Application Report
SPMA037–November 2011

MODBUS Bridge Using Stellaris

S M Narayanan

ABSTRACT

This application report describes the implementation of the MODBUS bridge using Stellaris® microcontrollers. The idea is to add the functionality of converting MODBUS TCP packets into MODBUS Serial packets on the MDL-S2E, which can be used to interface existing MODBUS Serial devices to Ethernet.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spma037.

Contents

List of Figures

List of Tables

1 Introduction

Modbus is serial communication protocol used in the industrial automation process. Due to advancement in technology, Modbus TCP/IP is used to provide more robustness and handle more Modbus clients. This application report enables customers to upgrade the existing legacy Modbus Serial devices to support Modbus TCP. By using this bridge, an IP can be assigned to an existing Modbus client. Modbus TCP/IP uses Modbus application protocol (MBAP) in the application layer.

2 Block Diagram

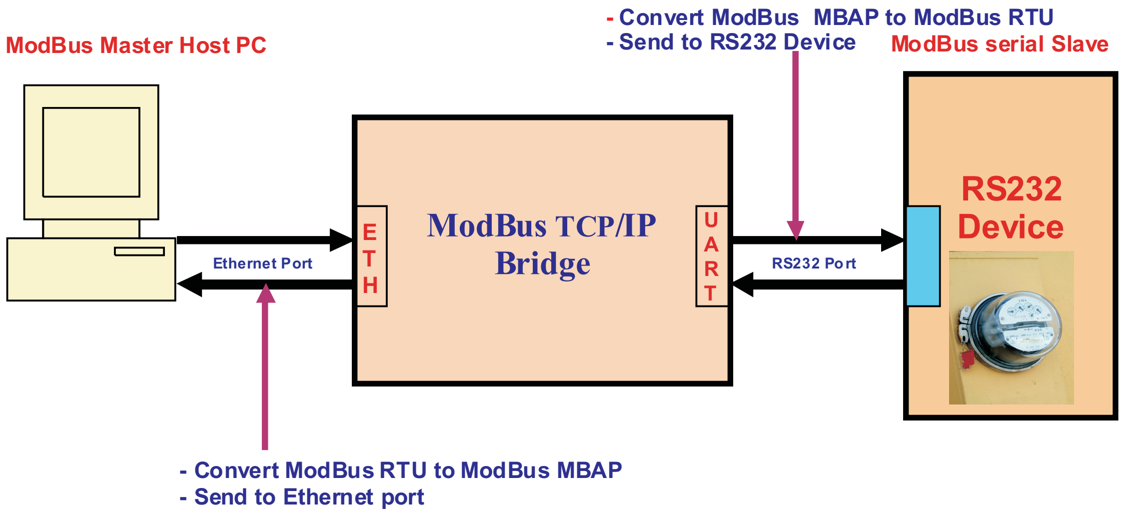

Figure 1. Stellaris Serial to Ethernet Block Diagram

3 Existing Serial to Ethernet Module

TI provides the Serial to Ethernet module (shown in Figure 2), which can be directly interfaced with the existing UART/RS232 devices to provide Ethernet connectivity. The S2E module supports two types of data flow by using TELNET mode and RAW mode. In case of the Telnet-based protocol implementation, TELNET data is used. This application uses lwip stack for Transport and Internet layer implementations. This Serial to Ethernet module is based on the Stellaris LM3S6432 device.

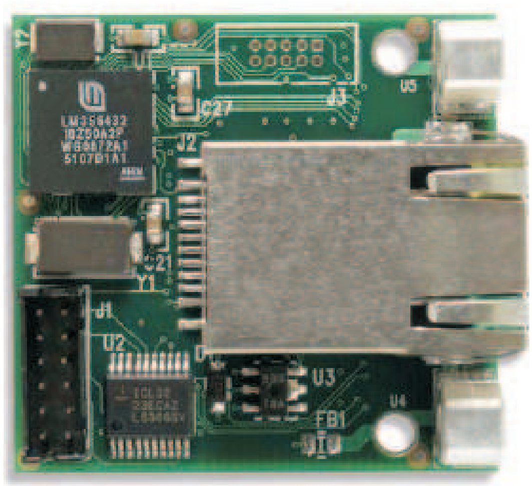

Figure 2. Stellaris Serial to Ethernet Module

4 Modbus RTU Frame

In the Modbus RTU frame, *Address* bits indicate that the Slave ID of the Modbus Serial device needs to be polled. The *Function* bits indicate whether it is Read or Write. This is followed by 2 bytes of OffsetAddress of the register that needs to be polled, which will be represented as Offset Address -1. For example, if you need to fetch the data from 0x5 location, it will be represented as 00 04 in the Modbus RTU frame. This is followed by 2 bytes of length of data that needs to read from the location. This is followed by 2 bytes of CRC of the whole frame.

Example of Modbus RTU Frame:

Address 0x1, Function 0x3 (Read Registers), Address of Register 0x5, Length 0x3

01 03 00 04 00 03 44 0A

Table 1. Modbus RTU Frame

Modus RTU Frame Format		
Name	**Length**	**Function**
Start	3.5c idle	At least 3-1/2 character times of silence (MARK condition)
Address	8 bits	Station address
Function	8 bits	Indicates the function codes like read coils/inputs
Data	n * 8 bits	Data + length will be filled depending on the message type
CRC Check	16 bits	Error checks
End	3.5c idle	At least 3-1/2 character times of silence between frames

5 Modbus TCP Frame

The Modbus TCP frame consist of 7 byte MBAP Header followed by the *Function Code* and *Data Bytes*. In the MBAP header, the first 2 bytes are known as the *Transaction Identifier*, which will be the same in the request and response. This is followed by 2 bytes of *Protocol Identifier*, which will be 0x0 for Modbus TCP. The *Length Field* is 2 bytes. The 1 byte *Unit Identifier* will provide the slave ID of the Modbus device. The function bits tell whether this is Read or Write. This is followed by application data unit (ADU), which contains the data.

Table 2. Modbus TCP Frame

Modus TCP Frame Format		
Name	**Length**	**Function**
Transaction Identifier	2 bytes	For synchronization between messages of server and client
Protocol Identifier	2 bytes	Zero for MODBUS/TCP
Length Field	2 bytes	Number of remaining bytes in this frame
Unit Identifier	1 byte	Slave Address (255 if not used)
Function Code	1 byte	Function codes as in other variants
Data Bytes	n byte	Data as response or commands

6 Implementation in Software

6.1 Modbus TCP to Modbus RTU

While receiving data from TCP/IP, Modbus TCP parser, you can parse the MBAP Header and extract the *Transaction Identfier*, *Protocol Identifier*, and ADU. You can also validate the Modbus TCP packet based on the *Protocol Identfier* field. If the packet is not Modbus TCP, then this packet will be rejected. You can also validate the *Slave Address* if they are in known range in your RS-485 network.

After receiving the whole Modbus TCP frame, the first 6 bytes need to be removed and CRC is calculated for the rest of the data packet. The calculated CRC is appended as the last 2 bytes. This frame is sent through the UART to Modbus RTU device.

6.2 Modbus RTU to Modbus TCP

While receiving data from the Serial side, the Modbus RTU parser module receives the whole packet byte-by-byte. Modbus RTU statemachine ensures and validates the Slave ID and Function ID. Once it receives the whole frame, this is written into the Ringbuffer. While writing into the Ringbuffer, the 2 bytes of CRC are ignored. The six bytes of data in MBAP header is appended to the frame. During the receiving of the bytes, idle gap should not be greater than 3.5 char. If the gap is more than 3.5 char, it is considered end of file (EOF). Finally, the constructed Modbus TCP frame will be sent across the TCP/IP network.

7 New Software Modules Added for Modbus Support

7.1 ModbusTCP Module (ModbusTCP.c)

This software module is responsible for maintaining states of Modbus TCP statemachine, receiving of Modbus TCP packets, converting into Modbus RTU packet and sending it on Serial side.

Below are the various states in the Modbus statemachine:

- MODBUS_STATE_INIT, // Initial State of ModbusTCP statemachine
- MODBUS_STATE_FRAME_START, // Start to receive the Modbus TCP frame
- MODBUS_STATE_TRANSACTION_IDENTIFIER_1, // To Receive Transaction ID Byte1
- MODBUS_STATE_TRANSACTION_IDENTIFIER_2, // To Receive Transaction ID Byte2
- MODBUS_STATE_PROTOCOL_IDENTIFIER_1, // To Receive Protocol ID Byte1
- MODBUS_STATE_PROTOCOL_IDENTIFIER_1, // To Receive Protocol ID Byte1
- MODBUS_STATE_FRAME_LENGTH_1, // To Receive length Byte 1
- MODBUS_STATE_FRAME_LENGTH_2, // To Receive length Byte 2
- MODBUS_STATE_SLAVE_ID, // To get the Slave Address ID
- MODBUS_STATE_FUNCTION_ID, // To get functional ID
- MODBUS_STATE_ADU_DATA, // To receive ADC packet
- MODBUS_STATE_CRC_CALC, // Calculating CRC for Frame
- MODBUS_STATE_SEND_SERIAL, // Send the frame to serial side
- MODBUS_STATE_WAIT_FOR_CLIENT_RESPONSE, // Waiting for response from Slave
- MODBUS_STATE_INVALID // Invalid state

Functions:

- `InitModbusTCPStatemachine()`

  This function initializes the whole statemachine. This resets all the global variables to its default value and also initializes the ADU buffer with all zeros.
- `ResetModbusTCPStatemachine()`

  This function sets the modbusTCP statemachine to its initial state MODBUS_STATE_INIT.
- `CheckModbusTCPStatemachine()`

  This function checks whether or not the Modbus TCP statemachine is waiting for a response from the Modbus RTU. This is used while arming and disarming timers.
- `ModbusCRC()`

  This function generates the calculate and returns CRC for the Modbus RTU payload.
- `ModbusTCPReceiveSerialSend()`

  This is the main function of this module. This is called from `TelnetProcessCharacter()` in the *Telnet.c*. It handles all the statemachine, receives the data and also processes the same. It converts the Modbus TCP to Modbus RTU frame and sends it across serially.

7.2 ModbusRTU Module (ModbusRTU.c)

Below are the various states of the RTU statemachines:

- MODBUS_RTU_STATE_IDLE // Idle state of Modbus RTU statemachine
- MODBUS_RTU_STATE_INIT, // Initial state of Modbus RTU statemachine
- MODBUS_RTU_STATE_FRAME_START, // Start of RTU Frame
- MODBUS_RTU_STATE_SLAVE_ID, // Wait for Slave Address
- MODBUS_RTU_STATE_FUNC_ID, // To receive for Functional ID
- MODBUS_RTU_STATE_FRAME_LENGTH, // To get the Frame Length bytes
- MODBUS_RTU_STATE_FRAME_CRC1, // To get the CRC Byte 1

- MODBUS_RTU_STATE_FRAME_CRC2, // To get the CRC Byte 2
- MODBUS_RTU_STATE_FRAME_SENT, // Sent the ModbusTCP frame into Ringbuffer
- MODBUS_RTU_STATE_INVALID // Invalid Error state

Functions:

- `initModbusRTUStatemachine()`

  This function initializes the whole Modbus STU statemachine. It resets all the global variables to its default value and also initializes the RTU buffer with all zeros and also appends the 6 bytes from the Modbus TCP request from the Modbus server.
- `ResetModbusRTUStatemachine()`

  This function re-initializes the Modbus RTU statemachine.
- `SetInitModbusRTUStatemachine()`

  This function sets RTU statemachine to MODBUS_RTU_STATE_INIT.
- `ModbusRTURecvTCPSend()`

  This function is called from the Serial interrupt handler whenever a character is received from UART. It receives Modbus RTU frames and validates the same. Then, it constructs the Modbus TCP frame and sends it across to Ethernet.
- `ModbusTimerInit()`

  This function initializes the Modbus timer.
- `ModbusTimerArm()`

  This function enables the timer. This is done after processing every character and waiting for new character, so that End of Frame and Start of Frame can be detected.
- `ModbusTimerDisArm()`

  This function disables the timer. This is done after receiving every character in UART interrupt handler so that End of Frame and Start of Frame can be detected.

8 Changes in Existing Software

The following changes are done in the *config.c* file.

- Default port of the S2E module is configured to Modbus Standard port 502 and is configured to use RAW protocol transfer.
- Default config for UART0 is set to 9600 bps, even parity.
- By default, S2E module supports two UART ports. Since memory is required, UART1 is disabled.
- The httpserver and CGI modules are disabled.

9 How to Use

Below are the steps to use the module:

1. Connect your MODBUS Serial enabled device to the DB9 connector in the S2E-RDK.
2. Program the S2E-RDK with *ModbusBridge.bin* in the *\boards\rdk-s2e\ModbusBridge\rvmdk* directory, using the LMFlash programmer utility (see Figure 3).

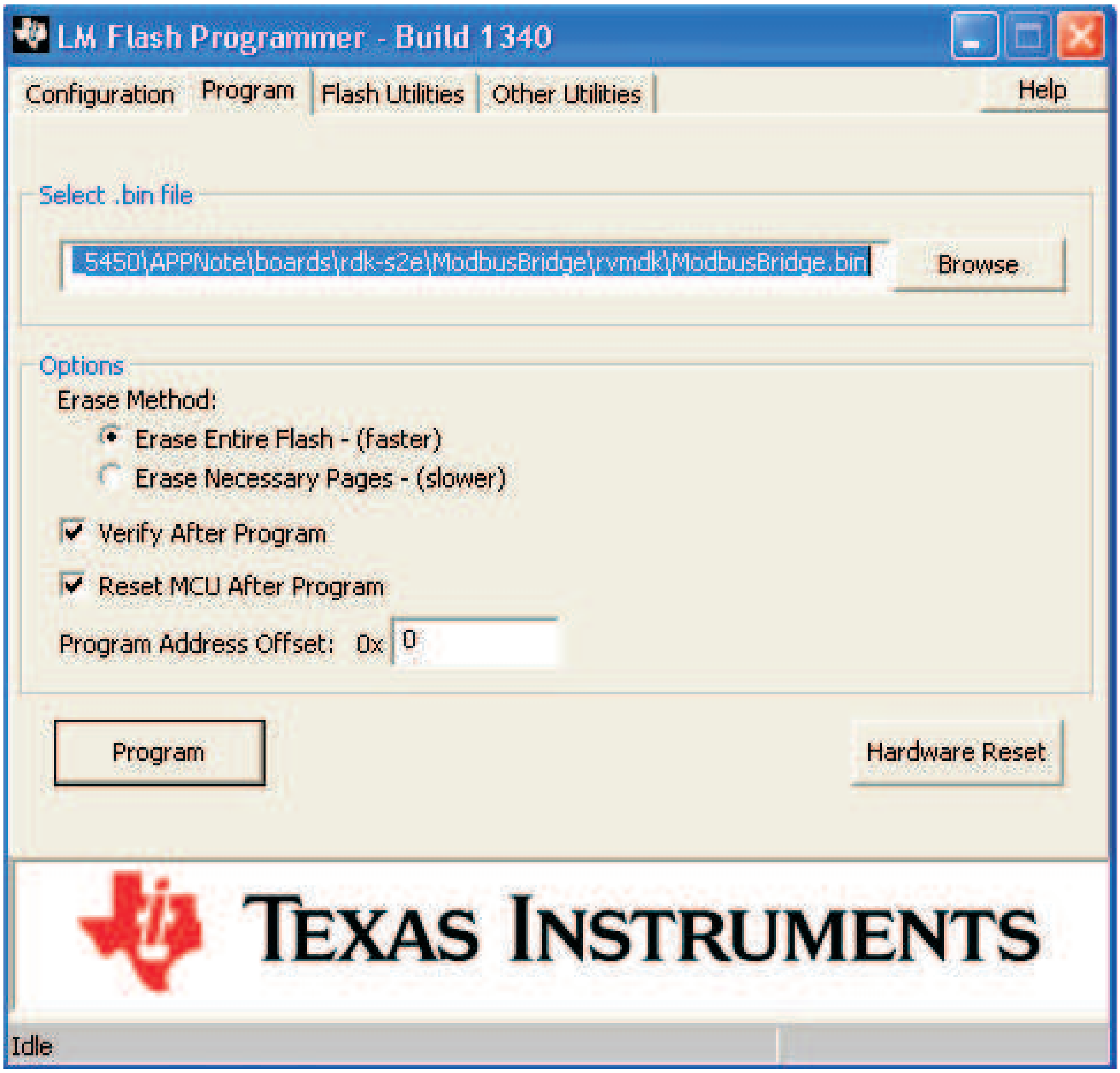

Figure 3. LMFlash Programmer Settings

3. Connect the S2E-RDK to local area network (LAN) through LAN cables in the RJ45 connector.

4. Power up the S2E-RDK by USB cable. Once the S2E-RDK is connected to Ethernet, it will acquire its IP address through the DHCP server; use *Finder.exe* utility in the *Stellarisware\tools\bin* directory.
   (a) Double click on the *finder.exe* icon. The finder utility starts and you should see the S2E module in the list of available Stellaris boards similar to Figure 4.

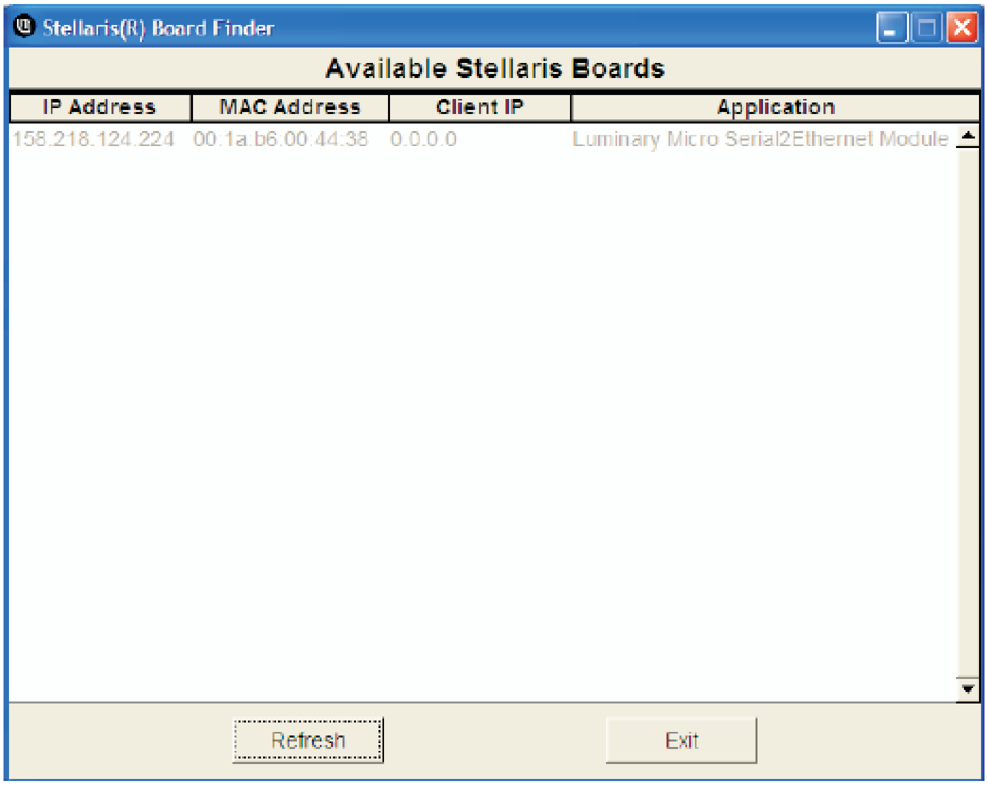

Figure 4. Finder Tool View

5. Install a PC-based Modbus tool that can poll a Modbus request through Modbus TCP (e.g., Modbus Poll).
   (a) Configure the connection setup to use TCP/IP and provide IP address of S2E module and Port 502 (see Figure 5).

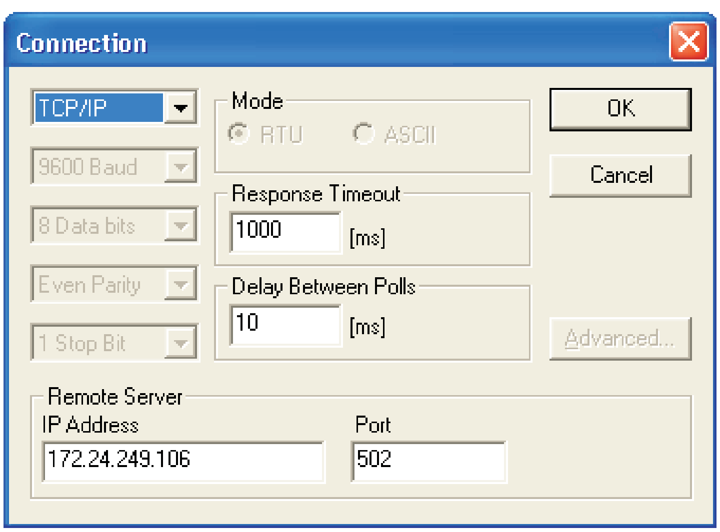

Figure 5. Connection Setup

6. Configure Modbus Poll by setting Slave ID, Function ID, number of registers to be read and start address (as shown in Figure 6). These values depend on connected Modbus Serial devices. For Slave ID and Start addresses of the registers, see the device-specific documentation.

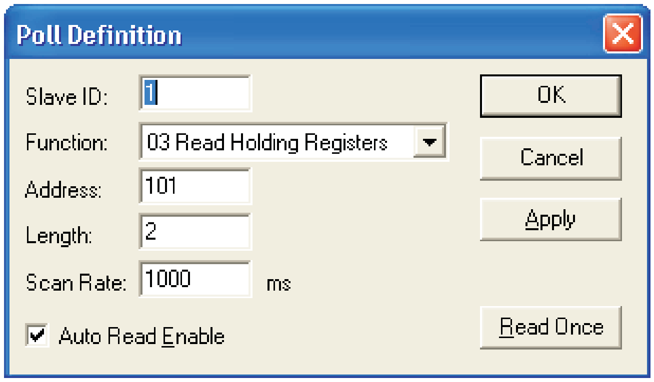

Figure 6. Poll Settings Setup

7. After setting poll definitions, the Modbus tool starts polling the client by sending the Modbus TCP request. You can view the polled register values and errors.

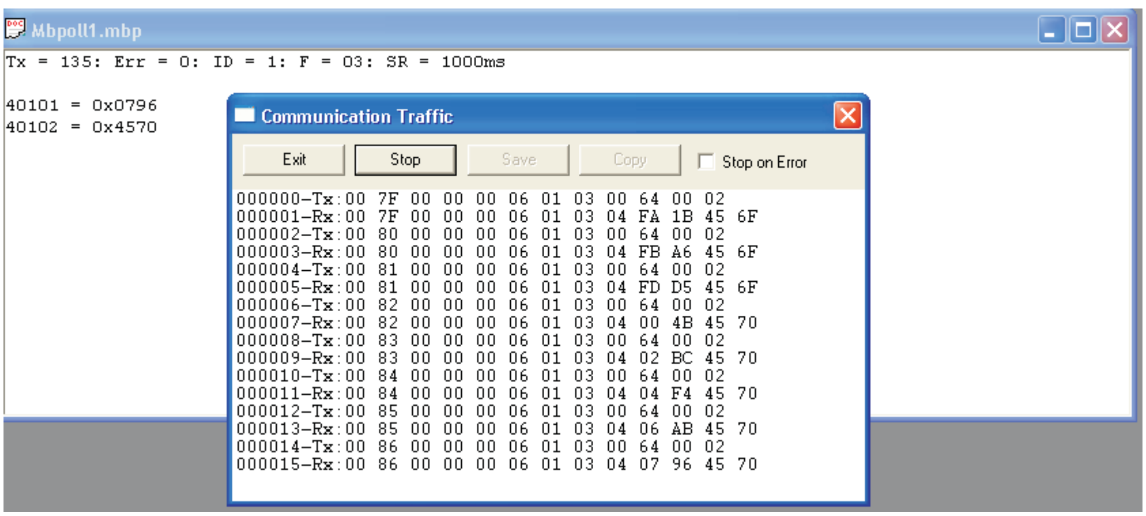

Figure 7. Communication in PC Tool

10 Project Information

This is implemented on the S2E framework in Stellarisware version 5450. This project workspace is created in Keil MicroVision integrated development environment (IDE). This can be easily ported to the latest version of Stellarisware and any IDE of your choice.

11 Limitation

- Current design supports Modbus polling from PC to one Modbus Client. Multiple clients are not supported.
- This supports only Read functions; Write functions are not supported. Write can be implemented in current statemachines.
- The timing parameter for Modbus RTU for char delay needs to be fined tuned.

12 References

- Stellaris Serial to Ethernet Reference Design Kit - http://www.ti.com/tool/rdk-s2e

IMPORTANT NOTICE