Serial Port Device Driver for Contiki Operating System

Milan Oklobdžija, Marko Nikolić, Vladimir Kovačević

Abstract — **Contiki is a fully optimized, lightweight, event-driven operating system developed for wireless sensor networks. It covers all major aspects of WSN operating systems, such as: processes (proto-threads), process interactions, communication stack and file system. However, Contiki OS lacks well defined driver structure. This paper proposes driver architecture for serial port devices. This architecture is based on object oriented approach and provides standard application interface enhanced with the great flexibility and code reuse.**

Keywords — **contiki OS, device driver, node, serial port driver, wireless sensor networks**

I. Introduction

Wireless sensor networks (WSN) are composed of a large number of autonomous tiny sensor devices with wireless communication capabilities for exchanging acquired information. The sensor devices are often severely resource constrained. An on-board battery or solar panel can only supply limited amounts of power. Moreover, the small physical size and low per-device cost limit the complexity of the system. Another constraint that opposes all mentioned is achieving fast system development time.

Considering all above mentioned demands, Swedish Institute for computer science has developed Contiki operating system (OS) [1]. Contiki kernel and libraries implement majority of features needed for functioning of the WSN node [2].

A. Contiki OS theory of operation

Contiki is multi-tasking operating system, especially designed for microcontrollers with small amount of memory, which are used in networked embedded systems and wireless sensor networks. A typical Contiki configuration needs 2 kilobytes of RAM and 40 kilobytes of ROM.

This work was supported by the Ministry of Science of Republic of Serbia, grant TR-11022 WSN and Remote Sensing – Foundations of Modern Agricultural Infrastructure.

M. N. Oklobdžija, Institute Mihajlo Pupin, Volgina 15, 11060 Belgrade, Serbia; (e-mail: milan.oklobdzija@institutepupin.com).

M. V. Nikolić, Institute Mihajlo Pupin, Volgina 15, 11060 Belgrade, Serbia; (e-mail: marko.nikolic@institutepupin.com).

V. B. Kovačević, Institute Mihajlo Pupin, Volgina 15, 11060 Belgrade, Serbia; (e-mail: vladimir.kovacevic@institutepupin.com).

Contiki is written in the C programming language. It is freely available as open source under a BSD-style license.

The main system parts are: kernel, libraries, program loader, and a set of processes. It contains all basic modules and features for the network node functioning.

Contiki was the first OS which introduced IP communication in low-power sensor networks [3]. Two communication stacks are available: micro IP and Rime. Micro IP is a small RFC-compliant TCP/IP stack which enables Contiki based system to communicate over the Internet. Rime is a lightweight communication stack designed especially for low-power radios.

B. Contiki kernel

Event driven programming model is widely used in resource constrained systems. "This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style" [4]. Contiki OS overcomes this drawback by introducing concept of proto-threads [5].

The kernel is event-driven, based on a lightweight event scheduler that dispatches events to running processes (proto-threads). Proto-threads offer almost the same functionality as the real thread, and they are specially designed for memory constrained system. Proto-thread does not have its own stack, but shares the same stack with the other proto-threads. Optional library for preemptive multi-tasking is also part of the Contiki OS.

Inter-process communication is achieved by posting events, which can be asynchronous or synchronous. Asynchronous events are queued by the kernel and submitted to the target process some time later. Synchronous events cause the target process to be scheduled immediately. Kernel function *process_post* is used for posting asynchronous event, while function *process_post_synch* posts synchronous event to the desired target process [6]. The kernel does not preempt a process once it has been scheduled.

The Contiki kernel also provides a polling mechanism. Process polling has higher priority than posting an asynchronous event. Function *process_poll* performs process polling. It is usually called in interrupt routines.

C. Motivation

However, the Contiki OS lacks well-defined and structured driver architecture. There is no standard driver

interface and it is different for each device.

The kernel does not provide a hardware abstraction layer, thus drivers and applications communicate directly with the hardware [1].

II. Proposed Driver Architecture

Serial interface is the most common way of communication between MCU and the other mote components (sensors, external memory, or RF transceiver). Thus, in this paper, special attention is devoted to the serial port device driver.

First, application layer is completely separated from the serial port device driver, then the driver is divided into a two sub-layers:

- external device driver (EDD) sub-layer
- serial port driver (SPD) sub-layer

EDD is a higher driver sub-layer and its function is to control external device through SPD interface. SPD is lower driver sub-layer. It is independent from a specific external device. This driver architecture concept provides great flexibility in case of hardware changes.

The proposed driver architecture is designed to fulfill several demands:

- separation of EDD and serial port driver
- serial port driver interface should not depend on serial interface mode
- serial ports must use interrupts for data exchange in order to avoid process blocking
- minimal overhead in the serial port interrupt routines
- providing reliable high speed serial communication with external device.

These demands can be achieved by assigning the single process (proto-thread) to all serial port drivers (serial port event process) and one process to each EDD. This process is introduced as adaptation layer to Contiki OS. Serial port event process checks if there is a new event at any serial port (new byte received or data packet sent) and informs EDD process which uses that serial port. EDD process accepts and processes application messages. It then forwards processed messages to the serial port driver (Fig. 1). In the opposite direction, EDD process parses bytes received from serial port driver and, after reception of complete message, sends processed message to the application. EDD and serial port driver exchange data through circular buffers. There is one circular buffer for reception and one circular buffer for transmission per EDD. Writing data to transmit circular buffer is interrupt-protected (interrupts are disabled). Reading data from receive circular buffer is protected in the same way.

During system initialization EDD should be created. Application assigns all the resources needed for EDD, such as: serial port, general purpose input/output pins and memory resources (for circular buffers). Function that creates EDD performs creation of circular buffers, serial port initialization and binds EDD and serial port driver.

A. Data Flow from the Application to the External Device

Complete data flow from the application to the external device is shown in Fig. 1.

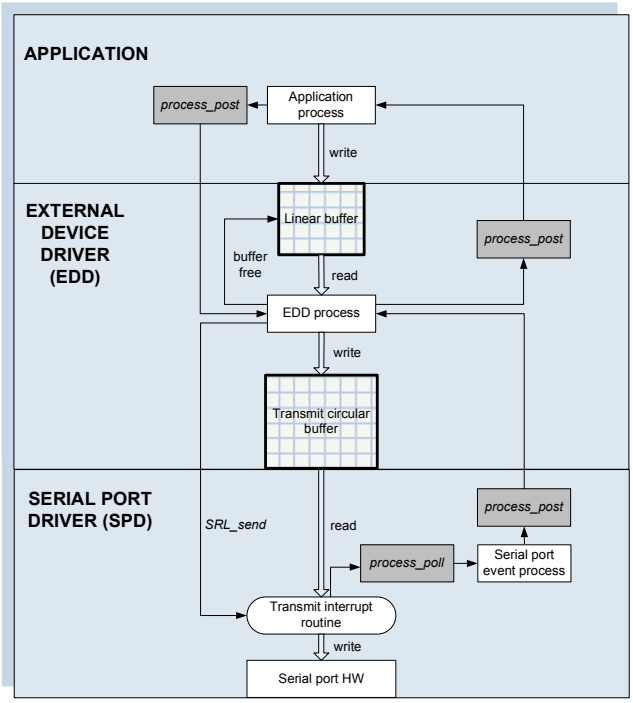

Fig. 1. Data flow from the application to external device

Application sends message to EDD by calling *process_post* function, which is part of OS kernel [6]. The arguments of this function are: pointer to EDD process, unique code representing application message, and pointer to the linear buffer containing message, respectively. EDD process performs message formatting depending on specific external device. Formatted message is then placed in the transmit circular buffer. After that, transmit cycle is initiated by calling serial port driver function *SRL_send*. This function has two arguments: serial port number and number of bytes to be sent. Inside *SRL_send* function, transmit interrupt is enabled for UART or SPI master mode of serial communication, while start condition is generated for I2C master mode. Transmit interrupt service routine reads one byte from transmit circular buffer and writes it to the serial port transmit buffer. After sending given number of bytes, appropriate serial event flag in serial driver object is set representing event "data packet sent," and *process_poll* function is called with pointer to serial port event process as only argument. When serial port event process executes, it sends appropriate message (according to serial port event flags) to EDD process bound to this serial port. Message is sent by using *process_post* function with following arguments: pointer to EDD process, unique code representing "data packet sent" event and NULL pointer. EDD process then informs application process by posting "data packet sent" event to it.

B. Data Flow from the External Device to the Application

Complete data flow from the external device to the application is shown in Fig. 2.

Receive interrupt service routine reads byte received from the external device and writes it to the receive circular buffer. If receive circular buffer was empty before byte reception, *process_poll* function is called with pointer to serial port event process as the only argument. Also, appropriate serial event flag in serial driver object is set representing "new data received" event. With conditional *process_poll* call, we avoid system message aggregation when they can not be processed immediately.

When serial port event process executes, it sends appropriate message (according to serial port event flags) to EDD process bound to this serial port. Message is sent by using *process_post* function with following arguments: pointer to EDD process, unique code representing "new data received" event and NULL pointer.

When EDD process is activated, it reads new data from receive circular buffer and performs data parsing. When whole message is received and processed it is sent to the application process. Only one message is processed at the time, even if several messages are received from the external device. In that case, EDD process has to post the same event to itself, before process exit. In this way long retention in one process is avoided.

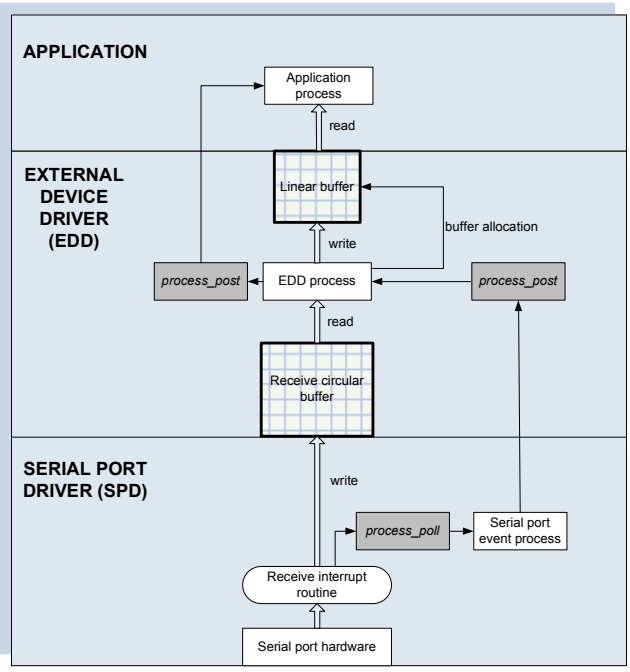

Fig. 2. Data flow from external device to the application

III. Implementation and Results

Proposed driver architecture for Contiki OS is implemented on a TI MSP430F5438 microcontroller. The Texas Instruments MSP430F5xx family of ultra low power microcontrollers supports all standard serial interface modes (UART, SPI and I2C) by just one configurable hardware module - USCI (Universal Serial Communication Interface) [7]. USCI module consists of two parts: USCI_A and USCI_B. USCI_A supports UART and SPI serial mode, while USCI_B supports I2C and SPI serial mode. Microcontrollers from MSP430F5xx family have different number of USCI modules (up to four identical USCI modules [8]). USCI module also exists in the older MSP430F2xx and MSP430F4xx families. Unique serial port driver (USCI driver) can be used for all those MSP430 microcontrollers, which have USCI modules, with negligible changes.

A. Code Description

Serial port driver (SPD) object is assigned to each serial port. SPD object contains all information about serial port state. It is not a real object and only represents abstraction consisting of the set of attributes and functions (separated from the attributes). SPD functions are unique for all SPD objects. They can be divided into two groups:

- general functions, independent of serial mode or specific MCU,
- specific functions, which depend on serial mode and specific MCU.

EDD communicates with SPD by calling SPD general functions only. Inside SPD general functions, specific SPD functions are called, depending on serial mode. Interrupt routines also call SPD general function, which invokes appropriate SPD specific function. General SPD functions and specific SPD functions for each serial mode are placed in different files.

SPD consists of the following functions:

SRL_create,
SRL_control,
SRL_sendData,
SRL_recvData,
SRL_sendRecvData,
SRL_processEvt,
SRL_processRxInt and
SRL_processTxInt.

For their operation these functions use SPD object that resides in RAM and consists of following attributes:

- mode of serial communication (UART, I2C master, I2C slave, SPI master or SPI slave)
- serial port clock source
- serial port clock frequency
- bitrate
- pointer to EDD process
- pointer to transmit circular buffer
- number of bytes to send
- pointer to receive circular buffer
- number of bytes to receive (I2C mode only)
- event status byte (each bit represents specific serial port event)

General function *SRL_create* creates SPD object and binds it with EDD. Function also performs initialization of serial port by using serial port settings structure placed in the ROM. Arguments of *SRL_create* function are: serial port ID, pointer to serial port settings structure, pointer to EDD process and pointers to EDD receive and transmit circular buffers. *SRL_create* checks if the desired mode is

supported by hardware and then calls specific functions for serial port initialization. Multiple initialization of the same serial port is prevented. Different functions are called for different communication mode (UART, I2C or SPI).

Serial port parameters such as bitrate and parity can be changed in run-time by calling general function *SRL_control*. Arguments of *SRL_control* function are: serial port ID, command and command arguments.

SRL_sendData is called from EDD process in order to send message to the external serial port device. Function argumets are: serial port ID and number of bytes to send. EDD writes the message in transmit circular buffer before calling this function.

Functions *SRL_recvData* and *SRL_sendRecvData* are used only in I2C master mode. By calling *SRL_recvData* function EDD process reads given number of bytes from external device. Function arguments are: serial port ID and number of bytes to be read. *SRL_sendRecvData* performs I2C write/read cycle. Its arguments are: serial port ID, number of bytes to send and number of bytes to be read.

Operations performed by following functions are described in Section II. in more detail. Processing of serial port events is performed by *SRL_processEvt* function. This function is called from serial port event process. It has a single argument – serial port ID. Functions *SRL_processRxInt* and *SRL_processTxInt* are called from serial port receive and transmit interrupt routines, respectively. They both have single argument – serial port ID.

B. Results and Performance

The driver code was developed and tested on the Texas Instruments MSP-EXP430F5438 experimenter board. SPD code was tested by using digital loop first. It was possible in case of UART and SPI serial mode. For I2C master mode driver code was tested using separate MCU with code that emulated slave device with I2C serial interface.

After that, serial communication with PC was tested using several instances of SPD simultaneously.

Code footprint is presented in Table 1.

TABLE 1: SERIAL PORT DRIVER CODE FOOTPRINT.

Code Segments	***ROM used (Bytes)***
UART mode	602
SPI mode	656
I2C mode	636
General	1824
Total	3718

Usage of RAM is 31 byte per serial port instance with 6 bytes of memory common for all instances.

Used bit-rate was up to 115200 bps for continuous transmission and reception. With CPU clock at 12MHz, three serial ports can perform simultaneous transmission and reception at 57600 bps bit-rate, with simple data processing. However, higher speed can be achieved when data is transmitted and received in the burst regime. In this case, size of receive circular buffers must be enough to accept all data.

For the test purpose, simple and fully functional EDD was developed.

C. Benefits of the Proposed Driver Architecture

The driver architecture proposed in this paper has several significant benefits. First of all, the application is separated from the drivers. This approach enables easier modifications of application software. Two sub-layer driver approach provides the great flexibility in case of hardware design changes. For example, if specific external device is replaced, it will be necessary to modify only EDD. On the other side, if MCU is changed, EDD will be retained, while SPD will have to be written for the new type of MCU. Both driver sub-layers have universal interface.

Separation of platform dependent and independent code is also provided.

Small code footprint is achieved, because all driver instances share the same functions, using principles of the object-oriented programming.

Maximal interrupt routine overhead is fixed and independent of specific application and communication protocol. Thus, high communication speed is achieved.

IV. CONCLUSION

Introduction of presented driver architecture in Contiki OS reduces application development time, both application and driver code footprint and increases system reliability.

This paper is focused on the serial interface. However, its main principles can be generalized for any type of MCU peripheral connected to the external device.

The results and contribution from this paper will be proposed to Contiki on-line community for becoming the part of the later versions of the Contiki OS.

REFERENCES

[1] Adam Dunkels, Bjorn Gronvall and Thiemo Voigt, “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”, Swedish Institute of Computer Science, 2004.

[2] Fredrik Osterlind, Adam Dunkels, “Contiki Programming Course: Hands-On Session Notes”, Swedish Institute of Computer Science, Siena, July 2009

[3] Adam Dunkels, “Towards TCP/IP for Wireless Sensor Networks”, Swedish Institute of Computer Science, March 2005

[4] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, “The Emergence of Networking Abstractions and Techniques in TinyOS”, NSDI 2004

[5] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, Muneeb Ali, “Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems, Swedish Institute of Computer Science, Box 1263, SE-16429 Kista, Sweden, 2005

[6] Adam Dunkels: Contiki OS on-line documentation, 2009, Available: http://www.sics.se/~adam/contiki/docs/

[7] Texas Instruments: SLAU208C, MSP430F5xx Family User’s Guide, June 2008 - revised February 2009, Available: http://www.ti.com

[8] Texas Instruments: SLAS612A, MSP430F543x Data Sheet, December 2008 - revised January 2009, Available: http://www.ti.com