what is SPI communication in LORA radio

SPI (Serial Peripheral Interface) communication is a common way of communicating between microcontrollers and other peripherals, such as sensors and radios. LoRa radio modules often use SPI as the communication protocol to interface with microcontrollers.

In LoRa radios, SPI communication is used to configure the radio module's parameters, such as frequency, bandwidth, and spreading factor. SPI is also used to send and receive data between the radio module and the microcontroller.

In SPI communication, data is transmitted in a serial fashion, one bit at a time, over a synchronous clock signal. The SPI interface typically has four signal lines: MOSI (Master Output Slave Input), MISO (Master Input Slave Output), SCK (Serial Clock), and SS (Slave Select).

The MOSI line is used to transmit data from the master (microcontroller) to the slave (radio module), while the MISO line is used to transmit data from the slave to the master. The SCK line provides the clock signal to synchronize the data transfer, and the SS line is used to select the slave device that the master wants to communicate with.

Overall, SPI communication is a simple and efficient way to communicate with LoRa radio modules and other peripherals, and is widely used in the embedded systems industry.

digitalWrite(\_hwConfig.PIN\_LORA\_NSS, HIGH);

The code digitalWrite(\_hwConfig.PIN\_LORA\_NSS, HIGH) sets the pin specified by the \_hwConfig.PIN\_LORA\_NSS variable to a high voltage level.

In LoRa radio modules, the NSS (Slave Select) pin is used to select the radio module as the target device for communication. By setting the NSS pin to a low voltage level, the microcontroller signals to the radio module that it is ready to send or receive data.

Conversely, setting the NSS pin to a high voltage level signals to the radio module that the communication is over and it can go into a low power consumption mode, also known as sleep mode.

Therefore, digitalWrite(\_hwConfig.PIN\_LORA\_NSS, HIGH) in your code is likely signaling the end of a communication session with the LoRa radio module and putting it into a low power consumption mode to save power.

SPI\_LORA.beginTransaction(spiSettings);

SPI\_LORA.transfer(RADIO\_READ\_REGISTER);

The code you provided is using the SPI interface to communicate with a LoRa radio module, likely to read a register value from the module.

SPI\_LORA.beginTransaction(spiSettings) initializes the SPI interface with the specified spiSettings configuration, such as clock frequency, data order, and clock polarity. This function prepares the SPI interface for communication with the LoRa radio module.

SPI\_LORA.transfer(RADIO\_READ\_REGISTER) sends the command to read the register from the radio module. The RADIO\_READ\_REGISTER variable likely contains the value of the command code or register address to be read from the radio module. This function sends the command byte over the SPI bus and returns the value received from the radio module in response to the command.

Note that after the data transfer is complete, the SPI interface should be properly deinitialized and any necessary chip select pins should be set back to their inactive state to prevent unwanted communication with other devices on the bus.

return (buf[0] << 24) | (buf[1] << 16) | (buf[2] << 8) | buf[3];

The code return (buf[0] << 24) | (buf[1] << 16) | (buf[2] << 8) | buf[3]; takes the four elements of the buf array, which are likely received over the SPI bus from the LoRa radio module, and combines them into a single 32-bit unsigned integer value.

The bitwise left shift (<<) operator shifts the binary representation of the value in buf[i] by the specified number of bits to the left. In this case, buf[0] is shifted by 24 bits, buf[1] is shifted by 16 bits, buf[2] is shifted by 8 bits, and buf[3] is not shifted. The result of each shift operation is a value with the bits of buf[i] in the appropriate position in a 32-bit integer.

The bitwise OR (|) operator combines the shifted values into a single 32-bit integer by performing a bitwise OR operation on each corresponding bit position of the shifted values. The resulting value is the 32-bit integer formed by concatenating the four bytes in buf.

Note that the order of the bytes in the resulting integer may depend on the endianness of the system architecture. Some architectures store the most significant byte first (big-endian), while others store the least significant byte first (little-endian). Therefore, it is important to ensure that the byte order is interpreted correctly when reading data from the SPI bus.

for (uint16\_t i = 0; i < size; i++)

{

buffer[i] = SPI\_LORA.transfer(0x00);

}

The code you provided reads size bytes of data from the SPI bus and stores them in the buffer array.

The for loop iterates over the size bytes of data to be read from the SPI bus. uint16\_t i declares a 16-bit unsigned integer i as the loop counter, which is initialized to zero. The loop continues as long as i is less than size, and increments i by one at the end of each iteration.

Inside the loop, SPI\_LORA.transfer(0x00) sends a dummy byte (0x00) over the SPI bus to trigger the response from the LoRa radio module, and reads the byte that is received in response. The byte that is read from the SPI bus is stored in the buffer[i] element, which is the i-th element of the buffer array.

After the loop is completed, the buffer array will contain size bytes of data read from the SPI bus. This data can then be processed as necessary in the rest of the program. Note that the SPI interface should be properly deinitialized and any necessary chip select pins should be set back to their inactive state to prevent unwanted communication with other devices on the bus.

SPISettings spiSettings = SPISettings(2000000, MSBFIRST, SPI\_MODE0);

The line SPISettings spiSettings = SPISettings(2000000, MSBFIRST, SPI\_MODE0); initializes an object spiSettings of the SPISettings class with specific parameters for configuring the SPI bus.

The SPISettings constructor takes three arguments:

uint32\_t clock: This is the clock frequency in hertz that the SPI bus should use for data transfer. In this case, the clock frequency is set to 2 MHz, or 2,000,000 Hz.

uint8\_t bitOrder: This specifies the bit order for data transfer. The MSBFIRST constant indicates that the most significant bit (MSB) is transmitted first.

uint8\_t dataMode: This specifies the SPI data mode, which determines the clock polarity and phase. The SPI\_MODE0 constant specifies a clock polarity of 0 (low when inactive) and a clock phase of 0 (data is captured on the first edge of the clock).

The spiSettings object is then used later in the code to configure the SPI bus with the specified clock frequency, bit order, and data mode. These settings must be consistent with the settings used by the LoRa radio module for successful communication over the SPI bus.

while( (b = readReg(LORARegRssiWideband) & 0x01) == (readReg(LORARegRssiWideband) & 0x01) );

The code while( (b = readReg(LORARegRssiWideband) & 0x01) == (readReg(LORARegRssiWideband) & 0x01) ); creates a loop that waits until the least significant bit (LSB) of the LORARegRssiWideband register changes.

Here's how the code works:

The readReg(LORARegRssiWideband) function is called twice within the same statement. The first call reads the current value of the LORARegRssiWideband register and applies a bitwise AND operation with 0x01 to extract the LSB. The result is then assigned to the variable b.

The second call to readReg(LORARegRssiWideband) reads the register value again and applies the same bitwise AND operation with 0x01 to extract the LSB.

The loop condition checks whether the value of b (which holds the LSB read from the first call to readReg(LORARegRssiWideband)) is equal to the LSB read from the second call to readReg(LORARegRssiWideband).

If the two values are equal, the loop continues and the process repeats. If the two values are not equal, the loop exits and the program continues with the next line of code.

In other words, the loop waits until the LSB of the LORARegRssiWideband register changes before continuing. This is likely used to synchronize the program with changes in the radio signal strength indicator (RSSI) of the LoRa radio module.

rxlora(RXMODE\_RSSI);

while( (readReg(RegOpMode) & OPMODE\_MASK) != OPMODE\_RX ); // continuous rx

The rxlora(RXMODE\_RSSI) function call sets the radio module to receive mode with a specific mode of operation (RXMODE\_RSSI). This mode is typically used to receive packets with the strongest signal, as it continuously listens to the channel and reports the RSSI (Received Signal Strength Indication) for each packet received.

The subsequent while loop waits for the radio to complete its transition to receive mode, indicated by the value of the RegOpMode register being equal to OPMODE\_RX. This loop ensures that the radio is fully configured and ready to receive before proceeding with any further operations.

void SX126xReadRegisters(uint16\_t address, uint8\_t \*buffer, uint16\_t size)

{

SX126xCheckDeviceReady();

digitalWrite(\_hwConfig.PIN\_LORA\_NSS, LOW);

SPI\_LORA.beginTransaction(spiSettings);

SPI\_LORA.transfer(RADIO\_READ\_REGISTER);

SPI\_LORA.transfer((address & 0xFF00) >> 8);

SPI\_LORA.transfer(address & 0x00FF);

SPI\_LORA.transfer(0x00);

for (uint16\_t i = 0; i < size; i++)

{

buffer[i] = SPI\_LORA.transfer(0x00);

}

SPI\_LORA.endTransaction();

digitalWrite(\_hwConfig.PIN\_LORA\_NSS, HIGH);

SX126xWaitOnBusy();

}

The function SX126xReadRegisters reads a block of registers from the SX126x LoRa radio module using SPI communication. The function takes three arguments:

uint16\_t address: This is the starting address of the block of registers to be read from the LoRa radio module.

uint8\_t \*buffer: This is a pointer to the buffer where the data read from the registers will be stored.

uint16\_t size: This is the number of bytes to be read from the registers.

The function starts by calling the SX126xCheckDeviceReady function, which checks if the LoRa radio module is ready for communication. This function ensures that the module is not currently transmitting or receiving data and can safely receive commands over the SPI bus.

The function then pulls the chip select (NSS) pin of the LoRa radio module low to select the device for communication over the SPI bus. The SPI\_LORA.beginTransaction function is then called with the spiSettings object to configure the SPI bus with the specified clock frequency, bit order, and data mode.

The function then sends a command over the SPI bus to read from the specified register address, followed by the address itself split into two bytes (high and low). The fourth byte sent over the bus is a dummy byte (0x00) used to trigger the LoRa radio module to respond with the register data.

The function then enters a loop that reads size bytes of data from the registers, one byte at a time, using the SPI\_LORA.transfer function. The bytes are stored in the buffer array pointed to by the buffer parameter.

After all bytes have been read, the SPI\_LORA.endTransaction function is called to end the SPI transaction, and the chip select (NSS) pin is set high to end communication with the LoRa radio module.

Finally, the function calls SX126xWaitOnBusy to wait for the LoRa radio module to finish any ongoing operations and become ready for further commands over the SPI bus.

uint64\_t frf = ((uint64\_t)LMIC.freq << 19) / 32000000;

This line of code is performing a frequency calculation for the LoRaWAN stack implemented in the LMIC library for Arduino. Specifically, it is calculating the frequency register value to be written to the radio hardware to configure the center frequency for LoRa transmission.

The formula for calculating the frequency register value is:

frf = (desired frequency in Hz / 32 MHz) \* 2^19

In this line of code, the desired frequency is stored in the LMIC.freq variable, which is likely set elsewhere in the code. The frequency is then left-shifted by 19 bits to perform the multiplication by 2^19. Finally, the result is divided by 32 MHz to complete the frequency calculation.

The resulting value frf is a 64-bit unsigned integer that represents the frequency register value to be written to the radio hardware. This value is typically split into two 32-bit registers to configure the frequency in the radio hardware.

static ostime\_t nextJoinState (void) { lmic.c

// Try the following:

// SF7/8/9/10 on a random channel 0..63

// SF8C on a random channel 64..71

//

u1\_t failed = 0;

if( LMIC.datarate != DR\_SF8C ) {

LMIC.txChnl = 64+(LMIC.txChnl&7);

setDrJoin(DRCHG\_SET, DR\_SF8C);

} else {

LMIC.txChnl = os\_getRndU1() & 0x3F;

s1\_t dr = DR\_SF7 - ++LMIC.txCnt;

if( dr < DR\_SF10 ) {

dr = DR\_SF10;

failed = 1; // All DR exhausted - signal failed

}

setDrJoin(DRCHG\_SET, dr);

}

This is a snippet of code written in C that is part of the LoRaWAN MAC in C (LMIC) library used in conjunction with Semtech's SX126x LoRa transceivers and the Arduino platform. Specifically, this function, nextJoinState(), is used during the process of joining a LoRaWAN network.

The purpose of this function is to set the next transmission channel and data rate (datarate) to use during the join process. The function first checks if the current datarate is not DR\_SF8C. If it is not, it sets the transmission channel to a random channel in the range of 64 to 71 and sets the datarate to DR\_SF8C. If the current datarate is DR\_SF8C, it sets the transmission channel to a random channel in the range of 0 to 63 and sets the datarate to a value between DR\_SF7 and DR\_SF10 based on the current transmission count.

The function also sets a flag called failed to 1 if all the datarates between DR\_SF7 and DR\_SF10 have been exhausted, indicating that the join process has failed.

Overall, this function helps to determine the best channel and datarate to use during the join process, ensuring reliable communication between the device and the LoRaWAN network.