International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:11, No:12, 2017

An Implementation of a Configurable UART-to-Ethernet Converter

Jungho Moon, Myunggon Yoon

***Abstract*—**This paper presents an implementation of a configurable UART-to-Ethernet converter using an ARM-based 32-bit microcontroller as well as a dedicated configuration program running on a PC for configuring the operating parameters of the converter. The program was written in Python. Various parameters pertaining to the operation of the converter can be modified by the configuration program through the Ethernet interface of the converter. The converter supports 3 representative asynchronous serial communication protocols, RS-232, RS-422, and RS-485 and supports 3 network modes, TCP/IP server, TCP/IP client, and UDP client. The TCP/IP and UDP protocols were implemented on the microcontroller using an open source TCP/IP protocol stack called lwIP (A lightweight TCP/IP) and FreeRTOS, a free real-time operating system for embedded systems. Due to the use of a real-time operating system, the firmware of the converter was implemented as a multi-thread application and as a result becomes more modular and easier to develop. The converter can provide a seamless bridge between a serial port and an Ethernet port, thereby allowing existing legacy apparatuses with no Ethernet connectivity to communicate using the Ethernet protocol.

***Keywords*—**Converter, embedded systems, Ethernet, lwIP, UART.

I. Introduction

THOUGH Ethernet communication is very common in a variety of devices ranging from tiny embedded devices to high-performance personal computers, plenty of legacy devices with no Ethernet connectivity are still in use in many fields. Such legacy devices are usually equipped with traditional serial communication interfaces such as RS-232 or RS-485. To enable existing legacy devices with such an interface to connect to an Ethernet network, it is required to upgrade the communication interface of the legacy devices. This may cause both considerable cost and time or be technically impossible. UART-to-Ethernet converters are devices for converting TCP/UDP packets into asynchronous serial data and vice versa. These devices can help solve the problem without the need for modifying the existing legacy devices. The UART-to-Ethernet converter bridges seamlessly asynchronous serial ports of the legacy devices and the Ethernet network.

An implementation of a serial-to-Ethernet converter was described in [1], wherein the converter was implemented using an ARM-based 32-bit microcontroller STM32F107 and a TCP/IP protocol stack called lwIP (A lightweight TCP/IP). The STM32F107 is a connectivity line microcontroller equipped with an Ethernet peripheral that supports both the media independent interface (MII) and the reduced media independent interface (RMII) [2]. lwIP is an open source TCP/IP stack designed for embedded systems having limited resources [3]. The converter described in [1] supports 3 asynchronous serial communication protocols, RS-232, RS-422, and RS-485 and 3 different network modes, TCP/IP server, TCP/IP client, and UDP client. Fig. 1 shows the picture of the converter introduced in [1].

Fig. 1 The picture of the converter introduced in [1]

The lwIP supports well-known protocols such as IP, ICMP, UDP, TCP, DHCP, and ARP required for the implementation of the converter [2]. It provides both a basic raw API and a high-level sequential API that requires a real-time operation system (RTOS) [4]. The firmware for the converter developed in [1] was implemented as a callback-based application by adopting the raw API. The raw API provides better execution speed and less memory usage than the high-level API but adds software complexity. This paper describes another implementation of the serial-to-Ethernet converter introduced in [1] using the high-level sequential API, which leads to simpler and more modular firmware but requires an operating system. We adopted a real-time operating system named FreeRTOS. FreeRTOS is a free real-time operating system designed to be small enough to run on a microcontroller having limited memory space. It has been ported to a variety of commercial microcontrollers [5]. Because FreeRTOS supports multi-thread operations, the firmware of the converter was implemented as a multi-thread application, which makes the firmware more modular and the development of the firmware much easier.

II. Configuration of Operating Parameters

The converter has many operating parameters that decide the operation of the converter. The parameters can be configured

J. Moon is with the Department of Electrical Engineering, Gangneung-Wonju National University, Gangneung, Gangwondo, South Korea (e-mail: itsmoon@gwnu.ac.kr).
M. Yoon is with the Department of Precision Mechanical Engineering, Gangneung-Wonju National University, Wonju, Gangwondo, South Korea (e-mail: mgyoon@gwnu.ac.kr).

by the user using a dedicated configuration program running on a PC. The user-configurable operating parameters include the following:

- Converter's IP address, subnet mask, and gateway's IP address
- Network mode: TCP/IP server, TCP/IP client, and UDP client
- Port number to which other TCP/IP clients connect when the converter acts as a TCP/IP server
- Server's IP address and port number when the converter acts as a TCP/IP client
- Server's IP address and port number when the converter acts as a UDP client
- Serial communication parameters: Baud rate, data bits, stop bits, and parity
- Size of serial data to send as a single packet over the Ethernet connection (SOD)
- Timer's fire interval for determining the boundary between two groups of serial data (TFI)

The parameters, SOD and the TFI, are described in detail in [1] and thus omitted here.

Fig. 2 shows a screen snapshot of the dedicated configuration program running on a PC, where the configuration program shows the operating parameters of a converter after receiving them from the converter.

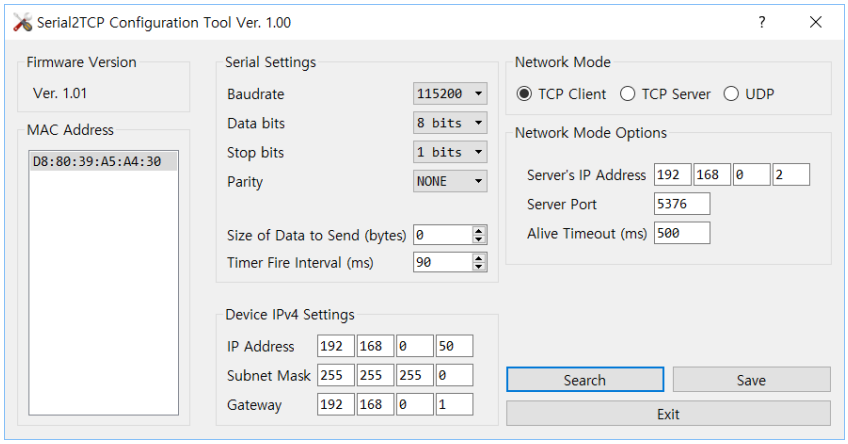

Fig. 2 A screen snapshot of the configuration program

As described in [1], the configuration program was written in Python using a graphic library called PyQT [6]-[9] and built into a standalone executable. Since the configuration program communicates with the converter using the UDP protocol, it is necessary to connect the converter to the same network as the PC on which the program is running to configure the converter. Alternatively, the PC and the converter can be connected directly using an Ethernet cable. It is not necessary for the user to know the IP address of the converter beforehand. Once the connection between the converter and the PC has been established, the configuration program is capable of reading the aforementioned operating parameters including the IP address of the converter from the converter and then modifying all the parameters.

Since the IP address of the converter is not known to the configuration program, the program communicates with the converter by exchanging UDP broadcast messages using a predefined format and port number. If more than one converter exists on the network, the configuration program receives multiple responses from all the converters but can identify each converter because the response from each converter includes its own MAC address. The configuration program shows MAC addresses on the left side as shown in Fig. 2. By selecting a specific MAC address, the user can see all the operating parameters of the converter associated with the selected MAC address and then modify them. On the microcontroller side, the task for transmitting/receiving the operating parameters to/from the configuration program and storing the operating parameters received from the configuration program in the flash memory of the microcontroller is handled by an RTOS thread created for this purpose, which will be described in detail in the following section.

III. The Firmware Structure

Fig. 3 shows the operation model of the lwIP when working with an RTOS.

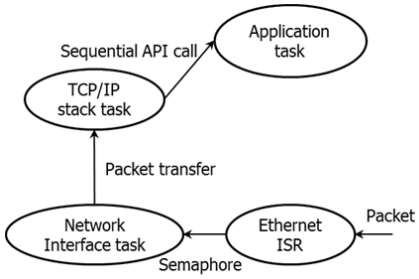

Fig. 3 lwIP operation model with an RTOS

The TCP/IP stack and the application run in separate threads. The application thread is created by the user and handles application-level tasks such as a HTTP server or a TCP/IP

client. The application thread communicates with the TCP/IP stack through sequential API calls for inter-thread communications [4]. The API calls are blocking calls, which indicates that the application thread is blocked after an API call until a desired response from the TCP/IP stack is received. An additional thread, the network interface thread, gets any received packets from driver buffers and provides them to the TCP/IP stack thread using an RTOS mailbox. The packet reception from the Ethernet is interrupt-driven. Once a packet is received, the network interface thread is notified of the reception through an RTOS semaphore. Though the operation of the application thread relies heavily on the underlying network interface thread and Ethernet ISR, the application thread does not have to know about them in detail. All the complicated tasks related to the network interface thread and the ISR are handled by the lwIP itself. The application thread therefore can simply communicate with the TCP/IP stack thread through sequential API calls without worrying about anything else.

Fig. 4 shows the application threads for the implementation of the parameter configuration and the three network modes supported by the converter.

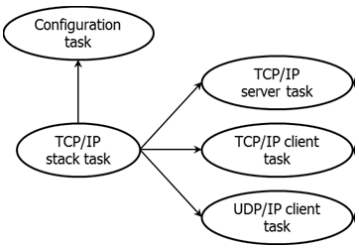

Fig. 4 Application threads

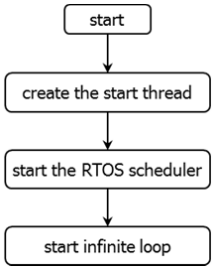

Fig. 5 Flowchart of the task done after power-up

The application is composed of four threads: Configuration thread, TCP/IP server thread, TCP/IP client thread, and UDP/IP client thread. After power-up, the configuration thread is always created and run but only one among the other three threads is created and run. When powered, one thread is chosen depending on the current network mode of the converter. Only the chosen thread is created and runs. If the user modifies the operating parameters of the converter and stores them, the whole parameters are overwritten without regard to the number of modified parameters and then the converter reboots. As a result, only the thread associated with the current network mode is created and run, thereby preventing the memory from being wasted.

Fig. 5 shows the flowchart of the task conducted after power-up. After power-up, a thread called the start thread is created first and then the RTOS kernel starts. The start thread does not start running until the kernel starts. After the kernel starts, the start thread initializes the lwIP and then creates the configuration thread. When the lwIP is initialized, all the system threads including the TCP/IP thread are created internally. After creating the configuration thread, the start thread creates one among the aforementioned three threads depending upon the current network mode.

The configuration thread handles the communication between the microcontroller and the configuration program running on the PC and storing received parameters in the flash memory of the microcontroller. Fig. 6 shows the flowchart of the task done by the configuration thread.

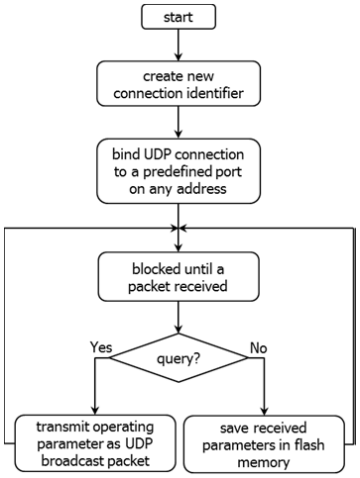

Fig. 6 Flowchart of the task done by the configuration thread

After creating a new connection identifier and biding the UDP connection to a predefined port on any IP address, the configuration threads blocked until a UDP packet is received on the port. As soon as a UDP packet sent from the configuration program is received on the port, the configuration thread gets unblocked and then analyzes the received packet and conducts the required operations.

If the received packet is a query for operating parameters, the configuration thread reads the current operating parameters of the converter from the flash memory of the microcontroller and transmits them as a UDP broadcast packet after appending the converter's unique MAC address to the operating parameters. When the configuration program transmits a command for storing operating parameters to a converter, the configuration program includes the parameters to store in the command. If the received packet is a command for strong operating parameters, the configuration thread stores the operating parameters contained in the received packet in the flash memory and resets the microcontroller.

As described earlier, only one thread is created and runs depending on the current network mode. Basically all of these threads work in a similar manner. After conducting necessary

initialization, these threads start an infinite loop in which API function calls are made. The API function calls are blocking calls which may block the caller. The TCP/IP server thread is blocked after entering a listening mode until it receives a request for connection from a TCP/IP client. The TCP/IP client thread makes a request for connection to a TCP/IP server and is blocked until it is accepted by the TCP/IP server. Likewise, these threads are blocked while waiting for a packet to arrive.

The tasks conducted after the threads are unblocked are quite similar to those conducted in the raw API implementation described in [1]. The way by which events are handled, however, differs completely. In the raw API implementation, the application layer communicates with the lwIP using event callback functions. These callback functions, therefore, must be registered before starting the communication process. If an event occurs, the registered callback function associated with the event is called automatically. In other words, the events are handled indirectly by the callback functions and therefore the code for processing the events is not shown in the main infinite loop. In the sequential API implementation using an RTOS, in contrast the application thread communicates with the lwIP through sequential API calls. It is thus more direct than the raw API implementation.

IV. Conclusion

The paper presented an implementation of a configurable UART-to-Ethernet converter using the lwIP and a real-time operating system named freeRTOS. Unlike the previous version, we adopted the high-level sequential API provided by the lwIP with the aid of freeRTOS. Compared to the raw API, the sequential API leads to a simpler implementation and less development time but results in more memory footprint and lower execution speed. The performance of the raw API implementation seems slightly better than its sequential API counterpart but the difference does not seem remarkable. The performance of the sequential API implementation is satisfactory as well.

References

[1] J. Moon, and M. Yoon, "An implementation of a configurable serial-to-Ethernet converter using lwIP", *Science International*, vol. ED-11, pp. 34–39, Jan. 1959.
[2] STMicroelectronics, *STM32F107xxx Reference Manual*, 2014.
[3] "lwIP Wiki", Available at: http://lwip.wikia.com/wiki/LwIP_Wiki. (Accessed Sep. 20, 2016)
[4] STMicroelectronics, *Developing Applications on STM32Cube with lwIP TCP/IP Stack*, 2015
[5] "FreeRTOS", Available at: http://www.freertos.org, (Accessed May 10, 2017)
[6] B. Lubanovic, *Introducing Python: Modern Computing in Simple Packages*, O'Reilly, 2014
[7] M. Lutz, *Learning Python 5/e*, O'Reilly , 2013
[8] M. Summerfield, *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*, Prentice Hall, 2015.
[9] B. M. Harwani, *Introduction to Python Programming and Developing GUI Applications with PyQT*, Cengage Learning PTR, 2011.