Application Note

Document No.: AN1127

G32R501 SDK Examples User Manual

Version: V1.0

1 Introduction

This user manual provides guidance on how to use the routines in the G32R5xx SDK of G32R5 series.

Note: Some SDK routines not covered in this document are included in the G32R5xx SDK extension package. If you need the content of the extension package, please visit the official website of Geehy to obtain it.

Contents

1 Introduction 1

2 Peripheral Library Routines 4

2.1 ADC Routine 4

2.2 Boot Routine Description 11

2.3 CAN Routine Description 14

2.4 CAP Routine 18

2.5 COMP Routine 20

2.6 DAC Routine 21

2.7 DCCOMP Routine 22

2.8 DMA Routine 24

2.9 Flash Routine Description 25

2.10 GPIO Routine Description 25

2.11 HRCAP Routine Description 27

2.12 HRPWM Routine Description 27

2.13 I2C Routine Description 29

2.14 Interrupt Routine Description 31

2.15 IPC Routine Description 35

2.16 LED Routine Description 39

2.17 LIN Routine Description 40

2.18 Low-power Mode Routine Description 41

2.19 PWM Routine 43

2.20 QEP Routine 51

2.21 SDF Routine Description 55

2.22 SPI Routine Description 58

2.23 SYSCTL Routine Description 63

2.24 Timer Routine Description 64

2.25 UART Routine Description 64
2.26 Watchdog Routine Description 68
2.27 Zidian Routine Description 68
3 Device Support Routine 69
3.1 ADC Routine Description 69
3.2 CAP Routine Description 69
3.3 DAC Routine Description 69
3.4 DMA Routine Description 70
3.5 Template Routine Description 70
3.6 GPIO Routine Description 70
3.7 HRPWM Routine Description 71
3.8 I2C Routine Description 73
3.9 LED Routine Description 74
3.10 SPI Routine Description 74
3.11 Timer Routine Description 75
3.12 UART Routine Description 76
4 Revision 77

2 Peripheral Library Routines

Note: These routines are located at the following positions of G32R5xx_SDK:

G32R5xx_SDK_VERSION：/driverlib/DEVICE_GPN/examples/CORE_IF_MULTICORE/

2.1 ADC Routine

2.1.1 ADC software trigger

File: adc_ex1_soc_software.c

This example is based on software triggering of conversion of some voltages on ADCA and ADCC. Since ADCC does not perform conversion before ADCA is completed, ADC will not run asynchronously. However, this is far less efficient than allowing parallel synchronous conversion of ADC (e.g. by using PWM trigger).

External connection:

- A0, A1, C2, and C3 shall be connected to the signal to be converted

Monitoring variables:

- myADC0Result0 - Digital representation of the voltage on Pin A0
- myADC0Result1 - Digital representation of the voltage on Pin A1
- myADC1Result0 - Digital representation of the voltage on Pin C2
- myADC1Result1 - Digital representation of the voltage on Pin C3

2.1.2 ADC PWM trigger

File: adc_ex2_soc_pwm.c

This example sets PWM1 to periodically trigger the conversion of ADCA.

External connection:

- A0 shall be connected to the signal to be converted.

Monitoring variables:

- myADC0Resulte - A series of analog-to-digital conversion samples from Pin A0. The time interval between samples is determined by the period of the PWM timer.

2.1.3 ADC software triggers synchronous conversion

File: adc_ex4_soc_software_sync.c

This example uses Input 5 of Input XBAR as the software forced trigger to convert some

voltages on ADCA and ADCC. Input 5 is triggered through GPIO0 flip, but any idle GPIO can be used. This method will ensure that both ADC start converting simultaneously.

External connection:

- A2, A3, C2 and C3 pins shall be connected to the signal to be converted

Monitoring variables:

- myADC0Result0 - Digital representation of the voltage on Pin A2
- myADC0Result1 - Digital representation of the voltage on Pin A3
- myADC1Result0 - Digital representation of the voltage on Pin C2
- myADC1Result1 - Digital representation of the voltage on Pin C3

2.1.4 ADC continuous conversion

File: adc_ex5_soc_continuous.c

This example sets ADC continuous conversion to reach maximum sampling rate.

External connection:

- A0 pin shall be connected to the signal to be converted.

Monitoring variables:

- adcAResults - A series of analog-to-digital conversion samples from Pin A0. The time interval between samples is based on the minimum possible value that the ADC speed can reach.

2.1.5 ADC continuous conversion reads data through DMA

File: adc_ex6_soc_continuous_dma.c

This example sets two ADC channels for simultaneous conversion. The results will be transferred to a buffer in RAM through DMA.

External connection:

- A3 and C3 pins shall be connected to the signal to be converted.

Monitoring variables:

- myADC0DataBuffer: Digital representation of the voltage on Pin A3
- myADC1DataBuffer: Digital representation of the voltage on Pin C3

2.1.6 ADC PPB offset

File: adc_ex7_ppb_offset.c

This example software triggers ADC. When ADC starts conversion (SOC), offset adjustment is automatically applied through the post-processing block. After the program runs, the corresponding memory will contain the results of ADC and post-processing block (PPB).

External connection:

- A2 and C2 pins shall be connected to the signal to be converted.

Monitoring variables:

- myADC0Result: Digital representation of the voltage on Pin A2
- myADC0PPBResult: Digital representation of the voltage on Pin A2, subtracting 100 automatically added LSB offsets
- myADC1Result: Digital representation of the voltage on Pin C2
- myADC1PPBResult: Digital representation of the voltage on Pin C2, adding 100 automatically added LSB offsets

2.1.7 ADC PPB limits

File: adc_ex8_ppb_limits.c

This example sets PWM to periodically trigger ADC. If the result exceeds the defined range, the post-processing block will generate an interrupt.

The default limits are 1000 LSB and 3000 LSB. When VREFHI is set to 3.3V, if the input voltage exceeds about 2.4V or falls below about 0.8V, PPB will generate an interrupt.

External connection:

- A0 pin shall be connected to the signal to be converted.

Monitoring variables:

- None

2.1.8 ADC PPB delayed capture

File: adc_ex9_ppb_delay.c

This example demonstrates delayed capture by using the post-processing block.

Two asynchronous ADC triggers are set:

- PWM1; the period is 2048, triggering SOC0 to convert on Pin A0

- PWM2; the period is 9999, triggering SOC1 to convert on Pin A2

Generate an ISR at the end of each conversion. In ISR of SOC0, the conversion counter is increased and the PPB is checked to determine whether the sample is delayed.

After the program runs, the memory will contain:

- conversion: Conversion delay sequence of SOC0.

Monitoring variables:

- delay: Delay time corresponding to each delayed conversion.

2.1.9 ADC PWM triggers multiple SOC

File: adc_ex10_multiple_soc_pwm.c

This example sets PWM1 to periodically trigger a set of conversions on ADCA and ADCC. This example demonstrates collaborative work of multiple ADC, and the parallelism among multiple ADC is used to process a batch of conversions.

External connection:

- A0, A1, A2 and C2, C3, C4 pins shall be connected to the signal to be converted.

Monitoring variables:

- adcAResult0 - Digital representation of the voltage on Pin A0
- adcAResult1 - Digital representation of the voltage on Pin A1
- adcAResult2 - Digital representation of the voltage on Pin A2
- adcCResult0 - Digital representation of the voltage on Pin C2
- adcCResult1 - Digital representation of the voltage on Pin C3
- adcCResult2 - Digital representation of the voltage on Pin C4

2.1.10 ADC burst mode

File: adc_ex11_burst_mode_pwm.c

This example sets PWM1 to periodically trigger ADCA using burst mode. This allows different channels to be sampled during each burst.

Each burst triggers 3 conversions. A0 and A1 are part of each burst, while the third conversion alternates among A2, A3, and A4. This allows high-speed sampling of the high-priority signals, while the low-priority signals can be sampled at a low rate.

Use ADCA interrupt ISR to read the results of ADCA.

External connection:

- A0, A1, A2, A3, and A4 pins shall be connected to the signal to be converted.

Monitoring variables:

- adcAResult0 - Digital representation of the voltage on Pin A0
- adcAResult1 - Digital representation of the voltage on Pin A1
- adcAResult2 - Digital representation of the voltage on Pin A2
- adcAResult3 - Digital representation of the voltage on Pin A3
- adcAResult4 - Digital representation of the voltage on Pin A4

2.1.11 ADC burst mode oversampling

File: adc_ex12_burst_mode_oversampling.c

This example is an ADC oversampling example implemented using software. ADC SOC is configured in burst mode, and triggered by the SOCA event of PWM.

External connection:

- A2 pin shall be connected to the signal to be converted.

Monitoring variables:

- lv_results - Array of numerical values measured on Pin A2 (the oversampling amount is configured by Oversampling_Amount)

2.1.12 ADC SOC oversampling

File: adc_ex13_soc_oversampling.c

This example sets PWM1 to periodically trigger a set of conversions on ADCA, including multiple SOC, all of which convert A2 to achieve oversampling of A2.

Use ADCA interrupt ISR to read the results of ADCA.

External connection:

- A0, A1, and A2 pins shall be connected to the signal to be converted.

Monitoring variables:

- adcAResult0 - Digital representation of the voltage on Pin A0
- adcAResult1 - Digital representation of the voltage on Pin A1
- adcAResult2 - Digital representation of the voltage on Pin A2

2.1.13 ADC PPB PWM trigger

File: adc_ex14_ppb_pwm_trip.c

This example demonstrates how to trigger PWM through ADC limit detection PPB block. ADCAINT1 is configured to periodically trigger ADCA channel 2 after initial software forced trigger. The amplitude limit detection post-processing block (PPB) is configured. If the ADC result exceeds the defined range, the PPB will generate an ADCxEVTy event. This event is configured as the trigger source of PWM by configuring the PWM XBAR and corresponding PWM trip zone and digital comparison submodule. This example demonstrates:

- One-time trigger
- Periodic trigger
- Directly trigger PWM through the digital comparison submodule

The default limits are 0 LSB and 3600 LSB. When VREFHI is set to 3.3V, if the input voltage exceeds approximately 2.9V, PPB will generate a trigger event.

External connection:

- A2 shall be connected to a signal to be converted
- Use an oscilloscope to observe the following signals:
  - PWM1 (GPIO0——GPIO1)
  - PWM2 (GPIO2——GPIO3)
  - PWM3 (GPIO4——GPIO5)

Monitoring variables:

- adcA2Results - Digital representation of the voltage on Pin A2

2.1.14 ADC open circuit and short circuit detection

File: adc_ex15_open_shorts_detection.c

This example demonstrates the circuit configuration of ADC open circuit and short circuit detection (ADCOSDETECT), which is used to detect pin faults in the system. This example enables open circuit and short circuit detection circuit and necessary ADC configuration, and diagnoses the ADCAA0 input pin status before starting normal ADC conversion.

To enable the ADC OSDetect circuit:

1. Configure ADC for conversion (e.g. channel, SOC, ACQPS, prescaler, trigger, etc.). The OSDetect function is only available in 12-bit mode.

2. Set the ADCOSDETECT register for the required voltage division connection. For detailed information on available OSDetect configuration, please refer to *G32R5xx User Manual*.

3. Start the conversion and check the conversion results. Note: The results must be interpreted based on the driving condition of the input terminal and the values of Rs and Cp. If the Vs signal can be disconnected from the input pin, the circuit can be used to detect the input pins of open circuit and short circuit. In the example, ADCAA0 channel is configured and the A0 pin status is checked using the following algorithm:

- Step 1: Configure the full-range OSDETECT mode and capture the ADC result (resultHi)
- Step 2: Configure the zero-range OSDETECT mode and capture the ADC result (resultLo)
- Step 3: Disable the OSDETECT mode and capture the ADC result (resultNormal)
- Step 4: Determine the status of the ADC pin.
  - If the pin is open, resultLo will be equal to Vreflo, and resultHi will be equal to Vrefhi.
  - If the pin is shorted to Vrefhi, resultLo shall be approximately equal to Vrefhi, and resultHi shall be equal to Vrefhi.
  - If the pin is shorted to Vreflo, resultLo shall be equal to Vreflo, and resultHi shall be approximately equal to Vreflo.
  - If the pin is connected to a valid signal, resultLo shall be greater than osdLoLimit but less than resultNormal, while resultHi shall be less than osdHiLimit but greater than resultNormal.

Table 1 Pin Status Judgment

Input	Full-range output	Zero-range output	Pin status
Unknown	VREFHI	VREFLO	Open circuit
VREFHI	VREFHI	Approximately equal to VREFHI	Shorted to VREFHI
VREFLO	Approximately equal to VREFLO	VREFLO	Shorted to VREFLO
Vn	Vn < resultHi < VREFHI	VREFLO < resultLo < Vn	Good

- Step 5: If osDetectStatusVal is greater than 4, it indicates that the pin is not faulty.
  - If osDetectStatusVal == 1, it indicates open circuit of Pin A0.
  - If osDetectStatusVal == 2, it indicates short circuit of Pin A0 to VREFLO.
  - If osDetectStatusVal == 4, it indicates short circuit of Pin A0 to VREFHI.
  - If osDetectStatusVal == 8, it indicates Pin A0 is in good/valid status.
  - If the value of osDetectStatusVal is greater than 4, it indicates Pin A0 status is valid.

When configuring ADC to enter OSDETECT mode, the followings shall be noted:

1. The tolerance of the voltage divider resistor may vary greatly, so this function shall not be used to check the conversion accuracy.
2. For the implementation and availability of analog input channels, please refer to *Chip Datasheet*.
3. Due to high drive impedance, an S+H duration longer than the minimum sampling and holding duration of the ADC is required.

External connection:

- Pin A0 shall be connected to a signal to be converted

Monitoring variables:

- osDetectStatusVal: Open/Short circuit detection status of the voltage on Pin A0
- adcAResult0: Digital representation of the voltage on Pin A0

2.2 Boot Routine Description

2.2.1 Boot error status Pin routine configured using DCS OTP

File: boot_ex1_errorstatuspin.c

This example demonstrates how to configure the boot mode, boot mode selection pin, and error status pin.

Note: DCS OTP (one-time programmable) memory is used to configure the boot mode, boot mode selection pin, and error status pin. Once the DCS OTP parts are programmed, they cannot be erased or reprogrammed.

This example aims to demonstrate how to configure the boot control and the functions of error status pins. Once the error status pin is enabled, the software will trigger NMI and the error status pin will change to a high level. Then, the NMI ISR will clear the error status and drive the error state pin to a low level, and return to a major cycle. In the major cycle, the NMI will be triggered again. For more details, please refer to ROM Code and Peripheral Startup in the *User Manual*.

External connection:

- Monitor the error status pins used according to the content programmed to GPREG2. The error status pin may be GPIO 24, GPIO 28, or GPIO 29.

Monitoring variables:

- None

2.2.2 Custom boot configuration routine

File: boot_ex2_customBootConfig.c

This example implements the custom boot configuration according to the selected project construction configuration: 0 boot mode selection pin (ZERO_BMSPS), 1 boot mode selection pin (ONE_BMSP), or 3 boot mode selection pins (THREE_BMSPS).

Select the construction configuration and run the program. To test a specific boot mode: pause the program, apply an appropriate voltage to BMSP if necessary, and then reset. If multiple specific boot modes need to be tested, reset the debugger and run the program before executing the above steps; this ensures that the device does not execute the “non-debuggable” codes from previous boot modes, and the emulation equivalent Boot ROM register contains appropriate values. Do not perform reset by powering off; if the device is powered off, the value in the emulation equivalent Boot ROM register may change.

By default, this example will configure the boot mode selection pin and boot mode options to perform the emulation boot process.

The following constants can be changed in the header file:

- STANDALONE_BOOT - Determine whether to simulate the independent boot or the emulation boot process when the device is reset (when the emulator is connected). Warning: The independent boot process requires the user-configurable DCS OTP (one-time programmable) register to be programmed. Before selecting to simulate the independent boot process, please ensure that the DCS OTP register has been programmed. The OTP registers can only be programmed once and cannot be erased.
- BOOTPIN_CONFIG_BMSP2 - Boot mode selection pin 2, GPIO2 by default
- BOOTPIN_CONFIG_BMSP1 - Boot mode selection pin 1, GPIO1 by default
- BOOTPIN_CONFIG_BMSP0 - Boot mode selection pin 0, GPIO0 by default

The BOOTDEF option can also be changed as needed. The default options in the example conform to the construction configuration described below.

Table 2 BOOTDEF Option Configuration

Options	Boot Mode Number	BMSP2	BMSP1	BMSP0	Boot Mode
ZERO_BMSPS	0	N/A	N/A	N/A	Flash Boot
ONE_BMSP	0	N/A	N/A	0	Flash Boot
	1	N/A	N/A	1	UART Boot
THREE_BMSPS	0	0	0	0	Flash Boot
	1	0	0	1	UART Boot

	2	0	1	0	Flash BootALT.1
	3	0	1	1	UART Boot ALT.1
	4	1	0	0	CAN Boot
	5	1	0	1	SPI Boot
	6	1	1	0	RAM Boot
	7	1	1	1	I2C Boot

When running the emulation boot configuration, monitor the following addresses in the memory browser:

- 0x50020000 and 0x50020003 are used for EMU BOOTPIN CONFIG
- 0x50020004 and 0x50020007 are used for EMU BOOTDEF LOW
- 0x50020008 and 0x5002000B are used for EMU BOOTDEF HIGH

When the STANDALONE_BOOT constant is set to a non-zero value, monitor the following addresses in the memory browser:

- 0x50020000 is used for EMU BOOTPIN CONFIG

External connection:

- Connect 3V or GND to BMSP2 as needed
- Connect 3V or GND to BMSP1 as needed
- Connect 3V or GND to BMSP0 as needed

Monitoring variables:

- None

2.2.3 CPU0 Secure Flash Boot

File: boot_ex3_cpu0_secure_flash.c

This example demonstrates how to use the secure flash boot mode of CPU0.

The secure flash boot performs CMAC authentication on the entry sector of the flash when device is started. If the authentication is passed, the application will start executing. For more information on secure flash boot mode, please refer to the *User Manual*.

This project demonstrates how to use the Geehy CMAC tool to generate the CMAC tag based on user CMAC key and embed this value into the flash application. In addition, the example also explains in detail how to call the CMAC API from the user application to calculate the CMAC values of other flash sectors outside the application entry flash sector.

Determine pass/fail without connecting to the debugger:

- LED2 and LED3 off=secure boot failed
- LED2 or LED3 flashing=secure boot passed, all flash CMAC failed
- LED2 and LED3 flashing=secure boot passed and all flash CMAC passed.

External connection:

- None

Monitoring variables:

- applicationCMACStatus: 0x00: CPU0's full flash CMAC authentication passed; 0xFFFFFFFF: CPU0's full flash CMAC authentication failed; other values: other errors.

2.3 CAN Routine Description

2.3.1 CAN external loopback

File: can_ex1_loopback.c

This example demonstrates the basic settings of CAN for transmitting and receiving messages on the CAN bus. The CAN peripheral is configured to transmit messages using a specific CAN ID. The messages are transmitted once per second, which is timed using a simple delay loop. The message transmitted is a 2-byte message containing an increasing mode.

This example sets the CAN controller to the external loopback test mode. The transmitted data is visible on the CANTXA pin and will be internally received back to the CAN core. Please refer to the CAN in G32R5xx *User Manual* for detailed information on the external loopback test mode.

External connection:

- None.

Monitoring variables:

- msgCount - Counter for successfully received messages
- txMsgData - Array for storing the transmitted data
- rxMsgData - Array for storing the received data

2.3.2 Interrupt used for CAN external loopback

File: can_ex2_loopback_interrupts.c

This example demonstrates the basic settings of CAN for transmitting and receiving messages on the CAN bus. The CAN peripheral is configured to transmit messages using a specific CAN ID. The messages are transmitted once per second, which is timed using a simple delay loop.

The transmitted message is a 4-byte message containing an increasing mode. Use CAN interrupt handler to confirm the message transmission and count the number of messages transmitted.

This example sets the CAN controller to the external loopback test mode. The transmitted data is visible on the CANTXA pin and will be internally received back to the CAN core. Please refer to the CAN in G32R5xx *User Manual* for detailed information on the external loopback test mode.

External connection:

- None.

Monitoring variables:

- txMsgCount - Counter for transmitted messages
- rxMsgCount - Counter for received messages
- txMsgData - Array for storing the transmitted data
- rxMsgData - Array for storing the received data
- errorFlag - Flag indicating the occurrence of an error

2.3.3 External transmission from CANA to CANB

File: can_ex3_external_transmit.c

This example initializes CAN module A and CAN module B for external communication. The CANA module is set to transmit incremental data to the CANB module, and the number of transmission times is “n”, where “n” is the value of TXCOUNT. The CANB module is set to trigger the interrupt service routine (ISR) upon receiving data. If the transmitted data does not match the received data, an error flag will be set.

The two CAN modules on the device need to be connected together through a CAN transceiver. Required hardware:

- A G32R5xx development board with two CAN transceivers.

External connection:

- Eval's CANA is located at DEVICE_GPIO_PIN_CANTXA(CANTXA) and DEVICE_GPIO_PIN_CANRXA(CANRXA)
- Eval's CANB is located at DEVICE_GPIO_PIN_CANTXB (CANTXB) and DEVICE_GPIO_PIN_CANRXB (CANRXB)

Monitoring variables:

- TXCOUNT - Adjust to set the number of messages to be transmitted

- txMsgCount - Counter for transmitted messages
- rxMsgCount - Counter for received messages
- txMsgData - Array for storing the transmitted data
- rxMsgData - Array for storing the received data
- errorFlag - Flag indicating the occurrence of an error

2.3.4 DMA used for CAN external loopback

File: can_ex4_loopback_dma.c

This example sets the CAN module for transmitting and receiving messages on the CAN bus. The CAN module is set to internally transmit a 4-byte message. Use interrupts to assert the DMA request lines, and then trigger DMA to transmit the received data from the CAN interface register to the receive buffer array. Once the transmission is completed, the data will be checked.

This example sets the CAN controller to the external loopback test mode. The transmitted data is visible on the CANTXA pin and will be internally received back to the CAN core. Please refer to the CAN in G32R5xx *User Manual* for detailed information on the external loopback test mode.

External connection:

- None.

Monitoring variables:

- txMsgCount - Counter for transmitted messages
- rxMsgCount - Counter for received messages
- txMsgData - Array for storing the transmitted data
- rxMsgData - Array for storing the received data

2.3.5 CAN transmitting and receiving configuration

File: can_ex5_transmit_receive.c

This example demonstrates the basic settings of CAN for transmitting or receiving messages on the CAN bus using a specific message ID. The CAN controller is configured according to the defined selection. When selecting the TRANSMIT definition, the CAN controller serves as a transmitter, transmitting data to the externally connected second CAN controller. If TRANSMIT is not defined, the CAN controller will serve as a receiver, waiting for the external CAN controller to transmit messages.

The CAN modules on the device need to be connected through a CAN transceiver. Required hardware:

- A G32R5xx development board with a CAN transceiver.

External connection:

- Eval's CANA is located at DEVICE_GPIO_PIN_CANTXA(CANTXA) and DEVICE_GPIO_PIN_CANRXA(CANRXA)

Monitoring variables:

- MSGCOUNT - Adjust to set the number of messages
- txMsgCount - Counter for transmitted messages
- txMsgData - Array for storing the transmitted data
- errorFlag - Flag indicating the occurrence of an error
- rxMsgCount - The initial value is the number of messages to be received, and it decreases every time a message is received

2.3.6 CAN error generation example

File: can_ex6_error_generation.c

This example demonstrates the method of processing CAN error conditions. It generates CAN data packets and transmits them through GPIO. The data packet is looped back externally and received in the CAN module. The CAN interrupt service routine reads the error status and demonstrates how to detect different error conditions.

Change the ERR_CFG definition to different error scenarios and run the example. The corresponding error flag will be set in the status variable of the canaISR() routine. Use a timer (Timer 0) to interrupt the timer periodically every CANBITRATE microseconds. When the timer is interrupted, it will transmit the required CAN frame type with the specified error conditions. The CAN modules on the device need to be connected through a CAN transceiver.

External connection:

- Eval's GPIOTX_PIN shall be connected to DEVICE_GPIO_PIN_CANRXA(CANRXA)

Monitoring variables:

- status - The variable in the canaISR function, used to check the error status.

2.3.7 CAN remote request loopback

File: can_ex7_loopback_tx_rx_remote_frame.c

This example demonstrates the basic settings of CAN for transmitting the remote frame and obtaining responses from the remote frame, and storing them in the receiving object. The CAN peripheral is configured to use specific CAN ID to transmit the remote request frames and the

remote response frame messages. Message object 3 is configured to transmit remote requests. Message object 2 is configured as a remote response object with a filter mask to accept the remote frames with any message ID and transmit the remote responses with message ID 7 and data length 8. Message object 1 is configured as the receiving object with filter messages of message ID 7, so as to store the remote response data transmitted by message object 2.

This example sets the CAN controller to the external loopback test mode. The transmitted data is visible on the CANTXA pin and will be internally received back to the CAN core. Please refer to the CAN in G32R5xx *User Manual* for detailed information on the external loopback test mode.

External connection:

- None.

Monitoring variables:

- txMsgData - Array for storing the transmitted data
- rxMsgData - Array for storing the received data

2.3.8 CAN example of demonstrating the use of mask register

File: can_ex8_mask.c

This example initializes CAN module A for receiving. When a frame that matches the filter condition is received, the data will be copied to mailbox 1, the LED will flash a few times and the code will prepare to receive the next frame. If a message from other MSGID is received, an ACK will be provided. The completion of receiving is determined by polling the CAN_NDAT_21 register. Interrupts are not used. Required hardware:

- An external CAN node that transmits messages to CAN-A on the G32R5xx MCU.

External connection:

- None.

Monitoring variables:

- rxMsgCount - Counter for received messages
- rxMsgData - Array for storing the received data

2.4 CAP Routine

2.4.1 CAP APWM application

File: cap_ex1_apwm.c

This program sets the CAP module in APWM mode. The PWM waveform will be output on GPIO5. The shadow register is used to load the next cycle/comparison value, with the PWM

frequency configured to vary between 5Hz and 10Hz.

2.4.2 CAP capture PWM application

File: cap_ex2_capture_pwm.c

This example configures PWM3A as follows:

- Counting forward mode
- The cycle starts from 500 and increases to 8000
- Switch output on PRD

CAP1 is configured to capture the time between the rising and falling edges of the output of PWM3A.

External connection

- CAP1 is on GPIO16
- PWM3A is on GPIO4
- Connect GPIO4 to GPIO16.

Monitoring variables

- cap1PassCount - Number of times of successful capture
- cap1IntCount - Interrupt count.

2.4.3 CAP APWM phase shift application

File: cap_ex3_apwm_phase_shift.c

This program sets the CAP1 and CAP2 modules to APWM mode, and generates PWM outputs of two phase shifts, with the same duty cycle and frequency values. The frequency, duty cycle, and phase values can be selected by updating the defined macro. 10KHz frequency, 50% duty cycle, and 30% phase shift value are used by default. CAP2 output is ahead of CAP1 output by 30%. GPIO5 and GPIO6 are used as CAP1/2 output, and can be detected using an analyzer/oscilloscope to observe the waveform.

2.4.4 CAP software synchronization application

File: cap_ex4_sw_sync.c

This example configures PWM3A as follows:

- Count-up mode
- The cycle starts from 500 and increases to 8000

- Switch output at the end of the cycle

CAP1, CAP2 and CAP3 are configured to capture the time between the rising and falling edges of the output of PWM3A.

External connection:

- CAP1, CAP2, and CAP3 are on GPIO16
- PWM3A is on GPIO4
- Connect GPIO4 to GPIO16.

Observed variables:

- capPassCount - Number of times of successful capture
- cap3IntCount - Interrupt count.

2.5 COMP Routine

2.5.1 COMP asynchronous configuration trigger

File: comp_ex1_asynch.c

This example enables the COMP1 COMPH comparator and feeds the asynchronous CTRIPOUTH signal to the GPIO14/OUTPUTXBAR3 pin, and feeds CTRIPH to the GPIO15/PWM8B.

COMP is configured to generate trigger signals that triggers PWM signals. CMPIN1P is used to provide positive input, while the internal DAC is configured to provide negative input. The internal DAC is configured to provide signals at VDD/2. Generate PWM signals in GPIO15 and configured to be triggered by CTRIPOUTH.

When providing low input (VSS) to CMPIN1P:

- Trigger signal (GPIO14) output is low
- PWM8B (GPIO15) provides PWM signals

When providing high input (higher than VDD/2) to CMPIN1P,

- Trigger signal (GPIO14) output is high
- PWM8B (GPIO15) is triggered and output is high

External connection

- Provide input on CMPIN1P (this pin is shared with ADCINB6)
- An oscilloscope can be used to observe the output on GPIO14 and GPIO15

Monitoring variables

- None

2.5.2 COMP digital filter configuration

File: comp_ex2_digital_filter.c

This example enables the COMP1 COMPH comparator and feeds the output to the GPIO14/OUTPUTXBAR3 pin through the digital filter.

CMPIN1P is used to provide positive input, while the internal DAC is configured to provide negative input. The internal DAC is configured to provide signals at VDD/2.

When providing low input (VSS) to CMPIN1P,

- GPIO14 output is low

When providing high input (higher than VDD/2) to CMPIN1P,

- GPIO14 output becomes high

External connection

- Provide input on CMPIN1P (this pin is shared with ADCINB6)
- The output can be observed on GPIO14

Monitoring variables

- None

2.6 DAC Routine

2.6.1 Buffer DAC enable

File: buffdac_ex1_enable.c

This example generates voltage on the buffered DAC output DACOUTA/ ADCINA0 and uses the default VDAC DAC reference setting.

External connection

- When the DAC reference setting is VDAC, the external reference voltage must be applied to the VDAC pin. This can be achieved by connecting a jumper wire from 3.3V to ADCINB3.

2.6.2 Buffer DAC random

File: buffdac_ex2_random.c

This example generates a random voltage on the buffered DAC output DACOUTA/ADCINA0, and uses the default DAC reference setting of VDAC.

External connection

- When the DAC reference setting is VDAC, the external reference voltage must be applied to the VDAC pin. This can be achieved by connecting a jumper wire from 3.3V to ADCINB3.

2.6.3 Buffer DAC sine wave (buffdac_sine)

File: buffdac_ex2_random.c

This example generates a sine wave on the buffered DAC output, DACOUTA/ADCINA0 (HSEC pin 9), and uses the default VDAC reference setting. When the DAC reference setting is VDAC, the external reference voltage must be applied to the VDAC pin. This can be achieved by connecting a jumper wire from 3.3V to ADCINB3.

2.7 DCCOMP Routine

2.7.1 DCCOMP single-shot clock verification

File: dccomp_ex1_single_shot_verification.c

This program uses the XTAL clock as the reference clock to verify the frequency of the PLLRAW clock. The dual-clock comparator module 0 is used for clock verification. clocksource0 is the reference clock (Fclk0=20MHz), and clocksource1 is the clock that needs to be verified (Fclk1=200MHz). Seed is the value loaded into the counter. Please refer to the *User Manual* for detailed information on the counter seed values to be set.

External connection

- None

Monitoring variables

- Status/Result - Status of PLLRAW clock verification

2.7.2 DCCOMP single-shot clock measurement

File: dccomp_ex2_single_shot_measurement.c

This program demonstrates single-shot measurement of INTOSC2 clock, with XTAL as the reference clock.

The dual-clock comparator module 0 is used for clock measurement. clocksource0 is the reference clock (Fclk0=20MHz), and clocksource1 is the clock that needs to be measured (Fclk1=10Mhz). As it is necessary to measure the frequency of clock 1, the initial seed is set to the maximum value of the counter.

Please refer to the *User Manual* for detailed information on the counter seed values to be set.

External connection

- None

Monitoring variables

- result - The status of whether the INTOSC2 clock measurement has been successfully completed.
- meas_freq1 - Measured clock frequency, specifically for INTOSC2.

2.7.3 DCCOMP continuous clock monitoring

File: dccomp_ex3_continuous_monitoring_of_clock.c

This program demonstrates continuous monitoring of PLL clock, with INTOSC2 as the reference clock in the system. This will trigger an interrupt in the event of any error, causing the counter to stop decreasing/reloading. The dual-clock comparator module 0 is used for clock monitoring. clocksource0 is the reference clock (Fclk0=10MHz), and clocksource1 is the clock that needs to be monitored (Fclk1=250MHz). The seeds of clock0 and clock1 are set to achieve a window of 300 microseconds. Seed is the value loaded into the counter. For demonstration, the seed value of clock1 is slightly changed to generate errors in continuous monitoring. Please refer to the *User Manual* for detailed information on the counter seed values to be set.

Note: When running in flash configuration, it is best to reset and restart after loading the example so as to eliminate any outdated flag/status.

External connection

- None

Monitoring variables

- status/result - Monitoring status of PLLRAW clock
- cnt0- Counter0 value when an error is generated
- cnt1- Counter01 value when an error is generated
- valid - Valid0 value when an error is generated

2.7.4 Detection of DCCOMP clock faults

File: dccomp_ex4_clock_fail_detect.c

This program demonstrates the clock failure detection in continuous monitoring of PLL clock, with XTAL as the oscillation clock source in the system. Once the oscillating clock fails, a DCCOMP error interrupt will be triggered, causing the counter to stop decreasing/reloading. In this example, clock failure is simulated by turning off the XTAL oscillator. Once ISR is processed, the oscillation source will be changed to INTOSC1, and PLL will be turned off.

The dual-clock comparator module 0 is used for clock monitoring. clocksource0 is the reference clock (Fclk0=20MHz), and clocksource1 is the clock that needs to be monitored (Fclk1=250MHz). Seed is the value loaded into the counter.

In the current example, XTAL is expected to be a resonator running in crystal mode, which will be turned off subsequently to simulate clock failure. If SE crystal is used, the clock connection needs to be physically disconnected on the board.

For detailed information on the seed values of the counters to be set, please refer to the User *Manual*.

Note: When running in flash configuration, it is best to reset and restart after loading the example so as to eliminate any outdated flag/status.

External connection

- None

Monitoring variables

- Status/Result - Status of clock failure detection

2.8 DMA Routine

2.8.1 DMA transfer application (dma_ex1_sram_transfer)

File: dma_ex1_sram_transfer.c

This example uses a DMA channel to transfer data from a buffer in RAMGS0 to a buffer in RAMGS1. This example repeatedly sets the PERINTFRC bit of the DMA channel until the transfer of 16 bursts (each burst is eight 16-bit words) is completed. When the entire transfer is completed, a DMA interrupt will be triggered.

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received

2.8.2 DMA transfer application (dma_ex2_sram_transfer)

File: dma_ex2_sram_transfer.c

This example uses a DMA channel to transfer the data from a buffer in RAMGS0 to a buffer in RAMGS1. This example repeatedly sets the PERINTFRC bit of the DMA channel until the transfer of 16 bursts (each burst is eight 16-bit words) is completed. When the entire transfer is completed, a DMA interrupt will be triggered.

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received

2.9 Flash Routine Description

2.9.1 Flash programming using AutoEcc

File: flashapi_ex1_program_autoecc.c

This example demonstrates how to use the API's AutoEcc generation option for Flash programming.

External connection:

- None.

Monitoring variables:

- None.

2.9.2 Flash programming switching bank mode

File: flashapi_ex6_bank_mode_switch.c

This example demonstrates how to use the Flash API to switch the Flash bank mode.

External connection:

- None.

Monitoring variables:

- None.

2.10 GPIO Routine Description

2.10.1 GPIO settings of the device

File: gpio_ex1_setup.c

Only GPIO is set. After setting, no operation is actually performed on the pins.

This routine configures the GPIO of the device into two different statuses. This segment of code is too long in order to explain how to set GPIO. In practical applications, the code lines can be combined to increase the code size and efficiency. This routine only sets GPIO, and no operation is actually performed on the pins after setting.

Generally speaking:

- All pull-up resistors have been enabled. For PWM, this may not be desirable.

- The input quality of communication ports (CAN, SPI, UART, I2C) is asynchronous.
- The input quality of the Trip pin (TZ) is asynchronous.
- The input quality of CAP and QEP signals is synchronous with SYSCLKOUT.
- The input quality of some I/O and interrupts may have a sampling window.

2.10.2 GPIO switching of the device

File: gpio_ex2_toggle.c

This routine demonstrates switching of GPIO pins in an infinite loop.

External connection:

- None

Monitoring variables:

- None

2.10.3 GPIO interrupt of the device

File: gpio_ex3_interrupt.c

This routine configures a GPIO output pin and a GPIO input pin, and then configures the GPIO input pin as the source of the external interrupt, which will switch the GPIO output pin.

External connection:

- None

Monitoring variables:

- error: None

2.10.4 External interrupt (EXTI)

File: gpio_ex4_aio_external_interrupt.c

In this routine, the AIO pin is configured as a digital input. The other two GPIO signals (connected externally to the AIO pin) are switched in the software to trigger the external interrupt through AIO224 and AIO225 (AIO224 is assigned to EXTI1, while AIO225 is assigned to EXTI2). Users need to connect these signals externally so that the program can work properly. Each interrupt is triggered in sequence: First EXTI1, then EXTI2.

External connection:

- Connect GPIO30 to AIO224. AIO224 will be assigned to EXTI1

- Connect GPIO31 to AIO225. AIO225 will be assigned to EXTI2
- GPIO34 can be monitored on an oscilloscope

Monitoring variables:

- xint1Count: The number of times through EXTI1 interrupt
- xint2Count: The number of times through EXTI2 interrupt
- loopCount: The number of times through idle loop

2.11 HRCAP Routine Description

2.11.1 HRCAP capture and calibration example

File: hrcap_ex1_capture.c

This routine configures a CAP to use the HRCAP function for capturing the time between the edges on the input GPIO2.

External connection:

- The user must provide signals to GPIO2. XCLKOUT has been configured to output GPIO, and an external jumper can be used to connect GPIO16 to GPIO2.

Monitoring variables:

- onTime1, onTime2
- offTime1, offTime2
- period1, period2

2.12 HRPWM Routine Description

2.12.1 HRPWM duty cycle control of SFO

File: hrpwm_ex1_duty_sfo.c

This routine modifies the MEP control register to display the edge displacement with a high-resolution cycle in the count-up mode due to the HRPWM control extension of the corresponding PWM module.

The routine calls the software library V8 function of MEP scale factor optimizer (SFO):

SFO v8:

- Dynamically update MEP_ScaleFactor when HRPWM is being used.
- Update the HRMSTEP register (only present in the Pwm1Regs register space) with the MEP_ScaleFactor value.

- If an error occurs: MEP_ScaleFactor is greater than the maximum value 255, return 2 (under this condition, automatic conversion may not work properly).
- Return 1 upon completion of the specified channel.
- Return 0 if the specified channel is not completed.

This routine aims to explain the function of HRPWM, and the code can be optimized to improve the code efficiency.

External connection:

- Monitor the PWM1/2/3/4 A/B pins on an oscilloscope

2.12.2 HRPWM slider

File: hrpwm_ex2_slider.c

This routine modifies the MEP control register to display the edge displacement caused by HRPWM. Due to the HRPWM logic, the control blocks for channels A and B of the corresponding PWM module will have fine edge movement.

External connection:

- Monitor the PWM1/2/3/4 A/B pins on an oscilloscope

2.12.3 HRPWM slider cycle control

File: hrpwm_ex3_prdupdown_sfo.c

This routine modifies the MEP control register to display the edge displacement of PWM in high-resolution cycle, which is in count -up and down mode due to the HRPWM control extension of their respective PWM module.

The example calls the software library V8 function of MEP scale factor optimizer (SFO):

SFO v8:

- Dynamically update MEP_ScaleFactor when HRPWM is used
- Update the HRMSTEP register (only present in the Pwm1Regs register space) with the MEP_ScaleFactor value
- If an error occurs, return 2: MEP_ScaleFactor is greater than the maximum value 255 (under this condition, automatic conversion may not work properly)
- Return 1 upon completion of the specified channel
- Return 0 if the specified channel is not completed

This program aims to explain the function of HRPWM. The code can be optimized to increase

the code efficiency.

External connection:

- Monitor the PWM1/2/3/4 A/B pins on an oscilloscope

2.12.4 HRPWM duty cycle control and UPDOWN mode

File: hrpwm_ex4_duty_updown_sfo.c

In this routine, the AIO pin is configured as a digital input. The other two GPIO signals (connected externally to the AIO pin) are switched in the software to trigger the external interrupt through AIO224 and AIO225 (AIO224 is assigned to EXTI1, while AIO225 is assigned to EXTI2). Users need to connect these signals externally so that the program can work properly. Each interrupt is triggered in sequence: First EXTI1, then EXTI2.

The example calls the software library V8 function of MEP scale factor optimizer (SFO):

SFO v8:

- Dynamically update MEP_ScaleFactor when HRPWM is being used
- Update the HRMSTEP register (only present in the Pwm1Regs register space) with the MEP_ScaleFactor value
- If an error occurs, return 2: MEP_ScaleFactor is greater than the maximum value 255 (under this condition, automatic conversion may not work properly)
- Return 1 upon completion of the specified channel
- Return 0 if the specified channel is not completed

This example aims to explain the function of HRPWM. The code can be optimized to increase the code efficiency.

External connection:

- Monitor the PWM1/2/3/4 A/B pins on an oscilloscope

2.13 I2C Routine Description

2.13.1 I2C digital loopback and FIFO interrupt

File: i2c_ex1_loopback.c

This routine uses the internal loopback test mode of the I2C module. TX and RX I2C FIFO and their interrupts are also used.

Transmit a series of data and compare it with the received data.

The data transmitted is similar to:

0000 0001

0001 0002

0002 0003

...

00FE 00FF

00FF 0000

And so on, this mode will repeat forever.

External connection:

- None

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received
- rDataPoint: Used to track the last position in the received stream for error checking

2.13.2 I2C EEPROM

File: i2c_ex2_eeprom.c

This routine will write 1-14 words to EEPROM and read them. The written data and EEPROM address are saved in the message structure i2cMsgOut. The read data will be saved in the message structure i2cMsgIn.

External connection:

For I2C <-> EEPROM communication on Eval:

- Connect the external I2C EEPROM at address 0x50
- Connect GPIO35 (SDAA) to the SDA (serial data) pin of the external EEPROM
- Connect GPIO37 (SCLA) to the SCL (serial clock) pin of the external EEPROM
- If EEPROM and r501 device are on different boards, please connect the GND pin

Monitoring variables:

- i2cMsgOut: Message of data to be written to EEPROM
- i2cMsgIn: Message of data read from EEPROM

2.13.3 I2C EEPROM POLLING

File: i2c_ex2_eeprom.c

This program will demonstrate how to use the I2C polling method to execute different EEPROM write and read commands.

External connection:

For I2C <-> EEPROM communication on Eval:

- Connect the external I2C EEPROM at address 0x50
- Connect GPIO35 (SDAA) to the SDA (serial data) pin of the external EEPROM
- Connect GPIO37 (SCLA) to the SCL (serial clock) pin of the external EEPROM
- If EEPROM and r501 device are on different boards, please connect the GND pin

Monitoring variables:

- i2cMsgOut: Message of data to be written to EEPROM
- i2cMsgIn: Message of data read from EEPROM

2.14 Interrupt Routine Description

2.14.1 External interrupt edge trigger

File: interrupt_ex1_external.c

This routine will set GPIO0 to EXTI1 (EXTI_LINE_4), and set GPIO1 to EXTI2 (EXTI_LINE_5). The other two GPIO signals are used to trigger interrupts (GPIO10 triggers EXTI1, and GPIO11 triggers EXTI2). Users need to connect these signals externally so that the program can work properly.

EXTI1 input is synchronized with SYSCLKOUT.

EXTI2 has a long limiting time - 6 samples each with 510*SYSCLKOUT.

GPIO16 will become high outside of interrupts and become low during interrupts. This signal can be monitored on an oscilloscope.

Each interrupt is triggered in sequence - First EXTI1, then EXTI2.

External connection:

- Connect GPIO10 to GPIO0. (GPIO0 will be assigned to EXTI1)
- Connect GPIO11 to GPIO1. (GPIO1 will be assigned to EXTI2)

Monitoring variables:

- xint1Count: The number of EXTI1 interrupts triggered
- xint2Count: The number of EXTI2 interrupts triggered
- loopCount: The number of execution times of idle loop

2.14.2 Multi-interrupt handling

File: interrupt_ex2_with_i2c_uart_spi_loopback.c

This routine is used to demonstrate how to handle multiple communication peripheral interrupts, such as I2C, UART, and SPI digital loopback, in one routine. Data transmission will be completed using FIFO interrupt.

It uses the internal loopback test mode of these modules. It also uses TX and RX FIFO and their interrupts. No other hardware configuration is required except for the boot mode pin configuration.

Transmit a series of data and compare it with the received data.

For I2C and UART, the data transmitted is as follows:

0000 0001

0001 0002

0002 0003

...

00FE 00FF

00FF 0000

...

For SPI, the data transmitted is as follows:

0000 0001

0001 0002

0002 0003

...

FFFE FFFF

FFFF 0000

...

This mode will repeat forever.

External connection:

- None

Monitoring variables:

- sDatai2cA: Data transmitted through I2C
- rDatai2cA: Data received through I2C
- rDataPoint: Used to track the last position in the received I2C data stream for error checking
- sDataspiA: Data transmitted through SPI
- rDataspiA: Data received through SPI
- rDataPointspiA: Used to track the last position in the received SPI data stream for error checking
- sDatauartA: Data to be transmitted by the serial port
- rDatauartA: Data received by the serial port
- rDataPointA: Track the position in the serial data stream. This is used to check the transmitted data

2.14.3 Interrupt priority setting (software)

File: interrupt_ex3_sw_prioritization.c

This routine demonstrates software priority ranking of interrupts through the CPU timer interrupt. By enabling the interrupt nesting, software priority ranking of interrupts can be achieved.

In this device, the hardware priority of CPU timers 0, 1, and 2 is so set as to ensure that Timer 0 has the highest priority and Timer 2 has the lowest priority. This example configures the priority of CPU timers 0, 1, and 2 in the software, with Timer 2 having the highest priority and Timer 0 having the lowest priority, and outputs the execution sequence.

For most applications, the interrupt priority ranking by hardware is sufficient. For the applications that require custom priority ranking, this example illustrates how to achieve this through software. The user-specific priority can be configured in the sw_prioritized_isr_level.h header file.

To enable interrupt nesting, operations need to be performed in ISR according to the following sequence:

- Step 1: Set the global priority: Modify the IER register to allow CPU interrupts to be serviced according to a higher user priority.

Note: At this point, IER has been saved in the stack.

- Step 2: Set the group priority (optional): Modify the group interrupts with a higher user

setting priority for service

- Step 3: Enable the interrupt
- Step 4: Run the main part of ISR.
- Step 5: Disable the interrupt
- Step 6: Return from ISR

External connection:

- None

Monitoring variables:

- traceISR: Execution sequence of ISR

2.14.4 PWM real-time interrupt

File: interrupt_ex4_pwm_realtime_interrupt.c

This example configures the PWM1 timer and a counter is added every time ISR is executed. PWM interrupts can be configured as the time critical to demonstrate the real-time mode function and real-time interrupt capability. This example uses 2 LED0 - LED1 switches in the main loop, and LED2 switches in the PWM timer interrupt. FREE_SOFT bit and DBGIER.INT3 bit must be set so that after the command halts, the PWM1 interrupt can enable the time critical and run in real-time mode.

Key points for running this example:

- Add the following observed variables and enable continuous refresh.
- Enable real-time mode (Run->Advanced->Enable real-time mode).
- At the beginning, the DBGIER register is set to 0, and the PWM emulation mode is set to PWM_EMULATION_STOP_AFTER_NEXT_TB (FREE_SOFT = 0).
- When the application is running, you will find two LED are switching, and the observed variables Pwm1TimerIntCount and Pwm1Regs.TBCTR are updated.
- When the application pauses, the two LED stop switching and the observed variables remain unchanged. When the debugger pauses, the PWM counter stops.
- To enable the PWM counter to run when the debugger pause, set the emulation mode to PWM_EMULATION_FREE_RUN (FREE_SOFT = 2). You will find Pwm1Regs.TBCTR is running, but Pwm1TimerIntCount remains unchanged. This means that the PWM counter is running, but the interrupt service routine has not been executed.
- To enable the real-time interrupt, please set DBGIER.INT3 = 1 (PWM1 interrupt is part of NVIC Group 3). You will find that Pwm1TimerIntCount is increasing and the LED starts to

switch. Even when the debugger pauses, the PWM interrupt service routine will still be executed.

External connection:

- None

Monitoring variables:

- Pwm1TimerIntCount: PWM1 interrupt service routine counter
- Pwm1Regs.TBCTR.TBCTR: PWM1 time reference counter
- Pwm1Regs.TBCTL.FREE_SOFT: Set it to 2 to enable free running
- DBGIER.INT3: Set it to 1 to enable real-time interrupts

2.14.5 Interrupt priority setting (hardware)

File: interrupt_ex5_hw_prioritization.c

This routine demonstrates the implementation of hardware interrupt priority through CPU timer interrupts.

The program configures the priority of CPU timers 0, 1, and 2, with Timer 2 having the highest priority and Timer 0 having the lowest priority, and prints the tracking of the execution sequence.

For most applications, the default interrupt priority setting is sufficient. For applications that require custom priority setting, this program demonstrates how to implement it.

External connection:

- None

Monitoring variables:

- traceISR: Execution sequence of ISR

2.15 IPC Routine Description

2.15.1 IPC mailbox uses interrupt mode

File: ipc_ex1_mailbox_interrupt_cpu0.c

This example demonstrates how to use the mailbox of the inter-processor communication (IPC) module to transmit and receive data between two CPU in interrupt mode.

In this example:

1. CPU0 transmits messages to CPU1 in interrupt mode through the IPC module.

2. CPU1 transmits messages back to CPU0 in interrupt mode.

3. CPU0 receives messages from CPU1 in interrupt mode.

Note: The IPC routine includes CPU0 and CPU1 projects. Before compiling the CPU0 project, please compile the CPU1 project first.

To run this application, it is necessary to use a serial terminal to open the COM port. The settings are as follows:

- Find the correct COM port
- Number of bits per second=115200
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None

External connection:

- GPIO28 is UART - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART - TXD (Pin2 connected to serial DB9 cable, PC - RX)

Monitoring variables:

- None.

2.15.2 IPC mailbox uses polling mode

File: ipc_ex2_mailbox_polling_cpu0.c

This example demonstrates how to use the mailbox of the inter-processor communication (IPC) module to transmit and receive data between two CPU in polling mode.

In this example:

1. CPU0 transmits messages to CPU1 in polling mode through the IPC module.

2. CPU1 transmits messages back to CPU0 in polling mode.

3. CPU0 receives messages from CPU1 in polling mode, and so on.

Note: The IPC routine includes CPU0 and CPU1 projects. Before compiling the CPU0 project, please compile the CPU1 project first.

To run this application, it is necessary to use a serial terminal to open the COM port. The settings are as follows:

- Find the correct COM port
- Number of bits per second=115200

- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None

External connection:

- GPIO28 is UART - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART - TXD (Pin2 connected to serial DB9 cable, PC - RX)

Monitoring variables:

- None.

2.15.3 IPC general interrupt

File: ipc_ex3_general_interrupt_cpu0.c

This example demonstrates how to use the general interrupt function of the inter-processor communication (IPC) module. CPU0 and CPU1 mutually trigger their respective general interrupts.

Note: The IPC routine includes CPU0 and CPU1 projects. Before compiling the CPU0 project, please compile the CPU1 project first.

To run this application, it is necessary to use a serial terminal to open the COM port. The settings are as follows:

- Find the correct COM port
- Number of bits per second=115200
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None

External connection:

- GPIO28 is UART - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART - TXD (Pin2 connected to serial DB9 cable, PC - RX)

Monitoring variables:

- None.

2.15.4 IPC resource sharing

File: ipc_ex4_resource_share_cpu0.c

This example demonstrates how to use the general interrupt and mailbox of the inter-processor communication (IPC) module, and CPU0 and CPU1 are mutually exclusive and can access the same memory space and the same peripheral (UART_A).

Note: The IPC routine includes CPU0 and CPU1 projects. Before compiling the CPU0 project, please compile the CPU1 project first.

To run this application, it is necessary to use a serial terminal to open the COM port. The settings are as follows:

- Find the correct COM port
- Number of bits per second=115200
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None

External connection:

- GPIO28 is UART - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART - TXD (Pin2 connected to serial DB9 cable, PC - RX)

Monitoring variables:

- None.

2.15.5 IPC event control

File: ipc_ex5_event_control_cpu0.c

This example demonstrates how to use the event control function of the inter-processor communication (IPC) module. CPU0 transmits tasks to CPU1 through the TASK register, and CPU1 executes different tasks based on different task ID.

In this example:

1. CPU0 transmits a task with task ID 0x55AA0001 to CPU1 through the PC_issueTask function, and then waits for CPU1 to execute the task.
2. CPU1 obtains the task with task ID 0x55AA0001 through the IPC_fetchTask function, and then executes the task. In this routine, LED2 flashes 10 times per second.

3. CPU0 continues to transmit the task with task ID 0x55AA0002, and then waits for CPU1 to complete the task.
4. CPU1 obtains the 0x55AA0002 task and executes it. In this task, CPU1 enters low-power mode through the __WFE instruction.
5. CPU0 judges whether CPU1 has entered low-power mode, and issues a task through the IPC_issueTask function to wake CPU1 up from low-power mode.
6. CPU1 is awakened from low-power mode by CPU0 and continues to execute the tasks.

Note: The IPC routine includes CPU0 and CPU1 projects. Before compiling the CPU0 project, please compile the CPU1 project first.

To run this application, it is necessary to use a serial terminal to open the COM port. The settings are as follows:

- Find the correct COM port
- Number of bits per second=115200
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None

External connection:

- GPIO28 is UART - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART - TXD (Pin2 connected to serial DB9 cable, PC - RX)

Monitoring variables:

- None.

2.16 LED Routine Description

2.16.1 LED flashing

File: led_ex1_blinky.c

This routine demonstrates how to make a LED flash.

External connection:

- None

Monitoring variables:

- None

2.17 LIN Routine Description

2.17.1 LIN internal loopback with interrupts

File: lin_ex1_loopback_interrupts.c

This routine configures the LIN module as the master mode for internal loopback with interrupts. This module is set to perform data transfer for eight times, using different transfer ID and changing transfer data. After receiving the ID header, an interrupt will be triggered on Line 0, and the interrupt service routine (ISR) will be called to check if the received data is accurate.

This routine can be adjusted to use interrupt line 1 by canceling the annotation "LIN_setInterruptLevel1()".

External connection:

- None

Monitoring variables:

- txData: Array of transmitted data
- rxData: Array of received data
- result: Routine completion status (PASS = 0xABCD, FAIL = 0xFFFF)
- level0Count: The quantity of interrupt line 0
- level1Count: The quantity of interrupt line 1

2.17.2 Internal loopback and interrupt in LIN UART mode

File: lin_ex2_uart_loopback.c

This routine configures the LIN module to UART mode and uses interrupts for internal loopback. The LIN module has UART function for specific characters and frame lengths in non-multi-buffer mode. This module is set to continuously transmit one character, wait for the character to be received, and repeat it.

External connection:

- None

Monitoring variables:

- None

2.17.3 Internal loopback and DMA in LIN UART mode

File: lin_ex3_uart_dma.c

This example configures the LIN module to use DMA for internal loopback in UART mode. The LIN module runs in multi-buffer mode with the set character and frame length as UART. When there is enough space in the transfer buffer of the LINTD0 and LINTD1 registers, DMA will transfer data from the global variable sData to these transfer registers. Once the receive buffer in the LINRD0 and LINRD1 registers contains data, DMA transfers the data to the global variable rdata.

When all data has been put into rData, the data validity will be checked in the ISR of a DMA channel.

External connection:

- None

Monitoring variables:

- sData: Data to be transmitted
- rData - Data received

2.17.4 LIN internal loopback without interrupts (polling mode)

File: lin_ex4_loopback_polling.c

This routine configures the LIN module as the master mode for internal loopback test without using interrupts. This module is set to perform data transfer for eight times, with transfer ID and changing transfer data. Wait to receive the ID header. Then check if the received data is accurate.

External connection:

- None

Monitoring variables:

- None

2.18 Low-power Mode Routine Description

2.18.1 Entering and exiting the idle mode

File: lpm_ex1_idlewake_gpio.c

This routine sets the device in idle mode, and then wakes it up by triggering the EXTI1 on the falling edge of GPIO0.

The GPIO0 pin must be pulled down from a high level by the external agent to wake up. The GPIO0 is configured as the EXTI1 pin to trigger an EXTI1 interrupt when a falling edge is detected.

Initially, GPIO0 is pulled up externally. To wake up the device from idle mode by triggering the EXTI1 interrupt, please pull down the GPIO0 (falling edge). When the time for GPIO0 remaining low reaches the time indicated in the device data sheet, the wake-up process begins.

Before entering idle mode, pull up the GPIO1 and pull it down in the external interrupt ISR.

External connection:

- Pull down the GPIO0 to wake up the device
- When the device wakes up, GPIO1 will be at a low level and LED1 will start flashing

2.18.2 Entering and exiting the idle mode

File: lpm_ex2_idlewake_watchdog.c

This routine sets the device in idle mode and then uses a watchdog timer to wake up the device.

When the watchdog timer overflows, the device will wake up from idle mode and trigger an interrupt.

To change the overflow time of the counter, a prescaler is set for the watchdog timer.

Before entering idle mode, GPIO1 is pulled up, and when waking up the interrupt service routine, GPIO1 is pulled down.

External connection:

- When the device wakes up, GPIO1 will be at a low level and LED1 will start flashing

2.18.3 Entering and exiting the halt mode

File: lpm_ex5_haltwake_gpio.c

This example sets the device in HALT mode. If the lowest possible current consumption is required in HALT mode, the JTAG connector must be unplugged from the device board when the device is in HALT mode.

For applications that require the minimum power consumption in HALT mode, the application software shall turn off the crystal oscillator (XTAL) by setting XTALCR.OSCOFF bit or using the driverlib function SysCtl_turnOffOsc(SYSCTL_OSCSRC_XTAL) before entering HALT mode. If the OSCCLK source is configured as XTAL, the application shall switch the OSSCLK source to INTOSC1 or INTOSC2 before setting XTALCR.OSCOFF.

This example sets the device in HALT mode and then wakes it up from HALT mode through an LPM wake-up pin.

The pin GPIO0 is configured as the LPM wake-up pin to trigger the WAKEINT interrupt when a low pulse is detected. The GPIO0 pin must be pulled down from a high level by the external

agent to wake up.

Before entering STANDBY mode, GPIO1 is pulled up, and is pulled down when waking up ISR.

External connection:

- When the device wakes up, GPIO1 will be at a low level and LED1 will start flashing

2.18.4 Entering and exiting the halt mode

File: lpm_ex6_haltwake_gpio_watchdog.c

This routine sets the device in HALT mode. If the lowest possible current consumption is required in HALT mode, the JTAG connector must be removed from the device board when the device is in HALT mode.

For applications that require the minimum power consumption in HALT mode, the application software shall turn off the XTAL by setting XTALCR.OSCOFF bit or using the driverlib function SysCtl_turnOffOsc(SYSCTL_OSCSRC_XTAL) before entering HALT mode. If the OSCCLK source is configured as XTAL, the application shall switch the OSSCLK source to INTOSC1 or INTOSC2 before setting XTALCR.OSCOFF.

This routine sets the device in HALT mode and then uses the LPM wake-up pin to wake up the device.

The pin GPIO0 is configured as the LPM wake-up pin to trigger the WAKEINT interrupt when a low pulse is detected. The GPIO0 pin must be pulled down from a high level by the external agent to wake up.

In this routine, the watchdog timer is clocked and configured as a timeout mechanism to generate watchdog reset.

Before entering STANDBY mode, GPIO1 is pulled up, and be pulled down when waking up ISR.

External connection:

- After the device wakes up, GPIO1 will be at a low level and LED1 will start flashing

2.19 PWM Routine

2.19.1 PWM trigger application

File: pwm_ex1_trip_zone.c

This example configures PWM1 and PWM2 as follows:

- The primary trigger source for PWM1 is TZ1
- The cycle-by-cycle trigger source for PWM2 is TZ1

Initially connect TZ1 to a high level. In the test process, monitor the output of PWM1 or PWM2

through an oscilloscope. Pull down TZ1 to see the effect.

External connection

- PWM1A is on GPIO0
- PWM2A is on GPIO2
- TZ1 is on GPIO12

This example also uses the input XBAR. GPIO12 (external trigger) is routed to input XBAR and then to TZ1.

The TZ event is defined as a one-time trigger for PWM1A and a cycle-by-cycle trigger for PWM2A.

2.19.2 PWM counting up and down

File: pwm_ex2_updown.c

This example configures PWM1, PWM2, and PWM3 to generate waveforms of PWMxA and PWMxB with independent modulation. The comparison values CMPA and CMPB are modified in the ISR of PWM. In this example, the TB counter is in count-up/down mode. View the waveforms of PWM1A/B (GPIO0 and GPIO1), PWM2A/B (GPIO2 and GPIO3) and PWM3A/B (GPIO4 and GPIO5) on an oscilloscope.

2.19.3 PWM synchronization application

File: pwm_ex3_synchronization.c

This example configures PWM1, PWM2, PWM3, and PWM4 as follows:

- As a synchronization source, PWM1 has no phase shift
- PWM2 phase shift of 300 TBCLKs
- PWM3 phase shift of 600 TBCLKs
- PWM4 phase shift of 900 TBCLKs

External connection:

- GPIO0 PWM1A
- GPIO1 PWM1B
- GPIO2 PWM2A
- GPIO3 PWM2B
- GPIO4 PWM3A

- GPIO5 PWM3B
- GPIO6 PWM4A
- GPIO7 PWM4B

Monitoring variables:

- None.

2.19.4 PWM digital comparison

File: pwm_ex4_digital_compare.c

This example configures PWM1 as follows:

- Use DCAEVT1 to make PWM output a low level
- GPIO25 is used as input to input the exchange input 1
- Input 1 (from input switching) is used as the source for DCAEVT1
- To test the trigger, enable the pull-up resistor of GPIO25 and pull this pin to ground

External connection

- GPIO0 PWM1A
- GPIO1 PWM1B
- GPIO25 TZ1: Pull down this pin to trigger PWM

Monitoring variables

- None.

2.19.5 PWM digital comparison event filter masking window

File: pwm_ex5_digital_compare_event_filter.c

This example configures PWM1 as follows:

- DCAEVT1 of PWM1 forces the PWM output to a low level
- GPIO25 is used as input to INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- To test the trip, enable the pull-up resistor of GPIO25 and pull this pin to GND
- DCBEVT1 of PWM1 forces the PWM output to a low level
- GPIO25 is used as input to INPUT XBAR INPUT1

- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- To test the trip, enable the pull-up resistor of GPIO25 and pull this pin to GND
- DCBEVT1 uses the filtered version of DCBEVT1
- The DCFILT signal uses a blank window to ignore the DC blank window for duration of the DCBEVT1

External connection

- GPIO0 PWM1A
- GPIO1 PWM1B
- GPIO25 TRIPIN1: Pull down this pin to trigger PWM

Monitoring variables

- None.

2.19.6 PWM valley switch

File: pwm_ex6_valley_switching.c

This example configures PWM1 as follows:

- PWM1 has DCAEVT1 to force the PWM output to a low level
- GPIO25 is used as input of INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25 is set as the output and switches in the main loop to trigger PWM
- PWM1 has DCBEVT1 to force the PWM output to a low level
- GPIO25 is used as input of INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25 is set as the output and switches in the main loop to trigger PWM
- DCBEVT1 uses the filtered version of DCBEVT1
- The DCFILT signal is delayed by a software-defined DELAY value using a valley switch module.

External connection

- GPIO0 PWM1A
- GPIO1 PWM1B

- GPIO25 TRIPIN1 (output pin, switched through software)

Monitoring variables

- None.

2.19.7 PWM digital comparison edge filter

File: pwm_ex7_edge_filter.c

This example configures PWM1 as follows:

- PWM1 forces the PWM output to a low level as the CBC source through DCBEVT2
- GPIO25 is used as input to INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCBEVT2
- GPIO25 is set as the output and switches in the main loop to trigger PWM
- DCBEVT2 is the source for DCFILT
- DCFILT will calculate the edge of DCBEVT2 and generate a signal at the fourth edge of DCBEVT2 to trigger PWM

External connection

- GPIO0 PWM1A
- GPIO1 PWM1B
- GPIO25 TRIPIN1 (output pin, switched through software)

Monitoring variables

- None.

2.19.8 PWM dead band

File: pwm_ex8_deadband.c

This example configures PWM1 to PWM6 as follows:

- PWM1 dead band is disabled (reference)
- PWM2 dead band activates high level
- PWM3 dead band activates low level
- PWM4 dead band activates high-level complementation
- PWM5 dead band activates low-level complementation

- PWM6 dead band output exchange (exchange A and B outputs)

External connection

- GPIO0 PWM1A
- GPIO1 PWM1B
- GPIO2 PWM2A
- GPIO3 PWM2B
- GPIO4 PWM3A
- GPIO5 PWM3B
- GPIO6 PWM4A
- GPIO7 PWM4B
- GPIO8 PWM5A
- GPIO9 PWM5B
- GPIO10 PWM6A
- GPIO11 PWM6B

Monitoring variables

- None.

2.19.9 PWM DMA

File: pwm_ex9_dma.c

This example configures PWM1f and DMA as follows:

- PWM1 is set to generate PWM waveform
- DMA5 is set to update CMPAHR, CMPA, CMPBHR and CMPB in each cycle, using the next value in the configuration array. This allows users to create a DMA enabled fifo for all CMPx and CMPxHR registers to generate non-traditional PWM waveforms.
- DMA6 is set to update TBPHSHR, TBPHS, TBPRDHR and TBPRD with the next value in the configuration array in each cycle.
- Other registers such as AQCTL can also be controlled through DMA by following the same process. (Not used in this example)

External connection

- GPIO0 PWM1A

- GPIO1 PWM1B

Monitoring variables

- None.

2.19.10 PWM chopper

File: pwm_ex10_chopper.c

This example configures PWM1, PWM2, PWM3, and PWM4 as follows:

- PWM1 chopper is disabled (reference)
- PWM2 chopper is enabled, with a duty cycle of 1/8
- PWM3 chopper is enabled, with a duty cycle of 6/8
- PWM4 chopper is enabled, with a duty cycle of 1/2, and the pulse is enabled

External connection

- GPIO0 PWM1A
- GPIO1 PWM1B
- GPIO2 PWM2A
- GPIO3 PWM2B
- GPIO4 PWM3A
- GPIO5 PWM3B
- GPIO6 PWM4A
- GPIO7 PWM4B

Monitoring variables

- None.

2.19.11 PWM signal configuration

File: pwm_ex11_configure_signal.c

This example configures PWM1, PWM2, and PWM3 to generate signals of the required frequency and duty cycle. It also configures the phase between these modules. A signal with a frequency of 10kHz and a duty cycle of 0.5 is configured on PWMxA and PWMxB, and PWMxB is inverted. In addition, a 120-degree phase is configured between PWM1 to PWM3 signals.

During the test period, use an oscilloscope to monitor the outputs of PWM1, PWM2, and/or

PWM3.

- PWM1A is on GPIO0
- PWM1B is on GPIO1
- PWM2A is on GPIO2
- PWM2B is on GPIO3
- PWM3A is on GPIO4
- PWM3B is on GPIO5

2.19.12 Implementation of single-pulse mode

File: pwm_ex12_monoshot_mode.c

This example demonstrates how to generate a single-pulse PWM output based on an external trigger, that is, to generate only one pulse output when receiving an external trigger. The next pulse will only be generated when the next trigger arrives. This example utilizes external synchronization and T1 action qualifier event function to implement the required output.

PWM1 is used to generate the single-pulse output, while PWM2 is used as an external trigger. No external connection is required, as PWM2A will automatically act as a trigger and feed through the input XBAR.

When receiving an external trigger, PWM1 is configured to generate a single pulse of 0.4 microsecond. This purpose is served by enabling the phase synchronization function and configuring PWMxSYNCI as EXTSYNCIN1. Moreover, configure PWMxSYNCI as the T1 event of the action qualifier so as to set the output to a high level during the "CTR=PRD" action.

PWM2 is configured to generate a signal with a frequency of 125 kHz and a duty cycle of 1% (used to simulate a rising edge trigger), and is routed to EXTSYNCIN1 through the input XBAR.

Observe GPIO0 (PWM1A: single-pulse output) and GPIO2 (PWM2: external trigger) on an oscilloscope.

2.19.13 PWM action qualifier (pwm_up_aq)

File: pwm_ex13_up_aq.c

This example configures PWM1, PWM2, and PWM3 to generate a waveform with independent modulation, where PWMxA and PWMxB are modulated separately. The comparison values CMPA and CMPB are modified in the ISR of PWM. In this example, the TB counter is in count-up mode. View the waveforms of PWM1A/B (GPIO0 and GPIO1), PWM2A/B (GPIO2 and GPIO3) and PWM3A/B (GPIO4 and GPIO5) on an oscilloscope.

2.19.14 Independent modulation waveform of PWM

File: pwm_ex14_global_load_use_case.c

The case described in the application report are is follows:

- The output frequency of PWM1/2/3 is 500 kHz
- PWM2 has a 120-degree phase shift relative to PWM1
- PWM3 has a 240-degree phase shift relative to PWM1
- The duty cycle of PWM1/2/3 is 45%
- The activation high-level complementary signal of PWM1/2/3 are paired, and the delay for the rising/falling edge is 200 nanoseconds
- Periodic trigger protection through the comparator signal on PWM2
- Single-trigger protection through GPIO on PWM3
- Whenever the time reference counter reaches zero, an interrupt signal will be generated
- Global loading to support the asynchronous updates of action qualifier settings
- Connect CMPA/CMPB of PWM1 to PWM2 and PWM3

2.20 QEP Routine

2.20.1 QEP frequency measurement

File: qep_ex1_freq_cal.c

This example will use the QEP module to calculate the frequency of the input signal. PWM1A is configured to generate input signals with a frequency of 5kHz. It will interrupt once per cycle and call the frequency calculation function. This example uses the IQMath library to simplify the high-accuracy calculation.

In addition to the main example file, the following files shall also be included in this project:

- qep_ex1_calculation.c – Contains the frequency calculation function
- qep_ex1_calculation.h - Contains the initialization value of frequency structure

The configuration of this example is as follows:

- The maximum frequency is configured to 10KHz (baseFreq)
- The minimum frequency is assumed to be 50Hz for capturing prescale selection

SPEED_FR: High-frequency measurement is made by calculating the external input pulse, with a counting time of 10ms (the timer is set to 100Hz).

SPEED_PR: Low-frequency measurement is obtained by measuring the time period of the input edge. The average value of time measurement is taken from 64 edges so as to obtain better results, and the capture unit uses the prescaled APBCLK for time measurement.

Please note that the prescaler of the capture unit clock is selected to ensure that the capture timer does not overflow at the required minimum frequency. This demonstration will continue to run until the user stops it.

For more information on frequency calculation, please refer to the annotation at the beginning of qep_ex1_calculation.c.

External connection

- Connect GPIO10/QEP1A to GPIO0/PWM1A

Monitoring variables

- freq.freqHzFR - Frequency measurement using the location counter/unit timeout
- freq.freqHzPR - Frequency measurement using the capture unit

2.20.2 QEP monitoring location and speed

File: qep_ex2_pos_speed.c

This example uses the capture unit of the QEP module to provide location and speed measurements, with the speed measurements in time unit. PWM1 and a GPIO are configured to generate analog QEP signals. The PWM module will interrupt once per cycle, and call the location/speed calculation function. This example uses the IQMath library to simplify the high-accuracy calculation.

In addition to the main example file, the following files shall also be included in this project:

- qep_ex2_calculation.c – Contains the location/speed calculation function
- qep_ex2_calculation.h - Contains the initialization value of location/speed structure

The configuration of this example is as follows:

- The maximum speed is configured to 6000rpm (baseRPM)
- The minimum speed is assumed to 10rpm for capture prescale selection
- Pole pairs are configured as 2 (polePairs)
- The encoder resolution is configured to 4000 counts per revolution (mechScaler)
- This means 4000/4=1000 lines per revolution quadrature encoder (simulated by PWM1)
- PWM1 (analog QEP encoder signal) is configured with a frequency of 5kHz or 300rpm (= 4 * 5000 cnts/sec * 60 sec/min)/ 4000 cnts/rev)

SPEEDRPM_FR: Obtain high-speed measurement by counting the QEP input pulses for 10ms (with the unit timer set to 100Hz).

SPEEDRPM_FR = (Location increment/10ms) * 60 rpm

SPEEDRPM_PR: Obtain low-speed measurements by measuring the time period of QEP edge. The time measurement is averaged on 64 edges to obtain better results, and the capture unit uses the prescale APBCLK for time measurement.

Please note that the prescaler of the capture unit clock is selected to ensure that the capture timer does not overflow at the required minimum frequency. This demonstration will run indefinitely until the user stops it.

For more information on location/speed calculation, please refer to the annotation at the beginning of qep_ex2_calculation.c.

External connection

- Connect GPIO6/QEP1A to GPIO0/PWM1A (simulating QEP A-phase signal)
- Connect GPIO7/QEP1B to GPIO1/PWM1B (simulating QEP B-phase signal)
- Connect GPIO9/QEP1I to GPIO4 (simulating QEP index signal)

Observed variables

- posSpeed.speedRPMFR - Speed measured using QEP location counter (rpm)
- posSpeed.speedRPMPR - Speed measured using the capture unit (rpm)
- posSpeed.thetaMech - Mechanical angle of the motor (Q15)
- posSpeed.thetaElec - Electrical angle of the motor (Q15)

2.20.3 QEP uses interrupts to calculate signal frequency

File: qep_ex4_freq_cal_interrupt.c

This example will use the QEP module to calculate the frequency of the input signal. PWM1A is configured to generate this input signal at a frequency of 5 kHz. QEP unit timeout has been set, and an interrupt will be generated every UNIT_PERIOD microseconds, and frequency calculation will continue.

The configuration of this example is as follows:

- The PWM frequency is specified as 5000Hz
- UNIT_PERIOD is specified as 10000 microseconds
- The minimum frequency is (1/(2*10 milliseconds)), i.e. 50Hz
- The maximum frequency can be (2^32)/((2*10 milliseconds))

- The resolution of frequency measurement is 50Hz

Frequency: Obtain frequency measurement by calculating the number of internal and external input pulses per unit cycle (with the unit timer set to 10 milliseconds).

External connection:

- Connect GPIO6/QEP1A to GPIO10/PWMA

Observed variables:

- freq - Frequency measurement using the location counter/unit timeout
- pass - If the measured frequency matches the PWM frequency, pass=1; otherwise, it is 0

2.20.4 Perception of motor speed and direction in quartile encoder mode

File: qep_ex5_speed_dir_motor.c

This example can be used for perception of the motor speed and direction using the QEP in quartile encoder mode. PWM1A is configured to simulate the motor encoder signals on Pins A and B, with a frequency of 5kHz and a phase difference of 90 degrees (in order to run this example without a motor). QEP unit timeout is set, an interrupt will be generated every UNIT_PERIOD microseconds, and the speed will be continuously calculated based on the direction of the motor.

The configuration of this example is as follows

- The PWM frequency is specified as 5000Hz
- UNIT_PERIOD is specified as 10000 microseconds
- The frequency of the simulated quartile coded signal is 20000Hz (4*5000)
- Assuming the number of encoder holes is 1000
- The simulated motor speed is 300rpm (5000 * (60/1000))

freq: Frequency of the simulated quartile coded signal measured by calculating the external input pulse within UNIT_PERIOD (the unit timer is set to 10 milliseconds).

speed: Motor speed measured in rpm

dir: Indicates clockwise (1) or counterclockwise (-1)

External connection (if the motor encoder signal is simulated through PWM)

- Connect GPIO6/QEP1A to GPIO10/PWMA (simulating QEP A-phase signal)
- Connect GPIO7/QEP1B to GPIO11/PWM1B (simulating QEP B-phase signal)

Monitoring variables

- freq: Simulated motor frequency measured by calculating the external input pulse within UNIT_PERIOD (the unit timer is set to 10 milliseconds).
- speed: Motor speed measured in rpm
- dir: Indicates clockwise (1) or counterclockwise (-1)
- pass - If the measured quartile frequency matches the input quartile frequency (4 * PWM frequency), pass=1; otherwise, fail=1 (** only when "MOTOR" is annotated)

2.21 SDF Routine Description

2.21.1 SDF filter CPU synchronization

File: sdf_ex1_filter_sync_cpuread.c

In this example, the SDF filter data is read by CPU in the SDF ISR routine. The SDF configuration is as follows:

- The SDF used in this example is SDF1
- Selected input control mode: MODE0
- Comparator settings:
  - Select Sinc3 filter
  - OSR = 32
  - HLT = 0x7FFF (high threshold setting)
  - LLT = 0x0000 (low threshold setting)
- Data filter settings:
  - Enable all the 4 filter modules
  - Select Sinc3 filter
  - OSR = 128
  - All the four filters are synchronized using MFE (main filter enable bit)
  - The filter output is represented in 16-bit format
  - To convert a 25-bit data filter to a 16-bit format, users need to shift the Sinc3 filter with OSR=128 to the right by 7 bits
- Interrupt module settings for SDF filter:
  - Disable all the four high threshold comparator interrupts
  - Disable all the four low threshold comparator interrupts

- Disable all the four modulator fault interrupts
- When there is new filter data available, all the four filters will generate interrupts

External connection:

- Connect Sigma - Delta stream to (SD-D1, SD-C1 to SD-D4, SD-C4) on GPIO24-GPIO31

Monitoring variables:

- filter1Result - Output of Filter 1
- filter2Result - Output of Filter 2
- filter3Result - Output of Filter 3
- filter4Result - Output of Filter 4

2.21.2 Synchronous DMA of SDF filter

File: sdf_ex3_filter_sync_dmaread.c

In this example, the SDF filter data is read by DMA. The SDF configuration is as follows:

- The SDF used in this example is SDF1
- Selected input control mode: MODE0
- Comparator settings:
  - Select Sinc3 filter
  - OSR = 32
  - HLT = 0x7FFF (high threshold setting)
  - LLT = 0x0000 (low threshold setting)
- Data filter settings:
  - Enable all the 4 filter modules
  - Select Sinc3 filter
  - OSR = 256
  - All the four filters are synchronized using MFE (main filter enable bit)
  - The filter output is represented in 16-bit format
  - To convert a 25-bit data filter to a 16-bit format, users need to shift the Sinc3 filter with OSR=256 to the right by 10 bits
- Interrupt module settings for SDF filter:

- Disable all the four high threshold comparator interrupts
- Disable all the four low threshold comparator interrupts
- Disable all the four modulator fault interrupts
- When there is new filter data available, all the four filters will generate interrupts

External connection:

- Connect Sigma - Delta stream to (SD-D1, SD-C1 to SD-D4, SD-C4) on GPIO24-GPIO31

Monitoring variables:

- filter1Result - Output of Filter 1
- filter2Result - Output of Filter 2
- filter3Result - Output of Filter 3
- filter4Result - Output of Filter 4

2.21.3 SDF PWM synchronization

File: sdf_ex4_pwm_sync_cpuread.c

In this example, the SDF filter data is read by CPU in the SDF ISR routine. The SDF configuration is as follows:

- The SDF used in this example is SDF1
- Selected input control mode: MODE0
- Comparator settings:
  - Select Sinc3 filter
  - OSR = 32
  - HLT = 0x7FFF (high threshold setting)
  - LLT = 0x0000 (low threshold setting)
- Data filter settings:
  - Enable all the 4 filter modules
  - Select Sinc3 filter
  - OSR = 256
  - All the four filters are synchronized using PWM (main filter enable bit)
  - The filter output is represented in 16-bit format

- To convert a 25-bit data filter to a 16-bit format, users need to shift the Sinc3 filter with OSR=256 to the right by 10 bits

- Interrupt module settings for SDF filter:
  - Disable all the four high threshold comparator interrupts
  - Disable all the four low threshold comparator interrupts
  - Disable all the four modulator fault interrupts
  - When there is new filter data available, all the four filters will generate interrupts

External connection:

- Connect Sigma - Delta stream to (SD-D1, SD-C1 to SD-D4, SD-C4) on GPIO24-GPIO31

Monitoring variables:

- filter1Result - Output of Filter 1
- filter2Result - Output of Filter 2
- filter3Result - Output of Filter 3
- filter4Result - Output of Filter 4

2.21.4 Use of SDF1 filter FIFO

File: sdf_ex5_filter_sync_fifo_cpuread.c

This example configures the SDF1 filter to demonstrate data reading through CPU in both FIFO and non-FIFO modes. Configure the data filter to mode 0, and select the SINC3 filter with an OSR of 256. Configure the filter output to 16-bit format and use 10-bit data shift.

This example demonstrates the use of FIFO (if enabled). The FIFO length is set to 16, and when FIFO is full, the data ready interrupt will be triggered. In this example, the SDF filter data is read by CPU in the SDF data ready ISR routine.

External connection:

- Connect Sigma - Delta stream to (SD-D1, SD-C1 to SD-D4, SD-C4) on GPIO24-GPIO31

Monitoring variables:

- filter1Result - Output of Filter 1

2.22 SPI Routine Description

2.22.1 SPI loopback test

File: spi_ex1_loopback.c

This program uses the internal loopback test mode of the SPI module. This is a very basic loopback that does not use FIFO or interrupts. Transmit a string of data and compare it with the received data. Pin multiplexing and SPI module can be reconfigured according to actual needs.

The transmitted data format is as follows:

0000 0001 0002 0003 0004 0005 0006 0007 ... FFFE FFFF 0000

This mode will repeat infinitely.

External connection:

- None

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received

2.22.2 SPI loopback test with FIFO interrupt

File: spi_ex2_loopback_fifo_interrupts.c

This program uses the internal loopback test mode of the SPI module, and FIFO and interrupts of SPI are used.

Transmit a string of data and compare it with the received data. The transmitted data format is as follows:

0000 0001

0001 0002

0002 0003

...

FFFE FFFF

FFFF 0000

And so on...

This mode will repeat infinitely.

External connection:

- None

Monitoring variables:

- sData - Data to be transmitted

- rData - Data received
- rDataPoint - Used to track the last position in the received stream for error checking

2.22.3 SPI external loopback test without FIFO interrupt

File: spi_ex3_external_loopback.c

This program uses an external loopback between two SPI modules. In this example, FIFO and interrupts of SPI are not used. SPIA is configured as the slave device, while SPIB is configured as the master controller. This example demonstrates full-duplex communication, so that the controller and the slave device can transmit and receive data simultaneously.

External connection:

- GPIO7 and GPIO16 - SPISIMO
- GPIO25 and GPIO17 - SPISOMI
- GPIO26 and GPIO56 - SPICLK
- GPIO27 and GPIO57 - SPISTE

Monitoring variables:

- TxData_SPIA - Data transmitted from SPIA (slave device)
- TxData_SPIB - Data transmitted from SPIB (master controller)
- RxData_SPIA - Data received from SPIA (slave device)
- RxData_SPIB - Data received from SPIB (master controller)

2.22.4 SPI external loopback test with FIFO interrupt

File: spi_ex4_external_loopback_fifo_interrupts.c

This program uses the external loopback between two SPI modules, and FIFO and interrupts of SPI are used. SPIA is configured as the slave device, and the slave SPIB (configured as the master controller) receives data.

Transmit a string of data and compare it with the received data. The transmitted data format is as follows:

0000 0001

0001 0002

0002 0003

...

FFFE FFFF

FFFF 0000

And so on...

This mode will repeat infinitely.

External connection:

- GPIO7 and GPIO16 - SPISIMO
- GPIO25 and GPIO17 - SPISOMI
- GPIO26 and GPIO3——SPICLK
- GPIO27 and GPIO11——SPISTE

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received
- rDataPoint - Used to track the last position in the received stream for error checking

2.22.5 SPI loopback test with DMA

File: spi_ex5_loopback_dma.c

This program uses the internal loopback test mode of the SPI module, and the DMA interrupts and FIFO of SPI are used. When there is enough space in the transmit FIFO of SPI (indicated by the FIFO level interrupt signal), DMA will transfer the data from the global variable sData to FIFO, and the data will be transmitted to the receive FIFO through internal loopback.

When there is enough data in the receive FIFO (indicated by the FIFO level interrupt signal), DMA will transfer the data from FIFO to the global variable rData.

When all data has been put into rData, the data validity will be checked in an ISR of the DMA channel.

External connection:

- None

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received

2.22.6 SPI EEPROM

File: spi_ex6_eeprom.c

This program writes 8-byte data into EEPROM and reads it back. The device communicates with EEPROM through SPI and specific opcodes. This example is applicable to serial EEPROM AT25128/256 of SPI.

External connection:

Connect external SPI EEPROM

- Connect GPIO16 (PICO) to the external EEPROM SI pin
- Connect GPIO17 (POCI) to the external EEPROM SO pin
- Connect GPIO56 (CLK) to the external EEPROM SCK pin
- Connect GPIO11 (CS) to the external EEPROM CS pin
- Connect VCC and GND pins of the external EEPROM

Monitoring variables:

- writeBuffer – Data written to external EEPROM
- readBuffer – Data read from EEPROM
- error - Error count

2.22.7 SPI DMA EEPROM

File: spi_ex7_eeprom_dma.c

This program writes 8-byte data into EEPROM and reads it back. The device communicates with EEPROM using DMA and specific opcodes through SPI. This example is applicable to serial EEPROM AT25128/256 of SPI.

External connection:

Connect external SPI EEPROM

- Connect GPIO16 (PICO) to the external EEPROM SI pin
- Connect GPIO17 (POCI) to the external EEPROM SO pin
- Connect GPIO56 (CLK) to the external EEPROM SCK pin
- Connect GPIO11 (CS) to the external EEPROM CS pin
- Connect VCC and GND pins of the external EEPROM

Monitoring variables:

- writeBuffer – Data written to external EEPROM
- SPI_DMA_Handle.RXdata - The data read back from EEPROM when the number of received bytes is less than 4
- SPI_DMA_Handle.pSPIRXDMA->pbuffer - Start address for receiving data from EEPROM
- error - Error count

2.23 SYSCTL Routine Description

2.23.1 Missing clock detection

File: sysctl_ex1_missing_clock_detection.c

This program demonstrates the missing clock detection (MCD) function and its processing method. When MCD is simulated by disconnecting OSCCLK from the MCD module, an NMI interrupt will be generated. This NMI interrupt indicates that MCD is generated due to clock fault and is processed in ISR.

Before MCD occurs, the clock frequency is set to 250MHz according to the device initialization. After MCD occurs, the frequency will be changed to 10MHz or INTOSC1.

This example also demonstrates how to relock the PLL after missing clock detection, first switching the clock source to INTOSC1, resetting the missing clock detection circuit, and then relocking the PLL. After relocking, the clock frequency will be 250MHz, but INTOSC1 will be used as the clock source.

External connection:

- None

Monitoring variables:

- fail - indicates that the missing clock has not been detected or processed correctly.
- mcd_clkfail_isr - indicates that an NMI interrupt was triggered due to the missing clock fault and ISR was called to process it.
- mcd_detect - indicates that the missing clock has been detected.
- result - indicates the successfully processed status of missing clock detection.

2.23.2 XCLKOUT configuration

File: sysctl_ex2_xclkout_config.c

This program demonstrates how to configure the XCLKOUT pin and observe the internal clock through external pins for the purpose of debugging and testing.

In this example, INTOSC1 is used as the XCLKOUT clock source, and the frequency divider is configured as 8.

The expected frequency of XCLKOUT is calculated as follows:

XCLKOUT frequency = INTOSC1 frequency / 8 =10MHz / 8 = 1.25MHz

External connection:

- Observe XCLKOUT on GPIO16 using an oscilloscope.

Monitoring variables:

- None

2.24 Timer Routine Description

2.24.1 CPU timer example

File: timer_ex1_tmrs.c（timer_ex1_cputimers.c）

This program example configures CPU timers 0, 1, and 2, and increases a counter each time the timer triggers an interrupt.

External connection:

- None

Monitoring variables:

- cpuTimer0IntCount - Interrupt count of CPU timer 0
- cpuTimer1IntCount - Interrupt count of CPU timer 1
- cpuTimer2IntCount - Interrupt count of CPU timer 2

2.25 UART Routine Description

2.25.1 Example of adjusting baud rate through UART

File: baud_tune_via_uart.c

This example demonstrates how to adjust the UART baud rate of the G32R501 device based on the UART input of another device. As there is no clock signal in UART, reliable communication requires a roughly matched baud rate. This example solves the problem that the clock mismatching between devices exceeds an acceptable range, requiring baud rate compensation between devices. Since reliable communication only requires matching effective baud rate, either device can be adjusted (the devices with inaccurate clock source do not need to be adjusted, and as long as one device is adjusted to the other, correct communication can be established).

To adjust the baud rate of this device, the UART data (required baud rate) must be transmitted to this device. The entered UART baud rate must be within the +/—— MARGINPERCENT range of the selected TARGETBAUD below. These two variables are defined below and shall be selected according to application requirements. In noisy environments, a higher MARGINPERCENT will allow more data to be considered “correct”, but may reduce the accuracy. TARGETBAUD is the expected baud rate, but due to clock differences, it needs to be adjusted to enhance the communication robustness with the other device.

For Eval and custom devices, if GPIO9 and GPIO8 cannot be used, different GPIO need to be configured for UART_RX and UART_TX pins. Open the UART peripheral and select the available GPIO on the given board. Update the GPIO_UARTRX_NUMBER below to match the RX selection. Please refer to the Eval User Guide to obtain a list of available GPIO.

Note: Lower baud rates have more granularities in register options, so it is more effective to adjust at these speeds.

External connection:

- UART_RX/eCAP1 is located in GPIO9 and is connected to the incoming UART communication
- UART_TX is located in GPIO8 and is used for external observation

Monitoring variables:

- avgBaud - Adjusted detection and set baud rate

2.25.2 UART FIFO loopback test

File: uart_ex1_loopback.c

This program uses the internal loopback test mode of the peripheral. No other hardware configuration is required except for the boot mode pin configuration. This test uses the loopback test mode of UART module to transmit characters from 0x00 to 0xFF. The test will transmit a character and then check if the receive buffer matches correctly.

Monitoring variables:

- loopCount - Number of characters transmitted
- errorCount - Number of errors detected
- sendChar - Characters transmitted
- receivedChar - Characters received

2.25.3 UART interrupt loopback

File: uart_ex2_interrupts.c

This test passes the UART-Port A by interrupting receiving and loopback data. Terminals such

as 'Putty' can be used to view the data from UART and transmit information to UART. The characters received by the UART port will be transmitted back to the host.

Run the application and use a terminal to open a COM port with the following settings:

- Find the correct COM port
- Bit rate=9600
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None

The program will print out greetings and then ask you to enter a character, which will be looped back to the terminal.

Monitoring variables:

- counter - Number of characters transmitted

External connection:

Connect the UART - Port A to PC through the transceiver and the cable.

- GPIO28 is UART_A - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART_A - TXD (Pin2 connected to serial DB9 cable, PC - RX)

2.25.4 UART interrupt loopback with FIFO

File: uart_ex3_interrupts_fifo.c

This test passes the UART-Port A by interrupting two characters, i.e. receiving and loopback, once. When the FIFO status level is two or more, trigger the Rx interrupt. Once the two characters are in the RXFIFO, the UART Rx ISR will be triggered and two characters will be read from the FIFO and written to the transmit buffer. Then UART Tx ISR will be triggered again to request more data from the terminal.

Run the application and use a terminal to open a COM port with the following settings:

- Find the correct COM port
- Bit rate=9600
- Data bit=8
- Check bit=None
- Stop bit=1

- Hardware control=None

The program will print out greetings and then ask you to enter two characters, which will be looped back to the terminal.

Monitoring variables:

- counter - Logarithm of characters transmitted

External connection:

Connect the UART - Port A to PC through the transceiver and the cable.

- GPIO28 is UART_A - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART_A - TXD (Pin2 connected to serial DB9 cable, PC - RX)

2.25.5 UART loopback test

File: uart_ex4_echoback.c

This test receives and loops back data through UART - Port A. Terminals such as 'Putty' can be used to view the data from UART and transmit information to UART. The characters received by the UART port will be transmitted back to the host.

Run the application and use a terminal to open a COM port with the following settings:

- Find the correct COM port
- Bit rate=9600
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None
- The program will print out greetings and then ask you to enter a character, which will be looped back to the terminal.

Monitoring variables:

- loopCounter - Number of characters transmitted

External connection:

Connect the UART - Port A to PC through the transceiver and the cable.

- GPIO28 is UART_A - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART_A - TXD (Pin2 connected to serial DB9 cable, PC - RX)

2.26 Watchdog Routine Description

2.26.1 Watchdog routine

File: watchdog_ex1_service.c

This program example demonstrates how to serve a watchdog or use a watchdog to generate a wake-up interrupt. By default, this example will generate a wake-up interrupt. To serve the watchdog without generating an interrupt, please uncomment the SysCtl_serviceWatchdog() line in the main loop.

External connection:

- None

Monitoring variables:

- WakeCount - The number of times of entering the watchdog ISR
- loopCount - The number of loops not executed in ISR

2.27 Zidian Routine Description

2.27.1 Zidian mathematics example

File: zidian_ex1_math.c

This program example demonstrates how to use zidian_math.h. Only the sqrtf(), sinf(), cosf(), atanf() and atan2f() in math.h can be accelerated through Zidian. Users can observe the acceleration effect through traceTime.

Note: Please avoid using compatibility functions directly, e.g. __sin(), __cos() and __atan().

External connection:

- None

Monitoring variables:

- traceResult - Calculation result
- traceTime - The time spent in calculating

3 Device Support Routine

Note: These routines are located at the following positions of G32R5xx_SDK:

G32R5xx_SDK_VERSION：/device_support/DEVICE_GPN/examples/CORE_IF_MULTICORE/

3.1 ADC Routine Description

3.1.1 ADC PWM trigger

File: adc_ex1_soc_pwm.c

This program example sets PWM1 to periodically trigger the conversion on ADCA.

External connection:

- A1 shall be connected to a signal to be converted

Monitoring variables:

- adcAResults - A series of analog-to-digital conversion samples from Pin A1. The time interval between samples is determined by the period based on the PWM timer.

3.2 CAP Routine Description

3.2.1 CAP APWM routine

File: cap_ex1_apwm.c

This program example sets the CAP module in APWM mode. The PWM waveform will appear on GPIO5. The shadow register is used to load the next cycle/comparison value, with the PWM frequency configured to vary between 5Hz and 10Hz.

External connection:

- None

Monitoring variables:

- None

3.3 DAC Routine Description

3.3.1 Buffer DAC enable routine

File: dac_ex1_enable.c

This example generates voltage on the buffered DAC output DACOUTA/ ADCINA0 and uses the default DAC reference setting VDAC.

External connection

- When the DAC reference setting is VDAC, an external reference voltage must be applied on the VDAC pin. This can be achieved by connecting a jumper wire from 3.3V to ADCINB3.

Monitoring variables

- None

3.4 DMA Routine Description

3.4.1 DMA SRAM transmission

File: dma_ex1_sram_transfer.c

This example uses a DMA channel to transfer data from a buffer in SRAM1 to a buffer in SRAM1. The example repeatedly sets the PERINTFRC bit of the DMA channel until the transfer of 16 bursts (each burst is eight 16-bit words) is completed. When the entire transfer is completed, a DMA interrupt will be triggered.

Monitoring variables

- sdata - Data transmitted
- rdata - Data received

3.5 Template Routine Description

3.5.1 Bit field library standard template

File: empty_bitfield_main.c

This example is an empty item set for bit field development.

3.5.2 Empty standard template for bit field library and driver library

File: empty_bitfield_driverlib_main.c

This example is an empty item set for bit field and driver library development.

3.6 GPIO Routine Description

3.6.1 GPIO Settings

File: gpio_ex1_setup.c

Configure G32R5xx GPIO with two different configurations. This code explains how to set GPIO in detail. In practical applications, the code lines can be combined to increase the code size and efficiency. This example only sets GPIO. No operation is performed on the pins after the setting is completed.

External connection

- None

Monitoring variables

- None

3.7 HRPWM Routine Description

3.7.1 HRPWM SFO V1 high-resolution cycle (up counting)

File: hrpwm_ex1_duty_sfo_v1.c

This example modifies the MEP control register to display the edge displacement of the high-resolution cycle in PWM count-up mode, which is due to the HRPWM control extension of the corresponding PWM module.

This example calls the following MEP scale factor optimizer (SFO) software library V1 function of Geehy:

int SFO();

- Dynamically update MEP_ScaleFactor when HRPWM is used
- Update the HRMSTEP register (only present in the Pwm1Regs register space) with the MEP_ScaleFactor value
- Return 2, indicating an error: MEP_ScaleFactor is greater than the maximum value 255 (under this condition, automatic conversion may not work properly)
- Return 1, indicating that the specified channel is completed
- Return 0, indicating that the specified channel is not completed

This example aims to explain the capability of HRPWM. The code can be optimized to increase the code efficiency.

Run this example:

- Run this example at the maximum SYSCLKOUT
- Activate real-time mode
- Run code

External connection

- Monitor the PWM1 A/B pins on an oscilloscope.

Monitoring variables

- Status - Running status of the example
- UpdateFine - Set to 1 to use the HRPWM function and observe the fine MEP step size

(default). Set to 0 to disable the HRPWM function and observe the coarse SYSCLKOUT cycle step size.

3.7.2 HRPWM SFO V1 high-resolution cycle (up/down counting)

File: hrpwm_ex2_prdupdown_sfo_v1.c

This example modifies the MEP control register to display the edge displacement of the high-resolution cycle in PWM count-up/down mode, which is due to the HRPWM control extension of the corresponding PWM module.

This example calls the following MEP scale factor optimizer (SFO) software library V1 function of Geehy:

int SFO();

- Dynamically update MEP_ScaleFactor when HRPWM is used
- Update the HRMSTEP register (only present in the Pwm1Regs register space) with the MEP_ScaleFactor value
- Return 2, indicating an error: MEP_ScaleFactor is greater than the maximum value 255 (under this condition, automatic conversion may not work properly)
- Return 1, indicating that the specified channel is completed
- Return 0, indicating that the specified channel is not completed

This example aims to explain the capability of HRPWM. The code can be optimized to increase the code efficiency.

Run this example:

- Run this example at the maximum SYSCLKOUT
- Activate real-time mode
- Run code

External connection

- Monitor the PWM1 A/B pins on an oscilloscope.

Monitoring variables

- UpdateFine - Set to 1 to use the HRPWM function and observe the fine MEP step size (default). Set to 0 to disable the HRPWM function and observe the SYSCLKOUT cycle step size.

3.8 I2C Routine Description

3.8.1 I2C master device

File: i2c_ex1_master.c

This program demonstrates how to use I2CA in the master configuration. This example uses polling method, without using interrupts or FIFO.

Two control cards are required:

- One is configured as the master device, and the other is configured as the slave device.
- The master device will run the binary file generated by "i2c_ex1_master.uvprojx".
- The slave device will run the binary file generated by "i2c_ex1_slave.uvprojx".

External connection:

The external connection on the control card shall be as follows:

Table 3 External Connection Diagram on Control Card

Signal	I2CA on Board1	I2CA on Board2
SCL	GPIO_PIN_SCLA	GPIO_PIN_SCLA
SDA	GPIO_PIN_SDAA	GPIO_PIN_SDAA
GND	GND	GND

- Example 1: Master transmitter and slave receiver
- Example 2: Master receiver and slave transmitter
- Example 3: Master transmitter and slave receiver, then master receiver and slave transmitter
- Example 4: Master receiver and slave transmitter, then master transmitter and slave receiver

Monitoring variables:

- I2CA_TXdata
- I2CA_RXdata

3.8.2 I2C slave device

File: i2c_ex1_slave.c

This program demonstrates how to use I2CA in the slave configuration. This example uses I2C interrupts without using FIFO.

Two control cards are required:

- One is configured as the master device, and the other is configured as the slave device.
- The master device will run the binary file generated by "i2c_ex1_master.uvprojx".
- The slave device will run the binary file generated by "i2c_ex1_slave.uvprojx".

External connection:

The external connection on the control card shall be as follows:

Table 4 External Connection Diagram on Control Card

Signal	I2CA on Board1	I2CA on Board2
SCL	GPIO_PIN_SCLA	GPIO_PIN_SCLA
SDA	GPIO_PIN_SDAA	GPIO_PIN_SDAA
GND	GND	GND

- Example 1: Master transmitter and slave receiver
- Example 2: Master receiver and slave transmitter
- Example 3: Master transmitter and slave receiver, then master receiver and slave transmitter
- Example 4: Master receiver and slave transmitter, then master transmitter and slave receiver

Monitoring variables:

- I2CA_TXdata
- I2CA_RXdata

3.9 LED Routine Description

3.9.1 LED flashing

File: led_ex1_blinky.c

This example demonstrates how to make the LED flash.

3.10 SPI Routine Description

3.10.1 SPI loopback test

File: spi_ex1_loopback.c

This program uses the internal loopback test mode of the SPI module. This is a very basic loopback that does not use FIFO or interrupts. Transmit a string of data and compare it with the

received data. Pin multiplexing and SPI module can be reconfigured according to actual needs.

The transmitted data format is as follows:

0000 0001 0002 0003 0004 0005 0006 0007 ... FFFE FFFF 0000

This mode will repeat infinitely.

External connection:

- None

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received

3.10.2 SPI loopback test with DMA

File: spi_ex2_dma_loopback.c

This program uses the internal loopback test mode of the SPI module, and the DMA interrupts and FIFO of SPI are used. When there is enough space in the transmit FIFO of SPI (indicated by the FIFO level interrupt signal), DMA will transfer the data from the global variable sData to FIFO, and the data will be transmitted to the receive FIFO through internal loopback.

When there is enough data in the receive FIFO (indicated by the FIFO level interrupt signal), DMA will transfer the data from FIFO to the global variable rData.

When all data has been put into rData, the data validity will be checked in an ISR of the DMA channel.

External connection:

- None

Monitoring variables:

- sData - Data to be transmitted
- rData - Data received

3.11 Timer Routine Description

3.11.1 CPU timer example

File: timer_ex1_tmrs.c（timer_ex1_cputimers.c）

This program example configures CPU timers 0, 1, and 2, and increases a counter each time the timer triggers an interrupt.

External connection:

- None

Monitoring variables:

- cpuTimer0IntCount - Interrupt count of CPU timer 0
- cpuTimer1IntCount - Interrupt count of CPU timer 1
- cpuTimer2IntCount - Interrupt count of CPU timer 2

3.12 UART Routine Description

3.12.1 UART loopback test

File: uart_ex4_echoback.c

This test receives and loops back data through UART - Port A. Terminals such as 'Putty' can be used to view the data from UART and transmit information to UART. The characters received by the UART port will be transmitted back to the host.

Run the application and use a terminal to open a COM port with the following settings:

- Find the correct COM port
- Bit rate=9600
- Data bit=8
- Check bit=None
- Stop bit=1
- Hardware control=None
- The program will print out greetings and then ask you to enter a character, which will be looped back to the terminal.

Monitoring variables:

- loopCounter - Number of characters transmitted

External connection:

Connect the UART - Port A to PC through the transceiver and the cable.

- GPIO28 is UART_A - RXD (Pin3 connected to serial DB9 cable, PC - TX)
- GPIO29 is UART_A - TXD (Pin2 connected to serial DB9 cable, PC - RX)

4 Revision

Table 5 Document Revision History

Date	Version	Change History
January, 2025	1.0	New

Statement

This document is formulated and published by Geehy Semiconductor Co., Ltd. (hereinafter referred to as “Geehy”). The contents in this document are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to make corrections and modifications to this document at any time. Read this document carefully before using Geehy products. Once you use the Geehy product, it means that you (hereinafter referred to as the “users”) have known and accepted all the contents of this document. Users shall use the Geehy product in accordance with relevant laws and regulations and the requirements of this document.

1. Ownership

This document can only be used in connection with the corresponding chip products or software products provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this document for any reason or in any form.

The “极海” or “Geehy” words or graphics with “®” or “™” in this document are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No Intellectual Property License

Geehy owns all rights, ownership and intellectual property rights involved in this document.

Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale or distribution of Geehy products or this document.

If any third party’s products, services or intellectual property are involved in this document, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property. Any information regarding the application of the product, Geehy hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party, unless otherwise agreed in sales order or sales contract.

3. Version Update

Users can obtain the latest document of the corresponding models when ordering Geehy products.

If the contents in this document are inconsistent with Geehy products, the agreement in the sales order or the sales contract shall prevail.

4. Information Reliability

The relevant data in this document are obtained from batch test by Geehy Laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing environment may occur inevitably. Therefore, users should understand that Geehy does not bear any responsibility for such errors that may occur in this document. The relevant data in this document are only used to guide users as performance parameter reference and do not constitute Geehy's guarantee for any product performance.

Users shall select appropriate Geehy products according to their own needs, and effectively verify and test the applicability of Geehy products to confirm that Geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If losses are caused to users due to user's failure to fully verify and test Geehy products, Geehy will not bear any responsibility.

5. Legality

USERS SHALL ABIDE BY ALL APPLICABLE LOCAL LAWS AND REGULATIONS WHEN USING THIS DOCUMENT AND THE MATCHING GEEHY PRODUCTS. USERS SHALL UNDERSTAND THAT THE PRODUCTS MAY BE RESTRICTED BY THE EXPORT, RE-EXPORT OR OTHER LAWS OF THE COUNTRIES OF THE PRODUCTS SUPPLIERS, GEEHY, GEEHY DISTRIBUTORS AND USERS. USERS (ON BEHALF OR ITSELF, SUBSIDIARIES AND AFFILIATED ENTERPRISES) SHALL AGREE AND PROMISE TO ABIDE BY ALL APPLICABLE LAWS AND REGULATIONS ON THE EXPORT AND RE-EXPORT OF GEEHY PRODUCTS AND/OR TECHNOLOGIES AND DIRECT PRODUCTS.

6. Disclaimer of Warranty

THIS DOCUMENT IS PROVIDED BY GEEHY "AS IS" AND THERE IS NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, TO THE EXTENT PERMITTED BY APPLICABLE LAW.

GEEHY'S PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED FOR USE AS CRITICAL COMPONENTS IN MILITARY, LIFE-SUPPORT, POLLUTION CONTROL, OR HAZARDOUS SUBSTANCES MANAGEMENT SYSTEMS, NOR WHERE FAILURE COULD RESULT IN INJURY, DEATH, PROPERTY OR ENVIRONMENTAL DAMAGE.

IF THE PRODUCT IS NOT LABELED AS "AUTOMOTIVE GRADE," IT SHOULD NOT BE CONSIDERED SUITABLE FOR AUTOMOTIVE APPLICATIONS. GEEHY ASSUMES NO LIABILITY FOR THE USE BEYOND ITS SPECIFICATIONS OR GUIDELINES.

THE USER SHOULD ENSURE THAT THE APPLICATION OF THE PRODUCTS COMPLIES WITH ALL RELEVANT STANDARDS, INCLUDING BUT NOT LIMITED TO SAFETY, INFORMATION SECURITY, AND ENVIRONMENTAL REQUIREMENTS. THE USER ASSUMES FULL RESPONSIBILITY FOR THE SELECTION AND USE OF GEEHY PRODUCTS. GEEHY WILL BEAR NO RESPONSIBILITY FOR ANY DISPUTES ARISING FROM THE SUBSEQUENT DESIGN OR USE BY USERS.

7. Limitation of Liability

IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL GEEHY OR ANY OTHER PARTY WHO PROVIDES THE DOCUMENT AND PRODUCTS "AS IS", BE LIABLE FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, DIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE DOCUMENT AND PRODUCTS (INCLUDING BUT NOT LIMITED TO LOSSES OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY USERS OR THIRD PARTIES). THIS COVERS POTENTIAL DAMAGES TO PERSONAL SAFETY, PROPERTY, OR THE ENVIRONMENT, FOR WHICH GEEHY WILL NOT BE RESPONSIBLE.

8. Scope of Application

The information in this document replaces the information provided in all previous versions of the document.