all enabled channels */
    R_DMAC_Control(
        PDL_DMAC_ALL,
        PDL_DMAC_SUSPEND,
        PDL_NO_DATA,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA
    );
}
```

4) R_DMAC_GetStatus

Synopsis

Check the status of a DMA channel.

Prototype

```
bool R_DMAC_GetStatus(
        uint8_t data1,      // Channel selection
        uint16_t * data2,   // Status flags pointer
        uint32_t * data3,   // Current source address pointer
        uint32_t * data4,   // Current destination address pointer
        uint32_t * data5    // Current transfer byte count pointer
);
```

Description

Return status flags and current channel registers.

[data1]
The channel number n (where n = 0 to 3).

[data2]
The status flags shall be stored in the following format.
Specify PDL_NO_PTR if the flags are not to be read.

b15 – b9	b8
0	0: Idle 1: Transfer End interrupt request set

b7	b6	b5	b4	b3	b2	b1	b0
Ch. 0	Ch. 1	Ch. 2	Ch. 3	Ch. 0	Ch. 1	Ch. 2	Ch. 3
0: Transfer end not detected 1: Transfer end detected				0 : Data transfer is not in progress 1: Data transfer in progress			

[data3]
Where the current source address shall be stored. Specify PDL_NO_PTR if it is not required.

[data4]
Where the current destination address shall be stored. Specify PDL_NO_PTR if it is not required.

[data5]
Where the current transfer byte count shall be stored. Specify PDL_NO_PTR if it is not required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

DMA controller

Reference

R_DMAC_Create

Remarks

- If a Transfer End Interrupt request flag is set to 1, the flag will be cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_dmac.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    uint16_t StatusValue;
    uint32_t SourceAddr;

    /* Read the status and current source address for channel 2 */
    R_DMAC_GetStatus(
        2,
        &StatusValue,
        &SourceAddr,
        PDL_NO_PTR,
        PDL_NO_PTR
    );
}
```

4.2.9. Data Transfer Controller

1) R_DTC_Set

Synopsis

Set the Data Transfer Controller options.

Prototype

```
bool R_DTC_Set (
        uint8_t data1,      // Configuration options
        uint32_t * data2    // Vector table base address
);
```

Description

Set the global options for the Data Transfer Controller.

[data1]
Configuration selections.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Chain transfer control

PDL_DTC_CHAIN_AFTER_REPEAT_DISABLE or PDL_DTC_CHAIN_AFTER_REPEAT_ENABLE	Disable or enable chain transfer after a repeat transfer completes.

- Read skip control

PDL_DTC_READ_SKIP_DISABLE or PDL_DTC_READ_SKIP_ENABLE	Disable or enable skipping of transfer data read when the vector numbers match.

- Address size control

PDL_DTC_ADDRESS_FULL or PDL_DTC_ADDRESS_SHORT	Select 32-bit (full) or 24-bit (short) address mode.

[data2]
The first address of the area of on-chip RAM where the DTC vector table shall be stored.
The address must be on a 4 kB boundary i.e. have the format xxxxx000h.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Data Transfer Controller

Reference

R_DTC_Create

Remarks

- Call this function once before calling R_DTC_Create.

Program example

```
/* RPDL definitions */
#include "r_pdl_dtc.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

/* Reserve an area for the DTC vector table */
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table [256];

void func(void)
{
    /* Configure the controller */
    R_DTC_Set(
        PDL_DTC_ADDRESS_SHORT,
        dtc_vector_table
    );
}
```

2) R_DTC_Create

Synopsis

Configure the Data Transfer Controller for a transfer.

Prototype

```
bool R_DTC_Create(
        uint32_t data1,     // Configuration selection
        uint32_t * data2,   // Transfer data start address
        void * data3,       // Source start address
        void * data4,       // Destination start address
        uint16_t data5,     // Transfer count
        uint8_t data6       // Block size
);
```

Description (1/3)

Configure DTC activation for one source.

[data1]
Configuration selections.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**.

- Transfer mode selection

PDL_DTC_NORMAL or PDL_DTC_REPEAT or PDL_DTC_BLOCK	Normal or Repeat or Block mode.
PDL_DTC_SOURCE or PDL_DTC_DESTINATION	If Repeat or Block mode is selected, select the source or destination side to be the Repeat or Block area.

- Address direction selection

PDL_DTC_SOURCE_ADDRESS_FIXED or PDL_DTC_SOURCE_ADDRESS_PLUS or PDL_DTC_SOURCE_ADDRESS_MINUS	After a data transfer, leave the source address unchanged, increment it or decrement it.
PDL_DTC_DESTINATION_ADDRESS_FIXED or PDL_DTC_DESTINATION_ADDRESS_PLUS or PDL_DTC_DESTINATION_ADDRESS_MINUS	After a data transfer, leave the destination address unchanged, increment it or decrement it.

- Transfer data size

PDL_DTC_SIZE_8 or PDL_DTC_SIZE_16 or PDL_DTC_SIZE_32	Select 1, 2 or 4 bytes to be transferred in one operation.

- Chain transfer control

PDL_DTC_CHAIN_DISABLE or PDL_DTC_CHAIN_CONTINUOUS or PDL_DTC_CHAIN_0	Disable, Enable continuous or Enable only when the transfer counter is 0.

- Interrupt generation

PDL_DTC_IRQ_COMPLETE or PDL_DTC_IRQ_TRANSFER	Select interrupt request generation when the transfer sequence completes, or for every transfer.

- Transfer trigger selection

Name	Module	Trigger cause
PDL_DTC_TRIGGER_CHAIN	None	Chain transfer
PDL_DTC_TRIGGER_CMT0	Compare match timer	Compare match on channel n (n = 0 to 3)
PDL_DTC_TRIGGER_CMT1		
PDL_DTC_TRIGGER_CMT2		
PDL_DTC_TRIGGER_CMT3		

Description (2/3)

PDL_DTC_TRIGGER_IRQ0	External interrupt pin	Valid edge detected on pin IRQn (n = 0 to 15)
PDL_DTC_TRIGGER_IRQ1		
PDL_DTC_TRIGGER_IRQ2		
PDL_DTC_TRIGGER_IRQ3		
PDL_DTC_TRIGGER_IRQ4		
PDL_DTC_TRIGGER_IRQ5		
PDL_DTC_TRIGGER_IRQ6		
PDL_DTC_TRIGGER_IRQ7		
PDL_DTC_TRIGGER_IRQ8		
PDL_DTC_TRIGGER_IRQ9		
PDL_DTC_TRIGGER_IRQ10		
PDL_DTC_TRIGGER_IRQ11		
PDL_DTC_TRIGGER_IRQ12		
PDL_DTC_TRIGGER_IRQ13		
PDL_DTC_TRIGGER_IRQ14		
PDL_DTC_TRIGGER_IRQ15		
PDL_DTC_TRIGGER_ADI0	Analog to Digital converter	Conversion completed on unit n (n = 0 to 3)
PDL_DTC_TRIGGER_ADI1		
PDL_DTC_TRIGGER_ADI2		
PDL_DTC_TRIGGER_ADI3		
PDL_DTC_TRIGGER_TGI0A	Timer Pulse Unit, channel 0	Input capture or compare match A
PDL_DTC_TRIGGER_TGI0B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI0C		Input capture or compare match C
PDL_DTC_TRIGGER_TGI0D		Input capture or compare match D
PDL_DTC_TRIGGER_TGI1A	Timer Pulse Unit, channel 1	Input capture or compare match A
PDL_DTC_TRIGGER_TGI1B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI2A	Timer Pulse Unit, channel 2	Input capture or compare match A
PDL_DTC_TRIGGER_TGI2B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI3A	Timer Pulse Unit, channel 3	Input capture or compare match A
PDL_DTC_TRIGGER_TGI3B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI3C		Input capture or compare match C
PDL_DTC_TRIGGER_TGI3D		Input capture or compare match D
PDL_DTC_TRIGGER_TGI4A	Timer Pulse Unit, channel 4	Input capture or compare match A
PDL_DTC_TRIGGER_TGI4B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI5A	Timer Pulse Unit, channel 5	Input capture or compare match A
PDL_DTC_TRIGGER_TGI5B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI6A	Timer Pulse Unit, channel 6	Input capture or compare match A
PDL_DTC_TRIGGER_TGI6B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI6C		Input capture or compare match C
PDL_DTC_TRIGGER_TGI6D		Input capture or compare match D
PDL_DTC_TRIGGER_TGI7A	Timer Pulse Unit, channel 7	Input capture or compare match A
PDL_DTC_TRIGGER_TGI7B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI8A	Timer Pulse Unit, channel 8	Input capture or compare match A
PDL_DTC_TRIGGER_TGI8B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI9A	Timer Pulse Unit, channel 9	Input capture or compare match A
PDL_DTC_TRIGGER_TGI9B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI9C		Input capture or compare match C
PDL_DTC_TRIGGER_TGI9D		Input capture or compare match D
PDL_DTC_TRIGGER_TGI10A	Timer Pulse Unit, channel 10	Input capture or compare match A
PDL_DTC_TRIGGER_TGI10B		Input capture or compare match B
PDL_DTC_TRIGGER_TGI11A	Timer Pulse Unit, channel 11	Input capture or compare match A
PDL_DTC_TRIGGER_TGI11B		Input capture or compare match B
PDL_DTC_TRIGGER_CMIA0	8-bit timer TMR, channel 0	Compare match A
PDL_DTC_TRIGGER_CMIB0		Compare match B
PDL_DTC_TRIGGER_CMIA1	8-bit timer TMR, channel 1	Compare match A
PDL_DTC_TRIGGER_CMIB1		Compare match B
PDL_DTC_TRIGGER_CMIA2	8-bit timer TMR, channel 2	Compare match A
PDL_DTC_TRIGGER_CMIB2		Compare match B
PDL_DTC_TRIGGER_CMIA3	8-bit timer TMR, channel 3	Compare match A
PDL_DTC_TRIGGER_CMIB3		Compare match B
PDL_DTC_TRIGGER_DMTEND0	Direct memory access controller	Transfer complete on channel n (n = 0 to 3)
PDL_DTC_TRIGGER_DMTEND1		
PDL_DTC_TRIGGER_DMTEND2		
PDL_DTC_TRIGGER_DMTEND3		

Description (3/3)

PDL_DTC_TRIGGER_RXI0	SCI, channel 0	Data received
PDL_DTC_TRIGGER_TXI0		Start of next data transfer
PDL_DTC_TRIGGER_RXI1	SCI, channel 1	Data received
PDL_DTC_TRIGGER_TXI1		Start of next data transfer
PDL_DTC_TRIGGER_RXI2	SCI, channel 2	Data received
PDL_DTC_TRIGGER_TXI2		Start of next data transfer
PDL_DTC_TRIGGER_RXI3	SCI, channel 3	Data received
PDL_DTC_TRIGGER_TXI3		Start of next data transfer
PDL_DTC_TRIGGER_RXI4	SCI, channel 4	Data received
PDL_DTC_TRIGGER_TXI4		Start of next data transfer
PDL_DTC_TRIGGER_RXI5	SCI, channel 5	Data received
PDL_DTC_TRIGGER_TXI5		Start of next data transfer
PDL_DTC_TRIGGER_RXI6	SCI, channel 6	Data received
PDL_DTC_TRIGGER_TXI6		Start of next data transfer
PDL_DTC_TRIGGER_ICRXI0	I²C bus interface, channel 0	Data received
PDL_DTC_TRIGGER_ICTXI0		Start of next data transfer
PDL_DTC_TRIGGER_ICRXI1	I²C bus interface, channel 1	Data received
PDL_DTC_TRIGGER_ICTXI1		Start of next data transfer

[data2]
The start address of the transfer data area. It must be a multiple of 4.
For short address mode, 12 bytes are required to store the transfer data.
For full address mode, 16 bytes are required.

[data3]
The source start address. The valid range depends on the address mode (short or full).

[data4]
The destination start address. The valid range depends on the address mode (short or full).

[data5]
The number of transfers to take place.
For normal or block mode, valid between 0 and 65535 (0 = 65536 transfers).
For repeat mode, valid between 0 and 255 (0 = 256 transfers).

[data6]
The block size for each transfer. Valid between 0 and 255 (0 = 256 units).
Ignored in normal or repeat mode.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Data Transfer Controller

Reference

R_DTC_Set, R_DTC_Control

Remarks

- If address increment or decrement is selected, the address changes according to the number of bytes (1, 2 or 4) in each transfer.
- Call this function before configuring the peripherals that will be involved in the data transfer.
- Call R_DTC_Set before calling this function.
- Call this function once for each peripheral that will trigger a transfer, and for each chained transfer.
- When all calls to this function are complete, call R_DTC_Control to start the DTC.

Program example

```
/* RPDL definitions */
#include “r_pdl_dtc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Reserve 16 bytes (full address mode) for the CMT0-triggered transfer data
area */
/* Use a 32-bit type to make the address a multiple of 4 */
uint32_t dtc_cmt0_transfer_data[4];

void func(void)
{
    /* Configure the DTC for CMT0 */
    R_DTC_Create(
        PDL_DTC_NORMAL | PDL_DTC_SOURCE_ADDRESS_FIXED | \
        PDL_DTC_DESTINATION_ADDRESS_PLUS | PDL_DTC_SIZE_8 | \
        PDL_DTC_TRIGGER_CMT0,
        dtc_cmt0_transfer_data,
        0x0000AA00,
        0x0000BB00,
        100,
        0
    );
}
```

3) R_DTC_Destroy

Synopsis	Shutdown the Data Transfer Controller.
Prototype	**bool R_DTC_Destroy(** **void** // No parameter is required **);**
Description	Shutdown the Data Transfer Controller.
Return value	True.
Category	Data Transfer Controller
Reference	R_DTC_Control
Remarks	• If another peripheral is being used to trigger a DTC transfer, stop the triggers from that peripheral (using Control or Destroy for that peripheral) before calling this function. • Use R_DTC_Control to stop the DTC before calling this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_dtc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Shutdown the DTC */
    R_DTC_Destroy(
    );
}
```

4) R_DTC_Control

Synopsis

Control the Data Transfer Controller.

Prototype

```
bool R_DTC_Control (
        uint32_t data1,     // Control options
        uint32_t * data2,   // Transfer data start address
        void * data3,       // Source start address
        void * data4,       // Destination start address
        uint16_t data5,     // Transfer count
        uint8_t data6       // Block size
);
```

Description

Modify the operation of the Data Transfer Controller.

[data1]
Control the operation.

- Stop / Start control

PDL_DTC_START or PDL_DTC_STOP	Enable / re-enable or suspend all DTC transfers.

- The registers to be modified, using the selected parameters.

PDL_DTC_UPDATE_SOURCE	The Source Address register, using parameter data3.
PDL_DTC_UPDATE_DESTINATION	The Transfer Address register, using parameter data4.
PDL_DTC_UPDATE_COUNT	The Transfer Count register, using parameter data5.
PDL_DTC_UPDATE_BLOCK_SIZE	The Block Size register, using parameter data6.

- Transfer trigger control
  After being triggered, the DTC will ignore further interrupts from that trigger source.
  Specify the trigger used in the relevant call of R_DTC_Create if you require the interrupt to trigger another transfer.

[data2]
The start address of the transfer data area (the same as that declared in R_DTC_Create).
Ignored if no registers are to be modified.

[data3]
The source start address. The valid range depends on the address mode (short or full).

[data4]
The destination start address. The valid range depends on the address mode (short or full).

[data5]
The number of transfers to take place.
For normal or block mode, valid between 0 and 65535 (0 = 65536 transfers).
For repeat mode, valid between 0 and 255 (0 = 256 transfers).

[data6]
The block size for each transfer. Valid between 0 and 255 (0 = 256 units).
Ignored in normal or repeat mode.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Data Transfer Controller

Reference

R_DTC_Create

Remarks

- This function must be called in order to start the DTC.

Program example

```
/* RPDL definitions */
#include “r_pdl_dtc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Start the controller */
    R_DTC_Control(
        PDL_DTC_START,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA,
        PDL_NO_DATA
    );

    /* Re-enable CMT0 as a DTC trigger */
    R_DTC_Control(
        PDL_DTC_TRIGGER_CMT0
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA,
        PDL_NO_DATA
    );

    /* Update the destination address for CMT0-triggered transfers */
    R_DTC_Control(
        PDL_DTC_STOP | PDL_DTC_START | PDL_DTC_UPDATE_DESTINATION,
        dtc_cmt0_transfer_data,
        PDL_NO_PTR,
        0x0000BB00,
        PDL_NO_DATA,
        PDL_NO_DATA
    );
}
```

5) R_DTC_GetStatus

Synopsis

Check the status of the Data Transfer Controller.

Prototype

```
bool R_DTC_GetStatus(
        uint32_t * data1,    // Transfer data start address
        uint8_t * data2,     // Status flags pointer
        uint32_t * data3,    // Current source address pointer
        uint32_t * data4,    // Current destination address pointer
        uint16_t * data5,    // Current transfer count pointer
        uint8_t * data6      // Current block size count pointer
);
```

Description

Return status flags and current channel registers.

[data1]
The start address of the transfer data area. Ignored if parameters data3, data4 and data5 are all PDL_NO_PTR.

[data2]
The status flags shall be stored in the following format.
Specify PDL_NO_PTR if the status flags are not required.

b7 – b1	b0
0	0: Normal 1: A DTC transfer stop request is generated

[data3]
Where the current source address shall be stored. Specify PDL_NO_PTR if it is not required.

[data4]
Where the current destination address shall be stored. Specify PDL_NO_PTR if it is not required.

[data5]
Where the current transfer count shall be stored. Specify PDL_NO_PTR if it is not required.

[data6]
Where the current block size count shall be stored. Specify PDL_NO_PTR if it is not required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Data Transfer Controller

Reference

R_DTC_Create

Remarks

- The start address of the transfer data area is the same as that declared in R_DTC_Create.

Program example

```
/* RPDL definitions */
#include “r_pdl_dtc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Declared in the R_DTC_Create example */
extern uint32_t dtc_cmt0_transfer_data[];

void func(void)
{
    uint8_t StatusValue;
    uint32_t SourceAddr;

    /* Read the status and current source address for the CMT0 transfer */
    R_DTC_GetStatus(
        dtc_cmt0_transfer_data,
        &StatusValue,
        &SourceAddr,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_PTR
    );
}
```

4.2.10. Timer Pulse Unit

1) R_TPU_Create

Synopsis

Configure a Timer Pulse Unit channel.

Prototype

```
bool R_TPU_Create(
    uint8_t data1,      // Channel selection
    uint32_t data2,     // Configuration selection
    uint32_t data3,     // Configuration selection
    uint32_t data4,     // Configuration selection
    uint32_t data5,     // Configuration selection
    uint16_t data6,     // Register value
    uint16_t data7,     // Register value
    uint16_t data8,     // Register value
    uint16_t data9,     // Register value
    uint16_t data10,    // Register value
    void * func1,       // Callback function
    void * func2,       // Callback function
    void * func3,       // Callback function
    void * func4,       // Callback function
    uint8_t data11,     // Interrupt priority level
    void * func5,       // Callback function
    void * func6,       // Callback function
    uint8_t data12      // Interrupt priority level
);
```

Description (1/5)

Set up a 16-bit TPU channel.

[data1]
The channel number n (where n = 0 to 11).

[data2]
Configure the channel mode.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Operation mode

PDL_TPU_MODE_NORMAL or	Normal operation.
PDL_TPU_MODE_PWM1 or PDL_TPU_MODE_PWM2 or	Pulse Width Modulation (PWM) mode 1 or 2.
PDL_TPU_MODE_PHASE1 or PDL_TPU_MODE_PHASE2 or PDL_TPU_MODE_PHASE3 or PDL_TPU_MODE_PHASE4	Phase counting mode 1, 2, 3 or 4. Valid for n = 1, 2, 4, 5, 7, 8, 10 and 11.

- Synchronous mode

PDL_TPU_SYNC_DISABLE or PDL_TPU_SYNC_ENABLE	Disable or enable synchronous operation.

- DMAC and / or DTC trigger control for TGRA

PDL_TPU_TGRA_DMAC_DTC_TRIGGER_DISABLE or PDL_TPU_TGRA_DMAC_TRIGGER_ENABLE or PDL_TPU_TGRA_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a TGRA compare match occurs.

- DTC trigger control for TGRB

PDL_TPU_TGRB_DTC_TRIGGER_DISABLE or PDL_TPU_TGRB_DTC_TRIGGER_ENABLE	Enable activation of the DTC when a TGRB compare match occurs.

- DTC trigger control for TGRC (valid for n = 0, 3, 6 and 9).

PDL_TPU_TGRC_DTC_TRIGGER_DISABLE or PDL_TPU_TGRC_DTC_TRIGGER_ENABLE	Enable activation of the DTC when a TGRB compare match occurs.

Description (2/5)

- DTC trigger control for TGRD (valid for n = 0, 3, 6 and 9).

PDL_TPU_TGRD_DTC_TRIGGER_DISABLE or PDL_TPU_TGRD_DTC_TRIGGER_ENABLE	Enable activation of the DTC when a TGRB compare match occurs.

[data3]
Configure the counter operation.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Counter clock source selection

PDL_TPU_CLK_PCLK_DIV_1 or PDL_TPU_CLK_PCLK_DIV_4 or PDL_TPU_CLK_PCLK_DIV_16 or PDL_TPU_CLK_PCLK_DIV_64 or	The internal clock signal PCLK ÷ 1, 4, 16 or 64.
PDL_TPU_CLK_PCLK_DIV_256 or	PCLK ÷ 256. Valid for n = 1, 3, 5, 7, 9 and 11.
PDL_TPU_CLK_PCLK_DIV_1024 or	PCLK ÷ 1024. Valid for n = 2, 3, 4, 8, 9 and 10.
PDL_TPU_CLK_PCLK_DIV_4096 or	PCLK ÷ 4096. Valid for n = 3 and 9.
PDL_TPU_CLK_TCLKA or	TCLKA-A or TCLKA-B pin input. Valid for n = 0 to 5.
PDL_TPU_CLK_TCLKB or	TCLKB-A or TCLKB-B pin input. Valid for n = 0, 1 and 2.
PDL_TPU_CLK_TCLKC or	TCLKC-A or TCLKC-B pin input. Valid for n = 0, 2, 4 and 5.
PDL_TPU_CLK_TCLKD or	TCLKD-A or TCLKD-B pin input. Valid for n = 0 and 5.
PDL_TPU_CLK_TCLKE or	TCLKE pin input. Valid for n = 6 to 11.
PDL_TPU_CLK_TCLKF or	TCLKF pin input. Valid for n = 6, 7 and 8.
PDL_TPU_CLK_TCLKG or	TCLKG pin input. Valid for n = 6, 8, 10, and 11.
PDL_TPU_CLK_TCLKH or	TCLKH pin input. Valid for n = 6 and 11.
PDL_TPU_CLK_TPU	The overflow / underflow signal from TPU(n+1). Valid for n = 1, 4, 7, and 10.

- Counter clock edge selection

PDL_TPU_CLK_FALLING or PDL_TPU_CLK_RISING or PDL_TPU_CLK_BOTH	The clock signal shall be counted on falling, rising or both edges.

- Counter clearing

PDL_TPU_CLEAR_DISABLE or	Clearing is disabled.
PDL_TPU_CLEAR_CM_A or	Cleared after a TGRA compare match occurs.
PDL_TPU_CLEAR_CM_B or	Cleared after a TGRB compare match occurs.
PDL_TPU_CLEAR_CM_C or	Cleared after a TGRC compare match occurs. Valid for n = 0, 3, 6 and 9.
PDL_TPU_CLEAR_CM_D or	Cleared after a TGRD compare match occurs. Valid for n = 0, 3, 6 and 9.
PDL_TPU_CLEAR_SYNC	Cleared by counter clearing on another channel configured for synchronous operation.

- Buffer operation (valid for channels 0, 3, 6 and 9)

PDL_TPU_BUFFER_AC_DISABLE or PDL_TPU_BUFFER_AC_ENABLE	Disable or enable buffer operation for registers TGRA and TGRC.
PDL_TPU_BUFFER_BD_DISABLE or PDL_TPU_BUFFER_BD_ENABLE	Disable or enable buffer operation for registers TGRB and TGRD.

- ADC trigger control

PDL_TPU_ADC_TRIG_DISABLE or PDL_TPU_ADC_TRIG_ENABLE	Disable or enable ADC conversion start requests on a TGRA input capture / compare match.

Description (3/5)

[data4]
Configure the operation for general registers A and B.
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Input capture / output compare control for register TGRA

PDL_TPU_A_OC_DISABLED or PDL_TPU_A_OC_LOW or PDL_TPU_A_OC_LOW_CM_HIGH or PDL_TPU_A_OC_LOW_CM_INV or PDL_TPU_A_OC_HIGH_CM_LOW or PDL_TPU_A_OC_HIGH or PDL_TPU_A_OC_HIGH_CM_INV or	TIOCAn output disabled. TIOCAn output low. TIOCAn initial output low; goes high at compare match. TIOCAn initial output low; toggles at compare match. TIOCAn initial output high; goes low at compare match. TIOCAn output high. TIOCAn initial output high; toggles at compare match.
PDL_TPU_A_IC_RISING_EDGE or PDL_TPU_A_IC_FALLING_EDGE or PDL_TPU_A_IC_BOTH_EDGES or	Input capture at TIOCAn rising edge. Input capture at TIOCAn falling edge. Input capture at TIOCAn both edges.
PDL_TPU_A_IC_TPU_COUNT_CLK or	Input capture at TPU(n+1) count clock count-up or count-down. Invalid if TPU(n+1) uses PCLK ÷ 1. Valid for n = 0, 3, 6 and 9.
PDL_TPU_A_IC_TPU_CM_IC	Input capture at TPU(n-1) TGRA compare match or input compare. Valid for n = 1, 4, 7 and 10.

- TIOCAn input capture pin selection

PDL_TPU_A_IC_SHARED or PDL_TPU_A_IC_SEPARATE	Input capture is shared with output compare or uses the adjacent pin.

- Input capture / output compare control for register TGRB

PDL_TPU_B_OC_DISABLED or PDL_TPU_B_OC_LOW or PDL_TPU_B_OC_LOW_CM_HIGH or PDL_TPU_B_OC_LOW_CM_INV or PDL_TPU_B_OC_HIGH_CM_LOW or PDL_TPU_B_OC_HIGH or PDL_TPU_B_OC_HIGH_CM_INV or	TIOCBn output disabled. TIOCBn output low. TIOCBn initial output low; goes high at compare match. TIOCBn initial output low; toggles at compare match. TIOCBn initial output high; goes low at compare match. TIOCBn output high. TIOCBn initial output high; toggles at compare match.
PDL_TPU_B_IC_RISING_EDGE or PDL_TPU_B_IC_FALLING_EDGE or PDL_TPU_B_IC_BOTH_EDGES or	Input capture at TIOCBn or TIOCAn rising edge. Input capture at TIOCBn or TIOCAn falling edge. Input capture at TIOCBn or TIOCAn both edges. See below for TIOCBn or TIOCAn pin selection.
PDL_TPU_B_IC_TPU_COUNT_CLK or	Input capture at TPU(n+1) count clock count-up or count-down. Invalid if TPU(n+1) uses PCLK ÷ 1. Valid for n = 0, 3, 6 and 9.
PDL_TPU_B_IC_TPU_CM_IC	Input capture at TPU(n-1) TGRC compare match or input compare. Valid for n = 1, 4, 7 and 10.

- TGRB input capture input selection

PDL_TPU_B_IC_TIOCB or PDL_TPU_B_IC_TIOCA	Input capture using pin TIOCBn or TIOCAn.

Description (4/5)

[data5]
Configure the operation for general registers C and D (valid for n = 0, 3, 6 and 9).
If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Input capture / output compare control for register TGRC.

PDL_TPU_C_OC_DISABLED or PDL_TPU_C_OC_LOW or PDL_TPU_C_OC_LOW_CM_HIGH or PDL_TPU_C_OC_LOW_CM_INV or PDL_TPU_C_OC_HIGH_CM_LOW or PDL_TPU_C_OC_HIGH or PDL_TPU_C_OC_HIGH_CM_INV or	TIOCCn output disabled. TIOCCn output low. TIOCCn initial output low; goes high at compare match. TIOCCn initial output low; toggles at compare match. TIOCCn initial output high; goes low at compare match. TIOCCn output high. TIOCCn initial output high; toggles at compare match.
PDL_TPU_C_IC_RISING_EDGE or PDL_TPU_C_IC_FALLING_EDGE or PDL_TPU_C_IC_BOTH_EDGES or	Input capture at TIOCCn rising edge. Input capture at TIOCCn falling edge. Input capture at TIOCCn both edges.
PDL_TPU_C_IC_TPU_COUNT_CLK	Input capture at TPU(n+1) count clock count-up or count-down. Invalid if TPU(n+1) uses PCLK ÷ 1.

- TIOCCn input capture pin selection.

PDL_TPU_C_IC_SHARED or PDL_TPU_C_IC_SEPARATE	Input capture is shared with output compare or uses the adjacent pin.

- Input capture / output compare control for register TGRD.

PDL_TPU_D_OC_DISABLED or PDL_TPU_D_OC_LOW or PDL_TPU_D_OC_LOW_CM_HIGH or PDL_TPU_D_OC_LOW_CM_INV or PDL_TPU_D_OC_HIGH_CM_LOW or PDL_TPU_D_OC_HIGH or PDL_TPU_D_OC_HIGH_CM_INV or	TIOCDn output disabled. TIOCDn output low. TIOCDn initial output low; goes high at compare match. TIOCDn initial output low; toggles at compare match. TIOCDn initial output high; goes low at compare match. TIOCDn output high. TIOCDn initial output high; toggles at compare match.
PDL_TPU_D_IC_RISING_EDGE or PDL_TPU_D_IC_FALLING_EDGE or PDL_TPU_D_IC_BOTH_EDGES or	Input capture at TIOCDn or TIOCCn rising edge. Input capture at TIOCDn or TIOCCn falling edge. Input capture at TIOCDn or TIOCCn both edges. See below for TIOCDn or TIOCCn pin selection.
PDL_TPU_D_IC_TPU_COUNT_CLK	Input capture at TPU(n+1) count clock count-up or count-down. Invalid if TPU(n+1) uses PCLK ÷ 1.

- TGRD input capture input selection

PDL_TPU_D_IC_TIOCD or PDL_TPU_D_IC_TIOCC	Input capture using pin TIOCDn or TIOCCn.

[data6]
The timer counter value.

[data7]
The register TGRA value.

[data8]
The register TGRB value.

[data9]
The register TGRC value (ignored for n ≠ 0, 3, 6 or 9).

[data10]
The register TGRD value (ignored for n ≠ 0, 3, 6 or 9).

[func1]
The function to be called when a TGRA event occurs. Specify PDL_NO_FUNC if not required.

Description (5/5)

[func2]
The function to be called when a TGRB event occurs. Specify PDL_NO_FUNC if not required.

[func3]
The function to be called when a TGRC event occurs. Specify PDL_NO_FUNC if not required.

[func4]
The function to be called when a TGRD event occurs. Specify PDL_NO_FUNC if not required.

[data11]
The interrupt priority level for TGRx events.
Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for all parameters func1, func2, func3 and func4.

[func5]
The function to be called when an overflow occurs. Specify PDL_NO_FUNC if not required.

[func6]
The function to be called when an underflow occurs. Specify PDL_NO_FUNC if not required.

[data12]
The interrupt priority level for overflow or underflow events.
Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for both parameters func5 and func6.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer Pulse Unit

Reference

Remarks

- If an external clock input pin (TCLKx) or I/O pin (TIOCxn) is made active, this function will configure that pin for input or output and disable other functions on that pin.
- The external clock inputs TCLKA, TCLKB, TCLKC and TCLKD are allocated to pins TCLKA-A, TCLKB-A, TCLKC-A and TCLKD-A by default.
  To select the –B group of pins, use API function R_PFC_Modify(5, PDL_PFC_OR, 0x08) to write 1 to bit TCLKS in register PFCR5 **before** calling this function.
  Note that these clock inputs are used by channels 0 to 5.
- If a callback function is specified, this function will enable the relevant CPU interrupt.
  Please see the notes on callback function usage in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- If the channel is configured for phase counting mode, the counter clock source setting is ignored
- If buffer operation is selected for registers TGRA and TGRC, input capture / output compare is not valid for register TGRC.
- If buffer operation is selected for registers TGRB and TGRD, input capture / output compare is not valid for register TGRD.
- If synchronous mode is required, at least two channels must be enabled for synchronous operation.

Program example

```
/* RPDL definitions */
#include “r_pdl_tpu.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure TPU0: PCLK, clear after a compare match A */
    R_TPU_Create(
          0,
          0,
          PDL_TPU_CLK_PCLK_DIV_1 | PDL_TPU_CLEAR_CM_A,
          0,
          199,
          99,
          PDL_NO_FUNC,
          PDL_NO_FUNC,
          PDL_NO_FUNC,
          PDL_NO_FUNC,
          0,
          PDL_NO_FUNC,
          PDL_NO_FUNC,
          0
    );
}
```

2) R_TPU_Destroy

Synopsis

Disable a timer unit.

Prototype

```
bool R_TPU_Destroy(
    uint8_t data    // Unit selection
);
```

Description

Shut down a timer pulse unit

[data]
The timer pulse unit n (where n = 0 or 1).
Unit 0 comprises channels TPU0 to TPU5.
Unit 1 comprises channels TPU6 to TPU11.

Return value

True if the unit selection is valid; otherwise false.

Category

Timer Pulse Unit

Reference

R_TPU_Create

Remarks

- The timer pulse unit is put into the stop state to reduce power consumption.

Program example

```
#include “r_pdl_tpu.h”

void func(void)
{
    /* Shutdown TPU channels 0 to 5 */
    R_TPU_Destroy(
        0
    );
}
```

3) R_TPU_Control

Synopsis

Control a timer channel.

Prototype

```
bool R_TPU_Control(
    uint8_t data1,      // Channel selection
    uint8_t data2,      // Register selection
    uint16_t data3,     // Register value
    uint16_t data4,     // Register value
    uint16_t data5,     // Register value
    uint16_t data6,     // Register value
    uint16_t data7      // Register value
);
```

Description

Modify a timer channel's registers.

[data1]
The channel number n (where n = 0 to 11).

[data2]
The channel settings to be modified.
If multiple selections are required, use "|" to separate each selection.

- Counter stop / re-start

PDL_TPU_STOP or PDL_TPU_START	Disable or re-enable the counter clock source.

- The registers to be modified.

PDL_TPU_COUNTER	Update the timer counter register (TCNT).
PDL_TPU_TGRA	Update the general register A (TGRA).
PDL_TPU_TGRB	Update the general register A (TGRB).
PDL_TPU_TGRC	Update the general register A (TGRC).
PDL_TPU_TGRD	Update the general register A (TGRD).

[data3]
The counter value. This will be ignored if the register is not selected.

[data4]
The general register A value. This will be ignored if the register is not selected.

[data5]
The general register B value. This will be ignored if the register is not selected.

[data6]
The general register C value. This will be ignored if the register is not selected.

[data7]
The general register D value. This will be ignored if the register is not selected.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer Pulse Unit

Reference

R_TPU_Create

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_pdl_tpu.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Load the counter on channel TPU channel 0 */
    R_TPU_Control(
         0,
         PDL_TPU_COUNTER,
         0xFFDD,
         0,
         0,
         0,
         0
    );
}
```

4) R_TPU_Read

Synopsis

Read from timer channel registers.

Prototype

```
bool R_TPU_Read(
    uint8_t data1,      // Channel selection
    uint8_t * data2,    // A pointer to the data storage location
    uint16_t * data3,   // A pointer to the data storage location
    uint16_t * data4,   // A pointer to the data storage location
    uint16_t * data5,   // A pointer to the data storage location
    uint16_t * data6,   // A pointer to the data storage location
    uint16_t * data7    // A pointer to the data storage location
);
```

Description

Read any of the timer's counter, compare or status flag registers.

[data1]
The channel number n (where n = 0 to 11).

[data2]
The status flags shall be stored in the format below.
The input capture / compare match flags A to D will be set to1 if the condition has been detected.
Specify PDL_NO_PTR if the flags are not to be read.

For n = 0, 3, 6 or 9

b7	b6	b5	b4	b3	b2	b1	b0
-	-	Overflow detection	Input capture / compare match detection				Count direction
-	-	V	D	C	B	A	0 = Counter counts down 1 = Counter counts up

For n = 1, 2, 4, 5, 7, 8, 10 or 11

b7	b6	b5	b4	b3	b2	b1	b0
-	Underflow detection	Overflow detection	Input capture / compare match detection				Count direction
-	U	V	-	-	B	A	0 = Counter counts down 1 = Counter counts up

[data3]
A pointer to where the counter value shall be stored. Specify PDL_NO_PTR if it is not required.

[data4]
Where the general register A value shall be stored. Specify PDL_NO_PTR if it is not required.

[data5]
Where the general register B value shall be stored. Specify PDL_NO_PTR if it is not required.

[data6]
Where the general register C value shall be stored. Specify PDL_NO_PTR if it is not required.

[data7]
Where the general register D value shall be stored. Specify PDL_NO_PTR if it is not required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer Pulse Unit

Reference

R_TPU_Create

Remarks

- If the flags are read, any detection flag that has been set to 1 shall be automatically cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_tpu.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

uint8_t Flags;
uint16_t General_A;
uint16_t General_D;

void func(void)
{
    /* Read the status flags and registers A and D for channel TPU0 */
    R_TPU_Read(
         0,
         &Flags,
         PDL_NO_PTR,
         &General_A,
         PDL_NO_PTR,
         PDL_NO_PTR,
         &General_D
    );
}
```

4.2.11. Programmable Pulse Generator

1) R_PPG_Create

Synopsis

Configure a PPG group.

Prototype

```
bool R_PPG_Create(
    uint32_t data1,    // Output pin selection
    uint16_t data2,    // Configuration selection
    uint8_t data3      // Output values
);
```

Description (1/2)

Set up a 4-bit PPG group.

[data1]
Select the outputs to be enabled.
If multiple selections are required, use "|" to separate each selection.
Select only outputs within one group.

- Output pin selection. Outputs are disabled by default.

PDL_PPG_PO0	Group 0.	Unit 0.
PDL_PPG_PO1		
PDL_PPG_PO2		
PDL_PPG_PO3		
PDL_PPG_PO4	Group 1.	
PDL_PPG_PO5		
PDL_PPG_PO6		
PDL_PPG_PO7		
PDL_PPG_PO8	Group 2.	
PDL_PPG_PO9		
PDL_PPG_PO10		
PDL_PPG_PO11		
PDL_PPG_PO12	Group 3.	
PDL_PPG_PO13		
PDL_PPG_PO14		
PDL_PPG_PO15		
PDL_PPG_PO16	Group 4.	Unit 1.
PDL_PPG_PO17		
PDL_PPG_PO18		
PDL_PPG_PO19		
PDL_PPG_PO20	Group 5.	
PDL_PPG_PO21		
PDL_PPG_PO22		
PDL_PPG_PO23		
PDL_PPG_PO24	Group 6.	
PDL_PPG_PO25		
PDL_PPG_PO26		
PDL_PPG_PO27		
PDL_PPG_PO28	Group 7.	
PDL_PPG_PO29		
PDL_PPG_PO30		
PDL_PPG_PO31		

Description (2/2)

[data2]
Operation control
If multiple selections are required, use "|" to separate each selection.

- Output trigger selection

PDL_PPG_TRIGGER_TPU0 or PDL_PPG_TRIGGER_TPU1 or PDL_PPG_TRIGGER_TPU2 or PDL_PPG_TRIGGER_TPU3 or	Select Compare Match on TPU channel 0 to 3 as the output trigger.
PDL_PPG_TRIGGER_TPU6 or PDL_PPG_TRIGGER_TPU7 or PDL_PPG_TRIGGER_TPU8 or PDL_PPG_TRIGGER_TPU9	Select Compare Match on TPU channel 6 to 9 as the output trigger (valid only for groups 4 to 7).

- Non-overlap control

PDL_PPG_NORMAL or PDL_PPG_NON_OVERLAP	Select normal (update on Compare Match A) or non-overlapping (update on Compare Match A or B) operation.

- Invert control

PDL_PPG_DIRECT or PDL_PPG_INVERT	Select direct or inverted output.

[data3]
The initial and next output values for the enabled pins, using the following format.

	b7	b6	b5	b4	b3	b2	b1	b0
Group	Next pulse output values				Initial output values			
0	PO3	PO2	PO1	PO0	PO3	PO2	PO1	PO0
1	PO7	PO6	PO5	PO4	PO7	PO6	PO5	PO4
2	PO11	PO10	PO9	PO8	PO11	PO10	PO9	PO8
3	PO15	PO14	PO13	PO12	PO15	PO14	PO13	PO12
4	PO19	PO18	PO17	PO16	PO19	PO18	PO17	PO16
5	PO23	PO22	PO21	PO20	PO23	PO22	PO21	PO20
6	PO27	PO26	PO25	PO24	PO27	PO26	PO25	PO24
7	PO31	PO30	PO29	PO28	PO31	PO30	PO29	PO28

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Programmable Pulse Generator

Reference

R_PPG_Control

Remarks

- If more than one group must be configured, use multiple calls of this function.
- The applicable PPG unit 0 or 1 is brought out of the stop state.
- This function disables the alternative modes on each PO pin that is enabled.

Program example

```
#include “r_pdl_ppg.h”

void func(void)
{
    /* Configure PPG outputs PO4 and PO6 (group 1) */
    R_PPG_Create(
        PDL_PPG_PO4 | PDL_PPG_PO6,
        PDL_PPG_TRIGGER_TPU2,
        0x15
    );
}
```

2) R_PPG_Destroy

Synopsis

Disable PPG outputs.

Prototype

```
bool R_PPG_Destroy(
    uint32_t data    // Output pin selection
);
```

Description

Disable the pulse output on the selected pins.

[data]
Select the outputs to be disabled.
If multiple selections are required, use "|" to separate each selection.
Select only outputs within one group.

- Output pin selection.

PDL_PPG_PO0	Group 0.	Unit 0.
PDL_PPG_PO1		
PDL_PPG_PO2		
PDL_PPG_PO3		
PDL_PPG_PO4	Group 1.	
PDL_PPG_PO5		
PDL_PPG_PO6		
PDL_PPG_PO7		
PDL_PPG_PO8	Group 2.	
PDL_PPG_PO9		
PDL_PPG_PO10		
PDL_PPG_PO11		
PDL_PPG_PO12	Group 3.	
PDL_PPG_PO13		
PDL_PPG_PO14		
PDL_PPG_PO15		
PDL_PPG_PO16	Group 4.	Unit 1.
PDL_PPG_PO17		
PDL_PPG_PO18		
PDL_PPG_PO19		
PDL_PPG_PO20	Group 5.	
PDL_PPG_PO21		
PDL_PPG_PO22		
PDL_PPG_PO23		
PDL_PPG_PO24	Group 6.	
PDL_PPG_PO25		
PDL_PPG_PO26		
PDL_PPG_PO27		
PDL_PPG_PO28	Group 7.	
PDL_PPG_PO29		
PDL_PPG_PO30		
PDL_PPG_PO31		

Return value

True if the unit selection is valid; otherwise false.

Category

Programmable Pulse Generator

Reference

R_PPG_Create

Remarks

- If all the outputs in a unit become disabled, that unit will be put into the stop state to reduce power consumption.

Program example

```
#include “r_pdl_ppg.h”

void func(void)
{
    /* Disable outputs PO24 and PO26 */
    R_PPG_Destroy(
        PDL_PPG_PO24 | PDL_PPG_PO26
    );
}
```

3) R_PPG_Control

Synopsis

Control a PPG group.

Prototype

```
bool R_PPG_Control(
    uint32_t data1,    // Group selection
    uint8_t data2      // Next output values
);
```

Description

Set the next output for a PPG group.

[data1]
Select the group(s) to be modified.
If multiple selections are required, use "|" to separate each selection.

- Group selection

PDL_PPG_GROUP_0 or PDL_PPG_GROUP_1 or	If a pair of groups (0-1, 2-3, 4-5 or 6-7) is using the same output trigger, both groups may be selected.
PDL_PPG_GROUP_2 or PDL_PPG_GROUP_3 or	
PDL_PPG_GROUP_4 or PDL_PPG_GROUP_5 or	
PDL_PPG_GROUP_6 or PDL_PPG_GROUP_7	

[data2]
The next output values (either for a single group, or a pair of groups), using the format:

	Group 1, 3, 5 or 7				Group 0, 2, 4 or 6			
Group pair	b7	b6	b5	b4	b3	b2	b1	b0
1 & 0	PO7	PO6	PO5	PO4	PO3	PO2	PO1	PO0
3 & 2	PO15	PO14	PO13	PO12	PO11	PO10	PO9	PO8
5 & 4	PO23	PO22	PO21	PO20	PO19	PO18	PO17	PO16
7 & 6	PO31	PO30	PO29	PO28	PO27	PO26	PO25	PO24

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Programmable Pulse Generator

Reference

R_PPG_Create

Remarks

- None.

Program example

```
/* RPDL definitions */
#include "r_pdl_ppg.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Load the next output values on group 6 */
    R_PPG_Control(
         PDL_PPG_GROUP_6,
         0x07
    );
}
```

4.2.12. 8-bit Timer

1) R_TMR_CreateChannel

Synopsis

Configure a timer TMR channel.

Prototype

```
bool R_TMR_CreateChannel(
    uint8_t data1,      // Channel selection
    uint32_t data2,     // Configuration selection
    uint8_t data3,      // Configuration selection
    uint8_t data4,      // Register value
    uint8_t data5,      // Register value
    uint8_t data6,      // Register value
    void * func1,       // Callback function
    void * func2,       // Callback function
    void * func3,       // Callback function
    uint8_t data7       // Interrupt priority level
);
```

Description (1/2)

Set up an 8-bit timer TMR channel.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
Configure the channel. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Counter clock source selection

PDL_TMR_CLK_OFF or	The clock input is disabled.
PDL_TMR_CLK_EXT_RISING or PDL_TMR_CLK_EXT_FALLING or PDL_TMR_CLK_EXT_BOTH or	The external clock signal TMCIn is used. Select rising, falling or both edges detected.
PDL_TMR_CLK_PCLK_DIV_1 or PDL_TMR_CLK_PCLK_DIV_2 or PDL_TMR_CLK_PCLK_DIV_8 or PDL_TMR_CLK_PCLK_DIV_32 or PDL_TMR_CLK_PCLK_DIV_64 or PDL_TMR_CLK_PCLK_DIV_1024 or PDL_TMR_CLK_PCLK_DIV_8192 or	The internal clock signal PCLK ÷ 1, 2, 8, 32, 64, 1024 or 8192.
PDL_TMR_CLK_TMR1_OVERFLOW or PDL_TMR_CLK_TMR3_OVERFLOW or	The overflow signal from TMR(n+1). Valid for n = 0 or 2.
PDL_TMR_CLK_TMR0_CM_A or PDL_TMR_CLK_TMR2_CM_A	The compare match A signal from TMR(n-1). Valid for n = 1 or 3.

- Counter clearing

PDL_TMR_CLEAR_DISABLE or	Clearing is disabled.
PDL_TMR_CLEAR_CM_A or	Cleared after a compare match A occurs.
PDL_TMR_CLEAR_CM_B or	Cleared after a compare match B occurs.
PDL_TMR_CLEAR_RESET_RISING or	Cleared by a rising edge on the external reset pin TMRIn.
PDL_TMR_CLEAR_RESET_HIGH	Cleared when the external reset pin TMRIn is high.

- ADC trigger control

PDL_TMR_ADC_TRIGGER_DISABLE or PDL_TMR_ADC_TRIGGER_ENABLE	Disable or enable ADC conversion start requests on a compare match A signal. Only applicable for channels TMR0 or TMR2.

- Compare Match A DTC trigger control

PDL_TMR_CM_A_DTC_TRIGGER_DISABLE or PDL_TMR_CM_A_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC when a Compare Match A occurs.

- Compare Match B DTC trigger control

PDL_TMR_CM_B_DTC_TRIGGER_DISABLE or PDL_TMR_CM_B_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC when a Compare Match B occurs.

Description (2/2)

[data3]
Configure the output control. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Output control for pin TMOn

PDL_TMR_OUTPUT_IGNORE_CM_A or PDL_TMR_OUTPUT_LOW_CM_A or PDL_TMR_OUTPUT_HIGH_CM_A or PDL_TMR_OUTPUT_INV_CM_A	No change if a compare match A occurs. 0 is output if a compare match A occurs. 1 is output if a compare match A occurs. The output toggles if a compare match A occurs.
PDL_TMR_OUTPUT_IGNORE_CM_B or PDL_TMR_OUTPUT_LOW_CM_B or PDL_TMR_OUTPUT_HIGH_CM_B or PDL_TMR_OUTPUT_INV_CM_B	No change if a compare match B occurs. 0 is output if a compare match B occurs. 1 is output if a compare match B occurs. The output toggles if a compare match B occurs.

[data4]
The counter value.

[data5]
The compare match A value.

[data6]
The compare match B value.

[func1]
The function to be called when an overflow occurs. Use PDL_NO_FUNC if not required.

[func2]
The function to be called when a Compare match A occurs. Use PDL_NO_FUNC if not required.

[func3]
The function to be called when a Compare match B occurs. Use PDL_NO_FUNC if not required.

[data7]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for all parameters func1, func2 and func3.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_Destroy, R_TMR_ControlChannel, R_TMR_ControlUnit, R_TMR_ReadChannel, R_TMR_ReadUnit

Remarks

- If an input pin (TMCIn or TMRIn) is selected, this function will configure the direction and input buffer control for that pin.
- If the output pin (TMOn) is made active, this function will disable other output functions on that pin.
- A closed clock loop will be created if:
  The overflow signal from TMR1 is selected for TMR0 and the compare match A signal from TMR0 is selected for TMR1, or
  The overflow signal from TMR3 is selected for TMR2 and the compare match A signal from TMR2 is selected for TMR3.
  Either case should be avoided.
- If a callback function is specified, this function will enable the relevant CPU interrupt.
  Please see the notes on callback function usage in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.

Program example

```
/* RPDL definitions */
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure TMR0: PCLK, clear after a compare match A */
    R_TMR_CreateChannel(
        0,
        PDL_TMR_CLK_PCLK_DIV_1 | PDL_TMR_CLEAR_CM_A,
        PDL_NO_DATA,
        0,
        199,
        99,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );
}
```

2) R_TMR_CreateUnit

Synopsis

Configure a timer TMR unit.

Prototype

```
bool R_TMR_CreateUnit(
    uint8_t data1,      // Channel selection
    uint32_t data2,     // Configuration selection
    uint8_t data3,      // Output control
    uint16_t data4,     // Register value
    uint16_t data5,     // Register value
    uint16_t data6,     // Register value
    void * func1,       // Callback function
    void * func2,       // Callback function
    void * func3,       // Callback function
    uint8_t data7       // Interrupt priority level
);
```

Description (1/2)

Set up a timer TMR unit in 16-bit count mode.

[data1]
The unit number n (where n = 0 or 1).

[data2]
Configure the unit. If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Counter clock source selection

PDL_TMR_CLK_OFF or	The clock input is disabled.
PDL_TMR_CLK_EXT_RISING or PDL_TMR_CLK_EXT_FALLING or PDL_TMR_CLK_EXT_BOTH or	The external clock signal TMCIx (x = 1 or 3 for n = 0 or 1) is used, with rising, falling or both edges detected.
PDL_TMR_CLK_PCLK_DIV_1 or PDL_TMR_CLK_PCLK_DIV_2 or PDL_TMR_CLK_PCLK_DIV_8 or PDL_TMR_CLK_PCLK_DIV_32 or PDL_TMR_CLK_PCLK_DIV_64 or PDL_TMR_CLK_PCLK_DIV_1024 or PDL_TMR_CLK_PCLK_DIV_8192	The internal clock signal PCLK ÷ 1, 2, 8, 32, 64, 1024 or 8192.

- Counter clearing

PDL_TMR_CLEAR_DISABLE or	Clearing is disabled.
PDL_TMR_CLEAR_CM_A or	Cleared after a compare match A occurs.
PDL_TMR_CLEAR_CM_B or	Cleared after a compare match B occurs.
PDL_TMR_CLEAR_RESET_RISING or	Cleared by a rising edge on the external reset pin TMRIy.
PDL_TMR_CLEAR_RESET_HIGH	Cleared when the external reset pin TMRIy (y = 0 or 2 for n = 0 or 1) is high.

- ADC trigger control

PDL_TMR_ADC_TRIGGER_DISABLE or PDL_TMR_ADC_TRIGGER_ENABLE	Disable or enable ADC conversion start requests on a compare match A signal.

- Compare Match A DTC trigger control

PDL_TMR_CM_A_DTC_TRIGGER_DISABLE or PDL_TMR_CM_A_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC when a Compare Match A occurs.

- Compare Match B DTC trigger control

PDL_TMR_CM_B_DTC_TRIGGER_DISABLE or PDL_TMR_CM_B_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC when a Compare Match B occurs.

Description (2/2)

[data3]
Configure the output control. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**.

- Output control for pin TMOy (y = 0 or 2 for n = 0 or 1)

PDL_TMR_OUTPUT_IGNORE_CM_A or PDL_TMR_OUTPUT_LOW_CM_A or PDL_TMR_OUTPUT_HIGH_CM_A or PDL_TMR_OUTPUT_INV_CM_A	No change if a compare match A occurs. 0 is output if a compare match A occurs. 1 is output if a compare match A occurs. The output toggles if a compare match A occurs.
PDL_TMR_OUTPUT_IGNORE_CM_B or PDL_TMR_OUTPUT_LOW_CM_B or PDL_TMR_OUTPUT_HIGH_CM_B or PDL_TMR_OUTPUT_INV_CM_B	No change if a compare match B occurs. 0 is output if a compare match B occurs. 1 is output if a compare match B occurs. The output toggles if a compare match B occurs.

[data4]
The 16-bit counter value.

[data5]
The 16-bit compare match A value.

[data6]
The 16-bit compare match B value.

[func1]
The function to be called when an overflow occurs. Use PDL_NO_FUNC if not required.

[func2]
The function to be called when a Compare match A occurs. Use PDL_NO_FUNC if not required.

[func3]
The function to be called when a Compare match B occurs. Use PDL_NO_FUNC if not required.

[data7]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for all parameters func1, func2 and func3.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_Destroy, R_TMR_ControlChannel, R_TMR_ControlUnit, R_TMR_ReadChannel, R_TMR_ReadUnit

Remarks

- If an input pin (TMCIx or TMRIy) is selected, this function will configure the direction and input buffer control for that pin.
- If the output pin (TMOy) is made active, this function will disable other output functions on that pin.
- If a callback function is specified, this function will enable the relevant CPU interrupt. Please see the notes on callback function usage in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.

Program example

```
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure TMR unit 0: PCLK, clear after a compare match A */
    R_TMR_CreateUnit(
         0,
         PDL_TMR_CLK_PCLK_DIV_1 | PDL_TMR_CLEAR_CM_A,
         0,
         0,
         199,
         99,
         PDL_NO_FUNC,
         PDL_NO_FUNC,
         PDL_NO_FUNC,
         0
    );
}
```

3) R_TMR_CreatePeriodic

Synopsis

Select periodic operation.

Prototype

```
bool R_TMR_CreatePeriodic(
    uint8_t data1,     // 8-bit (channel) or 16-bit (unit) selection
    uint32_t data2,    // Configuration selection
    float data3,       // Period or frequency
    float data4,       // Pulse width or duty cycle
    void * func1,      // Callback function
    void * func2,      // Callback function
    uint8_t data5      // Interrupt priority level
);
```

Description (1/2)

Set up a TMR timer channel or unit for periodic operation and start the timer.

[data1]

PDL_TMR_TMR0 or PDL_TMR_TMR1 or PDL_TMR_TMR2 or PDL_TMR_TMR3 or PDL_TMR_UNIT0 or PDL_TMR_UNIT1	The channel n (n = 0, 1, 2 or 3) or unit (n = 0 or 1) to be configured.

[data2]
Configure the timer. If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**.

- Period or frequency calculation

PDL_TMR_PERIOD or PDL_TMR_FREQUENCY	The parameters data3 and data4 will contain either period and pulse width or frequency and duty cycle.

- Output pin control

PDL_TMR_OUTPUT_HIGH or PDL_TMR_OUTPUT_LOW or **PDL_TMR_OUTPUT_OFF**	Start with a high-level or low-level output, or no output on pin TMOn. For 16-bit operation the pin shall be TMO2 when n = 1.

- ADC trigger control

PDL_TMR_ADC_TRIGGER_OFF or PDL_TMR_ADC_TRIGGER_ON	Disable or enable TMR-triggered ADC conversion start requests. Applicable only for channels TMR0 or TMR2, or either TMR unit.

- Pulse DTC trigger control

PDL_TMR_PULSE_DTC_TRIGGER_DISABLE or PDL_TMR_PULSE_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC at the pulse width interval.

- Period DTC trigger control

PDL_TMR_PERIOD_DTC_TRIGGER_DISABLE or PDL_TMR_PERIOD_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC at the periodic interval.

[data3]
The period (in seconds) or frequency (in Hz).

[data4]
The pulse width (in seconds) or duty cycle (%).

[func1]
The function to be called at the pulse width interval. Use PDL_NO_FUNC if not required.

[func2]
The function to be called at the periodic interval. Use PDL_NO_FUNC if not required.

Description (2/2)

[data5]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for both parameters func1and func2.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_Destroy

Remarks

- Function R_CGC_Set must be called before any use of this function.
- This function is an alternative to R_TMR_CreateChannel and R_TMR_CreateUnit.
- If an output pin (TMOn) is enabled, this function will disable other output functions on that pin.
- If a callback function is specified, this function will enable the relevant CPU interrupt. Please see the notes on callback function use in §6.
- The timing limits depend on the peripheral module clock, PCLK.

	Equation	f_{PCLK} (MHz)		
		50	**12.5**	**8**
Timer resolution	$\frac{1}{f_{PCLK}}$	20ns	80ns	125ns
$Period_{MIN}$	$\frac{2}{f_{PCLK}}$	40ns	160ns	250ns
$Period_{MAX\_CHANNEL}$	$\frac{2^{21}}{f_{PCLK}}$	41.9ms	167.7ms	262ms
$Period_{MAX\_UNIT}$	$\frac{2^{29}}{f_{PCLK}}$	10.7s	42.9s	67.1s
$Width_{MIN}$	$Period_{MIN}$			
$Width_{MAX\_CHANNEL}$	$Period_{MAX\_CHANNEL}$			
$Width_{MAX\_UNIT}$	$Period_{MAX\_UNIT}$			
f_{MAX}	$\frac{f_{PCLK}}{2}$	25 MHz	6.25 MHz	4 MHz
$f_{MIN\_CHANNEL}$	$\frac{f_{PCLK}}{2^{21}}$	23.8 Hz	5.96 Hz	3.81 Hz
f_{MIN_UNIT}	$\frac{f_{PCLK}}{2^{29}}$	0.0931 Hz	0.0232 Hz	0.0149 Hz

- If the requested period is not a multiple of the timer resolution, the actual time period will be more than the requested time period.
- The actual duty cycle will be less than the requested duty cycle if the resulting pulse width is not a multiple of the timer resolution.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.

Program example

```
/* RPDL definitions */
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Configure pin TMO1 for 500ns period, 200ns pulse width */
    R_TMR_CreatePeriodic(
        PDL_TMR_TMR1,
        PDL_TMR_PERIOD | PDL_TMR_OUTPUT_HIGH,
        500E-9,
        200E-9,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );

    /* Configure pin TMO1 for 5MHz frequency, 60% duty cycle */
    R_TMR_CreatePeriodic(
        PDL_TMR_TMR1,
        PDL_TMR_FREQUENCY | PDL_TMR_OUTPUT_HIGH,
        5E6,
        60,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );
}
```

4) R_TMR_CreateOneShot

Synopsis

Configure and use a one-shot timer.

Prototype

```
bool R_TMR_CreateOneShot(
    uint8_t data1,      // 8-bit (channel) or 16-bit (unit) timer selection
    uint32_t data2,     // Configuration selection
    float data3,        // Period
    void * func,        // Callback function
    uint8_t data4       // Interrupt priority level
);
```

Description

Set up a TMR timer channel or unit for one-shot operation and start the timer.

[data1]

PDL_TMR_TMR0 or PDL_TMR_TMR1 or PDL_TMR_TMR2 or PDL_TMR_TMR3 or PDL_TMR_UNIT0 or PDL_TMR_UNIT1	The channel n (n = 0, 1, 2 or 3) or unit n (n = 0 or 1) to be configured.

[data2]

Configure the timer. Use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Output pin control

PDL_TMR_OUTPUT_HIGH or PDL_TMR_OUTPUT_LOW or **PDL_TMR_OUTPUT_OFF**	For the duration of the one-shot period, generate a high-level output, low-level output or no output on pin TMOn. For 16-bit operation the pin shall be TMO2 when n = 1.

- DTC trigger control

PDL_TMR_PULSE_DTC_TRIGGER_DISABLE or PDL_TMR_PULSE_DTC_TRIGGER_ENABLE	Disable or enable activation of the DTC when the one-shot period ends.

- Control the CPU during the one-shot operation.

PDL_TMR_CPU_ON or	Allow the CPU to run normally while the one-shot operates.
PDL_TMR_CPU_OFF	Stop the CPU when the one-shot timer starts. The CPU will re-start when any valid interrupt occurs.

[data3]

The one-shot time period (in seconds).

[func]

The function to be called when the one-shot period ends.
Specify PDL_NO_FUNC for this function to wait for the timer to complete before returning.
You must always specify a function if PDL_TMR_CPU_OFF is selected, to ensure that an interrupt will re-start the CPU.

[data4]

The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_Destroy

Remarks

- Function R_CGC_Set must be called before any use of this function.
- This function is an alternative to R_TMR_CreateChannel and R_TMR_CreateUnit.
- This function stops the timer on completion, so no other TMR function calls are required.
- If an output pin (TMOn) is enabled, this function will disable other output functions on that pin.
- If a callback function is specified, this function will enable the relevant CPU interrupt. Please see the notes on callback function usage in §6.
- If no callback function is specified, this function waits for the CMIB flag to indicate that the one-shot time delay is complete. If the timer's control registers are directly modified by the user, this function may lock up.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- The timer period limits depend on the peripheral module clock, PCLK.

	Equation	f_{PCLK} (MHz)		
		50	12.5	8
T_{MIN}	$\frac{1}{f_{PCLK}}$	20ns	80ns	125ns
$T_{MAX\_CHANNEL}$	$\frac{2^{21}}{f_{PCLK}}$	41.9ms	167.7ms	262ms
T_{MAX_UNIT}	$\frac{2^{29}}{f_{PCLK}}$	10.7s	42.9s	67.1s

Program example

```
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Output a pulse and wait for 40ms */
    R_TMR_CreateOneShot(
        PDL_TMR_TMR0,
        PDL_TMR_OUTPUT_ON,
        40E-3,
        PDL_NO_FUNC,
        0
    );
}
```

5) R_TMR_Destroy

Synopsis

Disable a TMR timer unit.

Prototype

```
bool R_TMR_Destroy(
    uint8_t data    // Unit selection
);
```

Description

Shut down a TMR timer unit.

[data]
The timer unit n (where n = 0 or 1).
Unit 0 comprises channels TMR0 and TMR1.
Unit 1 comprises channels TMR2 and TMR3.

Return value

True if the unit selection is valid; otherwise false.

Category

Timer TMR

Reference

R_TMR_CreateChannel, R_TMR_CreateUnit

Remarks

- The timer unit is put into the stop state to reduce power consumption.

Program example

```
/* RPDL definitions */
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Shutdown channels 0 and 1 */
    R_TMR_Destroy(
        0
    );
}
```

6) R_TMR_ControlChannel

Synopsis

Write to timer channel registers.

Prototype

```
bool R_TMR_ControlChannel(
    uint8_t data1,     // Channel selection
    uint32_t data2,    // Configuration selection
    uint8_t data3,     // Register value
    uint8_t data4,     // Register value
    uint8_t data5      // Register value
);
```

Description

Modify a timer channel's operation, counter and compare registers.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
The channel settings to be modified.
If multiple selections are required, use "|" to separate each selection.

- Counter stop / re-start

PDL_TMR_STOP or PDL_TMR_START	Disable or re-enable the counter clock source.

- The counter or compare registers to be modified.

PDL_TMR_COUNTER	Update the timer counter register (TCNT).
PDL_TMR_TIME_CONSTANT_A	Update the timer compare match A register (TCORA).
PDL_TMR_TIME_CONSTANT_B	Update the timer compare match B register (TCORB).

[data3]
The counter value. This will be ignored if the register is not selected.

[data4]
The compare match A value. This will be ignored if the register is not selected.

[data5]
The compare match B value. This will be ignored if the register is not selected.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_CreateChannel, R_TMR_ReadChannel

Remarks

- None.

Program example

```
/* RPDL definitions */
#include "r_pdl_tmr.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Load the counter on channel TMR0 */
    R_TMR_ControlChannel(
        0,
        PDL_TMR_COUNTER,
        0xFF,
        PDL_NO_DATA,
        PDL_NO_DATA
    );
}
```

7) R_TMR_ControlUnit

Synopsis

Write to timer unit registers.

Prototype

```
bool R_TMR_ControlUnit(
    uint8_t data1,     // Unit selection
    uint32_t data2,    // Configuration selection
    uint16_t data3,    // Register value
    uint16_t data4,    // Register value
    uint16_t data5     // Register value
);
```

Description

Modify a timer unit's counter and compare registers.

[data1]
The unit number n (where n = 0 or 1).

[data2]
The channel settings to be modified.
If multiple selections are required, use "|" to separate each selection.

- Counter stop / re-start

PDL_TMR_STOP or PDL_TMR_START	Disable or re-enable the counter clock source.

- The counter or compare registers to be modified.

PDL_TMR_COUNTER	Update the timer counter register (TCNT).
PDL_TMR_TIME_CONSTANT_A	Update the timer compare match A register (TCORA).
PDL_TMR_TIME_CONSTANT_B	Update the timer compare match B register (TCORB).

[data3]
The 16-bit counter value. This will be ignored if the register is not selected.

[data4]
The 16-bit compare match A value. This will be ignored if the register is not selected.

[data5]
The 16-bit compare match B value. This will be ignored if the register is not selected.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_CreateUnit, R_TMR_ReadUnit

Remarks

- For unit 0, the upper byte is the value for TMR0 and the lower byte is the value for TMR1.
  For unit 1, the upper byte is the value for TMR2 and the lower byte is the value for TMR3.

Program example

```
/* RPDL definitions */
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Load the unit 1 counter and constants */
    R_TMR_ControlUnit(
        1,
        PDL_TMR_COUNTER | PDL_TMR_TIME_CONSTANT_A | \
        PDL_TMR_TIME_CONSTANT_B,
        0xAAFF,
        0x100,
        0x5600
    );
}
```

8) R_TMR_ControlPeriodic

Synopsis

Control periodic operation.

Prototype

```
bool R_TMR_ControlPeriodic(
    uint8_t data1,     // 8-bit (channel) or 16-bit (unit) selection
    uint32_t data2,    // Configuration selection
    float data3,       // The new period or frequency
    float data4        // The new pulse width or duty cycle
);
```

Description

Modify a periodic timer operation.

[data1]

PDL_TMR_TMR0 or PDL_TMR_TMR1 or PDL_TMR_TMR2 or PDL_TMR_TMR3 or PDL_TMR_UNIT0 or PDL_TMR_UNIT 1	The channel n (n = 0, 1, 2 or 3) or unit (n = 0 or 1) to be configured.

[data2]

Select the options to be modified. Use "|" to separate each selection.

- Period or frequency calculation

PDL_TMR_PERIOD or PDL_TMR_FREQUENCY	The parameters data3 and data4 will contain either period and pulse width or frequency and duty cycle.

- Output pin control

PDL_TMR_OUTPUT_ENABLE or PDL_TMR_OUTPUT_DISABLE	Enable or disable the periodic output on pin TMOn. For 16-bit operation the pin shall be TMO2 when n = 1.

- ADC trigger control

PDL_TMR_ADC_TRIGGER_OFF or PDL_TMR_ADC_TRIGGER_ON	Disable or enable periodic ADC conversion start requests. Applicable only for channels TMR0 or TMR2, or units 0 or 1.

- Counter stop / start

PDL_TMR_STOP or PDL_TMR_START	Disable or re-enable the counter clock source.

[data3]

The new period or frequency. This will be ignored if a timing change is not requested.

[data4]

The new pulse width or duty cycle (%). This will be ignored if a timing change is not requested.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_CreatePeriodic

Remarks

- See the remarks for R_TMR_CreatePeriodic.

Program example

```
/* RPDL definitions */
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Change timer TMR1 to 600ns period, 100ns pulse width */
    R_TMR_ControlPeriodic(
        PDL_TMR_TMR1,
        PDL_TMR_PERIOD,
        600E-9,
        100E-9
    );
}
```

9) R_TMR_ReadChannel

Synopsis

Read from timer channel registers.

Prototype

```
bool R_TMR_ReadChannel(
    uint8_t data1,      // Channel selection
    uint8_t * data2,    // A pointer to the data storage location
    uint8_t * data3,    // A pointer to the data storage location
    uint8_t * data4,    // A pointer to the data storage location
    uint8_t * data5     // A pointer to the data storage location
);
```

Description

Read any of the timer's counter, compare or status flag registers.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
The status flags shall be stored in the format below.
The flag will be set to 1 if the condition has been detected.
Specify PDL_NO_PTR if the flags are not to be read.

b7 – b3	b2	b1	b0
-	Overflow	Compare match B	Compare match A

[data3]
A pointer to where the counter value shall be stored. Specify PDL_NO_PTR if it is not required.

[data4]
Where the compare match A value shall be stored. Specify PDL_NO_PTR if it is not required.

[data5]
Where the compare match B value shall be stored. Specify PDL_NO_PTR if it is not required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_CreateChannel, R_TMR_ControlChannel

Remarks

- If the status flags are read, any flag that has been set to 1 shall be automatically cleared to 0 by this function.

Program example

```
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

uint8_t Flags;
uint8_t Counter;
uint8_t CompareMatchA;
uint8_t CompareMatchB;

void func(void)
{
    /* Read the status flags and registers for TMR0 */
    R_TMR_ReadChannel(
        0,
        &Flags,
        &Counter,
        &CompareMatchA,
        &CompareMatchB
    );
}
```

10) R_TMR_ReadUnit

Synopsis

Read from timer unit registers.

Prototype

```
bool R_TMR_ReadUnit(
    uint8_t data1,      // Unit selection
    uint8_t * data2,    // A pointer to the data storage location
    uint16_t * data3,   // A pointer to the data storage location
    uint16_t * data4,   // A pointer to the data storage location
    uint16_t * data5    // A pointer to the data storage location
);
```

Description

Read any of the timer's counter, compare or status flag registers.

[data1]
The unit number n (where n = 0 or 1).

[data2]
The status flags shall be stored in the format below.
A flag will be set to 1 if the condition has been detected.
Specify PDL_NO_PTR if the flags are not to be read.

The unit 0 status flags shall be stored in the format:

b7	b6	b5	b4	b3	b2	b1	b0
0	TMR0			0	TMR1		
	Overflow	Compare match B	Compare match A		Overflow	Compare match B	Compare match A

The unit 1 status flags shall be stored in the format:

b7	b6	b5	b4	b3	b2	b1	b0
0	TMR2			0	TMR3		
	Overflow	Compare match B	Compare match A		Overflow	Compare match B	Compare match A

[data3]
Where the counter value shall be stored. Specify PDL_NO_PTR if it is not required.

[data4]
Where the compare match A value shall be stored. Specify PDL_NO_PTR if it is not required.

[data5]
Where the compare match B value shall be stored. Specify PDL_NO_PTR if it is not required.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Timer TMR

Reference

R_TMR_CreateUnit, R_TMR_ControlUnit

Remarks

- If the status flags are read, any flag that has been set to 1 shall be automatically cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_tmr.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

uint8_t Flags;
uint16_t Counter;
uint16_t CompareMatchA;
uint16_t CompareMatchB;

void func(void)
{
    /* Read the status flags and registers for TMR unit 0 */
    R_TMR_ReadUnit(
        0,
        &Flags,
        &Counter,
        &CompareMatchA,
        &CompareMatchB
    );
}
```

4.2.13. Compare Match Timer

1) R_CMT_Create

Synopsis

Configure a CMT channel.

Prototype

```
bool R_CMT_Create(
    uint8_t data1,      // Timer channel selection
    uint16_t data2,     // Configuration selection
    float data3,        // Period, frequency or register data
    void * func,        // Callback function
    uint8_t data4       // Interrupt priority level
);
```

Description

Set up a Compare Match Timer channel and start the timer.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
Configure the timer. To set multiple options at the same time, use "|" to separate each value.
The default settings are shown in **bold**.

- Clock calculation

PDL_CMT_PERIOD or	The parameter data3 will specify the timer period. The counter clock source and compare match value will be calculated by this function.
PDL_CMT_FREQUENCY or	The parameter data3 will specify the timer frequency. The counter clock source and compare match value will be calculated by this function.
PDL_CMT_PCLK_DIV_8 or PDL_CMT_PCLK_DIV_32 or PDL_CMT_PCLK_DIV_128 or PDL_CMT_PCLK_DIV_512	Select the internal clock signal PCLK ÷ 8, 32, 128 or 512 as the counter clock source. The parameter data3 will be the register CMCOR value.

- DMAC / DTC trigger control

PDL_CMT_DMAC_DTC_TRIGGER_DISABLE or PDL_CMT_DMAC_TRIGGER_ENABLE or PDL_CMT_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a compare match occurs.

[data3]
The data to be used for the register value calculations.

Data use	Parameter type
The timer period in seconds or	float
The timer frequency in Hz or	float
The value to be put in register CMCOR	uint16_t

[func]
The function to be called at the periodic interval. Specify PDL_NO_FUNC if not required.

[data4]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Compare Match Timer

Reference

R_CMT_Destroy

Remarks

- Function R_CGC_Set must be called before any use of this function.
- If a callback function is specified, this function will enable the relevant CPU interrupt. Please see the notes on callback function use in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- The timing limits depend on the frequency of the peripheral module clock, PCLK.

	Equation	f_{PCLK} (MHz)			
		50	12.5	32	8
$Period_{MIN}$	$\frac{8}{f_{PCLK}}$	160ns	640ns	250ns	1.0µs
$Period_{MAX}$	$\frac{2^{25}}{f_{PCLK}}$	671ms	2.68s	1.05s	4.19s
f_{MAX}	$\frac{f_{PCLK}}{8}$	6.25 MHz	1.56 MHz	4.0 MHz	1.0 MHz
f_{MIN}	$\frac{f_{PCLK}}{2^{25}}$	1.49 Hz	0.37 Hz	0.95 Hz	0.24 Hz

- If the requested period is not a multiple of the minimum period, the actual time period will be more than the requested time period.

Program example

```
/* RPDL definitions */
#include “r_pdl_cmt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure CMT channel 0 for 10µs operation */
    R_CMT_Create(
         0,
         PDL_CMT_PERIOD,
         10E-6,
         PDL_NO_FUNC,
         0
    );

    /* Configure CMT channel 1 for 1kHz operation */
    R_CMT_Create(
         1,
         PDL_CMT_FREQUENCY,
         1E3,
         PDL_NO_FUNC,
         0
    );

    /* Configure CMT channel 2 using register values */
    R_CMT_Create(
         2,
         PDL_CMT_PCLK_DIV_32,
         0x55AA,
         PDL_NO_FUNC,
         0
    );
}
```

2) R_CMT_CreateOneShot

Synopsis

Configure a CMT channel as a one-shot event.

Prototype

```
bool R_CMT_CreateOneShot(
    uint8_t data1,      // Timer channel selection
    uint16_t data2,     // Configuration selection
    float data3,        // Period
    void * func,        // Callback function
    uint8_t data4       // Interrupt priority level
);
```

Description

Set up a Compare Match Timer channel and start the timer.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
Configure the timer.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Control the CPU during the one-shot operation.

PDL_CMT_CPU_ON or	Allow the CPU to run normally while the one-shot operates.
PDL_CMT_CPU_OFF	Stop the CPU when the one-shot timer starts. The CPU will re-start when any valid interrupt occurs.

- DMAC / DTC trigger control

PDL_CMT_DMAC_DTC_TRIGGER_DISABLE or PDL_CMT_DMAC_TRIGGER_ENABLE or PDL_CMT_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC when a compare match occurs.

[data3]
The one-shot time period (in seconds).

[func]
The function to be called when the one-shot period ends.
If you specify PDL_NO_FUNC, this function will wait for the timer to complete before returning.
You must always specify a function if PDL_CMT_CPU_OFF is selected to ensure that an interrupt will re-start the CPU.

[data4]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Compare Match Timer

Reference

None.

Remarks

- Function R_CGC_Set must be called before any use of this function.
- Function R_CMT_Create is not required.
- If a callback function is specified, this function will enable the relevant CPU interrupt. Please see the notes on callback function use in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- The timing limits depend on the peripheral module clock, PCLK.

	Equation	f_{PCLK} (MHz)			
		50	12.5	32	8
T_{MIN}	$\frac{8}{f_{PCLK}}$	160ns	640ns	250ns	1µs
T_{MAX}	$\frac{2^{25}}{f_{PCLK}}$	671ms	2.68s	1.05s	4.19s

- If the requested period is not a multiple of the minimum period, the actual time period will be more than the requested time period.

Program example

```
/* RPDL definitions */
#include “r_pdl_cmt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Use CMT channel 0 for a 1ms pause */
    R_CMT_CreateOneShot(
          0,
          0,
          1E-3,
          PDL_NO_FUNC,
          0
    );
}
```

3) R_CMT_Destroy

Synopsis

Disable a CMT unit.

Prototype

```
bool R_CMT_Destroy(
     uint8_t data     // Unit selection
);
```

Description

Shut down a CMT unit.

[data]
The timer unit n (where n = 0 or 1).
Unit 0 comprises channels CMT0 and CMT1.
Unit 1 comprises channels CMT2 and CMT3.

Return value

True if the unit selection is valid; otherwise false.

Category

Compare Match Timer

Reference

R_CMT_Create

Remarks

- The timer unit is put into the stop state to reduce power consumption.

Program example

```
/* RPDL definitions */
#include “r_pdl_cmt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Shutdown channels 0 and 1 */
    R_CMT_Destroy(
         0
    );
}
```

4) R_CMT_Control

Synopsis

Control CMT operation.

Prototype

```
bool R_CMT_Control(
     uint8_t data1,     // Channel selection
     uint16_t data2,    // Configuration selection
     float data3        // Period, frequency or register data
);
```

Description

Modify the operation of a CMT channel.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
Configure the timer channel. To set multiple options at the same time, use "|" to separate each value.

- Counter stop / re-start

PDL_CMT_STOP	Disable the counter clock source.
PDL_CMT_START	Enable the counter clock source.

- Value change request

PDL_CMT_PERIOD or PDL_CMT_FREQUENCY or PDL_CMT_CONSTANT or PDL_CMT_COUNTER	The parameter data3 will contain the new period, frequency, constant register (CMCOR) or counter register (CMCNT) value.

[data3]
The new period, frequency or register value. This will be ignored if a change is not requested.

Data use	Parameter type
The timer period in seconds or	float
The timer frequency in Hz or	float
The value to be put in the register	uint16_t

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Compare Match Timer

Reference

R_CMT_Create

Remarks

- R_CMT_Create must be first be used to configure the channel.
- The Stop operation is executed at the start of this function.
  The Start operation is executed at the end.
  Therefore, both options can be selected together with a value change in one function call.

Program example

```
/* RPDL definitions */
#include "r_pdl_cmt.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Change channel 2 to 1ms period */
    R_CMT_Control(
         2,
         PDL_CMT_PERIOD,
         1E-3
    );
}
```

5) R_CMT_Read

Synopsis

Read CMT channel status and registers.

Prototype

```
bool R_CMT_Read(
    uint8_t data1,      // Channel selection
    uint8_t * data2,    // A pointer to the data storage location
    uint16_t * data3    // A pointer to the data storage location
);
```

Description

Read and store the counter value and status flag.

[data1]
The channel number n (where n = 0, 1, 2 or 3).

[data2]
The compare match status flag shall be stored in the following format.
Specify PDL_NO_PTR if the flag is not to be read.

b7 – b1	b0
0	0: Idle 1: Compare match condition detected

[data3]
A pointer to where the counter value shall be stored. Specify PDL_NO_PTR if it is not required.

Return value

True if all parameters are valid; otherwise false.

Category

Compare Match Timer

Reference

R_CMT_Create

Remarks

- If the flag is read and is set to 1, it shall be automatically cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_cmt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

uint8_t Flags;
uint16_t Counter;

void func(void)
{
    /* Read the channel 2 values */
    R_CMT_Read(
         2,
         &Flags,
         &Counter
    );
}
```

4.2.14. Watchdog Timer

1) R_WDT_Create

Synopsis

Configure the Watchdog timer.

Prototype

```
bool R_WDT_Create(
    uint16_t data1,    // Configuration selection
    void * func,       // Callback function
    uint8_t data2      // Interrupt priority level
);
```

Description

Set up and start the Watchdog timer.

[data1]
Configure the timer. To set multiple options at the same time, use "|" to separate each value.
The default settings are shown in **bold**.

- Clock selection

PDL_WDT_PCLK_DIV_4 or PDL_WDT_PCLK_DIV_64 or PDL_WDT_PCLK_DIV_128 or PDL_WDT_PCLK_DIV_512 or PDL_WDT_PCLK_DIV_2048 or PDL_WDT_PCLK_DIV_8192 or PDL_WDT_PCLK_DIV_32768 or PDL_WDT_PCLK_DIV_131072	The division ratio for the internal clock signal PCLK.

- MCU reset control

PDL_WDT_RESET_DISABLE or PDL_WDT_RESET_ENABLE	Disable or enable reset of the MCU when the watchdog timer overflows with no callback function specified.

[func]
The function to be called at the periodic interval.

Specify PDL_NO_FUNC to have the timer output a WDTOVF# signal.
The MCU will also be reset (if selected above).

[data2]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Watchdog Timer

Reference

Remarks

- Function R_CGC_Set should be called before any use of this function.
- If a callback function is specified, this function will enable the relevant CPU interrupt. Please see the notes on callback function use in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- The timing limits depend on the frequency of the peripheral module clock, PCLK.

$$Period = \frac{n \times 256}{f_{PCLK}} \text{ or } Frequency = \frac{f_{PCLK}}{n \times 256}$$

Where n = 4, 64, 128, 512, 2048, 8192, 32768 or 131072.
Examples for different values of f_{PCLK} are given below.

	f_{PCLK} (MHz)			
	50	**12.5**	**32**	**8**
$Period_{PCLK÷4}$	20.5 µs	81.9 µs	32.0 µs	128 µs
$Period_{PCLK÷64}$	328 µs	1.31 ms	512.0 µs	2.05 ms
$Period_{PCLK÷128}$	655 µs	2.62 ms	1.02 ms	4.10 ms
$Period_{PCLK÷512}$	2.62 ms	10.5 ms	4.10 ms	16.4 ms
$Period_{PCLK÷2048}$	10.5 ms	41.9 ms	16.4 ms	65.5 ms
$Period_{PCLK÷8192}$	41.9 ms	168 ms	65.5 ms	262 ms
$Period_{PCLK÷32768}$	168 ms	671 ms	262 ms	1.05 s
$Period_{PCLK÷131072}$	671 ms	2.68 s	1.05 s	4.19 s
$f_{PCLK÷4}$	48.8 kHz	12.2 kHz	31.3 kHz	7.81 kHz
$f_{PCLK÷64}$	3.05 kHz	763 Hz	1.95 kHz	488 Hz
$f_{PCLK÷128}$	1.53 kHz	381 Hz	977 Hz	244 Hz
$f_{PCLK÷512}$	381 Hz	95.4 Hz	244 Hz	61.0 Hz
$f_{PCLK÷2048}$	95.4 Hz	23.8 Hz	61.0 Hz	15.3 Hz
$f_{PCLK÷8192}$	23.8 Hz	5.96 Hz	15.3 Hz	3.81 Hz
$f_{PCLK÷32768}$	5.96 Hz	1.49 Hz	3.81 Hz	0.954 Hz
$f_{PCLK÷131072}$	1.49 Hz	0.373 Hz	0.954 Hz	0.238 Hz

Program example

```
/* RPDL definitions */
#include “r_pdl_wdt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Configure the watchdog timer for PCLK/4 operation */
    R_WDT_Create(
         PDL_WDT_PCLK_DIV_4,
         WDT_handler,
         7
    );

    /* Configure the watchdog timer for PCLK/131072 operation with output
and reset enable */
    R_WDT_Create(
         1,
         PDL_WDT_PCLK_DIV_131072 | PDL_WDT_RESET_ENABLE,
         PDL_NO_FUNC,
         0
    );
}
```

2) R_WDT_Control

Synopsis

Control the Watchdog operation.

Prototype

```
bool R_WDT_Control(
    uint8_t data    // Control selection
);
```

Description

Modify the operation of the Watchdog timer.

[data]
Configure the timer channel. To set multiple options at the same time, use "|" to separate each value.

- Counter stop

PDL_WDT_STOP	Disable the counter clock source.

- Counter update

PDL_WDT_RESET_COUNTER	Reset the counter.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

Watchdog Timer

Reference

R_WDT_Create

Remarks

- R_WDT_Create must be first be used to configure the timer.

Program example

```
/* RPDL definitions */
#include "r_pdl_wdt.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Prevent the watchdog timer from overflowing */
    R_WDT_Control(
         PDL_WDT_RESET_COUNTER
    );
}
```

3) R_WDT_Read

Synopsis

Read the Watchdog timer status register.

Prototype

```
bool R_WDT_Read(
     uint8_t * data,     // A pointer to the data storage location
);
```

Description

Read and store the status flags.

[data]
The timer status shall be stored in the following format.

b7 – b1	b0
0	0: Not overflowed 1: Overflow has occurred

Return value

True if all parameters are valid; otherwise false.

Category

Watchdog Timer

Reference

R_WDT_Create

Remarks

- If the flag is set to 1, it shall be automatically cleared to 0 by this function.

Program example

```
/* RPDL definitions */
#include “r_pdl_wdt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

uint8_t Flags;

void func(void)
{
    /* Read the timer values */
    R_WDT_Read(
         &Flags
    );
}
```

4.2.15. Serial Communication Interface

1) R_SCI_Create

Synopsis

SCI channel setup.

Prototype

```
bool R_SCI_Create(
    uint8_t data1,     // Channel selection
    uint32_t data2,    // Channel configuration
    uint32_t data3,    // Bit rate or register value
    uint8_t data4      // Interrupt priority level
);
```

Description (1/3)

Set up the selected SCI channel.

[data1]
Select channel SCIn (where n = 0 to 6).

[data2]
Configure the channel. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**.

- Operation mode

PDL_SCI_ASYNC or PDL_SCI_SYNC or PDL_SCI_SMART	Choose between Asynchronous, Clock synchronous or Smart Card Interface operation.

- Data transfer format

PDL_SCI_LSB_FIRST or PDL_SCI_MSB_FIRST	Select least- or most-significant bit first. In 7-bit mode the format is fixed to LSB first.

- Data inversion

PDL_SCI_INVERSION_OFF or PDL_SCI_INVERSION_ON	Control data inversion (transmission and reception).

- Transmit / Receive connections

PDL_SCI_TX_CONNECTED or PDL_SCI_TX_DISCONNECTED or	The TXDn output is required / not required.
PDL_SCI_RX_CONNECTED or PDL_SCI_RX_DISCONNECTED	The RXDn input is required / not required.

Options which are available in Asynchronous mode

- Data clock source selection

PDL_SCI_CLK_INT_IO or	Select the on-chip baud rate generator.	The SCKn pin functions as an I/O pin.
PDL_SCI_CLK_INT_OUT or		The SCKn pin outputs the bit clock.
PDL_SCI_CLK_EXT_DIV_8 or PDL_SCI_CLK_EXT_DIV_16 or	Input a clock of 8 or 16 times the desired bit rate to the SCKn pin.	
PDL_SCI_CLK_TMR	For SCI5, select Timer output TMO0. SCK5 is set to high-impedance. For SCI6, select Timer output TMO2. SCK6 is set to high-impedance.	

- Data length

PDL_SCI_8_BIT_LENGTH or PDL_SCI_7_BIT_LENGTH	8- or 7-bit data length.

- Parity mode

PDL_SCI_PARITY_NONE or PDL_SCI_PARITY_EVEN or PDL_SCI_PARITY_ODD	No parity bit, even parity bit or odd parity bit.

- Stop bit length

PDL_SCI_STOP_1 or PDL_SCI_STOP_2	One or two stop bits.

Description (2/3)

The option “PDL_SCI_8N1” can be used to select 8-bit data length, no parity and one stop bit.

Options which are available in Clock Synchronous mode

- Data clock source selection

PDL_SCI_CLK_INT_OUT or	Select the On-chip baud rate generator. The SCKn pin outputs the bit clock.
PDL_SCI_CLK_EXT	Input the clock to the SCKn pin.

Options which are available in Smart Card Interface mode

- Base clock pulse cycle count

PDL_SCI_BCP_32 or PDL_SCI_BCP_64 or **PDL_SCI_BCP_93** or PDL_SCI_BCP_128 or PDL_SCI_BCP_186 or PDL_SCI_BCP_256 or PDL_SCI_BCP_372 or PDL_SCI_BCP_512	The number of base clock cycles in a 1-bit data transfer period.

- Parity selection

PDL_SCI_PARITY_EVEN or PDL_SCI_PARITY_ODD	Select even or odd parity bit.

- Block transfer mode selection

PDL_SCI_BLOCK_MODE_OFF or PDL_SCI_BLOCK_MODE_ON	Control Block transfer mode.

- GSM mode selection

PDL_SCI_GSM_MODE_OFF or PDL_SCI_GSM_MODE_ON	Control GSM mode.

- SCKn pin output control

	Normal mode	GSM mode
PDL_SCI_SCK_OUTPUT_OFF or	I/O pin	Not applicable
PDL_SCI_SCK_OUTPUT_LOW or	Not applicable.	Fixed low.
PDL_SCI_SCK_OUTPUT_ON or	Outputs the bit clock.	
PDL_SCI_SCK_OUTPUT_HIGH	Not applicable	Fixed high.

[data3]
The format must be either:

- The transfer bit rate in bits per second.
  If the on-chip baud rate generator is selected, the clock source and division values will be calculated using this value.

Or one selection from each of the following, using “|” to separate each selection.

- CKS selection

PDL_SCI_PCLK_DIV_1 or PDL_SCI_PCLK_DIV_4 or PDL_SCI_PCLK_DIV_16 or PDL_SCI_PCLK_DIV_64	Select the internal clock signal PCLK ÷ 1, 4, 16 or 64 as the baud rate generator clock source.

- ABCS selection (ignored for synchronous or smart card mode)

PDL_SCI_CYCLE_BIT_16 or PDL_SCI_CYCLE_BIT_8	Select 16 or 8 base clock cycles for one bit period.

- b31 to b24 b23 to b8 b7 – b0

0	A value between 0x100 and 0xFFFF00 that is nearest to the transfer bit rate.	The BRR register value.

Description (3/3)

[data4]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority). This parameter will be ignored if PDL_NO_FUNC is specified for parameter func in functions R_SCI_Send or R_SCI_Receive.

Return value

True if all parameters are valid, exclusive and achievable; otherwise false.

Category

SCI

Reference

R_SCI_Destroy, R_SCI_Send, R_SCI_Receive

Remarks

- Function R_CGC_Set must be called before any use of this function.
- This function configures each SCI pin that is required for operation. It also disables the alternative modes on those pins.
- For pin TXD5 it is not possible for this function to ensure that external bus signals CS4 or CS7 are not output on the same pin.
  If pin TXD5 is required for transmission, avoid selecting PDL_BSC_CS4_D or PDL_BSC_CS7_D when calling function R_BSC_Create.
- For pin SCK5 it is not possible for this function to ensure that external bus signal CS5 is not output on the same pin.
  If pin SCK5 is required, avoid selecting PDL_BSC_CS5_D when calling function R_BSC_Create.
- The wait time of 1 data bit period that is required during configuration is handled within this function.
- The range of achievable bit rates is listed below.

Mode	Data clock source	Limit	f_{PCLK}			
			50 MHz	12.5 MHz	32 MHz	8 MHz
Asynchronous	Internal	Minimum	95	24	61	15
		Maximum	3,125,000	781,250	2,000,000	500,000
	External		1,562,500	390,625	1,000,000	250,000
Synchronous	Internal	Minimum	763	191	488	122
		Maximum	6,250,000	1,562,500	4,000,000	1,000,000
	External		8,333,333	2,083,333	5,333,333	1,333,333
Smart card	Internal	Minimum	3	0.75	2	0.5
		Maximum	781,250	195,313	500,000	125,000

Program example

```
/* RPDL definitions */
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Configure SCI0 for asynchronous, 8N1, 38400 baud */
    R_SCI_Create(
        0,
        PDL_SCI_8N1,
        38400,
        1
    );

    /* Configure SCI1 for asynchronous, 8N1, register values supplied */
    R_SCI_Create(
        1,
        PDL_SCI_8N1,
        PDL_SCI_PCLK_DIV_1 | PDL_SCI_CYCLE_BIT_16 | \
         (115200 & 0x00FFF00) |0x50,
        1
    );
}
```

2) R_SCI_Destroy

Synopsis

Shut down a SCI channel.

Prototype

```
bool R_SCI_Destroy(
    uint8_t data    // Channel selection
);
```

Description

Stop data flow and shutdown the selected SCI channel.

[data]
Select channel SCIn (where n = 0 to 6).

Return value

True if all parameters are valid; otherwise false.

Category

SCI

Reference

R_SCI_Create

Remarks

- The SCI channel is put into the power-down state.

Program example

```
/* RPDL definitions */
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Shutdown SCI channel 1 */
    R_SCI_Destroy(
        1
    );
}
```

3) R_SCI_Send

Synopsis

Transmit data on a SCI channel.

Prototype

```
bool R_SCI_Send(
    uint8_t data1,      // Channel selection
    uint8_t data2,      // Channel configuration
    uint8_t * data3,    // Data start address
    uint16_t data4,     // Data count
    void * func         // Callback function
);
```

Description

Transmit data on the specified serial channel.

[data1]
Select channel SCIn (where n = 0 to 6).

[data2]
Control the transfer. If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- DMAC / DTC trigger control

PDL_SCI_DMAC_DTC_TRIGGER_DISABLE or PDL_SCI_DMAC_TRIGGER_ENABLE or PDL_SCI_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a data byte is transmitted.

[data3]
The start address of the data to be sent.
If the DMAC or DTC shall be used to transfer the data, specify PDL_NO_PTR.

[data4]
Set this to 0 if the transmit data is a character string (ending with a null character).

For sending binary data, set this to the number of bytes to be sent.
If the DMAC or DTC shall be used to transfer the data, specify PDL_NO_DATA.

[func]
The function to be called when the last byte has been sent.
Use R_SCI_Control to terminate this operation early.
R_SCI_GetStatus can be used to find out how many characters have been transmitted.

Specify PDL_NO_FUNC for this function to wait until the last byte has been sent.

Return value

True if all parameters are valid and the operation completed without errors;
False if a parameter was out of range or if the channel was already transmitting or if an error occurred during transmission.

Category

SCI

Reference

R_SCI_Create, R_SCI_Control, R_SCI_GetStatus

Remarks

- The compiler adds a null character to the end of string constants.
- If a callback function is specified, transmission interrupts are used.
  Please see the notes on callback function usage in §6.
- If a callback function is specified, avoid enabling activation of the DMAC or DTC for data transmission.
- If no callback function func is specified, this function will operate in polling mode.
  The TXI and TEND flags will be used to manage the data transmission.
  If the SCI channel's control registers are directly modified by the user, this function may lock up.
- The maximum number of characters to be transmitted is 65535.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- If reception is enabled and receive errors occur, transmission will be blocked until the errors are cleared.

Program example

```
/* RPDL definitions */
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    uint8_t data_store[100];

    /* Send a string on channel 2 */
    R_SCI_Send(
        2,
        PDL_NO_DATA,
        “Renesas RX”,
        0,
        PDL_NO_FUNC
    );

    /* Send 50 bytes of binary data on channel 1 */
    R_SCI_Send(
        2,
        PDL_NO_DATA,
        data_store,
        50,
        PDL_NO_FUNC
    );
}
```

4) R_SCI_Receive

Synopsis

Receive data on a SCI channel.

Prototype

```
bool R_SCI_Receive(
    uint8_t data1,      // Channel selection
    uint8_t data2,      // Channel configuration
    uint8_t * data3,    // Data start address
    uint16_t data4,     // Receive threshold
    void * func1,       // Callback function
    void * func2        // Callback function
);
```

Description

Enable SCI reception and acquire any incoming data.

[data1]
Select channel SCIn (where n = 0 to 6).

[data2]
Control the transfer.
The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- DMAC / DTC trigger control

PDL_SCI_DMAC_DTC_TRIGGER_DISABLE or PDL_SCI_DMAC_TRIGGER_ENABLE or PDL_SCI_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a data byte is received.

[data3]
The start address of the storage area for the expected data.
If the DMAC or DTC shall be used to handle the received data, specify PDL_NO_PTR.

[data4]
The number of bytes that must be received before the function completes or the callback function is called.
If the DMAC or DTC shall be used to handle the received data, specify PDL_NO_DATA.

[func1]
Specify PDL_NO_FUNC or a callback function name, depending on the required transfer method.

Transfer method	Parameter
Polling	PDL_NO_FUNC. This function will continue until the required number of bytes has been received.
Interrupts	The function to be called when the number of received bytes reaches the threshold number.
DMAC	Either the function to be called when each byte is received, or PDL_NO_FUNC if the callback function specified in R_DMAC_Create will be used.
DTC	The function to be called at the interval specified in R_DTC_Create.

[func2]
The function to be called if a receive error occurs. Specify PDL_NO_FUNC to ignore errors.

Return value

True if all parameters are valid and the operation completed; False if a parameter was out of range.

Category

SCI

Reference

R_SCI_Create, R_SCI_Control, R_SCI_GetStatus

Remarks

- The maximum number of characters to be received is 65535.
- Wait until a transmission on the same channel is complete before calling this function.
- If a callback function is used, please see the notes on callback function usage in §6.
- If polling mode is used, the RXI flag will be used to manage the data reception.
  If the SCI channel's control registers are directly modified by the user, this function may lock up.
- If no error callback function func2 is specified, the error flags are cleared automatically to allow the reception process to complete.
- Callback functions are executed by the interrupt processing function. This means that no other interrupt can be processed until a callback function has completed.
- While the receive operation is in progress:
  R_SCI_GetStatus can be used to find out how many bytes have been received so far.
  R_SCI_Control can be used to terminate the reception early.

Program example

```
/* PDL functions */
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

volatile uint8_t gSCI1ReceiveBuffer[10];

/* SCI channel 1 receive data handler */
void SCI1RxFunc(void){}

/* SCI channel 1 error handler */
void SCI1ErrFunc(void){}

void func( void )
{
    uint8_t temp;

    /* Put a Null character at the end of the string */
    gSCI1ReceiveBuffer[10] = NULL;

    /* Wait for 1 character to be received on channel 0 */
    R_SCI_Receive(
        0,
        PDL_NO_DATA,
        &temp,
        1,
        PDL_NO_FUNC,
        PDL_NO_FUNC
    );

    /* Start the reception of 9 characters on channel 1 */
    R_SCI_Receive(
        1,
        PDL_NO_DATA,
        gSCI1ReceiveBuffer,
        9,
        SCI1RxFunc,
        SCI1ErrFunc
    );
}
```

5) R_SCI_Control

Synopsis

Control the SCI channel.

Prototype

```
bool R_SCI_Control(
    uint8_t data1,    // Channel selection
    uint8_t data2     // Channel control
);
```

Description

Stops SCI transmission or reception.

[data1]
Select channel SCIn (where n = 0 to 6).

[data2]
Control the channel. If multiple selections are required, use "|" to separate each selection.

- Select the process to be stopped.

PDL_SCI_STOP_TX	Stop the transmission process. If a reception process is active, the transmit output will not become idle until the reception process has stopped.
PDL_SCI_STOP_RX	Stop the reception process. If a transmission process is active, the receive error flags may be set erroneously. These can be ignored and will be cleared when a new reception process is started.

The option "PDL_SCI_STOP_TX_AND_RX" can be used to select both processes.

If both processes are selected, transmission and reception will stop immediately.

- Generate a Space or Mark signal when idle.

PDL_SCI_OUTPUT_SPACE	Set the idle output to Space (logic 0). This can be used to generate a Break condition.
PDL_SCI_OUTPUT_MARK	Set the idle output to Mark (logic 1).

- Error flag control

PDL_SCI_CLEAR_RECEIVE_ERROR_FLAGS	Try to clear the receive error flags.

- Manual SCK control

PDL_SCI_GSM_SCK_STOP or PDL_SCI_GSM_SCK_START	Disable or enable the clock output (can be used while GSM mode is enabled).

Return value

True if all parameters are valid; otherwise false.

Category

SCI

Reference

R_SCI_Send, R_SCI_Receive, R_SCI_GetStatus

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Terminate SCI reception on channel 0 */
    R_SCI_Control(
        0,
        PDL_SCI_STOP_RX
    );
}
```

6) R_SCI_GetStatus

Synopsis

Check the status of a SCI channel.

Prototype

```
bool R_SCI_GetStatus(
    uint8_t data1,      // Channel selection
    uint8_t * data2,    // Status flags
    uint8_t * data3,    // Last byte received
    uint16_t * data4,   // Bytes transmitted
    uint16_t * data5    // Bytes received
);
```

Description

Acquires the channel status and the byte counts

[data1]
Select channel SCIn (where n = 0 to 6).

[data2]
The status flags shall be stored in the format:
Asynchronous or Synchronous mode:

b7 to b6	b5	b4	b3	b2	b1	b0
0	Reception error detection			Transmit status	0	RxD pin voltage level
	Overrun	Framing	Parity	0: Active 1: Complete		
	0: No error 1: Detected	0: No error 1: Detected	0: No error 1: Detected			0: Low 1: High

Smart card mode:

b7 to b6	b5	b4	b3	b2	b1	b0
0	Error detection			Transmit status	0	RxD pin voltage level
	Overrun	Error signal	Parity			
	0: No error 1: Detected	0: No error 1: Detected	0: No error 1: Detected	0: Active 1: Complete		0: Low 1: High

[data3]
The storage location for the last byte that was received. Specify PDL_NO_PTR if this information is not required.

[data4]
The storage location for the number of characters that are have been transmitted in the current transmission. Specify PDL_NO_PTR if this information is not required.

[data5]
The storage location for the number of characters that are have been received in the current reception process. Specify PDL_NO_PTR if this information is not required.

Return value

True if all parameters are valid and the operation completed; false if a parameter was out of range.

Category

SCI

Reference

R_SCI_Send, R_SCI_Receive

Remarks

- The error flags are not modified by this function. They are cleared when a new reception process is started.

Program example

```
/* RPDL definitions */
#include “r_pdl_sci.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

uint8_t StatusValue;
uint16_t TxChars;
uint16_t RxChars;

void func(void)
{
    /* Read the status of SCI channel 0 */
    R_SCI_GetStatus(
        0,
        &StatusValue,
        PDL_NO_PTR,
        &TxChars,
        &RxChars
    );
}
```

4.2.16. CRC calculator

1) R_CRC_Create

Synopsis

Configure the CRC calculator.

Prototype

```
bool R_CRC_Create(
    uint8_t data        // Configuration
);
```

Description

Enable the CRC and set the operating conditions.

[data]
Calculation options. To set multiple options at the same time, use "|" to separate each value.

- Polynomial selection

PDL_CRC_POLY_CRC_8 or PDL_CRC_POLY_CRC_16 or PDL_CRC_POLY_CRC_CCITT	$X^8 + X^2 + X + 1$ $X^{16} + X^{15} + X^2 + 1$ $X^{16} + X^{12} + X^5 + 1$

- Bit order

PDL_CRC_LSB_FIRST or PDL_CRC_MSB_FIRST	Select LSB or MSB-first operation.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

CRC

References

Remarks

- None.

Program example

```
/* RPDL definitions */
#include "r_pdl_crc.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Set up the CRC in 8-bit mode with LSB first */
    R_CRC_Create(
         PDL_CRC_POLY_CRC_8 | PDL_CRC_LSB_FIRST
    );
}
```

2) R_CRC_Destroy

Synopsis	Shut down the CRC calculator.
Prototype	**bool R_CRC_Destroy(** **void** // No parameter is required **);**
Description	Put the CRC calculator into the Power-down state, with minimal power consumption.
Return value	True.
Category	CRC
Reference	R_CRC_Create
Remarks	• None.
Program example	

```
/* RPDL definitions */
#include “r_pdl_crc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Shut down the CRC */
    R_CRC_Destroy(
    );
}
```

3) R_CRC_Write

Synopsis

Write data into the CRC calculation register.

Prototype

```
bool R_CRC_Write(
    uint8_t data      // The data to be used for the calculation
);
```

Description

Write the data into the data input register.

[data]
The data to be written into the register.

Return value

True.

Category

CRC

Reference

R_CRC_Create

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_crc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Write F0h into the CRC calculation register */
    R_CRC_Write(
        0xF0
    );
}
```

4) R_CRC_Read

Synopsis

Read the CRC calculation result.

Prototype

```
bool R_CRC_Read(
    uint8_t data1,      // Control
    uint16_t * data2    // Data storage location
);
```

Description

Reads and stores the CRC calculation result.

[data1]
Control the behaviour of the CRC unit.
The default setting is shown in **bold**. Specify PDL_NO_DATA to use the default.

- Result register clearing

PDL_CRC_CLEAR_RESULT or PDL_CRC_RETAIN_RESULT	Clear or retain the value in the result register.

[data2]
The address of the location where the result shall be stored.
For the 8-bit polynomial, the results are stored in the lower-order byte.

Return value

True.

Category

CRC

Reference

R_CRC_Create, R_CRC_Write

Remarks

- None.

Program example

```
/* RPDL definitions */
#include “r_crc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    uint16_t CRCresult;

    /* Read the CRC result and retain it */
    R_CRC_Read(
        PDL_CRC_RETAIN_RESULT,
        &CRCresult,
    );
}
```

4.2.17. I²C Bus Interface

1) R_IIC_Create

Synopsis

I²C channel setup.

Prototype

```
bool R_IIC_Create(
    uint8_t data1,     // Channel selection
    uint32_t data2,    // Channel configuration
    uint32_t data3,    // Detection configuration
    uint16_t data4,    // Slave address
    uint16_t data5,    // Slave address
    uint16_t data6,    // Slave address
    uint32_t data7,    // Transfer rate control
    uint32_t data8     // Rise and fall time correction
);
```

Description (1/3)

Set up the selected I²C channel.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
Configure the channel. If multiple selections are required, use "|" to separate each selection.
The default settings are shown in **bold**.

- Bus mode selection

PDL_IIC_MODE_IIC or PDL_IIC_MODE_IIC_FMP or PDL_IIC_MODE_SMBUS	Choose between I²C Bus, I²C Bus with Fast-mode Plus (for data rate > 400 kbps) or SMBus mode.

- Internal reference clock

PDL_IIC_INT_PCLK_DIV_1 or PDL_IIC_INT_PCLK_DIV_2 or PDL_IIC_INT_PCLK_DIV_4 or PDL_IIC_INT_PCLK_DIV_8 or PDL_IIC_INT_PCLK_DIV_16 or PDL_IIC_INT_PCLK_DIV_32 or PDL_IIC_INT_PCLK_DIV_64 or PDL_IIC_INT_PCLK_DIV_128	The reference clock source, used inside the I²C module.

- Timeout detection control

PDL_IIC_TIMEOUT_DISABLE or PDL_IIC_TIMEOUT_LOW or PDL_IIC_TIMEOUT_HIGH or PDL_IIC_TIMEOUT_BOTH	Disable timeout detection, or enable for SCL stuck at a low level high level or both low and high level.

- Timeout mode

PDL_IIC_TIMEOUT_LONG or PDL_IIC_TIMEOUT_SHORT	Select 16-bit (long) or 14-bit (short) mode.

- SDA output delay count

PDL_IIC_SDA_DELAY_0 or PDL_IIC_SDA_DELAY_1 or PDL_IIC_SDA_DELAY_2 or PDL_IIC_SDA_DELAY_3 or PDL_IIC_SDA_DELAY_4 or PDL_IIC_SDA_DELAY_5 or PDL_IIC_SDA_DELAY_6 or PDL_IIC_SDA_DELAY_7	Select the number of cycles for the SDA output delay counter.

- SDA output delay clock source

PDL_IIC_SDA_DELAY_DIV_1 or PDL_IIC_SDA_DELAY_DIV_2	Select the clock source (internal reference clock ÷ 1 or ÷ 2) for the SDA output delay counter.

Description (2/3)

- Noise filter control

PDL_IIC_NF_DISABLE or **PDL_IIC_NF_1** or PDL_IIC_NF_2 or PDL_IIC_NF_3 or PDL_IIC_NF_4	Select the number of stages in the noise filter.

[data3]
Detection settings. Specify PDL_NO_DATA to use the defaults.

- NACK Transmission Arbitration Lost Detection control

PDL_IIC_NTALD_DISABLE or PDL_IIC_NTALD_ENABLE	Disable or enable arbitration to be lost when an ACK is detection during transmission of a NACK in receive mode.

- Slave Arbitration Lost Detection control

PDL_IIC_SALD_DISABLE or PDL_IIC_SALD_ENABLE	Disable or enable arbitration to be lost when a mismatch occurs during slave data transmission.

- Slave address detection control

PDL_IIC_SLAVE_0_DISABLE or PDL_IIC_SLAVE_0_ENABLE_7 or PDL_IIC_SLAVE_0_ENABLE_10	Disable or enable detection of slave address 0 in 7-bit or 10-bit format.
PDL_IIC_SLAVE_1_DISABLE or PDL_IIC_SLAVE_1_ENABLE_7 or PDL_IIC_SLAVE_1_ENABLE_10	Disable or enable detection of slave address 1 in 7-bit or 10-bit format.
PDL_IIC_SLAVE_2_DISABLE or PDL_IIC_SLAVE_2_ENABLE_7 or PDL_IIC_SLAVE_2_ENABLE_10	Disable or enable detection of slave address 2 in 7-bit or 10-bit format.
PDL_IIC_SLAVE_GCA_DISABLE or PDL_IIC_SLAVE_GCA_ENABLE	Disable or enable detection of the General Call address.

- Device-ID detection control

PDL_IIC_DEVICE_ID_DISABLE or PDL_IIC_DEVICE_ID_ENABLE	Disable or enable detection of the Device-ID address (1111 100b).

- Host Address detection control

PDL_IIC_HOST_ADDRESS_DISABLE or PDL_IIC_HOST_ADDRESS_ENABLE	Disable or enable detection of the SMBus host address.

[data4]
Slave address 0. Ignored if slave address 0 detection is disabled.

[data5]
Slave address 1. Ignored if slave address 1 detection is disabled.

[data6]
Slave address 2. Ignored if slave address 2 detection is disabled.

[data7]
Transfer rate control.
Either:
The maximum bit rate in bits per second.
For Master mode, the clock division values will be calculated using a 50% duty cycle.
For Slave mode, the rate will be used to calculate the clock stretching period.
Or:

b31	b30 - b13	b12 - b8	b7 - b5	b4 - b0
1	-	Bit rate high-level register (ICBRH) value.	-	Bit rate low-level register (ICBRL) value.

Description (3/3)

[data8]
Rise and fall time compensation.
If the transfer rate is specified in bits per second, the high-level and low-level durations can be adjusted to allow for application-dependent rise and fall times. If unsure, use 0.

b31 - b16	b15 - b0
The SCL rise time in nanoseconds. Valid from 0 to 65535.	The SCL fall time in nanoseconds. Valid from 0 to 65535.

Return value

True if all parameters are valid, exclusive and achievable; otherwise false.

Category

I²C

Reference

R_IIC_Destroy

Remarks

- Function R_CGC_Set must be called before any use of this function.
- This function configures each I²C pin that is required for operation. It also disables the alternative modes on those pins.
- The 7 or 10-bit slave addresses should use the format:

b15 - b8	b7 - b1	b0
-	7-bit address	-

b15 - b11	b10 - b1	b0
-	10-bit address	-

- The timing limits depend on the frequency of the internal reference clock (IRC).

$$Transfer\_rate = \frac{1}{t_{rise} + t_{fall} + (ICBRH + 1)t_{IRC} + (ICBRL + 1)t_{IRC}}$$

The maximum transfer rate is given when ICBRH = ICBRL = 0; the minimum when ICBRH = ICBRL = 31.

The absolute limits (with zero rise and fall times) are:

	f_{PCLK} (MHz)			
f_{IRC}	**50**	**12.5**	**32**	**8**
f_{PCLK} ÷ 1	781 kbps to 25.0 Mbps	195 kbps to 6.25 Mbps	500 kbps to 16.0 Mbps	125 kbps to 4.00 Mbps
f_{PCLK} ÷ 2	391 kbps to 12.5 Mbps	97.7 kbps to 3.13 Mbps	250 kbps to 8.00 Mbps	62.5 kbps to 2.00 Mbps
f_{PCLK} ÷ 4	195 kbps to 6.25 Mbps	48.8 kbps to 1.56 Mbps	125 kbps to 4.00 Mbps	31.3 kbps to 1.00 Mbps
f_{PCLK} ÷ 8	97.7 kbps to 3.13 Mbps	24.4 kbps to 781 kbps	62.5 kbps to 2.00 Mbps	15.6 kbps to 500 kbps
f_{PCLK} ÷ 16	48.8 kbps to 1.56 Mbps	12.2 kbps to 391 kbps	31.3 kbps to 1.00 Mbps	7.81 kbps to 250 kbps
f_{PCLK} ÷ 32	24.4 kbps to 781 kbps	6.10 kbps to 195 kbps	15.6 kbps to 500 kbps	3.91 kbps to 125 kbps
f_{PCLK} ÷ 64	12.2 kbps to 391 kbps	3.05 kbps to 97.7 kbps	7.81 kbps to 250 kbps	1.95 kbps to 62.5 kbps
f_{PCLK} ÷ 128	6.10 kbps to 195 kbps	1.53 kbps to 48.8 kbps	3.91 kbps to 125 kbps	977 bps to 31.3 kbps

The actual rise and fall times will not be zero.
Using the limits from the I²C specification:
Rise time: (rate ≤ 100 kbps): 1000 ns; (100 kbps < rate ≤ 400 kbps): 300 ns; (400 kbps < rate ≤ 1 Mbps): 120 ns
Fall time: (rate ≤ 400 kbps): 300 ns; (400 kbps < rate ≤ 1 Mbps): 120 ns
Maximum rate: 1 Mbps

The achievable transfer rates are:

	f_{PCLK} (MHz)			
IRC	**50**	**12.5**	**32**	**8**
PCLK ÷ 1	658 kbps to 1 Mbps	175 kbps to 1 Mbps	446 kbps to 1 Mbps	116 kbps to 1 Mbps
PCLK ÷ 2	316 kbps to 1 Mbps	86.7 kbps to 1 Mbps	217 kbps to 1 Mbps	57.8 kbps to 1 Mbps
PCLK ÷ 4	175 kbps to 1 Mbps	45.9 kbps to 1 Mbps	116 kbps to 1 Mbps	30.0 kbps to 806 kbps
PCLK ÷ 8	86.7 kbps to 1 Mbps	23.7 kbps to 658 kbps	57.8 kbps to 1 Mbps	15.3 kbps to 446 kbps
PCLK ÷ 16	45.9 kbps to 1 Mbps	12.0 kbps to 316 kbps	30.0 kbps to 806 kbps	7.73 kbps to 217 kbps
PCLK ÷ 32	23.7 kbps to 658 kbps	6.06 kbps to 175 kbps	15.3 kbps to 446 kbps	3.89 kbps to 116 kbps
PCLK ÷ 64	12.0 kbps to 316 kbps	3.04 kbps to 86.7 kbps	7.73 kbps to 217 kbps	1.95 kbps to 57.8 kbps
PCLK ÷ 128	6.06 kbps to 175 kbps	1.52 kbps to 45.9 kbps	3.89 kbps to 116 kbps	975 bps to 30.0 kbps

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Select I²C mode at 100kHz, 100ns rise and fall times */
    R_IIC_Create(
        0,
        PDL_IIC_MODE_IIC | PDL_IIC_INT_PCLK_DIV_8,
        PDL_NO_DATA,
        PDL_NO_DATA,
        PDL_NO_DATA,
        PDL_NO_DATA,
        100E3,
        (100 << 16) | 100
    );

    /* Select I²C mode with two slave addresses */
    R_IIC_Create(
        1,
        PDL_IIC_MODE_IIC,
        PDL_IIC_SLAVE_0_ENABLE_7 | PDL_IIC_SLAVE_1_ENABLE_7,
        0x0020,
        0x0056,
        PDL_NO_DATA,
        100E3,
        (300 << 16) | 200
    );
}
```

2) R_IIC_Destroy

Synopsis	Disable an I²C channel.
Prototype	**bool R_IIC_Destroy(** **uint8_t data** // Channel selection **);**
Description	Shut down the selected I²C module. **[data]** Select channel IICn (where n = 0 or 1).
Return value	True if the parameter is valid; otherwise false.
Category	I²C
Reference	R_IIC_Create
Remarks	• The I²C module is put into the power-down state.
Program example	

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Shutdown IIC channel 1 */
    R_IIC_Destroy(
        1
    );
}
```

3) R_IIC_MasterSend

Synopsis

Write data to a slave device.

Prototype

```
bool R_IIC_MasterSend(
    uint8_t data1,      // Channel selection
    uint16_t data2,     // Channel configuration
    uint16_t data3,     // Slave address
    uint8_t * data4,    // Data start address
    uint16_t data5,     // Data count
    void * func,        // Callback function
    uint8_t data6       // Interrupt priority level
);
```

Description

Transmit data on the specified channel.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
Configure the channel. If multiple selections are required, use "|" to separate each selection. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Start / Repeated Start condition control

PDL_IIC_START_ENABLE or PDL_IIC_START_DISABLE	Choose whether or not to issue a Start or Repeated Start condition at the beginning of the transfer.

- Stop condition control

PDL_IIC_STOP_ENABLE or PDL_IIC_STOP_DISABLE	Choose whether or not to issue a Stop condition at the end of the transfer.

- DMAC / DTC trigger control

PDL_IIC_DMAC_DTC_TRIGGER_DISABLE or PDL_IIC_DMAC_TRIGGER_ENABLE or PDL_IIC_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a data byte is transmitted.

[data3]
The address of the slave device. Ignored if the Start condition is disabled.

[data4]
The start address of the data to be sent.
If the DMAC or DTC shall be used to transfer the data, specify PDL_NO_PTR.

[data5]
The number of bytes to be sent.
If the DMAC or DTC shall be used to transfer the data, specify PDL_NO_DATA.

[func]
Specify PDL_NO_FUNC or a callback function name, depending on the required transfer method.

Transfer method	Parameter
Polling	PDL_NO_FUNC. This function will continue until the required number of bytes has been sent (or another event occurs).
Interrupts	The function to be called when bus activity has stopped.
DMAC	Either the function to be called when each byte is sent, or PDL_NO_FUNC if the callback function specified in R_DMAC_Create will be used.
DTC	The function to be called at the interval specified in R_DTC_Create.

[data6]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid, exclusive and achievable and a normal transfer completed; otherwise false.

Category	I²C
Reference	R_IIC_Create, R_IIC_GetStatus
Remarks	• If a callback function is specified, please see the notes on callback function usage in §6. • If the Start condition is enabled and the previous transfer did not issue a Stop condition, a Repeated Start condition shall be generated. • If the Start condition is disabled, the slave address will not be transmitted. • If polling mode is selected, this function will monitor the status flags to manage the data flow. If an error occurs during this polling process, the function will terminate. If the I²C channel's registers are modified directly by the user, this function may lock up. • Use R_IIC_GetStatus to determine if the transfer was successful. If the transfer has ended prematurely, use R_IIC_Control to issue a Stop condition.

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

const uint8_t data_array[5] = {0x23, 0x48, 0x59, 0x60, 0xFE};

void func(void)
{
    /* Send 5 bytes to device 0x0A0 on channel 1, using polling */
    R_IIC_MasterSend(
        1,
        PDL_NO_DATA,
        0x0A0,
        data_array,
        5,
        PDL_NO_FUNC,
        0
    );
}
```

4) R_IIC_MasterReceive

Synopsis

Read data from a slave device.

Prototype

```
bool R_IIC_MasterReceive(
    uint8_t data1,      // Channel selection
    uint16_t data2,     // Channel configuration
    uint16_t data3,     // Slave address
    uint8_t * data4,    // Data start address
    uint16_t data5,     // Receive threshold
    void * func,        // Callback function
    uint8_t data6       // Interrupt priority level
);
```

Description

Read data over an I²C channel and store it.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
Configure the channel.
The default setting is shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- DMAC / DTC trigger control

PDL_IIC_DMAC_DTC_TRIGGER_DISABLE or PDL_IIC_DMAC_TRIGGER_ENABLE or PDL_IIC_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a data byte is received.

[data3]
The address of the slave device.

[data4]
The start address of the storage area for the expected data.
If the DMAC or DTC shall be used to handle the received data, specify PDL_NO_PTR.

[data5]
The number of bytes to be received.
If the DMAC or DTC shall be used to handle the received data, specify PDL_NO_DATA.

[func]
Specify PDL_NO_FUNC or a callback function name, depending on the required transfer method.

Transfer method	Parameter
Polling	PDL_NO_FUNC. This function will continue until the required number of bytes has been received (or another event occurs).
Interrupts	The function to be called when bus activity has stopped.
DMAC	Either the function to be called when each byte is received, or PDL_NO_FUNC if the callback function specified in R_DMAC_Create will be used.
DTC	The function to be called at the interval specified in R_DTC_Create.

[data6]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid, exclusive and achievable; otherwise false.

Category

I²C

Reference

R_IIC_Create, R_IIC_GetStatus, R_IIC_Control, R_IIC_MasterReceiveLast

Remarks

- If a callback function is specified, please see the notes on callback function usage in §6.
- If the previous transfer did not issue a Stop condition, a Repeated Start condition shall be generated.
- If polling or interrupts are used for data reception, the last byte to be read shall be completed with a NACK signal.
  If the DMAC or DTC is used, use R_IIC_MasterReceiveLast to read the last byte.
- If polling mode is selected, the status flags will be used to manage the data flow.
  If an error occurs during this polling process, the function will terminate.
  If the I²C channel's control registers are directly modified by the user, this function may lock up.
- Use R_IIC_GetStatus to determine if the transfer was successful.

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

volatile uint8_t data_array[5];

void func(void)
{
    /* Read 5 bytes from device 0xAA on channel 1, using polling */
    R_IIC_MasterReceive(
        1,
        PDL_NO_DATA,
        0xAA,
        data_array,
        5,
        PDL_NO_FUNC,
        0
    );
}
```

5) R_IIC_MasterReceiveLast

Synopsis

Complete a DMAC or DTC-based read process.

Prototype

```
bool R_IIC_MasterReceiveLast(
    uint8_t data1,      // Channel selection
    uint8_t * data2     // Data storage address
);
```

Description

Read one data byte with NACK and stop.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
The storage location for the data byte.

Return value

True if all parameters are valid and the function completed; otherwise false.

Category

I²C

Reference

R_IIC_MasterReceive

Remarks

- This function must only be used to terminate a Read process that has used the DMAC or DTC.
- Use R_IIC_GetStatus to determine if the transfer was successful.

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

volatile uint8_t data_array[5];

void func(void)
{
    /* Read 1 byte on channel 1 and stop */
    R_IIC_MasterReceiveLast(
        1,
        &data_array[4]
    );
}
```

6) R_IIC_SlaveMonitor

Synopsis

Monitor the bus.

Prototype

```
bool R_IIC_SlaveMonitor(
    uint8_t data1,      // Channel selection
    uint16_t data2,     // Channel configuration
    uint8_t * data3,    // Receive data start address
    uint16_t data4,     // Receive threshold
    void * func,        // Callback function
    uint8_t data5       // Interrupt priority level
);
```

Description

Monitor the bus until an address match occurs and store any data received.
Register the storage area and transfer method for data received on the selected I²C channel.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
Select the operation options.
The default setting is shown in **bold**. Specify PDL_NO_DATA to use the default.

- DMAC / DTC trigger control

PDL_IIC_RX_DMAC_DTC_TRIGGER_DISABLE or PDL_IIC_RX_DMAC_TRIGGER_ENABLE or PDL_IIC_RX_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a byte is received.
PDL_IIC_TX_DMAC_DTC_TRIGGER_DISABLE or PDL_IIC_TX_DMAC_TRIGGER_ENABLE or PDL_IIC_TX_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC for data transmission.

[data3]
The start address of the storage area for any received data.
If the DMAC or DTC shall be used to handle the received data, specify PDL_NO_PTR.

[data4]
The number of bytes in the storage area.
If the DMAC or DTC shall be used to handle the received data, specify PDL_NO_DATA.

[func]
Specify PDL_NO_FUNC or a callback function name, depending on the required transfer method.

Transfer method	Parameter
Polling	PDL_NO_FUNC. This function will continue until a Stop condition is detected or the master tries to read data from this slave.
Interrupts	The function to be called when a Stop condition is detected or the master tries to read data from this slave.
DMAC or DTC	The function to be called when a Stop or error condition is detected.

[data5]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid, exclusive and achievable; otherwise false.

Category

I²C

Reference

R_IIC_Create, R_IIC_GetStatus, R_IIC_SlaveSend

Remarks

- If a callback function is specified, interrupts are used.
  Use R_IIC_GetStatus in the callback function to identify the activity that has occurred.
  Please see the notes on callback function usage in §6.
- If no callback function is specified, this function will read the status flags to monitor the bus activity. Use R_IIC_GetStatus to identify the activity that has occurred.
  If the I²C channel's control registers are directly modified by the user, this function may lock up.
- If the master sends more data than is expected and the DMAC / DTC trigger is disabled, this function will issue a NACK to the master.
- When a Stop condition is detected, if the DMAC or DTC is used for transferring data, use R_DMAC_Control or R_DTC_Control to re-set the address and count before the next transfer begins.

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

volatile uint8_t data_array[5];

void func(void)
{
    /* Monitor channel 0, using polling */
    R_IIC_SlaveMonitor(
        0,
        PDL_NO_DATA,
        data_array,
        5,
        PDL_NO_FUNC,
        0
    );
}
```

7) R_IIC_SlaveSend

Synopsis

Write data to a master device.

Prototype

```
bool R_IIC_SlaveSend(
    uint8_t data1,      // Channel selection
    uint8_t * data2,    // Data start address
    uint16_t data3      // Data count
);
```

Description

Transmit data on the specified channel.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
The start address of the data to be sent.

[data3]
The number of bytes available to be sent.

Return value

True if all parameters are valid, exclusive and achievable; otherwise false.
If this function is not called from the R_IIC_SlaveMonitor callback function, it will complete when a stop condition is detected.

Category

I²C

Reference

R_IIC_SlaveMonitor

Remarks

- Use this function in conjunction with R_IIC_SlaveMonitor.
- If the master requires more data than is supplied, and polling or interrupt-based transfers are used, this function shall loop back to the start of the data. The transmitted byte count will also be reset to 0.

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

const uint8_t data_array[5] = {0x23, 0x48, 0x59, 0x60, 0xFE};

void func(void)
{
    /* Assign 5 bytes to be read by a master on channel 0 */
    R_IIC_SlaveSend(
        0,
        data_array,
        5
    );
}
```

8) R_IIC_Control

Synopsis

I²C channel control.

Prototype

```
bool R_IIC_Control(
    uint8_t data1,      // Channel selection
    uint8_t data2       // Control options
);
```

Description

Modify the operation of the selected I²C channel.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
Control the channel. If multiple selections are required, use "|" to separate each selection.

- Stop generation

PDL_IIC_STOP	Issue a Stop condition.

- NACK generation

PDL_IIC_NACK	Set the Acknowledge bit to the NACK state.

- Pin control

PDL_IIC_SDA_LOW or PDL_IIC_SDA_HI_Z	Set the SDA pin to low level or high-impedance.
PDL_IIC_SCL_LOW or PDL_IIC_SCL_HI_Z	Set the SCL pin to low level or high-impedance.

- Extra clock cycle generation

PDL_IIC_CYCLE_SCL	Generate an extra clock cycle on the SCL pin. This can be used in Master mode to try and unlock a slave device that is holding the SDA signal low.

- Reset control

PDL_IIC_RESET	Carry out an internal reset of the I²C module (the settings are preserved).

Return value

True if all parameters are valid, exclusive and achievable; otherwise false.

Category

I²C

Reference

R_IIC_Create

Remarks

- None.

Program example

```
/* RPDL definitions */
#include "r_pdl_iic.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Issue a Stop condition on channel 0 */
    R_IIC_Control(
        0,
        PDL_IIC_STOP
    );
}
```

9) R_IIC_GetStatus

Synopsis

Read the status for an I²C channel.

Prototype

```
bool R_IIC_GetStatus(
    uint8_t data1,      // Channel selection
    uint16_t * data2,   // Status flags
    uint16_t * data3,   // Transmitted bytes
    uint16_t * data4    // Received bytes
);
```

Description

Read the status registers for the selected I²C channel.

[data1]
Select channel IICn (where n = 0 or 1).

[data2]
The status flags shall be stored in the format below.
Specify PDL_NO_PTR if this information is not required.

b15	b14	b13	b12	b11	b10	b9	b8
Bus state	Pin level		Event detection (0 = Not detected, 1 = detected)				
0: Idle 1: Busy	SCL	SDA	NACK	Stop condition	Start condition	Arbitration lost	Timeout

b7	b6	b5	b4	b3	b2	b1	b0
Transmission	Mode	Address detection (0 = Not detected, 1= detected)					
0: Active 1: Idle	0: Receive 1: Transmit	SMBus host	Device-ID	General call	Slave		
					2	1	0

[data3]
The address for storing the number of bytes that are have been transmitted in the current transfer.
Specify PDL_NO_PTR if this information is not required.

[data4]
The address for storing for the number of bytes that are have been received in the current transfer.
Specify PDL_NO_PTR if this information is not required.

Return value

True if all parameters are valid; otherwise false.

Category

I²C

Reference

R_IIC_Create

Remarks

- The flags are not modified by this function. The event detection flags are cleared when a new transfer is started.

Program example

```
/* RPDL definitions */
#include “r_pdl_iic.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    uint16_t status_flags;
    uint16_t tx_count;

    /* Read the status of channel 0 */
    R_IIC_GetStatus(
        0,
        &status_flags,
        tx_count,
        PDL_NO_PTR
    );
}
```

4.2.18. 10-bit Analog to Digital Converter

1) R_ADC_10_Create

Synopsis

Configure an ADC unit.

Prototype

```
bool R_ADC_10_Create(
    uint8_t data1,      // ADC unit selection
    uint32_t data2,     // ADC configuration
    uint32_t data3,     // ADC conversion clock frequency
    float data4,        // ADC input sampling time
    void * func,        // Callback function
    uint8_t data5       // Interrupt priority level
);
```

Description (1/2)

Set the ADC's mode and operating condition.

[data1]
Select the ADC unit (0, 1, 2 or 3) to be configured.

[data2]
Conversion options. To set multiple options at the same time, use "|" to separate each value. The default settings are shown in **bold**. Specify PDL_NO_DATA to use the defaults.

- Input channel selection

PDL_ADC_10_CHANNELS_OPTION_1 or	Any mode: For unit 0, channel AN0. For unit 1, channel AN4. For unit 2, channel AN8. For unit 3, channel AN12.
PDL_ADC_10_CHANNELS_OPTION_2 or	Single mode: For unit 0, channel AN1. For unit 1, channel AN5. For unit 2, channel AN9. For unit 3, channel AN13. Scan mode: For unit 0, channels AN0 and AN1. For unit 1, channels AN4 and AN5. For unit 2, channels AN8 and AN9. For unit 3, channels AN12 and AN13.
PDL_ADC_10_CHANNELS_OPTION_3 or	Single mode: For unit 0, channel AN2. For unit 1, channel AN6. For unit 2, channel AN10. For unit 3, channel AN14. Scan mode: For unit 0, channels AN0 to AN2. For unit 1, channels AN4 to AN6. For unit 2, channels AN8 to AN10. For unit 3, channels AN12 to AN14.
PDL_ADC_10_CHANNELS_OPTION_4	Single mode: For unit 0, channel AN3. For unit 1, channel AN7. For unit 2, channel AN11. For unit 3, channel AN15. Scan mode: For unit 0, channels AN0 to AN3. For unit 1, channels AN4 to AN7. For unit 2, channels AN8 to AN11. For unit 3, channels AN12 to AN15.

Description (2/2)

- Scan mode

PDL_ADC_10_MODE_SINGLE or PDL_ADC_10_MODE_CONTINUOUS_SCAN or PDL_ADC_10_MODE_ONE_CYCLE_SCAN	Select single mode, continuous scan mode or one-cycle scan mode.

- Trigger selection

PDL_ADC_10_TRIGGER_SOFTWARE or	Software
PDL_ADC_10_TRIGGER_TPU_UNIT0 or	Timer TPU channels 0 to 5
PDL_ADC_10_TRIGGER_TPU_UNIT1 or	Timer TPU channels 6 to 11
PDL_ADC_10_TRIGGER_TMR or	Units 0 and 1: compare match A from TMR channel 0. Units 2 and 3: compare match A from TMR channel 2.
PDL_ADC_10_TRIGGER_ADTRG0 or PDL_ADC_10_TRIGGER_ADTRG1 or PDL_ADC_10_TRIGGER_ADTRG2 or PDL_ADC_10_TRIGGER_ADTRG3 or	ADTRG0 input pin (valid for units 0 or 1). ADTRG1 input pin (valid for unit 1 only). ADTRG2 input pin (valid for units 2 or 3). ADTRG3 input pin (valid for unit 3 only).
PDL_ADC_10_TRIGGER_TPU0_CM	A signal from TPU channel 0: Unit 0: compare match / input capture A. Unit 1: compare match / input capture B. Unit 2: compare match / input capture C. Unit 3: compare match / input capture D.

- Data alignment selection

PDL_ADC_10_DATA_ALIGNMENT_LEFT or **PDL_ADC_10_DATA_ALIGNMENT_RIGHT**	The alignment of the 10-bit ADC conversion result within the 16-bit storage area. Left: padded at the MSB end. Right: aligned to the LSB end.

- DMAC / DTC trigger control

PDL_ADC_10_DMAC_DTC_TRIGGER_DISABLE or PDL_ADC_10_DMAC_TRIGGER_ENABLE or PDL_ADC_10_DTC_TRIGGER_ENABLE	Disable or enable activation of the DMAC or DTC when a conversion or scan cycle completes.

- Sampling time calculation

PDL_ADC_10_ADSSTR_CALCULATE or PDL_ADC_10_ADSSTR_SPECIFY	Select whether parameter data4 is used to calculate the ADSSTR value, or contains the value to be stored in register ADSSTR.

[data3]
The desired frequency of the conversion clock (ADCLK) in Hertz.

[data4]
The data to be used for the sampling state register value calculations.

Data use	Parameter type
The timer period in seconds or	float
The value to be put in register ADSSTR	uint8_t

[func]
The function to be called when the ADC conversion or scan cycle is complete.
Specify PDL_NO_FUNC if no callback function is required.

[data5]
The interrupt priority level. Select between 1 (lowest priority) and 7 (highest priority).
This parameter will be ignored if PDL_NO_FUNC is specified for parameter func.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

ADC

References

R_ADC_10_Destroy, R_ADC_10_Read

Remarks

- This function configures the selected pin(s) for ADC operation by setting the direction to input and turning off the input buffer.
  The port control settings for any ADC pins that subsequently become inactive are not modified.
- This function brings the selected converter unit out of the power-down state.
- Interrupts are enabled automatically if a callback function is specified.
  Please see the notes on callback function usage in §6.
- A callback function is executed by the interrupt processing function. This means that no other interrupt can be processed until the callback function has completed.
- Function R_CGC_Set must be called before any use of this function.
- The available values for the conversion clock are PCLK ÷ 8, 4, 2 or 1. If the desired frequency is not an exact match, the actual frequency will be the next highest frequency.
  The timing limits depend on the peripheral module clock, PCLK.

Parameter	Limit	Equation	f_{PCLK} (MHz)			
			50	**12.5**	**32**	**8**
Conversion clock (ADCLK)	Minimum	(f_{PCLK} ÷ 8, 4, 2, 1) ≥ 4.0	6.25 MHz	6.25 MHz	4.0 MHz	4.0 MHz
	Maximum	f_{PCLK}	50.00 MHz	12.50 MHz	32.00 MHz	8.00 MHz
Conversion time	Maximum	25 ÷ ADCLK	4.0µs	4.0µs	6.25µs	6.25µs
	Minimum		0.5µs	2µs	0.781µs	3.13µs
Sampling time	Minimum	-	0.5µs			
	Maximum	255 ÷ ADCLK	e.g. 5.1µs at 50 MHz			
Total conversion time	Minimum	Conversion time + sampling time	1.0µs	2.5µs	1.28µs	3.63µs

Program example

```
/* RPDL definitions */
#include “r_pdl_adc_10.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* ADC unit 1 callback function */
void ADC1IntFunc(void){}

void func(void)
{
    /* Set up ADC 1 at 50 MHz in single mode using AN1 with 0.6µs sampling
time */
    R_ADC_10_Create(
         1,
         PDL_ADC_10_CHANNELS_OPTION_2,
         50E6,
         0.6E-6,
         ADC1IntFunc,
         2
    );

    /* Set up ADC 1 at 50 MHz in single mode using AN1 */
    R_ADC_10_Create(
         1,
         PDL_ADC_10_CHANNELS_OPTION_2 | PDL_ADC_10_ADSSTR_SPECIFY,
         50E6,
         0x40,
         ADC1IntFunc,
         2
    );
}
```

2) R_ADC_10_Destroy

Synopsis

Shut down an ADC unit.

Prototype

```
bool R_ADC_10_Destroy(
    uint8_t data    // ADC unit selection
);
```

Description

Put the ADC into the Power-down state, with minimal power consumption.

[data]
Select the ADC unit (0, 1, 2 or 3) to be shut down.

Return value

True if a valid unit is selected; otherwise false.

Category

ADC

Reference

R_ADC_10_Create

Remarks

- This function waits for the ADST flag to indicate that the converter has stopped. If the ADC unit's control registers are directly modified by the user, this function may lock up.

Program example

```
/* RPDL definitions */
#include “r_pdl_adc_10.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func( void )
{
    /* Shut down ADC unit 1 */
    R_ADC_10_Destroy(
        1
    );
}
```

3) R_ADC_10_Control

Synopsis

Start or stop an ADC unit.

Prototype

```
bool R_ADC_10_Control(
    uint16_t data     // Conversion unit control
);
```

Description

Controls start / stop operation of the specified ADC.

[data]
To select multiple units at the same time, use "|" to separate each value.

- On / off control

PDL_ADC_10_0_ON or PDL_ADC_10_0_OFF	Start or stop ADC unit 0 conversion.
PDL_ADC_10_1_ON or PDL_ADC_10_1_OFF	Start or stop ADC unit 1 conversion.
PDL_ADC_10_2_ON or PDL_ADC_10_2_OFF	Start or stop ADC unit 2 conversion.
PDL_ADC_10_3_ON or PDL_ADC_10_3_OFF	Start or stop ADC unit 3 conversion.

- Control the CPU during the ADC conversion. The default setting is shown in **bold**.

PDL_ADC_10_CPU_ON or	Allow the CPU to run normally during the conversion.
PDL_ADC_10_CPU_OFF	Stop the CPU when the conversion starts. The CPU will re-start when any valid interrupt occurs.

Return value

True if all parameters are valid and exclusive; otherwise false.

Category

ADC

Reference

R_ADC_10_Create, R_ADC_10_Read

Remarks

- Use this API function only when the software trigger option is selected.
- For single or one-cycle scan modes, the ADC will stop automatically when the conversion is complete.
- The time delay between starting conversions on multiple units is minimised, but has to use separate instructions. This function minimises the delay between starts by using an interrupt to prevent other interrupts from occurring during the start sequence. If the user has disabled interrupts (cleared the 'I' bit in the PSW register) in their own code, this function will lock up. For true simultaneous starting of ADC units, select an appropriate hardware trigger e.g. timer TMR.

Program example

```
/* RPDL definitions */
#include "r_pdl_adc_10.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Stop ADC unit 0 and start ADC unit 1 */
    R_ADC_10_Control(
         PDL_ADC_10_0_OFF | PDL_ADC_10_1_ON
    );
}
```

4) R_ADC_10_Read

Synopsis

Read the ADC conversion results.

Prototype

```
bool R_ADC_10_Read(
    uint8_t data1,      // ADC unit selection
    uint16_t * data2    // Pointer to the buffer where the converted values are to be stored
);
```

Description

Reads the conversion values for an ADC unit.

[data1]
Select the ADC unit (0, 1, 2 or 3) to be read.

[data2]
Specify a pointer to a variable or array where the results shall be stored.

Return value

True if a valid unit is selected; otherwise false.

Category

ADC

Reference

R_ADC_10_Create, R_ADC_10_Control

Remarks

- Between 1 and 4 conversion results will be read and stored. The number depends on the settings for “Input channel selection” and “Scan mode” when R_ADC_10_Create is used to configure the ADC unit.
- The 10-bit data alignment is controlled using the R_ADC_10_Create function.
- Ensure that the buffer is big enough for the requested number of values.
- If no callback function is used, this function waits for the ADI flag to indicate that conversion is complete before reading the results. If the ADC unit’s control registers are directly modified by the user, this function may lock up.

Program example

```
/* RPDL definitions */
#include “r_pdl_adc_10.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    uint16_t ADCresult[2];

    /* Read the ADC values for unit 2 */
    R_ADC_10_Read(
        2,
        ADCresult
    );
}
```

4.2.19. 10-bit Digital to Analog Converter

1) R_DAC_10_Create

Synopsis

Configure the 10-bit DAC module.

Prototype

```
bool R_DAC_10_Create(
    uint8_t data1,     // Configuration
    uint16_t data2,    // Output value
    uint16_t data3     // Output value
);
```

Description

Enable the DAC module and set the operating conditions.

[data1]
Configuration options. To set multiple options at the same time, use "|" to separate each value. The default settings are shown in **bold**.

- Channel enable

PDL_DAC_10_CHANNEL_0	Enable channel 0
PDL_DAC_10_CHANNEL_1	Enable channel 1

- Data alignment selection

PDL_DAC_10_ALIGN_LEFT or **PDL_DAC_10_ALIGN_RIGHT**	The alignment of the 10-bit output data within the 16-bit parameters data2 and data3. Left: padded at the MSB end. Right: aligned to the LSB end.

[data2]
The value to be written to the channel 0 output register. Ignored if the channel is not enabled.

[data3]
The value to be written to the channel 1 output register. Ignored if the channel is not enabled.

Return value

True if all parameters are valid and exclusive; otherwise false.

Functionality

DAC

References

R_DAC_10_Destroy, R_DAC_10_Write

Remarks

- This function configures the relevant pin for DAC operation.
  The port control settings for any DAC pins that subsequently become inactive are not modified.
- This function brings the converter module out of the power-down state.

Program example

```
/* RPDL definitions */
#include "r_pdl_dac_10.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func(void)
{
    /* Set up DAC channel 0 with default operation, mid voltage */
    R_DAC_10_Create(
         PDL_DAC_10_CHANNEL_0,
         1024 / 2,
         0
    );
}
```

2) R_DAC_10_Destroy

Synopsis

Disable a DAC channel.

Prototype

```
bool R_DAC_10_Destroy(
     uint8_t data     // Channel selection
);
```

Description

Disable the channel output.

[data1]

Disable selection. To set multiple options at the same time, use "|" to separate each value.

PDL_DAC_10_CHANNEL_0	Disable channel 0
PDL_DAC_10_CHANNEL_1	Disable channel 1

Return value

True if the parameter is valid; otherwise false.

Category

DAC

Reference

R_DAC_10_Create

Remarks

- Once both channels are disabled, the module is put into the power-down state.

Program example

```
/* RPDL definitions */
#include "r_pdl_dac_10.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

void func( void )
{
    /* Shut down both DAC channels */
    R_DAC_10_Destroy(
         PDL_DAC_10_CHANNEL_0 | PDL_DAC_10_CHANNEL_1
    );
}
```

3) R_DAC_10_Write

Synopsis

Write data to a DAC channel.

Prototype

```
bool R_DAC_10_Control(
    uint8_t data1,     // Channel selection
    uint16_t data2,    // Output value
    uint16_t data3     // Output value
);
```

Description

Write data to the selected DAC channel(s).

[data1]
Select the DAC channel output to be modified.

PDL_DAC_10_CHANNEL_0	Select channel 0
PDL_DAC_10_CHANNEL_1	Select channel 1

[data2]
The value to be written to the channel 0 output register. Ignored if the channel is not selected.

[data3]
The value to be written to the channel 1 output register. Ignored if the channel is not selected.

Return value

True if all parameters are valid; otherwise false.

Category

DAC

Reference

R_DAC_10_Create

Remarks

- Refer to the data alignment that was selected when R_DAC_10_Create was called.

Program example

```
/* RPDL definitions */
#include “r_pdl_dac_10.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void func(void)
{
    /* Write new data to DAC channel 0 */
    R_DAC_10_Write(
         PDL_DAC_10_CHANNEL_0,
         100,
         0
    );
}
```

5. Usage Examples

This chapter shows programming examples for each driver in this library.

5.1. Interrupt control

Figure 5-1 shows an example of external interrupt use.

Pin IRQ0-A is used to detect a falling edge and generates an interrupt.
Pin IRQ1-B is used to detect a falling edge and is polled.
Pin IRQ2-A is used to detect a low-level signal and generates an interrupt. Further interrupts are prevented until the signal has returned to the high level.

```
/* Peripheral driver function prototypes */
#include “r_pdl_intc.h”
#include “r_pdl_io_port.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

volatile uint8_t switch_sw1_pressed;
volatile uint8_t irq2_low;

/* Callback function prototypes */
void IRQ0Handler(void);
void IRQ0Handler(void);
static void ReEnableIRQ2(void);

void main(void)
{
    uint8_t irq_status;

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Configure the IRQ0 interrupt on pin IRQ0-A */
    R_INTC_CreateExtInterrupt(
        PDL_INTC_IRQ0,
        PDL_INTC_FALLING | PDL_INTC_A, 7,
        IRQ0Handler
    );

    /* Configure the IRQ1 interrupt on pin IRQ1-B */
    R_INTC_CreateExtInterrupt(
        PDL_INTC_IRQ1,
        PDL_INTC_FALLING | PDL_INTC_B,
        0,
        PDL_NO_FUNC
    );

    /* Configure the IRQ2 interrupt on pin IRQ2-A */
    R_INTC_CreateExtInterrupt(
        PDL_INTC_IRQ2,
        PDL_INTC_LOW | PDL_INTC_A,
        7,
        IRQ2Handler
    );

    irq2_low = false;

    while(1)
    {
    /* Poll the IRQ1 flag */
        R_INTC_GetExtInterruptStatus(
            PDL_INTC_IRQ1,
            &irq_status
```

```
            );
            if ((irq_status & 0x01) == 0)
            {
                /* Disable IRQ1 */
                R_INTC_ControlExtInterrupt(
                    PDL_INTC_IRQ1,
                    PDL_INT_DISABLE
                );
            }

            /* Has IRQ2 triggered? */
            if (irq2_low == true)
            {
                /* Re-enable the interrupt if the signal has returned to the high level */
                R_IO_PORT_Compare(
                    PDL_IO_PORT_3_2,
                    1,
                    ReEnableIRQ2
                );
            }
        }
    }

    void IRQ0Handler(void)
    {
        /* Process the IRQ0 event here (the flag is cleared automatically */
        switch_sw1_pressed = true;
    }

    void IRQ2Handler(void)
    {
        /* Disable the level-triggered interrupt */
        R_INTC_ControlExtInterrupt(
            PDL_INTC_IRQ2,
            PDL_INTC_DISABLE
        );
        irq2_low = true;
    }

    static void ReEnableIRQ2(void)
    {
        /* Re-enable the interrupt (and try to clear it) */
        R_INTC_ControlExtInterrupt(
            PDL_INTC_IRQ2,
            PDL_INTC_ENABLE | PDL_INTC_CLEAR_IR_FLAG
        );
    }
```

Figure 5-1: Example of External Interrupt

5.2. I/O Port

Figure 5-2 shows examples of I/O port configuration, reading and writing.

```
/* Peripheral driver function prototypes */
#include “r_pdl_io_port.h”
#include “r_pdl_pfc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void main(void)
{
    uint8_t result;

    /* Configure port 4 as an input */
    R_IO_PORT_Set(
        PDL_IO_PORT_4,
        PDL_IO_PORT_INPUT,
        PDL_IO_PORT_INPUT_BUFFER_ON,
        0,
        0
    );

    /* Configure port pin P21 as an open-drain output */
    R_IO_PORT_Set(
        PDL_IO_PORT_2_1,
        PDL_IO_PORT_OUTPUT,
        0,
        0,
        PDL_IO_PORT_OPEN_DRAIN
    );

    /* Write 0x44 to register PFC2 */
    R_PFC_Write(
        2,
        0x44
    );

    /* Read the value of all the pins on port 4 */
    R_IO_PORT_Read(
        PDL_IO_PORT_4,
        &result
    );

    /* Set pin P21 to output high */
    R_IO_PORT_Write(
        PDL_IO_PORT_2_1,
        1
    );

    /* Invert pin P21 */
    R_IO_PORT_Modify(
        PDL_IO_PORT_2_1,
        PDL_IO_PORT_XOR,
        1
    );

    /* And the value on port 4 with 55h */
    R_IO_PORT_Modify(
        PDL_IO_PORT_4,
        PDL_IO_PORT_AND,
        0x55
    );

    /* Read the control registers for port PC */
    R_IO_PORT_ReadControl(
        PDL_IO_PORT_1,
```

```
            &direction,
            &buffer,
            &pull_up,
            &output
        );

        /* Read the direction for pin P03 */
        R_IO_PORT_ReadControl(
            PDL_IO_PORT_0_3,
            &direction,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_PTR
        );

        /* Set the lower 4 bits on port P1 to output */
        R_IO_PORT_ModifyControl(
            PDL_IO_PORT_1,
            PDL_IO_PORT_DIRECTION | PDL_IO_PORT_OR,
            0x0F
        );

        /* Enable the pull-up on pin PA3 */
        R_IO_PORT_ModifyControl(
            PDL_IO_PORT_A_3,
            PDL_IO_PORT_PULL_UP | PDL_IO_PORT_OR,
            1
        );
    }
```

Figure 5-2: Example of I/O Port Operations

5.3. Bus Controller

Figure 5-3 shows an example of external bus controller usage.

```
/* Peripheral driver function prototypes */
#include “r_pdl_bsc.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void BSC_error_handler(void)
{
    /* Clear the error signals */
    R_BSC_Control(
        PDL_BSC_ERROR_CLEAR
    );
}

void main(void)
{
    uint8_t  * cs1_location_8;
    uint16_t * cs7_location_16;

    cs1_location_8  = ( uint8_t *)0x07000000ul;
    cs7_location_16 = (uint16_t *)0x01000000ul;

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Configure the bus controller */
    R_BSC_Create(
        PDL_BSC_CS2_B,
        0,
        PDL_BSC_ERROR_ILLEGAL_ADDRESS_ENABLE | PDL_BSC_ERROR_TIME_OUT_ENABLE,
        BSC_error_handler,
        5
    );

    /* Configure area CS7 */
    R_BSC_CreateArea(
        7,
        PDL_BSC_WRITE_SINGLE,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        1,
        0,
        0,
        1,
        0
    );

    /* Configure area CS1 */
    R_BSC_CreateArea(
        1,
        PDL_BSC_WIDTH_8 | PDL_BSC_WRITE_SINGLE,
```

```
            0,
            0,
            0,
            0,
            0,
            0,
            0,
            0,
            1,
            0,
            0,
            1,
            0
        );

        /* Initialise the system clocks and enable the BCLK output */
        R_CGC_Set(
            12.5E6,
            100E6,
            50E6,
            25E6,
            PDL_CGC_BCLK_ENABLE
        );

        /* Write to external areas */
        *cs7_location_16 = 0xAA55;
        *cs1_location_8 = 0xAA;

        /* Disable area CS1 */
        R_BSC_Destroy(
            1
        );
    }
```

Figure 5-3: Example of Bus Controller use

5.4. DMA controller

The following examples show the use of triggers by software, IRQ pin edge detection and SCI transmission.

(1) Software and IRQ triggers

Channel 0 will copy the string “Renesas RX610” into the destination area when a falling edge occurs on pin IRQ3-B.
Channel 1 will copy the string “Hello, World” into the destination area as soon as it is enabled.

```
/* PDL functions and definitions */
#include “r_pdl_dmac.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Required for this example */
#include <string.h>

/* Callback function prototype */
void DMAC0_transfer_end_handler(void);

/* Data source and destination declarations */
const uint8_t source_string_1[]="Renesas RX610";
const uint8_t source_string_2[]="Hello, World";
volatile uint8_t destination_string_1[]=".....................";
volatile uint8_t destination_string_2[]=".....................";

void main(void)
{
    uint16_t StatusValue;
    uint32_t SourceAddr;
    uint32_t DestAddr;
    uint32_t ByteCount;

    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Enable control of LED0 */
    R_IO_PORT_Set(
        PDL_IO_PORT_3_6,
        PDL_IO_PORT_OUTPUT,
        0,
        0,
        0
    );

    /* Configure channel 0 */
    R_DMAC_Create(
        0,
        PDL_DMAC_CONSECUTIVE | PDL_DMAC_SOURCE_ADDRESS_PLUS | \
```

```
             PDL_DMAC_DESTINATION_ADDRESS_PLUS,
            PDL_DMAC_REQUEST_IRQ3,
            source_string_1,
            destination_string_1,
            strlen((char *)source_string_1),
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            DMAC0_transfer_end_handler,
            7
        );

        /* Configure channel 1 */
        R_DMAC_Create(
            1,
            PDL_DMAC_CONSECUTIVE | PDL_DMAC_SOURCE_ADDRESS_PLUS | \
             PDL_DMAC_DESTINATION_ADDRESS_PLUS,
            PDL_DMAC_REQUEST_SW,
            source_string_2,
            destination_string_2,
            strlen((char *)source_string_2),
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_FUNC,
            0
        );

        /* Enable the SW3 interrupt */
        R_INTC_CreateExtInterruptAll(
            PDL_INTC_IRQ3,
            PDL_INTC_FALLING | PDL_INTC_B | PDL_INTC_DMAC_TRIGGER_ENABLE,
            PDL_NO_FUNC,
            0
        );

        /* Enable channel 0 */
        R_DMAC_Control(
            PDL_DMAC_0,
            PDL_DMAC_ENABLE,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Enable and start channel 1 */
        R_DMAC_Control(
            PDL_DMAC_1,
            PDL_DMAC_ENABLE | PDL_DMAC_START,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Read the status for channel 0 */
        R_DMAC_GetStatus(
            0,
            &StatusValue,
            &SourceAddr,
            &DestAddr,
```

```
            &ByteCount
        );
    }

    void DMAC0_transfer_end_handler(void)
    {
        /* Invert the port pin */
        R_IO_PORT_Modify(
            PDL_IO_PORT_3_6,
            PDL_IO_PORT_XOR,
            1
        );

        /* Stop all channels */
        R_DMAC_Control(
            PDL_DMAC_ALL,
            PDL_DMAC_SUSPEND,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Stop the DMAC */
        R_DMAC_Destroy();
    }
```

Figure 5-4: Two examples of DMAC use

(2) SCI transmission trigger

DMAC Channel 3 will be used to transmit the string “Renesas RX610”.
Then the string “Hello, World” will be transmitted by polling.

```
    /* PDL functions and definitions */
    #include “r_pdl_dmac.h”
    #include “r_pdl_cgc.h”
    #include “r_pdl_intc.h”
    #include "r_pdl_sci.h"

    /* RPDL device-specific definitions */
    #include “r_pdl_definitions.h”

    /* Required for this example */
    #include <string.h>

    /* Callback function prototype */
    void DMAC3_transfer_end_handler(void);

    /* Data source and destination declarations */
    const uint8_t source_string_1[]="Renesas RX610";
    const uint8_t source_string_2[]="Hello, World";

    /* Global flags */
    volatile uint8_t sci_dma_transfer_complete;
    volatile uint8_t break_required;

    void main(void)
    {
        /* Initialise the system clocks */
        R_CGC_Set(
            12.5E6,
            100E6,
```

```
            50E6,
            PDL_NO_DATA,
            PDL_CGC_BCLK_DISABLE
        );

        /* Set the CPU's Interrupt Priority Level to 0 */
        R_INTC_Write(
            PDL_INTC_REG_IPL,
            PDL_NO_DATA,
            0
        );

        /* Configure the RS232 port */
        R_SCI_Create(
            1,
            PDL_SCI_RX_DISCONNECTED | PDL_SCI_8N1,
            115200,
            0
        );

        /* Configure channel 3 */
        R_DMAC_Create(
            3,
            PDL_DMAC_SINGLE | PDL_DMAC_SOURCE_ADDRESS_PLUS,
            PDL_DMAC_REQUEST_SCI1_TX,
            source_string_1,
            (uint8_t *)&SCI1.TDR,
            strlen(source_string_1),
            PDL_NO_DATA,
            PDL_NO_DATA,
            PDL_NO_DATA,
            DMAC3_transfer_end_handler,
            7
        );

        /* Enable channel 3 */
        R_DMAC_Control(
            PDL_DMAC_3,
            PDL_DMAC_ENABLE,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Initialise the flags */
        sci_dma_transfer_complete = false;
        break_required = false;

        /* Enable the transmission using the DMAC */
        R_SCI_Send(
            1,
            PDL_SCI_DMAC_TRIGGER_ENABLE,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_FUNC
        );

        /* Wait for the DMAC to complete the transfer */
        while (sci_dma_transfer_complete == false);

        /* Send the next string using polling mode */
        R_SCI_Send(
            1,
            PDL_NO_DATA,
```

```
            source_string_2,
            0,
            PDL_NO_FUNC
        );
    }

    void DMAC3_transfer_end_handler(void)
    {
        uint8_t SCI_status;

        /* Wait for the SCI transmission to end */
        do
        {
            R_SCI_GetStatus(
                1,
                &SCI_status,
                PDL_NO_PTR,
                PDL_NO_PTR,
                PDL_NO_PTR
            );
        } while ((SCI_status & 0x04) == 0);

        if (break_required == true)
        {
            /* Stop the SCI to allow the break signal to be output */
            R_SCI_Control(
                1,
                PDL_SCI_STOP_TX | PDL_SCI_OUTPUT_SPACE
            );
        }
        else
        {
            /* Stop the SCI */
            R_SCI_Control(
                1,
                PDL_SCI_STOP_TX | PDL_SCI_OUTPUT_MARK
            );
        }

        sci_dma_transfer_complete = true;
    }
```

Figure 5-5: An example of using the DMAC for serial port transmission

(3) SCI reception trigger

DMAC channel 2 will transfer 5 received characters into the assigned storage area and then call the callback function.

```
    /* PDL functions and definitions */
    #include “r_pdl_dmac.h”
    #include “r_pdl_cgc.h”
    #include “r_pdl_intc.h”
    #include "r_pdl_sci.h"

    /* RPDL device-specific definitions */
    #include “r_pdl_definitions.h”

    /* Callback function prototype */
    void DMAC2_transfer_end_handler(void);

    /* Data destination area */
    volatile uint8_t destination_string_1[]=".....................";

    void main(void)
    {
```

```
        /* Initialise the system clocks */
        R_CGC_Set(
            12.5E6,
            100E6,
            50E6,
            PDL_NO_DATA,
            PDL_CGC_BCLK_DISABLE
        );

        /* Set the CPU's Interrupt Priority Level to 0 */
        R_INTC_Write(
            PDL_INTC_REG_IPL,
            PDL_NO_DATA,
            0
        );

        /* Configure the RS232 port */
        R_SCI_Create(
            1,
            PDL_SCI_TX_DISCONNECTED | PDL_SCI_8N1,
            115200,
            0
        );

        /* Configure channel 2 */
        R_DMAC_Create(
            2,
            PDL_DMAC_SINGLE | PDL_DMAC_DESTINATION_ADDRESS_PLUS,
            PDL_DMAC_REQUEST_SCI1_RX,
            (uint8_t *)&SCI1.RDR,
            destination_string_1,
            5,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            DMAC2_transfer_end_handler,
            7
        );

        /* Enable channel 2 */
        R_DMAC_Control(
            PDL_DMAC_2,
            PDL_DMAC_ENABLE,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Initiate reception, triggering the DMAC when data is received */
        R_SCI_Receive(
            1,
            PDL_SCI_DMAC_TRIGGER_ENABLE,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_FUNC,
            PDL_NO_FUNC
        );
    }

    void DMAC2_transfer_end_handler(void)
    {
        /* Disable channel 2 */
        R_DMAC_Control(
            PDL_DMAC_2,
```

```
            PDL_DMAC_SUSPEND,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );
    }
```

Figure 5-6: An example of using the DMAC for serial port reception

5.5. Data Transfer Controller

Figure 5-8 shows an example of Data Transfer Controller usage.

```
/* Peripheral driver function prototypes */
#include “r_pdl_dtc.h”
#include “r_pdl_cgc.h”
#include “r_pdl_io_port.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Required for this example */
#include <string.h>

/* Reserve an area for the DTC vector table */
#pragma address dtc_vector_table = 0x00015000
uint32_t dtc_vector_table[256];

/* Reserve 16 bytes for the IRQ3-triggered transfer data area */
uint32_t dtc_irq3_transfer_data[4];

void main(void)
{
    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Enable control of LED0 */
    R_IO_PORT_Set(
        PDL_IO_PORT_3_6,
        PDL_IO_PORT_OUTPUT,
        0,
        0,
        0
    );

    /* Set the DTC options */
    R_DTC_Set(
        PDL_NO_DATA,
        dtc_vector_table
    );

    /* Configure the DTC for IRQ3 */
    if (R_DTC_Create(
        PDL_DTC_BLOCK | PDL_DTC_DESTINATION | \
        PDL_DTC_SOURCE_ADDRESS_PLUS | PDL_DTC_DESTINATION_ADDRESS_PLUS | \
        PDL_DTC_SIZE_8 | \
        PDL_DTC_IRQ_COMPLETE | \
        PDL_DTC_TRIGGER_IRQ3,
        dtc_irq3_transfer_data,
        source_string_1,
        destination_string_1,
        1,
```

```
            (uint8_t)(strlen((char *)source_string_1))
        )==false)
        {
            while(1);
        }


        /* Enable the SW3 interrupt */
        R_INTC_CreateExtInterruptAll(
            PDL_INTC_IRQ3,
            PDL_INTC_FALLING | PDL_INTC_B | PDL_INTC_DTC_TRIGGER_ENABLE,
            IRQ3_handler,
            7
        );

        /* Start the DTC */
        R_DTC_Control(
            PDL_DTC_START,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_DATA
        );
    }

    void IRQ3_handler(void)
    {
        uint8_t StatusValue;
        uint32_t SourceAddr;
        uint32_t DestAddr;
        uint32_t TransferCount;

        /* Read the status and current source address for the IRQ3 transfer */
        R_DTC_GetStatus(
            dtc_irq3_transfer_data,
            &StatusValue,
            &SourceAddr,
            &DestAddr,
            &TransferCount
        );

        /* Normal transfer? */
        if (StatusValue == 0x00)
        {
            /* Invert the port pin */
            R_IO_PORT_Modify(
                PDL_IO_PORT_3_6,
                PDL_IO_PORT_XOR,
                1
            );

            /* Re-enable IRQ3 as a DTC trigger */
            R_DTC_Control(
                PDL_DTC_TRIGGER_IRQ3,
                PDL_NO_PTR,
                PDL_NO_PTR,
                PDL_NO_PTR,
                PDL_NO_DATA,
                PDL_NO_DATA
            );
        }
    }
```

Figure 5-7: Example of DTC use

5.6. Timer Pulse Unit

Figure 5-8 shows an example of Timer Pulse Unit usage.

```
/* Peripheral driver function prototypes */
#include “r_pdl_tpu.h”
#include “r_pdl_cgc.h”
#include “r_pdl_pfc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void main(void)
{
    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Select the –B pins for TLCKA, TCLKB, TCLKC and TCLKD by setting bit TCLKS in register
PFCR5 to 1 */
    R_PFC_Modify(
        5,
        PDL_PFC_OR,
        0x08
    );

    /* Configure channel 0 for dual-waveform (A and B) output */
    R_TPU_Create(
        0,
        0,
        PDL_TPU_CLK_PCLK_DIV_1 | PDL_TPU_CLEAR_CM_B,
        PDL_TPU_A_OC_LOW_CM_INV | PDL_TPU_B_OC_HIGH_CM_INV,
        0,
        0,
        200 - 1,
        400 - 1,
        0,
        0,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );

    /* Read the status flags and resisters A and D for channel 0 */
    R_TPU_Read(
        0,
        &Flags,
        PDL_NO_PTR,
        &General_A,
```

```
            PDL_NO_PTR,
            PDL_NO_PTR,
            &General_D
        );

        /* Modify channel 0 */
        R_TPU_Control(
            0,
            PDL_TPU_COUNTER,
            0xFFDD,
            PDL_NO_DATA,
            PDL_NO_DATA,
            PDL_NO_DATA,
            PDL_NO_DATA
        );

        /* Shutdown channels 0 to 5 */
        R_TPU_Destroy(
            0
        );
    }
```

Figure 5-8: Example of Timer Pulse Unit use

The counter is reset when it reaches 399. The 0 value is a valid state so the output toggle frequency is 50 MHz ÷ 400.

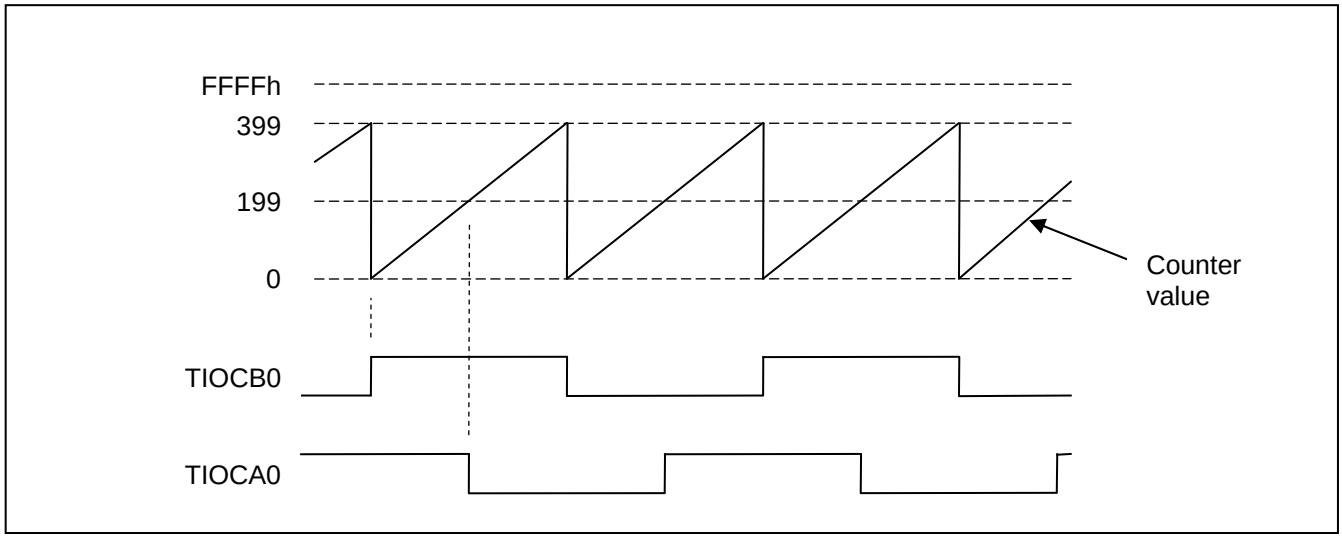


Figure 5-9: Example of TPU operation

5.7. Compare Match Timer

Figure 5-10 shows an example of Compare Match Timer usage. One channel is used to generate interrupts at regular intervals.

```
/* Peripheral driver function prototypes */
#include “r_pdl_cmt.h”
#include “r_pdl_cgc.h”
#include “r_pdl_io_port.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Callback function prototype */
void CMT0_handler(void);

void main(void)
{
    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Configure a port pin for output */
    R_IO_PORT_Set(
        PDL_IO_PORT_3_6,
        PDL_IO_PORT_OUTPUT,
        0,
        0,
        0
    );

    /* Configure CMT channel 0 for 1kHz operation */
    R_CMT_CreatePeriodic(
        0,
        PDL_CMT_FREQUENCY,
        1E3,
        CMT0_handler,
        7
    );

    /* Change the frequency to 10kHz */
    R_CMT_Control(
        0,
        PDL_CMT_FREQUENCY,
        10E3
    );
}
void CMT0_handler(void)
{
    /* Invert the port pin */
    R_IO_PORT_Modify(
        PDL_IO_PORT_3_6,
        PDL_IO_PORT_XOR,
```

```
            1
        );
    }
```

Figure 5-10: Example of Compare Match Timer use

5.8. 8-bit Timer

(1) Periodic operation

Timer channel 0 is configured to provide pulses on pin TMO0, with a pulse width of 500µs and an on-time of 200µs.

```
/* Peripheral driver function prototypes */
#include “r_pdl_tmr.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void main(void)
{
    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Configure TMR0 for 500µs pulse width, 200µs on-time */
    R_TMR_CreatePeriodic(
        PDL_TMR_TMR0,
        PDL_TMR_PERIOD | PDL_TMR_OUTPUT_ON,
        500E-6,
        200E-6,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );

    /* The same operation, using frequency and duty cycle */
    R_TMR_CreatePeriodic(
        PDL_TMR_TMR0,
        PDL_TMR_FREQUENCY | PDL_TMR_OUTPUT_ON,
        2E6,
        40,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );
}
```

Figure 5-11: Example of Pulse Output code

For full flexibility, the R_TMR_CreateChannel() function can be used.
In this example, Timer channel 0 is configured to provide pulses on pin TMO0, with a pulse width of 200 ticks of PCLK and a duty cycle of 50%.
Note that the output transitions and counter clearing occur after the compare match has occurred. So the values for compare match A and compare match B should be 1 less than the required count.

```
/* Peripheral driver function prototypes */
#include “r_pdl_tmr.h”
#include “r_pdl_definitions.h”

void main(void)
{
    /* Configure TMR0 to clear on a compare match A, output 1 at a compare match A and output
0 at a compare match B */
    R_TMR_CreateChannel(
        0,
        PDL_TMR_CLK_PCLK_DIV_1 | PDL_TMR_CLEAR_CM_A | PDL_TMR_OUTPUT_HIGH_CM_A | \
         PDL_TMR_OUTPUT_LOW_CM_B,
        PDL_NO_DATA,
        (200 – 1),
        (200 / 2) – 1,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        PDL_NO_FUNC,
        0
    );
}
```

Figure 5-12: Example of Pulse Output code

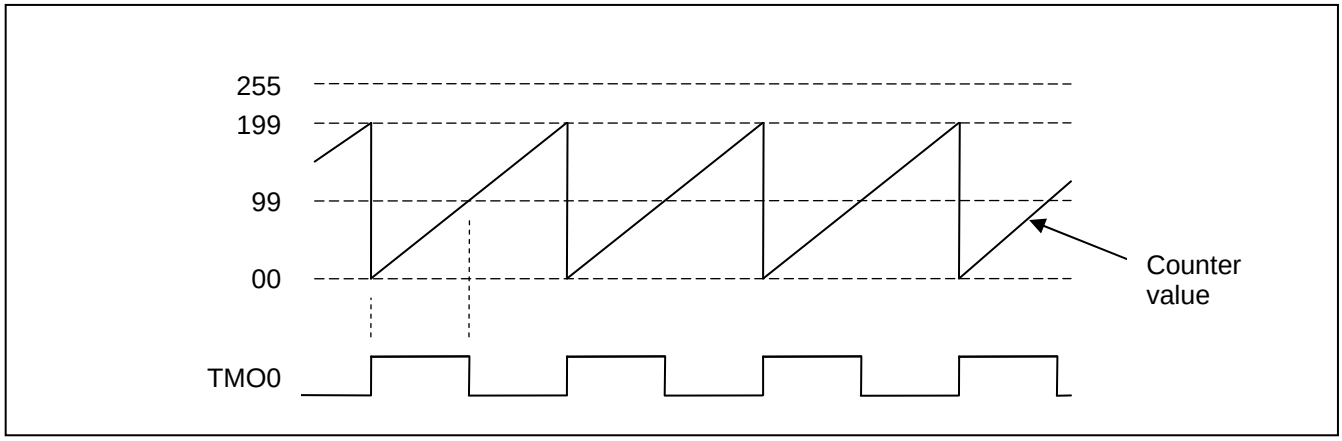


Figure 5-13: Example of pulse output operation

5.9. Serial Communication Interface

(1) Asynchronous Reception

Figure 5-14 shows the setting of SCI channels 0 and 1 and the reception of data using interrupts (channel 0) and polling (channel 1).

```
/* Peripheral driver function prototypes */
#include “r_pdl_sci.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Callback function prototypes */
void System_failed(void){};
void System_reset(void){};
void SCI0RxFunc(void);
void SCI0ErrFunc(uint8_t);

volatile char rx_string[10];

void main(void)
{
    uint8_t result;

    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Set up SCI channel 0: Async, 8N1, 38400 baud */
    R_SCI_Create(
        0,
        PDL_SCI_ASYNC | PDL_SCI_8N1,
        38400,
        1
    );

    /* Set up SCI channel 1: Async, 8N1, 19200 baud */
    R_SCI_Create(
        1,
        PDL_SCI_ASYNC | PDL_SCI_8N1,
        19200,
        0
    );

    /* Start the interrupt-based reception of 9 characters on channel 0 */
    R_SCI_Receive(
        0,
        PDL_NO_DATA,
        rx_string,
        9,
        SCI0RxFunc,
        SCI0ErrFunc
```

```
        );

        /* Wait for 1 character to be received on channel 1 */
        R_SCI_Receive(
            1,
            PDL_NO_DATA,
            &result,
            1,
            PDL_NO_FUNC,
            PDL_NO_FUNC
        );

        /* Check that channel 0 has completed */
        do
        {
            R_SCI_GetStatus(
                   0,
                   &result,
                   PDL_NO_PTR,
                   PDL_NO_PTR
            );
        } while ((result & 0x10) != 0);

        /* Shut down channel 0 */
        R_SCI_Destroy(
            0
        );
    }

    /* SCI channel 0 receive data handler */
    void SCI0RxFunc(void)
    {
        char * str_ptr = rx_string;
        while (*str_ptr-- != 0)
        {
            /* Process the string contents */
        }
    }

    /* SCI channel 0 error handler */
    void SCI0ErrFunc(void)
    {
        uint8_t error_flags;

        /* Read the status */
        R_SCI_GetStatus(
            0,
            &error_flags,
            PDL_NO_PTR,
            PDL_NO_PTR
        );

        /* Overrun error? */
        if ((error_flags & 0x20) != 0)
        {
            System_failed();
        }
    }
```

Figure 5-14: Example of Asynchronous Reception code

(2) Asynchronous Transmission

Figure 5-15 shows the configuration of SCI channels 0 and 1 and the transmission of data (on channel 0) using polling and then interrupts.

```
/* Peripheral driver function prototypes */
#include “r_pdl_sci.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Callback function prototype */
void SCI0TxFunc(void);

volatile char result[2];

void main(void)
{
    /* Put a null at the end */
    result[1] = 0;

    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Set up SCI channel 0: Async, 8N1, 38400 baud */
    R_SCI_Create(
        0,
        PDL_SCI_ASYNC | PDL_SCI_8N1,
        38400,
        0
    );

    /* Set up SCI channel 1: Async, 8N1, 19200 baud */
    R_SCI_Create(
        1,
        PDL_SCI_ASYNC | PDL_SCI_8N1,
        19200,
        0
    );

    /* Wait for 1 character to be received on channel 1 */
    R_SCI_Receive(
        1,
        PDL_NO_DATA,
        result,
        1,
        PDL_NO_FUNC,
        PDL_NO_FUNC
    );

    /* Send a string on channel 0 (wait for completion) */
    R_SCI_Send(
```

```
            0,
            PDL_NO_DATA,
            “Renesas RX”,
            0,
            PDL_NO_FUNC
        );

        /* Send another string on channel 0 */
        R_SCI_Send(
            0,
            PDL_NO_DATA,
            “www.renesas.com”,
            0,
            SCI0TxFunc
        );

        /* Echo the character on channel 1 */
        R_SCI_Send(
            1,
            PDL_NO_DATA,
            result,
            0,
            PDL_NO_FUNC
        );
    }

    /* SCI channel 0 transmit complete handler */
    void SCI0TxFunc(void)
    {
        /* Shut down channel 0 */
        R_SCI_Destroy(
            0
        );
    }
```

Figure 5-15: Example of Asynchronous Transmission code

(3) Synchronous Transmission and Reception

Figure 5-16 shows the configuration of SCI channel 3 as the clock master followed by the simultaneous transmission and reception of data.
The Receive function call uses interrupts while the Transmit function uses polling.

```
/* Peripheral driver function prototypes */
#include “r_pdl_sci.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Callback function prototype */
void SCI3RxFunc(void);

volatile uint8_t data_received;

#define BUFFER_SIZE 10

void main(void)
{
    uint8_t Tx_Data[BUFFER_SIZE];
    uint8_t Rx_Data[BUFFER_SIZE];
    uint8_t transfer_size = 8;
    uint8_t i;

    /* Configure the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        PDL_NO_DATA,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Set up SCI channel 3: Sync, MSb first, 2 Mbps */
    R_SCI_Create(
        3,
        PDL_SCI_SYNC | PDL_SCI_MSB_FIRST,
        2E6,
        1
    );

    /* Load the required data into the transmit buffer */
    for (i = 0; i < BUFFER_SIZE; i++)
    {
        Tx_Data[i] = (uint8_t)(i + 1);
    }

    data_received = false;

    /* Set up the receive process (no bus activity will occur) */
    R_SCI_Receive(
        3,
        PDL_NO_DATA,
        Rx_Data,
        transfer_size,
        SCI3RxFunc,
```

```
            PDL_NO_FUNC
        );

        /* Send data (which will also receive data at the same time) */
        R_SCI_Send(
            3,
            PDL_NO_DATA,
            Tx_Data,
            transfer_size,
            PDL_NO_FUNC
        );

        /* Ensure the receive interrupt has processed the last byte */
        while (data_received == false);

        /* Process the received data here */
    }

    /* SCI channel 3 receive complete handler */
    void SCI3RxFunc (void)
    {
        data_received = true;
    }
```

Figure 5-16: Example of Synchronous Transmission and Reception code

5.10. CRC calculator

Figure 5-17 shows an example of CRC usage.
The payload and CRC checksum have been received from a remote unit.
The CRC calculator is used to check that the payload is correct.

```
/* Peripheral driver function prototypes */
#include “r_pdl_crc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void main(void)
{
    uint16_t crc_result;

    /* Configure the CRC to use the CCITT polynomial; LSB first */
    R_CRC_Create(
        PDL_CRC_POLY_CRC_CCITT | PDL_CRC_LSB_FIRST
    );

    /* Write the payload data */
    R_CRC_Write(
        0xF0
    );

    /* Write the first half of the CRC checksum */
    R_CRC_Write(
        0x8F
    );

    /* Write the second half of the CRC checksum */
    R_CRC_Write(
        0xF7
    );

    /* Read the CRC calculation result */
    R_CRC_Read(
        PDL_NO_DATA,
        &Result
    );

    /* Shutdown the CRC unit */
    R_CRC_Destroy(
    );
}
```

Figure 5-17: Example of CRC calculation

5.11. I²C Bus Interface

In the following examples, the bus activity will be illustrated using the following format.

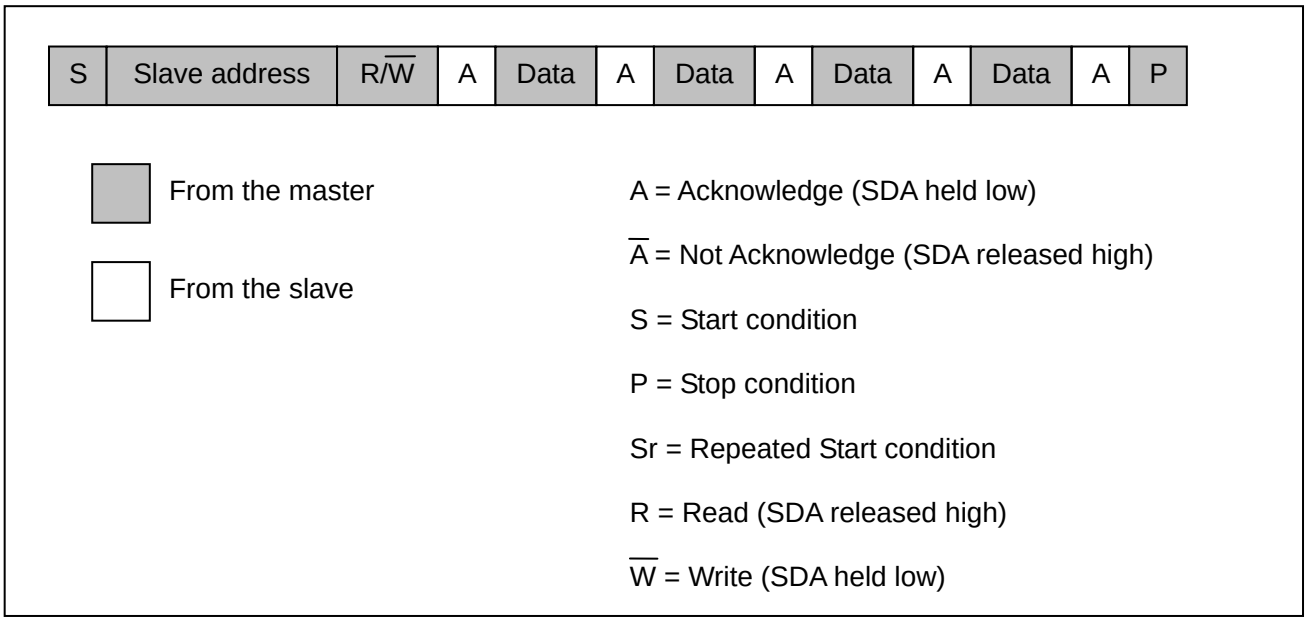


Figure 5-18: I²C bus activity notation

5.11.1. Master mode

In this example an EEPROM device has been connected to channel 1.

The EEPROM responds to the 7-bit slave address 1010xxxb.

During a read process the bits “xxx” can be any value.

During a write process:

i) The bits “xxx” represent the EEPROM memory address bits a10, a9 and a8.
ii) The first byte after the slave address is the EEPROM memory address bits a7 to a0.

The EEPROM has a write cycle time of 5 ms.

The following examples illustrate the use of Master mode.

1) Configuration and transmission

```
/* Peripheral driver function prototypes */
#include “r_pdl_iic.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”
#include “r_pdl_cmt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

#define EEPROM_ADDRESS 0xA0

void main(void)
{
    const uint8_t eeprom_data_array_1[5] = {0x00, 0x01, 0x02, 0x03, 0x04};
    uint16_t status_flags = 0;
    uint16_t TxChars;
    uint16_t RxChars;

    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        PDL_NO_DATA,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Select I²C mode at 100kHz, 300ns rise time, 200ns fall time */
    R_IIC_Create(
        1,
        PDL_IIC_MODE_IIC | PDL_IIC_INT_PCLK_DIV_8,
        PDL_NO_DATA,
        PDL_NO_DATA,
        PDL_NO_DATA,
        PDL_NO_DATA,
        100E3,
        (300 << 16) | 200
    );

    /* Send the lower address and 3 bytes to the EEPROM, using polling */
    if (R_IIC_MasterSend(
        1,
        PDL_NO_DATA,
        EEPROM_ADDRESS,
        eeprom_data_array_1,
        4,
        PDL_NO_FUNC,
        0
    ) == false)
    {
        /* Read the channel and transfer status */
        R_IIC_GetStatus(
            1,
            &status_flags,
            &TxChars,
            PDL_NO_PTR
        );
        /* Review the flags and transmit count to decide on the next action */
```

```
        }
        else
        {
            /* Wait for 5ms while the EEPROM updates */
            R_CMT_CreateOneShot(
                0,
                0,
                5E-3,
                PDL_NO_FUNC,
                0
            );
        }
```

Figure 5-19: Configure the I²C channel and write 3 data bytes to the first locations

2) Reception

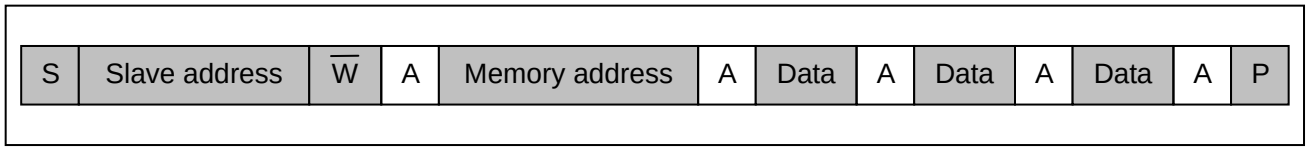


Figure 5-20: The bus activity, showing 4 bytes being transmitted to the EEPROM

```
/* Read data from the EEPROM, using polling */
if (R_IIC_MasterReceive(
    1,
    PDL_NO_DATA,
    EEPROM_ADDRESS,
    data_storage,
    4,
    PDL_NO_FUNC,
    0
) == false)
{
    /* Read the channel and transfer status */
    R_IIC_GetStatus(
        1,
        &status_flags,
        PDL_NO_PTR,
        &RxChars
    );
    /* Review the flags and transmit count to decide on the next action */
}
```

Figure 5-21: An example of reading data from the EEPROM

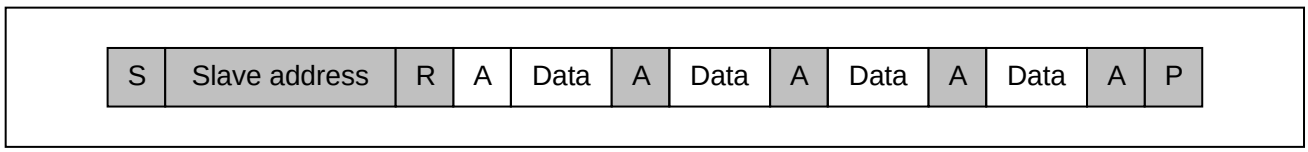


Figure 5-22: The bus activity, showing 4 bytes being transmitted by the EEPROM

3) Repeated Start

```
/* Send 1 byte to the EEPROM to update the lower address bits and do not stop */
R_IIC_MasterSend(
    1,
    PDL_IIC_STOP_DISABLE,
    EEPROM_ADDRESS,
    0x37,
    1,
    PDL_NO_FUNC,
    0
);

/* Read data from the EEPROM.  A repeated start will occur. */
R_IIC_MasterReceive(
    1,
    PDL_NO_DATA,
    EEPROM_ADDRESS,
    data_storage,
    2,
    PDL_NO_FUNC,
    0
);
```

Figure 5-23: Set the lower address to 37h and then read 2 bytes.

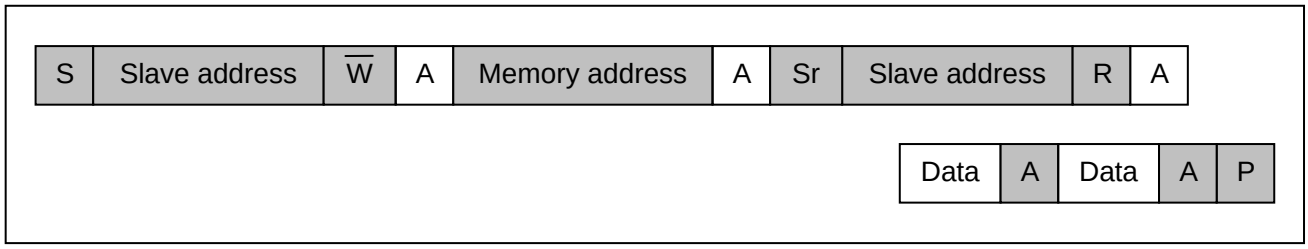


Figure 5-24: The bus activity, showing the Repeated Start condition when switching to the Read process

5.11.2. Master mode with DMAC

In the following example, data is written to an EEPROM in two bursts. DMAC channel 3 is used to handle the data transfer.
The same EEPROM address locations are then read out in two bursts. DMAC channel 2 is used to handle the data transfer

```
/* Peripheral driver function prototypes */
#include “r_pdl_iic.h”
#include “r_pdl_cgc.h”
#include “r_pdl_cmt.h”
#include “r_pdl_dmac.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

static void write_eeprom_data(void);
static void read_eeprom_data(void);
void iic_tx_dmac_end_handler(void);
void iic_rx_dmac_end_handler(void);

#define EEPROM_MEMORY_ADDRESS_UPPER 0x00
#define EEPROM_MEMORY_ADDRESS_LOWER 0x00
#define EEPROM_ADDRESS (0x00A0 | EEPROM_MEMORY_ADDRESS_UPPER)

volatile uint8_t bus_busy;
volatile uint8_t data_storage[20];

void main(void)
{
    const uint8_t eeprom_data_array_1[] = {EEPROM_MEMORY_ADDRESS_LOWER, 0x01, 0x02, 0x03,
0x04, 0x05};
    const uint8_t eeprom_data_array_2[] = {EEPROM_MEMORY_ADDRESS_LOWER + 5, 0x06, 0x07,
0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};
    uint8_t i;

    /* Configure the clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        PDL_NO_DATA,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set up a DMAC channel for IIC transmission */
    R_DMAC_Create(
        3,
        PDL_DMAC_SINGLE | PDL_DMAC_SOURCE_ADDRESS_PLUS,
        PDL_DMAC_REQUEST_IIC1_TX,
        eeprom_data_array_1,
        (uint8_t *)&RIIC1.ICDRT,
        6,
        PDL_NO_PTR,
        PDL_NO_PTR,
        PDL_NO_DATA,
        iic_tx_dmac_end_handler,
        7
    );

    /* Set up a DMAC channel for IIC reception */
    R_DMAC_Create(
        2,
        PDL_DMAC_SINGLE | PDL_DMAC_DESTINATION_ADDRESS_PLUS,
        PDL_DMAC_REQUEST_IIC1_RX,
        (uint8_t *)&RIIC1.ICDRR,
```

```
            data_storage,
            4,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            iic_rx_dmac_end_handler,
            7
        );

        /* Select I²C mode at 100kHz, 300ns rise time, 200ns fall time */
        R_IIC_Create(
            1,
            PDL_IIC_MODE_IIC | PDL_IIC_INT_PCLK_DIV_8,
            0,
            0,
            0,
            0,
            100E3,
            (300 << 16) | 200
        );

        /* Enable the DMAC channels */
        R_DMAC_Control(
            PDL_DMAC_2 | PDL_DMAC_3,
            PDL_DMAC_ENABLE,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Write the data into the EEPROM */
        write_eeprom_data();

        /* Prepare the next data for the EEPROM */
        R_DMAC_Control(
            PDL_DMAC_3,
            PDL_DMAC_SUSPEND | PDL_DMAC_ENABLE | \
            PDL_DMAC_UPDATE_SOURCE | PDL_DMAC_UPDATE_COUNT | PDL_DMAC_CLEAR_DREQ,
            PDL_NO_DATA,
            eeprom_data_array_2,
            PDL_NO_PTR,
            8,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Write the data into the EEPROM */
        write_eeprom_data();

        /* Clear the data storage area */
        for (i = 0; i < 20; i++) data_storage[i] = 0x00;

        /* Reset the EEPROM sub-address to 0, using polling */
        R_IIC_MasterSend(
            1,
            PDL_IIC_STOP_DISABLE,
            EEPROM_ADDRESS,
            eeprom_data_array_1,
            1,
            PDL_NO_FUNC,
            0
        );
```

```
        /* Read data from the EEPROM on channel 1, using the DMAC */
        read_eeprom_data();

        /* Prepare the next data */
        R_DMAC_Control(
            PDL_DMAC_2,
            PDL_DMAC_SUSPEND | PDL_DMAC_ENABLE | \
             PDL_DMAC_UPDATE_DESTINATION | PDL_DMAC_UPDATE_COUNT,
            PDL_NO_DATA,
            PDL_NO_PTR,
            &data_storage[5],
            5,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        /* Read data from the EEPROM on channel 1, using the DMAC */
        read_eeprom_data();
    }

    static void write_eeprom_data(void)
    {
        bus_busy = true;
        /* Send data to the EEPROM on channel 1, using the DMAC */
        R_IIC_MasterSend(
            1,
            PDL_IIC_DMAC_TRIGGER_ENABLE,
            EEPROM_ADDRESS,
            PDL_NO_PTR,
            0,
            PDL_NO_FUNC,
            0
        );
        while (bus_busy == true);

        /* Wait for 5ms while the EEPROM updates */
        R_CMT_CreateOneShot(
            0,
            0,
            5E-3,
            PDL_NO_FUNC,
            0
        );
    }

    static void read_eeprom_data(void)
    {
        bus_busy = true;
        /* Read data from the EEPROM on channel 1, using the DMAC */
        R_IIC_MasterReceive(
            1,
            PDL_IIC_DMAC_TRIGGER_ENABLE,
            EEPROM_ADDRESS,
            PDL_NO_PTR,
            0,
            PDL_NO_FUNC,
            0
        );
        while (bus_busy == true);
    }

    void iic_tx_dmac_end_handler(void)
    {
        uint16_t status_flags = 0;

        /* Wait for the transmission to complete */
        do
```

```
        {
            R_IIC_GetStatus(
                1,
                &status_flags,
                PDL_NO_PTR,
                PDL_NO_PTR
            );
        } while((status_flags & 0x0080u) == 0x0u);

        /* Issue a Stop condition on channel 1 */
        R_IIC_Control(
            1,
            PDL_IIC_STOP
        );

        bus_busy = false;
    }

    void iic_rx_dmac_end_handler(void)
    {
        uint32_t DestAddr = 0;

        /* Read the next destination address for the current transfer */
        R_DMAC_GetStatus(
            2,
            PDL_NO_PTR,
            PDL_NO_PTR,
            &DestAddr,
            PDL_NO_PTR
        );

        /* Read one more byte with NACK condition on channel 1 and stop */
        R_IIC_MasterReceiveLast(
            1,
            (uint8_t *)DestAddr
        );

        bus_busy = false;
    }
```

Figure 5-25: An example of write data to and reading data from an EEPROM, using two DMAC channels

5.11.3. Master mode with DTC

In the following example, data is written to an EEPROM in two bursts. The DTC is used to handle the data transfer.
The same EEPROM address locations are then read out in two bursts. The DTC is used to handle the data transfer

```
/* Peripheral driver function prototypes */
#include “r_pdl_iic.h”
#include “r_pdl_cgc.h”
#include “r_pdl_cmt.h”
#include “r_pdl_dtc.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

static void write_eeprom_data(void);
static void read_eeprom_data(void);
void iic_tx_dmac_end_handler(void);
void iic_rx_dmac_end_handler(void);

#define EEPROM_MEMORY_ADDRESS_UPPER 0x00
#define EEPROM_MEMORY_ADDRESS_LOWER 0x00
#define EEPROM_ADDRESS (0x00A0 | EEPROM_MEMORY_ADDRESS_UPPER)

volatile uint8_t bus_busy;
volatile uint8_t data_storage[20];

/* Reserve an area for the DTC vector table */
#pragma address dtc_vector_table = 0x00001000
uint32_t dtc_vector_table[256];

/* Reserve 16 bytes (full address mode) for the transfer data areas */
uint32_t dtc_iic1_tx_transfer_data[4];
uint32_t dtc_iic1_rx_transfer_data[4];

void main(void)
{
    const uint8_t eeprom_data_array_1[] = {EEPROM_MEMORY_ADDRESS_LOWER, 0x01, 0x02, 0x03,
0x04, 0x05};
    const uint8_t eeprom_data_array_2[] = {EEPROM_MEMORY_ADDRESS_LOWER + 5, 0x06, 0x07,
0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};
    uint8_t i;

    /* Configure the clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        PDL_NO_DATA,
        PDL_CGC_BCLK_DISABLE
    );

    /* Configure the DTC controller */
    R_DTC_Set(
        PDL_DTC_ADDRESS_FULL,
        dtc_vector_table
    );

    /* Set up a DTC channel for IIC transmission */
    R_DTC_Create(
        PDL_DTC_NORMAL | \
        PDL_DTC_SOURCE_ADDRESS_PLUS | PDL_DTC_DESTINATION_ADDRESS_FIXED | \
        PDL_DTC_SIZE_8 | \
        PDL_DTC_IRQ_COMPLETE | \
        PDL_DTC_TRIGGER_ICTXI1,
```

```
            dtc_iic1_tx_transfer_data,
            eeprom_data_array_1,
            (uint8_t *)&RIIC1.ICDRT,
            6,
            PDL_NO_DATA
        );

        /* Set up a DTC channel for IIC reception */
        R_DTC_Create(
            PDL_DTC_NORMAL | \
            PDL_DTC_SOURCE_ADDRESS_FIXED | PDL_DTC_DESTINATION_ADDRESS_PLUS | \
            PDL_DTC_SIZE_8 | \
            PDL_DTC_IRQ_COMPLETE | \
            PDL_DTC_TRIGGER_ICRXI1,
            dtc_iic1_rx_transfer_data,
            (uint8_t *)&RIIC1.ICDRR,
            data_storage,
            4,
            PDL_NO_DATA
        );

        /* Select I²C mode at 100kHz, 300ns rise time, 200ns fall time */
        R_IIC_Create(
            1,
            PDL_IIC_MODE_IIC | PDL_IIC_INT_PCLK_DIV_8,
            0,
            0,
            0,
            0,
            100E3,
            (300 << 16) | 200
        );

        /* Enable the DTC */
        R_DTC_Control(
            PDL_DTC_START,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_DATA
        );

        /* Write the data into the EEPROM */
        write_eeprom_data();

        /* Prepare the next data for the EEPROM */
        R_DTC_Control(
            PDL_DTC_UPDATE_SOURCE | PDL_DTC_UPDATE_COUNT,
            dtc_iic1_tx_transfer_data,
            eeprom_data_array_2,
            PDL_NO_PTR,
            8,
            PDL_NO_DATA
        );

        /* Write the data into the EEPROM */
        write_eeprom_data();

        /* Clear the data storage area */
        for (i = 0; i < 20; i++) data_storage[i] = 0x00;

        /* Reset the EEPROM sub-address to 0, using polling */
        R_IIC_MasterSend(
            1,
            PDL_IIC_STOP_DISABLE,
            EEPROM_ADDRESS,
            eeprom_data_array_1,
```

```
            1,
            PDL_NO_FUNC,
            0
        );

        /* Read data from the EEPROM on channel 1, using the DTC */
        read_eeprom_data();

        /* Prepare the next data */
        R_DTC_Control(
            PDL_DTC_UPDATE_DESTINATION | PDL_DTC_UPDATE_COUNT,
            dtc_iic1_rx_transfer_data,
            PDL_NO_PTR,
            &data_storage[5],
            5,
            PDL_NO_DATA
        );

        /* Read data from the EEPROM on channel 1, using the DTC */
        read_eeprom_data();
    }

    static void write_eeprom_data(void)
    {
        bus_busy = true;
        /* Send data to the EEPROM on channel 1, using the DTC */
        R_IIC_MasterSend(
            1,
            PDL_IIC_DTC_TRIGGER_ENABLE,
            EEPROM_ADDRESS,
            PDL_NO_PTR,
            0,
            PDL_NO_FUNC,
            0
        );
        while (bus_busy == true);

        /* Wait for 5ms while the EEPROM updates */
        R_CMT_CreateOneShot(
            0,
            0,
            5E-3,
            PDL_NO_FUNC,
            0
        );
    }

    static void read_eeprom_data(void)
    {
        bus_busy = true;
        /* Read data from the EEPROM on channel 1, using the DTC */
        R_IIC_MasterReceive(
            1,
            PDL_IIC_DTC_TRIGGER_ENABLE,
            EEPROM_ADDRESS,
            PDL_NO_PTR,
            0,
            PDL_NO_FUNC,
            0
        );
        while (bus_busy == true);
    }

    void iic_tx_dtc_end_handler(void)
    {
        uint16_t status_flags = 0;

        /* Wait for the transmission to complete */
```

```
        do
        {
            R_IIC_GetStatus(
                1,
                &status_flags,
                PDL_NO_PTR,
                PDL_NO_PTR
            );
        } while((status_flags & 0x0080u) == 0x0u);

        /* Issue a Stop condition on channel 1 */
        R_IIC_Control(
            1,
            PDL_IIC_STOP
        );

        bus_busy = false;
    }

    void iic_rx_dtc_end_handler(void)
    {
        uint32_t DestAddr = 0;

        /* Read the next destination address for the current transfer */
        R_DTC_GetStatus(
            dtc_iic1_rx_transfer_data,
            PDL_NO_PTR,
            PDL_NO_PTR,
            &DestAddr,
            PDL_NO_PTR,
            PDL_NO_PTR
        );

        /* Read one more byte with NACK condition on channel 1 and stop */
        R_IIC_MasterReceiveLast(
            1,
            (uint8_t *)DestAddr
        );

        bus_busy = false;
    }
```

Figure 5-26: An example of write data to and reading data from an EEPROM, using two DMAC channels

5.11.4. Slave mode

In this example the MCU behaves as a slave device on channel 0.
It will respond to 7-bit address 0001001b or 10-bit address 0010010010b.

```
/* Peripheral driver function prototypes */
#include “r_pdl_iic.h”
#include “r_pdl_cgc.h”
#include “r_pdl_intc.h”
#include “r_pdl_cmt.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

void slave_event_handler(void);

const uint8_t mcu_data_array[10] = {0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x5A};
volatile bool all_data_read;
volatile bool all_data_sent;
volatile uint8_t slave_data_storage[10];

void main(void)
{
    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        PDL_NO_DATA,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU’s Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Select I²C mode at 100kHz, 300ns rise time, 200ns fall time */
    R_IIC_Create(
        0,
        PDL_IIC_MODE_IIC | PDL_IIC_INT_PCLK_DIV_8,
        PDL_IIC_SLAVE_0_ENABLE_7 | PDL_IIC_SLAVE_1_ENABLE_10,
        0x12,
        0x0124,
        PDL_NO_DATA,
        100E3,
        (300 << 16) | 200
    );

    all_data_read = false;

    /* Monitor the channel.  Any data received will be stored in the receive buffer */
    R_IIC_SlaveMonitor(
        0,
        PDL_NO_DATA,
        slave_data_storage,
        10,
        slave_event_handler,
        7
    );
    while (all_data_read == false);
}

void slave_event_handler(void)
```

```
{
    uint16_t status_flags = 0;
    uint16_t tx_count = 0;
    uint16_t rx_count = 0;

    /* Read the channel status */
    R_IIC_GetStatus(
        0,
        &status_flags,
        &tx_count,
        &rx_count
    );

    /* Slave address 0 detected with a Read? */
    if ((status_flags & 0x0047) == 0x0041)
    {
        /* Assign 5 bytes to be read by a master */
        R_IIC_SlaveSend(
            0,
            mcu_data_array,
            5
        );
    }

    /* Slave address 1 detected with a Read? */
    else if ((status_flags & 0x0047) == 0x0042)
    {
        /* Assign 5 bytes to be read by a master */
        R_IIC_SlaveSend(
            0,
            slave_data_storage,
            5
        );
    }

    /* NACK and Stop detected */
    else if ((status_flags & 0x1800) == 0x1800)
    {
        all_data_sent = true;
    }

    /* Stop detected */
    else if ((status_flags & 0x1800) == 0x0800)
    {
        all_data_read = true;
        do
        {
            R_IIC_GetStatus(
                0,
                &status_flags,
                PDL_NO_PTR,
                PDL_NO_PTR
            );
        } while ((status_flags & 0x8000) != 0x0u);
    }
    else
    {
        /* Process any other conditions here */
    }
}
```

Figure 5-27: Configure the I²C channel and write 3 data bytes to the first locations

5.11.5. Slave mode with DMAC

In the following example, data is received using DMAC channel 2 and transmitted using DMAC channel 3.

The slave will respond to one 7-bit address and one 10-bit address.

The same EEPROM address locations are then read out in two bursts. DMAC channel 2 is used to handle the data transfer

```
/* Peripheral driver function prototypes */
#include “r_pdl_iic.h”
#include “r_pdl_cgc.h”
#include “r_pdl_dmac.h”

/* RPDL device-specific definitions */
#include “r_pdl_definitions.h”

/* Callback prototype */
void slave_event_handler(void);

/* IIC application definitions */
#define SLAVE_CHANNEL   0

#define MCU_ADDRESS_UPPER 0x0100
#define MCU_ADDRESS_LOWER 0x12
#define MCU_ADDRESS_0 0x12
#define MCU_ADDRESS_1 0x0124

#define BUFFER_SIZE 10

/* Global variables */
volatile bool transmission_completed;
volatile bool reception_completed;
volatile uint8_t slave_data_received[BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00};
volatile uint8_t slave_data_storage_0[BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00};
volatile uint8_t slave_data_storage_1[BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00};

void main(void)
{
    uint16_t status_flags = 0;
    uint32_t Count = 0;
    uint32_t i;
    uint32_t slave_0_ptr = 0;
    uint32_t slave_1_ptr = 0;

    /* Configure the clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        PDL_NO_DATA,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set up a DMAC channel for IIC reception */
    R_DMAC_Create(
        2,
        PDL_DMAC_SINGLE | PDL_DMAC_DESTINATION_ADDRESS_PLUS,
        PDL_DMAC_REQUEST_IIC0_RX,
        (uint8_t *)&RIIC0.ICDRR,
        slave_data_received,
        BUFFER_SIZE,
        PDL_NO_PTR,
```

```
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_FUNC,
            0
        );

        /* Set up a DMAC channel for IIC transmission */
        R_DMAC_Create(
            3,
            PDL_DMAC_SINGLE | PDL_DMAC_SOURCE_ADDRESS_PLUS,
            PDL_DMAC_REQUEST_IIC0_TX,
            slave_data_storage_0,
            (uint8_t *)&RIIC0.ICDRT,
            BUFFER_SIZE,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_FUNC,
            0
        );

        /* Select I²C mode at 100kHz, 300ns rise time, 200ns fall time */
        R_IIC_Create(
            SLAVE_CHANNEL,
            PDL_IIC_MODE_IIC | PDL_IIC_INT_PCLK_DIV_8,
            PDL_IIC_SLAVE_0_ENABLE_7 | PDL_IIC_SLAVE_1_ENABLE_10,
            MCU_ADDRESS_0,
            MCU_ADDRESS_1,
            PDL_NO_DATA,
            100E3,
            (300 << 16) | 200
        );

        /* Enable the DMAC channels */
        R_DMAC_Control(
            PDL_DMAC_2 | PDL_DMAC_3,
            PDL_DMAC_ENABLE,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA,
            PDL_NO_PTR,
            PDL_NO_PTR,
            PDL_NO_DATA
        );

        transmission_completed = false;
        reception_completed = false;

        while(1)
        {
            /* Any bus activity? */
            if ( (transmission_completed == true) || (reception_completed == true) )
            {
                /* Read the status flags */
                R_IIC_GetStatus(
                    SLAVE_CHANNEL,
                    &status_flags,
                    PDL_NO_PTR,
                    PDL_NO_PTR
                );

                if (transmission_completed == true)
                {
                    /* Which address was detected? */
                    if ((status_flags & 0x0001) != 0x0u)
                    {
                        /* Prepare the next data for the Master */
```

```
                    R_DMAC_Control(
                        PDL_DMAC_3,
                        PDL_DMAC_SUSPEND | PDL_DMAC_ENABLE | \
                        PDL_DMAC_UPDATE_SOURCE | PDL_DMAC_UPDATE_COUNT | \
                        PDL_DMAC_CLEAR_DREQ,
                        PDL_NO_DATA,
                        slave_data_storage_0,
                        PDL_NO_PTR,
                        BUFFER_SIZE,
                        PDL_NO_PTR,
                        PDL_NO_PTR,
                        PDL_NO_DATA
                    );
                }
                else if ((status_flags & 0x0002) != 0x0u)
                {
                    /* Prepare the next data for the Master */
                    R_DMAC_Control(
                        PDL_DMAC_3,
                        PDL_DMAC_SUSPEND | PDL_DMAC_ENABLE | \
                        PDL_DMAC_UPDATE_SOURCE | PDL_DMAC_UPDATE_COUNT | \
                        PDL_DMAC_CLEAR_DREQ,
                        PDL_NO_DATA,
                        slave_data_storage_1,
                        PDL_NO_PTR,
                        BUFFER_SIZE,
                        PDL_NO_PTR,
                        PDL_NO_PTR,
                        PDL_NO_DATA
                    );
                }

                transmission_completed = false;
            }

            else if (reception_completed == true)
            {
                /* Read the receive count */
                R_DMAC_GetStatus(
                    2,
                    PDL_NO_PTR,
                    PDL_NO_PTR,
                    PDL_NO_PTR,
                    &Count
                );
                Count = BUFFER_SIZE - Count;

                /* Which address was detected? */
                if ((status_flags & 0x0001) != 0x0u)
                {
                    for (i = 0; i < Count; i++)
                    {
                        slave_data_storage_0[slave_0_ptr] = slave_data_received[i];
                        slave_0_ptr ++;
                        if (slave_0_ptr == BUFFER_SIZE)
                        {
                            slave_0_ptr = 0;
                        }
                    }
                }
                else if ((status_flags & 0x0002) != 0x0u)
                {
                    for (i = 0; i < Count; i++)
                    {
                        slave_data_storage_1[slave_1_ptr] = slave_data_received[i];
                        slave_1_ptr ++;
                        if (slave_1_ptr == BUFFER_SIZE)
                        {
```

```
                                slave_1_ptr = 0;
                            }
                        }
                    }

                    /* Reset the receive buffer DMAC channel */
                    R_DMAC_Control(
                        PDL_DMAC_2,
                        PDL_DMAC_SUSPEND | PDL_DMAC_ENABLE | \
                        PDL_DMAC_UPDATE_DESTINATION | PDL_DMAC_UPDATE_COUNT | \
                        PDL_DMAC_CLEAR_DREQ,
                        PDL_NO_DATA,
                        PDL_NO_PTR,
                        slave_data_received,
                        BUFFER_SIZE,
                        PDL_NO_PTR,
                        PDL_NO_PTR,
                        PDL_NO_DATA
                    );

                    reception_completed = false;
                }

                /* Wait for the bus to become idle */
                do
                {
                    R_IIC_GetStatus(
                        SLAVE_CHANNEL,
                        &status_flags,
                        PDL_NO_PTR,
                        PDL_NO_PTR
                    );
                } while ((status_flags & 0x8000) != 0x0u);

                /* Re-start monitoring the channel */
                R_IIC_SlaveMonitor(
                    SLAVE_CHANNEL,
                    PDL_IIC_RX_DMAC_TRIGGER_ENABLE | PDL_IIC_TX_DMAC_TRIGGER_ENABLE,
                    PDL_NO_PTR,
                    PDL_NO_DATA,
                    slave_event_handler,
                    7
                );
            }
        }
    }

    void slave_event_handler(void)
    {
        uint16_t status_flags = 0;

        R_IIC_GetStatus(
            SLAVE_CHANNEL,
            &status_flags,
            PDL_NO_PTR,
            PDL_NO_PTR
        );

        /* NACK and Stop detected (end of transmission to the master)? */
        if ((status_flags & 0x1800) == 0x1800)
        {
            if (transmission_completed == false)
            {
                transmission_completed = true;
            }
            else
            {
                /* The main loop has failed to process the last transfer in time */
```

```
                nop();
            }
        }

        /* Stop detected? */
        else if ((status_flags & 0x1800) == 0x0800)
        {
            if (reception_completed == false)
            {
                reception_completed = true;
            }
            else
            {
                /* The main loop has failed to process the last transfer in time */
                nop();
            }
        }
    }
```

Figure 5-28: An example of IIC Slave operation, using two DMAC channels

5.12. Analog to Digital Converter

Figure 5-29 shows an example of ADC usage. ADC unit 2 is polled until the conversion is complete. Interrupts are enabled for ADC unit 3, which operates in the one-cycle scan mode.

```
/* Peripheral driver function prototypes */
#include "r_pdl_adc_10.h"
#include "r_pdl_cgc.h"
#include "r_pdl_intc.h"

/* RPDL device-specific definitions */
#include "r_pdl_definitions.h"

volatile uint8_t adc2_result;
volatile uint8_t adc3_result[4];

void ADC3Handler(void);

void main(void)
{
    /* Initialise the system clocks */
    R_CGC_Set(
        12.5E6,
        100E6,
        50E6,
        25E6,
        PDL_CGC_BCLK_DISABLE
    );

    /* Set the CPU's Interrupt Priority Level to 0 */
    R_INTC_Write(
        PDL_INTC_REG_IPL,
        PDL_NO_DATA,
        0
    );

    /* Configure ADC unit 2 for single scan on pin AN9, polled */
    R_ADC_10_Create(
        2,
        PDL_ADC_10_MODE_SINGLE | PDL_ADC_10_CHANNELS_OPTION_2,
        50E6,
        0.5E-6,
        PDL_NO_FUNC,
        0
    );

    /* Configure ADC unit 3 for one cycle scan on pins AN12 to AN15, interrupts */
    R_ADC_10_Create(
        3,
        PDL_ADC_10_MODE_ONE_CYCLE_SCAN | PDL_ADC_10_CHANNELS_OPTION_4,
        50E6,
        0.5E-6,
        ADC3Handler,
        6
    );

    /* Start conversions on ADC units 2 and 3 */
    R_ADC_10_Control(
        PDL_ADC_10_2_ON | PDL_ADC_10_3_ON
    );

    /* Read the level on AN9 */
    R_ADC_10_Read(
        2,
        &adc2_result
    );
```

```
        /* Shutdown unit 2 */
        R_ADC_10_Destroy(
            2
        );
    }

    void ADC3Handler(void)
    {
        R_ADC_10_Read(
            3,
            adc3_result
        );
    }
```

Figure 5-29: Example of ADC Conversion

6. RX-specific notes

6.1. Interrupts and processor mode

The RX CPU has two processor modes; supervisor and user.
The API driver functions will be executed by the CPU in user mode.
However any callback functions which are called by the API interrupt handlers will be executed by the CPU in supervisor mode.
This means that the privileged CPU instructions (RTFI, RTE and WAIT) can be executed by the callback function and any function that is called by the callback function.

The user must:

1. Avoid using the RTFI and RTE instructions.

These instructions are issued by the API interrupt handlers, so there should be no need for the user's code to use these instructions.

2. Use the wait() intrinsic function with caution.

This instruction is used by some API functions as part of power management, so there should be no need for the user's code to use this instruction.
More information on the processor modes can be found in §1.4 of the RX Family software manual.

6.2. Interrupts and DSP instructions

The accumulator (ACC) register is modified by the following instructions:

i. DSP (MACHI, MACLO, MULHI, MULLO, MVTACHI, MVTACLO and RACW).
ii. Multiply and multiply-and-accumulate (EMUL, EMULU, FMUL, MUL, and RMPA)

The accumulator (ACC) register is not pushed onto the stack by the API interrupt handlers.

If DSP instructions are being utilised in the users' code, callback functions which are called by the API interrupt handlers should either

a) Avoid using instructions which modify the ACC register.
b) Take a copy of the ACC register and restore it before exiting the callback function.

Revision History	RX610 Group User's Manual

		Description	
Rev.	Date	Page	Summary
0.13	Nov. 24, 2009	—	First edition issued
0.14	Dec. 02, 2009	—	Issued with WS-3C driver source.
0.15	Dec. 11, 2009	—	Updated with feedback from RSO.
0.16	Dec. 16, 2009	—	Created a single I/O pin definitions table.
0.17	Dec. 23, 2009	—	Added DTC and PPG functions.
0.18	Jan. 08, 2010	—	Corrected errors; updated IIC and PPG functions.
0.19	Feb. 03, 2010	—	Synchronised with some of the changes in the hardware manual; Updated IIC; Added MCU, Added LPC; Updated DTC; Standardised the declaration for prototypes with unknown types.
0.20	Feb. 25, 2010	—	Re-wrote the installation guide; Added general definitions; Re-designed the use of fast interrupts; Re-designed the SCI control to match the restrictions in the control of TE and RE bits.
0.21	Mar. 04, 2010	—	Updates to ADC and IIC descriptions.
0.22	Mar. 26, 2010	—	Added support for the 176-pin package; Separated DMAC and DTC control; Finished IIC.
0.23	Apr. 06, 2010.	1-1	Changed the company name.
		4-68	Corrected the TGRC and TGRD trigger options.
		4-120	Added a remark about avoiding combining interrupts and DMAC/DTC triggering.
		4-124	Removed the comment that Mark is the default.
		4-128	Corrected the category.
		4-129	Corrected the example code.
		4-130	Corrected the example code.
		5-12	Added SCI_Stop call in DMAC callback.
		5-27	Added I²C usage examples.
0.24	Apr. 26, 2010.	4-72	Updated the PFC function call in remark 2.
		4-142	Added a note that the byte count is reset if the data loops back.
		5-26	Added an example of synchronous reception and transmission.
0.25	Jun. 11, 2010.	3-1	Added the device group.
		4-6	Added the DDR register to the remarks.
		4-20	Re-designed the parameter list.
		4-21	Re-designed the parameter list.
		4-53	Separated the Enable and Suspend control; Added access to the current address and count registers; Added Transfer Request flag control.
		4-60	Removed level valid for IRQn.
		4-64	Added register parameters
		4-66	Added the block size parameter.
		4-75	Changed two channel references to unit.
		4-115	Removed the reference to reading the counter value.
		4-118	Added a remark for SCK5.
		4-124	Added GSM clock control options.
		4-139	Removed the Stop and Start control options.
		4-141	New function.
		4-142	Added TX DMAC/DTC control.
		4-144	Removed TX DMAC/DTC control.
		4-145	Added NACK control.

Rev.	Date	Description	
		Page	Summary
		4-146	Added Transmission status bit.
		4-154	Removed the reference to right-aligned limits.
		4-156	Removed the reference to right-aligned limits.
		5-34	Added IIC with DMAC example.
		5-38	Added IIC with DTC example.
0.26	Jun. 21, 2010.	1-5	Re-generated the screen shots.
		4-80	Re-named the overlap parameter.
		4-83	Moved the odd group next-data settings to the upper nybble.
		4-91	Corrected one Description page number.
		4-130	Swapped the parameters.
		4-133	Corrected one entry in the second timing table.
		4-142	Added a remark about re-setting the DMAC or DTC after a Stop condition is detected.
		5-9	Updated the DMAC_Control parameters.
		5-16	Updated the DTC_Control parameters.
		5-29	Updated the CRC_Read parameters.
		5-45	Added a usage example for Slave with DMAC.
1.00	Jul. 23, 2010	1-1	Added a note about required documents.
		4-59	Added the chain transfer trigger option.
		4-87	Corrected the TMRI pin assignment.
		4-91	Corrected the cross-reference and modified the second remark.
		4-93	Updated the func parameter description.
		4-94	Corrected the cross-reference and modified the second remark.
		4-106	Updated the func parameter description.
		4-109	Added counter control to R_CMT_Control.
		4-119	Updated the descriptions and remarks for DMAC and DTC usage.
		4-121	Updated the descriptions and remarks for DMAC and DTC usage.
		4-136	Updated the descriptions and remarks for DMAC and DTC usage.
		4-138	Updated the descriptions and remarks for DMAC and DTC usage.
		4-141	Updated the descriptions and remarks for DMAC and DTC usage.
		4-148	Added clarification for the data alignment.
		4-153	Added clarification for the data alignment.

Renesas Peripheral Driver Library
User's Manual
RX610 Group

Publication Date: Rev.1.00 Jul. 23, 2010

Published by: Renesas Electronics Corporation

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 1.0

RX610 Group

Renesas Electronics Corporation

R20UT0083EE0100