Design and Implementation of an Embedded Python Run-Time System

Thomas W. Barr, Rebecca Smith, Scott Rixner
Rice University, Department of Computer Science
USENIX Annual Technical Conference, June 2012
The computing landscape
Microcontrollers: They’re everywhere.

Uncountable? ~10bil ~1bil ~1000s
Microcontrollers: They’re everywhere.

- Virtually no run-time support (*yet*)
- Storage?
- Memory allocation?
- Interrupt handlers?
- Console?
Microcontrollers: They’re everywhere.

- 32-bit ARM Cortex-M3
  - 64-128KB RAM
  - Up to 512KB Flash
  - 50-100MHz
- Interpreters on microcontrollers?
  - Java Card
  - eLua
  - Python-on-a-chip
Owl: A Development Environment for Microcontrollers

- Run-time and toolchain
  - Open-source
    - Based on p14p
  - Interactive prompt
  - Profilers
  - Programmers
  - A lot more
- It works. Today.
- Focus on two parts today
  - Store Python objects
  - Call C functions
python> a = 3
python> print a + 1
4
python> import lights
python> lights.one.on()
python> def f(x):
    return x * x
python> f(3)
9
python>
Representing programs

- Python source code

def foo():
a = 'foo'
b = (42, 'bar')
c = 27 + 3
Representing programs

- Compiler builds code objects
  - Bytecode to represent user program

```python
def foo():
    a = 'foo'
    b = (42, 'bar')
    c = 27 + 3
```

Bytecodes:

```
LOAD_CONST 0
STORE_NAME 0
...
```
Representing programs

- Must include data along with code
  - Constants, names, etc.
  - .pyc file, .class file

Bytecodes:

LOAD_CONST 0
STORE_NAME 0
...

Constants:

0: 'foo'
1: (42, 'bar')
...

rice computer architecture group
Loading programs

File:

0: 'foo'
1: (42, 'bar')
...

Memory
Loading programs

File:

0: 'foo'
1: (42, 'bar')
...

Memory

str:
Loading programs

File:

0: 'foo'
1: (42, 'bar')
...

Memory

str: 'foo'
Loading programs

File:

0: 'foo'
1: (42, 'bar')
...

Memory:

int: 42
str: 'foo'
str: 'bar'
tup: _, _
Loading programs

- Intermediate files
  - Python .pyc file, Java .class file, etc.
  - Loaded at runtime, then executed
- Load before execution
  - Copy and transform
    - Link compound objects with references
  - Makes sense on x86
    - RAM is plentiful
- Doesn’t on microcontroller
  - Can’t afford to copy anything
Loading programs

- Owl memory image
  - Unique on interpreted embedded systems
  - Store objects in form that is used by the VM
    - Object headers
    - Byte-order
  - How do we store compound objects?

Memory image

- int: 42
- str: 'foo'
- str: 'bar'
- tup: _, _
Loading programs

- Packed tuples
  - Store objects inside of nested containers
  - Transplantable
    - Computer → Controller
    - Controller → Controller
  - No pointers to fix up
Memory images

- p14p, eLua use dynamic loaders
  - Objects copied from flash to RAM
- Owl uses memory images
  - Can stay in flash, not SRAM
  - 512KB flash vs. 96KB SRAM
- Results
  - 4x reduced SRAM footprint
Calling C library functions

- **C is a necessary evil**
  - Peripheral I/O libraries
    - Generally vendor provided
  - Must provide a way for Python to call C

- **Two techniques compared**
  - Automatically expose functions from .h file
  - Trade-offs in space and speed
Calling C library functions

- User cannot directly control execution
  - In an interpreter loop
  - Need some way to let programmer break out of loop

```c
bytecode = *IP;
switch bytecode:
    case BINARY_ADD:
        // add two Python ints
        // together
        IP++;
    case JUMP:
        // change IP
    case CALL_FUNCTION:
        // create new frame
        // change IP to new frame
```
Calling C library functions

```c
bytecode = *IP;
switch bytecode:
  case BINARY_ADD:
    // add two Python ints together
    IP++;
  case JUMP:
    // change IP
  case CALL_FUNCTION:
    // create new frame
    // change IP to new frame
```
Calling C library functions

bytecode = *IP;
switch bytecode:
    case BINARY_ADD:
        // add two Python ints together
        IP++;
    case JUMP:
        // change IP
    case CALL_FUNCTION:
        if (func_name == "uart_init") {
            call_uart_init();
        } else {
            // create new frame
            // change IP to new frame
        }
Calling C library functions

bytecode = *IP;
switch bytecode:
    case BINARY_ADD:
        // add two Python ints together
        IP++;
    case JUMP:
        // change IP
    case CALL_FUNCTION:
        if (func_name == "uart_init") {
            call_uart_init();
        }
        else if (func_name == "uart_send") {
            call_uart_send();
        }
        else {
            // create new frame
            // change IP to new frame
        }
call_uart_init()

/* Variable declarations */
PmReturn_t retval = PM_RET_OK;
pPmObj_t p0;
uint32_t peripheral;

/* If wrong number of arguments, raise TypeError */
if (NATIVE_GET_NUM_ARGS() != 1) {
    PM_RAISE(retval, PM_RET_EX_TYPE);
    return retval;
}

/* Get Python argument */
p0 = NATIVE_GET_LOCAL(0);

/* If wrong argument type, raise TypeError */
if (OBJ_GET_TYPE(p0) != OBJ_TYPE_INT) {
    PM_RAISE(retval, PM_RET_EX_TYPE);
    return retval;
}

/* Convert Python argument to C argument */
peripheral = ((pPmInt_t)p0)->val;

/* Actual call to native function */
SysCtlPeripheralEnable(peripheral);

/* Return Python object */
NATIVE_SET_TOS(PM_NONE);
return retval;

- Wrappers for functions
  - Interpreter calls wrapper
  - Wrapper converts arguments
  - Calls function
  - Wrapper converts result
  - Returns result on stack

- Call C as you call Python
import gpio

class Output:
    def __init__(self, portpin):
        self.port = PORTS[portpin[0]]
        self.pin = PINS[portpin[1]]

        # turn on GPIO module
        init_port(portpin[0])

        # configure pin for output
        gpio.GPIOPinTypeGPIOOutput(self.port, self.pin)

    def write(self, value):
        if value:
            # turn pin on
            gpio.GPIOPinWrite(self.port, self.pin, self.pin)
        else:
            # turn pin off
            gpio.GPIOPinWrite(self.port, self.pin, 0)
import gpio

class Output:
    def __init__(self, portpin):
        self.port = PORTS[portpin[0]]
        self.pin = PINS[portpin[1]]

        # turn on GPIO module
        init_port(portpin[0])

        # configure pin for output
        gpio.GPIOPinTypeGPIOOutput(self.port, self.pin)

    def write(self, value):
        if value:
            # turn pin on
            gpio.GPIOPinWrite(self.port, self.pin, self.pin)
        else:
            # turn pin off
            gpio.GPIOPinWrite(self.port, self.pin, 0)
Calling C library functions

- Wrapped functions
  - Manually write wrappers
    - p14p, eLua
  - Autowrapping
    - Similar to SWIG
- Simple
  - Any compiler
  - Any architecture
  - Lots of duplicated code

/* Variable declarations */
PmReturn_t retval = PM_RET_OK;
pPmObj_t p0;
uint32_t peripheral;

/* If wrong number of arguments, raise TypeError */
if (NATIVE_GET_NUM_ARGS() != 1) {
    PM_RAISE(retval, PM_RET_EX_TYPE);
    return retval;
}

/* Get Python argument */
p0 = NATIVE_GET_LOCAL(0);

/* If wrong argument type, raise TypeError */
if (OBJ_GET_TYPE(p0) != OBJ_TYPE_INT) {
    PM_RAISE(retval, PM_RET_EX_TYPE);
    return retval;
}

/* Convert Python argument to C argument */
peripheral = ((pPmInt_t)p0)->val;

/* Actual call to native function */
SysCtlPeripheralEnable(peripheral);

/* Return Python object */
NATIVE_SET_TOS(PM_NONE);
return retval;
Foreign Function Interface

- Based on libffi
  - Calls C functions with arbitrary signature
  - Implements calling convention
    - Custom port to Cortex-M3
- eFFI
  - Functions statically linked into program
  - Function pointer tables instead of DWARF symbol headers
    - Compatible with statically linked embedded system
  - No duplicated code
- **eFFI smaller**
  - VM 2KB larger
  - Eliminates 21KB of wrappers
- **Wrapper functions simpler**
  - 2x faster, largely irrelevant in practice
Owl: A Development Environment for Microcontrollers
Yes, you can run a managed run-time on a microcontroller.

- It works, right now.
  - Autonomous car
  - Toys
  - Courseware
    - ENGI128 at Rice
  - Cookware
  - ...
Yes, you can run a managed run-time on a microcontroller.

It works, right now.
- Autonomous car
- Toys
- Courseware
  - ENGI128 at Rice
- Cookware
- ...

Owl: A Development Environment for Microcontrollers
Yes, you can run a managed run-time on a microcontroller.

It works, right now.

- Autonomous car
- Toys
- Courseware
  - ENGI128 at Rice
- Cookware
- ...
Owl: A Development Environment for Microcontrollers

- Interesting area for research
  - Two innovations presented here
  - More in paper
  - Far more in the future?
    - Compare to C?
    - To MATLAB?

- Vital area for research
  - Microcontrollers are getting more common
  - Larger peripheral set is making programming harder
    - We can and must reverse this trend
Owl: A Development Environment for Microcontrollers

embeddedowl@gmail.com