RX24T Group

Using the Driver for Resolver-to-Digital Converter Control

Introduction

This application note describes how to use the driver to control the resolver-to-digital converter IC (RDC).

Target Devices

- RX24T (R5F524TEADFP)
- RX24T (R5F524TAADFM)
- RDCs (RAA3064002GFP and RAA3064003GFP)

Contents

1. Overview 6
1.1 Functions of the Driver 6
1.2 Development Environment 6
1.3 Program Size 6

2. Overall Configuration 7
2.1 System Configuration 7
2.2 RDC Functions 8

3. Driver Functions 9
3.1 Initial Settings for the On-Chip Peripheral Modules of the MCU 9
3.1.1 Specifying System Information 10
3.1.2 Initial Settings for a Peripheral Module 11
3.1.3 Starting the Peripheral Module 11
3.2 RDC Settings 11
3.3 Output of the Excitation Signal 13
3.4 Output of the Phase Adjustment Signals for the Resolver Signals 15
3.5 Output of the Angle Error Correction Signal 17
3.6 Input of the Angle Signal 20
3.7 Output of the RDC Operating Clock 22
3.8 Automatic Calibration of Errors 23
3.8.1 Functions Used to Adjust Parameters 23
3.8.2 Adjustment of Gain and Phase of Resolver Signals 24
3.8.3 Adjustment of the Angle Error Correction Signal 26
3.9 Communications between the RDC and MCU 27
3.9.1 Setting Peripheral Modules 27
3.10 Detection of Disconnection from the Resolver Sensor 28
3.10.1 Functions Used for Detecting Disconnection 28

4. Software Configuration 29
4.1 Folder and File Configuration 29

5. Settings for Peripheral Modules 30
5.1 List of Macro-Defined Names of Driver Facilities 30
5.2 List of Macro-Defined Names of Peripheral Modules 30
5.3 List of Possible Combinations of Peripheral Modules and Driver Facilities 31
5.4 List of Port Pins Usable for Driver Facilities 32
5.4.1 TMR 32
5.4.2 MTU3 33
5.4.3 RSPI 34
5.4.4 SCI 35
5.5 Setting Interrupts for Driver Facilities 36
5.5.1 Availability of Interrupt Settings 36
5.5.2 Interrupt Sources for the Output of the Excitation Signals 36
5.5.3 Input Capture Interrupt Sources 40
5.5.4 Interrupt Sources for Updating the Angle Error Correction Duty Cycle 40
5.5.5 Interrupt Sources for Communications with the RDC 41
5.6 Capture Trigger Settings for Driver Facilities 41

6. APIs 42
6.1 List of API Functions 42
6.2 Descriptions of API Functions 46
6.2.1 API Function for Initial Settings of an On-Chip Peripheral Module of the MCU 46
6.2.2 API Function for Specifying System Information 46
6.2.3 API Function for Acquiring the RDC Driver Setting Information 46
6.2.4 API Function for Setting the Pointer to a User Function 47
6.2.5 API Function for Controlling Synchronous Starting of the MTU3 Timer Channels 47
6.2.6 API Function for Acquiring the RDC Driver Version Information 47
6.2.7 API Function for Starting the Output of the Angle Error Correction Signal 48
6.2.8 API Function for Stopping the Output of the Angle Error Correction Signal 48
6.2.9 API Function for Updating the Duty Cycle of the Angle Error Correction Signal 48
6.2.10 API Function for Synchronously Starting the Angle Error Correction Signal 48
6.2.11 API Function for Acquiring the Output State of the Angle Error Correction Signal 49
6.2.12 API Function for Starting the Input Capture Timer 49
6.2.13 API Function for Acquiring the Input Capture Value 49
6.2.14 API Function for Setting the Input Capture Timer Start Timing 49
6.2.15 API Function for Reading the Trigger Information for the Input Capture Interrupt 49
6.2.16 API Function for Reading the Resolver Position Count (Trigger: Falling Edge) 50
6.2.17 API Function for Acquiring the Resolver Position Difference Count (Trigger: Falling Edge) 50
6.2.18 API Function for Reading the Resolver Position Count (Trigger: Rising Edge) 50
6.2.19 API Function for Reading the Resolver Position Difference Count (Trigger: Rising Edge) 50

6.2.20 API Function for Starting the Output of the Excitation Signal 51
6.2.21 API Function for Stopping the Output of the Excitation Signal 51
6.2.22 API Function for Setting the Excitation Signal Output Start Timing 51
6.2.23 API Function for Counting the Wait Time 51
6.2.24 API Function for Starting the Output of the Phase Adjustment Signals 51
6.2.25 API Function for Stopping the Output of the Phase Adjustment Signals 52
6.2.26 API Function for Setting the Phase Adjustment Signal Duty Cycle in the Buffer 52
6.2.27 API Function for Setting the Phase Adjustment Signal Duty Cycle in the Register 52
6.2.28 API Function for Reading the Phase Adjustment Signal Duty Cycle from the Buffer 52
6.2.29 API Function for Initializing Variables for RDC Communications 53
6.2.30 API Function for Executing RDC Initialization Sequence 53
6.2.31 API Function for Handling RDC Communications 53
6.2.32 API Function for Writing to an RDC Register 53
6.2.33 API Function for Reading from an RDC Register 54
6.2.34 API Function for Acquiring the RDC Register Access State 54
6.2.35 API Function for Reading Data from the RDC Register Buffer 54
6.2.36 API Function for Writing Data to the RDC Register Buffer 54
6.2.37 API Function for Reporting Errors in RDC Communications 54
6.2.38 API Function for a Reception Interrupt in RDC Communications 55
6.2.39 API Function for a Transmission Interrupt in RDC Communications 55
6.2.40 API Function for an Error Interrupt in RDC Communications 55
6.2.41 API Function for an Idle Interrupt in RDC Communications 55
6.2.42 API Function for Starting RDC Alarm Cancellation 55
6.2.43 API Function for Controlling the RDC Alarm Cancellation Sequence 55
6.2.44 API Function for Adjusting the Gain and Phase of the Resolver Signals 56
6.2.45 API Function for Adjusting the Angle Error Correction Signal 56
6.2.46 API Function for Setting the Pointer to the User-Created Callback Function 56
6.2.47 API Function for Acquiring the A/D Conversion State 56
6.2.48 API Function for Detecting Disconnection 56
6.3 Structures 57
6.3.1 Structure for R_RSLV_CreatePeripheral 57
6.3.2 Structure for R_RSLV_SetSystemInfo 60
6.3.3 Structure for R_RSLV_GetRdcDrvSettingInfo 60
6.3.4 Structure for R_RSLV_ADJST_GainPhase 61
6.3.5 Structure for R_RSLV_ADJST_Carrier 63
6.3.6 Structure for R_RSLV_ADJST_SetPtrFunc 64
6.3.7 Structure for R_RSLV_DiscDetection_Seq 64

7. Examples of Implementing API Functions 65
7.1 Overview 65
7.2 Initialization 66

7.2.1 Initialization Procedure 66
7.2.2 Details of Initialization Processing 67
7.2.3 Sample Code 68
7.3 Main Loop 70
7.4 Output of the Excitation Signal 71
7.4.1 Example of Using API Functions 71
7.4.2 Details of the Output of the Excitation Signal 72
7.4.3 Sample Code 73
7.5 Output of the Phase Adjustment Signals 74
7.5.1 Example of Using API Functions 74
7.5.2 Sample Code 75
7.6 Output of the Angle Error Correction Signal 76
7.6.1 Example of Using API Functions 76
7.6.2 Details of the Output of the Angle Error Correction Signal 77
7.6.3 Sample Code 78
7.7 Input of Angle Signal 79
7.7.1 Example of Using API Functions 79
7.7.2 Sample Code 80
7.8 Automatic Adjustment of the Gain and Phase 81
7.8.1 Example of Using API Functions 81
7.8.2 Details of Gain and Phase Adjustment 82
7.8.3 Sample Code 84
7.9 Automatic Adjustment of the Angle Error Correction Signal 87
7.9.1 Example of Using API Functions 87
7.9.2 Details of Angle Error Correction Signal Adjustment 88
7.9.3 Sample Code 92
7.10 Communications with RDC 96
7.10.1 Example of Using API Functions 96
7.10.2 Sample Code 97
7.11 Detection of Disconnection from the Resolver Sensor 98
7.11.1 Example of Using API Functions 98
7.11.2 Sample Code 100
7.12 Cancelling an Alarm 103
7.12.1 Example of Using API Functions 103
7.12.2 Sample Code 104

8. Notes 105
8.1 Initial Setting Procedure 105
8.2 Specifying System Information 105
8.3 Assigning Multiple Driver Facilities to a Single Peripheral Module 105
8.4 Assigning Multiple Peripheral Modules to a Single Driver Facility 105

8.5 Initializing Variables for Communications with the RDC 105
8.6 Specifying Peripheral Modules for Phase Adjustment Signals 105
8.7 Setting Timer Start Timing 105

9. Troubleshooting 106
9.1 Counter Value Errors 107
9.2 Rotation Direction Errors 108
9.3 Angle Errors 109
9.4 Detection of Disconnection 110

Website and Support 115

Revision History 116

1. Overview

1.1 Functions of the Driver

This driver has the following functions.

- Initial settings for the on-chip peripheral modules of the RX24T
- RDC settings
- Output of the excitation reference signal
- Output of the phase adjustment signals
- Output of the angle error correction signal
- Input of the angle signal
- Output of the RDC operating clock
- Automatic calibration of errors
- Communications between the RDC and MCU
- Detection of disconnection from the resolver sensor
- Deassertion of the ALARM# signal

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
CS+: V8.04.00 e^2 studio: V2020-10	CC-RX V3.01.00

1.3 Program Size

Table 1.2 shows the program size of this driver.

Table 1.2 Program Size

ROM Size	RAM Size
45527 bytes	1206 bytes

2. Overall Configuration

2.1 System Configuration

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

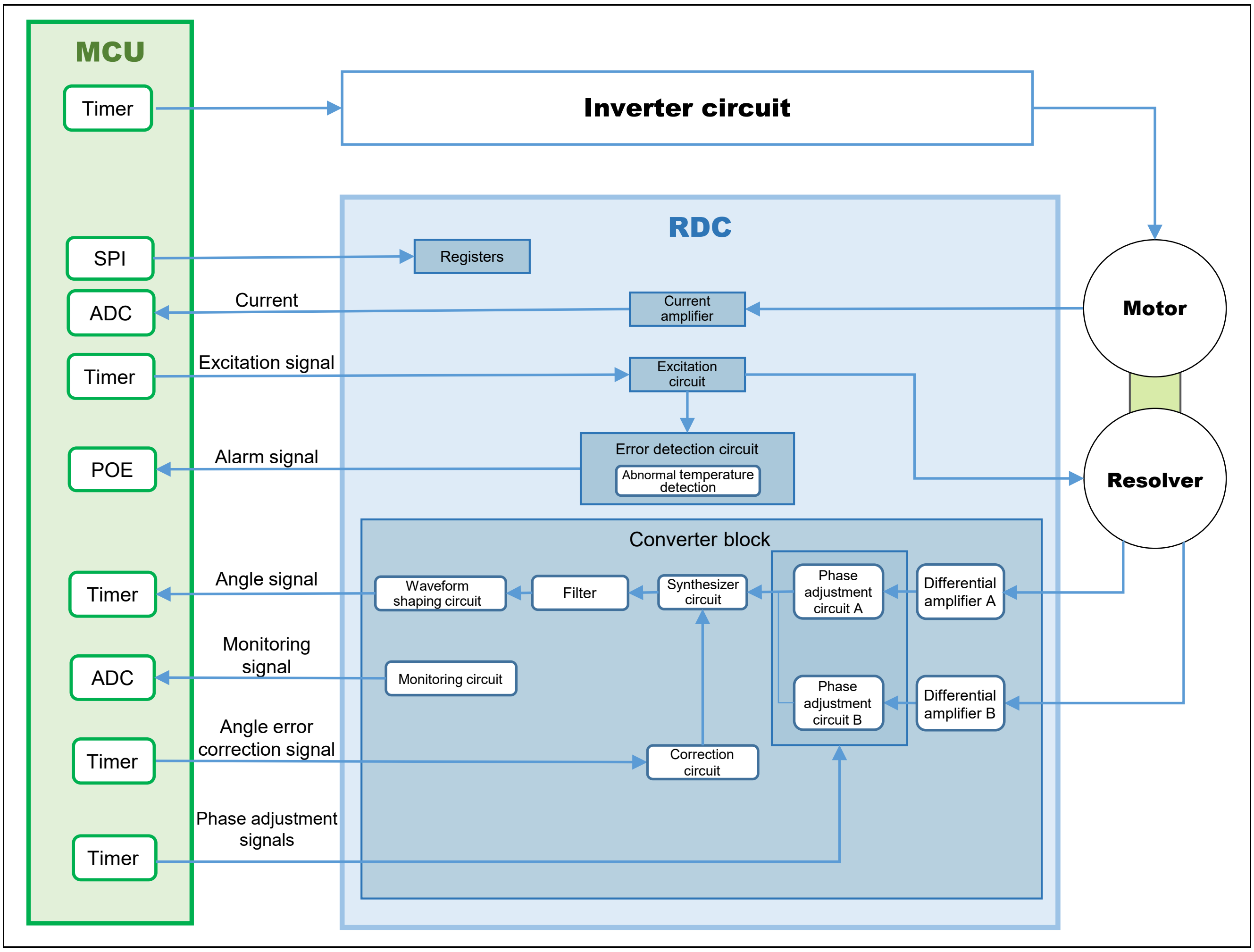


Figure 2.1 Configuration of the System Incorporating RDC and MCU

2.2 RDC Functions

The RDC incorporates an excitation circuit to excite the resolver sensor and a converter 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 changes the RDC settings to adjust the amplitudes of the two-phase signals of the resolver sensor to the same level.

The phase adjustment function outputs correction signals for phase adjustment from the MCU to the RDC to adjust 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. Driver Functions

This section describes the functions of the driver software.

3.1 Initial Settings for the On-Chip Peripheral Modules of the MCU

The RDC driver allows the user to specify which peripheral modules implement individual driver functions and to make settings for those modules. To set up peripheral modules, call the API function for initial settings provided in the driver with the necessary arguments specified.

Follow the procedure below to make the initial settings of the peripheral module registers.

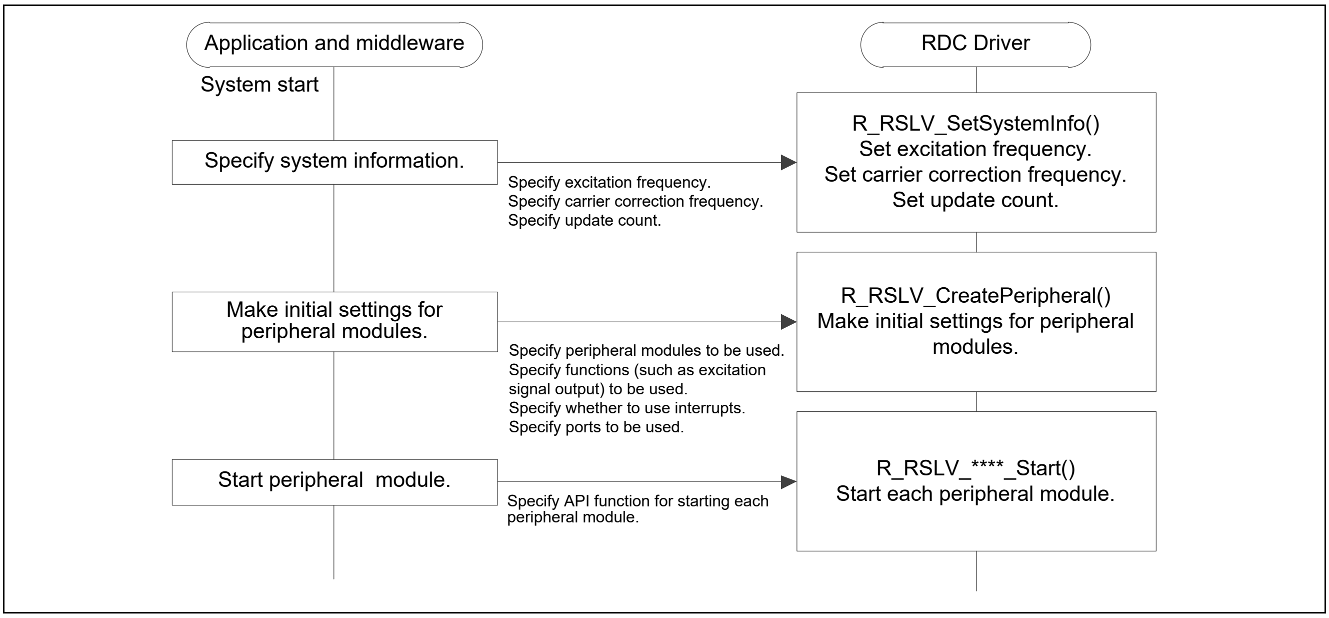


Figure 3.1 Initial Setting Sequence

3.1.1 Specifying System Information

Specify the system information, such as the excitation frequency, the angle error correction frequency, and the number of updates of the angle error correction signal, and then execute the following API function. For details of the system information settings, see section 6.3.2, Structure for R_RSLV_SetSystemInfo.

API function: R_RSLV_SetSystemInfo (ST_SYSTEM_PARAM *rdc_sys_param)

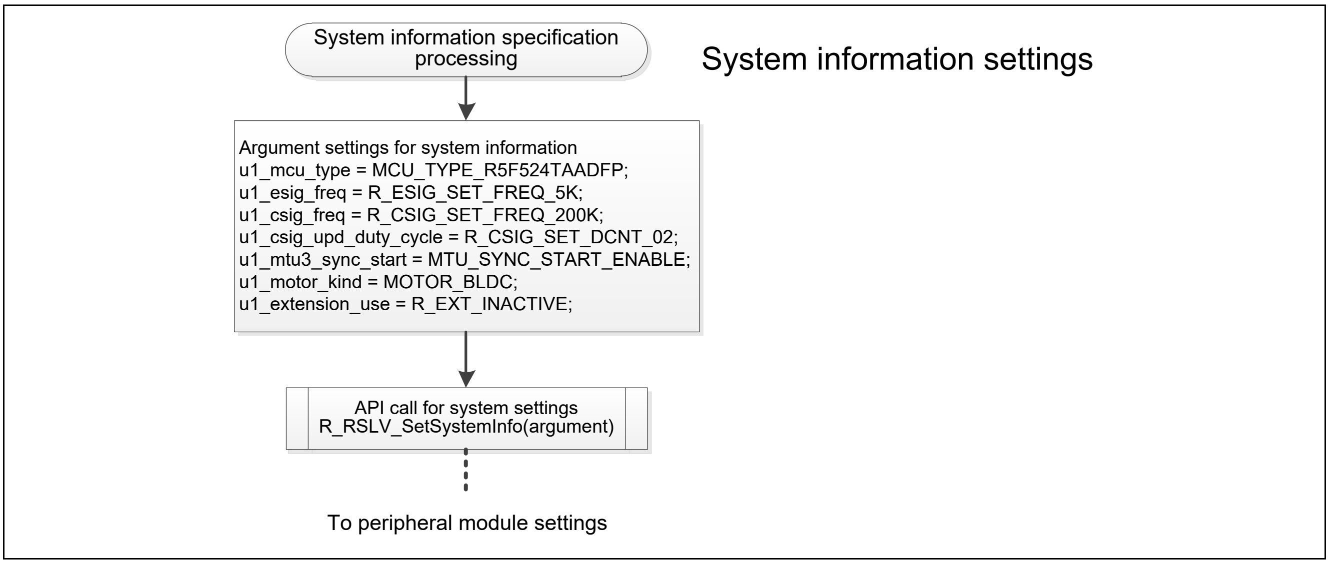


Figure 3.2 Processing for Specifying System Information

3.1.2 Initial Settings for a Peripheral Module

Execute the following API function to make initial settings for the target peripheral module, such as the type of peripheral module and the hardware facilities to be used. For details of the initial settings of peripheral modules, see section, 6.3.1, Structure for R_RSLV_CreatePeripheral.

API function: R_RSLV_CreatePeripheral (ST_INIT_REG_PARAM *rdc_init_param)

This API function makes settings of the peripheral module facilities such as the timer counters to be used and control of the duty cycles for the angle error correction signal and phase adjustment signals according to the information passed through the arguments of this API function and the API function described in section 3.1.1, Specifying System Information.

3.1.3 Starting the Peripheral Module

A specific API function is provided to start each peripheral module. For details, see section 6.1, List of API Functions.

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.

(1) Writing data to an RDC register

Call the API function for starting writing to an RDC register to start the process of writing. The user can check the state of writing through reference to the *write_status argument of this API function. The following shows the flow of processing for writing.

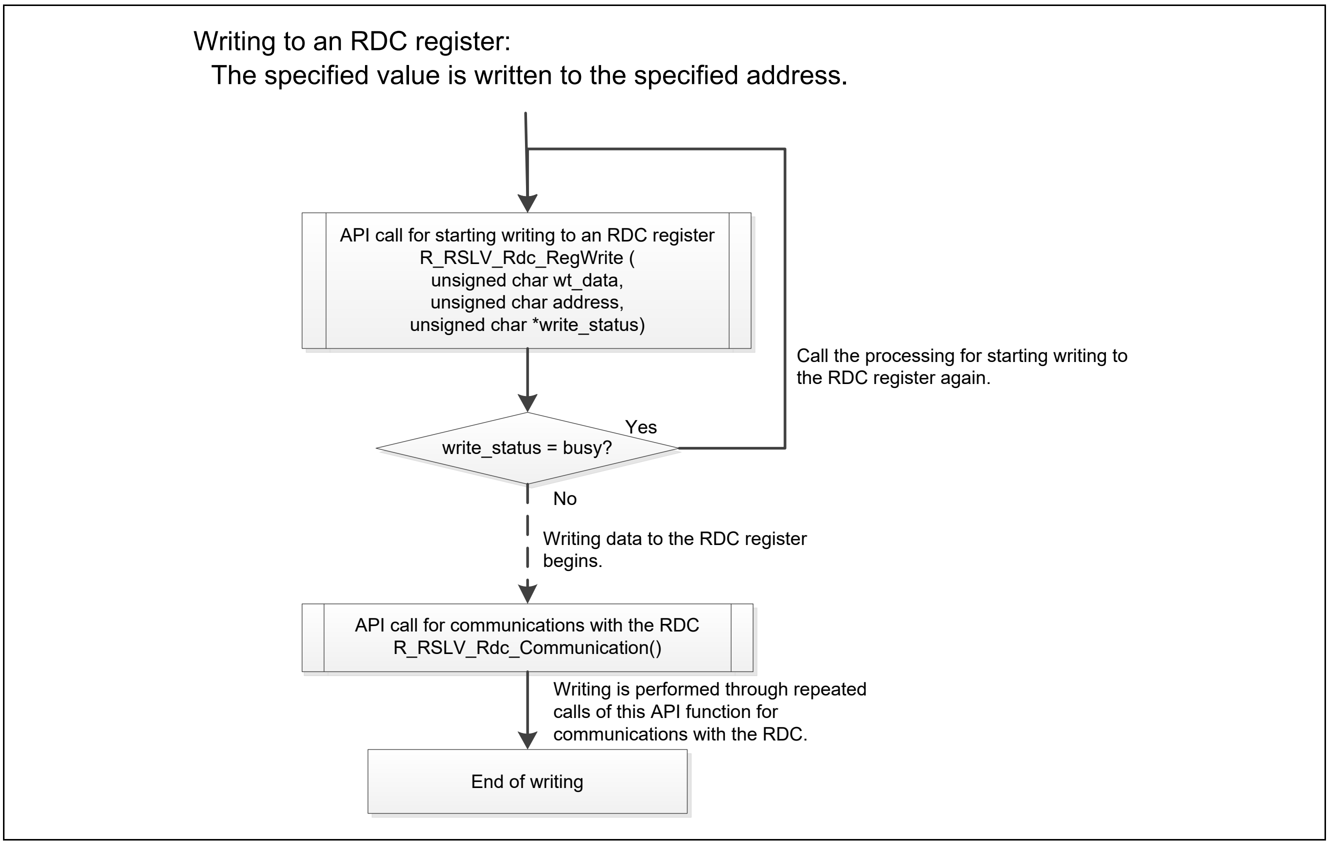


Figure 3.3 Processing for Writing to an RDC Register

(2) Reading data from the RDC register

Call the API function for starting reading from an RDC register to start the process of reading. The following shows the flow of processing for reading.

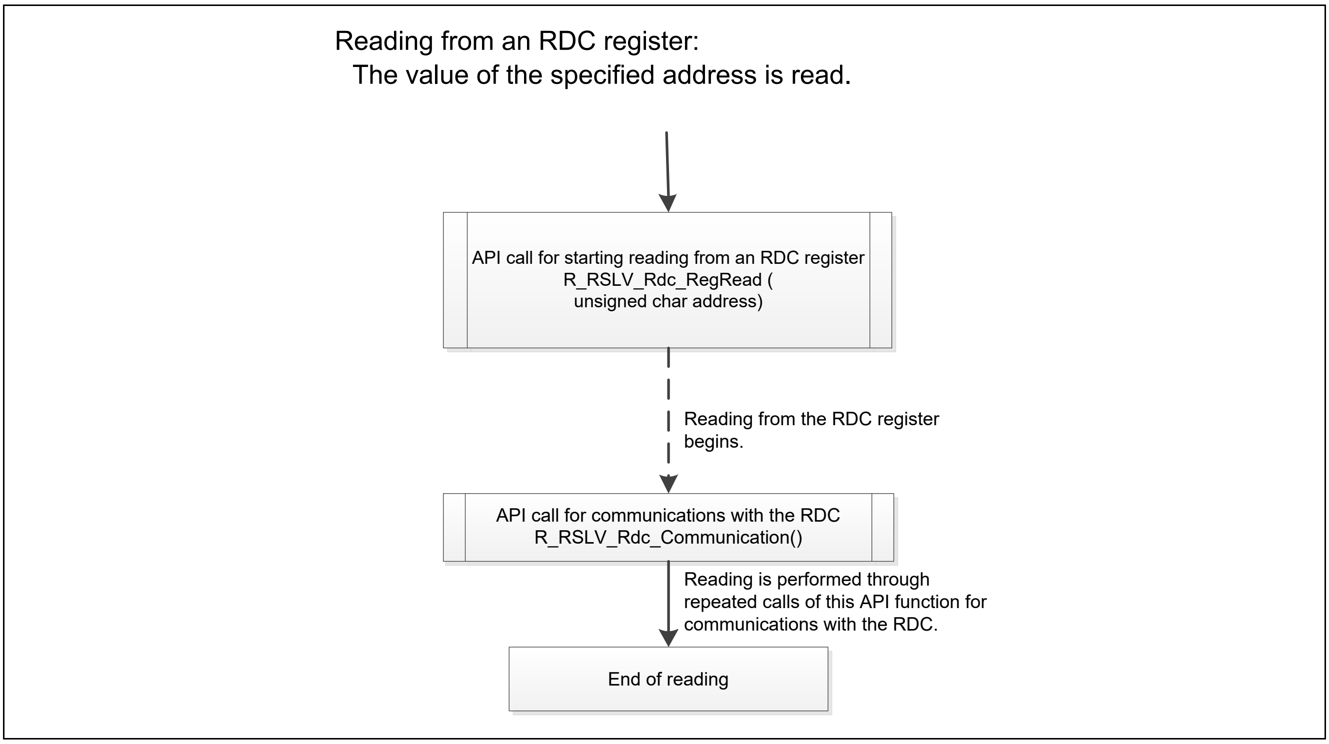


Figure 3.4 Processing for Reading from an RDC Register

3.3 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 a signal for synthesis, which 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.

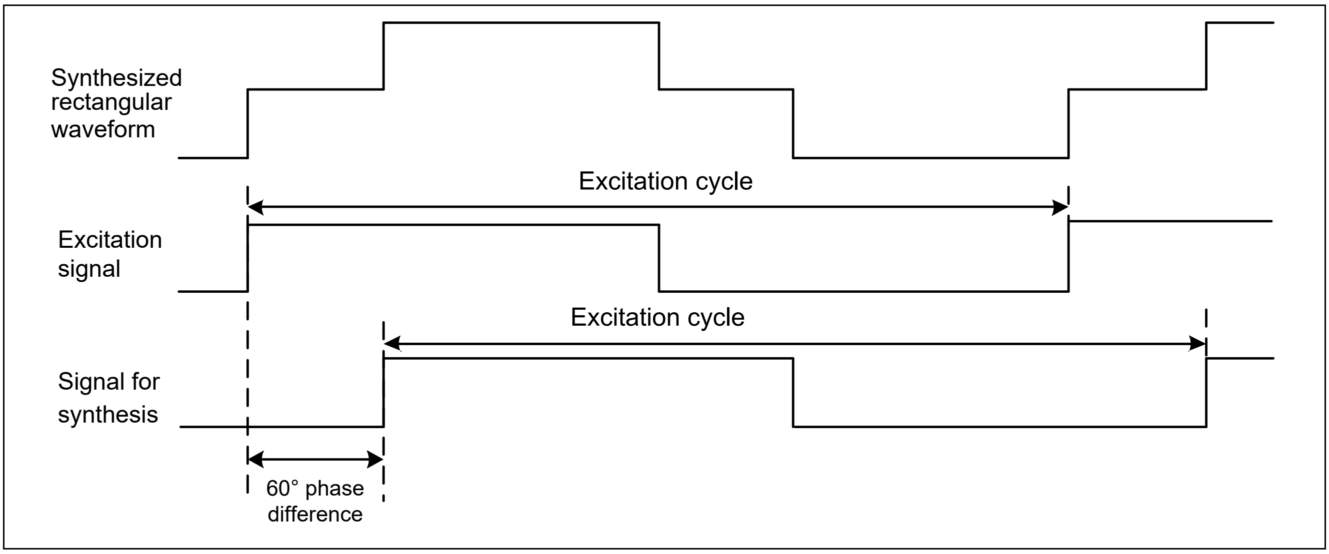


Figure 3.5 Synthesized Rectangular Wave Signal

(1) Procedure for setting output of the excitation signal

Follow procedure below to output the excitation signal. For details of the initial settings of the peripheral module in this figure, see section 6.3.1, Structure for R_RSLV_CreatePeripheral.

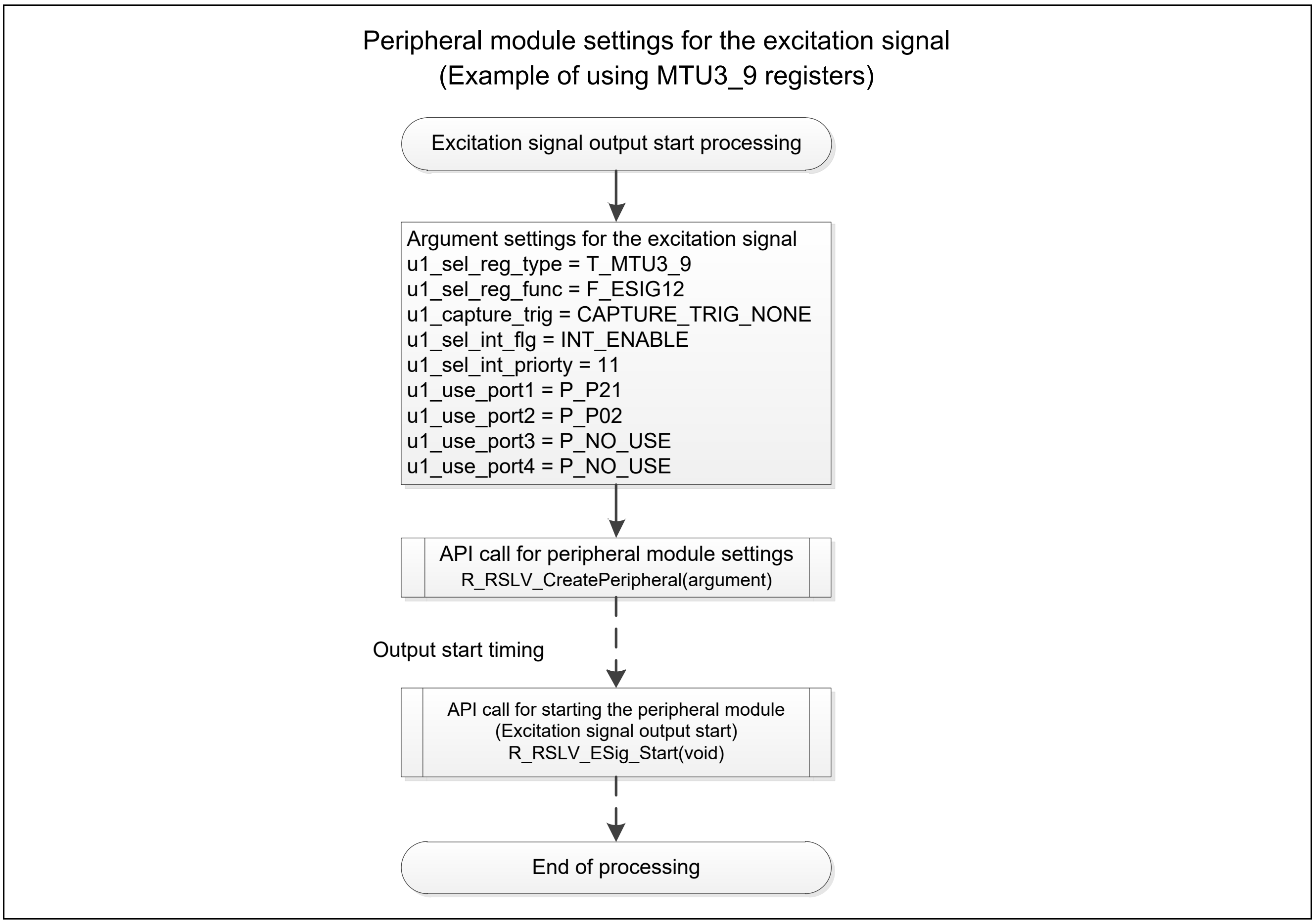


Figure 3.6 Processing for Output of the Excitation Signal

(2) Interrupt

In the excitation signal output processing, interrupts can be generated at intervals of the excitation signal output. In synchronization with this interrupt, the processing for outputting the angle error correction signal and the processing for updating the duty cycle of the angle error correction signal are activated. When a single-phase excitation signal is used, interrupts are generated on the rising edges of the excitation signal. When a synthesized signal is used, interrupts are generated with a phase delay of 30 degrees from the rising edges of the reference excitation signal (excitation signal 1 in Figure 3.5, Synthesized Rectangular Wave Signal).

3.4 Output of the Phase Adjustment Signals for the Resolver Signals

The RDC converts the two-phase signals output from the resolver sensor into to an angle signal, and then outputs the converted angle signal to the MCU. However, 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.

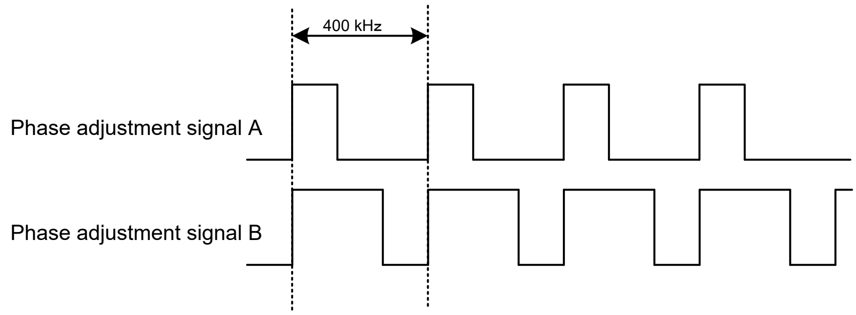


Figure 3.7 Example of Phase Adjustment Signals

(1) Procedure for setting output of the phase adjustment signals

Follow the procedure below to output the phase adjustment signals. For details of the initial settings of the peripheral modules in this figure, see section 6.3.1, Structure for R_RSLV_CreatePeripheral.

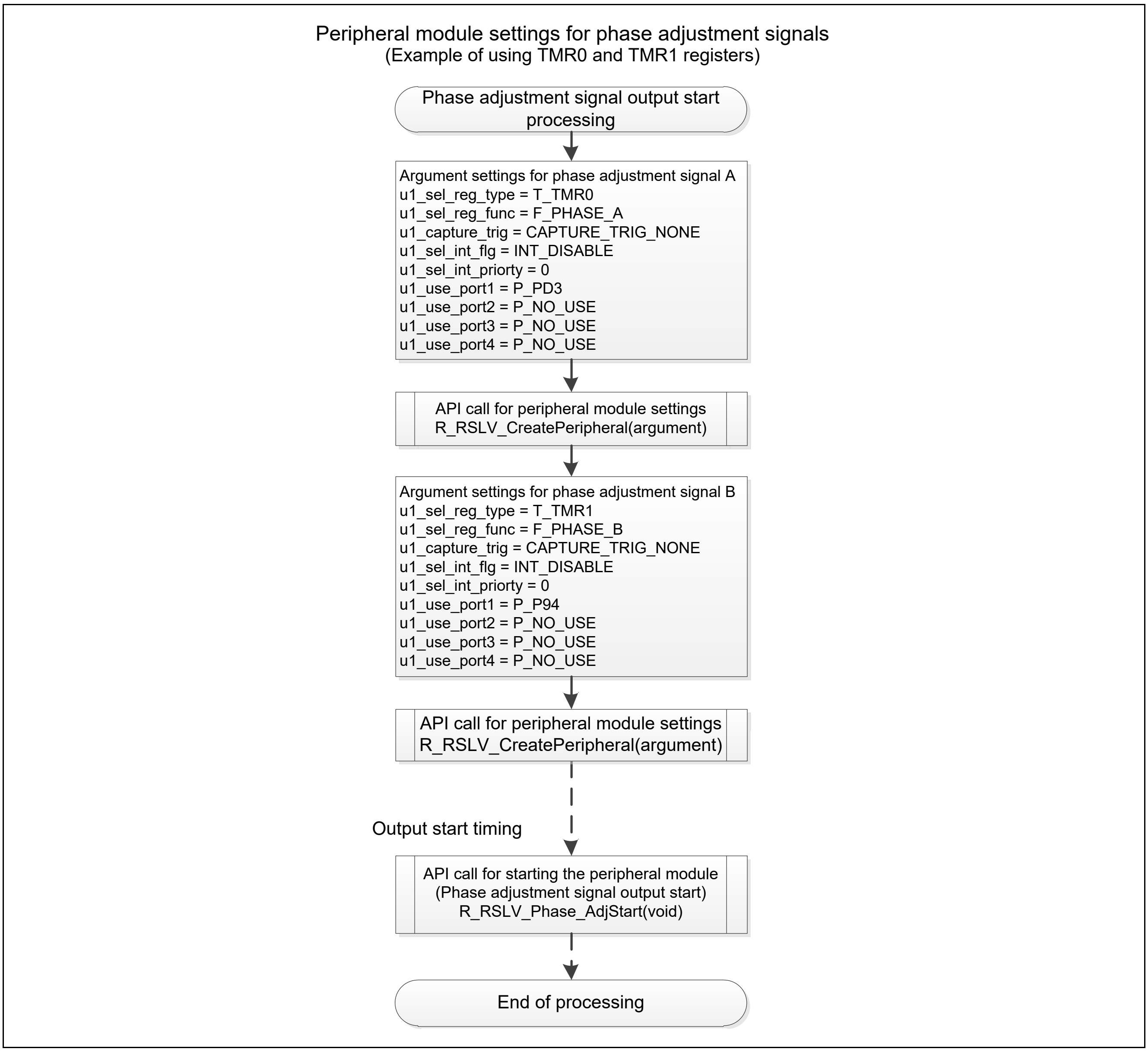


Figure 3.8 Processing for Output of the Phase Adjustment Signals

(2) Interrupt

No interrupt is generated for output of the phase adjustment signals.

(3) Timer settings

The frequency of the phase adjustment signals is always 400 kHz.

3.5 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.

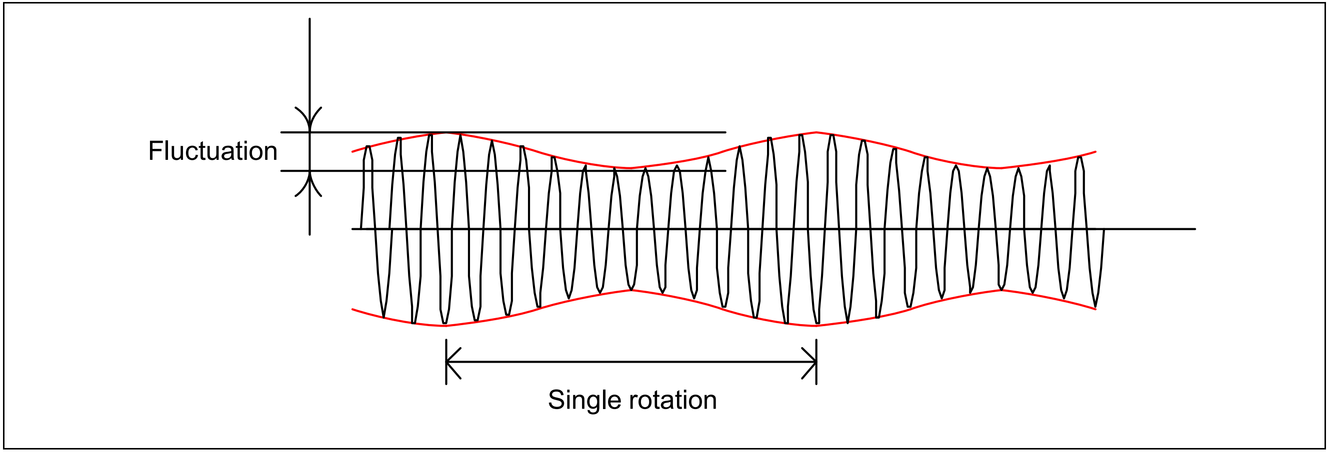


Figure 3.9 Fluctuation of Amplitude (RDC Internal Signal)

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.

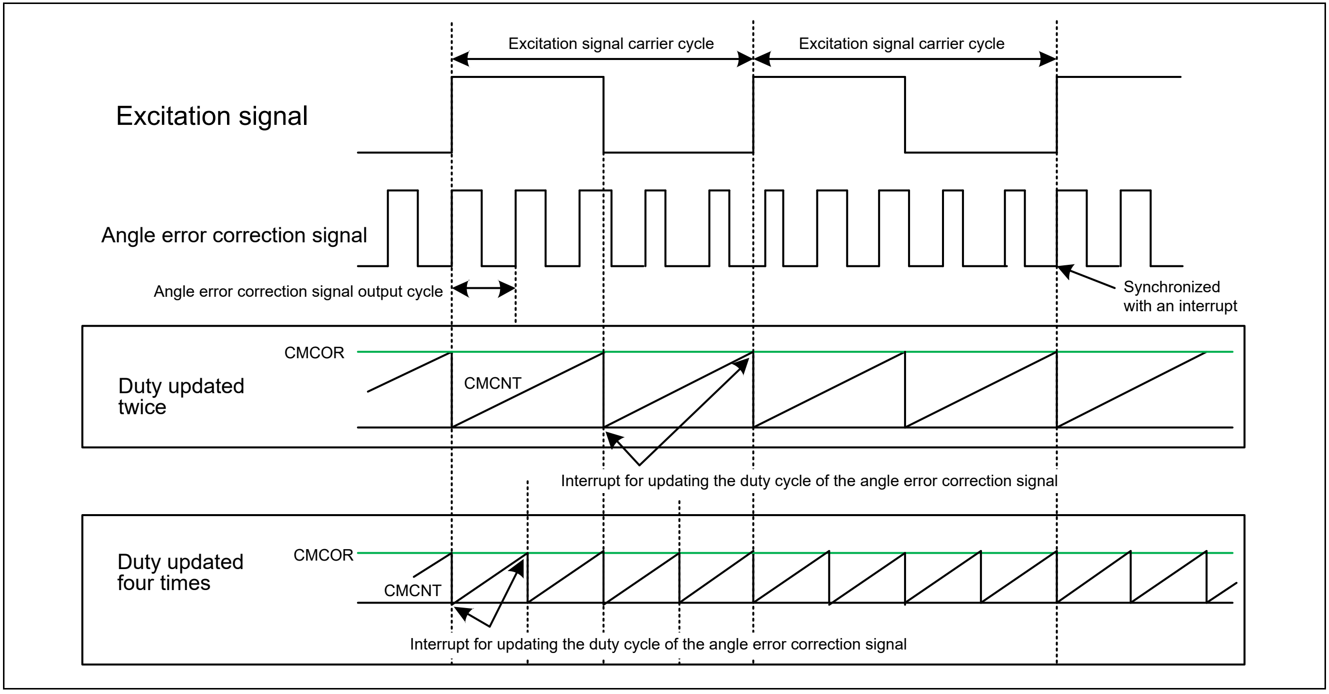


Figure 3.10 Output of the Angle Error Correction Signal

The duty cycle of the angle error correction signal (PWM signal) is changed using a duty cycle updating interrupt. Figure 3.10 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.

(1) Procedure for setting output of the angle error correction signal

Follow the procedure below to output the angle error correction signal. For details of the initial settings of the peripheral module in this figure, see section 6.3.1, Structure for R_RSLV_CreatePeripheral.

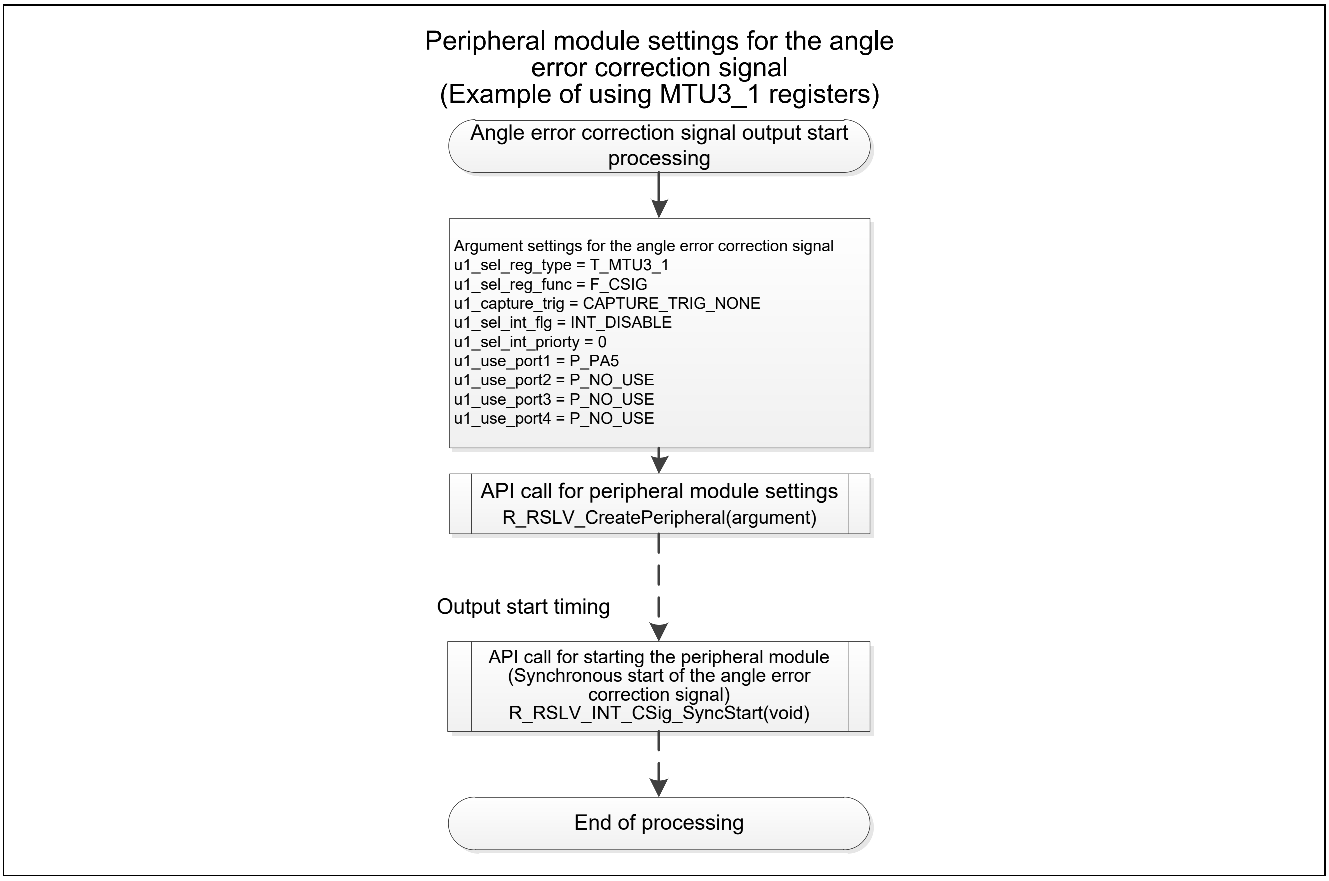


Figure 3.11 Processing for Output of the Angle Error Correction Signal

The following shows the procedure for setting the duty cycle updating interrupt for the angle error correction signal. For details of the initial settings of the peripheral module in this figure, see section 6.3.1, Structure for R_RSLV_CreatePeripheral.

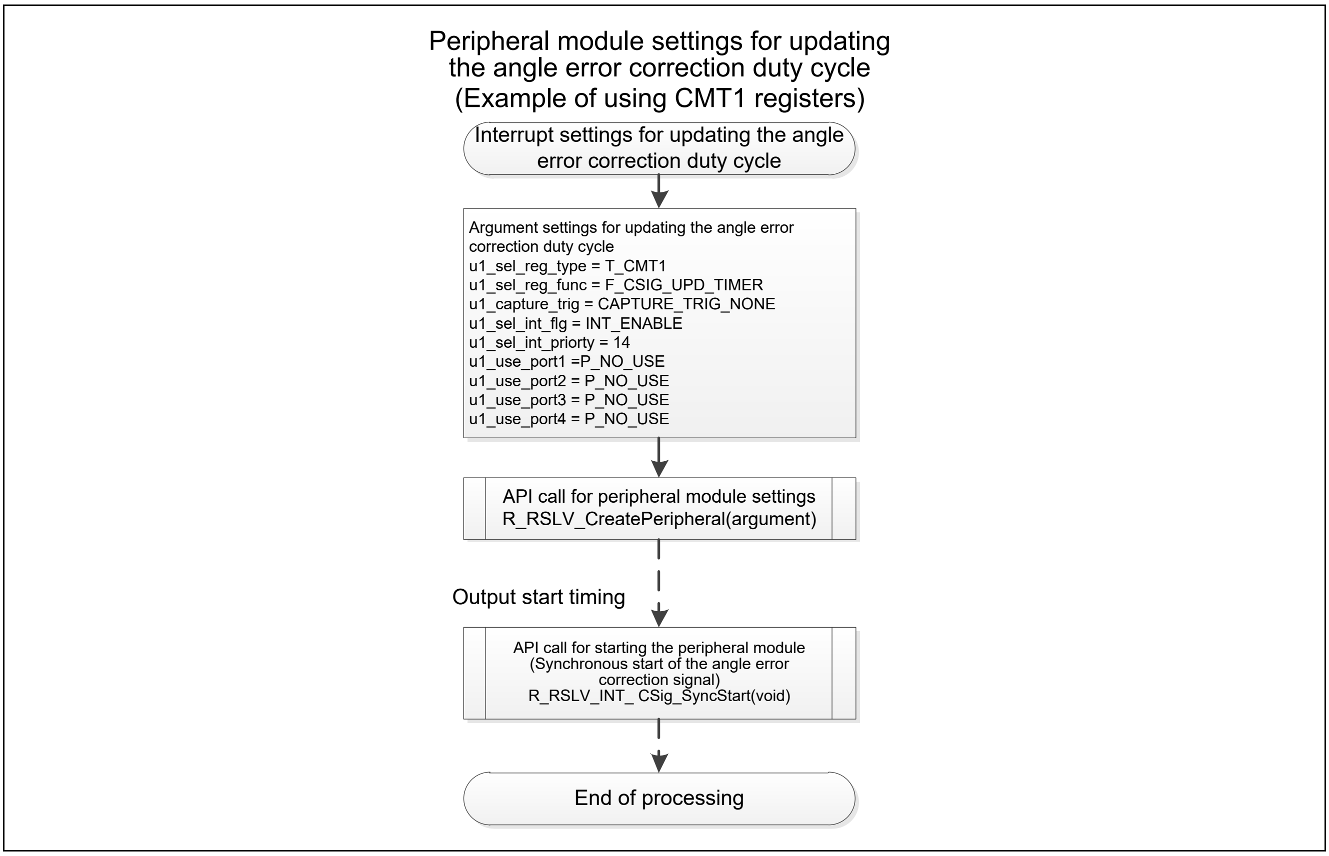


Figure 3.12 Processing for Updating the Duty Cycle of the Angle Error Correction Signal

(2) Interrupt

Interrupts for updating the duty cycle of the angle error correction signal are generated at intervals of 1/2 or 1/4 (depending on the number of duty cycle updates) of the excitation signal cycle. In the interrupt processing, the PWM duty value of the angle error correction signal is set in the register for the angle error correction 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). The input capture function can be set only in the MTU3.

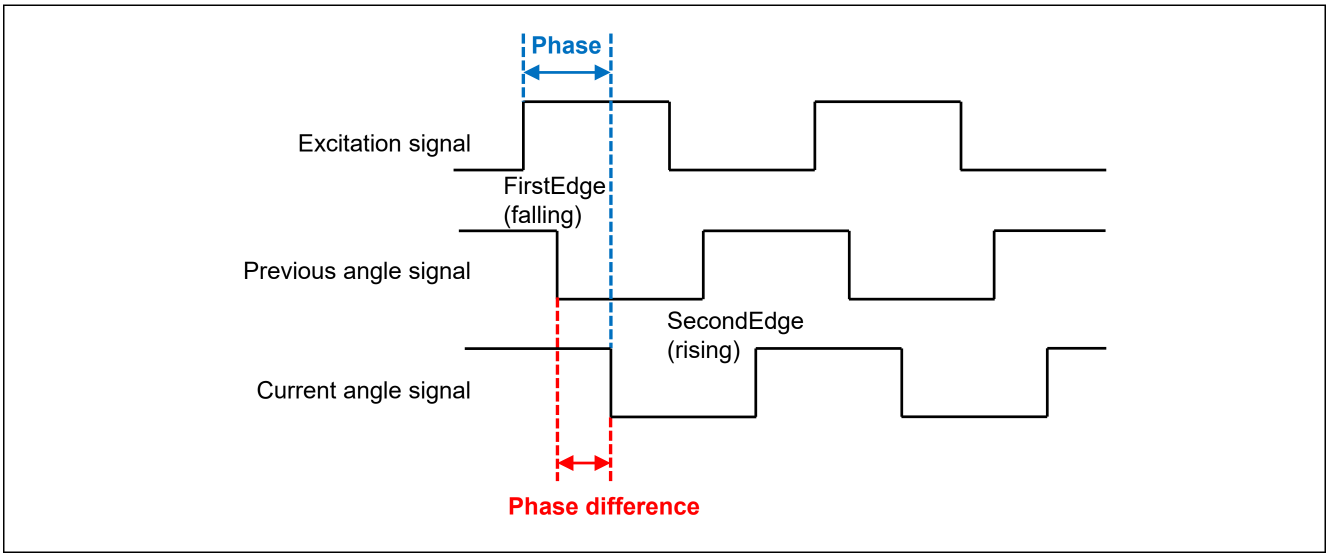


Figure 3.13 Angle Signal

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.

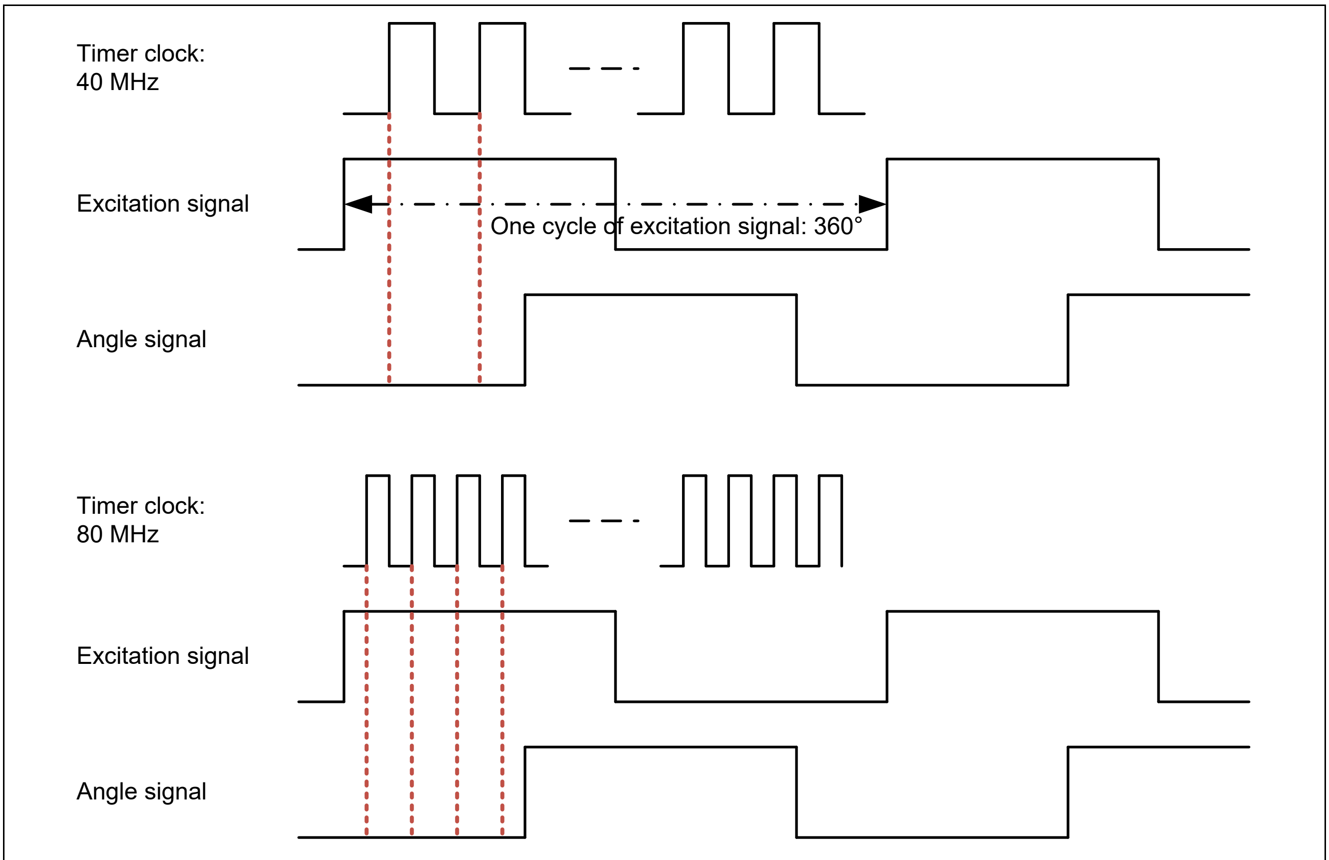


Figure 3.14 Concept of Resolution

The resolution (in terms of mechanical angle) of the angle signal can be obtained by multiplying the 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. When the timer clock is at 80 MHz, the resolution corresponds to 32000 values.

(1) Procedure for setting angle signal input interrupts

Follow the procedure below to set angle signal input interrupts. For details of the initial settings of the peripheral module in this figure, see section 6.3.1, Structure for R_RSLV_CreatePeripheral.

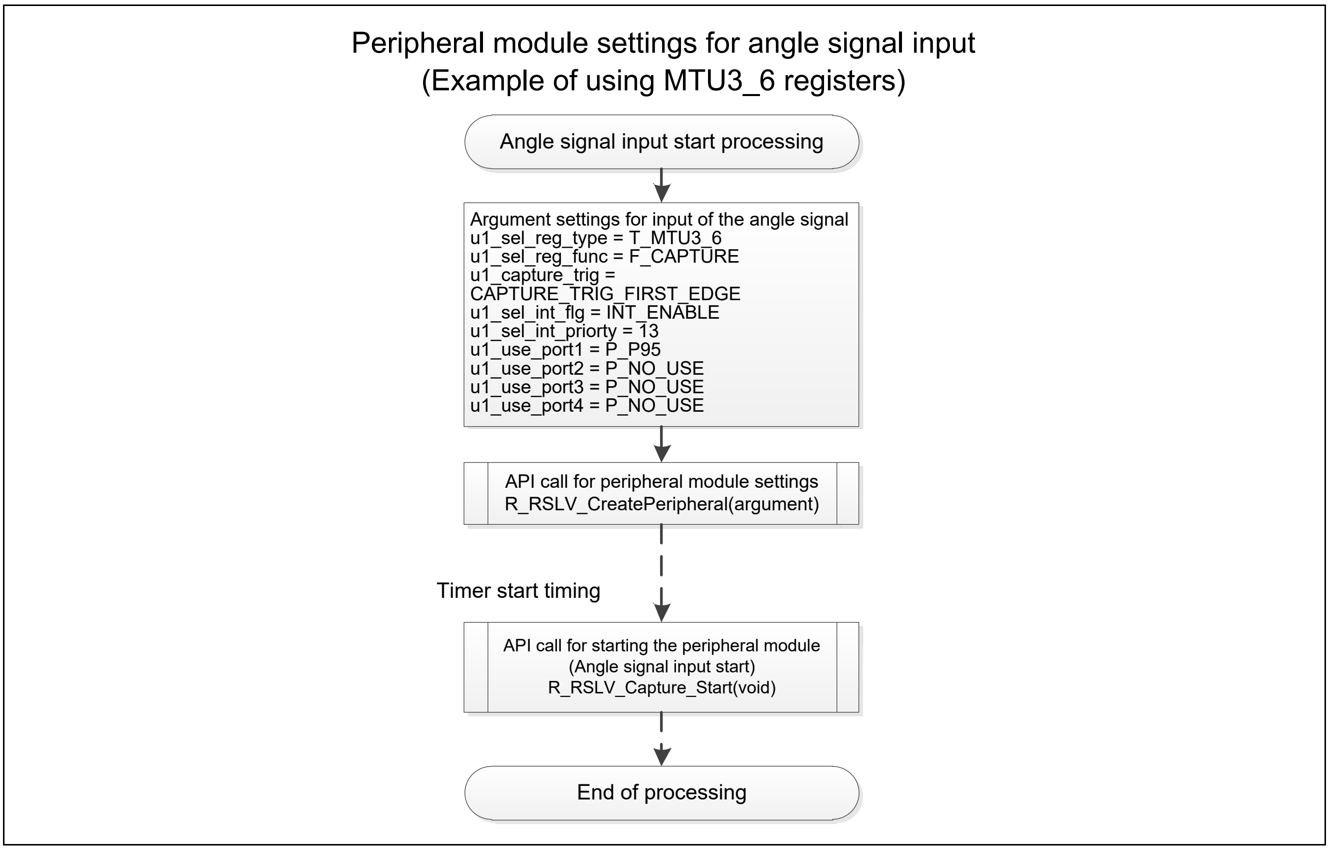


Figure 3.15 Angle Signal Input Processing Settings

(2) Interrupt

Input capture interrupts are generated on the specified edges of the input angle signal. The position of rotation 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.

3.7 Output of the RDC Operating Clock

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

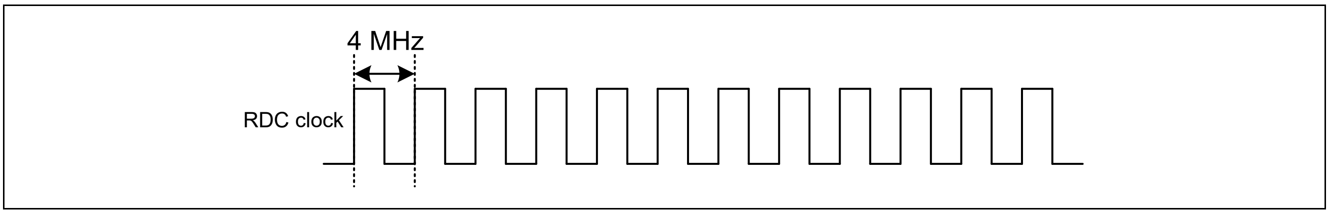


Figure 3.16 RDC Clock

(1) Procedure for setting output of the RDC clock

Follow the procedure below to set RDC clock output. For details of the initial settings of the peripheral module in this figure, see section 6.3.1, Structure for R_RSLV_CreatePeripheral.

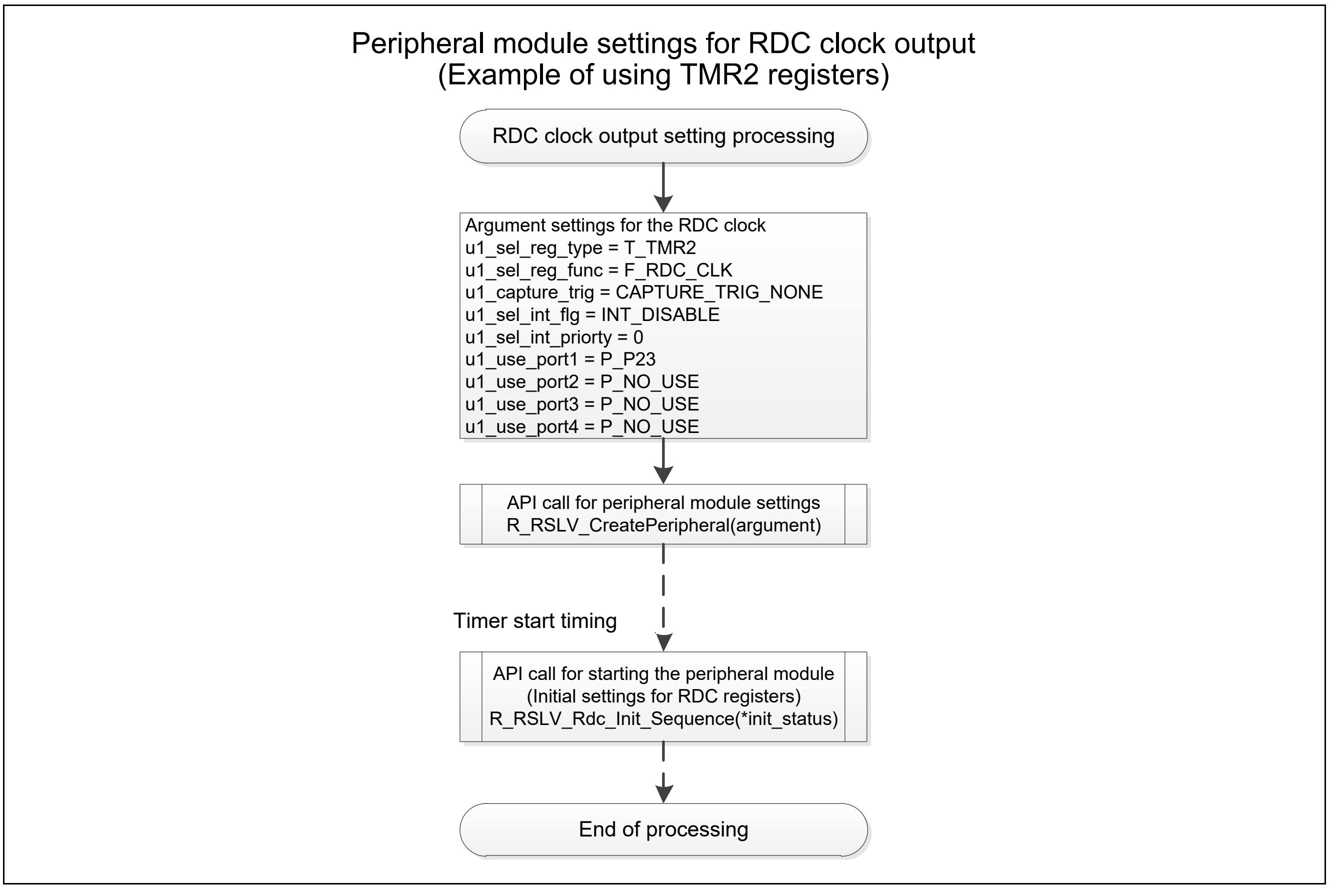


Figure 3.17 RDC Clock Processing

(2) Interrupt

No interrupt is generated for output of the RDC operating clock.

(3) Timer setting

The RDC clock frequency is always 4 MHz.

3.8 Automatic Calibration of Errors

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

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

3.8.1 Functions Used to Adjust Parameters

Automatic calibration uses the following functions to adjust parameters.

(1) RDC communications

RDC registers are manipulated through SPI communications.

(2) Angle error correction signal output

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

(3) PWM output for phase adjustment

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

(4) Acquiring phase count

This phase count is angle information obtained from the RDC.

(5) 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 facility for access to the 12-bit A/D converter must be prepared in the application.

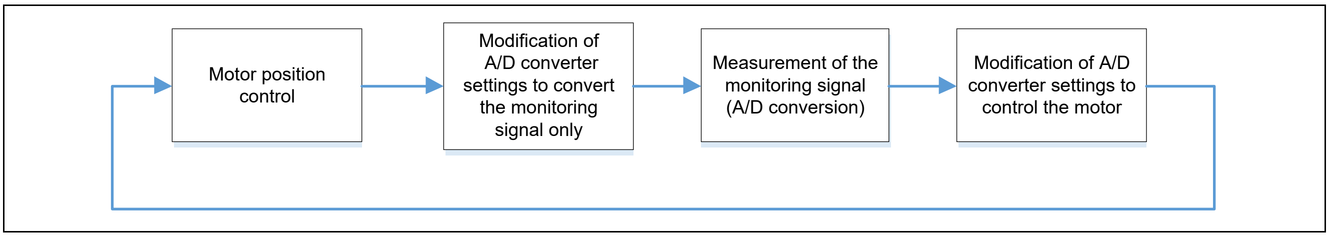


Figure 3.18 Schematic Processing Flow for Measuring Monitoring Signal for Correcting Angle Errors

(6) Motor position control

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

(7) Motor speed control

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

(8) Referencing speed data

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

3.8.2 Adjustment of Gain and Phase of Resolver Signals

(1) Resolver signal gain adjustment

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

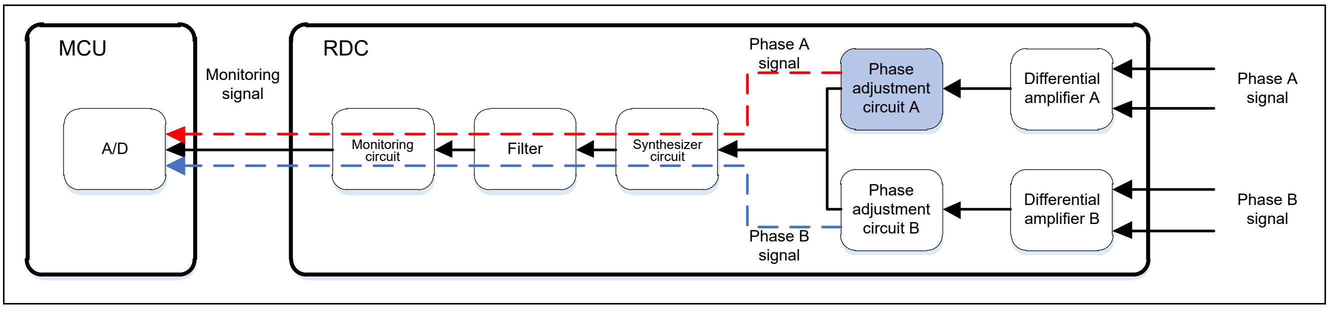


Figure 3.19 Block Diagram of Resolver Signal Gain Adjustment

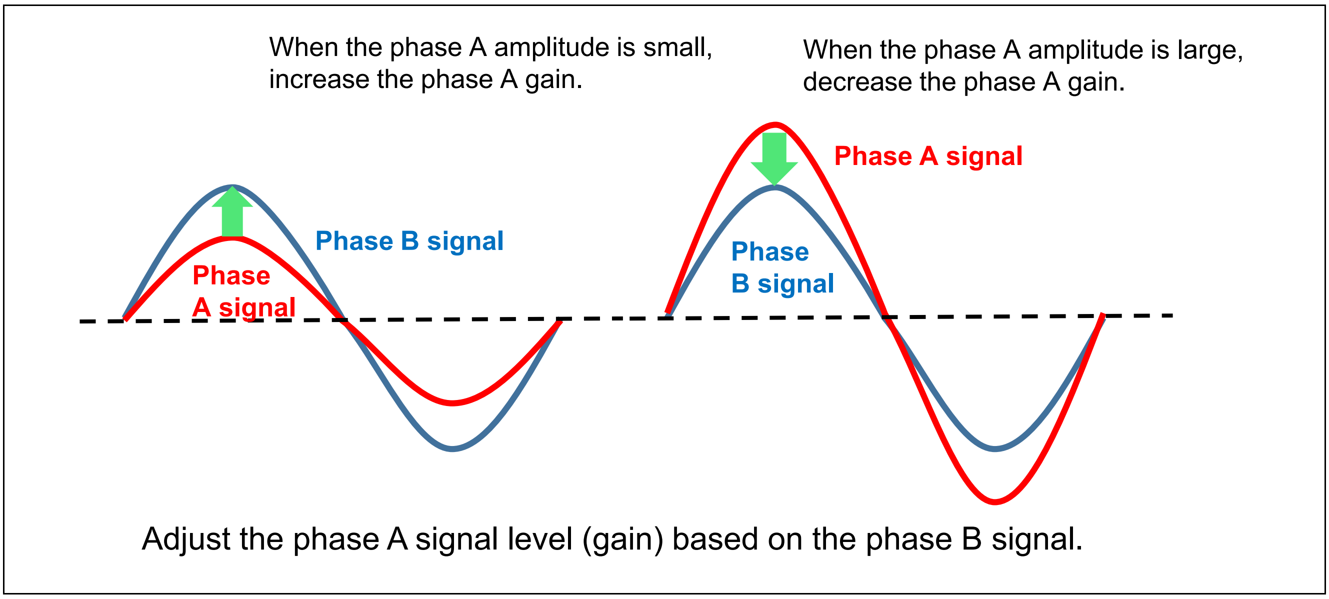


Figure 3.20 Resolver Signal Gain Adjustment

The phase A and phase B signals having different amplitudes produces 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%.

(2) Resolver signal phase adjustment

Figure 3.21 shows a block diagram for resolver signal phase adjustment.

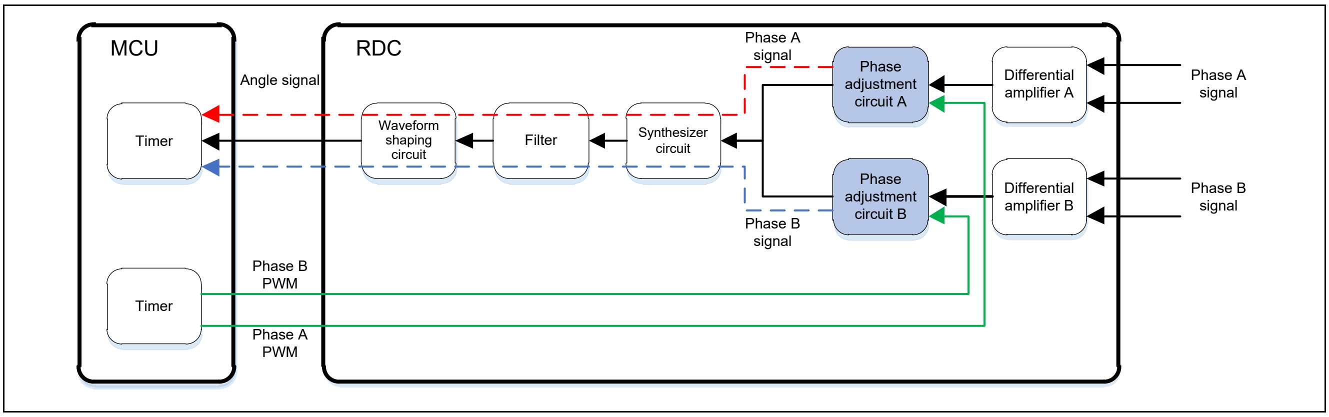


Figure 3.21 Block Diagram of Resolver Signal Phase Adjustment

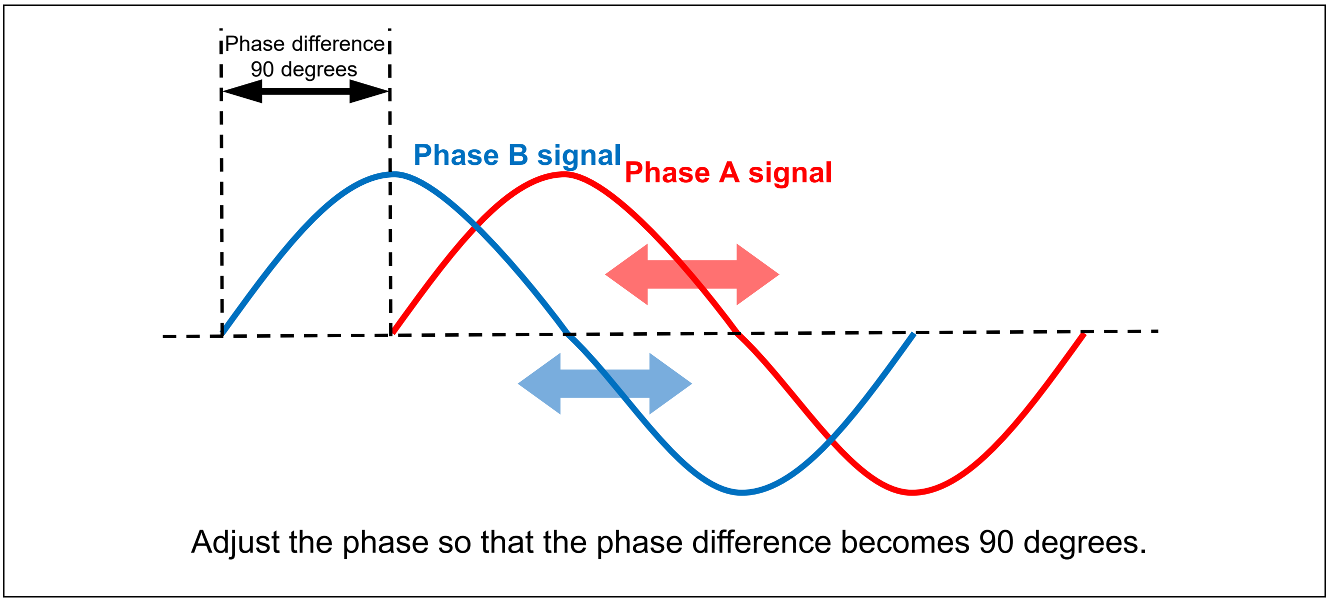


Figure 3.22 Resolver Signal Phase Adjustment

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)

3.8.3 Adjustment of the Angle Error Correction Signal

Figure 3.23 shows a block diagram for angle error correction signal adjustment.

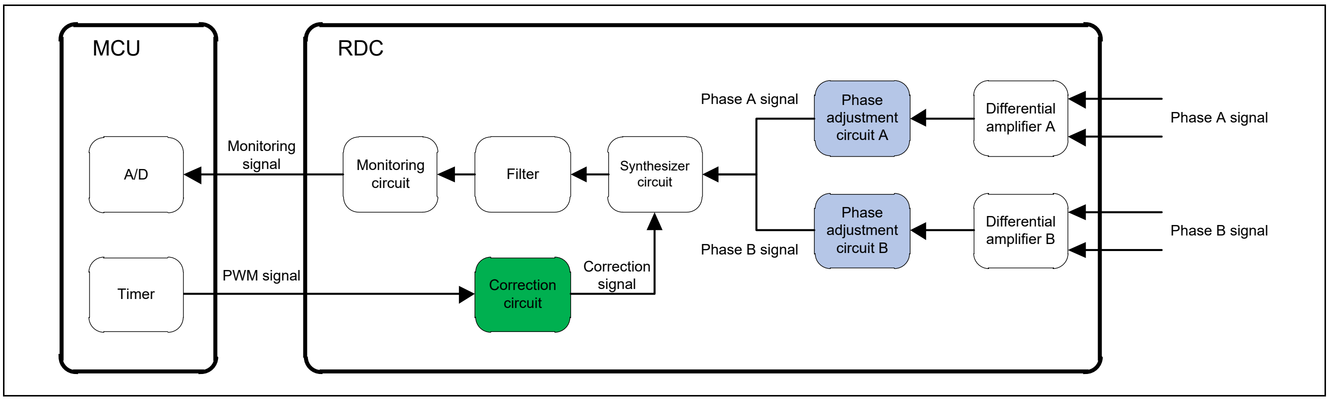


Figure 3.23 Block Diagram of Angle Error Correction Signal Adjustment

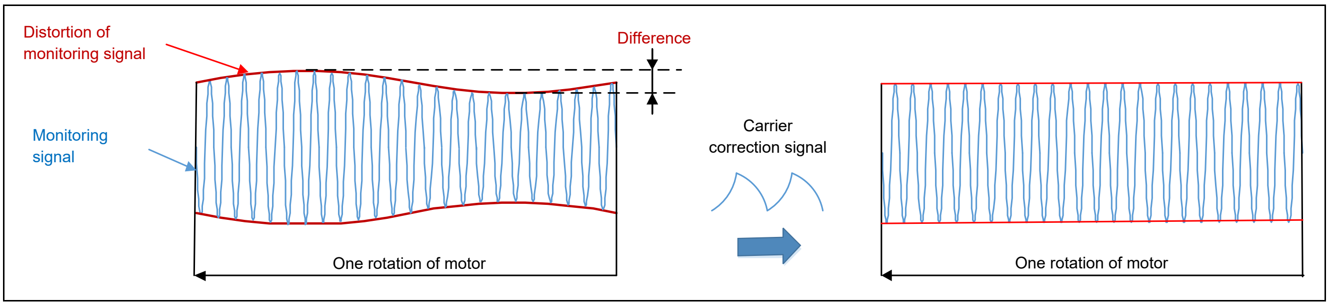


Figure 3.24 Angle Error Correction Signal Adjustment

This facility adjusts the amount of 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 shift and amplitude value for the angle error correction signal are shown below.

Amount of shift:

0 to 999 (1000 steps) at an excitation frequency of 5 kHz

0 to 499 (500 steps) at an excitation frequency of 10 kHz

0 to 249 (250 steps) at an excitation frequency of 20 kHz

Amplitude value:

0 to 199 (200 steps) at an angle error correction signal PWM cycle of 200 kHz

0 to 99 (100 steps) at an angle error correction signal PWM cycle of 400 kHz

3.9 Communications between the RDC and MCU

SPI communications are used between the MCU and the RDC. Figure 3.25 shows a system overview of the RDC communications block.

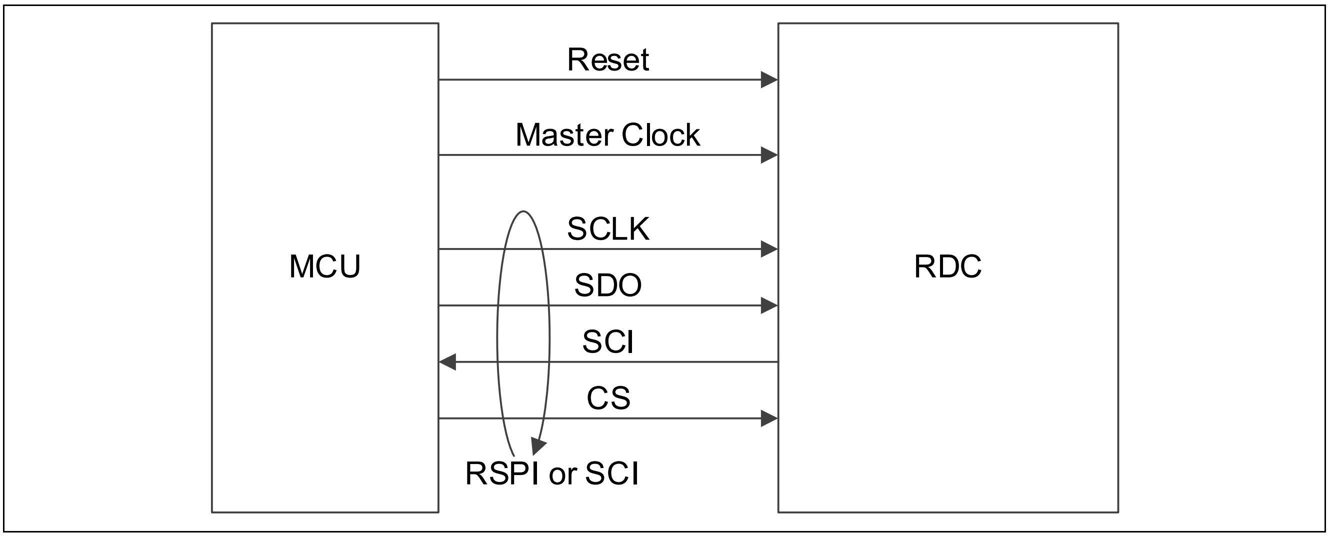


Figure 3.25 System Overview of the RDC Communications Block

3.9.1 Setting Peripheral Modules

The following describes the settings of peripheral modules.

(1) RDC master clock output

A 4-MHz rectangular wave signal is output from the MCU as an internal master clock of the RDC.

(2) Serial peripheral interface

When the RSPI module is selected, it is used with the following settings.

- When MOSI is idle, the MOSIA pin output level is low. RSPI loopback is set to normal mode.
- Bit rate: 1 Mbps
- Slave signal x (SSLx) (active low) is used.
- Delay from SSLx assertion: 1 RSPCK cycle. Negation delay: 1 RSPCK cycle
- Delay between slave signals: 1 RSPCK + 2 PCLK cycles
- The RSPI idle interrupt is disabled and no parity bit is attached.
- Data are sampled on even edges. The RSPCK level is high in idle state.
- Data length: 16 bits, MSB first

When the SCI module is selected, it is used with the following settings.

- Data transfer direction: MSB first
- Receive data level: Normal setting (Stored in the RDR register without being inverted)
- A clock with a frequency identical to the bit rate is output from the SCK pin.
- Bit rate: 1 Mbps

3.10 Detection of Disconnection from the Resolver Sensor

Figure 3.26 shows a system overview of detection of disconnection from the resolver sensor.

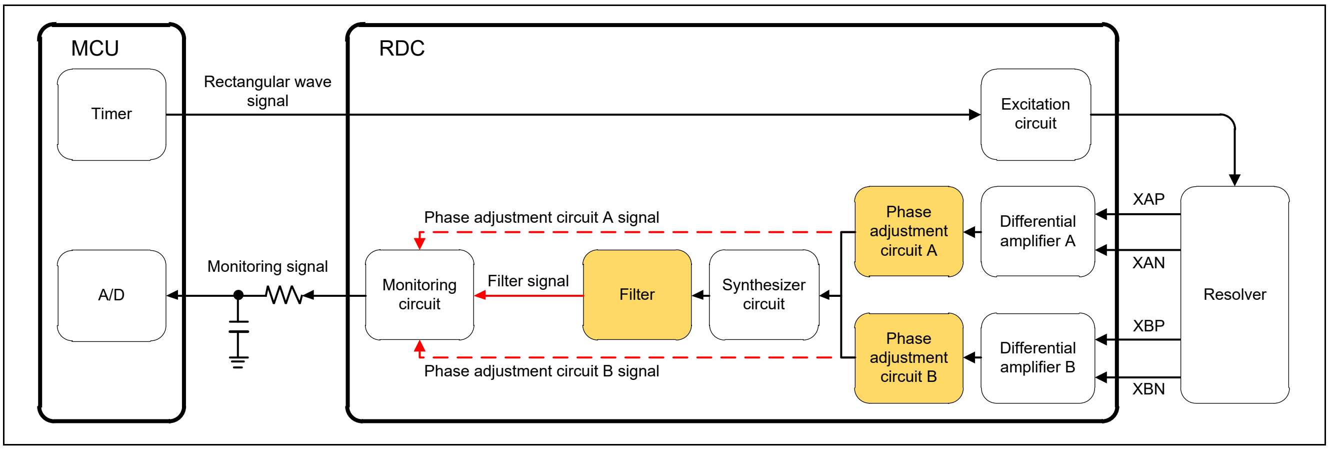


Figure 3.26 System Overview of Detection of Disconnection from the Resolver Sensor

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)

3.10.1 Functions Used for Detecting Disconnection

The following functions are used to detect disconnection.

(1) RDC communications

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

(2) Measuring the RDC monitoring signal

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

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_RX24T.lib	Library file for RX24T

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_ESIG12	3	Facility for setting the excitation signal (synthesized output)
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_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 Macro-Defined Names of Peripheral Modules

Table 5.2 lists the macro-defined names of the peripheral modules that can be used for each facility of this driver.

Table 5.2 List of Macro-Defined Names of Peripheral Modules

Defined Name	Defined Value	Description
T_TMR0	0	Peripheral module TMR0
T_TMR1	1	Peripheral module TMR1
T_TMR2	2	Peripheral module TMR2
T_TMR3	3	Peripheral module TMR3
T_TMR4	4	Peripheral module TMR4
T_TMR5	5	Peripheral module TMR5
T_TMR6	6	Peripheral module TMR6
T_TMR7	7	Peripheral module TMR7
T_MTU3_0	8	Peripheral module MTU3_0
T_MTU3_1	9	Peripheral module MTU3_1
T_MTU3_2	10	Peripheral module MTU3_2
T_MTU3_6	12	Peripheral module MTU3_6
T_MTU3_7	13	Peripheral module MTU3_7
T_MTU3_9	14	Peripheral module MTU3_9
T_CMT0	15	Peripheral module CMT0
T_CMT1	16	Peripheral module CMT1
T_CMT2	17	Peripheral module CMT2
T_CMT3	18	Peripheral module CMT3
T_RSPI	19	Peripheral module RSPI
T_SCI1	21	Peripheral module SCI1
T_SCI5	22	Peripheral module SCI5
T_SCI6	23	Peripheral module SCI6

5.3 List of Possible Combinations of Peripheral Modules and Driver Facilities

Table 5.3 lists the peripheral modules that can be assigned to serve the individual driver facilities.

Table 5.3 List of Possible Combinations of Peripheral Modules and Driver Facilities

				Name Defined for Facility								
				F_ESIG1	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
			Defined Value	0	3	4	5	6	8	9	10	11
Name Defined for Peripheral Module	TMR	T_TMR0	0				√	√				√
		T_TMR1	1				√	√				√
		T_TMR2	2				√	√				√
		T_TMR3	3				√	√				√
		T_TMR4	4				√	√				√
		T_TMR5	5				√	√				√
		T_TMR6	6				√	√				√
		T_TMR7	7				√	√				√
	MTU	T_MTU3_0	8	√	√	√						
		T_MTU3_1	9			√			√			
		T_MTU3_2	10			√			√			
		T_MTU3_6	12			√			√			
		T_MTU3_7	13			√			√			
		T_MTU3_9	14	√	√							
	CMT	T_CMT0	15							√		
		T_CMT1	16							√		
		T_CMT2	17							√		
		T_CMT3	18							√		
	Communications	T_RSPI	19								√	
		T_SCI1	21								√	
		T_SCI5	22								√	
		T_SCI6	23								√	

√: Possible

5.4 List of Port Pins Usable for Driver Facilities

This section lists the port pins of peripheral modules that can be used to serve the individual driver facilities.

5.4.1 TMR

Table 5.4 lists the TMR channels and port pins that can be used to serve the individual driver facilities.

Table 5.4 List of Possible Combinations of TMR Port Pins and Driver Facilities

				Name Defined for Facility								
				F_ESIG1	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
		Name Defined for Port Pin	Defined Value	0	3	4	5	6	8	9	10	11
Name Defined for Peripheral Module	T_TMR0	P_PD3	34				√	√				√
		P_PB0	27				√	√				√
		P_P33	13				√	√				√
	T_TMR1	P_PD6	37				√	√				√
	T_TMR2	P_PD1	32				√	√				√
		P_PA0	21				√	√				√
		P_P23	8				√	√				√
	T_TMR3	P_P11	4				√	√				√
	T_TMR4	P_PD2	33				√	√				√
		P_PA1	22				√	√				√
		P_P82	14				√	√				√
		P_P22	7				√	√				√
	T_TMR5	P_PE1	40				√	√				√
	T_TMR6	P_PD0	31				√	√				√
		P_P32	12				√	√				√
		P_P24	9				√	√				√
	T_TMR7	P_PA2	23				√	√				√

√: Possible

5.4.2 MTU3

Table 5.5 lists the MTU3 channels and port pins that can be used to serve the individual driver facilities.

Table 5.5 List of Possible Combinations of MTU Port Pins and Driver Facilities

				Name Defined for Facility								
				F_ESIG1	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
		Name Defined for Port Pin	Defined Value	0	3	4	5	6	8	9	10	11
Name Defined for Peripheral Module	T_MTU3_0	P_PB3	30	√	√	√						
		P_PB2	29	√	√	√						
		P_PB1	28		√	√						
		P_PB0	27		√	√						
		P_P31	11		√	√						
		P_P30	10		√	√						
	T_MTU3_1	P_PA5	26			√			√			
		P_PA4	25			√			√			
	T_MTU3_2	P_PA3	24			√			√			
		P_PA2	23			√			√			
	T_MTU3_6	P_PA1	22			√			√			
		P_PA0	21			√			√			
		P_P95	20						√			
		P_P92	17						√			
	T_MTU3_7	P_P94	19			√			√			
		P_P93	18						√			
		P_P91	16			√			√			
		P_P90	15						√			
	T_MTU3_9	P_PE1	40	√								
		P_PE0	39	√								
		P_PD7	38	√								
		P_PD6	37	√								
		P_P21	6	√								
		P_P20	5	√								
		P_P10	3	√								
		P_P02	2	√								

√: Possible

5.4.3 RSPI

Table 5.6 lists the RSPI port pins that can be used to serve the communications with the RDC.

Table 5.6 List of Possible Combinations of RSPI Port Pins and Driver Facilities

				Name Defined for Facility			
				F_RDC_COM			
				Port to be Set			
				Port 1	Port 2	Port 3	Port 4
		Name Defined for Port Pin	Defined Value	RSPCKA	MISOA	MOSIA	SSLA
Name Defined for Peripheral Module	T_RSPI	P_PE1	40				√
		P_PE0	39				√
		P_PD7	38				√
		P_PD6	37				√
		P_PD2	33		√		
		P_PD1	32			√	
		P_PD0	31	√			
		P_PB3	30	√			
		P_PB0	27		√		
		P_PA5	26			√	
		P_PA4	25	√			
		P_PA3	24				√
		P_PA2	23				√
		P_PA1	22				√
		P_PA0	21				√
		P_P33	13				√
		P_P32	12				√
		P_P31	11				√
		P_P30	10				√
		P_P24	9	√			
		P_P23	8		√		
		P_P22	7			√	

√: Possible

5.4.4 SCI

Table 5.7 lists the SCI port pins that can be used to serve the communications with the RDC.

Table 5.7 List of Possible Combinations of SCI Port Pins and Driver Facilities

				Name Defined for Facility			
				F_RDC_COM			
				Port to be Set			
				Port 1	Port 2	Port 3	Port 4
		Name Defined for Port Pin	Defined Value	SCK	SMOSI	SMISO	—
Name Defined for Peripheral Module	T_SCI1	P_PD5	36		√		
		P_PD4	35	√			
		P_PD3	34			√	
	T_SCI5	P_PE0	39		√		
		P_PB7	43	√			
		P_PB6	41		√		
		P_PB5	42			√	
		P_PD7	38			√	
		P_PD2	33	√			
	T_SCI6	P_PB3	30	√			
		P_PB2	29			√	
		P_PB1	28		√		
		P_PB0	27			√	
		P_PA5	26		√		
		P_PA2	23	√			
		P_P82	14	√			
		P_P81	45			√	
		P_P80	44		√		

√: Possible

5.5 Setting Interrupts for Driver Facilities

5.5.1 Availability of Interrupt Settings

Table 5.8 lists the availability of interrupt settings for the individual driver facilities.

Table 5.8 Availability of Interrupt Settings

			Name Defined for Facility								
			F_ESIG1	F_ESIG12	F_CSIG	F_PHASE_A	F_PHASE_B	F_CAPTURE	F_CSIG_UPD_TIMER	F_RDC_COM	F_RDC_CLK
		Defined Value	0	3	4	5	6	8	9	10	11
Name Defined for Interrupt Setting	INT_DISABLE	0	√	√	√	√	√	—	—	*	√
	INT_ENBALE	1	√	√	—	—	—	√	√	*	×

√: Available —: Not available

Note: * Interrupts are always enabled regardless of the settings.

5.5.2 Interrupt Sources for the Output of the Excitation Signals

Table 5.9 and Table 5.10 list the interrupt sources for the output of the excitation signals.

Table 5.9 Interrupt Sources for the Output of the Single-Phase Excitation Signal

			Interrupt Source							
		Name Defined for Port Pin	TGIA0	TGIB0	TGIC0	TGID0	TGIA9	TGIB9	TGIC9	TGID9
Name Defined for Peripheral Module	T_MTU3_0	P_PB3			√					
		P_PB2			√					
		P_PB1		√						
		P_PB0		√						
		P_P31			√					
		P_P30			√					
	T_MTU3_9	P_PE1						√		
		P_PE0							√	
		P_PD7							√	
		P_PD6						√		
		P_P21							√	
		P_P20						√		
		P_P10							√	
		P_P02						√		

√: Available

Table 5.10 Interrupt Sources for the Output of the Synthesized Excitation Signal (1/3)

Name Defined for Peripheral Module		Name Defined for Port Pin 1	Name Defined for Port Pin 2	Interrupt Source: TGIA0	TGIB0	TGIC0	TGID0	TGIA9	TGIB9	TGIC9	TGID9
Name Defined for Peripheral Module	T_MTU3_0	P_PB3	P_PB2			√					
			P_PB1				√				
			P_PB0			√					
			P_P30			√					
		P_PB2	P_PB3			√					
			P_PB1	√							
			P_PB0			√					
			P_P31			√					
		P_PB1	P_PB3				√				
			P_PB2				√				
			P_PB0			√					
			P_P31				√				
			P_P30				√				
		P_PB0	P_PB3			√					
			P_PB2			√					
			P_PB1		√						
			P_P31			√					
			P_P30			√					
		P_P31	P_PB2			√					
			P_PB1				√				
			P_PB0			√					
			P_P30			√					
		P_P30	P_PB3			√					
			P_PB1	√							
			P_PB0			√					
			P_P31			√					

√: Available

Table 5.10 Interrupt Sources for the Output of the Synthesized Excitation Signal (2/3)

				Interrupt Source							
		Name Defined for Port Pin 1	Name Defined for Port Pin 2	TGIA0	TGIB0	TGIC0	TGID0	TGIA9	TGIB9	TGIC9	TGID9
Name Defined for Peripheral Module	T_MTU3_9	P_PE1	P_PE0							√	
			P_PD7							√	
			P_PD6						√		
			P_P21							√	
			P_P20						√		
			P_P10							√	
		P_PE0	P_PE1							√	
			P_PD7							√	
			P_PD6					√			
			P_P21							√	
			P_P20					√			
			P_P02							√	
		P_PD7	P_PE1							√	
			P_PE0							√	
			P_PD6								√
			P_P20								√
			P_P10							√	
			P_P02							√	
		P_PD6	P_PE1						√		
			P_PE0								√
			P_PD7								√
			P_P21								√
			P_P10								√
			P_P02						√		

√: Available

Table 5.10 Interrupt Sources for the Output of the Synthesized Excitation Signal (3/3)

				Interrupt Source							
		Name Defined for Port Pin 1	Name Defined for Port Pin 2	TGIA0	TGIB0	TGIC0	TGID0	TGIA9	TGIB9	TGIC9	TGID9
Name Defined for Peripheral Module	T_MTU3_9	P_P21	P_PE1							√	
			P_PE0							√	
			P_PD6								√
			P_P20								√
			P_P10							√	
			P_P02							√	
		P_P20	P_PE1						√		
			P_PE0								√
			P_PD7								√
			P_P21								√
			P_P10								√
			P_P02						√		
		P_P10	P_PE1							√	
			P_PD7							√	
			P_PD6					√			
			P_P21							√	
			P_P20					√			
			P_P02							√	
		P_P02	P_PE0							√	
			P_PD7							√	
			P_PD6						√		
			P_P21							√	
			P_P20						√		
			P_P10							√	

√: Available

5.5.3 Input Capture Interrupt Sources

Table 5.11 lists the input capture interrupt sources.

Table 5.11 Input Capture Interrupt Sources

Name Defined for Facility		Name Defined for Peripheral Module		Name Defined for Port Pin	Interrupt Source															
					TGIA1	TGIB1	TGIC1	TGID1	TGIA2	TGIB2	TGIC2	TGID2	TGIA6	TGIB6	TGIC6	TGID6	TGIA7	TGIB7	TGIC7	TGID7
	F_CAPTURE		T_MTU_1	P_PA5	√															
				P_PA4		√														
			T_MTU_2	P_PA3					√											
				P_PA2						√										
			T_MTU_6	P_PA1									√							
				P_PA0											√					
				P_P95										√						
				P_P92												√				
			T_MTU_7	P_P94													√			
				P_P93														√		
				P_P91															√	
				P_P90																√

√: Available

5.5.4 Interrupt Sources for Updating the Angle Error Correction Duty Cycle

Table 5.12 lists the interrupt sources for updating the angle error correction duty cycle.

Table 5.12 Interrupt Sources for Updating the Angle Error Correction Duty Cycle

Name Defined for Facility		Name Defined for Peripheral Module		Interrupt Source			
				CMI0	CMI1	CMI2	CMI3
	F_CSIG_UPD_TIMER		T_CMT0	√			
			T_CMT1		√		
			T_CMT2			√	
			T_CMT3				√

√: Available

5.5.5 Interrupt Sources for Communications with the RDC

Table 5.13 lists the interrupt sources for communications with the RDC.

Table 5.13 Interrupt Sources for Communications with the RDC

Name Defined for Facility	Name Defined for Peripheral Module	Interrupt Source															
		SPTI0	SPRI0	SPEI0	SPII0	TXI1	RXI1	ERI1	TEI1	TXI5	RXI5	ERI5	TEI5	TXI6	RXI6	ERI6	TEI6
F_RDC_COM	T_RSPI	√	√	√	√												
	T_SCI1					√	√	√	√								
	T_SCI5									√	√	√	√				
	T_SCI6													√	√	√	√

√: Available

5.6 Capture Trigger Settings for Driver Facilities

Table 5.14 lists the availability of capture trigger settings for individual driver facilities.

Table 5.14 Capture Trigger Edge Settings for Driver Facilities

Name Defined for Facility	Defined Value	Availability of Capture Trigger Setting		
		CAPTURE_TRIG_FRIST_EDGE	CAPTURE_TRIG_SECOND_EDGE	CAPTURE_TRIG_BOTH_EDGE
		1	2	3
F_ESIG1	0			
F_ESIG12	3			
F_CSIG	4			
F_PHASE_A	5			
F_PHASE_B	6			
F_CAPTURE	8	√	√	√
F_CSIG_UPD_TIMER	9			
F_RDC_COM	10			
F_RDC_CLK	11			

√: Available

Capture triggers can only be set for the input capture facility for detecting the angle signal. When initializing the other facilities, specify CAPTURE_TRIG_NONE.

6. APIs

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_CreatePeripheral Input: ST_INIT_REG_PARAM *rdc_init_param / Register initialization parameters Output: unsigned char result / Processing result	Initializes the settings of the peripheral module registers specified by a member variable of the argument. That is, the required register settings can be made from the member variables that specify the desired peripheral module, driver facility, and so on.
		R_RSLV_SetSystemInfo Input: ST_SYSTEM_PARAM *rdc_sys_param / System information Output: unsigned char result / Processing result	Specifies the system information such as the timer counter value to be used according to the information passed through the argument.
		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 input capture timer counter specified in the RDC driver, sets the information in the pointer variable argument, and reports it to the user.
		R_RSLV_SetFunctionPointer Input: UNSIGNED_CHAR_POINTER func / Function pointer unsigned char func_id / Type of function passed through argument Output: unsigned char result / Processing result	Sets the function pointer passed through the argument in the relevant function pointer variable. (CS manipulation in RDC communications and reading of the value of the RDC alarm cancellation port)
		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 difference 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 the angle error correction signal in synchronization with the excitation signal interrupt processing.
		R_RSLV_GetCSigStatus Input: unsigned char *status/ Pointer to angle error correction signal output state to be acquired Output: unsigned char result / Processing result	Acquires the output state of the 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	Input capture (angle signal)	R_RSLV_Capture_Start Input: None Output: unsigned char result / Processing result	Starts the input capture timer.
		R_RSLV_INT_GetCaptureCount Input: None Output: unsigned char result / Processing result	Acquires the input capture value (current angle count), calculates the difference from the previous value, and then sets it in the variable.
		R_RSLV_SetCaptureTiming Input: unsigned short tcnt / Timer counter value to be adjusted Output: unsigned char result / Processing result	Adjusts the count start timing of the input capture timer for angle detection.
		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	Reads 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	Reads 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	Reads 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	Reads 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_EsigStartTiming Input: unsigned short tcnt / Timer counter value Output: unsigned char result / Processing result	Adjusts the excitation signal output timing.
		R_RSLV_INT_ESigCounter Input: None Output: unsigned char result / Processing result	Starts counting down by the wait timer in the adjustment 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 communications sequence is provided and repeated calls of this function cause progress through the sequence.
		R_RSLV_Rdc_RegWrite Input: unsigned char wt_data / Write data unsigned char address / Write address 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_RdcCom_GetErrorInfo Input: unsigned char *err_info / RDC communications error information Output: unsigned char result / Processing result	Acquires information about whether an RDC communications error has occurred.
		R_RSLV_INT_RdcCom_Recv Input: None Output: unsigned char result / Processing result	Performs the reception interrupt callback processing after being called from the reception interrupt processing of RDC communications.
		R_RSLV_INT_RdcCom_Trans Input: None Output: unsigned char result / Processing result	Transmits data, if any, after being called from the transmission interrupt processing of RDC communications.
		R_RSLV_INT_RdcCom_Error Input: None Output: unsigned char result / Processing result	Performs the error callback processing after being called from the error interrupt processing of RDC communications.

Table 6.1 API Functions (r_rslv_api.h) (4/4)

File Name	Category	Interface Function Name	Processing
r_rslv_api.h	RDC settings	R_RSLV_INT_RdcCom_Idle Input: None Output: unsigned char result / Processing result	Performs the transmission end callback processing after being called from the transmission end interrupt processing of RDC communications.
		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.
	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 twice alternately.
		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 Initial Settings of an On-Chip Peripheral Module of the MCU

Item	Description	
Function name	R_RSLV_CreatePeripheral	
Argument	ST_INIT_REG_PARAM *rdc_init_param	Parameters for selecting a peripheral module and a driver facility
Return value	unsigned char	Processing result
Function	Initializes the driver. • Selecting a peripheral module • Selecting a facility Input capture Output of the excitation signal Output of the angle error correction signal Timer for updating the angle error correction signal, etc. • Setting whether to use interrupts • Specifying ports	
Remark	ST_INIT_REG_PARAM is a structure. For the settings of the peripheral module, facility, and other items, see section 6.3.1 Structure for R_RSLV_CreatePeripheral. For possible combinations of peripheral modules and driver facilities, see section 5.3, List of Possible Combinations of Peripheral Modules and Driver Facilities.	

6.2.2 API Function for Specifying System Information

Item	Description	
Function name	R_RSLV_SetSystemInfo	
Argument	ST_SYSTEM_PARAM *rdc_sys_param	System setting information
Return value	unsigned char	Processing result
Function	Specifies the following system information. • MCU type • 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	
Remark	ST_SYSTEM_PARAM is a structure. For details of system information settings, see section 6.3.2 Structure for R_RSLV_SetSystemInfo. Because this API function makes the settings of peripheral modules such as the timer values for individual peripheral module registers, this API function must be called prior to the function described in section 6.2.1, API Function for Initial Settings of an On-Chip Peripheral Module.	

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 input capture 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 Setting the Pointer to a User Function

Item	Description	
Function name	R_RSLV_SetFunctionPointer	
Argument	UNSIGNED_CHAR_POINTER func unsigned char func_id	Pointer to a user-created function Facility for which the function is to be set
Return value	unsigned char	Processing result
Function	Sets the pointer to a user-created function. The following can be specified. • Function for setting the RDC chip select port to a low level (the RDC is selected) • Function for setting the RDC chip select port to a high level (the RDC is deselected) • Function for acquiring the value of the RDC alarm cancellation port "func_id" specifies the type of the user-created function to be set. The following can be set. • RDC_CS_ON (2): ID of the user function for setting the RDC chip select port to a low level • RDC_CS_OFF (3): ID of the user function for setting the RDC chip select port to a high level • RDC_ALARM_PORT (4): ID of the user function for acquiring the value of the RDC alarm cancellation port	
Remark	This API function stores the function pointer passed through an argument in the specific pointer variable provided for each func_id.	

6.2.5 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 carrier correction signal timer simultaneously.	

6.2.6 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.	

6.2.7 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 3.8.3, 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.	

6.2.8 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.	

6.2.9 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		

6.2.10 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		

6.2.11 API Function for Acquiring the Output State of the Angle Error Correction Signal

Item	Description	
Function name	R_RSLV_GetCSigStatus	
Argument	unsigned char *status	Output state of the angle error correction signal
Return value	unsigned char	Processing result
Function	Acquires the output state of the angle error correction signal.	
Remark		

6.2.12 API Function for Starting the Input Capture Timer

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

6.2.13 API Function for Acquiring the Input Capture 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 read 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 read using the following API function. R_RSLV_GetCaptureEdge	
Remark		

6.2.14 API Function for Setting the Input Capture Timer Start Timing

Item	Description	
Function name	R_RSLV_SetCaptureTiming	
Argument	unsigned short tcnt	Counter value for adjusting the start timing
Return value	unsigned char	Processing result
Function	Sets the adjustment value for starting the input capture interrupt timer.	
Remark		

6.2.15 API Function for Reading the Trigger Information for the Input Capture Interrupt

Item	Description	
Function name	R_RSLV_GetCaptureEdge	
Argument	unsigned char *cap_edge	Variable to store input capture trigger information
Return value	unsigned char	Processing result
Function	Reads the trigger information for the input capture interrupt. (Rising edge or falling edge can be determined according to the port level.)	
Remark		

6.2.16 API Function for Reading the Resolver Position Count (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	Reads the counter value detected by the input capture facility.	
Remark	• The counter value detected on the falling edge of the angle signal is read. • Use the R_RSLV_INT_GetCaptureCount function to acquire the counter value.	

6.2.17 API Function for Acquiring the Resolver Position Difference Count (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	Reads 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.	

6.2.18 API Function for Reading the Resolver Position Count (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	Reads the counter value detected by the input capture facility.	
Remark	• The counter value detected on the rising edge of the angle signal is read. • Use the R_RSLV_INT_GetCaptureCount function to acquire the counter value.	

6.2.19 API Function for Reading the Resolver Position Difference Count (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	Reads 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.	

6.2.20 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		

6.2.21 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 input capture timer also stop.	

6.2.22 API Function for Setting the Excitation Signal Output Start Timing

Item	Description	
Function name	R_RSLV_ESigStartTiming	
Argument	unsigned short tcnt	Setting of the excitation signal output start timing
Return value	unsigned char	Processing result
Function	Sets the timing to start outputting the excitation signal.	
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.	

6.2.23 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 adjustment processing.	
Remark	Counting down is performed only in the adjustment processing.	

6.2.24 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.	

6.2.25 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		

6.2.26 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		

6.2.27 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.	

6.2.28 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		

6.2.29 API Function for Initializing Variables for RDC Communications

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	This function initializes the following registers. PS1 (Power-saving control register 1 at address 02h) PS2 (Power-saving control register 2 at address 04h) PS3 (Power-saving control register 3 at address 0Ah) ALMOUT (ALARM# output setting register at address 16h) GCGSL (Differential amplification circuit gain selection register at address 2Eh) CSACTL (Shunt current amplification circuit control register at address 42h)	

6.2.30 API Function for Executing 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		

6.2.31 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.	

6.2.32 API Function for Writing to an RDC Register

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

6.2.33 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.	

6.2.34 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		

6.2.35 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 buffered value of the RDC register address specified by an argument.	
Remark		

6.2.36 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		

6.2.37 API Function for Reporting Errors in RDC Communications

Item	Description	
Function name	R_RSLV_RdcCom_GetErrorInfo	
Argument	unsigned char *err_info	Storage of RDC communications 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		

6.2.38 API Function for a Reception Interrupt in RDC Communications

Item	Description	
Function name	R_RSLV_INT_RdcCom_Recv	
Argument	void	
Return value	unsigned char	Processing result
Function	Reads data from the receive register at a reception end interrupt of RDC communications.	
Remark		

6.2.39 API Function for a Transmission Interrupt in RDC Communications

Item	Description	
Function name	R_RSLV_INT_RdcCom_Trans	
Argument	void	
Return value	unsigned char	Processing result
Function	Writes data to the transmit register at a transmission end interrupt of RDC communications.	
Remark		

6.2.40 API Function for an Error Interrupt in RDC Communications

Item	Description	
Function name	R_RSLV_INT_RdcCom_Error	
Argument	void	
Return value	unsigned char	Processing result
Function	Stops RDC communications at an error interrupt of RDC communications.	
Remark		

6.2.41 API Function for an Idle Interrupt in RDC Communications

Item	Description	
Function name	R_RSLV_INT_RdcCom_Idle	
Argument	void	
Return value	unsigned char	Processing result
Function	Stops RDC communications at an idle interrupt of RDC communications.	
Remark		

6.2.42 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		

6.2.43 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.	

6.2.44 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.	

6.2.45 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, Structure for R_RSLV_ADJST_Carrier.	

6.2.46 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	void	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.	

6.2.47 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		

6.2.48 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 return_val	Processing result
Function	Performs the sequence for detection 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_CreatePeripheral (section 6.2.1)
- R_RSLV_SetSystemInfo (section 6.2.2)
- R_RSLV_GetRdcDrvSettingInfo (section 6.2.3)
- R_RSLV_ADJST_GainPhase (section 6.2.44)
- R_RSLV_ADJST_Carrier (section 6.2.45)
- R_RSLV_ADJST_SetPtrFunc (section 6.2.46)
- R_RSLV_DiscDetection_Seq (section 6.2.48)

6.3.1 Structure for R_RSLV_CreatePeripheral

The argument of the R_RSLV_CreatePeripheral API function is an ST_INIT_REG_PARAM structure defined as shown below.

API function: R_RSLV_CreatePeripheral (ST_INIT_REG_PARAM *rdc_init_param)

Table 6.2 Structure Definitions for R_RSLV_CreatePeripheral (1/2)

Member Name	Type	Description		Defined Value	Macro-Defined Name
u1_sel_reg_type	unsigned char	Type of peripheral module	TMR0	0	T_TMR0
			TMR1	1	T_TMR1
			TMR2	2	T_TMR2
			TMR3	3	T_TMR3
			TMR4	4	T_TMR4
			TMR5	5	T_TMR5
			TMR6	6	T_TMR6
			TMR7	7	T_TMR7
			MTU3_0	8	T_MTU3_0
			MTU3_1	9	T_MTU3_1
			MTU3_2	10	T_MTU3_2
			MTU3_6	12	T_MTU3_6
			MTU3_7	13	T_MTU3_7
			MTU3_9	14	T_MTU3_9
			CMT0	15	T_CMT0
			CMT1	16	T_CMT1
			CMT2	17	T_CMT2
			CMT3	18	T_CMT3
			RSPI	19	T_RSPI
			SCI1	21	T_SCI1
			SCI5	22	T_SCI5
			SCI6	23	T_SCI6

Table 6.2 Structure Definitions for R_RSLV_CreatePeripheral (2/2)

Member Name	Type	Description		Defined Value	Macro-Defined Name
u1_sel_reg_func	unsigned char	Driver facility	ESIG1	0	F_ESIG1
			ESIG12	3	F_ESIG12
			CSIG	4	F_CSIG
			PHASE_A	5	F_PHASE_A
			PHASE_B	6	F_PHASE_B
			CAPTURE	8	F_CAPTURE
			CSIG_UPD_TIMER	9	F_CSIG_UPD_TIMER
			RDC_COM	10	F_RDC_COM
			RDC_CLK	11	F_RDC_CLK
u1_sel_int_flg	unsigned char	Whether to use interrupts	Not used	0	INT_DISABLE
			Used	1	INT_ENABLE
u1_sel_int_priorty	unsigned char	Priority of interrupts	Highest: 15 Lowest: 1 Prohibited: 0	0 to 15	—
u1_capture_trig	unsigned char	Input capture trigger	Not set	0	CAPTURE_TRIG_NONE
			Falling edge	1	CAPTURE_TRIG_FIRST_EDGE
			Rising edge	2	CAPTURE_TRIG_SECOND_EDGE
			Rising and falling edges	3	CAPTURE_TRIG_BOTH_EDGE
u1_use_port1	unsigned char	ID of the port to be used	Specifies the ID of the port to be used. For respective facilities, see Table 6.3, Settings of Ports for Driver Facilities.		For defined values and names, see Table 6.4, Macro-Defined Names of Ports.
u1_use_port2	unsigned char	ID of the port to be used			
u1_use_port3	unsigned char	ID of the port to be used			
u1_use_port4	unsigned char	ID of the port to be used			

Table 6.3 Settings of Ports for Driver Facilities

		Member Name of Port			
		u1_use_port1	**u1_use_port2**	**u1_use_port3**	**u1_use_port4**
Name Defined for Facility	**F_ESIG1**	Excitation signal output	—	—	—
	F_ESIG12	Excitation signal output	Synthesized signal output	—	—
	F_CSIG	Angle error correction signal output	—	—	—
	F_PHASE_A	Phase adjustment signal A output	—	—	—
	F_PHASE_B	Phase adjustment signal B output	—	—	—
	F_CAPTURE	Angle signal input	—	—	—
	F_CSIG_UPD_TIMER	—	—	—	—
	F_RDC_COM(RSPI)	Clock output	Data reception	Data transmission	Slave selection
	F_RDC_COM(SCI)	Clock output	Data transmission	Data reception	—
	F_RDC_CLK	RDC clock output	—	—	—

Table 6.4 Macro-Defined Names of Ports

Macro-Defined Name	Defined Value	Peripheral Module
P_P02	2	MTU3_9: MTIOC9D
P_P10	3	MTU3_9: MTIOC9B
P_P11	4	TMO3
P_P20	5	MTU3_9: MTIOC9C
P_P21	6	MTU3_9: MTIOC9A
P_P22	7	TMO4, MISOA
P_P23	8	TMO2, MOSIA
P_P24	9	TMO6, RSPCKA
P_P30	10	MTU3_0: MTIOC0B, SSLA0
P_P31	11	MTU3_0: MTIOC0A, SSLA1
P_P32	12	TMO6, SSLA2
P_P33	13	TMO0, SSLA3
P_P82	14	TMO4, SCK6
P_P90	15	MTU3_7: MTIOC7D
P_P91	16	MTU3_7: MTIOC7C
P_P92	17	MTU3_6: MTIOC6D
P_P93	18	MTU3_7: MTIOC7B
P_P94	19	MTU3_7: MTIOC7A
P_P95	20	MTU3_6: MTIOC6B
P_PA0	21	TMO2, MTU3_6: MTIOC6C, SSLA3
P_PA1	22	TMO4, MTU3_6: MTIOC6A, SSLA2
P_PA2	23	TMO7, MTU3_2: MTIOC2B, SSLA1
P_PA3	24	MTU3_2: MTIOC2A, SSLA0
P_PA4	25	MTU3_1: MTIOC1B, SCK6, RSPCKA
P_PA5	26	MTU3_1: MTIOC1A, SMISO6, MISOA
P_PB0	27	TMO0, MTU3_0: MTIOC0D, SMOSI6, MOSIA
P_PB1	28	MTU3_0: MTIOC0C, SMISO6
P_PB2	29	MTU3_0: MTIOC0B, SMOSI6
P_PB3	30	MTU3_0: MTIOC0A, SCK6, RSPCKA
P_PD0	31	TMO6, RSPCKA
P_PD1	32	TMO2, MISOA
P_PD2	33	TMO4, SCK5, MOSIA
P_PD3	34	TMO0, SMOSI1
P_PD4	35	SCK1
P_PD5	36	SMISO1
P_PD6	37	TMO1, MTU3_9: MTIOC9C, SSLA0
P_PD7	38	MTU3_9: MTIOC9A, SMOSI5, SSLA1
P_PE0	39	MTU3_9: MTIOC9B, SMISO5, SSLA2
P_PE1	40	TMO5, MTU3_9: MTIOC9D, SSLA3
P_PB6	41	SMISO5
P_PB5	42	SMOSI5
P_PB7	43	SCK5
P_P80	44	SMISO6
P_P81	45	SMOSI6
P_NO_USE	0xff	When the port is not used

6.3.2 Structure for R_RSLV_SetSystemInfo

The argument of the R_RSLV_SetSystemInfo API function is an ST_SYSTEM_PARAM structure defined as shown below.

API function: R_RSLV_SetSystemInfo (ST_SYSTEM_PARAM *rdc_sys_param)

Table 6.5 Structure Definitions for R_RSLV_SetSystemInfo

Member Name	Type	Description		Defined Value	Macro-Defined Name
u1_mcu_type	unsigned char	MCU type	RX24T-R5F524TAADFM	1	MCU_TYPE_R5F524TAADFM
			RX24T-R5F524TAADFP	2	MCU_TYPE_R5F524TAADFP
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 and capture timer start flag	SYNC_START_NONE	0	MTU_SYNC_START_NONE
			SYNC_START_DISABLE	0	MTU_SYNC_START_DISABLE
			SYNC_START_ENABLE	1	MTU_SYNC_START_ENABLE
u1_motor_kind	unsigned char	Motor type	BLDC type	1	MOTOR_BLDC
			Stepper type	2	MOTOR_STM
u1_extension_use	unsigned char	For future extension		0	R_EXT_INACTIVE (fixed)*

Note: * Always set this member to 0.

6.3.3 Structure for R_RSLV_GetRdcDrvSettingInfo

The argument of the R_RSLV_GetRdcDrvSettingInfo API function is an ST_RDC_DRV_SETTING_INFO structure defined as shown below.

API function: R_RSLV_GetRdcDrvSettingInfo (ST_RDC_DRV_SETTING_INFO *rdc_setting_info)

Table 6.6 Structure Definitions for R_RSLV_GetRdcDrvSettingInfo

Member Name	Type	Description			Remark
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 input capture 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 return value of the R_RSLV_ADJST_GainPhase API function is an st_adjst_gainphase_return_t structure defined as shown below.

API function: st_adjst_gainphase_return_t R_RSLV_ADJST_GainPhase (unsigned char u1_call_state)

Table 6.7 Structure Definitions 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 a 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 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 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 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
	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	—	—

Table 6.7 Structure Definitions for R_RSLV_ADJST_GainPhase (2/2)

Structure	Member Name	Type	Description		Defined Value	Macro-Defined Name
st_adjst_gainphase_return_t (return value)	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 Structure for R_RSLV_ADJST_Carrier

The return value and argument of the R_RSLV_ADJST_Carrier API function are, respectively, st_adjst_carrier_return_t and st_adjst_carrier_arg_t structures defined as shown below.

API function: st_adjst_carrier_return_t R_RSLV_ADJST_Carrier (st_adjst_carrier_arg_t arg_value)

Table 6.8 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	—
				Adjustment completed	0 to 5	—
				Terminated with an error	User-set value	—
	res_csig_shift	unsigned short	Adjustment result: shift amount	Adjustment in progress	0xFF	—
				Adjustment completed	0 to 999	—
				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	 0 to 199 0 to 99	—
				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

6.3.6 Structure for R_RSLV_ADJST_SetPtrFunc

The argument of the R_RSLV_ADJST_SetPtrFunc API function is an st_ptr_func_arg_t structure defined as shown below.

API function: void R_RSLV_ADJST_SetPtrFunc (st_ptr_func_arg_t *ptr_arg)

Table 6.9 Structure Definitions 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	Speed reference value for calibration to compensate for errors	rpm	—	—

6.3.7 Structure for R_RSLV_DiscDetection_Seq

The argument of the R_RSLV_DiscDetection_Seq API function is an st_rdc_ddmnt_arg_t structure defined as shown below.

API function: unsigned char R_RSLV_DiscDetection_Seq (st_rdc_ddmnt_arg_t arg_value)

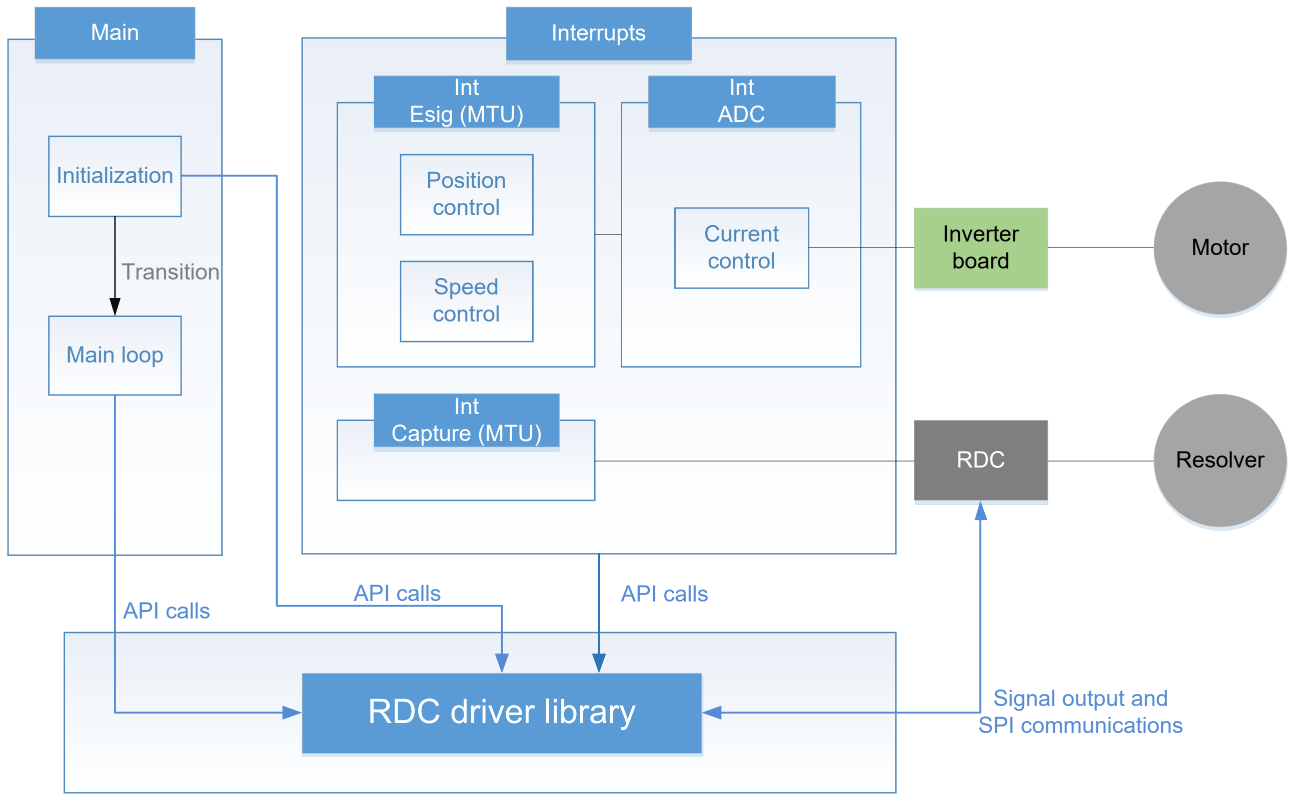
Table 6.10 Structure Definitions for R_RSLV_DiscDetection_Seq

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

7.1 Overview

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



Int: Interrupt processing
Esig: Excitation signal processing

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 (in response to input capture interrupts) 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.

7.2 Initialization

7.2.1 Initialization Procedure

Figure 7.2 shows the initialization flow.

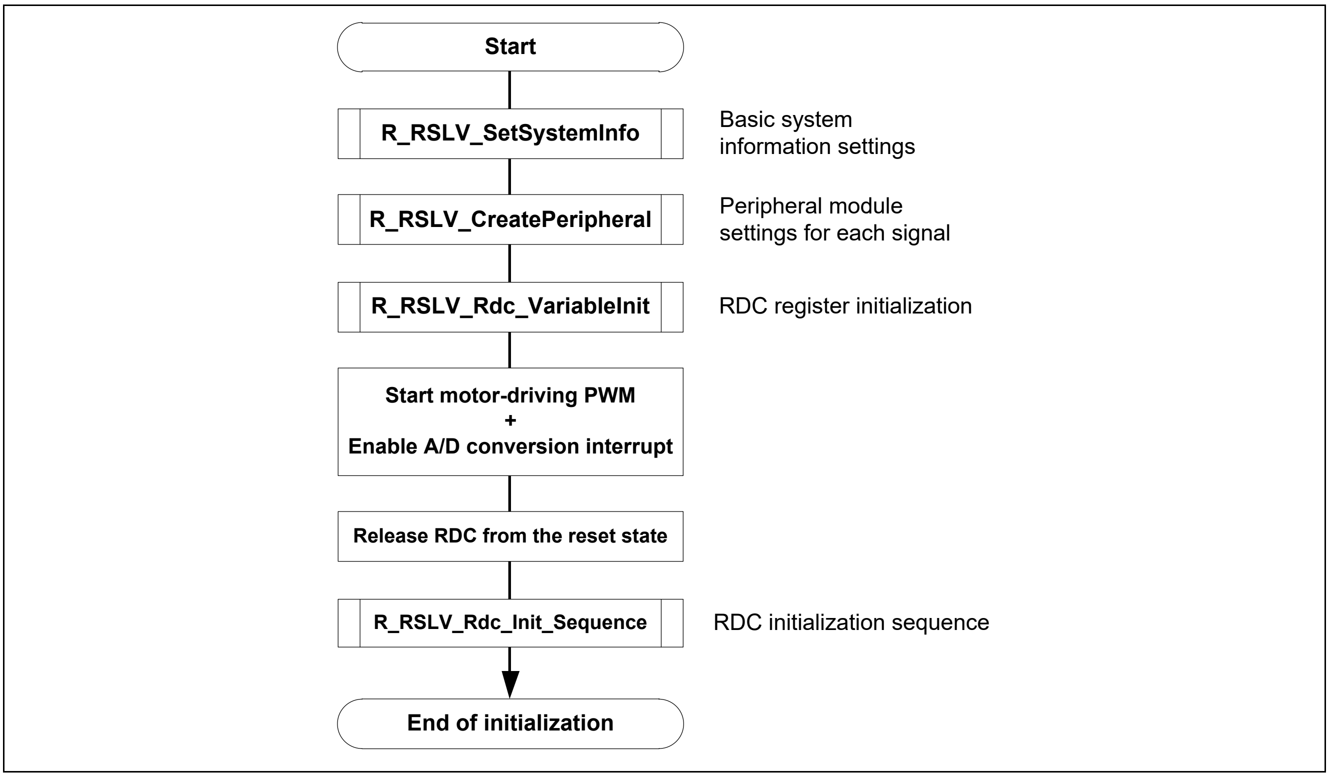


Figure 7.2 Initialization Flow

Create code for the above flow of initialization in the user application program which is to incorporate this driver.

7.2.2 Details of Initialization Processing

- Basic system information settings (R_RSLV_SetSystemInfo)
  Set basic information for using this driver. For information to be set, see section 6.3.2, Structure for R_RSLV_SetSystemInfo.

- Peripheral module settings for each signal (R_RSLV_CreatePeripheral)
  Set up the peripheral modules to be used for the signals that control the RDC. Assign peripheral modules to all facilities of this driver by using this API function. There is no constraint on the order of facility settings. For peripheral modules that can be assigned to each facility of this driver, see section 5.3, List of Possible Combinations of Peripheral Modules and Driver Facilities.

- RDC register initialization (R_RSLV_Rdc_VariableInit)
  This API function initializes each RDC register according to the specifications of the resolver sensor. Determine the initial value of each register according to the specifications of the resolver sensor to be connected and set the value through this API function. See section 6.2.29, API Function for Initializing Variables for RDC Communications, for the registers to be initialized.

- RDC initialization sequence (R_RSLV_Rdc_Init_Sequence)
  This API function implements the procedure for initializing the RDC. This API function takes the form of a state machine and so must be called periodically. Furthermore, a call of this API function must only be made at least 30 ms after deassertion of the reset signal for the RDC, which is in accordance with the RDC specifications (see section 4.3, Power-up Sequences and Reset Operation, in the RAA3064002GFP/RAA3064003GFP Renesas ICs Resolver-to-Digital Converters User's Manual: Hardware). When the initialization sequence is completed, RSLV_MD_OK is set in the argument init_status of this API function.

7.2.3 Sample Code

The following shows sample code.

```
/*******************************************************************************
* Function Name: r_RESOLVER_init_peripheral_sample
* Description  : Sample code to initialize RDC Driver Peripherals
* Arguments    : None
* Return Value : None
*******************************************************************************/
static void r_RESOLVER_init_peripheral_sample(void)
{
    ST_SYSTEM_PARAM   st_system_param;/* First-order variable for system initialization */
    ST_INIT_REG_PARAM rdc_init_param;  /* First-order variable for peripheral module initialization */

    /* System initialization settings */
    /* RX24T 100pin */
    st_system_param.u1_mcu_type = MCU_TYPE_R5F524TAADFP;
    /* ESig freq 5kHz */
    st_system_param.u1_esig_freq = R_ESIG_SET_FREQ_5K;
    /* CSig freq 200kHz */
    st_system_param.u1_csig_freq = R_CSIG_SET_FREQ_200K;
    /* Update Count 2 times*/
    st_system_param.u1_csig_upd_duty_cycle = R_CSIG_SET_DCNT_02;
    /* Use MTU synchronize start */
    st_system_param.u1_mtu3_sync_start = MTU_SYNC_START_ENABLE;
    /* Target Motor is a Stepper motor */
    st_system_param.u1_motor_kind = MOTOR_STM;
    /* Dither signal isn't used */
    st_system_param.u1_extension_use = R_EXT_INACTIVE;

    R_RSLV_SetSystemInfo(&st_system_param);

    // MTU3_9 ESig12       /* Channel 9 of MTU3 is used to output a synthesized excitation signal. */
    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);

    // MTU3_0 CSig     /* Channel 0 of MTU3 is used to output an angle error correction signal. */
    rdc_init_param.u1_sel_reg_type    = T_MTU3_0;
    rdc_init_param.u1_sel_reg_func    = F_CSIG;
    rdc_init_param.u1_sel_int_flg     = INT_DISABLE;
    rdc_init_param.u1_sel_int_priorty = 0;
    rdc_init_param.u1_capture_trig    = CAPTURE_TRIG_NONE;
    rdc_init_param.u1_use_port1       = P_P31;
    rdc_init_param.u1_use_port2       = 0xFF;      // Not used
    rdc_init_param.u1_use_port3       = 0xFF;      // Not used
    rdc_init_param.u1_use_port4       = 0xFF;      // Not used
    R_RSLV_CreatePeripheral(&rdc_init_param);

    /***** Repeat necessary peripheral module settings (omitted in this example). *****/

    /* Initialize valiables related to Resolver Converter Control */
    R_RSLV_Rdc_VariableInit((unsigned char*)s_u1_rdc_init_data);

    /* Get Resolver Setting Information *//* Acquisition of information setting */
    R_RSLV_GetRdcDrvSettingInfo(&s_st_rslv_drv_info);
```

```
    /* Start motor-driving PWM and A/D conversion. */
    R_MTU3_C3_Start();
    R_S12AD0_Start();

    /* Go to the main loop. */
}

void main( void )
{
    unsigned char   u1_state_rdc_init = 0U;        /* Variable for RDC initialization order */
    unsigned short  u2_rdc_result = RSLV_MD_BUSY1;/* Variable for RDC initialization API argument */

    r_RESOLVER_init_peripheral_sample();

    /* Main loop */
    while(1)
    {
        /* RDC communications processing (including initialization) */
        {
            switch (u1_state_rdc_init)
            {
                default:
                    /* Do nothing */
                break;

                case 0:
                    /* Release the RDC from the reset state. */
                    u1_state_rdc_init = 1U;
                break;

                case 1:
                    /* Wait for 30 ms. */
                    if (30 ms elapses)
                    {
                        u1_state_rdc_init = 2U;
                    }
                break;

                case 2:
                    /* Perform RDC initialization processing. (Periodic call in the main loop) */
                    R_RSLV_Rdc_Init_Sequence(&u2_rdc_result);

                    /* When initialization is completed, RSLV_MD_OK is returned. */
                    if (RSLV_MD_OK == u2_rdc_result)
                    {
                        u1_state_rdc_init = 3U;
                    }
                break;
            }

            /* Always perform RDC communications processing. */
            R_RSLV_Rdc_Communication();
        }
    }
}
```

7.3 Main Loop

Figure 7.3 shows an example of implementing the main loop.

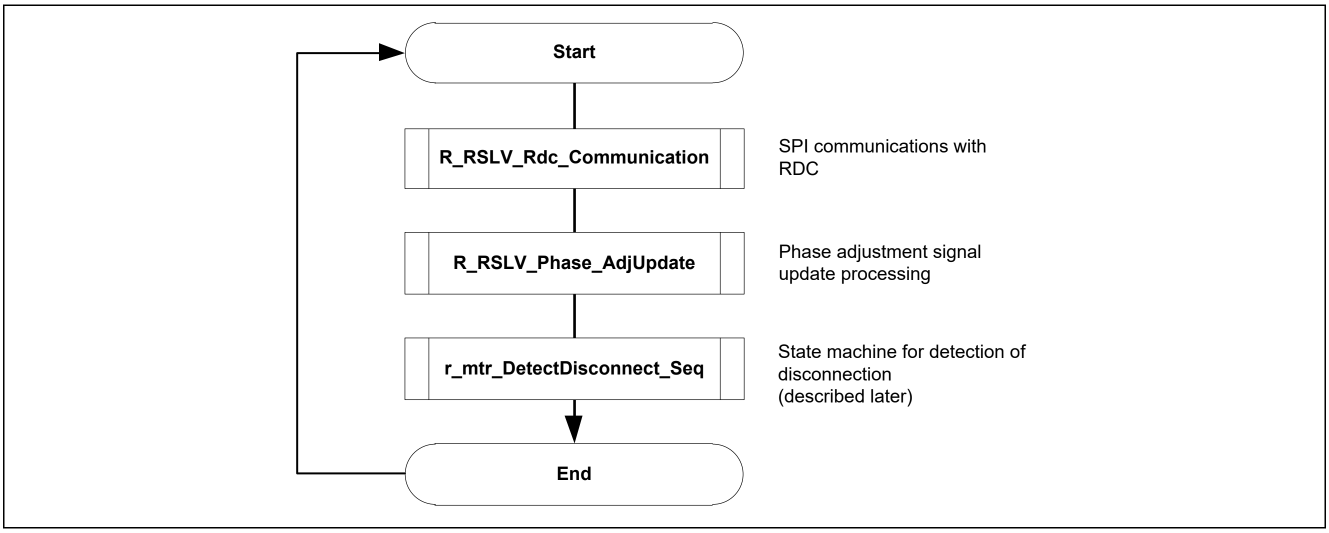


Figure 7.3 Example of Implementing the Main Loop

In the main loop, call the RDC communications processing and the phase adjustment signal update processing 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.

7.4 Output of the Excitation Signal

7.4.1 Example of Using API Functions

Figure 7.4 shows a block diagram of implementation by using API functions related to the output of the excitation signal.

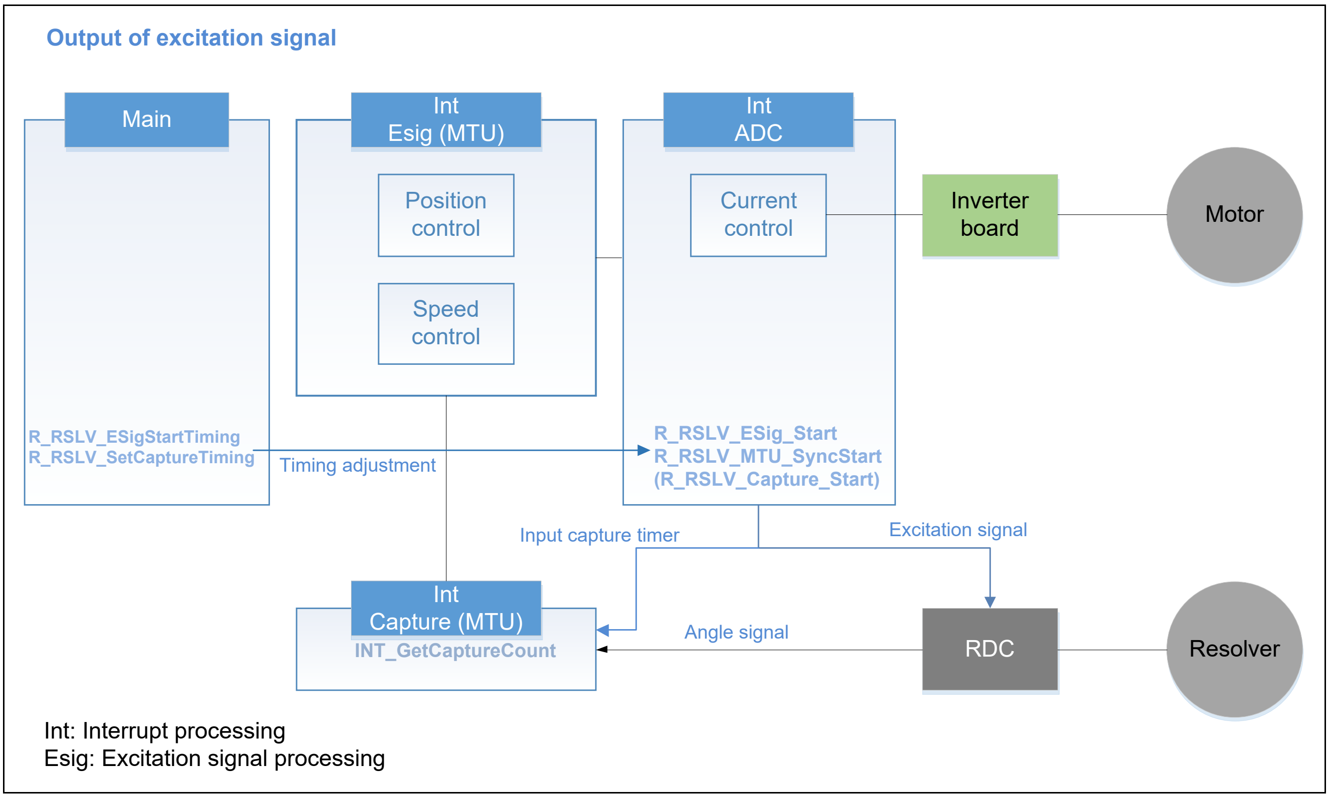


Figure 7.4 Example of Implementation by Using API Functions Related to the Output of the Excitation Signal

7.4.2 Details of the Output of the Excitation Signal

R_RSLV_CSig_Start (section 6.2.20, API Function for Starting the Output of the Excitation Signal) is used to output the excitation signal. This function can be called any time after the peripheral module to be used for the excitation signal is set up by R_RSLV_CreatePeripheral.

To start the excitation signal timer and the input capture timer simultaneously (if it is specified by the API function for specifying system information (section 6.2.2)), call R_RSLV_MTU_SyncStart to start counting by the input capture timer in synchronization with the output of the excitation signal. The MTU3 channel to be synchronized is selected by the argument of R_RSLV_MTU_SyncStart. For details, see section 6.2.5, API Function for Controlling Synchronous Starting of the MTU3 Timer Channels. When starting the timers simultaneously, call R_RSLV_ESig_Start and then call R_RSLV_MTU_SyncStart.

Figure 7.5 shows an example of using R_RSLV_ESigStatTiming.

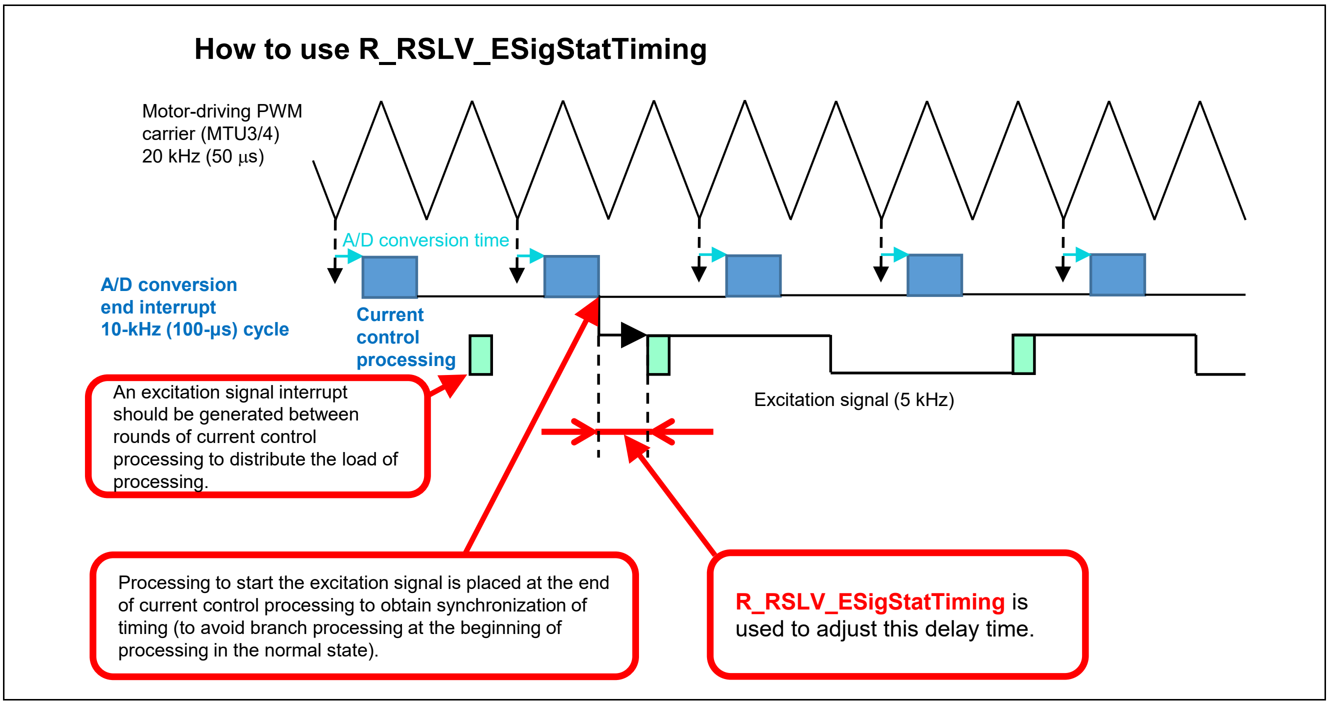


Figure 7.5 Example of Using R_RSLV_ESigStatTiming

The R_RSLV_ESigStartTiming function is used to adjust the excitation signal start timing. See the figure above.

Call R_RSLV_ESigStartTiming after setting up the peripheral module for the excitation signal by R_RSLV_CreatePeripheral and before calling R_RSLV_ESig_Start or R_RSLV_MTU_SyncStart.

Figure 7.6 shows an example of using R_RSLV_SetCaptureTiming.

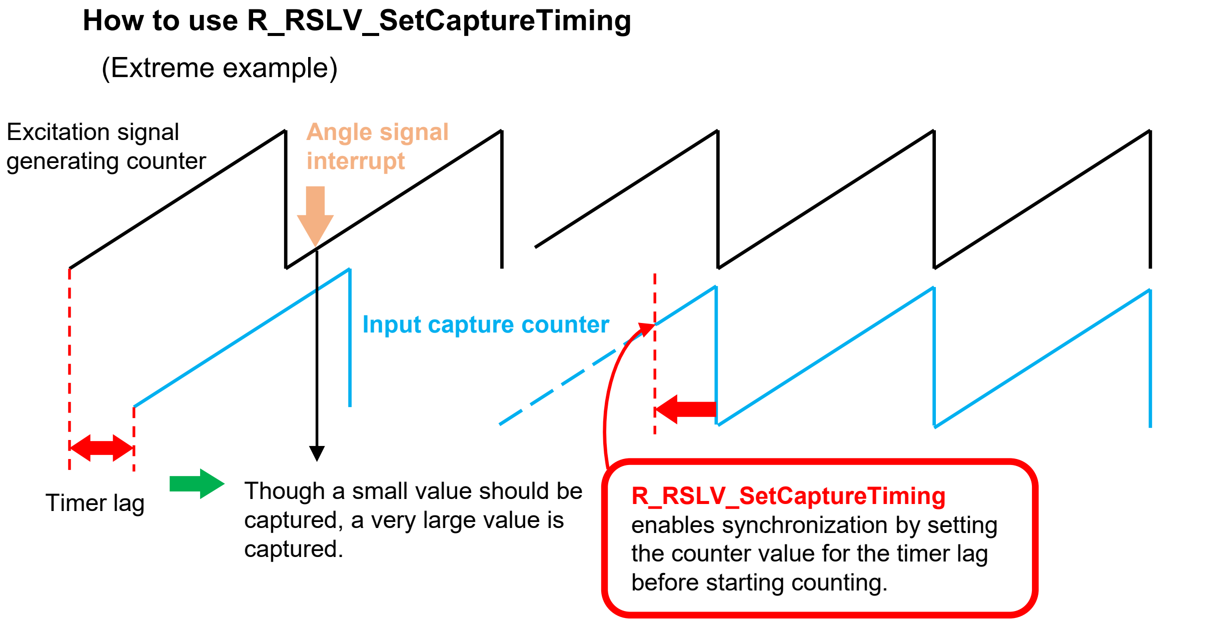


Figure 7.6 Example of Using R_RSLV_SetCaptureTiming

The R_RSLV_SetCaptureTiming function is used to synchronize the input capture counter with the excitation signal. See the figure above. Call R_RSLV_SetCaptureTiming after setting up the peripheral module for input capture by R_RSLV_CreatePeripheral and before calling R_RSLV_Capture_Start or R_RSLV_MTU_SyncStart.

7.4.3 Sample Code

The following shows sample code.

Example: When the excitation signal is generated by channel 9 of MTU3, input capture is handled by channel 6, and the timers are started simultaneously

```
#define    MTU_SYNC_START_CH    (MTU_SYNCSTART_BIT_MTU6 | MTU_SYNCSTART_BIT_MTU9)
unsigned char u1_flg_esig_started = 0U; /* Excitation signal start flag */

/* This example uses the A/D conversion end interrupt processing to start the output of the
signal as shown in Figure 7.5. */
#pragma interrupt r_s12ad_interrupt(vect=VECT(S12AD1,S12ADI1))
static void r_s12ad_interrupt(void)
{
    /* Perform phase current acquisition processing. */

    /* Perform current control processing. */

    /* Excitation signal starting block */
    if (0U == u1_flg_esig_started)
    {
        R_RSLV_ESigStartTiming(DEF_DELAY_ADJ_ESIG);     /* Excitation signal start timing*/
        R_RSLV_SetCaptureTiming(DEF_SFT_ADJ_ESIG);      /* Capture start timing */
        R_RSLV_ESig_Start();
        R_RSLV_MTU_SyncStart(MTU_SYNC_START_CH);
        u1_flg_esig_started = 1U;
    }
}
```

7.5 Output of the Phase Adjustment Signals

7.5.1 Example of Using API Functions

Figure 7.7 shows a block diagram of implementation by using API functions for outputting the phase adjustment signals.

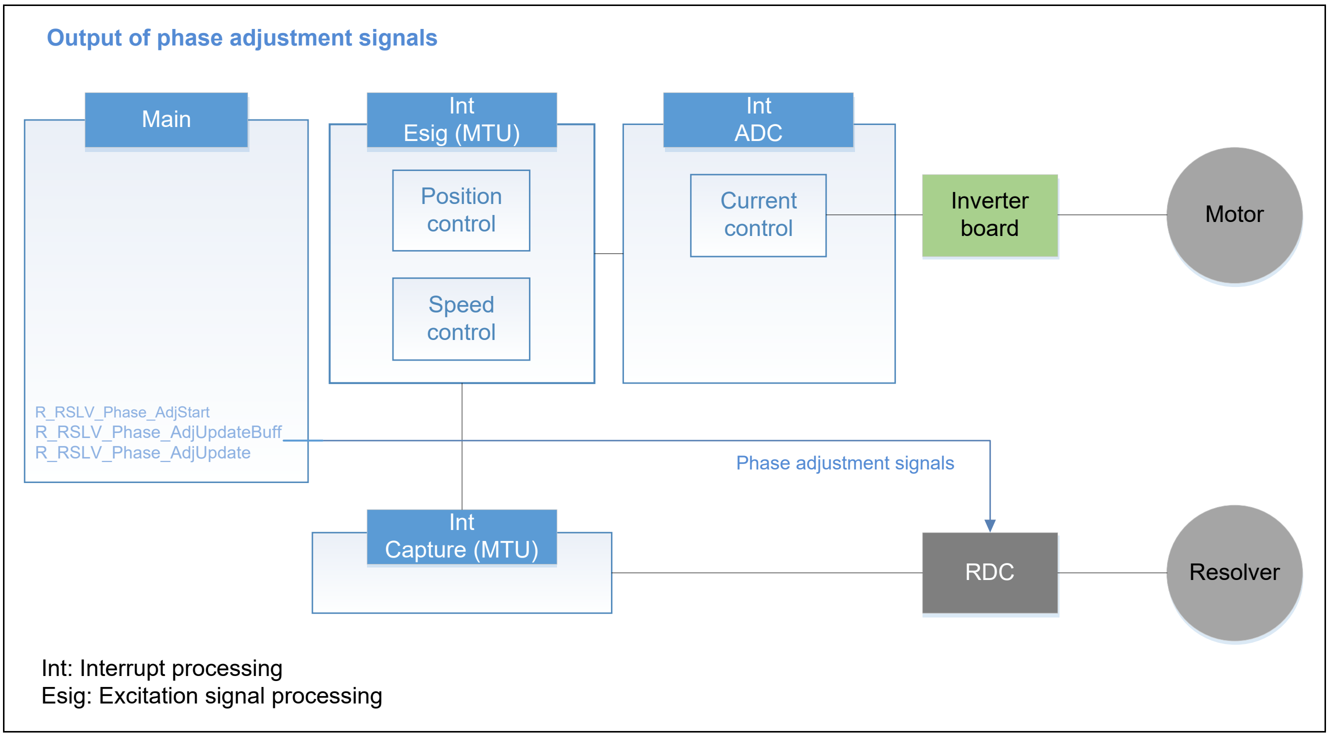


Figure 7.7 Example of Implementation by Using API Functions for Outputting the Phase Adjustment Signals

To output the phase adjustment signals, use the functions R_RSLV_Phase_AdjStart (section 6.2.24, API Function for Starting the Output of the Phase Adjustment Signals), R_RSLV_Phase_AdjUpdateBuff (section 6.2.26, API Function for Setting the Phase Adjustment Signal Duty Cycle in the Buffer), and R_RSLV_Phase_AdjUpdate (section 6.2.27, API Function for Setting the Phase Adjustment Signal Duty Cycle in the Register). Use R_RSLV_Phase_AdjUpdateBuff to update the duty cycle to be set for the phase adjustment signals. After that, call R_RSLV_Phase_AdjUpdate to change the duty cycle of the PWM output signal. Finally, call R_RSLV_Phase_AdjStart.

7.5.2 Sample Code

The following shows sample code.

Example: Phase A duty cycle: 65%, phase B duty cycle: 22%

```
/* The main loop processing shown in Figure 7.3 is arranged. */
unsigned char u1_flg_phase_started = 0U;  /* Phase adjustment signal start flag */

void main(void)
{
    /* Initialization processing */

    /* Main loop */
    while (1)
    {
        /* RDC communications processing */

        /* 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 7.8 shows a block diagram of implementation by using API functions for outputting the angle error correction signal.

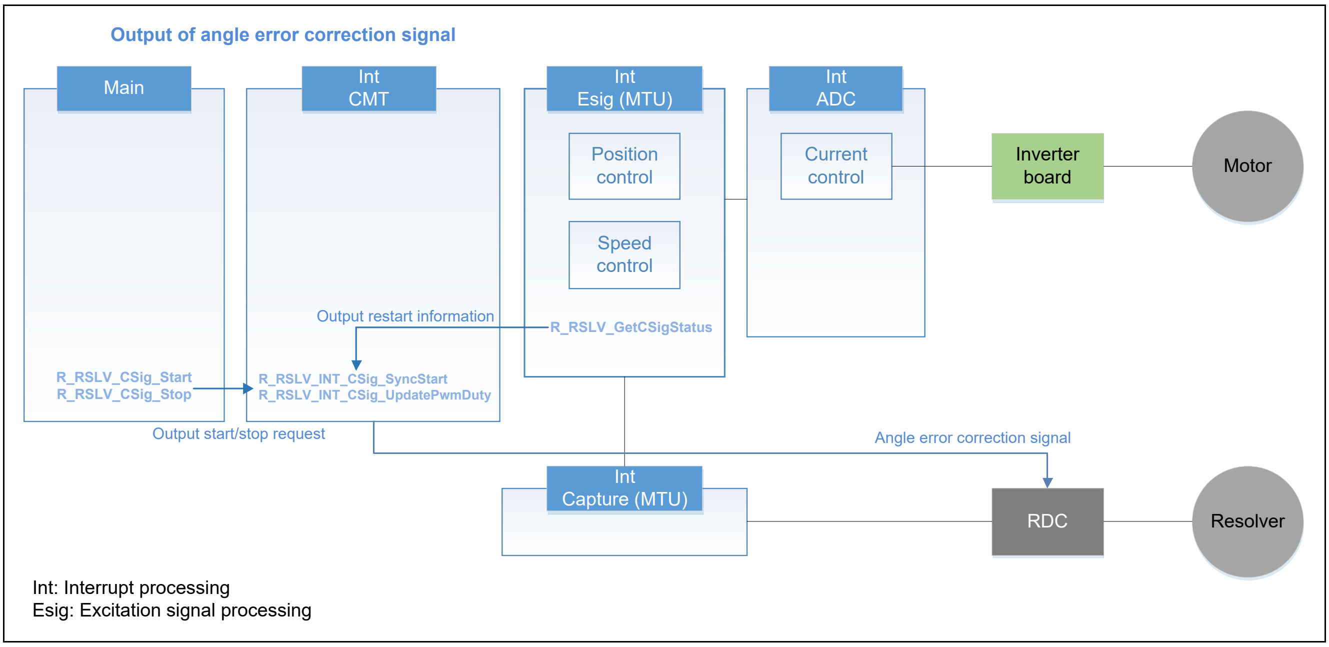


Figure 7.8 Example of Implementation by Using API Functions for Outputting the Angle Error Correction Signal

To output the angle error correction signal, use the API functions R_RSLV_CSig_Start (section 6.2.7, API Function for Starting the Output of the Angle Error Correction Signal), R_RSLV_INT_CSig_SyncStart (section 6.2.10, API Function for Synchronously Starting the Angle Error Correction Signal), and R_RSLV_INT_CSig_UpdatePwmDuty (section 6.2.9, API Function for Updating the Duty Cycle of the Angle Error Correction Signal).

7.6.2 Details of the Output of the Angle Error Correction Signal

Figure 7.9 shows details about the output of the angle error correction signal.

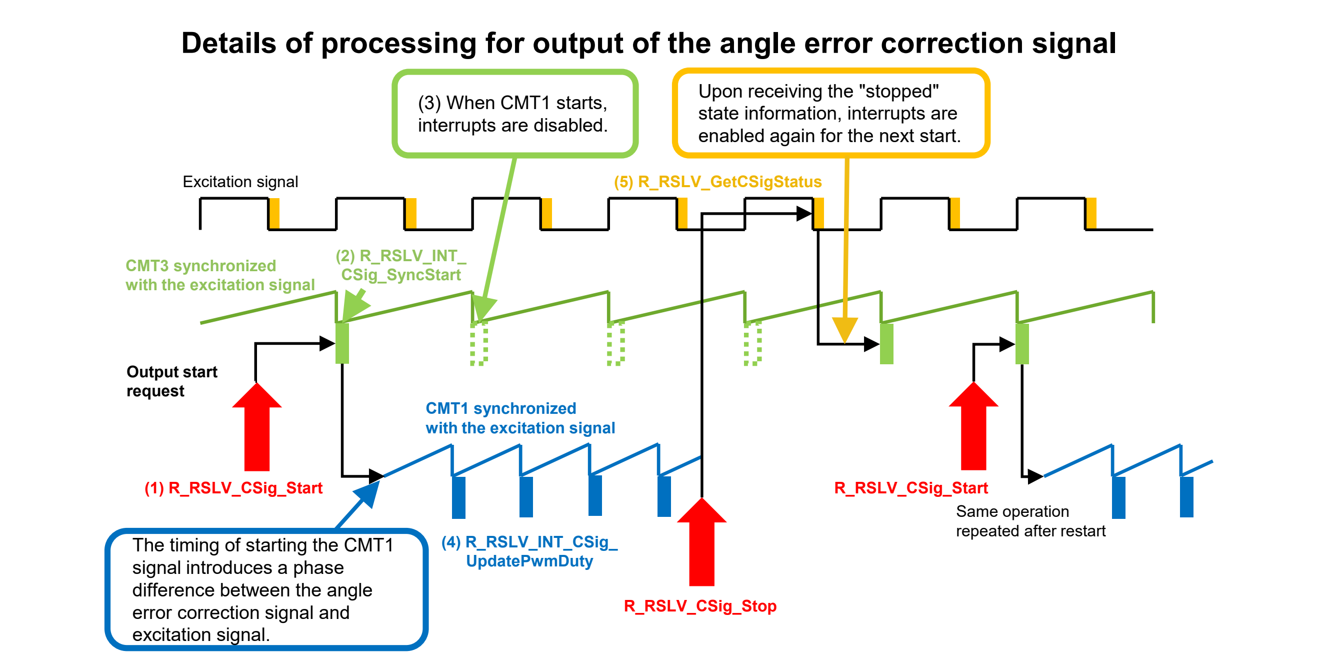


Figure 7.9 Details of Implementation of the Angle Error Correction Signal

(1) Use R_RSLV_CSig_Start to issue a request to start outputting the angle error correction signal. R_RSLV_CSig_Start can be called any time after the peripheral modules for the angle error correction signal are set by R_RSLV_CreatePeripheral.

(2) To synchronize the angle error correction signal with the excitation signal, use the API function for synchronous starting (section 6.2.10, API Function for Synchronously Starting the Angle Error Correction Signal). Use this API function in the timer interrupt processing (CMT3 is used in this example) synchronized with the excitation signal as shown in Figure 7.9. When R_RSLV_CSig_Start is called, R_RSLV_INT_CSigStart starts output of the angle error correction signal.
The reason for calling R_RSLV_INT_CSig_SyncStart from processing by another timer instead of from the excitation signal interrupt processing is as follows: if an interrupt with a higher priority than the excitation signal interrupt is present, the interrupts may be in contention with each other. Processing for the higher-priority interrupt delays the call of the API function for synchronous starting which in turn delays switching of the angle error correction signal. Therefore, the correction signal is output with a different phase from that corresponding to the setting for phase. To obtain output of the correction signal according to the setting for phase, set the priority of the timer interrupt that calls the API function for synchronous starting higher than the priority of other interrupts.

(3) R_RSLV_INT_CSig_SyncStart starts output of the angle error correction signal, and at the same time starts the timer for updating the PWM duty cycle in synchronization with the excitation signal. CMT1 is used as this timer in Figure 7.9. R_RSLV_CreatePeripheral selects the peripheral module to be used for F_CSIG_UPD_TIMER. R_RSLV_INT_CSig_SyncStart will not require a further call after this processing to start, so disable the interrupt used for issuing the call of R_RSLV_INT_CSig_SyncStart.

(4) Call the API function for updating the duty cycle (section 6.2.9, API Function for Updating the Duty Cycle of the Angle Error Correction Signal) from the duty cycle updating timer interrupt processing. For details of operation using this API function, see Figure 3.1, Initial Setting Sequence.

(5) When changing the settings of the angle error correction signal, call R_RSLV_CSig_Stop and R_RSLV_CSig_Start in that order. Also call R_RSLV_GetCSigStatus from the excitation signal interrupt processing (Figure 7.9, Details of Implementation of the Angle Error Correction Signal). When the state information acquired by this API function is not E_OUTPUT_SIGNAL_ON, enable the interrupt for calling the API function for synchronous starting again.

7.6.3 Sample Code

The following shows sample code.

(1) Excitation signal synchronous timer interrupt processing (CMT3 in Figure 7.9)

```
#pragma interrupt r_Config_CMT3_cmi3_interrupt(vect=VECT(CMT3,CMI3))
static void r_Config_CMT3_cmi3_interrupt(void)
{
   /* Start the angle error correction signal in synchronization with the excitation signal.*/
   R_RSLV_INT_CSig_SyncStart();

   R_Config_CMT3_Set_IntEnable( OFF ); /* Disable the interrupt after the timer starts. */
}
```

(2) PWM duty cycle updating interrupt processing (CMT1 in Figure 7.9)

```
#pragma interrupt r_cmt_cmi1_interrupt(vect=VECT(CMT1,CMI1),enable)
static void r_cmt_cmi1_interrupt( void )
{
    R_RSLV_INT_CSig_UpdatePwmDuty();
}
```

(3) Excitation signal interrupt processing (channel 9 of MTU3 in this example)

```
static uint8_t s_u1_get_csig_state = E_OUTPUT_SIGNAL_OFF;

#pragma interrupt r_mtr_Esig_Intr_process(vect=VECT(MTU9,TGIC9))
static void r_mtr_Esig_Intr_process( void )
{
    /* When the correction signal is disabled (not enabled),
       CSig_SyncStart should be called to enable the signal again. */
    R_RSLV_GetCSigStatus( &s_u1_get_csig_state );
    if(E_OUTPUT_SIGNAL_ON != s_u1_get_csig_state)
    {
       R_Config_CMT3_Set_IntEnable( ON ); /* Enable the interrupt again
                                              after the timer stops. */
    }
}
```

7.7 Input of Angle Signal

7.7.1 Example of Using API Functions

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

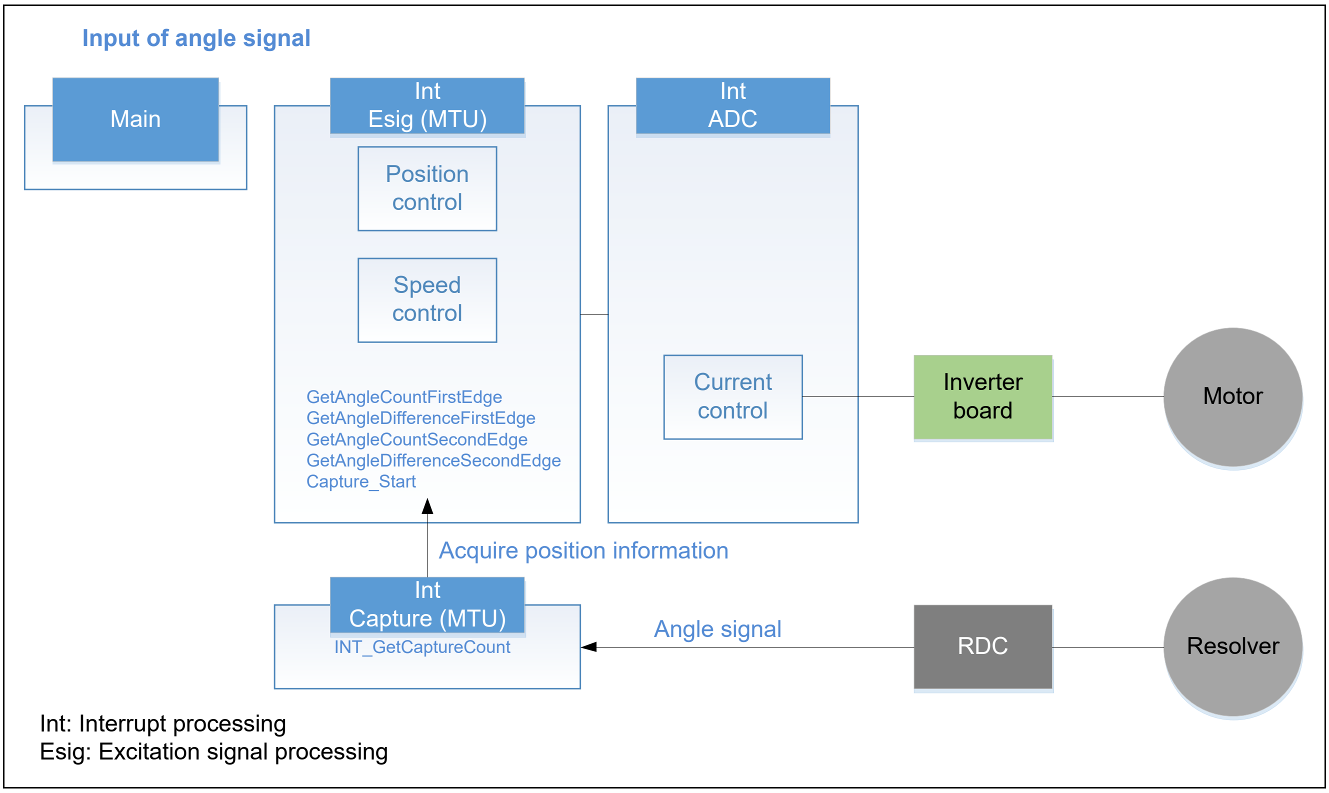


Figure 7.10 Example of Implementation by Using API Functions for Inputting the Angle Signal

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.

(1) Angle signal interrupt processing (Channel 6 of MTU3 is used as an input capture counter in this example.)

```
#pragma interrupt r_mtr_InputCapture_Intr_process(vect=VECT(MTU6,TGIB6),enable)

static void r_mtr_InputCapture_Intr_process( void )
{
    /* Get current capture count. */
    R_RSLV_INT_GetCaptureCount();
}
```

(2) Position information capture processing

```
/* These functions are usually called prior to position control. */
R_RSLV_GetAngleCountFirstEdge(&temp_cnt);
R_RSLV_GetAngleDifferenceFirstEdge(&temp_diff);

/* Apply position control. */
```

7.8 Automatic Adjustment of the Gain and Phase

7.8.1 Example of Using API Functions

Figure 7.11 shows a block diagram of implementation using the API functions for automatic adjustment of the gain and phase.

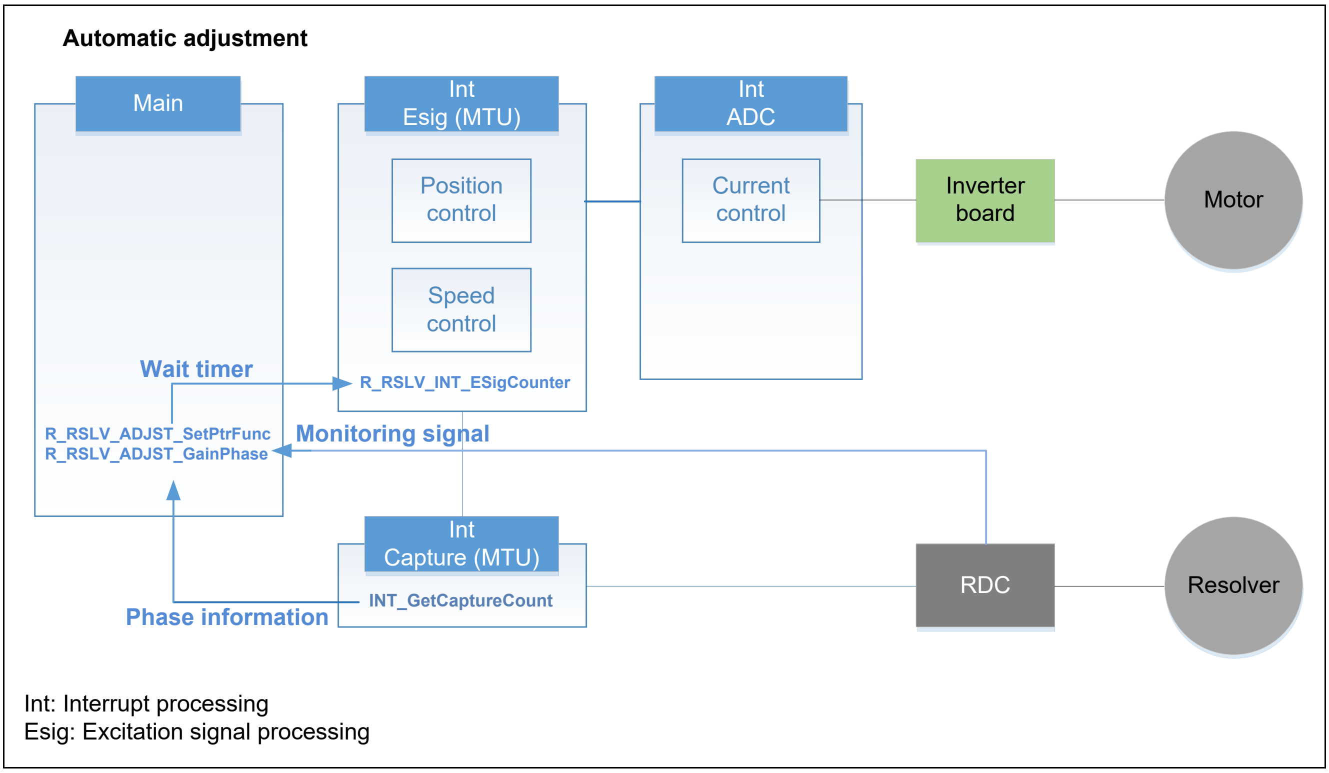


Figure 7.11 Example of Implementation by Using API Functions for Automatic Adjustment of the Gain and Phase

To execute the automatic adjustment of the gain and phase, use the functions R_RSLV_ADJST_SetPtrFunc (section 6.2.46, API Function for Setting the Pointer to the User-Created Callback Function), R_RSLV_ADJST_GainPhase (section 6.2.44, API Function for Adjusting the Gain and Phase of the Resolver Signals), and R_RSLV_INT_ESigCounter (section 6.2.23, API Function for Counting the Wait Time).

In this adjustment, R_RSLV_INT_GetCaptureCount (section 6.2.13, API Function for Acquiring the Input Capture 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 7.12 shows an example of implementing adjustment of the gain and phase.

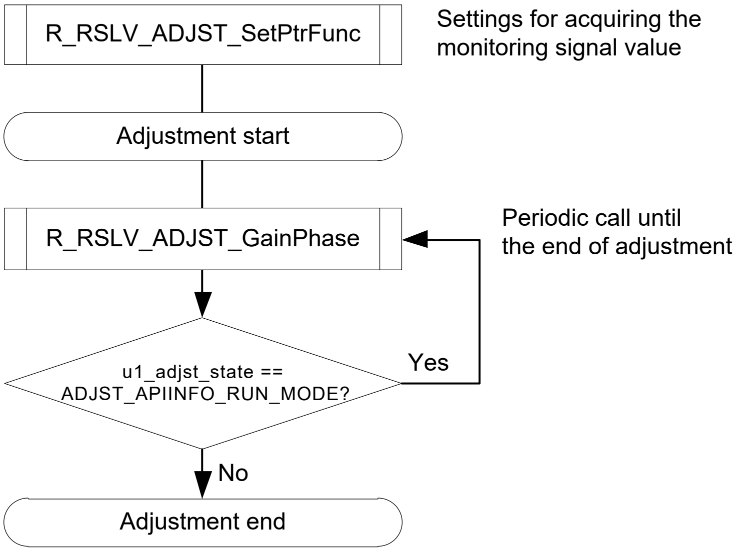


Figure 7.12 Gain and Phase Adjustment Sequence

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.

The API function R_RSLV_ADJST_GainPhase for adjusting the gain and phase must be repeatedly called until the adjustment is completed.

(a) Starting adjustment

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

(b) 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 variable u1_adjst_state becomes ADJST_APIINFO_END_USER_STOP (13U).

(c) 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_GainPhase (1/2).

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.

(1) Callback function settings (input of monitoring signal is assigned to AN103)

```
/*****************************************************************************
* Function Name: R_S12AD_GetMntOut
* Description  : Get A/D converted data at RDC monitor output
* Arguments    : void
* Return Value : Converted A/D data at RDC monitor output
*****************************************************************************/
uint16_t R_S12AD_GetMntOut( void )
{
    return ((uint16_t)S12AD1.ADDR3);
}
/*****************************************************************************
* Function Name: R_S12AD_StartByAdjst
* Description  : A/D conversion control I/F
* Arguments    : ctrl -
*                A/D conversion start/stop (1:Start 0:Stop)
* Return Value : void
*****************************************************************************/
void R_S12AD_StartByAdjst( uint8_t ctrl )
{
    S12AD1.ADCSR.BIT.ADST = ctrl;
}

/*****************************************************************************
 * Function Name: R_S12AD_ChgSettingForAdjst
 * Description  : Change A/D settings for adjustment
 * Arguments    : None
 * Return Value : None
 ****************************************************************************/
void R_S12AD_ChgSettingForAdjst( void )
{
    S12AD.ADCSR.BIT.TRGE = 0;          /* Disable start of A/D converter. */
    S12AD.ADCSR.BIT.ADST = 0;
    S12AD1.ADCSR.BIT.TRGE = 0;         /* Disable start of A/D converter. */
    S12AD1.ADCSR.BIT.ADST = 0;
    S12AD2.ADCSR.BIT.TRGE = 0;         /* Disable start of A/D converter. */
    S12AD2.ADCSR.BIT.ADST = 0;
    IEN(S12AD1, S12ADI1) = 0;          /* Disable interrupts. */
    IR(S12AD1, S12ADI1) = 0;           /* Clear interrupt flags. */

    S12AD.ADANSA0.WORD = 0x0000;   /* Cancel A/D converter channel selection for AN0XX. */

  /* Cancel A/D converter channel selection for AN1XX except AN103. */
    S12AD1.ADANSA0.WORD = 0x0008;

    S12AD2.ADANSA0.WORD = 0x0000;  /* Cancel A/D converter channel selection for AN2XX. */

    S12AD1.ADCSR.WORD = _0000_AD_DBLTRIGGER_DISABLE | _0000_AD_SYNC_TRIGGER |
                        _0200_AD_SYNCASYNCTRG_ENABLE |
                        _1000_AD_SCAN_END_INTERRUPT_ENABLE |
                        _4000_AD_CONTINUOUS_SCAN_MODE;
}
```

```
/***********************************************************************
 * Function Name: R_S12AD_ResetSettigForNormal
 * Description  : Reset general A/D settings
 * Arguments    : None
 * Return Value : None
 ***********************************************************************/
void R_S12AD_ResetSettigForNormal( void )
{
    /* Select A/D converter channels AN100 and AN101 again. */
    S12AD1.ADANSA0.WORD = 0x0003;

    /* Select A/D converter channels AN2XX again. */
    S12AD2.ADANSA0.WORD = 0x0811;

    S12AD1.ADCSR.WORD = _0000_AD_DBLTRIGGER_DISABLE | _0000_AD_SYNC_TRIGGER |
                        _0200_AD_SYNCASYNCTRG_ENABLE |
                        _1000_AD_SCAN_END_INTERRUPT_ENABLE |
                        _0000_AD_SINGLE_SCAN_MODE;

    R_S12AD0_Start();
}

/***********************************************************************
* Function Name: r_mtr_init_adjst_interface
* Description  : Initialize interface functions and variables of 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 );
}
```

(2) Call of API function for adjusting gain and phase

```
/***************************************************************************
* Function Name: r_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)
***************************************************************************/
static uint8_t r_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. */
    if (ADJST_APIINFO_RUN_MODE == gp_api_ret.u1_adjst_state)
    {
        result = TRUE;
    }
    /* On completion of processing, the end of processing is reported. */
    else
    {
        result = FALSE;
    }

    return (result);
}
```

r_mtr_rdc_AdjstGainPhaseProcess is called periodically.

7.9 Automatic Adjustment of the Angle Error Correction Signal

7.9.1 Example of Using API Functions

Figure 7.13 shows an example of implementation by using API functions for automatic adjustment of the angle error correction signal.

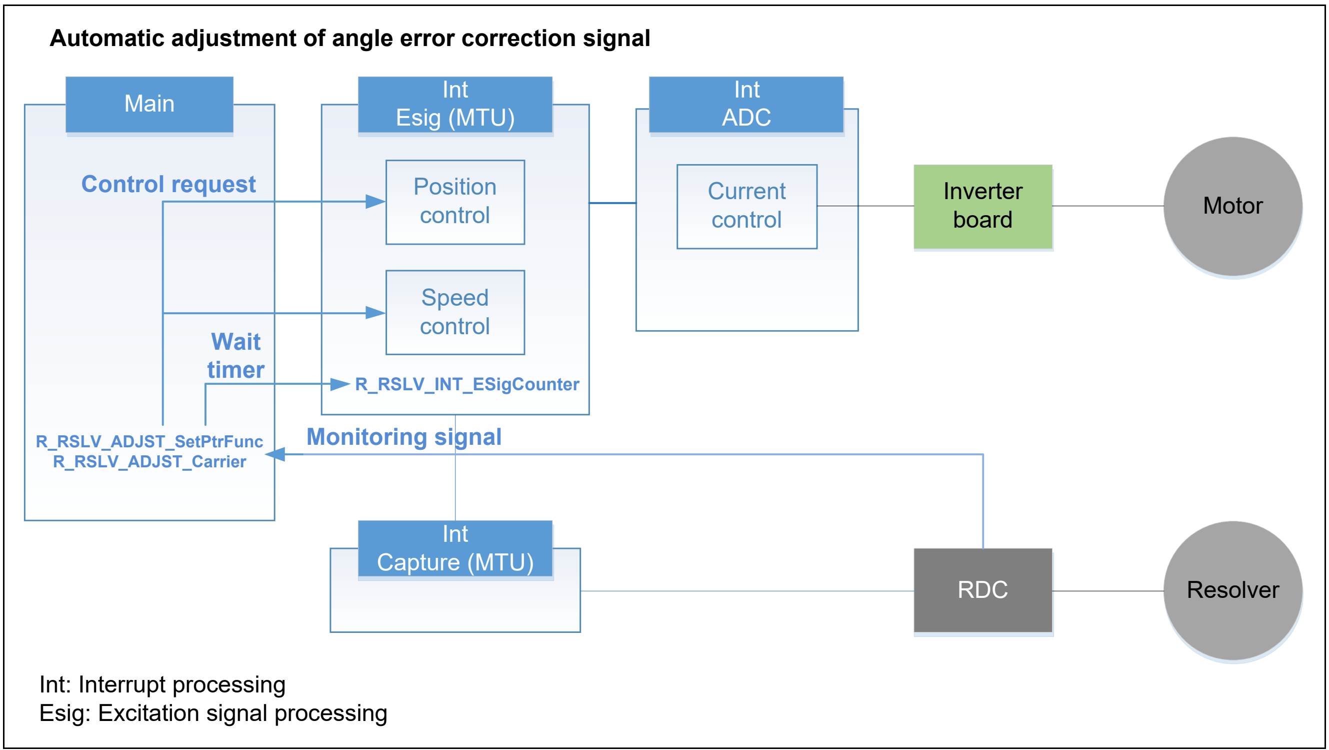


Figure 7.13 Example of Implementation by Using API Functions for Automatic Adjustment of the Angle Error Correction Signal

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

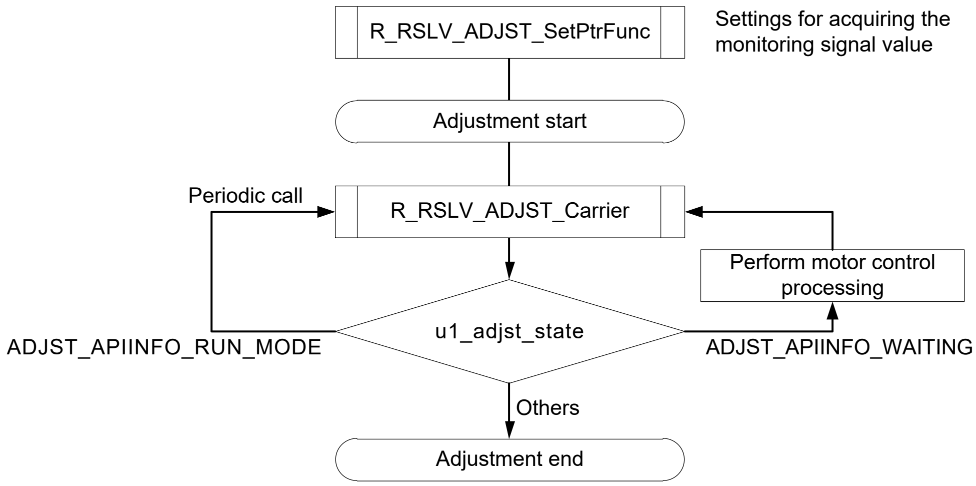


Figure 7.14 Angle Error Correction Signal Adjustment Sequence

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 u1_adjst_state value. When the adjustment requires the application of motor control, the return value becomes ADJST_APIINFO_WAITING.

Figure 7.15 shows the sequence between the caller (application) and the driver from the start of adjustment until the completion of adjustment.

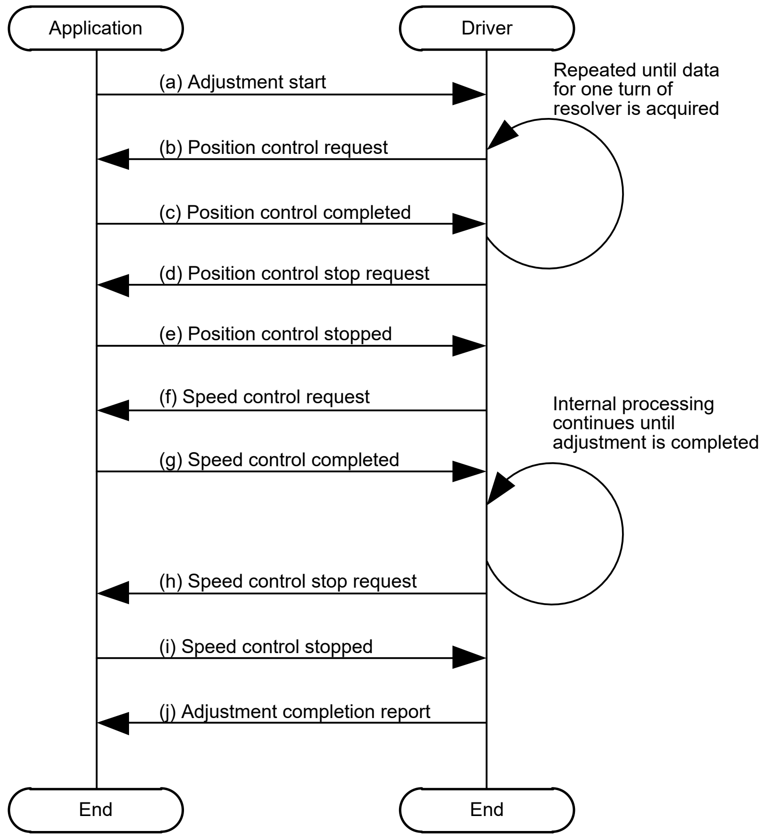


Figure 7.15 Angle Error Correction Signal Adjustment Sequence

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 Definitions for R_RSLV_ADJST_Carrier.

(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 request in step (b), 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 steps (b) to (c) 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 step (d).

(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 step (b).

(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 step (f).

(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 step (h).

(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 step (j).

(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 Definitions for R_RSLV_ADJST_Carrier.

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.

(1) Periodic call processing

```
/***************************************************************************
* 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 carrier 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:
            {
                /* Function for response to the motor control request */
                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);

}
```

(2) Function for response to the motor control request

```
/*******************************************************************************
* Function Name: r_mtr_ctrl_posspd_for_ccadjust_seq
* Description  : Control sequence for adjustment of angle error correction signal
* Arguments    : None
* Return Value : None
*******************************************************************************/
static void r_mtr_ctrl_posspd_for_ccadjust_seq( void )
{
    float temp_pos;
    r_mtr_rslv_err_flg_t temp_err_flg;

    /* Receive the request from adjustment process. */
    if (POSSPD_SEQ_NONE == s_u1_ctrl_posspd_seq)
    {
        /* Switch the execution state according to the return value. */
        switch (cc_api_ret.req_mtr_ctrl)
        {
            default:
                /* Do nothing. */
            break;

            /* Position control request */
            case ADJST_APIREQ_POS_CTRL:
                {
                   /* Start position control. */
                    s_u1_ctrl_posspd_seq = POSSPD_SEQ_POS_REF;
                }
            break;

            /* Speed control request */
            case ADJST_APIREQ_SPD_CTRL:
                {
                    /* Start speed control. */
                    s_u1_ctrl_posspd_seq = POSSPD_SEQ_SPD_REF;
                }
            break;

            /* Control stop request */
            case ADJST_APIREQ_POS_STOP:
            case ADJST_APIREQ_SPD_STOP:
                {
                    /* Start control stop. */
                    s_u1_ctrl_posspd_seq = POSSPD_SEQ_STOP;
                }
            break;
        }
    }

    switch (s_u1_ctrl_posspd_seq)
    {
        default:
        case POSSPD_SEQ_NONE:
            {
                /* Return value is always set to "COMPLETE". */
                cc_api_req.req_state = ADJST_USRINFO_COMPLETE;
```

```
            }
        break;

        /* Response to a position control request */
        case POSSPD_SEQ_POS_REF:
            {
                /* Receiving position reference value */
                temp_pos = cc_api_ret.mtr_ctrl_data;

                /* Return value is set to "PROCESSING". */
                cc_api_req.req_state = ADJST_USRINFO_PROCESSING;

                /* Start position control according to the reference value. */

                s_u1_ctrl_posspd_seq = POSSPD_SEQ_POS_WAIT;
            }
        break;

        /* State of waiting for the stability of position */
        case POSSPD_SEQ_POS_WAIT:
            {
               /* Wait until the rotor position is stabilized by position control.
 */
               /* After the rotor position is stable, the value for completion
                  of control is returned. */
                {
                   /* Request of adjustment process was completed. */
                   cc_api_req.req_state = ADJST_USRINFO_COMPLETE;
                   s_u1_ctrl_posspd_seq = POSSPD_SEQ_NONE;
                }
            }
        break;

        /* Response to a speed control request */
        case POSSPD_SEQ_SPD_REF:
            {
                cc_api_req.req_state = ADJST_USRINFO_PROCESSING;

                /* Start speed control according to the reference value. */

                s_u1_ctrl_posspd_seq = POSSPD_SEQ_SPD_WAIT;
            }
        break;

        /* State of waiting until the speed reaches the target */
        case POSSPD_SEQ_SPD_WAIT:
            {
                /* Wait until the speed is stabilized by speed control. */
                /* After the speed is stable, the value for completion of control
                   is returned. */
                {
                       /* Request of adjustment process was completed. */
                       cc_api_req.req_state = ADJST_USRINFO_COMPLETE;
                       s_u1_ctrl_posspd_seq = POSSPD_SEQ_NONE;
                }
        break;
```

```
        /* Response to a control stop request */
        case POSSPD_SEQ_STOP:
            {
                /* Return value is set to "PROCESSING". */
                cc_api_req.req_state = ADJST_USRINFO_PROCESSING;

                /* Apply the control stop processing. */

                s_u1_ctrl_posspd_seq = POSSPD_SEQ_STOP_WAIT;
            }
        break;

        /* State of waiting for stop */
        case POSSPD_SEQ_STOP_WAIT:
            {
                cc_api_req.req_state = ADJST_USRINFO_PROCESSING;
                /* After the stop of control is confirmed, the value for
                   completion is returned. */
                {
                    /* Request of adjustment process was completed. */
                    cc_api_req.req_state = ADJST_USRINFO_COMPLETE;
                    s_u1_ctrl_posspd_seq = POSSPD_SEQ_NONE;
                }
            }
        break;
    }
}
```

7.10 Communications with RDC

7.10.1 Example of Using API Functions

Figure 7.16 shows a block diagram of implementation by using API functions for communications with the RDC.

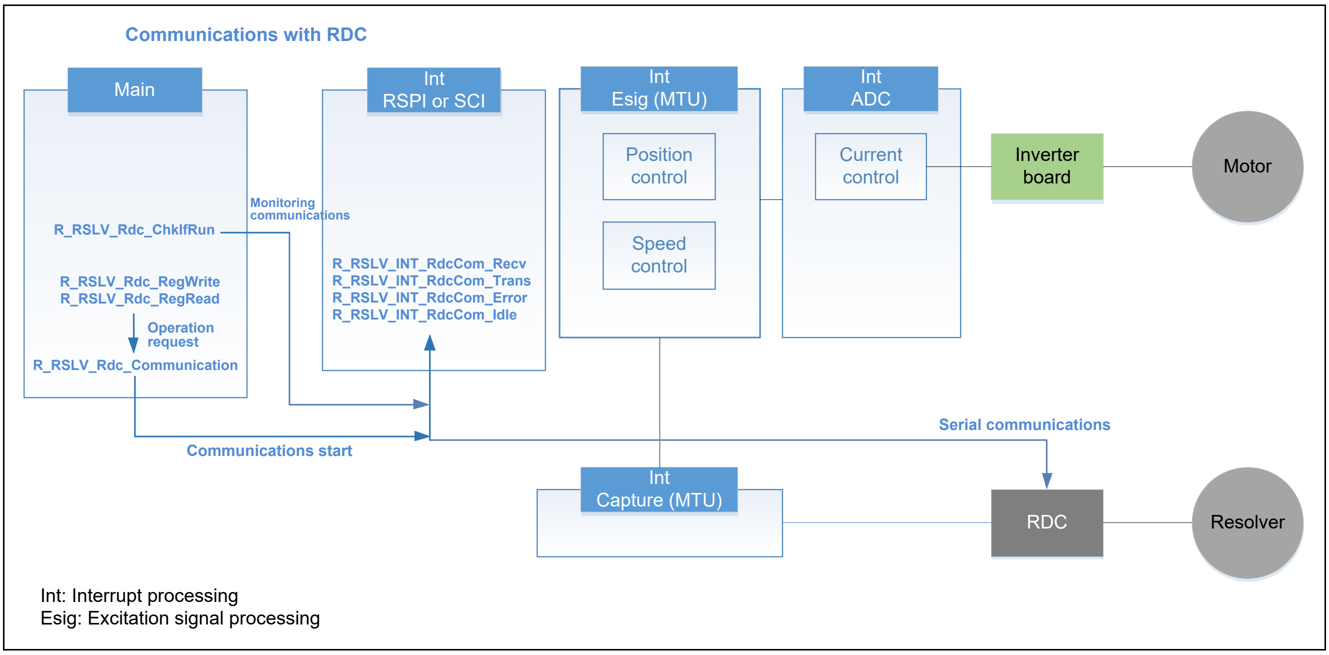


Figure 7.16 Example of Implementing Communications with RDC

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 6.2.31, 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 issue read and write requests, use R_RSLV_Rdc_RegWrite (section 6.2.32, API Function for Writing to an RDC Register) and R_RSLV_Rdc_RegRead (section 6.2.33, API Function for Reading from an RDC Register). The current communication state is returned by R_RSLV_Rdc_ChkIfRun (section 6.2.34, API Function for Acquiring the RDC Register Access State). Do not issue a read or write request during execution.

R_RSLV_INT_RdcCom_XXX is the interrupt processing of the communications by the RSPI or SCI. Use R_RSLV_INT_RdcCom_XXX in each interrupt processing.

7.10.2 Sample Code

The following shows sample code. The RSPI is used in this example.

```
/*************************************************************************
* Function Name: r_rspi0_transmit_interrupt
* Description  : Handler function at RSPI transmit interrupt
* Arguments    : None
* Return Value : None
*************************************************************************/
#pragma interrupt r_rspi0_transmit_interrupt(vect=VECT(RSPI0,SPTI0))
static void r_rspi0_transmit_interrupt( void )
{
   setpsw_i();     // After this, enable other interrupts.

   R_RSLV_INT_RdcCom_Trans();
} /* End of function r_rspi0_transmit_interrupt() */

/*************************************************************************
* Function Name: r_rspi0_receive_interrupt
* Description  : Handler function at RSPI receive interrupt
* Arguments    : None
* Return Value : None
*************************************************************************/
#pragma interrupt r_rspi0_receive_interrupt(vect=VECT(RSPI0,SPRI0))
static void r_rspi0_receive_interrupt( void )
{
   setpsw_i();     // After this, enable other interrupts.

   R_RSLV_INT_RdcCom_Recv();
} /* End of function r_rspi0_receive_interrupt() */

/*************************************************************************
* Function Name: r_rspi0_error_interrupt
* Description  : Handler function at RSPI error interrupt
* Arguments    : None
* Return Value : None
*************************************************************************/
#pragma interrupt r_rspi0_error_interrupt(vect=VECT(RSPI0,SPEI0))
static void r_rspi0_error_interrupt( void )
{
   setpsw_i();     // After this, enable other interrupts.

   R_RSLV_INT_RdcCom_Error();
} /* End of function r_rspi0_error_interrupt() */

/*************************************************************************
* Function Name: r_rspi0_idle_interrupt
* Description  : Handler function at RSPI idle interrupt
* Arguments    : None
* Return Value : None
*************************************************************************/
#pragma interrupt r_rspi0_idle_interrupt(vect=VECT(RSPI0,SPII0))
static void r_rspi0_idle_interrupt( void )
{
   setpsw_i();     // After this, enable other interrupts.

   R_RSLV_INT_RdcCom_Idle();
} /* End of function  r_rspi0_idle_interrupt()*/

Call R_RSLV_Rdc_Communication periodically in the main loop. (See section 7.3.)
```

7.11 Detection of Disconnection from the Resolver Sensor

7.11.1 Example of Using API Functions

Figure 7.17 shows a block diagram of implementation by using API functions for detection of disconnection from the resolver sensor.

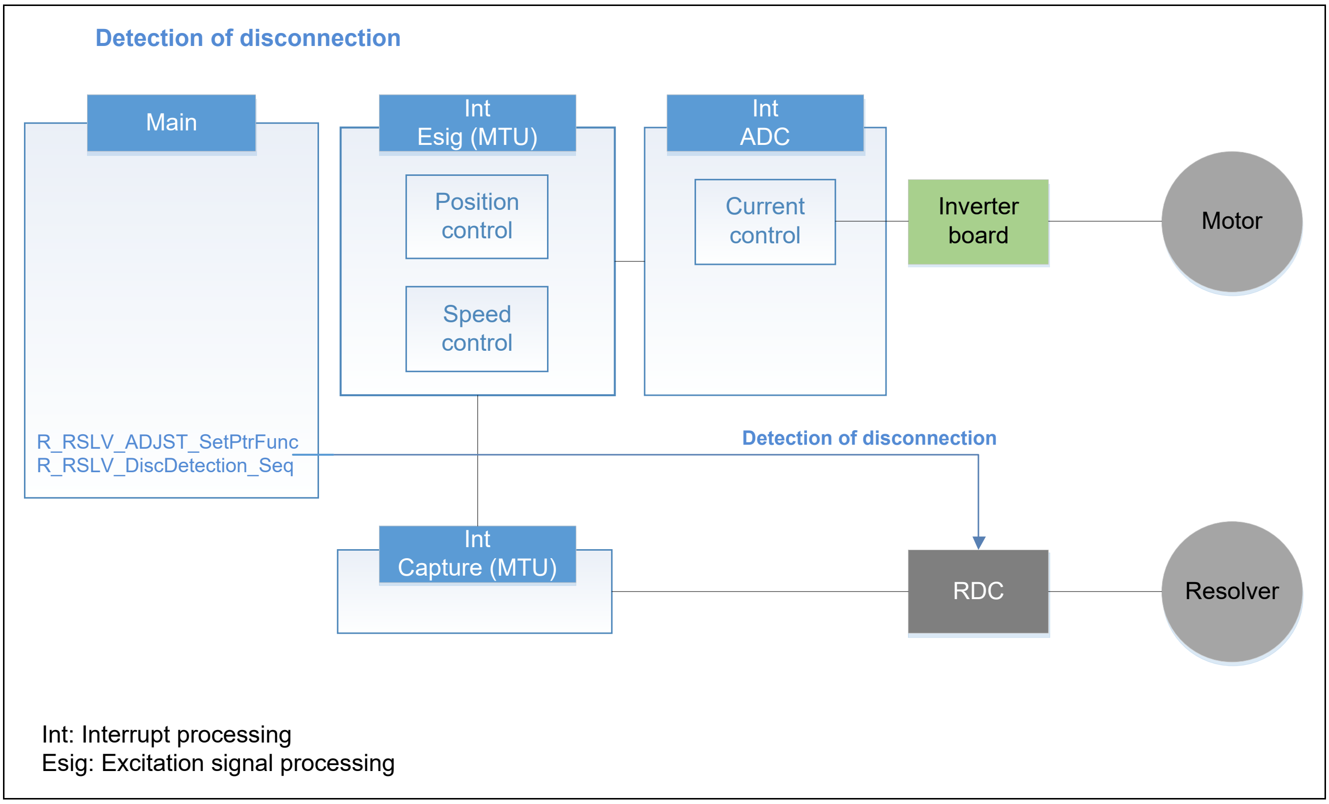


Figure 7.17 Example of Implementing Detection of Disconnection from the Resolver Sensor

To detect disconnection, use R_RSLV_ADJST_SetPtrFunc (section 6.2.46, API Function for Setting the Pointer to the User-Created Callback Function) and R_RSLV_DiscDetection_Seq (section 6.2.48, API Function for Detecting Disconnection).

For how to use the API function for setting callback, see section 7.8, Automatic Adjustment of the Gain and Phase.

Repeated calls of the API function R_RSLV_DiscDetection_Seq for detecting disconnection are required to progress the sequence for detection of disconnection, so periodically call the function.

Figure 7.18 shows an example of implementing the processing for detecting disconnection.

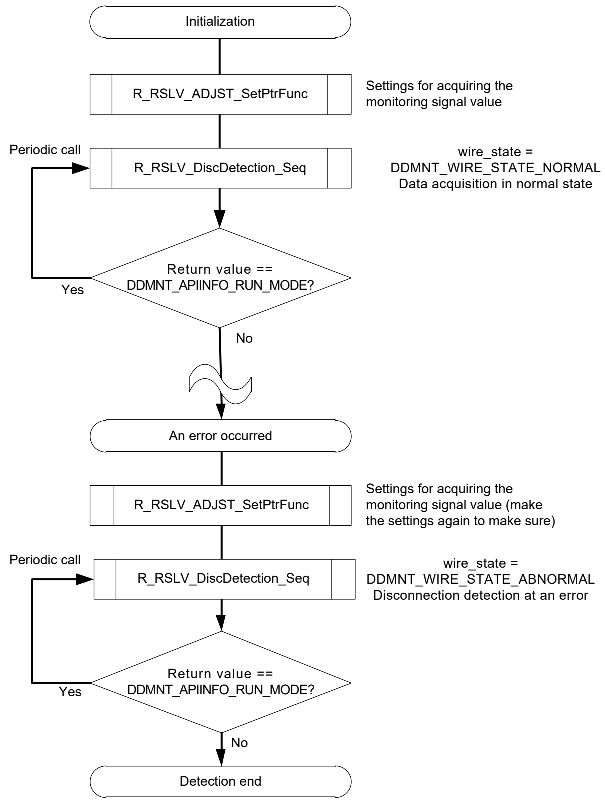


Figure 7.18 Example of Disconnection Detection Sequence

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 initialization settings using 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.

```
/* State of Sequence for Detection of Resolver Disconnection */
enum
{
    STS_DDCNCT_NONE = 0,
    STS_DDCNCT_INIT_START,
    STS_DDCNCT_INIT,
    STS_DDCNCT_INIT_FIN,
    STS_DDCNCT_CONF_START,
    STS_DDCNCT_CONF,
    STS_DDCNCT_CONF_FIN,
};

unsigned char  s_u1_sts_ddcnct = STS_DDCNCT_NONE;   /* State management variable */

/* State setting macro */
#define SetDdiscnctStatus(status)          (s_u1_sts_ddcnct = status)

/* 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);
            }
        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;
    }
}
```

Call r_mtr_DetectDisconnect_Seq periodically in the main loop. (See section 7.3.)

7.12 Cancelling an Alarm

7.12.1 Example of Using API Functions

Figure 7.19 shows a block diagram of implementation by using API functions for cancelling an alarm.

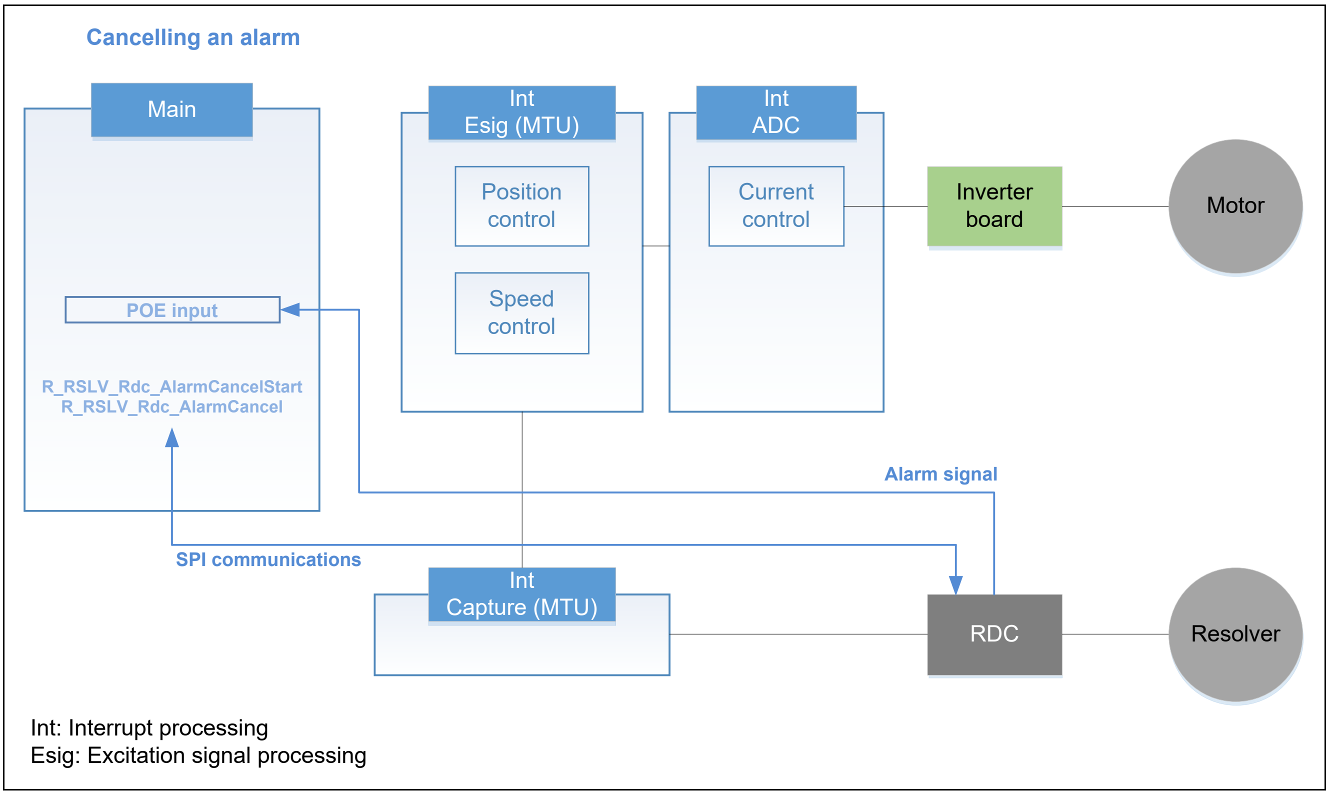


Figure 7.19 Example of Implementing Processing to Cancel an Alarm

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 6.2.42, 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 6.2.43, 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.

(a) R_RSLV_Rdc_AlarmCancel

This API function can be called at any time after an alarm is generated. In this 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();
}
```

(b) R_RSLV_Rdc_AlarmCancel

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

```
main( void )
{
    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. Notes

Note the following when making initial settings.

8.1 Initial Setting Procedure

Follow the steps below to make initial settings.

1. Specify system information (R_RSLV_SetSystemInfo()).
1. Make settings for each peripheral module (R_RSLV_CreatePeripheral()).
2. Acquire RDC driver setting information (R_RSLV_GetRdcDrvSettingInfo()).
3. 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 or peripheral module settings.

8.2 Specifying System Information

When specifying system information such as the MCU type, do not specify values that are not listed in Table 6.5, Structure Definitions for R_RSLV_SetSystemInfo. The values not listed in the table are reserved for possible future extensions. If a reserved value is specified, subsequent settings for peripheral modules will be faulty.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

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

8.7 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. 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 9.1.

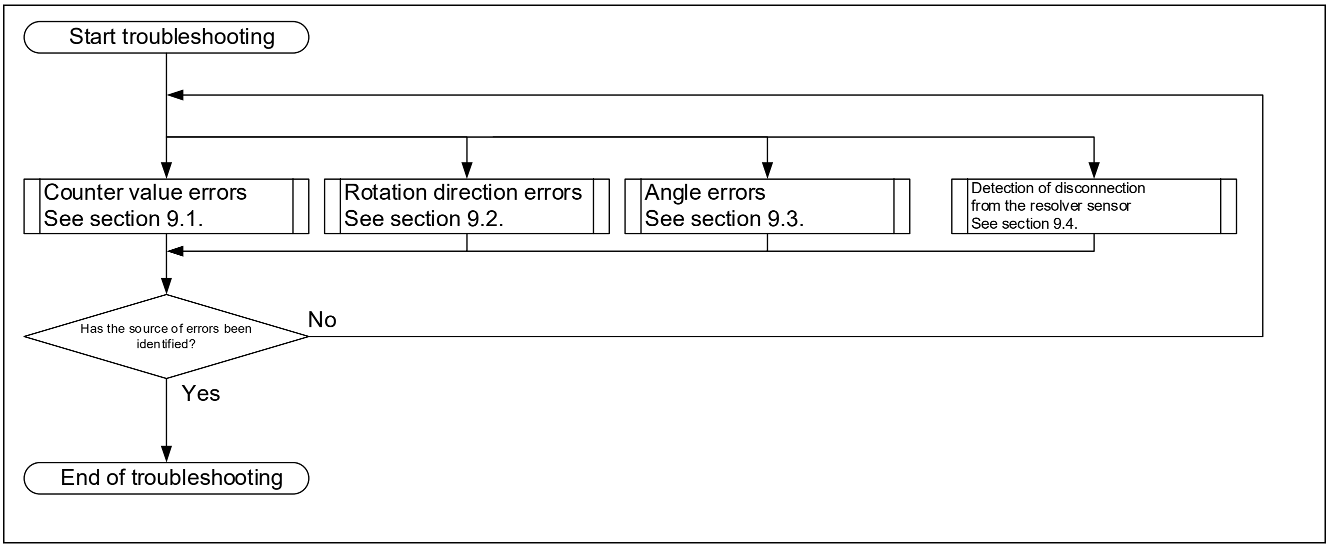


Figure 9.1 Overall Flow of Troubleshooting

9.1 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 9.2. For details of detection of disconnections from the resolver sensor, see section 9.4, Detection of Disconnection from the Resolver Sensor.

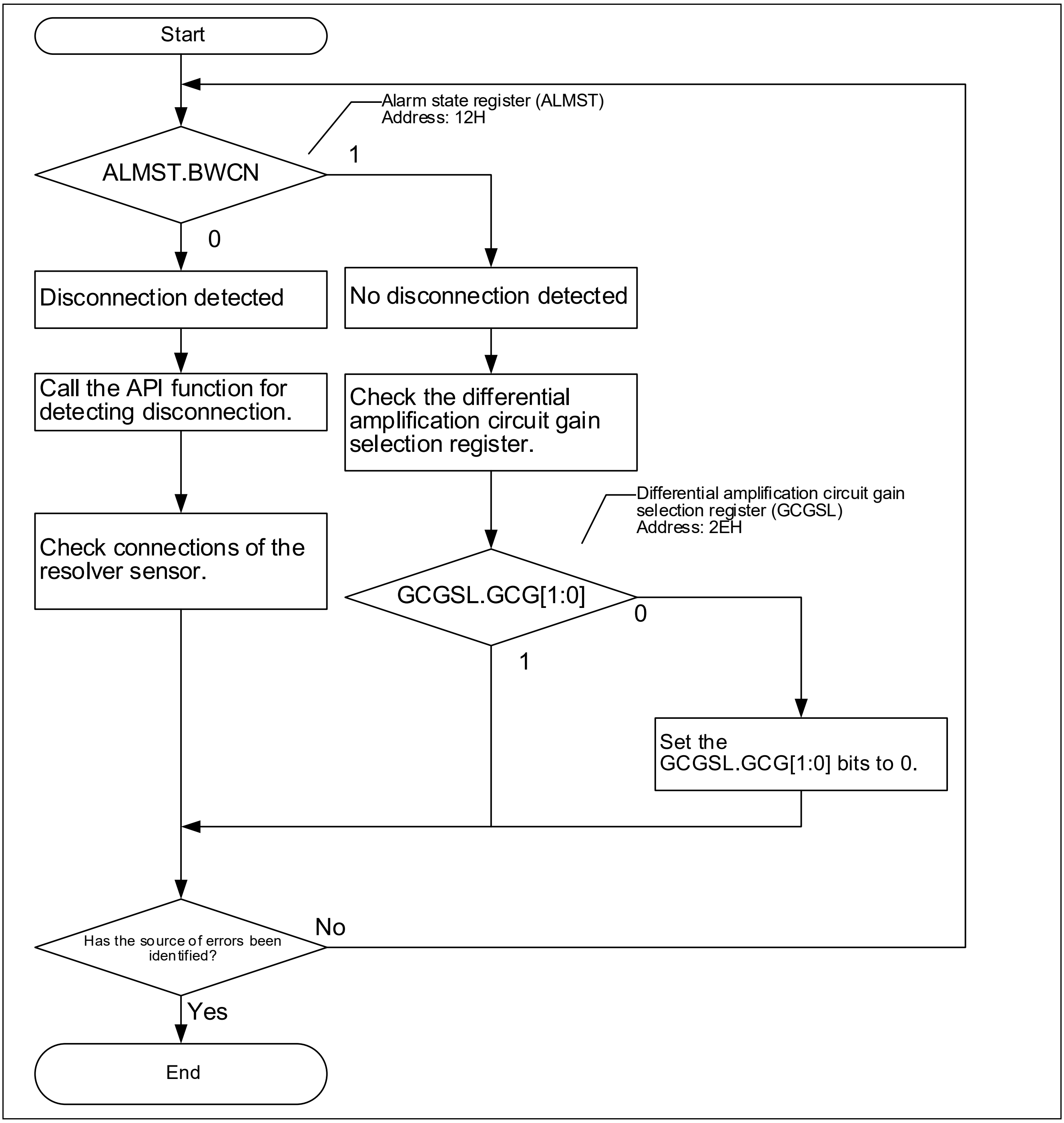


Figure 9.2 Counter Value Errors

9.2 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 9.3.

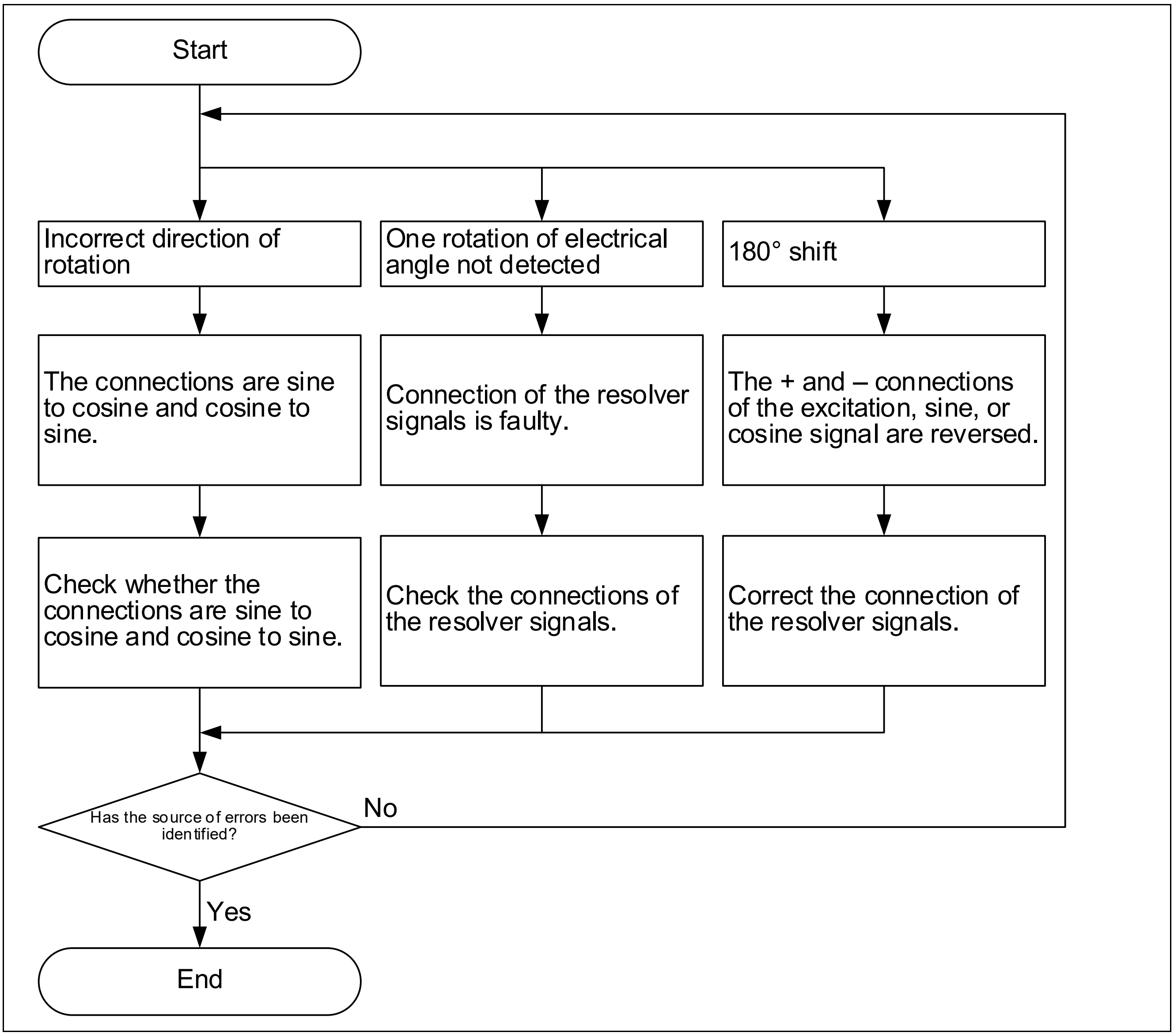


Figure 9.3 Rotation Direction Errors

9.3 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).

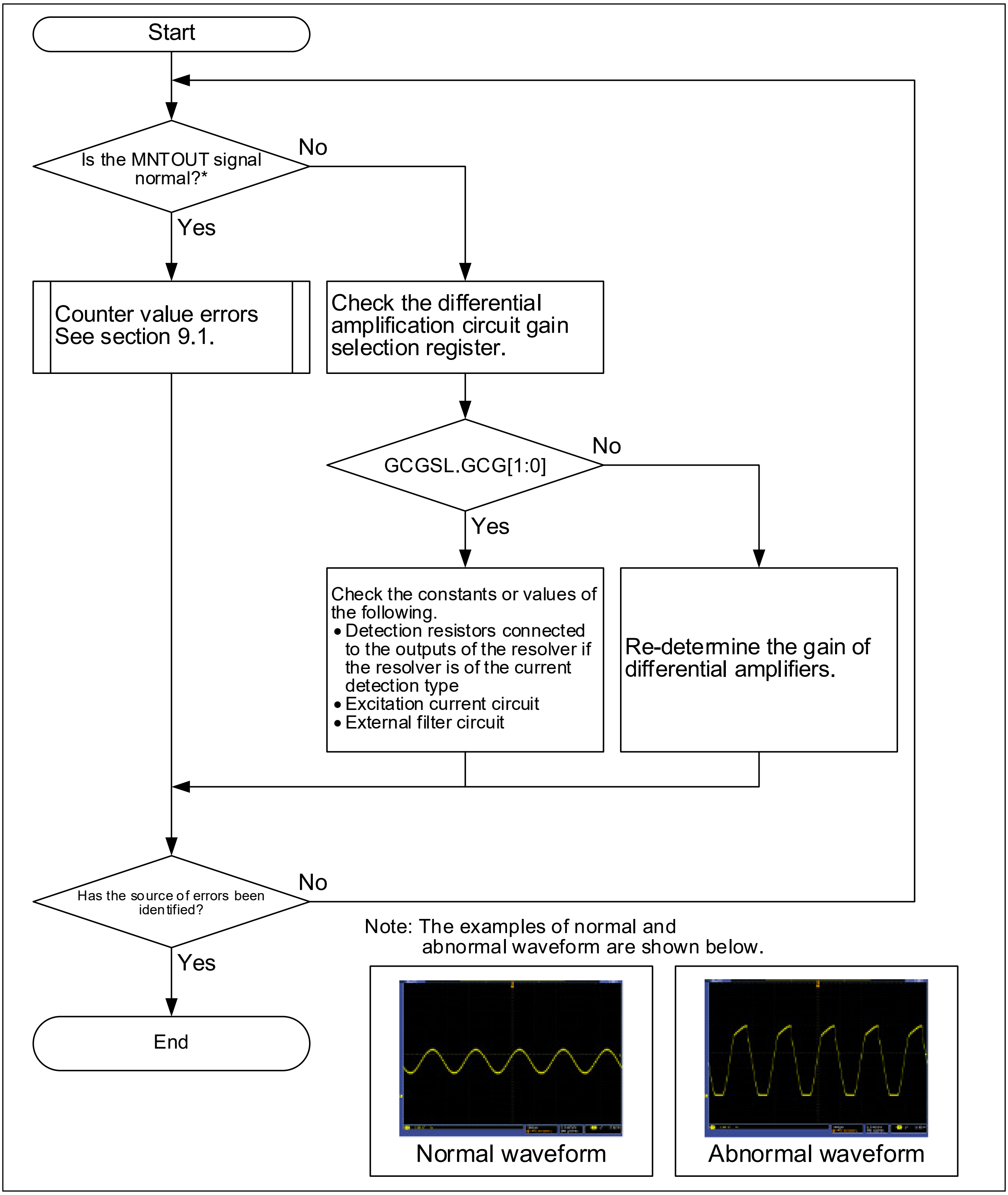


Figure 9.4 Angle Errors

9.4 Detection of Disconnection

Note that 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 the 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 9.5 shows normal operation and Figures 9.6 to 9.8 show cases of the detection of disconnection when the resolver is of the transformer type.

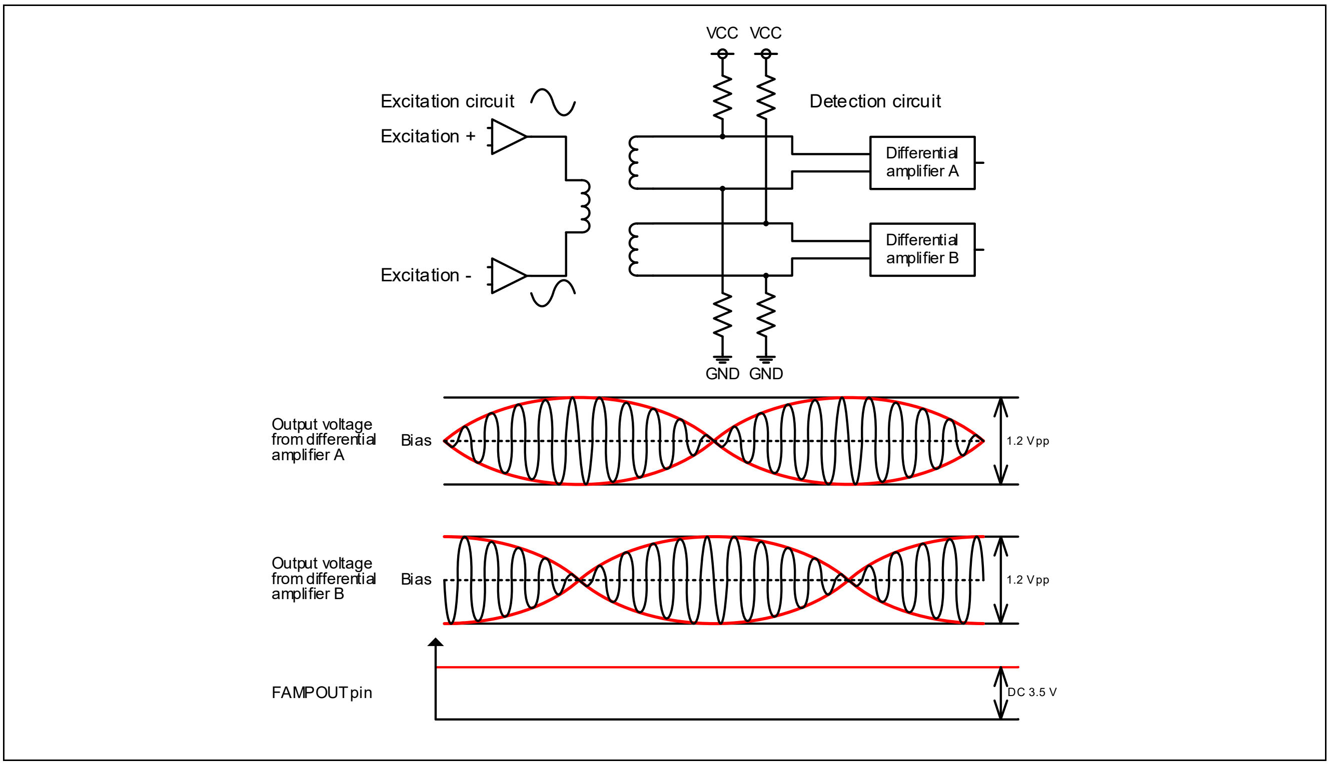


Figure 9.5 Normal State

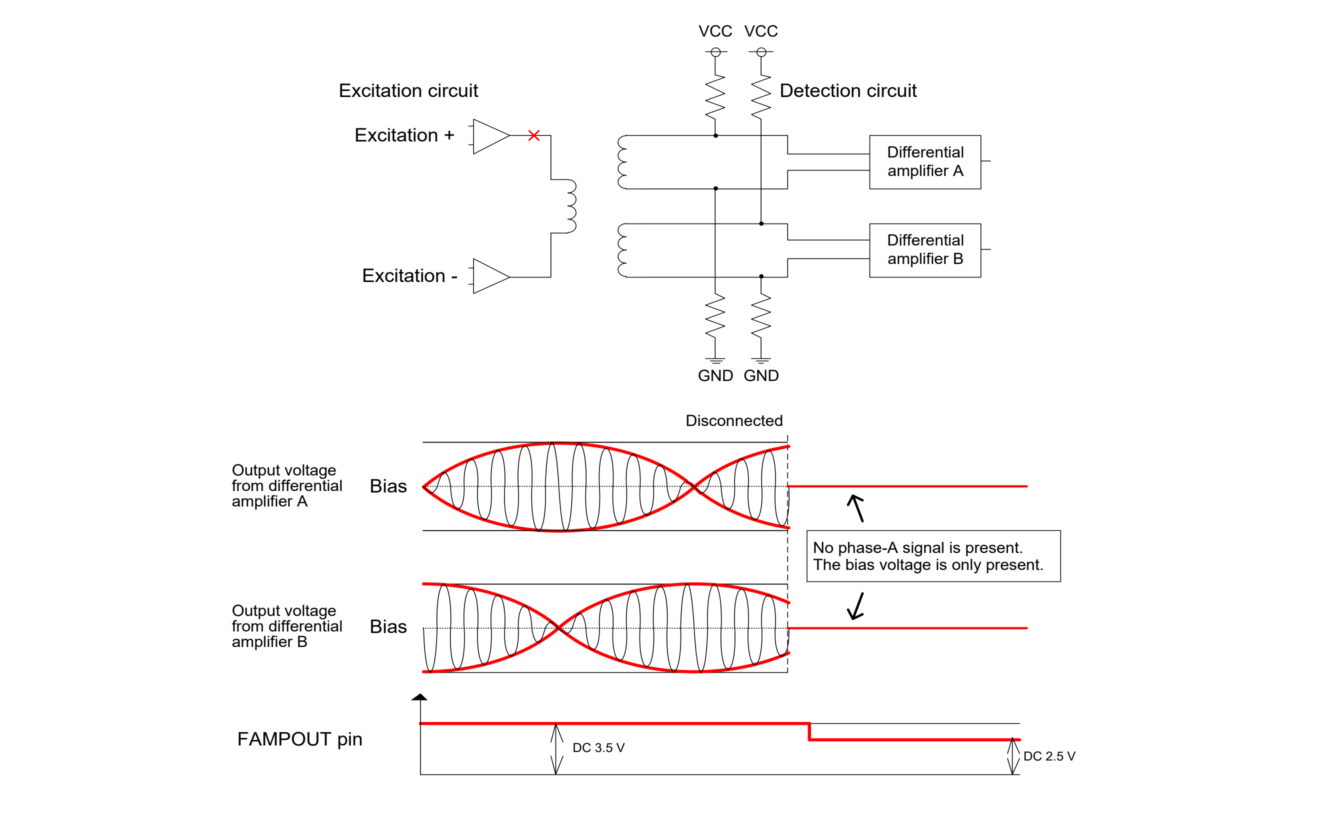


Figure 9.6 Disconnection on the Excitation Side

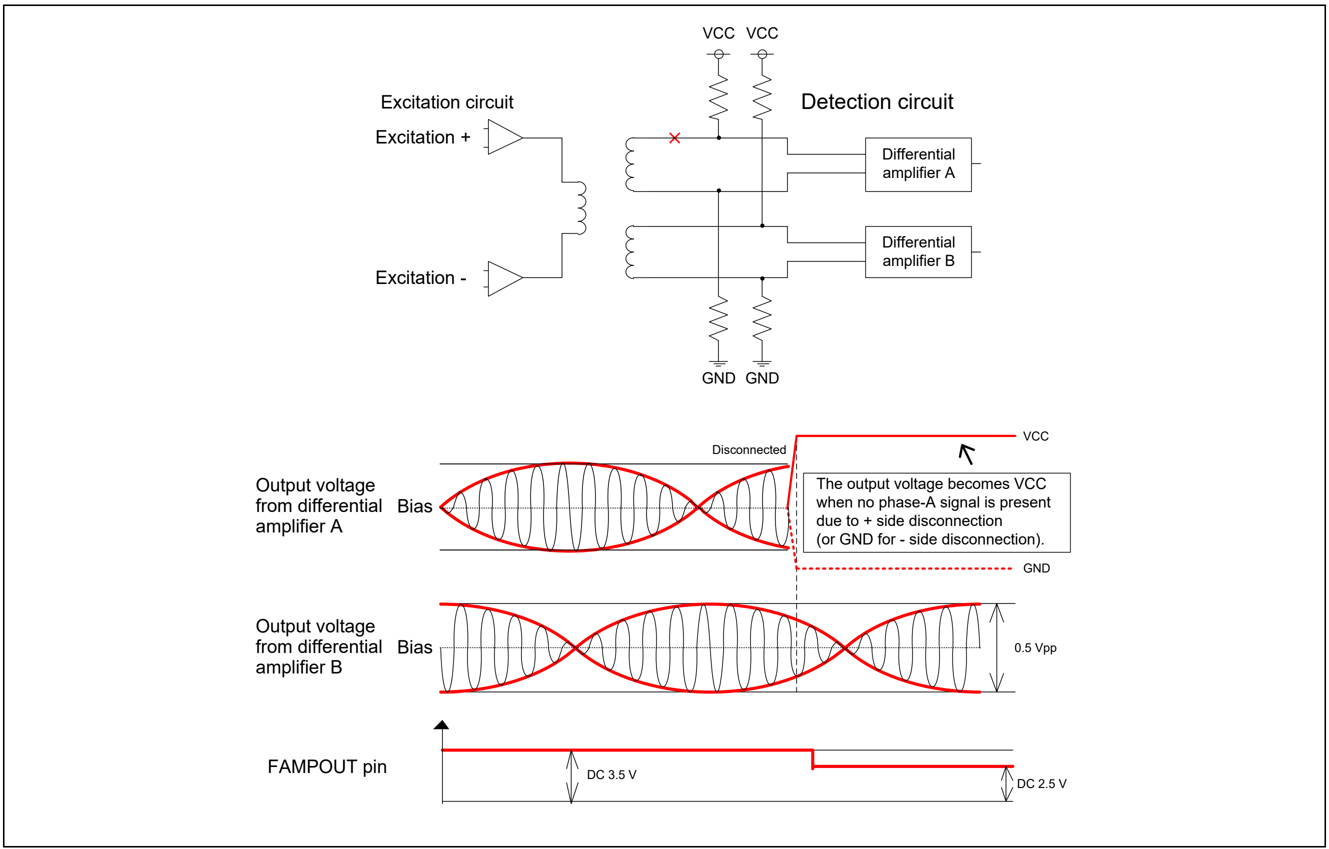


Figure 9.7 Disconnection on the SIN+ Side

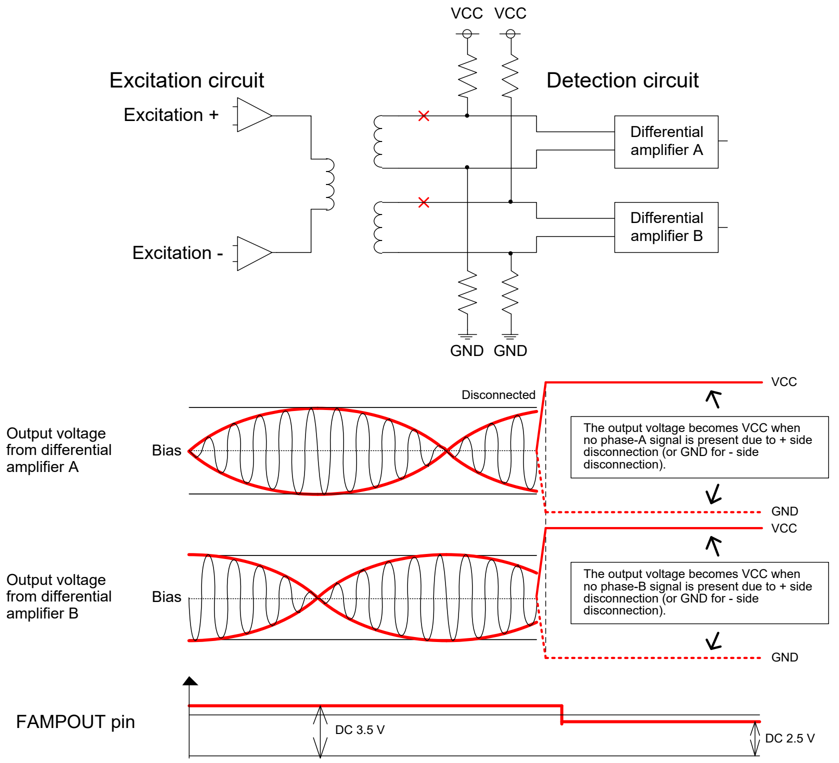


Figure 9.8 Disconnection on the SIN+ and COS+ Sides

Figure 9.9 shows normal operation and Figures 9.10 to 9.13 show cases of the detection of disconnection when the resolver is of the current-detection type.

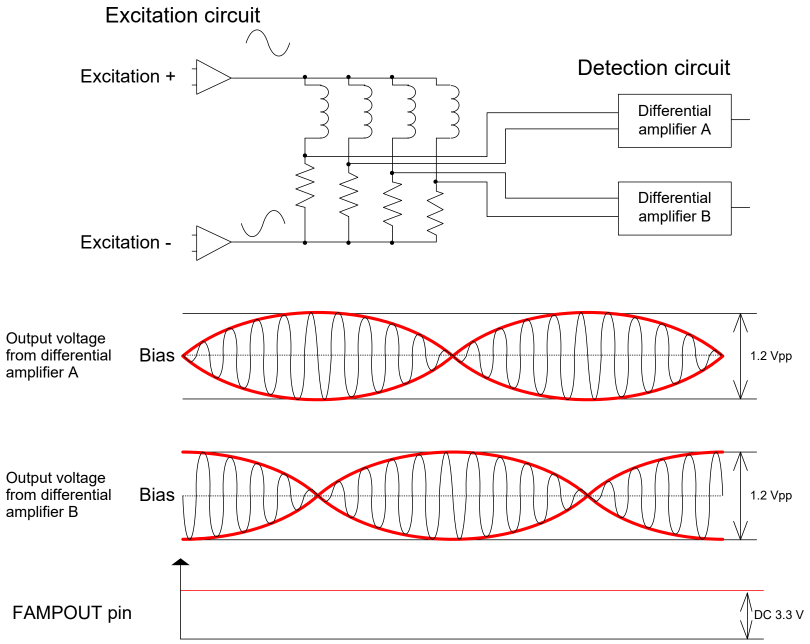


Figure 9.9 Normal State

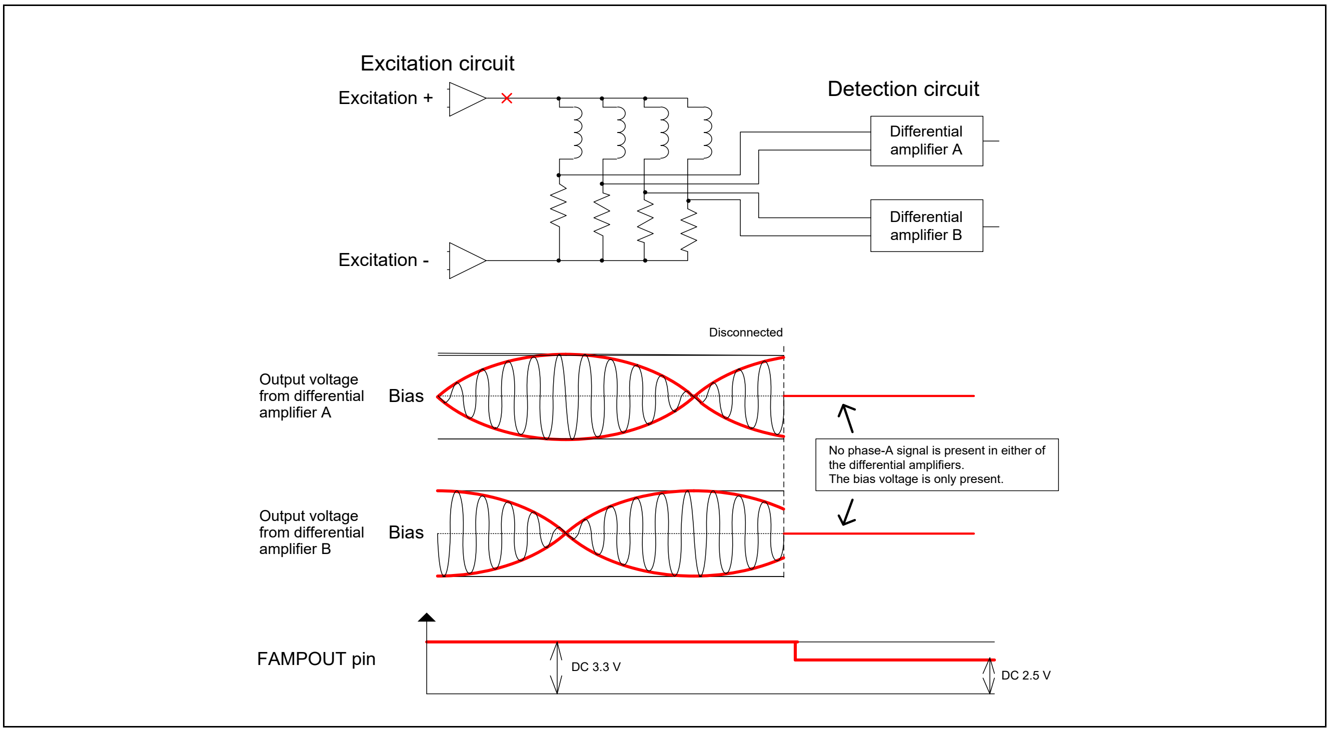


Figure 9.10 Disconnection on the Excitation Side

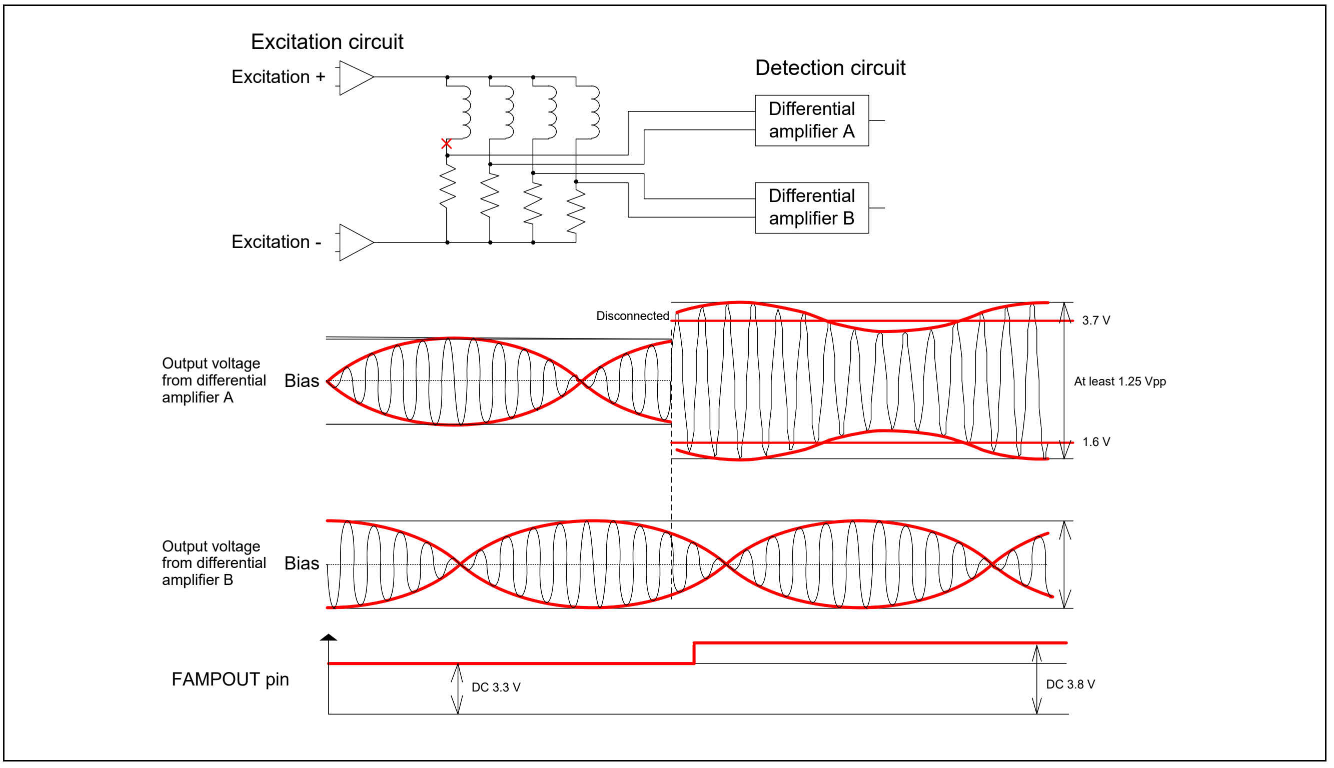


Figure 9.11 Disconnection on the 0°- Side

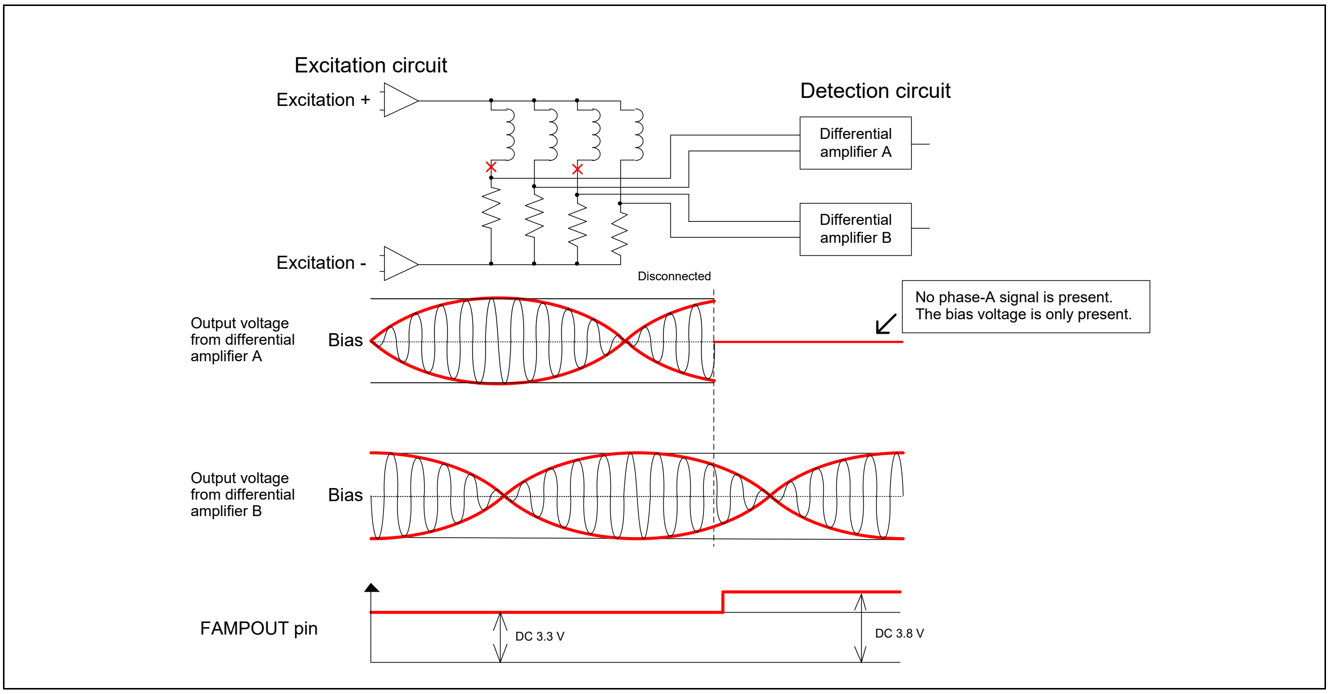


Figure 9.12 Disconnection on the 0°- and 180°- Sides

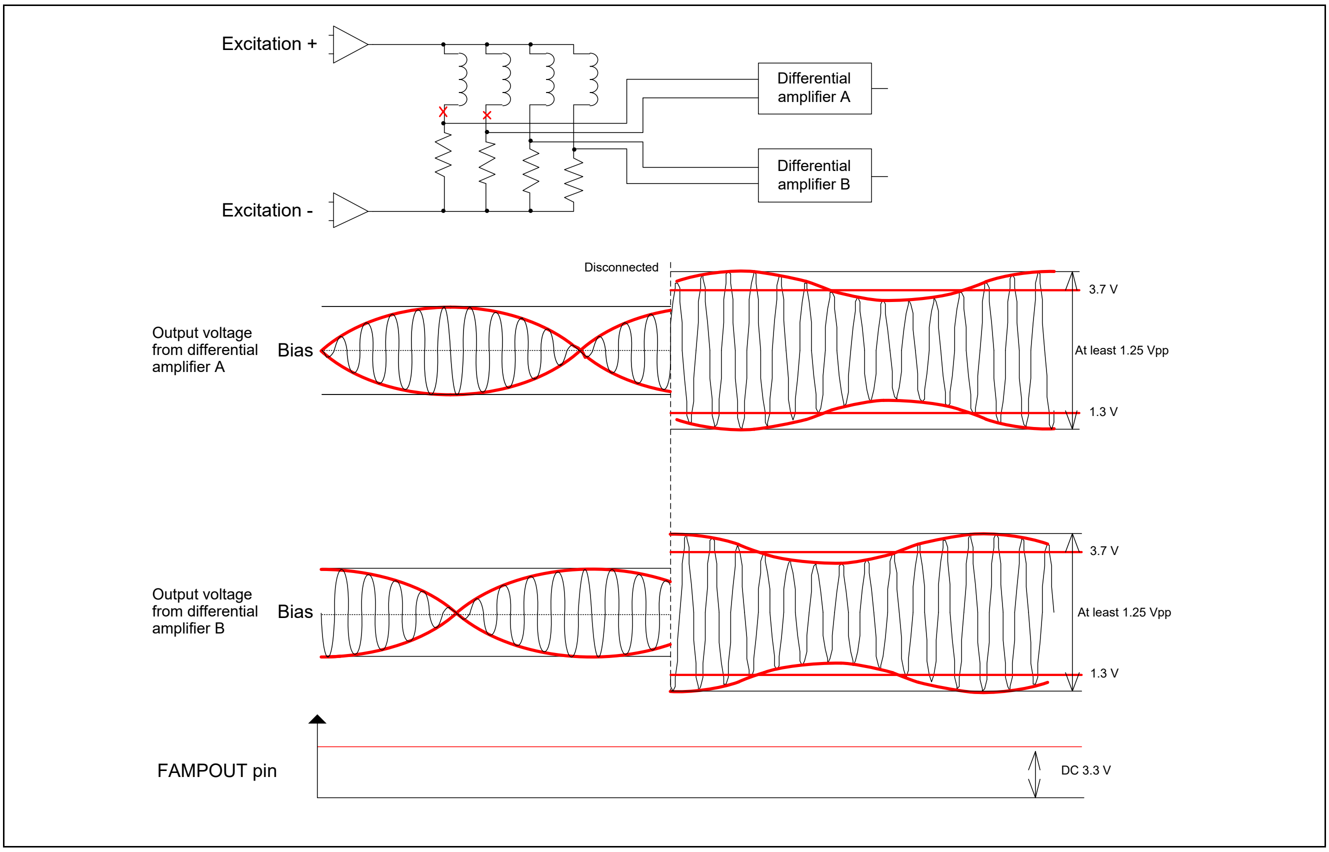


Figure 9.13 Disconnection on the 0°- and 90°- Sides

Website and Support

Renesas Electronics Website

http://www.renesas.com/

Inquiries

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

Revision History

Rev.	Date	Description	
		Page	**Summary**
1.00	Dec 25, 2019	—	First edition issued
1.10	Mar 23, 2020	6	Table 1.2 Program Size: ROM Size/RAM Size corrected
		10	Figure 3.2 Processing for Specifying System Information: Corrected
		29	Table 4.1 Folder and File Configuration: Corrected
		53	Section 6.2.29, API Function for Initializing Variables for RDC Communications: The RDC registers to be initialized were added in Remark.
		60	Table 6.5 Structure Definitions for R_RSLV_SetSystemInfo: Corrected
		64	Table 6.9 Structure Definitions for R_RSLV_ADJST_SetPtrFunc: Member “req_speed” was added.
		67	7.2.2 Details of Initialization Processing: Addition to the text
		68	7.2.3 Sample Code: Sample code corrected
		85	7.8.3 Sample Code (1) Callback function settings: Sample code corrected
		106 to 114	Section 9, Troubleshooting: Added
1.20	Apr 01, 2021	1, 6, 36, 42 to 45, 56, 63, 89, 91, 105, 112	The errors were corrected.

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.

Notice

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

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

Trademarks

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/.