VSPIN: A Framework for Developing Incremental Sensor Network Reprogramming Strategies

Biswajit Mazumder, Jason O. Hallstrom
School of Computing
Clemson University
{brazumder, jasonoh}@cs.clemson.edu

Abstract—We present VSPIN, a framework for developing incremental code update mechanisms to support efficient reprogramming of wireless sensor networks. The presentation emphasizes the supporting framework rather than the reprogramming mechanisms themselves. Concretely, VSPIN provides a modular testing platform on a host desktop system to plug-in and evaluate various incremental code update algorithms. The framework supports Avrdude, among the most popular programming tools for AVR microcontrollers.

VSPIN consists of a virtual serial port device driver which executes in Linux kernel space, a user space component, and a corresponding boot loader which executes on the sensing (microcontroller) core. The framework is tailored for AVR microcontrollers, which support in-system reprogramming of on-chip flash memory through a serial interface (SPI or UART). On the host side, VSPIN can function with wireless communication devices (802.11b/g/n or 802.15.4), or wired communication devices which expose a serial device interface on the Linux platform (RS-232 or USB). We describe the overall architecture and detail the individual components of VSPIN. The development of VSPIN is the first such attempt to facilitate ease of development, testing, and use of incremental code update algorithms for efficient network reprogramming.

Index Terms—Wireless sensor networks, reprogramming, incremental code update, software tools, Linux-based frameworks.

I. INTRODUCTION

Wireless sensor network (WSN) applications, such as disaster management, structural health monitoring, and volcanic activity monitoring [5], [10], [14], usually consist of a large deployment of resource-constrained sensor nodes. After the device-specific application code is developed (on a desktop machine), the compiled code is transferred to the sensor nodes and programmed into flash program memory.

Embedded system designers use a variety of tools to achieve network reprogramming. In addition to the compilers and assemblers required to generate the code, designers also need robust reprogramming tools, like Avrdude [6], which is a popular Linux-based tool used for programming Atmel AVR microcontrollers (MCUs). These software reprogramming tools can be used along with various hardware programming solutions such as in-system-programmers (ISP), or other serial-based reprogramming approaches [11], [12].

Reprogramming a sensor application may be required after a WSN has been deployed due to bugs or feature enhancements. Even with the aid of existing tools, reprogramming a large sensor network is a time-consuming task. Brute force code update mechanisms which require the entire code image to be resent are inefficient. To reduce the amount of time required for reprogramming a WSN, many incremental code update strategies have been proposed [8], [9], [13]. However, very little information has been reported in the literature regarding the supporting software engineering tools and frameworks used to develop these reprogramming strategies. As a result, the design of every new incremental update mechanism requires the designer to develop an ad hoc reprogramming framework.

We present VSPIN, a framework for developing and testing incremental code update mechanisms to support efficient reprogramming of wireless sensor networks. We do not focus on the particular mechanisms, but rather, a software framework used to support their development. We built VSPIN to facilitate implementation, deployment, testing, and use of incremental code update algorithms by providing a unified development framework for embedded system designers. The current implementation is tailored for reprogramming of Atmel AVR MCUs, which have on-chip, in-system reprogrammable flash program memory, using Avrdude [6]. However, the solution design can be easily adopted for use with any MCU which supports in-system programming and uses a set of standardized reprogramming tools.

The VSPIN framework consists of a virtual serial port kernel device driver, a user space program, and a bootloader executing on the AVR MCU core. VSPIN connects to the MCU through any communication device which terminates with a serial interface (SPI or UART) on the sensor node. On the host side, it is able to use wireless communication devices (Wi-Fi and ZigBee), or wired communication devices which are capable of exporting a serial device on the Linux platform (RS-232 or USB).

II. BACKGROUND

Avrdude [6] is a popular Linux-based command line tool capable of programming flash and EEPROM memory, as well as the fuse and lock bits of an AVR MCU. Avrdude supports specialized hardware programmers which comply with programming protocols specified in AVR068, AVR069, and AVR910 [1]–[3], by Atmel, including the popular AVRISP and AVRISPmini devices. Avrdude also works with a variety of other hardware programmers which connect to the host system using a parallel (ppi, parport) or serial port.
Listing 1 presents three different usage scenarios where Avrdude is used for programming a sensor node. The first example presents a typical scenario using an AVRISPmkII programmer; the subsequent examples show its usage when using VSPIN. Avrdude takes as input the type of MCU that is being programmed (-p), the communication port (-P), the programmer type (-c), and the input file containing the binary application data (-U) that is to be written to flash memory (flash:w). The application data (app.hex) sent to the boot loader is typically in the 16-bit Intel HEX format [7]. First, Avrdude performs checks to confirm the presence and status of the communication port and the programmer. Next, it queries and checks the values set on the various fuse bits in the MCU. (Fuse bit values are stored in specialized registers and determine the behavior of the MCU.) Finally, Avrdude parses the HEX file and transfers the data.

![Virtual Serial Port](image)

Listing 1. Avrdude usage example

Depending on the type of programming mechanism in use, application data can either be transferred to an intermediate device programmer, or sent directly to the MCU to be programmed. If an intermediate programmer is used, the programmer is responsible for transferring the data to the MCU. In the absence of a programmer, the sensor node requires a specialized boot loader capable of communicating with Avrdude and emulating the behavior of a programmer. After the application data has been written to the sensor node, Avrdude compares the MCU fuse values from before and after programming to check for consistency. Finally, it reads the data back from program flash memory, conducts a byte-by-byte comparison with the original binary file, and reports the result of the comparison.

Commercially available hardware programmers require system designers to use a brute force reprogramming approach, i.e., the entire program image must be sent to the sensor nodes. As a result, incremental code update mechanisms designed to achieve more efficient network reprogramming rely on ad hoc methods of binary image differencing and data transmission. Specialized boot loaders are used to support the transfer and decoding of the resulting code increments, as well as subsequent reconstruction and programming of the application data image. The boot loader is initialized and programmed onto the MCU using a programmer. After the boot loader has been installed, it interacts with the host system using a wired/wireless, serial/parallel communication device, using strategy-specific data transmission protocols.

In contrast, VSPIN provides a transparent solution to enable incremental sensor network reprogramming by allowing use of Avrdude without requiring any changes to its code base or usage (Listing 1). VSPIN uses the STK500 Communication Protocol [1], supported by Avrdude, to communicate between the host system and the boot loader. As a result, VSPIN is capable of allowing the use of Avrdude in incremental, as well as non-incremental programming modes.

### III. Architecture and Component Overview

The kernel module and user process components of VSPIN execute between Avrdude and the physical serial device module, while the boot loader is installed on the sensor node. Figure 1 presents an overview of the VSPIN architecture. Using the kernel module and user process, VSPIN has the ability to intercept messages sent by and sent to Avrdude. This is achieved by providing a virtual serial communication port, used as the communication port input parameter to Avrdude. The port is emulated by the VSPIN kernel module.

The messages intercepted by the kernel module are transferred to the VSPIN user process, which recognizes the syntax and semantics of the messages specified in the STK500 communication protocol. Depending on the type of message, the user process either forwards the message to the boot loader via the physical communication layer (represented by the /dev/ttyUSB0 and ftdi_sio modules in Figure 1), or responds to the message itself. If the message is forwarded, the boot loader processes the message and sends the corresponding response back over the serial connection, as per the STK500 protocol. The user process reads the response sent by the boot loader and relays it back to Avrdude via the kernel module. In addition to its ability to parse and forward messages to the MCU, and to respond to selected messages on its own, the VSPIN user process also contains incremental code update logic. The update logic resides in a separate module in the user process; it accepts pointers to the old and new program image versions as input, and outputs an edit script. VSPIN transmits the information in this edit script to the MCU, where it is used to reconstruct the new program image from the old version. As a result, we are able to “plug-in” any algorithm in VSPIN for incremental sensor network reprogramming.

#### A. Kernel Module

The VSPIN kernel module creates two devices on initialization: one virtual serial device /dev/vspins, and one character device /dev/vspinc, where each device has its own set of file operations. Figure 2 provides a detailed overview of the VSPIN kernel module. While the virtual

---

**Fig. 1. VSPIN architecture**

Listing 1 presents three different usage scenarios where Avrdude is used for programming a sensor node. The first example presents a typical scenario using an AVRISPmkII programmer; the subsequent examples show its usage when using VSPIN. Avrdude takes as input the type of MCU that is being programmed (-p), the communication port (-P), the programmer type (-c), and the input file containing the binary application data (-U) that is to be written to flash memory (flash:w). The application data (app.hex) sent to the boot loader is typically in the 16-bit Intel HEX format [7]. First, Avrdude performs checks to confirm the presence and status of the communication port and the programmer. Next, it queries and checks the values set on the various fuse bits in the MCU. (Fuse bit values are stored in specialized registers and determine the behavior of the MCU.) Finally, Avrdude parses the HEX file and transfers the data.

**Listing 1. Avrdude usage example**

```bash
// Usage with AVRISPmkII programmer
avrdude -p m644 -P /dev/ttyUSB0 -c avr ispmkII -U flash:w app.hex

// Usage with VSPIN boot loader, no kernel module, and no user process
avrdude -p m644 -P /dev/vspins -c stk500v2 -U flash:w app.hex

// Usage with VSPIN boot loader, kernel module, and user process
avrdude -p m644 -P /dev/vspins -c stk500v2 -U flash:w app.hex
```
serial port (/dev/vspins) is used to communicate with Avrdude, the character device (/dev/vspinc) is used for communication with the VSPIN user process. The two devices act as two end points of a communication channel and are used to efficiently channelize the transfer of data from Avrdude to the user process and vice-versa.

VSPIN makes use of two separate devices to solve simultaneous read and write synchronization issues and race conditions arising from the use of a single device. The /dev/vspins device is required because Avrdude expects to communicate with a serial device and issues Linux terminos library calls to the communication port it connects to. The virtual serial driver implements all of the file operations required to emulate a serial device, as shown in Figure 2. It supports all the terminos library calls (cfsetispeed(), cfsetospeed(), tcgetattr(), tcsetattr(), etc.), as well as fcntl() and select() calls made from Avrdude. It also provides support for a select number of options within the ioctl() system call.

While the serial device is dedicated for use with Avrdude, the character device driver provides another set of file operation callbacks for handling data transmission with the VSPIN user process using read(), write(), and mmap() calls. If VSPIN had been designed with a single device with simultaneous opens from Avrdude and the user process, proper channelization of data between Avrdude and the user process would have been substantially more complex. Also, serial device drivers do not provide a read() callback [4]; the read mechanism in serial tty drivers functions by forwarding any data received from the communication channel to the flip buffer in the Linux tty core module. Figure 3 illustrates the interaction between the Linux tty modules and the VSPIN kernel modules. The tty core module handles forwarding this data to any one process which has an open on the serial device; the behavior is undefined in the event of multiple opens on a single device.

On module initialization, the kernel module handles the registration of the serial device driver with the tty core module and creates all the necessary sysfs entries and devices. The serial device driver also interacts with the Linux tty line discipline module (Figure 3), required for supporting the select() call. The kernel module provides a single fixed size buffer which is shared by the two devices and used to transfer data from Avrdude to the user process, and vice-versa. The module also provides a semaphore to protect the buffer.

Avrdude communicates with the boot loader by issuing an open on its communication port, /dev/vspins, and issuing read and write calls to it. When Avrdude intends to send any message to the boot loader, it writes the message to /dev/vspins, the emulated serial port. Immediately after the write, Avrdude issues a timed read on /dev/vspins to collect the response to the message it just sent. When the virtual serial device receives any data from Avrdude via the write callback, it stores the data in the shared buffer and hands the buffer over to the read call back of the character device. Similarly, when the character device receives any data from the user process via the write call back, it stores the data in the shared buffer and hands the buffer over to the virtual serial device. The virtual serial device treats the character device as the communication channel. On receiving data, the serial device schedules a read task, which forwards the received data to the flip buffer of the tty core module. The data from the flip buffer is then forwarded to Avrdude, which has been blocking on the timed read call.

B. User Process

The VSPIN user process consists of three parts: the transmission module, the processing module, and the incremental code update module. Figure 4 presents a brief overview of the user process. The transmission module is responsible for all transmissions to and from the user process. The module issues two opens: one on the /dev/vspinc character device, to interact with Avrdude via the kernel module, and the other on

---

1 Reproduced and adapted from [4].
VSPIN assumes that incremental code update strategies require comparison of the new version of the program image with the old version already installed on the sensor node, and that the old image version is not available on the host system. Figure 5 illustrates the incremental code update process used by VSPIN. The process begins when Avrdude starts sending the new program image to the boot loader via the virtual serial device of the kernel module. The data for the new program image is intercepted by the user process and stored in a data buffer $d_1$. The user process emulates the boot loader behavior and sends a response back to Avrdude confirming receipt of the data. On receiving the confirmation, Avrdude attempts to read the programmed data back from the boot loader. The user process receives the read request and forwards the request to the boot loader. The boot loader reads data from flash memory and sends it back to the user process. The user process receives the old program image data from the boot loader and stores it in another data buffer $d_2$.

Next, the user process sends data from buffer $d_1$ to Avrdude instead of from buffer $d_2$. Avrdude receives the program image, verifies that the program image received is identical to the program image it sent, and exits. At this point, the user process has access to both the old and new versions of the program image in data buffers $d_1$ and $d_2$. The two program images are sent to the incremental code update module inside the user process, where differencing techniques are used to compute the differences between the two binary images. The incremental code update module creates an edit script which encodes the differences, and can be used by the boot loader to translate the old program image into the new image. The edit script is sent by the user process to the boot loader. On receipt of the edit script, the boot loader sends a confirmation back to the user process.

Next, the boot loader reconstructs the new program image from the old image in flash memory using the edit script it received. In the final step, the user process attempts to read the (now) programmed data from the boot loader. The boot loader reads the newly reconstructed program image data from flash and sends it back to the user process. The user process uses this data to verify successful reprogramming of the sensor node, failing which it restarts from step 7 in Figure 5.

D. Boot Loader

The boot loader provided by VSPIN has a simplistic design, closely resembling the boot loader design described in [12]. The boot loader receives messages via a serial interface, interprets the message, reacts based on the command contained in the message, and finally sends a response back via the serial interface. The boot loader supports the set of commands listed in the STK500 communication protocol. In addition to these commands, it supports user-defined commands for incremental code updates, which typically involve receiving the edit script, reconstructing the program image, and finally programming flash.

IV. Conclusion

In this paper, we presented VSPIN, a framework to facilitate ease of development, testing, and use of incremental code update algorithms for efficient sensor network reprogramming. We described the overall architecture of VSPIN, and then detailed the individual components. We presented details of
the VSPIN kernel module, where we use a virtual serial and a character device as the two end points of a communication channel. We presented our two device solution for read and write synchronization issues, and described the components of the VSPIN user process and the steps used to achieve a modular incremental code update solution. The design of the incremental code update module supports plug-in use of binary image differencing algorithms which take two versions of the program image as input, and output an edit script. This is the first such attempt to facilitate simplified development, testing, and use of incremental code update algorithms.

**ACKNOWLEDGMENT**

This work was supported in part by the National Science Foundation (CNS-0745846, CNS-1116976, CNS-1126344).

**REFERENCES**