AC 2007-1250: USING THE RUBY LANGUAGE AS A PROGRAMMING ENVIRONMENT FOR A ROBOTICS LAB-BASED CLASS

C. Richard Helps, Brigham Young University

Richard Helps is the Program Chair of the Information Technology program at BYU and has been a faculty member in the School of Technology since 1986. His primary scholarly interests are in embedded and real-time computing and in technology education. He also has interests in human-computer interfacing. He has been involved in ABET accreditation for about 8 years and is a Commissioner of CAC-ABET and a CAC accreditation team chair. He is a SIGITE executive committee member and an ASEE Section Chair. He spent ten years in industry designing industrial automation systems and in telecommunications. Professional memberships include IEEE, IEEE-CS, ACM, SIGITE, ASEE.

Andrew Arnott, Brigham Young University

Andrew Arnott was a student at Brigham Young University with a strong interest in applications of the Ruby language. He is currently working for Microsoft Corp.

Using the Ruby Language as a Programming Environment for a Robotics Lab-Based Class

Abstract

The object-oriented, scripting language Ruby, is becoming popular in information technology and computer-oriented educational programs. Yukihiro Matsumoto has indicated that the language was designed with the principle of "least surprise" to help programmers convert ideas into working programs quickly and to have fun. Ruby was designed for applications in many of the modern application areas of software development, including text processing, network programming, interfacing to CGI and XML and addressing Internet-oriented applications. It is also intended to make programming fun for students.

Ruby can also be used for embedded system programming. The general advantages of Ruby programming also apply to this domain. The Ruby interpreter allows rapid development and testing, including wireless tele-operation of mobile robots for prototyping. Since Ruby was not primarily designed for this type of application some adaptations are necessary. Extensions to the language are necessary to allow for real-time interfacing. These and related topics are discussed in the paper.

This paper discusses a semester-long experience of adapting Ruby to serve the needs of a robotics-design, lab-oriented course and evaluates the benefits and disadvantages of Ruby both for embedded development in general and as a teaching tool.

Introduction

Although desktop and laptop computers are ubiquitous they are outnumbered by at least an order of magnitude by embedded computer systems. These embedded systems appear in devices ranging from microwave ovens and cars to PDAs, cell-phones and even network routers and switchers. Embedded systems were once dominated by small 8-bit microcontroller systems, with kilobytes or less of memory; Megahertz or slower CPU clocks and were programmed in assembly-language or in C. However as systems have been developed for small low-power, low-cost, high-capability devices, such as cell-phones and PDAs so 32 bit processing power with megabytes of memory; complete operating systems; networking and object-oriented languages, have started to appear even in small embedded systems. This trend has been further accelerated as notebook computers, and the compact low-power technologies they need, have overtaken desktops in sales[7]. These low-power, high performance, compact technologies also become available for embedded systems development.

BYU has offered courses in embedded systems for some years. These elective courses are targeted at students who wish to explore computer applications beyond the traditional desktop or laptop environment. Over recent years this class has, like the marketplace, moved its focus from small 8-bit system to modern operating systems and programming environments[3], but is still focused on physically small, mobile systems with real-time needs and IO capabilities far beyond the traditional keyboard/mouse/monitor peripherals.

Applications are traditionally programmed in a remote desktop development system and then the executable code is uploaded into the target system, which may be running a different CPU, architecture and operating system than the development system.

One of the elective courses offered is in Mechatronics. This course requires students to design and build a small autonomous robot with motors, sensors and actuators. The course typically attracts Mechanical Engineering (ME) and Information Technology (IT) students who work in teams. The ME students provide expertise in mechanical design and related engineering topics and the IT students provide expertise in operating systems, programming and computer systems integration. Both groups learn new skills in input/output handling, autonomous intelligence and related mechatronic concepts. The IT students have already completed several required courses in programming and computer configuration while the ME students have completed courses in mechanisms, instrumentation and related subjects, as well as a minimum of one required course in programming.

The traditional languages of embedded systems development, mostly C and assembly-language, have much to recommend them. With no operating system, they are fast, compact, and capable and they have good access to hardware and peripherals. In addition they are well known and accepted by the many engineers working in this field. However they are, in some senses, holding back the development of embedded systems in that developers cannot easily take full advantage of the many libraries and utilities developed for 32-bit systems using modern operating systems like UNIX and Windows CE, nor do they have the advantages of modern object-oriented languages

Many of these newer languages that may allow programmers to be more productive come with system costs that make them prohibitive for small embedded systems. These languages make the programmer's job easier at the expense of requiring better hardware and a full operating system on the embedded system to run the resulting program. Many of these more sophisticated systems also require per-user license fees, which poses a problem for price-sensitive embedded systems.

Recently we explored the use of the popular Ruby scripting language as an alternative to C and assembly-language and found some interesting advantages in using this language for embedded systems development. This report will discuss how Ruby can be applied to the task of real-time embedded system programming, including some extensions to overcome difficulties. It will then analyze the suitability of Ruby for this class of applications. To more fully describe the role that Ruby can play the viability of several other modern languages will be discussed.

Development Language Comparison

There are several issues that must be considered when choosing an appropriate language for embedded systems development. One important issue is the difference between compiled and interpreted languages. In general compiled languages, such as Pascal, Java, or C++, are structured to help the programmer write error-free and maintainable code. Modular structure and variable typing, for example, constrain the programmer in ways that lead to better code being developed. These languages can be optimized by a good compiler and can produce fast and reliable executable code. This comes at a cost of being more verbose in programming and requiring compiling for each development cycle.

One of the problems of embedded systems development is the fact that code is almost always developed on a development machine. The development system typically has a different architecture and OS than the target system, requiring special cross-compiler libraries to be used.

When compiled the executable code is ported over to the target system where it is run. This makes debugging and changing code tedious. The target system is typically much less capable than the development system in terms of speed, memory and it usually doesn't support a standard keyboard, monitor and mouse.

Typical workarounds include using very simple interpreted versions of languages like C which can be edited on the target system; running simulators on the development system to test the code before porting it; automating the compile and port process or, most expensively, running an emulator "pod" connecting the target and development systems so the target can be run under direct control of the development system. The cabling and connecting of emulators becomes very impractical in the later stages of developing very small systems or mobile systems.

Most modern languages offer some provision for object-oriented programming. In an object-oriented language, a programmer's focus shifts from writing a list of instructions to writing the model to represent a real-life situation. This model can then be taught to solve problems in a very intuitive and maintainable way. Life for the programmer is easier, coding often happens faster, and teamwork and code reuse is encouraged. Object-oriented programs can suffer from a performance hit when compared to compiled procedural programs, but this can depend on compiler efficiency and, with modern fast hardware platforms, this may not be a significant factor. The weaknesses of OO languages for embedded systems development are the compile and upload time, the size of the executable code with associated libraries, the run-time performance, and, very significantly, the relative difficulty in directly accessing the hardware from the application language.

In contrast to compiled languages interpreted languages can be quickly programmed and instantly run. Languages such as Perl or any of the many dialects of BASIC are loosely typed, relatively unstructured and designed for rapid code development by non-professional programmers. This has many advantages for an embedded systems development environment characterized by rapid development with many cycles of creation, testing and incremental improvement. The extensive IO requirements of embedded systems development are well supported by a development environment that permits many short experimental development loops as hardware communication and response is explored and tuned. The well-known costs of this approach are the lack of software structure and the consequent risk of poor coding habits and also code that is difficult to debug and maintain. Consequently interpreted code is often only recommended for short utility programs.

For scripting applications run-time speed is not usually a major factor and the execution-time penalty of interpreted languages is acceptable, however for real-time applications the run-time speed may be a serious barrier.

Many large applications are being written in interpreted or scripting languages. Faster computers make the slower speed of scripts less important. As these interpreted languages mature, they become more aligned with the good programming design that favors large and maintainable programs.

Scripted languages also tend to be very portable, making programs written in them instantly usable on multiple devices across hardware and operating systems

Strong typing and structure is not always associated with compiled code, nor is loose typing always associated with interpreted code. They commonly are paired that way, however. With a strongly typed language, since the compiler has more information available, it can catch more errors automatically than a loosely typed language can. The trade-off for using a strongly

typed language is that it typically requires more code writing, and in some cases can lead to harder-to-read code.
Some languages lie between these two extremes, to try to bring in the best of both worlds – with varying success.

Another important issue for embedded systems is the ability to access the hardware easily. This requires a language that can manipulate bits easily and a language/operating system combination that allows free access to the hardware directly from the application source code. Procedural languages such as C and assembly-code are both very good in this respect. They can relatively easily access the hardware and give the programmer good control over the inputs and outputs that comprise so much of embedded-system development. These languages also produce very compact and fast executable programs. What they lack is the stronger structure of newer object-oriented languages and they are also compiled (or assembled), and thus they lack the immediacy of interpreted languages for development.

Real-time applications

A real-time application is a system that must respond to stimuli in a set period of time. A real-time system is not necessarily a *fast* system. The required response time of a real-time system may be a microsecond, an hour, or a day. However real-time constraints become difficult to meet as the specified response time moves into the millisecond or faster range.

Real time constraints can be hard, meaning the system has failed if the constraint is not met, or soft – meaning the performance is degraded but the system can continue to function if the constraint is not met.

Embedded systems often have both hard and soft real-time constraints imposed on them. When a consumer purchases an electronic device, the consumer takes for granted that the device will be responsive in its interactions. As a result, manufacturers must put hardware and software into the device that is capable of responding fast enough to qualify within the implied real-time constraints, or in other words, to be fast enough to appease the user. In many cases they also have hard real-time constraints. For instance software to speed control a motor must meet its various control deadlines or the motor will malfunction, with possibly very serious consequences.

Computer languages vary in their ability to produce real-time capable code. The Assembly and C languages are very widely used because of their fine-grained control of exactly what code gets executed when, and their free access to hardware. Often these languages run on systems with little or no operating system or monitor so the programmer can (and must) directly control the real-time response within the application suite. These systems offer no services that run in the background to help the programmer get the job done. While this makes the programmer's job more tedious, the programmer can ensure that a routine will take a constant time to run and thus satisfy real-time requirements.

Multi-threading can also contribute to real-time capability. Multi-threading enables a program to have multiple strings of instructions executing simultaneously on the system. Multi-threading can make a system more responsive and capable of handling more than one simple task at any given time. While nothing in assembly or C precludes writing multi-threaded software, these languages do nothing to assist the programmer in writing it. Other more modern languages include constructs that greatly aid the programmer in writing these responsive, capable programs.

For the best embedded system programming we would like to have a language that supports modern programming style well, including support for multi-threading and real-time implementations. A language that also supports fast development and direct interpreted control also has great benefits. The Ruby language has many of these features.

Ruby Programming Language

Ruby is a modern, object-oriented, interpreted language[4, 5] that is quickly gaining popularity, especially in the open-source world. Ruby's strengths include terse yet readable code, an interactive interpreter, multithreading support, a vast set of available libraries, and platform independence.

Ruby provides a very dynamic and somewhat loosely-typed language. This makes writing and deploying code across platforms very quick and easy. And though the interpreted nature of Ruby code can add some non-deterministic timing on its programs, this need not prohibit some real-time applications from being written using it because the variation in timing brought in by the Ruby interpreter does stay within predictable limits.

Ruby is specifically designed to be easy for programmers to create code[9]. On the other hand Ruby's creator, Yukihiro Matsumoto, has also specifically stated that run-time performance was deliberately ignored in creating the language[10]. Thus using the language in embedded real-time applications requires that we pay attention to these aspects. Mr. Matsumoto makes the point that run-time performance and reliability is achieved by a combination of modern, fast hardware and a well-written interpreter. This needs to be carefully considered in a real-time application.

Ruby has advantages over other popular scripting languages for embedded systems development. Baas[1] describes the problems of Perl's syntax for non-experienced programmers, Javascript's web-orientation and Python's weakness in object-oriented support. Ruby has none of these weaknesses. Ruby may also have a speed advantage over Python[2].

Ruby compilers are starting to become available. When they have matured, faster execution time for Ruby will be available where needed.

The view has been expressed that the growth of Java applications environments, with developments such as .NET, will eliminate the niche occupied by scripting languages[8]. Ruby still has a significant advantage over object-oriented languages, such as Java, in that it allows for an interpreted mode. Code can be edited directly on the target system over an ftp link and then run using the 'irb' interpreter. The tedious steps of compiling and porting from the development system to the target system step can be eliminated, allowing for much quicker development cycles.

The Development Project

The class project requires that students design and build a small autonomous robot. A typical design from a previous class is shown below

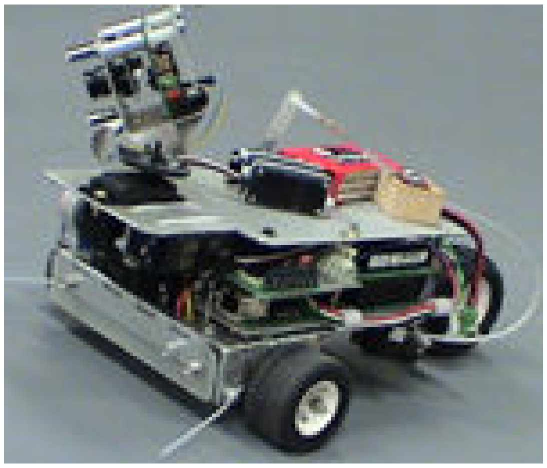
Figure 1 Miniature autonomous robot, showing overall structure and camera. The computer platform is visible as two layers of circuit board between the wheels and the camera tower.

The system peripherals include motors, motion sensors, a camera and actuators. The computer system is a TS5500 X86-based Pentium class system on a PC104-compliant miniature motherboard with multiple and varied IO capabilities. Communication with the system is by serial port, Ethernet cable or wireless networking. The system can run various operating systems but usually is used with a variant of the Linux OS.

The tasks that the miniature vehicle is required to complete include navigation around a track, obstacle detection and avoidance, seeking colored balls, capturing and storing them and then returning to a base station and discharging the balls into a 'goal'. All operations must be completed autonomously, using only the on-board intelligence programmed into the system. The project had a number of real-time requirements. Sensor and camera data has to be processed fast enough to make navigation, motor-speed and steering decisions to prevent the car from going off a ledge or colliding with a wall. When the video camera found a correctly colored ball, the car has to change course quickly so the ball stays in the camera's view. The motors themselves are speed controlled.

The Solution

The student team tried a few language alternatives, such as C, C++ and C#, before selecting Ruby. Initially the student team, with one of the authors as lead programmer, developed the software structure that would be used for the rest of the design. A set of libraries were created. The libraries were divided into these layers, starting at the bottom:

- Utilities - wrappers, methods, and other useful code that is not at all hardware-specific
- *EmbeddedX86.Ts5500* - wraps the Ts-5500 card itself, and nothing more
- Devices - all the hardware devices that can be attached to the target system
- *Target system* - wraps the whole embedded system, comprising of both the Ts-5500 card and the devices attached to the IO ports
- *"`car`"* - the specific classes used to get the car to drive around the track and follow programmed instructions

Code was designed to be reusable. Ruby's cross-platform nature means that any of the code could be ported to other embedded systems with minimal effort. Reusability was supported in the application development by writing libraries in this layered fashion to maximize their usefulness to other developers.

The Object-oriented features of Ruby were used to create the libraries. Class libraries defined all the functions the car would use, using several layers of abstraction, so that in the future when the availability of add-on devices changes, or if the IO subsystem is changed, then only the hardware-specific library layer would need to be re-written.

One of the problems with a cross-platform language is accessing the hardware, however library functions are available. Ruby does not support memory-mapped I/O or serial port control (baud rate, etc.). Two very convenient Ruby extensions[6] added support for these operations: `ioport(i-o_port_access)` and `serialport (ruby-serialport)`. By cross-compiling these for the target system, and copying the resulting binary files onto the system, Ruby was able to access programs support for serial communication and memory-mapped I/O on the target system very easily. For example communication routines written in C program were much shorter in Ruby. There are supporting libraries, of course, but they are short and readable.

Once the libraries were created the team found that software development was much easier. Software could be written at a fairly high level of abstraction. Direct control of IO was possible through the library functions, code development and test cycles were very fast since design intentions could be very easily expressed in terms of the available classes and they could be implanted immediately using the on-board interpreter. This capability became very important at various stages of the project, such as tuning delay loops for accurate navigation and for motor speed control. In fact, the ability to very quickly write small pieces of code and test them became a hall-mark of this team's development and differed noticeably from previous years.

Support for real-time applications was still constrained by the capability of the hardware platform and the Linux OS. Ruby's compliance with threaded code was fast enough for most navigation tasks and for real-time object identification but the load on the CPU was too large to permit direct Pulse-Width Modulation (PWM) control of the motors for speed control. PWM control of multiple motors requires toggling individual IO ports at 15 KHz or more, and recalculating delays until the next toggle point dynamically. The combination of Ruby and hardware could not do this. This task was delegated to a separate hardware chip with built-in PWM capability.

Summary and Conclusions

Overall the implementation of Ruby as a embedded systems development language was very successful. The inherent programmer-friendly features of Ruby and its object-orientation made software development clean and relatively bug free. The interpreted nature of the language allowed for very fast development cycles directly on the target system.

The software structure that has been developed is extensible and portable. Ruby does not inherently support real-time programming but a combination of IO libraries and with some help from hardware allowed all the real-time tasks to be handled well. This disadvantage is outweighed by the significant advantages.

In future the usefulness of OO scripting languages for embedded systems development will increase. Key improvements that could help the process would include:

- Integration of the language with a real-time operating system allowing for true real-time control
- Availability of a full binary-level compiler to increase execution speed of time critical modules.

We expect future development with this platform or similar language/OS/hardware platforms in the future to extend the capabilities of the system.

Bibliography

1. Baas, B *Ruby in the CS Curriculum.* Journal of Computing Sciences in Colleges, Vol 17, No. 5 (April 2002) , Pages: 95 - 103

2. Baird Kevin C. *Generating music notation in real time* Linux Journal, Vol 2004, Issue 128 (Dec 2004) Page 3

3. Helps R. *Teaching Embedded Systems From Eight Bits to Operating Systems and Networks,* Proceedings, ASEE Annual Conference 2002 (Montreal). Session 3647

4. Matsumoto, Yukihiro. *The Ruby Programming Language* Jun 12, 2000. http://www.informit.com/articles/article.asp?p=18225&redir=1&rl=1

5. RubyCentral, *What is Ruby?* http://www.rubycentral.com/misc/intro.html

6. RAA *Ruby Application Archive,* http://raa.ruby-lang.org/ Accessed April 2006.

7. Singer, Michael. *PC milestone--notebooks outsell desktops*. Cnet News.com. http://news.co.com/PC+milestone--notebooks+outsell+desktops/2100-1047_3-5731417.html

8. Spinellis, D. *Java makes scripting languages irrelevant* Software, IEEE Vol 22, Issue 3, May-June 2005 Page(s):70 – 71

9. Venner, Bill, *The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I*. Artima.com, On-line interview, September 29, 2003, http://www.artima.com/intv/ruby4.html

10. Venner, Bill, *The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part II*. Artima.com, On-line interview, p3, November 17, 2003, http://www.artima.com/intv/tuesday.htm