The Owl Embedded Python Environment

Microcontroller Development for the Modern World

THOMAS W. BARR AND SCOTT RIXNER

Thomas W. Barr is a fifth-year PhD student at Rice University in the Department of Computer Science. He received his BS degree in engineering and music from Harvey Mudd College in 2008. He has published research in computer architecture, embedded systems software, and high-performance computing. Outside of graduate school, he has worked as an expert witness and in litigation support for intellectual property.
twb@rice.edu

Scott Rixner is an Associate Professor of Computer Science at Rice University. His research focuses on the interaction between operating systems, runtime systems, and computer architectures; memory controller architectures; and hardware and software architectures for networking. He works with both large server-class systems and small embedded systems. Prior to joining Rice, he received his PhD from MIT.
rixner@rice.edu

Imagine my typical day. My alarm clock goes off and I immediately check my email on my iPhone. I stumble out of bed, make myself a cup of coffee and watch the morning news that my TiVo kindly recorded for me. I unlock my car and am presented with a map that shows the traffic on my route to work.

While there were three obvious computers in this little story, there are dozens more unsung heroes you probably didn't even think about. My alarm clock, my coffee maker, my TiVo remote, and my car key all are built around a microcontroller. My car is built around dozens of them. This article is about programming these very real, very complex computer systems.

Modern microcontrollers are almost always programmed in C. Applications run at a very low level without a real operating system. They are painfully difficult to debug, analyze, and maintain. At best, a simple real-time operating system (RTOS) is used for thread scheduling, synchronization, and communication [3]. These systems provide primitive, low-level mechanisms that require expert knowledge to use and do very little to simplify programming. At worst, they are programmed on the bare metal, perhaps even without a C standard library. As the electronic devices of the world become more and more complex, we absolutely have to do something to make embedded development easier.

We believe that the best way to do this is to run embedded software on top of a managed runtime system. We have developed and released as open source an efficient embedded Python programming environment named Owl. Owl is a complete Python development toolchain and runtime system for microcontrollers. Specifically, Owl targets systems that lack the resources to run a traditional operating system, but are still capable of running sophisticated software systems. Our work focuses on the ARM Cortex-M3 class of devices. These microcontrollers typically have 64–128 KB of SRAM, and have up to 1 MB of on-chip flash. Surprisingly, though, they are quite fast, executing at up to 100 MHz. This makes them more than fast enough to run an interpreter. These devices are absolutely everywhere; by 2015, ARM Cortex-M3-based systems are estimated to outsell x86 systems by a factor of 40.

Owl is a complete system that includes an interpreter, a programmer, an IDE, and a set of profilers and memory analyzers. Owl is derived from portions of several open-source projects, including CPython and Baobab. Most notably, the core runtime system for Owl is based on Dean Hall's Python-on-a-Chip (p14p) [2]. We support it on Texas Instruments LM3S9x9x Cortex-M3 microcontrollers as well as STM ST32F4 Cortex-M4 microcontrollers.

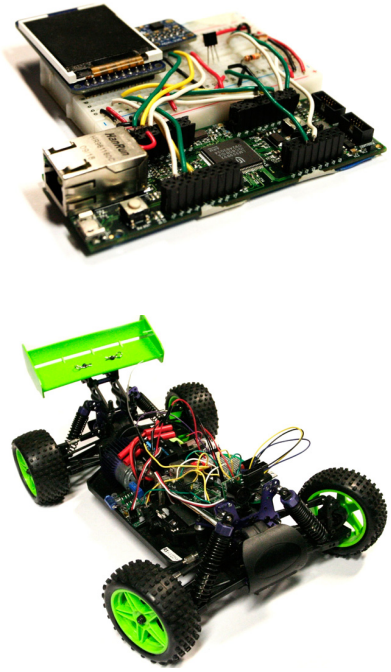

Figure 1: An artificial horizon (a) and an autonomous car (b) built with Owl

Owl demonstrates that it is possible to develop complex embedded systems using a high-level programming language. Many software applications have been developed within the Owl system, including a GPS tracker, a Web server, a read/write FAT32 file system, and an artificial horizon display. Furthermore, Owl is capable of running soft real-time systems; we've built an autonomous RC car and a pan-and-tilt laser pointer mount. These applications were written entirely in Python by programmers with no previous embedded systems experience, showing that programming microcontrollers with a managed runtime system is not only possible but easy. Additionally, Owl is used as the software platform for Rice University's r-one educational robot [5]. A class using this robot is now being taught for the third time, and groups of first-semester college students have been able to program their robots in Python successfully without any problems from the virtual machine. Moreover, at a demo, we had children as young as seven years old programming robots.

The cornerstone of this productivity is the interactive development process. A user can connect to the microcontroller and type statements to be executed immediately. This allows easy experimentation with peripherals and other functionality, making incremental program development for microcontrollers almost trivial. In a traditional development environment, the programmer has to go through a tedious compile/link/flash/run cycle repeatedly as code is written and debugged. Alternatively, in the Owl system a user can try one thing at the interactive prompt and then immediately try something else after simply hitting "return." The cost of experimentation is almost nothing.

This sort of capability is invaluable to both the novice and expert embedded programmer. While there are certainly many people in the world who are skilled in the art of low-level microcontroller programming, the process is and always will be expensive, slow, and error-prone. By raising the level of abstraction, we can allow expert embedded developers to spend their limited time making more interesting and complex systems, not debugging simple ones.

Finally, Owl is fun to use. Microcontrollers help put a lot of the joy back into programming because they make it possible to build real, physical systems. Go out and build a super-intelligent barbeque, a home automation controller, or a fearsome battle robot. We'll make sure that register maps and funky memory layouts don't get in your way.

The Owl System

Modern 32-bit ARM-based microcontrollers now have enough performance to run a relatively sophisticated runtime system. Such systems include eLua [1], the Python-on-a-Chip project [2], and Owl, our project. These systems execute modern high-level languages and provide system support for everything from object-oriented code to multithreading to networking.

The Owl runtime natively executes a large subset of the standard Python bytecodes. This allows us to use the standard Python compiler and even to execute many existing programs. Owl natively supports multithreading and includes a novel feature to call C functions. This is critical on microcontrollers because programmers must call into a C driver library to control peripherals. Our system allows Python programmers to call functions in these libraries exactly as if they were standard Python functions.

The high-level design of Owl allows a user to build embedded systems without having low-level knowledge about microcontrollers. Users start by connecting to the microcontroller from a standard desktop computer over USB. Owl then shows a Python prompt, just like regular Python. When the user types a statement, the host computer compiles the statement into bytecodes, sends it to the controller where it is executed. Any resulting output is sent back for display and the process repeats.

Along the way, Owl automatically manages resources on the controller. The prompt is a built-in feature, as is the thread scheduler, the memory manager and countless others. A user doesn't need to write code on the microcontroller to connect over USB; it just works. The user doesn't need to allocate memory for a variable, nor remember to free it later. These automatic features are nothing particularly new in large computer systems, but they are nearly unheard of in the embedded space. Higher level languages including Python are heavily used in everything from package managers to cell phones to scientific computing. We believe that the time has come to use these ideas to make microcontroller programming easier.

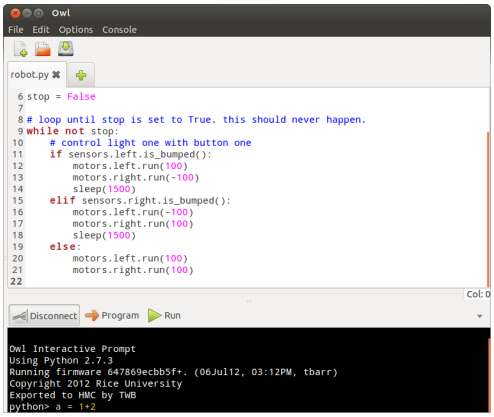

Figure 2: Owl can be programmed from the command line, or using our cross-platform, Arduino-like IDE

Does It Work?

Often, people tell us that building a high-level language interpreter for microcontrollers must be "impossible." There's simply not enough RAM or enough flash or enough speed! We've found this to be false and refer you to our research paper on the Owl system [4] for a more in-depth look at these issues. We would like to take this opportunity to address some specific questions people have asked us.

Question: Surely a complete virtual machine takes up a lot of flash?

Indeed, flash memory that stores programs and data is a precious resource on a microcontroller; however, it turns out that Owl doesn't need much more flash than a traditional RTOS does.

The Owl VM itself is actually quite small, around 35 KB, and it contains all of the code necessary for manipulating objects, interpreting bytecodes, managing threads, and calling C functions. When compared to the 256 KB or more available on a microcontroller, this is not much larger than the so-called "light weight" FreeRTOS, which requires 22 KB.

The largest fraction of this space is used by C libraries, such as a network stack, a USB library or specialized math routines. The size of the standard Owl distribution is on the order of 150 KB, the majority of which are compiled C libraries. Any C application that uses these libraries would have to include them, just like Owl. Therefore, the overhead incurred by Owl for any complex application that utilizes a large set of peripherals and C libraries will be quite low.

Question: Okay, but won't it be very, very slow?

This depends greatly on what you're doing. At one extreme, the bytecode to add two numbers together takes about 10 μs. This is 500 times slower than the single cycle 32-bit add that the Cortex-M3 is theoretically capable of; however, the interpreter supports much more complicated operations, including function calls into native C code, which incur far lower overhead.

Overall, the proof is in what you can build. We've implemented many applications using Owl, and the performance has always been sufficient. We've connected our controller to a GPS receiver, three-axis accelerometer, three-axis MEMS

gyroscope, digital compass, LCD display, microSD card reader, ultrasonic range finder, steering servo, and motor controller. We then built an artificial horizon display (using the display and accelerometer), a GPS tracker (using the GPS, compass, microSD, and display), and an autonomous RC car (using the gyroscope, GPS, range finder, steering servo, and motor controller). All of these applications work just fine.

Question: Aren't all embedded systems real-time? You've said nothing about building real-time systems.

There is some truth to this. Owl is not a hard real-time system. Owl provides no guarantees about when code will run; however, most real-time systems only need soft real-time guarantees. There is no loss of life if a thermostat takes an extra few milliseconds to switch on.

Our autonomous car is one such system. The car is based on an off-the-shelf remote controlled car that has had the R/C receiver disabled. Instead, a microcontroller running Owl drives the outputs. The car senses its position with GPS, navigating to waypoints. Meanwhile, it monitors a rangefinder to detect obstacles and uses a gyroscope to drive straight. All of these functions are "real-time systems," and they all work to form a functional autonomous car.

Question: Well, okay, but what about garbage collection? Doesn't that ruin everything?

The impact of garbage collection on embedded workloads is much smaller than even we expected it to be.

Owl's garbage collector (GC) is a simple mark-and-sweep collector that occasionally stops execution for a variable period of time. We found that the garbage collector has the largest impact on applications that use complex data structures, such as CPU benchmarks that we ported to Owl. These structures take a long time to traverse during the mark phase of collection. Additionally, there are a large number of objects in total, slowing the sweep phase. Overall, garbage collection can take up to 65 ms or 41% of execution time on these types of programs.

The embedded systems we have examined use much simpler data structures. This means that GC runs more rarely, and for shorter periods of time. For the worst-case embedded workload we tested, this takes 8 ms on average, only 11% of the application's running time.

Further reducing the impact of GC on embedded workloads, our virtual machine runs the collector when the system is otherwise idle. In an event-driven system, there are often idle times waiting for events. We take advantage of those idle times to run the garbage collector preemptively. In practice, this works quite well. For example, when we tested our autonomous car, all GC happened during sleep times. In other words, the garbage collector never interrupted or slowed useful work.

Sounds great for a beginner. What about me, though? I've been writing low-level assembler and C for decades! I can already build microcontroller applications. Why do we need yet another development system?

Of course it is possible for one skilled in the art to build a complex embedded system using low-level programming tools. Owl itself is an example of such a system;

however, just because it's possible to build a system using these tools doesn't mean we can't do better!

The time of an expert embedded systems programmer is a precious commodity. A higher level language makes building complicated algorithms and data structures easier. Tools such as profilers and interactive prompts make exploring the performance and behavior of a system possible. These tools mean that a programmer has to spend less time debugging and can spend more time building products. In the time that it might take an expert engineer to build a programmable thermostat in C, the same expert engineer might be able build a machine learning thermostat in Python. The fact that an expert programmer is capable of repeatedly writing low-level code doesn't mean that it should be necessary.

Perhaps more critically, though, raising the level of abstraction can make programs more reliable. Programs are simpler, so they are less error-prone. Owl can detect internal errors, such as stack overflow, turning a catastrophic memory corruption bug into a properly detected and reported error condition. Owl can detect programming bugs, such as array bounds violations, reporting a sensible error before a device is deployed into the field.

Finally, Owl allows users to reprogram part or all of their devices easily, sometimes without even needing to restart the controller. This means that deployed devices can be tested, modified, fixed, and upgraded without having to take critical systems offline. This would be extremely difficult to accomplish with normal embedded toolchains and is rarely, if ever, done.

We don't see the Owl system as a replacement for skilled embedded systems programmers. Rather, we see it as a productivity multiplier. We are skilled embedded systems programmers—we built the Owl system—yet we can accomplish a lot more a lot faster using the Owl system itself!

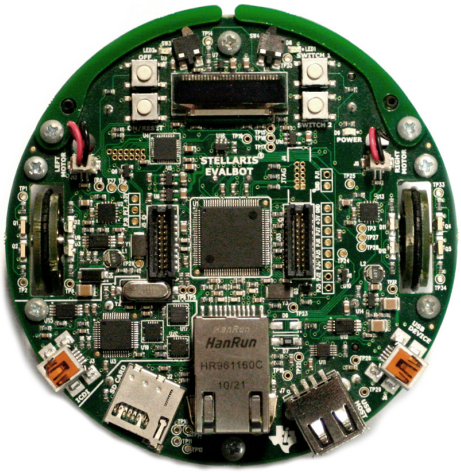

Figure 3: The Texas Instruments Evalbot is a commercially available robot that can be programmed using Owl

Using Owl

Getting started with Owl is easy. Here, we show a simple robotic example using an off-the-shelf Texas Instruments "Evalbot." The Evalbot is a simple, two-motor turtle, or Roomba-like, robot. You can download all the software we use here and find links to the hardware from our Web site. We demonstrate how to use the interactive prompt to control the hardware using both prepackaged libraries as well as through low-level driver library calls. Finally, we show how to flash a program onto the robot and run it while disconnected from the host computer.

First, we assume that the Owl tools distribution is installed onto a UNIX-like system and that the robot is connected over USB. From a prompt, type:

```
$ mcu interactive
Owl Interactive Prompt
Using Python 2.7.3
Running release firmware 0.01. (05Sep12, 01:37AM, twb)
mcu> 1+1
2
mcu>
```

This prompt looks and works just like any other Python prompt. You can assign values to variables, evaluate expressions, call functions, and even define functions and classes. Of course, this would be a very boring robot if we didn't dig into controlling the hardware. The Owl distribution for the TI Evalbot contains prebuilt

modules to control some of the robot peripherals. As a simple example, let's play with the motors module. After each statement, the robot responds immediately. First, we run the left motor 100% forward, then 100% backward, then we stop it:

```
mcu> import motors
mcu> motors.left.run(100)
mcu> motors.left.run(-100)
mcu> motors.left.run(0)
mcu>
```

Suppose, however, that you were designing your own device. You won't have access to high-level peripheral libraries for it, so you'll need to make calls directly into the low-level driver library. Owl makes this relatively easy by making those libraries appear just like any other Python module. In fact, the conversion from C to Python is so transparent that you can use the original C documentation provided by the microcontroller vendor.

Suppose we are trying to read the current value of the bump sensor, which is just a simple push button attached to a general purpose I/O pin. We will first need to enable the GPIO module. In C, we would do this with the line `SysCtlPeripheralEnable (SYSCTL_PERIPH_GPIOE)`. In Python, this translates very simply. In fact, we can call this function from the prompt:

```
mcu> import sysctl
mcu> sysctl.PeripheralEnable(sysctl.PERIPH_GPIOE)
mcu>
```

Similarly, we will call `gpio.PinTypeGPIOInput` and `gpio.PadConfigSet` to configure the correct pin. Finally, we will read the value by calling `gpio.PinRead`. Putting all of this together, we can write a simple program to emulate the "bounce against walls" behavior of a Roomba:

```
# robot.py
import motors, sysctl, gpio, sys

# initialize the bump sensors
sysctl.PeripheralEnable(sysctl.PERIPH_GPIOE)

gpio.PinTypeGPIOInput(gpio.PORTE_BASE, gpio.PIN_0) # right bumper
gpio.PinTypeGPIOInput(gpio.PORTE_BASE, gpio.PIN_1) # left bumper

gpio.PadConfigSet(gpio.PORTE_BASE, gpio.PIN_0, gpio.GPIO_STRENGTH_2MA,
                      gpio.GPIO_PIN_TYPE_STD_WPU)
gpio.PadConfigSet(gpio.PORTE_BASE, gpio.PIN_1, gpio.GPIO_STRENGTH_2MA,
                      gpio.GPIO_PIN_TYPE_STD_WPU)
# loop forever
while True:
   if gpio.PinRead(gpio.PORTE_BASE, gpio.PIN_1): # bumped!
      motors.left.run(100) # full forward
      motors.right.run(-100) # full backwards
      sys.sleep(1500) # wait 1500 ms
   elif gpio.PinRead(gpio.PORTE_BASE, gpio.PIN_0):
      motors.left.run(-100)
      motors.right.run(100)
      sys.sleep(1500)
```

```
else: # go straight ahead.
    motors.left.run(100)
    motors.right.run(100)
```

We can now flash this program as a module onto the robot. This process erases all user-programmed modules from the device and programs one or more new files. In this case, we only program one module, robot.py, by resetting the robot and calling mcu robot.py at the UNIX prompt. This module could be imported (and therefore executed) from the Python prompt, or it can be run in stand-alone mode. When the microcontroller starts up, it checks to see if it is connected to USB. If it is not, it automatically runs the primary module, which was the first module listed when the device was programmed.

Now, our robot is free to explore the world!

Next Steps

There are unfathomable numbers of microcontrollers in the world, but for some reason, we don't think of them as "real" computer systems. While we've developed incredible programming environments for cell phones and Web apps, we still program most embedded systems as if they were PDP-8s. As a result, we have countless lines of unportable, unreliable, and unsafe code that we use every day. Programmers have very little visibility over what their software is doing and must debug software using multimeters and oscilloscopes. This is expensive, painful, and error-prone. Furthermore, the process is not really all that fun. Let's start thinking of these tiny devices as the fully fledged computer systems that they are. We think Owl and the other open-source projects are a great start. They enable interactive software development that's high-level, safe, and easy as opposed to the current approach that is more akin to flipping front-panel switches on an Altair.

Go try Owl out! Microcontroller development boards are cheap nowadays. Numerous boards are available for less than $100 and some for as low as $15. Companies such as SparkFun Electronics and AdaFruit Industries sell a lifetime worth of peripherals that are easy to work with. Check out our Web site at http://embeddedpython.org/ for links to these products, buy some of them, download Owl, and get out there and build something. We promise that you'll have a lot of fun!

References

[1] eLua: http://www.eluaproject.net/.

[2] Python-on-a-Chip: http://code.google.com/p/python-on-a-chip/.

[3] T.N.B. Anh and S.-L. Tan, "Real-Time Operating Systems for Small Microcontrollers," *IEEE Micro,* vol. 29, no. 5), 2009.

[4] T.W. Barr, R. Smith, and S. Rixner, "Design and Implementation of an Embedded Python Runtime System," USENIX ATC, 2012.

[5] J. McLurkin, A. Lynch, S. Rixner, T. Barr, A. Chou, K. Foster, and S. Bilstein. A Low-Cost Multi-Robot System for Research, Teaching, and Outreach. Distributed Autonomous Robotic Systems, pages 597–609, 2010.