RH850/U2B Group

How to implement FreeRTOS

Introduction

This application note introduces how to implement FreeRTOS, and describes the procedure for using sample programs.

Target Device

RH850/U2B Group Microcontrollers

Operation Confirmed Device

RH850/U2B-FCC (U2B24 mode) Microcontrollers

CAUTION

There is no guarantee to update in this document and software to reflect the latest manual, errata, technical update and development environment. You are fully responsible for the incorporation or any other use of the information of this document in the design of your product or system, and please refer to latest manual, errata, technical update and development environment.

Reference Document

RH850/U2B Group User's Manual: Hardware (R01UH0923EJxxxx)

FreeRTOS official site: ***https://www.freertos.org***

Abbreviations

Symbol	Description
RTOS	Real time operating system
API	Application programming interface
ISR	Interrupt Service Routine
MemMang	Memory management

Table of Contents

Section 1 Overview

1.1 Overview of this application note

This application note explains how to implement FreeRTOS on RH850/U2B group products and the operation of the sample programs using FreeRTOS attached to this application note.

Section 2 describes instruction on how to implement FreeRTOS on RH850/U2B group products.

CAUTION

The driver program for using FreeRTOS described in Section 2.2 only implements the minimum required to use FreeRTOS.

When creating an application using FreeRTOS, please carefully consider your system configuration and consider and create driver program.

The driver program in the sample program has at least the following limitations:

- Does not support multiple interrupts
- Does not support FE level interrupts

Section 3 describes the sample programs attached to this application note to explain how to use FreeRTOS. The operation of the sample programs has been confirmed in the environment shown in **Table 1-1**.

Table 1-1 Operation confirmation environment

Category	Item	Description
Software	IDE	CS+ for CC V8.10.00 [06 Jun 2023]
	FreeRTOS	FreeRTOS 202212.01
Hardware	Debug tool	E2 Emulator (RTE0T00020KCE00000R)
	Evaluation board	RH850/U2B Piggyback Board BGA 468-pin (Y-RH850-U2B-468PIN-PB-T1-V1)
	Logic analyzer	ZEROPLUS LAP-C Pro (16064M)

Section 2 How to port FreeRTOS

This section describes how to download and setup the FreeRTOS source code, and how to use the needed the hardware resources in RH850/U2B group products to implement FreeRTOS.

2.1 Preparation before installing FreeRTOS

2.1.1 Downloading FreeRTOS for Porting

The FreeRTOS kernel and other FreeRTOS libraries are distributed under the MIT open-source license.

The latest version of FreeRTOS source code can be downloaded from the FreeRTOS official website below.

https://www.freertos.org/a00104.html

This application note uses the "FreeRTOS 202212.01" shown in the red frame in **Figure 2-1**, which contains sample programs for various products.

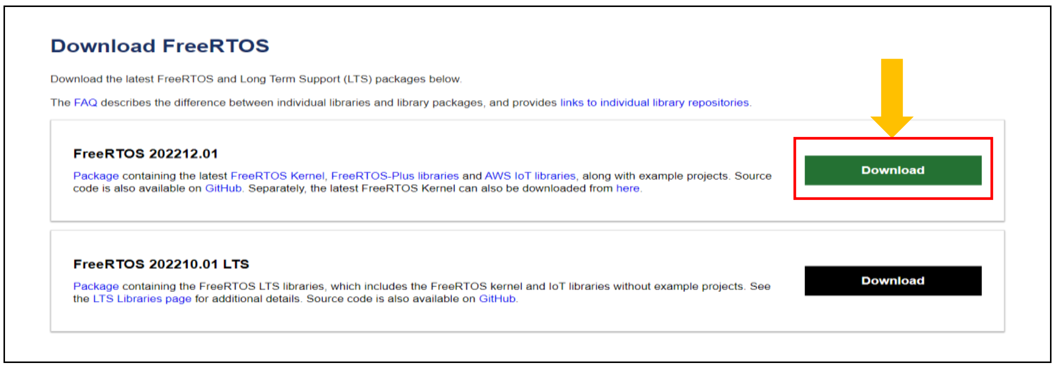


Figure 2-1 FreeRTOS Source download from FreeRTOS official website

2.1.2 Folder structure

The application note uses the "FreeRTOS 202212.01.zip" downloaded from FreeRTOS official site to perform porting to project. Below describes this downloaded zip file description.

After downloading and unzipping, the folder “FreeRTOS 202212.01” will be created.

Figure 2-2 shows the folder structure, in which there are two main subfolders under “FreeRTOSv202212.01” and “FreeRTOS” and “FreeRTOS-Plus”.

- The folder "FreeRTOS" contains the kernel source and its demo projects.
- The folder "FreeRTOS-Plus" contains components of FreeRTOS-Plus-TCP, FreeRTOS-Plus-CLI, etc. and their demo projects.

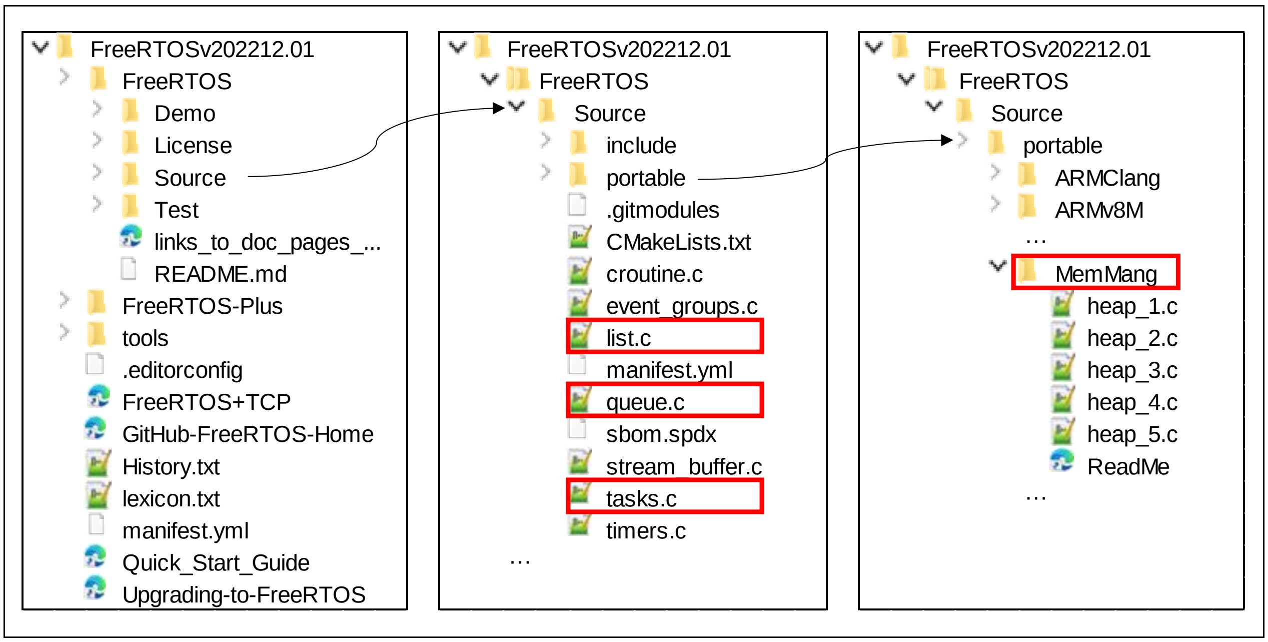


Figure 2-2 The folder structure of “FreeRTOS 202212.01.zip”

The source code is in “FreeRTOS/Source”, this folder contains several files. There are 3 core files in the red box in **Figure 2-2**: task.c, queue.c and list.c contain Task Management functions for the RTOS scheduling system.

Another important folder is the "MemMang" folder located in "FreeRTOS/Source/portable". This folder contains codes for managing memory when using FreeRTOS features.

2.1.3 Project organization

This section describes the configuration of the sample project “CSP_Projects_U2B_RTOS”.

(1) platform\third-party

Figure 2-3 shows the folder structure of “platform\third-party” in the sample projects “CSP_Projects_U2B_RTOS”. The folder “third-party” is import to the category “third-party” in each CS+ projects.

The files contained in this folder are FreeRTOS related files. "Copy from" in **Figure 2-3** indicates the downloaded FreeRTOS folder. Among these, the "(New create)" file is a file created for the RH850/U2B product. For details about these files, refer to **Section 2.2**.

Folder structure of CSP_Projects_U2B_RTOS	Copy from	Description
CSP_Projects_U2B_RTOS	-	-
platform	-	-
third-party	-	-
FreeRTOS	-	-
core	-	-
include	-	-
atomic.h	Source\include	Atomic functions by disabling interrupts globally
croutine.h		The macro implementation of the co-routine functionality
deprecated_definitions.h		The correct portmacro.h file for the port being used
event_groups.h		Definitions of event group function
FreeRTOS.h		The generic headers required for the FreeRTOS port being used
list.h		The list implementation used by the scheduler
message_buffer.h		Message buffers build functionality on top of FreeRTOS stream buffers
mpu_prototypes.h		Definitions the standard API functions
mpu_wrappers.h		API functions to be called through a wrapper macro
portable.h		Definitions of the portable layer API
projdefs.h		The prototype definition that the task functions must conform
queue.h		Definitions of queue function
semphr.h		Definitions of semaphore function
StackMacros.h		Include stack_macros.h
stack_macros.h		Macros to check the current stack state only
stream_buffer.h		Definitions of stream buffer function
task.h		Definitions of task function
timers.h		Definitions of software timer function
portable	-	-
MemMang	-	-
heap_4.c	Source\portable\MemMang	The file added to use dynamic memory allocation
Renesas_U2B24	(New create)	-
portasm.asm		Interrupt handlers related to switch context
porting.c		Functions of stack initalize and scheduler
portmacro.h		FreeRTOS correctly for the given hardware and compiler
contextop.h		macro of context switching
croutine.c	Source	The FreeRTOS co-routine functionality
event_groups.c		Event group functionality
list.c		Scheduling management functions
queue.c		Both queue and semaphore services
stream_buffer.c		Stream buffer functionality
tasks.c		Task services
timers.c		Software timer functionality

Figure 2-3 The folder structure of “platform\third-party” in sample projects

(2) platform\drivers

Figure 2-4 shows the folder structure of "platform\drivers" in the sample projects "CSP_Projects_U2B_RTOS". The folder "drivers" is import to the category "drivers" in each CS+ projects.

The file of "U2B_Driver_Lib.lib" in the folder "library" contains the body of the function whose prototype is declared in each header file included in the include folder.

The functions included in "library" are basic settings for U2B clock functions, port functions, and peripheral functions, so the source code is not included in this application note. For details on these setting methods, please refer to the user's manual of each product.

Folder structure of CSP_Projects_U2B_RTOS	Description
CSP_Projects_U2B_RTOS	-
platform	-
drivers	-
include	-
U2B_clock.h	Clock controller related functions
U2B_interrupt.h	Interrupt controller related functions
U2B_OSTM.h	OSTM related functions
U2B_pin.h	Port related functions
U2B_RLIN3_UART.h	RLIN3 of UART mode related functions
U2B_standby.h	Standby controller related functions
U2B_TAUD.h	TAUD related functions
U2x_general.h	General functions such as register manipulations
U2x_typedef.h	The typedef for driver software
library	-
U2B_Driver_Lib.lib	Library file for driver software

Figure 2-4 The folder structure of "platform\drivers" in sample projects

(3) samples

Figure 2-5 shows the folder structure of "samples" in the sample projects "CSP_Projects_U2B_RTOS".

This folder contains the CS+ project folder of each sample program, and source files and header files commonly used by each sample program.

Folder structure of CSP_Projects_U2B_RTOS	Description
CSP_Projects_U2B_RTOS	-
samples	-
1_Static_Memory	CS+ project folder for the sample program of Static Memory
2_Dynamic_Memory	CS+ project folder for the sample program of Dynamic Memory
3_Round_Robin_Scheduling	CS+ project folder for the sample program of Round Robin Scheduling
4_Preemption_Scheduling	CS+ project folder for the sample program of Preemption Scheduling
5_Queue_Management	CS+ project folder for the sample program of Queue Management
6_Binary_Semaphores	CS+ project folder for the sample program of Binary Semaphores
7_Counting_Semaphores	CS+ project folder for the sample program of Counting Semaphores
8_Mutexes	CS+ project folder for the sample program of Mutexes
9_Gatekeeper Tasks	CS+ project folder for the sample program of Gatekeeper Tasks
common	The source and header files commonly used in each project
U2B_TAUD_App.c	TAUD setting functions called in the sample programs of 6.Binary_Semaphores and 7.Counting_Semaphores
U2B_TAUD_App.h	
U2B_UART_App.c	RLIN3 setting functions called in the sample programs of 8. Mutexes and 9. Gatekeeper Tasks
U2B_UART_App.h	
user_freeRTOS.c	Functions called directly from the FreeRTOS kernel

Figure 2-5 The folder structure of "samples" in sample projects

(4) CS+ project folder of each sample program

Figure 2-6 shows the CS+ project folder of the sample program "6_Binary_Semaphores" as an example of the folder structure of each sample project.

These .h file, .asm files, .c file, and config folder will be imported directly under the "files" category of this CS+ project.

Folder structure of each CS+ folder	Description
CSP_Projects_U2B_RTOS	-
samples	-
6_Binary_Semaphores	CS+ project folder for the sample program of Binary Semaphores
6_Binary Semaphores.mtpj	CS+ project file for the sample program of Binary Semaphores
config	
FreeRTOSConfig.h	Application specific definitions
boot0.asm	The vetor table and entry function of boot up
cstart.asm	Start up functions
main.c	main function of this sample program
user_interrupt.asm	Functions that preprocesses C language functions that are processed at the time of an interrupt.

Figure 2-6 The folder structure of "sample/6_Binary_Semaphores"

2.2 Make driver program and configuration of the FreeRTOS kernel

This section describes the files should be created when implementing FreeRTOS on RH850/U2B product. **Table 2-1** shows the list of files that needs to be created.

Table 2-1 The files need to be modified for MCU products to use

File name	Location in sample projects	Description
port.c	CSP_Projects_U2B_RTOS\platform\third-party\FreeRTOS\core\portable\Renesas_U2B24	The files that need to be edited for the RH850 architecture
portasm.s		
portmacro.h		
contextop.h		
FreeRTOSConfig.h	CSP_Projects_U2B_RTOS\samples\(project folder)\config	The application specific definitions for FreeRTOS API

2.2.1 porting.c

This file implements following functions:

- Function pxPortInitialiseStack
- Function xPortStartScheduler
- Function vPortEndScheduler

Of these, the function vPortEndScheduler is not used in sample programs, so its explanation will be omitted.

(1) Function pxPortInitialiseStack

This function initializes the stack area at the address specified by the parameter pxTopOfStack as shown in **Table 2-2**. To confirm operation, this sample program sets various general-purpose registers to unique values other than 0.

This function has three parameters:

- StackType_t *pxTopOfStack:
  Address of top of the stack for task.
- TaskFunction_t pxCode:
  Pointer to the task entry function
- void *pvParameters:
  The value that is passed as the parameter to the created task

This function returns the top address of the stack area after executing this function.

Table 2-2 Value of stack after initialization with function pxPortInitialiseStack

Address	Value after initialization	Description	Address	Value after initialization	Description
&pxTopOfStack-0	0x01010101	Initial Value of R1	&pxTopOfStack-68	0x19191919	Initial Value of R19
&pxTopOfStack-4	0x02020202	Initial Value of R2	&pxTopOfStack-72	0x20202020	Initial Value of R20
&pxTopOfStack-8	0x04040404	Initial Value of R4	&pxTopOfStack-76	0x21212121	Initial Value of R21
&pxTopOfStack-12	0x05050505	Initial Value of R5	&pxTopOfStack-80	0x22222222	Initial Value of R22
&pxTopOfStack-16	pvParameters	Initial Value of R6	&pxTopOfStack-84	0x23232323	Initial Value of R23
&pxTopOfStack-20	0x07070707	Initial Value of R7	&pxTopOfStack-88	0x24242424	Initial Value of R24
&pxTopOfStack-24	0x08080808	Initial Value of R8	&pxTopOfStack-92	0x25252525	Initial Value of R25
&pxTopOfStack-28	0x09090909	Initial Value of R9	&pxTopOfStack-96	0x26262626	Initial Value of R26
&pxTopOfStack-32	0x10101010	Initial Value of R10	&pxTopOfStack-100	0x27272727	Initial Value of R27
&pxTopOfStack-36	0x11111111	Initial Value of R11	&pxTopOfStack-104	0x28282828	Initial Value of R28
&pxTopOfStack-40	0x12121212	Initial Value of R12	&pxTopOfStack-108	0x29292929	Initial Value of R29
&pxTopOfStack-44	0x13131313	Initial Value of R13	&pxTopOfStack-112	0x30303030	Initial Value of R30
&pxTopOfStack-48	0x14141414	Initial Value of R14	&pxTopOfStack-116	pxCode	Initial Value of R31
&pxTopOfStack-52	0x15151515	Initial Value of R15	&pxTopOfStack-120	pxCode	Initial Value of EIPC
&pxTopOfStack-56	0x16161616	Initial Value of R16	&pxTopOfStack-124	0x03F38000	Initial Value of EIPSW
&pxTopOfStack-60	0x17171717	Initial Value of R17	&pxTopOfStack-128	0	*1
&pxTopOfStack-64	0x18181818	Initial Value of R18			

Note 1. This value defined by portNO_CRITICAL_SECTION_NESTING in pormacro.h.
These sample programs does not support nesting of the interrupt processing.

NOTE

In this sample program, interrupts are enabled at the start of each task by setting the above value in PSW.

(2) Function xPortStartScheduler

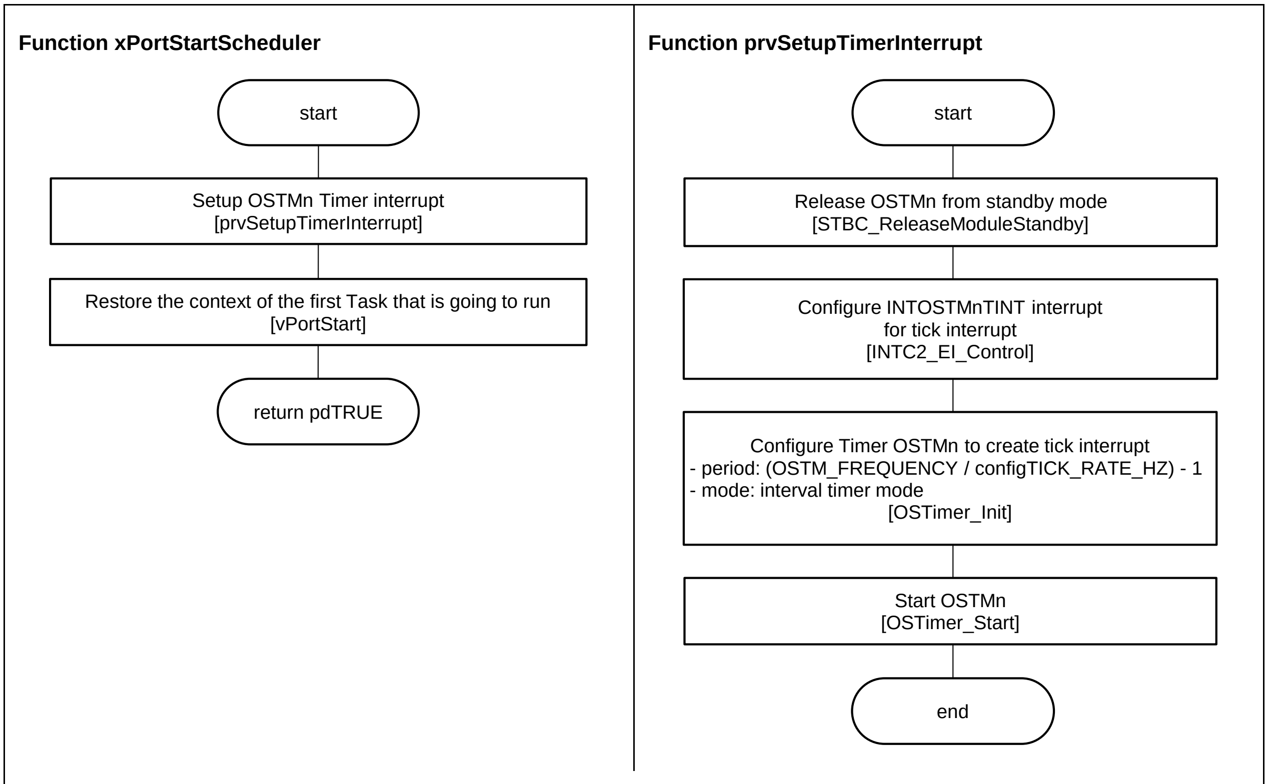
This function executes the function prvSetupTimerInterrupt, which sets up OSTM0 and starts operation, and the function vPortStart, which starts the operation of the first Task.

The function prvSetupTimerInterrupt sets and starts the tick interrupt, which is an interrupt that switches tasks. This sample program uses OSTM0 in interval timer mode as a tick interrupt.

The tick interrupt interval is typically 1ms to 10ms. In this sample program, the tick interrupt interval is set to 1ms.

About the function vPortStart, refer to **Section 2.2.2**.

Figure 2-7 shows the flowchart of the function xPortStartScheduler and the function prvSetupTimerInterrupt.



Note. In the sample programs, OSTMn = OSTM0.

Figure 2-7 Flowchart of the function xPortStartScheduler and the function prvSetupTimerInterrupt

2.2.2 portasm.asm

The file portasm.asm contains the following functions:

- Function _Intfunc_INTOSTM0TINT
- Function _vPortStart
- Function _vPortYield
- Function _trap_0

(1) Function _Intfunc_INTOSTM0TINT

Function _Intfunc_INTOSTM0TINT is the interrupt handler for interrupt INTOSTM0TINT, which is used as a tich interrupt.This function suspends the task that was being executed before the interrupt occurred, determines the next task to execute, and executes that task. **Figure 2-8** shows the flowchart of this function.

The macro portSAVE_CONTEXT is a macro that stores the context in the stack area, and the macro portRESTORE_CONTEXT is the macro that restores the context from the stack area. See **Section 2.2.4** for more information on these macros.

To monitor the timing of task switching due to the INTOSTM0TINT interrupt, monitor port P21_0 outputs high level near the start of this function, and P21_0 outputs low level near the end of this function. As a result, the sample program does not retain the value of R10 before and after switching tasks. If need to retain the value of R10, remove the output part to the monitor port.

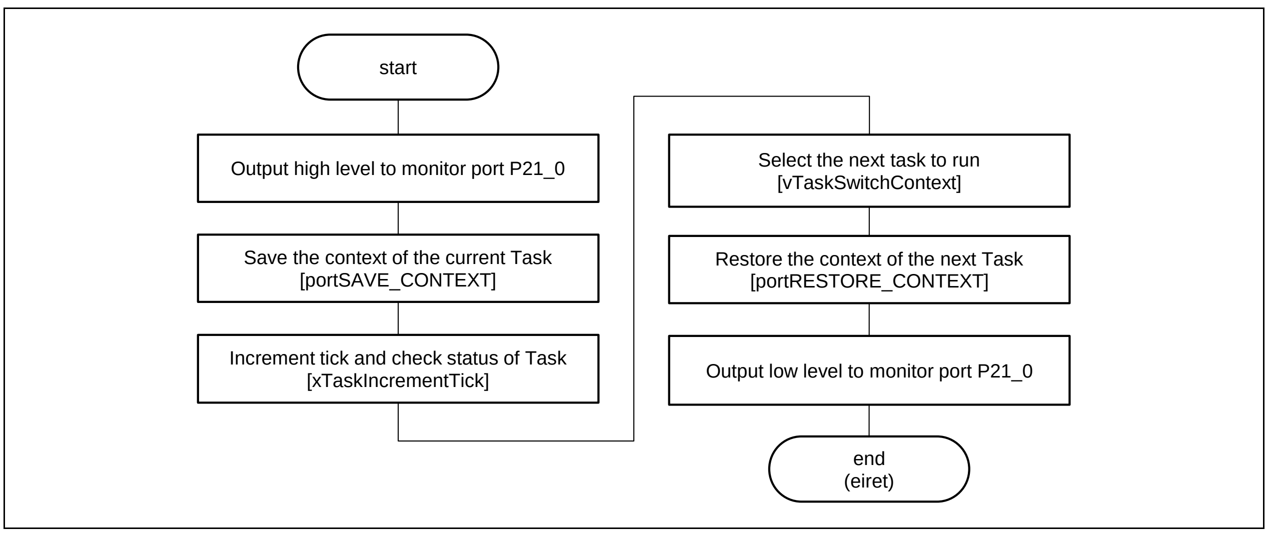


Figure 2-8 Flowchart of the function _Intfunc_INTOSTM0TINT

(2) Function _vPortStart

The function _vPortStart restores the context, and the first task run. This function is called by the function xPortStartScheduler.

Figure 2-9 shows the flowchart of the function _vPortStart.

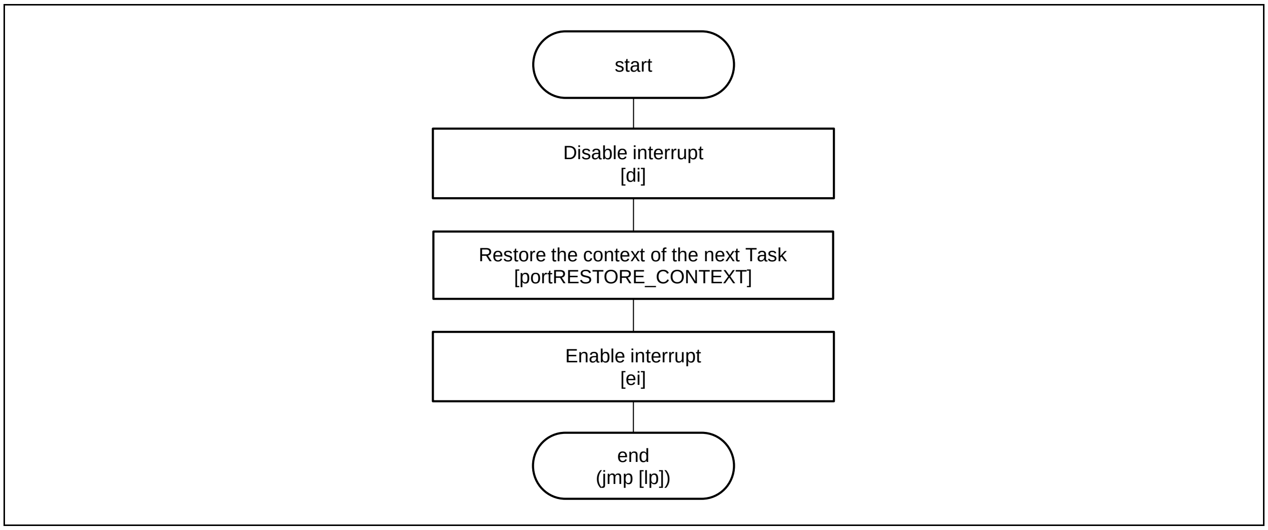


Figure 2-9 Flowchart of the function _vPortStart

(3) Function _vPortYield

The function _vPortYield is called when trap exception occurred. This function suspends the task that was running before the trap exception occurred, determines the next task to execute, and executes that task.

Figure 2-10 shows the flowchart of the function _vPortYield.

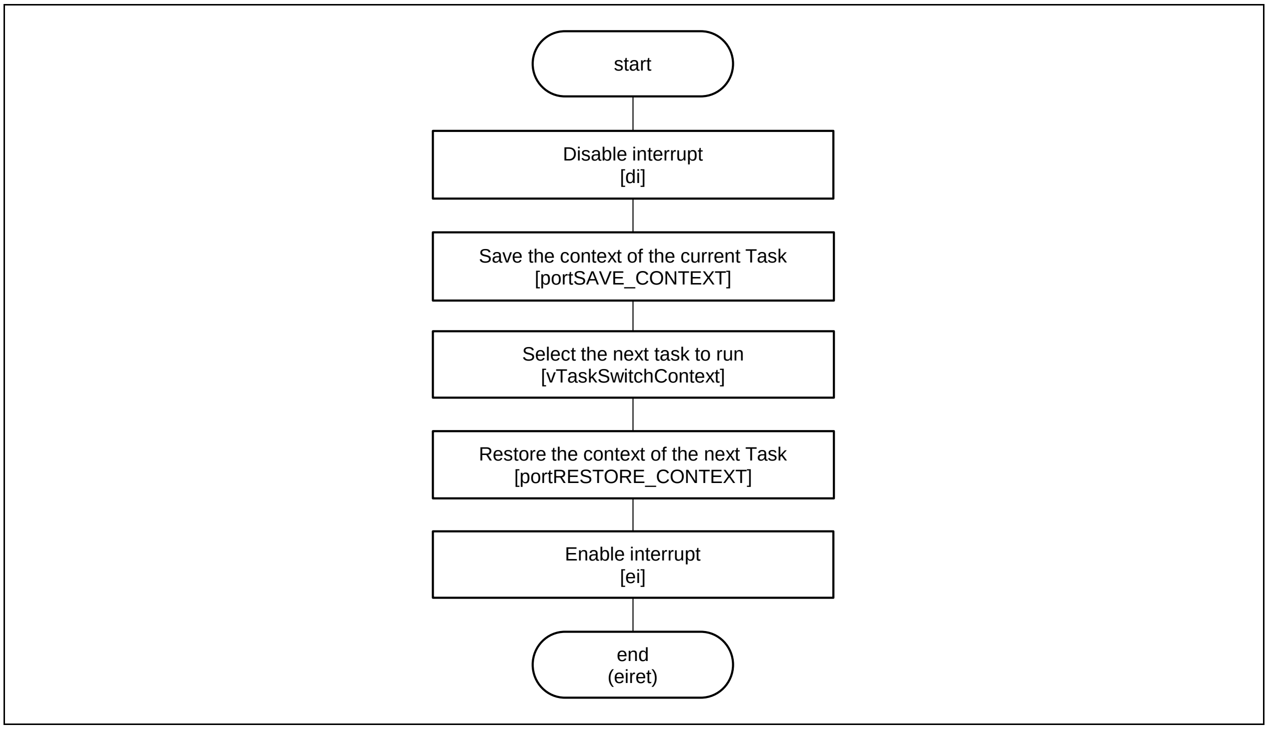


Figure 2-10 Flowchart of the function _vPortYield

(4) Function _trap_0

Function _trap_0 is a function to generate a trap exception.

2.2.3 portmacro.h

This file is a header file for the files port.asm and porting.c. In addition to prototype declarations for functions in these files, there are macros to disable and enable interrupts, and definitions used in FreeRTOS API functions.

2.2.4 contextop.h

This file contains the macro portSAVE_CONTEXT, which stores the context in the stack area, and the macro portRESTORE_CONTEXT, which restores the context from the stack area.

(1) macro portSAVE_CONTEXT

Figure 2-11 shows the flowchart of macro portSAVE_CONTEXT.

This macro stores the values of the general-purpose registers, EIPC, EIPSW, and global variable usCriticalNesting to the address in the minus direction from the stack area address indicated by the stack pointer SP at the time this macro is called. Each value is stored in the stack area in the order shown in Table 2-2.

Finally, this macro stores the value of the current stack pointer SP in the global variable pxCurrentTCB.pxTopOfStack, which indicates the address of the current stack pointer.

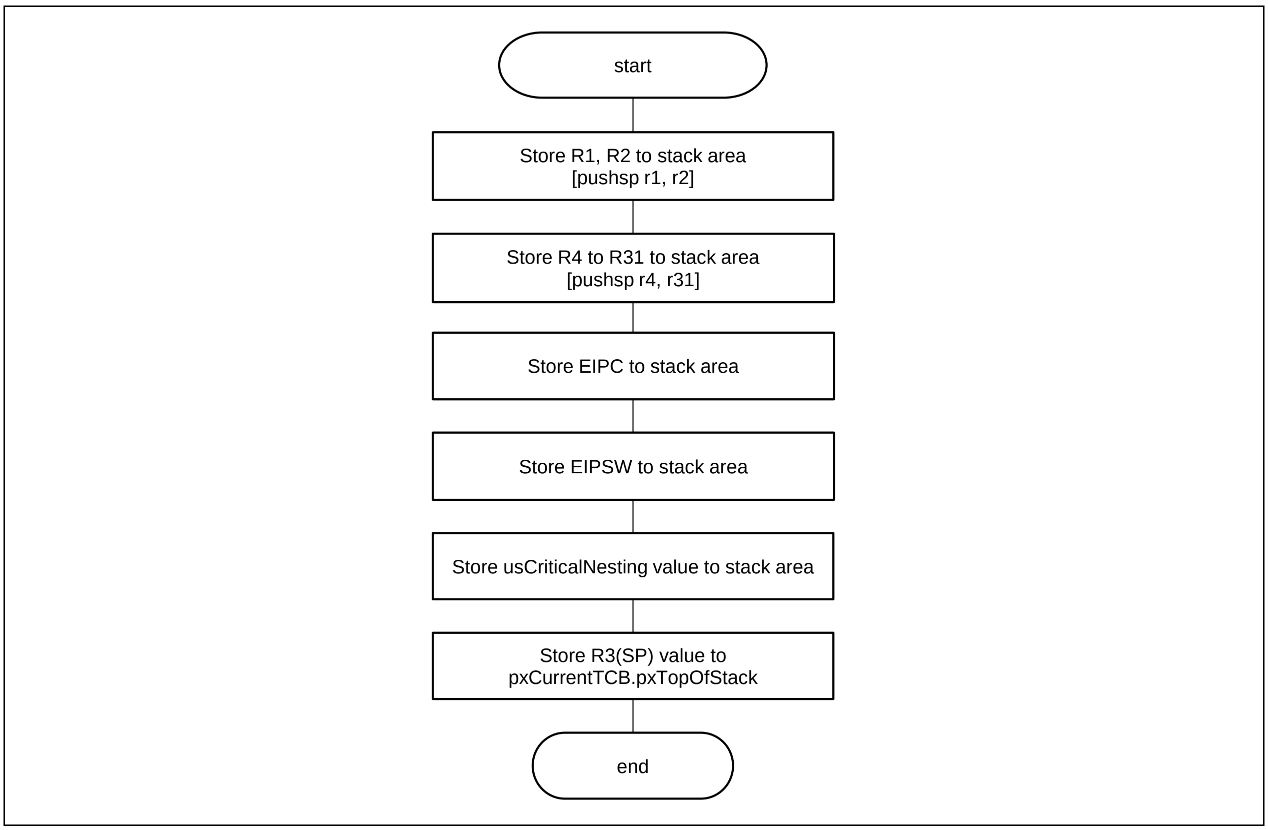


Figure 2-11 Flowchart of the macro portSAVE_CONTEXT

(2) macro portRESTORE_CONTEXT

Figure 2-12 shows the flowchart of macro portRESTORE_CONTEXT.

This macro obtains the value of the global variable usCriticalNesting, EIPSW, EIPC, and general-purpose register values from the address in the stack area indicated by the obtained stack pointer value to the address in the plus direction, and restores them to each register. Each value is retrieved from the stack area in the order shown in **Table 2-2**.

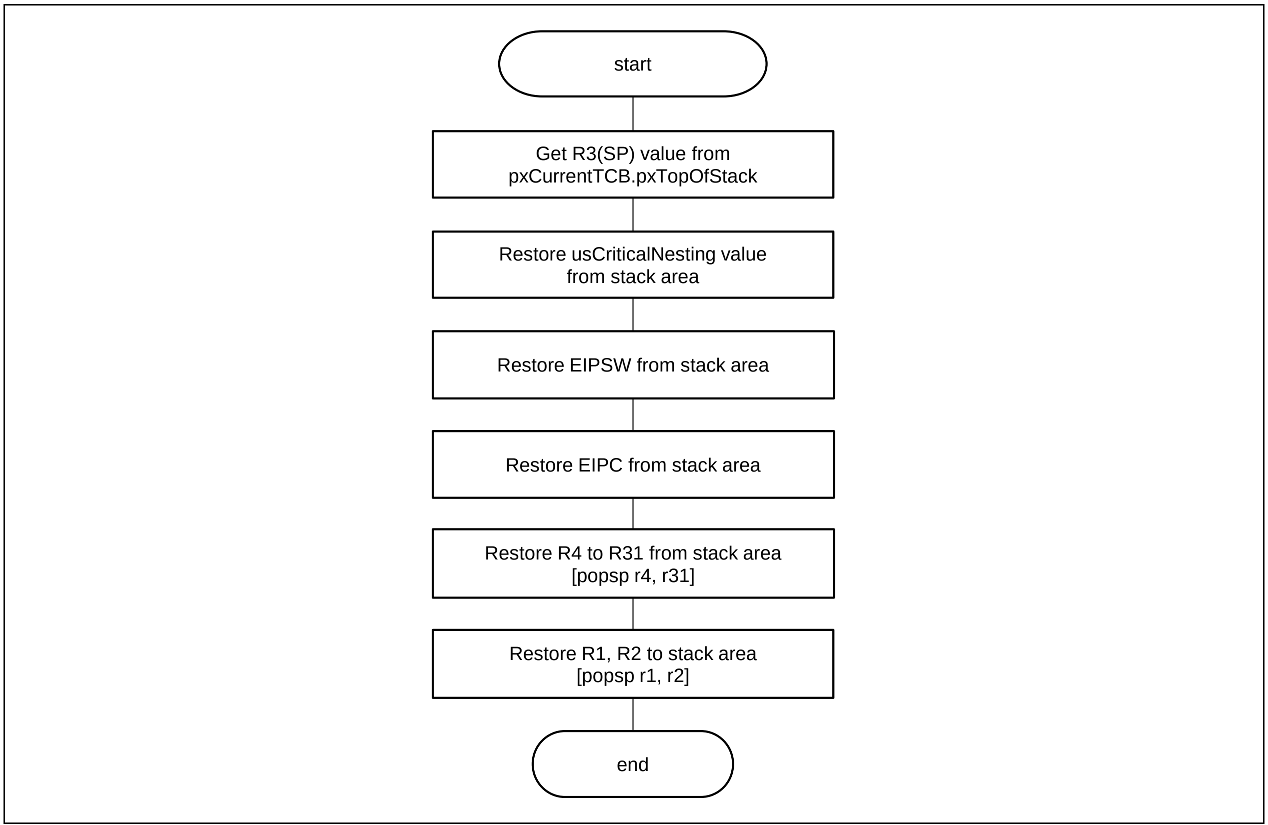


Figure 2-12 Flowchart of the macro portRESTORE_CONTEXT

2.2.5 FreeRTOSConfig.h

FreeRTOS is customized through the FreeRTOSConfig.h file. Each FreeRTOS application requires the FreeRTOSConfig.h header file. **Table 2-3** below shows configuration information for the sample program. For the contents of each definition, please refer to the FreeRTOS official website.

Kernel > Developer Docs > FreeRTOSConfig.h: ***https://www.freertos.org/a00110.html***

Table 2-3 Configuration parameters in FreeRTOS kernel (1/2)

Definition	Sample program								
	1_Static_ Memory	2_Dynamic_ Memory	3_Round_Robin_ Scheduling	4_Preemption_ Scheduling	5_Queue_ Management	6_Binary_ Semaphores	7_Counting_ Semaphores	8_Mutexes	9_Gatekeeper_ Tasks
configUSE_COUNTING_SEMAPHORES	0	0	0	0	0	0	1	0	0
configSUPPORT_STATIC_ALLOCATION	1	0	0	0	0	0	0	0	0
configSUPPORT_DYNAMIC_ALLOCATION	0	1	1	1	1	1	1	1	1
configUSE_PREEMPTION	1								
configUSE_IDLE_HOOK	1								
configUSE_TICK_HOOK	0								
configTICK_RATE_HZ	(TickType_t) 1000								
configMAX_PRIORITIES	4								
configMINIMAL_STACK_SIZE	(unsigned short) 128								
configMAX_TASK_NAME_LEN	10								
configUSE_TRACE_FACILITY	0								
configUSE_16_BIT_TICKS	0								
configIDLE_SHOULD_YIELD	0								
configUSE_CO_ROUTINES	0								
configUSE_MUTEXES	1								
configCHECK_FOR_STACK_OVERFLOW	2								
configUSE_RECURSIVE_MUTEXES	1								
configQUEUE_REGISTRY_SIZE	0								
configUSE_MALLOC_FAILED_HOOK	1								
configUSE_QUEUE_SETS	0								
configUSE_CO_ROUTINES	0								
configMAX_CO_ROUTINE_PRIORITIES	2								
configHEAP_CLEAR_MEMORY_ON_FREE	1								
configUSE_TIMERS	1								
configTIMER_TASK_PRIORITY	configMAX_PRIORITIES - 1								
configTIMER_QUEUE_LENGTH	5								
configTIMER_TASK_STACK_DEPTH	configMINIMAL_STACK_SIZE*2								

Table 2-3 Configuration parameters in FreeRTOS kernel (2/2)

Definition	Sample program								
	1_Static_ Memory	2_Dynamic_ Memory	3_Round_Robin_ Scheduling	4_Preemption_ Scheduling	5_Queue_ Management	6_Binary_ Semaphores	7_Counting_ Semaphores	8_Mutexes	9_Gatekeeper_ Tasks
INCLUDE_vTaskPrioritySet	1								
INCLUDE_uxTaskPriorityGet	1								
INCLUDE_vTaskDelete	1								
INCLUDE_vTaskCleanUpResources	0								
INCLUDE_vTaskSuspend	1								
INCLUDE_vTaskDelayUntil	1								
INCLUDE_vTaskDelay	1								
INCLUDE_eTaskGetState	1								
configTOTAL_HEAP_SIZE	(size_t) (8*1024)								
configCPU_CLOCK_HZ	(unsigned long) 400000000								

Section 3 Sample programs

This section describes sample programs. **Table 3-1** shows the sample programs attached to this application note.

The "Location" in **Table 3-1** indicates the relative path from the folder "CSP_Projects_U2B_RTOS" in the attached sample projects file.

Table 3-1 List of sample program

Title	Location	Refer to
Static memory in Memory Management	samples/1_Static_Memory	**Section 3.1.1**
Dynamic memory in Memory Management	samples/2_Dynamic_Memory	**Section 3.1.2**
Scheduling in Task Management	samples/3_Round_Robin_Scheduling	**Section 3.2.1**
Preemption Scheduling in Task Management	samples/4_Preemption_Scheduling	**Section 3.2.2**
Queue Management	samples/5_Queue_Management	**Section 3.3.1**
Binary Semaphores in Interrupt Management	samples/6_Binary_Semaphores	**Section 3.4.1**
Counting Semaphores in Interrupt Management	samples/7_Counting_Semaphores	**Section 3.4.2**
Mutexes in Resource Management	samples/8_Mutexes	**Section 3.4.3**
Gatekeeper Tasks in Resource Management	samples/9_Gatekeeper_Tasks	**Section 3.4.4**

3.1 Memory Management

3.1.1 Static memory

3.1.1.1 Overview

This sample program demonstrates the steps to create two simple tasks using static memory. Use the function xTaskCreateStatic to allocates memory for two tasks. Both tasks are created at the same priority and execute an infinite loop.

3.1.1.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-2**.

Table 3-2 API functions used in Static Memory program

Function name	Description	Link to FreeRTOS official site
xTaskCreateStatic	Create a new task and add it to the list of tasks that are ready to run. The RAM is statically allocated at compile time.	https://www.freertos.org/xTaskCreateStatic.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html

(2) Main program (Function main in main.c)

The main function of this sample program uses the function xTaskCreateStatic to create two tasks as **Figure 3-1**. The stack area and TCB area of the two tasks are allocated the variables declared below as static variables. The size and assigned address of these variables are determined at build time, so the stack area and TCB area of these tasks can be said to be static.

- static StackType_t Gx_Task1_stack[configMINIMAL_STACK_SIZE];
- static StaticTask_t Gx_Task1_TCB;
- static StackType_t Gx_Task2_stack[configMINIMAL_STACK_SIZE];
- static StaticTask_t Gx_Task2_TCB;

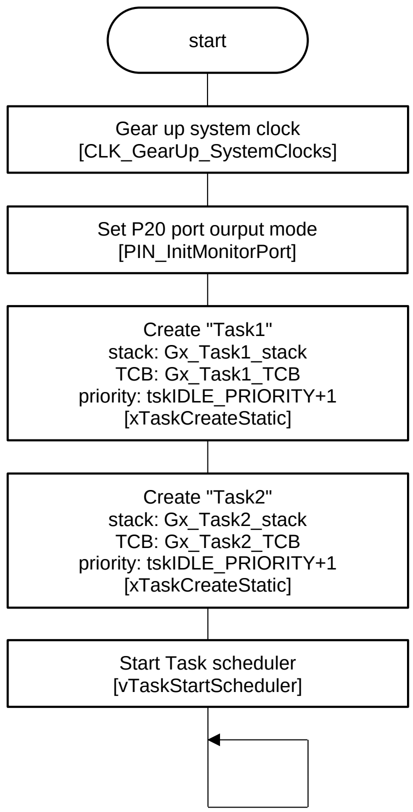
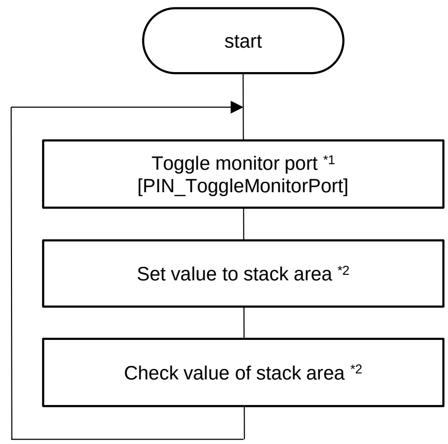


Figure 3-1 Flowchart of main of Static Memory program

(3) Main program for each task (Function Task1_main and Task2_main in main.c)

Figure 3-2 shows the main program flow for Task1 and Task2. After each task toggles the monitor port, it sets arbitrary values in its own stack area and checks the set values.



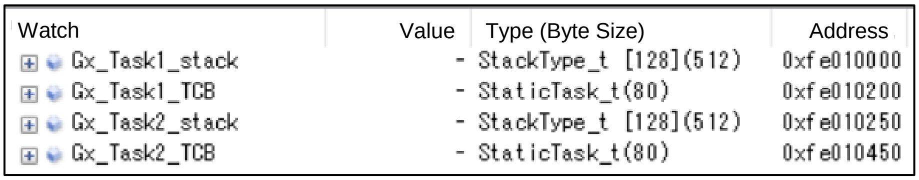
Note 1. The monitor port of Task1 is P20_1.
The monitor port of Task2 is P20_2.

Note 2. Task1 sets and checks 0x1111 1100, 0x1111 1101, 0x1111 1102, ... in its own stack area.
Task2 sets and checks 0x2222 2200, 0x2222 2201, 0x2222 2202, ... in its own stack area.

Figure 3-2 Flowchart of each task of Static Memory program

3.1.1.3 Operation result

When this sample program runs, it can be seen that the stack areas for Task1 and Task2 are allocated as variables Gx_Task1_stack and Gx_Task2_stack, respectively, and the TCB areas for Task1 and Task2 are allocated as variables Gx_Task1_TCB and Gx_Task2_TCB in the RAM area as **Figure 3-3**.



Watch	Value	Type (Byte Size)	Address
Gx_Task1_stack	-	StackType_t [128](512)	0xfe010000
Gx_Task1_TCB	-	StaticTask_t(80)	0xfe010200
Gx_Task2_stack	-	StackType_t [128](512)	0xfe010250
Gx_Task2_TCB	-	StaticTask_t(80)	0xfe010450

Figure 3-3 Memory allocation for Static Memory program

Figure 3-4 shows the memory map of Task1 and Task2 in this sample program.

The local variables variable_in_task1 and variable_in_task2 used in Task1 and Task2 are 312 bytes from 0xFE01 00C0 and 0xFE01 0310, respectively.

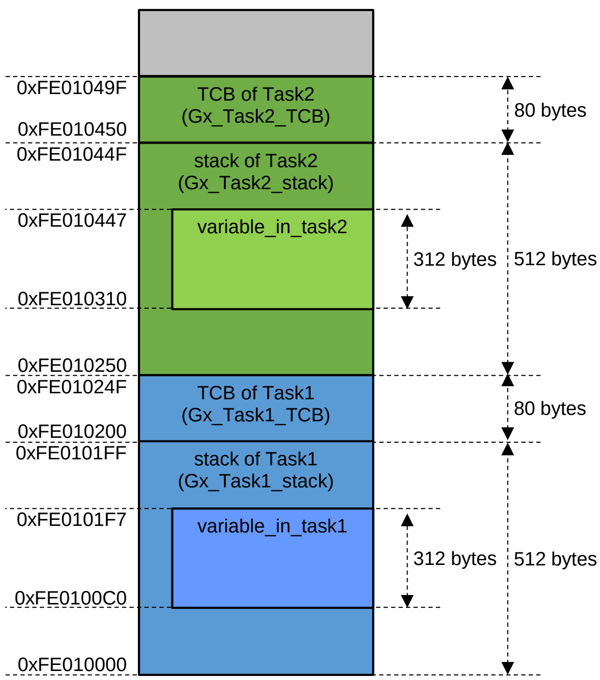


Figure 3-4 Memory allocation result for Static Memory program

Figure 3-5 shows the operating waveforms of this sample program.

The meaning of each port are as follows.

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: Task1 toggles output level of this port before set and check into its own stack area.
  If the check result does not match, the output level of this port is fixed.
- P20_2: Task2 toggles output level of this port before set and check into its own stack area.
  If the check result does not match, the output level of this port is fixed.

If the values set in the stack areas of Task1 and Task2 are the expected values, the monitor port will continue toggling. This indicates that the stack area value of each task is preserved even when switching between Task1 and Task2 occurs.

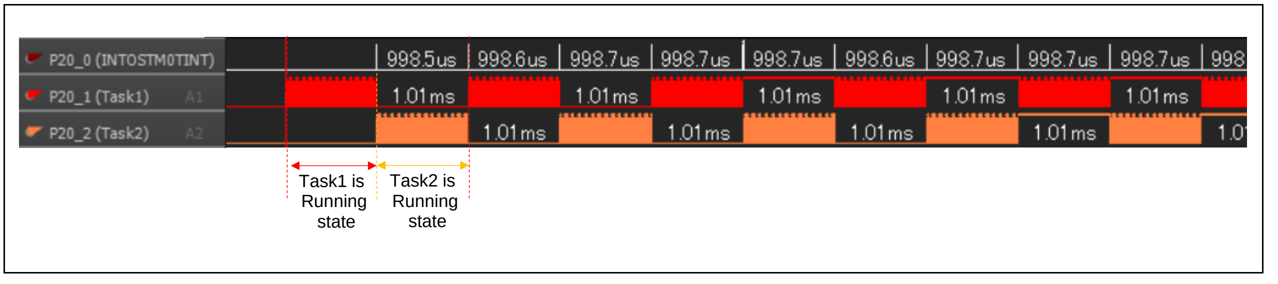


Figure 3-5 Operation waveform of Static Memory program

3.1.2 Dynamic memory

3.1.2.1 Overview

To allocate memory dynamically, use the heap_4 for memory allocation, described in this section.

The heap_4 uses a First Fit algorithm to allocate memory. The First Fit algorithm is an algorithm that searches through the list of free spaces of memory, starting from the beginning of the list, until it finds a free space that is large enough to accommodate the memory request from the process.

The heap_4 combines adjacent free blocks of memory into a single larger block, which minimizes the risk of memory fragmentation. The heap_4 should be used when the program continuously creates/deletes tasks, Queues, etc.

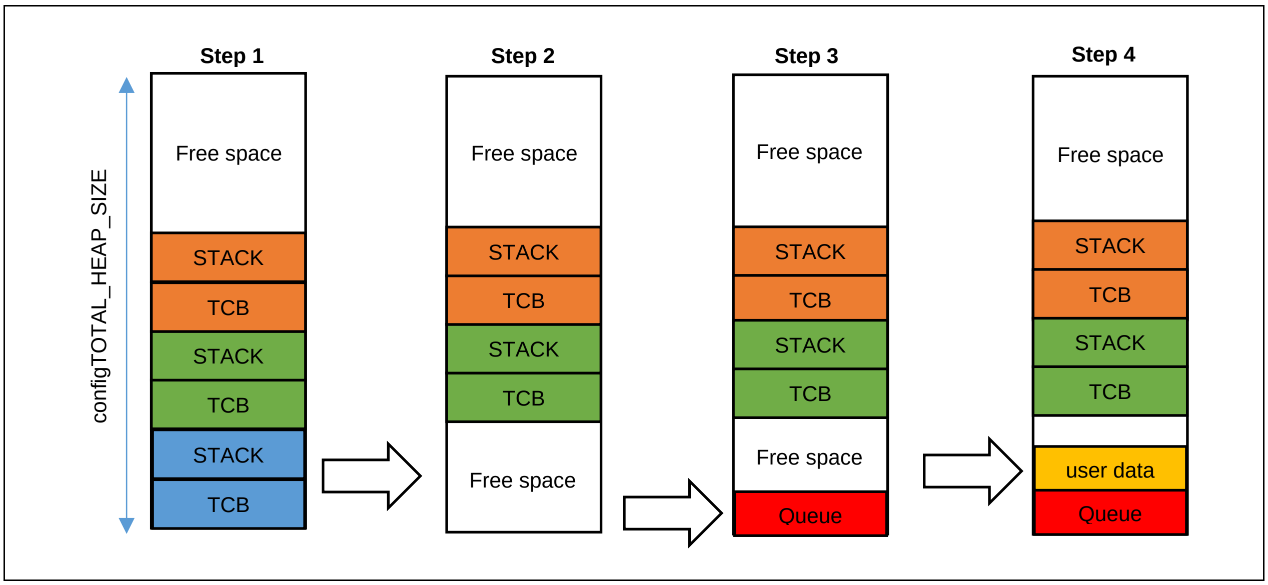


Figure 3-6 Memory allocation image after deleting Task and creating Queue

Figure 3-6 shows how the heap_4 works, and memory is allocated and freed.

Step 1: The three tasks have been created.

Step 2: One of the tasks has been deleted. The large free space at the top of the array remains.

Step 3: The Queue has been created. As the heap_4 uses a First Fit algorithm, the system will allocate RAM from the first free RAM block that is large enough to hold the Queue.

Step 4: The user allocates data. Since the user data is small enough, it fits in between the Queue and the memory.

This sample program demonstrates the steps to create three simple tasks using dynamic memory. By using the function xTaskCreate in **Table 3-3** it allocates memory for three tasks.

3.1.2.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-3**.

Table 3-3 API functions used in Dynamic Memory program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
vTaskDelete	Remove a task from the RTOS kernels management.	https://www.freertos.org/a00126.html
xQueueCreate	Creates a new queue and returns a handle by which the queue can be referenced.	https://www.freertos.org/a00116.html
xQueueSend	Post an item on a queue. The item is queued by copy, not by reference.	https://www.freertos.org/a00117.html
xQueueReceive	Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided.	https://www.freertos.org/a00118.html

(2) Main program (Function main in main.c)

The main function of this sample program uses the function xTaskCreate to create three tasks as Figure 3-7. The function xTaskCreate creates stack areas and TCB areas for Task1, Task2, and Task3 in the free space of the heap. Unlike function xTaskCreateStatic, the addresses of each task's stack area and TCB area are determined after function xTaskCreate is executed.

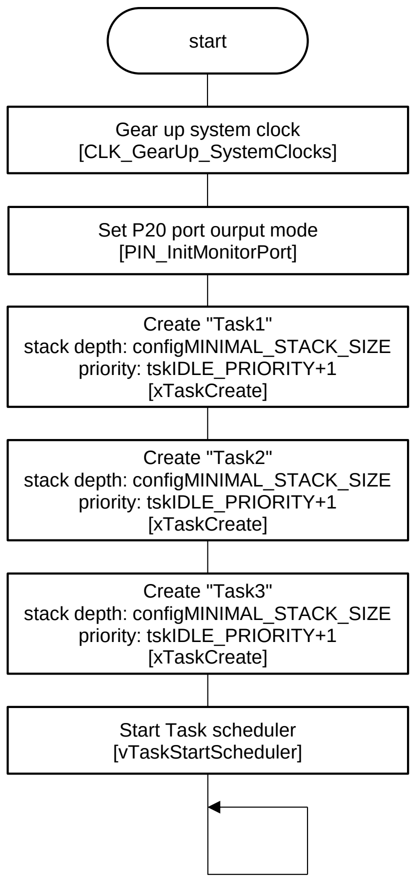
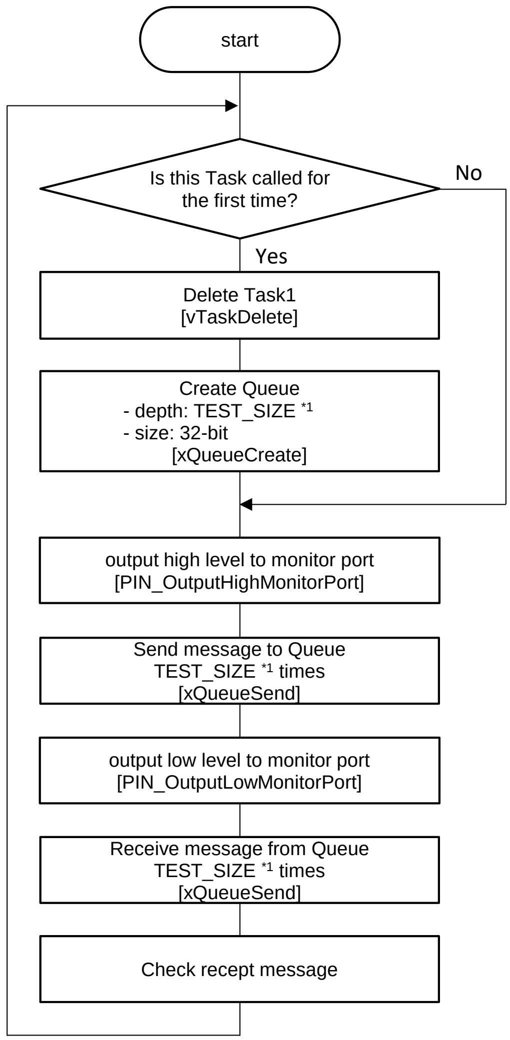


Figure 3-7 Flowchart of main of Dynamic Memory program

(3) Main program for Task2 (Function Task2_main in main.c)

Figure 3-8 shows the flowchat of the main function of Task2. When Task2 is executed for the first time, Task2 deletes Task1 and creates Queue. After then, Task2 sends message to Queue and receives the sent message from same Queue, and checks received message.



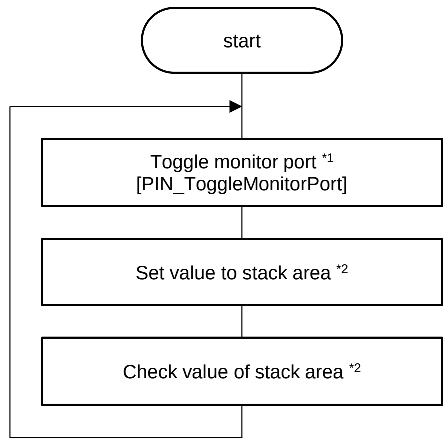
Note 1. TEST_SIZE is 66. This size depends on Task2's stack size.

Note 2. The monitor port of Task2 is P20_2.

Figure 3-8 Flowchart of Task2 of Dynamic Memory program

(4) Main program for Task1 and Task3 (Function Task1_main and Task3_main in main.c)

Figure 3-9 shows the main program flow for Task1 and Task3. After each task toggles the monitor port, it sets arbitrary values in its own stack area and repeatedly checks the set values.



Note 1. The monitor port of Task1 is P20_1.
The monitor port of Task3 is P20_3.

Note 2. Task1 sets and checks 0x1111 1100, 0x1111 1101, 0x1111 1102, ... in its own stack area.
Task3 sets and checks 0x3333 3300, 0x3333 3301, 0x3333 3302, ... in its own stack area.

Figure 3-9 Flowchart of Task1 and Task3 of Dynamic Memory program

3.1.2.3 Operation result

By running this sample program, the RAM memory area will change to the following state over time:

After creating three tasks

(A) in **Figure 3-10** shows the address map after successfully creating three tasks.

After deleting Task1

(B) in **Figure 3-10** shows that when the scheduling process starts, Task2 removed Task1 and freed memory in TCB of Task1 and the stack of Task1.

After putting user data to Queue

(C) in **Figure 3-10** shows the situation after creating a Queue and putting user data to Queue by Task2. The block between the memory is allocated to the Queue and user data.

Figure 3-10 Memory allocation result for Dynamic Memory program

The address of each TCB and stack area can know the variable pxNewTCB in the function xTaskCreate.

Figure 3-11 shows the operating waveforms of this sample program.

The meaning of each port are as follows.

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: Task1 toggles output level of this port before set and check into its own stack area.
  If Task1 is stopped, the output level of this port is fixed.
- P20_2: Task2 outputs high level to this port before sending message to Queue.
  Task2 outputs low level to this port before receiving message to Queue.
  If the check result does not match, the output level of this port is fixed.
- P20_3: Task3 toggles output level of this port before set and check into its own stack area.
  If the check result does not match, the output level of this port is fixed.

If the values set in the stack areas of Task1 and Task3 are the expected values, the monitor port will continue toggling. This shows that the stack area value of each task is preserved even when switching between Task1 and Task3 occurs.

Also, after Task2 starts for the first time, Task2 deletes Task1, so even if the next Tick interrupt occurs, Task1 will not work.

After Task2 deletes Task1, it creates a Queue and sends and receives the Queue. If the received message has the expected value, P20_2 continues toggling. All received messages are temporarily held in the stack area within Task2, so the fact that P20_2 continues toggling means that even if a switch between Task2 and Task3 occurs, the value of the stack area of each task indicates that it is retained.

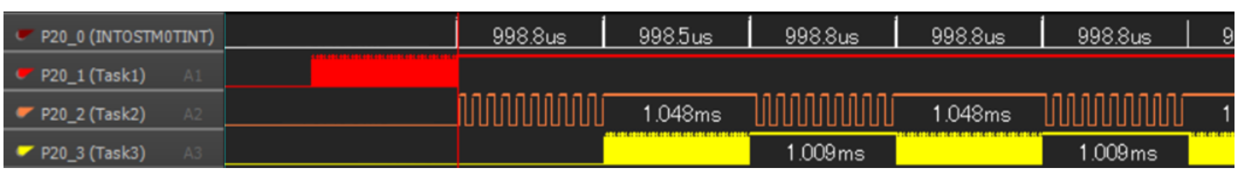


Figure 3-11 Operation waveform of Dynamic Memory program

3.2 Task Management

The most used algorithms in RTOS system are Round Robin Scheduling and Preemption Scheduling. This section describes the operation of these two algorithms.

3.2.1 Round Robin Scheduling

3.2.1.1 Overview

Round Robin is a simple scheduling algorithm in which tasks with the same priority will run alternately at regular intervals. These regular intervals are called time slices and are generated by timer interrupts. This timer interrupt is called a tick interrupt.

The tick interrupt frequency is configured by the application-defined configTICK_RATE_HZ compile-time configuration constant in FreeRTOSConfig.h.

This sample program generates Task1, Task2 with the same priority, and Task3 with a lower priority than them, and check when each task enters the Running state.

Figure 3-12 shows the operation timing of this sample program. Since Task3 has a lower priority than Tack1 and Task2, Task3 remains in Ready state until Task1 and Task2 complete their processing. Task1 and Task2 alternately switch to Running state at the timing of a timer interrupt.

This sample program uses INTOSTM0TINT as a tick interrupt, and the time slice is 1 ms.

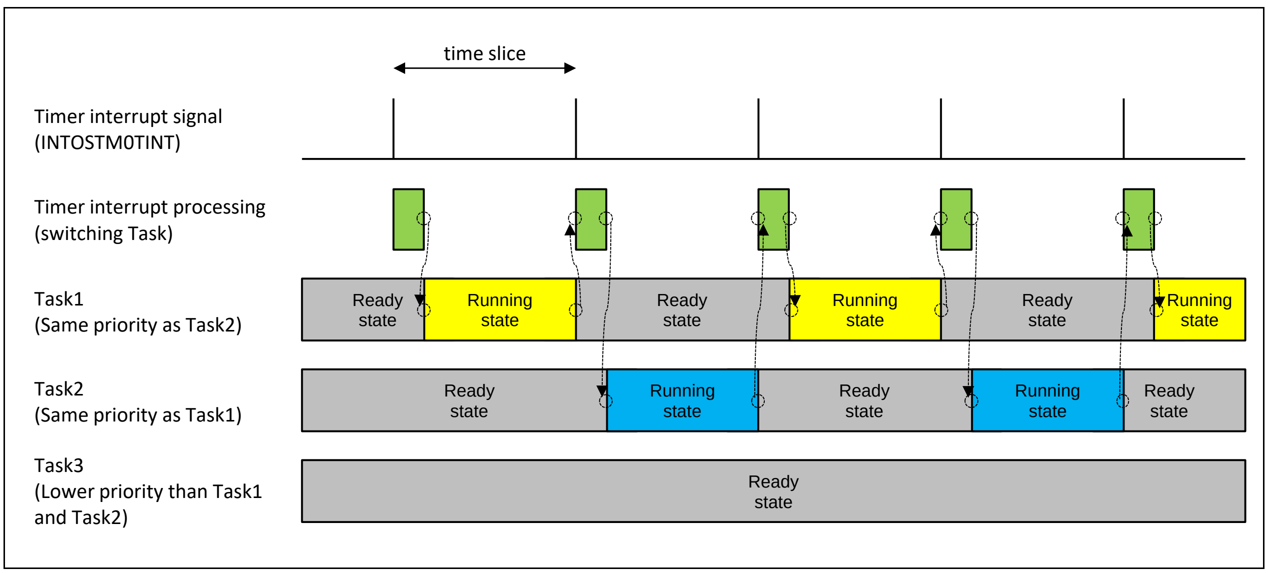


Figure 3-12 Timing chart of Round Robin Scheduling program

3.2.1.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-4**.

Table 3-4 API functions used in Round Robin Scheduling program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html

(2) Main program (Function main in main.c)

Figure 3-13 is the flowchart of the main program that creates Task1, Task2, and Task3.

After creating each task, the tick operation is started by setting and starting operation of OSTM0 in the function vtaskstartScheduler.

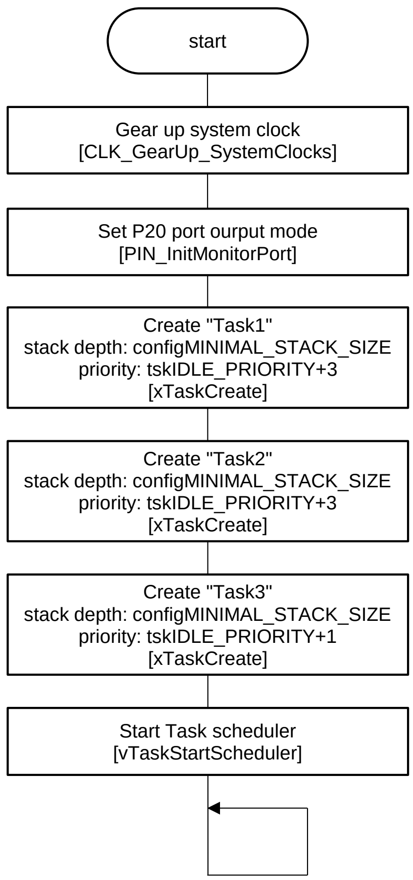
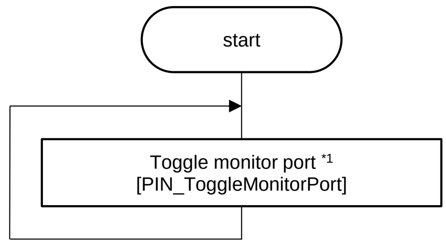


Figure 3-13 Flowchart of main of Round Robin Scheduling program

(3) Main program for each task (Function Task1_main, Task2_main, and Task3_main in main.c)

Figure 3-14 shows the flow of the main program of Task1, Task2, and Task3. These three tasks differ in the ports they toggle. To observe that each task has become Running state, each task toggles the output level of unique ports for each task.



Note 1. The monitor port of Task1 is P20_1.
The monitor port of Task2 is P20_2.
The monitor port of Task3 is P20_3.

Figure 3-14 Flowchart of each task of Round Robin Scheduling program

3.2.1.3 Operation result

Figure 3-15 shows the execution results of the sample program. The meaning of each port are as follows.

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: Task1 operating state. The output level is toggle at the time of Running state
- P20_2: Task2 operating state. The output level is toggle at the time of Running state
- P20_3: Task3 operating state. The output level is toggle at the time of Running state

Since Task1 and Task2 have the same priority, when a tick interrupt occurs at approximately 1ms intervals, Task1 and Task2 alternately transition to the execution state. Toggling the output level of P20_1 indicates that Task1 is in the Running state, and toggling the output level of P20_2 indicates that Task2 is in the Running state.

Task3 priority are less than Task1 and Task2 Task3 will never enter the Running state and the output level of P20_3 will not toggle.

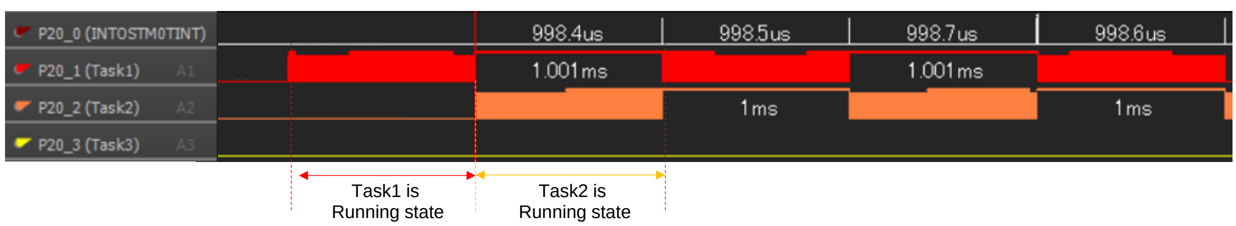


Figure 3-15 Operation waveform of Round Robin Scheduling program

3.2.2 Preemption Scheduling

3.2.2.1 Overview

Preemptive scheduling is the default scheduling algorithm in FreeRTOS. If multiple tasks exist, the Task with the highest priority becomes the Running state. A Task with a lower priority will not enter the Running state unless a Task with a higher priority enters the Blocked state or Suspend state.

This sample program uses the function vTaskDelay to transition a high priority task to the Blocked state, allowing lower priority tasks to operate.

Figure 3-16 shows the execution status of each task in preemption scheduling.

(1) After Task1 enters the Running state for the second time, Task1 executes the function vTaskDelay at the end of processing to put itself in the Blocked state and request a task switch.

(2) Task1, which has the highest priority, is in the Blocked state, so Task2, which has the next highest priority, is in the Running state.

(3) Task2 executes the function vTaskDelay at the end of processing to put itself into a Blocked state and request a task switch.

(4) Since Task1 and Task2 with the highest priority are in the Blocked state, Task3 with the lowest priority is in the Running state.

(5) During the processing of Task3, Task1 returned from the suspended state, so Task1 goes into the Running state at the next tick interrupt.

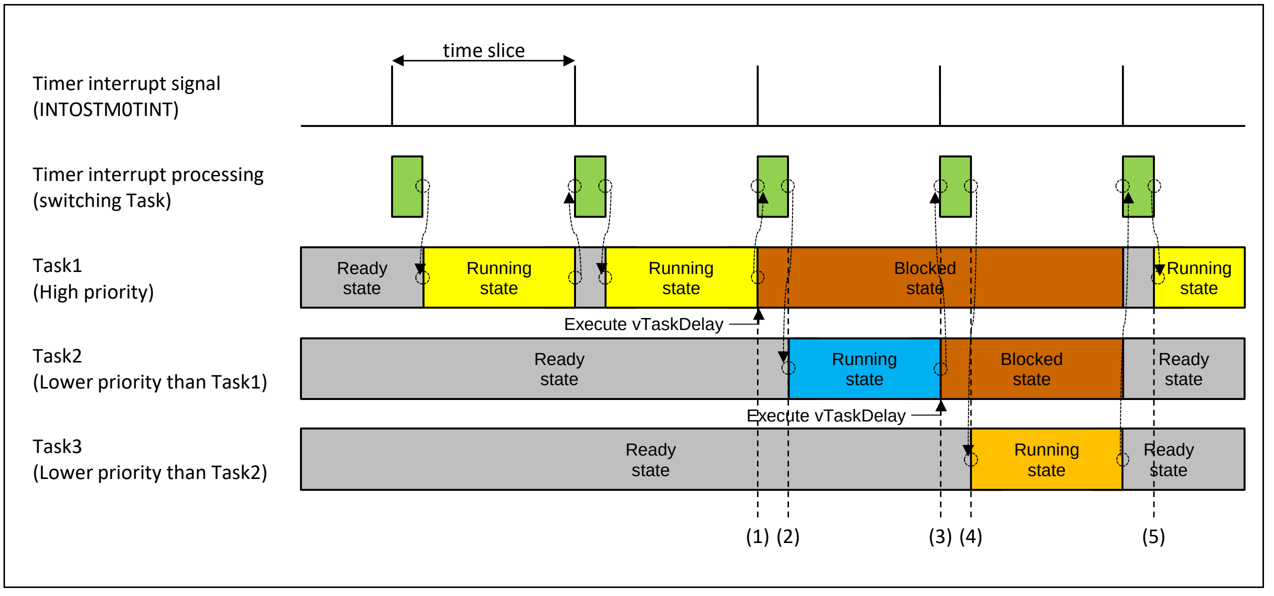


Figure 3-16 Timing chart of Preemption Scheduling program

3.2.2.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-5**.

Table 3-5 API functions used in Preemption Scheduling program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
vTaskDelay	Delay a task for a given number of ticks.	https://www.freertos.org/a00127.html

(2) Main program (Function main in main.c)

Figure 3-17 is the flowchart of the main program that creates Task1, Task2, and Task3. The difference from the Round Robin sample program is that each task has a different priority.

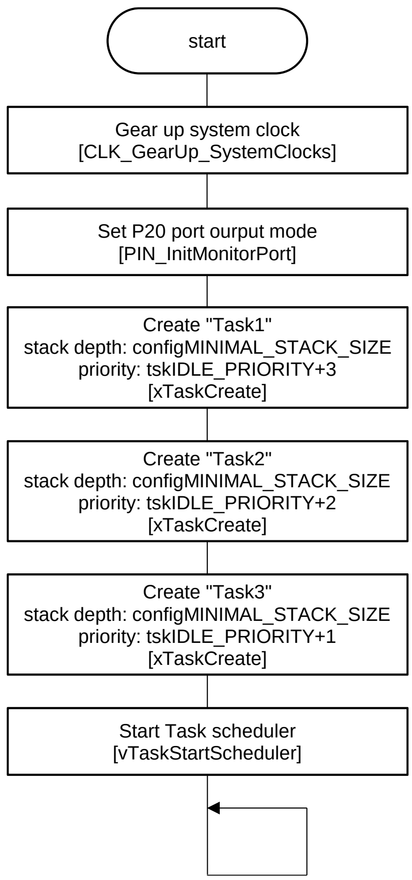
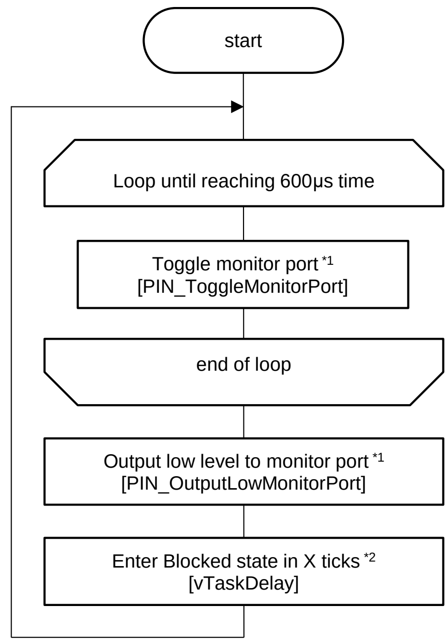


Figure 3-17 Flowchart of main of Preemption Scheduling program

(3) Main program for Task1 and Task2 (Function Task1_main, Task2_main in main.c)

After toggling the port output for a certain period, Task1 and Task2 execute the function vTaskDelay to transition itself to the Blocked state. The only difference between Task1 and Task2 is the port they toggle and the duration of the Blocked state. **Figure 3-18** is the flowchart of the main program of Task1 and Task2.



Note 1. The monitor port of Task1 is P20_1.
The monitor port of Task2 is P20_2.

Note 2. Blocked state period of Task1 is 3 ticks.
Blocked state period of Task2 is 4 ticks.

Figure 3-18 Flowchart of Task1 and Task2 of Preemption Scheduling program

(4) Main program for Task3 (Function Task3_main in main.c)

To ensure that Task3 is in the Running state, Task3 repeatedly toggles the output level of P20_3.

3.2.2.3 Operation result

Figure 3-19 shows the execution result of sample program for preemption scheduling. The meaning of each port are as follows.

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: Task1 operating state. The output level is toggle at the time of Running state
- P20_2: Task2 operating state. The output level is toggle at the time of Running state
- P20_3: Task3 operating state. The output level is toggle at the time of Running state

Task1 toggles the output level of P20_1 for 600µs. After that, P20_1 remains at the Low output because Task1 remains in the Blocked state until the third tick interrupt occurs due to the execution of the function vTaskDelay.

Since task 1 is in the blocked state, task 2, which has the next highest priority after task 1, transitions to the running state. Similar to Task1, Task2 toggles the output level of P20_2 for 600and then becomes into a Blocked state until the fourth tick interrupt occurs.

Since both Task1 and Task2 are in Blocked state, Task3 transitions to Running state and toggles the output level of P20_3.

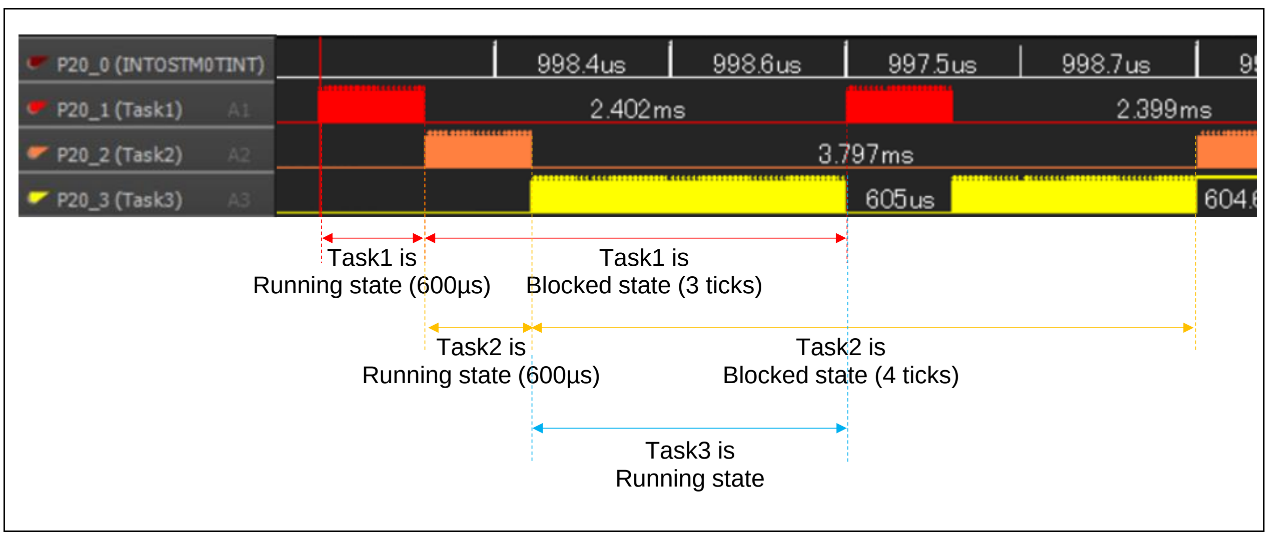


Figure 3-19 Operation waveform of Preemption Scheduling program

3.3 Queue Management

3.3.1 Queue operation

3.3.1.1 Overview

This sample program creates one Queue and three tasks, Task1, Task2, and Task3. Task2 and Task3 send data to the Queue, and Task1 receives data from the Queue.

The sample program creates Queue using function xQueueCreate. The Queue holds data items of type uint32_t, which is an unsigned long type.

Task2 and Task3 send data to the Queue using function xQueueSend after a delay of 4 ticks. This delay is used to ensure that Task3, which receives data from the Queue, is kept waiting while the Queue is empty.

As shown **Figure 3-20**, Task1 transitions to the Blocked state when it requests to receive data from the Queue using the function xQueueReceive. As soon as the data arrives in the Queue, the Blocked state is released and Task1 can read the data in the Queue.

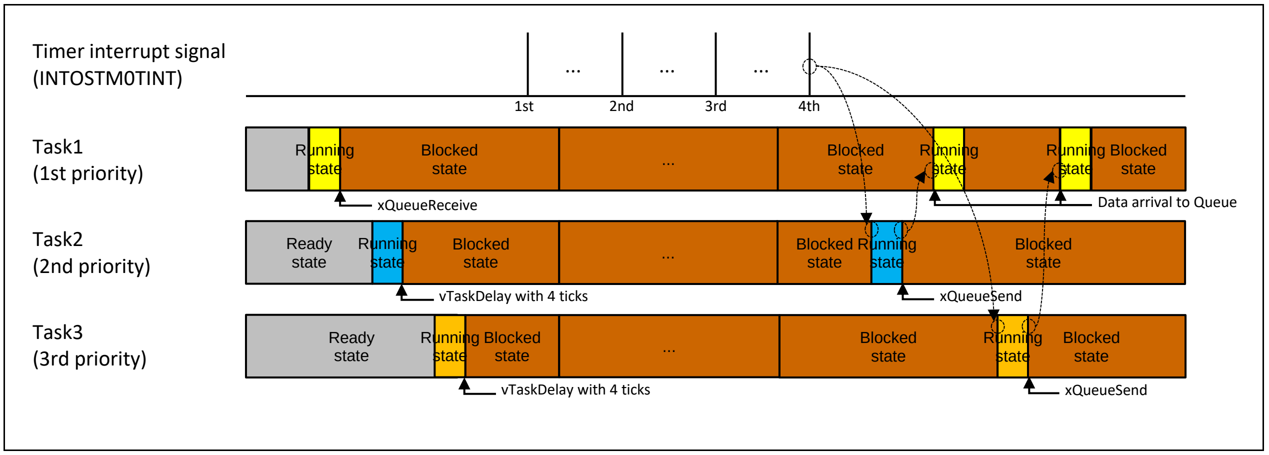


Figure 3-20 Timing chart of Queue program

3.3.1.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-6**.

Table 3-6 API functions used in Queue program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
xQueueCreate	Creates a new queue and returns a handle by which the queue can be referenced.	https://www.freertos.org/a00116.html
xQueueSend	Post an item on a queue. The item is queued by copy, not by reference.	https://www.freertos.org/a00117.html
xQueueReceive	Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided.	https://www.freertos.org/a00118.html

(2) Main program (Function main in main.c)

Figure 3-21 is the flowchart of the main program that creates one Queue and three tasks. The Task that receives data from the Queue, Task1, has a higher priority than the two tasks that send data to the Queue, Task2 and Task3.

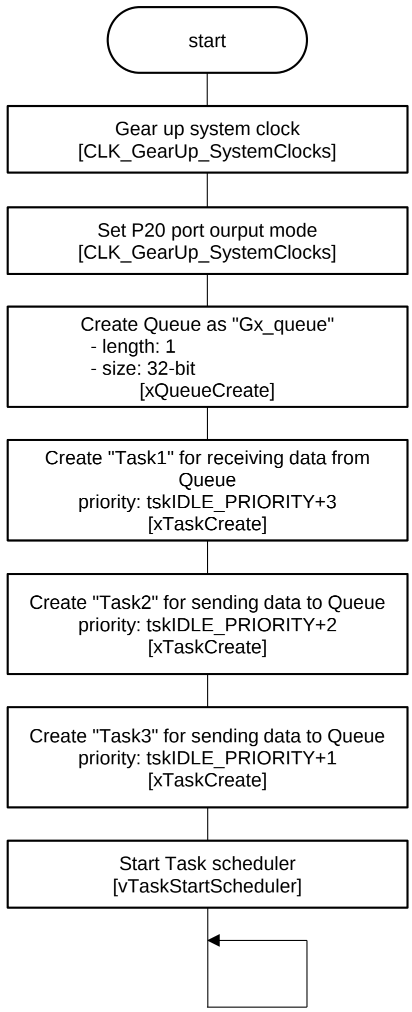


Figure 3-21 Flowchart of main of Queue program

(3) Main program for Task1 (Function Task1_main in main.c)

Figure 3-22 is the flowchart of the main program of Task1. Task1 receives data from the Queue and checks the received data. If the received data is the data sent by Task2, outputs P20_2 to low level, and if the received data is the data sent by Task3, outputs P20_3 to low level.

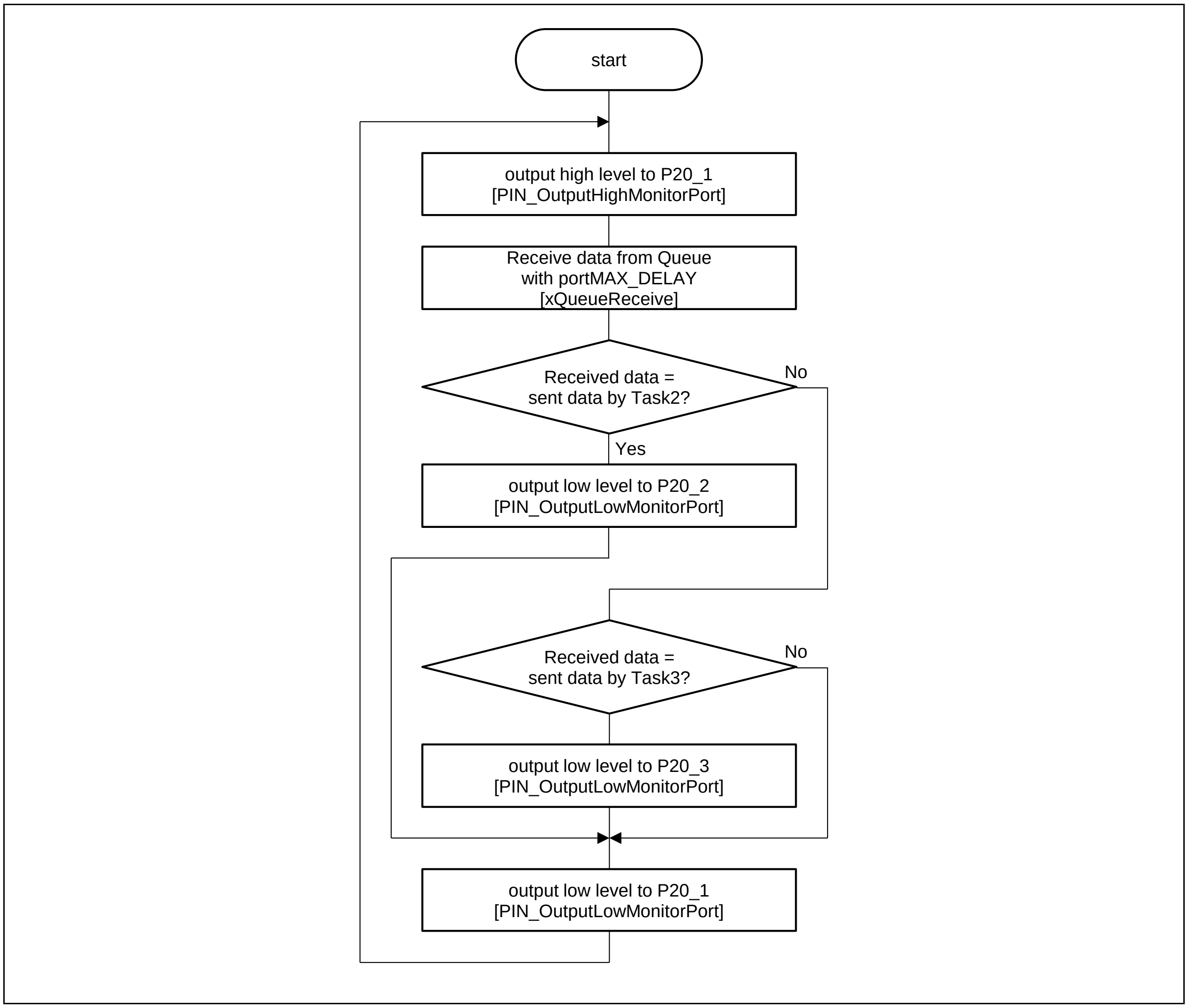
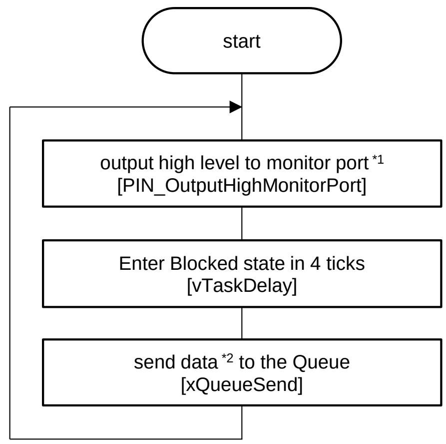


Figure 3-22 Flowchart of Task1 of Queue program

(4) Main program for Task2 and Task3 (Function Task2_main and Task3_main in main.c)

Figure 3-23 is the flowchart of the main program of Task2 and Task3. After outputting high level to P20_2 and P20_3 respectively, Task2 and Task3 execute the function vTaskDelay to transition itself to the Blocked state. After returning from the Blocked state after 4 ticks, these tasks send the data to the Queue.



Note 1. The monitor port of Task2 is P20_2.
The monitor port of Task3 is P20_3.

Note 2. Task2 sends 0x5A5A5A5A to Queue.
Task3 sends 0xA5A5A5A5 to Queue.

Figure 3-23 Flowchart of Task2 and Task3 of Queue program

3.3.1.3 Operation result

Figure 3-24 shows the execution result of sample program for Queue operation. The meaning of each port are as follows.

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: Task1 makes this port a high level output before executing the function xQueueReceive.
  Task1 makes this port to a low level after receiving Queue data.
- P20_2: Task2 makes this port a high level output before transition to Blocked state.
  When Task1 receives the Queue data sent by Task2, Task1 sets this port to a low level.
- P20_3: Task3 makes this port a high level output before sending data to Queue.
  When Task1 receives the Queue data sent by Task3, Task1 sets this port to a low level.

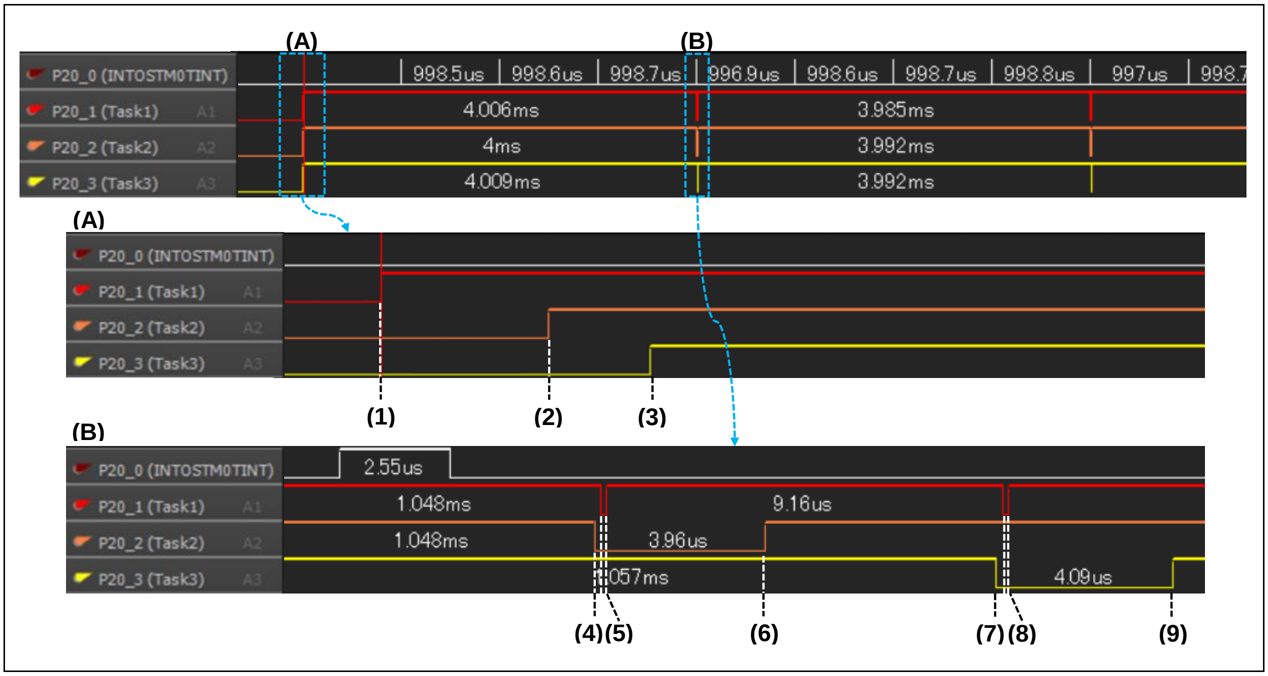


Figure 3-24 Operation waveform of Queue program

(A) in Figure 3-24

(1) Task1 executes the function xQueueReceive after setting P20_1 to High level output. This causes Task1 to transition to the Blocked state.

(2) Since Task1 is in the Blocked state, Task2, which has the next highest priority after Task1, will be executed. Task2 executes the function vTaskDelay after outputing P20_2 to high level. This causes Task2 to transition to the Blocked state.

(3) Since Task1 an Task2 are in the Blocked state, Task3 will be executed. Task3 executes the function vTaskDelay after outputing P20_3 to High level. This causes Task3 to transition to the Blocked state.

(B) in Figure 3-24

(4) After the 4th INTOSTM0TINT occurs after Task2 goes to Blocked state, Task2 transitions to Running state and sends data to the Queue. Since the data has arrived in the Queue, Task1 returns from the Blocked state. Task1 checks the received Queue data, and since the data was the data sent by Task2, it outputs P20_2 to low level.

(5) Task1 sets P20_1 to low level, then to high level, and executes the function xQueueReceive again. This causes Task1 to transition to the Blocked state.

(6) After the task switch processing from (5), Task2, which has the next highest priority, becomes Running state. After Task2 outputs P20_2 to high level, it executes the function vTaskDelay and transitions to Blocked state.

(7) After the 4th INTOSTM0TINT occurs after Task3 enters the Blocked state, Task3 transitions to the Running state and sends data to the Queue. Since the data has arrived in the Queue, Task1 returns from the Blocked state. Task1 checks the received Queue data, and since that data was the data sent by Task3, it outputs P20_3 to low level.

(8) Same as (5).

(9) After the task switch processing from (8), Task3, which has the next highest priority, becomes Running. After Task3 outputs P20_3 to high level, it executes the function vTaskDelay and transitions to Blocked state.

3.4 Resource Management

3.4.1 Binary Semaphores

3.4.1.1 Overview

This sample program uses Binary Semaphore to synchronize Task and interrupt processing by releasing a blocked Task using ISR. TAUD2 ch.0 is used as ISR.

Use the function xSemaphoreCreateBinary to create one Binary Semaphore and use the function xTaskCreate to create one Task, Task1.

Task1 uses the function xSemaphoreTake to take a Semaphore with a timeout of 3 ticks. Also, use the function xSemaphoreGive in the INTTAUD2I0 interrupt handler to release the Binary Semaphore.

Figure 3-25 shows the case where no Semaphore is given because no TAUD2 ch.0 interrupt occurs.

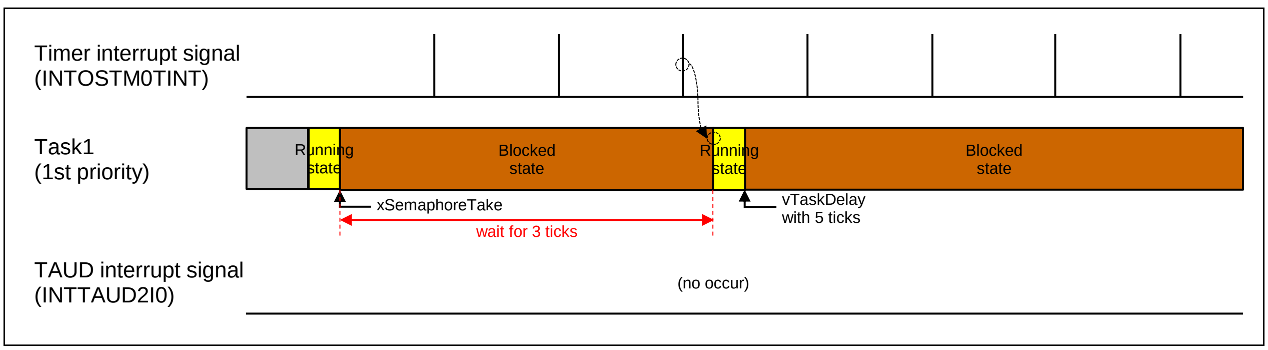


Figure 3-25 Timing chart of Binary Semaphore program when Semaphore is not released

Figure 3-26 shows the case where Semaphore is given by TAUD2 ch.0 interrupt processing.

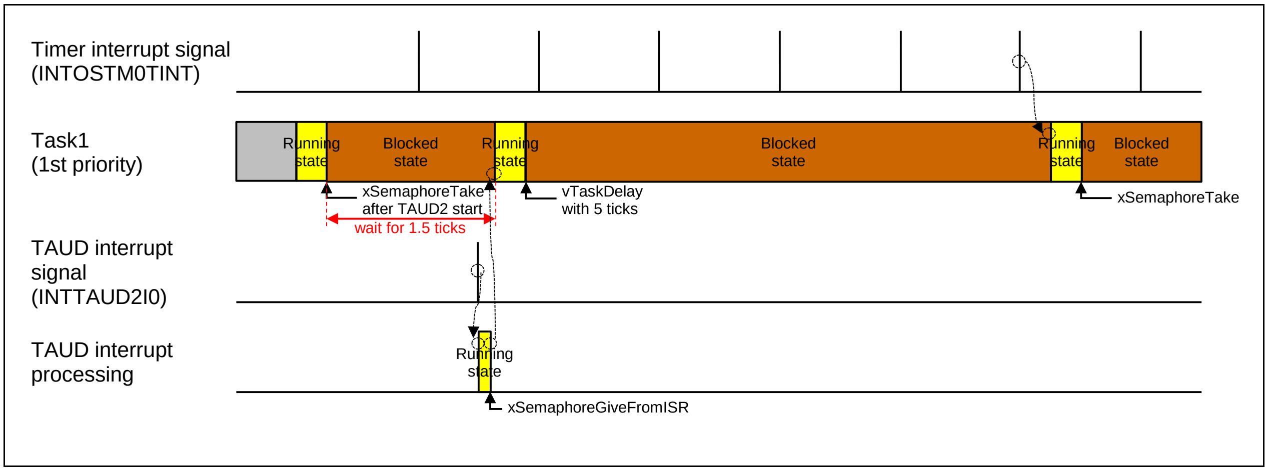


Figure 3-26 Timing chart of Binary Semaphore program when Semaphore is released

There is a difference between **Figure 3-25** and **Figure 3-26**, as shown by the red arrow in these figure.

In case **Figure 3-25**, Task1 remains Blocked state until a timeout period of 3 ticks elapses after the function xSemaphoreTake executes.

In case **Figure 3-26**, since Semaphore is released by the INTTAUD2I0 interrupt handler, Task1 can take Semaphore and transition to Running state.

3.4.1.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-7**.

Table 3-7 API functions used in Binary Semaphore program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
xSemaphoreCreateBinary	Creates a binary semaphore, and returns a handle by which the semaphore can be referenced.	https://www.freertos.org/xSemaphoreCreateBinary.html
xSemaphoreTake	Macro to obtain a semaphore.	https://www.freertos.org/a00122.html
xSemaphoreGiveFromISR	Macro to release a semaphore. This macro can be used from an ISR.	https://www.freertos.org/a00124.html

(2) Main program (Function main in main.c)

Figure 3-27 is the flowchart of the main program that creates one Binary Semaphore and one task. TAUD2 ch.0, which releases Semaphore, starts by Task1, so main function only performs the initial settings for TAUD2 ch.0, and do not start it.

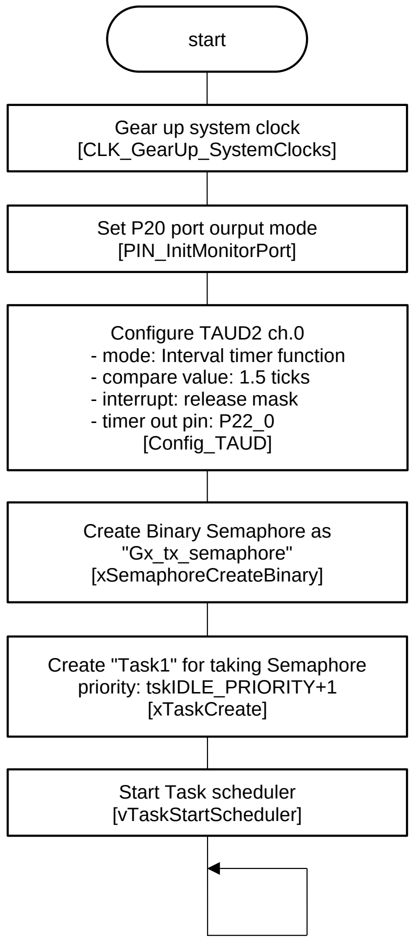


Figure 3-27 Flowchart of main of Binary Semaphore program

(3) Main program for Task1 (Function Task1_main in main.c)

Figure 3-28 is the flowchart of the main program of Task1. After setting P20_1 to high level output, Task1 starts the TAUD2 counter if the number of executions of Task1's loop processing is an even number. After that, Task1 requests to take the Semaphore regardless of the number of executions of Task1's loop processing. After the Semaphore is given or the timeout period of 3 ticks has elapsed, sets P20_1 to low level output and then execute the function vTaskDelay for 5 ticks.

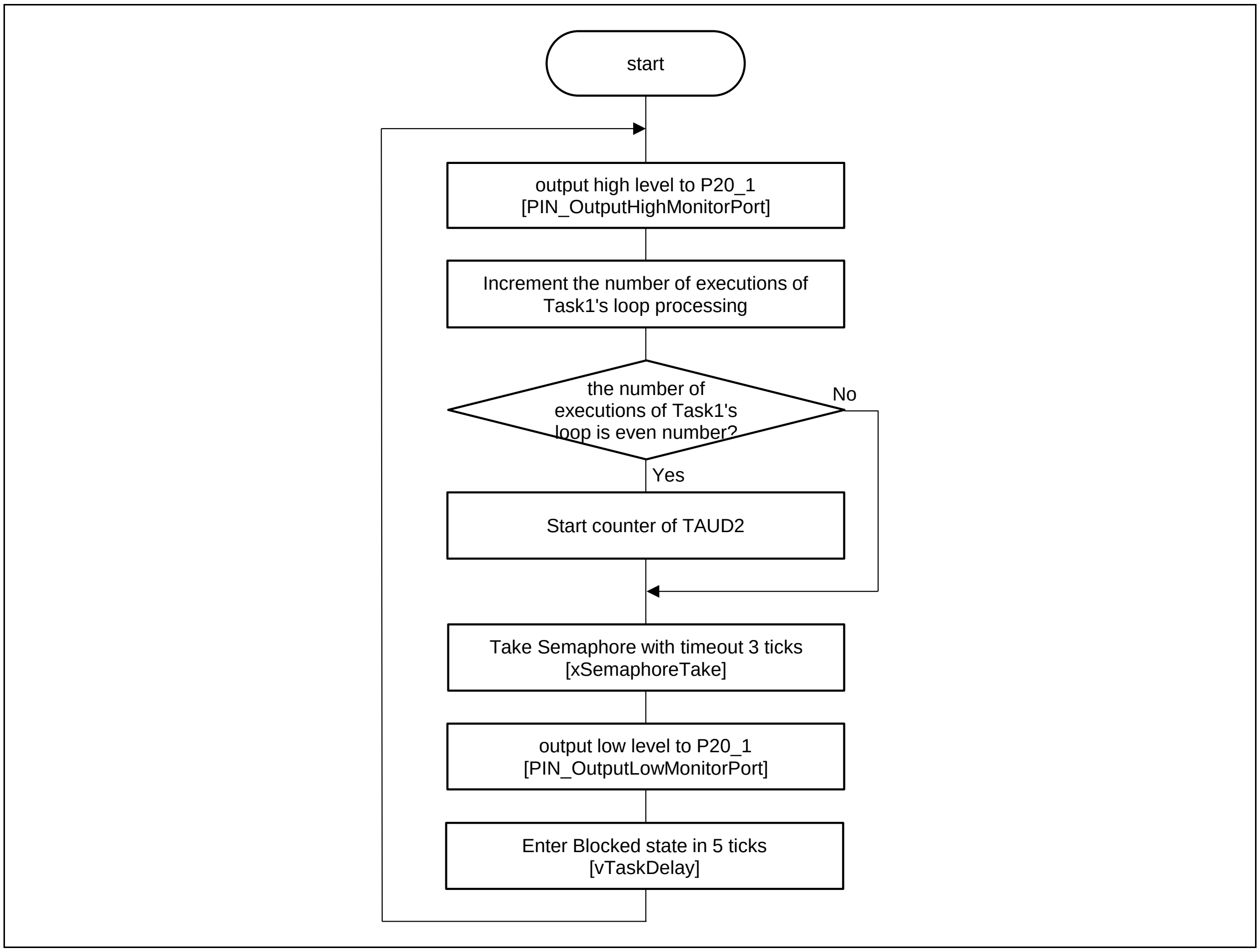


Figure 3-28 Flowchart of Task1 of Binary Semaphore program

(4) ISR processing (Function Intfunc_INTTAUD2I0 in main.c)

Figure 3-29 is the flowchart of the program for INTTAUD2I0 interrupt. This program outputs the TAUD2O0 pin to low level after stopping the TAUD2 counter. Then take the Semaphore and switches task using the portYIELD_FROM_ISR function.

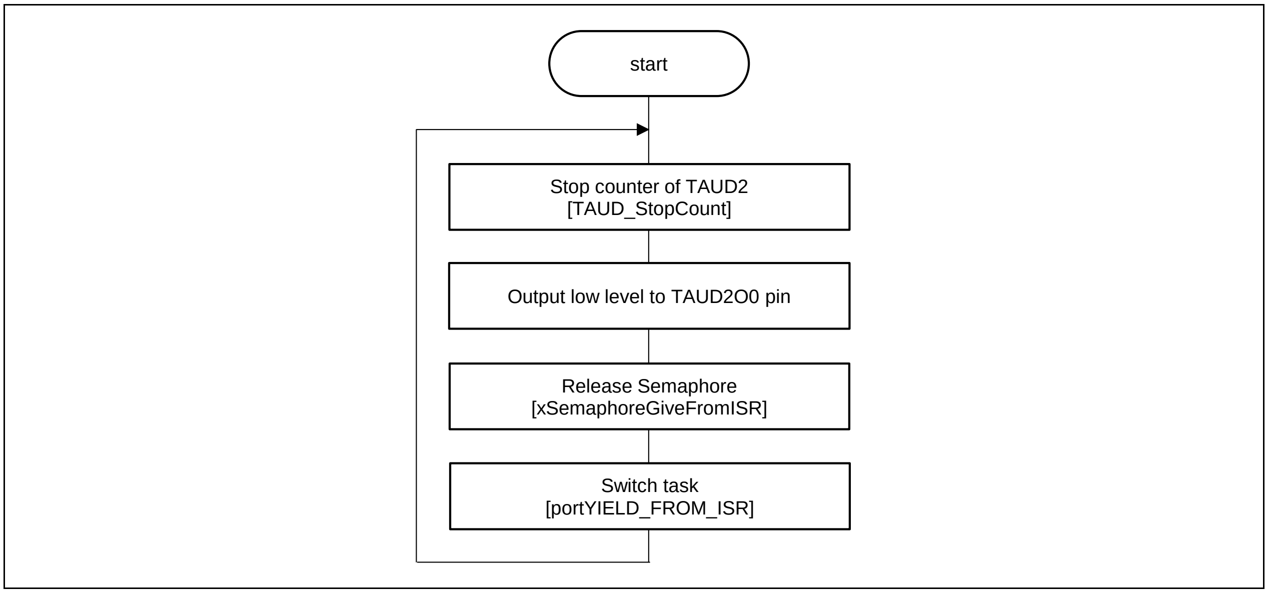


Figure 3-29 Flowchart of ISR of Binary Semaphore program

3.4.1.3 Operation result

Figure 3-30 and **Figure 3-31** show the operation result of this sample program. The meaning of each port are as follows:

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: When Task1 is in a Blocked state to request Semaphore, this port is a high level output.
  When Task1 is in a Blocked state due to execute function vTaskDelay, this port is a low level output.

(1) If the Semaphore is not released

Figure 3-30 shows the execution results of a sample program focusing on the timing when INTTAUD2I0 does not occur.

Since INTTAUD2I0 is not raised, the function xSemaphoreGive is not executed and Task1 is in Blocked state until the function xSemaphoreTake times out. This period is about 3ms from (1) to (2) in **Figure 3-30**.

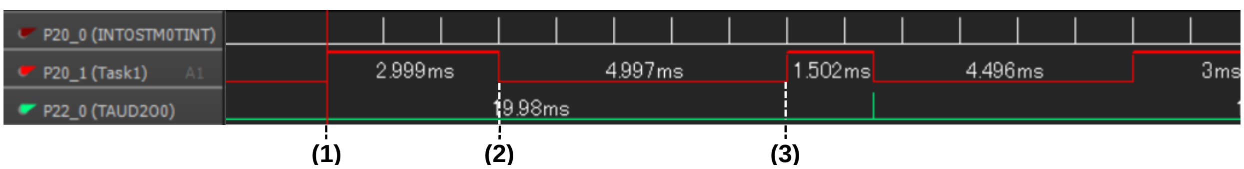


Figure 3-30 Operation waveform of Binary Semaphore program when Semaphore is not released

(1) After setting P20_1 to high level output, Task1 requests the Semaphore take with the timeout of 3 ticks by using the function xSemaphoreTake, and transitions to the Blocked state. Since the initial state of Semaphore after creation is in an empty state, Task1 cannot obtain the Semaphore at this time.

(2) After the third tick interrupt occurs from (1), Task1 returns from the Blocked state to the Running state. After that, Task1 transitions to the Blocked state for 5 ticks using the function vTaskDelay after setting P20_1 to low level output..

(3) After the fifth tick interrupt occurs from (2), Task1 returns from the Blocked state to the Running state.

(2) If the Semaphore is released

Figure 3-31 shows the execution results of a sample program focusing on the timing when INTTAUD2I0 releases the Semaphore to Task1. The handling of the INTTAUD2I0 interrupt that occurs 1.5 ticks after Task1 requests the Semaphore executes the function xSemaphoreGive.

The difference with **Figure 3-30** and **Figure 3-31** is that in **Figure 3-30** it takes 3 ticks between (1) and (2), while in **Figure 3-31** it takes 1.5ms between (3) and (4).

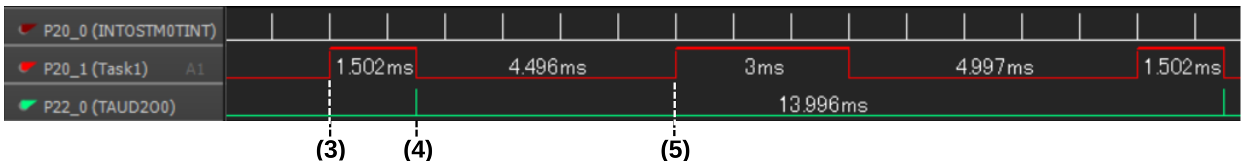


Figure 3-31 Operation waveform of Binary Semaphore program when Semaphore is released

(3) After setting P20_1 to high level output, Task1 requests the Semaphore take with the timeout of 3 ticks by using the function xSemaphoreTake, and transitions to the Blocked state.

(4) When the INTTAUD2I0 interrupt occurs, the function xSemaphoreGive is executed in the INTTAUD2I2 interrupt proessing. After that, Task1 transitions to Running state.

Task1 transitions to the Blocked state for 5 ticks using the function vTaskDelay after making P20_1 a low level output.

(5) After the fifth tick interrupt occurs from (2), Task1 returns from the Blocked state to the Running state and becomes the state shown in (1) of **Figure 3-30**.

3.4.2 Counting Semaphores

3.4.2.1 Overview

This sample program uses the Counting Semaphore that is incremented by the ISR to synchronize the Task and interrupt processing by changing the flow of the Task when the value of Counting Semaphore reaches a certain value. TAUD2 ch.0 is used as ISR.

Use the function xSemaphoreCreateBinary to create one Counting Semaphore and use the function xTaskCreate to create one task, Task1.

When the INTTAUD2I0 interrupt occurs, increment the value of Counting Semaphore in the INTTAUD2I0 interrupt handler.

Task1 uses the function uxSemaphoreGetCount to get the value of Counting Semaphore. If the obtained value is less than or equal to 4, execute the function vTaskDelay for 1 tick. If it is 5 or more, execute the function vTaskDelay for 5 ticks.

Task2 repeats the transition between the Running state and the Blocked state for 2 ticks.

Figure 3-32 shows the operation timing of this sample program. Counting Semaphore is incremented by INTTAUD2I0 interrupt processing, and when the value of Counting Semaphore becomes 4 or more, the period of Blocked state of Task1 changes to 5 ticks.

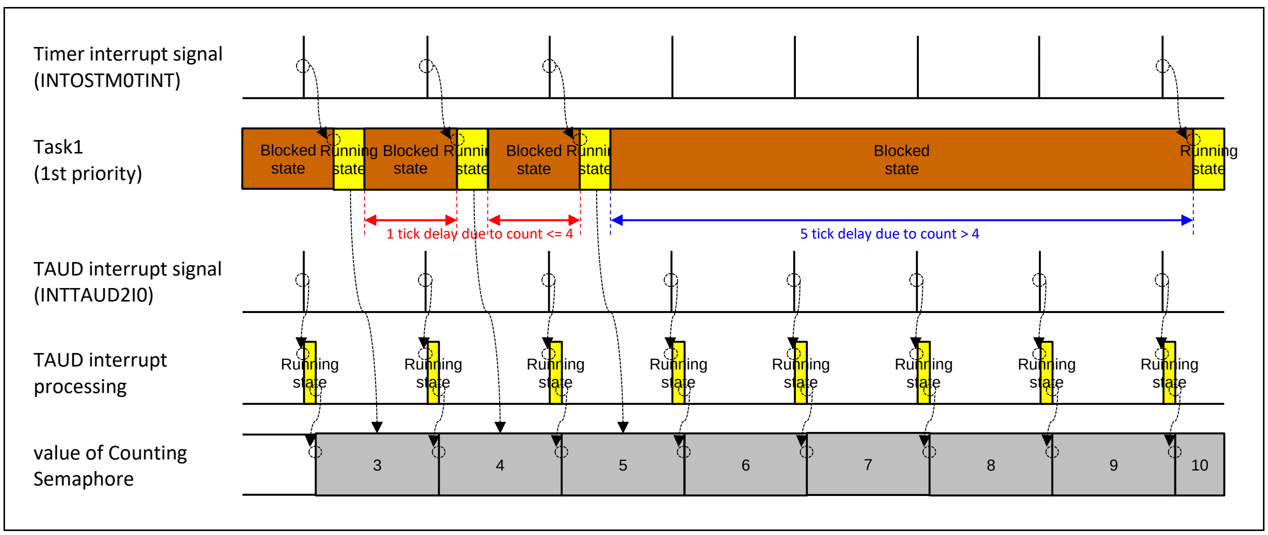


Figure 3-32 Timing chart of Counting Semaphore program

3.4.2.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-8**.

Table 3-8 API functions used in Counting Semaphore program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
xSemaphoreCreateCounting	Creates a counting semaphore and returns a handle by which the newly created semaphore can be referenced.	https://www.freertos.org/CreateCounting.html
xSemaphoreGiveFromISR	Macro to release a semaphore. This macro can be used from an ISR.	https://www.freertos.org/a00124.html
uxSemaphoreGetCount	Returns the count of a semaphore.	https://www.freertos.org/uxSemaphoreGetCount.html

(2) Main program (Function main in main.c)

Figure 3-33 is the flowchart of the main program that creates one Counting Semaphore and one task. After creating the Counting Semaphore and task, start the TAUD2 ch.0 counter and the scheduler.

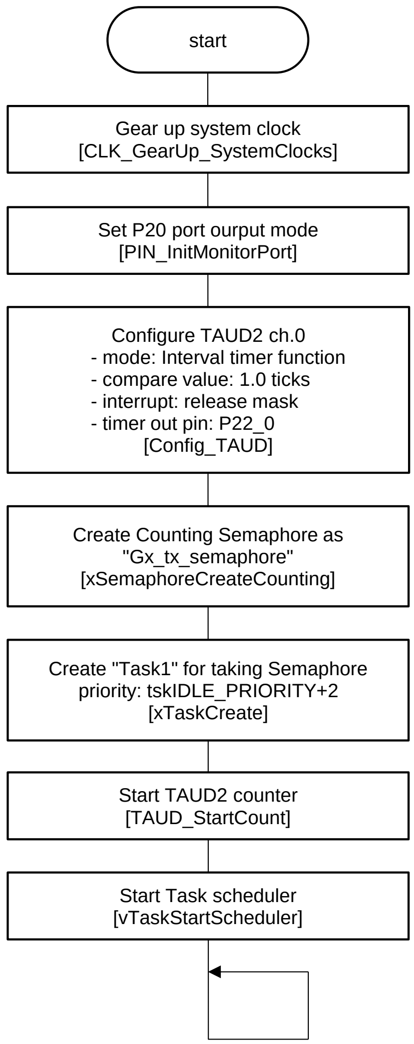


Figure 3-33 Flowchart of main of Counting Semaphore program

(3) Main program for Task1 (Function Task1_main in main.c)

Figure 3-34 is the flowchart of the main program for Task1. After setting P20_1 to high level output, Task1 gets the value of Counting Semaphore, and changes the pulse pattern of P20_1 and the delay value of vTaskDelay according to the got value.

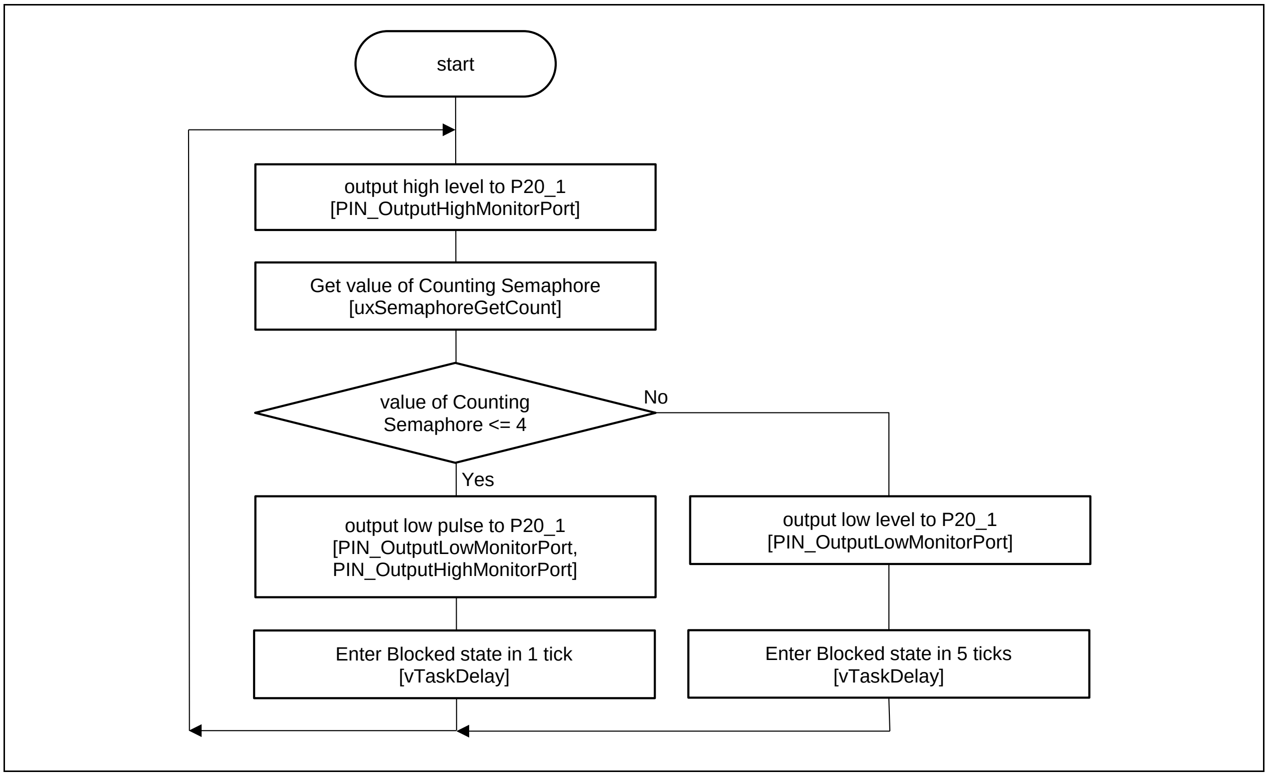


Figure 3-34 Flowchart of Task1 of Counting Semaphore program

(4) ISR processing (Function Intfunc_INTTAUD2I0 in main.c)

Figure 3-35 is the flowchart of the program for the INTTAUD2I0 interrupt. This program increments the value of Counting Semaphore, gets the value of Semaphore, and outputs it to the port. Then switches Task using the portYIELD_FROM_ISR function.

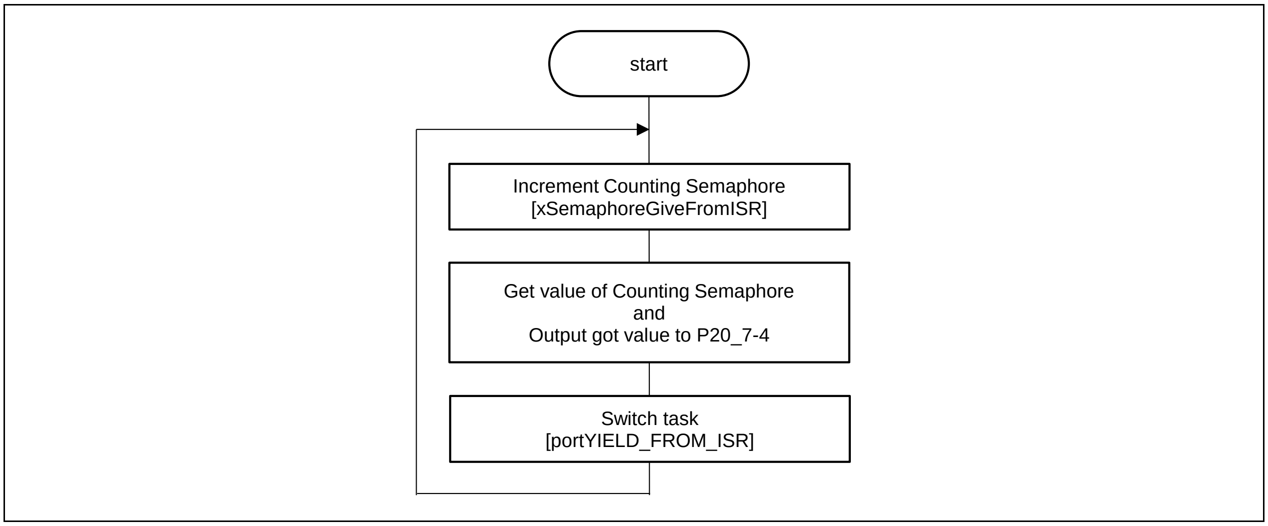


Figure 3-35 Flowchart of ISR of Counting Semaphore program

3.4.2.3 Operation result

Figure 3-36 shows the execution result of this sample program.

It can be seen that when the Count value is 5 or more, the waveform pattern of P20_1 output by Task1 has changed.

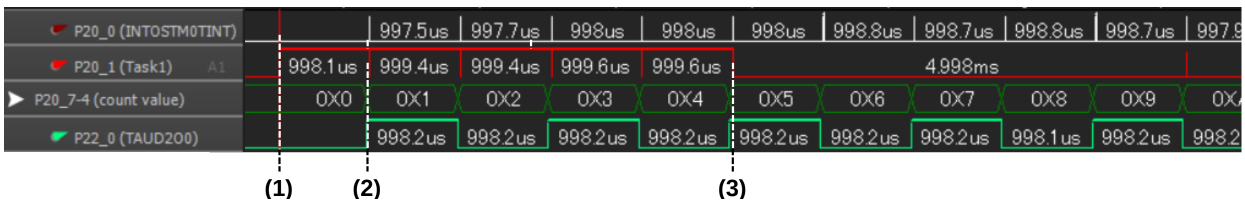


Figure 3-36 Operation waveform of Counting Semaphore program

About **Figure 3-36**, The rising edge and falling edge of TAUD2O0 indicates the occurrence timing of an INTTAUD2I0 interrupt. When the INTTAUD2I0 interrupt occurs, the value of Counting Semaphore is incremented by 1.

(1) Task1 sets P20_1 to high level output, then uses the function uxSemaphoreGetCount to get the value of Counting Semaphore and checks the obtained value. At this time, the value of Counting Semaphore was 0, so after outputting a low pulse to P20_1, Task1 transitions to the Blocked state for 1 tick using the function vTaskDelay.

(2) After the first tick interrupt occurs from (1), Task1 returns from the Blocked state to the Running state. After that, get the value of Counting Semaphore in the same way as in (1) and check the obtained value. At this time, the value of Counting Semaphore was 1, so after outputting a low pulse to P20_1, Task1 transitions to the Blocked state for 1 tick using the function vTaskDelay.

(3) Get the value of Counting Semaphore in the same way as in (1) and check the obtained value. As a result, the value of Counting Semaphore was 4, so after outputting a low level to P20_1, Task1 transitions to the Blocked state for 5 ticks using the function vTaskDelay.

3.4.3 Mutexes

3.4.3.1 Overview

This sample program shows how two tasks can take exclusive control of a hardware resource using Mutex.

Two tasks attempt to obtain the Mutex using the function xSemaphoreTake before using shared hardware resources. If the Task obtained the shared hardware resources.

In this sample program, the shared hardware resource is RLIN30. Task that obtained the Mutex uses RLIN30 to send messages specific to each task.

After the sending process is complete, use the function xSemaphoreGive to release the Mutex.

Figure 3-37 shows an image where two tasks use a shared resource exclusively using a Mutex. Shared resources can only be accessed by the Task that has obtained the Mutex. After using a shared resource, Task can release the Mutex and other tasks can obtain the Mutex.

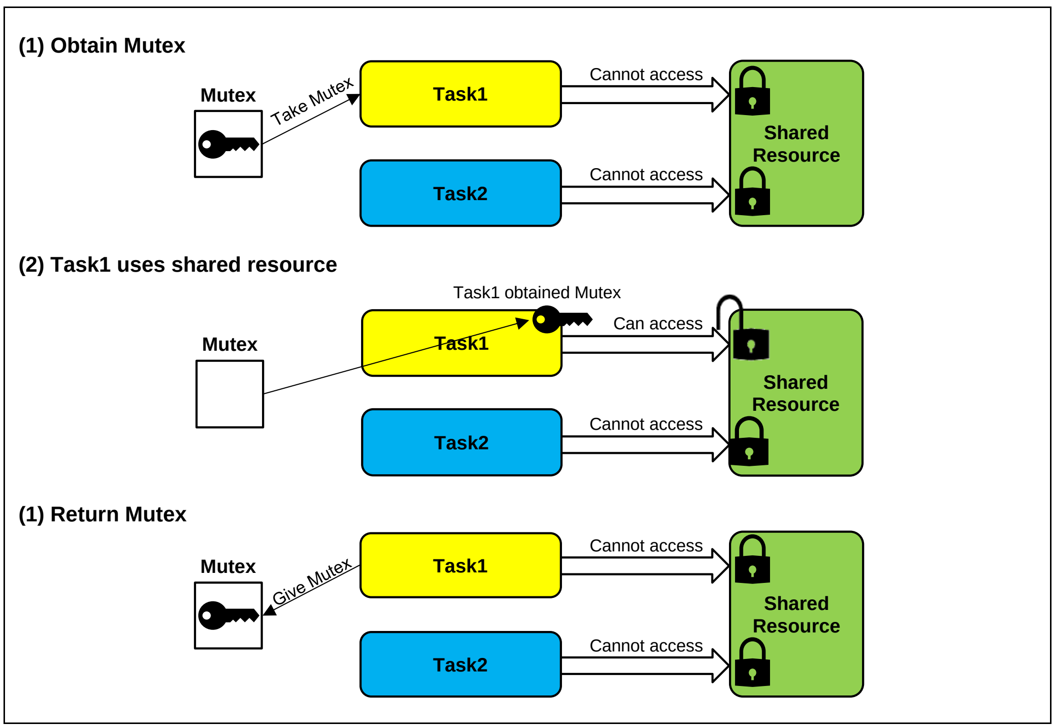


Figure 3-37 Image of exclusive access using Mutex

3.4.3.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-9**.

Table 3-9 API functions used in Mutex program

Function name	Description	Link to FreeRTOS official site
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
xSemaphoreCreateMutex	Creates a mutex, and returns a handle by which the created mutex can be referenced.	https://www.freertos.org/CreateMutex.html
xSemaphoreTake	Macro to obtain a semaphore.	https://www.freertos.org/a00122.html
xSemaphoreGive	Macro to release a semaphore.	https://www.freertos.org/a00123.html
vTaskDelay	Delay a task for a given number of ticks.	https://www.freertos.org/a00127.html

(2) Main program (Function main in main.c)

Figure 3-38 is the flowchart of the main program that creates one Mutex and two tasks. Set RLIN30 to UART mode within this program. Task1 and Task2 both execute the function Tasks_main. However, the value of the parameter *pvParameters, which is the message sent by RLIN30 with this function, is different between Task1 and Task2.

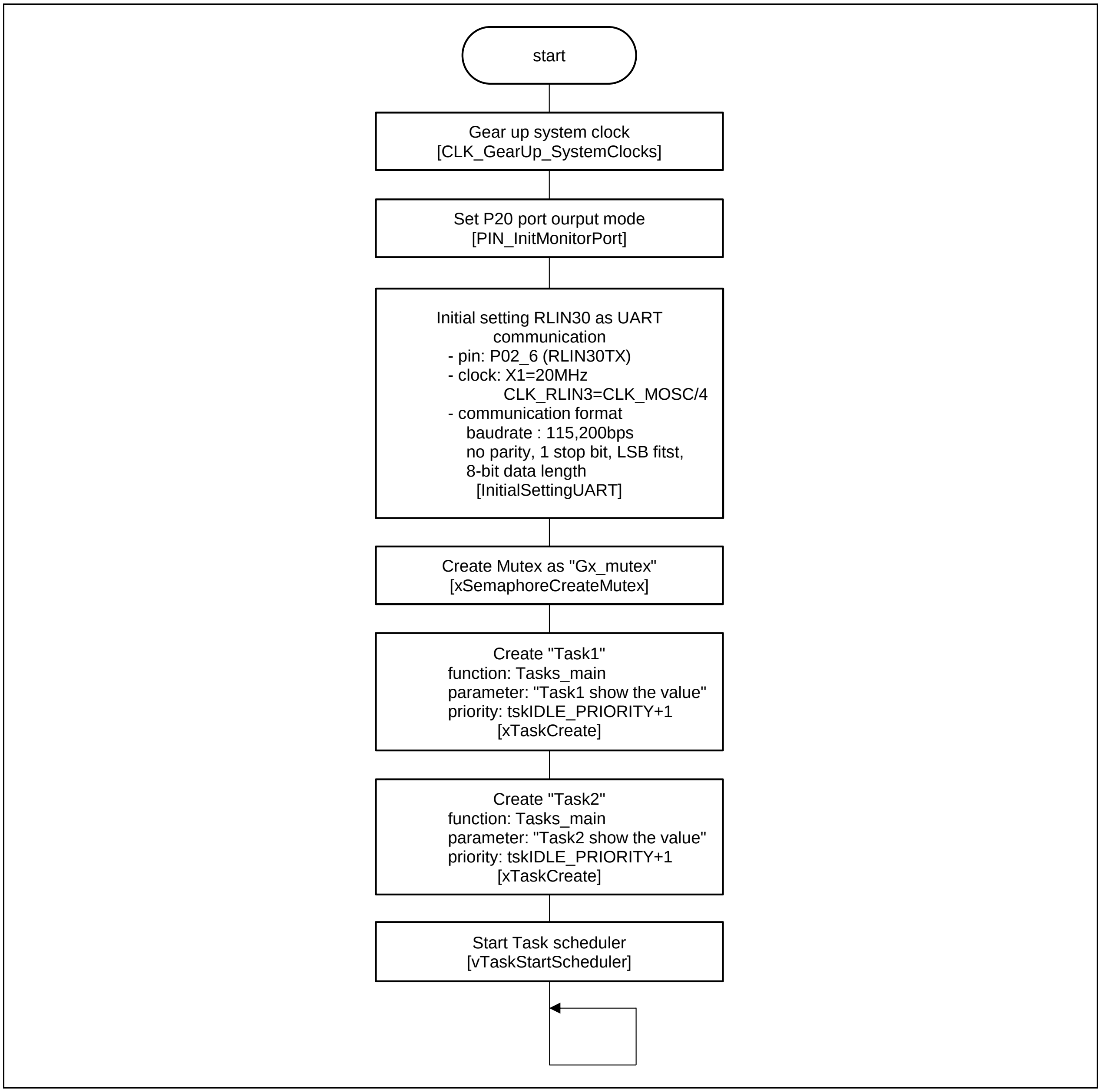


Figure 3-38 Flowchart of main of Mutex program

(3) Main program for Task1 and Task2 (Function Task1_main and Task2_main in main.c)

Figure 3-39 is the flowchart of the main program of each task.

When one task obtained the Mutex, it outputs a message to RLIN30, then releases the Mutex.

To check which Task is holding the Mutex, P20_1 will be output at high level while Task1 is holding the Mutex, and P20_2 will be output at high level while Task2 is holding the Mutex.

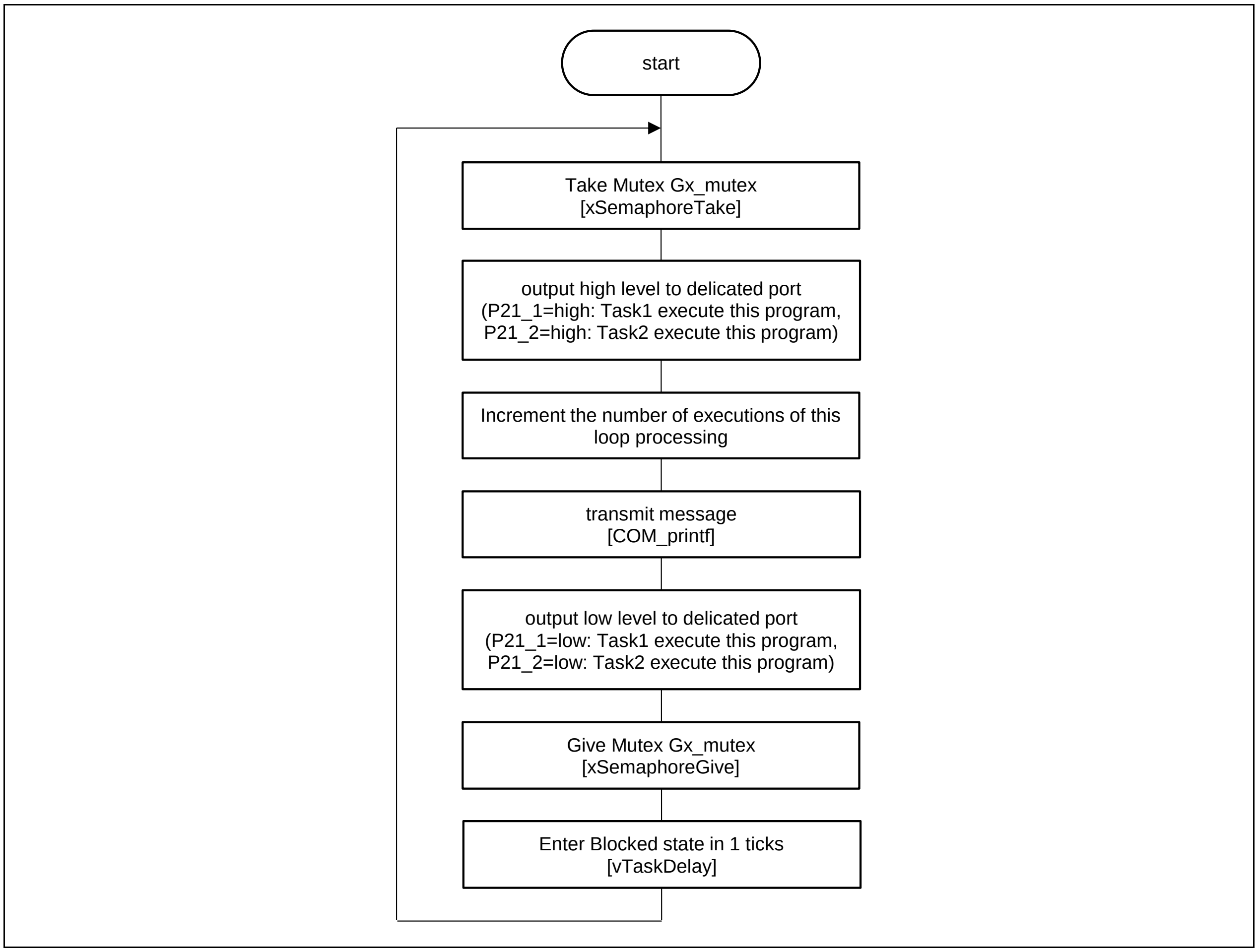


Figure 3-39 Flowchart of each task of Mutex program

3.4.3.3 Operation result

(1) In case of using Mutex for exclusive control

Figure 3-40 shows the message sent by RLIN30 as a result of running this sample program. It can be seen that messages sent by Task1 and messages sent by Task2 are displayed alternately.

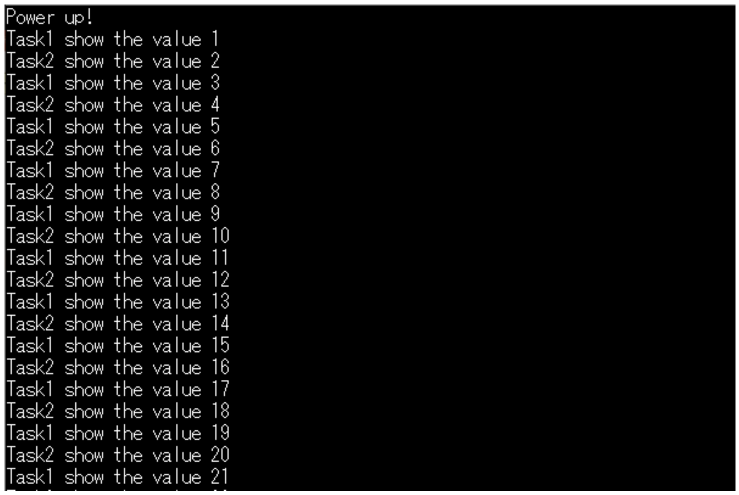


Figure 3-40 Operation log of Mutex program

Figure 3-41 shows the output status of P20_1 and P20_2 when this sample program is executed. The meaning of each port are as follows:

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: While Task1 holds Mutex and outputs a message on RLIN30, this port is a high level output.
- P20_2: While Task2 holds Mutex and outputs a message on RLIN30, this port is a high level output.

Since the high pulses of P20_1 and P20_2 appear alternately without overlapping, it can be seen that Task1 and Task2 hold the Mutex alternately.

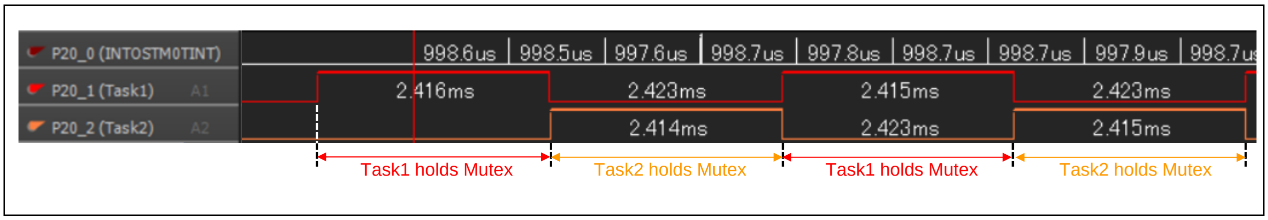


Figure 3-41 Operation waveform of Mutex program

(2) In case of not using Mutex for exclusive control

As shown in **Figure 3-42**, delete the Mutex take and give descriptions from the function Tasks_main and try running a program in which two tasks execute one program without exclusive control by Mutex.

In this sample program, if you comment out USE_MUTEX defined in main.c, the flow will be **Figure 3-42**.

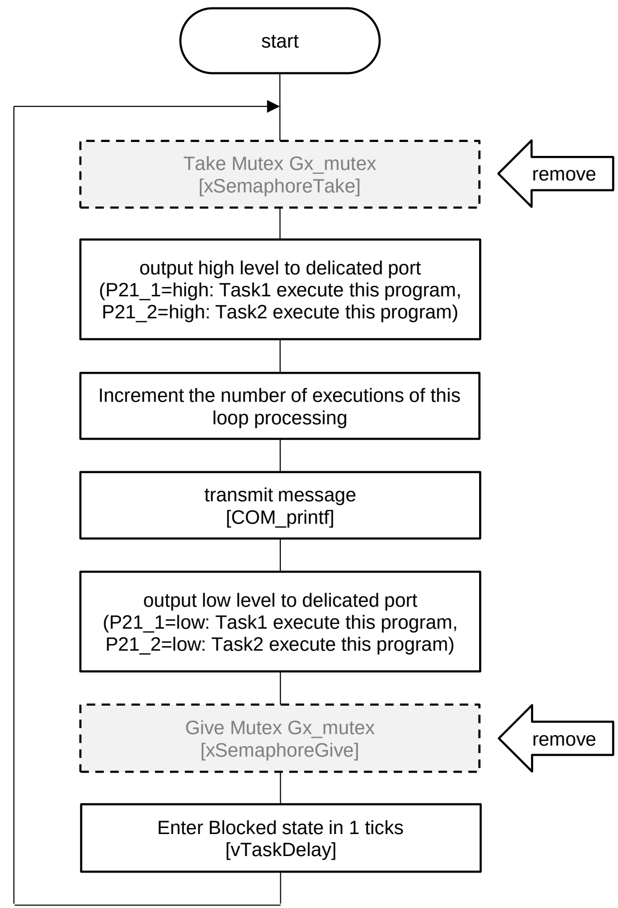


Figure 3-42 Flowchart of each task of Mutex program without using Mutex

Figure 3-43 shows the message sent by RLIN30 as a result of running the sample program without exclusive control by Mutex. It can be seen that the messages sent by Task1 and the messages sent by Task2 are mixed.

```
Power up!
Task1 show Task2 showthe value 1 the value
Task1 show  2
Task2 show the value the value 43
Task1 show
Task2 show the value the value 5
Task1 show 6
Task2 show the value the value 7
Task1 show 8
Task2 show the value the value 9
Task1 show 10
Task2 show the value the value 111
Task1 show 2
Task2 show the value the value 113
Task1 show 4
Task2 show the value the value 115
Task1 show 6
Task2 show the value the value 17
Task1 show 18
Task2 show the value the value 19
Task1 show 20
Task2 show the value the value 21
```

Figure 3-43 Operation log of Mutex program that does not use Mutex

NOTE

If the UART communication time is short compared to the Tich interrupt cycle, the message will be displayed normally because the task will not be switched during the processing of each task.

Depending on the following conditions, the message may be displayed correctly.

- UART baud rate
- Message length
- Value of vTaskDelay after sending message
- Tick interrupt cycle
- others

Use waveforms to analyze the phenomenon where the messages sent by Task1 and the messages sent by Task2 are mixed.

Figure 3-44 shows the output status of P20_1 and P20_2 when running the sample program without exclusive control by Mutex. The high level output of P20_1 and P20_2 overlap indicates that the task is switched while RLIN30 is sending a message.

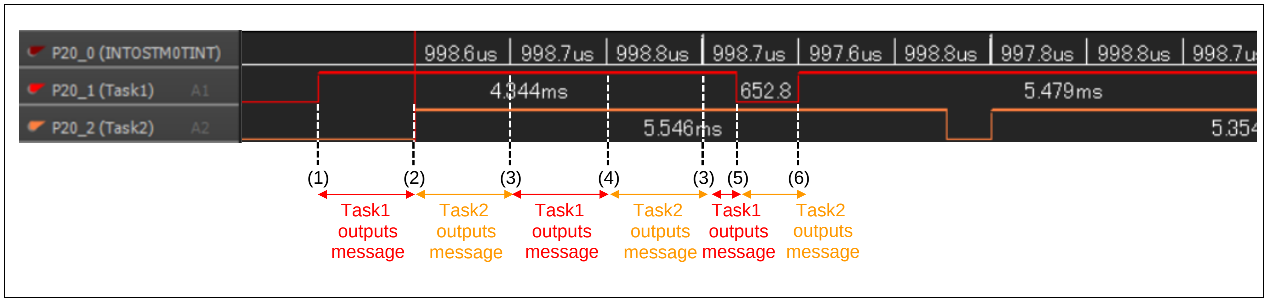


Figure 3-44 Operation waveform of Mutex program that does not use Mutex

(1) Task1 starts sending message.

(2) After 1 tick has passed, the scheduler suspends the processing of Task1, and Task2 starts sending message.

(3) After 1 tick has elapsed, the scheduler suspends the processing of Task2, and Task1 resumes sending message.

(4) After 1 tick has elapsed, the scheduler suspends the processing of Task1, and Task2 resumes sending message.

(5) After completing sending the message, Task1 will be in the Blocked state until the next tick with vTaskDelay. At this time, Task2 transitions to Running state and resumes sending message.

(6) When the next tick interrupt occurs, the scheduler suspends the processing of Task2, and Task1 starts sending the next message.

3.4.4 Gatekeeper Tasks

3.4.4.1 Overview

This sample program shows how two tasks can use one hardware resource via a Gatekeeper Task.

Only one task can use shared hardware resources. The only task that has the right to use shared hardware resources is called the Gatekeeper Task. This sample program uses RLIN30 as a shared hardware resource.

Queue is used to send/receive data between each task and Gatekeeper Task. Each task writes messages to the Queue to send. The Gatekeeper Task monitors the Queue and transmits the message on RLIN30 when a message arrives on the Queue.

Figure 3-45 shows two tasks sending messages via the Gatekeeper Task. RLIN30, which is a shared resource, can only be accessed by Gatekeeper Task, so exclusive control by Mutex or Semaphore is not required.

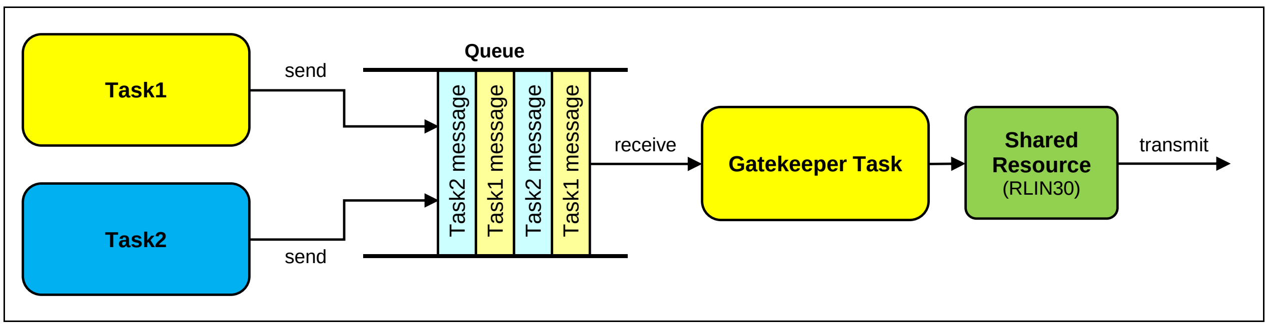


Figure 3-45 Image of messages transmission via Gatekeeper Task

3.4.4.2 Program

(1) API function

About description of the API functions of FreeRTOS used in this sample program, refer to the FreeRTOS official site shown **Table 3-10**.

Table 3-10 API functions used in Gatekeeper Task program

Function name	Description	Link to FreeRTOS official site
xQueueCreate	Creates a new queue and returns a handle by which the queue can be referenced.	https://www.freertos.org/a00116.html
xTaskCreate	Create a new task and add it to the list of tasks that are ready to run.	https://www.freertos.org/a00125.html
vTaskStartScheduler	Starts the RTOS scheduler.	https://www.freertos.org/a00132.html
xQueueSendToBack	Post an item to the back of a queue.	https://www.freertos.org/xQueueSendToBack.html
vTaskDelay	Delay a task for a given number of ticks.	https://www.freertos.org/a00127.html
xQueueReceive	Receive an item from a queue.	https://www.freertos.org/a00118.html

(2) Main program (Function main in main.c)

Figure 3-46 is the flowchart of the main program that creates one Queue and three tasks. Gatekeeper Task is created using the function xTaskCreate like Task1 and Task2.

Set RLIN30 to UART mode within this program.

Task1 and Task2 both execute the function Tasks_main. Gatekeeper Task executes the function GatekeeperTask_main.

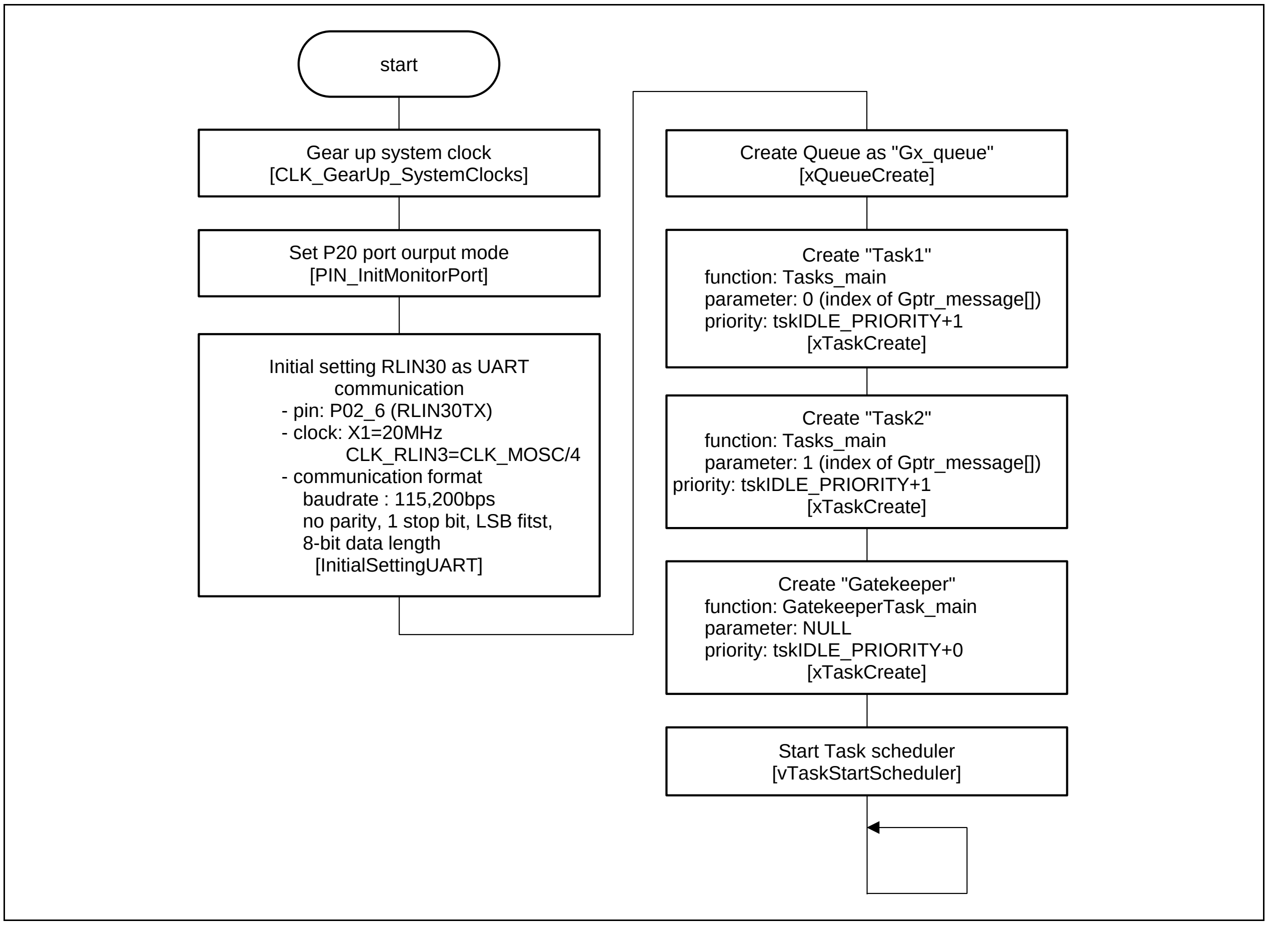


Figure 3-46 Flowchart of main of Gatekeeper Task program

(3) Main program for Task1 and Task2 (Function Task1_main and Task2_main in main.c)

Figure 3-47 is the flowchart of the main program of each task.

Each Task attempts to send a message to the Queue after setting its corresponding monitor port to high level output. If the Queue is full, Task will wait until it is no longer full.

After finishing sending messages to the Queue, set the monitor port to low level output and execute the function vTaskDelay to switch tasks.

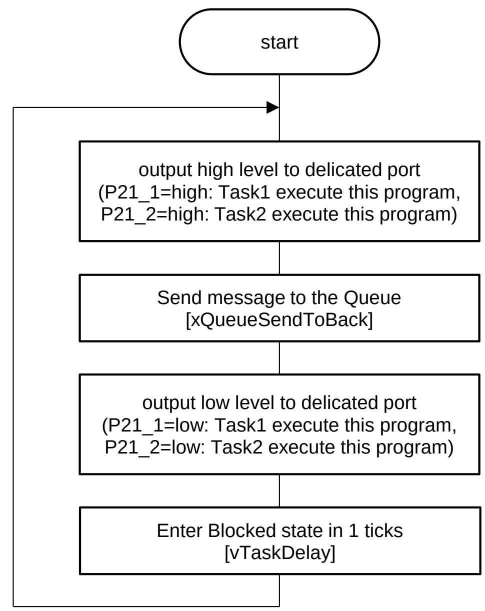


Figure 3-47 Flowchart of each task of Gatekeeper Task program

(4) Main program for Gatekeeper Task (Function GatekeeperTask_main in main.c)

Figure 3-48 is the flowchart of the Gatekeeper Task main program.

The Gatekeeper Task waits until a message arrives on the Queue.

When a message arrives at the Queue, Gatekeeper Task transmits the message via RLIN30 after setting P20_3 to high level output.

After the transmission is complete, Gatekeeper Task sets P20_3 to low level output and wait for the next message to arrive in the Queue.

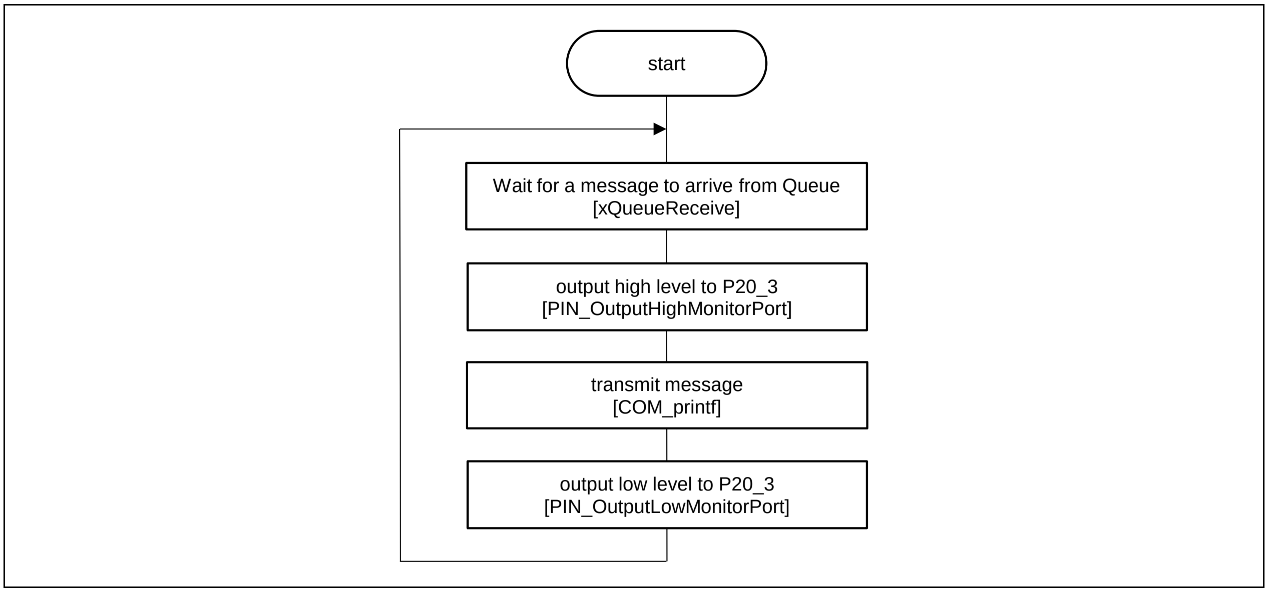


Figure 3-48 Flowchart of Gatekeeper Task of Gatekeeper Task program

3.4.4.3 Operation result

Figure 3-49 shows the message sent by RLIN30 as a result of running this sample program. It can be seen that messages sent by Task1 and messages sent by Task2 are displayed alternately.

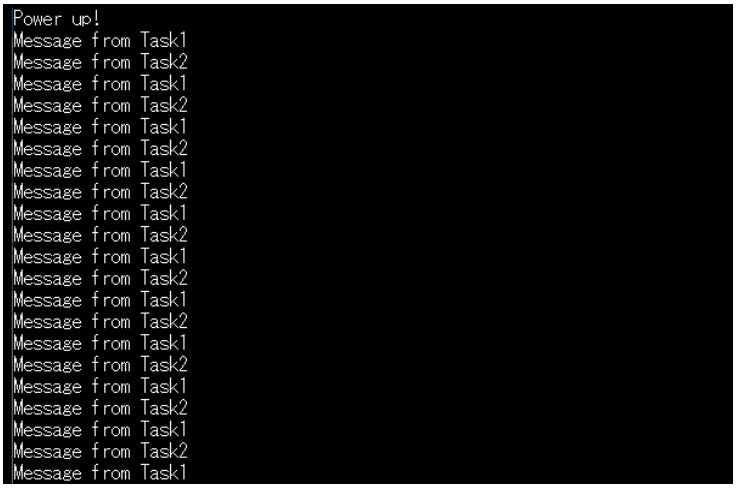


Figure 3-49 Operation log of Gatekeeper Task program

Figure 3-50 shows the operation timing of each task when this sample program is executed.

P20_1 and P20_2 show the timing of sending messages to the Queue of Task1 and Task2, respectively, and P20_3 shows the timing of transmitting messages using RLIN30 by Gatekeeper Task.

The meaning of each port are as follows:

- P20_0: Generation timing of the tick interrupt INTOSTM0TINT
- P20_1: When Task1 tries to transmit message to Queue, this port is a high level output.
  When Task1 completed to transmit message to Queue, this port is a low level output.
- P20_2: When Task2 tries to transmit message to Queue, this port is a high level output.
  When Task2 completed to transmit message to Queue, this port is a low level output.
- P20_3: When Gatekeeper Task received transmit message from Queue, this port is a high level output.
  When Gatekeeper Task completed to transmit message by RLIN30, this port is a low level output.

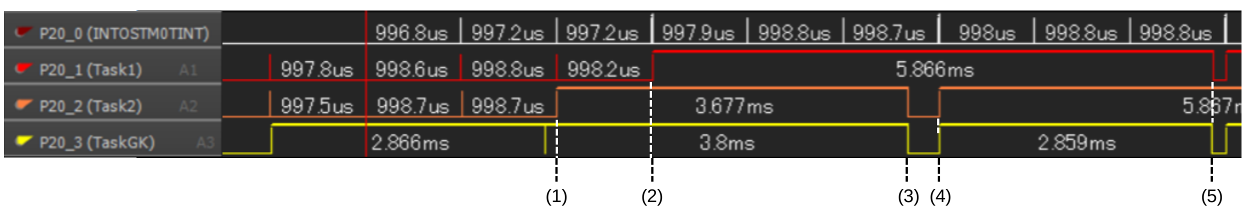


Figure 3-50 Operation waveform of Gatekeeper Task program

The continued high level output at (2) in P20_1, and at (1) and (4) in P20_2 indicates that when the Queue becomes full, Task2 and Task1 are waiting until there is space in the Queue.

The negative edge of (3) and (5) in P20_3 indicates that the Gatekeeper Task has finished transmitting one message. After this, the Gatekeeper Task will receive the messages from the Queue, so there will be space in the Queue.

When the Queue become not full, Task2 sends a message to the Queue at timing (3), and Task1 sends a message to the Queue at timing (5).

REVISION HISTORY

Revision	Description	Date
Rev.1.00	New release	2023.10.27

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit 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. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity.

   Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time 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 time 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 time 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 time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to power supply or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.
5. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
6. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
7. Power ON/OFF sequence

   In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply

   after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current. The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.