CE226537 – CYW20819 SPI-Based Datalogger

Objective

This code example includes two applications that demonstrate the operation of multiple Serial Peripheral Interface (SPI) interfaces using the CYW20819 Bluetooth SoC and ModusToolbox™ Integrated Development Environment (IDE). The first application demonstrates operation of two SPI masters – one for collecting sensor data and the other for logging the data to external Flash. The second application demonstrates operation of an SPI slave which is used for providing sensor data to the first application.

Requirements

Tool: ModusToolbox IDE 1.1 or later version

Programming Language: C

Associated Parts: CYW20819

Related Hardware: CYW920819EVB-02 Evaluation Kit

Overview

The application *dual_spi_master* demonstrates two SPI master instances using multiple threads. The SPI1 instance is used to communicate with an external SPI sensor and the SPI2 instance is used to communicate with an SFLASH on the kit. The received sensor values are written to SFLASH by coupling with a timestamp of the temperature reading using the RTC. The user can initiate a read operation on the SFLASH by pressing user button (SW3) on the CYW920819EVB-02 kit to obtain and display the stored temperature records. A NVRAM module using internal flash is used to store the last temperature record number that was stored in SFLASH, so that the application resumes without overwriting the stored records during reset or power failure.

This application showcases usage of:

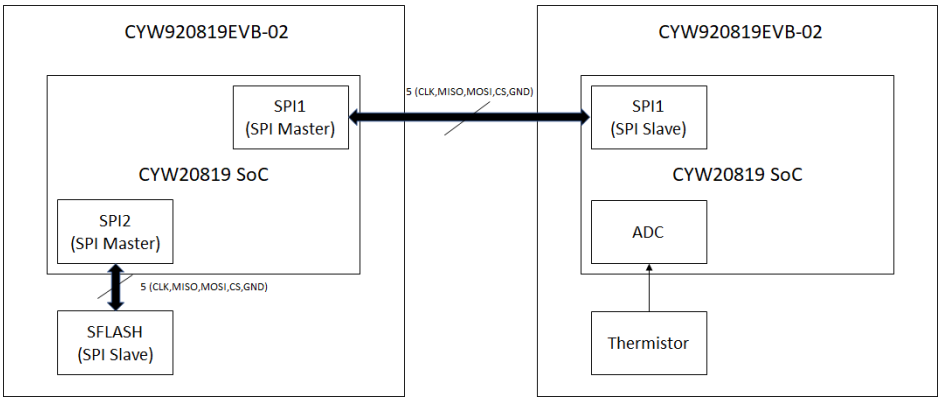
1. Multiple threads communicating via an RTOS semaphore and message queue
2. Two SPI instances available on the device in the same application
3. SFLASH middleware library
4. NVRAM middleware for application data

The application *spi_slave_sensor* demonstrates how to use the SPI slave interface to send and receive bytes or a stream of bytes over the SPI hardware. This kit emulates a SPI based temperature sensor by reading an on-board thermistor using the ADC on the device and sending it to the master whenever a new value is requested. These two applications should be used in conjunction with each other.

Hardware Setup

These applications run on two separate CYW920819EVB-02 kits. Both applications use the kit's default configuration. Refer to the kit guide, if required, to ensure the kit is configured correctly. The block diagram depicting the connections between different blocks of two CYW920819EVB-02 kits is shown in Figure 1.

Figure 1. Block Diagram



Make the connections as shown in Table 1.

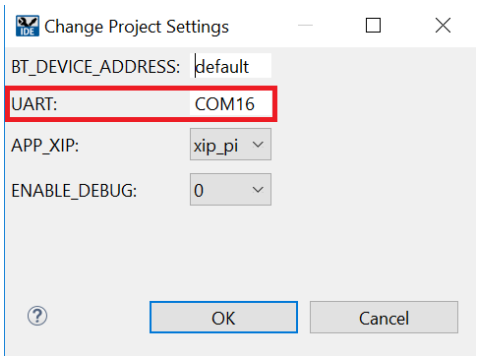
Table 1. Hardware Connections

Function	Master			Slave		
CLK	WICED_P15	J3.8	D10	WICED_P09	J3.5	D13
MISO	WICED_P14	J3.10	D8	WICED_P17	J3.6	D12
MOSI	WICED_P13	J12.6	A05	WICED_P14	J3.10	D8
CS	WICED_P12	J12.5	A04	WICED_P15	J3.8	D10
GND	GND	J11.6	GND	GND	J11.6	GND

Operation

1. Connect two kits to your PC using the provided USB cable. Note down the port enumerations for each device in **Device Manager > Ports (COM & LPT)** (Windows only). The enumeration with a smaller number is the HCI UART port and the other is Peripheral UART port.
2. Import the 2 code examples into a workspace. See KBA225201.
3. In the project explorer, right-click each **<App Name>_mainapp** project one at a time and select **Change Application Settings** in the drop-down menu. Configure the UART port to match the HCI UART port enumeration as shown in Figure 2 instead of using AUTO so that the master application is set for the HCI UART port of one of the kits and the slave application is set for the HCI UART of the other kit.

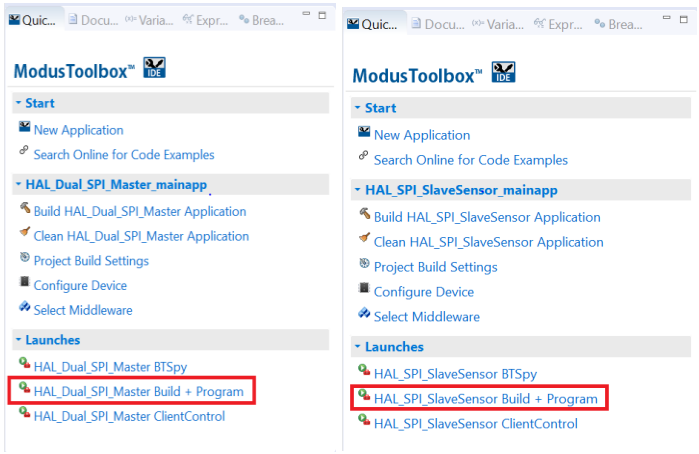
Figure 2. Change Project Settings



Alternatively, instead of manually setting the UART port in the application settings, you can connect the kits one at a time to program the applications and then connect both kits once programming is done.

4. Program the CYW20819 device on the kit with the application *dual_spi_master*. In the project explorer, select the **<App Name>_mainapp** project for the master. In the Quick Panel, scroll to the **Launches** section, and click the **<App Name> Build + Program** configuration as shown in Figure 3. Repeat this step to program the other kit with *spi_slave_sensor* and ensure that the UART port selected is for the other kit.

Figure 3. Programming the CYW20819 Device from ModusToolbox



Note: If the download fails, it is possible that a previously loaded application is preventing programming. For example, application might use a custom baud rate that the download process does not detect or it might be in a low power mode. In that case, it may be necessary to put the board in recovery mode, and then try the programming operation again from the IDE. To enter recovery mode, first, press and hold the **Recover** button (SW1), then press the **Reset** button (SW2), release the **Reset** button (SW2), and then release the **Recover** button (SW1).

5. Open a serial terminal application, such as Tera Term, and connect to the **WICED Peripheral UART** port of both the kits.

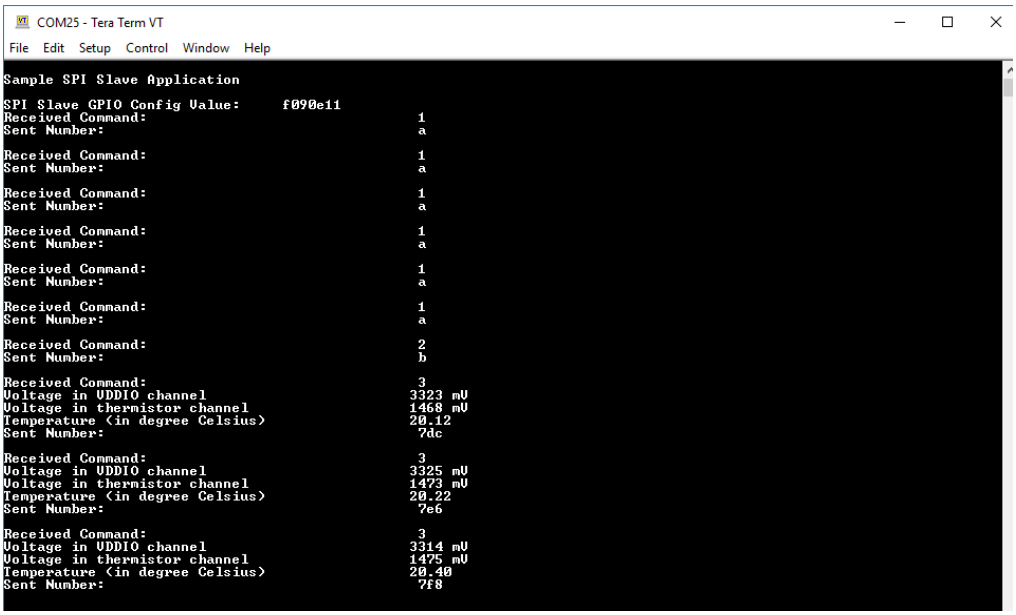
   Configure the terminal application to access the serial port using settings listed in Table 2.

Table 2. WICED Peripheral UART Settings

WICED Peripheral UART Serial Port Configuration	Value
Baud rate	115200 bps
Data	8 bits
Parity	None
Stop	1 bit
Flow control	None
New-line for Receive data	Line Feed (LF) or Auto setting

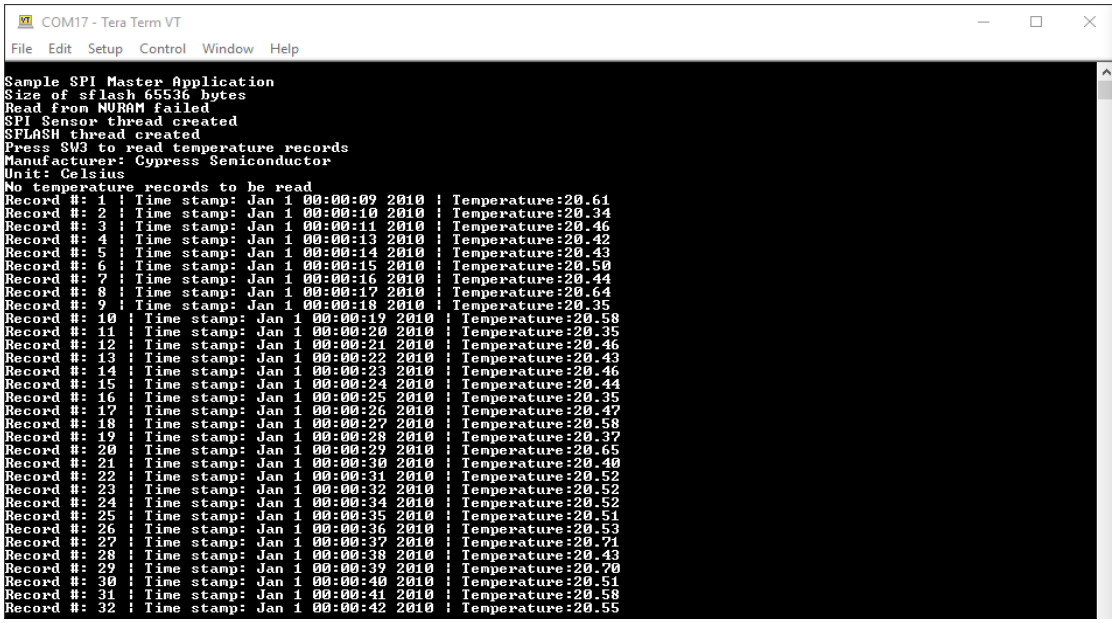
   a. The slave serial terminal window displays the SPI command received and the accompanying response on the terminal window, as shown in Figure 4.

Figure 4. Serial terminal Output for SPI Slave



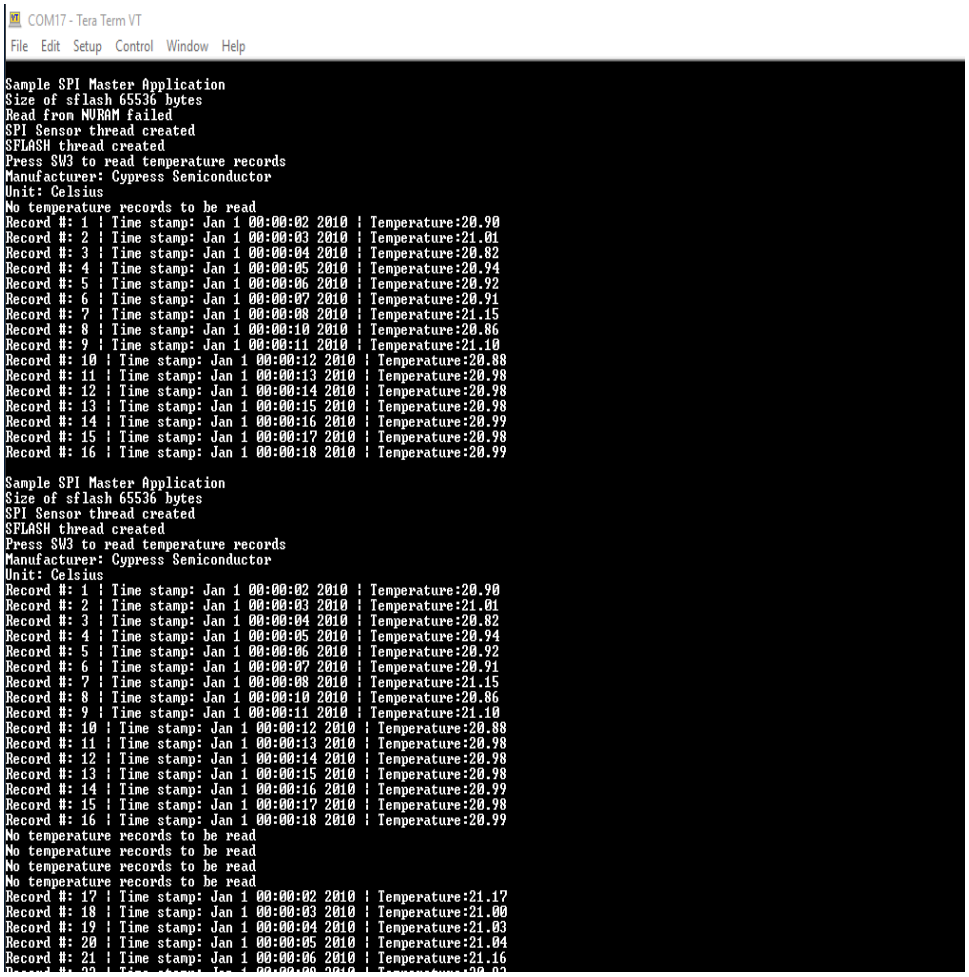
b. The master serial terminal window displays a message that prompts the user to press the user button to view the temperature records. When the user presses the button, if there are no temperature records to be read from SFLASH, an appropriate message is displayed on the window. Otherwise, the temperature records that have not already been read by the user are displayed on the terminal window as shown in Figure 5.

Figure 5. Serial Terminal Output of SPI Master



6. The master stores in the NVRAM the number of the record that was last stored in the SFLASH. This feature ensures that even when the master is powered down or reset, prior temperature records are not lost by overwriting the previous data. The user can test this feature by pressing the reset button (SW2) and then user button (SW3). The terminal displays all the stored temperature records from the beginning as shown in Figure 6. Note that the RTC timestamp value is reset for the new values that are stored, so there won't be continuity in the timestamp values between the stored records before reset and the stored records after reset.

Figure 6. Serial Terminal Output of SPI Master After Reset



Design and Implementation

SPI Master

This section describes the details of the implementation of the SPI Master.

On startup, the application sets up the UART and then starts the Bluetooth stack in application_start(). Once the stack is started (`BTM_ENABLED_EVT`), it calls the `initialize_app()` function which handles the remaining functionality. Note that the Bluetooth stack is running but we are not using Bluetooth in this application, so it doesn't do anything once the stack is started. The `initialize_app()` function initializes both the SPI interfaces, RTC, GPIO and two separate threads which handle the two different SPI transactions – one to read the SPI sensor and one to write the SFLASH. The two threads communicate with each other through a queue to transfer temperature records. A semaphore is used to signal the thread handling SFLASH when data of one page size is available in the queue for writing. In this implementation, the page size is 256 bytes and the size of each temperature record is 16 bytes, so records are written when 256/16 = 16 temperature values have been received.

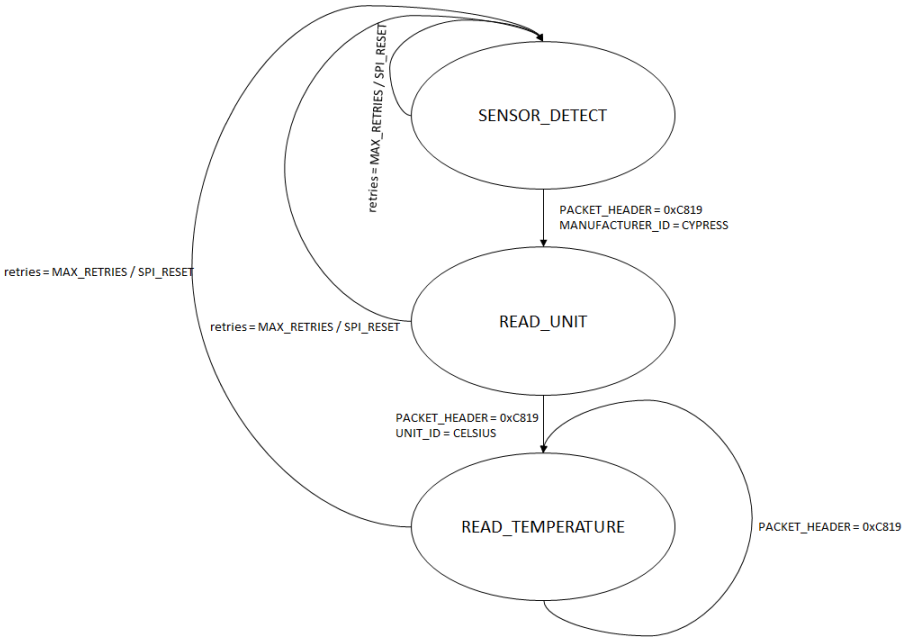
Also, during initialization the number of the last record that was stored in NVRAM is obtained. The NVRAM read provides the number of temperature records that are present in SFLASH so that if the master is reset or powered down, the current address (i.e. where new temperature records will be written) is updated to address the next free location. This prevents overwriting over stored temperature records upon reset. When the application is running for the first time after download the NVRAM read fails since no write has been performed. Here, the record number read from NVRAM is re-assigned to 0 to prevent using an erroneous value for the record number.

The temperature readings are held in a structure called “temperature_record” which contains the following elements:

- record_no: stores the number of the temperature record
- dec_temp: holds the decimal part of the temperature reading
- frac_temp: holds the fractional part of the temperature reading
- timestamp: holds the time when temperature reading was received. Note that the timestamp values by default start from 00:00:00 Hrs, January 1, 2010.

In the thread handling SPI communication with the slave (spi_sensor_thread), a finite state machine is used to determine the data that the master requests as shown in Figure 7.

Figure 7. FSM Adopted for Communicating with Slave



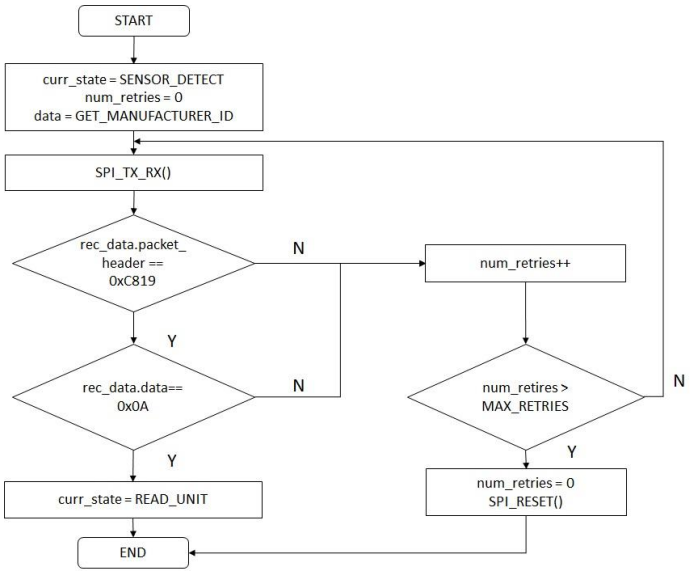
The finite state machine contains three states:

- SENSOR_DETECT
- READ_UNIT
- READ_TEMPERATURE

In each state, the slave is verified to be a known slave using a packet header before processing the data that is sent from the slave. If the master is not able to authenticate the slave, the master remains in the same state and retries. After five retries, the SPI interface is reset, and the master starts from the SENSOR_DETECT state.

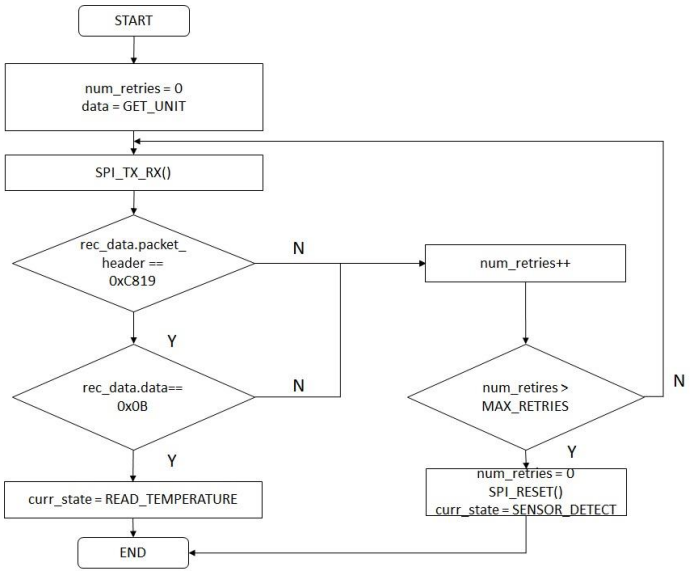
In the SENSOR_DETECT state, the master requests the Manufacturer ID to verify that the slave's manufacturer is Cypress. If the slave responds with an unknown Manufacturer ID, the master informs the user that the slave's identity could not be authenticated. If the slave responds with the expected Manufacturer ID, the master enters the next state, READ_UNIT. A flowchart illustrating the operation is shown in Figure 8.

Figure 8. Flowchart of SENSOR_DETECT State



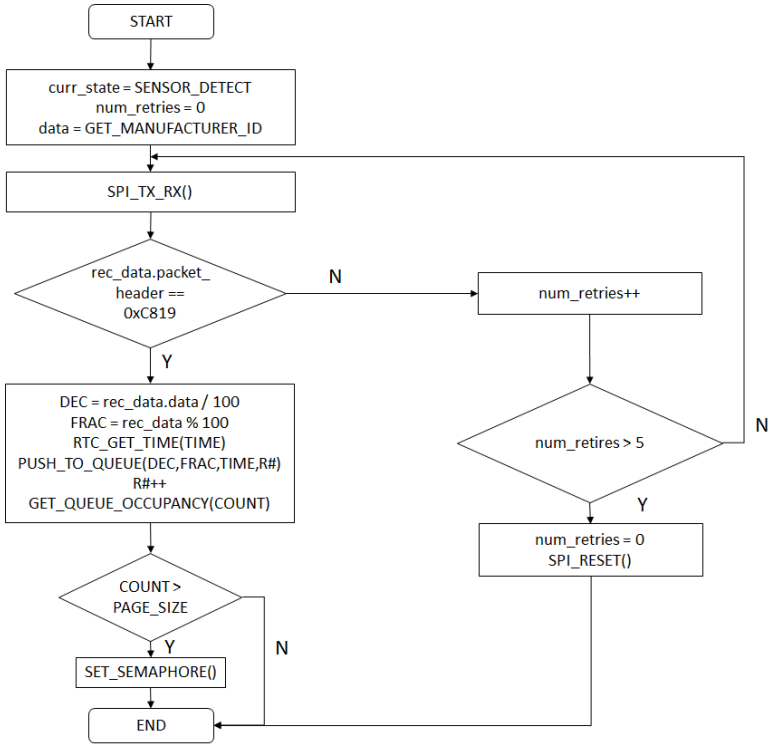
In the READ_UNIT state, the master requests the Unit ID to know the unit of temperature values provided by the slave. If the slave responds with an unknown Unit ID, the master informs the user that the unit of temperature is unknown and tries to obtain the unit again. If the number of retries exceeds 5, the master changes its state to SENSOR_DETECT. Otherwise, if the slave responds as expected, the master enters the next state, READ_TEMPERATURE. A flowchart illustrating the operation is shown in Figure 9.

Figure 9 Flowchart Showing READ_UNIT State



In the READ_TEMPERATURE state, the master requests the temperature from the slave. The temperature reading received comprises the decimal and fractional parts of the temperature. For instance, if the temperature reading is 24.44 C, the slave sends 2444 as the data. The Master then stores the quotient as the decimal part of temperature and remainder as the fractional part of temperature in the temperature record. The RTC is used to obtain the time when the temperature reading was received so that it can be included in the temperature record. The number of the temperature record is also updated and pushed to the queue. The queue occupancy is checked every time after a temperature record is pushed to the queue. Once the queue occupancy equals the size of a page in SFLASH, the thread sets a semaphore signaling the thread handling SFLASH to start popping data from queue to write to SFLASH. A flowchart illustrating the operation is shown in Figure 10.

Figure 10. Flowchart Showing READ_TEMPERATURE State



The thread handling SFLASH (sflash_thread) performs an erase if the current page being written to belongs to a new sector and waits for the semaphore to be set. Once the semaphore is set, it pops the data from the queue and writes it to SFLASH. The size of the data written is one page (256 bytes) which is 16 temperature records. The size of the data written is chosen as one page to maximize efficiency of the write operation. The record number of the last temperature record that was stored in SFLASH is written to NVRAM in this thread after every page write.

The application code and Bluetooth stack run on the Arm® Cortex®-M4 core of the CYW20819 SoC. The application level source files for *dual_spi_master* is listed in Table 3.

Table 3. Application Source Files

File Name	Comments
dual_spi_master.c	Contains the application_start() function which is the entry point for execution of the user application code after device startup and the threads that handle SPI communication with sensor and SFLASH.
cycfg_pins.c cycfg_pins.h	These files reside in the “GeneratedSource” folder under the application folder. They contain the pin configuration information generated using the Device Configurator tool.

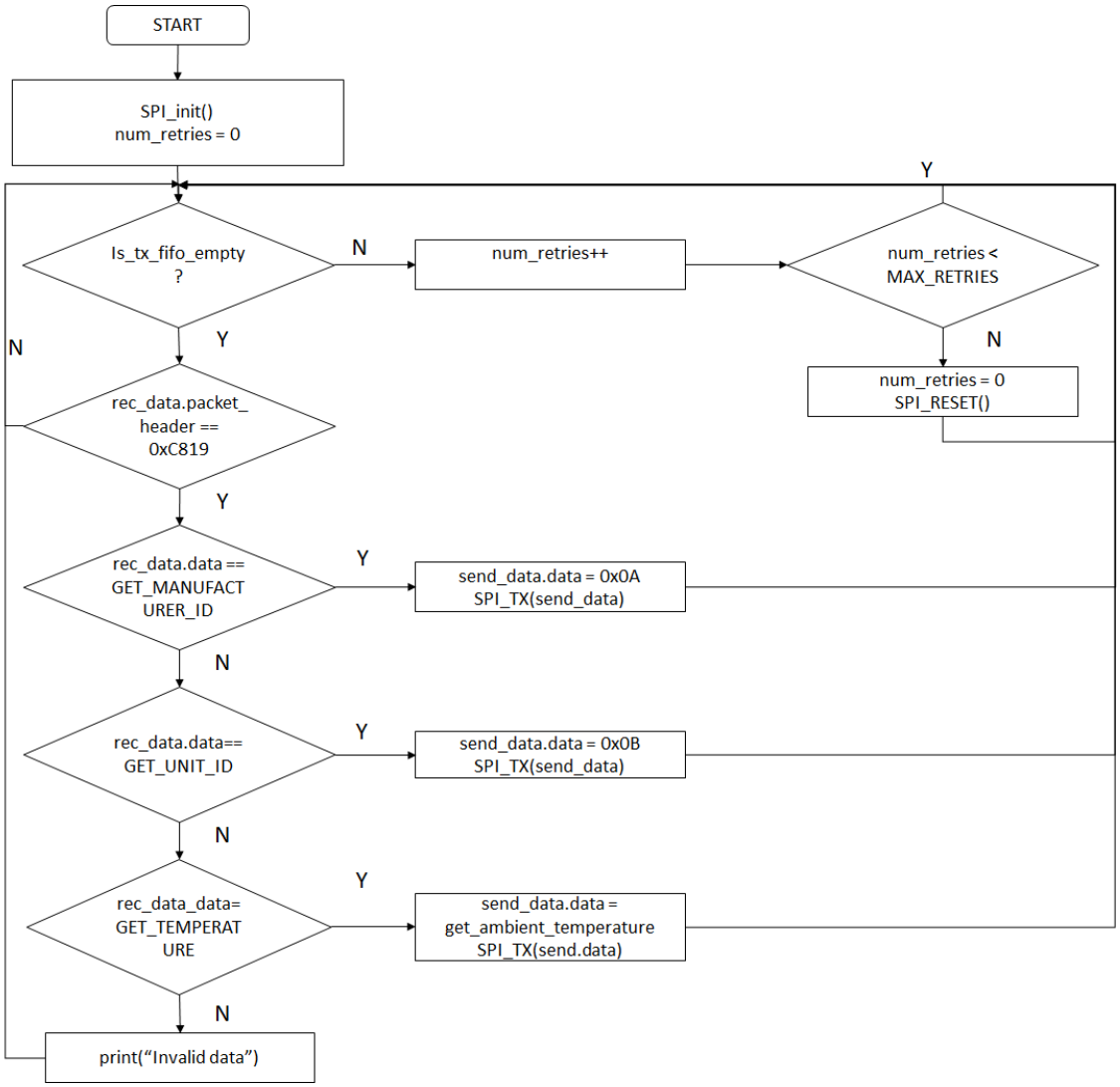
SPI Slave

This section describes the operation of the Slave. As with the master, application_start() sets up the UART and then starts the Bluetooth stack. Once the stack is started (BTM_ENABLED_EVT), it initializes the ADC and then calls the `initialize_app()` function which handles the remaining functionality. Note that again the Bluetooth stack is running but we are not using Bluetooth in this application, so it doesn't do anything once the stack is started. The `initialize_app()` function sets up the SPI interface and then waits for and responds to SPI master commands. There are three commands that the slave will respond to:

- Manufacturer ID: The slave responds with its Manufacturer ID.
- Unit ID: The slave responds with its Unit ID
- Temperature: The slave responds with a temperature reading obtained by acquiring ADC samples

The Slave reads from SPI Rx buffers only when its Tx buffers are empty. If the Slave is unable to empty the Tx buffers after several retries, the SPI interface is reset. A flowchart illustrating the operation of the slave is shown in Figure 11.

Figure 11. Flowchart Showing Slave Operation



The application code and Bluetooth stack run on the Arm® Cortex®-M4 core of the CYW20819 SoC. The application level source files for "spi_slave_sensor" are listed in Table 4.

Table 4. Application Source Files

File Name	Comments
spi_slave_thermistor.c	Contains the `application_start()` function which is the entry point for execution of the user application code after device startup.
thermistor_temp_db.c, thermistor_temp_db.h	These files contain the function to map resistance to temperature values of the thermistor using a lookup table (from the thermistor datasheet).
cycfg_pins.c, cycfg_pins.h	These files reside in the GeneratedSource folder under the application folder. They contain the pin configuration information generated using the Device Configurator tool.

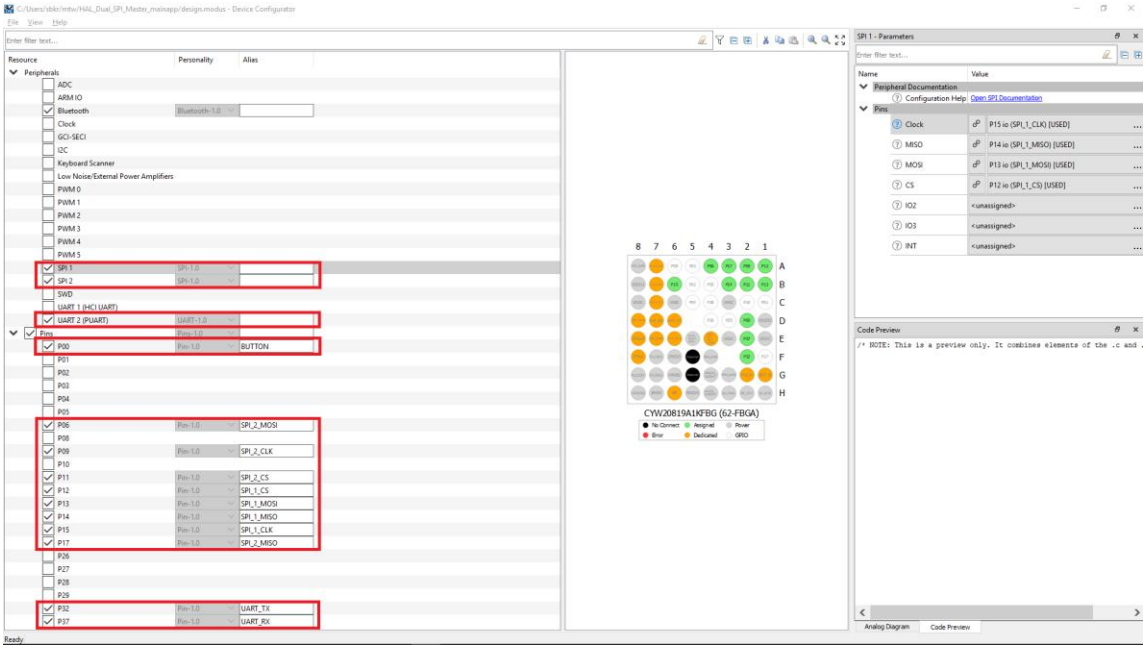
Resources and Settings

SPI Master

This section explains the ModusToolbox resources and their configuration as used for SPI Master. Note that all the configuration explained in this section has already been done in the code example. The ModusToolbox IDE stores the configuration settings of the application in the *design.modus* file. This file is used by the graphical configurators, which generate the configuration firmware. This firmware is stored in the application's *GeneratedSource* folder. For example, the **Peripherals** section of the design file allows you to select the peripherals needed for the application and to configure them as needed. For pin assignment and configuration, review the **Pins** section of the design file.

Figure 12 shows the Device Configurator settings for this code example. The Device Configurator is used to enable/configure the peripherals and the pins used in the application. To launch the Device Configurator, double-click the *design.modus* file or click on **Configure Device** in the Quick Panel. Any time you make a change in the Device Configurator, click **File > Save** to save the updated configuration.

Figure 12. Device Configurator



- **Peripherals:** The design uses two SPI instances and one PUART which are enabled under the Peripherals section. The SPI1 instance communicates with the SPI sensor and its configuration details are provided in Figure 13. The SPI2 instance communicates with the SFLASH and its configuration details are provided in Figure 14. The PUART configuration details are provided in Figure 15.

Figure 13. SPI1 Configuration

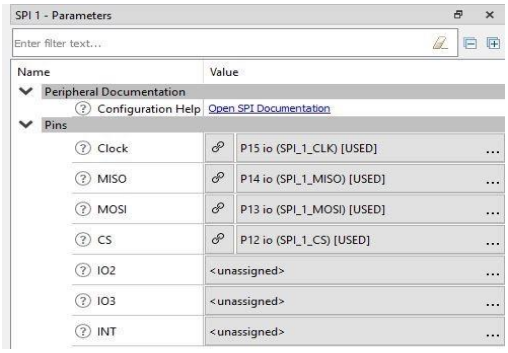


Figure 14. SPI2 Configuration

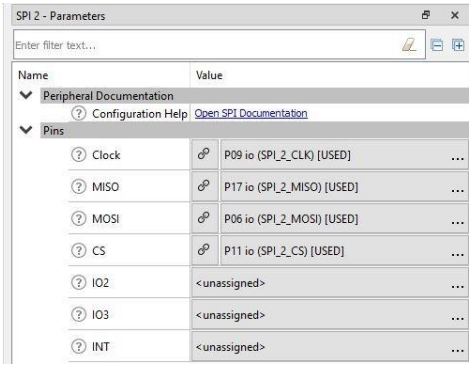
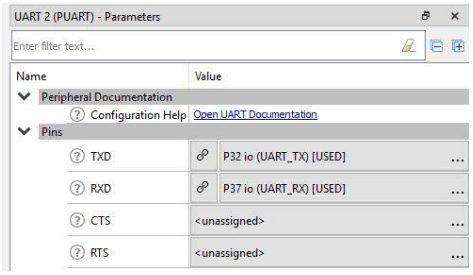


Figure 15. PUART Configuration



- **Pins:** The pins used in the application are enabled in the Pins section of the Device Configurator. Table 5 provides more information on these pins.

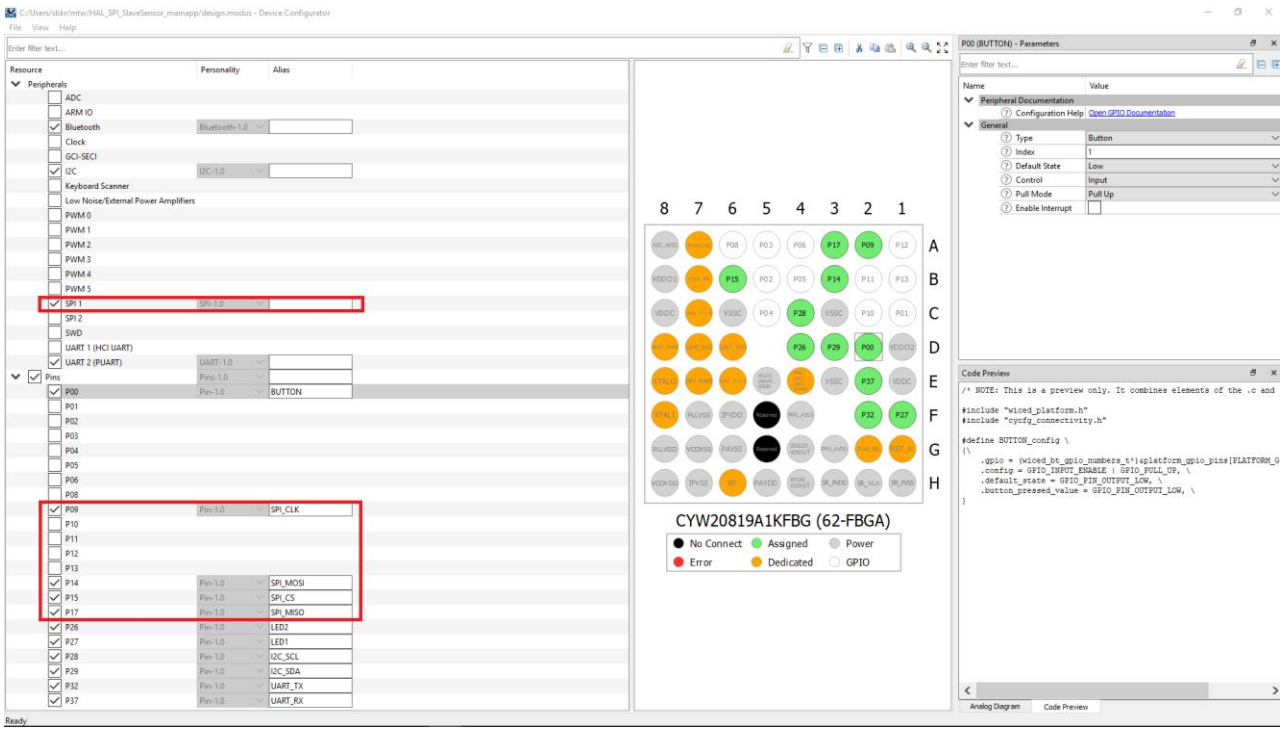
Table 5. Pin Mapping Details for Master

Pin	Alias	Purpose
P32	UART_TX	Used for PUART transmit (Tx)
P37	UART_RX	Used for PUART receive (Rx)
P12	SPI_1_CS	Chip select for sensor
P13	SPI_1_MOSI	MOSI pin to sensor
P14	SPI_1_MISO	MISO pin to sensor
P15	SPI_1_CLK	Clock
P11	SPI_2_CS	Chip select for SFLASH
P06	SPI_2_MOSI	MOSI pin to SFLASH
P17	SPI_2_MISO	MISO pin to SFLASH
P09	SPI_2_CLK	Clock
P00	BUTTON	Button

SPI Slave

This section explains the ModusToolbox resources and their configuration as used for the SPI Slave. Note that all the configuration explained in this section has already been done in the code example. Figure 16 shows the Device Configurator settings for this code example.

Figure 16. Device Configurator for Slave



- **Peripherals:** The example uses a SPI and PUART peripheral. The configuration details of the SPI used to communicate with the master are provided in Figure 17. The configuration details of the PUART are provided in Figure 18.

Figure 17. SPI1 Configuration

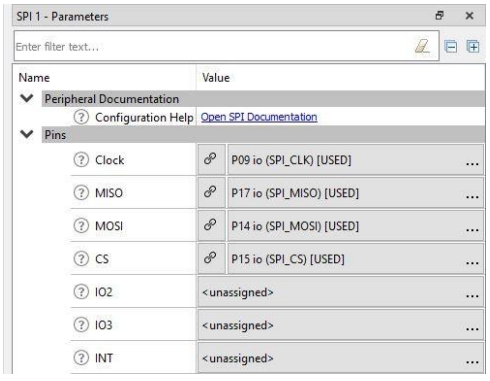
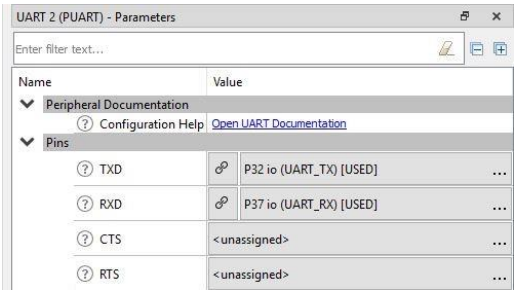


Figure 18. PUART Configuration



- **Pins:** The pins used in the application are enabled in the Pins section of the Device Configurator. Table 6 provides more information on these pins.

Table 6. Pin Mapping Details for Slave

Pin	Alias	Purpose
P32	UART_TX	Used for PUART transmit (Tx)
P37	UART_RX	Used for PUART receive (Rx)
P15	SPI_1_CS	Chip select for sensor
P14	SPI_1_MOSI	MOSI pin to sensor
P17	SPI_1_MISO	MISO pin to sensor
P09	SPI_1_CLK	Clock

Related Documents

Application Notes	
NA	
Code Examples	
Visit the Cypress GitHub site for a comprehensive collection of code examples using ModusToolbox IDE	
Device Documentation	
CYW20819 Device Datasheet	CYW20819 Product and Peripheral Guide
Development Kits	
CYW920819EVB-02 Evaluation Kit	
Tool Documentation	
ModusToolbox IDE	The Cypress IDE for IoT designers

Document History

Document Title: CE226537 – CYW20819 SPI-Based Datalogger

Document Number: 002-26537

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	6489982	SBKR	02/20/2019	New code example

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Code Examples | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2019. This document is the property of Cypress Semiconductor Corporation and its subsidiaries ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress shall have no liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. CYPRESS DOES NOT REPRESENT, WARRANT, OR GUARANTEE THAT CYPRESS PRODUCTS, OR SYSTEMS CREATED USING CYPRESS PRODUCTS, WILL BE FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (collectively, "Security Breach"). Cypress disclaims any liability relating to any Security Breach, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any Security Breach. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. "High-Risk Device" means any device or system whose failure could cause personal injury, death, or property damage. Examples of High-Risk Devices are weapons, nuclear installations, surgical implants, and other medical devices. "Critical Component" means any component of a High-Risk Device whose failure to perform can be reasonably expected to cause, directly or indirectly, the failure of the High-Risk Device, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any use of a Cypress product as a Critical Component in a High-Risk Device. You shall indemnify and hold Cypress, its directors, officers, employees, agents, affiliates, distributors, and assigns harmless from and against all claims, costs, damages, and expenses, arising out of any claim, including claims for product liability, personal injury or death, or property damage arising from any use of a Cypress product as a Critical Component in a High-Risk Device. Cypress products are not intended or authorized for use as a Critical Component in any High-Risk Device except to the limited extent that (i) Cypress's published data sheet for the product explicitly states Cypress has qualified the product for use in a specific High-Risk Device, or (ii) Cypress has given you advance written authorization to use the product as a Critical Component in the specific High-Risk Device and you have signed a separate indemnification agreement.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.