RX23T, RX24T, RX66T, and RX72M Groups

Using the Driver (Rev. 2.00) for Resolver-to-Digital Converter Control

Introduction

This application note describes how to use the driver (Rev. 2.00) to control the resolver-to-digital converter IC (RDC). This driver is an upgraded version of the driver library supplied together with Rev. 1.20 of the application note "Using the Driver for Resolver-to-Digital Converter Control" for the RX24T group.

Target Devices

- RX23T (R5F523T5ADFM)
- RX24T (R5F524TAADFM)
- RX66T (R5F566TKCDFB)
- RX72M (R5F572MNDDFC)
- RDCs (RAA3064002GFP and RAA3064003GFP)

Contents

1. Overview

1.1 Functions of the Driver

This driver has the following functions.

- RDC settings
- Output of the RDC operating clock
- Communications between the RDC and MCU
- Output of the excitation reference signal
- Input of the angle signal
- Detection of disconnection from the resolver sensor
- Deassertion of the ALARM# signal
- Output of the phase adjustment signals
- Output of the angle error correction signal
- Automatic calibration of errors

1.2 Development Environment

Table 1-1 shows the environment in which operations of this driver have been verified.

Table 1-1 Software Development Environment

IDE Version	Toolchain	Smart Configurator
CS+: V8.03.00 e^2 studio: V2020-10	CC-RX V3.02.00	Version: 2.7.0

1.3 Program Size

Table 1-2 shows the program size of this driver.

Table 1-2 Program Size

ROM Size	RAM Size	Size of Stack Area Used
12298 bytes	1067 bytes	164 bytes

1.4 Related Documents

RX23T Group User's Manual: Hardware (R01UH0520)

RX24T Group User's Manual: Hardware (R01UH0576)

RX66T Group User's Manual: Hardware (R01UH0749)

RX72M Group User's Manual: Hardware (R01UH0804)

RX Smart Configurator User's Guide: e^2 studio (R20AN0451)

RX Smart Configurator User's Guide: CS+ (R20AN0470)

Resolver-to-Digital Converters User’s Manual: Hardware (R03UZ0002)

2. Overall Configuration

2.1 System Configuration

Figure 2.1 shows the configuration of the system incorporating the RDC and the MCU.

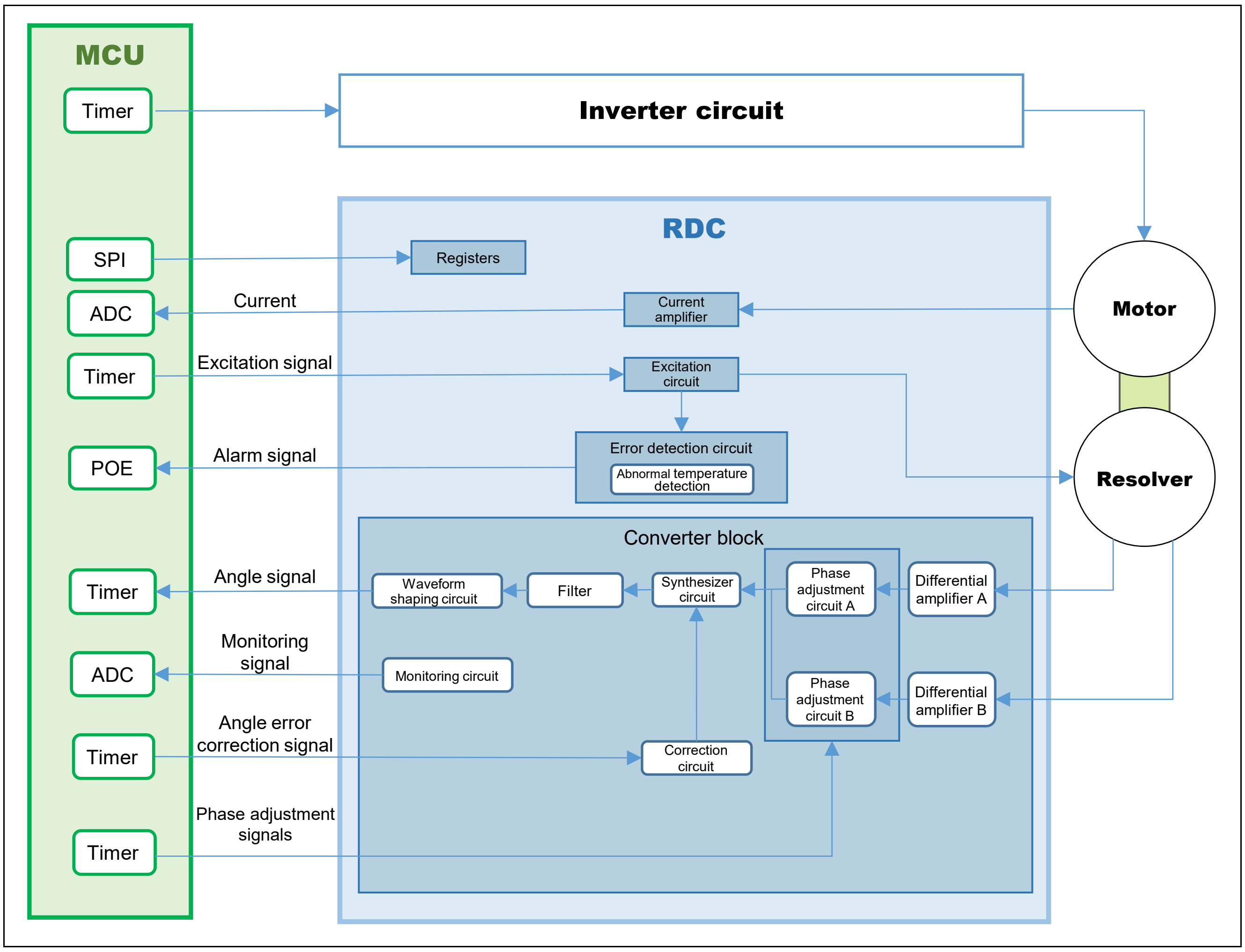


Figure 2.1 Configuration of the System Incorporating RDC and MCU

2.2 RDC Functions

The RDC incorporates an excitation circuit to excite the resolver sensor and a converter block to convert an analog signal output from the resolver sensor into a digital signal.

The excitation circuit converts a rectangular wave output from the MCU to an analog signal to excite the resolver sensor.

The converter block generates an angle signal (rectangular wave) from the two-phase signals (electrical angle information) detected by the resolver sensor, and outputs the angle signal to the host MCU. A rotor angle can be obtained by using the timer of the host MCU to measure the phase difference between the rectangular excitation wave and angle signal. Furthermore, the converter block has gain adjustment, phase adjustment, and angle error correction functions.

The gain adjustment function adjusts the amplitudes of the two-phase signals of the resolver sensor to the same level according to the changes in the RDC settings.

The phase adjustment function receives the correction signals for phase adjustment output from the MCU to the RDC and adjusts the phase difference between the two-phase signals of the resolver sensor to 90 degrees.

The angle error correction function corrects analog errors of the resolver sensor. The angle error correction signal output from the MCU to the RDC is combined with the angle signal through the correction circuit in the converter block.

This driver software provides functions to output the rectangular wave signal and the correction signal from the MCU to the RDC and detect the angle signal output from the converter block.

3. Functions

This section describes the functions of the driver software.

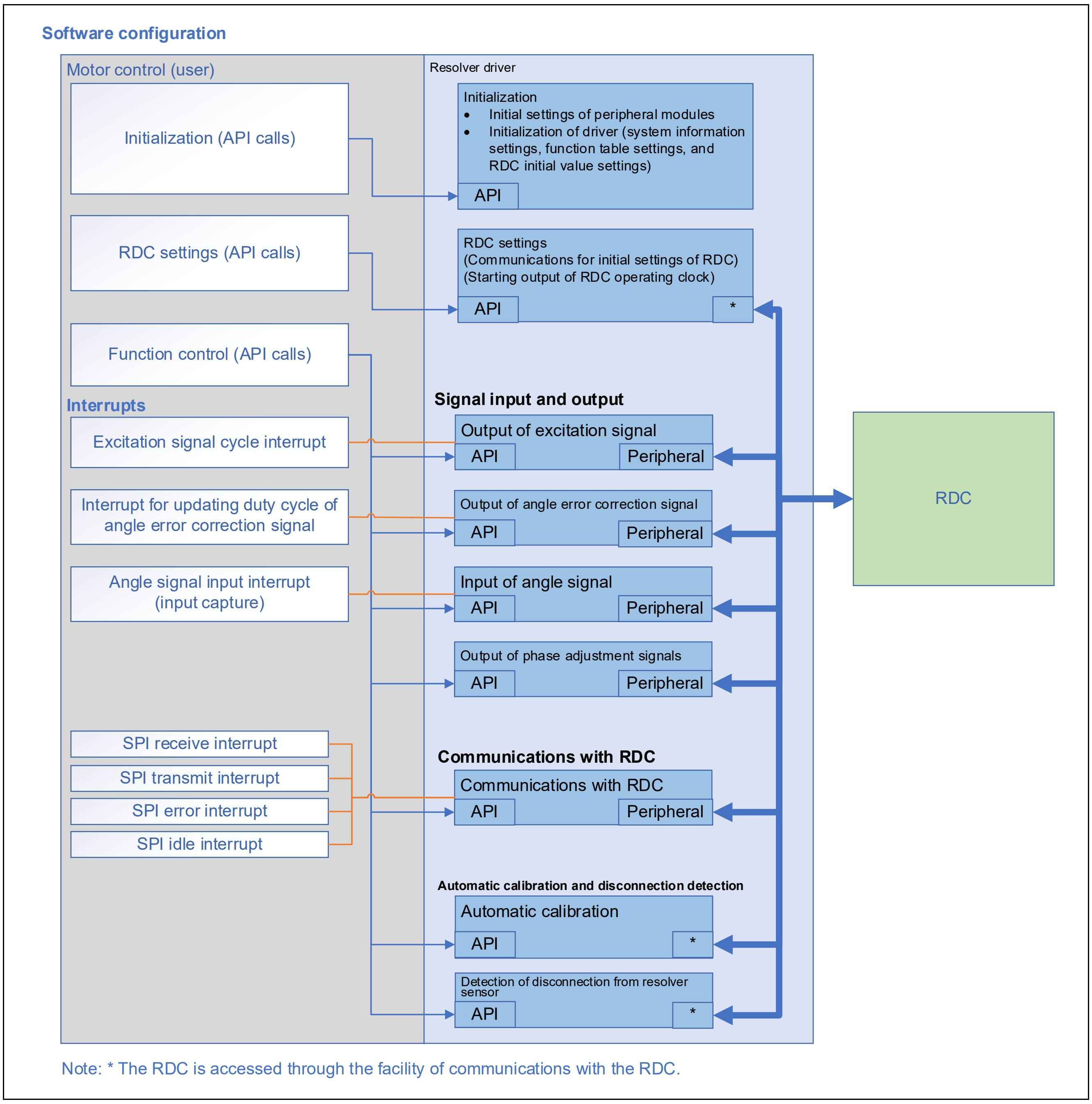

Software configuration
Motor control (user)
Initialization (API calls)
RDC settings (API calls)
Function control (API calls)
Interrupts
Excitation signal cycle interrupt
Interrupt for updating duty cycle of angle error correction signal
Angle signal input interrupt (input capture)
SPI receive interrupt
SPI transmit interrupt
SPI error interrupt
SPI idle interrupt
Resolver driver
Initialization
• Initial settings of peripheral modules
• Initialization of driver (system information settings, function table settings, and RDC initial value settings)
API
RDC settings
(Communications for initial settings of RDC)
(Starting output of RDC operating clock)
API
*
Signal input and output
Output of excitation signal
API
Peripheral
Output of angle error correction signal
API
Peripheral
Input of angle signal
API
Peripheral
Output of phase adjustment signals
API
Peripheral
Communications with RDC
Communications with RDC
API
Peripheral
Automatic calibration and disconnection detection
Automatic calibration
API
*
Detection of disconnection from resolver sensor
API
*
RDC
Note: * The RDC is accessed through the facility of communications with the RDC.

Figure 3.1 Software Configuration

3.1 Initialization of the Driver

To initialize the resolver driver, make initial settings for the peripheral modules of the MCU, specify system information, and make settings of function tables. After that, start the peripheral modules assigned to the driver facilities. To make initial settings of the peripheral modules, use the functions generated by the smart configurator (hereafter called the SC) or created by the user.

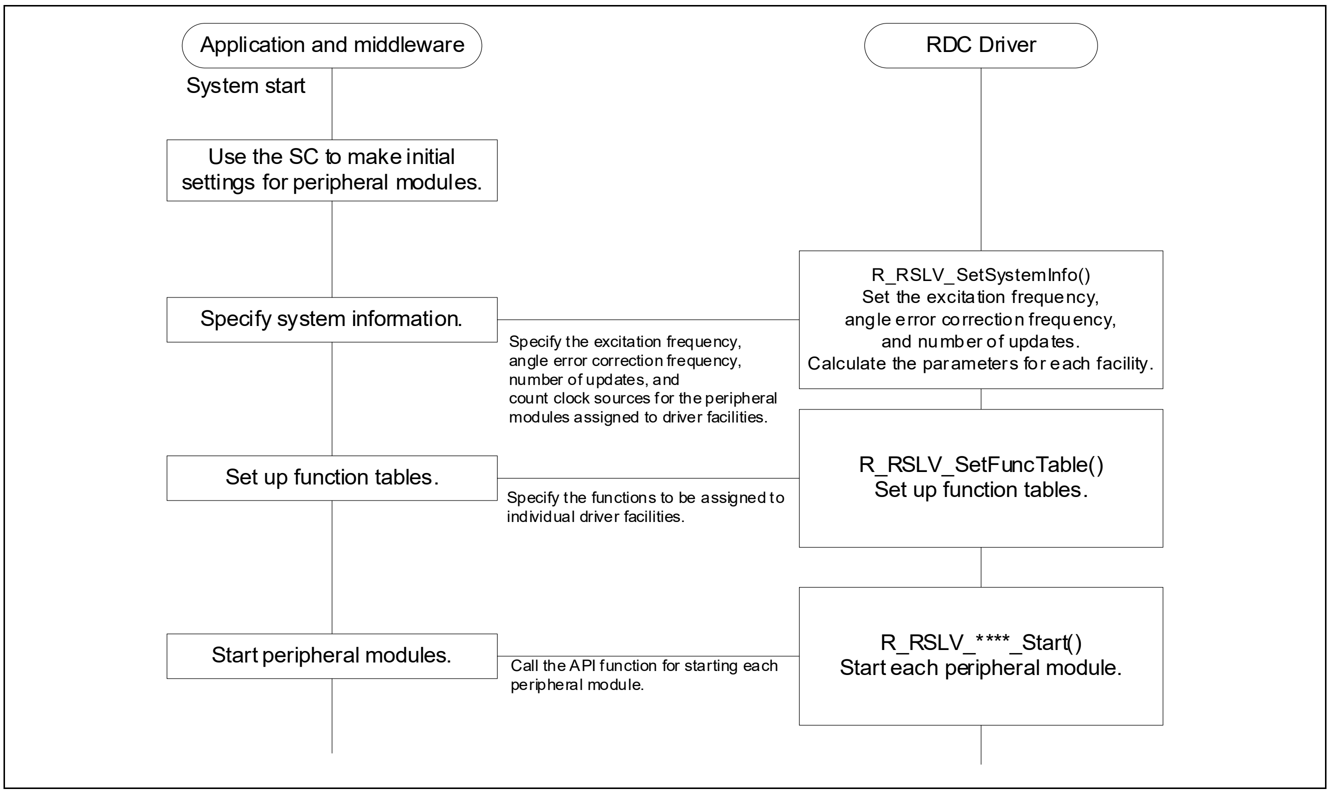

Application and middleware
RDC Driver
System start
Use the SC to make initial settings for peripheral modules.
Specify system information.
Specify the excitation frequency, angle error correction frequency, number of updates, and count clock sources for the peripheral modules assigned to driver facilities.
R_RSLV_SetSystemInfo()
Set the excitation frequency, angle error correction frequency, and number of updates.
Calculate the parameters for each facility.
Set up function tables.
Specify the functions to be assigned to individual driver facilities.
R_RSLV_SetFuncTable()
Set up function tables.
Start peripheral modules.
Call the API function for starting each peripheral module.
R_RSLV_****_Start()
Start each peripheral module.

Figure 3.2 Initialization Sequence

3.1.1 Initial Settings for the On-Chip Peripheral Modules by the SC

The user should use the SC to generate functions for initializing the peripheral modules assigned to individual driver facilities. When the MCU is started, the generated functions are called to initialize the peripheral modules.

This driver package includes a sample code of functions for initializing the peripheral modules, which were generated by the SC and can be used for reference.

3.1.2 Specifying System Information

Specify the system information, such as the excitation frequency, the angle error correction frequency, the number of updates of the angle error correction signal, and the clock sources for the peripheral modules assigned to individual driver facilities, and then execute the API function for specifying the system information. For details of the system information settings, see section 6.3.2, Structures for R_RSLV_SetSystemInfo.

Calling this function sets up the initial duty cycles of the phase adjustment signals, the maximum and minimum numbers counted for the angle error correction signal, the interval for updating the duty cycle of the angle error correction signal, and the maximum and minimum numbers counted for the input angle signal, which are used in the driver.

API function: R_RSLV_SetSystemInfo
(ST_SYSTEM_PARAM *rdc_sys_param, ST_USER_PERI_PARAM *user_peri_param)

3.1.3 Setting up Function Tables

The resolver driver uses function tables to access peripheral module registers. Specify register access functions generated by the SC or created by the user in function tables so that the driver can access peripheral module registers. For details of the function table settings, see section 6.2.1, API Function for Setting up a Function Table.

API function: R_RSLV_SetFuncTable
(unsigned char set_func, ST_FUNCTION_TABLE user_func_table)

3.1.4 Starting Peripheral Modules

This driver provides API functions for starting peripheral modules to activate individual driver facilities. For details, see section 6.1, List of API Functions. Specific functions are prepared for output of the excitation signal, output of the angle error correction signal, output of the phase adjustment signals, and input of the angle signal.

<API functions for starting peripheral modules>

Output of the excitation signal: R_RSLV_ESig_Start(void)
Output of the angle error correction signal: R_RSLV_CSig_Start
(unsigned short phase_diff, unsigned short amp_level)
Output of the phase adjustment signals: R_RSLV_Phase_AdjStart(void)
Input of angle signal: R_RSLV_Capture_Start(void)

3.2 RDC Settings

To control the resolver, the operation of the RDC must be set up. Use SPI communications to set up RDC registers.

3.2.1 Initial Settings of the RDC

To initialize the operation of the RDC, use the API function for setting the initial values of the registers in the RDC and then call the API function for executing the RDC initialization sequence. The user should specify the initial values of the registers according to the specifications of the resolver sensor used.

API function for setting the RDC initial values:
R_RSLV_Rdc_VariableInit((unsigned char*)s_u1_rdc_init_data)
API function for executing the RDC initialization sequence
R_RSLV_Rdc_Init_Sequence(unsigned short *init_status)

3.3 Output of the RDC Operating Clock

The MCU outputs an operating clock signal (4-MHz rectangular wave) for the RDC.

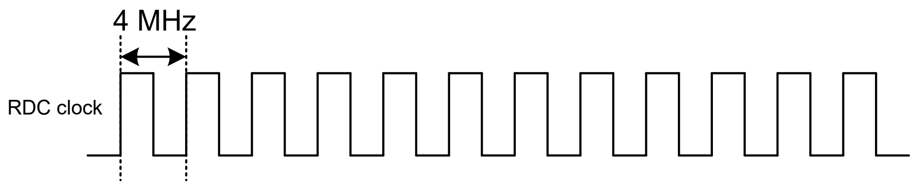


Figure 3.3 RDC Clock

3.3.1 Starting the Output of the RDC Operating Clock

The RDC operating clock is started by the API function for executing the RDC initialization sequence. This driver does not stop the RDC operation clock.

API function: R_RSLV_Rdc_Init_Sequence(unsigned short *init_status)

3.4 Communications between the RDC and MCU

SPI communications are used between the MCU and the RDC. Figure 3.4 shows a system overview of the RDC communications block.

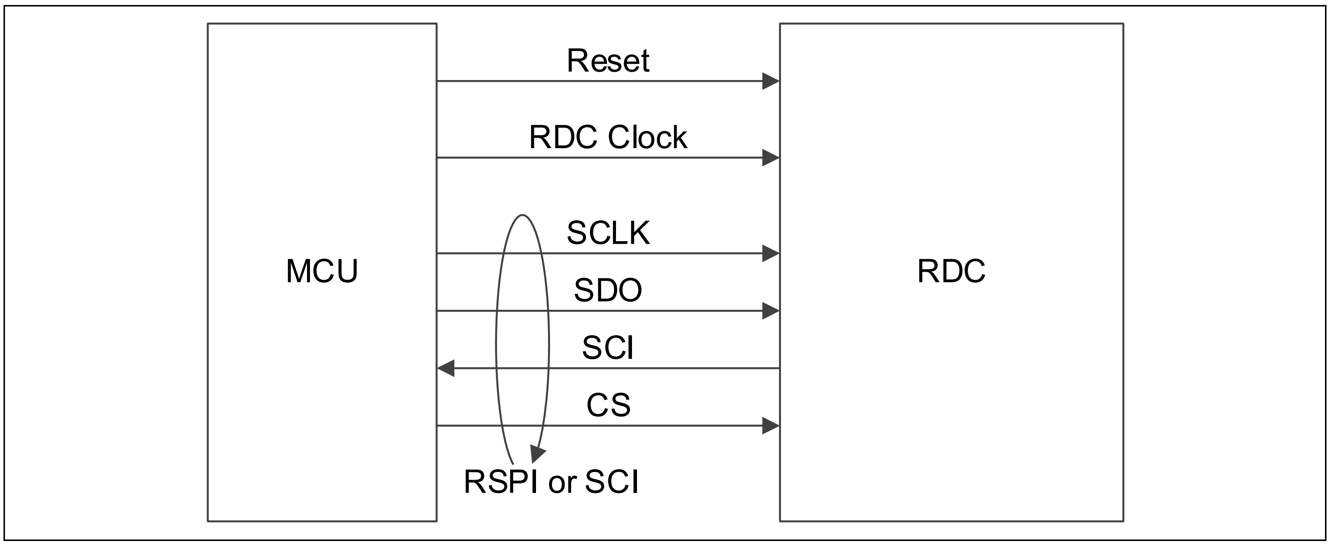


Figure 3.4 System Overview of the RDC Communications Block

3.4.1 Writing Data to an RDC Register

To write data to an RDC register, call the API function for passing a register value to the resolver driver and then the API function for starting writing.

API function for writing to the RDC register buffer (passing a register value to the resolver driver):
R_RSLV_Rdc_SetRegisterVal(unsigned char wt_data, unsigned char address)
API function for writing to an RDC register (starting writing):
R_RSLV_Rdc_RegWrite(unsigned char *write_status)

3.4.2 Reading Data from an RDC Register

To read data from an RDC register, call the API function for starting reading from an RDC register and then the API function for receiving a register value from the resolver driver.

API function for reading from an RDC register (starting reading):
R_RSLV_Rdc_RegRead(unsigned char address)
API function for reading from the RDC register buffer (receiving a register value from the resolver driver):
R_RSLV_Rdc_GetRegisterVal(unsigned char *rd_data, unsigned char address)

3.4.3 Communications with the RDC

To communicate with the RDC, call the API function for handling communications with the RDC. This function should be called repeatedly (for example, in the main loop) to write to or read from RDC registers.

API function: R_RSLV_Rdc_Communication(void)

3.5 Output of the Excitation Signal

To detect the position and speed of rotation, an excitation signal must be output to the resolver. A rectangular wave is output as the excitation signal and is converted to a sine wave by the external circuit between the MCU and RDC.

Either a single excitation signal or a signal synthesized from two rectangular waves (an excitation signal and another signal that differs from the excitation signal in phase by 60 degrees) is input to the RDC.
An excitation frequency of 5 kHz, 10 kHz, or 20 kHz is selectable. The following figure shows the waveform of the excitation signal synthesized from two rectangular waves.

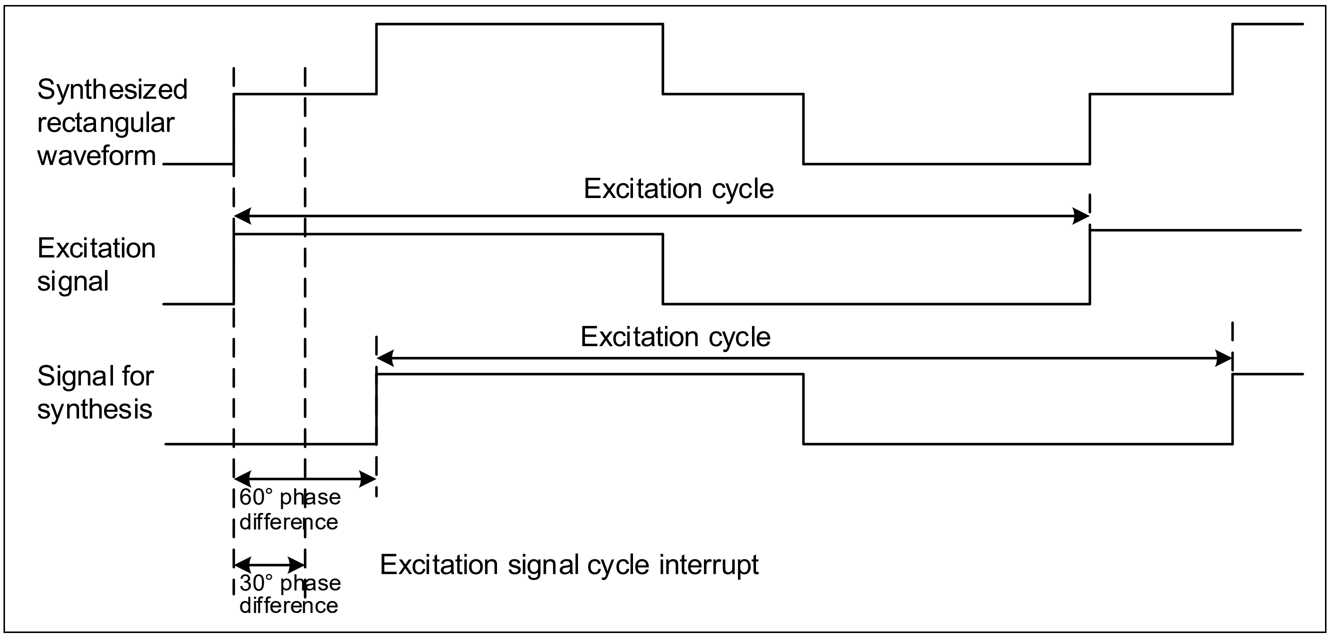


Figure 3.5 Synthesized Rectangular Wave Signal

3.5.1 Excitation Signal Cycle Interrupt

Excitation signal cycle interrupts are generated at intervals of the excitation signal output. When a single PWM signal is output as the excitation signal, interrupts are generated on the rising edges of the rectangular wave. When two PWM signals are output, interrupts are delayed by 30 degrees from the excitation signal. This interrupt should be set up in the initial settings of peripheral modules generated by the SC.

This interrupt is used to synchronize the start of the timers for outputting the excitation signal, outputting the angle error correction signal, and generating the interrupt for updating the duty cycle of the angle error correction signal.

To output two PWM signals from a single timer channel, the timer should be set up so that the output of the excitation signal toggles at every compare match of the timer. In this case, interrupts are generated twice in a single excitation signal cycle; ignore the second interrupt in an excitation signal cycle. The following gives an overview of the timing of the excitation signals and interrupts.

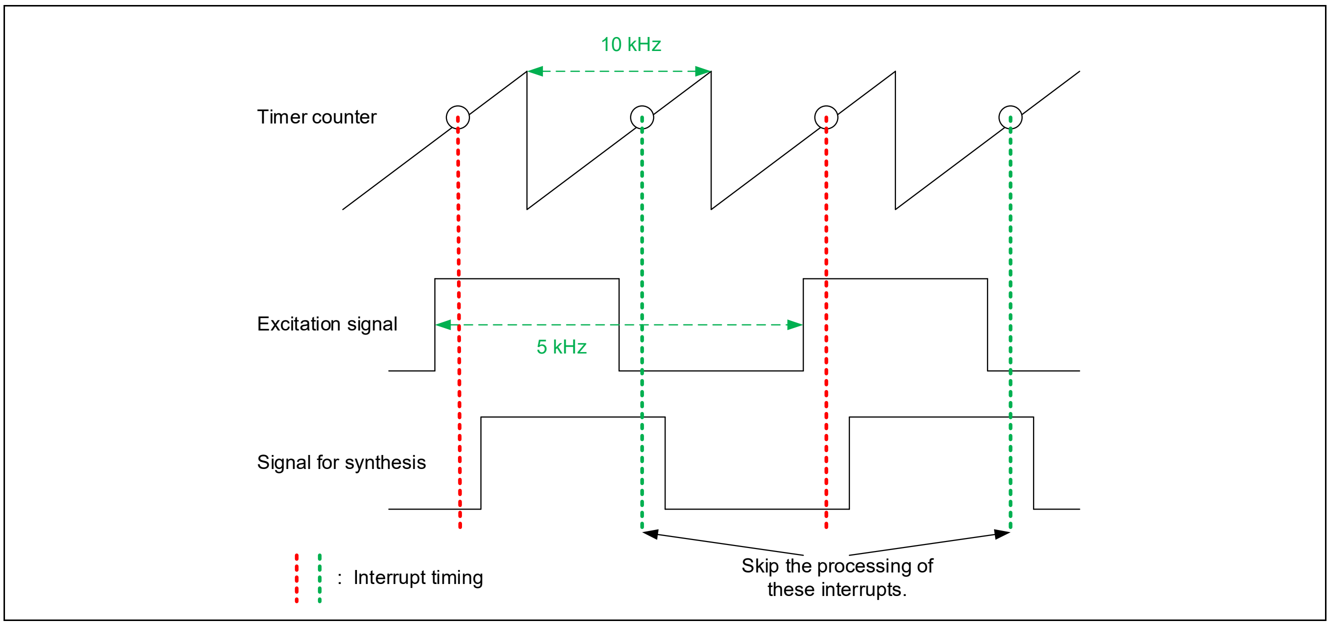


Figure 3.6 Timing of the Excitation Signals (Two Signals from a Timer Channel) and Interrupts

3.5.2 Starting the Output of the Excitation Signal

To start the output of the excitation signal, call the API function shown below. The timers for outputting the excitation signal and inputting the angle signal should be started simultaneously. For synchronous start of the timers, see section 3.6.2, Starting the Input of the Angle Signal.

API function: R_RSLV_ESig_Start(void)

3.5.3 Stopping the Output of the Excitation Signal

To stop the output of the excitation signal, call the API function shown below. The input of the angle signal started in synchronization with the output of the excitation signal is also stopped by this API function.

API function: R_RSLV_ESig_Stop(void)

3.5.4 Adjusting the Timing for Starting Output of the Excitation Signal

The resolver driver has a function for adjusting the interrupt timing for the excitation signal. The load of processing can be distributed by delaying the timing of the excitation signal interrupt from that of another interrupt process in the motor control block. Call the API function for adjusting the timing for outputting the excitation signal and inputting the angle signal.

API function: R_RSLV_ESigCapStartTiming
(unsigned short esig_start_tcnt, unsigned short cap_start_tcnt)

The following shows how to use the R_RSLV_ESigCapStartTiming function.

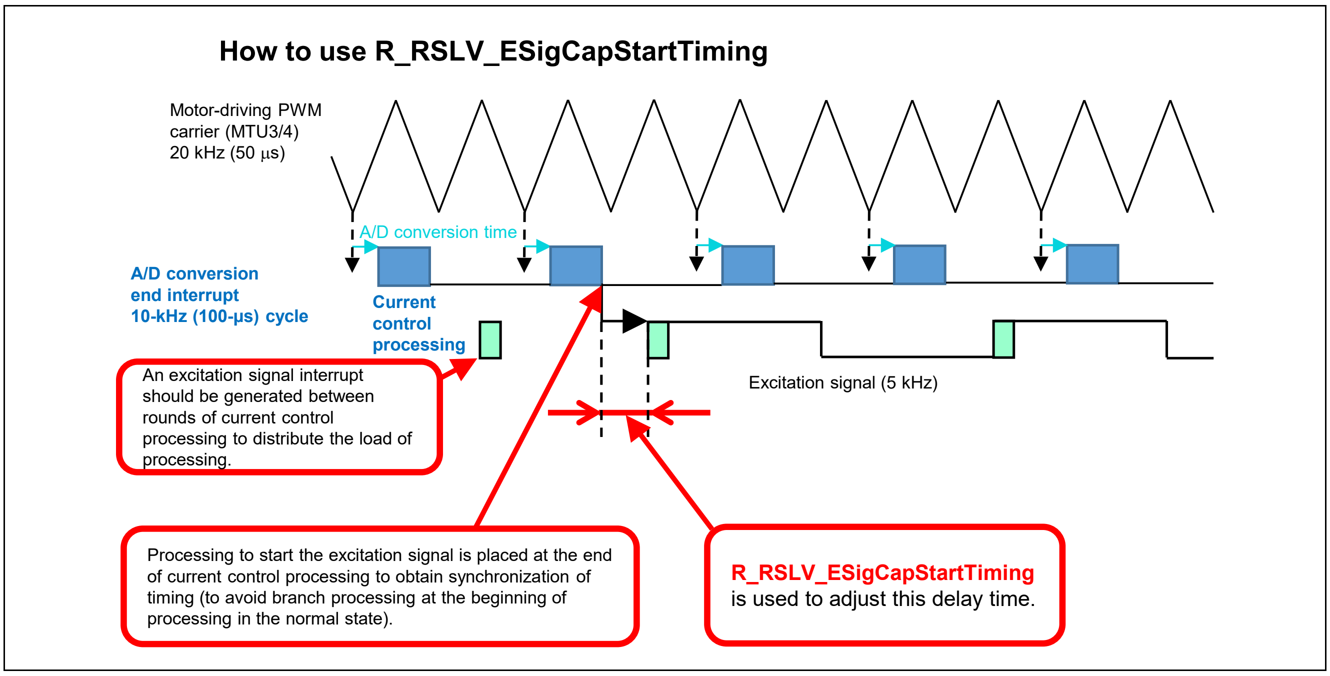

How to use R_RSLV_ESigCapStartTiming
Motor-driving PWM carrier (MTU3/4) 20 kHz (50 μs)
A/D conversion time
A/D conversion end interrupt 10-kHz (100-μs) cycle
Current control processing
Excitation signal (5 kHz)
An excitation signal interrupt should be generated between rounds of current control processing to distribute the load of processing.
Processing to start the excitation signal is placed at the end of current control processing to obtain synchronization of timing (to avoid branch processing at the beginning of processing in the normal state).
R_RSLV_ESigCapStartTiming is used to adjust this delay time.

Figure 3.7 Example of Using R_RSLV_ESigCapStartTiming (ESig)

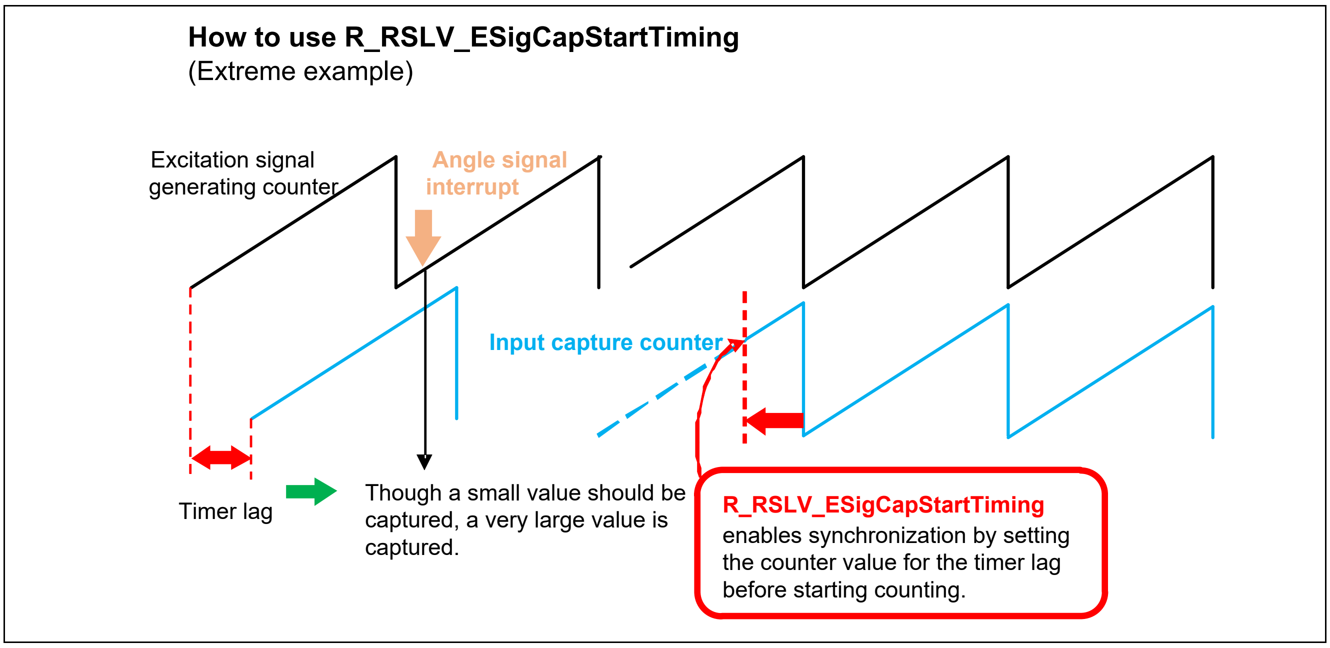

How to use R_RSLV_ESigCapStartTiming
(Extreme example)
Excitation signal generating counter
Angle signal interrupt
Input capture counter
Timer lag
Though a small value should be captured, a very large value is captured.
R_RSLV_ESigCapStartTiming enables synchronization by setting the counter value for the timer lag before starting counting.

Figure 3.8 Example of Using R_RSLV_ESigCapStartTiming (Capture)

The timing for starting the output of the excitation signal can be adjusted within the range shown below.

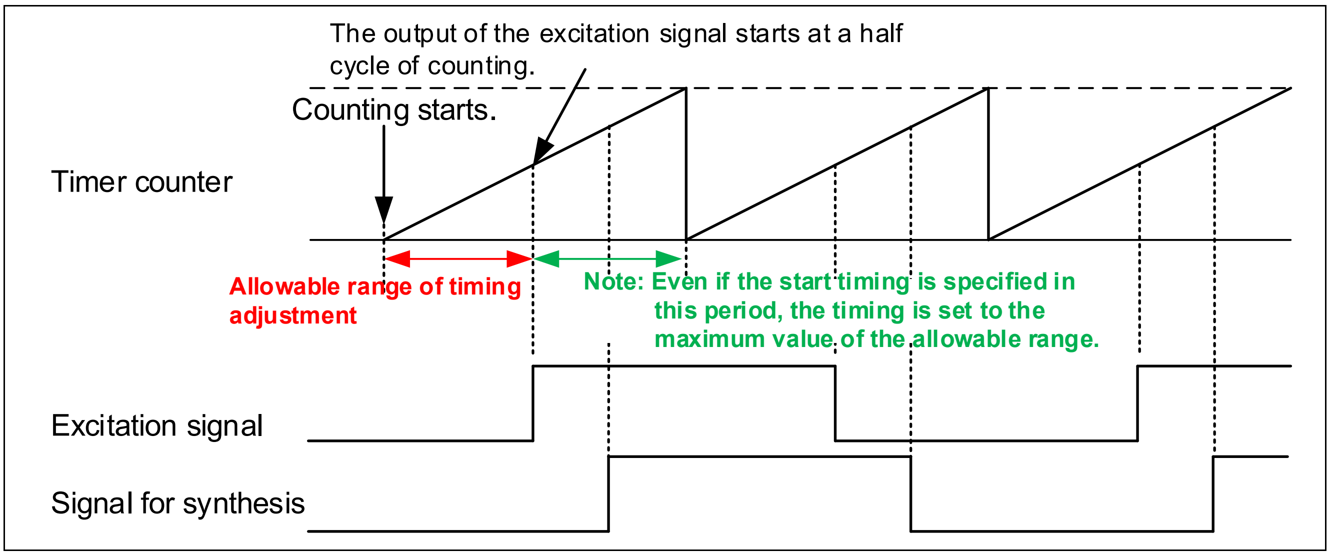

The output of the excitation signal starts at a half cycle of counting.
Counting starts.
Timer counter
Allowable range of timing adjustment
Note: Even if the start timing is specified in this period, the timing is set to the maximum value of the allowable range.
Excitation signal
Signal for synthesis

Figure 3.9 Allowable Range for Adjusting the Timing for Starting the Output of the Excitation Signal

3.6 Input of the Angle Signal

The angle signal output from the RDC is detected by using an external interrupt (input capture function). A timer having the input capture function such as MTU3, GPT, and TPU can be used to detect the signal.

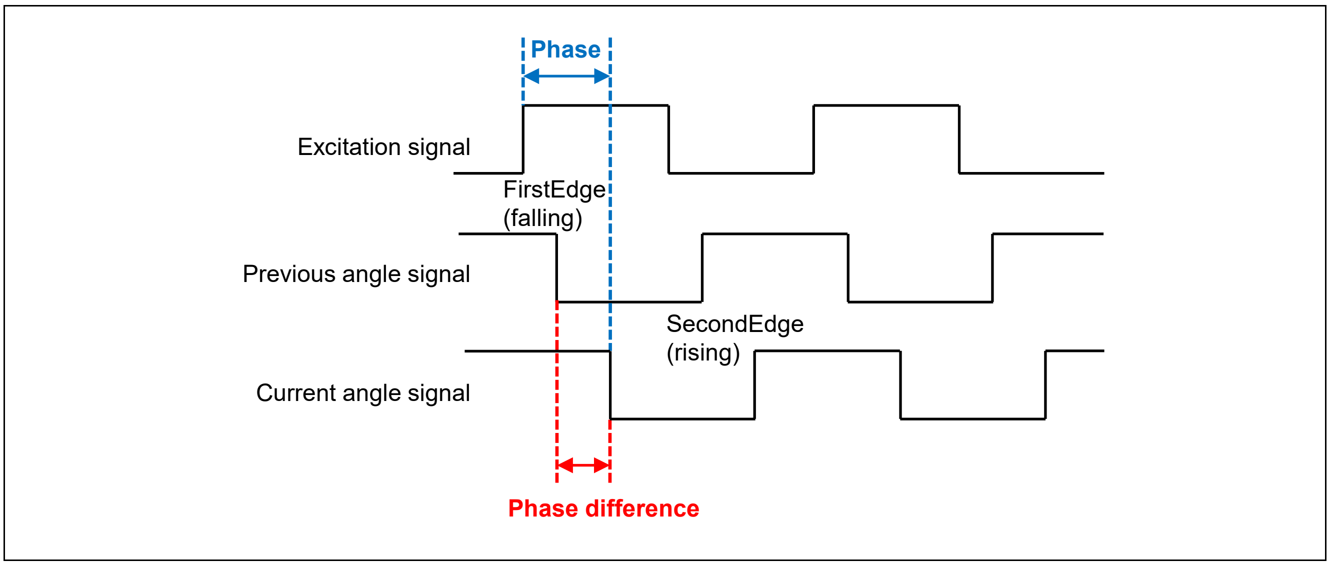


Figure 3.10 Angle Signal

The resolution of the angle signal depends on the excitation signal frequency, timer count clock, and the number of pole pairs of the resolver sensor.

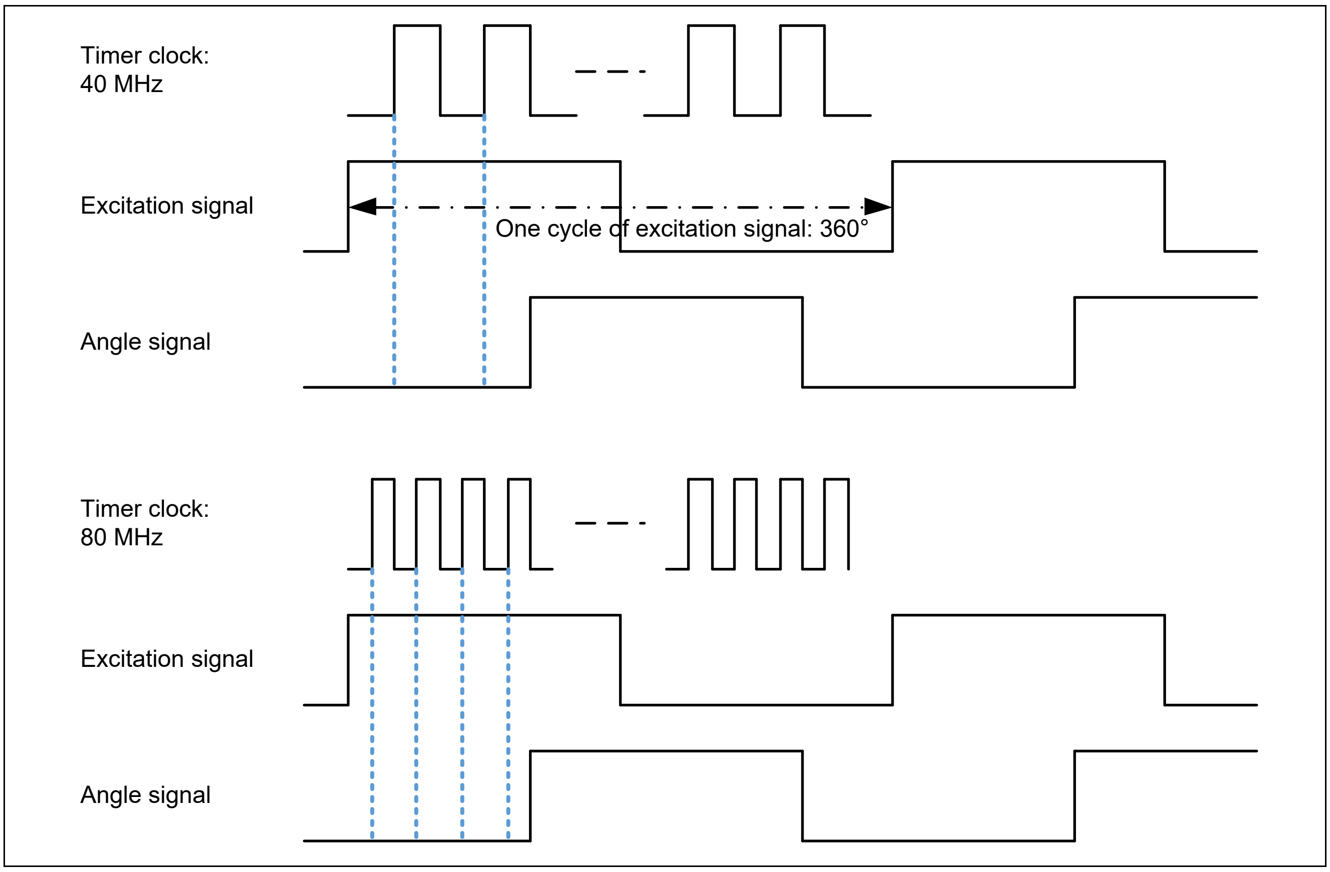


Figure 3.11 Concept of Resolution

The resolution (in terms of mechanical angle) of the angle signal can be obtained by multiplying the maximum timer counter value for a single excitation signal cycle by the number of pole pairs of the resolver sensor. The maximum number counted in a single excitation signal cycle depends on the frequencies of the output excitation signal and the clock that drives the timer counter. Assuming that the timer clock is at 40 MHz and excitation signal is at 10 kHz as in the first example in the figure above and the resolver sensor has four pole pairs, the maximum number counted in a single excitation signal cycle becomes 4000 (40 MHz/10 kHz). Therefore, the resolution of the angle signal corresponds to 16000 values (4000 × 4). When the timer clock is at 80 MHz, the resolution corresponds to 32000 values.

3.6.1 Angle Signal Input Interrupt

An input capture interrupt is generated on the specified edge of the input angle signal. The angle is obtained from the timer counter value at that time. The first-edge (falling), the second-edge (rising), or both rising and falling edges can be selected as the interrupt timing.

3.6.2 Starting the Input of the Angle Signal

To input the angle signal, counting in the timer should be started in synchronization with the output of the excitation signal. Synchronous starting can be controlled in the following three ways: starting the timers simultaneously in the API function for starting the output of the excitation signal, calling the API function for controlling synchronous starting of the MTU3 timer channels (only when using the MTU), and calling the API function for starting the angle detection timer when an excitation signal interrupt occurs.

API function for starting the output of the excitation signal:
See section 3.5.2, Starting the Output of the Excitation Signal.
API function for starting the angle detection timer: R_RSLV_Capture_Start(void)
API function for controlling synchronous starting of the MTU3 timer channels:
R_RSLV_MTU_SyncStart(unsigned char start_ch)

3.6.3 Stopping the Input of the Angle Signal

To stop the input of the angle signal, the excitation signal should be stopped. Call the API function for stopping the output of the excitation signal.

API function: See section 3.5.3, Stopping the Output of the Excitation Signal.

3.6.4 Adjusting the Timing for Starting Input of the Angle Signal

A correct angle can be obtained only when the timer counters for the input of the angle signal and the output of the excitation signal are started simultaneously. The driver has a facility for adjusting the timing for starting counting in the timer for angle signal input. Call the API function for adjusting the timing for outputting the excitation signal and inputting the angle signal. For details, see section 3.5.4, Adjusting the Timing for Starting Output of the Excitation Signal.

3.7 Detection of Disconnection from the Resolver Sensor

Figure 3.12 shows a system overview of detection of disconnection from the resolver sensor.

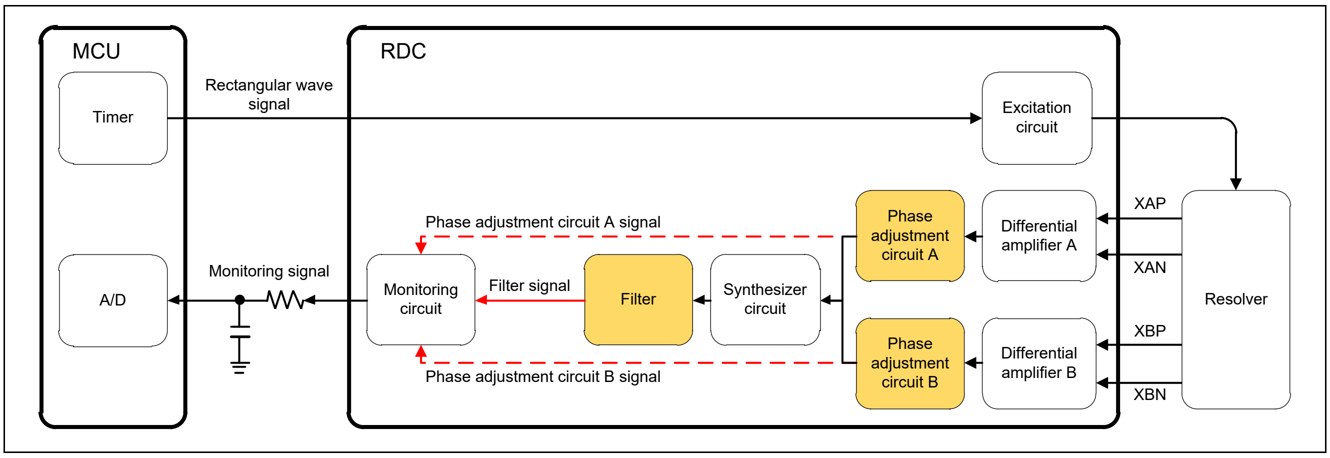


Figure 3.12 System Overview of Detection of Disconnection from the Resolver Sensor

The normal voltages of the resolver signals are compared with abnormal voltages to detect disconnection based on the difference in voltage.

To this end, the normal voltages of the resolver signals must be obtained in advance. The output signal from the monitoring circuit is used to check the voltages. Voltages of the following five signals are checked.

- Filter signal (Monitored circuit: Filter output circuit 1 output)
- XAP signal (Monitored circuit: Phase adjustment circuit A output)
- XAN signal (Monitored circuit: Phase adjustment circuit A output)
- XBP signal (Monitored circuit: Phase adjustment circuit B output)
- XBN signal (Monitored circuit: Phase adjustment circuit B output)

3.7.1 Functions Used for Detecting Disconnection

The following functions are used to detect disconnection.

3.7.1.1 Communications with the RDC

RDC register settings required for detection of disconnection are made through SPI communications.

3.7.1.2 Measuring the RDC Monitoring Signal

The RDC monitoring signal is measured by continuous scan of the 12-bit A/D converter.

3.8 Alarm Cancellation

When the RDC detects an excessive temperature, an alarm is output. To cancel this alarm, call the API functions show below. After starting alarm cancellation, call the API function for controlling the alarm cancellation sequence repeatedly.

API function for starting alarm cancellation: R_RSLV_Rdc_AlarmCancelStart(void)
API function for controlling the alarm cancellation sequence: R_RSLV_Rdc_AlarmCancel(void)

3.9 Output of the Phase Adjustment Signals for the Resolver Signals

The RDC converts the two-phase signals output from the resolver sensor into an angle signal, and then outputs the converted angle signal to the MCU. Here, unless the phase difference between the two-phase signals A and B is 90 degrees, a correct angle signal cannot be output to the MCU. For this reason, adjustment signals for resolver phase signals A and B are output from the MCU to the RDC to adjust the phase difference to 90 degrees. Phase adjustment signals are 400-kHz PWM signals.

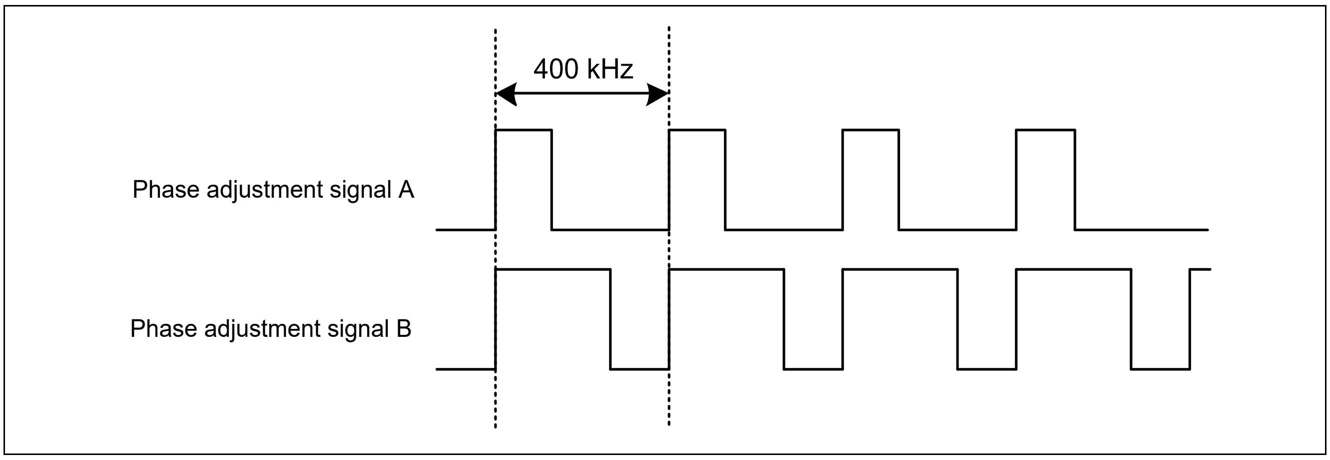


Figure 3.13 Example of Phase Adjustment Signals

3.9.1 Starting the Output of the Phase Adjustment Signals

To start the output of the phase adjustment signals, call the API function shown below.

API function: R_RSLV_Phase_AdjStart(void)

3.9.2 Stopping the Output of the Phase Adjustment Signals

To stop the output of the phase adjustment signals, call the API function shown below.

API function: R_RSLV_Phase_AdjStop(void)

3.9.3 Setting the Duty Cycles of the Phase Adjustment Signals in the Buffers

To setting the duty cycle of a phase adjustment signal in a buffer, call the API function shown below.

API function: R_RSLV_Phase_AdjUpdateBuff(unsigned short duty, unsigned char ch)

3.9.4 Setting the Duty Cycles of the Phase Adjustment Signals in the Registers

To reflect the duty cycle specified as described in section 3.9.3 in the timer for phase adjustment, call the API function shown below.

API function: R_RSLV_Phase_AdjUpdate(void)

3.9.5 Reading the Duty Cycles of the Phase Adjustment Signals from the Buffers

To read the duty cycle of a phase adjustment signal, call the API function shown below.

API function: R_RSLV_Phase_AdjReadBuff(unsigned short *duty, unsigned char ch)

3.10 Output of the Angle Error Correction Signal

When the motor is actuated, analog errors of the resolver sensor generate first-order distortion in the signal synthesized from the two-phase signals. This makes the amplitude of the synthesized signal fluctuate. This fluctuation is superposed as an error on the angle signal to be output from the RDC to the MCU.

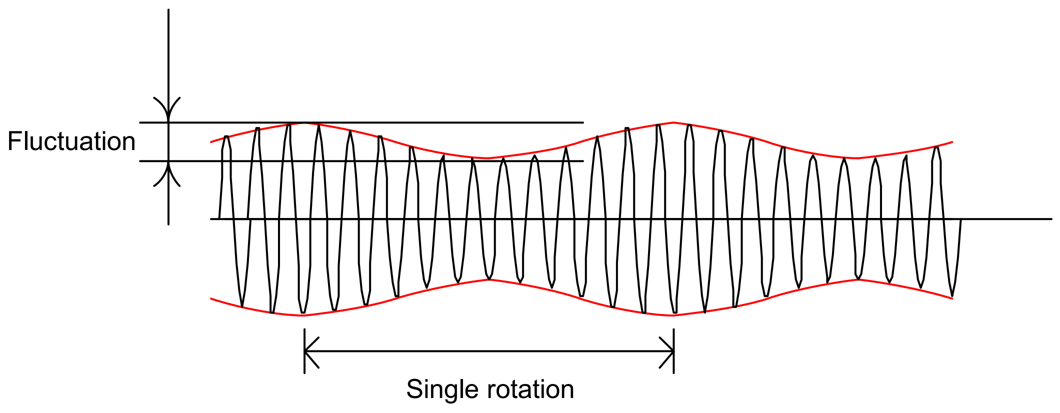


Figure 3.14 Fluctuation of Amplitude (RDC Internal Signal)

A correction signal is output from the MCU to the RDC to reduce this fluctuation. The correction signal is identical in amplitude but its phase is the inverse of that of the first-order distortion.

The angle error correction signal is a PWM signal with a carrier frequency of 200 kHz or 400 kHz (selectable). This signal is input to the RDC through a low-pass filter as an analog signal (sine wave). The angle error correction signal must be synchronized with the excitation signal. The duty cycle for generation of the sine wave is updated two or four times (selectable) per cycle of the excitation signal. The following shows a schematic diagram of angle error correction signal output.

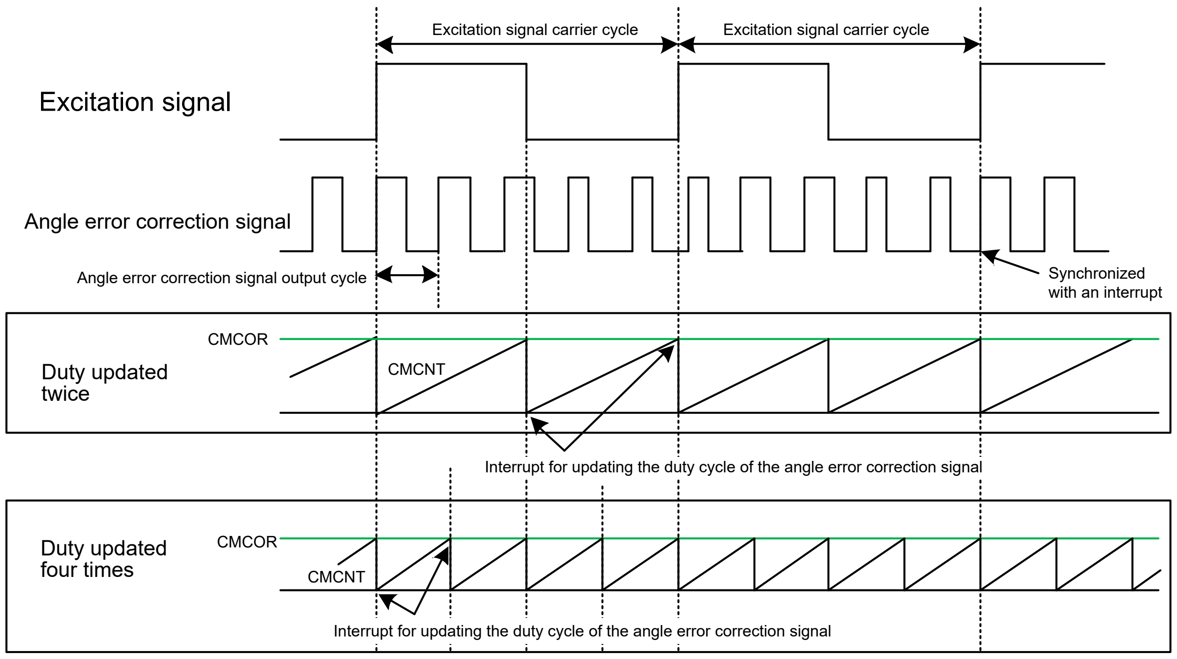

Excitation signal carrier cycle
Excitation signal carrier cycle
Excitation signal
Angle error correction signal
Angle error correction signal output cycle
Synchronized with an interrupt
CMCOR
Duty updated twice
CMCNT
Interrupt for updating the duty cycle of the angle error correction signal
Duty updated four times
CMCOR
CMCNT
Interrupt for updating the duty cycle of the angle error correction signal

Figure 3.15 Output of the Angle Error Correction Signal

The duty cycle of the angle error correction signal (PWM signal) is changed using a duty cycle updating interrupt. Figure 3.15 shows an example of using the CMT to generate duty cycle updating interrupts. The CMT counter value is set to 1/2 or 1/4 of the excitation signal cycle to select updating of the duty cycle twice or four times per cycle.

3.10.1 Starting the Output of the Angle Error Correction Signal

To output the angle error correction signal, call the API function for starting the output of the angle error correction signal. A value to be set in the timer for outputting the angle error correction signal is calculated from the phase shift amount and amplitude level specified by this API function and the number of updates of the angle error correction duty cycle. In addition, a value to be set in the timer for updating the duty cycle of the angle error correction signal is calculated. The timers for outputting the angle error correction signal and updating the duty cycle of the correction signal should be started in synchronization with the excitation signal. For synchronous starting, call the API function for synchronously starting the angle error correction signal.

API function for starting the output of the angle error correction signal:
R_RSLV_CSig_Start(unsigned short phase_diff, unsigned short amp_level)
API function for synchronously starting the angle error correction signal:
R_RSLV_INT_CSig_SyncStart (void)

3.10.2 Stopping the Output of the Angle Error Correction Signal

To stop the output of the angle error correction signal (for example, when the settings of the correction signal need to be changed), call the API function shown below. The timer for updating the duty cycle of the correction signal is also stopped at the same time.

API function: R_RSLV_CSig_Stop(void)

3.10.3 Settings for Updating the Duty Cycle of the Angle Error Correction Signal

The output frequency and the number of duty cycle updates of the angle error correction signal are specified by using the API function for specifying system information. According to the settings, this driver calculates the adjustment ranges of the phase shift amount and amplitude level of the angle error correction signal.

API function: See section 3.1.2, Specifying System Information.

3.10.4 Interrupt for Updating the Duty Cycle of the Angle Error Correction Signal

This interrupt is used to update the duty cycle of the angle error correction signal. An interrupt is generated in synchronization with the excitation signal and the duty cycle is updated by the API function for updating the duty cycle of the correction signal called within the interrupt processing. This interrupt is generated twice or four times per cycle of the excitation signal. The number of interrupt occurrences depends on the number of duty cycle updates specified in the system information settings.

API function: R_RSLV_INT_CSig_UpdatePwmDuty(void)

3.11 Automatic Calibration of Errors

This driver has functions to automatically adjust for errors in the following items:

- Resolver signal gain
- Resolver signal phase
- Angle error correction signal

3.11.1 Functions Used to Adjust Parameters

Automatic calibration uses the following driver functions to adjust parameters.

- RDC communications
  RDC registers are manipulated through SPI communications.

- Output of the angle error correction signal
  This signal is output to correct the first-order distortion error of the resolver sensor.

- PWM output for phase adjustment
  This PWM signal is output to adjust the phase difference between two-phase signals from the resolver sensor.

- Acquiring the phase count
  This phase count is angle information obtained from the RDC.

- Measuring the monitoring signal from the RDC
  The internally-synthesized signal of the RDC is output from the monitoring pin, which is used in adjusting the resolver signal gain and the angle error correction signal. To detect the monitoring signal, a function for access to the 12-bit A/D converter must be prepared in the application.

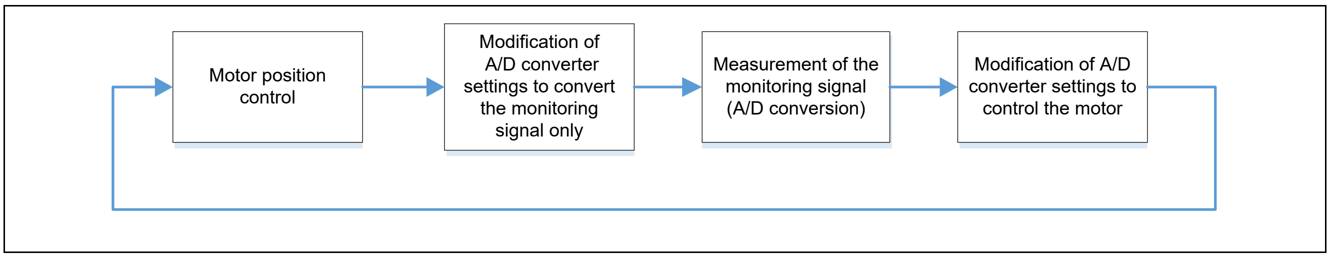

Motor position control
Modification of A/D converter settings to convert the monitoring signal only
Measurement of the monitoring signal (A/D conversion)
Modification of A/D converter settings to control the motor

Figure 3.16 Schematic Processing Flow for Measuring Monitoring Signal for Correcting Angle Errors

- Controlling the motor position
  Motor position control is used for adjusting the angle error correction signal. Control in units of one degree of the resolver angle is required.

- Controlling the motor speed
  Motor speed control is used for adjusting the angle error correction signal.

- Referencing the speed data
  The speed data (unit: rad/s) is referenced to control the speed for adjusting the angle error correction signal.

3.11.2 Adjustment of Gain and Phase of Resolver Signals

3.11.2.1 Resolver Signal Gain Adjustment

Figure 3.17 shows a block diagram for resolver signal gain adjustment.

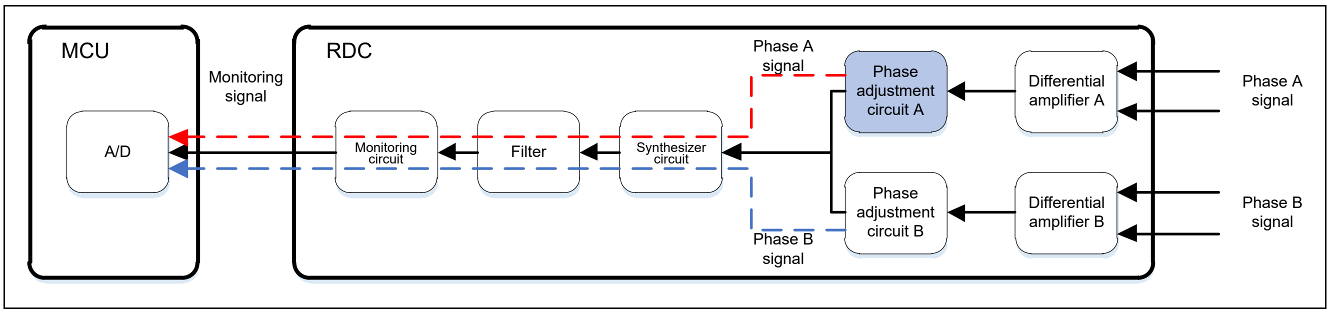


Figure 3.17 Block Diagram of Resolver Signal Gain Adjustment

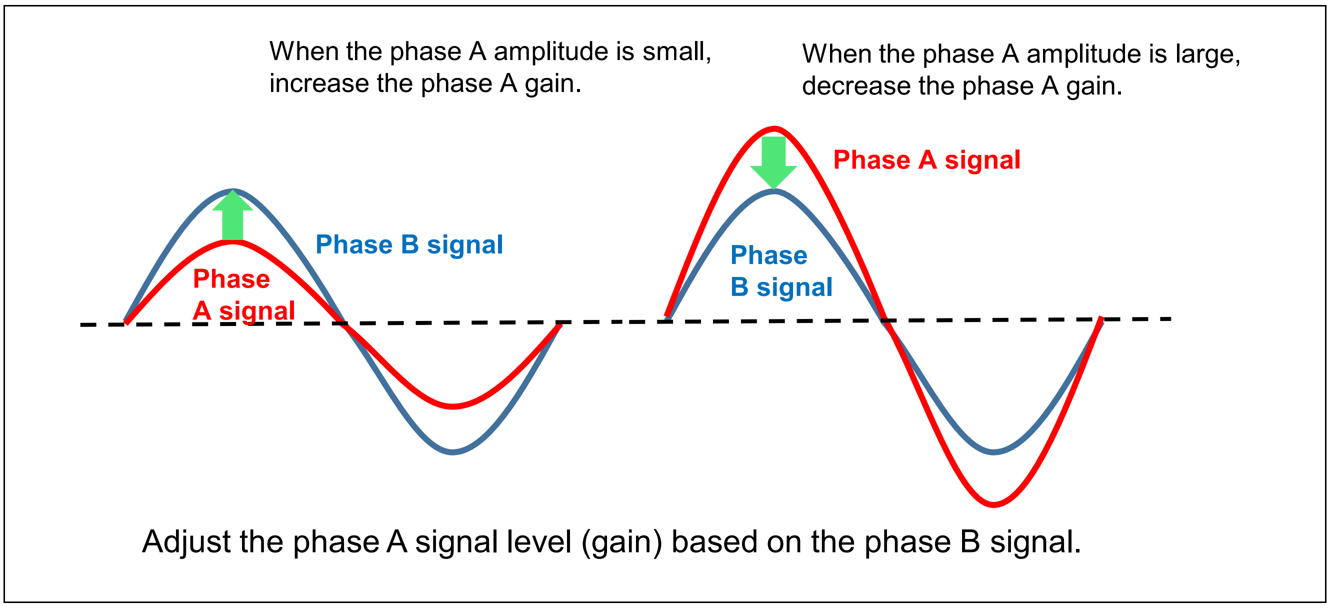

When the phase A amplitude is small, increase the phase A gain.
When the phase A amplitude is large, decrease the phase A gain.
Phase B signal
Phase A signal
Phase A signal
Phase B signal
Adjust the phase A signal level (gain) based on the phase B signal.

Figure 3.18 Resolver Signal Gain Adjustment

The phase A and phase B signals having different amplitudes produce an error in the angle information sent from the resolver. Therefore, the phase A and phase B signal amplitudes are adjusted to the same level — that is, so that the relative error between their amplitudes falls within the range ±0.28%.

3.11.2.2 Resolver Signal Phase Adjustment

Figure 3.19 shows a block diagram for resolver signal phase adjustment.

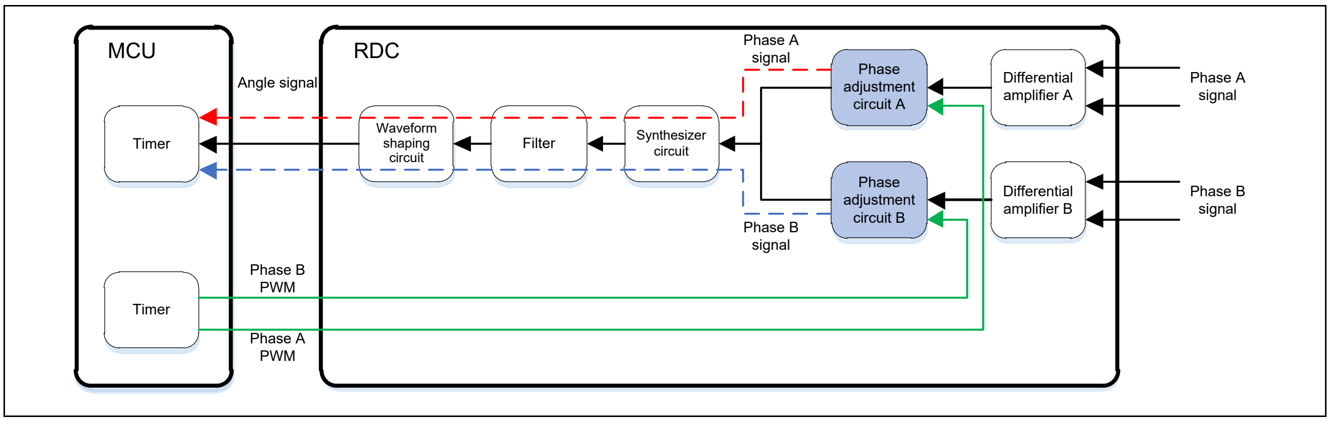


Figure 3.19 Block Diagram of Resolver Signal Phase Adjustment

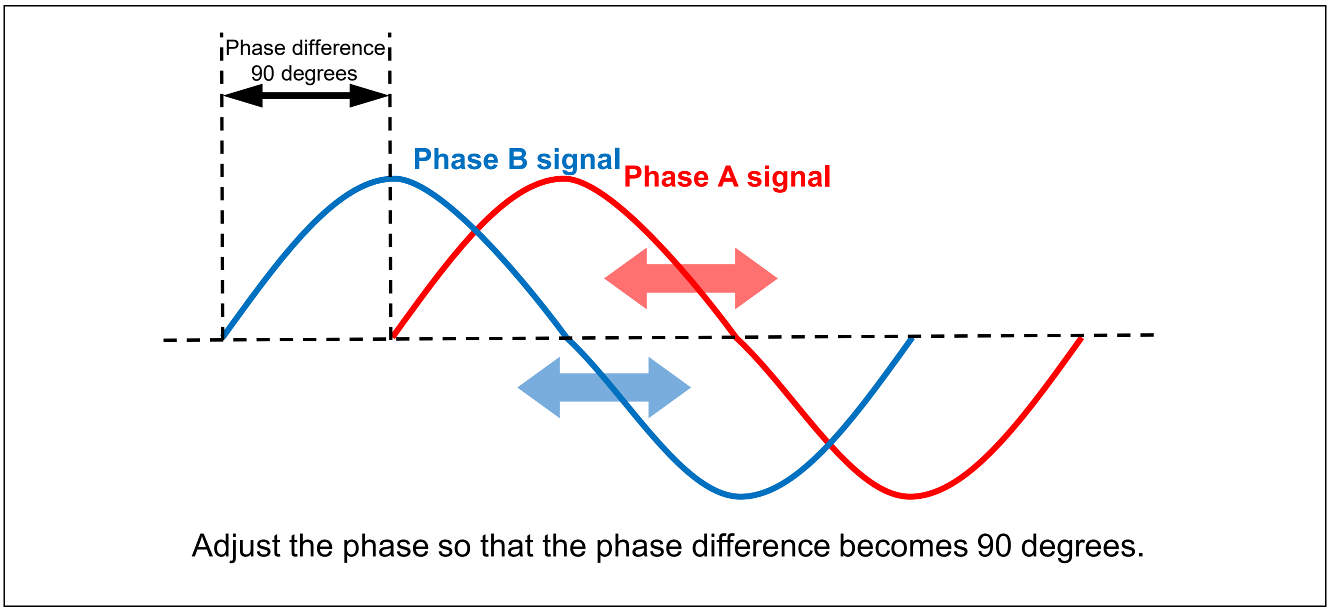

Phase difference 90 degrees
Phase B signal
Phase A signal
Adjust the phase so that the phase difference becomes 90 degrees.

Figure 3.20 Resolver Signal Phase Adjustment

The duty cycles of the phase adjustment signals for the phase A signal and phase B signal are changed so that the phase difference between the phase A signal and phase B signal falls within the range of 90 degrees ±0.3% (more precisely, ±0.27 degrees).

Duty cycle adjustment range: 5% to 90% (1% steps)

3.11.3 Adjustment of the Angle Error Correction Signal

Figure 3.21 shows a block diagram for angle error correction signal adjustment.

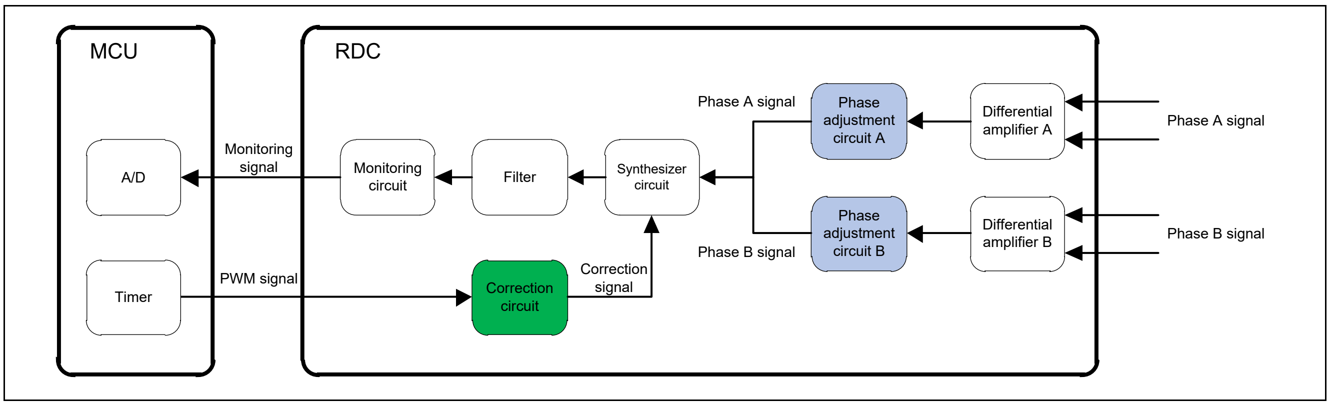


Figure 3.21 Block Diagram of Angle Error Correction Signal Adjustment

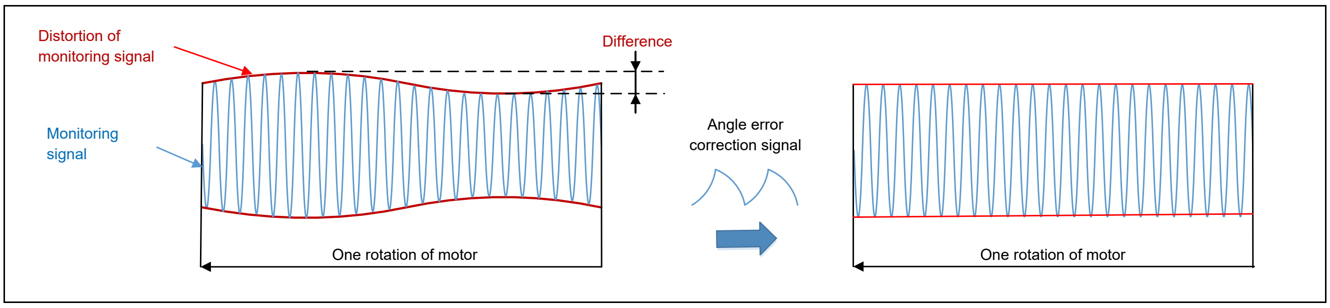


Figure 3.22 Angle Error Correction Signal Adjustment

This facility adjusts the amount of phase shift and the amplitude for the angle error correction signal input to the correction circuit. The adjusted correction signal is superposed on the angle signal in the RDC to correct angle errors due to analog errors of the resolver sensor.
The specifiable ranges of the amount of phase shift and the value of amplitude for the angle error correction signal are shown below.

Table 3-1 Specifiable Range of the Amount of Phase Shift (0 to the Value Shown Below)

MCU		RX23T	RX24T			RX66T			RX72M			
Peripheral module		CMT	MTU	GPT	CMT	MTU	GPT	CMT	MTU	GPT	TPU	CMT
Source clock setting (MHz)		5	80	80	5	160	160	7.5	120	120	60	7.5
Excitation frequency	5 kHz	999	15999	15999	999	31999	31999	1499	23999	23999	11999	1499
	10 kHz	499	7999	7999	499	15999	15999	749	11999	11999	5999	749
	20 kHz	249	3999	3999	249	7999	7999	374	5999	5999	2999	374

Note: For the CMT, specify the peripheral module clock divided by 8 as the source clock.

Table 3-2 Specifiable Range of the Amplitude (0 to the Value Shown Below)

MCU		RX23T	RX24T		RX66T		RX72M		
Peripheral module		MTU	MTU	GPT	MTU	GPT	MTU	GPT	TPU
Source clock setting (MHz)		40	80	80	160	160	120	120	60
Angle error correction signal cycle	200 kHz	199	399	399	799	799	599	599	299
	400 kHz	99	199	199	399	399	299	299	149

4. Software Configuration

4.1 Folder and File Configuration

Table 4-1 shows the configuration of the project folder and files of this driver.

Table 4-1 Folder and File Configuration

¥rx_rslv_drv			
	¥api*		
		r_rslv_api.h	Header file for the RDC driver (File for definitions of parameter structures, API functions, and common constants)
	¥lib		
		rdc_driver_library_RX.lib	Library file
¥sample¥PeripheralCode_XXX (XXX : product name of MCU)			
	¥src¥smc_gen¥Config_peri_func		
		Config_peri_func.c Config_peri_func_user.c Config_peri_func.h	Sample source files generated by the SC peri: Peripheral module name (MTU0, TMR0, etc.) func: Driver facility name: (Esig, Csig, etc.) Note: The same naming method is applied to the functions generated by the SC.
	¥src¥sample_src		
		r_sample_func_table.c	Sample source file for function tables

Note: This driver is provided as a library. The file contained in ¥api is provided to be used for access to the library.

5. Settings for Peripheral Modules

5.1 List of Macro-Defined Names of Driver Facilities

Table 5-1 lists the macro-defined names of the facilities of this driver.

Table 5-1 List of Macro-Defined Names of Driver Facilities

Defined Name	Defined Value	Description
F_ESIG1	0	Facility for setting the excitation signal (single-phase output)
F_ESIG2_1	1	Facility for setting the excitation signal (synthesized output with a phase difference of 0 degrees, two timers are used)
F_ESIG2_2	2	Facility for setting the excitation signal (synthesized output with a phase difference of 60 degrees, two timers are used)
F_ESIG12	3	Facility for setting the excitation signal (synthesized output, one timer is used)
F_CSIG	4	Facility for setting the output of the angle error correction signal
F_PHASE_A	5	Facility for setting the output of the phase adjustment signal (phase A)
F_PHASE_B	6	Facility for setting the output of the phase adjustment signal (phase B)
F_PHASE_AB	7	Facility for setting the output of the phase adjustment signal (for the output of phase A or B of one timer)
F_CAPTURE	8	Facility for setting the input of the angle signal
F_CSIG_UPD_TIMER	9	Facility for setting the timer for updating the angle error correction duty cycle
F_RDC_COM	10	Facility for setting RDC communications
F_RDC_CLK	11	Facility for setting the output of the RDC clock

5.2 List of Peripheral Modules Assigned to Driver Facilities (Recommended)

Table 5-2 to Table 5-6 list the (recommended) peripheral modules that can be assigned to serve individual driver facilities.

Table 5-2 List of Possible Combinations of Peripheral Modules and Driver Facilities (RX23T)

			Defined Name of Driver Facility											
			F_ESIG1	F_ESIG2_1	F_ESIG2_2	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_PHASE_AB	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
Peripheral Module	TMR	TMR0						√	√					√
		TMR1						√	√					√
		TMR2						√	√					√
		TMR3						√	√					√
	MTU	MTU0	√			√								
		MTU1					√				√			
		MTU2					√				√			
	CMT	CMT0										√		
		CMT1										√		
		CMT2										√		
		CMT3										√		
	RSPI	RSPI0											√	
	SCI	SCI1											√	
		SCI5											√	

Table 5-3 List of Possible Combinations of Peripheral Modules and Driver Facilities (RX24T)

			Defined Name of Driver Facility											
			F_ESIG1	F_ESIG2_1	F_ESIG2_2	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_PHASE_AB	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
Peripheral Module	TMR	TMR0						√	√					√
		TMR1						√	√					√
		TMR2						√	√					√
		TMR3						√	√					√
		TMR4						√	√					√
		TMR5						√	√					√
		TMR6						√	√					√
		TMR7						√	√					√
	MTU	MTU0	√	√	√	√	√	√	√	√	√	√		√
		MTU1	√	√	√		√	√	√		√	√		√
		MTU2	√	√	√		√	√	√		√	√		√
		MTU6	√	√	√		√	√	√	√	√	√		√
		MTU7	√	√	√		√	√	√	√	√	√		√
		MTU9	√	√	√	√	√	√	√	√	√	√		√
	GPT	GPT0	√	√	√	√	√	√	√	√	√	√		√
		GPT1	√	√	√	√	√	√	√	√	√	√		√
		GPT2	√	√	√	√	√	√	√	√	√	√		√
		GPT3	√	√	√	√	√	√	√	√	√	√		√
	CMT	CMT0										√		
		CMT1										√		
		CMT2										√		
		CMT3										√		
	RSPI	RSPI0											√	
	SCI	SCI1											√	
		SCI5											√	
		SCI6											√	

Table 5-4 List of Possible Combinations of Peripheral Modules and Driver Facilities (RX66T)

			Defined Name of Driver Facility											
			F_ESIG1	F_ESIG2_1	F_ESIG2_2	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_PHASE_AB	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
Peripheral Module	TMR	TMR0						√	√					√
		TMR1						√	√					√
		TMR2						√	√					√
		TMR3						√	√					√
		TMR4						√	√					√
		TMR5						√	√					√
		TMR6						√	√					√
		TMR7						√	√					√
	MTU	MTU0	√	√	√	√	√	√	√	√	√	√		√
		MTU1	√	√	√		√	√	√		√	√		√
		MTU2	√	√	√		√	√	√		√	√		√
		MTU6	√	√	√		√	√	√	√	√	√		√
		MTU7	√	√	√		√	√	√	√	√	√		√
		MTU9	√	√	√	√	√	√	√	√	√	√		√
	GPT	GPT0	√	√	√	√	√	√	√	√	√	√		√
		GPT1	√	√	√	√	√	√	√	√	√	√		√
		GPT2	√	√	√	√	√	√	√	√	√	√		√
		GPT3	√	√	√	√	√	√	√	√	√	√		√
		GPT4	√	√	√	√	√	√	√	√	√	√		√
		GPT5	√	√	√	√	√	√	√	√	√	√		√
		GPT6	√	√	√	√	√	√	√	√	√	√		√
		GPT7	√	√	√	√	√	√	√	√	√	√		√
		GPT8	√	√	√	√	√	√	√	√	√	√		√
		GPT9	√	√	√	√	√	√	√	√	√	√		√
	CMT	CMT0										√		
		CMT1										√		
		CMT2										√		
		CMT3										√		
	RSPI	RSPI0											√	
	SCI	SCI1											√	
		SCI5											√	
		SCI6											√	
		SCI8											√	
		SCI9											√	
		SCI11											√	
		SCI12											√	

Table 5-5 List of Possible Combinations of Peripheral Modules and Driver Facilities (RX72M) [1/2]

			Defined Name of Driver Facility											
			F_ESIG1	F_ESIG2_1	F_ESIG2_2	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_PHASE_AB	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
Peripheral Module	TMR	TMR0						√	√					√
		TMR1						√	√					√
		TMR2						√	√					√
		TMR3						√	√					√
	MTU	MTU0	√	√	√	√	√	√	√	√	√	√		√
		MTU1	√	√	√		√	√	√		√	√		√
		MTU2	√	√	√		√	√	√		√	√		√
		MTU6	√	√	√	√	√	√	√	√	√	√		√
		MTU7	√	√	√	√	√	√	√	√	√	√		√
		MTU8	√	√	√	√					√	√		√
	GPT	GPT0	√	√	√	√	√	√	√	√	√	√		√
		GPT1	√	√	√	√	√	√	√	√	√	√		√
		GPT2	√	√	√	√	√	√	√	√	√	√		√
		GPT3	√	√	√	√	√	√	√	√	√	√		√
	TPU	TPU0	√	√	√	√	√	√	√	√	√	√		√
		TPU1	√	√	√		√	√	√		√	√		√
		TPU2	√	√	√		√	√	√		√	√		√
		TPU3	√	√	√	√	√	√	√	√	√	√		√
		TPU4	√	√	√		√	√	√		√	√		√
		TPU5	√	√	√		√	√	√		√	√		√
	CMT	CMT0										√		
		CMT1										√		
		CMT2										√		
		CMT3										√		

Table 5-6 List of Possible Combinations of Peripheral Modules and Driver Facilities (RX72M) [2/2]

			Defined Name of Driver Facility											
			F_ESIG1	F_ESIG2_1	F_ESIG2_2	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_PHASE_AB	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
Peripheral Module	RSPI	RSPI0											√	
		RSPI1											√	
		RSPI2											√	
	SCI	SCI0											√	
		SCI1											√	
		SCI2											√	
		SCI3											√	
		SCI4											√	
		SCI5											√	
		SCI6											√	
		SCI7											√	
		SCI8											√	
		SCI9											√	
		SCI10											√	
		SCI11											√	
		SCI12											√	

5.3 Setting Driver Facilities by the SC

The initialization functions output from the SC are used to initialize the peripheral modules assigned to individual driver facilities. Examples of SC settings are shown in this section. Note that the MCU is RX72M and the system information to be set is as follows:

<Conditions>

Frequency of the excitation signal: 5 kHz
Frequency of the angle error correction signal: 200 kHz
Number of times the angle error correction duty cycle is to be updated: Twice

5.3.1 Output of the Excitation Signal

The MTU, GPT, and TPU (TPU is only for RX72M) are the peripheral modules recommended for assigning outputting of the excitation signal. The modes for outputting the excitation signal are single-phase output mode and synthesized output mode. The following tables show examples of setting the SC when selecting synthesized output mode for a 1-channel timer.

5.3.1.1 Examples of SC Settings When Using the MTU

Table 5-7 Selecting the Component for Outputting the Excitation Signal (MTU)

Component Selection	Selected Contents
Component	Normal mode timer
Configuration name	Config_MTU0_Esig12
Input capture/output compare pins	4 pins
Resource	MTU0

Table 5-8 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Output Pins are MTIOC0A and MTIOC0B

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRD0 compare match/input capture
Count clock	PCLK
Group: Setting the external clock pins	Setting not required.
Group: Setting the general registers	Set the following. Other settings are not required.
TGRA	Output compare register (50 μs)
TGRB	Output compare register (83.33 μs)
TGRC	Output compare register (66.67 μs)
TGRD	Output compare register (100 μs)
Group: Setting the I/O pins	Set the following. Other settings are not required. (Pin output is disabled.)
MTIOC0A pin	The initial output value of the pin is 0. The output is toggled at a compare match.
MTIOC0B pin	The initial output value of the pin is 0. The output is toggled at a compare match.
Group: Setting the noise filter	Setting not required.
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required.
TGRC	Enabled Priority: Level 11

When the excitation signal is output with ESIG12, set the start of outputting the excitation signal to the position of half of the count cycle of the timer counter. Since the count cycle is 100 μs in this example, outputting of the excitation signal starts at the position of 50 μs.

5.3.1.2 Examples of SC Settings When Using the GPT

Table 5-9 Selecting the Component for Outputting the Excitation Signal (GPT)

Component Selection	**Selected Contents**
Component	General PWM timer
Configuration name	Config_GPT0_Esig12
Operation	Saw-wave PWM mode
Resource	GPT0

Table 5-10 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Output Pins are GTIOC0A and GTIOC0B

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLKA (120.000 MHz)
Timer operation cycle	100 µs
Cycle register value	11999
Buffer operation	Buffer operation is not performed.
Count direction	Up-counting
Initial value of counter	0
To perform input capture when counting is stopped	Setting not required.
Group: Setting the compare match registers and pins	—
TAB: GTCCRA	Set the following.
GTCCRA function	Compare matches: 5999
Buffer operation	Buffer operation is not performed.
GTIOC0A pin function	PWM output pin
Noise filter	Setting not required.
Duty cycle of GTIOC0A pin output	Determined by a compare match.
Negate control of GTIOC0A pin	Disabled
Output level at starting or stopping of the counter	0 is output when started and 0 is output when stopped.
Output level at compare match	Toggle output
Output level at end of cycle	Output is retained.
Output after release of duty cycle	Setting not required.
TAB: GTCCRA input capture source	Setting not required.
TAB: GTCCRB	Set the following.
GTCCRB function	Compare matches: 9999
Buffer operation	Buffer operation is not performed.
GTIOC0B pin function	PWM output pin
Noise filter	Setting not required.
Duty cycle of GTIOC0B pin output	Determined by a compare match.
Negate control of GTIOC0B pin	Disabled
Output level at starting or stopping of the counter	0 is output when started and 0 is output when stopped.
Output level at compare match	Toggle output
Output level at end of cycle	Output is retained.
Output after release of duty cycle	Setting not required.
TAB: GTCCRB input capture source	Setting not required.
Group: Setting GTCCRC, GTCCRD, GTCCRE, and GTCCRF	Set the following. Other settings are not required.
GTCCRC function	Compare matches: 7999
Group: Setting the count source	Setting not required.
Group: Setting the stopping of output	Setting not required.
Group: Setting the A/D conversion start request	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required.
Enabling the GTCCRC compare match interrupt	Priority: Level 11
Group: Setting the function of skipping interrupts and A/D conversion start requests	Setting not required.
Group: Setting the extended function of interrupt skipping	Setting not required.
Group: Setting the extended function of buffer transfer skipping	Setting not required.

5.3.1.3 Examples of SC Settings When Using the TPU

Table 5-11 Selecting the Component for Outputting the Excitation Signal (TPU)

Component Selection	Selected Contents
Component	Normal mode timer
Configuration name	Config_TPU0_Esig12
Input capture/output compare pins	4 pins
Resource	TPU0

Table 5-12 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Output Pins are TIOCA0 and TIOCB0

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRD0 compare match/input capture
Count clock	PCLK
Group: Setting the general registers	Set the following. Other settings are not required.
TGRA0	Output compare register (50 µs)
TGRB0	Output compare register (83.333 µs)
TGRC0	Output compare register (66.667 µs)
TGRD0	Output compare register (100 µs)
Group: Setting the I/O pins	Set the following. Other settings are not required.
TIOCA0 pin	The initial output value of the pin is 0. The output is toggled at a compare match.
TIOCB0 pin	The initial output value of the pin is 0. The output is toggled at a compare match.
Group: Setting the noise filter	Setting not required.
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required (setting prohibited).
Enabling the TGRC input capture/compare match interrupt	Priority: Level 11

5.3.2 Output of the Phase Adjustment Signals for the Resolver Signals

The MTU, GPT, TMR, and TPU (TPU is only for RX72M) are the peripheral modules recommended for assigning outputting of the phase adjustment signals. The following tables show examples of setting the SC.

5.3.2.1 Examples of SC Settings When Using the MTU

Table 5-13 Selecting the Component for Outputting the Phase Adjustment Signal (MTU)

Component Selection	Selected Contents
Component	PWM mode timer
Configuration name	Config_MTU0_PhaseA
Operation	PWM mode 1
Resource	MTU0

Table 5-14 Case in Which the PWM Frequency of the Phase Adjustment Signals is 400 kHz and the Output Pin is MTIOC0A

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match/input capture
Count clock	PCLK
Group: Setting the external clock pins	Setting not required.
Group: Setting the general registers	Setting not required.
Group: Setting the output pins	Set the following. Other settings are not required. (Pin output is disabled.)
MTIOC0A pin	The initial output value of the pin is 1. 1 is output at a compare match.
Operation at TGRB compare match	0 is output from the MTIOC0A pin.
Group: Setting the PWM output	Set the following. Other settings are not required.
PWM cycle	2.5 μs
Initial value of TGRA	299
Initial value of TGRB	149
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Setting not required.

5.3.2.2 Examples of SC Settings When Using the GPT

Table 5-15 Selecting the Component for Outputting the Phase Adjustment Signal (GPT)

Component Selection	Selected Contents
Component	General PWM timer
Configuration name	Config_GPT0_PhaseA
Operation	Saw-wave PWM mode
Resource	GPT0

Table 5-16 Case in Which the PWM Frequency of the Phase Adjustment Signals is 400 kHz and the Output Pin is GTIOC0A

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLKA (120.000 MHz)
Timer operation cycle	2.5 µs
Cycle register value	299
Buffer operation	Buffer operation is not performed.
Count direction	Up-counting
Initial value of counter	0
To perform input capture when counting is stopped	Setting not required.
Group: Setting the compare match registers and pins	—
TAB: GTCCRA	Set the following.
GTCCRA function	Compare matches: 149
Buffer operation	Buffer operation is not performed.
GTIOC0A pin function	PWM output pin
Noise filter	Setting not required.
Duty cycle of GTIOC0A pin output	Determined by a compare match.
Negate control of GTIOC0A pin	Disabled
Output level at starting or stopping of the counter	1 is output when started and 0 is output when stopped.
Output level at compare match	0 is output.
Output level at end of cycle	1 is output.
Output after release of duty cycle	Setting not required.
TAB: GTCCRA input capture source	Setting not required.
TAB: GTCCRB	Setting not required.
TAB: GTCCRB input capture source	Setting not required.
Group: Setting GTCCRC, GTCCRD, GTCCRE, and GTCCRF	Setting not required.
Group: Setting the count source	Setting not required.
Group: Setting the stopping of output	Setting not required.
Group: Setting the A/D conversion start request	Setting not required.
Group: Setting the interrupts	Setting not required.
Group: Setting the function of skipping interrupts and A/D conversion start requests	Setting not required.
Group: Setting the extended function of interrupt skipping	Setting not required.
Group: Setting the extended function of buffer transfer skipping	Setting not required.

5.3.2.3 Examples of SC Settings When Using the TMR

Table 5-17 Selecting the Component for Outputting the Phase Adjustment Signal (TMR)

Component Selection	Selected Contents
Component	8-bit timer
Configuration name	Config_TMR0_PhaseA
Counting mode	8-bit counting mode
Resource	TMR0

Table 5-18 Case in Which the PWM Frequency of the Phase Adjustment Signals is 400 kHz and the Output Pin is TMO0

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLK (60000.0 kHz)
Clearing of counter	Cleared by compare match A
Value of compare match A	2.5 μs
A/D conversion start request for S12AD	Setting not required.
Value of compare match B	1.25 μs
Group: Setting the TMO0 output	Set the following.
Output level at compare match A	1 is output.
Output level at compare match B	0 is output.
Group: Setting the interrupts	Setting not required.

5.3.2.4 Examples of SC Settings When Using the TPU

Table 5-19 Selecting the Component for Outputting the Phase Adjustment Signal (TPU)

Component Selection	Selected Contents
Component	PWM mode timer
Configuration name	Config_TPU0_PhaseA
Operation	PWM mode 1
Resource	TPU0

Table 5-20 Case in Which the PWM Frequency of the Phase Adjustment Signals is 400 kHz and the Output Pin is TIOCA0

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match
Count clock	PCLK
Group: Setting the general registers	Setting not required.
Group: Setting the I/O pins	Set the following. Other settings are not required. (Pin output is disabled.)
TIOCA0 pin	The initial output value of the pin is 1. 1 is output at a compare match.
Operation at TGRB compare match	0 is output from the TIOCA0 pin.
Group: Setting the PWM output	Set the following. Other settings are not required.
PWM cycle	2.5 μs
Initial value of TGRA	149
Initial value of TGRB	78
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Setting not required.

5.3.3 Output of the Angle Error Correction Signal

The MTU, GPT, and TPU (TPU is only for RX72M) are the peripheral modules recommended for assigning outputting of the angle error correction signal. The following tables show examples of setting the SC.

5.3.3.1 Examples of SC Settings When Using the MTU

Table 5-21 Selecting the Component for Outputting the Angle Error Correction Signal (MTU)

Component Selection	Selected Contents
Component	PWM mode timer
Configuration name	Config_MTU0_Csig
Operation	PWM mode 1
Resource	MTU0

Table 5-22 Case in Which the Frequency of the Angle Error Correction Signal is 200 kHz and the Output Pin is MTIOC0A

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match
Count clock	PCLK
Group: Setting the external clock pins	Setting not required.
Group: Setting the general registers	Setting not required.
Group: Setting the output pins	Set the following. Other settings are not required. (Pin output is disabled.)
MTIOC0A pin	The initial output value of the pin is 1. 1 is output at a compare match.
Operation at TGRB compare match	0 is output from the MTIOC0A pin.
Group: Setting the PWM output	Set the following. Other settings are not required.
PWM cycle	5 µs
Initial value of TGRA	599
Initial value of TGRB	299
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Setting not required.

5.3.3.2 Examples of SC Settings When Using the GPT

Table 5-23 Selecting the Component for Outputting the Angle Error Correction Signal (GPT)

Component Selection	Selected Contents
Component	General PWM timer
Configuration name	Config_GPT0_Csig
Operation	Saw-wave PWM mode
Resource	GPT0

Table 5-24 Case in Which the Frequency of the Angle Error Correction Signal is 200 kHz and the Output Pin is GTIOC0A

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLKA (120.000 MHz)
Timer operation cycle	5 µs
Cycle register value	599
Buffer operation	Buffer operation is not performed.
Count direction	Up-counting
Initial value of counter	0
To perform input capture when counting is stopped	Setting not required.
Group: Setting the compare match registers and pins	—
TAB: GTCCRA	Set the following.
GTCCRA function	Compare matches: 299
Buffer operation	Buffer operation is not performed.
GTIOC0A pin function	PWM output pin
Noise filter	Setting not required.
Duty cycle of GTIOC0A pin output	Determined by a compare match.
Negate control of GTIOC0A pin	Disabled
Output level at starting or stopping of the counter	1 is output when started and 0 is output when stopped.
Output level at compare match	0 is output.
Output level at end of cycle	1 is output.
Output after release of duty cycle	Setting not required.
TAB: GTCCRA input capture source	Setting not required.
TAB: GTCCRB	Setting not required.
TAB: GTCCRB input capture source	Setting not required.
Group: Setting GTCCRC, GTCCRD, GTCCRE, and GTCCRF	Setting not required.
Group: Setting the count source	Setting not required.
Group: Setting the stopping of output	Setting not required.
Group: Setting the A/D conversion start request	Setting not required.
Group: Setting the interrupts	Setting not required.
Group: Setting the function of skipping interrupts and A/D conversion start requests	Setting not required.
Group: Setting the extended function of interrupt skipping	Setting not required.
Group: Setting the extended function of buffer transfer skipping	Setting not required.

5.3.3.3 Examples of SC Settings When Using the TPU

Table 5-25 Selecting the Component for Outputting the Angle Error Correction Signal (TPU)

Component Selection	**Selected Contents**
Component	PWM mode timer
Configuration name	Config_TPU0_Csig
Operation	PWM mode 1
Resource	TPU0

Table 5-26 Case in Which the Frequency of the Angle Error Correction Signal is 200 kHz and the Output Pin is TIOCA0

Item	**Setting**
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match
Count clock	PCLK
Group: Setting the general registers	Setting not required.
Group: Setting the I/O pins	Set the following. Other settings are not required. (Pin output is disabled.)
TIOCA0 pin	The initial output value of the pin is 1. 1 is output at a compare match.
Operation at TGRB compare match	0 is output from the TIOCA0 pin.
Group: Setting the PWM output	Set the following. Other settings are not required.
PWM cycle	5 μs
Initial value of TGRA	299
Initial value of TGRB	149
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Setting not required.

5.3.4 Interrupt for Updating the Duty Cycle of the Angle Error Correction Signal

The MTU, GPT, CMT, and TPU (TPU is only for RX72M) are the peripheral modules recommended for assigning the interrupt for updating the duty cycle of the angle error correction signal. The following tables show examples of setting the SC.

5.3.4.1 Examples of SC Settings When Using the MTU

Table 5-27 Selecting the Component for the Interrupt for Updating the Duty Cycle of the Angle Error Correction Signal (MTU)

Component Selection	Selected Contents
Component	Normal mode timer
Configuration name	Config_MTU0_CsigUpdTim
Input capture/output compare pins	Either 2 pins or 4 pins
Resource	MTU0

Table 5-28 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Number of Updates is Twice

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match/input capture
Count clock	PCLK
Group: Setting the external clock pins	Setting not required.
Group: Setting the general registers	Set the following. Other settings are not required.
TGRA0	Output compare register (100 μs)
Group: Setting the I/O pins	Setting not required.
Group: Setting the noise filter	Setting not required.
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required (setting prohibited).
TGRA	Enabled Priority: Level 14

5.3.4.2 Examples of SC Settings When Using the GPT

Table 5-29 Selecting the Component for the Interrupt for Updating the Duty Cycle of the Angle Error Correction Signal (GPT)

Component Selection	Selected Contents
Component	General PWM timer
Configuration name	Config_GPT0_CsigUpdTim
Operation	Saw-wave PWM mode
Resource	GPT0

Table 5-30 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Number of Updates is Twice

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLKA (120.000 MHz)
Timer operation cycle	100 μs
Cycle register value	11999
Buffer operation	Buffer operation is not performed.
Count direction	Up-counting
Initial value of counter	0
To perform input capture when counting is stopped	Setting not required.
Group: Setting the compare match registers and pins	—
TAB: GTCCRA	Setting not required.
TAB: GTCCRA input capture source	Setting not required.
TAB: GTCCRB	Setting not required.
TAB: GTCCRB input capture source	Setting not required.
Group: Setting GTCCRC, GTCCRD, GTCCRE, and GTCCRF	Setting not required.
Group: Setting the count source	Setting not required.
Group: Setting the stopping of output	Setting not required.
Group: Setting the A/D conversion start request	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required.
Enabling the GTCNT overflow (GTPR compare match) interrupt	Priority: Level 14
Group: Setting the function of skipping interrupts and A/D conversion start requests	Setting not required.
Group: Setting the extended function of interrupt skipping	Setting not required.
Group: Setting the extended function of buffer transfer skipping	Setting not required.

5.3.4.3 Examples of SC Settings When Using the TPU

Table 5-31 Selecting the Component for the Interrupt for Updating the Duty Cycle of the Angle Error Correction Signal (TPU)

Component Selection	Selected Contents
Component	PWM mode timer
Configuration name	Config_TPU0_CsigUpdTim
Operation	PWM mode 1
Resource	TPU0

Table 5-32 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Number of Updates is Twice

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match/input capture
Count clock	PCLK
Group: Setting the general registers	Setting not required.
TGRA0	Output compare register (100 μs)
Group: Setting the I/O pins	Setting not required.
Group: Setting the PWM output	Setting not required.
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required.
Enabling the TGRA input capture/compare match interrupt	Priority: Level 14

5.3.4.4 Examples of SC Settings When Using the CMT

Table 5-33 Selecting the Component for the Interrupt for Updating the Duty Cycle of the Angle Error Correction Signal (CMT)

Component Selection	Selected Contents
Component	Compare match timer
Configuration name	Config_CMT0_CsigUpdTim
Resource	CMT0

Table 5-34 Case in Which the Frequency of the Excitation Signal is 5 kHz and the Number of Updates is Twice

Item	Setting
Group: Setting the clock	Set the following.
PCLK/8, PCLK/32, PCLK/128, or PCLK/512	PCLK/8
Group: Setting the I/O pins	Set the following.
Interval time	100 μs
Register value	749
Enabling the compare match interrupt	Enabled
Priority	Level 14

5.3.5 Input of the Angle Signal

The MTU, GPT, and TPU (TPU is only for RX72M) are the peripheral modules recommended for assigning inputting of the angle signal. The following tables show examples of setting the SC.

5.3.5.1 Examples of SC Settings When Using the MTU

Table 5-35 Selecting the Component for Inputting the Angle Signal (MTU)

Component Selection	Selected Contents
Component	Normal mode timer
Configuration name	Config_MTU0_Cap
Input capture/output compare pins	Either 2 pins or 4 pins (only 2 pins for MTU1 or MTU2)
Resource	MTU0

Table 5-36 Case in Which the Frequency of the Angle Signal is 5 kHz and the Input Pin is MTIOC0B

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match/input capture
Count clock	PCLK
Group: Setting the external clock pins	Setting not required.
Group: Setting the general registers	Set the following. Other settings are not required.
TGRA0	Output compare register (200 μs)
TGRB0	Input capture register
Group: Setting the I/O pins	Set the following. Other settings are not required. (Pin output is disabled.)
MTIOC0B pin	Input capture at the falling edge of the MTIOC0B pin
Group: Setting the noise filter	Setting not required.
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required (setting prohibited).
Enabling the TGRB input capture/compare match interrupt	Priority: Level 13

5.3.5.2 Examples of SC Settings When Using the GPT

Table 5-37 Selecting the Component for Inputting the Angle Signal (GPT)

Component Selection	**Selected Contents**
Component	General PWM timer
Configuration name	Config_GPT0_Cap
Operation	Saw-wave PWM mode
Resource	GPT0

Table 5-38 Case in Which the Frequency of the Angle Signal is 5 kHz and the Input Pin is GTIOC0A

Item	**Setting**
Group: Setting the counting mode	Set the following.
Clock source	PCLKA (120.000 MHz)
Timer operation cycle	200 μs
Cycle register value	23999
Buffer operation	Buffer operation is not performed.
Count direction	Up-counting
Initial value of counter	0
To perform input capture when counting is stopped	Setting not required.
Group: Setting the compare match registers and pins	—
TAB: GTCCRA	Set the following. Other settings are not required.
GTCCRA function	Input capture
Buffer operation	Buffer operation is not performed.
GTIOC0A pin function	Input pin
TAB: GTCCRA input capture source	Set the following. Other settings are not required.
Selecting the falling edge of the GTIOC0A pin	Falling of GTIOC0A input
TAB: GTCCRB	Setting not required.
TAB: GTCCRB input capture source	Setting not required.
Group: Setting GTCCRC, GTCCRD, GTCCRE, and GTCCRF	Setting not required.
Group: Setting the count source	Setting not required.
Group: Setting the stopping of output	Setting not required.
Group: Setting the A/D conversion start request	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required.
Enabling the GTCCRA compare match/input capture interrupt	Priority: Level 13
Group: Setting the function of skipping interrupts and A/D conversion start requests	Setting not required.
Group: Setting the extended function of interrupt skipping	Setting not required.
Group: Setting the extended function of buffer transfer skipping	Setting not required.

5.3.5.3 Examples of SC Settings When Using the TPU

Table 5-39 Selecting the Component for Inputting the Angle Signal (TPU)

Component Selection	Selected Contents
Component	Normal mode timer
Configuration name	Config_TPU0_Cap
Input capture/output compare pins	Either 2 pins or 4 pins
Resource	TPU0

Table 5-40 Case in Which the Frequency of the Angle Signal is 5 kHz and the Input Pin is TIOCB0

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match/input capture
Count clock	PCLK
Group: Setting the general registers	Set the following. Other settings are not required.
TGRA0	Output compare register (200 μs)
TGRB0	Input capture register
Group: Setting the I/O pins	Setting not required.
TIOCB0 pin	Input capture at the falling edge of the MTIOC0B pin
Group: Setting the noise filter	Setting not required.
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Set the following. Other settings are not required (setting prohibited).
Enabling the TGRB input capture/compare match interrupt	Priority: Level 13

5.3.6 Output of the RDC Operating Clock

The MTU, GPT, TMR, and TPU (TPU is only for RX72M) are the peripheral modules recommended for assigning outputting of the RDC operating clock. The following tables show examples of setting the SC.

5.3.6.1 Examples of SC Settings When Using the MTU

Table 5-41 Selecting the Component for Outputting the RDC Operating Clock (MTU)

Component Selection	Selected Contents
Component	PWM mode timer
Configuration name	Config_MTU0_RdcClk
Operation	PWM mode 1
Resource	MTU0

Table 5-42 Case in Which the Frequency of the RDC Clock is 4 MHz and the Output Pin is MTIOC0A

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match (TGRA0 is used as a cycle register.)
Count clock	PCLK
Group: Setting the external clock pins	Setting not required.
Group: Setting the general registers	Setting not required.
Group: Setting the output pins	Set the following. Other settings are not required.
MTIOC0A pin	The initial output value of the pin is 1. 1 is output at a compare match.
Operation at TGRB compare match	0 is output from the MTIOC0A pin.
Group: Setting the PWM output	Set the following. Other settings are not required.
PWM cycle	250 ns
Initial value of TGRA	29
Initial value of TGRB	14
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Setting not required.

5.3.6.2 Examples of SC Settings When Using the GPT

Table 5-43 Selecting the Component for Outputting the RDC Operating Clock (GPT)

Component Selection	Selected Contents
Component	General PWM timer
Configuration name	Config_GPT0_RdcClk
Operation	Saw-wave PWM mode
Resource	GPT0

Table 5-44 Case in Which the Frequency of the RDC Clock is 4 MHz and the Output Pin is GTIOC0A

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLKA (120.000 MHz)
Timer operation cycle	250 ns
Cycle register value	29
Buffer operation	Buffer operation is not performed.
Count direction	Up-counting
Initial value of counter	0
To perform input capture when counting is stopped	Setting not required.
Group: Setting the compare match registers and pins	—
TAB: GTCCRA	Set the following.
GTCCRA function	Compare matches: 14
Buffer operation	Buffer operation is not performed.
GTIOC0A pin function	PWM output pin
Noise filter	Setting not required.
Duty cycle of GTIOC0A pin output	Determined by a compare match.
Negate control of GTIOC0A pin	Disabled
Output level at starting or stopping of the counter	1 is output when started and 0 is output when stopped.
Output level at compare match	0 is output.
Output level at end of cycle	1 is output.
Output after release of duty cycle	Setting not required.
TAB: GTCCRA input capture source	Setting not required.
TAB: GTCCRB	Set the following. Other settings are not required.
GTCCRB function	Compare matches: 28*
TAB: GTCCRB input capture source	Setting not required.
Group: Setting GTCCRC, GTCCRD, GTCCRE, and GTCCRF	Set the following.
GTCCRC function	Compare matches: 28*
GTCCRD function	Compare matches: 28*
GTCCRE function	Compare matches: 28*
GTCCRF function	Compare matches: 28*
Group: Setting the count source	Setting not required.
Group: Setting the stopping of output	Setting not required.
Group: Setting the A/D conversion start request	Setting not required.
Group: Setting the interrupts	Setting not required.
Group: Setting the function of skipping interrupts and A/D conversion start requests	Setting not required.
Group: Setting the extended function of interrupt skipping	Setting not required.
Group: Setting the extended function of buffer transfer skipping	Setting not required.

Note: * Set the maximum value because the initial value will generate an out-of-range error.

5.3.6.3 Examples of SC Settings When Using the TMR

Table 5-45 Selecting the Component for Outputting the RDC Operating Clock (TMR)

Component Selection	Selected Contents
Component	8-bit timer
Configuration name	Config_TMR0_RdcClk
Counting mode	8-bit counting mode
Resource	TMR0

Table 5-46 Case in Which the Frequency of the RDC Clock is 4 MHz and the Output Pin is TMO0

Item	Setting
Group: Setting the counting mode	Set the following.
Clock source	PCLK (60000.0 kHz)
Clearing of counter	Cleared by compare match A
Value of compare match A	250 ns
A/D conversion start request for S12AD	Do not make a setting.
Value of compare match B	125 ns
Group: Setting the TMO0 output	Set the following.
Enabling the TMO0 output	Enabled
Output level at compare match A	1 is output.
Output level at compare match B	0 is output.
Group: Setting the interrupts	Setting not required.

5.3.6.4 Examples of SC Settings When Using the TPU

Table 5-47 Selecting the Component for Outputting the RDC Operating Clock (TPU)

Component Selection	Selected Contents
Component	PWM mode timer
Configuration name	Config_TPU0_RdcClk
Operation	PWM mode 1
Resource	TPU0

Table 5-48 Case in Which the Frequency of the RDC Clock is 4 MHz and the Output Pin is TIOCA0

Item	Setting
Group: Setting the synchronous operation	Setting not required.
Group: Setting the TCNT0 counter	Set the following.
Counter clearing source	TGRA0 compare match (TGRA0 is used as a cycle register.)
Count clock	PCLK
Group: Setting the general registers	Setting not required.
Group: Setting the output pins	Set the following. Other settings are not required.
TIOCA0 pin	The initial output value of the pin is 1. 1 is output at a compare match.
Operation at TGRB compare match	0 is output from the TIOCA0 pin.
Group: Setting the PWM output	Set the following. Other settings are not required.
PWM cycle	250 ns
Initial value of TGRA	14
Initial value of TGRB	7
Group: Setting the A/D conversion start triggers	Setting not required.
Group: Setting the interrupts	Setting not required.

5.3.7 RDC Communications

The RSPI and SCI are the peripheral modules for assigning RDC communications. The following tables show examples of setting the SC.

5.3.7.1 Examples of SC Settings When Using the RSPI (SSLA0 is Selected)

Table 5-49 Selecting the Component for RDC Communications (RSPI)

Component Selection	Selected Contents
Component	SPI operation mode (4-wire method)
Configuration name	Config_RSPI0_RdcCom
Operation	Master transmission/reception
Resource	RSPI0

Table 5-50 Case in Which RDC Communications is Assigned to RSPI0 (1/2)

Item	Setting
Group: Setting the transmit/receive buffers	Set the following.
Buffer access width	16 bits
Group: Setting the parity bit	Set the following.
Byte swapping	Disabled
Parity bit	The parity bit is not added to transmit data. The parity bit is not checked in receive data.
Group: Setting the transfer rate	Set the following.
Base bit rate	1000 kbps
Group: Setting the output timing	Set the following.
Period from the beginning of SSL signal assertion to RSPCK oscillation	1 RSPCK
Period from the transmission of a final RSPCK edge to the negation of the SSL signal	1 RSPCK
Non-active period of the SSL signal after termination of a serial transfer	1 RSPCK + 2 PCLK
Group: Setting the auto-stop function	Set the following.
Enabling the auto-stop function	Disabled (Do not make a setting.)
Group: Setting the pin control	Set the following.
Idle value of MISO	Low
SSLA0 pin	Active low
SSLA1 pin	Invalid (Clear the checkbox.)
SSLA2 pin	Invalid (Clear the checkbox.)
SSLA3 pin	Invalid (Clear the checkbox.)
RSPI pin control	CMOS output
Loopback mode	Normal mode
Group: Setting the data processing	Set the following.
Transmit/receive data processing	Processed by an interrupt service routine.
Group: Setting the interrupts	Set the following.
Priority of SPTI0	Level 9
Priority of SPRI0	Level 9
Enabling the error interrupt	Enabled
Priority of SPEI0 and SPII0	Level 9

Table 5-50 Case in Which RDC Communications is Assigned to RSPI0 (2/2)

Item	Setting
Group: Setting the commands	Set the following.
TAB: Command 0	
Number of commands and number of frames	Number of commands: 1, number of transfer frames: 1
Data length	16 bits
Format	MSB first
RSPCK phase	Data variation on odd edge, data sampling on even edge
RSPCK polarity	RSPCK is high when idle
Bit rate	Base bit rate
SSL signal assertion	SSL0 (board-dependent)
SSL negation	Negates all SSL signals upon completion of transfer
RSPCK delay	1 RSPCK
SSL negation delay	1 RSPCK
Next-access delay	1 RSPCK + 2 PCLK

5.3.7.2 Examples of SC Settings When Using the SCI

Table 5-51 Selecting the Component for RDC Communications (SCI)

Component Selection	Selected Contents
Component	SPI clock synchronous operation mode (3-wire method)
Configuration name	Config_SCI0_RdcCom
Operation	Master transmission/reception
Resource	SCI0

Table 5-52 Case in Which RDC Communications is Assigned to SCI0

Item	Setting
Group: Setting the data transfer direction	Set the following.
LSB first or MSB first	MSB first
Group: Setting the transmit/receive data level	Set the following.
Standard or inverted	Standard
Group: Setting the transfer rate	Set the following.
Transfer clock	Internal clock
Base bit rate	1000 kbps
Group: Setting the clock	Set the following.
Clock delay and inversion of clock polarity	Both are disabled. (Do not check the checkboxes.)
Group: Setting the data processing	Set the following.
Transmit data processing	Processed by an interrupt service routine.
Receive data processing	Processed by an interrupt service routine.
Group: Setting the interrupts	Set the following.
Priority of TXI0	Level 9
Priority of RXI0	Level 9
Enabling the receive error interrupt	Enabled
Priority of TEI0 and ERI0	Level 9
Group: Setting the callback functions	Set the following.
Transmission end, reception end, and error detection	All callback functions are enabled. (Check the checkboxes.)

The RDC chip select processing needs to be implemented at either of the following locations when the SCI performs RDC communications. For examples of implementing the RDC chip select processing, see section 5.4.6, Function for SPI Transmission/Reception.

Chip selection is ON (active): Add to the transmission/reception start processing.
Chip selection is OFF (inactive): Add to the reception end processing.

5.4 Setting up Function Tables

Functions generated by the SC and created by the user need to be set to function tables in order to access registers of the peripheral modules assigned to individual driver facilities. The following tables list the function tables to be set to individual driver facilities.

Table 5-53 List of Function Tables Set to Driver Facilities (1/2)

Driver Facility / Function Table	ESIG1	ESIG2_1	ESIG2_2	ESIG12	CSIG	PHASE_A
Function for starting the timer	√	√	√	√	√	√
Function for stopping the timer	√	√	√	√	√	√
Function for acquiring the counter value	Δ	Δ	Δ	Δ	Δ	×
Function for setting the counter value	Δ	Δ	Δ	Δ	Δ	×
Function for acquiring the duty value	×	×	×	×	Δ	×
Function for setting the duty value	×	×	×	×	Δ	Δ
Function for setting the duty value (phase A or B of one timer)	×	×	×	×	×	×
Function for acquiring the capture value	×	×	×	×	×	×
Function for acquiring the port level	×	×	×	×	×	×
Function for SPI transmission/reception with RDC	×	×	×	×	×	×

√: Setting required (code generated by the SC), Δ: Setting required (code created by the user),
×: Setting not required.

Table 5-54 List of Function Tables Set to Driver Facilities (2/2)

Driver Facility / Function Table	PHASE_B	PHASE_AB	CAPTURE	CSIG_UPD _TIMER	RDC_CLK	RDC_COM
Function for starting the timer	√	√	√	√	√	×
Function for stopping the timer	√	√	√	√	√	×
Function for acquiring the counter value	×	×	Δ	×	×	×
Function for setting the counter value	×	×	Δ	×	×	×
Function for acquiring the duty value	×	×	×	Δ	×	×
Function for setting the duty value	Δ	×	×	Δ	×	×
Function for setting the duty value (phase A or B of one timer)	×	Δ	×	×	×	×
Function for acquiring the capture value	×	×	Δ	×	×	×
Function for acquiring the port level	×	×	Δ	×	×	×
Function for SPI transmission/reception with RDC	×	×	×	×	×	√

√: Setting required (code generated by the SC), Δ: Setting required (code created by the user),
×: Setting not required.

The details of processing to be set in function tables are shown in the following pages.

5.4.1 Functions for Starting and Stopping the Timer

The functions for starting and stopping a module, which are generated by the SC, are set to function tables.

5.4.2 Functions for Acquiring and Setting the Counter Value

The functions for acquiring and setting the counter value are not generated by the SC and so the user has to create them and set them to function tables.

The functions for acquiring and setting the counter value when using the MTU0 are shown below as examples (angle error correction signal (CSig) is the facility example).

```
/* Function to get the counter value */
void R_Config_MTU0_Csig_GetTcnt (unsigned short *tcnt)
{
   *tcnt = MTU0.TCNT;
}

/* Function to set the counter value */
void R_Config_MTU0_Csig_SetTcnt (unsigned short tcnt)
{
   MTU0.TCNT = tcnt;
}
```

5.4.3 Functions for Acquiring and Setting the Duty Value

The functions for acquiring and setting the duty value are not generated by the SC and so the user has to create them and set them to function tables.

The functions for acquiring and setting the duty value when using TGRA of the MTU0 as a general register that can change the duty cycle of the output signal are shown below as examples (angle error correction signal (CSig) is the facility example).

```
/* Function to get the duty value */
void R_Config_MTU0_Csig_GetDuty (unsigned short *duty)
{
   *duty = MTU0.TGRA;
}

/* Function to set the duty value */
void R_Config_MTU0_Csig_SetDuty (unsigned short duty)
{
   MTU0.TGRA = duty;
}

/* Function to set the duty value */
void R_Config_MTU0_Csig_SetDuty_2val (unsigned short ch, unsigned short duty)
{
   If (PHASE_CH_A == ch)
   {
      MTU0.TGRA = duty;       /* Phase A signal duty setting */
   }
   else if (PHASE_CH_B == ch)
   {
      MTU0.TGRC = duty;       /* Phase B signal duty setting */
   }
}
```

5.4.4 Function for Acquiring the Capture Value

The function for acquiring the capture value is not generated by the SC and so the user has to create it and set it to function tables.

The function for acquiring the capture value when using the MTU2 is shown below as an example.

```
/* Function to get the capture value */
void R_Config_MTU2_Cap_GetCapVal (unsigned short *current_angle_count)
{
    *current_angle_count= MTU2.TGRA;
}
```

5.4.5 Function for Acquiring the Port Level

The function for acquiring the port level is not generated by the SC and so the user has to create it and set it to function tables.

The function for acquiring the level of the P00 port is shown below as an example.

```
/* Function to get the port level */
void R_Config_MTU2_Cap_GetPortLvl (unsigned char *port_level)
{
    *port_level = PORT0.PIDR.BIT.B0;
}
```

5.4.6 Function for SPI Transmission/Reception

The function for transmission or reception by the RSPI or SCI, which is generated by the SC, is set to function tables. When the function is created for the SCI, a chip select signal needs to be output using a general port. Since the SCI uses the 8-bit communication format, the 16-bit communication format has to be supported for achieving communications with the RDC. For code examples, see section 7.10.2.3, Example of Using the SCI. The functions for transmission or reception by the RSPI and SCI are shown in the following sub-sections as respective examples.

5.4.6.1 When Using the SCI

```
/* Transmission/reception start processing (code generated by the SC)*/
MD_STATUS R_Config_SCI1_RdcCom_SPI_Master_Send_Receive
(uint8_t * const tx_buf, uint16_t tx_num, uint8_t * const rx_buf, uint16_t
rx_num)
{
    MD_STATUS status = MD_OK;

    if (1U > tx_num)
    {
        status = MD_ARGERROR;
    }
    else
    {

        R_Config_SCI0_Start();       // Start SCI (requires to be added)

        g_sci0_tx_count = tx_num;
        gp_sci0_tx_address = tx_buf;
        gp_sci0_rx_address = rx_buf;
        g_sci0_rx_count = 0U;
        g_sci0_rx_length = rx_num;

        /* Set SMOSI0 pin */
        PORT2.PMR.BYTE |= 0x01U;

        /* Set low to CS port */
        PORT9.PODR.BIT.B2 = 0U;    // Select chip: Chip ACTIVE (requires to be
added)

        /* Set TE, TIE, RE, RIE bits simultaneously */
        SCI0.SCR.BYTE |= 0xF0U;
    }

    return (status);
}
```

5.4.6.2 When Using the RSPI

```
/* Transmission/reception start processing (code generated by the SC)*/
MD_STATUS R_Config_RSPI0_RdcCom_Send_Receive
(uint16_t * const tx_buf, uint16_t tx_num, uint16_t * const rx_buf)
{
    MD_STATUS status = MD_OK;

    if (tx_num < 1U)
    {
        status = MD_ARGERROR;
    }
    else
    {
        R_Config_RSPI0_RSPI0_Start();     // Start RSPI (requires to be added)

        /* Initialize the global counters */
        gp_rspi0_tx_address = tx_buf;
        g_rspi0_tx_count = tx_num;
        gp_rspi0_rx_address = rx_buf;
        g_rspi0_rx_length = tx_num;
        g_rspi0_rx_count = 0U;

        /* Enable transmit interrupt */
        RSPI0.SPCR.BIT.SPTIE = 1U;

        /* Enable receive interrupt */
        RSPI0.SPCR.BIT.SPRIE = 1U;

        /* Enable error interrupt */
        RSPI0.SPCR.BIT.SPEIE = 1U;

        /* Enable RSPI function */
        RSPI0.SPCR.BIT.SPE = 1U;
    }

    return (status);
}
```

6. API Functions

6.1 List of API Functions

The driver provides API functions that can be called from the application or middleware. The following tables list the API functions. For details of API functions, see section 6.2, Descriptions of API Functions.

Table 6-1 API Functions (r_rslv_api.h) (1/4)

File Name	Category	Interface Function Name	Processing
r_rslv_api.h	Initialization System information	R_RSLV_SetSystemInfo Input: ST_SYSTEM_PARAM *rdc_sys_param / System information ST_USER_PERI_PARAM *user_peri_param / Setting information of user peripheral module Output: unsigned char result / Processing result	Selects system information, such as the timer counter value to be used, from the information passed through the argument.
		R_RSLV_SetFuncTable Input: unsigned char set_func, / Driver facility FUNCTION_TABLE user_func_table / Pointer to functions Output: unsigned char result / Processing result	Sets the function pointer passed through the argument to the function table.
		R_RSLV_GetRdcDrvSettingInfo Input: ST_RDC_DRV_SETTING_INFO *rdc_setting_info / Pointer to the setting information structure Output: unsigned char result / Processing result	Obtains the excitation frequency and the maximum value of the angle detection timer counter specified in the RDC driver, sets the information in the pointer variable argument, and reports it to the user.
		R_RSLV_MTU_SyncStart Input: unsigned char start_ch / MTU channel Output: unsigned char result / Processing result	Writes the value passed through the argument to the timer counter synchronous start register in the MTU to simultaneously start the timer counters of the selected channels of the MTU.
		R_RSLV_GetDriverVer Input: unsigned long *drv_ver / Pointer to driver version storage buffer Output: unsigned char result / Processing result	Acquires the RDC driver version information.
	Angle error correction signal	R_RSLV_CSig_Start Input: unsigned short phase_diff / Phase shift amount unsigned short amp_level / Amplitude level Output: unsigned char result / Processing result	Makes necessary preparations to start outputting the angle error correction signal including calculation of the angle error correction duty cycle.
		R_RSLV_CSig_Stop Input: None Output: unsigned char result / Processing result	Stops outputting the angle error correction signal.
		R_RSLV_INT_CSig_UpdatePwmDuty Input: None Output: unsigned char result / Processing result	Updates the PWM duty cycle of the angle error correction signal.
		R_RSLV_INT_CSig_SyncStart Input: None Output: unsigned char result / Processing result	Starts synchronization between the excitation signal and angle error correction signal.

Table 6-1 API Functions (r_rslv_api.h) (2/4)

File Name	Category	Interface Function Name	Processing
r_rslv_api.h	Angle signal input	R_RSLV_Capture_Start Input: None Output: unsigned char result / Processing result	Starts the angle detection timer.
		R_RSLV_INT_GetCaptureCount Input: None Output: unsigned char result / Processing result	Acquires the angle detection value (current angle count), calculates the difference from the previous value, and then sets it in the variable.
		R_RSLV_GetCaptureEdge Input: unsigned char *cap_edge / Capture port state Output: unsigned char result / Processing result	Acquires the information to determine whether the previous capture was made on a rising edge or a falling edge.
		R_RSLV_GetAngleCountFirstEdge Input: unsigned short *angle_cnt / Angle Output: unsigned char result / Processing result	Acquires the current angle count stored in the variable (on a falling edge).
		R_RSLV_GetAngleDifferenceFirstEdge Input: unsigned short *angle_diff_cnt / Angle difference Output: unsigned char result / Processing result	Acquires the difference between the current angle and the previous angle stored in the variable (on a falling edge).
		R_RSLV_GetAngleCountSecondEdge Input: unsigned short *angle_cnt / Angle Output: unsigned char result / Processing result	Acquires the current angle count stored in the variable (on a rising edge).
		R_RSLV_GetAngleDifferenceSecondEdge Input: unsigned short *angle_diff_cnt / Angle difference Output: unsigned char result / Processing result	Acquires the difference between the current angle and the previous angle stored in the variable (on a rising edge).
	Excitation signal	R_RSLV_ESig_Start Input: None Output: unsigned char result / Processing result	Starts outputting the excitation signal.
		R_RSLV_ESig_Stop Input: None Output: unsigned char result / Processing result	Stops outputting the excitation signal.
		R_RSLV_EsigCapStartTiming Input: unsigned short esig_start_tcnt / ESIG timer counter value unsigned short cap_start_tcnt / Angle detection timer counter value Output: unsigned char result / Processing result	Adjusts the timing to start outputting the excitation signal and the timing to start the angle detection timer.
		R_RSLV_INT_ESigCounter Input: None Output: unsigned char result / Processing result	Starts counting down by the wait timer in the automatic calibration processing.
	Phase adjustment signals	R_RSLV_Phase_AdjStart Input: None Output: unsigned char result / Processing result	Starts outputting the phase adjustment signals.
		R_RSLV_Phase_AdjStop Input: None Output: unsigned char result / Processing result	Stops outputting the phase adjustment signals.
		R_RSLV_Phase_AdjUpdateBuff Input: unsigned short duty / Duty value unsigned char ch / Selection of phase A or phase B Output: unsigned char result / Processing result	Sets the duty cycle of a phase adjustment signal in the buffer.
		R_RSLV_Phase_AdjUpdate Input: None Output: unsigned char result / Processing result	Sets the duty cycle of a phase adjustment signal in the register.

Table 6-1 API Functions (r_rslv_api.h) (3/4)

File Name	Category	Interface Function Name	Processing
r_rslv_api.h	Phase adjustment signals	R_RSLV_Phase_AdjReadBuff Input: unsigned short *duty / Read duty value unsigned char ch / Specification of phase A or B to be read Output: unsigned char result / Processing result	Reads the duty cycle of the phase adjustment signal from the register.
	RDC settings	R_RSLV_Rdc_VariableInit Input: unsigned char *u1_init_data / RDC initialization command table Output: unsigned char result / Processing result	Sets the initial values of RDC communications.
		R_RSLV_Rdc_Init_Sequence Input: unsigned short *init_status / Communication state Output: unsigned char result / Processing result	Makes initial settings of the RDC.
		R_RSLV_Rdc_Communication Input: None Output: unsigned char result / Processing result	Handles communications with the RDC. A communication sequence is provided and repeated calls of this function cause progress through the sequence.
		R_RSLV_Rdc_RegWrite Input: unsigned char *write_status / Write state Output: unsigned char result / Processing result	Writes a value to the RDC register buffer variable.
		R_RSLV_Rdc_RegRead Input: unsigned char address / Read address Output: unsigned char result / Processing result	Starts reading data from the RDC register. Note: This function is a trigger to start reading.
		R_RSLV_Rdc_ChkIfRun Input: None Output: unsigned char result / Processing result	Returns the RDC register access state as a return value.
		R_RSLV_Rdc_GetRegisterVal Input: unsigned char *rd_data / Data read from variable unsigned char address / Read address Output: unsigned char result / Processing result	Reads the RDC register value from the variable.
		R_RSLV_Rdc_SetRegisterVal Input: unsigned char wt_data / Data written to variable unsigned char address / Write address Output: unsigned char result / Processing result	Writes the RDC register value to the variable.
		R_RSLV_Rdc_CallComEndCb Input: None Output: unsigned char result / Processing result	Performs the callback processing for the RDC communication transmit/receive end interrupt.
		R_RSLV_Rdc_CallErrorCb Input: None Output: unsigned char result / Processing result	Performs the callback processing for the RDC communication error interrupt.
		R_RSLV_RdcCom_GetErrorInfo Input: unsigned char *err_info / RDC communication error information Output: unsigned char result / Processing result	Acquires information about whether an RDC communication error has occurred.
		R_RSLV_Rdc_AlarmCancelStart Input: None Output: unsigned char result / Processing result	Starts the RDC alarm cancellation sequence.
		R_RSLV_Rdc_AlarmCancel Input: None Output: unsigned char result / Processing result	Controls the RDC alarm cancellation sequence.

Table 6-1 API Functions (r_rslv_api.h) (4/4)

File Name	Category	Interface Function Name	Processing
r_rslv_api.h	Automatic calibration of errors	R_RSLV_ADJST_GainPhase Input: unsigned char u1_call_state / Adjustment execution request Output: st_adjst_gainphase_return_t / Processing result	Performs resolver signal gain adjustment and resolver signal phase adjustment.
		R_RSLV_ADJST_Carrier Input: st_adjst_carrier_arg_t arg_value / Adjustment execution request Output: st_adjst_carrier_return_t return_val / Adjustment processing execution state or processing result	Adjusts the angle error correction signal.
		R_RSLV_ADJST_SetPtrFunc Input: st_ptr_func_arg_t *ptr_arg / Pointer to callback function Output: None	Sets the pointer to the user-created callback function and variables in the automatic calibration facility.
		R_RSLV_ADJST_Ad_Processing Input: None Output: unsigned char gs_u1_ad_condition / A/D conversion execution state	Returns 1 during A/D conversion of the monitoring signal or returns 0 in other cases.
	Detection of disconnection	R_RSLV_DiscDetection_Seq Input: st_rdc_ddmnt_arg_t arg_value / Disconnection detection parameter Output: unsigned char return_valt / Operation state	Performs processing for the disconnection detection sequence.

6.2 Descriptions of API Functions

6.2.1 API Function for Setting up a Function Table

Item	Description	
Function name	R_RSLV_SetFuncTable	
Argument	unsigned char set_func, ST_FUNCTION_TABLE user_func_table	Driver facility to which a function table is set Function table
Return value	unsigned char	Processing result
Function	Sets up a function table to be used in the driver. • Specifies a driver facility. • Specifies a function table.	
Remark	ST_FUNCTION_TABLE is a structure. For details on setting up a function table, see section 5.4, Setting up Function Tables. For possible combinations of peripheral modules and driver facilities, see section 5.2, List of Peripheral Modules Assigned to Driver Facilities (Recommended).	

6.2.2 API Function for Specifying System Information

Item	Description	
Function name	R_RSLV_SetSystemInfo	
Argument	ST_SYSTEM_PARAM *rdc_sys_param, ST_USER_PERI_PARAM *user_peri_param	System setting information Count clock source of used peripheral module
Return value	unsigned char	Processing result
Function	Specifies the following system information. • Frequency of the excitation signal • Frequency of the output angle error correction signal • Number of times the angle error correction duty cycle is to be updated • Motor type • Output mode of the MNTOUT pin of the RDC • Count clock source (MHz) of the peripheral module assigned for outputting the excitation signal • Count clock source (MHz) of the peripheral module assigned for outputting the angle error correction signal • Count clock source (MHz) of the peripheral module assigned for inputting the angle signal • Count clock source (MHz) of the peripheral module assigned for updating the duty cycle of the angle error correction signal • Count clock source (MHz) of the peripheral module assigned for outputting the phase adjustment signal A • Count clock source (MHz) of the peripheral module assigned for outputting the phase adjustment signal B	
Remark	ST_SYSTEM_PARAM is a structure. For details of system information settings, see section 6.3.2, Structures for R_RSLV_SetSystemInfo.	

6.2.3 API Function for Acquiring the RDC Driver Setting Information

Item	Description	
Function name	R_RSLV_GetRdcDrvSettingInfo	
Argument	ST_RDC_DRV_SETTING_INFO *rdc_setting_info	Pointer to the driver setting information structure
Return value	unsigned char	Processing result
Function	Acquires information including counter values set in the driver. • Frequency of the excitation signal • Maximum value of the angle detection timer counter • Motor type	
Remark	ST_RDC_DRV_SETTING_INFO is a structure. For details, see section 6.3.3, Structure for R_RSLV_GetRdcDrvSettingInfo.	

6.2.4 API Function for Controlling Synchronous Starting of the MTU3 Timer Channels

Item	Description	
Function name	R_RSLV_MTU_SyncStart	
Argument	unsigned char start_ch	Channels to be started simultaneously (Multiple channels should be specified.)
Return value	unsigned char	Processing result
Function	Simultaneously starts the specified channels of MTU3.	
Remark	If MTU3_0 is used to generate the angle error correction signal, do not start it and the angle error correction signal timer simultaneously.	

6.2.5 API Function for Acquiring the RDC Driver Version Information

Item	Description	
Function name	R_RSLV_GetDriverVer	
Argument	unsigned long *drv_ver	Pointer to the RDC driver version storage buffer
Return value	unsigned char	Processing result
Function	Sets the RDC driver version in the specified buffer.	
Remark	Example: When the value is 0x00010000, the RDC driver version is Rev. 1.00.00.	

6.2.6 API Function for Starting the Output of the Angle Error Correction Signal

Item	Description	
Function name	R_RSLV_CSig_Start	
Argument	unsigned short phase_diff unsigned short amp_level	Phase shift amount Amplitude level
Return value	unsigned char	Processing result (the "normal end" information is always returned)
Function	Outputs the angle error correction signal according to the phase shift amount and amplitude level specified by arguments. For the ranges of setting values, see section 3.11.3, Adjustment of the Angle Error Correction Signal.	
Remark	• This API function sets the output of the angle error correction signal according to the arguments. Before changing the settings, be sure to execute the R_RSLV_CSig_Stop function to stop the signal.	

6.2.7 API Function for Stopping the Output of the Angle Error Correction Signal

Item	Description	
Function name	R_RSLV_CSig_Stop	
Argument	void	
Return value	unsigned char	Processing result (the "normal end" information is always returned)
Function	Stops outputting the angle error correction signal.	
Remark	• Calling this API function immediately stops the signal output. • To change the correction signal settings, call this API function in advance to stop the signal output, and then execute the R_RSLV_CSig_Start function to re-set the correction signal settings.	

6.2.8 API Function for Updating the Duty Cycle of the Angle Error Correction Signal

Item	Description	
Function name	R_RSLV_INT_CSig_UpdatePwmDuty	
Argument	void	
Return value	unsigned char	Processing result
Function	Updates the PWM duty cycle of the angle error correction signal. Call this API function from the processing of the timer interrupt for updating the angle error correction duty cycle.	
Remark		

6.2.9 API Function for Synchronously Starting the Angle Error Correction Signal

Item	Description	
Function name	R_RSLV_INT_CSig_SyncStart	
Argument	void	
Return value	unsigned char	Processing result
Function	Starts outputting the angle error correction signal in synchronization with the excitation signal. Call this API function from the interrupt processing in synchronization with the excitation signal.	
Remark		

6.2.10 API Function for Starting the Angle Detection Timer

Item	**Description**	
Function name	R_RSLV_Capture_Start	
Argument	void	
Return value	unsigned char	Processing result
Function	Enables input capture facility interrupts and starts the timer.	
Remark		

6.2.11 API Function for Acquiring the Angle Detection Value

Item	Description	
Function name	R_RSLV_INT_GetCaptureCount	
Argument	void	
Return value	unsigned char	Processing result
Function	Acquires the counter value detected by the input capture facility. • The counter value can be acquired using the following API functions. Current position (falling edge): R_RSLV_GetAngleCountFirstEdge Difference between previous and current positions (between falling edges): R_RSLV_GetAngleDifferenceFirstEdge Current position (rising edge): R_RSLV_GetAngleCountSecondEdge Difference between previous and current positions (between rising edges): R_RSLV_GetAngleDifferenceSecondEdge • Trigger edge information can be acquired using the following API function. R_RSLV_GetCaptureEdge	
Remark		

6.2.12 API Function for Acquiring the Trigger Information of the Interrupt for Acquiring the Angle Detection Value

Item	Description	
Function name	R_RSLV_GetCaptureEdge	
Argument	unsigned char *cap_edge	Variable to store angle detection trigger information
Return value	unsigned char	Processing result
Function	Acquires the trigger information of the interrupt generated by angle detection. (Rising edge or falling edge can be determined according to the port level.)	
Remark		

6.2.13 API Function for Acquiring the Resolver Angle Count (Acquisition Trigger: Falling Edge)

Item	Description	
Function name	R_RSLV_GetAngleCountFirstEdge	
Argument	unsigned short *angle_cnt	Pointer to the counter value storage
Return value	unsigned char	Processing result
Function	Acquires the counter value detected by the input capture facility.	
Remark	• The counter value detected on the falling edge of the angle signal is acquired. • Use the R_RSLV_INT_GetCaptureCount function to acquire the counter value.	

6.2.14 API Function for Acquiring the Resolver Angle Difference Count (Acquisition Trigger: Falling Edge)

Item	Description	
Function name	R_RSLV_GetAngleDifferenceFirstEdge	
Argument	signed short *angle_diff_cnt	Pointer to the difference value storage
Return value	unsigned char	Processing result
Function	Acquires the difference between the previous captured counter value and the current captured value.	
Remark	• The counter values detected on the falling edges of the angle signal are used for calculation. • Use the R_RSLV_INT_GetCaptureCount function to acquire the counter value.	

6.2.15 API Function for Acquiring the Resolver Angle Count (Acquisition Trigger: Rising Edge)

Item	Description	
Function name	R_RSLV_GetAngleCountSecondEdge	
Argument	unsigned short *angle_cnt	Pointer to the counter value storage
Return value	unsigned char	Processing result
Function	Acquires the counter value detected by the input capture facility.	
Remark	• The counter value detected on the rising edge of the angle signal is acquired. • Use the R_RSLV_INT_GetCaptureCount function to acquire the counter value.	

6.2.16 API Function for Acquiring the Resolver Angle Difference Count (Acquisition Trigger: Rising Edge)

Item	Description	
Function name	R_RSLV_GetAngleDifferenceSecondEdge	
Argument	signed short *angle_diff_cnt	Pointer to the difference value storage
Return value	unsigned char	Processing result
Function	Acquires the difference between the previous captured counter value and the current captured value.	
Remark	• The counter values detected on the rising edges of the angle signal are used for calculation. • Use the R_RSLV_INT_GetCaptureCount function to acquire the counter value.	

6.2.17 API Function for Starting the Output of the Excitation Signal

Item	Description	
Function name	R_RSLV_ESig_Start	
Argument	void	
Return value	unsigned char	Processing result
Function	Starts outputting the excitation signal.	
Remark		

6.2.18 API Function for Stopping the Output of the Excitation Signal

Item	Description	
Function name	R_RSLV_ESig_Stop	
Argument	void	
Return value	unsigned char	Processing result
Function	Stops outputting the excitation signal.	
Remark	When the excitation signal is stopped, the angle error correction signal and the angle detection timer also stop.	

6.2.19 API Function for Setting the Timing to Start the Excitation Signal Output

Item	Description	
Function name	R_RSLV_ESigCapStartTiming	
Argument	unsigned short esig_start_tcnt unsigned short cap_start_tcnt	Setting of the excitation signal output start timing Setting of the timing to start the angle detection timer
Return value	unsigned char	Processing result
Function	Sets the timing to start outputting the excitation signal and the timing to start the angle detection timer.	
Remark	If the specified value is greater than the upper limit of the timing value, the upper limit value is set and the "NG" information is returned as the processing result.	

6.2.20 API Function for Counting the Wait Time

Item	Description	
Function name	R_RSLV_INT_ESigCounter	
Argument	void	
Return value	unsigned char	Processing result
Function	Starts counting down by the wait timer in the automatic calibration processing.	
Remark	Counting down is performed only in the automatic calibration processing.	

6.2.21 API Function for Starting the Output of the Phase Adjustment Signals

Item	Description	
Function name	R_RSLV_Phase_AdjStart	
Argument	void	
Return value	unsigned char	Processing result
Function	Starts outputting the phase adjustment signals.	
Remark	This API function starts the timers for the phase adjustment signals specified by F_PHASE_A and F_PHASE_B.	

6.2.22 API Function for Stopping the Output of the Phase Adjustment Signals

Item	Description	
Function name	R_RSLV_Phase_AdjStop	
Argument	void	
Return value	unsigned char	Processing result
Function	Stops outputting the phase adjustment signals.	
Remark		

6.2.23 API Function for Setting the Phase Adjustment Signal Duty Cycle in the Buffer

Item	Description	
Function name	R_RSLV_Phase_AdjUpdateBuff	
Argument	unsigned short duty unsigned char ch	Duty value to be set Selection of phase A or phase B (0: Phase A, 1: Phase B)
Return value	unsigned char	Processing result
Function	Sets the duty cycle of the phase adjustment signal in the buffer.	
Remark		

6.2.24 API Function for Setting the Phase Adjustment Signal Duty Cycle in the Register

Item	Description	
Function name	R_RSLV_Phase_AdjUpdate	
Argument	void	
Return value	unsigned char	Processing result
Function	Sets the duty cycle of the phase adjustment signal in the register.	
Remark	This API function updates the duty value when the duty value set in the buffer differs from the current duty value.	

6.2.25 API Function for Reading the Phase Adjustment Signal Duty Cycle from the Buffer

Item	Description	
Function name	R_RSLV_Phase_AdjReadBuff	
Argument	unsigned short *duty unsigned char ch	Duty value of the phase adjustment signal Selection of phase A or phase B (0: Phase A, 1: Phase B)
Return value	unsigned char	Processing result
Function	Reads the duty cycle of the phase adjustment signal from the storage buffer.	
Remark		

6.2.26 API Function for Setting RDC Initial Values

Item	Description	
Function name	R_RSLV_Rdc_VariableInit	
Argument	unsigned char *u1_init_data	Pointer to a set of data for initializing RDC communications
Return value	unsigned char	Processing result
Function	Sets data for initializing RDC communications.	
Remark	RDC registers to be initialized PS1 (02h: Power-saving control register 1) PS2 (04h: Power-saving control register 2) PS3 (0Ah: Power-saving control register 3) ALMOUT (16h: ALARM# output setting register) GCGSL (2Eh: Differential amplification circuit gain selection register) CSACTL (42h: Shunt current amplification circuit control register)	

6.2.27 API Function for Executing the RDC Initialization Sequence

Item	Description	
Function name	R_RSLV_Rdc_Init_Sequence	
Argument	unsigned short *init_status	Initialization processing state ("processing in progress" or "processing terminated")
Return value	unsigned char	Processing result
Function	Executes the RDC initialization sequence.	
Remark		

6.2.28 API Function for Handling RDC Communications

Item	Description	
Function name	R_RSLV_Rdc_Communication	
Argument	void	
Return value	unsigned char	Processing result
Function	Handles communications with the RDC. The sequence of communications is made to progress through repeated calls of this API function from the application.	
Remark	Call this API function periodically to control the sequence of communications.	

6.2.29 API Function for Writing to an RDC Register

Item	Description	
Function name	R_RSLV_Rdc_RegWrite	
Argument	unsigned char *write_status	Write state
Return value	unsigned char	Processing result
Function	Writes the value specified by an argument to the specified RDC register.	
Remark		

6.2.30 API Function for Reading from an RDC Register

Item	Description	
Function name	R_RSLV_Rdc_RegRead	
Argument	unsigned char address	RDC register address to be read
Return value	unsigned char	Processing result
Function	Reads the RDC register value from the address specified by the argument and stores it in the buffer.	
Remark	Use the R_RSLV_Rdc_GetRegisterVal function to acquire the read data.	

6.2.31 API Function for Acquiring the RDC Register Access State

Item	Description	
Function name	R_RSLV_Rdc_ChkIfRun	
Argument	void	
Return value	unsigned char	Processing result
Function	Returns the processing result regarding whether the RDC register was accessed (read or written to).	
Remark		

6.2.32 API Function for Reading Data from the RDC Register Buffer

Item	Description	
Function name	R_RSLV_Rdc_GetRegisterVal	
Argument	unsigned char *rd_data unsigned char address	Pointer to the read data RDC register address to be read
Return value	unsigned char	Processing result
Function	Reads the buffer value of the RDC register address specified by an argument.	
Remark		

6.2.33 API Function for Writing Data to the RDC Register Buffer

Item	Description	
Function name	R_RSLV_Rdc_SetRegisterVal	
Argument	unsigned char wt_data unsigned char address	Data to be written RDC register address to be written to
Return value	unsigned char	Processing result
Function	Writes the specified data to the buffer for the RDC register at the address specified by an argument.	
Remark		

6.2.34 API Function for Calling the Callback Processing for the RDC Communication Transmit/Receive End Interrupt

Item	Description	
Function name	R_RSLV_Rdc_CallComEndCb	
Argument	void	
Return value	unsigned char	Processing result
Function	Calls the transmit/receive end interrupt callback processing and terminates read or write access from the driver to the RDC.	
Remark	Call this API function from the transmit interrupt processing or receive interrupt processing.	

6.2.35 API Function for Calling the Callback Processing for the RDC Communication Error Interrupt

Item	Description	
Function name	R_RSLV_Rdc_CallErrorCb	
Argument	void	
Return value	unsigned char	Processing result
Function	Calls the error interrupt callback processing.	
Remark		

6.2.36 API Function for Reporting Errors in RDC Communications

Item	Description	
Function name	R_RSLV_RdcCom_GetErrorInfo	
Argument	unsigned char *err_info	Storage of RDC communication error information
Return value	unsigned char	Processing result
Function	Acquires error information in RDC communications. RSLV_MD_OK: No error occurred. RSLV_MD_ERROR: An error occurred.	
Remark		

6.2.37 API Function for Starting RDC Alarm Cancellation

Item	Description	
Function name	R_RSLV_Rdc_AlarmCancelStart	
Argument	void	
Return value	unsigned char	Processing result
Function	Starts the processing for cancelling an alarm in the RDC.	
Remark		

6.2.38 API Function for Controlling the RDC Alarm Cancellation Sequence

Item	Description	
Function name	R_RSLV_Rdc_AlarmCancel	
Argument	void	
Return value	unsigned char	Processing result
Function	Performs the sequence for cancelling the alarm detection state of the RDC.	
Remark	Call this API function periodically for sequence control.	

6.2.39 API Function for Adjusting the Gain and Phase of the Resolver Signals

Item	Description	
Function name	R_RSLV_ADJST_GainPhase	
Argument	unsigned char u1_call_state	User-specified state Selection of whether to perform or cancel the gain and phase adjustment of the resolver signals 0: Performed (Constant: ADJST_USRREQ_RUN) 1: Cancelled (Constant: ADJST_USRREQ_STOP)
Return value	st_adjst_gainphase_return_t	Processing result
Function	Performs the sequence for adjusting the gain and phase of the resolver signals.	
Remark	st_adjst_gainphase_return_t is a structure. For details of the information regarding the end of resolver signal gain and phase adjustment, the gain adjustment result, the phase adjustment result, see section 6.3.4, Structure for R_RSLV_ADJST_GainPhase.	

6.2.40 API Function for Adjusting the Angle Error Correction Signal

Item	Description	
Function name	R_RSLV_ADJST_Carrier	
Argument	st_adjst_carrier_arg_t arg_value	User-specified state Motor control information
Return value	st_adjst_carrier_return_t	Processing result
Function	Performs the sequence for adjusting the angle error correction signal.	
Remark	st_adjst_carrier_arg_t and st_adjst_carrier_return_t are structures. For details of these structures, see section 6.3.5, Structures for R_RSLV_ADJST_Carrier.	

6.2.41 API Function for Setting the Pointer to the User-Created Callback Function

Item	Description	
Function name	R_RSLV_ADJST_SetPtrFunc	
Argument	st_ptr_func_arg_t *ptr_arg	Pointer to the user-created function
Return value	unsigned char	Processing result
Function	Sets the pointer to the user-created callback function in the pointer variable used in the automatic calibration processing.	
Remark	st_ptr_func_arg_t is a structure. For the setting of the callback function pointer, see section 6.3.6, Structure for R_RSLV_ADJST_SetPtrFunc.	

6.2.42 API Function for Acquiring the A/D Conversion State

Item	Description	
Function name	R_RSLV_ADJST_Ad_Processing	
Argument	void	
Return value	unsigned char	Processing result (A/D conversion execution state)
Function	Returns the A/D conversion execution state. While A/D conversion is in progress, 1 is returned. In other cases, 0 is returned.	
Remark		

6.2.43 API Function for Detecting Disconnection

Item	Description	
Function name	R_RSLV_DiscDetection_Seq	
Argument	st_rdc_ddmnt_arg_t arg_value	Structure for processing detection of disconnection
Return value	unsigned char	Processing result
Function	Performs the sequence for detecting disconnection.	
Remark	st_rdc_ddmnt_arg_t is a structure. For details of the structure, see section 6.3.7, Structure for R_RSLV_DiscDetection_Seq.	

6.3 Structures

The following API functions use respective structures. This section describes the structures for these API functions.

- R_RSLV_SetFuncTable (section 6.2.1)
- R_RSLV_SetSystemInfo (section 6.2.2)
- R_RSLV_GetRdcDrvSettingInfo (section 6.2.3)
- R_RSLV_ADJST_GainPhase (section 6.2.39)
- R_RSLV_ADJST_Carrier (section 6.2.40)
- R_RSLV_ADJST_SetPtrFunc (section 6.2.41)
- R_RSLV_DiscDetection_Seq (section 6.2.43)

6.3.1 Structure for R_RSLV_SetFuncTable

The definitions of the set_func argument and the ST_FUNCTION_TABLE structure for the R_RSLV_SetFuncTable API function are shown below.

API function: R_RSLV_SetFuncTable (unsigned char set_func,
ST_FUNCTION_TABLE user_func_table)

Table 6-2 Macro-Defined Names Specified in set_func of R_RSLV_SetFuncTable

Variable Name	Type	Description		Defined Value	Macro-Defined Name
set_func	unsigned char	Driver facility	ESIG1	0	F_ESIG1
			ESIG2_1	1	F_ESIG2_1
			ESIG2_2	2	F_ESIG2_2
			ESIG12	3	F_ESIG12
			CSIG	4	F_CSIG
			PHASE_A	5	F_PHASE_A
			PHASE_B	6	F_PHASE_B
			PHASE_AB	7	F_PHASE_AB
			CAPTURE	8	F_CAPTURE
			CSIG_UPD_TIMER	9	F_CSIG_UPD_TIMER
			RDC_COM	10	F_RDC_COM
			RDC_CLK	11	F_RDC_CLK

Table 6-3 Structure Definition for R_RSLV_SetFuncTable

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
ST_FUNCTION_TABLE (argument)	void (*Start)(unsigned char u1_sync_start)	void	Pointer to the function for starting the timer	—	—	—
	void (*Stop)(void)	void	Pointer to the function for stopping the timer	—	—	—
	void (*GetTcnt)(unsigned short *tcnt)	void	Pointer to the function for acquiring the timer value	—	—	—
	void (*SetTcnt)(unsigned short tcnt)	void	Pointer to the function for setting the timer value	—	—	—
	void (*GetDuty)(unsigned short *duty)	void	Pointer to the function for acquiring the duty value	—	—	—
	void (*SetDuty)(unsigned short duty)	void	Pointer to the function for setting the duty value	—	—	—
	void (*SetDuty_2val)(unsigned short ch, unsigned short duty)	void	Pointer to the function for setting the duty value (for PHASE_AB)	—	—	—
ST_FUNCTION_TABLE (argument)	void (*GetCaptureVal)(unsigned short *capture_val)	void	Pointer to the function for acquiring the angle detection value	—	—	—
	void (*GetPortLevel)(unsigned char *port_level)	void	Pointer to the function for acquiring the port level	—	—	—
	void (*ComSendReceive) (unsigned short *tx_buf, unsigned short tx_num, unsigned short *rx_buf)	void	Pointer to the function for starting RDC transmission/reception	—	—	—

6.3.2 Structures for R_RSLV_SetSystemInfo

The structure definitions of the ST_SYSTEM_PARAM and ST_USER_PERI_PARAM arguments for the R_RSLV_SetSystemInfo API function are shown below.

API function: R_RSLV_SetSystemInfo (ST_SYSTEM_PARAM *rdc_sys_param,
ST_USER_PERI_PARAM *user_peri_param)

Table 6-4 Structure Definitions for R_RSLV_SetSystemInfo

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
ST_SYSTEM_PARAM	u1_esig_freq	unsigned char	Frequency of the excitation signal	5 kHz	1	R_ESIG_SET_FREQ_5K
				10 kHz	2	R_ESIG_SET_FREQ_10K
				20 kHz	3	R_ESIG_SET_FREQ_20K
	u1_csig_freq	unsigned char	Frequency of the output angle error correction signal	200 kHz	1	R_CSIG_SET_FREQ_200K
				400 kHz	2	R_CSIG_SET_FREQ_400K
	u1_csig_upd_duty_cycle	unsigned char	Number of update times of the angle error correction duty cycle	Two times	1	R_CSIG_SET_DCNT_02
				Four times	2	R_CSIG_SET_DCNT_04
	u1_mtu3_sync_start	unsigned char	Excitation signal timer and angle detection timer start flag	Synchronous start*1	0	SYNCMD_ESIG_API
				Synchronous start*2	1	SYNCMD_OTHER_API
	u1_motor_kind	unsigned char	Motor type	BLDC type	1	MOTOR_BLDC
				Stepper type	2	MOTOR_STM
	u1_mntout_type	unsigned char	Output mode of the MNTOUT pin of the RDC	AC output	1	RSLV_MNTOUT_TYPE_AC
				DC output	2	RSLV_MNTOUT_TYPE_DC
ST_USER_PERI_PARAM	f_esig1_peri_clk_src	float	Count clock source of the peripheral module assigned for outputting the excitation signal		—	—
	f_csig_peri_clk_src	float	Count clock source of the peripheral module assigned for outputting the angle error correction signal		—	—
	f_capture_peri_clk_src	float	Count clock source of the peripheral module assigned for inputting the angle signal		—	—
	f_csig_upd_timer_peri_clk_src	float	Count clock source of the peripheral module assigned for updating the duty cycle of the angle error correction signal		—	—
	f_phase1_peri_clk_src	float	Count clock source of the peripheral module assigned for outputting the phase adjustment signal A		—	—
	f_phase2_peri_clk_src	float	Count clock source of the peripheral module assigned for outputting the phase adjustment signal B		—	—

Note 1. When SYNCMD_ESIG_API is specified, start counting by the timer for the excitation signal and the timer for angle detection in the API function for starting the output of the excitation signal.

Note 2. When SYNCMD_OTHER_API is specified, call the API function for starting the angle detection timer from the API function for controlling synchronous starting of the MTU3 timer channels or the excitation signal interrupt processing and start counting.

6.3.3 Structure for R_RSLV_GetRdcDrvSettingInfo

The structure definition of the ST_RDC_DRV_SETTING_INFO argument for the R_RSLV_GetRdcDrvSettingInfo API function is shown below.

API function: R_RSLV_GetRdcDrvSettingInfo (ST_RDC_DRV_SETTING_INFO *rdc_setting_info)

Table 6-5 Structure Definition for R_RSLV_GetRdcDrvSettingInfo

Structure	Member Name	Type	Description			Remark
ST_RDC_DRV_SETTING_INFO	f_esig_freq	float	Excitation signal frequency 5 kHz: 5000, 10 kHz: 10000, 20 kHz: 20000			
	u2_capture_cnt_max	unsigned short	Maximum value of the angle detection timer counter			
	u1_motor_kind	unsigned char	Motor Type	Defined Value	Macro-Defined Name	
			BLDC type	1	MOTOR_BLDC	
			Stepper type	2	MOTOR_STM	

6.3.4 Structure for R_RSLV_ADJST_GainPhase

The structure definition of the st_adjst_gainphase_return_t return value for the R_RSLV_ADJST_GainPhase API function is shown below.

API function: st_adjst_gainphase_return_t R_RSLV_ADJST_GainPhase (unsigned char u1_call_state)

Table 6-6 Structure Definition for R_RSLV_ADJST_GainPhase (1/2)

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
st_adjst_gainphase_return_t (return value)	u1_adjst_state	unsigned char	Execution in progress	Waiting for internal processing	0	ADJST_APIINFO_RUN_MODE
			Normal end	Phase adjustment is successfully completed.	1	ADJST_APIINFO_END_NORMAL
			Gain adjustment: Terminated with an upper-limit amplification error	When the adjustment result does not fall within the acceptable range even if the upper-limit amplification value of the resolver phase A signal of the RDC is reached	3	ADJST_APIINFO_ERR_GAIN_HI_LMT
			Gain adjustment: Terminated with a lower-limit amplification error	When the adjustment result does not fall within the acceptable range even if the lower-limit amplification value of the resolver phase A signal of the RDC is reached	4	ADJST_APIINFO_ERR_GAIN_LO_LMT
			Gain adjustment: Terminated with an unstable gain error	When the adjustment result of the resolver phase A signal of the RDC does not fall within the acceptable range	5	ADJST_APIINFO_ERR_GAIN_SWAY
			Phase adjustment: Terminated with a phase A upper-limit or phase B lower-limit duty value error	When the adjustment result does not fall within the acceptable range even if the phase A upper-limit or phase B lower-limit duty value is reached	6	ADJST_APIINFO_ERR_PHASE_AHI_BLO
			Phase adjustment: Terminated with a phase A lower-limit or phase B upper-limit duty value error	When the adjustment result does not fall within the acceptable range even if the phase A lower-limit or phase B upper-limit duty value is reached	7	ADJST_APIINFO_ERR_PHASE_ALO_BHI
			Phase adjustment: Terminated with an unstable phase error	When the phase B duty cycle does not reach the upper-limit or lower-limit value and the adjustment result does not fall within the acceptable range	8	ADJST_APIINFO_ERR_PHASE_SWAY
			Phase adjustment: Terminated with a phase adjustment error	When the difference between phase A count and phase B count exceeds the acceptable adjustment range	9	ADJST_APIINFO_ERR_PHASE_OUT_RANGE
			Gain or phase adjustment: Terminated with an RDC error	When acquisition of the monitoring signal or phase A or phase B count is not successful	10	ADJST_APIINFO_ERR_RDC
			Terminated by cancellation	When execution is cancelled by the u1_call_state setting	13	ADJST_APIINFO_END_USER_STOP

Table 6-6 Structure Definition for R_RSLV_ADJST_GainPhase (2/2)

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
st_adjst_gainphase_ return_t (return value)	u1_res_dlcgsl	unsigned char	u1_adjst_state = "execution in progress (0)"	—	0xFF	—
			u1_adjst_state = "normal end (1)"	RDC register DLCGSL adjustment result	0 to 31	—
			u1_adjst_state = "error (3 to 10, or 13)"	Value of the RDC register DLCGSL specified by the user before adjustment	—	—
	u2_res_a_duty	unsigned short	u1_adjst_state = "execution in progress (0)"	—	0xFFFF	—
			u1_adjst_state = "normal end (1)"	Result of PWM duty cycle adjustment for phase A [%]	5 to 90	—
			u1_adjst_state = "error (3 to 10, or 13)"	Phase A PWM duty cycle specified by the user before adjustment	—	—
	u2_res_b_duty	unsigned short	u1_adjst_state = "execution in progress (0)"	—	0xFFFF	—
			u1_adjst_state = "normal end (1)"	Result of PWM duty cycle adjustment for phase B [%]	5 to 90	—
			u1_adjst_state = "error (3 to 10, or 13)"	Phase B PWM duty cycle specified by the user before adjustment	—	—

6.3.5 Structures for R_RSLV_ADJST_Carrier

The structure definitions of the st_adjst_carrier_return_t return value and the st_adjst_carrier_arg_t argument for the R_RSLV_ADJST_Carrier API function are shown below.

API function: st_adjst_carrier_return_t R_RSLV_ADJST_Carrier (st_adjst_carrier_arg_t arg_value)

Table 6-7 Structure Definitions for R_RSLV_ADJST_Carrier

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
st_adjst_carrier_return_t (return value)	adjst_state	unsigned char	Angle error correction signal adjustment state	Execution in progress	0	ADJST_APIINFO_RUN_MODE
				Normal end	1	ADJST_APIINFO_END_NORMAL
				Waiting for control completion	2	ADJST_APIINFO_WAITING
				Terminated with an angle error correction error	11	ADJST_APIINFO_ERR_CARRIER
				Terminated with a motor rotation error	12	ADJST_APIINFO_ERR_MOTOR
				Terminated by cancellation	13	ADJST_APIINFO_END_USER_ STOP
	req_mtr_ctrl	unsigned char	Motor control request for angle error correction signal adjustment	No control request	0	ADJST_APIREQ_NONE
				Position control request	1	ADJST_APIREQ_POS_CTRL
				Position control stop request	2	ADJST_APIREQ_POS_STOP
				Speed control request	3	ADJST_APIREQ_SPD_CTRL
				Speed control stop request	4	ADJST_APIREQ_SPD_STOP
	mtr_ctrl_data	unsigned short	req_mtr_ctrl (1)	Position control angle	0 to 360	—
			req_mtr_ctrl (3)	Speed data [rpm]	—	—
	res_ccgsl	unsigned char	Adjustment result	Adjustment in progress	0xFF	—
				Terminated with an error	User-set value	—
	res_csig_shift	unsigned short	Adjustment result: Phase shift amount	Adjustment in progress	0xFF	—
				Adjustment completed	*	—
				Terminated with an error	User-set value	—
	res_csig_amp	unsigned short	Adjustment result: Amplitude value	Adjustment in progress	0xFF	—
				Adjustment completed CSIG: 200 kHz CSIG: 400 kHz	*	—
				Terminated with an error	User-set value	—
st_adjst_carrier_arg_t (argument)	call_state	unsigned char	Execution or cancellation of angle error correction signal adjustment	Execution continued	0	ADJST_USRREQ_RUN
				Execution cancelled	1	ADJST_USRREQ_STOP
	req_state	unsigned char	Motor control execution state	Motor control completed	0	ADJST_USRINFO_COMPLETE
				Motor control in progress	1	ADJST_USRINFO_PROCESSING

Note: * For the defined value, see section 3.11.3, Adjustment of the Angle Error Correction Signal.

6.3.6 Structure for R_RSLV_ADJST_SetPtrFunc

The structure definition of the st_ptr_func_arg_t argument for the R_RSLV_ADJST_SetPtrFunc API function is shown below.

API function: void R_RSLV_ADJST_SetPtrFunc (st_ptr_func_arg_t *ptr_arg)

Table 6-8 Structure Definition for R_RSLV_ADJST_SetPtrFunc

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
st_ptr_func_arg_t (argument)	(*ad_data)(void);	unsigned short	Pointer to the function for referencing A/D data	—	—	—
	(*ad_ctrl)(unsigned char);	void	Pointer to the function for starting or stopping A/D conversion	—	—	—
	(*ad_peri_adjst)(void);	void	Pointer to the function for adjusting the settings of the A/D converter	—	—	—
	(*ad_peri_user)(void);	void	Pointer to the user-created function for setting the AD converter	—	—	—
	resolver_pole_num	unsigned short	Number of poles in the resolver of the motor to be used	—	—	—
	*mtr_speed	float	Pointer to the variable for referencing the speed data	[rad/s]	—	—
	req_speed	unsigned short	Reference speed of error when automatic calibration is executed	[rpm]	—	—

6.3.7 Structure for R_RSLV_DiscDetection_Seq

The structure definition of the st_rdc_ddmnt_arg_t argument for the R_RSLV_DiscDetection_Seq API function is shown below.

API function: unsigned char R_RSLV_DiscDetection_Seq (st_rdc_ddmnt_arg_t arg_value)

Table 6-9 Structure Definition for R_RSLV_DiscDetection_Seq

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
st_rdc_ddmnt_arg_t (argument)	call_state	unsigned char	Disconnection detection processing state	Execution in progress	0	DDMNT_APIINFO_RUN_MODE
				Disconnection not detected	1	DDMNT_APIINFO_END_NOMAL
				Disconnection detected	2	DDMNT_APIINFO_ERR_DISCONNECT
				Terminated by cancellation	3	DDMNT_APIINFO_ENC_USER_STOP
	wire_state	unsigned char	Resolver line state	Normal	0	DDMNT_WIRE_STATE_NOMAL
				Abnormal	1	DDMNT_WIRE_STATE_ABNOMAL

7. Examples of Implementing API Functions

The following shows an example of a software architecture using this driver.

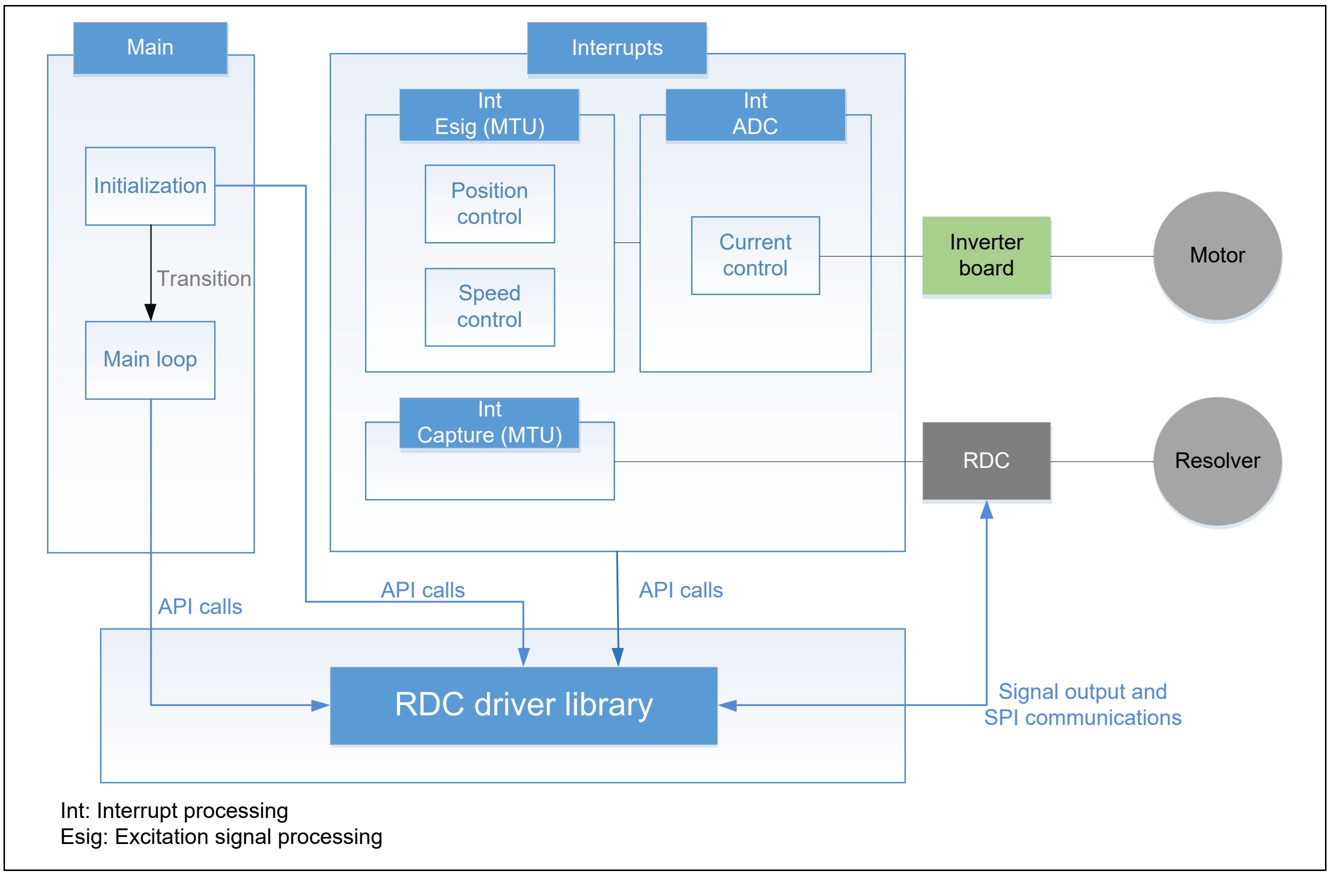


Figure 7.1 Example of Software Architecture

The driver is initialized in the initialization processing. After that, the main loop calls API functions for the execution of processing such as starting the generation of signals and the interrupt processing calls API functions to acquire rotor positional information (input capture function) or to synchronize signals and so on. Furthermore, this driver handles SPI communications with the RDC and the output of signals.

The following describes implementation of each processing.

7.1 Preparation for the Use of Peripheral Modules

The user should create functions for setting up peripheral modules by using the SC. The SC can generate functions for initializing peripheral modules and starting or stopping the timers in peripheral modules. The user should also prepare the necessary additional functions that are not generated by the SC as user-created code.

These functions for handling peripheral modules as well as the user-created code are prepared in the sample code supplied together with this application note: use them as necessary.

7.1.1 SC Settings

Use the SC to set up the peripheral module assigned to each facility of the driver. For the recommended settings of the assigned peripheral modules, see section 5, Settings for Peripheral Modules. When the SC is used, the Config_(peri_func).c, Config_(peri_func)_user.c, and Config_(peri_func).h file are generated. For the name of each file, see section 4.1, Folder and File Configuration.

7.1.2 User-Created Code

In addition to the code generated by the SC, the user should create the following functions for accessing peripheral modules, which should be set in function tables.

- Function for acquiring the timer counter value
- Function for setting the timer counter value
- Function for acquiring the duty value
- Function for setting the duty value
- Function for acquiring the capture value
- Function for acquiring the port level

For the settings in function tables, see section 5.4, Setting up Function Tables.

7.2 Initialization

7.2.1 Initialization of the MCU

The R_Systeminit function is automatically created when a code is generated by the SC and the functions for initializing peripheral modules are included in this function. When the MCU is started, the R_Systeminit function is called and the peripheral modules are initialized.

Function for initializing a peripheral module: R_Config_(peri_func)_Create()

7.2.2 Initialization of the Driver

To initialize the driver, the following settings are necessary after initialization of the MCU.

- System information
- Timer start timing for the excitation signal output and angle signal input
- Function tables
- Pointer to the callback function for automatic calibration
- Initial values of the RDC registers

See section 7.2.3.2, Initialization of the Driver, for the sample code.

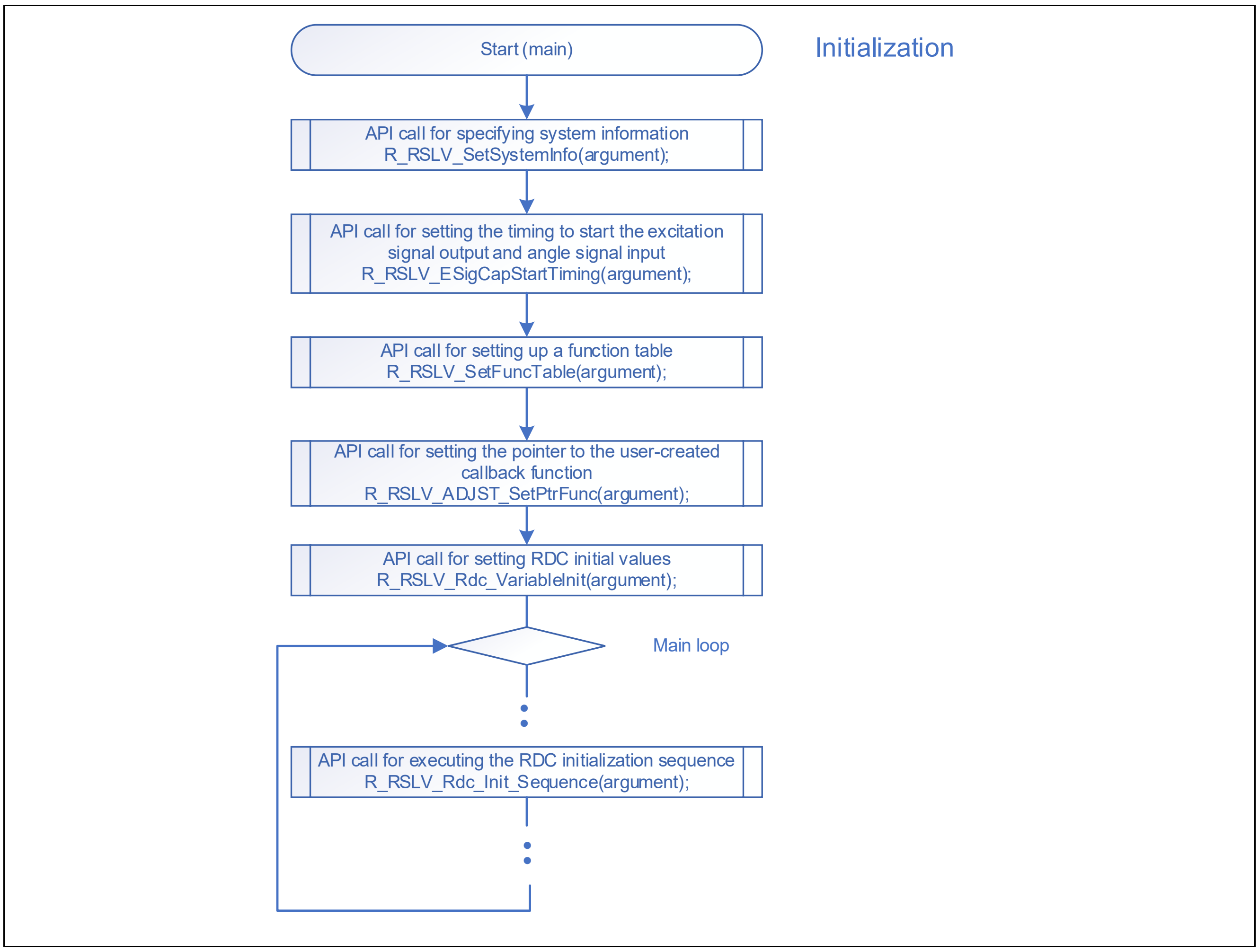
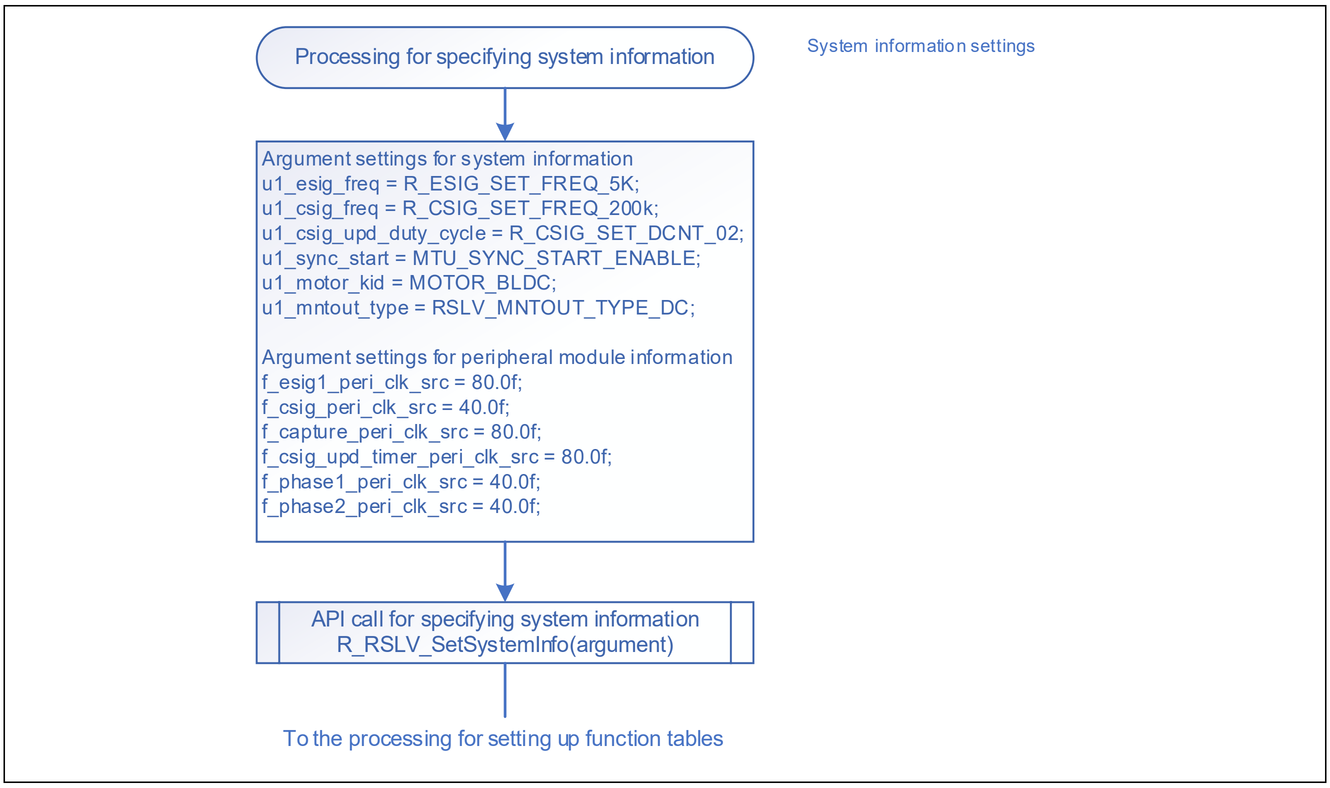

Initialization
Start (main)
API call for specifying system information
R_RSLV_SetSystemInfo(argument);
API call for setting the timing to start the excitation signal output and angle signal input
R_RSLV_ESigCapStartTiming(argument);
API call for setting up a function table
R_RSLV_SetFuncTable(argument);
API call for setting the pointer to the user-created callback function
R_RSLV_ADJST_SetPtrFunc(argument);
API call for setting RDC initial values
R_RSLV_Rdc_VariableInit(argument);
Main loop
API call for executing the RDC initialization sequence
R_RSLV_Rdc_Init_Sequence(argument);

Figure 7.2 Initialization Flow

7.2.2.1 Specifying the System Information

Specify the system information, such as the excitation frequency, the angle error correction signal frequency, the number of updates of the angle error correction signal, and the clock source for the peripheral module assigned to each driver facility, and execute the API function for specifying the system information. To divide the frequency of the clock for counting, specify the value of "clock source/divider value".

See section 7.2.3.2, Initialization of the Driver, for the sample code.


System information settings
Processing for specifying system information
API call for specifying system information
R_RSLV_SetSystemInfo(argument)
To the processing for setting up function tables

```
Argument settings for system information
u1_esig_freq = R_ESIG_SET_FREQ_5K;
u1_csig_freq = R_CSIG_SET_FREQ_200k;
u1_csig_upd_duty_cycle = R_CSIG_SET_DCNT_02;
u1_sync_start = MTU_SYNC_START_ENABLE;
u1_motor_kid = MOTOR_BLDC;
u1_mntout_type = RSLV_MNTOUT_TYPE_DC;

Argument settings for peripheral module information
f_esig1_peri_clk_src = 80.0f;
f_csig_peri_clk_src = 40.0f;
f_capture_peri_clk_src = 80.0f;
f_csig_upd_timer_peri_clk_src = 80.0f;
f_phase1_peri_clk_src = 40.0f;
f_phase2_peri_clk_src = 40.0f;
```

Figure 7.3 Processing for Specifying the System Information

7.2.2.2 Specifying the Timer Start Timing for the Excitation Signal Output and Angle Signal Input

To specify the timing for starting the timers for the excitation signal output and angle signal input, use the API function for setting the timing to start the excitation signal output. The sample code executes this function in the driver initialization processing but it can be executed in any processing before starting the timers for the excitation signal output and angle signal input.

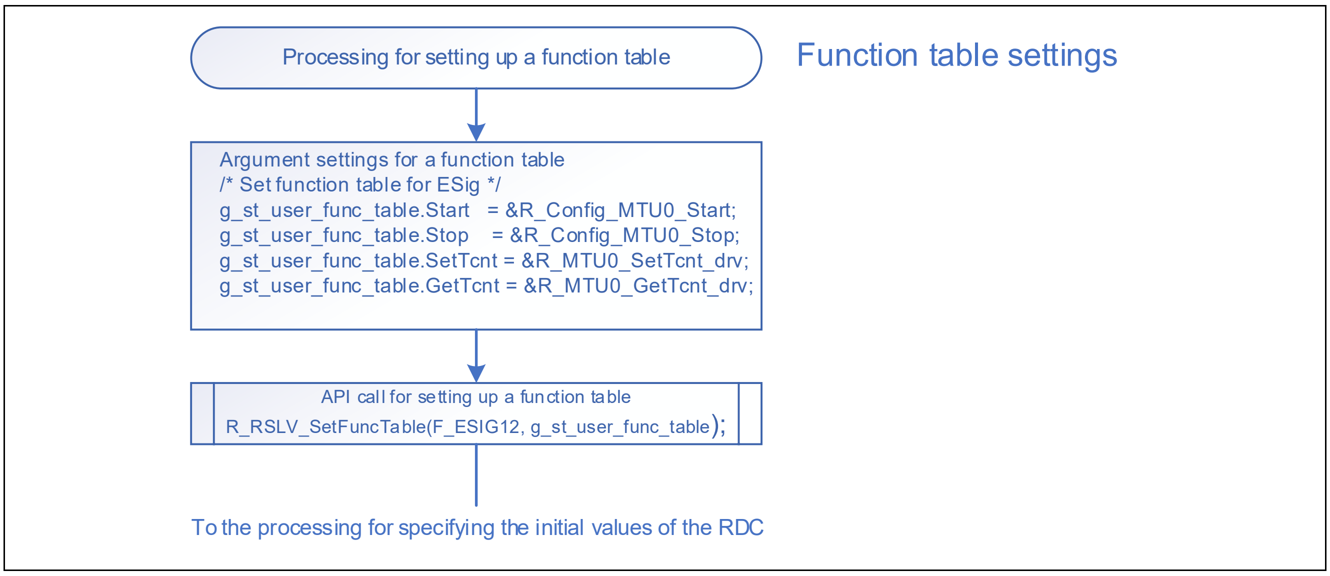
See section 7.2.3.2, Initialization of the Driver, for the sample code.

API function: R_RSLV_ESigCapStartTiming(DEF_DELAY_ADJ_ESIG, DEF_SFT_ADJ_ESIG);

7.2.2.3 Setting up Function Tables

To set up a function table, specify the code generated by the SC or created by the user in the target table and execute the API function for setting up a function table.

See section 7.2.3.2, Initialization of the Driver, for the sample code.


Processing for setting up a function table
Function table settings
Argument settings for a function table
API call for setting up a function table
To the processing for specifying the initial values of the RDC

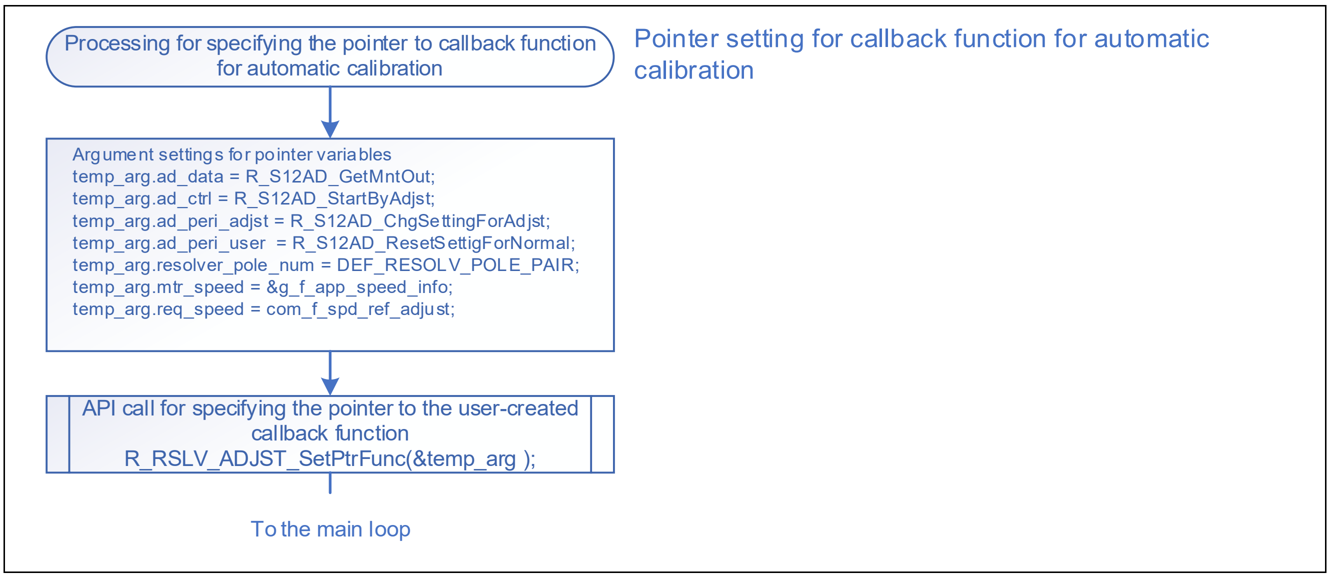
```
/* Set function table for ESig */
g_st_user_func_table.Start   = &R_Config_MTU0_Start;
g_st_user_func_table.Stop    = &R_Config_MTU0_Stop;
g_st_user_func_table.SetTcnt = &R_MTU0_SetTcnt_drv;
g_st_user_func_table.GetTcnt = &R_MTU0_GetTcnt_drv;

R_RSLV_SetFuncTable(F_ESIG12, g_st_user_func_table);
```

Figure 7.4 Processing for Setting up a Function Table

7.2.2.4 Specifying the Pointer to the User-Created Callback Function

To specify the pointer to the callback function for automatic calibration, set the pointer to the A/D conversion function for automatic calibration and the necessary values for the adjustment processing in the members of a structure and execute the API function for specifying the pointer to the user-created callback function.

See section 7.2.3.2, Initialization of the Driver, for the sample code.


Processing for specifying the pointer to callback function for automatic calibration
Pointer setting for callback function for automatic calibration
Argument settings for pointer variables
API call for specifying the pointer to the user-created callback function
To the main loop

```
temp_arg.ad_data = R_S12AD_GetMntOut;
temp_arg.ad_ctrl = R_S12AD_StartByAdjst;
temp_arg.ad_peri_adjst = R_S12AD_ChgSettingForAdjst;
temp_arg.ad_peri_user  = R_S12AD_ResetSettigForNormal;
temp_arg.resolver_pole_num = DEF_RESOLV_POLE_PAIR;
temp_arg.mtr_speed = &g_f_app_speed_info;
temp_arg.req_speed = com_f_spd_ref_adjust;

R_RSLV_ADJST_SetPtrFunc(&temp_arg );
```

Figure 7.5 Processing for Initial Settings for Automatic Calibration (Function Pointer Setting)

7.2.2.5 Specifying the Initial Values of the RDC

To initialize the registers in the RDC, use the API function for setting the RDC initial values. The user should specify the initial value of each register. For the registers to be set up, see section 6.2.26, API Function for Setting RDC .

See section 7.2.3.2, Initialization of the Driver, for the sample code.

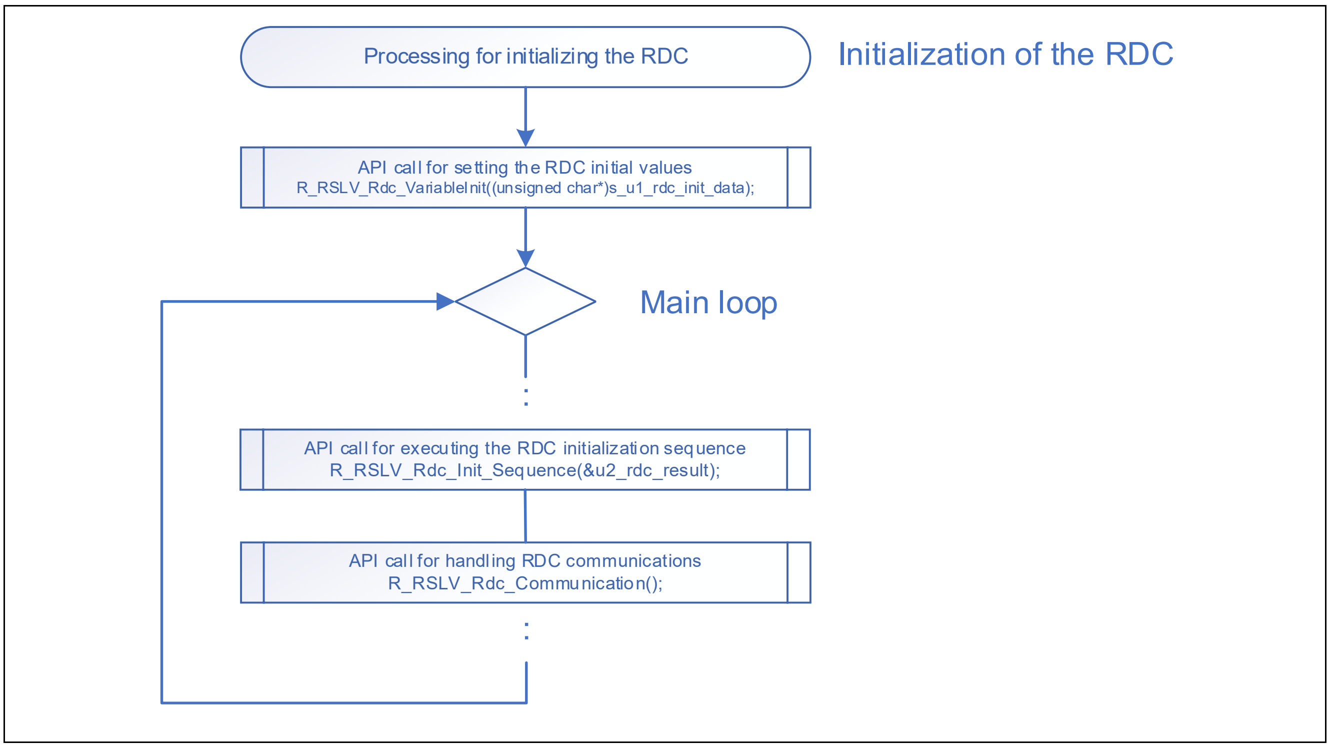

Processing for initializing the RDC
Initialization of the RDC
API call for setting the RDC initial values
R_RSLV_Rdc_VariableInit((unsigned char*)s_u1_rdc_init_data);
Main loop
API call for executing the RDC initialization sequence
R_RSLV_Rdc_Init_Sequence(&u2_rdc_result);
API call for handling RDC communications
R_RSLV_Rdc_Communication();

Figure 7.6 Processing for Specifying the Initial Values of the RDC

After specifying the initial values of the RDC, call the API function for executing the RDC initialization sequence in the main loop. The initialization state can be acquired through the argument of this API function; continue the execution of this function until the sequence ends. As this API function executes RDC communications, the API function for handling RDC communications should also be called in the main loop.

7.2.3 Sample Code

7.2.3.1 Initialization of the MCU (Initialization of the Peripheral Modules)

The following shows an example of code for initializing the peripheral modules. This example uses the initialization function R_Systeminit generated by the SC. When not using R_Systeminit, refer to the example of migration in section 8.2.1, Initialization Processing of Peripheral Modules.

```
/******************************************************************************
* Function Name: R_Systeminit
* Description  : This function initializes every configuration.
* Arguments    : None
* Return Value : None
******************************************************************************/
void R_Systeminit(void)
{
    /* Enable writing to registers related to operating modes, LPC, CGC, and
software reset */
    SYSTEM.PRCR.WORD = 0xA50BU;

    /* Enable writing to MPC pin function control registers. */
    MPC.PWPR.BIT.B0WI = 0U;
    MPC.PWPR.BIT.PFSWE = 1U;

    /* Write 0 to the target bits in the POECR2 register. */
    POE.POECR2.WORD = 0x0000U;

    /* Initialize clock settings. */
    R_CGC_Create();

    /* Make peripheral module settings. */
    R_Config_RSPI0_RdcCom_Create();
    R_Config_TMR0_PhaseA_Create();
    R_Config_TMR3_RdcClk_Create();
    R_Config_TMR4_PhaseB_Create();
    R_Config_MTU2_Cap_Create();
    R_Config_CMT1_CsigUpdTim_Create();
    R_Config_MTU0_Csig_Create();
    R_Config_MTU9_Esig_Create();

    /* Make interrupt settings. */
    R_Interrupt_Create();

    /* Register undefined interrupt. */
    R_BSP_InterruptWrite(BSP_INT_SRC_UNDEFINED_INTERRUPT,(bsp_int_cb_t)
    r_undefined_exception);

    /* Register group BL0 interrupt TEI5 (SCI5). */
    R_BSP_InterruptWrite(BSP_INT_SRC_BL0_SCI5_TEI5,(bsp_int_cb_t)
    r_Config_SCI5_transmitend_interrupt);

    /* Register group BL0 interrupt ERI5 (SCI5). */
    R_BSP_InterruptWrite(BSP_INT_SRC_BL0_SCI5_ERI5,(bsp_int_cb_t)
    r_Config_SCI5_receiveerror_interrupt);

    /* Register group BL0 interrupt TEI12 (SCI12). */
    R_BSP_InterruptWrite(BSP_INT_SRC_BL0_SCI12_TEI12,(bsp_int_cb_t)
    r_Config_SCI12_transmitend_interrupt);
    /* Register group BL0 interrupt ERI12 (SCI12). */
    R_BSP_InterruptWrite(BSP_INT_SRC_BL0_SCI12_ERI12,(bsp_int_cb_t)
    r_Config_SCI12_receiveerror_interrupt);

    /* Register group BL1 interrupt OEI2 (POE3). */
    R_BSP_InterruptWrite(BSP_INT_SRC_BL1_POE3_OEI2,(bsp_int_cb_t)
    r_Config_POE_oei2_interrupt);

    /* Register group BL1 interrupt OEI3 (POE3). */
    R_BSP_InterruptWrite(BSP_INT_SRC_BL1_POE3_OEI3,(bsp_int_cb_t)
    r_Config_POE_oei3_interrupt);

    /* Register group AL0 interrupt SPII0 (RSPI0). */
    R_BSP_InterruptWrite(BSP_INT_SRC_AL0_RSPI0_SPII0,(bsp_int_cb_t)
    r_Config_RSPI0_idle_interrupt);

    /* Register group AL0 interrupt SPEI0 (RSPI0). */
    R_BSP_InterruptWrite(BSP_INT_SRC_AL0_RSPI0_SPEI0,(bsp_int_cb_t)
    r_Config_RSPI0_error_interrupt);

    /* Disable writing to MPC pin function control registers. */
    MPC.PWPR.BIT.PFSWE = 0U;
    MPC.PWPR.BIT.B0WI = 1U;

    /* Enable protection. */
    SYSTEM.PRCR.WORD = 0xA500U;
}
```

7.2.3.2 Initialization of the Driver

The following shows an example of code for initializing the driver. Call this processing from the main loop. For the example of the code (main loop) that calls this processing, see section 7.3.2, Sample Code.

```
/*******************************************************************
* Function Name : R_RSLVADP_Init
* Description   : Resolver related processing initialization
* Arguments     : None
* Return Value  : None
********************************************************************/
void R_RSLVADP_Init (void)
{

   /* Setting of function for resolver */
   RESOLVER_init_func();    // Specify the system information and function
tables.

   /////////////////////////////////////////
   /// RDC initial value settings
   /////////////////////////////////////////
   /* Initializes RDC register values. */
   R_RSLV_Rdc_VariableInit((unsigned char*)s_u1_rdc_init_data);

   /* Get resolver settings.  */
   R_RSLV_GetRdcDrvSettingInfo(&st_drv_info);

}
```

The following shows an example of code for specifying the system information and function tables. Call this processing from the main loop. This example calls the processing from the R_RSLVADP_Init() function shown above.

```
/*******************************************************************
* Function Name : RESOLVER_init_func
* Description   : Resolver driver system initialization
* Arguments     : None
* Return Value  : None
*******************************************************************/
static void RESOLVER_init_func(void)
{
   ST_SYSTEM_PARAM    st_system_param;
   ST_USER_PERI_PARAM st_user_peri_param;

   /* Initialize GPIO to output a low level as the reset signal
and place the RDC in the reset state. */
   /* STM board uses P43 as RDC reset pin */
   PORT4.PODR.BIT.B3   = 0;
   PORT4.PDR.BIT.B3    = 1;

   ////////////////////////////////
   /// System information settings
   ////////////////////////////////
   /* Excitation signal (ESig) frequency 20 kHz */
   st_system_param.u1_esig_freq = R_ESIG_SET_FREQ_20K;
   /* Correction signal (CSig) frequency 200 kHz */
   st_system_param.u1_csig_freq = R_CSIG_SET_FREQ_200K;
   /* Update the duty cycle 2 times. */
   st_system_param.u1_csig_upd_duty_cycle = R_CSIG_SET_DCNT_02;
   /* Use MTU synchronous start. */
   st_system_param.u1_sync_start = SYNCMD_OTHER_API;
   /* Target motor is a STM motor. */
   st_system_param.u1_motor_kind = MOTOR_STM;
   /* RDC IC MNTOUT output method */
   st_system_param.u1_mntout_type = RSLV_MNTOUT_TYPE_AC;


   st_user_peri_param.f_esig1_peri_clk_src = 80.0f;
   st_user_peri_param.f_csig_peri_clk_src = 80.0f;
   st_user_peri_param.f_csig_upd_timer_peri_clk_src = 80.0f;
   st_user_peri_param.f_capture_peri_clk_src = 80.0f;
   st_user_peri_param.f_phase1_peri_clk_src = 40.0f;
   st_user_peri_param.f_phase2_peri_clk_src = 40.0f;

   R_RSLV_SetSystemInfo(&st_system_param, &st_user_peri_param);

   ///////////////////////////////////////////////////////////////////////
   /// Settings of timer start timing for excitation signal output and angle
       signal input
   ///////////////////////////////////////////////////////////////////////
   /* Esig & Capture start timing*/
   R_RSLV_ESigCapStartTiming(DEF_DELAY_ADJ_ESIG, DEF_SFT_ADJ_ESIG);

   ////////////////////////////////////////////////////
   /// Function table settings (excitation signal output)
   ////////////////////////////////////////////////////
```

```
/* Set up the function table for ESig. */
g_st_user_func_table.Start   = &R_Config_MTU9_Esig12_Start;
g_st_user_func_table.Stop    = &R_Config_MTU9_Esig12_Stop;
g_st_user_func_table.SetTcnt = &R_Config_MTU9_Esig12_SetTcnt;
g_st_user_func_table.GetTcnt = &R_Config_MTU9_Esig12_GetTcnt;
R_RSLV_SetFuncTable(F_ESIG12, g_st_user_func_table);

//////////////////////////////////////////////////////////////////
/// Function table settings (angle error correction signal output)
//////////////////////////////////////////////////////////////////
/* Set up the function table for CSig. */
g_st_user_func_table.Start   = &R_Config_MTU0_Csig_Start;
g_st_user_func_table.Stop    = &R_Config_MTU0_Csig_Stop;
g_st_user_func_table.SetTcnt = &R_Config_MTU0_Csig_SetTcnt;
g_st_user_func_table.GetTcnt = &R_Config_MTU0_Csig_GetTcnt;
g_st_user_func_table.SetDuty = &R_Config_MTU0_Csig_SetDuty;
g_st_user_func_table.GetDuty = &R_Config_MTU0_Csig_GetDuty;
R_RSLV_SetFuncTable(F_CSIG, g_st_user_func_table);

/////////////////////////////////////////////////
/// Function table settings (angle signal input)
/////////////////////////////////////////////////
/* Set up the function table for Capture. */
g_st_user_func_table.Start   = &R_Config_MTU2_Cap_Start;
g_st_user_func_table.Stop    = &R_Config_MTU2_Cap_Stop;
g_st_user_func_table.SetTcnt = &R_Config_MTU2_SetTcnt;
g_st_user_func_table.GetTcnt = &R_Config_MTU2_GetTcnt;
g_st_user_func_table.GetCaptureValue = &R_Config_MTU2_GetCapVal;
g_st_user_func_table.GetPortLevel = &R_Config_MTU2_GetPortLvl;
R_RSLV_SetFuncTable(F_CAPTURE, g_st_user_func_table);

/////////////////////////////////////////
/// Function table settings (RDC clock)
/////////////////////////////////////////
/* Set up the function table for RDC IC clock. */
g_st_user_func_table.Start   = &R_Config_TMR3_RdcClk_Start;
g_st_user_func_table.Stop    = &R_Config_TMR3_RdcClk_Stop;
R_RSLV_SetFuncTable(F_RDC_CLK, g_st_user_func_table);

//////////////////////////////////////////////////////////////
/// Function table settings (phase adjustment signal output A)
//////////////////////////////////////////////////////////////
/* Set up the function table for phase A/B. */
g_st_user_func_table.Start   = &R_Config_TMR0_PhaseA_Start;
g_st_user_func_table.Stop    = &R_Config_TMR0_PhaseA_Stop;
g_st_user_func_table.SetDuty = &R_Config_TMR0_PhaseA_SetDuty;
R_RSLV_SetFuncTable(F_PHASE_A, g_st_user_func_table);

//////////////////////////////////////////////////////////////
/// Function table settings (phase adjustment signal output B)
//////////////////////////////////////////////////////////////
/* Set up the function table for phase A/B. */
g_st_user_func_table.Start   = &R_Config_TMR4_PhaseB_Start;
g_st_user_func_table.Stop    = &R_Config_TMR4_PhaseB_Stop;
g_st_user_func_table.SetDuty = &R_Config_TMR4_PhaseB_SetDuty;
R_RSLV_SetFuncTable(F_PHASE_B, g_st_user_func_table);
```

```
    ////////////////////////////////////////////////
    /// Function table settings (Communications with RDC)
    ////////////////////////////////////////////////
    /* Set up the function table for phase B. */
    g_st_user_func_table.ComSendReceive =
    & R_Config_RSPI0_RdcCom_Send_Receive;
    R_RSLV_SetFuncTable(F_RDC_COM, g_st_user_func_table);

}
```

The following shows an example of code for specifying the pointer to the user-created callback function for automatic calibration. Call this processing from the main loop.

For the example of the code (main loop) that calls this processing, see section 7.3.2, Sample Code.

```
/***************************************************************************
* Function Name: r_mtr_init_adjst_interface
* Description  : Initialize interface functions and variables with library
* Arguments    : void
* Return Value : void
***************************************************************************/
void r_mtr_init_adjst_interface( void )
{
    st_ptr_func_arg_t    temp_arg;

    temp_arg.ad_data = R_S12AD_GetMntOut;
    temp_arg.ad_ctrl = R_S12AD_StartByAdjst;
    temp_arg.ad_peri_adjst = R_S12AD_ChgSettingForAdjst;
    temp_arg.ad_peri_user  = R_S12AD_ResetSettigForNormal;
    temp_arg.resolver_pole_num = DEF_RESOLV_POLE_PAIR;
    temp_arg.mtr_speed = &(mtr_p[0]->spd_ctrl.f_speed);
    temp_arg.req_speed = com_f_spd_ref;

    R_RSLV_ADJST_SetPtrFunc( &temp_arg );
}
```

The following shows an example of code for the RDC initialization sequence. Call this processing from the main loop.

```
/*******************************************************************************
* Function Name : R_RSLVADP_MainLoopProcess
* Description   : Resolver management process for main loop
* Arguments     : None
* Return Value  : None
*******************************************************************************/
void R_RSLVADP_MainLoopProcess(void)
{
    uint16_t rdc_result = RSLV_MD_BUSY1;

    resolver_csig_ui();

    if (TRUE == com_u1_flg_rdc_sequence)
    {
         g_u1_flg_rdc_state_ready = FALSE;

        if(RDC_RESET_STATE_NON == g_u1_rdc_reset_wait_status)
        {
            write_rdc_reset_gpio(1);
            g_u1_rdc_reset_wait_status = RDC_RESET_STATE_ACT;
            g_u2_rdc_reset_wait_count = 0;
        }
        else if(RDC_RESET_STATE_ACT == g_u1_rdc_reset_wait_status)
        {
            if(PRV_RDC_SPI_WAIT < g_u2_rdc_reset_wait_count)   /* Wait 1000 count
* 50[us] = 50[ms] */
            {
                g_u1_rdc_reset_wait_status = RDC_RESET_STATE_FIN;
            }
        }
        else if(RDC_RESET_STATE_FIN == g_u1_rdc_reset_wait_status)
        {
            R_RSLV_Rdc_Init_Sequence(&rdc_result);
            if (RSLV_MD_OK == rdc_result)
            {
                 com_u1_flg_rdc_sequence = FALSE;
                 g_u1_flg_rdc_state_ready = TRUE;
                 /* Start of IRQ5 */
                 R_ICU_Start_irq5();
            }
        }
        else
        {
            ;
        }
    }

    /* RDC SPI main function */
    R_RSLV_Rdc_Communication();

    /* Setting PWM duty of MTU3 channel 7 */
    R_RSLV_Phase_AdjUpdate();
}
```

7.3 Main Loop

7.3.1 Example of Implementation

Figure 7.7 shows an example of implementing the main loop.

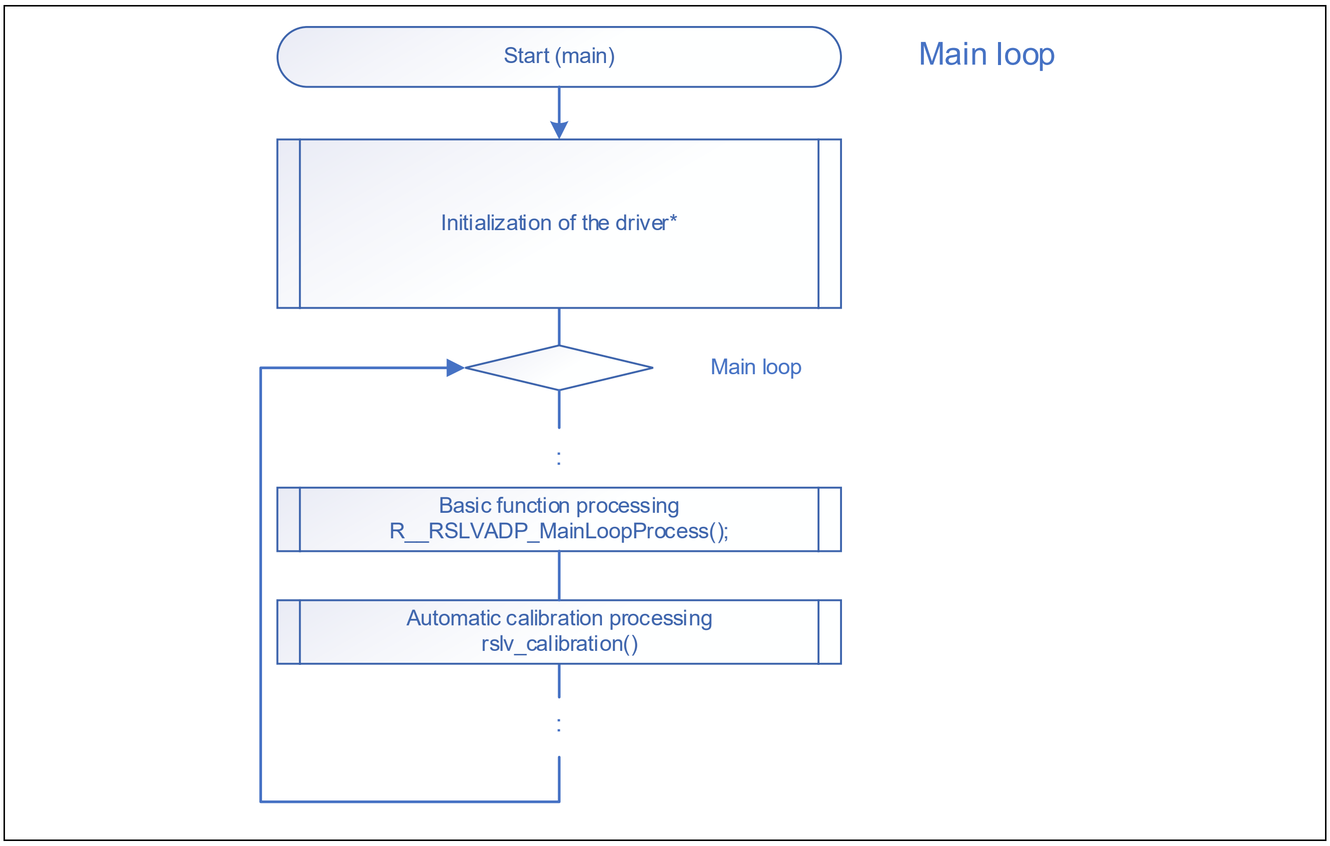

Main loop
Start (main)
Initialization of the driver*
Main loop
:
Basic function processing
R__RSLVADP_MainLoopProcess();
Automatic calibration processing
rslv_calibration()
:

Figure 7.7 Example of Implementing the Main Loop

In the main loop, call the processing for communications with the RDC and the processing for updating the duty cycles of the phase adjustment signals periodically. Furthermore, it is recommended that the processing for detecting disconnection described in section 7.11, Detection of Disconnection from Resolver Sensor, be also implemented. This sample code makes initial settings and updates the duty cycles of the phase adjustment signals in the basic function processing. It also performs automatic adjustment of the gain and phase of the resolver signals and automatic adjustment of the angle error correction signal in the automatic calibration processing.

Note: * For initialization of the driver, see section 3.1, Initialization of the Driver.

7.3.2 Sample Code

The following shows an example of the main function (main loop) code.

```
/******************************************************************************
* Function Name : main
* Description   : Initialization and main routine
* Arguments     : None
* Return Value  : None
******************************************************************************/
void main(void)
{
    float f4_temp;
clrpsw_i();                                 /* Interrupt disabled */
    /* Initialize peripheral functions */
  // Initialize the MCU. See section 7.2.3.1, Initialization of the MCU
      (Initialization of the Peripheral Modules).
    R_MTR_InitHardware()
    // Initialize the driver. See section 7.2.3.2, Initialization of the
       Driver.
    R_RSLVADP_Init();

    /* Initialize ICS. */
    ics2_init((void*)dtc_table, ICS_SCI1_PD3_PD5, ICS_INT_LEVEL, ICS_BRR,
    ICS_INT_MODE);

    /* Start of A/D converter */
    R_MTR_Start_s12ad();

    /* Start of CMT0 */
    R_MTR_Start_cmt0();

    /* Initialize private global variables. */
    variables_init();

    /* Execute reset event. */
    R_MTR_SR_Foc_ExecEvent( MTR_ID_A,MTR_EVENT_RESET);

setpsw_i();                                 /* Interrupt enabled */

    /* Start peripheral modules related to the resolver. The following must be
       called after enabling interrupts. */
  // Start the output of the excitation signal. See section 7.4.2, Sample Code.
    R_RSLVADP_Start();
    // Specify the pointer to the user-created callback function. See section
       7.2.3.2, Initialization of the Driver.
    mtr_init_adjst_interface();

    /*** Main routine ***/
    while (1)
    {
        /* User interface */
        ui_main();

        R_MTR_SR_Foc_GetSpeed(MTR_ID_A, &f4_temp, &g_f4_adjst_rslv_speed_rad);
        // Basic function processing: Communications with RDC (RDC initial
           settings) and updating of the duty cycles of the phase adjustment
           signals
        R_RSLVADP_MainLoopProcess();
        // Automatic calibration: adjustment of the gain and phase and
           adjustment of the angle error correction signal
        rslv_calibration();
        /* Clear watch dog timer. */
        R_MTR_ClearWdt();
    }
} /* End of function main */
```

7.4 Output of the Excitation Signal

7.4.1 Example of Using API Functions

Figure 7.8 shows a block diagram of implementation by using API functions related to the output of the excitation signal.

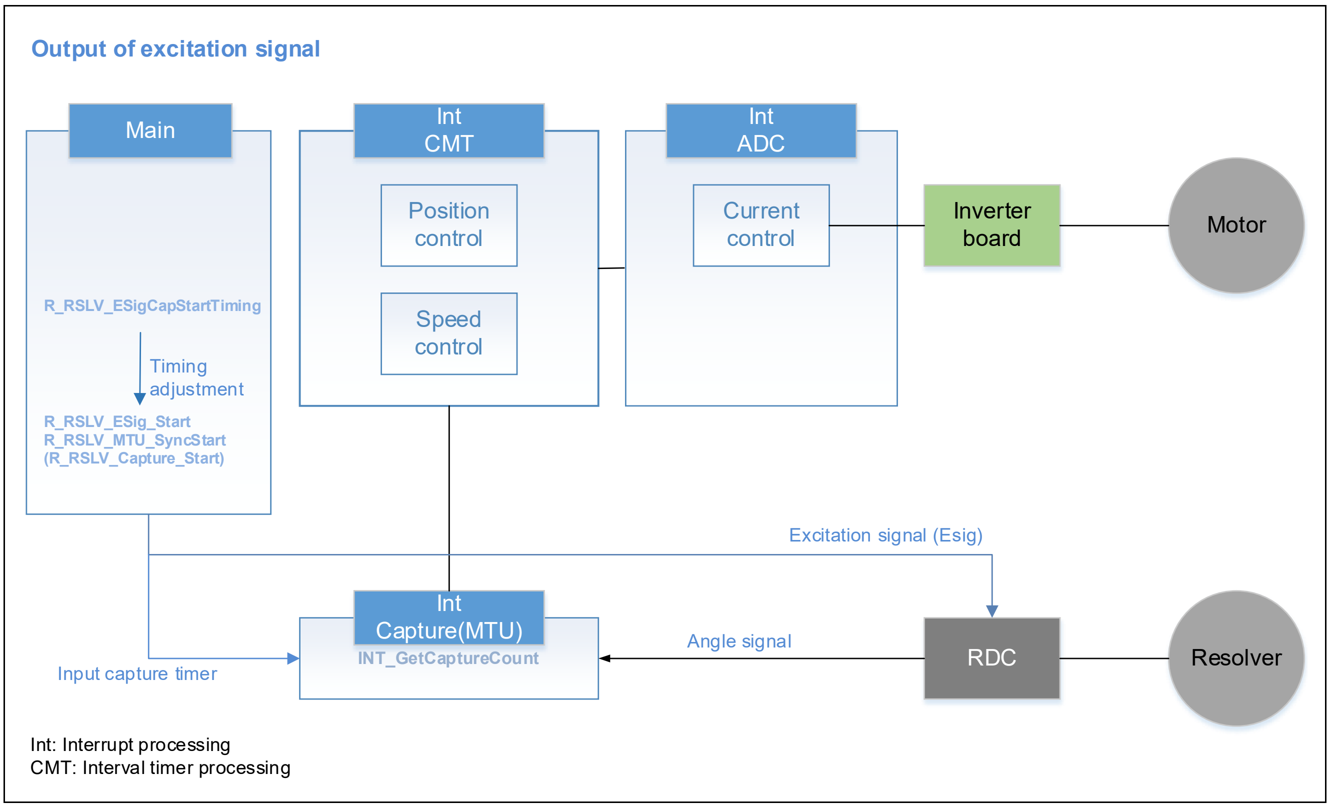


Figure 7.8 Example of Implementation by Using API Functions Related to the Output of the Excitation Signal

To start the output of the excitation signal, use the R_RSLV_ESig_Start function (section 6.2.17, API Function for Starting the Output of the Excitation Signal).

7.4.2 Sample Code

The following shows sample code.

In this example, the timers for outputting the excitation signal and detecting the angle signal are started simultaneously.

```
/*****************************************************************************
* Function Name : R_RSLVADP_Start
* Description   : Resolver start processing
* Arguments     : None
* Return Value  : None
*****************************************************************************/
void R_RSLVADP_Start(void)
{
    /* Initialize resolver settings */
    R_RSLV_ESig_Start();
    R_RSLV_MTU_SyncStart(MTU_TCSYSTR_BIT_MTU9 | MTU_TCSYSTR_BIT_MTU2);

    /* Output the angle error correction signal (current default is "TRUE"). */
    if (TRUE == com_u1_flg_csig)
    {
        R_RSLV_CSig_Start(com_u2_csig_shiftnum, com_u2_csig_amplvl);
    }
    else
    {
        R_RSLV_CSig_Stop();
    }
    g_u1_flg_pre_csig = com_u1_flg_csig;
}
```

7.5 Output of the Phase Adjustment Signals

7.5.1 Example of Using API Functions

Figure 7.9 shows a block diagram of implementation by using API functions for outputting the phase adjustment signals.

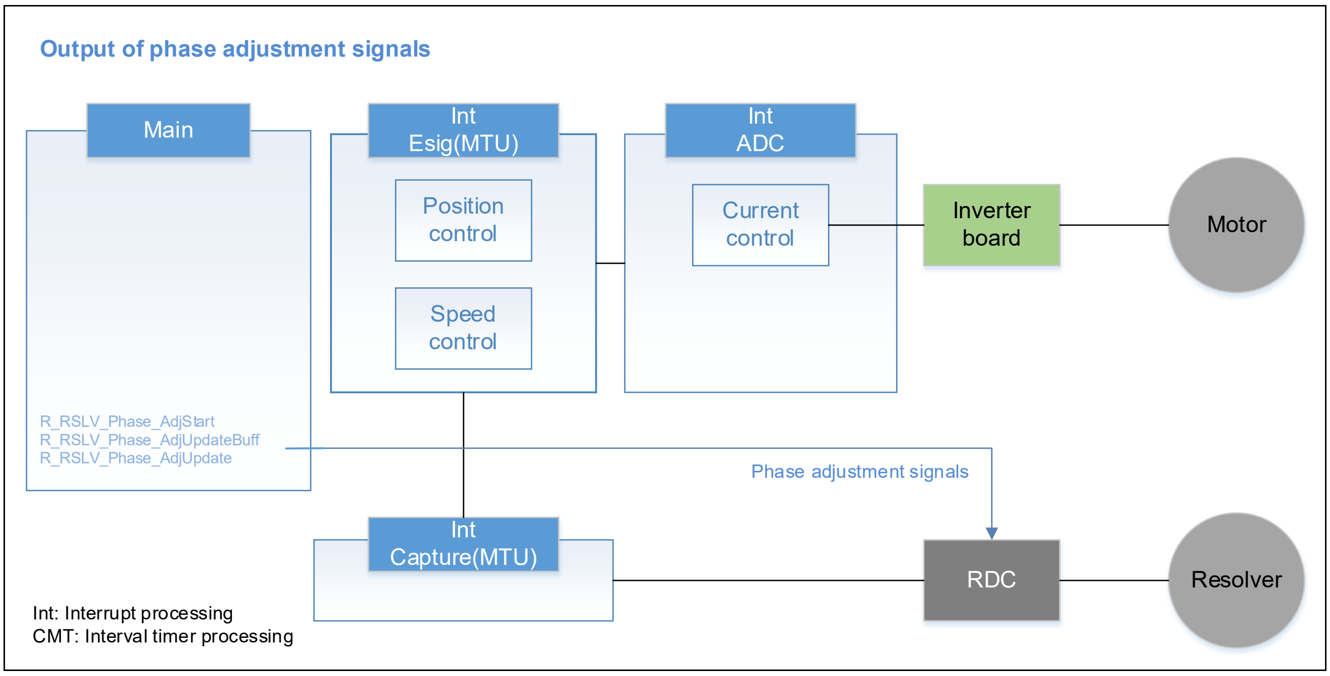


Figure 7.9 Example of Implementation by Using API Functions for Outputting the Phase Adjustment Signals

To output the phase adjustment signals, use the API functions R_RSLV_Phase_AdjStart (section 6.2.21, API Function for Starting the Output of the Phase Adjustment Signals), R_RSLV_Phase_AdjUpdateBuff (section 6.2.23, API Function for Setting the Phase Adjustment Signal Duty Cycle in the Buffer), and R_RSLV_Phase_AdjUpdate (section 6.2.24, API Function for Setting the Phase Adjustment Signal Duty Cycle in the Register).

After updating the duty cycle information in the driver by the R_RSLV_Phase_AdjUpdateBuff function, execute the R_RSLV_Phase_AdjUpdate function to reflect the information in the duty output register. Then, call the R_RSLV_Phase_AdjStart function to start the output of PWM signals.

7.5.2 Sample Code

The following shows sample code.

7.5.2.1 Outputting the Phase Adjustment Signals

The following shows an example of implementing the output of the phase A signal with 65% duty cycle and the phase B signal with 22% duty cycle in the main loop.

```
unsigned char u1_flg_phase_started = 0U;   /* Phase adjustment signal start
flag */

void main(void)
{
    /* Initialization */

    /* Main loop */
    while (1)
    {
        /* Communications with RDC */

        /* Phase adjustment signal processing */
        if (0U == u1_flg_phase_started)
        {
            R_RSLV_Phase_AdjUpdateBuff(65, PHASE_CH_A);
            R_RSLV_Phase_AdjUpdateBuff(22, PHASE_CH_B);
        }

        R_RSLV_Phase_AdjUpdate();  /* Call R_RSLV_Phase_AdjUpdate
        periodically.*/

        if (0U == u1_flg_phase_started)
        {
            R_RSLV_Phase_AdjStart();
            u1_flg_phase_started = 1U;
        }
    }
}
```

7.6 Output of the Angle Error Correction Signal

7.6.1 Example of Using API Functions

Figure 7.10 shows a block diagram of implementation by using API functions for outputting the angle error correction signal.

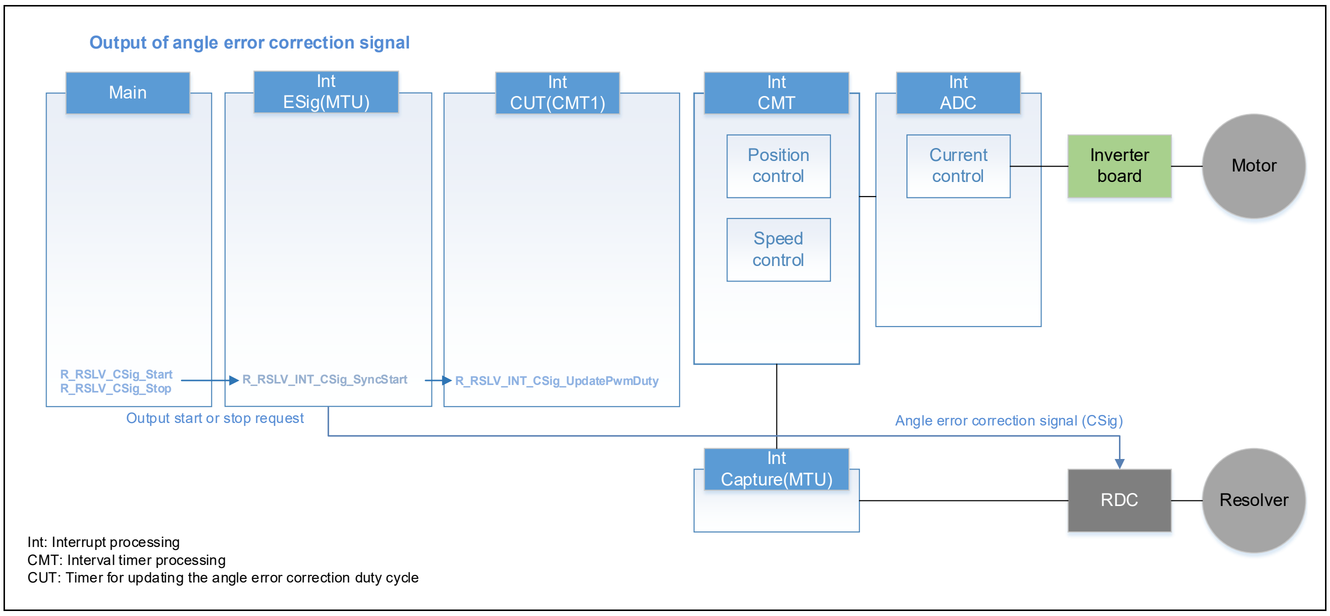

Output of angle error correction signal
Main
Int ESig(MTU)
Int CUT(CMT1)
Int CMT
Position control
Speed control
Int ADC
Current control
Inverter board
Motor
R_RSLV_CSig_Start
R_RSLV_CSig_Stop
R_RSLV_INT_CSig_SyncStart
R_RSLV_INT_CSig_UpdatePwmDuty
Output start or stop request
Angle error correction signal (CSig)
Int Capture(MTU)
RDC
Resolver
Int: Interrupt processing
CMT: Interval timer processing
CUT: Timer for updating the angle error correction duty cycle

Figure 7.10 Example of Implementation by Using API Functions for Outputting the Angle Error Correction Signal

To output the angle error correction signal, use the API functions R_RSLV_CSig_Start (section 6.2.6, API Function for Starting the Output of the Angle Error Correction Signal), R_RSLV_INT_CSig_SyncStart (section 6.2.9, API Function for Synchronously Starting the Angle Error Correction Signal), and R_RSLV_INT_CSig_UpdatePwmDuty (section 6.2.8, API Function for Updating the Duty Cycle of the Angle Error Correction Signal).

7.6.2 Sample Code

The following shows sample code.

7.6.2.1 Starting and Stopping the Output of the Angle Error Correction Signal

Call this processing from the main loop.

```
/******************************************************************************
* Function Name : R_RSLVADP_Start
* Description   : Resolver start processing
* Arguments     : None
* Return Value  : None
******************************************************************************/
void R_RSLVADP_Start(void)
{
    /* Initialize resolver settings. */
    R_RSLV_ESig_Start();
    R_RSLV_MTU_SyncStart(MTU_TCSYSTR_BIT_MTU9 | MTU_TCSYSTR_BIT_MTU2);

    /* Output the angle error correction signal (current default is "TRUE"). */
    if (TRUE == com_u1_flg_csig)
    {
        R_RSLV_CSig_Start(com_u2_csig_shiftnum, com_u2_csig_amplvl);
    }
    else
    {
        R_RSLV_CSig_Stop();
    }
    g_u1_flg_pre_csig = com_u1_flg_csig;
}
```

7.6.2.2 Interrupt Processing for Updating the PWM Duty Cycle

Call the following API function from the timer interrupt processing for updating the duty cycle of the angle error correction signal.

```
#pragma interrupt (mtr_csig_interrupt(vect = VECT_RSLV_CSIG))
static void mtr_csig_interrupt(void)
{
setpsw_i();                                  /* Interrupt enabled */
    R_RSLV_INT_CSig_UpdatePwmDuty();
} /* End of function mtr_csig_interrupt */
```

7.6.2.3 Synchronous Start of the Angle Error Correction Signal

Call the following API function from the excitation interrupt processing.

```
#pragma interrupt (rslv_esig_interrupt(vect = VECT_RSLV_ESIG))
static void rslv_esig_interrupt(void)
{
setpsw_i();

    if(mtu9_interrupt_decimation_flag == 0)
    {
        R_RSLV_INT_CSig_SyncStart();
        mtu9_interrupt_decimation_flag ++;
        R_RSLV_INT_ESigCounter();
    }
    else
    {
        mtu9_interrupt_decimation_flag = 0;
    }

} /* End of function rslv_esig_interrupt */
```

7.7 Input of Angle Signal

7.7.1 Example of Using API Functions

Figure 7.11 shows a block diagram of implementation by using API functions for inputting the angle signal.

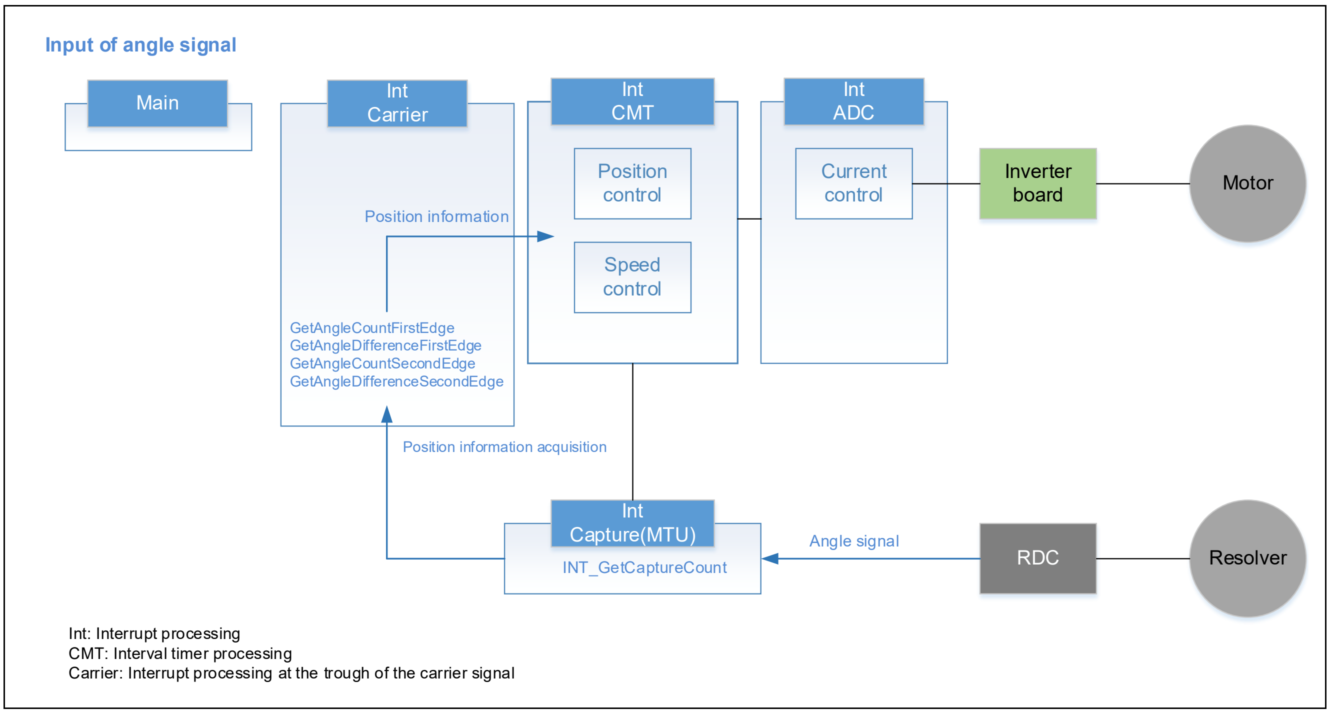


Figure 7.11 Example of Implementation by Using API Functions for Inputting the Angle Signal

Use the FirstEdge API functions to acquire the counter value and counter difference information on the falling edge of the angle signal. Use the SecondEdge API functions to acquire the values on the rising edge of the angle signal.

7.7.2 Sample Code

The following shows sample code.

7.7.2.1 Processing of Angle Signal Interrupt

Call the following API function from the input capture interrupt processing.

```
#pragma interrupt (rslv_capture_interrupt(vect = VECT_RSLV_CAPTURE))
static void rslv_capture_interrupt(void)
{
    R_RSLV_INT_GetCaptureCount();
} /* End of function rslv_capture_interrupt */
```

7.7.2.2 Acquiring the Position Information

The following shows an example of information acquisition in the interrupt processing at the trough of the motor-driving PWM timer counter.

```
#pragma interrupt mtr_mtu3_tciv4_interrupt(vect=VECT(MTU4,TCIV4))
static void mtr_mtu3_tciv4_interrupt( void )
{
    uint16_t u2_angle_cnt;
    int16_t  s2_angle_diff;

// Acquire the counter value at both the first and second edges.
//   uint16_t s2_AngleDiffHi;
//   uint16_t s2_AngleDiffLo;

setpsw_i();                                   /* Interrupt enabled *

    R_RSLV_GetAngleCountFirstEdge(&u2_angle_cnt);
    R_RSLV_GetAngleDifferenceFirstEdge(&s2_angle_diff);
    R_MTR_SR_Foc_SetAngleInfo(MTR_ID_A, u2_angle_cnt, s2_angle_diff);

// The following is an example of processing added to acquire the counter
value at both the first and second edges.
//   /* Get angle count value of resolver. */
//   if(RSLV_HIGH == R_RSLV_GetCaptureEdge())
//   {
//       R_RSLV_GetAngleCountFirstEdge(&g_st_foc.u2_rslv_angle_cnt);
//   }
//   else
//   {
//       R_RSLV_GetAngleCountSecondEdge(&g_st_foc.u2_rslv_angle_cnt);
//   }
//   R_RSLV_GetAngleDifferenceFirstEdge(&s2_AngleDiffHi);
//   R_RSLV_GetAngleDifferenceSecondEdge(&s2_AngleDiffLo);
//   g_st_foc.s2_angle_err_cnt = u2_AngleDiffHi + u2_AngleDiffLo;
//   g_st_foc.s2_angle_err_cnt *= 0.5f;
//

    R_RSLVADP_IncreaseWaitTimer();


} /* End of function mtr_mtu3_tciv4_interrupt */
```

7.8 Automatic Adjustment of the Gain and Phase

7.8.1 Example of Using API Functions

Figure 7.12 shows a block diagram of implementation using the API functions for automatic adjustment of the gain and phase.

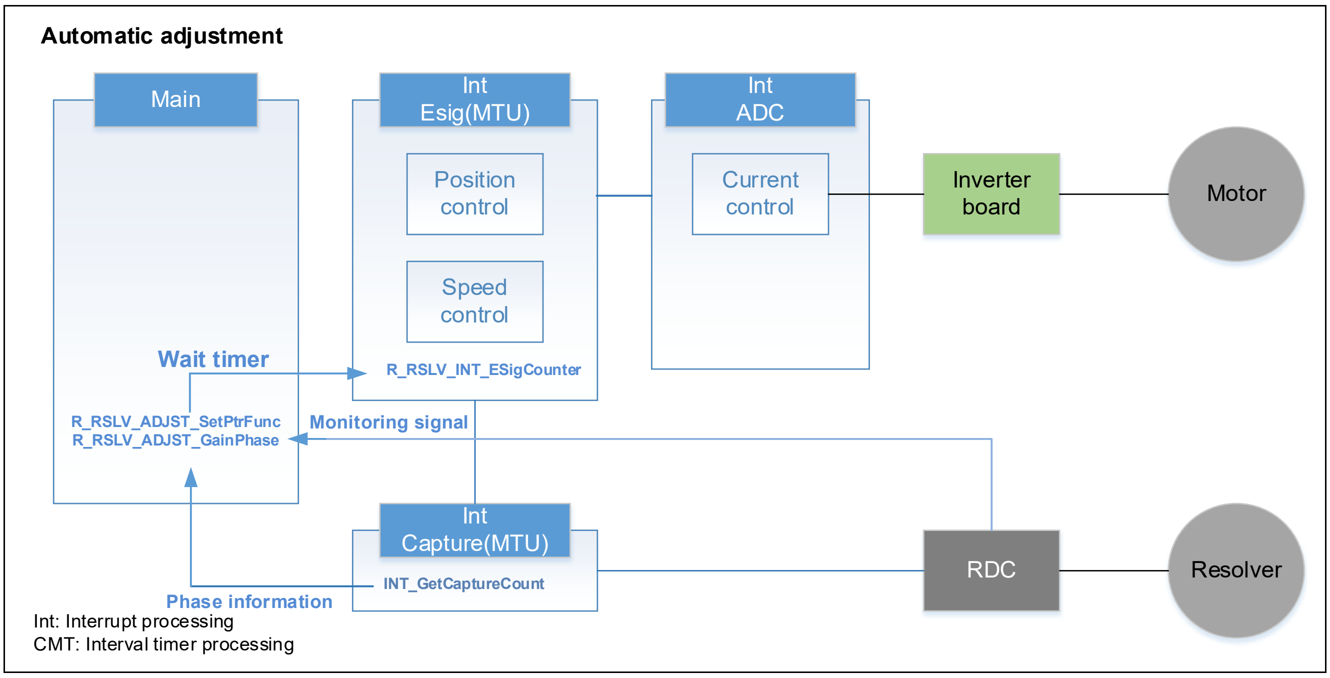


Figure 7.12 Example of Implementation by Using API Functions for Automatic Adjustment of the Gain and Phase

To execute the automatic adjustment of the gain and phase, use the API functions R_RSLV_ADJST_SetPtrFunc (section 6.2.41, API Function for Setting the Pointer to the User-Created Callback Function), R_RSLV_ADJST_GainPhase (section 6.2.39, API Function for Adjusting the Gain and Phase of the Resolver Signals), and R_RSLV_INT_ESigCounter (section 6.2.20, API Function for Counting the Wait Time).

R_RSLV_INT_GetCaptureCount (section 6.2.11, API Function for Acquiring the Angle Detection Value) is used to acquire phase information during phase adjustment. Call this function from the input capture interrupt processing.

R_RSLV_INT_ESigCounter is used as a wait timer in the adjustment processing. Call this function from the excitation signal interrupt processing.

7.8.2 Details of Gain and Phase Adjustment

Figure **7.13** shows an example of implementing adjustment of the gain and phase.

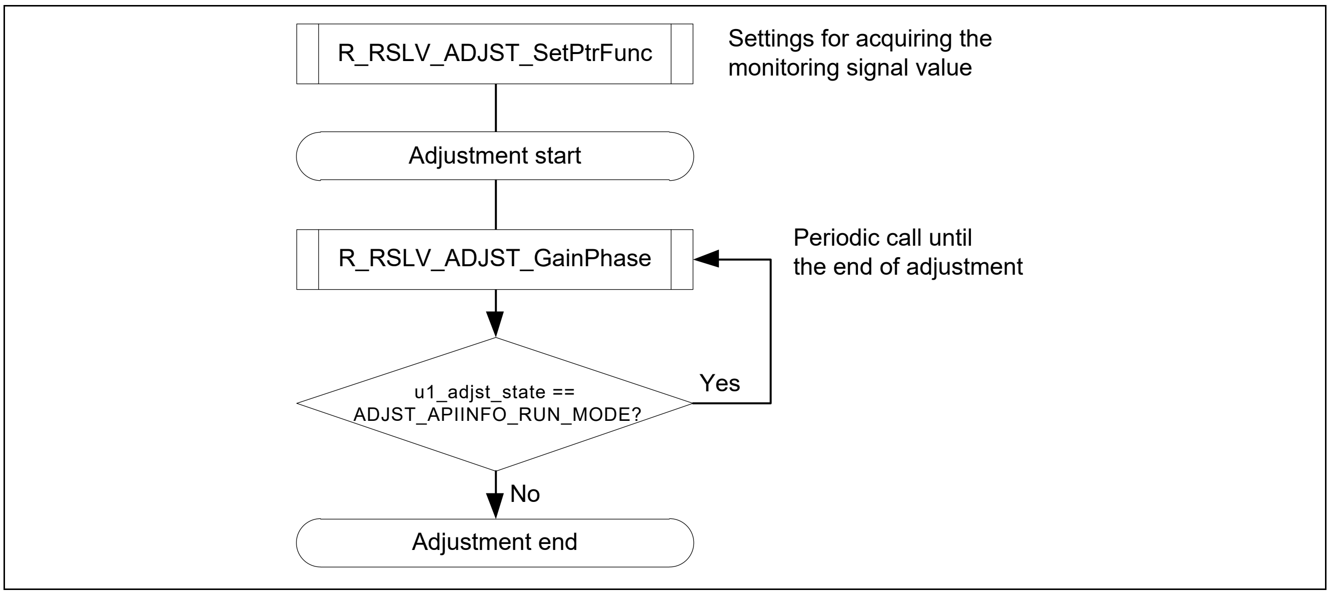

R_RSLV_ADJST_SetPtrFunc
Settings for acquiring the monitoring signal value
Adjustment start
R_RSLV_ADJST_GainPhase
Periodic call until the end of adjustment
u1_adjst_state == ADJST_APIINFO_RUN_MODE?
Yes
No
Adjustment end

Figure 7.13 Gain and Phase Adjustment Sequence

Adjustment of gain and phase uses the A/D converter to convert the monitoring signal output from the RDC. Therefore, it is necessary to use the API function for setting the callback function to specify the information on the A/D channel to which the monitoring signal is assigned for the driver. For details, see section 6.3.6, Structure for R_RSLV_ADJST_SetPtrFunc.

Repeatedly call the API function R_RSLV_ADJST_GainPhase for adjusting the gain and phase of the resolver signal until the adjustment is completed.

7.8.2.1 Starting Adjustment

To start adjustment, call R_RSLV_ADJST_GainPhase with ADJST_USRREQ_RUN (0) set as the argument of the API function.

7.8.2.2 Continuing Adjustment

The value of the member u1_adjst_state of the return value structure st_adjst_gainphase_return_t of R_RSLV_ADJST_GainPhase being ADJST_APIINFO_RUN_MODE (0U) indicates that adjustment remains in progress. As long as this is the case, repeatedly call R_RSLV_ADJST_GainPhase with ADJST_USRREQ_RUN (0) set as the argument of the API function.

To suspend the adjustment process, call the API function with ADJST_USRREQ_STOP (1) set as the argument.

Processing to return from the suspended state to the normal state is required, and this involves repeatedly calling R_RSLV_ADJST_GainPhase until the return value u1_adjst_state becomes ADJST_APIINFO_END_USER_STOP (13U).

7.8.2.3 Determining Completion of Adjustment

When u1_adjst_state is not ADJST_APIINFO_RUN_MODE (0U), adjustment is complete. Stop calling R_RSLV_ADJST_GainPhase.

The adjustment completion state indicator is stored in u1_adjst_state. In the case of normal end (ADJST_APIINFO_END_NORMAL (1U)), the result of adjustment is reflected in a member of the return value structure st_adjst_gainphase_return_t.

The required information is modified within the adjustment processing according to the result of adjustment, so there is no need to use API functions to re-make the settings and so on.

Table 7-1 lists the members of the return value structure st_adjst_gainphase_return_t. For details, see Table 6-7 Structure Definitions for R_RSLV_ADJST_Carrier.

Table 7-1 st_ptr_func_arg_t Structure Members

Member Name	Type	Description
u1_adjst_state	unsigned char	Gain and phase adjustment processing state and processing completion state
u1_res_dlcgsl	unsigned char	Adjustment result value for the RDC register DLCGSL (adjustment result value for the phase A gain)
u2_res_a_duty	unsigned short	Adjustment result duty value of the phase A adjustment signal
u2_res_b_duty	unsigned short	Adjustment result duty value of the phase B adjustment signal

7.8.3 Sample Code

The following shows sample code.

7.8.3.1 Call of API Function for Adjusting Gain and Phase

Repeatedly call the following processing from the main loop.

```
/***************************************************************************
* Function Name: mtr_rdc_AdjstGainPhaseProcess
* Description  : Process for adjustment of RDC gain & phase parameters
* Arguments    : req -
*                Request of sequence continuation (0:Continue, 1:Halt)
* Return Value : Active status of process (1:Active, 0:Finished)
***************************************************************************/
uint8_t mtr_rdc_AdjstGainPhaseProcess( uint8_t req )
{
    uint8_t result = TRUE;

    /* Call gain & phase adjustment API function. */
    gp_api_ret = R_RSLV_ADJST_GainPhase(req);

    /* Processing branches according to the return value. */
    /* While the processing is in progress, continuation of processing is
       reported. */
    switch (gp_api_ret.u1_adjst_state)
    {
        default:
        case ADJST_APIINFO_RUN_MODE:
            {
                result = TRUE;
            }
        break;

        case ADJST_APIINFO_END_NORMAL:
        case ADJST_APIINFO_ERR_GAIN_HI_LMT:
        case ADJST_APIINFO_ERR_GAIN_LO_LMT:
        case ADJST_APIINFO_ERR_GAIN_SWAY:
        case ADJST_APIINFO_ERR_PHASE_AHI_BLO:
        case ADJST_APIINFO_ERR_PHASE_ALO_BHI:
        case ADJST_APIINFO_ERR_PHASE_SWAY:
        case ADJST_APIINFO_ERR_MOTOR:
        case ADJST_APIINFO_END_USER_STOP:
            {
                result = FALSE;
            }
        break;
    }

    return (result);
}
```

7.9 Automatic Adjustment of the Angle Error Correction Signal

7.9.1 Example of Using API Functions

Figure 7.14 shows an example of implementation by using API functions for automatic adjustment of the angle error correction signal.

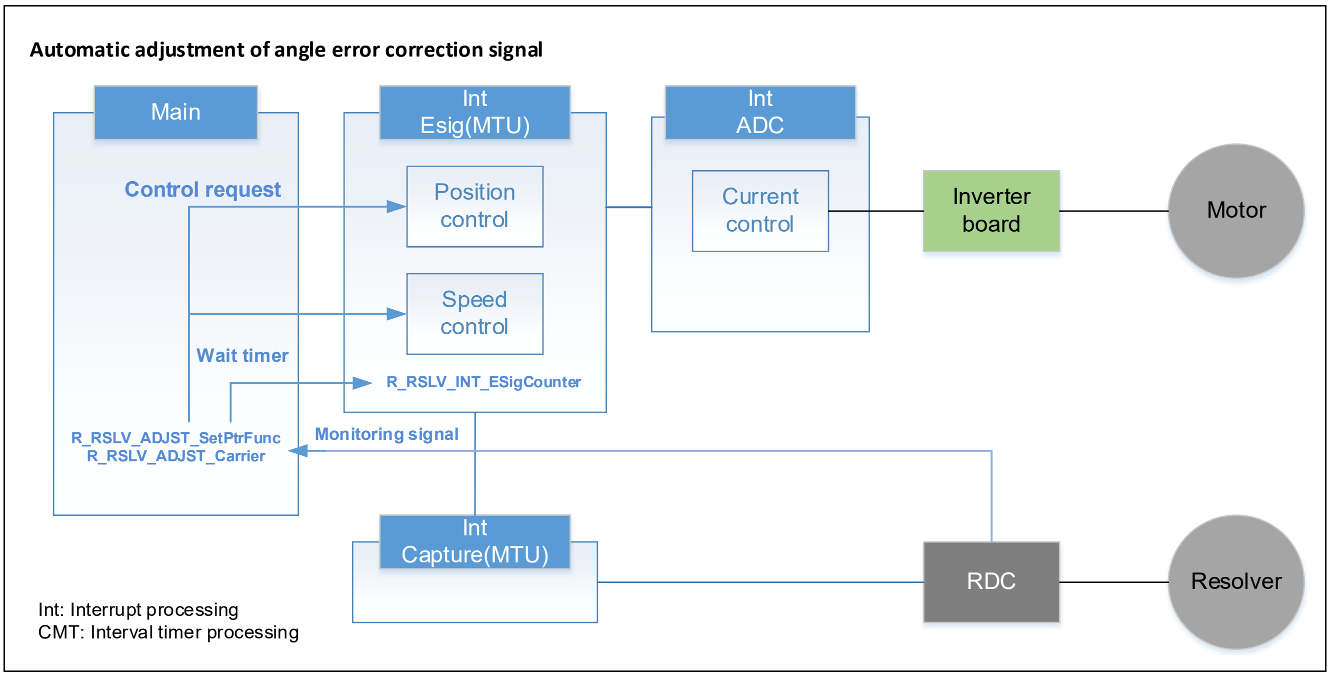


Figure 7.14 Example of Implementation by Using API Functions for Automatic Adjustment of the Angle Error Correction Signal

To automatically adjust the angle error correction signal, use the R_RSLV_ADJST_Carrier function (section 6.2.40, API Function for Adjusting the Angle Error Correction Signal).

The functionality of R_RSLV_INT_ESigCounter() is the same as that described in section 7.8, Automatic Adjustment of the Gain and Phase.

7.9.2 Details of Angle Error Correction Signal Adjustment

The motor must be controlled during adjustment of the angle error correction signal.

Figure 7.15 shows an example of implementing adjustment of the angle error correction signal.

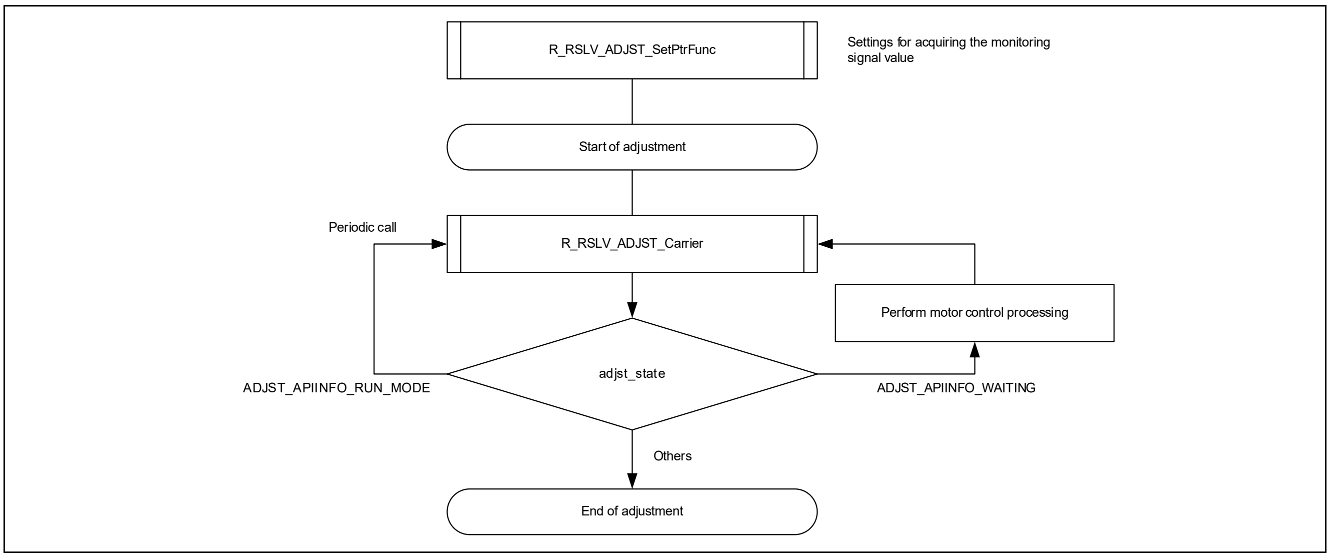


Figure 7.15 Angle Error Correction Signal Adjustment Sequence

The same processing is performed before the start of adjustment as that stated in section 7.8, Automatic Adjustment of the Gain and Phase. Processing after that depends on the adjst_state value. When the adjustment requires the application of motor control, the return value becomes ADJST_APIINFO_WAITING.

Figure 7.16 shows the sequence between the caller (application) and the driver from the start of adjustment until the completion of adjustment.

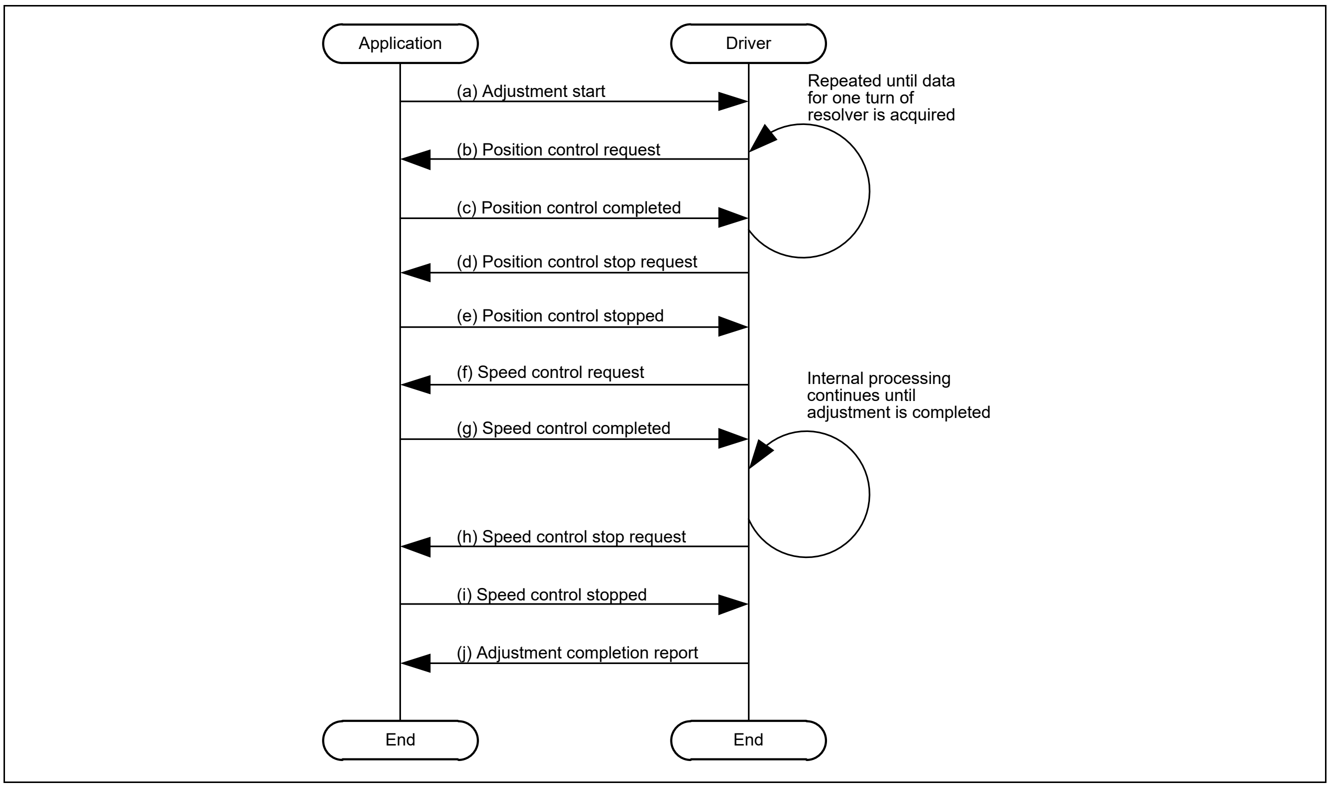

Application
Driver
(a) Adjustment start
Repeated until data for one turn of resolver is acquired
(b) Position control request
(c) Position control completed
(d) Position control stop request
(e) Position control stopped
(f) Speed control request
Internal processing continues until adjustment is completed
(g) Speed control completed
(h) Speed control stop request
(i) Speed control stopped
(j) Adjustment completion report
End
End

Figure 7.16 Angle Error Correction Signal Adjustment Sequence

The following describes processing steps (a) to (j) of the sequence.

(a) Adjustment Start

To start adjustment, call R_RSLV_ADJST_Carrier with ADJST_USRREQ_RUN (0) set as the member call_state of the structure argument st_adjst_carrier_arg_t for the API function. For details, see Table 6-8 Structure Definition for R_RSLV_ADJST_SetPtrFunc.

(b) Position Control Request

When adjustment starts, R_RSLV_ADJST_Carrier issues a position control request. This request is sent through members adjst_state and req_mtr_ctrl of the return value structure st_adjst_carrier_return_t of R_RSLV_ADJST_Carrier.

adjst_state = ADJST_APIINFO_WAITING (2)
req_mtr_ctrl = ADJST_APIREQ_POS_CTRL (1)
mtr_ctrl_data = 0 (beginning with a resolver angle of 0 degrees)

This adjustment processing requests the motor control settings as a return value as stated above, so start position control in accord with the control setting.

When calling R_RSLV_ADJST_Carrier again while making the motor control settings, set ADJST_USRINFO_PROCESSING (1) for the member req_state of the structure argument to notify the driver that the setting is in progress in the user application.

(c) Position Control Completed

When the position control (to the requested specified angle) has been completed according to the position control request, set ADJST_USRINFO_COMPLETE (0) for the member req_state of the structure argument.

After that, the driver starts acquisition of data. Upon completion of data acquisition, the driver requests position control again. At this time, the requested position information mtr_ctrl_data will have been updated. Apply position control again according to this position information. Repeat position control request and position control completion steps until the driver has completed acquisition of the required data. When data for one rotation of the resolver angle have been acquired, the processing proceeds to the step of position control stop request.

(d) Position Control Stop Request

When all data have been acquired, R_RSLV_ADJST_Carrier issues a position control stop request.

adjst_state = ADJST_APIINFO_WAITING (2)
req_mtr_ctrl = ADJST_APIREQ_POS_STOP (2)

When the return values of the API function have been updated as shown above, stop position control. When calling R_RSLV_ADJST_Carrier during position control stop processing, set ADJST_USRINFO_PROCESSING (1) as the member req_state of the structure argument in the same way as the step of position control request.

(e) Position Control Stopped

When the position control has been terminated, set ADJST_USRINFO_COMPLETE (0) as the member req_state of the structure argument. The processing proceeds to the step of speed control request.

(f) Speed Control Request

R_RSLV_ADJST_Carrier issues a speed control request.

adjst_state = ADJST_APIINFO_WAITING (2)
req_mtr_ctrl = ADJST_APIREQ_SPD_CTRL (3)
mtr_ctrl_data = 1000 rpm

When the return values of the API function have been updated as shown above, start speed control.

(g) Speed Control Completed

When the specified speed is reached, set ADJST_USRINFO_COMPLETE (0) for the member req_state of the structure argument for R_RSLV_ADJST_Carrier as an indicator of completion.

At the start of speed control, the adjustment processing involves manipulating the adjustment parameters of the angle error correction signal to make the adjustments. Call R_RSLV_ADJST_Carrier repeatedly until the adjustment processing is completed. Upon completion of the adjustment process, the processing proceeds to the step of speed control stop request.

(h) Speed Control Stop Request

After the adjustment has been completed, R_RSLV_ADJST_Carrier issues a request to stop speed control.

adjst_state = ADJST_APIINFO_WAITING (2)
req_mtr_ctrl = ADJST_APIREQ_SPD_STOP (4)

When the return values of the API function have been updated as shown above, stop the speed control.

(i) Speed Control Stopped

When the speed control has been stopped, set ADJST_USRINFO_COMPLETE (0) as the member req_state of the structure argument for R_RSLV_ADJST_Carrier. The processing proceeds to the step of adjustment completion report.

(j) Adjustment Completion Report

Upon completion of all processing for adjustment, completion of adjustment is reported by R_RSLV_ADJST_Carrier.

When adjst_state is not ADJST_APIINFO_RUN_MODE (0) or ADJST_APIINFO_WAITING (2), adjustment is complete.

For details of each return value, see Table 6-8 Structure Definition for R_RSLV_ADJST_SetPtrFunc.

When the return value is ADJST_APIINFO_END_NORMAL (1), the adjustment has been successfully completed and the adjusted values are returned as the members res_XXXX of the return value structure.

The required information is modified within the adjustment processing according to the result of adjustment, so there is no need to use API functions to re-make the settings and so on.

7.9.3 Sample Code

The following shows sample code.

7.9.3.1 Periodic Call Processing

Call the following processing from the main loop.

```
/***************************************************************************
* Function Name: r_mtr_rdc_AdjstCarrierProcess
* Description  : Process for adjustment of angle error correction signal
* Arguments    : req -
*                Request of sequence continuation (0:Continue, 1:Halt)
* Return Value : Active status of process (1:Active, 0:Finished)
***************************************************************************/
static uint8_t r_mtr_rdc_AdjstCarrierProcess( uint8_t req )
{
   uint8_t result = TRUE;

   cc_api_req.call_state = req;

   /* Call angle error adjustment API function. */
   cc_api_ret = R_RSLV_ADJST_Carrier (cc_api_req);

   /* The required control varies with the return value. */
   switch (cc_api_ret.adjst_state)
   {
      default:
      case ADJST_APIINFO_RUN_MODE:
         {
            result = TRUE;      /* Continuation of execution is reported. */
         }
      break;

      /* Application of motor control is required. */
      case ADJST_APIINFO_WAITING:
         {
            /* Execute the motor control processing. */
            r_mtr_ctrl_posspd_for_ccadjust_seq();
         }
      break;

      case ADJST_APIINFO_END_NORMAL:
      case ADJST_APIINFO_ERR_CARRIER:
      case ADJST_APIINFO_ERR_MOTOR:
      case ADJST_APIINFO_END_USER_STOP:
         {
            result = FALSE;     /* The end of execution is reported. */
         }
      break;
   }

   return (result);

}
```

7.10 Communications with RDC

7.10.1 Example of Using API Functions

Figure 7.17 shows a block diagram of implementation by using API functions for communications with the RDC.

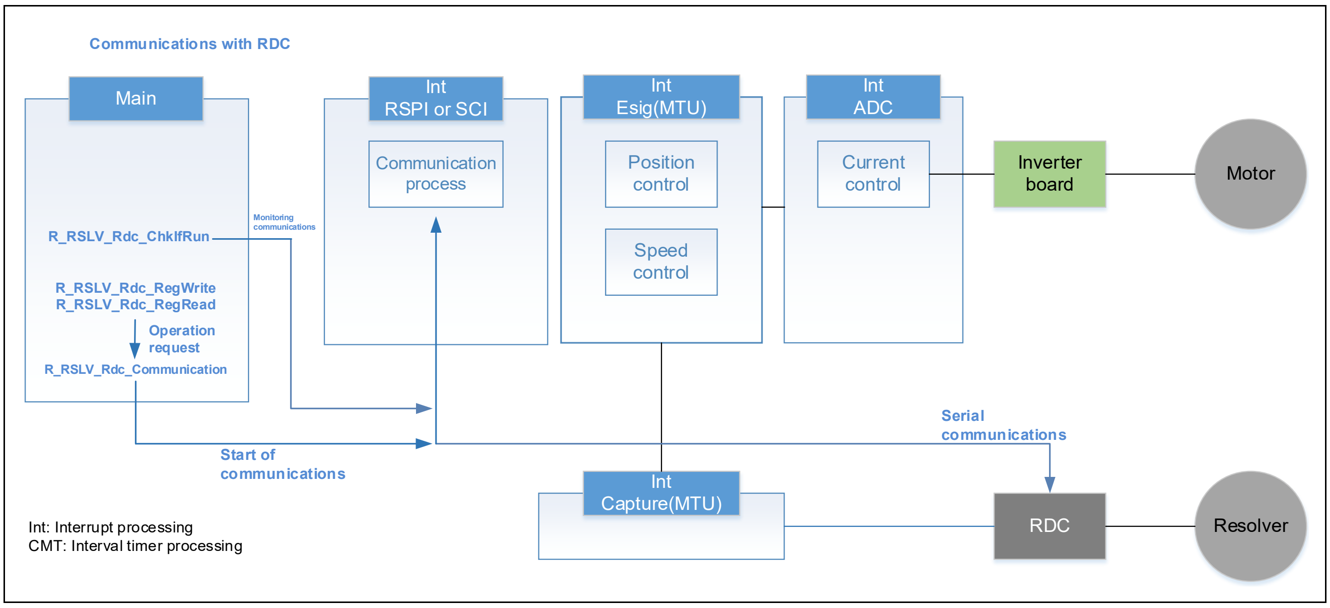


Figure 7.17 Example of Implementing Communications with RDC

An RSPI or SCI channel is used for communications with the RDC. The same API functions are used regardless of the selected type of peripheral module. The R_RSLV_Rdc_Communication function (section 6.2.28, API Function for Handling RDC Communications) is used to handle communications processing. Repeated calls of this API function are required to progress the sequence for communications, so periodically call the function. To read data from the RDC, use R_RSLV_Rdc_RegRead (section 6.2.30, API Function for Reading from an RDC Register). To write data to the RDC, use R_RSLV_Rdc_RegWrite (section 6.2.29, API Function for Writing to an RDC Register). The current communication state is returned by R_RSLV_Rdc_ChkIfRun (section 6.2.31, API Function for Acquiring the RDC Register Access State). Do not issue a read or write request during execution.

For the processing of communication interrupts, use the code generated by the SC when the RSPI is used for communications. When the SCI is used, modify the code of transmit interrupt processing generated by the SC so that 16-bit communications are supported because the SC does not generate code for the 16-bit communication format. Refer to section 7.10.2.3, Example of Using the SCI, to implement the code for supporting the 16-bit format.

7.10.2 Sample Code

The following shows sample code.

7.10.2.1 Writing to an RDC Register

The following shows an example of code for writing to an RDC register.

```
main( void )
{
   while (1U)
   {
     if (TRUE == flg_write_req)
     {
         /* Write data to the RDC register buffer. */
         R_RSLV_Rdc_SetRegisterVal(rdc_write_data, rdc_address);
         /* Issue a write request. */
         R_RSLV_Rdc_RegWrite(&rdc_write_status);
         flg_write_req = FALSE;
     }
     /* Sequence of communications with the RDC */
     R_RSLV_Rdc_Communication();
   }
}
```

7.10.2.2 Reading from an RDC Register

The following shows an example of code for reading from an RDC register.

```
main( void )
{
   while (1U)
   {
     if (TRUE == flg_read_req)
     {
         /* Read data from an RDC register to the buffer. */
         R_RSLV_Rdc_RegRead(rdc_address);
         flg_read_req = FALSE;
     }
     /* Sequence of communications with the RDC */
     R_RSLV_Rdc_Communication();

     /* Get data from the RDC register buffer. */
     R_RSLV_Rdc_GetRegisterVal(&rdc_read_data, rdc_address);
   }
}
```

7.10.2.3 Example of Using the SCI

When using the SCI for communications, modify the transmit interrupt processing automatically generated by the SC to support 16-bit communications. In the following example, a new function for 16-bit transmit interrupt processing is created and added to the transmit interrupt processing in the code generated by the SC.

```
/* Transmit interrupt processing (generated by the SC)*/
#pragma interrupt r_Config_SCI0_transmit_interrupt(vect=VECT(SCI0, TXI0))
static void r_Config_SCI0_transmit_interrupt(void)
{
    // Delete the following processing and call the function for 16-bit
       transmit interrupt processing.
//    if (0U < g_sci0_tx_)
//    {
//        SCI0.TD count R = *gp_sci0_tx_address;
//        gp_sci0_tx_address++;
//        g_sci0_tx_count--;
//    }
//    else
//    {
//        SCI0.SCR.BIT.TIE = 0U;
//        SCI0.SCR.BIT.TEIE = 1U;
//    }
    R_SCI0_Trans_Intr_Process();     // Add the new function. See the code for 16-bit
    transmit interrupt processing shown later.
}

/* Receive interrupt processing (generated by the SC)*/
#pragma interrupt r_Config_SCI0_receive_interrupt(vect=VECT(SCI0, RXI0))
static void r_Config_SCI0_receive_interrupt(void)
{
    if (g_sci0_rx_length > g_sci0_rx_count)
    {
        *gp_sci0_rx_address = SCI0.RDR;
        gp_sci0_rx_address++;
        g_sci0_rx_count++;

        if (g_sci0_rx_length == g_sci0_rx_count)
        {
            SCI0.SCR.BIT.RIE = 0;

              /* Set the CS port to the high level.*/
              PORT9.PODR.BIT.B2 = 1U;     // Chip select signal: Chip inactive (Add this
              line.)

            /* Clear the TE and RE bits. */
            if((0U == SCI0.SCR.BIT.TIE) && (0U == SCI0.SCR.BIT.TEIE))
            {
                SCI0.SCR.BYTE &= 0xCFU;
                R_Config_SCI0_Stop();       // Place the SCI in the module stop state.
                (Add this line.)
            }

            r_Config_SCI0_callback_receiveend();
        }
        else
        {
            R_SCI0_Trans_Intr_Process();  // Prepare for the next data reception. (Add
            this line.)
        }
    }
}
/* Receive error interrupt processing (generated by the SC)*/
#pragma interrupt r_Config_SCI0_receiveerror_interrupt(vect=VECT(SCI0, ERI0))
void r_Config_SCI0_receiveerror_interrupt(void)
{
    uint8_t err_type;

    r_Config_SCI0_callback_receiveerror();

    /* Clear the overrun error flag. */
    err_type = SCI0.SSR.BYTE;
    err_type &= 0xDFU;
    err_type |= 0xC0U;
    SCI0.SSR.BYTE = err_type;
}
```

Create the following code even when the RSPI is used.

```
/* Transmit end callback processing (generated by the SC)*/
void r_Config_SCI1_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI1_callback_transmitend. Do not edit
comment generated here. */
    R_RSLV_Rdc_CallComEndCb();     // Add the API function for communication end
                                      callback processing.
    /* End user code. Do not edit comment generated here. */
}


/* Receive end callback processing (generated by the SC)*/
void r_Config_SCI1_callback_receiveend(void)
{
    /* Start user code for r_Config_SCI1_callback_receiveend. Do not edit
comment generated here. */
    R_RSLV_Rdc_CallComEndCb();     // Add the API function for communication end
                                      callback processing.
    /* End user code. Do not edit comment generated here. */
}


/* Receive error callback processing (generated by the SC)*/
void r_Config_SCI1_callback_receiveerror(void)
{
    /* Start user code for r_Config_SCI1_callback_receiveerror. Do not edit
comment generated here. */
    R_RSLV_Rdc_CallErrorCb();       // Add the API function for receive error
                                       callback processing.
    /* End user code. Do not edit comment generated here. */
}
```

```
/* Function for 16-bit transmit interrupt processing (user-created code) */
static void R_SCI0_Trans_Intr_Process(void)
{
   uint16_t com_data;

   if (0U == s_u1_pass_flg)
   {
      if (g_sci0_tx_count > 0U)
      {
         /* Determine whether to send the upper data or lower data */
         if (g_sci0_tx_count & 0x01)
         {
            com_data = *gp_sci0_tx_address & 0x00FF;
         }
         else
         {
            com_data = *gp_sci0_tx_address & 0xFF00;
            com_data >>= 8;
            s_u1_pass_flg = 1U;
         }
         /* Write data for transmission. */
         SCI0.TDR = com_data;
         g_sci0_tx_count--;
      }
      else
      {
         SCI0.SCR.BIT.TIE = 0U;
         SCI0.SCR.BIT.TEIE = 0U;
      }
   }
   else
   {
      s_u1_pass_flg = 0U;
   }
```

7.11 Detection of Disconnection from Resolver Sensor

7.11.1 Example of Using API Functions

Figure 7.18 shows a block diagram of implementation by using API functions for detection of disconnection from the resolver sensor.

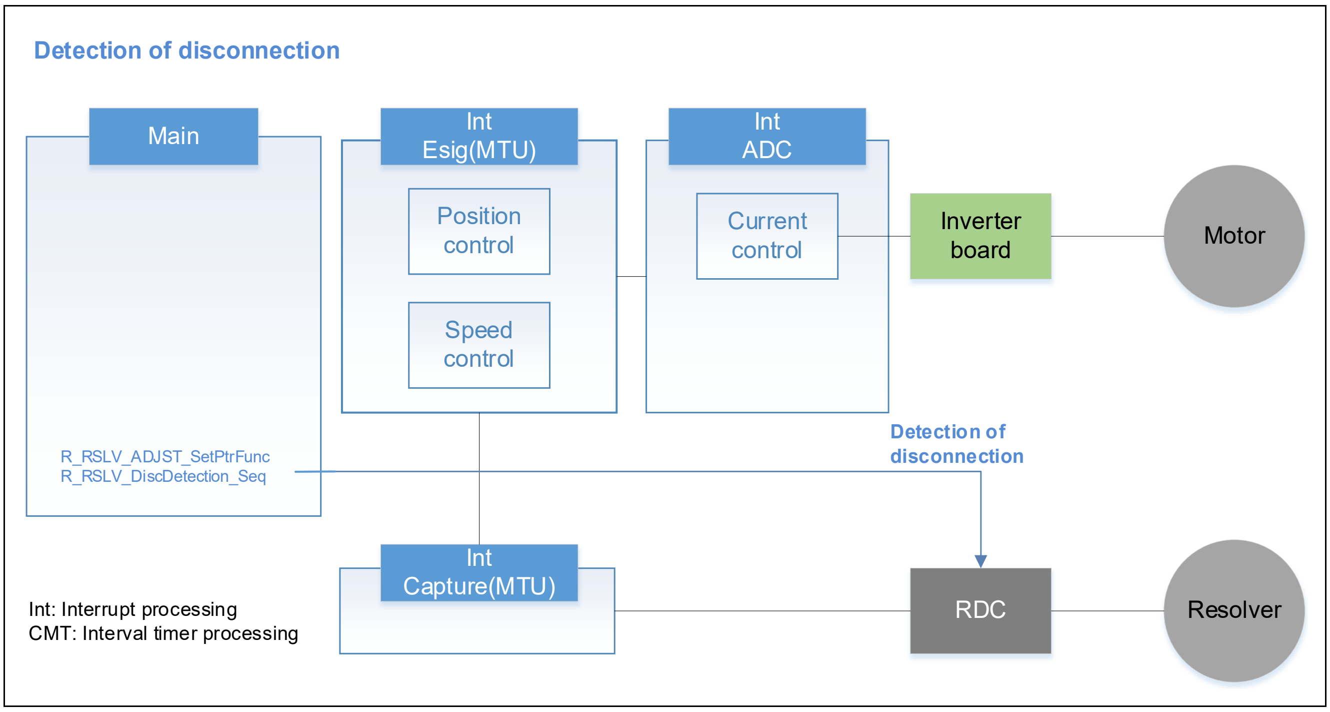


Figure 7.18 Example of Implementing Detection of Disconnection from the Resolver Sensor

To detect disconnection, use R_RSLV_ADJST_SetPtrFunc (section 6.2.41, API Function for Setting the Pointer to the User-Created Callback Function) and R_RSLV_DiscDetection_Seq (section 6.2.43, API Function for Detecting Disconnection). Repeated calls of the API function for detecting disconnection are required to progress the sequence for detection of disconnection, so periodically call the function.

For how to use the API function for specifying the pointer to the user-created callback function, see section 7.8, Automatic Adjustment of the Gain and Phase.

Figure 7.19 shows an example of implementing the processing for detecting disconnection.

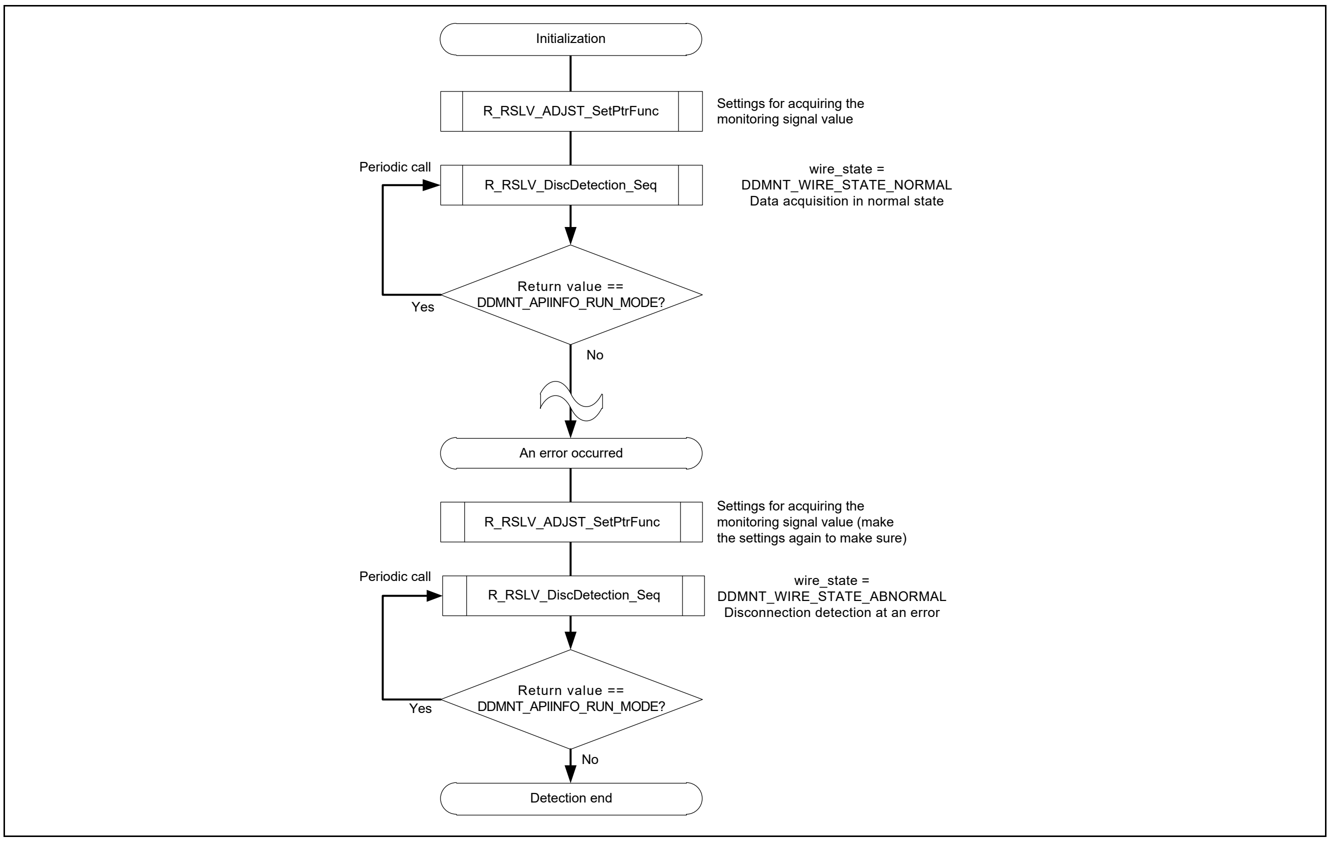

Initialization
R_RSLV_ADJST_SetPtrFunc
Settings for acquiring the monitoring signal value
Periodic call
R_RSLV_DiscDetection_Seq
wire_state = DDMNT_WIRE_STATE_NORMAL Data acquisition in normal state
Return value == DDMNT_APIINFO_RUN_MODE?
Yes
No
An error occurred
R_RSLV_ADJST_SetPtrFunc
Settings for acquiring the monitoring signal value (make the settings again to make sure)
Periodic call
R_RSLV_DiscDetection_Seq
wire_state = DDMNT_WIRE_STATE_ABNORMAL Disconnection detection at an error
Return value == DDMNT_APIINFO_RUN_MODE?
Yes
No
Detection end

Figure 7.19 Example of Disconnection Detection Sequence

In detection of disconnection, the normal connection state is compared with the error connection state to check the disconnection state of the resolver signal lines. For this reason, data in the normal connection state must be acquired in advance.

To acquire data in the normal state, call the API function R_RSLV_DiscDetection_Seq with DDMNT_WIRE_STATE_NORMAL (0U) set as the member wire_state of the structure argument st_rdc_ddmnt_arg_t for the API function. For details, see section 6.3.7, Structure for R_RSLV_DiscDetection_Seq. When the return value of this API function is not DDMNT_APIINFO_RUN_MODE (detection of disconnection is in progress), data acquisition in the normal state is complete.

Perform this processing for acquiring data in the normal state after the end of the initialization processes by the RDC driver and before the start of normal operation.

If an error occurs in operation of the motor (such as failure to update position information at the time of speed control), the disconnection detection processing is used to identify whether the error is due to disconnection of a resolver signal. Therefore, apply disconnection detection processing as required (when an error occurs) on the user side.

To check the disconnection state, call R_RSLV_DiscDetection_Seq with the arguments set as follows.

```
arg_value.call_state = DDMNT_USRREQ_RUN
arg_value.wire_state = DDMNT_WIRE_STATE_ABNORMAL
```

When the return value of this API function is not DDMNT_APIINFO_RUN_MODE (disconnection detection is in progress), the processing is complete.

In either initialization processing or disconnection detection processing (at an error), call R_RSLV_DiscDetection_Seq with the argument arg_value.call_state set to DDMNT_USRREQ_STOP to suspend the processing.

7.11.2 Sample Code

The following shows sample code.

7.11.2.1 Detection of Disconnection from the Resolver Sensor

The following shows an example of the processing for detecting disconnection.
r_mtr_DetectDisconnect_Seq() is called from the main loop.

```
/*********************************************************************
* Function Name: r_mtr_DetectDisconnect_Seq
* Description  : Sequence to detect resolver disconnection
* Arguments    : None
* Return Value : None
*********************************************************************/
/* State machine implementing detection of disconnection */
static void r_mtr_DetectDisconnect_Seq( void )
{
    st_rdc_ddmnt_arg_t temp_arg;     /* Temporary variable for API arguments */
    unsigned char dd_ret = DDMNT_APIINFO_RUN_MODE;   /* Variable for receiving
                                                        return value */

    /* A stop request is always made while detection is not being executed. */
    temp_arg.call_state = DDMNT_USRREQ_STOP;

    switch (s_u1_sts_ddcnct)
    {
        case STS_DDCNCT_NONE:
        default:
            /* Do nothing. */
        break;

        /* Start initialization. */
        case STS_DDCNCT_INIT_START:
            {
                /* Set interface functions */
                /*R_RSLV_ADJST_SetPtrFunc is called in this function. */
                r_mtr_init_ddiscnct_interface();
                SetDdiscnctStatus(STS_DDCNCT_INIT);   /* State setting macro */
            }
        break;

        /* Periodic call for waiting for the completion of initialization */
        case STS_DDCNCT_INIT:
            {
                temp_arg.call_state = DDMNT_USRREQ_RUN;
                temp_arg.wire_state = DDMNT_WIRE_STATE_NORMAL;
                dd_ret = R_RSLV_DiscDetection_Seq(temp_arg);

                /* When the return value is not DDMNT_APIINFO_RUN_MODE,
                    the processing is complete. */
                if (DDMNT_APIINFO_RUN_MODE != dd_ret)
                {
                    SetDdiscnctStatus(STS_DDCNCT_INIT_FIN);
                }
            }
        break;

        /* Post-initialization processing */
case STS_DDCNCT_INIT_FIN:
```

```
            {
                /* Set interface functions for adjustment. */
                r_mtr_init_adjst_interface();
                /* All system initialization finished. */
                s_u1_flg_system_init_fin = TRUE;
                SetDdiscnctStatus(STS_DDCNCT_NONE);
            }
        break;

        /* Start detection of disconnection at an error. */
        case STS_DDCNCT_CONF_START:
            {
                /* Set SetPtrFunc again. */
                r_mtr_init_ddiscnct_interface();
                SetDdiscnctStatus(STS_DDCNCT_CONF);
            }
        break;

        /* Periodic call for waiting for detection of disconnection */
        case STS_DDCNCT_CONF:
            {
                temp_arg.call_state = DDMNT_USRREQ_RUN;
                temp_arg.wire_state = DDMNT_WIRE_STATE_ABNORMAL;
                dd_ret = R_RSLV_DiscDetection_Seq(temp_arg);

                /* In the case of normal termination, execution is ended
                    without any further processing. */
                if (DDMNT_APIINFO_END_NORMAL == dd_ret)
                {
                    SetDdiscnctStatus(STS_DDCNCT_CONF_FIN);
                }
                /* When disconnection is detected, the disconnection information
                    is set in the variable. */
                else if (DDMNT_APIINFO_ERR_DISCONNECT == dd_ret)
                {
                    g_u2_err_status |= MTR_ERR_RSLV_DISCNCT;
                    SetDdiscnctStatus(STS_DDCNCT_CONF_FIN);
                }
                /* Periodic call in the other cases */
                else
                {
                    /* Do nothing. */
                }
            }
        break;

        /* Post-detection processing */
        case STS_DDCNCT_CONF_FIN:
                /* Set interface functions for adjustment again. */
                r_mtr_init_adjst_interface();
                SetDdiscnctStatus(STS_DDCNCT_NONE);
        break;
    }
} /* End of function r_mtr_DetectDisconnect_Seq() */
```

7.12 Cancelling an Alarm

7.12.1 Example of Using API Functions

Figure 7.20 shows a block diagram of implementation by using API functions for cancelling an alarm.

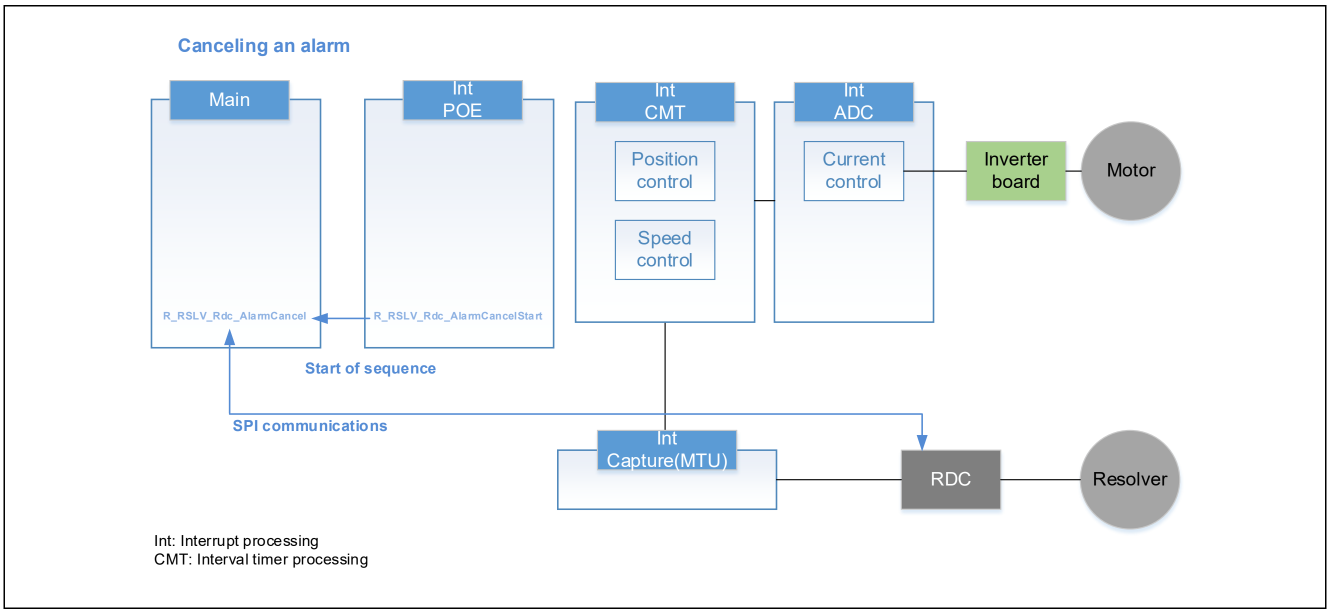


Figure 7.20 Example of Implementing Processing to Cancel an Alarm

When the RDC detects an excessive temperature, the low level is output on the alarm signal pin. In general, connect the alarm signal to a POE pin and stop the motor through forced shutdown.

To reset an alarm of the RDC, execute R_RSLV_Rdc_AlarmCancelStart (section 6.2.37, API Function for Starting RDC Alarm Cancellation) to change the driver state to the alarm reset state, and then execute R_RSLV_Rdc_AlarmCancel (section 6.2.38, API Function for Controlling the RDC Alarm Cancellation Sequence).

The API function R_RSLV_Rdc_AlarmCancel for starting alarm cancellation internally takes the form of a state machine, and so must be called periodically. R_RSLV_Rdc_AlarmCancel usually returns RSLV_MD_BUSY1. When an alarm has successfully been cancelled, RSLV_MD_OK is returned. If an alarm cannot be cancelled (continuous alarm state), RSLV_MD_ERROR is returned.

7.12.2 Sample Code

The following shows sample code.

7.12.2.1 R_RSLV_Rdc_AlarmCancelStart

This API function can be called at any time after an alarm is generated. In the following example, this function is called from the processing for the POE interrupt (POE1) generated by the ALARM signal.

```
#pragma interrupt r_mtr_rslv_foc_poe3_oei1_intr_example (vect=VECT(POE,OEI1))
void r_mtr_rslv_foc_poe3_oei1_intr_example( void )
{
    /* Post-POE processing */
    R_POE3_Stop();

    /* Start the alarm cancellation sequence. */
    R_RSLV_Rdc_AlarmCancelStart();
}
```

7.12.2.2 R_RSLV_Rdc_AlarmCancel

In the following example, this API function is called periodically in the main loop.

```
main( void )
{
  while (1)
  {
     unsigned char ret;

     ret = R_RSLV_Rdc_AlarmCancel();

     if (RSLV_MD_OK == ret)
     {
        /* Processing for successful cancellation */
     }
     else if (RSLV_MD_ERROR == ret)
     {
       /* Processing for failure in cancellation */
     }
  }
}
```

8. Migration from Rev. 1.20 and Earlier Versions to Rev. 2.00

This section shows the procedure for migrating from Rev. 1.20 and earlier versions to Rev. 2.00 of the RX24T-version resolver driver. The sample code used in migration examples is assumed to be RX24T_MRSSK_STM_RSLV_FOC_CSP_RV120 (hereafter referred to as the STM-version sample code).

8.1 Changing the Configuration of Folders and Files

The necessary steps for migrating to Rev. 2.00 are to replace the library and header files of the resolver driver and add the SC-generated code for peripheral modules.

8.1.1 Replacing the Library and Header Files and Adding the SC Code

Replace the library and header files of the resolver driver, which are located under rdc_driver_RX and shown in the following figure. Create the src folder and copy the ¥smc_gen folder into it. For details on registering files to a project, see section 8.1.2, Registering Files to a Project.

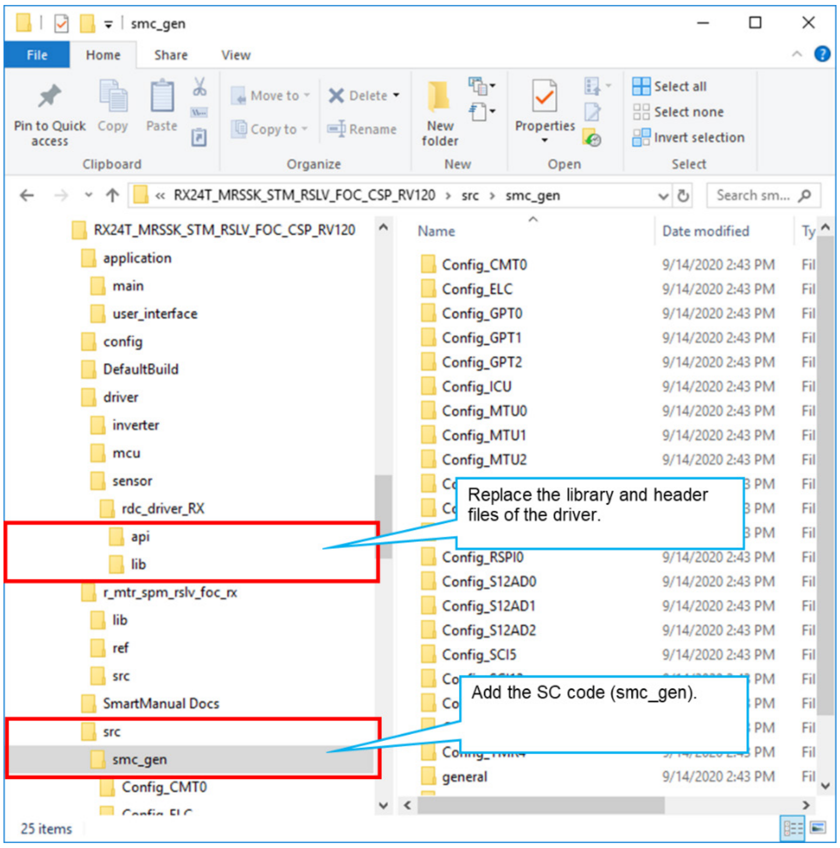


Figure 8.1 Replacing Files and Adding the SC Code

When generating code by the SC, the created code (SC code) is saved in the following folder.

```
¥smc_gen¥
        ¥Config_(peri_func)
        ¥general
        ¥r_bsp
        ¥r_config
        ¥r_pincfg
```

¥r_bsp, ¥r_config, and ¥r_pincfg are not used in this migration example. Register only ¥Config_(peri_func) and ¥general to a project. The SC code needs to be partially modified. For details on modifying the SC code, see section 8.2.2, Modifying the SC Code.

8.1.2 Registering Files to a Project

After the files have been replaced and the SC code has been added, register each file to an IDE project as shown in the following figure.

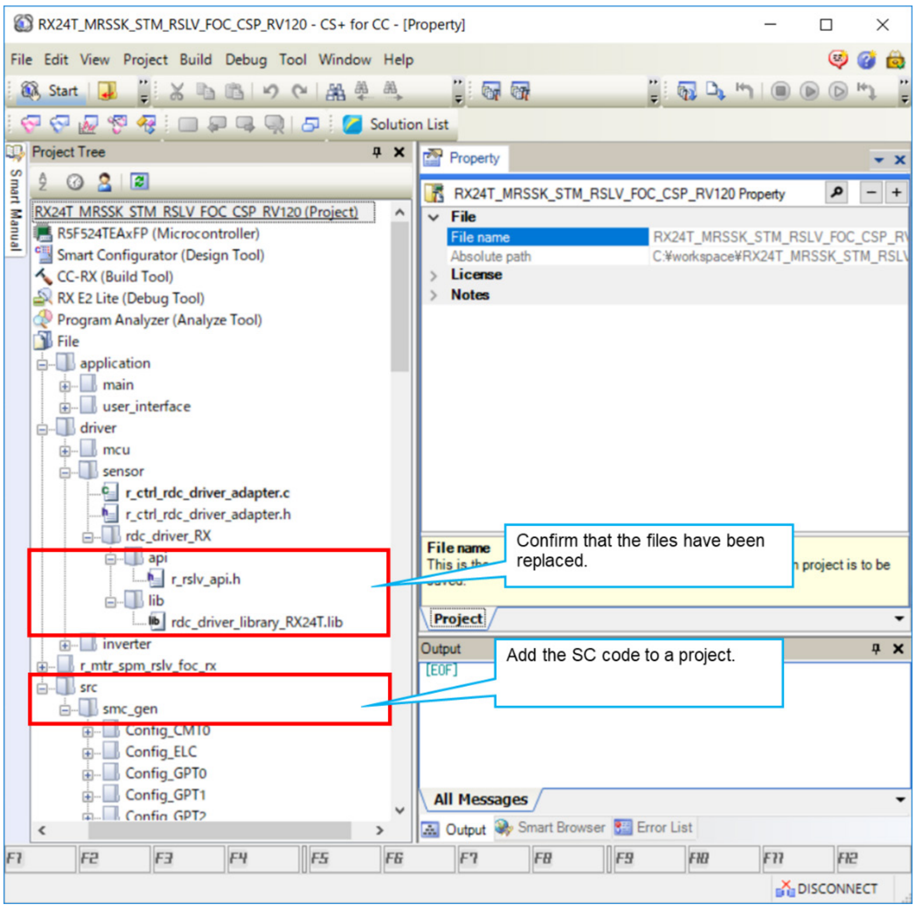

RX24T_MRSSK_STM_RSLV_FOC_CSP_RV120 - CS+ for CC - [Property]
File Edit View Project Build Debug Tool Window Help
Project Tree
RX24T_MRSSK_STM_RSLV_FOC_CSP_RV120 (Project)
R5F524TEAxFP (Microcontroller)
Smart Configurator (Design Tool)
CC-RX (Build Tool)
RX E2 Lite (Debug Tool)
Program Analyzer (Analyze Tool)
File
application
main
user_interface
driver
mcu
sensor
r_ctrl_rdc_driver_adapter.c
r_ctrl_rdc_driver_adapter.h
rdc_driver_RX
api
r_rslv_api.h
lib
rdc_driver_library_RX24T.lib
inverter
r_mtr_spm_rslv_foc_rx
src
smc_gen
Config_CMT0
Config_ELC
Config_GPT0
Config_GPT1
Config_GPT2
Property
RX24T_MRSSK_STM_RSLV_FOC_CSP_RV120 Property
File
File name
Absolute path
License
Notes
Confirm that the files have been replaced.
Add the SC code to a project.
Output
All Messages
DISCONNECT

Figure 8.2 Registering Files to a Project

8.2 Modifying the Source Code

8.2.1 Initialization Processing of Peripheral Modules

Add the SC-generated functions for initializing the peripheral modules to R_MTR_InitHardware.

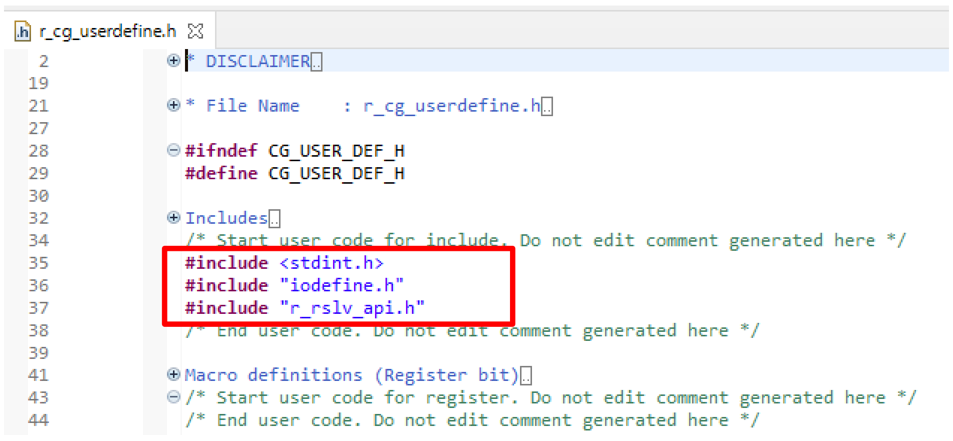
When code is generated by the SC, the initialization functions are called from R_Systeminit(). Here, however, an example of using the function for initializing the motor control block is shown.

```
void R_MTR_InitHardware (void)
{

    /*=========================*/
    /*    Initialize port    */
    /*=========================*/
    mtr_init_port();

    /*=========================*/
    /*    Initialize clock    */
    /*=========================*/
    mtr_init_clock();

    /*========================*/
    /*    Initialize WDT    */
    /*========================*/
    mtr_init_wdt();

    /*========================*/
    /*    Initialize CMT0    */
    /*========================*/
    mtr_init_cmt0();
        .
        .
    SYSTEM.PRCR.WORD = 0xA50FU;
    /* Enable writing to MPC pin function control registers */
    MPC.PWPR.BIT.B0WI = 0U;
    MPC.PWPR.BIT.PFSWE = 1U;

    R_Config_MTU9_Esig12_Create();
    R_Config_MTU0_Csig_Create ();
    R_Config_MTU2_Cap_Create ();
    R_Config_CMT1_CsigUpdTim_Create ();
    R_Config_TMR0_PhaseA_Create ();
    R_Config_TMR4_PhaseB_Create ();
    R_Config_TMR3_RdcClk_Create ();
    R_Config_RSPI0_RdcCom_Create ();

    /* Disable writing to MPC pin function control registers */
    MPC.PWPR.BIT.PFSWE = 0U;
    MPC.PWPR.BIT.B0WI = 1U;
    /* Enable protection */
    SYSTEM.PRCR.WORD = 0xA500U;
        .
        .
```

8.2.2 Modifying the SC Code

After registering the SC code to a project, modify the SC code as shown in the following.

8.2.2.1 Adding an Include Declaration to r_cg_userdefine.h

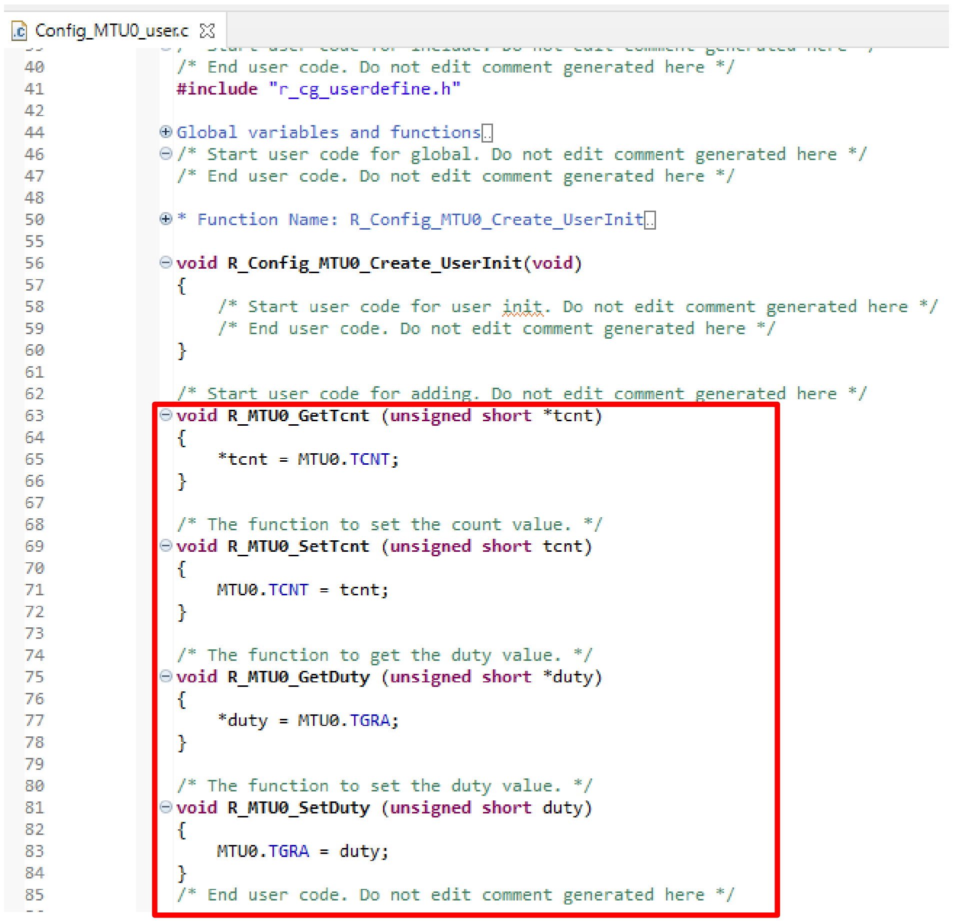


r_cg_userdefine.h

```
/* DISCLAIMER

* File Name    : r_cg_userdefine.h

#ifndef CG_USER_DEF_H
#define CG_USER_DEF_H

Includes
/* Start user code for include. Do not edit comment generated here */
#include <stdint.h>
#include "iodefine.h"
#include "r_rslv_api.h"
/* End user code. Do not edit comment generated here */

Macro definitions (Register bit)
/* Start user code for register. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */
```

8.2.2.2 Adding User-Created Code to Each Config_(peri_func)_user.c File

Add the user-created code for function tables, according to section 5.4, Setting up Function Tables. Also, add a prototype declaration to Config_(peri_func).h. The following code shows an example in which the angle error correction signal is assigned to the MTU0.



Config_MTU0_user.c

```
/* End user code. Do not edit comment generated here */
#include "r_cg_userdefine.h"

Global variables and functions
/* Start user code for global. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */

* Function Name: R_Config_MTU0_Create_UserInit

void R_Config_MTU0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    /* End user code. Do not edit comment generated here */
}

/* Start user code for adding. Do not edit comment generated here */
void R_MTU0_GetTcnt (unsigned short *tcnt)
{
    *tcnt = MTU0.TCNT;
}

/* The function to set the count value. */
void R_MTU0_SetTcnt (unsigned short tcnt)
{
    MTU0.TCNT = tcnt;
}

/* The function to get the duty value. */
void R_MTU0_GetDuty (unsigned short *duty)
{
    *duty = MTU0.TGRA;
}

/* The function to set the duty value. */
void R_MTU0_SetDuty (unsigned short duty)
{
    MTU0.TGRA = duty;
}
/* End user code. Do not edit comment generated here */
```

8.2.2.3 Porting Each Interrupt Processing of the Project before Migration to the Interrupt Processing of the SC Code

When an interrupt setting is enabled by the SC, an interrupt processing function is automatically created. Port the interrupt processing that was created before migration to the interrupt processing generated by the SC. After that, delete the interrupt processing created before migration (enable the interrupt processing function created in the SC code).

- Excitation signal interrupt
- Interrupt for updating the duty cycle of the angle error correction signal
- Interrupt for acquiring the angle detection value

Coding example: Interrupt for updating the duty cycle of the angle error correction signal

(Before migration)

```
r_mtr_interrupt.c
/***********************************************************************************
* Function Name : mtr_csig_interrupt
* Description   : CMI1 interrupt(Duty update of PWM for angle error correction)
* Arguments     : None
* Return Value  : None
**********************************************************************************/
#pragma interrupt (mtr_csig_interrupt(vect = VECT_RSLV_CSIG))
static void mtr_csig_interrupt(void)
{
setpsw_i();                                 /* Interrupt enabled */
   R_RSLV_INT_CSig_UpdatePwmDuty();
} /* End of function mtr_csig_interrupt */

Config_CMT0.c:
#pragma interrupt r_Config_CMT0_cmi0_interrupt(vect=VECT(CMT0,CMI0))
static void r_Config_CMT0_cmi0_interrupt(void)
{
   /* Start user code for r_Config_CMT0_ cmi0_interrupt. Do not edit comment
generated here. */
   /* End user code. Do not edit comment generated here. */
}
```

(After migration)

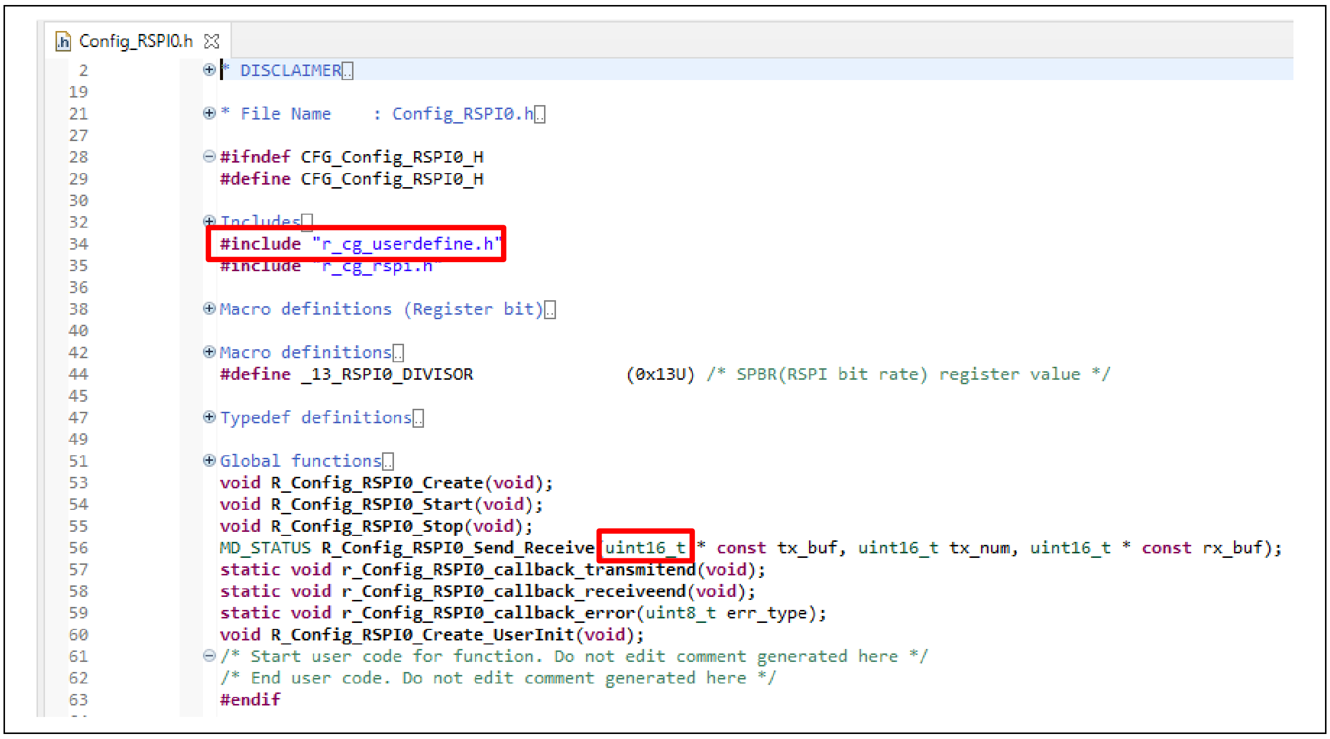
```
r_mtr_interrupt.c
```

Delete the whole function.

```
/***********************************************************************************
* Function Name : mtr_csig_interrupt
* Description   : CMI1 interrupt(Duty update of PWM for angle error correction)
* Arguments     : None
* Return Value  : None
**********************************************************************************/
// #pragma interrupt (mtr_csig_interrupt(vect = VECT_RSLV_CSIG))
// static void mtr_csig_interrupt(void)
// {
// setpsw_i();                                 /* Interrupt enabled */
//   R_RSLV_INT_CSig_UpdatePwmDuty();  // Deleted
//} /* End of function mtr_csig_interrupt */

Config_CMT0.c:
#pragma interrupt r_Config_CMT0_cmi0_interrupt(vect=VECT(CMT0,CMI0))
static void r_Config_CMT0_cmi0_interrupt(void)
{
    /* Start user code for r_Config_CMT0_ cmi0_interrupt. Do not edit comment
    generated here. */
    R_RSLV_INT_CSig_UpdatePwmDuty();   // Added
    /* End user code. Do not edit comment generated here. */
}
```

8.2.2.4 Modifying the Header File for Communication Peripheral Modules (RSPI and SCI)

Since definitions (e.g., uint16_t) are written in the prototype declaration of the SC-generated header file for communication peripheral modules, including "r_cg_userdefine.h" is necessary.

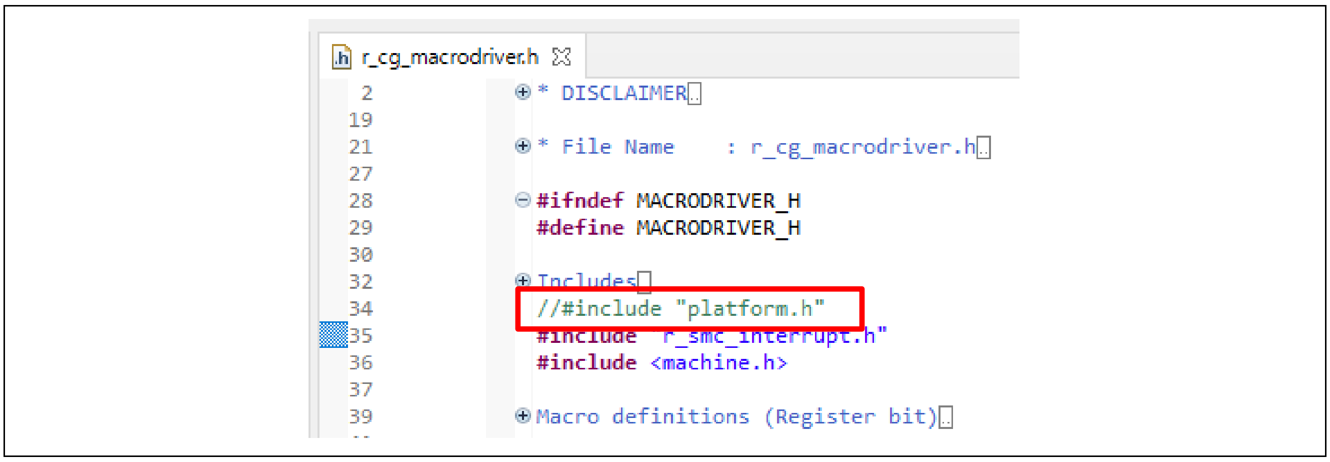


Config_RSPI0.h

```
* DISCLAIMER

* File Name    : Config_RSPI0.h

#ifndef CFG_Config_RSPI0_H
#define CFG_Config_RSPI0_H

Includes
#include "r_cg_userdefine.h"
#include "r_cg_rspi.h"

Macro definitions (Register bit)

Macro definitions
#define _13_RSPI0_DIVISOR                (0x13U) /* SPBR(RSPI bit rate) register value */

Typedef definitions

Global functions
void R_Config_RSPI0_Create(void);
void R_Config_RSPI0_Start(void);
void R_Config_RSPI0_Stop(void);
MD_STATUS R_Config_RSPI0_Send_Receive(uint16_t * const tx_buf, uint16_t tx_num, uint16_t * const rx_buf);
static void r_Config_RSPI0_callback_transmitend(void);
static void r_Config_RSPI0_callback_receiveend(void);
static void r_Config_RSPI0_callback_error(uint8_t err_type);
void R_Config_RSPI0_Create_UserInit(void);
/* Start user code for function. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */
#endif
```

8.2.2.5 Modifying r_cg_macrodriver.h

Since r_bsp is not used in this migration method, delete the #include line of platform.h from r_cg_macrodriver.h. This step is unnecessary when r_bsp is used.



r_cg_macrodriver.h

```
* DISCLAIMER

* File Name    : r_cg_macrodriver.h

#ifndef MACRODRIVER_H
#define MACRODRIVER_H

Includes
//#include "platform.h"
#include "r_smc_interrupt.h"
#include <machine.h>

Macro definitions (Register bit)
```

8.2.3 Modifying the API Functions

When changing the resolver driver from Rev. 1.20 and earlier versions to Rev. 2.00, the implementation method also has to be changed because the specifications of some API functions differ. The following table lists the API functions that require change. For details on modification, see section 8.2.3.1, Deletion of the R_RSLV_CreatePeripheral API Function, and onwards.

Table 8-1 List of API Functions Requiring Modification

API Function	Modification	Modification Method
R_RSLV_CreatePeripheral(ST_INIT_REG_PARAM *rdc_init_param)	Deletion	Delete the code where this function is used.
R_RSLV_SetFuncTable(unsigned char set_func, FUNCTION_TABLE user_func_table)	Addition	Implement this function in the initialization processing. Delete the R_RSLV_CreatePeripheral() setting in RESOLVER_peripheral_init() and RDC_peripheral_init(), and at that location write the processing for setting up a function table.
R_RSLV_SetSystemInfo(ST_SYSTEM_PARAM *rdc_sys_param)	Change	Change the function as follows: R_RSLV_SetSystemInfo(ST_SYSTEM_PARAM *rdc_sys_param, ST_USER_PERI_PARAM *user_peri_param)
R_RSLV_SetCaptureTiming(uint16_t tcnt)	Change	Replace these functions with the following function. R_RSLV_ESigCapStartTiming (uint16_t esig_start_tcnt, uint16_t cap_start_tcnt)
R_RSLV_EsigStartTiming(uint16_t tcnt)	Change	

R_RSLV_Rdc_RegWrite(
uint8_t wt_data,
uint8_t address,
uint8_t *write_status
)
	Change	Change the calling function as follows:
		R_RSLV_Rdc_RegWrite(
		unsigned char *write_status
		)
		In addition, change the calling method as follows:
		(Before modification)
		R_RSLV_Rdc_SetRegisterVal (data1,address1);
		R_RSLV_Rdc_SetRegisterVal (data2,address2);
		R_RSLV_Rdc_RegWrite(data3,address3,&com_sts);
		(After modification)
		R_RSLV_Rdc_SetRegisterVal (data1,address1);
		R_RSLV_Rdc_SetRegisterVal (data2,address2);
		R_RSLV_Rdc_SetRegisterVal (data3,address3);
		R_RSLV_Rdc_RegWrite(&com_sts);

R_RSLV_INT_RdcCom_Recv(void)	Deletion	Delete the code where this function is called.
R_RSLV_INT_RdcCom_Trans(void)	Deletion	Delete the code where this function is called.
R_RSLV_INT_RdcCom_Error(void)	Deletion	Delete the code where this function is called.
R_RSLV_INT_RdcCom_Idle(void)	Deletion	Delete the code where this function is called.
R_RSLV_SetFunctionPointer(UNSIGNED_CHAR_POINTER *func, unsigned char func_id)	Deletion	Delete the code where this function is called.
R_RSLV_Rdc_CallComEndCb(void)	Addition	Add this function to the transmit/receive end interrupt callback processing.
R_RSLV_Rdc_CallErrorCb()	Addition	Add this function to the error interrupt callback processing.
R_RSLV_GetCSigStatus(unsigned char *status)	Deletion	Delete the code where this function is called.
R_RSLV_ADJST_SetPtrFunc(st_ptr_func_arg_t *ptr_arg)	Change	This function is changed so that the return value is returned. Determine the return value as required.

8.2.3.1 Deleting the R_RSLV_CreatePeripheral API Function

Since this API function for initializing the peripheral modules was deleted from Rev. 2.00, delete the code where this function is used. The following function is used in the STM-version sample code.

- RESOLVER_peripheral_init(void)
- RDC_peripheral_init(void)

Example:
(Deleted)
```
     // MTU3_9 ESig12
 //    rdc_init_param.u1_sel_reg_type    = T_MTU3_9;
 //    rdc_init_param.u1_sel_reg_func    = F_ESIG12;
 //    rdc_init_param.u1_sel_int_flg     = INT_ENABLE;
 //    rdc_init_param.u1_sel_int_priorty = 11;
 //    rdc_init_param.u1_capture_trig    = CAPTURE_TRIG_NONE;
 //    rdc_init_param.u1_use_port1      = P_P21;
 //    rdc_init_param.u1_use_port2      = P_PE0;
 //    rdc_init_param.u1_use_port3      = 0xFF;      // Not used
 //    rdc_init_param.u1_use_port4      = 0xFF;      // Not used
 //    R_RSLV_CreatePeripheral(&rdc_init_param);
```

8.2.3.2 Adding the R_RSLV_SetFuncTable API Function

Add the processing to set up a function table at the locations where R_RSLV_CreatePeripheral was deleted. The following function is used in the STM-version sample code, as in section 8.2.3.1.

- RESOLVER_peripheral_init(void)
- RDC_peripheral_init(void)

Example:
(Deleted)
```
     // MTU3_9 ESig12
 //    rdc_init_paramu1_sel_reg_type    = T_MTU3_9;
 //    rdc_init_param.u1_sel_reg_func    = F_ESIG12;
 //    rdc_init_param.u1_sel_int_flg     = INT_ENABLE;/
 //    rdc_init_param.u1_sel_int_priorty = 11;
 //    rdc_init_param.u1_capture_trig    = CAPTURE_TRIG_NONE;
 //    rdc_init_param.u1_use_port1      = P_P21;
 //    rdc_init_param.u1_use_port2      = P_PE0;
 //    rdc_init_param.u1_use_port3      = 0xFF;      // Not used
 //    rdc_init_param.u1_use_port4      = 0xFF;      // Not used
 //    R_RSLV_CreatePeripheral(&rdc_init_param);
```
(Added)
```
     /* Set up the function table for ESig */
     g_st_user_func_table.Start   = &R_Config_MTU9_Esig_Start;
     g_st_user_func_table.Stop    = &R_Config_MTU9_Esig_Stop;
     g_st_user_func_table.SetTcnt = &R_Config_MTU9_Esig_SetTcnt;
     g_st_user_func_table.GetTcnt = &R_Config_MTU9_Esig_GetTcnt;
     R_RSLV_SetFuncTable(F_ESIG12, g_st_user_func_table);
```

8.2.3.3 Changing the R_RSLV_SetSystemInfo API Function

Change the arguments (parameters) of R_RSLV_SetSystemInfo. For the arguments (parameters), see section 6.3.2, Structures for R_RSLV_SetSystemInfo. The following function is where the STM-version sample code has to be changed.

- RESOLVER_peripheral_init(void)

  In this function, modify the code as follows:

(Before modification)

```
/* RX24T 100 pins */
st_system_param.u1_mcu_type = MCU_TYPE_R5F524TAADFP;
/* Excitation signal (ESig) frequency 5 kHz */
st_system_param.u1_esig_freq = R_ESIG_SET_FREQ_20K;
/* Correction signal (CSig) frequency 400 kHz */
st_system_param.u1_csig_freq = R_CSIG_SET_FREQ_200K;
/* Update the duty cycle 2 times.*/
st_system_param.u1_csig_upd_duty_cycle = R_CSIG_SET_DCNT_02;
/* Use MTU synchronous start. */
st_system_param.u1_mtu3_sync_start = MTU_SYNC_START_ENABLE;
/* Target motor is a BLDC motor. */
st_system_param.u1_motor_kind = MOTOR_STM;
st_system_param.u1_extension_use = R_EXT_INACTIVE;
R_RSLV_SetSystemInfo(&st_system_param);
```

(After modification)

```
/* Excitation signal (ESig) frequency 20 kHz */
st_system_param.u1_esig_freq = R_ESIG_SET_FREQ_20K;
/* Correction signal (CSig) frequency 200 kHz */
st_system_param.u1_csig_freq = R_CSIG_SET_FREQ_200K;
/* Update the duty cycle 2 times.*/
st_system_param.u1_csig_upd_duty_cycle = R_CSIG_SET_DCNT_02;
/* Use MTU synchronous start. */
st_system_param.u1_sync_start = SYNCMD_OTHER_API;
/* Target motor is a STM motor. */
st_system_param.u1_motor_kind = MOTOR_STM;
/* RDC IC MNTOUT output method */
st_system_param.u1_mntout_type = RSLV_MNTOUT_TYPE_AC;
st_user_peri_param.f_esig1_peri_clk_src = 80.0f;
st_user_peri_param.f_csig_peri_clk_src = 80.0f;
st_user_peri_param.f_csig_upd_timer_peri_clk_src = 5.0f; // CMT:PCLKB/8
st_user_peri_param.f_capture_peri_clk_src = 80.0f;
st_user_peri_param.f_phase1_peri_clk_src = 40.0f;
st_user_peri_param.f_phase2_peri_clk_src = 40.0f;
R_RSLV_SetSystemInfo(&st_system_param, &st_user_peri_param);
```

8.2.3.4 Changing the R_RSLV_SetCaptureTiming and R_RSLV_EsigStartTiming API Functions

Delete R_RSLV_SetCaptureTiming and R_RSLV_EsigStartTiming, and add R_RSLV_Set_EsigCapTiming to an appropriate location. For details on how to use the API function, see section 6.2.19, API Function for Setting the Timing to Start the Excitation Signal Output.

Modify the code as follows:

(Before modification)

```
R_RSLV_SetCaptureTiming(DEF_SFT_ADJ_ESIG);      /* Capture start timing */
R_RSLV_ESigStartTiming(DEF_DELAY_ADJ_ESIG);     /* Esig start timing*/
```

(After modification)

```
    R_RSLV_ESigCapStartTiming(DEF_DELAY_ADJ_ESIG, DEF_SFT_ADJ_ESIG); /* Esig &
Capture start timing*/
```

8.2.3.5 Changing the R_RSLV_Rdc_RegWrite API Function

Change the arguments of this function. For details on how to use the API function, see section 6.2.29, API Function for Writing to an RDC Register. Also, change the method of calling the API function as shown below.

(Before modification)

```
R_RSLV_Rdc_SetRegisterVal (data1,address1);
R_RSLV_Rdc_SetRegisterVal (data2,address2);
R_RSLV_Rdc_RegWrite(data3,address3,&com_sts);
```

(After modification)

```
R_RSLV_Rdc_SetRegisterVal (data1,address1);
R_RSLV_Rdc_SetRegisterVal (data2,address2);
R_RSLV_Rdc_SetRegisterVal (data3,address3);
R_RSLV_Rdc_RegWrite(&com_sts);
```

8.2.3.6 Deleting the R_RSLV_INT_RdcCom_Recv API Function

This API function was deleted from Rev. 2.00 because a receive interrupt processing for SPI communications is to be created by the SC. Therefore, use the receive interrupt processing that was generated by the SC. For the receive interrupt processing generated by the SC, see the receive interrupt processing (generated by the SC) in section 7.10.2.3, Example of Using the SCI.

8.2.3.7 Deleting the R_RSLV_INT_RdcCom_Trans API Function

This API function was deleted from Rev. 2.00 because a transmit interrupt processing for SPI communications is to be created by the SC. Therefore, use the transmit interrupt processing that was generated by the SC. For the transmit interrupt processing generated by the SC, see the transmit interrupt processing (generated by the SC) in section 7.10.2.3, Example of Using the SCI.

8.2.3.8 Deleting the R_RSLV_INT_RdcCom_Error API Function

This API function was deleted from Rev. 2.00 because an error interrupt processing for SPI communications is to be created by the SC. Therefore, use the error interrupt processing that was generated by the SC. For the error interrupt processing generated by the SC, see the error interrupt processing (generated by the SC) in section 7.10.2.3, Example of Using the SCI.

8.2.3.9 Deleting the R_RSLV_INT_RdcCom_Idle API Function

This API function was deleted from Rev. 2.00 because an idle interrupt processing for SPI communications is to be created by the SC. Therefore, use the idle interrupt processing that was generated by the SC. However, if an SCIx module is set by the SC, the idle interrupt processing is not created and so the only change is that this API function was deleted.

8.2.3.10 Deleting the R_RSLV_SetFunctionPointer API Function

This API function was deleted from Rev. 2.00 because the chip select signal is to be output in the code generated by the SC. Therefore, delete the code where this function is called.

8.2.3.11 Adding the R_RSLV_Rdc_CallComEndCb API Function

Call this API function from the callback functions (r_Config_(peri_func)_callback_transmitend() and r_Config_(peri_func)_callback_ receiveend ()) for SPI communication end interrupts, which are generated by the SC. For details, see the callback processing in section 7.10.2.3, Example of Using the SCI.

8.2.3.12 Adding the R_RSLV_Rdc_CallErrorCb API Function

Call this API function from the callback function (r_Config_(peri_func)_callback_error()) for the SPI communication error interrupt, which is generated by the SC. For details, see the callback processing in section 7.10.2.3, Example of Using the SCI.

8.2.3.13 Deleting the R_RSLV_GetCSigStatus API Function

This API function was deleted from Rev. 2.00 because the output state of the angle error correction signal can be determined from the settings of peripheral module registers. Therefore, delete the code where this function is called.

8.2.3.14 Modifying the R_RSLV_ADJST_SetPtrFunc API Function

This API function was modified so that the return value is returned. Handle the return value as required. For details on how to use the API function, see section 6.2.41, API Function for Setting the Pointer to the User-Created Callback Function.

8.2.4 Other Modifications

8.2.4.1 Interrupt

When an interrupt of each driver facility is enabled by the SC, an interrupt function is created. If the same interrupt function has already been created, the recommended actions are to implement the processing in the interrupt function that was generated by the SC and delete the former interrupt function.

8.2.4.2 Adding and Deleting Structures

In Rev. 2.00, ST_USER_PERI_PARAM has been added and ST_INIT_REG_PARAM has been deleted. Modify the STM-version sample code as follows:

- RESOLVER_peripheral_init(void)
  Delete the definition of ST_INIT_REG_PARAM and the code where it is used.
- RDC_peripheral_init(void)
  Delete the definition of ST_INIT_REG_PARAM and the code where it is used.
  Add the definition of ST_USER_PERI_PARAM, and then add ST_USER_PERI_PARAM to the arguments of R_RSLV_SetSystemInfo().

9. Notes

Note the following when making initial settings.

9.1 Initial Setting Procedure

Follow the steps below to make initial settings.

1. Specify system information (R_RSLV_SetSystemInfo()).
2. Specify each function table (R_RSLV_SetFuncTable()).
3. Acquire RDC driver setting information (R_RSLV_GetRdcDrvSettingInfo()).
4. Make other settings.

Using a different procedure for settings might lead to timer values being other than as intended or abnormal RDC driver setting information.

9.2 Assigning Multiple Driver Facilities to a Single Peripheral Module

Do not assign more than one driver facility to a single peripheral module. Doing so does not lead to a faulty setting but only the last setting to have been made is effective.

Examples of setting:	ESIG12 and CAPTURE are assigned to MTU3_9.
	RDC_CLK and PHASE_A are assigned to TMR0.

9.3 Assigning Multiple Peripheral Modules to a Single Driver Facility

Do not assign more than one peripheral module to a single driver facility. Doing so does not lead to a faulty setting but only the last setting to have been made is effective.

Examples of setting:	MTU3_0 and MTU3_9 are assigned to ESIG12.
	TMR0 and TMR1 are assigned to PHASE_A.

9.4 Initializing Variables for Communications with the RDC

Do not perform RDC communications processing before initialization of the communications variables for the RDC (R_RSLV_Rdc_VariableInit). Doing so may lead to faulty settings in the RDC registers.

9.5 Specifying Peripheral Modules for Phase Adjustment Signals

Do not specify a single peripheral module for both phase adjustment signals (F_PHASE_A and F_PHASE_B). Doing so does not lead to a faulty setting but the output phase adjustment signals will not be correct.

Examples of setting:	TMR0 is assigned to PHASE_A and PHASE_B.

9.6 Setting Timer Start Timing

Set the timing for starting the timers for output of the excitation signal and input of the angle signal before starting the timers. Failure to do so may lead to a timer count error and an unexpected value of the angle signal may be obtained.

9.7 Adjustment Operation

The adjustment facilities operate only while the basic facilities are operating. Do not start the adjustment operation while the facilities described in sections 7.4 to 7.7 and 7.10 are stopped.

9.8 Amount of Phase Shift for Angle Error Correction

If the amount of phase shift is set to a value close to 0, the phase of the angle error correction signal may not change as specified. This is because the start of the timer for generating the duty cycle updating interrupt is delayed due to the program processing time and interrupt disabled period when the API function for synchronously starting the angle error correction signal (R_RSLV_INT_CSig_SyncStart) is executed in the processing of the exciting signal interrupt. The updating of the duty cycle of the correction signal is also delayed if a timer interrupt for updating the duty cycle of the angle error correction signal is generated while the processing of another interrupt (such as angle detection interrupt) is in progress. This is because the duty cycle updating interrupt is processed after the processing of the other interrupt generated first is completed.

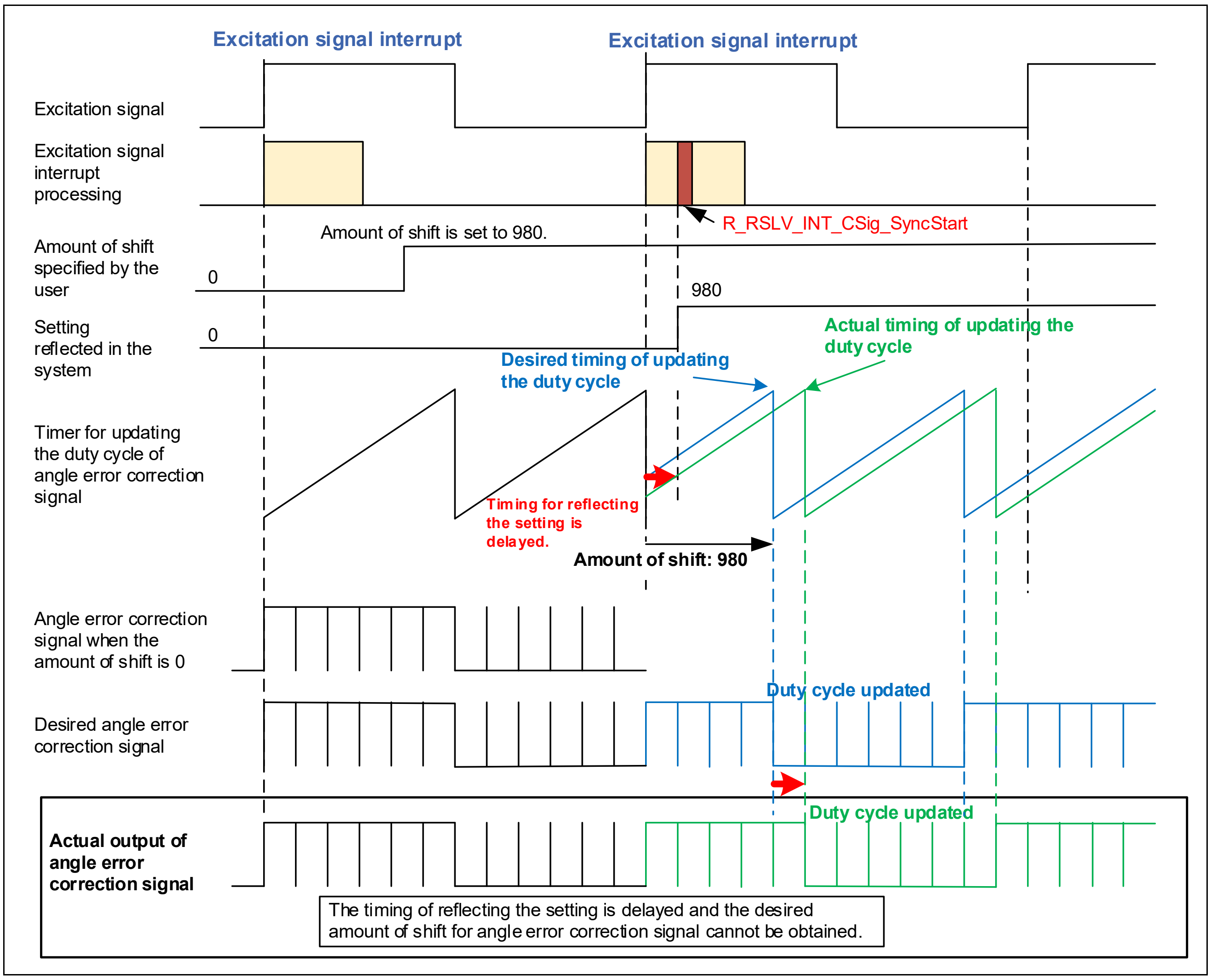

Excitation signal interrupt
Excitation signal interrupt
Excitation signal
Excitation signal interrupt processing
Amount of shift is set to 980.
R_RSLV_INT_CSig_SyncStart
Amount of shift specified by the user
0
980
Setting reflected in the system
0
Actual timing of updating the duty cycle
Desired timing of updating the duty cycle
Timer for updating the duty cycle of angle error correction signal
Timing for reflecting the setting is delayed.
Amount of shift: 980
Angle error correction signal when the amount of shift is 0
Duty cycle updated
Desired angle error correction signal
Duty cycle updated
Actual output of angle error correction signal
The timing of reflecting the setting is delayed and the desired amount of shift for angle error correction signal cannot be obtained.

Figure 9.1 Mechanism of Incorrect Shift in Phase

To avoid this problem, make the following settings.

1. Set up the timer for starting the duty cycle updating interrupt for the angle error correction signal as follows.
   — Use an unused timer to generate interrupts with the same cycle as the excitation signal cycle (hereafter, this timer is called the timer for starting the updating interrupt).
   — Start counting in the timer for starting the updating interrupt within the excitation signal interrupt processing.

   In addition, make the following settings before starting counting.
   - Specify an appropriate initial value in the timer for starting the updating interrupt so that the timer generates an interrupt in synchronization with the excitation signal interrupt.
   - Enable interrupts from the timer for starting the updating interrupt.
   - Set the priory of the interrupt from the timer for starting the updating interrupt to a higher level than those of the excitation signal interrupt, angle detection interrupt, and the interrupt for updating the duty cycle of the angle error correction signal so that the interrupt from the timer for starting the updating interrupt is processed before the other interrupts. In addition, enable nesting of excitation signal interrupts.

   — Make these settings while the angle error correction signal is stopped (for example, when the setting of the angle error correction signal is modified).

2. Perform the following in the processing of the timer for starting the updating interrupt.
- Call R_RSLV_INT_CSig_SyncStart.
- Disable the timer for starting the updating interrupt.

These settings enable R_RSLV_INT_CSig_SyncStart to be executed with the correct timing even if the amount of phase shift for the angle error correction signal is set to a value close to 0 (almost the same phase as the excitation signal). The timing of duty cycle updating interrupts for the angle error correction signal is also corrected. The following shows the operations with these settings.

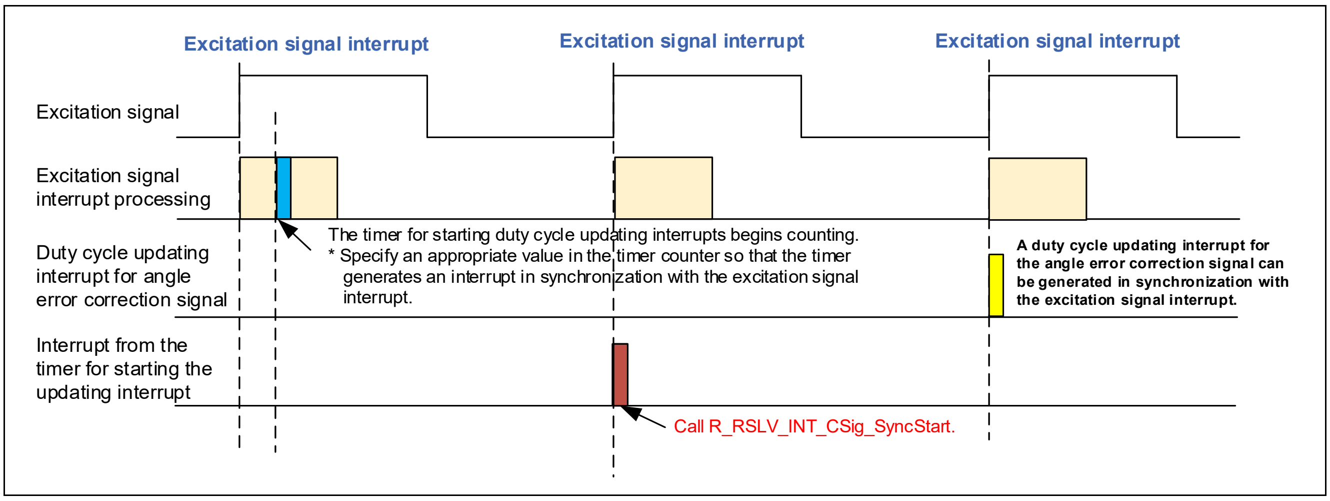

Excitation signal interrupt
Excitation signal interrupt
Excitation signal interrupt
Excitation signal
Excitation signal interrupt processing
Duty cycle updating interrupt for angle error correction signal
Interrupt from the timer for starting the updating interrupt
The timer for starting duty cycle updating interrupts begins counting.
* Specify an appropriate value in the timer counter so that the timer generates an interrupt in synchronization with the excitation signal interrupt.
A duty cycle updating interrupt for the angle error correction signal can be generated in synchronization with the excitation signal interrupt.
Call R_RSLV_INT_CSig_SyncStart.

Figure 9.2 Example of Avoiding the Problem

9.9 Order of Function Table Settings

When F_ESIG2_1 and F_ESIG2_2 are used to output the excitation signal, specify the function table for F_ESIG2_1 before that for F_ESIG2_2. Otherwise, the correct excitation signal is not output.

10. Troubleshooting

This section provides examples of actions to be taken when resolver signals are not detectable. If you have errors, identify the source of errors with reference to the flow in Figure 10.1.

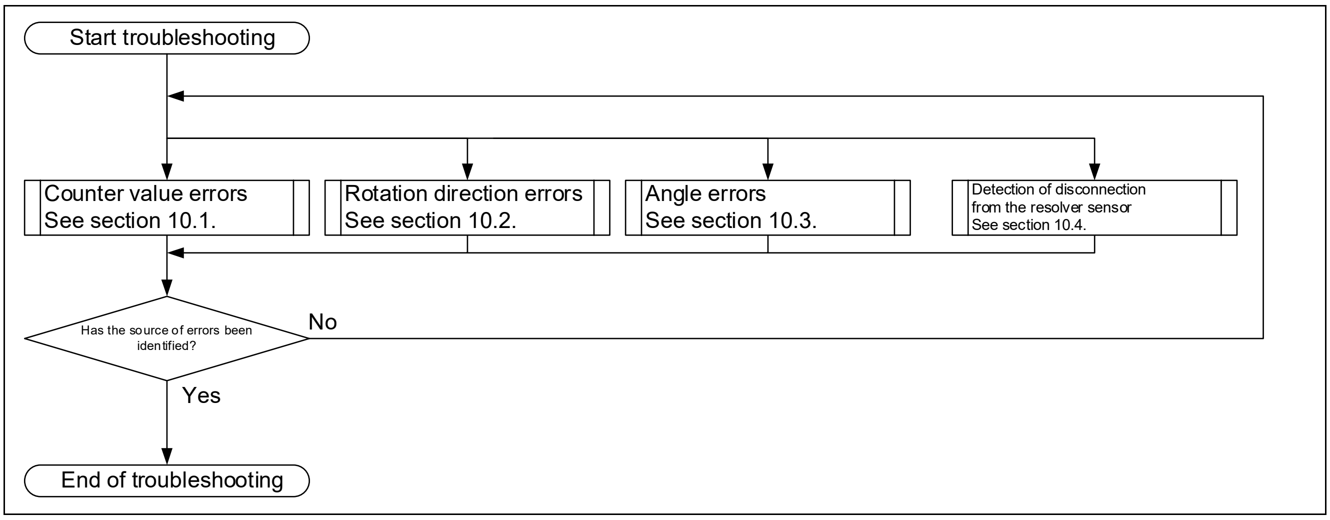

Start troubleshooting
Counter value errors See section 10.1.
Rotation direction errors See section 10.2.
Angle errors See section 10.3.
Detection of disconnection from the resolver sensor See section 10.4.
Has the source of errors been identified?
No
Yes
End of troubleshooting

Figure 10.1 Overall Flow of Troubleshooting

10.1 Counter Value Errors

If a counter value error is found in the phase information in the MCU, identify the source of errors with reference to the flow in Figure 10.2. For details of detection of disconnections from the resolver sensor, see section 10.4, Detection of Disconnection from the Resolver Sensor.

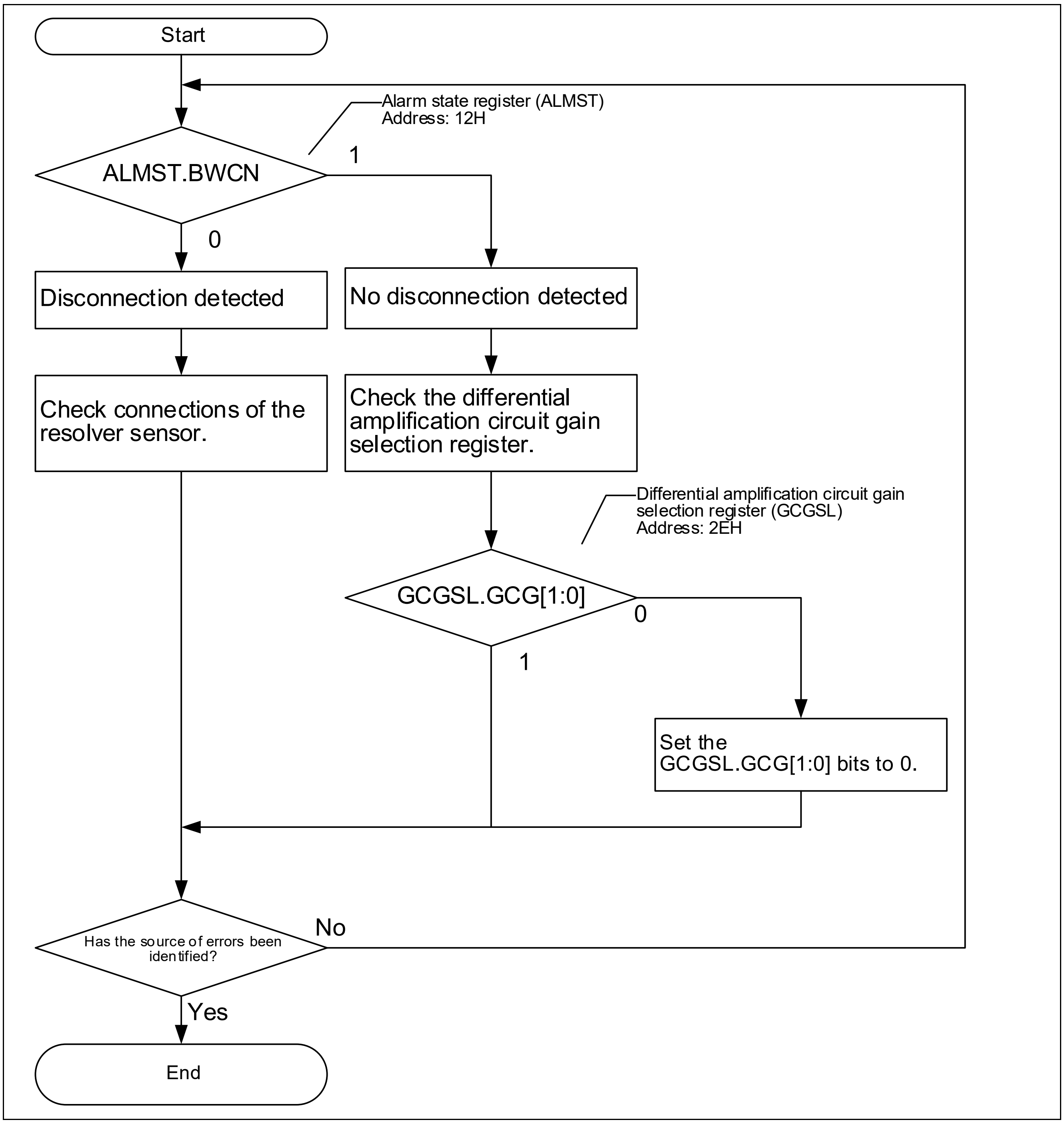

Start
Alarm state register (ALMST)
Address: 12H
ALMST.BWCN
1
0
Disconnection detected
No disconnection detected
Check connections of the resolver sensor.
Check the differential amplification circuit gain selection register.
Differential amplification circuit gain selection register (GCGSL)
Address: 2EH
GCGSL.GCG[1:0]
0
1
Set the GCGSL.GCG[1:0] bits to 0.
Has the source of errors been identified?
No
Yes
End

Figure 10.2 Counter Value Errors

10.2 Rotation Direction Errors

If the direction of rotation is not as expected, or if the resolver is not rotating in accordance with the phase information even though the resolver was physically rotated through one rotation of electrical angle, identify the source of errors with reference to the flow in Figure 10.3.

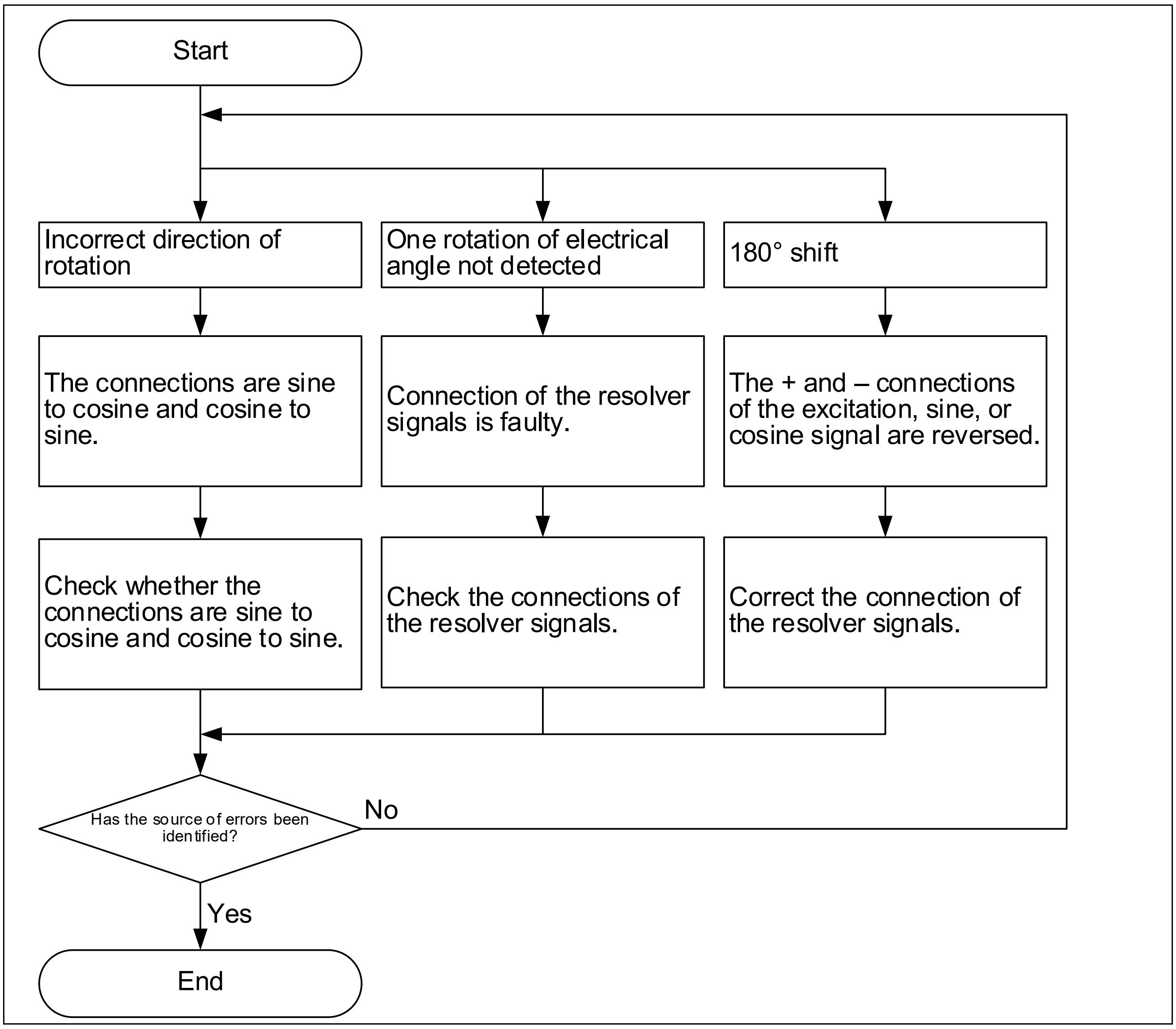

Start
Incorrect direction of rotation
One rotation of electrical angle not detected
180° shift
The connections are sine to cosine and cosine to sine.
Connection of the resolver signals is faulty.
The + and – connections of the excitation, sine, or cosine signal are reversed.
Check whether the connections are sine to cosine and cosine to sine.
Check the connections of the resolver signals.
Correct the connection of the resolver signals.
Has the source of errors been identified?
No
Yes
End

Figure 10.3 Rotation Direction Errors

10.3 Angle Errors

If the phase information from the resolver differs from the expected angle, an abnormality may be present in the signal waveform. In such cases, check the output waveform from the analog monitoring signals. To output waveforms to the analog monitoring output, set the PSMON bit in power-saving control register 3 (PS3) to 1 and make the appropriate settings in the monitor output selection register (MNTSL).

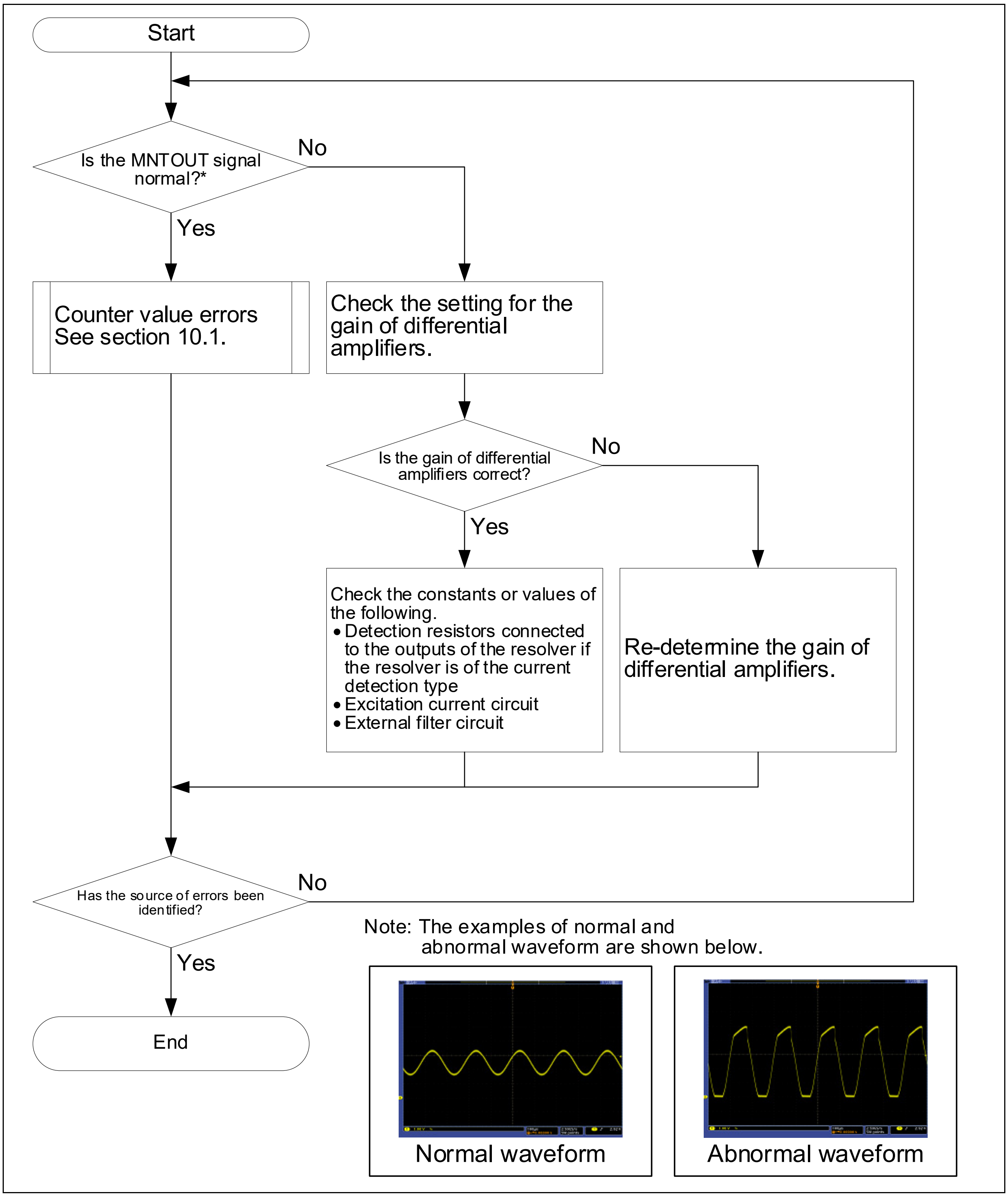

Start
Is the MNTOUT signal normal?*
No
Yes
Counter value errors See section 10.1.
Check the setting for the gain of differential amplifiers.
Is the gain of differential amplifiers correct?
No
Yes
Check the constants or values of the following.
• Detection resistors connected to the outputs of the resolver if the resolver is of the current detection type
• Excitation current circuit
• External filter circuit
Re-determine the gain of differential amplifiers.
Has the source of errors been identified?
No
Yes
End
Note: The examples of normal and abnormal waveform are shown below.
Normal waveform
Abnormal waveform

Figure 10.4 Angle Errors

10.4 Detection of Disconnection from the Resolver Sensor

The RAA3064002GFP and RAA3064003GFP only detect disconnection from the resolver sensor. After disconnection is detected, handling such as the MCU applying control to stop the output of the excitation actuating signal is required. For details on the settings for the detection of disconnection, see section 7.11, Detection of Disconnection from Resolver Sensor.

The following describes the patterns that may lead to the detection of disconnection. How disconnection is detected depends on the configuration of the resolver in use.

Figure 10.5 shows normal operation and Figures 10.6 to 10.8 show cases of the detection of disconnection when the resolver is of the transformer type.

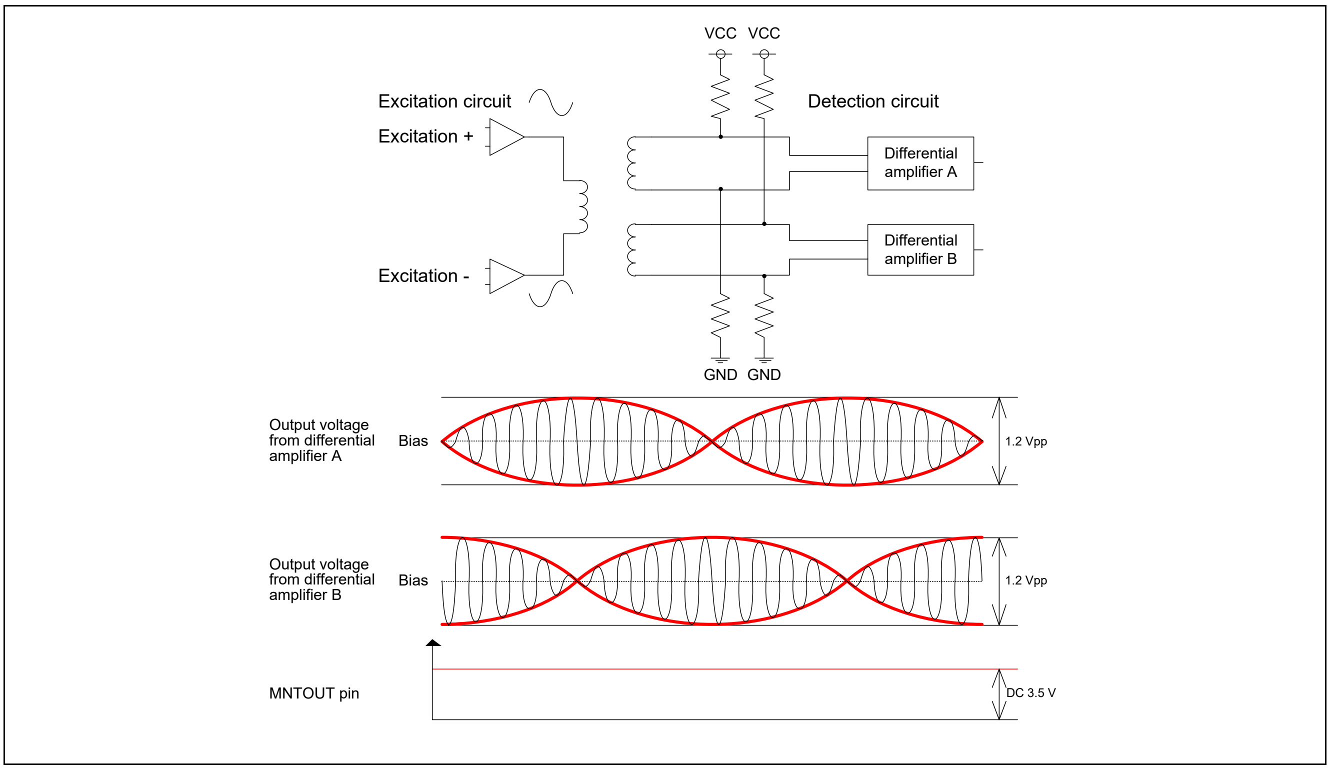


Figure 10.5 Normal State

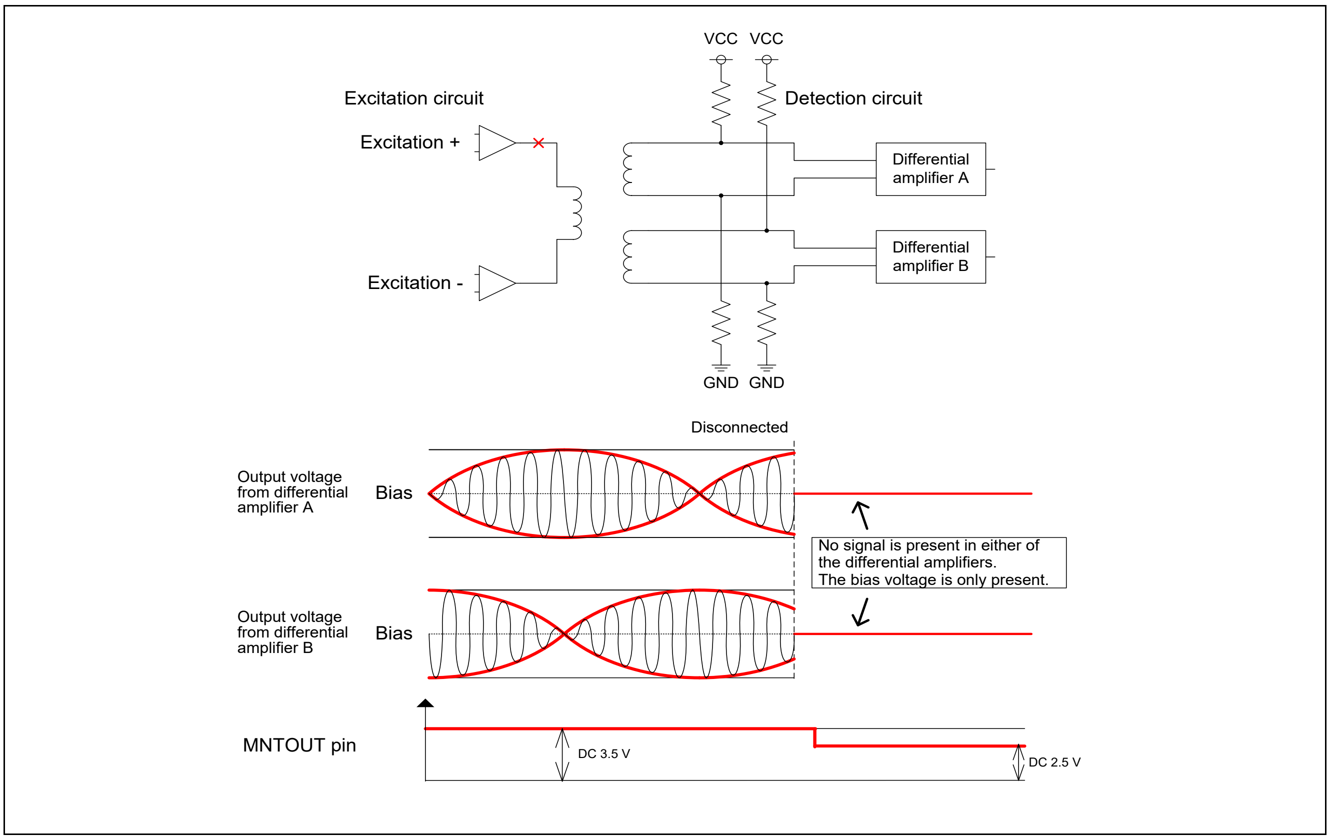

VCC
VCC
Excitation circuit
Detection circuit
Excitation +
Excitation -
Differential amplifier A
Differential amplifier B
GND
GND
Disconnected
Output voltage from differential amplifier A
Bias
No signal is present in either of the differential amplifiers.
The bias voltage is only present.
Output voltage from differential amplifier B
Bias
MNTOUT pin
DC 3.5 V
DC 2.5 V

Figure 10.6 Disconnection on the Excitation Side

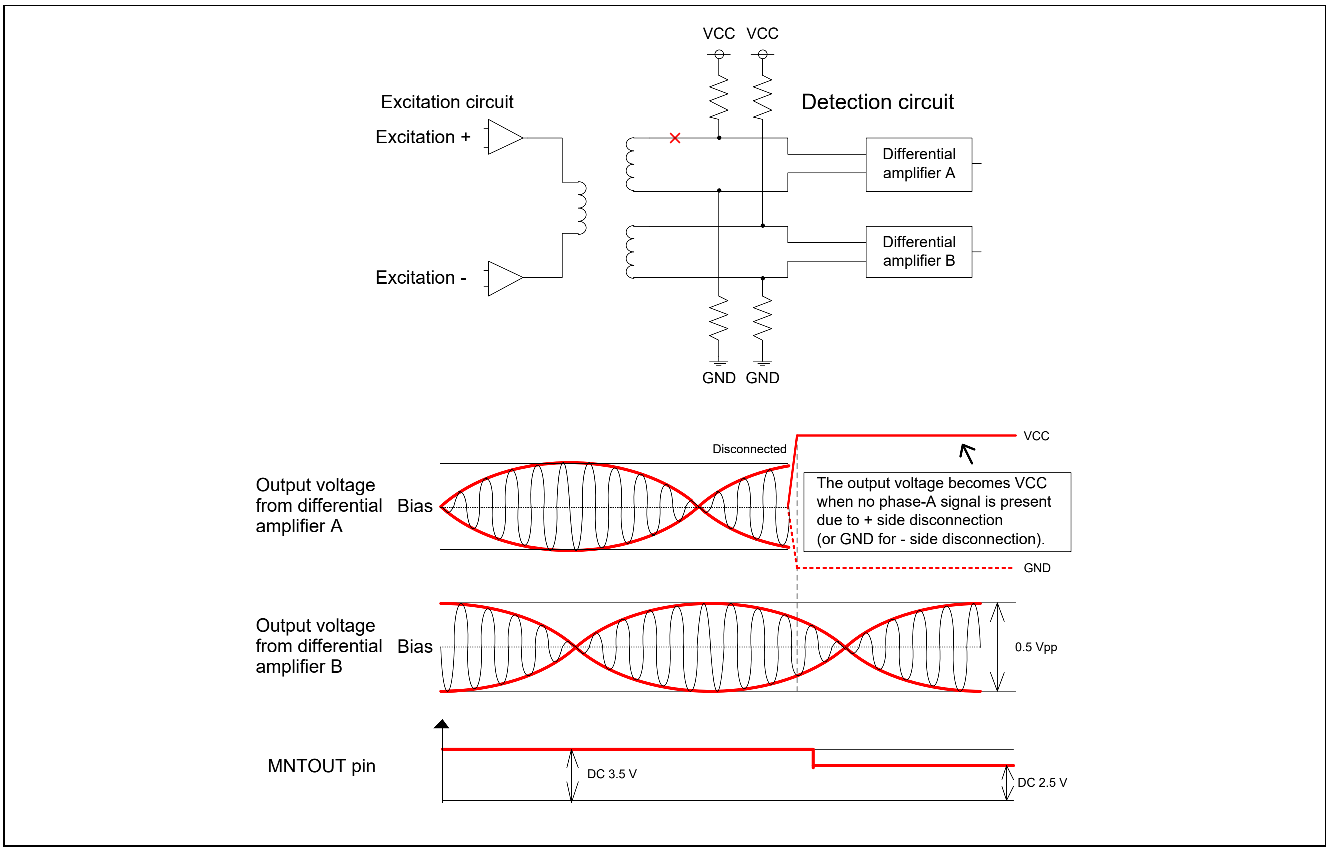

VCC
VCC
Excitation circuit
Detection circuit
Excitation +
Excitation -
Differential amplifier A
Differential amplifier B
GND
GND
Disconnected
VCC
Output voltage from differential amplifier A
Bias
The output voltage becomes VCC when no phase-A signal is present due to + side disconnection
(or GND for - side disconnection).
GND
Output voltage from differential amplifier B
Bias
0.5 Vpp
MNTOUT pin
DC 3.5 V
DC 2.5 V

Figure 10.7 Disconnection on the SIN+ Side

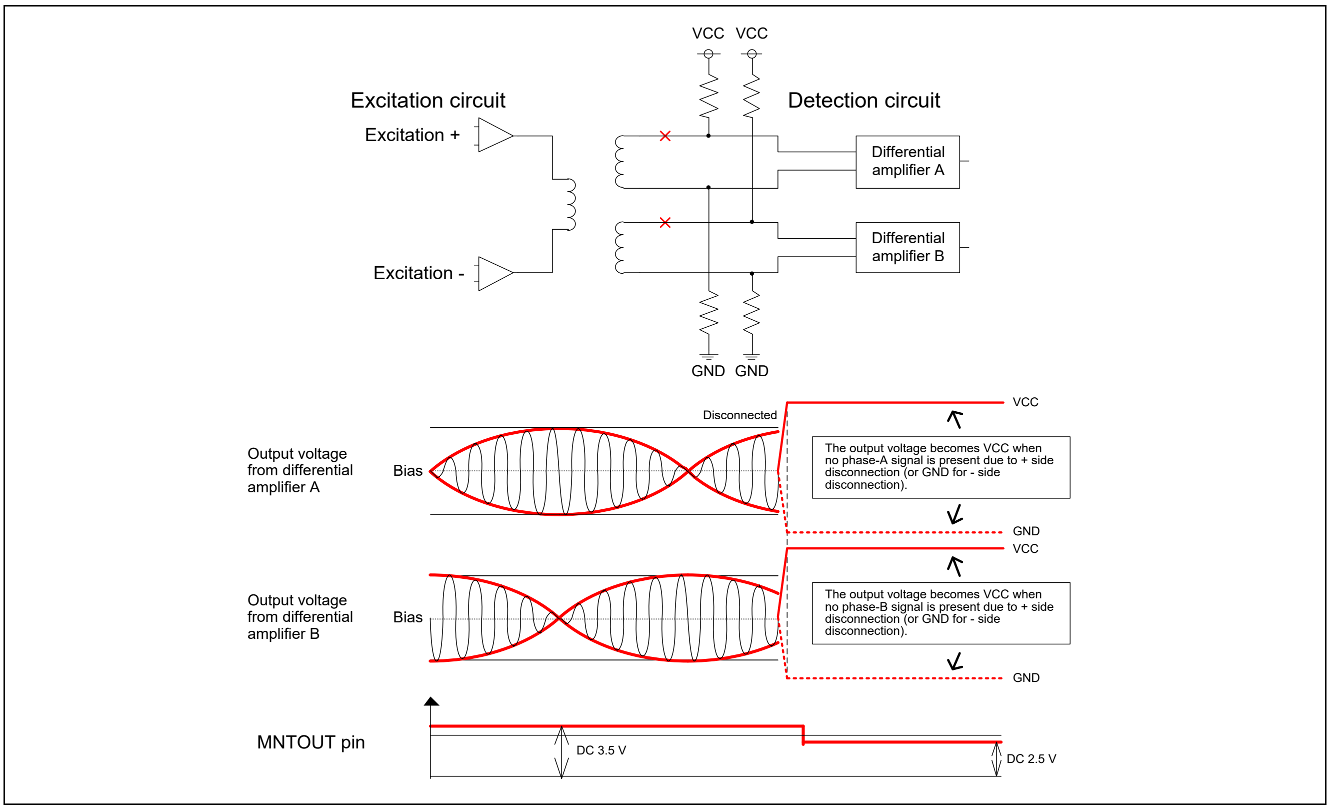

VCC VCC
Excitation circuit
Detection circuit
Excitation +
Excitation -
Differential amplifier A
Differential amplifier B
GND GND
Disconnected
VCC
GND
Output voltage from differential amplifier A
Bias
The output voltage becomes VCC when no phase-A signal is present due to + side disconnection (or GND for - side disconnection).
Output voltage from differential amplifier B
The output voltage becomes VCC when no phase-B signal is present due to + side disconnection (or GND for - side disconnection).
MNTOUT pin
DC 3.5 V
DC 2.5 V

Figure 10.8 Disconnection on the SIN+ and COS+ Sides

Figure 10.9 shows normal operation and Figures 10.10 to 10.13 show cases of the detection of disconnection when the resolver is of the current-detection type.

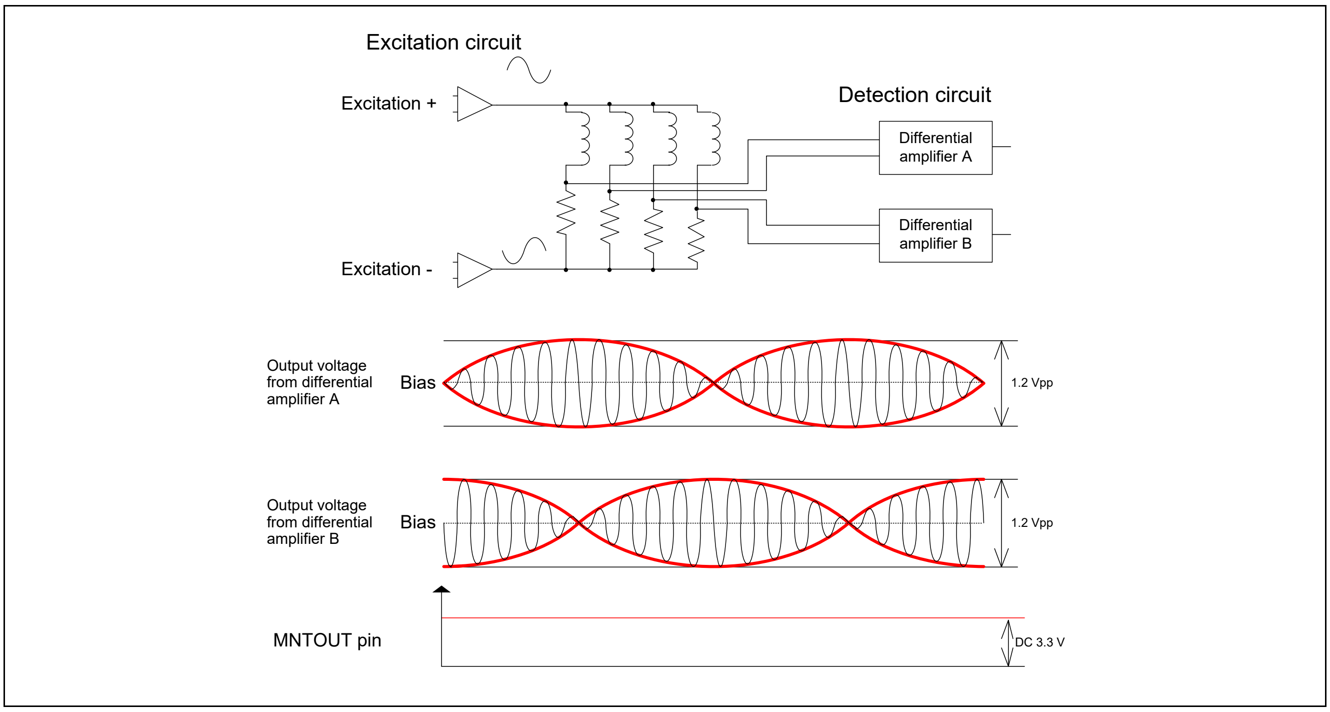


Figure 10.9 Normal State

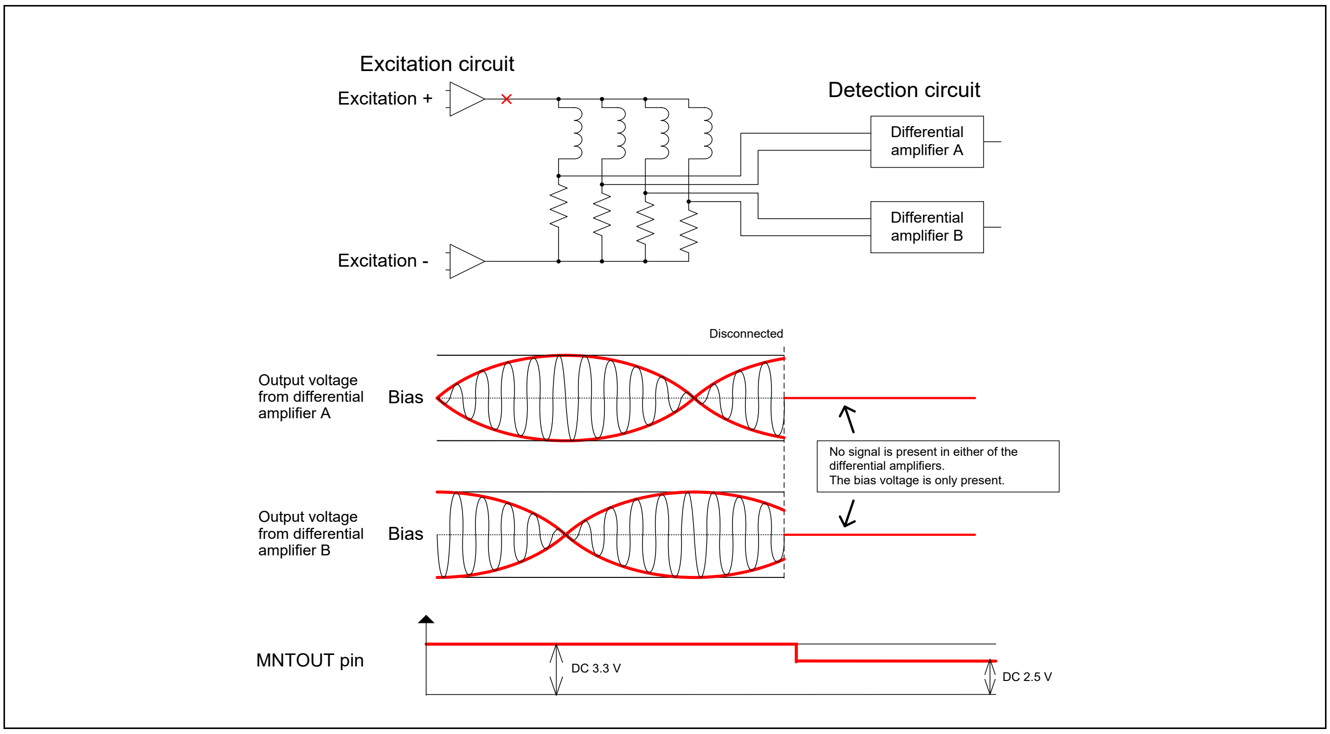

Excitation circuit
Excitation +
Excitation -
Detection circuit
Differential amplifier A
Differential amplifier B
Disconnected
Output voltage from differential amplifier A
Bias
No signal is present in either of the differential amplifiers.
The bias voltage is only present.
Output voltage from differential amplifier B
Bias
MNTOUT pin
DC 3.3 V
DC 2.5 V

Figure 10.10 Disconnection on the Excitation Side

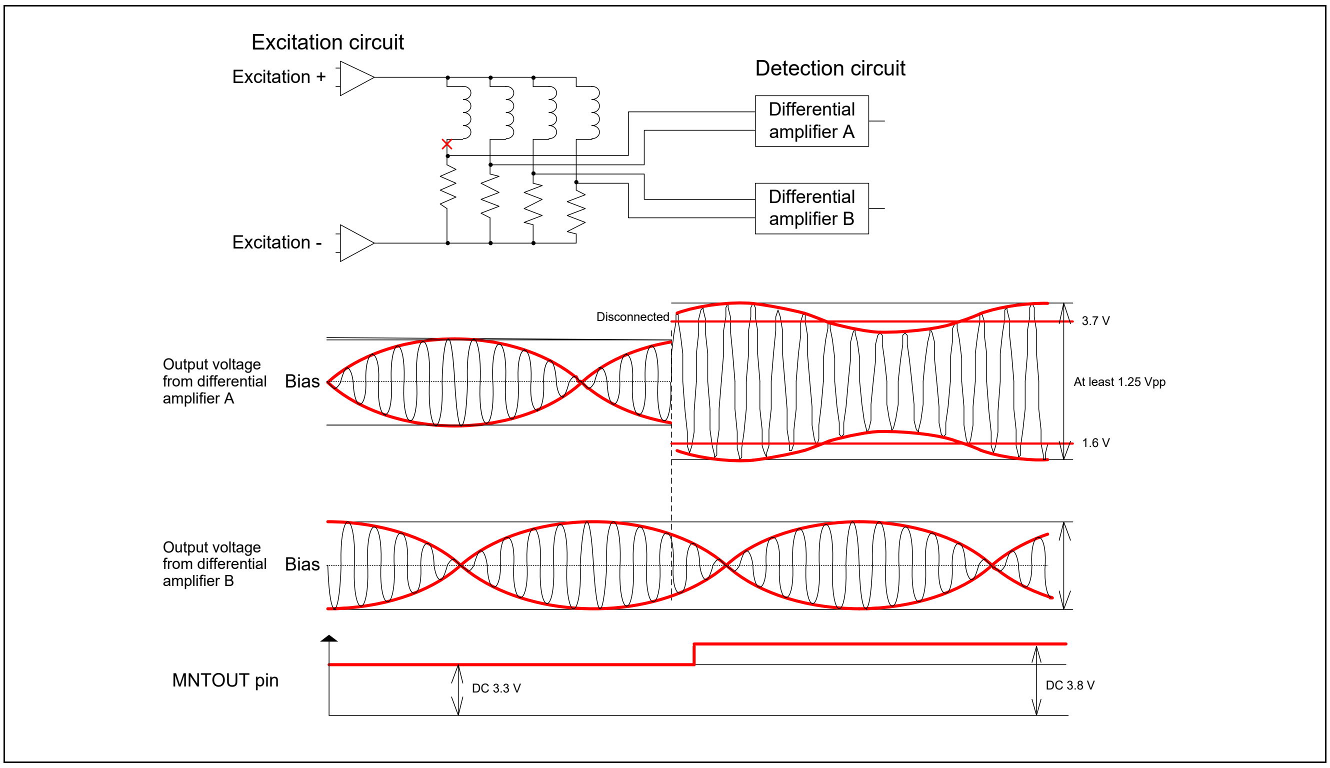


Figure 10.11 Disconnection on the Negative Side of 0 Degrees

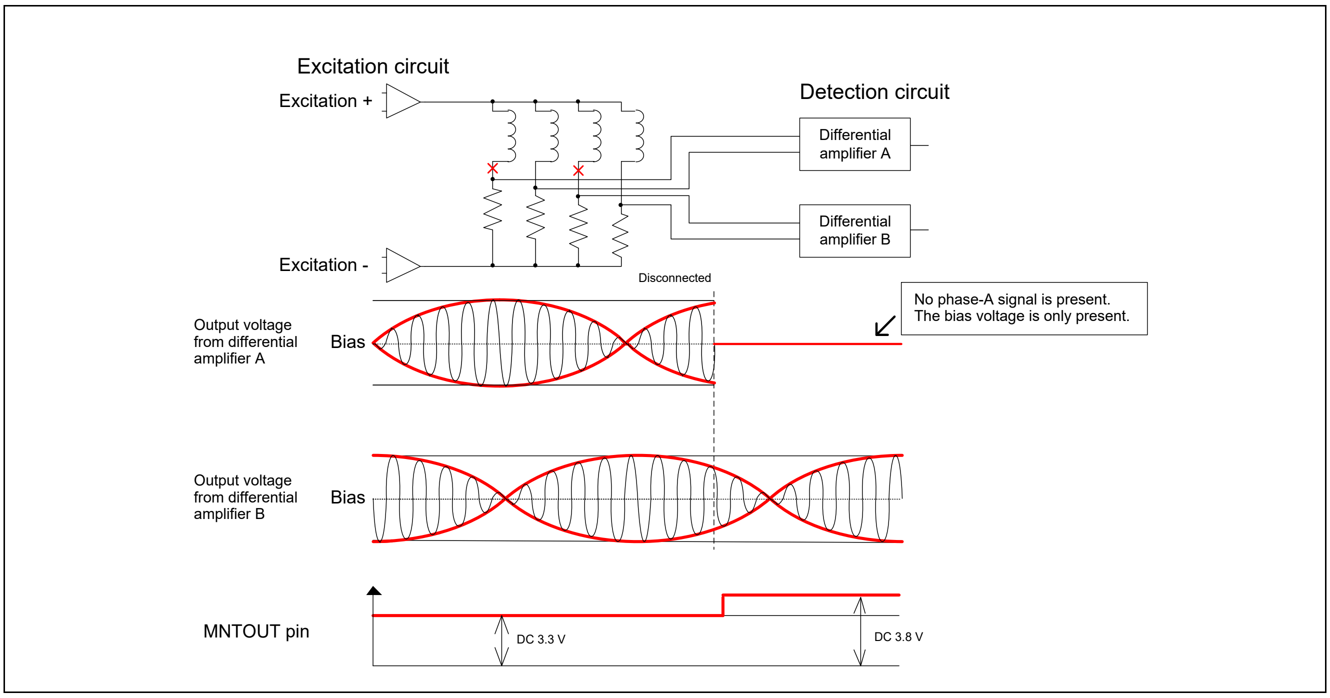


Figure 10.12 Disconnection on the Negative Side of 0 and 180 Degrees

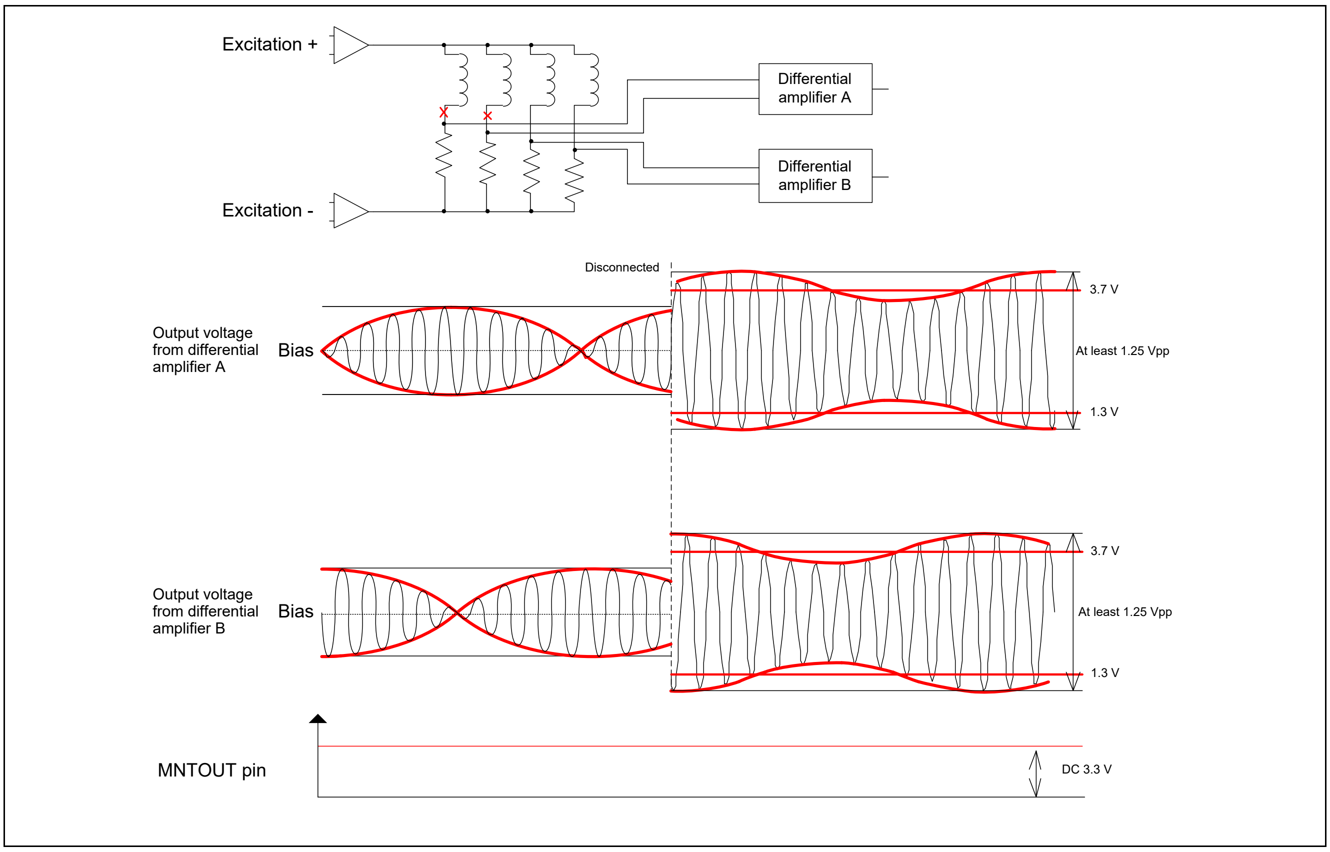


Figure 10.13 Disconnection on the Negative Side of 0 and 90 Degrees

Website and Support

Revision History

Rev.	Date	Description	
		Page	**Summary**
1.00	Jan. 29, 2021	—	First edition issued

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

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. 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.).
7. 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.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

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

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/.