A COMING OF AGE FOR GPS: A RTLINUX BASED GPS RECEIVER

Brent Ledvina − Cornell University − Ithaca, NY
ledvina@cornell.edu

Francisco Mota − UFRN − Brazil
fmota@ee.cornell.edu

Paul Kintner − Cornell University − Ithaca, NY
paul@ee.cornell.edu

Abstract

The Global Positioning System, or GPS, is a key technology for both the civilian and scientific research markets. GPS provides users with precise geodetic positioning and time determination. Most GPS receivers require specialized hardware and complex software to operate. One receiver, originally designed to run in DOS, will be the focus of a code migration to RTLinux. The major aspects of the migrated code are an interrupt service routine, multiple threads, and TCP connectivity. Following the migration we noticed that not only did receiver stability increase dramatically, but also errors in positioning decreased. To provide insight into code migration from DOS to RTLinux, we will focus on the specific difficulties, discoveries, and enlightenments encountered.

1. Introduction

Once the focus of many "realtime" computing applications, the DOS operating system has not followed a fruitful evolutionary path. With the ability for software to directly access hardware and its ubiquiteness, DOS levied an important niche in inexpensive realtime computing platforms. Today, DOS is dead, supplanted by various formulations of Windows. Windows succesfully nullifies the attributes which made DOS attractive to realtime computing, by providing a platform that responds slowly to interrupts, limits direct access to hardware and responds slowly and unreliably to interrupts. On the other hand, Linux, when modified to perform realtime computing, is an ideal candidtate for inexpensive realtime applications [3].

With intentions toward migrating realtime code from DOS to Linux, this paper provides a general guideline for those interested. Sacrificing extensive details over general ideas, this is an attempt to provide insight to those involved in such a task. Additionally, highlights of running realtime code in Linux are discussed.

2. GPS Basics

2.1. *System Basics*

GPS is a navigational system that provides precise determination of position on or near Earth. Currently, consisting of 27 satellites in medium Earth orbit, GPS allows users to ascertain their position to within approximately 20 meters [2]. Since its inception on Janurary 5th, 1980, GPS has continued to grow in popularity, while the breadth of appliations using GPS has increased dramatically. Indeed, it is common to find GPS's influence in navigational systems designed for geographic surveying, hiking, and even driving a car.

To take advantage of GPS, a user requires a receiver. Receiver designs vary significantly depending upon intended use, but are fundamentally similar.

2.2. *Receiver Basics*

The original design of GPS dates back to the early 1970's. Due to the lack of computational speed at this time, receiver design considerations were in favor of simplicity, in terms of hardware and software requirements. Most modern receivers take an approach that places a large portion of the receiver in hardware, which implies an embedded system with a proprietary operating system that executes code to track the satellites, produce a navigation solution (determination of position and time) and interact with the user. There is typically another layer of software that takes the form of mapping or navigational utilites for the user.

For a GPS receiver to function, it needs to lock onto satellite signals. Each satellite broadcasts two signals at 1.57542GHz and 1.2276GHz, denoted as L1 and L2, respectively. A satellite specific code, known as the course acquisition (C/A) code, is used to discern satellites. Correlation of the transmitted codes against local codes is needed to locate satellites in frequency space. The 1023 bit

C/A code modulates the L1 at 1.023MHz, repeating every millisecond. Accumulation of this 1000Hz data is required for a receiver to operate.

2.3. *The Development Receiver*
The receiver that we focus our attention on was orginally part of the GPSBuilder−2 development kit created by GEC Plessy and now owned by Mitel. This devlopment kit, released in 1995, was intended to provoke innovative designs that take advantage of the chipset. This receiver is unique in a number of ways. First, it is open source. Hardware descriptions along with the source code were provided for a development fee. Second, the system is split into two components. There is a chipset which resides on an ISA card for a PC, while all the software runs on a Intel x86 computer. Since its inception, another version designed as an embedded system using the same chipset and an ARM processor, was released by Mitel. This receiver along with more mature software is known as the GPS Architect. Due to the modern code and fact that both receivers rely upon the same chipset, we chose to combine the two, using the ISA card from the GPSBuilder−2 and the software from the GPS Architect.

3.GPS Architect Code Structure

The GPS Architect code is broken into task switching operating system , an interrupt service routine and five tasks. The operating system provides task sceduling and mutex operations. The interrupt service routine (ISR) performs two basic functions. It maintains the GPS Architect operating system and processes raw accumulation data from the hardware to maintain the signal tracking loops [1]. The ISR runs with a 900μs period and has a higher priority of execution than all the tasks. Additionally, the ISR is non−reentrant and it is important that its processing time does not inhibit other tasks. The remaining five tasks work together to provide I/O and compute the navigation solution. Figure 3.1. provides a visual description of the code structure, showing the interactions between the active task, operating system, ISA card and CPU.

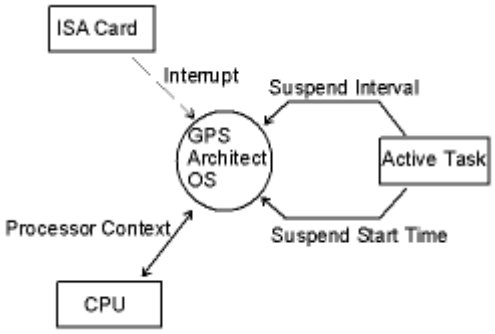

Figure 3.1. *Structure of GPS Architect Cod*e

3.1. *Thoughts about the Code Structure*
The Architect's structure is modular: it can be divided into two parts. First, there is the *application code*, which contains an ISR and various tasks used to track the satellites and compute the navigation solution. Second, there is the *user interface*, which interacts with the application code and user, providing user control and displaying information.

The DOS version of the code has several drawbacks; the first one is that the application code and the user interface are together in the same piece of code, and adding a new service would imply changing all of the receiver code. The other problem is that task parallelism is implemented inside the receiver code, that is, task timing and mutexes are all controlled by assembly code.

In the Linux version we chose to divide the code, so that the application code runs in kernel space and the user interface runs in user space. The main advantage of this is the level of abstraction between the two codes. For example, the user interface could be improved without (or with minor) changes in the receiver code itself; similarily changes in the receiver code will not interfere in the interface.

3.2. *The Linux Application Code*
In the Linux version, the code consists of five periodic threads, an ISR and a FIFO handler, all running in the kernel space under RTLinux v2.3. The threads exchange data using global variables while communication with the user interface occurs via FIFOs. The ISR executes the tracking loops and is executed every 800μs (but could be any time between 505μs and 900μs). The five threads, having periods ranging from 0.1 to 1 seconds, operate cooperatively to calculate the navigation solution. The FIFO handler is used to process user commands and is activated when there is data pending in the command FIFO. To implement this we changed the DOS code in the following ways:

1) Exclusion of all assembly code. The original DOS code used assembly code to implement task switching and mutexes. Since RTLinux implements these functions, removal of the assembly code was allowed, greatly increasing portability.

2) ISR definition. Since the DOS ISR contained a hefty amount of code, we created a minimalist ISR, transferring extraneous code to other tasks.

3) Thread definition syntax/scheduling. Specific changes to the task definitions were made to follow that of periodic theads.

4) Mutex definition. The DOS code implemented (in assembly) a recursive mutex; since RTLinux implements binary mutexes, we defined a function that makes a binary mutex behave like a recursive mutex.

5) User input command processing modifications. The user commands are now processed by a FIFO handler instead of the ISR.

3.3. *Comments about the Migration*
Using Linux was possible only due to the realtime extensions; RTLinux provides the necessary timing for the ISR, that could not be fulfilled with standard Linux [3]. From the viewpoint of technical work, this was relatively easy, but we should mention that this was possible only due to the similarity between the concept of task (as is was defined in the DOS code) and threads (specially "periodic pthreads", as it is defined in RTLinux). Had the DOS code not been split into several tasks, the translation would have been very difficult, if not impossible. Some technical issues appeared during the translation; but, in general, they were solved easily. Only one issue took more time to be solved: the implementation of a recursive mutex in RTLinux v2.3. This version of the code only supports binary mutexes. Since the synchronization between the threads in DOS code used recursive mutexes, we worked around this problem defining a function that uses the thread id as parameter to implement a recursive mutex from a binary one. Beyond this, other changes to the code were mostly of the "cut and paste" type.

4. User Interface

The user interface interacts with the application code via FIFOs. This bidirectional interation can take many forms, but generally the user interface displays current information and allows the user to control the application code. For example the user interface may provide the user's position, time, date and allow the user to adjust receiver operating parameters.

4.1. *Investigations into User Interface Designs*
The generic user interface mentioned above can certainly be augmented. To extend the concepts of abstraction and modularity we designed a TCP server that reads from the FIFOs and forwards the data to clients connected over sockets. Adding the TCP server allows flexibility in the locality and type of user interface. Figure 4.1. depicts the possible layers of code for the receiver, showing that the application code can run independently of the user interface or in conjunction with a local client or with one or more remote clients.

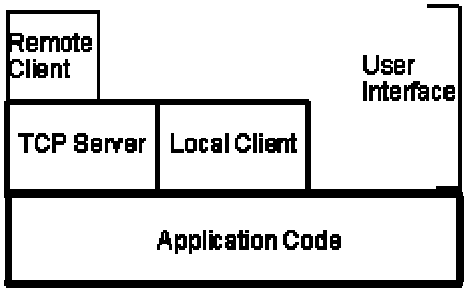

Figure 4.1. *Conceptual Layered View of the Receiver*

As a demonstration of concepts and to showcase flexibility of the receiver we devloped three unique interfaces. First, there is a graphical user interface designed with the open source library Qt. Second, there is an ncurses version. Third, a java applet.

The first interface, designed with the GPL'd Qt graphics libraries, provides a visually enhanced design that is flexible in terms of graphic possiblities and user friendliness. Using the Qt classes, we can easily design 3−D plotting routines, real time analysis of data, advanced mapping utilities, and a user friendly interface that runs on a myriad of platforms. This version can directly write to and read from the FIFOs or connect to the server over a TCP connection. Control of the application code is implemented with dialog boxes.

The next interface uses the ncurses libraries commonly found in Linux and other UNIX−like operating systems, to generate a text based display. This user interface communicates over TCP sockets to the server, provides command line control of the application code and presents minimal information to the user.

Third, to exemplify the internet connectivity of this receiver we wrote a java applet which runs in a web browser. It displays the receiver's position, date, time and a graph of the error in position with respect to a reference position. This interface is only for monitoring the receiver's operation.

Each interface provides a different level of sophistication, highlighting modularity while providing different modes of user interaction.

4.2. *Uses of this Receiver*
Due to RTLinux's platform options and the modular receiver design, considerable flexibility is available for applications of this receiver. The receiver will be part of ionospheric monitoring stations in equatorial regions of the world. It will also be incorporated into a course on GPS taught at Cornell University. There is also the possibility that this code will be added to embedded systems used in sounding rockets and satellites.

5. Summary of Advantages Regarding Migration

Our goals in migration were centered around insufficiencies in DOS. Choosing Linux provides a modern operating system, while the RTLinux extensions provide hard realtime support required by a GPS receiver. These choices allow for many of the shortcomings of DOS to be circumvented.

5.1 *Linux Advantages*
Exiting the dark ages, so to speak, is the result of migrating from DOS to Linux. In the DOS regime, due to the lack of concurrect processing, the receiver is the only application that can run. With Linux, the computer is capable of running multiple other applications. For example we are able to run the receiver concurrently with a distributed.net client, Netscape and a whole slew of other programs.

Another feature of Linux is its intimicy with networking. In DOS, networking options are limited, but in Linux, doors swing wide open regarding possibilites of network connectivity. For example, attempts to implement a TCP server in DOS is a daunting, if not impossible, task, but is straightforward in Linux.

5.2 *RTLinux Advantages*
Since RTLinux provides hard realtime computing [3], the 1000Hz interrupts are guarenteed serviceable and with minimal delay. Guarantee of this servicing is required in a realtime system like GPS and for this reason receiver operation is more reliable. The DOS version frequently lost data due to delays in processing interupts. This loss is not fatal, but is cause for potentially unreliable behavior.

In the GPSBuilder−2 specifications, the interrupt period is 505us, whereas the RTLinux code uses a 800us period. By using a more frequent interrupt rate, DOS has a greater likelihood of actually servicing the interrupt within the 1 ms window. With RTLinux, interrupt servicing has incredibly low latency, thus it is allowable to increase the interrupt period, thereby reducing loading of the processor.

Another advantage gained by using RTLinux is its support of scheduling and mutexes. As previously mentioned, RTLinux's support of these features provides for an implementation that is platform independent and due to the exclusion of assembly code, easier to code from a programmer's point of view.

Finally, RTLinux support for FIFOs allows for a layer of abstraction between the application code and user interface.

5.3 *Other Advantages*
Quantification of stability is a difficult task. When an application crashes frequently, it is considered unstable, while infrequent crashes render a notion of stability. From our experiences, the use of RTLinux provides for an incredibly stable receiver. The DOS version is plagued with hardware initialization problems and frequent crashes, while the RTLinux version runs uninhibitted for weeks.

6. Acknowledgements

This work was supported in part by the Office of Naval Research grant ONR 00014−92−J−1822.

7. References

[1] Mitel Corporation, *GPS Architect Software Design Manual*,Mitel Corporation, 1999
[2] Parkinson and Spilker, *Global Positioning System: Theory and Applications*, American Insitute of Aeronautics and Astronautics Inc, 1996
[3] Yodaiken, Victor, *The RTLinux Manifesto*, New Mexico Institute of Technology, 1999