Application Note

AN1182

G32A14xx

Peripheral CFGIO Application Note

Version: V 1.0

1 Introduction

This application note describes how the CFGIO general-purpose peripheral module in the G32A14x5 series emulates serial and parallel communication protocols, including UART, I2C master, SPI master/slave, I2S master/slave with PCM digital audio support, LIN, and others. A key advantage of CFGIO is that it allows users to implement custom peripherals directly inside the MCU.

Content

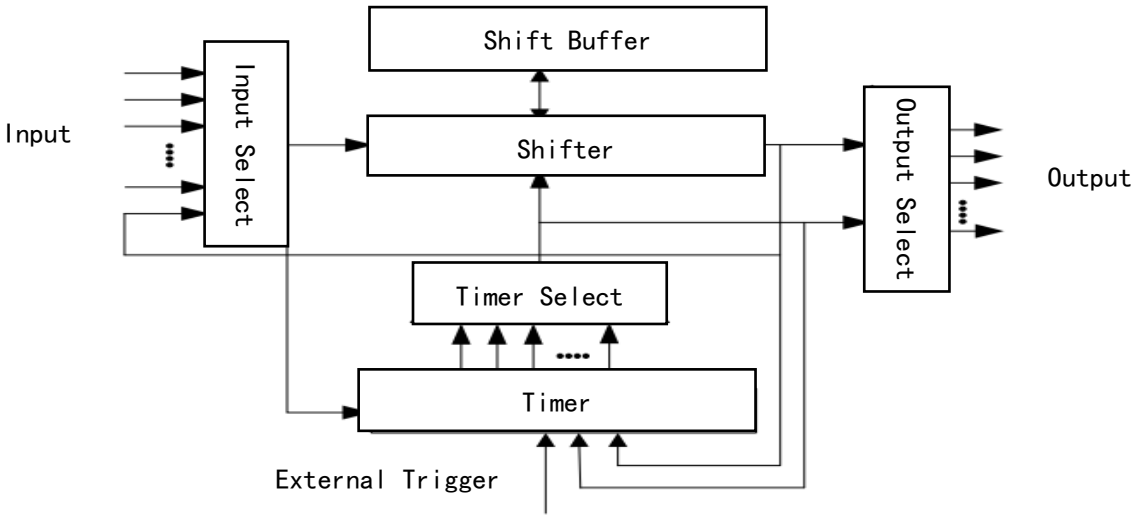
2 Introduction to the General-Purpose Peripheral Module CFGIO

CFGIO provides the following main features:

- Shifter cascading to support larger transfer sizes
- Automatic start-bit and stop-bit generation
- A 32-bit shift-register array with transmit, receive, and data-match modes
- Double-buffered shifter operation for continuous data transmission
- Transmit and receive operation by interrupt, DMA, or polling
- Programmable baud rate independent of the bus clock frequency, with asynchronous operation supported in Stop mode
- A highly configurable 16-bit timer that supports various internal and external trigger, reset, enable, and disable conditions

The CFGIO block diagram is shown below:

Figure 1 CFGIO Block Diagram



Resource usage of CFGIO when emulating each protocol:

Emulated Serial Communication Protocol	Resources Used	CFGIO Resource Occupancy
LPUART	Transmit and receive are independent sections: Transmit: 1 timer, 1 shifter, 1 pin; Receive: 1 timer, 1 shifter, 1 pin	50%
LPI2C master	2 timers, 2 shifters, 2 pins	50%

Emulated Serial Communication Protocol	Resources Used	CFGIO Resource Occupancy
SPI master	2 timers, 2 shifters, 4 pins	50%
SPI slave	1 timer, 2 shifters, 4 pins	50%
I2S master	2 timers, 2 shifters, 4 pins	50%
LIN master	2 timers, 2 shifters, 4 pins	50%
LIN slave	2 timers, 2 shifters, 4 pins	50%

3 Emulating LPUART with CFGIO

3.1 Implementation Principle

3.1.1 LPUART Transmit:

One timer, one shifter, and one pin can be used to support LPUART transmission. Start-bit and stop-bit insertion is handled automatically, and DMA can be used for multiple transfers. The timer status flag indicates when a stop bit occurs.

Break and idle characters require software handling. Before sending a break or idle character, configure SSTAFSEL and SSTOPFSEL to output the required state. The transmitted data must be 0xFF or 0x00. To support a second stop bit, software must insert the stop bit into the data stream. When a byte is written to CFGIO_SBUF, the register address-mark bit and other stop bits remain unchanged. CFGIO does not support automatic parity-bit insertion.

3.1.2 LPUART Receive:

One timer, one shifter, and one pin can be used to support LPUART reception. Start-bit and stop-bit checking is handled automatically, and DMA can be used for multiple transfers. The timer status flag indicates when a stop bit is received. CFGIO does not support triple voting for received data; data is sampled once at the middle of each bit. Another timer can be used for input glitch filtering, and an additional timer can detect an idle line with a programmable length. A break character sets the error flag, and the shift-buffer register returns 0x00. CFGIO does not support automatic parity-bit checking.

3.2 Driver Component Configuration and API Function Descriptions

User structure:

```
typedef struct
{
    CFGIO_UART_DRIVER_DIR_T direction:      /* Driver direction: receive (Rx) or transmit (Tx) */
    CFGIO_DRIVER_TYPE_T transferType:       /* Data transfer type: interrupt/DMA/polling */
    uint32_t baudrate:                      /* Baud rate */
    uint8_t numBits:                        /* Number of bits per word */
    uint8_t cfgioPin:                       /* CFGIO pin used as the receive (Rx) or transmit (Tx) pin */
    uint8_t dmaChannel:                     /* DMA channel number, used only in DMA mode */
```

```
    UART_CALLBACK_T callback:                /* User callback function */
    void *callbackParam:                     /* Parameter for the callback function */
} CFGIO_UART_USER_CONFIG_T;
```

Initialization function:

```
STATUS_T CFGIO_UART_Init(                                    /* Returns error status */
    uint32_t instance,                                       /* CFGIO peripheral instance number */
    const CFGIO_UART_USER_CONFIG_T *configPtr,               /* Pointer to the CFGIO_UART user configuration structure */
    CFGIO_UART_STATE_T *states);                             /* Pointer to the CFGIO_UART driver state structure */
```

Set baud rate and bit width:

```
STATUS_T CFGIO_UART_SetBaudrate(          /* Returns error status */
    CFGIO_UART_STATE_T *states,           /* Pointer to the CFGIO_UART driver state structure */
    uint32_t baudrate,                    /* Desired baud rate */
    uint8_t numBits):                     /* Number of bits per word */
```

Blocking receive:

```
STATUS_T CFGIO_UART_RxDataBlocking(       /* Returns error status */
    CFGIO_UART_STATE_T *states,           /* Pointer to the CFGIO_UART driver state structure */
    uint8_t *rxBuffer,                    /* Pointer to the data buffer */
    uint32_t rxLength,                    /* Length of data to receive, in bytes */
    uint32_t timeout);                    /* Timeout, in milliseconds */
```

Blocking transmit:

```
STATUS_T CFGIO_UART_TxDataBlocking(       /* Returns error status */
    CFGIO_UART_STATE_T *states,           /* Pointer to the CFGIO_UART driver state
```

```
structure */
    const uint8_t *txBuffer,                    /* Pointer to the data buffer */
    uint32_t txLength,                          /* Length of data to transmit, in bytes */
    uint32_t timeout);                          /* Timeout, in milliseconds */
```

3.3 Test Case

3.3.1 Case Description

- This example shows how to use the CFGIO_UART driver.
- It uses CFGIO_D2 and CFGIO_D3 as a UART port configured for 115200 baud, 8 data bits, no parity, and 1 stop bit.
- After startup, the board prints a welcome message and accepts user commands to turn on LEDs.
- Entering commands such as "red", "green", or "blue" turns on the corresponding LED.

3.3.2 Hardware Preparation

- G32A1445 EVAL board × 1
- J-Link programmer × 1, including Type-B to Type-A USB cable and FC-20P dual-head cable/2.54mm pitch
- Mini USB data cable × 1
- Several jumper wires
- Computer × 1

3.3.3 Software Preparation

- G32A1xxx_SDK_x.x.x
- Keil MDK and G32A series support package
- SEGGER – J-Link Vx.xx
- Serial port assistant

3.3.4 Test Procedure

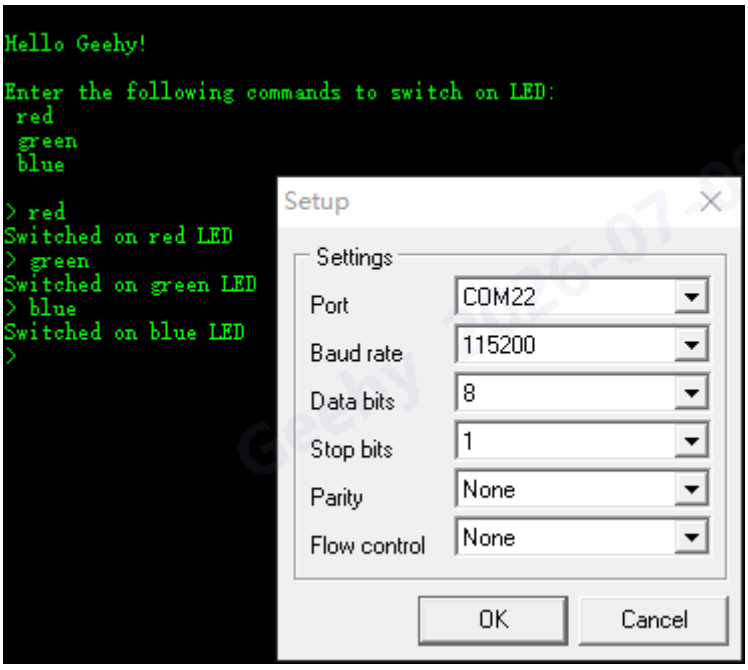
(1) Open the corresponding Keil MDK project, compile it, and download it to the evaluation board:
“G32A1xxx_SDK_x.x.x\Examples\G32A1445\CFGIO\CFGIO_UART_Echo\Project\MDK

\CFGIO_UART_Echo.uvprojx". Compile the project, download it to the G32A1445 EVAL board, and then disconnect the J-Link programmer from the board.

(2) Device connection: Remove the jumper caps at J9 and J10 on the G32A1445 EVAL board. Use one jumper wire to connect A0 to J9, and another jumper wire to connect A1 to J10. Connect one end of the Mini USB cable to the board and the other end to the computer. Open the serial port assistant on the computer, configure the serial parameters, and open the corresponding port.

(3) Press the RST reset button. The serial port assistant receives the command information sent by the board. Send the commands "redr", "greenr", and "bluer" through the serial port assistant. The corresponding LED on the board will turn on.

Figure 2 CFGIO_UART_Echo Example Serial Send/Receive



4 Emulating an LPI2C Master with CFGIO

4.1 Implementation Principle

(1) Two timers, two shifters, and two pins can be used to support LPI2C master mode. One timer controls the shifters, and the other generates the SCL output. The shifters transmit and receive each word. During reception, the transmitter must send 0xFF to keep the output in a tri-state condition.

(2) CFGIO inserts a stop bit after each word to generate or check ACK/NACK. Before enabling SCL generation, CFGIO waits for the first write to the transmit data buffer. DMA can be used for data transmission. The shifter error flag is set if a transmit underrun or receive overflow occurs.

(3) The first timer generates the bit clock for the full packet, so its compare register must be configured with the total number of clock edges in the packet. When the pin level matches the output level, the timer uses the reset counter to support clock stretching. The second timer uses the SCL input pin to control the transmit and receive shifters, adding two CFGIO clock cycles to the SDA data hold time.

(4) Each transmitted word must be processed by both the transmit and receive shifters. During reception, the transmit shifter sends 0xFF, while the receive shifter returns the data present on the SDA pin. The transmit shifter loads one word on the last falling edge of SCL. This word should be 0x00 to generate a stop condition, or 0xFF to generate a repeated start condition.

(5) During the last word sent by the master receiver, software should set the transmit SSTOPFSEL bit to 1 to generate a NACK. If a NACK is detected, the receive shift register generates an error interrupt, and software must generate a stop or repeated start condition. If a NACK is detected during master transmission, the interrupt routine should immediately write 0x00 or 0xFF to the transmit shift register. Software should then wait for the next rising edge of SCL and disable both timers. After the setup delay for a repeated start or stop condition, the transmit shifter should be disabled.

(6) Because of synchronization delay, transmit output data is valid for two CFGIO clock cycles. Therefore, the maximum baud rate is the CFGIO clock frequency divided by 6. LPI2C master data is delayed by two cycles because the clock output passes through the synchronizer before driving the transmit and receive shifters. The SCL output is synchronized to the CFGIO clock, which adds one synchronization cycle and one output-generation cycle.

4.2 Driver Component Configuration and API Function Descriptions

User structure:

```
typedef struct
{
    uint16_t slaveAddr;         /* Slave address, 7-bit */
    uint8_t sdaPin;             /* CFGIO pin used as the I2C SDA pin */
    uint8_t sclPin;             /* CFGIO pin used as the I2C SCL pin */
    uint32_t baudrate;          /* Baud rate, in Hz */
    CFGIO_DRIVER_TYPE_T transferType;     /* Data transfer type: interrupt/DMA/polling */
    uint8_t txDmaChannel;                 /* Tx DMA channel number */
    uint8_t rxDmaChannel;                 /* Rx DMA channel number */
    I2C_MASTER_CALLBACK_T callback; /* User callback function */
    void *callbackParam;                  /* Parameter for the callback function */
} CFGIO_I2C_MASTER_USER_CONFIG_T;
```

Initialization function:

```
STATUS_T CFGIO_I2C_MasterInit(                                  /* Returns error status */
    uint32_t instance,                                          /* CFGIO instance number */
    const CFGIO_I2C_MASTER_USER_CONFIG_T *configPtr, /* Pointer to the CFGIO_I2C master user configuration */
    CFGIO_I2C_MASTER_STATE_T *states);                  /* Pointer to the master context structure */
```

Set baud rate function:

```
STATUS_T CFGIO_I2C_MasterSetBaudrate(       /* Returns error status */
    CFGIO_I2C_MASTER_STATE_T *states,       /* Pointer to the master context structure */
    uint32_t baudrate);       /* Desired baud rate, in Hz */
```

Set slave device address:

```
STATUS_T CFGIO_I2C_MasterSetSlaveAddr(        /* Returns error status */
    CFGIO_I2C_MASTER_STATE_T *states,      /* Pointer to the master context structure */
    uint16_t address);      /* Slave address, 7-bit */
```

Non-blocking transmit:

```
STATUS_T CFGIO_I2C_MasterWriteNonBlocking(        /* Returns error status */
    CFGIO_I2C_MASTER_STATE_T *states,      /* Pointer to the master context structure */
    const uint8_t *txBuf,      /* Pointer to the data to transmit */
    uint32_t txLen,      /* Length of data to transmit, in bytes */
    bool sendStop);      /* Whether to generate a stop condition after transfer */
```

Non-blocking receive:

```
STATUS_T CFGIO_I2C_MasterReadNonBlocking(        /* Returns error status */
    CFGIO_I2C_MASTER_STATE_T *states,      /* Pointer to the master context structure */
    uint8_t *rxBuf,      /* Pointer to the receive buffer */
    uint32_t rxLen,      /* Length of data to receive, in bytes */
    bool sendStop);      /* Whether to generate a stop condition after reception */
```

4.3 Test Case

4.3.1 Case Description

This example shows how to use the CFGIO_I2C driver. It uses CFGIO_I2C as the I2C master and LPI2C as the I2C slave.

4.3.2 Hardware Preparation

- G32A1445 EVAL board × 1
- J-Link programmer × 1, including Type-B to Type-A USB cable and FC-20P dual-head cable/2.54mm pitch
- Mini USB data cable × 1

- Several jumper wires
- Computer × 1

4.3.3 Software Preparation

- G32A1xxx_SDK_x.x.x
- Keil MDK and G32A series support package
- SEGGER – J-Link Vx.xx
- Serial port assistant

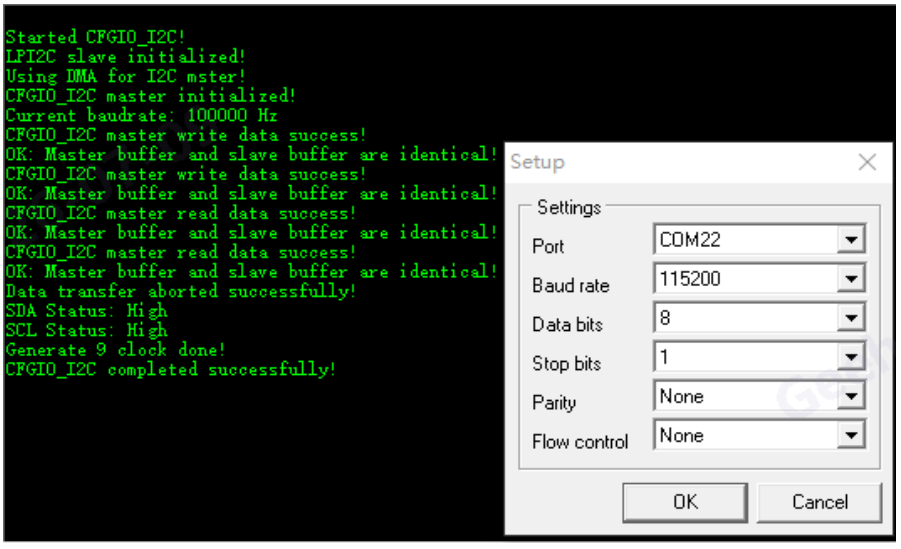
4.3.4 Test Procedure

(1) Open the corresponding Keil MDK project, compile it, and download it to the evaluation board: “G32A1xxx_SDK_x.x.x\Examples\G32A1445\CFGIO\CFGIO_I2C_Master\Project\MDK\CFGIO_I2C_Master.uvprojx”. Compile the project, download it to the G32A1445 EVAL board, and then disconnect the J-Link programmer from the board.

(2) (2) Device connection: Use one jumper wire to connect A0 to A3 on the board, and another jumper wire to connect A1 to A2. Connect one end of the Mini USB cable to the board and the other end to the computer. Open the serial port assistant on the computer, configure the serial parameters, and open the corresponding port.

(3) (3) Press the RST reset button. The serial port assistant receives the print information sent by the board. If I2C communication is successful, the green LED lights up; otherwise, the red LED lights up.

Figure 3 CFGIO_I2C_Master Example I2C Send/Receive



5 Emulating an SPI Master/Slave with CFGIO

5.1 Implementation Principle

5.1.1 Resource Configuration Required to Emulate an SPI Bus Master

Two timers, two shifters, and four pins can be used to support SPI master mode. Both CPHA = 0 and CPHA = 1 are supported, and DMA can be used for transmission. For CPHA = 1, slave select can remain active for multiple transfers, and the timer status flag can indicate the end of a transfer. Each transfer starts when the core or DMA writes to the transmit buffer. The stop bit ensures at least one clock cycle between slave-select deassertion and the next transfer. Because of synchronization delay, the serial input data setup time is 1.5 CFGIO clock cycles. Therefore, the maximum baud rate is the CFGIO clock frequency divided by 4.

5.1.2 Resource Configuration Required to Emulate an SPI Bus Slave

One timer, two shifters, and four pins can be used to support SPI slave mode. Both CPHA = 0 and CPHA = 1 are supported, and DMA can be used for transmission. For CPHA = 1, slave select can remain active for multiple transfers, and the timer status flag can indicate the end of a transfer. Transmit data must be written to the transmit buffer before the external slave-select signal becomes active; otherwise, the shifter error flag is set. Because of synchronization delay, serial output data is valid after 2.5 CFGIO clock cycles. Therefore, the maximum baud rate is the CFGIO clock frequency divided by 6.

5.2 Driver Component Configuration and API Function Descriptions

Master device user structure:

```
typedef struct
{
    uint8_t txDMAChannel;                    /* Tx DMA channel number */
    uint8_t rxDMAChannel;                    /* Rx DMA channel number */
    void *callbackParam;                     /* Callback function parameter */
    SPI_CALLBACK_T callback;                 /* User callback, non-interrupt callback */
    uint8_t ssPin;                           /* CFGIO pin configured as SS */
    uint8_t sckPin;                          /* CFGIO pin configured as SCK */
    uint8_t misoPin;                         /* CFGIO pin configured as MISO */
    uint8_t mosiPin;                         /* CFGIO pin configured as MOSI*/
    uint8_t clockPha;          /* Clock phase: 0 = sample on leading edge; 1 = sample
```

```
on trailing edge */
    uint8_t clockPol;                          /* Clock polarity: 0 = clock idle high; 1 = clock
idle low */
    CFGIO_SPI_DATA_SIZE_T transferSize;        /* Transfer size, in bytes: 1/2/4 */
    CFGIO_SPI_SEND_BIT_FIRST_T firstBitOrder; /* First-bit order: LSB/MSB */
    CFGIO_DRIVER_TYPE_T cfgspiDirType;         /* CFGIO_SPI driver type:
interrupt/polling/DMA */
    uint32_t baudrate;                         /* Baud rate */
} CFGIO_SPI_MASTER_CFG_T;
```

Slave device user structure:

```
typedef struct
{
    uint8_t txDMAChannel;                      /* Tx DMA channel number */
    uint8_t rxDMAChannel;                      /* Rx DMA channel number */
    void *callbackParam;                       /* Callback function parameter */
    SPI_CALLBACK_T callback;                   /* User callback, non-interrupt callback */
    uint8_t ssPin;                             /* CFGIO pin configured as SS */
    uint8_t sckPin;                            /* CFGIO pin configured as SCK */
    uint8_t misoPin;                           /* CFGIO pin configured as MISO */
    uint8_t mosiPin;                           /* CFGIO pin configured as MOSI */
    uint8_t clockPha;                  /* Clock phase: 0 = sample on leading edge; 1 = sample
on trailing edge */
    uint8_t clockPol;                          /* Clock polarity: 0 = clock idle high; 1 = clock
idle low */
    CFGIO_SPI_DATA_SIZE_T transferSize;        /* Transfer size, in bytes: 1/2/4 */
    CFGIO_SPI_SEND_BIT_FIRST_T firstBitOrder; /* First-bit order: LSB/MSB */
    CFGIO_DRIVER_TYPE_T cfgspiDirType;         /* CFGIO_SPI driver type:
interrupt/polling/DMA   */
```

```
} CFGIO_SPI_SLAVE_CFG_T;
```

Master initialization function:

```
STATUS_T CFGIO_SPI_MasterInit(                  /* Returns error status */
uint32_t ins,                                   /* CFGIO peripheral instance number */
const CFGIO_SPI_MASTER_CFG_T *configPtr,        /* Pointer to the CFGIO_SPI master user
configuration structure */
CFGIO_SPI_MASTER_STATE_T *master);              /* User-defined CFGIO_SPI master state */
```

Slave initialization function:

```
STATUS_T CFGIO_SPI_SlaveInit(                   /* Returns error status */
uint32_t ins,                                   /* CFGIO peripheral instance number */
const CFGIO_SPI_SLAVE_CFG_T *configPtr,         /* Pointer to the CFGIO_SPI slave user
configuration structure */
CFGIO_SPI_SLAVE_STATE_T *slave);                /* User-defined CFGIO_SPI slave state */
```

Master set baud rate function:

```
STATUS_T CFGIO_SPI_MasterSetBaudrate(           /* Returns error status */
CFGIO_SPI_MASTER_STATE_T *master,               /* User-defined CFGIO_SPI master state */
uint32_t baudrate);                             /* Current baud rate */
```

Master non-blocking transmit/receive:

```
STATUS_T CFGIO_SPI_MasterTransferNonBlocking( /* Returns error status */
    CFGIO_SPI_MASTER_STATE_T *master,          /* User-defined CFGIO_SPI master state */
    const uint8_t *txData,                     /* User transmit data buffer */
    uint8_t *rxData,                           /* User receive data buffer */
    uint32_t dataSize);                        /* Length of data to transfer, in bytes */
```

Slave non-blocking transmit/receive:

```
STATUS_T CFGIO_SPI_SlaveTransferNonBlocking(        /* Returns error status */
    CFGIO_SPI_SLAVE_STATE_T *slave,                 /* User-defined CFGIO_SPI slave state */
    const uint8_t *txData,                          /* User transmit data buffer */
    uint8_t *rxData,                                /* User receive data buffer */
    uint32_t dataSize);                             /* Length of data to transfer, in bytes*/
```

5.3 Test Case

5.3.1 Case Description

This case uses the CFGIO module to emulate SPI communication and requires two development boards. The code contains the SPI_MASTER macro. When it is set to 1, SPI works as the master. When it is set to 0, SPI works as the slave.

5.3.2 Hardware Preparation

- G32A1445 EVAL board × 1
- J-Link programmer × 1, including Type-B to Type-A USB cable and FC-20P dual-head cable/2.54mm pitch
- Mini USB data cable × 2
- Several jumper wires
- Computer × 1

5.3.3 Software Preparation

- G32A1xxx_SDK_x.x.x
- Keil MDK and G32A series support package
- SEGGER – J-Link Vx.xx
- Serial port assistant

5.3.4 Test Procedure

(1) Open the corresponding Keil MDK project, modify it, compile it, and download it to the

evaluation boards: “G32A1xxx_SDK_x.x.x\Examples\G32A1445\CFGIO\CFGIO_SPI_Dual\Project\MDK\CFGIO_SPI_Dual.uvprojx”. Set the CFGIOSPI_MASTER macro in “main.c” to 1, compile the project, and download it to one G32A1445 EVAL board as the master. Set the CFGIOSPI_MASTER macro in “main.c” to 0, compile the project, and download it to another G32A1445 EVAL board as the slave. Then disconnect the J-Link programmer from the boards.

(2) Device connection: Connect the two evaluation boards as follows:

master		slave	
PD9	<==>	PD9	mosi
PA11	<==>	PA11	miso
PA0	<==>	PA0	sck
PA1	<==>	PA1	ss

(3) Connect the two Mini USB cables to the two evaluation boards, and connect the other ends to the computer. Open the serial port assistant on the computer, configure the serial parameters, and open the corresponding ports.

(4) First press the RST reset button on the slave device, and then immediately press the RST reset button on the master device. The serial port assistant receives the print information sent by the boards. If SPI communication is successful, the green LED lights up; otherwise, the red LED lights up.

Figure 4 CFGIO_SPI_Dual.uvprojx Example SPI Master Send/Receive

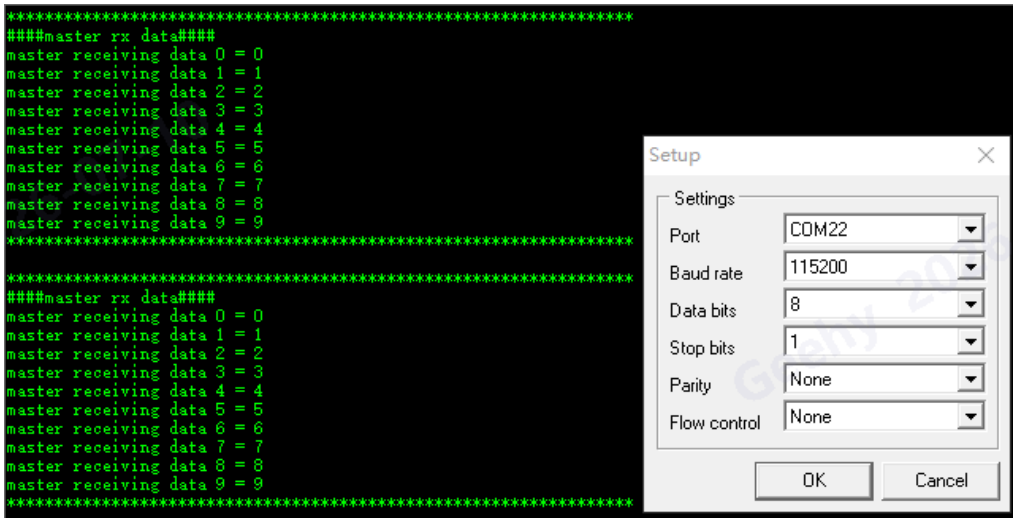
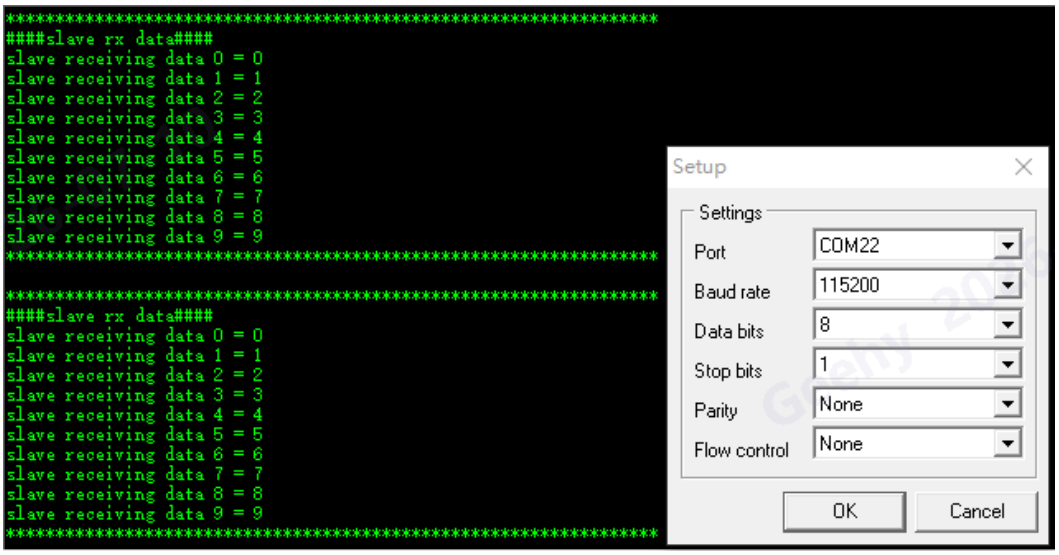


Figure 5 CFGIO_SPI_Dual.uvprojx Example SPI Slave Send/Receive



6 CFGIO_SPI_Dual.uvprojx Example SPI Slave Send/Receive

6.1 Implementation Principle

6.1.1 Resource Configuration Required to Emulate an I2S Bus Master

Two timers, two shifters, and four pins can be used to support I2S master mode. One timer generates the frame sync, and the other generates the bit clock and controls the shifters. Before enabling bit-clock and frame-sync generation, CFGIO waits for the first write to the transmit data buffer. DMA can be used for data transmission. The shifter error flag is set if a transmit underrun or receive overflow occurs. The timer uses the start bit to ensure that frame sync is generated one clock cycle before the first output data bit. The bit-clock frequency is an even integer multiple of the CFGIO clock frequency, and the initial frame sync occurs at the same time as the first bit-clock edge. Because of synchronization delay, the receiver input setup time is 1.5 CFGIO clock cycles. Therefore, the maximum baud rate is the CFGIO clock frequency divided by 4.

6.1.2 Resource Configuration Required to Emulate an I2S Bus Slave

Three timers, two shifters, and four pins can be used to support I2S slave mode. Transmit data must be written to the transmit buffer before the external frame sync becomes active; otherwise, the shifter error flag is set. Because the maximum clock synchronization delay is 1.5 cycles, plus one more cycle to output data, the I2S slave output is valid after up to 2.5 CFGIO clock cycles. Therefore, the maximum baud rate is the CFGIO clock frequency divided by 6. When timer 1 output is active, timer 1 detects the rising edge of the bit clock and controls the shift register for a 32-bit transfer. Timer 2 detects the falling edge of frame sync and keeps its output active until the falling edge of the bit clock. Timer 0 detects the falling edge of the bit clock while timer 2 output is active and asserts its output for the frame length.

6.2 Driver Component Configuration and API Function Descriptions

Master device user structure:

```
typedef struct
{
    CFGIO_DRIVER_TYPE_T driverType;      /* Driver type: interrupt/polling/DMA */
    uint32_t baudRate;                   /* Baud rate, in Hz */
    uint8_t bitsWidth;                   /* Number of bits in a word; must be a multiple of 8 */
    uint8_t txPin;                       /* CFGIO pin used for transmission */
    uint8_t rxPin;                       /* CFGIO pin used for reception */
```

```
    uint8_t sckPin;                    /* CFGIO pin used for the serial clock */
    uint8_t wsPin;                     /* CFGIO pin used for word select */
    I2S_CALLBACK_T callback;           /* User callback function */
    void *callbackParam;               /* Parameter for the callback function */
    uint8_t rxDMAChannel;              /* Rx DMA channel number, used only in DMA mode */
    uint8_t txDMAChannel;              /* Tx DMA channel number, used only in DMA mode */
} CFGIO_I2S_MASTER_CFG_T;
```

Slave device user structure:

```
typedef struct
{
    CFGIO_DRIVER_TYPE_T driverType;    /* Driver type: interrupt/polling/DMA */
    uint8_t bitsWidth;                 /* Number of bits in a word; must be a multiple of 8 */
    uint8_t txPin;                     /* IO pin used for transmission */
    uint8_t rxPin;                     /* IO pin used for reception */
    uint8_t sckPin;                    /* CFGIO pin used for the serial clock */
    uint8_t wsPin;                     /* CFGIO pin used for word select */
    I2S_CALLBACK_T callback;           /* User callback function */
    void *callbackParam;               /* Parameter for the callback function */
    uint8_t rxDMAChannel;              /* Rx DMA channel number */
    uint8_t txDMAChannel;              /* Tx DMA channel number, used only in DMA mode */
} CFGIO_I2S_SLAVE_CFG_T;
```

Master initialization function:

```
STATUS_T CFGIO_I2S_DRV_MasterInit(          /* Returns error status */
uint32_t instance,                          /* CFGIO peripheral instance number */
const CFGIO_I2S_MASTER_CFG_T * userConfigPtr, /* Pointer to the CFGIO_I2S master user
configuration structure */
```

```
CFGIO_I2S_MASTER_STATE_T * master);          /* User-defined CFGIO_I2S master state */
```

Slave initialization function:

```
STATUS_T CFGIO_I2S_DRV_SlaveInit(            /* Returns error status */
uint32_t instance,                           /* CFGIO peripheral instance number */
const CFGIO_I2S_SLAVE_CFG_T * userConfigPtr, /* Pointer to the CFGIO_I2S slave user
configuration structure */
cfgio_i2s_slave_state_t * slave);            /* User-defined CFGIO_I2S slave state */
```

Master blocking send:

```
STATUS_T CFGIO_I2S_DRV_MasterSendDataBlocking( /* Returns error status */
CFGIO_I2S_MASTER_STATE_T * master,            /* User-defined CFGIO_I2S master state */
const uint8_t * txBuff,                      /* User transmit data buffer */
uint32_t txSize,                             /* Length of data to send, in bytes */
uint32_t timeout);                           /* Timeout, in milliseconds */
```

Slave blocking send:

```
STATUS_T CFGIO_I2S_DRV_SlaveSendDataBlocking( /* Returns error status */
cfgio_i2s_slave_state_t * slave,              /* User-defined CFGIO_I2S slave state */
const uint8_t * txBuff,                       /* User transmit data buffer */
uint32_t txSize,                              /* Length of data to send, in bytes */
uint32_t timeout);                            /* Timeout, in milliseconds */
```

Master blocking receive:

```
STATUS_T CFGIO_I2S_DRV_MasterReceiveDataBlocking( /* Returns error status */
CFGIO_I2S_MASTER_STATE_T * master,            /* User-defined CFGIO_I2S master state */
uint8_t * rxBuff,                            /* User transmit data buffer */
```

```
    uint32_t rxSize,                                    /* Length of data to receive, in bytes */
    uint32_t timeout);                                  /* Timeout, in milliseconds */
```

Slave blocking receive:

```
STATUS_T CFGIO_I2S_DRV_SlaveReceiveDataBlocking( /* Returns error status */
    cfgio_i2s_slave_state_t * slave,                    /* User-defined CFGIO_I2S slave state */
    uint8_t * rxBuff,                                   /* User transmit data buffer */
    uint32_t rxSize,                                    /* Length of data to send, in bytes */
    uint32_t timeout);                                  /* Timeout, in milliseconds */
```

6.3 Test Case

6.3.1 Case Description

This case uses the CFGIO module to emulate I2S communication. One CFGIO instance is configured as the slave, and another CFGIO instance is configured as the master. Data is transferred over the I2S bus by interrupt.

6.3.2 Hardware Preparation

- G32A1445 EVAL board × 1
- J-Link programmer × 1, including Type-B to Type-A USB cable and FC-20P dual-head cable/2.54mm pitch
- Mini USB data cable × 1
- Several jumper wires
- Computer × 1

6.3.3 Software Preparation

- G32A1xxx_SDK_x.x.x
- Keil MDK and G32A series support package
- SEGGER – J-Link Vx.xx
- Serial port assistant

6.3.4 Test Procedure

(1) Open the corresponding Keil MDK project, compile it, and download it to the evaluation board: “G32A1xxx_SDK_x.x.x\Examples\G32A1445\CFGIO\CFGIO_I2S\Project\MDK\CFGIO_I2S.uvprojx”. Compile the project, download it to the G32A1445 EVAL board, and then disconnect the J-Link programmer from the board.

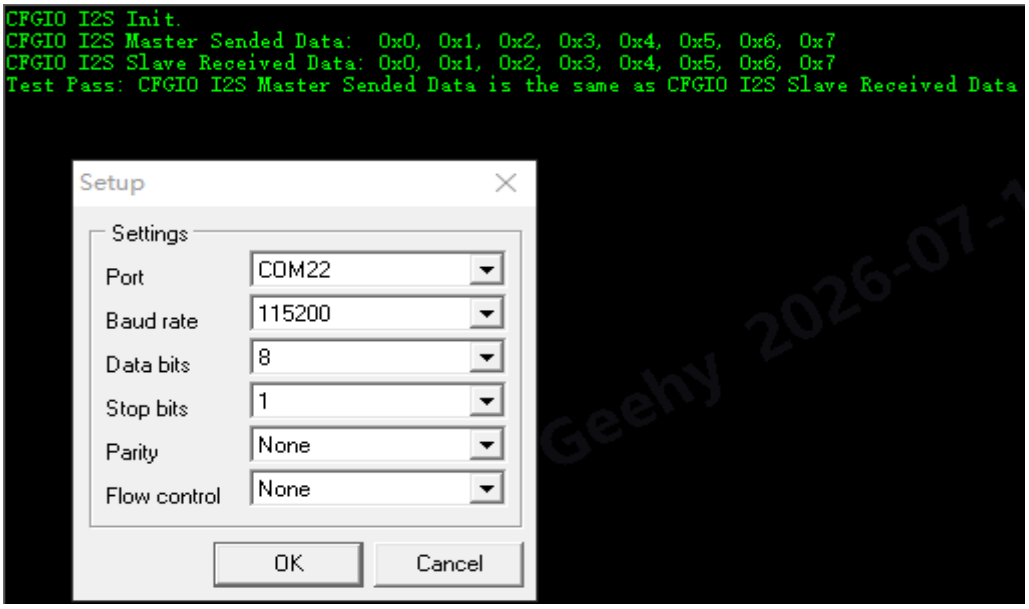
(2) Device connection: Connect the pins on the evaluation board as follows:

	master		slave	
sck	PD9	<==>	PA2	sck
ws	PA11	<==>	PA3	ws
tx	PA0	<==>	PA9	rx
rx	PA1	<==>	PA8	tx

(3) Connect one end of the Mini USB cable to the evaluation board and the other end to the computer. Open the serial port assistant on the computer, configure the serial parameters, and open the corresponding port.

(4) Press the RST reset button. The serial port assistant receives the print information sent by the board. If I2S communication is successful, the green LED lights up; otherwise, the red LED lights up.

Figure 6 CFGIO_I2S.uvprojx Example I2S Master/Slave Send/Receive



7 Revision History

Table 1 Document Revision History

Date	Version	Revision History
July, 2026	1.0	● Initial release

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.