RX Family

MTU Driver Module Using Firmware Integration Technology (FIT)

R01AN2329EU0100
Rev.1.00
Sept 30, 2014

Introduction

This application note describes a Multi-Function Timer Pulse Unit 2 (MTU2a) driver module using Firmware Integration Technology (FIT) for the supported RX family MCUs that have the MTU2a peripheral. Details are provided that describe the architecture of this driver, integration of the FIT module into a user's application, and how to use the driver's Applications Programming Interface (**API**).

Typical operations that can be performed include basic timing, pulse generation, input capture, and pulse width modulation.

Target Devices

The list of devices that are currently supported by this API:

- RX63N Group, RX631 Group
- RX210 Group
- RX111 Group
- RX110 Group

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Related Documents

- Firmware Integration Technology User's Manual (R01AN1833EU)
- Board Support Package FIT Module (R01AN1685EU)
- Adding FIT Modules to Projects (R01AN1723EU)

Contents

1. Overview

This software provides an applications programing interface (API) for using the Multi-Function Timer Pulse Unit 2 (MTU2a) peripheral of supported RX Family microcontrollers. This API is designed to make it easy to add timing operations to your application software by providing a highly abstracted, use-case oriented interface to the hardware. Rather than referring to hardware specific register names and internal control bits, this API lets you set up your timing operations in terms of 'what you want to do'.

Use this driver to easily operate the MTU for basic timing operations, pulse generation, input capture, and pulse width modulation tasks. Options include generating interrupts based on timing events, outputting waveforms, and one-shot or continuous operation. The driver also has scalability to help reduce its memory size. It can be built into your project to include all features or a subset of only the features you need

The MTU driver module fits between the user application and the physical hardware to take care of the low-level hardware configuration and control tasks that manage the MTU2a peripheral.

While this driver strives to reduce the need to fully know the details of the MTU2a hardware, it is recommended to review the MTU2a peripheral chapter in the RX MCU hardware user's manual before using this software.

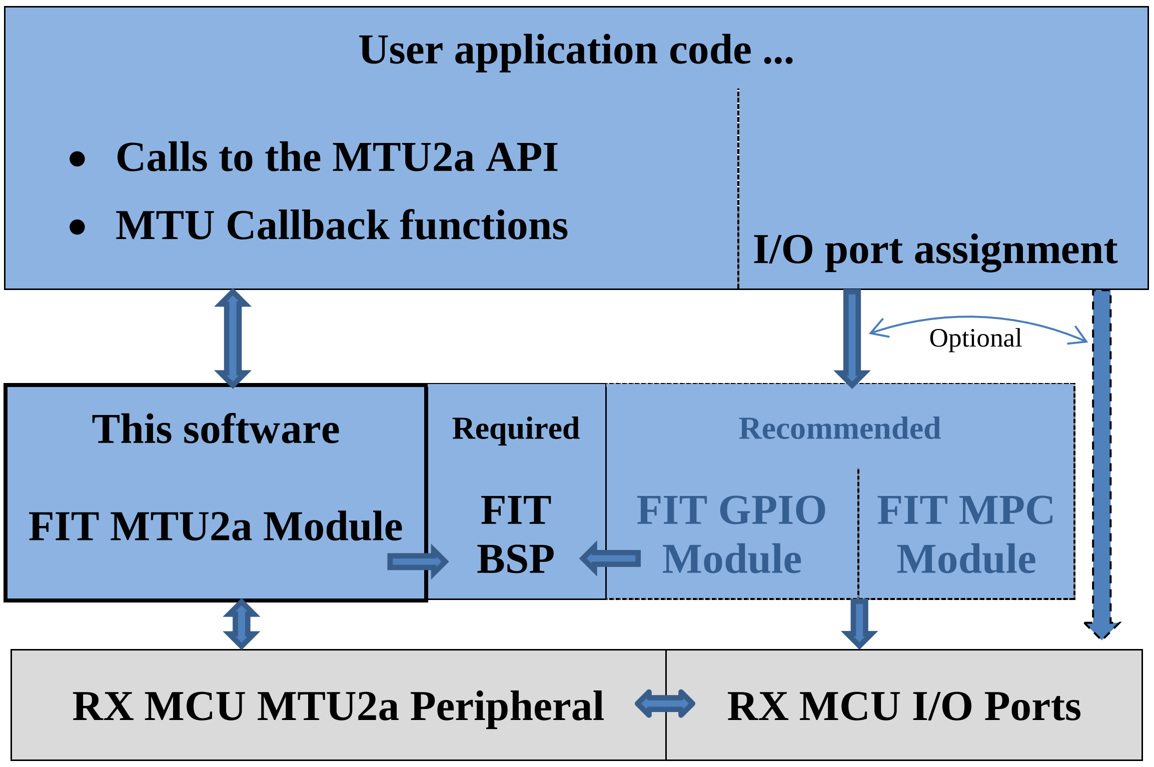

Figure 1 : Example Figure Showing Project Layers

1.1 Features

This software provides a general purpose Timer mode, an Input Capture mode, and two PWM mode.

The following subset of the hardware features available with the MTU2a peripheral are supported:

Table 1 Supported MTU Features

Item		**MTU0**	**MTU1**	**MTU2**	**MTU3**	**MTU4**
Count clock		PCLK/1 PCLK/4 PCLK/16 PCLK/64 — — MTCLKA MTCLKB MTCLKC MTCLKD	PCLK/1 PCLK/4 PCLK/16 PCLK/64 PCLK/256 — MTCLKA MTCLKB — —	PCLK/1 PCLK/4 PCLK/16 PCLK/64 — PCLK/1024 MTCLKA MTCLKB MTCLKC —	PCLK/1 PCLK/4 PCLK/16 PCLK/64 PCLK/256 PCLK/1024 MTCLKA MTCLKB — —	PCLK/1 PCLK/4 PCLK/16 PCLK/64 PCLK/256 PCLK/1024 MTCLKA MTCLKB — —
General registers (TGR)		TGRA TGRB	TGRA TGRB	TGRA TGRB	TGRA TGRB	TGRA TGRB
General registers/ buffer registers		TGRC TGRD	—	—	TGRC TGRD	TGRC TGRD
I/O pins		MTIOC0A MTIOC0B MTIOC0C MTIOC0D	MTIOC1A MTIOC1B — —	MTIOC1A MTIOC1B — —	MTIOC0A MTIOC0B MTIOC0C MTIOC0D	MTIOC0A MTIOC0B MTIOC0C MTIOC0D
Counter clear function		TGR compare match or input capture	TGR compare match or input capture	TGR compare match or input capture	TGR compare match or input capture	TGR compare match or input capture
Compare match output	0 output	✓	✓	✓	✓	✓
	1 output	✓	✓	✓	✓	✓
	Toggle output	✓	✓	✓	✓	✓
Input capture function		✓	✓	✓	✓	✓
Synchronous operation		✓	✓	✓	✓	✓
PWM mode 1		✓	✓	✓	✓	✓
PWM mode 2		✓	✓	✓	—	—
Interrupt sources		4 sources • Compare match or input capture 0A • Compare match or input capture 0B • Compare match or input capture 0C • Compare match or input capture 0D	2 sources • Compare match or input capture 1A • Compare match or input capture 1B	2 sources • Compare match or input capture 2A • Compare match or input capture 2B	4 sources • Compare match or input capture 3A • Compare match or input capture 3B • Compare match or input capture 3C • Compare match or input capture 3D	4 sources • Compare match or input capture 4A • Compare match or input capture 4B • Compare match or input capture 4C • Compare match or input capture 4D

1.1.1 Timer Function and Compare Match or Output Compare

Compare match, (also known as Output Compare when outputs are used) is a variation of the general Timer function. The Timer function provides the following features:

1. Specify MTU channel
2. Specify clocking source.
    a. Internal
    b. External
    c. Output from another timer (cascaded).
3. Specify compare-match count, or, if internal clock source, specify the frequency
4. What to do at compare/match:
    a. Optionally change state of output pin.
        i. Specify transition polarities
    b. Optionally generate interrupt request.
    c. Optional user-defined callback may be assigned to timer interrupt.
    d. Repeat cycle for continuous operation, or halt after one cycle (one-shot mode)
5. Optional validation of function arguments.

1.1.2 Timer Function Operation Examples.

These are some examples of the kinds of timer operations that can be performed using the Timer mode configuration with various optional settings. Many settings can be used in different combinations to produce other operations.

An interrupt or Callback function that repeats at a regular frequency can be created.

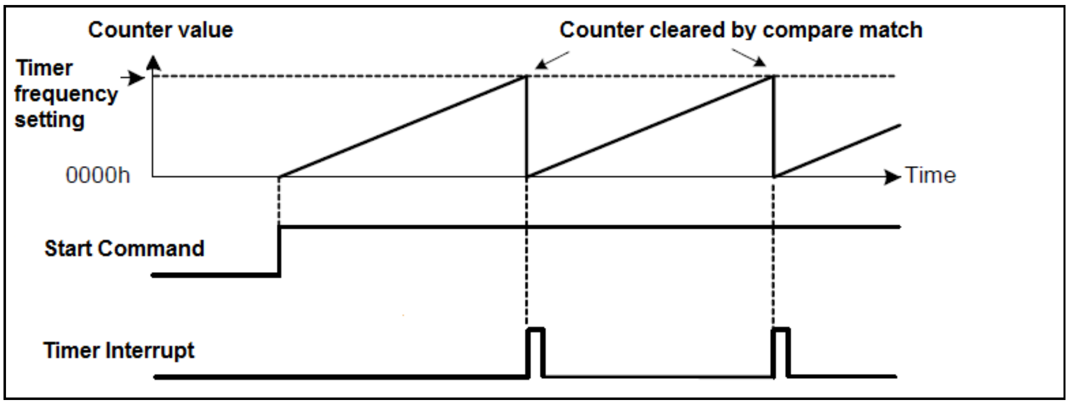

Figure 2 Periodic Counter Operation with interrupts

Output pins can be switched from their initial state after a specific interval.

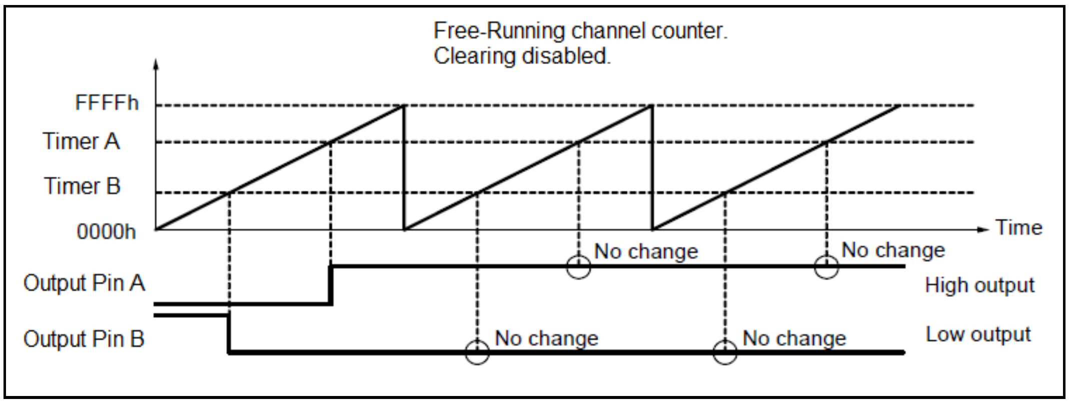

Figure 3 Example Output of 1 and 0 at Timer compare match

Using the toggle output option, multiphase clock outputs can be created.

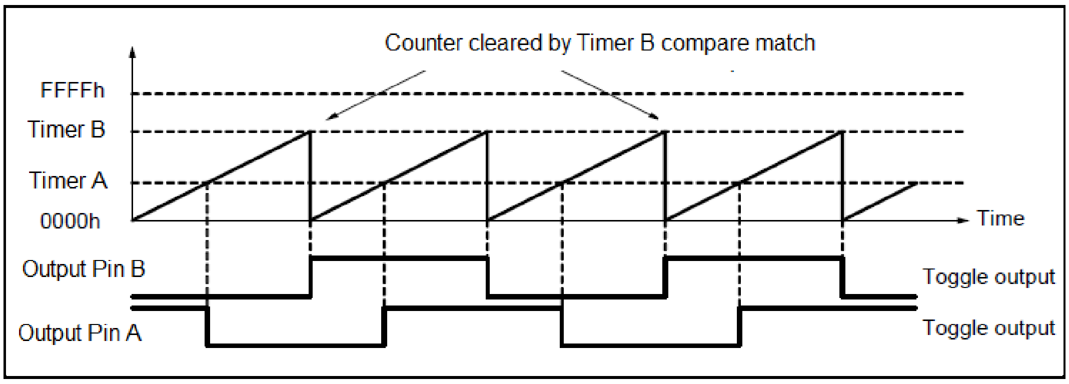

Figure 4 Example of Repeating Toggle Output Operation

1.1.3 Input Capture

Input capture is the sampling of the of clock tick count at the time of the input event. By taking multiple successive samples, the duration between input transitions (i.e.: pulse width) can be measured.

The Input Capture operation provides the following features:

1. Specify channel
2. Specify clock source –
   a. Internal
   b. External
   c. Output from another timer (cascaded).
3. Specify input capture transition polarities
4. Supports noise filter feature
5. What to do at input event:
   a. Optionally generate interrupt request.
   b. Optional user-defined callback may be assigned to input capture interrupt.
   c. Repeat cycle for continuous operation, or halt after one cycle (one-shot mode)
6. Status command to poll for event detection as alternative to interrupt/callback.
7. Optional validation of function arguments.

1.1.4 Input Capture Function Operation Example.

Figure 5 shows an example of input capture operation that can be performed using the input capture mode configuration with certain optional settings. Many settings can be used in different combinations to produce other related operations.

In this example, both rising and falling edges have been selected as the Capture A input pin capture input edge, the falling edge has been selected as the Capture B input pin input capture input edge, and counter clearing by Capture B input capture has been selected.

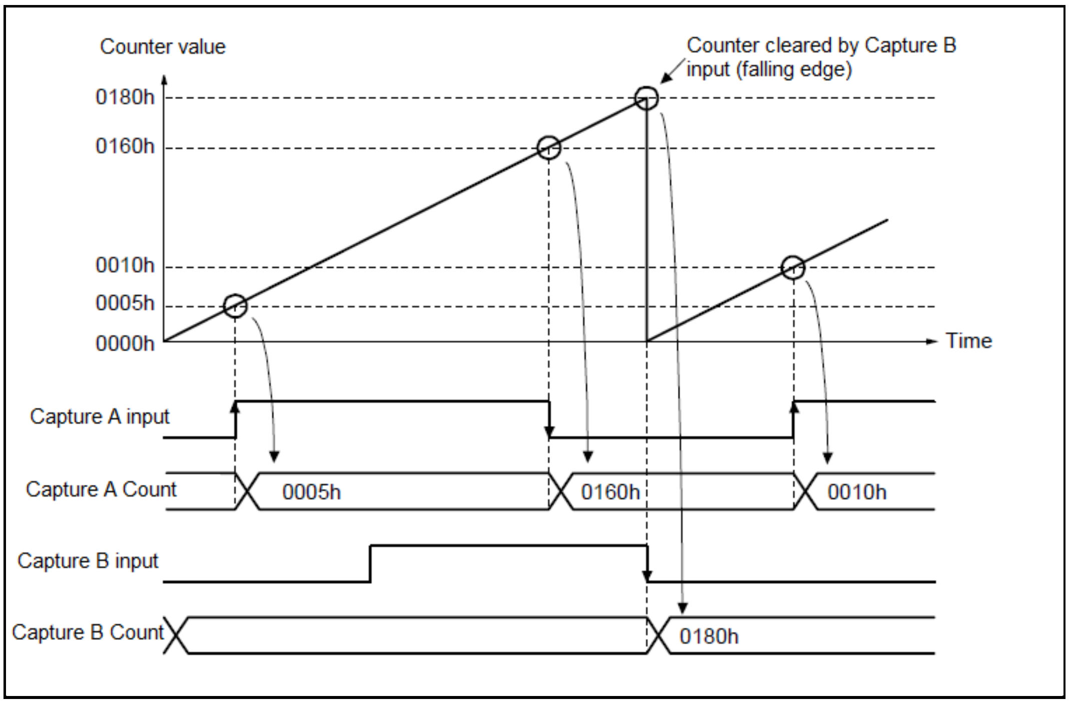

Figure 5 Example of Input Capture Operation

1.1.5 PWM Mode1

The PWM Mode1 operation supports the following features:

1. Specify MTU channel
2. Specify clocking source.
    a. Internal
    b. External
        i. Specify input pin
3. Specify cycle:
    a. if internal clock source, specify the cycle frequency.
    b. if external clock source specify cycle tick count instead of frequency.
4. Specify PWM duty:
    a. Duty is specified in terms of 1/10 % units, i.e.; 25.9% is specified as 259.
5. TGRA and TGRC will always be used as cycle timers, and TGRB and TGRD as duty timers.
6. What to do at compare-match of cycle and duty timers:
    a. Optionally change state of output pin.
        i. Specify transition polarities
    b. Optionally generate interrupt request.
    c. Optional user-defined callback may be assigned to timer interrupt.
6. Limited range checking of input arguments.

1.1.6 PWM Mode2

The PWM Mode2 operation supports the following features:

1. Specify MTU channel
2. Specify clocking source.
    a. Internal
    b. External
        i. Specify input pin
    c. Specify active clocking edge polarity.
3. Specify cycle:
    a. if internal clock source specify the cycle frequency.
   or
    b. if external clock source specify cycle tick count instead of frequency.
4. Specify PWM duty:
    a. Duty is specified in terms of 1/10 % units, i.e.; 25.9% is specified as 259.
5. One TGR must serve as cycle timer, all others can be duty timers.
6. What to do at compare-match of cycle and duty timers:
    a. Optionally change state of output pin.
        i. Specify transition polarities
    b. Optionally generate interrupt request.
    c. Optional user-defined callback may be assigned to timer interrupt.
7. Limited range checking of input arguments.

Table 2 PWM mode availability and output pins

Channel	Register	Output Pins	
		PWM Mode 1	PWM Mode 2
MTU0	MTU0.TGRA	MTIOC0A	MTIOC0A
	MTU0.TGRB		MTIOC0B
	MTU0.TGRC	MTIOC0C	MTIOC0C
	MTU0.TGRD		MTIOC0D
MTU1	MTU1.TGRA	MTIOC1A	MTIOC1A
	MTU1.TGRB		MTIOC1B
MTU2	MTU2.TGRA	MTIOC2A	MTIOC2A
	MTU2.TGRB		MTIOC2B
MTU3	MTU3.TGRA	MTIOC3A	Setting prohibited
	MTU3.TGRB		
	MTU3.TGRC	MTIOC3C	
	MTU3.TGRD		
MTU4	MTU4.TGRA	MTIOC4A	
	MTU4.TGRB		
	MTU4.TGRC	MTIOC4C	
	MTU4.TGRD		

Note: In PWM mode 2, PWM output is not possible for the TGR register in which the PWM cycle is set.

1.1.7 Examples of PWM Mode Operation.

These are some examples of the kinds of PWM operations that can be performed with various optional settings. Settings can be used in different combinations to produce other variations of PWM operation.

Figure 6 shows an example of operation in PWM mode 1. In this example, TGRA compare match is set as the TCNT clearing source, 0 is set as the initial output value and output value for TGRA, and 1 is set as the output value for TGRB. In this case, the value set in TGRA is used as the cycle, and the value set in TGRB is used as the duty.

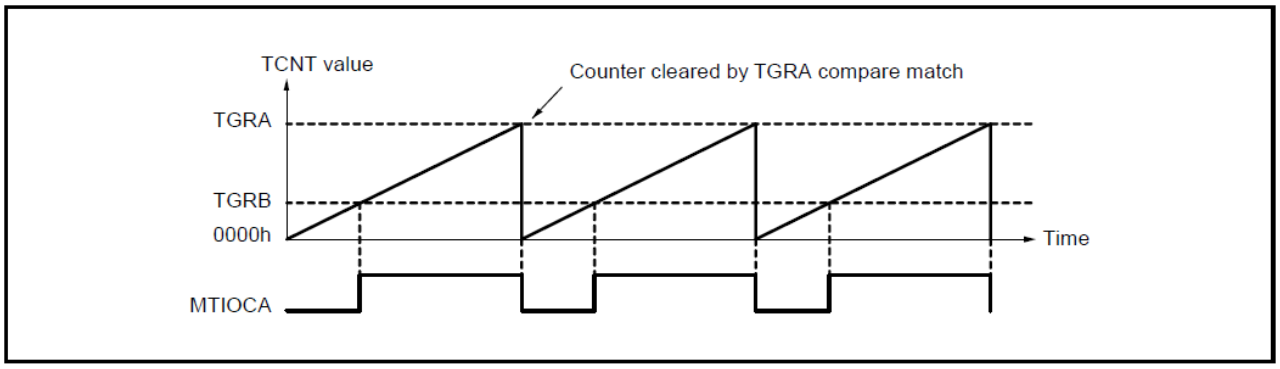

Figure 6 PWM Mode 1 Operation

Figure 7 shows an example of operation in PWM mode 2. In this example, synchronous operation is designated for MTU0 and MTU1, MTU1.TGRB compare match is set as the TCNT clearing source, and 0 is set as the initial output value and 1 as the output value for the other TGR registers (MTU0.TGRA to MTU0.TGRD and MTU1.TGRA), outputting 5-phase PWM waveforms. In this case, the value set in MTU1.TGRB is used as the cycle, and the values set in the other TGRs are used as the duty.

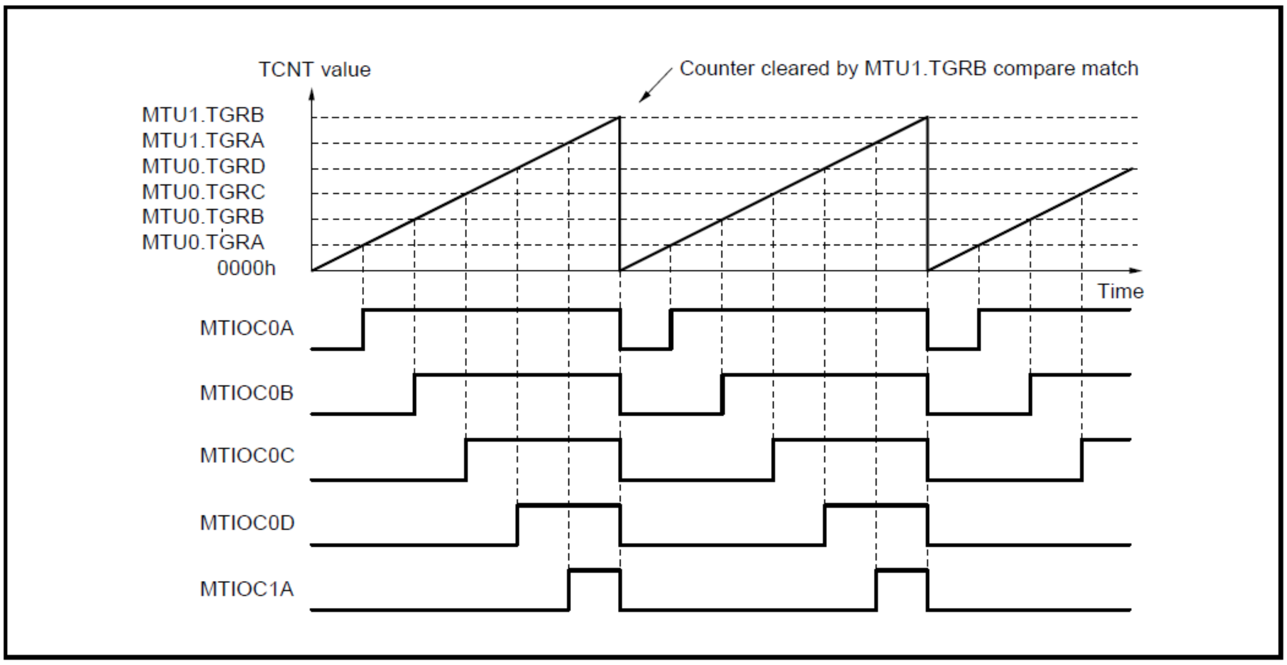

Figure 7 Example of PWM Mode 2 operation

2. Adding the MTU2a FIT Module to Your Project

To use this MTU2a Module you will need to add its source files to your software project and then add the desired function calls to your application code. This section will first describe how to add the FIT module files to your project.

This FIT module was developed using the e2studio development environment with Renesas RX compiler toolchain, so the installation instructions will assume that is what you are using. Use with other toolchains will require some modifications to the code which not covered in this application note.

2.1 Using this FIT Module with e2Studio

The driver must be added to an existing e2Studio project. It is best to use the e2Studio FIT plugin to add the driver to your project as that will automatically update the include file paths for you. See the external document, “Adding FIT Modules to Projects (R01AN1723EU)” for details.

Alternatively, the driver can be imported from the archive that accompanies this application note and manually added by following these steps:

1. This application note is distributed with a zip file package that includes the FIT MTU2a Module in its own folder *r_mtu_rx*.
2. Unzip the package into the location of your choice.
3. In a file browser window, browse to the directory where you unzipped the distribution package and locate the *r_mtu_rx* folder
4. Open your e2Studio workspace.
5. In the e2Studio project explorer window, select the project that you want to add the MTU2a Timers module to.
6. Drag and drop the *r_mtu_rx* folder from the browser window (or copy/paste) into your e2Studio project at the top level of the project.
7. Update the source search/include paths for your project by adding the paths to the module files:
    a. Navigate to the "Add directory path" control:

       'project name'->properties->C/C++ Build->Settings->Compiler->Source -Add (green + icon)

    b. Add the following paths:

       "${workspace_loc:/${ProjName}/ r_mtu_rx}"

       "${workspace_loc:/${ProjName}/ r_mtu_rx/src}"

8. Locate the *r_mtu_rx_config_reference.h*file in the *r_mtu_rx /ref/* source folder in your project and copy it to your project's *r_config* folder.
9. Change the name of the copy in the r_config folder to *r_mtu_rx_config.h*
10. Whether you used the plug-in or manually added the package to your project, it is necessary to configure the driver for your application.
    a. Make the required configuration settings by editing the copied *r_mtu_rx_config.h* file. See Configuration Overview.

The MTU2a Module uses the *r_bsp* package for certain MCU information and support functions The r_bsp package is easily configured through the *platform.h* header file which is located in the r_bsp folder. To configure the r_bsp package, open up *platform.h* and uncomment the #include for the board you are using.

3. MTU2a Module API Information

This Driver API follows the Renesas API naming standards.

3.1 Hardware Requirements

This driver requires that your MCU support the following features:

- **RX600, RX200, or RX100 series MCU with one MTU2a peripheral circuit.**

3.2 Hardware Resource Requirements

This section details the hardware peripherals that this driver requires. Unless explicitly stated, these resources must be reserved for the driver and the user cannot use them independently.

- **MTU2a**

3.3 Software Requirements

This driver is dependent upon the support from the following software:

- This software depends on a FIT compliant BSP module being present. It assumes that the related I/O ports have been correctly initialized elsewhere prior to calling this software's API functions.
- This software requires that the peripheral clock (PCLKB) has been initialized by the BSP prior to calling the APIs of this module. The r_bsp macro ‘BSP_PCLKB_HZ’ is used by the driver for calculating bit-rate register settings. If the user modifies the PCLKB setting outside of the r_bsp module or the r_cgc module, then calculations based on the frequency settings parameters will differ from actual timing.

3.4 Supported Toolchains

This driver is tested and working with the following toolchains:

- Renesas RX Compiler Toolchain v2.01.00

3.5 Header Files

All API calls are accessed by including a single file "r_mtu_rx_if.h" which is supplied with this software’s project code. Build-time configuration options are selected or defined in the file "r_mtu_rx_config.h"

3.6 Integer Types

This project uses ANSI C99 “Exact width integer types” in order to make the code clearer and more portable. These types are defined in *stdint.h*.

3.7 MTU2a features Supported by Driver

Compare match, input capture, and PWM modes 1 and 2 are supported on MTU channels 0 to 4.

3.8 MTU2a features not supported

Phase Counting, Complimentary PWM, and Reset Synchronized PWM operations are not supported.
MTU2a channel 5 is not supported by this driver due to substantial differences in its features.

3.9 Multi-Channel MTU Timer Support

For supported RX family MCUs this driver will operate all available channels on an individually selectable basis with the same body of code. Each channel can be configured with its own setup independently of the other channels in use.

3.10 Configuration Overview

Some features or behavior of the software are determined at build-time by configuration options that the user must select.

Configuration options in *r_mtu_rx_config.h*	
`MTU_CFG_REQUIRE_LOCK`	If this is set to (1) then the MTU driver will attempt to obtain the lock for the channel when performing certain operations to prevent concurrent access conflicts. This option requires support from a FIT BSP.
`MTU_CFG_PARAM_CHECKING_ENABLE`	Validation of arguments passed to MTU API functions can be enabled or disabled. Disabling parameter checking is provided for systems that absolutely require faster and smaller code. By default the module is configured to use the setting of the system-wide BSP_CFG_PARAM_CHECKING_ENABLE macro. This can be locally overridden for the MTU module by redefining MTU_CFG_PARAM_CHECKING_ENABLE. To control parameter checking locally, set MTU_CFG_PARAM_CHECKING_ENABLE to 1 to enable it, otherwise set to 0 skip checking.
`MTU_CFG_USE_CHn`	Enable the MTU channels to use at build-time. Portions of code that support the unused channels will be omitted from the build, resulting in smaller code size. (0) = not used. (1) = used.

`MTU_CFG_USE_TIMER`
 	
`MTU_CFG_USE_CAPTURE`
 	
`MTU_CFG_USE_PWM`
	Code for unused MTU functions can be excluded from the build to reduce size. Any combination can be selected, but at least one must be used.
	
	(0) = not used. (1) = used.

`MTU_IR_PRIORITY_CHANn`	Sets the shared interrupt priority for the channel. This is provided as a convenience. Priority can still be changed outside of this module at run time *after* a call to R_MTU_Timer_Open or R_MTU_Capture_Open has been made to a channel. However the next call to R_MTU_Timer_Open or R_MTU_Capture_Open for that channel will change it back to this configuration value.
`MTU_CFG_FILT_CHANn`	Input capture operations can apply digital filtering to the input capture signal. This option sets the filter clock divisor. Larger Divisor values result in longer filtering periods. The available options for setting this configuration are defined in " r_mtu_rx_if.h" as MTU_FILTER_PCLK_DIV_1, MTU_FILTER_PCLK_DIV_8, MTU_FILTER_PCLK_DIV_32, and MTU_FILTER_PCLK_EXTERNAL. When the MTU_FILTER_PCLK_EXTERNAL option is chosen there is no division of the externally supplied filter clock.

Table 3 : List of MTU Timers driver module configuration options

4. Port Configuration

4.1 System Initialization

MTU I/O signals may be optionally assigned to different I/O ports. The initialization of the Multi-function Pin Controller (MPC) and the PORT registers must be handled by the application, either through calls to the FIT MPC module, or directly. This initialization must be performed before a call is made to the driver Open function. A sample of direct I/O pin initialization for the RX111 channel 0 is provided here, assigning the 4 MTIOC0* pins to the MTU:

```
/* Assign all MTIOC outputs for MTU channel 0. */

/* Use BSP function to enable writing to MPC registers. */
R_BSP_RegisterProtectDisable(BSP_REG_PROTECT_MPC);

/* Make the MTU peripheral signal assignment selections in the MPC registers. */
MPC.PE3PFS.BYTE = 0x03;   /* PE3 assigned to MTIOC0A */
MPC.PA1PFS.BYTE = 0x01;   /* PA1 assigned to MTIOC0B */
MPC.P17PFS.BYTE = 0x03;   /* P17 assigned to MTIOC0C */
MPC.PA3PFS.BYTE = 0x01;   /* PA3 assigned to MTIOC0D */

/* Use BSP function to protect writing to MPC registers. */
R_BSP_RegisterProtectEnable(BSP_REG_PROTECT_MPC);

/* Switch over pin control from GPIO to peripheral mode. */
PORT1.PMR.BIT.B7  = 1;
PORTA.PMR.BIT.B1  = 1;
PORTA.PMR.BIT.B3  = 1;
PORTE.PMR.BIT.B3  = 1;
```

5. Driver Architecture

5.1 Interrupts

Interrupts may be enabled individually for each MTU event source. When the callback function is enabled, the interrupt is also automatically enabled. However, interrupts may be enabled without using callbacks. The result of this will be that the interrupt ISR will be invoked, some minimal processing will be completed to see that the callback is not enabled, and the interrupt will return. This feature is provided to support the use of DMAC or DTC operations that require interrupts from the peripheral and where the user does not want a callback on completion. When DMAC or DTC operation is active the interrupt requests will be intercepted by the DMAC or DTC peripheral and the ISR for this MTU module will not execute until the DMAC/DTC operation completes and stops taking the interrupts.

Not all MTU interrupt sources are supported.

5.2 Callback Functions

The definition of callbacks follows the FIT 1.0 specification rules:

a. Callback functions take one argument. This argument is ‘void * pdata’.

b. Before calling a callback function the function pointer is be checked to be valid. At a minimum the pointer is be checked to be:

i. Non-null

ii. Not equal to FIT_NO_FUNC macro.

5.3 Example callback function prototype declaration.

```
    void callback(void * pdata)
```

5.4 Using Callback functions

When a callback function is provide for an MTU channel operation at the “Open” procedure, the same callback is used for every event source that has its callback action enabled on that channel. For compare/match timer operations, the user defined callback that was specified in the “Timer Open” call will be called from within an interrupt at the time of compare/match for each event. For input capture operations, the user defined callback that was specified in the “Capture Open” call will be called from within an interrupt at the time of each input capture event. Event information is passed to the callback function from the respective interrupt that called it.

A pointer to a data structure containing the channel number, the event source, and event status data is passed as the only argument. It is up to the user application to process the provided information appropriately. Since callbacks are being processed within the context of the interrupt, and interrupts are disabled at this time, it is strongly recommended that the user-defined callback function complete as quickly as possible to avoid missing further system interrupts.

The most typical use of the callback function is to inform the application that a timer event has occurred.

Example callback function:

```
    mtu_callback_data_t  g_my_capture_cb_data = {0};
    bool                       g_my_capture_occured = false;

    void my_capture_callback(void * pdata)
    {
        mtu_callback_data_t * cb_data = (mtu_callback_data_t *)pdata;

        g_my_capture_cb_data.channel   = cb_data->channel;   // Which channel? 0-4?
        g_my_capture_cb_data.timer_num = cb_data->timer_num; // Which capture A,B,C or D?
        g_my_capture_cb_data.count     = cb_data->count;     // The capture timer count.

        g_my_capture_occured = true;
    }
```

6. Applications Programming Interface (API)

6.1 Summary

The following functions are included in this design:

Function	Description
R_MTU_Timer_Open ()	Sets up an MTU channel for compare/match timer operations. Applies power to the MTU channel, initializes the associated registers, enables interrupts, and takes an optional callback function pointer for responding to interrupt events.
R_MTU_Capture_Open()	Sets up an MTU channel for input capture operations. Applies power to the MTU channel, initializes the associated registers, enables interrupts, and takes an optional callback function pointer for responding to interrupt events.
R_MTU_PWM_Open ()	Sets up an MTU channel for basic pwm operations. Applies power to the MTU channel, initializes the associated registers, enables interrupts, and takes an optional callback function pointer for notifying the user application at interrupt level when pwm events have occurred.
R_MTU_Close ()	Disables the specified MTU channel and returns it to 'available' status.
R_MTU_Control ()	Handles special hardware or software operations for the MTU channel .
R_MTU_GetVersion ()	Returns the driver version number.

6.2 API Data Structures

This section introduces the data structures that are used with the driver's API functions.

6.2.1 Special Data Types

To provide strong type checking and reduce errors, most of the parameters used in API functions require arguments to be passed using the provided type definitions. Allowable values are defined in the public interface file "*r_mtu_rx_if.h*". The following special types have been defined:

Enumeration of MTU channel numbers
Type: mtu_channel_t

Values: MTU_CHANNEL_0, MTU_CHANNEL_1, MTU_CHANNEL_2,
MTU_CHANNEL_3, MTU_CHANNEL_4 (Note: channels 3 and 4 not available for RX110)

MTU Clocking source selections
Type: mtu_clk_sources_t

Values:

```
MTU_CLK_SRC_INTERNAL        // Use internal clock (PCLK)
MTU_CLK_SRC_EXT_MTCLKA      // External clock input on MTCLKA pin
MTU_CLK_SRC_EXT_MTCLKB      // External clock input on MTCLKB pin
MTU_CLK_SRC_EXT_MTCLKC      // External clock input on MTCLKC pin
MTU_CLK_SRC_EXT_MTCLKD      // External clock input on MTCLKD pin
MTU_CLK_SRC_CASCADE         // Clocking by overflow from other channel counter.  (only on
                            certain channels. Not for PWM)
```

Possible settings for MTU output pins
Type: mtu_output_states_t

Values:

```
MTU_PIN_NO_OUTPUT  // Output high impedance.
MTU_PIN_LO_GOLO    // Initial output is low. Low output at compare match.
MTU_PIN_LO_GOHI    // Initial output is low. High output at compare match.
MTU_PIN_LO_TOGGLE  // Initial output is low. Toggle (alternate) output at compare match.
MTU_PIN_HI_GOLO    // Initial output is high. Low output at compare match.
MTU_PIN_HI_GOHI    // Initial output is high. High output at compare match.
```

MTU_PIN_HI_TOGGLE // Initial output is high. Toggle (alternate) output at compare match.

Settings for counting clock active edge

Type: mtu_clk_edges_t

Values: MTU_CLK_RISING_EDGE, MTU_CLK_FALLING_EDGE, MTU_CLK_ANY_EDGE

Settings for input capture signal active edge

Type: mtu_cap_edges_t

Values: MTU_CAP_RISING_EDGE, MTU_CAP_FALLING_EDGE, MTU_CAP_ANY_EDGE

MTU counter clearing source selections

Type: mtu_clear_src_t

Values:	MTU_CLR_TIMER_A	// Clear the channel counter on the "A" compare or capture event.
	MTU_CLR_TIMER_B	// Clear the channel counter on the "B" compare or capture event.
	MTU_CLR_TIMER_C	// Clear the channel counter on the "C" compare or capture event.
	MTU_CLR_TIMER_D	// Clear the channel counter on the "D" compare or capture event.
	MTU_CLR_SYNC	// Clear the channel counter when another sync'ed channel clears.
	MTU_CLR_DISABLED	// Never clear the channel counter.

Actions to be done upon timer, capture, or PWM event

Type: mtu_actions_t

Values:	MTU_ACTION_OUTPUT	// Change state of output pin.
	MTU_ACTION_CAPTURE	// Perform input capture for this event.
	MTU_ACTION_INTERRUPT	// Generate interrupt request on event.
	MTU_ACTION_CALLBACK	// Execute user-defined callback on event (also generates interrupt).
	MTU_ACTION_TRIGGER_ADC	// Trigger ADC on this event. Timer A events only.
	MTU_ACTION_REPEAT	// Continuously repeat the timer cycle and actions
	MTU_ACTION_NONE	// Do nothing with this timer.

PCLK divisor for internal clocking source

Used when doing input capture and the MTU internal clocking source is selected. Divides the Peripheral Clock (PCLK) by this. Choose a clocking period long enough to capture the expected input event within a 16-bit timer count. Larger divisor values mean a longer period can be captured, but resolution is reduced (longer time per counter tick).

Type: mtu_src_clk_divisor_t

Values:	MTU_SRC_CLK_DIV_1	// Counter tick = PCLK
	MTU_SRC_CLK_DIV_4	// Counter tick = PCLK/4
	MTU_SRC_CLK_DIV_16	// Counter tick = PCLK/16
	MTU_SRC_CLK_DIV_64	// Counter tick = PCLK/64
	MTU_SRC_CLK_DIV_256	// Counter tick = PCLK/256
	MTU_SRC_CLK_DIV_1024	// Counter tick = PCLK/1024

MTU control command codes

Type: mtu_cmd_t

Values:	MTU_CMD_START	// Activate clocking
	MTU_CMD_STOP	// Pause clocking
	MTU_CMD_SAFE_STOP	// Stop clocking and set outputs to safe state
	MTU_CMD_RESTART	// Zero the counter then resume clocking
	MTU_CMD_SYNCHRONIZE	// Specify channels to group for synchronized clearing.
	MTU_CMD_GET_STATUS	// Retrieve the current status of the channel
	MTU_CMD_CLEAR_EVENTS	// Clears the interrupt flags for the channel
	MTU_CMD_SET_CAPT_EDGE	// Sets the detection edge polarity for input capture.

MTU Timers control command data structures

See R_MTU_Control() chapter.

Channel Settings structure for R_MTU_Timer_Open()

The R_MTU_Timer_Open() function requires a pointer to an initialized instance of this structure to set various operating modes at the channel open.

Type: mtu_timer_chnl_settings_t

Members:

```
mtu_clk_src_t  clock_src          // Specifies the MTU counter clocking source.
mtu_clear_src_t      clear_src    // Specifies the counter clearing source.
mtu_timer_settings_t    timer_a
mtu_timer_settings_t    timer_b
mtu_timer_settings_t    timer_c
mtu_timer_settings_t    timer_d
```

This structure has a number of sub-structures as members:

Type: mtu_timer_settings_t

Members:

```
uint32_t   freq                         // If internal clock source, the desired event frequency, or if external
                                        // clock the Compare-match count.
mtu_timer_actions_cfg_t actions         // Things to do when this timers event occurs.
```

Type: mtu_timer_actions_cfg_t

Members:

```
mtu_actions_t    do_action
mtu_output_states_t     output          // Output pin transition type when output action is selected
```

Channel Settings structure for R_MTU_Capture_Open()

The R_MTU_Capture_Open() function requires a pointer to an initialized instance of this structure to set various operating modes at the channel open. This structure has a number of sub-structures as members:

Type: mtu_capture_chnl_settings_t

Members:

```
mtu_clk_src_t     clock_src         // Specifies the MTU counter clocking source.
mtu_src_clk_divisor_t  clock_div          // Internal clock divisor selection
mtu_clear_src_t         clear_src         // Specifies the counter clearing source.
mtu_capture_settings_t  capture_a
mtu_capture_settings_t  capture_b
mtu_capture_settings_t  capture_c
mtu_capture_settings_t  capture_d
```

Type: mtu_capture_settings_t

Members:

```
mtu_actions_t          actions           // Things to do when this capture event occurs.
mtu_cap_edges_t        capture_edge      // Specify the input transition polarity to capture.
bool                   filter_enable     // Digital noise filter on or off.
```

Channel Settings structure for R_MTU_PWM_Open()

The R_MTU_PWM_Open() function requires a pointer to an initialized instance of this structure to set various operating modes at the channel open.

Type: mtu_pwm_chnl_settings_t

Members:

```
mtu_pwm_clk_src_t          clock_src;        // Specify clocking source.
uint32_t                   cycle_freq;       // Cycle frequency for the channel
mtu_clear_src_t            clear_src;        // Specify the counter clearing source.
mtu_pwm_mode_t              pwm_mode;        // Specify mode 1 or mode 2
mtu_pwm_settings_t         pwm_a;
mtu_pwm_settings_t         pwm_b;
mtu_pwm_settings_t         pwm_c;
mtu_pwm_settings_t         pwm_d;
```

This structure has a number of sub-structures as members:

Type: mtu_pwm_settings_t

Members:

```
uint16_t            duty;
mtu_actions_t       actions;
mtu_output_states_t outputs;   // Specify transition polarities.
```

Callback function data structure

The channel number, the event source, and the count value of the event source are passed in this data structure to the user defined callback function upon timer or capture interrupt.

Type: mtu_callback_data_t

Members:

mtu_channel_t	channel	// The channel number.
mtu_timer_num_t	timer_num	// The interrupt event source.
uint32_t	count	// The input capture count value or compare-match setting (TGR read value).

6.3 API Function Return Values

The different values API functions can return. These values are enumerated.

Return Type: mtu_err_t

Values:	**Cause**
MTU_SUCCESS	Function completed without errors
MTU_ERR_BAD_CHAN	Invalid channel number
MTU_ERR_CH_NOT_OPENED	Channel not yet opened. Function can not be completed.
MTU_ERR_CH_NOT_CLOSED	Channel still open from previous open.
MTU_ERR_UNKNOWN_CMD	Control command is not recognized.
MTU_ERR_INVALID_ARG	Argument is not valid for parameter.
MTU_ERR_ARG_RANGE	Argument is out of range for parameter.
MTU_ERR_NULL_PTR	Received null pointer; missing required argument.
MTU_ERR_LOCK	A lock procedure failed
MTU_ERR_UNDEF	Undefined/unknown error

6.4 R_MTU_Timer_Open()

The **R_MTU_Timer_Open** function is responsible for preparing an MTU channel for compare/match type timing operations.

6.4.1 R_MTU_Timer_Open() Summary

Format

```
   mtu_err_t  R_MTU_Timer_Open (mtu_channel_t              channel,
                                mtu_timer_chnl_settings_t *pconfig,
                                void          (*pcallback)(void *pdata));
```

Parameters

channel

  Number of the MTU channel to be initialized

pconfig

  Pointer to MTU channel configuration data structure.

pcallback

  Pointer to user defined function called from interrupt.

Return Values

MTU_SUCCESS:	*Successful; channel initialized*
MTU_ERR_BAD_CHAN:	*Channel number is not available*
MTU_ERR_CH_NOT_CLOSED:	*Channel currently in operation; Perform R_MTU_Close() first*
MTU_ERR_NULL_PTR:	*pconfig pointer is NULL*
MTU_ERR_INVALID_ARG:	*The pconfig structure contains an invalid value.*
MTU_ERR_ARG_RANGE:	*Could not configure for frequency requested.*
MTU_ERR_LOCK:	*The lock could not be acquired. Channel is busy.*

Properties

Prototyped in file "r_mtu_rx_if.h"

Description

The Open function is responsible for preparing an MTU channel for compare/match type timing operations. This function must be called once before starting compare/match type timing operations an a given channel. Once successfully completed, the status of the selected MTU channel will be set to "open". After that this function should not be called again for the same MTU channel without first performing a "close" by calling R_MTU_Close().

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a 'Lock error' return.

Example

```
   my_result = R_MTU_Timer_Open(MTU_CHANNEL_0, &my_timer_configs, &my_callback_func);
```

6.4.2 R_MTU_Timer_Open() Usage Details

Compare/match operations on the MTU2a using this driver are generally referred to as "Timer mode". Compare/match works by setting a "compare" count value and then clocking a counter until the counter value matches the target compare value. When the counter matches the compare value a "compare/match event" is said to have occurred. It is at this point that the driver will perform the assigned event "actions".

Timer mode Channel Configuration Structure

Most of the configuration of this driver for timer mode consists of setting up the configuration data structure. Your application must declare a structure of type *mtu_timer_chnl_settings_t* and then initialize it with all the settings needed to perform the desired timing operation. You will then pass the address of this data structure to the driver in the call to R_MTU_Timer_Open(). This driver provides a lot of flexibility in determining what specific actions to perform and the timing parameters you want to apply. Please study the following information that describes the initialization and usage of the various configuration elements.

The mtu_timer_chnl_settings_t structure is explained in detail:

```
typedef struct mtu_timer_chnl_settings_s
{
    mtu_clk_src_t  clock_src;      // Specify clocking source.
    mtu_clear_src_t      clear_src;      // Specify the counter clearing source.
    mtu_timer_settings_t timer_a;
    mtu_timer_settings_t timer_b;
    mtu_timer_settings_t timer_c;
    mtu_timer_settings_t timer_d;
} mtu_timer_chnl_settings_t;
```

Setting up the time-base

The MTU2a needs a clocking source to provide ‘ticks’ to its counters. This can either come from the internal RX peripheral clock (PCLK), or it can come from an external source on an input pin (MTCLKA through MTCLKD). When the internal clock is selected the MTU2a has the ability to divide the clock using one of several clock divisors of ‘prescalers’. This provides a wide range of resolution or clock periods to be programmed. In addition, since the internal PCLK frequency is know to this driver, timings can be specified in real-time units.

Alternatively, when an external clock source is selected, the actual frequency is not known to this driver. So in that case the timing settings will be expressed in tick counts, and it will be up to the user’s application software to manage any real-time interpretation, if needed.

Make selections for the clocking source settings structure**, mtu_clk_src_t clock_src:**

```
typedef struct mtu_clk_src_s
{
    mtu_clk_sources_t   source;       // Internal clock or external clock input
    mtu_clk_edges_t     clock_edge;  // Specify the clock active edge.
} mtu_clk_src_t
```

source

Type	Choose one from this column	Description
mtu_clk_sources_t	MTU_CLK_SRC_INTERNAL	Use internal clock (PCLK)
	MTU_CLK_SRC_EXT_MTCLKA	External clock input on MTCLKA pin
	MTU_CLK_SRC_EXT_MTCLKB	External clock input on MTCLKB pin
	MTU_CLK_SRC_EXT_MTCLKC	External clock input on MTCLKC pin *1
	MTU_CLK_SRC_EXT_MTCLKD	External clock input on MTCLKD pin *1
	MTU_CLK_SRC_CASCADE	Clocking by overflow from other channel counter *2

Note1: Only on channels 0, 3, and 4.
Note2: This clock source only available for channel 1. Channel 1 counts on channel 2 overflow.

clock_edge

Type	Choose one from this column	Description
mtu_clk_edges_	MTU_CLK_RISING_EDGE	Counter is incremented on rising edge of clock
	MTU_CLK_FALLING_EDGE,	Counter is incremented on falling edge of clock
	MTU_CLK_ANY_EDGE	Counter is incremented on both rising and falling clock edges.

Many timer operations require that the tick counter be cleared to 0 upon occurrence of a compare match event or upon the clearing of the counter in another channel. Or, counter clearing may be disabled, in which case the counter will continue to increment until it overflows the 16-bit counter register at which point it will start over at 0.

Specify the counter clearing source - **mtu_clear_src_t clear_src**

```
typedef struct mtu_timer_chnl_settings_s
{
    mtu_clk_src_t  clock_src;      // Specify clocking source.
    mtu_clear_src_t      clear_src;      // Specify the counter clearing source.
    mtu_timer_settings_t timer_a;
    mtu_timer_settings_t timer_b;
    mtu_timer_settings_t timer_c;
    mtu_timer_settings_t timer_d;
} mtu_timer_chnl_settings_t;
```

clear_src

Type	Choose one from this column	Description
mtu_clear_src_t	MTU_CLR_TIMER_A	Clears counter on the "A" compare event.
	MTU_CLR_TIMER_B	Clears counter on the "B" compare event.
	MTU_CLR_TIMER_C	Clears counter on the "C" compare event.
	MTU_CLR_TIMER_D	Clears counter on the "D" compare event.
	MTU_CLR_SYNC	Clears counter when another sync'ed channel clears.
	MTU_CLR_DISABLED	Never clear the channel counter. Let it continue.

Setting up the individual timer events.

The MTU2a has up to four timer general registers (TGRs) per channel, each of which can hold a different 16-bit timer compare value. This driver treats these as ‘event sources’. There is one tick counter per channel. When the counter value increments to a value that matches the value in one of the TGRs a compare/match event occurs. This continues, unless stopped, and eventually all of the TGR values will be matched. Each of these four match events can trigger its own set of actions. In this driver we call these four compare/match settings **timer_a** through **timer_d**. Each timer must be configured to set its match count and to assign the actions to perform on its compare/match event.

Selectable timer event actions can include; request an **interrupt**, execute the **callback** function, drive an **output** pin, **repeat** the timer operation indefinitely, or do **nothing**.

The settings choices are, for the most part, identical for all timers available on a channel. However, not all channels have all four timers, and some timers have differing capabilities. Consult the Timers Features table for a list of available timers and settings they support.

```
typedef struct mtu_timer_chnl_settings_s
{
    mtu_clk_src_t  clock_src;      // Specify clocking source.
    mtu_clear_src_t      clear_src;      // Specify the counter clearing source.
    mtu_timer_settings_t timer_a;
    mtu_timer_settings_t timer_b;
    mtu_timer_settings_t timer_c;
    mtu_timer_settings_t timer_d;
} mtu_timer_chnl_settings_t;
```

Timer frequency setting

```
typedef struct mtu_timer_settings_s
{
    uint32_t   freq;      // If internal clock source, the desired event frequency
                          // or if external the Compare-match count.
     mtu_timer_actions_cfg_t   actions;
} mtu_timer_settings_t;
```

Internal clocking

The **freq** setting is interpreted differently by the driver depending on whether the internal clock source is selected or an external clock source. When the internal clock is selected, then the **freq** setting is to be the desired event frequency in 1Hz units. For example, if you want the timer event to occur at a 1MHz rate, assign a value of 1000000 to this structure member.

This value will be used by the driver to attempt to provide the closest possible event frequency. If the selected frequency is higher than the driver can deliver, then an out of range error will be returned. If the value is in range, then the closest possible frequency that is equal to or lower than the requested frequency will be applied. The accuracy of the resulting frequency chosen by the driver depends on the value requested. This is determined by the base PCLK frequency setting of the RX MCU, and the available clock divisors for the given channel. Furthermore, when multiple event timers are being used on a given MTU channel, the timer with the longest period (largest count) will be used to determine the clock prescaler (PCLK divisor) that will be applied to all of the events for that channel.

Specifying the event timing in terms of frequency is natural when a continuous, repeating event stream or output clock waveform is desired. When only a single event is desired, then typically it would be preferable to specify the timing in terms of 'period' units such as seconds or microseconds. However this would require floating-point math operations which we chose to avoid in this driver to help reduce code size. To specify a period for a single or "one-shot" event, simply calculate the mathematical inverse of the desired period to get the frequency equivalent, and then apply that to the **freq** setting.

External clocking

When an external clock has been chosen as the clock source, no real-time based frequency calculations are done by the driver. In this case the freq setting is interpreted simply as an absolute timer count value. Be sure to specify a value no greater than 16-bits in size, or 65535 maximum.

Timer actions settings

One or more actions must be assigned to every timer event source when configuring a channel.

```
typedef struct mtu_timer_settings_s
{
    uint32_t   freq;         // If internal clock source, the desired event frequency
                             // or if external the Compare-match count.
    mtu_timer_actions_cfg_t   actions;
} mtu_timer_settings_t;
```

actions

```
typedef struct mtu_timer_actions_config_s
{
    mtu_actions_t        do_action;  // Various actions that can be done at timer
                                     // event.
    mtu_output_states_t  output;     // Output pin transition type when output
                                     // action is selected.
} mtu_timer_actions_cfg_t;
```

do_action

Type	Choose one or more from this column	Description
mtu_actions_t	MTU_ACTION_OUTPUT	Control state of an output pin.
	MTU_ACTION_INTERRUPT	Generate interrupt request on event.
	MTU_ACTION_CALLBACK	Execute user-defined callback on event [*1] (also generates interrupt).
	MTU_ACTION_TRIGGER_ADC	Trigger ADC on this event. Timer A events only.
	MTU_ACTION_REPEAT	Continuously repeat the timer cycle and actions
	MTU_ACTION_NONE	Don't use this timer event. [*2]

Note 1: If assigning the MTU_ACTION_CALLBACK action, an interrupt is automatically assigned. It is not necessary to also specify MTU_ACTION_INTERRUPT for the timer event.

Note 2: Do not combine MTU_ACTION_NONE with other actions for the same timer event. The result is undefined.

When assigning multiple actions to a timer event the actions must be combined to a single value by logical ‘OR’ing the individual selections.

Example:
```
    mtu_timer_actions_cfg_t   my_timer_a_config;
    my_timer_a_config.do_action = (mtu_actions_t)(MTU_ACTION_OUTPUT | MTU_ACTION_REPEAT);
```

Note: When a combination do_action value is created it should be typecast to (mtu_actions_t) to prevent a compiler warning.

Note: All timer events on the channel must be configured, even those that will not be used. Unused timers must have the MTU_ACTION_NONE action assigned. When a timer has bee configured with the MTU_ACTION_NONE action, the rest of that timer’s settings are ignored and so need not be set.

Output pin actions

If the MTU_ACTION_OUTPUT action is to be used to control an output pin, then further settings are required to choose the output pin state transition mode.

```
typedef struct mtu_timer_actions_config_s
{
    mtu_actions_t        do_action;  // Various actions that can be done at timer
                                     // event.
    mtu_output_states_t  output;     // Output pin transition type when output
                                     // action is selected.
} mtu_timer_actions_cfg_t;
```

output

Type	Choose one from this column	Description
mtu_output_states_t	MTU_PIN_NO_OUTPUT	Output is high impedance.
	MTU_PIN_LO_GOLO	Initial output is low. Low output at compare match
	MTU_PIN_LO_GOHI	Initial output is low. High output at compare match.
	MTU_PIN_LO_TOGGLE	Initial output is low. Toggle (alternate) output at compare match.
	MTU_PIN_HI_GOLO	Initial output is high. Low output at compare match.
	MTU_PIN_HI_GOHI	Initial output is high. High output at compare match.
	MTU_PIN_HI_TOGGLE	Initial output is high. Toggle (alternate) output at compare match.

The value specified in the output structure member will be used by the driver to set the MTU Timer I/O Control Register (TIOR). The initial output is valid when the counter is stopped. So upon completion of the R_MTU_Timer_Open() function call, the output pin will be set to the state specified as ‘Initial’. The output pin will remain at the initial state until the timer is started and a compare match event occurs, then it will change to the next assigned state. Thereafter, outputs do not return to their initial state unless the timer is stopped with the MTU_CMD_SAFE_STOP command of the R_MTU_Timer_ Control() function

For the _GOLO and _GOHI output settings, this means that once the output changes to low or high on compare match, it will remain in that state and does not switch on subsequent compare match events.

For the _TOGGLE setting, the output pin will alternately switch to the opposite state at each compare match event. This behavior makes it possible to create a continuous 50% duty output clock signal. Since it is toggling, one side effect of this is that output cycle frequency will be 1/2 of the compare match event frequency.

6.5 R_MTU_Capture_Open()

The **R_MTU_Capture_Open** function is responsible for preparing an MTU channel for input capture timing operations.

6.5.1 R_MTU_Capture_Open() Summary

Format

```
   mtu_err_t  R_MTU_Capture_Open (mtu_channel_t                channel,
                                  mtu_capture_chnl_settings_t *pconfig,
                                  void       (*pcallback)(void *pdata));
```

Parameters

channel

Number of the MTU channel to be initialized

pconfig

Pointer to MTU channel configuration data structure.

pcallback

Pointer to user defined function called from interrupt.

Return Values

MTU_SUCCESS:	*Successful; channel initialized*
MTU_ERR_BAD_CHAN:	*Channel number is not available*
MTU_ERR_CH_NOT_CLOSED:	*Channel currently in operation; Perform R_MTU_Close() first*
MTU_ERR_NULL_PTR:	*pconfig pointer is NULL*
MTU_ERR_INVALID_ARG:	*The pconfig structure contains an invalid value.*
MTU_ERR_LOCK:	*The lock could not be acquired. Channel is busy.*

Properties

Prototyped in file “r_mtu_rx_if.h”

Description

The Open function is responsible for preparing an MTU channel for input capture timing operations. This function must be called once before starting input capture timing operations an a given channel. Once successfully completed, the status of the selected MTU channel will be set to "open". After that this function should not be called again for the same MTU channel without first performing a "close" by calling R_MTU_Close().

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a ‘Lock error’ return.

Example

```
   my_result = R_MTU_Capture_Open(MTU_CHANNEL_0, &my_timer_configs, &my_callback_func);
```

6.5.2 R_MTU_Capture_Open() Usage Details

Input capture operations on the MTU2a using this driver are generally referred to as “Capture mode”. Input capture works by setting the MTU2a MTIOC pins to input capture mode, and then clocking a counter until an input signal state change matches chosen transition edge. When the correct edge is detected the value in the counter is saved to the corresponding timer general register (TGR) for that input pin. It is at this point that an “input capture event” has occurred, and the driver will perform the assigned event “actions”.

Capture mode Channel Configuration Structure

Most of the configuration of this driver for capture mode consists of setting up the configuration data structure. Your application must declare a structure of type *mtu_ capture_chnl_settings_t* and then initialize it with all the settings needed to perform the desired capture operation. You will then pass the address of this data structure to the driver in the call to R_MTU_ Capture_Open(). This driver provides a lot of flexibility in determining what specific actions to perform and the parameters you want to apply. Please study the following information that describes the initialization and usage of the various configuration elements.

The mtu_timer_chnl_settings_t structure is explained in detail:

```
typedef struct mtu_capture_chnl_settings_s
{
    mtu_clk_src_t      clock_src;   // Specify clocking source.
    mtu_src_clk_divisor_t   clock_div;   // Internal clock divisor selection.
    mtu_clear_src_t         clear_src;   // Specify the counter clearing source.
    mtu_capture_settings_t  capture_a;
    mtu_capture_settings_t  capture_b;
    mtu_capture_settings_t  capture_c;
    mtu_capture_settings_t  capture_d;
} mtu_capture_chnl_settings_t;
```

Setting up the time-base

The MTU2a needs a clocking source to provide 'ticks' to its counters. This can either come from the internal RX peripheral clock (PCLK), or it can come from an external source on an input pin (MTCLKA through MTCLKD). When the internal clock is selected the MTU2a has the ability to divide the clock using one of several clock divisors of 'prescalers'. This provides a wide range of resolution or clock periods to be programmed.

When an external clock source is selected no additional division of the clock is available, so the **clock_div** setting is ignored and may be omitted. The timing values returned by the input capture operation will be expressed in tick counts, and it will be up to the user's application software to convert these values to real-time units if needed.

Make selections for the clocking source settings structure**, mtu_clk_src_t clock_src:**

```
typedef struct mtu_clk_src_s
{
    mtu_clk_sources_t   source;      // Internal clock or external clock input
    mtu_clk_edges_t     clock_edge;  // Specify the clock active edge.
} mtu_clk_src_t
```

source

Type	Choose one from this column	Description
mtu_clk_sources_t	MTU_CLK_SRC_INTERNAL	Use internal clock (PCLK)
	MTU_CLK_SRC_EXT_MTCLKA	External clock input on MTCLKA pin
	MTU_CLK_SRC_EXT_MTCLKB	External clock input on MTCLKB pin
	MTU_CLK_SRC_EXT_MTCLKC	External clock input on MTCLKC pin [*1]
	MTU_CLK_SRC_EXT_MTCLKD	External clock input on MTCLKD pin [*1]
	MTU_CLK_SRC_CASCADE	Clocking by overflow from other channel counter [*2]

Note1: Only on channels 0, 3, and 4.
Note2: This clock source only available for channel 1. Channel 1 counts on channel 2 overflow.

clock_edge

Type	Choose one from this column	Description
mtu_clk_edges_	MTU_CLK_RISING_EDGE	Counter is incremented on rising edge of clock
	MTU_CLK_FALLING_EDGE,	Counter is incremented on falling edge of clock
	MTU_CLK_ANY_EDGE	Counter is incremented on both rising and falling clock edges.

Many input capture operations require that the tick counter be cleared to 0 upon occurrence of the input capture event or upon the clearing of the counter in another channel. Or, counter clearing may be disabled, in which case the counter will continue to increment until it overflows the 16-bit counter register at which point it will start over at 0.

Specify the counter clearing source - **mtu_clear_src_t clear_src**

```
typedef struct mtu_capture_chnl_settings_s
{
    mtu_clk_src_t     clock_src;   // Specify clocking source.
    mtu_src_clk_divisor_t   clock_div;   // Internal clock divisor selection.
    mtu_clear_src_t         clear_src;   // Specify the counter clearing source.
    mtu_capture_settings_t  capture_a;
    mtu_capture_settings_t  capture_b;
    mtu_capture_settings_t  capture_c;
    mtu_capture_settings_t  capture_d;
} mtu_capture_chnl_settings_t;
```

clear_src

Type	Choose one from this column	Description
mtu_clear_src_t	MTU_CLR_TIMER_A	Clears counter on the "A" capture event.
	MTU_CLR_TIMER_B	Clears counter on the "B" capture event.
	MTU_CLR_TIMER_C	Clears counter on the "C" capture event.
	MTU_CLR_TIMER_D	Clears counter on the "D" capture event.
	MTU_CLR_SYNC	Clears counter when another sync'ed channel clears.
	MTU_CLR_DISABLED	Never clear the channel counter. Let it continue.

Setting up the individual capture events.

The MTU2a has up to four MTIOC pins per channel that can be configured as input pins for input capture. For each input capture pin there is a corresponding timer general register (TGR), each of which can hold a different 16-bit timer count value. There is one tick counter per channel. When the MTU channel clocking is started, the counter increments until an input signal state change matches chosen transition edge. When the correct edge is detected the value in the counter is saved to the corresponding TGR for that input pin. At this point an “input capture event” has occurred, and the driver will perform the assigned event “actions”. Each of the four input capture events on a channel can trigger its own actions settings. In this driver we call these four input capture source **capture_a** through **capture_d**. Each capture source must be configured to set the actions to perform on its input capture event.

To perform input capture on a capture source the capture action must be enabled. Additional optional capture event actions can include; request an **interrupt**, execute the **callback** function, and **repeat** the capture operation indefinitely If the capture source will not be used the **do nothing** action must be selected.

The settings choices are, for the most part, identical for all capture sources available on a channel. However, not all channels have all four capture sources. Consult the Supported Features table for a list of available capture sources and settings they support.

```
typedef struct mtu_capture_chnl_settings_s
{
    mtu_clk_src_t     clock_src;   // Specify clocking source.
    mtu_src_clk_divisor_t   clock_div;   // Internal clock divisor selection.
    mtu_clear_src_t         clear_src;   // Specify the counter clearing source.
    mtu_capture_settings_t  capture_a;
    mtu_capture_settings_t  capture_b;
    mtu_capture_settings_t  capture_c;
    mtu_capture_settings_t  capture_d;
} mtu_capture_chnl_settings_t;
```

Capture actions settings

One or more actions must be assigned to every capture event source when configuring a channel.

```
typedef struct mtu_capture_settings_s
{
    mtu_actions_t      actions;
    mtu_cap_edges_t   capture_edge;       // Specify transition polarities.
    bool               filter_enable;     // Noise filter on or off.
} mtu_capture_settings_t;
```

The enumerated list of actions is defined in mtu_actions_t. This list is shared with the Timer mode operations, however not all of the actions in the enumeration are used in Capture mode. The following table describes the actions for use with Compare mode.

actions

Type	Choose one or more from this column	Description
mtu_actions_t	MTU_ACTION_CAPTURE	Perform input capture from this capture source. [*1]
	MTU_ACTION_INTERRUPT	Generate interrupt request on capture event.
	MTU_ACTION_CALLBACK	Execute user-defined callback on capture event [*2] (also generates interrupt).
	MTU_ACTION_TRIGGER_ADC	Trigger ADC on this event. Timer A events only.
	MTU_ACTION_REPEAT	Continuously repeat the capture cycle and actions
	MTU_ACTION_NONE	Don't use this capture event. [*3]

Note 1: At a minimum, the MTU_ACTION_CAPTURE action must be selected to perform input capture from this source.

Note 2: If assigning the MTU_ACTION_CALLBACK action, an interrupt is automatically assigned. It is not necessary to also specify MTU_ACTION_INTERRUPT for the timer event.

Note 3: Do not combine MTU_ACTION_NONE with other actions for the same timer event. The result is undefined.

When assigning multiple actions to a timer event the actions must be combined to a single value by logical 'OR'ing the individual selections.

Example:

```
mtu_capture_settings_t my_capture_a_config;
my_capture_a_config.action = (mtu_actions_t)(MTU_ACTION_CALLBACK | MTU_ACTION_REPEAT);
```

Note: When a combination action value is created it should be typecast to (mtu_actions_t) to prevent a compiler warning.

Note: All capture source events on the channel must be configured, even those that will not be used. Unused capture sources must have the MTU_ACTION_NONE action assigned. When a capture source has bee configured with the MTU_ACTION_NONE action, the rest of that source's settings are ignored and so need not be set.

Input capture edge setting.

The input signal transition edge polarity must be selected. This is the active edge that will trigger the input capture event.

```
typedef struct mtu_capture_settings_s
{
    mtu_actions_t      actions;
    mtu_cap_edges_t   capture_edge;       // Specify transition polarities.
    bool               filter_enable;     // Noise filter on or off.
} mtu_capture_settings_t;
```

capture_edge

Type	Choose one from this column	Description
mtu_cap_edges_t	MTU_CAP_RISING_EDGE	Only rising edges trigger input capture event.
	MTU_CAP_FALLING_EDGE	Only falling edges trigger input capture event.
	MTU_CAP_ANY_EDGE	Both rising and falling edges trigger input capture event.

Example:

```
mtu_capture_settings_t my_capture_a_config;
```

```
   my_capture_a_config.capture_edge = MTU_CAP_RISING_EDGE;
```

Input capture noise filter setting.

A digital noise filter can be applied to the capture input signal. This setting enable or disables the use of the noise filter. Additional configuration of the filter timing (filter clock) is available at build time in the driver configuration file.

```
typedef struct mtu_capture_settings_s
{
    mtu_actions_t     actions;
    mtu_cap_edges_t   capture_edge;       // Specify transition polarities.
    bool              filter_enable;      // Noise filter on or off.
} mtu_capture_settings_t;
```

Example:

```
   mtu_capture_settings_t my_capture_a_config;
   my_capture_a_config. filter_enable = true;  // Turn noise filtering on.
```

6.6 R_MTU_PWM_Open()

The **R_MTU_PWM_Open** function is responsible for preparing an MTU channel for PWM timing operations.

6.6.1 R_MTU_PWM_Open() Summary

Format

```
   mtu_pwm_err_t  R_MTU_PWM_Open (mtu_channel_t            channel,
                                  mtu_pwm_chnl_settings_t *pconfig,
                                  void                (*pcallback)(void *pdata));
```

Parameters

channel

Number of the MTU channel to be initialized

pconfig

Pointer to configuration structure that has been instantiated and set by user application..

pcallback

Pointer to user defined function called from interrupt.

Return Values

MTU_SUCCESS:	*Successful; channel initialized*
MTU_ERR_BAD_CHAN:	*Channel number is not available*
MTU_ERR_CH_NOT_CLOSED:	*Channel currently in operation; Perform R_MTU_PWM_Close() first*
MTU_ERR_NULL_PTR:	*pconfig pointer is NULL*
MTU_ERR_INVALID_ARG:	*The pconfig structure contains an invalid value.*
MTU_PWM_ERR_ARG_RANGE:	*Could not configure for frequency requested.*
MTU_ERR_LOCK:	*The lock could not be acquired. Channel is busy.*

Properties

Prototyped in file “r_mtu_pwm_rx_if.h”

Description

The R_MTU_PWM_Open function sets an MTU channel up for basic pwm operations. It applies power to the MTU channel, initializes the associated registers, enables interrupts, and takes a callback function pointer for notifying the user application at interrupt level when pwm compare-match events have occurred. This function must be called once before starting PWM operations on a given channel.

After successful completion of this function, pwm operations on the channel will not begin until execution of the R_MTU_PWM_Control() “MTU_PWM_CMD_START” command.

The MTU channel will be reserved as “in use” after successful completion of this function. This protects against resource conflicts from other drivers or application tasks that might try to re-open the channel while it is in use. A local locking feature shall be used to accomplish this. The channel will remain in the locked (reserved in-use) state until released by a call to R_MTU_PWM_Close ().

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a ‘Lock error’ return.

Example

```
       mtu_pwm_err_t  result;
       mtu_pwm_chnl_settings_t ex_pwm_settings;

       ex_pwm_settings.pwm_mode              = MTU_PWM_MODE_1;
       ex_pwm_settings.clock_src.source      = MTU_CLK_SRC_INTERNAL;
       ex_pwm_settings.clock_src.clock_edge  = MTU_CLK_RISING_EDGE;
       ex_pwm_settings.clear_src             = MTU_CLR_TIMER_A;
       ex_pwm_settings.pwm_a.cycle_freq      = 10000; // 10KHz.
       ex_pwm_settings.pwm_a.actions         = MTU_ACTION_OUTPUT;
       ex_pwm_settings.pwm_a.outputs         = MTU_PIN_HI_GOHI;
       ex_pwm_settings.pwm_b.duty            = 600; // 60.0 %
       ex_pwm_settings.pwm_b.actions         = MTU_ACTION_OUTPUT;
       ex_pwm_settings.pwm_b.outputs         = MTU_PIN_HI_GOLO;

       ex_pwm_settings.pwm_c.cycle_freq      = 30000;  // 30KHz
```

```
        ex_pwm_settings.pwm_c.actions        = MTU_ACTION_NONE;
        ex_pwm_settings.pwm_c.outputs        = MTU_PIN_LO_GOLO;

        ex_pwm_settings.pwm_d.duty           = 250; // 25.0%
        ex_pwm_settings.pwm_d.actions        = MTU_ACTION_OUTPUT;
        ex_pwm_settings.pwm_d.outputs        = MTU_PIN_LO_GOHI;

        result = R_MTU_PWM_Open(MTU_CHANNEL_1, &ex_pwm_settings, FIT_NO_FUNC);
        result = R_MTU_PWM_Control(MTU_CHANNEL_1, MTU_PWM_CMD_START, FIT_NO_PTR);
```

6.6.2 R_MTU_PWM_Open() Usage Details

PWM Channel Configuration Structure

Most of the configuration of this driver for pwm mode consists of setting up the configuration data structure. Your application must declare a structure of type *mtu_pwm_chnl_settings_t* and then initialize it with all the settings needed to perform the desired operation. You will then pass the address of this data structure to the driver in the call to R_MTU_PWM_Open(). This driver provides a lot of flexibility in determining what specific actions to perform and the timing parameters you want to apply. Please study the following information that describes the initialization and usage of the various configuration elements.

The mtu_pwm_chnl_settings_t structure is explained in detail:

```
    typedef struct mtu_pwm_chnl_settings_s
    {
        mtu_pwm_clk_src_t   clock_src;   // Specify clocking source.
        uint32_t            cycle_freq;  // Cycle frequency for the channel
        mtu_clear_src_t     clear_src;   // Specify the counter clearing source.
        mtu_pwm_mode_t      pwm_mode;    // Specify mode 1 or mode 2
        mtu_pwm_settings_t  pwm_a;
        mtu_pwm_settings_t  pwm_b;
        mtu_pwm_settings_t  pwm_c;
        mtu_pwm_settings_t  pwm_d;
    } mtu_pwm_chnl_settings_t;
```

Setting up the time-base

The MTU2a needs a clocking source to provide ‘ticks’ to its counters. This can either come from the internal RX peripheral clock (PCLK), or it can come from an external source on an input pin (MTCLKA through MTCLKD). When the internal clock is selected the MTU2a has the ability to divide the clock using one of several clock divisors of ‘prescalers’. This provides a wide range of resolution or clock periods to be programmed. In addition, since the internal PCLK frequency is know to this driver, timings can be specified in real-time units.

Alternatively, when an external clock source is selected, the actual frequency is not known to this driver. So in that case the timing settings will be expressed in tick counts, and it will be up to the user’s application software to manage any real-time interpretation, if needed.

Make selections for the clocking source settings structure**, mtu_pwm_clk_src_t clock_src:**

```
    typedef struct mtu_pwm_clk_src_s
    {
        mtu_clk_sources_t    source;      // Internal clock or external clock input
        mtu_clk_edges_t      clock_edge;  // Specify the clock active edge.
    } mtu_pwm_clk_src_t
```

source

Type	Choose one from this column	Description
mtu_clk_sources_t	MTU_CLK_SRC_INTERNAL	Use internal clock (PCLK)
	MTU_CLK_SRC_EXT_MTCLKA	External clock input on MTCLKA pin
	MTU_CLK_SRC_EXT_MTCLKB	External clock input on MTCLKB pin
	MTU_CLK_SRC_EXT_MTCLKC	External clock input on MTCLKC pin [*1]
	MTU_CLK_SRC_EXT_MTCLKD	External clock input on MTCLKD pin [*1]

Note1: Only on channels 0, 3, and 4.

clock_edge

Type	Choose one from this column	Description
mtu_clk_edges_	MTU_CLK_RISING_EDGE	Counter is incremented on rising edge of clock
	MTU_CLK_FALLING_EDGE,	Counter is incremented on falling edge of clock
	MTU_CLK_ANY_EDGE	Counter is incremented on both rising and falling clock edges.

Specify the PWM cycle frequency - **uint32_t** **cycle_freq**

```
    typedef struct mtu_pwm_chnl_settings_s
    {
        mtu_pwm_clk_src_t   clock_src;   // Specify clocking source.
        uint32_t            cycle_freq;  // Cycle frequency for the channel
        mtu_clear_src_t     clear_src;   // Specify the counter clearing source.
        mtu_pwm_mode_t      pwm_mode;    // Specify mode 1 or mode 2
        mtu_pwm_settings_t  pwm_a;
        mtu_pwm_settings_t  pwm_b;
        mtu_pwm_settings_t  pwm_c;
        mtu_pwm_settings_t  pwm_d;
    } mtu_pwm_chnl_settings_t;
```

When Internal clocking is selected

The **cycle_freq** setting is interpreted differently by the driver depending on whether the internal clock source is selected or an external clock source. When the internal clock is selected, then the **cycle_freq** setting is to be the desired frequency in 1Hz units. For example, if you want the pwm to have a 1MHz cycle rate, assign a value of 1000000 to this structure member.

This value will be used by the driver to attempt to provide the closest possible frequency. If the selected frequency is higher than the driver can deliver, then an out of range error will be returned. If the value is in range, then the closest possible frequency that is equal to or lower than the requested frequency will be applied. The accuracy of the resulting frequency chosen by the driver depends on the value requested. This is determined by the base PCLK frequency setting of the RX MCU, and the available clock divisors for the given channel.

External clocking

When an external clock has been chosen as the clock source, no real-time based frequency calculations are done by the driver. In this case the freq setting is interpreted simply as an absolute pwm count value. Be sure to specify a value no greater than 16-bits in size, or 65535 maximum.

Clear source
Pwm operations require that the tick counter be cleared to 0 upon occurrence of the compare match event for the PWM cycle. This is automatic for PWM mode1, but for PWM mode 2 it must be specified. This can be a PWM timer in the same channel or a timer that clears the counter in another channel In PWM mode 2 the timer chosen as the cycle clearing source becomes unavailable for PWM output.

Counter clearing may also be disabled, in which case the counter will continue to increment until it overflows the 16-bit counter register at which point it will start over at 0, however this setting may be of very limited usefulness.

Specify the counter clearing source - **mtu_clear_src_t clear_src**

```
    typedef struct mtu_pwm_chnl_settings_s
    {
        mtu_pwm_clk_src_t   clock_src;   // Specify clocking source.
        uint32_t            cycle_freq;  // Cycle frequency for the channel
        mtu_clear_src_t     clear_src;   // Specify the counter clearing source.
        mtu_pwm_mode_t      pwm_mode;    // Specify mode 1 or mode 2
        mtu_pwm_settings_t  pwm_a;
        mtu_pwm_settings_t  pwm_b;
        mtu_pwm_settings_t  pwm_c;
        mtu_pwm_settings_t  pwm_d;
    } mtu_pwm_chnl_settings_t;
```

clear_src

Type	Choose one from this column	Description
mtu_clear_src_t	MTU_CLR_TIMER_A	Clears counter on the "A" compare event.
	MTU_CLR_TIMER_B	Clears counter on the "B" compare event.
	MTU_CLR_TIMER_C	Clears counter on the "C" compare event.
	MTU_CLR_TIMER_D	Clears counter on the "D" compare event.
	MTU_CLR_SYNC	Clears counter when another sync'ed channel clears.
	MTU_CLR_DISABLED	Never clear the channel counter. Let it continue.

Note: In PWM Mode 2, the same cycle frequency must be set in the synchronized channel performing the cycle clear operation and in all other channels that are depending on that clear source. Otherwise duty settings will not be correctly calculated.

pwm_mode
Under PWM Mode 1 PWM waveforms are output from the MTIOCnA and MTIOCnC pins by pairing TGRA with TGRB and TGRC with TGRD. The levels specified by bits IOA3 to IOA0 and IOC3 to IOC0 in TIOR are output from the MTIOCnA and MTIOCnC pins at compare matches A and C, and the level specified by bits IOB3 to IOB0 and IOD3 to IOD0 in TIOR are output at compare matches B and D. The initial output value is set in TGRA or TGRC

In PWM Mode 2 PWM output is generated using one TGR as the cycle register and the others as duty registers. The level specified in TIOR is output at compare matches. Upon counter clearing by a synchronized register compare match, the initial value set in TIOR is output from each pin. If the values set in the cycle and duty registers are identical, the output value doesnot change even when a compare match occurs. In both PWM mode 1 and PWM mode 2, up to eight phases of PWM waveforms can be output. However the output pins are differently available based on mode.

Specify the PWM mode - **mtu_pwm_mode_t pwm_mode**

```
    typedef struct mtu_pwm_chnl_settings_s
    {
        mtu_pwm_clk_src_t   clock_src;   // Specify clocking source.
        uint32_t            cycle_freq;  // Cycle frequency for the channel
        mtu_clear_src_t     clear_src;   // Specify the counter clearing source.
        mtu_pwm_mode_t      pwm_mode;    // Specify mode 1 or mode 2
        mtu_pwm_settings_t  pwm_a;
        mtu_pwm_settings_t  pwm_b;
        mtu_pwm_settings_t  pwm_c;
        mtu_pwm_settings_t  pwm_d;
    } mtu_pwm_chnl_settings_t;
```

pwm_mode

Type	Choose one from this column	Description
mtu_pwm_mode_t	MTU_PWM_MODE_1	Selects PWM Mode 1 operation
	MTU_PWM_MODE_2	Selects PWM Mode 2 operation

Setting up the individual pwm events.

The MTU2a has up to four pwm general registers (TGRs) per channel, each of which can hold a different 16-bit pwm count value. There is one tick counter per channel. When the counter value increments to a value that matches the value in one of the TGRs a compare/match event occurs. This continues, unless stopped, and eventually all of the TGR values will be matched. Each of these four match events can trigger its own set of actions. In this driver we call these four compare/match settings **pwm_a** through **pwm_d**. Each pwm must be configured to set its match count and to assign the actions to perform on its compare/match event.

Selectable pwm event actions can include; request an **interrupt**, execute the **callback** function, drive an **output** pin, **repeat** the pwm operation indefinitely, or do **nothing**.

The settings choices are, for the most part, identical for all pwms available on a channel. However, not all channels have all four pwms, and some pwms have differing capabilities. Consult the Features table for a list of available pwms and settings they support.

```
typedef struct mtu_pwm_chnl_settings_s
{
    mtu_pwm_clk_src_t   clock_src;   // Specify clocking source.
    uint32_t            cycle_freq;  // Cycle frequency for the channel
    mtu_clear_src_t     clear_src;   // Specify the counter clearing source.
    mtu_pwm_mode_t      pwm_mode;    // Specify mode 1 or mode 2
    mtu_pwm_settings_t  pwm_a;
    mtu_pwm_settings_t  pwm_b;
    mtu_pwm_settings_t  pwm_c;
    mtu_pwm_settings_t  pwm_d;
} mtu_pwm_chnl_settings_t;
```

PWM setting

```
typedef struct mtu_pwm_settings_s
{
    uint16_t             duty;            // Percent duty in 0.1% units
    mtu_actions_t        actions;         // Specify action to perform at event.
    mtu_output_states_t  outputs;         // Specify transition polarities.
} mtu_pwm_settings_t;
```

PWM actions settings

One or more actions must be assigned to every pwm source when configuring a channel.

```
typedef struct mtu_pwm_settings_s
{
    uint16_t             duty;            // Percent duty in 0.1% units
    mtu_actions_t        actions;         // Specify action to perform at event.
    mtu_output_states_t  outputs;         // Specify transition polarities.
} mtu_pwm_settings_t;
```

actions

Type	Choose one or more from this column	Description
mtu_actions_t	MTU_ACTION_OUTPUT	Control state of an output pin.
	MTU_ACTION_INTERRUPT	Generate interrupt request on event.
	MTU_ACTION_CALLBACK	Execute user-defined callback on event [*1] (also generates interrupt).
	MTU_ACTION_TRIGGER_ADC	Trigger ADC on this event. Timer A events only.
	MTU_ACTION_REPEAT	Continuously repeat the pwm cycle and actions
	MTU_ACTION_NONE	Don't use this pwm. [*2]

Note 1: If assigning the MTU_ACTION_CALLBACK action, an interrupt is automatically assigned. It is not necessary to also specify MTU_ACTION_INTERRUPT for the pwm event.

Note 2: Do not combine MTU_ACTION_NONE with other actions for the same pwm event. The result is undefined.

When assigning multiple actions to a pwm event the actions must be combined to a single value by logical 'OR'ing the individual selections.

Example:

```
    mtu_pwm_settings_t    my_pwm_a_config;
    my_pwm_a_config.action = (mtu_actions_t)(MTU_ACTION_OUTPUT | MTU_ACTION_REPEAT);
```

Note: When a combination do_action value is created it should be typecast to (mtu_actions_t) to prevent a compiler warning.

Note: All pwm events on the channel must be configured, even those that will not be used. Unused pwms must have the MTU_ACTION_NONE action assigned. When a pwm has been configured with the MTU_ACTION_NONE action, the rest of that pwm's settings are ignored and so need not be set.

Output pin transitions

If the MTU_ACTION_OUTPUT action is to be used to enable a PWM output pin, then further settings are required to choose the output pin state transition mode.

```
typedef struct mtu_pwm_settings_s
{
    uint16_t             duty;      // Percent duty in 0.1% units
    mtu_actions_t        actions;   // Specify action to perform at event.
    mtu_output_states_t  outputs;   // Specify transition polarities.
} mtu_pwm_settings_t;
```

outputs

Type	Choose one from this column	Description
mtu_output_states_t	MTU_PIN_NO_OUTPUT	Output is high impedance.
	MTU_PIN_LO_GOLO	Initial output is low. Low output at compare match
	MTU_PIN_LO_GOHI	Initial output is low. High output at compare match.
	MTU_PIN_LO_TOGGLE	Initial output is low. Toggle (alternate) output at compare match.
	MTU_PIN_HI_GOLO	Initial output is high. Low output at compare match.
	MTU_PIN_HI_GOHI	Initial output is high. High output at compare match.
	MTU_PIN_HI_TOGGLE	Initial output is high. Toggle (alternate) output at compare match.

Under PWM Mode 1 PWM waveforms are output from the MTIOCnA and MTIOCnC pins by pairing TGRA with TGRB and TGRC with TGRD. The levels specified by bits IOA3 to IOA0 and IOC3 to IOC0 in TIOR are output from the MTIOCnA and MTIOCnC pins at compare matches A and C, and the level specified by bits IOB3 to IOB0 and IOD3 to IOD0 in TIOR are output at compare matches B and D. The initial output value is set in TGRA or TGRC

In PWM Mode 2 PWM output is generated using one TGR as the cycle register and the others as duty registers. The level specified in TIOR is output at compare matches. Upon counter clearing by compare match on the selected clearing source, the initial value set in TIOR is output from each pin.

For the _TOGGLE setting, the output pin will alternately switch to the opposite state at each compare match event. This behavior is probably of limited usefulness for PWM.

Special Notes

In PWM Mode 2, the same cycle frequency must be set in the synchronized channel performing the cycle clear operation and in all other channels that are depending on that clear source. Otherwise duty settings will not be correctly calculated.

6.7 R_MTU_Control()

The Control function is responsible for handling special hardware or software operations for the MTU channel.

6.7.1 R_MTU_Control () Summary

Format

```
   mtu_err_t  R_MTU_Control (mtu_channel_t channel,
                               mtu_cmd_t     cmd,
                               void         *pcmd_data);
```

Parameters

channel

The channel number from enumerated list

cmd

Enumerated command code.

Available command codes:

MTU_CMD_START	*Activate clocking*
MTU_CMD_STOP	Pause clocking
MTU_CMD_SAFE_STOP	Stop clocking and set outputs to safe state
MTU_CMD_RESTART	Zero the counter then resume clocking
MTU_CMD_SYNCHRONIZE	Specify channels to group for synchronized clearing.
MTU_CMD_GET_STATUS	Retrieve the current status of the channel
MTU_CMD_CLEAR_EVENTS	Clears the interrupt flags for the channel
MTU_CMD_SET_CAPT_EDGE	Sets the detection edge polarity for input capture.

pcmd_data

Pointer to the command-data structure parameter of type void that is used to reference the location of any data specific to the command that is needed for its completion. For commands that do not require supporting data use the FIT_NO_PTR argument.

Return Values

MTU_SUCCESS:	*Command successfully completed.*
MTU_ERR_CH_NOT_OPEN:	*The channel has not been opened.*
MTU_ERR_BAD_CHAN:	*Channel number is invalid for part*
MTU_ERR_UNKNOWN_CMD:	*Control command is not recognized.*
MTU_ERR_NULL_PTR:	*A required pcmd_data pointer is NULL or invalid*
MTU_ERR_INVALID_ARG:	*An element of the pcmd_data structure contains an invalid value.*
MTU_ERR_LOCK:	*The lock could not be acquired. The channel is busy.*

Properties

Prototyped in file “r_mtu_rx_if.h”

Description

This function is responsible for handling special hardware or software operations for the MTU channel. It takes an MTU channel number from the enumerated list, an enumerated command value to select the operation to be performed, and if required, a void pointer to a location that contains information or data needed to complete the operation. This pointer must point to storage that has been type-cast by the caller for the particular command using the appropriate type provided in " r_mtu_rx_if.h ". The R_MTU_Control() function requires that the specified channel is already in the “open” state.

Reentrant

Yes if locking is enabled. Reentrancy to operate on the same MTU is rejected by BSP lock. If locking is disabled then reentrancy is only safe for operating on a different channel. If locking is enabled special care should be taken to prevent deadlock. Caller should check return value and should handle a locked return status by permitting the process owning the lock to complete.

Example

```
   my_result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);
```

6.7.2 Description of R_MTU_Control function commands

All R_MTU_Control () commands require that the specified MTU channel has been put into the open state with either a call to R_MTU_Timer_Open() or R_MTU_Capture_Open(). After a channel is opened, it will remain in an idle state (timer clocking is stopped) until a start or restart command is executed. Any command can be executed while the channel is open, however commands should only be used when it makes logical sense to do so. For example, after the start command has been issued, it is useless to issue another start command if it has not been stopped first. It is up to the user to determine which command sequence makes sense for the operation desired. This gives a great deal of flexibility to the types of operations that can be performed.

Control function command codes.

This is a simple enumerated list of the available commands. Select one command for the **cmd** parameter for each call to the R_MTU_Control() function.

```
typedef enum mtu_cmd_e
{
    MTU_CMD_START,          // Activate clocking
    MTU_CMD_STOP,           // Pause clocking
    MTU_CMD_SAFE_STOP,      // Stop clocking. Set outputs to safe state
    MTU_CMD_RESTART,        // Zero the counter then resume clocking
    MTU_CMD_SYNCHRONIZE,    // Channels to group for synchronized clearing.
    MTU_CMD_GET_STATUS,     // Retrieve the current status of the channel
    MTU_CMD_CLEAR_EVENTS,   // Clears the interrupt flags for the channel
    MTU_CMD_UNKNOWN         // Not a valid command.
    MTU_CMD_SET_CAPT_EDGE   // Sets the detection edge polarity for input capture.
} mtu_cmd_t;
```

MTU_CMD_START

This command simply starts the clocking to the MTU channel counter register. The MTU counter begins or resumes counting from whatever value is already in the count register (TCNT).
Either a single channel may be started or a group of channels may be simultaneously started.
See Group command settings
When performing a group start, all channels in the group must be open, however the channel number for any member of the group may be used as the channel argument to the R_MTU_PWM_Control() function.

Examples

```
mtu_err_t  result;

/* Basic start of a single channel. */
result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);

/* Multi-channel Group Start */
mtu_group_t    my_group;

test_group = (mtu_group_t)(MTU_GRP_CH0 | MTU_GRP_CH3 | MTU_GRP_CH4);

result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, &my_group);
```

MTU_CMD_STOP

This command stops the clocking to the MTU channel counter register. The MTU counter immediately stops counting. The value in the count register (TCNT) is preserved, so this is equivalent to a ‘pause’ command. If the channel has been configured for output pin operation, outputs will remain in their present state.

Either a single channel may be stopped or a group of channels may be simultaneously stopped.
See Group command settings

When performing a group stop, all channels in the group must be open, however the channel number for any member of the group may be used as the channel argument to the R_MTU_Control() function.

Examples

```
    mtu_err_t  result;

    /* Basic stop of a single channel. */
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_STOP, FIT_NO_PTR);

    /* Multi-channel Group Start */
    mtu_group_t    my_group;

    my_group = (mtu_group_t)(MTU_GRP_CH0 | MTU_GRP_CH3 | MTU_GRP_CH4);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_STOP, &my_group);
```

Note: If the MTU is stopped while input capture is pending, then the capture event may be missed.

MTU_CMD_SAFE_STOP

This command is similar to the **STOP** command; the clocking of the MTU channel counter register is stopped. However if the channel has been configured for output pin operation, the output pins will be retuned to their 'initial' High or Low state. If a specific High or Low state is considered to be a safe state, then this command may be used for that purpose. This command does not support group operation; only the specified single channel is affected.

Example

```
    mtu_err_t  result;

    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_SAFE_STOP, FIT_NO_PTR);
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_SAFE_STOP, FIT_NO_PTR);
```

MTU_CMD_RESTART

Like the **START** command, the clocking of the MTU channel counter register is started. However the counter (TCNT) value is first cleared to 0. This command can be used while the counter is running to clear 'on-the-fly' or while the channel counter is stopped. This command does not support group operation; only the specified single channel is affected.

Example

```
    mtu_err_t  result;

    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_RESTART, FIT_NO_PTR);
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_RESTART, FIT_NO_PTR);
```

MTU_CMD_SYNCHRONIZE

The **SYNCHRONIZE** command is used to set the members of the group of channels that are to share simultaneous clearing of their counter registers. If a channel in this group is set to clear its counter based on an external clearing source, then when any other channel in the group clears its counter, channels set to "external clear" will also be cleared. Channels in this group that have their clearing source set to something other than external clear will not be affected by clearing on other channels, but will instead act as a clearing source for those channels in the group set for "external clear".

Examples

```
    mtu_err_t    result;
    mtu_group_t  my_group = (mtu_group_t)(MTU_GRP_CH1 | MTU_GRP_CH2);

    /* MTU channels not yet started. Set sync group first. */
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_ SYNCHRONIZE, &my_group);

    /* Now start the group simultaneously. */
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_START, &my_group);
```

MTU_CMD_GET_STATUS

The **GET_STATUS** command retrieves the current status of the channel and places it into the user-provided status command data structure.

Status command Data Structure

The status command returns information about the current state of the MTU channel and the operation in progress. Status information differs depending on whether the channel is configured for **timer** mode (compare/match) operations or for **input capture** mode operations.

Use the **mtu_timer_status_t** data structure with the status command when the channel is configured for **timer** mode operations:

```
    typedef struct mtu_timer_status_s
    {
        uint32_t timer_count;     // The current channel counter value.
        bool     timer_running;   // TRUE: timer is counting. FALSE: counting stopped.
    } mtu_timer_status_t;
```

Use the **mtu_capture_status_t** data structure with the status command when the channel is configured for **input capture** mode operations:

```
    typedef struct mtu_capture_status_s
    {
        uint32_t capt_a_count;    // The count at input capture A event.
        uint32_t capt_b_count;    // The count at input capture B event.
        uint32_t capt_c_count;    // The count at input capture C event.
        uint32_t capt_d_count;    // The count at input capture D event.
        uint32_t timer_count;     // The current channel counter value.
        uint8_t  capture_flags;   // 1 if a capture event occurred, 0 if still waiting.
    } mtu_capture_status_t;
```

Input capture operations may be done without the use of interrupts by polling with the status command. The **capture flags** status member can be monitored to detect an input capture event. The capture event source can be determined by checking which bits are set in the capture flags data.

capture_flags

b7	b6	b5	b4	b3	b2	b1	b0
--	--	--	--	capt D	capt C	capt B	capt A

Bit	Input Capture Source pin	Bit Name	Description
b0	MTIOCnA	capt A	1: Input capture detected 0: Input capture pending
b1	MTIOCnB	capt B	1: Input capture detected 0: Input capture pending
b2	MTIOCnC	capt C	1: Input capture detected 0: Input capture pending
b3	MTIOCnD	capt D	1: Input capture detected 0: Input capture pending
b7 to b4	--	--	Not used.

MTU_CMD_CLEAR_EVENTS

The **CLEAR_EVENTS** command clears the interrupt flags for the channel. When the channel configuration is set to not use interrupts, interrupt requests are still sent to the Interrupt Controller Unit (ICU), however the interrupt is not processed. This is done so that there is a means to poll for input capture events when system interrupts are not desired. After detection of an input capture event, the interrupt flag will be set and the MTU_CMD_GET_STATUS command can read it. Afterwards, another input capture event cannot be detected until this flag is cleared. Use this command under these circumstances to clear the detection flag in preparation for another input capture.

Examples: Polling status for capture event and then clearing for next event

```
    mtu_err_t       result;
    mtu_capture_status_t  my_capture_status = {0};
    uint32_t              capture_a_result;

    /* Start the capture process. (channel was already initialized for capture)*/
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_START, FIT_NO_PTR);

    /* Now poll for the capture event on capture A. */
    do
    {
        result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_GET_STATUS, &my_capture_status);
    } while(!(my_capture_status.capture_flags & 0x01)); //Check event bit

    /* We got the event on input A, so now the capture counter data is valid. */
    capture_a_result = my_capture_status.capt_a_count;

    /* Clear the event flags for next capture event. */
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_CLEAR_EVENTS, FIT_NO_PTR);
    ...
```

MTU_CMD_SET_CAPT_EDGE

Sets the detection edge polarity for input capture for a single capture source. This is useful for changing the detection edge polarity without having to first close then reopen the entire channel. The most typical use case is for measurement of a pulse width of a specific polarity. After the first edge is captured use this command to set the opposite edge for the next detection.

Use the **mtu_capture_set_edge_t** data structure with this command when the channel is configured for **capture** mode operations.

```
    typedef struct mtu_capture_set_edge_s
    {
        mtu_cap_src_t    capture_src;        // The capture source.
        mtu_cap_edges_t  capture_edge;       // Specify transition polarities.
    } mtu_capture_set_edge_t;
```

Set **capture_src** and **capture_edge** from the following choices:

capture_src

Type	Choose one from this column	Description
mtu_cap_edges_t	MTU_CAP_SRC_A	Input capture source A (MTIOCnA)
	MTU_CAP_SRC_B	Input capture source B (MTIOCnB)
	MTU_CAP_SRC_C	Input capture source C (MTIOCnC)
	MTU_CAP_SRC_D	Input capture source D (MTIOCnD)

capture_edge

Type	Choose one from this column	Description
mtu_cap_edges_t	MTU_CAP_RISING_EDGE	Only rising edges trigger input capture event.
	MTU_CAP_FALLING_EDGE	Only falling edges trigger input capture event.
	MTU_CAP_ANY_EDGE	Both rising and falling edges trigger input capture event.

Note: Do not use this command on a channel that is configured for Timer mode (compare/match).

Group command settings.

The START, and STOP commands can optionally accept a **group** specifier, and the SYNCHRONIZE command requires it.

```
    typedef enum
    {
        MTU_GRP_CH0 = 0x01,
        MTU_GRP_CH1 = 0x02,
        MTU_GRP_CH2 = 0x04,
        MTU_GRP_CH3 = 0x40,
        MTU_GRP_CH4 = 0x80,
    } mtu_group_t;
```

Two or more channels may be included in the group so that the Control command will be applied simultaneously to all channels in the group. To add channels to the group the group numbers for each channel must be OR'ed together so that they form a single value.

Example:

```
    mtu_group_t my_channel_group =
            (mtu_group_t)(MTU_GRP_CH0 | MTU_GRP_CH3 | MTU_GRP_CH4);
```

Notes:

1. The group should be typecast to (mtu_group_t) when defined to prevent a compiler warning.
2. All channels being assigned to a group must already have been opened or this command will return an MTU_ERR_INVALID_ARG error.
3. Group settings for a channel remain in effect even after a the channel has been closed. Always perform a new group command to update the group setting when closing channels that were in the group.

6.8 R_MTU_Close()

Disables the specified MTU channel and returns it to 'available' status.

Format

```
   mtu_err_t  R_MTU_Close(mtu_channel_t  channel);
```

Parameters

channel

The channel number

Return Values

MTU_SUCCESS:	*Successful; channel closed*
MTU_ERR_CH_NOT_OPEN:	*The channel has not been opened so closing has no effect.*
MTU_ERR_BAD_CHAN:	*Channel number is not available*

Properties

Prototyped in file "r_mtu_rx_if.h"

Description

Disables the specified MTU channel and returns it to 'available' status. This function first checks to see if the channel is currently initialized. Then it stops any current operation. All associated interrupts are disabled. If the specified channel is the last channel in use, then after stopping this channel the MTU2a unit is powered down.

Reentrant

Yes for other channels. Reentrance for same MTU channel may result in an MTU_ERR_ NOT_OPEN return code.

Example

```
    mtu_err_t my_result;

    my_result = R_MTU_Close (MTU_CHANNEL_1);

    if (MTU_SUCCESS != my_result)
    {
       return my_result;
    }
    else
    {
        ....
    }
```

6.9 R_MTU_GetVersion()

This function returns the driver version number at runtime.

Format

```
   uint32_t  R_MTU_GetVersion(void);
```

Parameters

None

Return Values

Version number with major and minor version digits packed into a single 32-bit value.

Properties

Prototyped in file “r_mtu_rx_if.h”

Description

The function returns the version of this module. The version number is encoded such that the top two bytes are the major version number and the bottom two bytes are the minor version number. For example, Rev 4.25 would be 0x00040019.

Reentrant

Yes

Example

Example showing this function being used.

```
/* Retrieve the version number and convert it to a string. */

uint32_t   version, version_high, version_low;
char       version_str[18];

version = R_MTU_GetVersion ();

version_high = (version >> 16)&0xf;
version_low  =  version & 0xff;

sprintf(version_str, "MTU Timers v%1.1hu.%2.2hu", version_high, version_low);
```

7. Application Code Examples

```
/******************************************************************************
* Function Name : compare_match_timer_outputs_transition
* Description   : Demonstrate output of a single transition on outputs A and B
*                 Of Channel 0. Timers C and D not used.
* Arguments     : none
* Return value  : API call result
******************************************************************************/

mtu_err_t compare_match_timer_outputs_transition(void)
{
    mtu_timer_chnl_settings_t  my_timer_cfg;
    mtu_err_t  result;

    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.actions.do_action = MTU_ACTION_OUTPUT;
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_LO_GOHI;
    my_timer_cfg.timer_a.freq              = 1000; //1KHz
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_OUTPUT;
    my_timer_cfg.timer_b.actions.output    = MTU_PIN_LO_GOHI;
    my_timer_cfg.timer_b.freq              = 2000; //2KHz
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    result = R_MTU_Timer_Open(MTU_CHANNEL_0, &my_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);

    return result;
}

/******************************************************************************
* Function Name : compare_match_timer_outputs_toggle
* Description   : Demonstrates toggle output setting to output a repeating 50%
*                 duty pulse stream with 2 phases on outputs A and B of Channel 0.
*                 Timers C and D not used.
* Arguments     : none
* Return value  : API call result
******************************************************************************/
mtu_err_t  compare_match_timer_outputs_toggle(void)
{
    mtu_timer_chnl_settings_t  my_timer_cfg;
    mtu_err_t  result;

    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.actions.do_action = (mtu_actions_t)(MTU_ACTION_OUTPUT |
                                                             MTU_ACTION_REPEAT);
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_a.freq              = 10000; //10KHz
    my_timer_cfg.timer_b.actions.do_action = (mtu_actions_t)(MTU_ACTION_OUTPUT );
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_b.freq              = 20000; //20KHz
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    result = R_MTU_Timer_Open(MTU_CHANNEL_0, &my_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);
    return result;
}
```

```
/******************************************************************************
* Function Name : my_callback
* Description   : called by interrupt at occurrance of timer event.
* Arguments     : pdata – pointer to callback data structure.
* Return value  :
******************************************************************************/
void my_callback(void * pdata)
{
    mtu_callback_data_t * cb_data = (mtu_callback_data_t *)pdata;
    mtu_callback_data_t my_cb_data;

    my_cb_data.channel   = cb_data->channel;
    my_cb_data.count     = cb_data->count;
    my_cb_data.timer_num = cb_data->timer_num;

    if (MTU_TIMER_A == my_cb_data.timer_num) // wait for the slowest timer to finish.
    {
        g_my_callback_called = true;
    }
}

/******************************************************************************
* Function Name : compare_match_timer_callback
* Description   : Demonstrate calling the user defined callback on compare match
*                 of Channel 1 timers A and B. Timers C and D not used.
* Arguments     : none
* Return value  : API call result
******************************************************************************/
mtu_err_t  compare_match_timer_callback(void)
{
    mtu_timer_chnl_settings_t  my_timer_cfg;
    mtu_err_t  result;

    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.actions.do_action = MTU_ACTION_CALLBACK;
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_NO_OUTPUT;
    my_timer_cfg.timer_a.freq              = 1000;
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_CALLBACK;
    my_timer_cfg.timer_b.actions.output    = MTU_PIN_NO_OUTPUT
    my_timer_cfg.timer_b.freq              = 2000;
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    result = R_MTU_Timer_Open(MTU_CHANNEL_1, &my_timer_cfg, my_callback);

    g_my_callback_called = false;

    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_START, FIT_NO_PTR);
    while (!g_my_callback_called)
    {
        nop(); // Wait for the callback to execute.
    }
    return result;
}

/******************************************************************************
* Function Name : compare_match_timer_interrupt
* Description   : Demonstrate triggering interrupt requests on compare/match
*                 of Channel 2 timer A. Timers B, C, and D not used.
*                 Could be used to supply timed interrupts to drive DTC.
* Arguments     : none
* Return value  : API call result
******************************************************************************/
mtu_err_t  compare_match_timer_interrupt(void)
{
    mtu_err_t  result;
    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.actions.do_action = MTU_ACTION_INTERRUPT;
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_NO_OUTPUT;
    my_timer_cfg.timer_a.freq              = 100000; // 100 KHz.
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    result = R_MTU_Timer_Open(MTU_CHANNEL_2, &my_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_2, MTU_CMD_START, FIT_NO_PTR);
    return result;
}


/******************************************************************************
* Function Name : input_capture_clock_period
* Description   : 1) Demonstrate input capture on channel 0 capture source A.
*                    capture sources B, C, and D not used.
*                 2) No interrupt enabled. Test for capture event by polling with
*                    status command.
*                 3) Wait for first edge then capture again on second edge to measure
*                    interval between edges.
*                 4) Use channel 1 timer A to generate a signal fed to channel 0
*                    capture input pin.
*                 5) Start channel 0 and channel 1 simultaneously using group command.
* Arguments     : none
* Return value  : the period measurement
******************************************************************************/
uint16_t input_capture_clock_period(void)
{
    mtu_err_t  result;
    mtu_group_t    my_group;
    mtu_capture_status_t my_capture_status = {0};
    uint16_t captured_period;

    my_group = (mtu_group_t)(MTU_GRP_CH0 | MTU_GRP_CH1);

    /* Set up the signal source timer. */
    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.actions.do_action = (mtu_actions_t)(MTU_ACTION_OUTPUT |
                                                 MTU_ACTION_REPEAT);
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_a.freq              = 20000;
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    /* Set up the input capture channel. */
    my_capture_cfg.clock_src.source     = MTU_CLK_SRC_INTERNAL;
    my_capture_cfg.clock_src.clock_edge = MTU_CLK_RISING_EDGE;
    my_capture_cfg.clock_div            = MTU_SRC_CLK_DIV_1;
    my_capture_cfg.clear_src            = MTU_CLR_TIMER_A;

    my_capture_cfg.capture_a.capture_edge = MTU_CAP_ANY_EDGE;
    my_capture_cfg.capture_a.actions = (MTU_ACTION_CAPTURE | MTU_ACTION_REPEAT);
    my_capture_cfg.capture_b.actions = MTU_ACTION_NONE;
    my_capture_cfg.capture_c.actions = MTU_ACTION_NONE;
    my_capture_cfg.capture_d.actions = MTU_ACTION_NONE;

    /* Open the timer and capture channels. */
    result = R_MTU_Timer_Open(MTU_CHANNEL_1, &my_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Capture_Open(MTU_CHANNEL_0, &my_capture_cfg, my_capture_callback);
    /* Issue Start command on the group to start both timers simultaneously. */
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, &my_group);

    /* Wait until first edge is captured. */
    do
    {
        result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_GET_STATUS, &my_capture_status);
    } while(!(my_capture_status.capture_flags));

    /* Got the first edge now clear event flag so we can poll for the next edge. */
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_CLEAR_EVENTS, FIT_NO_PTR);

    /* Wait until second edge is captured. */
    do
    {
        result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_GET_STATUS, &my_capture_status);
    } while(!(my_capture_status.capture_flags));

    /* capture status now contains the captured measurement data. */
    captured_period = my_capture_status. capt_a_count;
    return captured_period;
}

/*******************************************************************************
* Function Name : compare_match_timer_sync
* Description   : 1) Demonstrates output of 4 repeating 50% duty pulse streams
*                    with phase offsets on outputs A, B, C and D of Channel 0.
*                 2)Frequency is controlled by timer on channel 1.
*                 3)Channels 0 and 1 are synchronized.
*                 4)Counter clearing on channel 1 also clears channel 0.
* Arguments     : none
* Return value  : API call result
*******************************************************************************/
mtu_err_t  compare_match_timer_sync(void)
{
    mtu_err_t  result;
    mtu_timer_chnl_settings_t sync_timer_cfg;
    mtu_group_t    my_group;

    my_group = (mtu_group_t)(MTU_GRP_CH0 | MTU_GRP_CH1);

    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_SYNC;
    my_timer_cfg.timer_a.actions.do_action = MTU_ACTION_OUTPUT;
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_a.freq              = 10000;
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_OUTPUT;
    my_timer_cfg.timer_b.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_b.freq              = 20000;
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_OUTPUT;
    my_timer_cfg.timer_c.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_c.freq              = 30000;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_OUTPUT;
    my_timer_cfg.timer_d.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_d.freq              = 40000;

    sync_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    sync_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    sync_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    sync_timer_cfg.timer_a.actions.do_action = MTU_ACTION_REPEAT;
    sync_timer_cfg.timer_a.actions.output    = MTU_PIN_NO_OUTPUT;
    sync_timer_cfg.timer_a.freq              = 8000;
    sync_timer_cfg.timer_b.actions.do_action = MTU_ACTION_NONE;
    sync_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    sync_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    result = R_MTU_Timer_Open(MTU_CHANNEL_0, &my_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Timer_Open(MTU_CHANNEL_1, &sync_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_SYNCHRONIZE, &my_group);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, &my_group);
}

/*****************************************************************************
* Function Name : compare_match_timer_ext_clock
* Description   : 1) Demonstrates counter clocking by external clock source.
                  2) Outputs a repeating pulse stream on output A of Channel 1.
*                 2) Channel 0 is clocked by external input on MTCLKB pin.
*                 3) Outputs a repeating pulse stream on output A of Channel 0
*                    when clocked by jumpering channel 1 output to MTCLKB pin.
* Arguments     : none
* Return value  : API call result
*****************************************************************************/
mtu_err_t  compare_match_timer_ext_clock(void)
{
    mtu_err_t  result;
    mtu_timer_chnl_settings_t src_clk_timer_cfg;
    mpc_config_t pin_cfg;

    src_clk_timer_cfg.clock_src.source          = MTU_CLK_SRC_INTERNAL;
    src_clk_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    src_clk_timer_cfg.clear_src                 = MTU_CLR_TIMER_B;
    src_clk_timer_cfg.timer_b.freq              = 1000; // 1000Hz = 1ms.
    src_clk_timer_cfg.timer_b.actions.do_action = (mtu_actions_t)(MTU_ACTION_OUTPUT |
                                                                   MTU_ACTION_REPEAT);
    src_clk_timer_cfg.timer_b.actions.output    = MTU_PIN_HI_TOGGLE;
    src_clk_timer_cfg.timer_a.actions.do_action = MTU_ACTION_NONE;
    src_clk_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    src_clk_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

    my_timer_cfg.clock_src.source          = MTU_CLK_SRC_EXT_MTCLKB;
    my_timer_cfg.clock_src.clock_edge      = MTU_CLK_RISING_EDGE;
    my_timer_cfg.clear_src                 = MTU_CLR_TIMER_A;
    my_timer_cfg.timer_a.freq              = 1000; // 1000 ticks =  1 sec.
    my_timer_cfg.timer_a.actions.do_action = (mtu_actions_t)(MTU_ACTION_OUTPUT |
                                                              MTU_ACTION_REPEAT);
    my_timer_cfg.timer_a.actions.output    = MTU_PIN_LO_TOGGLE;
    my_timer_cfg.timer_b.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_c.actions.do_action = MTU_ACTION_NONE;
    my_timer_cfg.timer_d.actions.do_action = MTU_ACTION_NONE;

#if defined BSP_BOARD_RSKRX111 || defined BSP_BOARD_RSKRX110
    /* MTIOC3A PMOD1-1, J2-11 */
    R_GPIO_PinControl (GPIO_PORT_C_PIN_7, GPIO_CMD_ASSIGN_TO_PERIPHERAL);
    pin_cfg.pin_function = 0x02; // set as clock input MTCLKB
    R_MPC_Write( GPIO_PORT_C_PIN_7, &pin_cfg);
#endif

    result = R_MTU_Timer_Open(MTU_CHANNEL_0, &my_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Timer_Open(MTU_CHANNEL_1, &src_clk_timer_cfg, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);
    result = R_MTU_Control(MTU_CHANNEL_1, MTU_CMD_START, FIT_NO_PTR);
}
```

```
/******************************************************************************
* Function Name : pwm_mode1_simple
* Description   : 1) Demonstrates basic PWM mode 1 operation.
*                 2) Outputs a 33.3% duty PWM pulse stream on output A of Channel 0
*                    with a cycle frequency of 20KHz.
* Arguments     : none
* Return value  : API call result
******************************************************************************/
mtu_err_t  pwm_mode1_simple (void)
{
    mtu_err_t  result = MTU_SUCCESS;
    mtu_pwm_chnl_settings_t simple_pwm_cfg;

    ex_clk_pwm_cfg.pwm_mode               = MTU_PWM_MODE_1;
    ex_clk_pwm_cfg.clock_src.source       = MTU_CLK_SRC_INTERNAL;
    ex_clk_pwm_cfg.clock_src.clock_edge = MTU_CLK_RISING_EDGE;
    ex_clk_pwm_cfg.clear_src              = MTU_CLR_TIMER_A;
    ex_clk_pwm_cfg.cycle_freq             = 20000;  // 20KHz cycle frequency.

    ex_clk_pwm_cfg.pwm_a.actions = (mtu_actions_t)(MTU_ACTION_OUTPUT | MTU_ACTION_REPEAT);
    ex_clk_pwm_cfg.pwm_a.outputs = MTU_PIN_HI_GOHI;

    ex_clk_pwm_cfg.pwm_b.duty    = 333;        // 33.3% duty.
    ex_clk_pwm_cfg.pwm_b.actions = MTU_ACTION_OUTPUT;
    ex_clk_pwm_cfg.pwm_b.outputs = MTU_PIN_HI_GOLO;

    ex_clk_pwm_cfg.pwm_c.actions = MTU_ACTION_NONE;
    ex_clk_pwm_cfg.pwm_d.actions = MTU_ACTION_NONE;

    /* Call open function to initialize the MTU channel for PWM. */
    result = R_MTU_PWM_Open(MTU_CHANNEL_0, & ex_clk_pwm_cfg, FIT_NO_FUNC);

    if ((MTU_SUCCESS != result)
    {
        return result;
    }

    /* Call control function to start the MTU counter and begin outputting PWM. */
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);

    return result;
}
```

```
/*******************************************************************************
* Function Name : pwm_mode2_sync
* Description   : 1) Demonstrates PWM mode 2 multiphase output with synchronous
*                    clearing.
                  2) Outputs 4 phases of PWM on channel 0.
*                 3) Channel 0 is clocked by external input on MTCLKB pin.
*                 4) Outputs a 20% duty pulse stream on output A of Channel 0.
*                    Outputs a 40% duty pulse stream on output B of Channel 0.
*                    Outputs a 60% duty pulse stream on output C of Channel 0.
*                    Outputs a 80% duty pulse stream on output D of Channel 0.
*                 5) Cycle frequency is controlled by timer A of Channel 1.
* Arguments     : none
* Return value  : API call result
*******************************************************************************/
mtu_err_t  pwm_mode2_sync (void)
{
    mtu_err_t  result;
    mtu_pwm_chnl_settings_t output_pwm_settings;
    mtu_pwm_chnl_settings_t sync_pwm_settings;
    mtu_group_t     my_group;

    my_group = (mtu_group_t)(MTU_GRP_CH0 | MTU_GRP_CH1);

    output_pwm_settings.clock_src.source      = MTU_CLK_SRC_INTERNAL;
    output_pwm_settings.clock_src.clock_edge  = MTU_CLK_RISING_EDGE;
    output_pwm_settings.clear_src             = MTU_CLR_SYNC;
    output_pwm_settings.cycle_freq            = 10000; // 10KHz cycle frequency.
    output_pwm_settings.pwm_mode              = MTU_PWM_MODE_2;

    output_pwm_settings.pwm_a.duty            = 200; // 20.0% duty.
    output_pwm_settings.pwm_a.actions         = MTU_ACTION_OUTPUT;
    output_pwm_settings.pwm_a.outputs         = MTU_PIN_LO_GOHI;

    output_pwm_settings.pwm_b.duty            = 400; // 40.0% duty.
    output_pwm_settings.pwm_b.actions         = MTU_ACTION_OUTPUT;
    output_pwm_settings.pwm_b.outputs         = MTU_PIN_LO_GOHI;

    output_pwm_settings.pwm_c.duty            = 600; // 60.0% duty.
    output_pwm_settings.pwm_c.actions         = MTU_ACTION_OUTPUT;
    output_pwm_settings.pwm_c.outputs         = MTU_PIN_LO_GOHI;

    output_pwm_settings.pwm_d.duty            = 800; // 80.0% duty.
    output_pwm_settings.pwm_d.actions         = MTU_ACTION_OUTPUT;
    output_pwm_settings.pwm_d.outputs         = MTU_PIN_LO_GOHI;

    sync_pwm_settings.clock_src.source      = MTU_CLK_SRC_INTERNAL;
    sync_pwm_settings.clock_src.clock_edge  = MTU_CLK_RISING_EDGE;
    sync_pwm_settings.clear_src             = MTU_CLR_TIMER_A;
    sync_pwm_settings.pwm_mode              = MTU_PWM_MODE_1;
    sync_pwm_settings.cycle_freq            = 10000; // 10KHz cycle frequency
    sync_pwm_settings.pwm_a.actions         = MTU_ACTION_REPEAT;
    sync_pwm_settings.pwm_a.outputs         = MTU_PIN_NO_OUTPUT;

    sync_pwm_settings.pwm_b.actions = MTU_ACTION_NONE;
    sync_pwm_settings.pwm_c.actions = MTU_ACTION_NONE;
    sync_pwm_settings.pwm_d.actions = MTU_ACTION_NONE;

    result = R_MTU_PWM_Open(MTU_CHANNEL_0, &output_pwm_settings, FIT_NO_FUNC);
    result = R_MTU_PWM_Open(MTU_CHANNEL_1, &sync_pwm_settings, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_SYNCHRONIZE, &my_group);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, &my_group);
}
```

```
/*******************************************************************************
* Function Name : pwm_mode1_ext_clock
* Description   : 1) Demonstrates PWM counter clocking by external clock source.
*                 2) Outputs a 50% duty PWM pulse stream on output A of Channel 2.
*                    to generate the external clock. Jumper this output to MTCLKB pin.
*                 3) Channel 0 is clocked by external input on MTCLKB pin.
*                 4) Outputs a 60% duty pulse stream on output A of Channel 0
*                    and a 25% duty pulse stream on output C of Channel 0
*                    when clocked by jumpering channel 2 output to MTCLKB pin.
*                 5) Channel 0 cycle frequency is expressed in terms of tick count.
* Arguments     : none
* Return value  : API call result
*   Notes: Uses calls to FIT GPIO and MPC modules to set up MTCLKB pin
*******************************************************************************/
mtu_err_t  pwm_mode1_ext_clock (void)
{
    mtu_err_t  result;
    mtu_pwm_chnl_settings_t ex_clk_pwm_cfg;
    mtu_pwm_chnl_settings_t clk_src_pwm_cfg;
    mpc_config_t pin_cfg;

    ex_clk_pwm_cfg.pwm_mode               = MTU_PWM_MODE_1;
    ex_clk_pwm_cfg.clock_src.source       = MTU_CLK_SRC_EXT_MTCLKB;
    ex_clk_pwm_cfg.clock_src.clock_edge = MTU_CLK_RISING_EDGE;
    ex_clk_pwm_cfg.clear_src              = MTU_CLR_TIMER_A;
    ex_clk_pwm_cfg.cycle_freq             = 100;  // Ext clock so this is tick count.
    ex_clk_pwm_cfg.pwm_a.actions = (mtu_actions_t)(MTU_ACTION_OUTPUT | MTU_ACTION_REPEAT);
    ex_clk_pwm_cfg.pwm_a.outputs = MTU_PIN_HI_GOHI;
    ex_clk_pwm_cfg.pwm_b.duty    = 600;        // 60.0% duty.
    ex_clk_pwm_cfg.pwm_b.actions = MTU_ACTION_OUTPUT;
    ex_clk_pwm_cfg.pwm_b.outputs = MTU_PIN_HI_GOLO;
    ex_clk_pwm_cfg.pwm_c.actions = MTU_ACTION_OUTPUT;
    ex_clk_pwm_cfg.pwm_c.outputs = MTU_PIN_LO_GOLO;
    ex_clk_pwm_cfg.pwm_d.duty    = 250;        // 25.0% duty.
    ex_clk_pwm_cfg.pwm_d.actions = MTU_ACTION_OUTPUT;
    ex_clk_pwm_cfg.pwm_d.outputs = MTU_PIN_LO_GOHI;

    src_clk_pwm_cfg.pwm_mode               = MTU_PWM_MODE_1;
    src_clk_pwm_cfg.clock_src.source       = MTU_CLK_SRC_INTERNAL;
    src_clk_pwm_cfg.clock_src.clock_edge = MTU_CLK_RISING_EDGE;
    src_clk_pwm_cfg.clear_src              = MTU_CLR_TIMER_A;
    src_clk_pwm_cfg.cycle_freq             = 20000;  // Int clock, so this is frequency.
    src_clk_pwm_cfg.pwm_a.actions = (mtu_actions_t)(MTU_ACTION_OUTPUT | MTU_ACTION_REPEAT);
    src_clk_pwm_cfg.pwm_a.outputs = MTU_PIN_HI_GOHI;
    src_clk_pwm_cfg.pwm_b.duty    = 500;       // 50.0% duty.
    src_clk_pwm_cfg.pwm_b.actions = MTU_ACTION_OUTPUT;
    src_clk_pwm_cfg.pwm_b.outputs = MTU_PIN_HI_GOLO;
    src_clk_pwm_cfg.pwm_c.actions = MTU_ACTION_NONE;
    src_clk_pwm_cfg.pwm_d.actions = MTU_ACTION_NONE;

#if defined BSP_BOARD_RSKRX111 || defined BSP_BOARD_RSKRX110
    /* MTIOC3A PMOD1-1, J2-11 */
    R_GPIO_PinControl (GPIO_PORT_C_PIN_7, GPIO_CMD_ASSIGN_TO_PERIPHERAL);
    pin_cfg.pin_function = 0x02; // set as clock input MTCLKB
    R_MPC_Write( GPIO_PORT_C_PIN_7, &pin_cfg);
#endif

    result = R_MTU_PWM_Open(MTU_CHANNEL_0, & ex_clk_pwm_cfg, FIT_NO_FUNC);
    result = R_MTU_PWM_Open(MTU_CHANNEL_2, & src_clk_pwm_cfg, FIT_NO_FUNC);
    result = R_MTU_Control(MTU_CHANNEL_0, MTU_CMD_START, FIT_NO_PTR);
    result = R_MTU_Control(MTU_CHANNEL_2, MTU_CMD_START, FIT_NO_PTR);
}
```

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/contact/

Revision History

		Description	
Rev.	Date	Page	Summary
1.00	Sept. 30, 2014		First release

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 document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accordance 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 a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

RENESAS

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.
2801 Scott Boulevard Santa Clara, CA 95050-2549, 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: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

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: +852-2265-6688, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

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.
12F., 234 Teheran-ro, Gangnam-Ku, Seoul, 135-920, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141