AC 2007-2136: BRIDGING THE GAP WITH SLIP

Arlen Planting, Boise State University

Sin Ming Loo, Boise State University

Bridging the Gap with SLIP

Abstract

Many embedded system projects make use of some type of serial communications in order to transmit data packets between devices. The choice of methods to manage transmission and receiving of data is critical, and small systems are no exception. Communications of this type can be troublesome without borrowing techniques from other disciplines. However, one very important criterion is that the protocol must be lightweight. If the interface evolves from a "Hello world" message displayed on HyperTerminal to a full-blown data packet-passing system, chances are that the code is overly complex and difficult to keep running reliably. Techniques borrowed from internet protocols can help send messages reliably between two devices less than an inch apart with less effort than one might assume. Since most Electrical and Computer Engineering students may not have been exposed to extensive computer science courses on data structures or network protocols (usually elective courses), a few topics could be borrowed from these areas for a crash course before embedded systems are introduced. This paper presents the foundation for a crash course that we have researched and developed. The topics will include circular queues (ring buffer) from data structures and SLIP protocol from network protocols, which will provide compact, efficient and reliable solutions for many student projects. We have also tested the techniques presented in this paper in our FAA Airliner Cabin Environment Research sensor network prototype.

Introduction

Most electrical engineering students are not exposed to the courses needed to address many of the critical issues required for embedded system implementation, e.g. data structures, network protocols, and operating system device drivers. A computer science person would have had many of these courses, but minimal experience working with devices without an operating system. In addition, electrical engineering students traditionally have a C++ or Java course plus a microprocessors course covering an assembler language. Learning a more generalized, relatively low-level, portable language such as C with an emphasis on small embedded systems (a term we use to denote devices without an operating system) is more useful for these types of applications.

Courses on data structures, network protocols and operating systems are valuable, but adding these to an already tight schedule of electrical engineering curriculum is asking too much. This is why Boise State University is introducing an experimental one-semester crash course that will have the material in this Serial Line Internet Protocol (SLIP) article as its backbone [1,2]. The process of bridging the gap with SLIP will be used as a case study, with other devices and protocols covered as homework and projects. We selected this route due to the lack of computer engineering program at Boise State. The course will emphasize object-oriented techniques that are applicable to small embedded systems, and cover select topics from data structures, network protocols and operating systems. Though not everything can be covered, hopefully enough areas will be surveyed that the student will gain an idea of possible techniques for approaching software problems so as not to feel as lost on projects involving software.

This paper describes the process that led us to the discovery that internet protocols might be applicable to send messages reliably between two devices less than an inch apart, and the development of an experimental course to expose students to computer science topics in order to find better solutions for small embedded system applications. Section 2 describes our experience with a project where the apparently logical choice of methods to manage transmission and receiving of data initially produced code that was overly complex and difficult to keep running reliably. Various attempts to fix the problems and provide a robust protocol were unsuccessful, until an internet protocol was applied. Section 3 outlines the plan for an experimental one-semester crash course to introduce select topics from data structures, network protocols and operating systems applicable to small embedded systems.

SLIP

In this section, we describe in detail the implementation of and justification for utilizing the SLIP protocol as a means of bridging the gap between two very small devices communicating information.

Background

The process began with a project by Boise State University involving FAA research into airliner cabin air quality. The design included sensor nodes to collect information on airliner cabin air quality and relay it to a central node that would forward data via radio. The original concept was to cluster multiple sensors at a central location, with the central processor communicating with each attached sensor node.

As with many projects, ours included devices that use Universal Asynchronous Receiver/Transmitter (UART) as their primary communications interface. The UART itself was not a mystery – many of us are familiar with HyperTerminal or have used a modem to access the internet when traveling. In addition, we had used a UART to print debug messages to HyperTerminal on previous projects. We were also aware that any project involving serial communications might need to address the issues of master/slave designation, connection method (RS232, DB9, DB25), DTC, DCE, voltage levels/conversion, signals used (RTS, CTS, RX, TX), and hardware/software flow of control.

The devices used in our project were microcontrollers with no operating system. After some investigation, we determined that we could use 3.3 or 5 volts when talking device-to-device but needed to convert to 12 volts when connecting to a PC. The only signals needed initially were RX, TX and ground (though we found later that implementing RTS/CTS signals would eliminate some problems). To test transmission, we first sent Hello world to HyperTerminal. This was successful. Then we removed the DB9, directly connected another microcontroller and developed code to receive the Hello world message. Everything continued to work well - when we ran the program again, Hello world came out the other end.

The adventure began at the point when some real work needed to be done. The real work involved transmitting a series of messages, consisting of multiple characters in meaningful groupings (packets), from one microcontroller to another. The first problem was determining

how to distinguish between one message and the next. One idea was to borrow from the typewriter and use a line feed-carriage return (LF+CR) combination to designate the end of a message.

This practice of separating messages with LF+CR sequences works well for printable characters, but what about binary data (non-printable characters)? Can the LF+CR combination still be used to send the data packet? Should all data be converted to printable ASCII to transmit it, and then converted back to binary format on the other side (similar to using Uuencode/Uudecode)? How would we know when all of the data had been received? These are all issues that must be addressed to make sure the data transmission is reliable.

The LF+CR protocol might be a satisfactory method for some projects, but not all projects fit the limitations of transmitting only printable data. So we tried to design a robust protocol to handle everything that we might send through the pipeline, which could include different types of packets, variable length packets, and printable or non-printable data. The ability to delete a bad packet is also a requirement for reliable data transmission.

Implementation

One approach that incorporates these requirements for a robust protocol is a packet with the following format:

Type	Length	Data	Checksum

```
#define byte unsigned char

struct our_packet
{
        byte            type;
        unsigned int    length;         // assume same byte order
        byte            *data;
        byte            checksum;
};
```

Figure 1. Typical data packet

This packet format uses printable characters to define the type of message and binary values to define the Length field, while the Data field can contain either printable characters or binary data. The Length value at the beginning of the packet is used to determine when all the data has been received (With this format, the LF+CR combination is no longer usable since the Length and/or Data field could contain the LF+CR combination.). This appears to be a usable packet and it is easily sent over the UART, but receiving it reliably is another story when there is some kind of error.

For example, suppose you are in the middle of receiving data, counting the number of bytes received, and you don't get them all. What does the program do then? Or, maybe the Length field gets corrupted and the program counts too many or not enough bytes. Both of these scenarios will cause the embedded system to appear as if it has frozen up.

To correct this situation, we decided to use non-blocking calls instead of blocking calls for reading data. The problem with the non-blocking version is that if data does not arrive fast enough, the program may decide that the end of data has been reached and assume that the packet is full - only to find that the checksum does not check out. So the next step was to make a new function that was somewhere between blocking and non-blocking - a blocking get with timeout.

Though the timeout resolved the immediate problem, it did not appear to be a robust long-term solution as a timeout value is very specific to a particular set of hardware and software conditions. In addition, the timeout can introduce its own set of issues. When you do not reach the end of the packet, if you throw away the data since the checksum is invalid and go back and start reading more bytes you may find that the next byte is not a valid packet type, or the length is outside of an acceptable range. You seem to be caught in a place where you can never get back to a clean starting point.

By this time we had so much invested in this approach that we just kept hoping that with a little more tweaking the program would all settle down. Eventually we were able to get the routine to work fairly reliably by inserting a lot of delay loops throughout the code, but any changes would send the system back into the wild gyrations experienced previously. At this point, it appeared there was no better solution available.

Internet protocols had never been considered because they didn't seem applicable to our application, plus their computational and resource requirements were thought to be too extensive. Implementing the entire TCP/IP protocol stack was out of the question, as our system had limited computing resources that needed to be dedicated to the sensor tasks. However, selective implementation of portion(s) of the stack could be appropriate [3]. Since the TCP/IP protocol uses SLIP for data transmission over serial lines, it seemed reasonable to investigate how SLIP handled the task.

It turns out that SLIP is similar to our original approach of terminating a packet with an LF+CR pair. However, SLIP does not assume that the packet contains only printable ASCII characters. SLIP uses a single termination character (not printable). If the data sent just happens to contain this special termination character, it is escaped with another special character. You might fear that this escape process will go on recursively, but with further consideration you should be able to convince yourself that it works out very nicely.

The amount of code needed for encoding and decoding SLIP packets is minimal. Figure 2 is a condensed implementation of the SLIP protocol (additional error checking has been removed) to show just how little code is necessary for its inclusion in a project. It does not require extra buffers. It can be encapsulated into function calls like receivepacket() and sendpacket(). A better protocol might be point-to-point protocol (PPP), since it includes a checksum which

addresses a few other problems. However, it does require a bit more code and might be overkill for really simple applications.

Lessons Learned

What can be learned from this adventure? Specific lessons learned related to data communications include:

- SLIP (or PPP) will not work unless it is on both sides of the conversation. So when out shopping for devices, it might pay to investigate the types of API interfaces supported by those devices or see if an SPI or I^2C type device is available.

- When you are talking to a device in a pass-through mode and you have control of both sides, either SLIP or PPP will work great. However, when you need to talk to the device for configuration purposes, you do not have control of both sides and you will need to temporarily switch to some other mode. Most of the time this type of interface uses an AT command type interface. All characters are printable and terminated with LF or CR or both.

- There are still a few more problems that can occur. The SLIP (or PPP) protocol does not guarantee that you will get all packets. It just guarantees you can get re-synchronized when you have a bad packet, and do not continue mangling subsequent packets. If you need guaranteed delivery, then it would pay to spend even more time digging into the old network protocols textbooks.

The study of network protocols is typically focused on the big applications across miles of distance, and it may not be intuitively obvious that it could be applied to communications between two devices less than an inch apart. There may be people out there who progress from the simple LF+CR protocol to a SLIP-like protocol without all of the other adventures, but there are also probably others like us. Hopefully this article will save those others from so many gray hairs.

In general, the adventure also served as a validation of the heuristic approach and exposure to a wide range of technologies. As we discovered, the solution may lie in a completely different direction than originally thought.

```
#define byte                    unsigned char

#define slip_end                0xc0
#define slip_esc                0xdb
#define slip_esc_end            0xdc
#define slip_esc_esc            0xdd

void slip_send_packet ( byte * buff, int len )
{
        while ( len-- )
                slip_send_byte ( *buff ++ );

        uart_put_byte ( slip_end );
}

void slip_send_byte ( byte ch )
{
        switch ( ch )
        {
                case slip_end:
                        uart_put_byte ( slip_esc );
                        uart_put_byte ( slip_esc_end );
                        break;
                case slip_esc:
                        uart_put_byte ( slip_esc );
                        uart_put_byte ( slip_esc_esc );
                        break;
                default:
                        uart_put_byte ( ch );
                        break;
        }
}

int slip_get_packet ( byte * buff )             // return length
{
        byte    ch;
        byte    *pch = buff;

        while (1)
        {
                uart_get_byte ( &ch );
                switch ( ch )
                {
                        case slip_end:
                                return pch - buff;
                                break;
                        case slip_esc:
                                uart_get_byte ( &ch );
                                switch ( ch )
                                {
                                        case slip_esc_end:
                                                *pch++ = slip_end;
                                                break;
                                        case slip_esc_esc:
                                                *pch++ = slip_esc;
                                                break;
                                }
                                break;
                        default:
                                *pch++ = ch;
                                break;
                }
        }
}
```

Figure 2. Minimal implementation of SLIP protocol

Course Plan

The discovery that network protocols could prove useful for embedded systems led to the idea for a crash course to cover select topics from several areas outside the traditional electrical engineering and computer science curricula. Neither electrical engineering nor computer science students have been exposed to the full range of topics needed to address many of the critical issues required for small embedded systems. Most electrical engineering students have not had courses on data structures, network protocols and operating system device drivers; computer science students would have had many of these courses, but minimal experience working with devices without an operating system. Object-oriented techniques, though introduced in other courses, do not receive the emphasis needed to facilitate application to small embedded systems.

The principles of object oriented programming are just as applicable to small systems as to large systems. The computer science curriculum has evolved to starting with a language such as Java in order to teach object oriented techniques, and then progressing to other less object oriented languages such as C. This approach has proven to be less difficult to master than trying to apply object oriented methods after first learning the procedural approach to C. The wisdom of this approach applies equally well to persons wishing to develop code for embedded systems.

Electrical engineering students traditionally have a C++? or Java course, plus a microprocessors course covering an assembler language. None of these languages are ideal for small embedded systems, which require the reliability provided by object-oriented programming without the overhead of a high-level language (such as Java). A more universal, relatively low-level, portable language such as C is more useful for small embedded system applications.

The crash course is designed to cover small embedded systems (devices without an operating system) plus programming techniques applicable to these systems, for both Computer Science and Electrical Engineering students. The C Programming Language by Kernighan and Ritchie will be used as the basis for the C programming portion of the course [4]. Since the rest of the course material will consist of selected topics drawn from numerous other areas, reference materials on these topics will be supplied by the instructor.

Embedded systems will be introduced, with a discussion of both the differences between embedded systems and general purpose computers, and the differences between embedded devices with and without an operating system. The course will teach computer architecture as it applies to small embedded systems, including memory devices and memory-mapped versus isolated (port) devices. The appropriate choice of programming language and usage of object-oriented techniques will be discussed. The course will present the application of a layered approach to the development of device drivers in order to provide an effective abstraction of the underlying hardware, and will cover how to split the solution into layers that are useful for testing and portability.

The UART/SLIP example is ideal for the course backbone, since it incorporates elements of all areas to be covered in the course; specifically, operating system concepts (issues of concurrency with ISRs), data structures (circular buffers), network protocols in SLIP, and low level devices access [5]. Additionally, the solution to the problems encountered in the case study can be

structured utilizing layering techniques which go hand-in-hand with encapsulation methods. The value of object oriented techniques can be demonstrated by the reliability, flexibility and portability of code developed for the FAA research project. The research project integrated multiple sensors controlled by a schedule using an interrupt-based timer with asynchronous radio communications for data transmission. Minimal effort was required to add new sensors and port the code base to new processors and/or new devices.

Conclusion

Many embedded system projects make use of some type of serial communications in order to transmit data packets between devices, and the choice of methods to manage transmission and receiving of data is critical. Network protocols show promise in providing compact, efficient and reliable solutions for many student projects.

Since most Electrical and Computer Engineering students may not have been exposed to extensive computer science courses on data structures or network protocols, a few topics will be borrowed from these areas for a crash course before embedded systems are introduced. The course is designed to cover small embedded systems (devices without an operating system), including programming techniques applicable to these systems, for both Computer Science and Electrical Engineering students. A follow-up paper will be published to describe our experiences with this experimental crash course.

Bibliography

1. G. Skelton, "Introducing Software Engineering to Computer Engineering Students," Computer Engineering Department, Jackson State University, 0-4244-0169-0/06, 2006 IEEE.
2. K.G. Ricks, W.A. Stapleton, D.J. Jackson, "An Embedded Systems Course and Course Sequence," presented at Workshop on Computer Architecture Education held in conjunction with 32nd International Symposium on Computer Architecture, Madison, Wisconsin, June 4, 2005.
3. Stevens, W. Richard, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Reading, MA, 1994.
4. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language, Second Edition,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
5. Rubini, Alessandro, *Linux Device Drivers,* O'Reilly and Associates, Sebastopol, California, 1998.