TI Designs

Bluetooth® Smart to RS-485 Gateway With Modbus Application Software

TI Designs

TI Designs provide the foundation that you need including methodology, testing and design files to quickly evaluate and customize the system. TI Designs help *you* accelerate your time to market.

Design Resources

TIDC-Bluetooth-Smart-to-RS-485-Gateway	Tool Folder Containing Design Files
CC2540T	Product Folder
SN65HVD72	Product Folder
TPS76933	Product Folder
CC Debugger	Product Folder

TI E2E™ Community

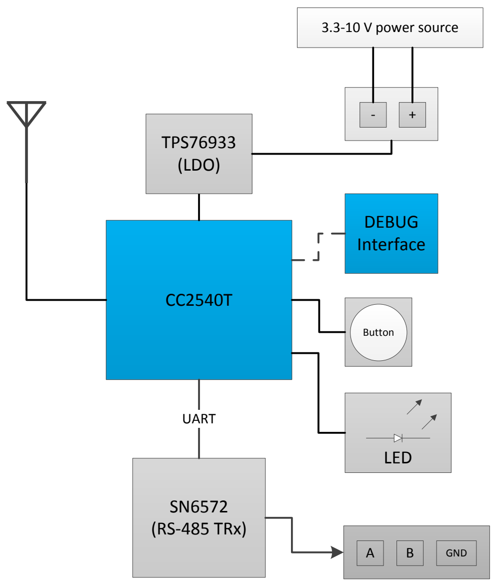
ASK Our E2E Experts
WEBENCH® Calculator Tools

Description

This design implements a gateway between Bluetooth Smart and RS-485 networks. The design includes software made for communicating with Modbus devices. A detailed user guide is included to simplify modifications and use.

Design Features

- Simple Interfacing With RS-485 Networks
- Simple Modification
- Ready to Use With Modbus Networks



All trademarks are the property of their respective owners.

1 Introduction

1.1 Solution Overview

This design implements a gateway between Modbus and Bluetooth Smart/Low Energy (BLE). The design serves as a replacement for wires in an RS-485 network and allows another BLE-compatible device like a computer or smartphone to easily connect to an existing RS-485 network.

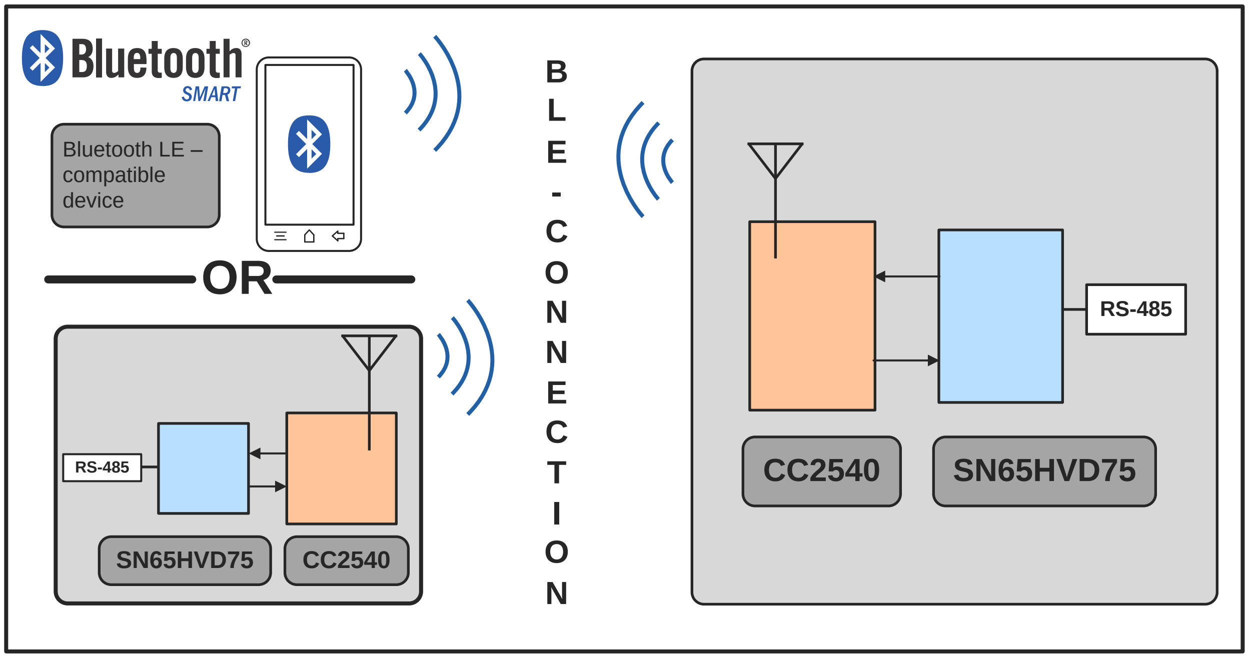


Figure 1. BLE Connection

The gateway consists of a small PCB module with a CC2540T wireless MCU and an SN65HVD72 RS-485 transceiver. Two different software applications are provided, one for the BLE peripheral role and one for the central role.

The PCB module can be connected to an RS-485 network, but a separate application layer software implementation is necessary to translate the incoming bytes to BLE. This design describes an implementation handling the Modbus application protocol.

1.2 RS-485

The RS-485 is a two-wire serial data transfer standard. The device is officially called TIA/EIA-485-A and is maintained by the Telecommunications Industry Association/Electric Industries Alliance (TIA/EIA). Data transmission is done by sending differential signals on a twisted pair of cables to make the line resistant against electromagnetic interference. Because the data is sent differentially, there is often tolerance for different ground levels for devices on the network. The RS-485 uses half-duplex communications, meaning only one device can send data at a time. This method means the transceiver needs control signals to enable sending and receiving. A typical transceiver has four connections to a node: the Driver, Receiver, Driver Enable (DE), and Receiver Enable (RE). The signal DE enables transmission. Usually, a higher layer specification makes sure this requirement is met.

1.3 Modbus

Modbus is an application protocol for serial data transmission that often uses the RS-485 for serial data transfer. The Modbus protocol is defined for different ways to transmit data, but the solution described in this design assumes transmission on an RS-485 serial bus in RTU mode. A detailed specification of Modbus over a serial line is found in *MODBUS over Serial Line Specification and Implementation Guide*[5].

Modbus is an open, royalty free protocol created by Modicon (now Schneider Electric). It is currently maintained by the Modbus Organization. More information can be found in the specification[4] and at the Modbus Organization's website, www.modbus.org.

Network Configuration: A Modbus network consists of a single master and one or more slaves. The slaves are silent until they are addressed by the master. All messages specified for a specific slave generate a response unless the parity or cyclic redundancy check (CRC) fails. Messages sent with recipient 0 are considered broadcast messages. The master makes sure only one device tries to communicate at the same time.

Message Transmission: Modbus supports a total message length of 256 bytes, which includes the recipient address field and error-detection bytes. Modbus messages are separated by at least 3.5-byte times of silence. This is the marker used by the connected devices to determine the start and end of messages. A Modbus message starts with a single byte address field, identifying which slave the message is intended for (or the slave from which the response originated). The next byte specifies the function code (See the application specifications[4] for a detailed overview of function codes). The rest of the message consists of the actual data transmitted in 16-bit segments. The last two bytes of the message are the Modbus-CRC word.

Byte Transmission: Bytes are transmitted as frames. A start bit indicates the beginning of a new frame and must be the opposite of the idle line. The following eight bits are the data bits. After the data bits, a parity bit may be appended, followed by a stop bit. The parity bit may be exchanged for a second stop bit or removed altogether. This versatility means that a data byte consists of 10 or 11 bits, depending on the parity and stop bit settings.

Some Software Considerations: Modbus supports message lengths up to 256 bytes, and the software implementation must be able to handle this. For BLE, it might not be possible to transmit an entire message in a single connection event. This transmission must be handled in software. Because the Modbus protocol uses timing to mark the end of a message, the application needs to keep track of this when handling messages. The Modbus protocol uses byte framing similar to that of UART.

More details on these considerations follow in Section 2.

1.4 Plug and Play, Getting Started

This section explains how to connect the PCB to an RS-485 network and how to set up a BLE link between two boards.

1.4.1 Connecting the PCB Module to an RS-485 Network

The PCB has two terminal connectors. One connects to the RS-485 transceiver and the other connects to the power supply. Connect the non-inverting RS-485 wire to the connector labeled with 'A', and the inverting wire to 'B'. If the network has a common ground wire, it can be connected to "GND", but this is often not necessary. The module needs an external power supply connected to the terminal pins labeled "VDD" and "GND". Input voltage can range from 3.3-V to 10-V DC. The module starts immediately once the power supply connects.

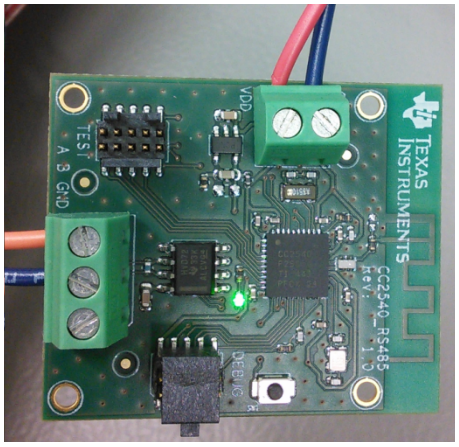


Figure 2. Connected PCB Module

1.4.2 Linking Two PCB Modules Connected to Physically Separate Modbus Networks

Two separate RS-485 networks can be connected using two PCB modules. One of the modules must run the ModbusPeripheral application, and the other must run ModbusCentral. The peripheral device starts advertising for a predefined amount of time when the button is pushed. Pushing the button on the central device starts device discovery, and the central device will automatically connect to an advertising device that implements the SerialProfile service. During discovery and advertising, the LED will blink quickly. When the device is idle, the LED will toggle more slowly, about once per second. Once the two devices have been connected, the LED will stop blinking and stay on. Push the button again on any connected device to terminate the connection, which makes the LED blink again. Figure 3 illustrates the button behavior.

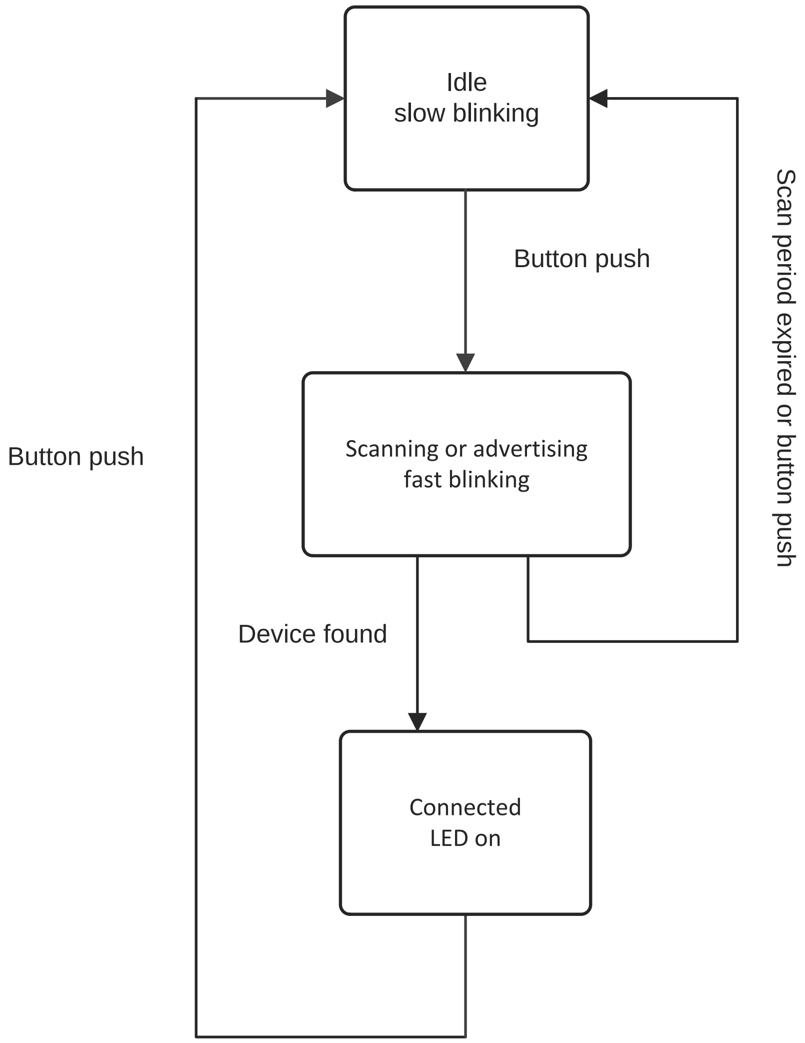


Figure 3. Button Behavior

NOTE: Peripheral and central behavior are separated by a backslash.

1.4.3 Loading New Software onto the Chip

The PCB module includes a header for easy connection to a CC Debugger. Connect the CC Debugger as shown in Figure 4 and the device is ready to be programmed. Make sure the chip is supplied with power. If connected correctly, the light on the CC Debugger will be green (you may have to press the reset button). Consult the *CC Debugger's User Guide* for additional help[6].

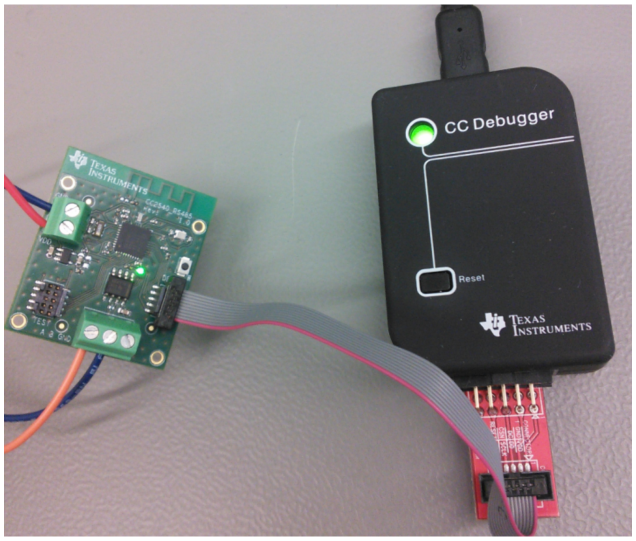


Figure 4. PCB Module With CC Debugger Connected and Ready to Download

1.5 Short BLE Introduction

This chapter introduces some of the aspects of BLE that are relevant to this application. Find more details about BLE in the software developer's guide[1] and the Bluetooth core specification[3].

Connection Events: BLE devices exchange information only in predetermined connection events; for a majority of the time, the connected devices have no RF activity, which is the main reason BLE allows low power consumption. A connection interval is the idle time between each connection event. A shorter connection interval allows faster communication, but it also leads to higher power consumption. Large connection intervals consume less power, but can lead to significant delays in data transmission.

Peripheral and Central Roles: The device initiating a BLE link will get the client role in the connection. In the software, this is called the central role or central device. The other device is the peripheral. The peripheral is the device hosting the GATT server where the characteristics are stored. The central device can request to read or write to characteristics on the server.

Characteristics: BLE devices transmit data through reading and writing GATT characteristics. A characteristic is a collection of attributes describing the same thing. Typical attributes are the characteristic value and a descriptor containing the user description. A service is a collection of characteristics. Services are stored on the GATT server, which is hosted on the peripheral device. The central device can access characteristics through write or read requests. A characteristic usually consist of several attributes with different permissions, and the central device can access attributes through write or read requests, according to the specified permissions. This means that only an attribute with write permission can be written by the GATT client (central). Characteristics can be associated with several descriptors, called the Characteristic User Description, which is a string containing a short user description. The Characteristic Client Configuration descriptor is another descriptor which is very relevant in this application. This is the descriptor used to enable notifications.

Notifications and Indications: The GATT client can request to be notified of a change in characteristic values by writing to the Characteristic Client Configuration descriptor. Writing a one to the least significant bit enables notifications, and writing a one to the second least significant bit enables indications. When enabled, notifications are sent by the GATT server whenever a characteristic value changes and do not require confirmation from the client. An indication does the same thing, but a confirmation is required before another indication can be sent. Notifications and indications are the only means for the GATT server (peripheral) to send data without being specifically requested to by the server.

2 Modbus Application Software Overview

This section introduces the example software and explains how to adapt and use the reference design in an existing Modbus network. This section also contains information on how to modify and test the software for different applications. An overview of the SerialProfile service is also provided. This service implements a two-way asynchronous data transfer and can also be used in other applications. Figure 5 shows the software architecture on a BLE device. The lower levels are included in the BLE stack and library files from Texas Instruments. The ModbusPeripheral and ModbusCentral software introduces an Application task, and a new GATT Profile is introduced. They also run different GAP Roles. All other layers remain untouched.

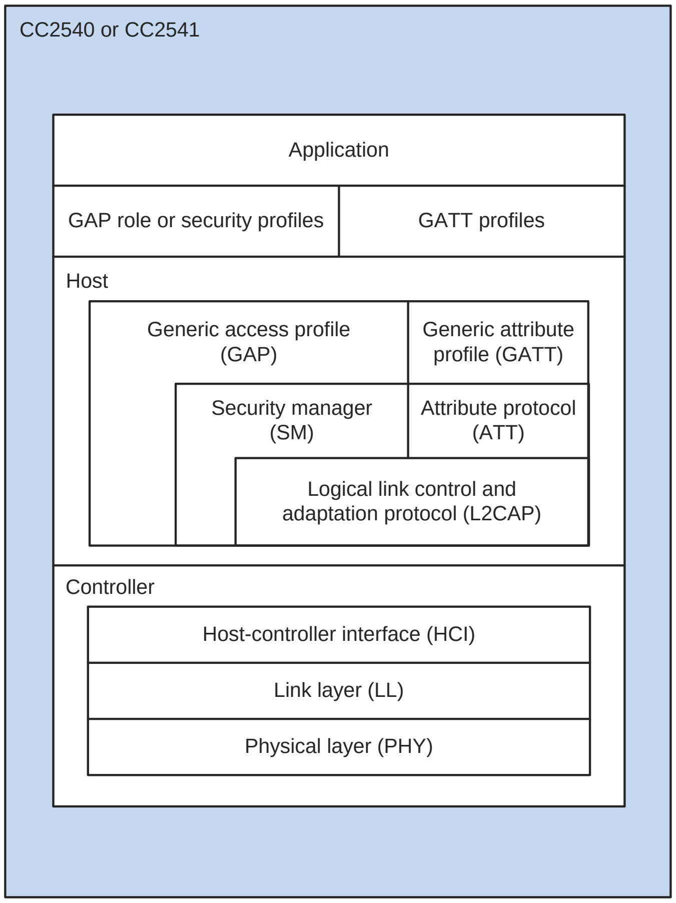

CC2540 or CC2541
Application
GAP role or security profiles
GATT profiles
Host
Generic access profile (GAP)
Generic attribute profile (GATT)
Security manager (SM)
Attribute protocol (ATT)
Logical link control and adaptation protocol (L2CAP)
Controller
Host-controller interface (HCI)
Link layer (LL)
Physical layer (PHY)

Figure 5. Software Architecture of BLE Device

2.1 SerialProfile

The SerialProfile service serves to simulate full-duplex two-way serial data transfer. It is designed to let the two connected devices be peers; both devices can initiate communication at any time. To achieve this, the data characteristics needs write permission and ability to send notifications. The peripheral device initiates communication through sending notifications, and the central device uses write commands. [(1)]

Characteristics: The SerialProfile has two characteristics: Data In and Data Out. Data In is the data flowing into the peripheral device, meaning Data In needs write permission to let the central device write new data. Data Out contains the data flowing out from the peripheral device. This characteristic uses notifications to let the peripheral device send new data without being asked to by the central device. The peripheral device should not change the Data In characteristic, and the central device should not modify Data Out.

(1) The profile allows full-duplex communication because it includes two Data-characteristics, one for each direction. This application uses half-duplex communication, but a two-characteristic solution is used for robustness and portability.

2.1.1 Software Implementation

The implementation of the SerialProfile is based on the simpleGATTProfile provided with the example projects from TI. The SerialProfile implementation is very similar to the simpleGATTProfile. Consult the software developer's guide[1] for more detailed information on the simpleGATTProfile. The functions have been modified to correctly interface with the characteristics, which involves changing the validity checks with respect to characteristic length and removing references to other characteristics than Data In and Data Out.

A brief walkthrough of the SerialProfile API functions:

- *SerialProfile_AddService*: Initialization function that registers the service attributes and callback functions with the GATT server
- *SerialProfile_RegisterAppCBs*: Initialization function informs SerialProfile which application functions should be notified of changes in characteristic values
- *SerialProfile_SetParameter*: The only valid argument for the first parameter "param" is SERIALPROFILE_DATA_OUT (0x06). If a different argument is passed, the function will return "INVALIDPARAMETER" (0x02) and do nothing. If the application writes to Data Out, this function will validate the data size, copy the new data, and attempt to send a notification to the central device. If the length is invalid, the function will return "bleInvalidRange" (0x18). If the write succeeds, the function returns "SUCCESS" (0x00)
- *SerialProfile_GetParameter*: API function that lets the application read a characteristic value. Data In is the only characteristic the application needs read access to, so if it tries to read any other characteristic the function will return "INVALIDPARAMETER" (0x02). When the application reads Data In, the current characteristic value is copied into the memory segment pointed to by value and made available to the application. In this case, the return value is "SUCCESS" (0x00)

An overview over the local functions used in SerialProfile:

- *serialProfile_ReadAttrCB*: This callback function is called whenever a GATT client tries to read a characteristic value.
- *serialProfile_WriteAttrCB*: This callback function is called whenever a GATT client tries to write a characteristic value. The function checks permissions as well as the length and offset of the message to make sure the write operation is valid. If it is, the write is carried out and the application is notified with the specified callback functions. This function also handles changes to the Client Characteristic Configuration descriptor.
- *serialProfile_HandlConnStatusCB*: This function is called when the link status is changed. When a link is terminated, this function resets the client characteristic configuration to disable notifications.

An overview of the two characteristics:

- *Data In*: Data is sent into the peripheral device from the BLE link. This data is transmitted serially from the peripheral. The value of this characteristic is an array of length "CHARACTERISTIC_LEN", which is defined in SerialProfile.h. By default this is "ATT_MTU_SIZE-3", because that is the largest number of bytes that can be written during a single connection event (using GATT_WriteNoRsp). This characteristic needs write permission to let the central device change it remotely. (2)
- *Data Out*: Data is sent out from the peripheral device on the BLE link. This data is transmitted serially from the central device. The value of this characteristic is also an array of length "CHARACTERISTIC_LEN". This characteristic should not be writable or readable; therefore, it does not need any permissions. The peripheral device communicates changes in the value by sending notifications, which is achieved by giving it the "GATT_PROP_NOTIFY" property (see the definition of serialProfileDataOutProps in SerialProfile.c). The property also needs a client characteristic configuration-descriptor for the central to request notifications.

(2) ATT_MTU_SIZE is defined in att.h as L2CAP_MTU_SIZE, which is defined in l2cap.h as 23.

2.1.2 Usage

Throughput: The SerialProfile simulates two-way data communication without synchronization, which means that no acknowledgment is needed from either side that a message is received. This communication is the reason notifications can be used instead of indications. Another upside of using notifications is that communication goes faster and several notifications can be sent on the same connection event. On the central side a similar approach should be used. If the usual write request-function "GATT_WriteCharValue" is used, the peripheral will have to transmit an "ATT_WRITE_RSP" on the next connection event, and the central will be idle while waiting for this response, which means that a write is only executed on every second connection event. To avoid this and achieve a higher throughput, use the sub-procedure "GATT_WriteNoRsp" to transfer data on every connection event because the client does not need to wait for the response before writing again. It is also possible to execute more than one write per connection event. The difference is illustrated in Figure 6.

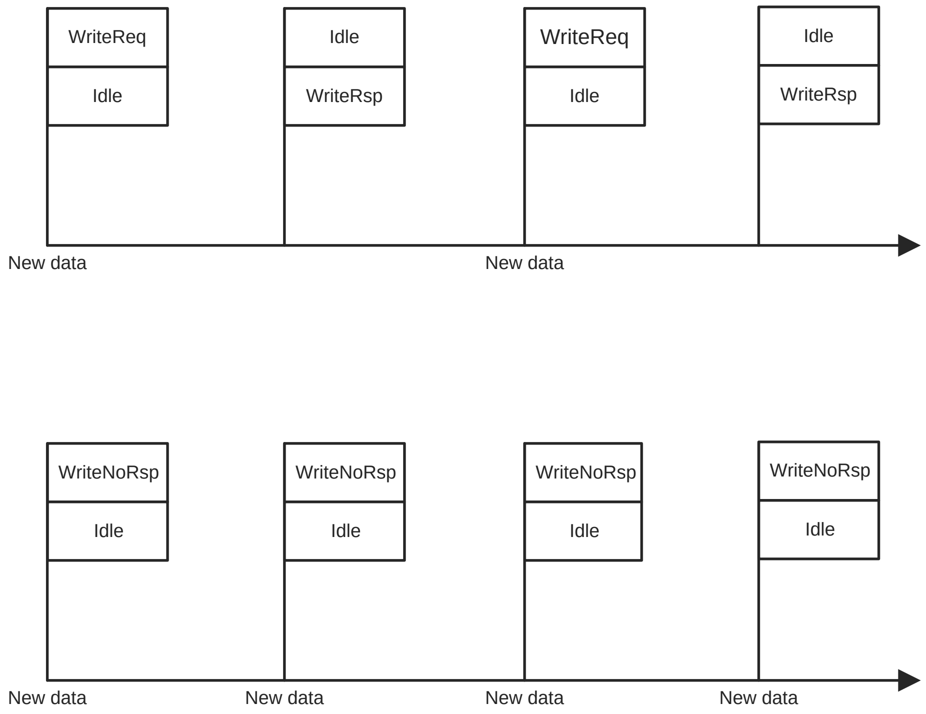


Figure 6. Transmitting Using Regular Write Command versus No Response Command

NOTE: Each vertical line denotes the start of a connection event.

Notifications: To simulate a peer-to-peer configuration, notifications have to be enabled before the data transmission can begin. If this is not done, the peripheral device is unable to transmit data. The central device cannot request data because none of the characteristics have read permissions.

Message Size: To transmit messages longer than ATT_MTU_SIZE-3 longer characteristics can be used, but they cannot be transmitted in entirety during a single connection interval. Furthermore, a notification will only send the first ATT_MTU_SIZE-3 bytes of the characteristic value, regardless of its size. Because of these (and other) reasons, long characteristics might not be a suitable way to transmit long messages. A better way will often be to split the message into smaller fragments that can be sent in one go. The fragments have to be rebuilt on the other side. The ModbusPeripheral and ModbusCentral projects show a way to achieve this. [3]

[3] A blob write requests require a blob write response. This introduces the problem discussed under throughput.

Modbus Application Considerations: The application design described in the rest of this document is an implementation of a Modbus interface that uses half-duplex communications over an RS-485 network. It would be sufficient to implement a half-duplex SerialProfile with only one shared characteristic. The two-characteristic solution is chosen for several reasons. One of them is to achieve robustness with regard to timing discrepancies. A single-characteristic solution is dependent on half-duplex operation, where both devices does not try to transmit at the same time. Using two characteristics, there is no risk that incoming messages interfere with the outgoing. Half-duplex operation is usually the responsibility of the application layer, but because of the potentially significant delays introduced by the BLE link (see Section 1.5) a two-way solution is chosen for robustness. Another reason is portability. The present implementation of the SerialProfile service allows for future applications to implement a full-duplex communication protocol without having to change the implementation of the service. Additionally, the cost of using a two-characteristic solution is small, especially when using small characteristic value sizes.

Identifiers: The SerialProfile is not an official Bluetooth profile and does not have official UUIDs, meaning that the user is free to choose UUIDs. The chosen UUIDs should not collide with official UUIDs. Table 1 shows the 16-bit UUIDs and other identifiers used in the SerialProfile by default.

Table 1. Identifiers Used by SerialProfile

DESCRIPTION	NAME	VALUE
SerialProfile service UUID	SERIALPROFILE_SERV_UUID	0xABCD
Data In UUID	SERIALPROFILE_DATA_IN_UUID	0xFFF6
Data Out UUID	SERIALPROFILE_DATA_OUT_UUID	0xFFF7
Data In identifier	SERIALPROFILE_DATA_IN	5
Data Out identifier	SERIALPROFILE_DATA_OUT	6

2.2 ModbusPeripheral and ModbusCentral

The main tasks of the ModbusPeripheral and ModbusCentral applications are to receive messages through UART and transmit them on the BLE link, and vice versa. These parts of the application are almost identical on the ModbusPeripheral and ModbusCentral, and will be presented in a separate chapter. The main difference between the two implementations is how they interface with the BLE link. This chapter will shortly introduce the differences between these two implementations. Find more information about how to build applications in the CC2540 software developer's guide[1].

Peripheral Role: The server side of a client-server configuration. This role is the only implementation needed to connect a BLE-compatible device, like a smart phone, to a Modbus network. The smart phone will then act as the central device. The ModbusPeripheral sends data using notifications. The API function SerialProfile_SetParameter from SerialProfile takes care of sending notifications when the application updates the Data Out characteristic value. The application is notified of newly written values to Data In by the callback function serialProfileChangeCB. To enable a device running ModbusCentral to connect, the peripheral needs to send connectable advertisements.

Central Role: The device that initiates a connection gets the central role in a BLE link. The ModbusCentral application scans for devices implementing the SerialProfile service, specified by its UUID. When such a device is found, the devices connect automatically. After discovering the handles for the two characteristics, notifications are enabled by writing 0x0001 to Data Out's Client Characteristic Configuration descriptor, which is always found at the handle following the characteristic it belongs to. The ModbusCentral application is notified of changes in Data Out by a GATT message with the method specifier ATT_HANDLE_VALUE_NOTI. This message is handled in modbusCentralProcessGATTMsg. To send new data to the peripheral, GATT_WriteNoRsp from gatt.h is used. This function is called from Modbus_WriteDataIn.

2.3 Modbus Serial Transfer Functions

This section covers the contents of modbusDataTransfer.c and modbusDataTransfer.h. The application is made to transfer Modbus messages over the air between two devices. As mentioned in Section 1.3, the Modbus protocol has certain demands that the software must meet. The most important demands in this application are the maximum supported message length, byte framing, and the timing demands. This section introduces a solution that meets these demands.

API Functions: There are some API functions used by the application to communicate with the serial interface. These functions are defined in modbusDataTransfer.c.

- *Modbus_initUART*: Configures the UART with the desired baud rate and parity settings, as well as sets up the pin used for the DE control signal on the RS-485 transceiver
- *Modbus_initTimer*: Sets up Timer1 according to the timing demands defined by the Modbus protocol and baud rate settings. This function also configures a DMA channel to automatically reset Timer1 when a new byte is received on the UART
- *RS485_initBoard*: Initialization function specific to the PCB module described in this document. This function initializes the I/O pins connected to peripherals on the board.
- *RS485_enableTest*: Enables a test pin used to configure new baud rates. This procedure will be discussed in Section 2.4
- *RS485_WriteSerial*: Sets the control signals to the RS-485 transceiver and writes the data to UART. This function also checks the validity of the data with respect to length
- *Modbus_GetFragment*: Tries to read a fragment of the specified length from the UART RX buffer. If the buffer contains fewer bytes than the desired amount, no bytes will be read. The retrieved bytes are returned in the data pointer

Message Fragments: A Modbus message can contain up to 256 bytes, but BLE is not able to transfer that much in one connection event. Long messages need to be fragmented and potentially transmitted over several connection events. For the receiving device to rebuild the original message, each fragment is tagged with the number of valid bytes in the fragment and a flag to signify if it is the last message or not. The most significant bit of the tag byte is reserved for the flag, meaning the total number of bytes contained in the fragment must not exceed 127, because this is the largest number that can be represented using seven bits. (It is assumed that the messages will arrive in the order they are sent). [1]

2.3.1 Serial Data Transfer With UART0

Motivation: The UART is a suitable option for serial I/O in this application because the Modbus protocol specifies the same framing schemes as UART.

UART Configuration: The UART module on CC2540 is configurable with many choices with regard to parity settings, start and stop bits, and baud rates. The desired settings are defined in the header file, modbusDataTransfer.h, and the program is designed to easily let the user change these settings to match the network the device will be connected to. Section 2.4 describes how to change the UART settings. The UART is configured and enabled in the public function Modbus_initUART. This function is called during the application's initialization routine. In this design, the DMA is used for RX and ISR used for TX, as recommended in hal_uart.c.

2.3.2 Timing with Timer1

Motivation: Modbus uses timing to mark the end of a message, 1.5 byte times of silence means a message is finished, and a new message cannot be sent until after 3.5 byte times of silence. At slow baud rates, 1.5 byte times is a significant amount of clock cycles, even with high clock-prescaling values. Timer1 is a suitable choice, as it is the only 16-bit timer available on CC2540.

Timer Configuration: The timer is configured in the public function Modbus_initTimer. This function is called during the application's initialization routine. This function also sets up a DMA channel to clear the timer every time a byte is received on UART0. This way the Timer1 count registers T1CNTL and T1CNTH always contain the number of ticks since the last byte was received. When the count reaches the predefined value, defined in MODBUS_RX_TIMEOUT, the appropriate amount of time has passed, and the message can be assumed to be finished. The ISR for Timer1 notifies the application of the finished message.

[1] The fragment length is less than 127 in this application because ATT_MTU_SIZE is defined as 23.

2.3.3 Receiving a Modbus Message on UART and Sending on BLE

This section describes what happens when a Modbus message is received on the UART of the CC2540, from the first byte is received until the complete message is transmitted over the air to the linked device. Message fragments are continually being transmitted over the air as they come in. Figure 7 shows the process of receiving a message serially.

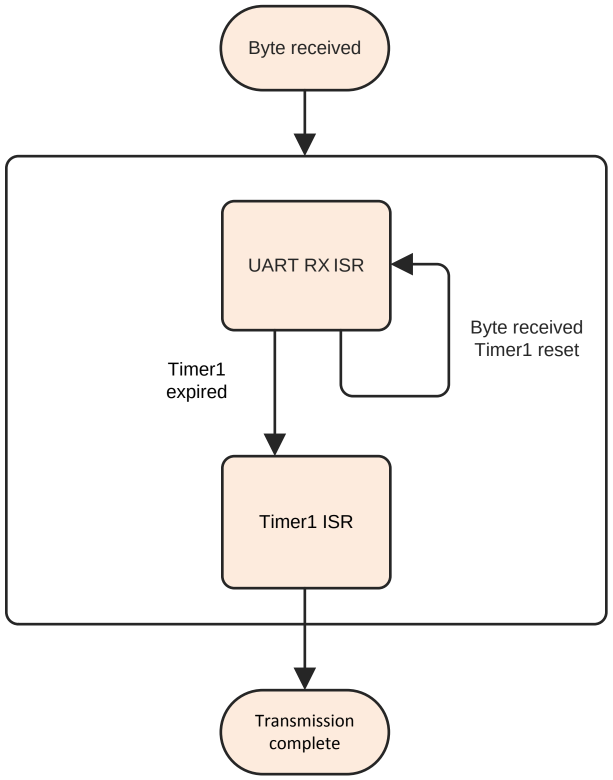


Figure 7. UART RX Process

Receiving Bytes: When a byte is received on the UART, two things happen: the DMA clears the Timer1 counter registers, and the UART RX ISR starts. The ISR is responsible for starting the timer and incrementing the byte counter, which keeps track of the total number of bytes received. If CHARACTERISTIC_LEN bytes have been received, the ISR also takes care of sending CHARACTERISTIC_LEN-1 bytes over the air. This method sends the message in fragments while it is received. This lowers the total delay through the link, as opposed to waiting for the entire message to arrive before sending it.

Message Received: As long as new bytes are arriving on the UART, the DMA keeps clearing the Timer1 count registers. Once the specified time has passed since the last byte arrived, Timer1 reaches its goal, and the Timer1 ISR runs, which means the entire message has arrived. After stopping Timer1, the ISR transmits the remaining bytes in fragments of CHARACTERISTIC_LEN-1 bytes. The last fragment is marked as the final fragment by letting the most significant bit in the first byte be zero.

Transmitting Fragments on BLE: Fragments are transmitted over the air using different methods on ModbusPeripheral and ModbusCentral. ModbusPeripheral simply writes the new characteristic value to the GATT server using SerialProfile_SetParameter, which also takes care of sending a notification to the central. ModbusCentral uses the API function Modbus_WriteDataIn. The two different functions take (almost) the same arguments and are called at the same spot in the code. In both implementations, sending a fragment is initiated by setting an operating system abstraction layer (OSAL) event. Read more about OSAL in the software developer's guide[1].

2.3.4 Sending a Modbus Message on UART and Receiving on BLE

Because of the timing requirements in Modbus, the entire message must transmit at the same time. Store load fragments in a separate buffer and wait until the final fragment has arrived rather than loading them into the UART buffer as they arrive. Figure 8 shows the process of receiving a message over the air.

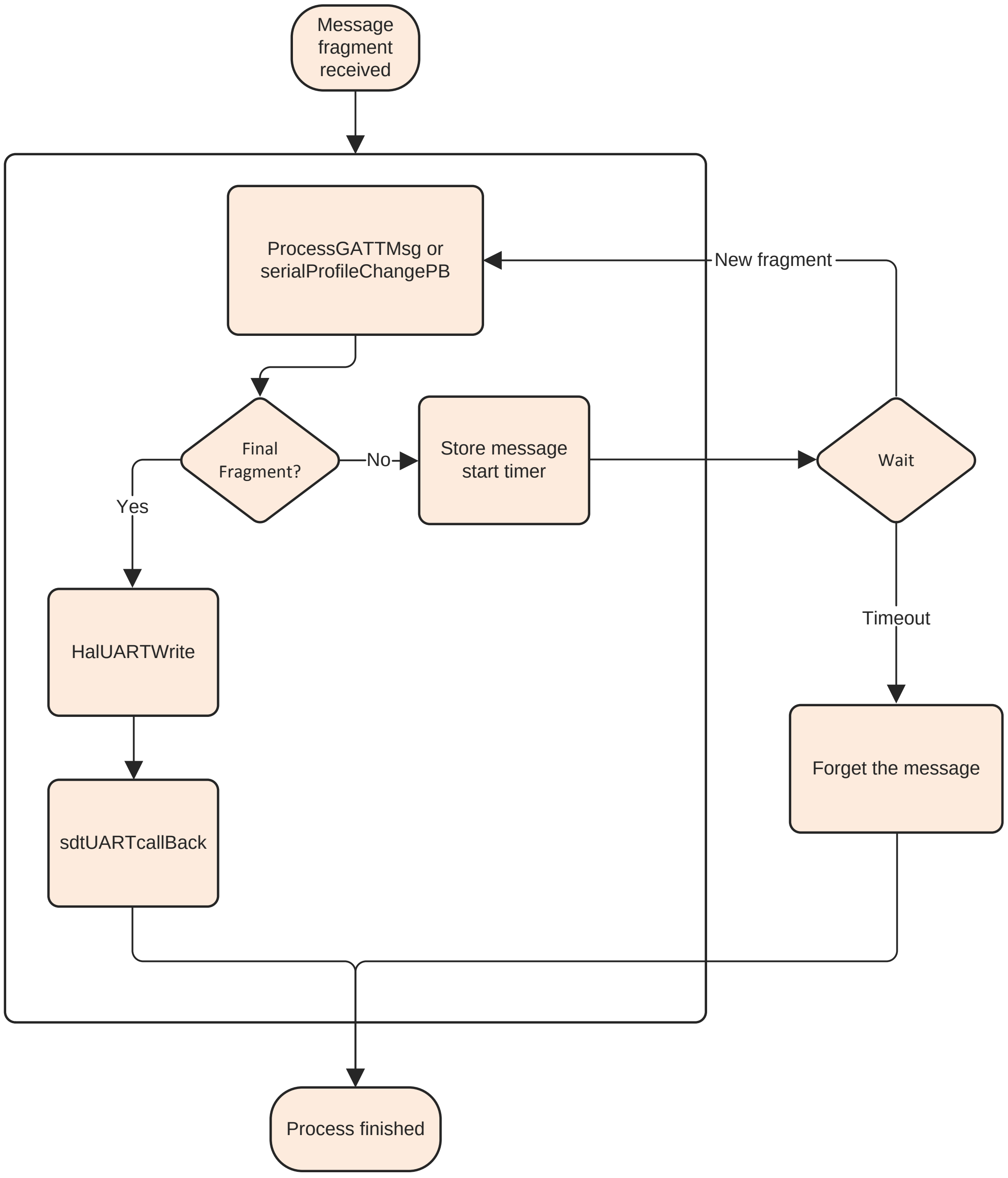


Figure 8. UART TX Process

Receive New BLE Data: Because fragments are transmitted differently on ModbusPeripheral and ModbusCentral, they are also received differently. The code looks the same for both cases but is placed in different functions. For ModbusPeripheral, new characteristic values are received by the application in the serialProfileChangeCB callback. ModbusCentral handles notifications in modbusCentralHandleGATTMsg when the method field of the message is ATT_HANDLE_VALUE_NOTI. [2]

[2] Fragments are sent in the same format in both implementations, but using different GATT commands.

Handle New Message Fragments: In both implementations, fragments are stored in a global array called modbusMessage. When a new fragment is added, modbusCurrentLength is updated with the new length. If the fragment tag marks it as the final fragment, the contents of modbusMessage is copied into the UART TX buffer and transmitted to the connected network using Modbus_WriteSerial. This function writes the message to the UART and sets the DE control signal, in addition to disabling UART RX since only half-duplex communication is supported. After the message has been written, modbusCurrentLength is set to zero. If there are fragments remaining, an MBP_FRAGMENT_TIMEOUT_EVT is scheduled using osal_startTimerEx. The fragment timeout period must be long enough to allow for missed connection events. When a new fragment is received, this event is removed by osal_stopTimerEx. If the specified time is allowed to pass (MBP_FRAGMENT_TIMEOUT_PERIOD), the message is assumed to be invalid and it is forgotten by setting modbusCurrentLength to zero.

End of Transmission: When the UART TX buffer is empty, the code in the UART callback function modbusUARTcallback is run. This function clears the DE signal and re-enables UART RX once the transmission is complete. Because this call only means the buffer is empty, the byte may not yet be finished transmitting on the UART when the function starts running. The while loop keeps polling the UART status register (U0CSR) until the ACTIVE-bit is 0 to indicate that the UART is idle. Once this happens, the DE signal is cleared, and UART RX is re-enabled. (3)

2.4 Modification

This section explains how to modify the software to fit the serial network it is connected to, including changing the baud rate, changing parity settings, and meeting timing demands.

Baud Rate and Parity: By default, five different baud rates are supported by hal_uart: 9600, 19200, 38400, 57600, and 115200 bits per second. To modify the software to use one of these, simply change the defined value of SDT_UART_BR to HAL_UART_BR_XXXX, where XXXX is the desired baud rate, and download the program. (4)

If other baud rates are desired, they need to be included in _hal_uart_dma.c. This file is a library file, so changing something in this file applies the change for all other projects using this file. Specifically, the values for the UxBAUD_M and UxBAUD_E must be set for the desired baud rate. The procedure for finding the values of these registers is found in the CC2540 user guide’s USART section[2].

Because Modbus relates its timing demands to the time a byte takes to transmit, which is inversely related to the baud rate, the application needs to take this process into account when changing baud rates. The timing is controlled by ISRs, introduces delays between when an event should have occurred and when it actually occurs. These delays depend amongst other things on other running ISRs, and how busy the CPU is (for example, scheduled events). The simplest way to adjust the timing is using a logic analyzer and control bits on one of the I/O pins.

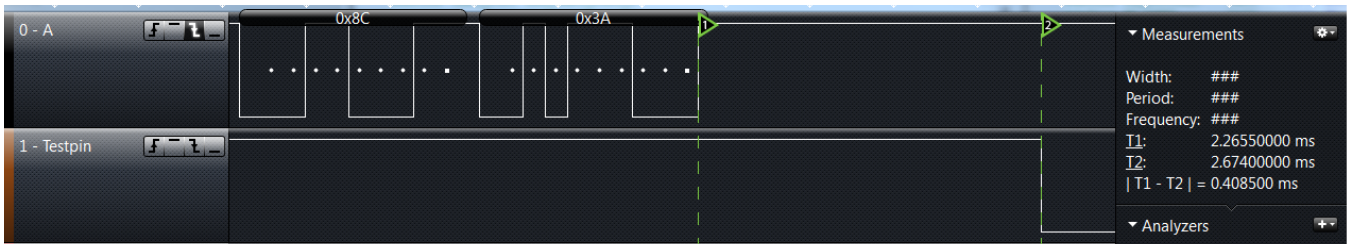
The timing constant that must be defined when changing baud rates is MODBUS_RX_TIMEOUT, which is used as the compare value for Timer1. This value must correspond to a duration of between 1.5 and 3.5 bytes to comply with the Modbus protocol. Because so much is happening on the chip, it is difficult to precisely predict when ISRs will run, so calculating the theoretical number of ticks for Timer1 will often not work.

Using the method described below, values have been found for the supported baud rates. Based on these values, the timeout value should be approximately 23 µs/b. The easiest way to determine the timeout value is by testing. To facilitate this, on startup call a function named RS485_enableTesting, which sets up a test pin to measuring. Change the output on this pin using the macros SET_TEST and CLEAR_TEST. The pin should be set at the beginning of uartRxIsr and cleared at the beginning of timer1isr. These settings make it is possible to see when the ISR symbolizing the end of a message is actually run. Use a logic analyzer to verify that the waveform of the test pin high for the entire duration of the message and that it is cleared no earlier than 1.5-byte times and no later than 3.5-byte times after the message is over. This verification is illustrated in Figure 9 and Figure 10.

(3) modbusUARTcallback is actually run every time an UART event is complete, but the if-statement discards all other events.

(4) When using high baud rates, the application may be unable to keep up with the messages. Longer silences than the specified 3.5-byte times are necessary in these cases.

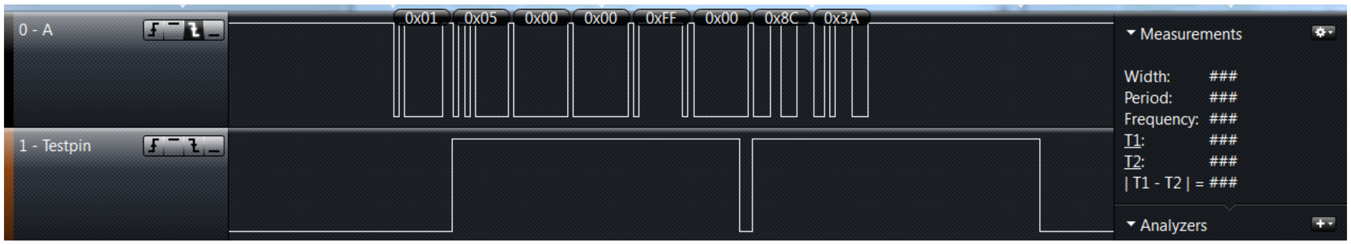
For Figure 9, the timeout value is set so that at least 1.5 byte time elapses. The baud rate is 38400, which corresponds to a duration of approximately 0.39 ms for 1.5 bytes. The measured time is 0.40 ms. A larger delay would also be acceptable.



Width: ###
Period: ###
Frequency: ###
T1: 2.26550000 ms
T2: 2.67400000 ms
| T1 - T2 | = 0.408500 ms

Figure 9. Logic Analyzer Plot of Data Transfer at 38400 bps With Long Timeout

In Figure 10, the timeout value is too short. The timer expires before the message is finished, which creates transitions in the test pin. These transitions split up the message before transmitted over the air.



Width: ###
Period: ###
Frequency: ###
T1: ###
T2: ###
| T1 - T2 | = ###

Figure 10. Logic Analyzer Plot of Data Transfer at 38400 bps With Short Timeout

The logic analyzer is connected to P0_6 on the test header and either the A or B output pin on the RS-485 transceiver to measure MODBUS_RX_TIMEOUT, as shown in Figure 11.

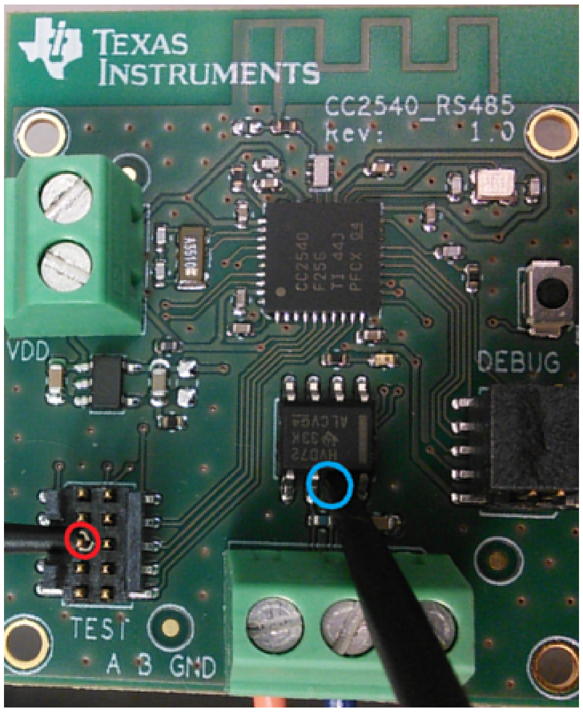


Figure 11. Logic Analyzer Connected to P0_6 (Red) and Either A or B (Blue)

Table 2 shows the baud rates available by default and their corresponding values for MODBUS_RX_TIMEOUT.

Table 2. Supported Baud Rates With Corresponding Values for MODBUS_RX_TIMEOUT

BAUD RATE (bps)	MODBUS_RX_TIMEOUT
9600	0x2000
19200	0x1000
38400	0x0650
57600	0x0450
115200	0x0228

BLE Link Settings: The delay through the BLE link is very dependent on the length of the connection interval. This interval can be set by changing the value of the defined DEFAULT_DESIRED_MIN_CONN_INTERVAL and DEFAULT_DESIRED_MAX_CONN_INTERVAL in modbusPeripheral.c. The central can request an update of the connection parameters within the bounds specified by the peripheral, but this is not implemented this design. Specifying shorter connection intervals gives shorter delay and higher power consumption.

Other Protocols: This software is designed for Modbus, which uses the same byte framing as the UART module. Other protocols that use a similar framing may be implemented by changing the requirements for when a message starts and ends (and maybe other parameters). This depends on the protocol and might not even require the timer. RS-485 networks that implement protocols using smaller message sizes than 256 bytes, and some delay between messages may be able to use the ModbusPeripheral and ModbusCentral software without modification.

Protocols that do not use UART framing for byte transfers cannot use the UART for serial transfer; however, bit banging might be a viable option. The paths on the PCB connect the Driver input on the RS-485 transceiver to P0_3 and the Receiver input is connected to P0_2. These pins must be used as the serial I/O pins for the module to work. According to the CC2540 user's guide[2], these pins also correspond to Timer1 channel 0 and 1 when configured as peripheral I/O pins.

2.5 Special Considerations

Long Messages: By default hal_uart.c does not support writes to the buffer longer than 255 bytes. The reason for this is that many of the index variables are uint8. To enable a message length of 256 bytes, change the types of all the index variables in _hal_uart_dma.c to uint16. Because _hal_uart_dma.c is a library file, any changes to this file will be effective for all other projects using it. There may also an issue with memory on the ModbusCentral software. This issue is fixed by defining a smaller INT_HEAP_LEN in the preprocessor defines (right-click the on the project in the IAR’s file explorer and choose *Options*, then select *C/C++ Compiler* and change the defined value listed under *Defined Symbols*). The maximum supported message length for the project is the same as the maximum size of the UART buffers, defined as HAL_UART_DMA_TX_MAX, and should also be listed under *Defined Symbols*.

Power Saving: This application must be ready to receive messages from the UART at all times, and keep track of the timing. This means that POWER_SAVING cannot be defined for the project, as this will disable operation of the UART during sleep.

3 Hardware Overview

3.1 Components

The PCB module is made from the schematic included as a PDF. An important part of the circuit is the RS-485 transceiver. This design uses the SN65HVD72 from Texas Instruments, which uses the same supply voltage as the CC2540 (3.3 V) and is capable of switching speeds up to 250 kbps. The design includes two headers to access I/O pins on the CC2540. The header marked with DEBUG can be connected to a CC Debugger as described in Section 1.4. A linear voltage regulator (TPS76933 from Texas Instruments) is used to supply the circuit with 3.3-V DC. The regulator accepts input voltages up to 10-V DC.

4 Problems and FAQs

This section covers solutions to common problems.

Problem: The Modbus network connected to the PCB module does not respond.
Solution: Check if the message is actually transmitted serially from the PCB module by connecting a logic analyzer to the A or B output from the RS-485 transceiver (see Figure 11). If the message is transmitted from the RS-485 transceiver, verify that the baud rate and parity settings are the same for all the devices on the network. The settings defined for the software running on the PCB module need to correspond to the settings used on the network.

Problem: The two PCB modules cannot connect.
Solution: Make sure the peripheral device is advertising when the central is scanning. Push the buttons on the two devices at approximately the same time to be sure. The LED will blink during scanning or advertising. If this does not work, verify that ModbusPeripheral is running on one device and ModbusCentral is running on the other. Two devices running the same role are not able to connect. If the software has not been modified, the LED on ModbusPeripheral will blink faster than on ModbusCentral.

Problem: The message does not get through the RS-485 transceiver or is cut short.
Solution: Check that the message is actually transmitted on the UART TX pin. If it is, verify the presence and timing of the DE-signal. Check that the DE signal is high for the duration of the message by connecting a logic analyzer to the DE or /RE pins (which are connected to each other) and the Driver pin on the RS-485 transceiver.

Problem: The message coming out from the PCB module (serially) is split into smaller fragments.
Solution: The timer controlling the silence time reaches its end too soon. Change the value of MODBUS_RX_TIMEOUT. See the modification section for more details.

Problem: The application does not receive anything on the UART module.
Solution: Check the preprocessor settings by right clicking the project in IAR and selecting Settings. In the window that appears, select the *C/C++ Compiler* and the *Preprocessor* tab. In the list of defined symbols, make sure that POWER_SAVING is not defined, HAL_UART = TRUE, and HAL_DMA = TRUE. These settings are necessary for the DMA to be able to control the UART as desired.

Problem: The application starts running, but it appears to halt during initialization. The application also cannot connect. Some compiler warnings about conditions always being true or false appear.
Solution: If HAL_UART_DMA_TX_MAX is larger than 255, problems arise in the UART driver because of overflows. Change all indexes in _hal_uart_dma.c to uint16 instead of uint8, or use a smaller message length. Changes in _hal_uart_dma.c affect all other projects using this library file.

5 Abbreviations

BLE	Bluetooth Low Energy
GATT	Generic Attribute Profile
UART	Universal Asynchronous Receiver/Transmitter
RX	Receive
TX	Transmit
UUID	Universally Unique Identifier
ISR	Interrupt Service Routine
DMA	Direct Memory Access

6 References

1. Texas Instruments, *CC2540/41 Bluetooth Low Energy Software Developer's Guide v1.3.2* (SWRU271F)
2. Texas Instruments, *CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth Low Energy Applications User's Guide* (SWRU191)
3. Bluetooth Special Interest Group, *Bluetooth Core Specification 4.2* (https://www.bluetooth.org/en-us/specification/adopted-specifications)
4. Modbus, *Modbus Application Protocol Specification* (http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf)
5. Modbus, *MODBUS over Serial Line Specification and Implementation Guide* (http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf)
6. Texas Instruments, *CC Debugger User's Guide* (SWRU197)