



esc

EMBEDDED SYSTEMS CONFERENCE '15

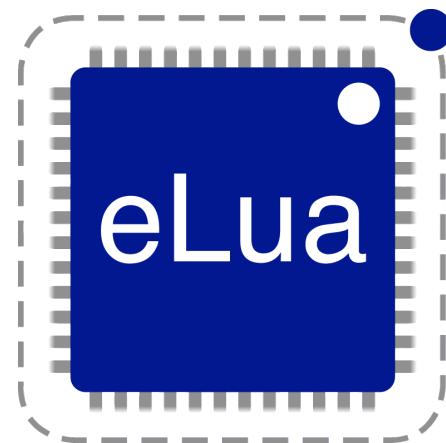


BOSTON CONVENTION CENTER
BOSTON, MA
MAY 6-7, 2015



UBM

Rapid Application Development with Embedded Lua



James Snyder, Board Member and Core Developer, eLua Project

Traditional Tools

- Development Suite
 - Large footprint, not always cross-platform
 - Free options are vendor-specific
- Language
 - ASM, C, C++
- Development cycle
 - Compile, Flash, Test/Run

Traditional Tools

Disadvantages

- Steep learning curve
- Portability
- Code complexity

Advantages

- Low-level access/control
- Deterministic behavior
- Performance

New Traditional Tools: mbed

- Development Suite
 - Online IDE/compiler, local IDE/compiler
- Language
 - C, C++ with cross-platform HAL, networking, components, etc..
- Development cycle
 - Compile, Download, Copy, Test/Run

Alternative

- Development Environment
 - Self-hosted, local IDE or RPC
- Language
 - High level, cross platform with HAL
 - Easy to extend in C
- Development cycle
 - Type, hit enter, see result – compiler is on the MCU

Disadvantages

- Performance
- High memory usage
- Dynamic memory allocation
- Non-deterministic execution time
 - Virtual Machine with Garbage Collection
 - Interrupts handled with VM-scheduled queuing



Advantages

- Performance
- Low barrier to get started / shallow learning curve
- Interactive development and debugging
- Update behavior/logic without re-flash
- Easy end user or integrator customization

Disadvantages?

Performance & Memory

- 32-bit MCUs
 - Inexpensive
 - Fast
 - Lots of SRAM + external mem

Determinism & Dyn. Allocation

- Some applications are out:
 - ECUs and other Real-Time
- Lots of applications where it doesn't matter

Options

- Lua
 - eLua
- Squirrel
 - Electric Imp
- Python
 - MicroPython
 - Python On A Chip
 - SNAPpy
- FORTH
- JavaScript
 - Tessel
 - Espruino
- BASIC
 - BASIC Stamp
- BGScript (BlueGiga/SiLabs)

*Not all of these provide interactive programming

What is Lua?



- Powerful dynamic language
- Compact enough to run on a 32-bit microcontroller
- Written in ANSI C
- Simple, flexible C API
- Used for scripting in Photoshop Lightroom, numerous games and other tools

Syntax

- Types
 - nil, boolean, number, string, function, userdata, thread, table
 - All first class
 - Fundamental data structure is table
- a = **nil**
- b, c = **true**, 3.14159
- d = "Hello" .. "World"
- e = **function** () **print**("Hi!") **end**
- f = **coroutine.create**(e)
- g = {a=a,b=b,c=c,d=d,e=e}
- g.f = f

Control Structures

if elseif else

```
if a == 1 then
    print("1")
elseif a == 2 then
    print("2")
else
    print("error")
end
```

while

```
i = 1
while i <= 10 do
    print(i)
    i = i + 1
end
```

for

```
for i=1,10 do
    print(i)
end

t = { 1, 2, 3, 4,
5, 6}
for k in pairs(t)
do
    print(k)
end
```

break, return

```
for i=1,10 do
    if i = 5
then
    break
end

function test (a,b)
    return a*2,b/2
end
```

Closures

```
function newSquares ()  
    local i = 2  
    return function ()  
        i = i*i  
        return i  
    end  
end  
  
square = newSquares()  
print(square())
```

Co-routines

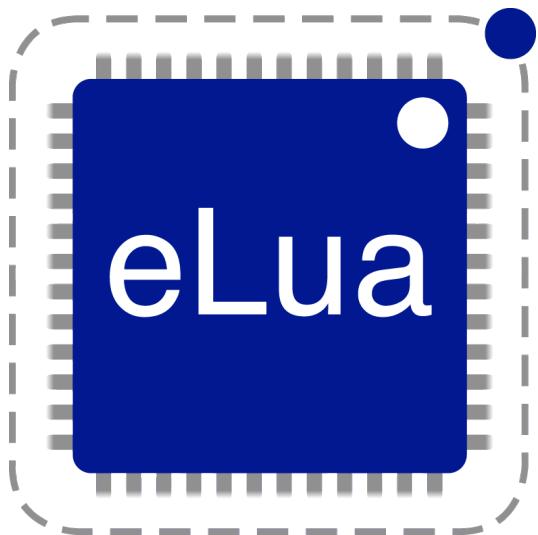
```
co = coroutine.create( function ()  
    t = { 1, 2, 3, 4, 5, 6}  
    for k in pairs(t) do  
        print(k)  
        coroutine.yield(k)  
    end  
end)  
  
coroutine.resume(co)
```



Learning Lua

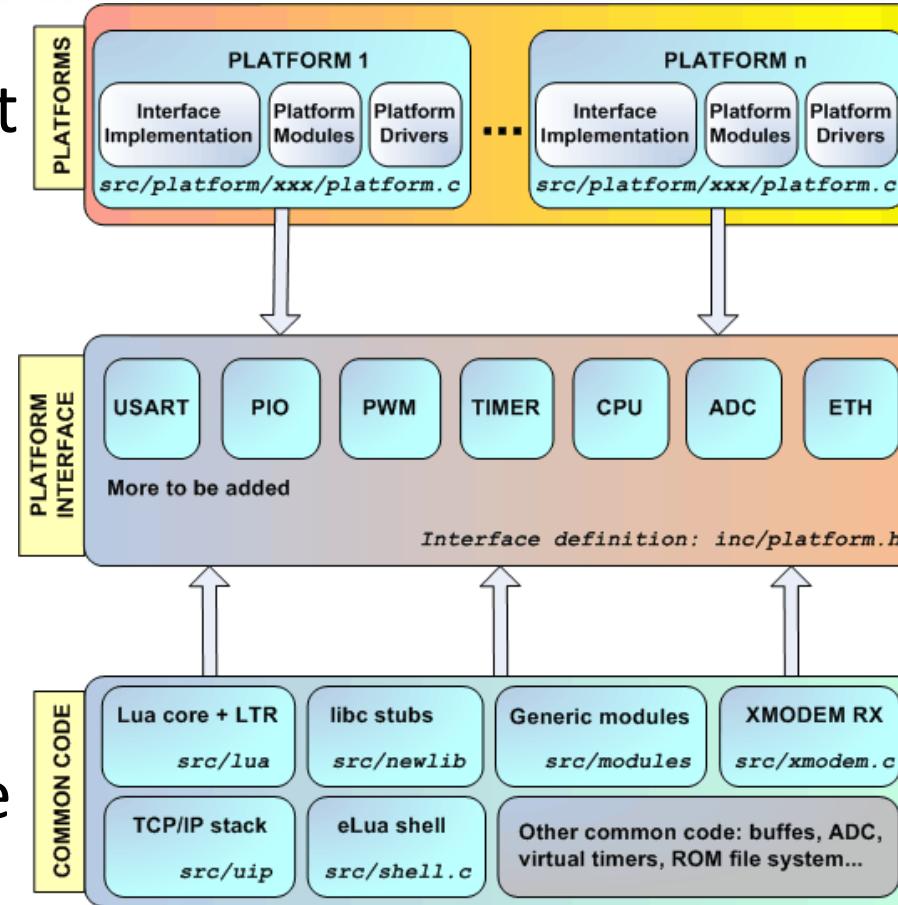
- Learn Lua in 15 Minutes
 - <http://tylerneylon.com/a/learn-lua/>
- Programming in Lua
 - <http://www.lua.org/pil/>
- Reference Manual
 - <http://www.lua.org/manual/5.1/>

What is eLua



- Full implementation of Lua for microcontrollers
- Hardware Abstraction Layer (C & Lua APIs)
- Filesystem and hooking of system calls
- Added Lua modules (bit manipulation, byte packing, etc..)
- Permissively Licensed (MIT for Lua & eLua core)

Platform Support



Common Code



Supported Boards

- [STM32F4DISCOVERY](#) (master)
- [NUCLEO-F401RE](#), [NUCLEO-F411RE](#) (master)
- [Mizar32](#)
- [mbed](#) (LPC1768)
- [ET-STM32 Stamp](#)
- Many others: <http://www.eluaproject.net/overview/status>

Lua Core Modifications



Lua 5.1.5

- + Emergency Garbage Collection
- + Read-only tables
- + Byte-code execution from flash
- + NaN packing for more efficient types
- + Integer-only support (64, 32-bit)

Common Code

- Filesystems
 - MCU Flash-based ROMFS & WOFS
 - SD/MMC over SPI with FatFS
 - Semihosting (MBED local mass storage over debug)
 - Remote Filesystem (over serial/network)
- Modules
 - bit manipulation, byte packing, rpc

Common Code

- Shell
 - start Lua
 - xmodem file transfer (recv)
 - filesystem interaction (ls, cp, cat, etc..)

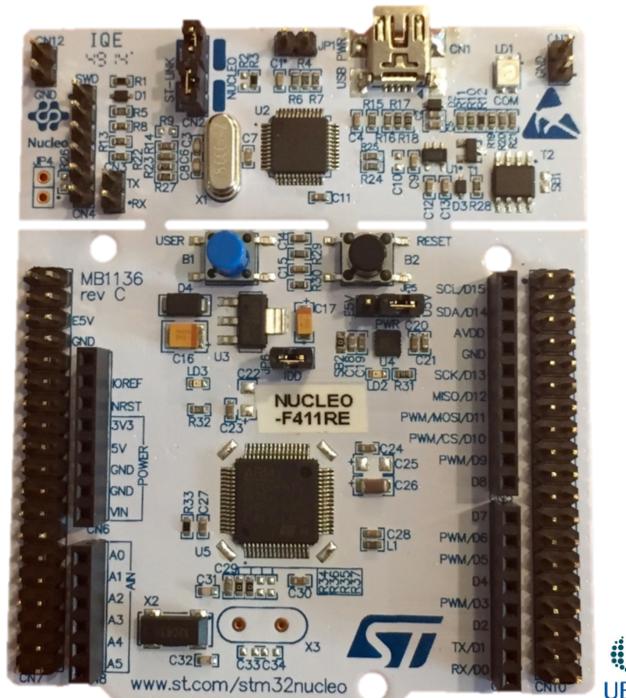


Startup

- LuaRPC (if enabled)
- autorun.lua (if present)
- Shell (if enabled)
- Lua REPL

Getting Started

- Get a supported board
 - NUCLEO-F401RE
 - NUCLEO-F411RE
- Generate a build:
 - <http://builder.eluaproject.net/>
- Download & copy bin file to mass storage
- Connect to Serial/COM port





Quick Demonstration

Example Project

- Lets blink some LEDs!
- WS2812B-based Neopixel LED array
 - Individually controllable RGB LEDs w/ 8-bits/channel brightness
 - Single-wire control using NZR protocol at 800 kHz
 - 24-bits/controller, GRB
 - 0=short high (~0.35 µs), long low (~0.8µs)
 - 1=long high (~0.7 µs), short low (~0.6µs)
 - ends with a reset (>=50 µs low)
 - Signals cascaded from controller to controller



Example Project

- Naïve implementation (bit bang)
 - Pure Lua not fast enough for GPIO toggling at 800 kHz update rate
 - Implement in C and provide a Lua API
 - Update algorithm & parameters on the fly

Code Available: <http://github.com/jsnyder/elua-escbos>



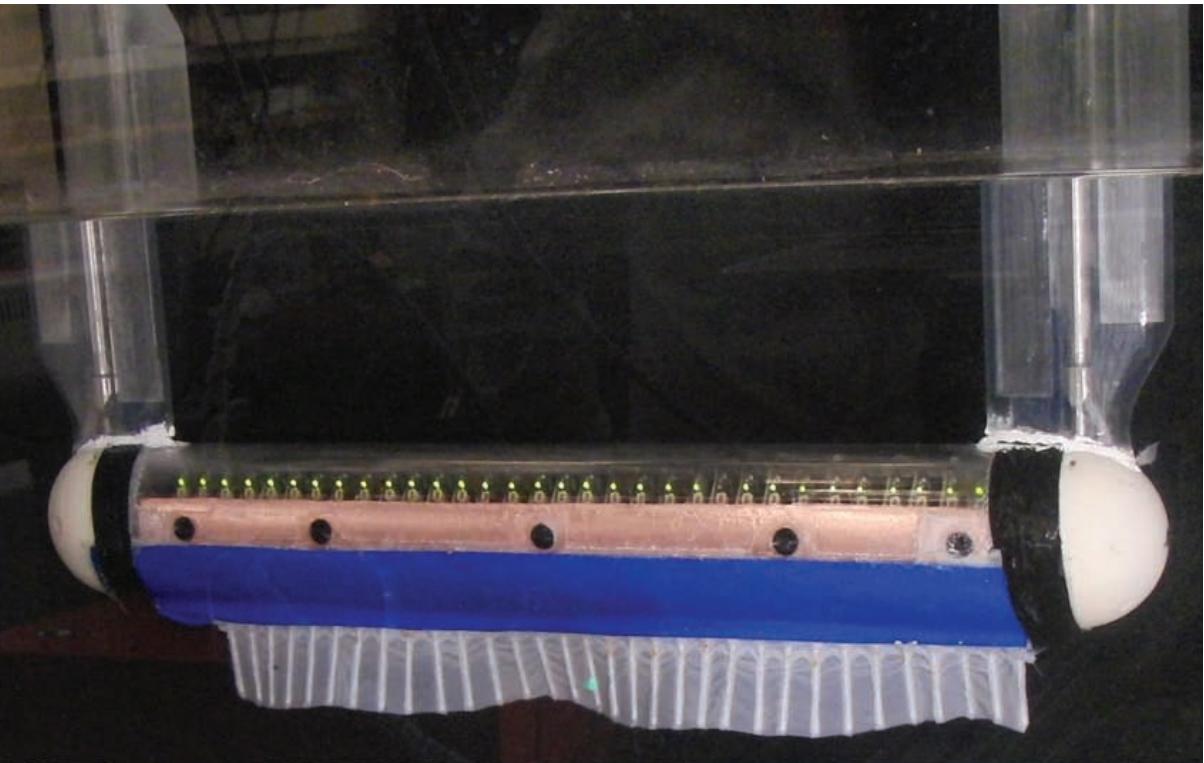
Live Demonstration



In The Field / Lab (with some lessons learned)



Taming GhostBot



Lessons Learned / Implementation

- 32 independent rays, controllable over CAN at 100 Hz
- eLua provided a dynamic environment where parameters and code could be updated on the fly over LuaRPC
- Performance constrained doing emulated floating point on 72 MHz Cortex-M3 (STM32F1)
- Upgraded to a higher clock rate Cortex-M4 to provide more headroom (STM32F4)

GhostBot Code Structure

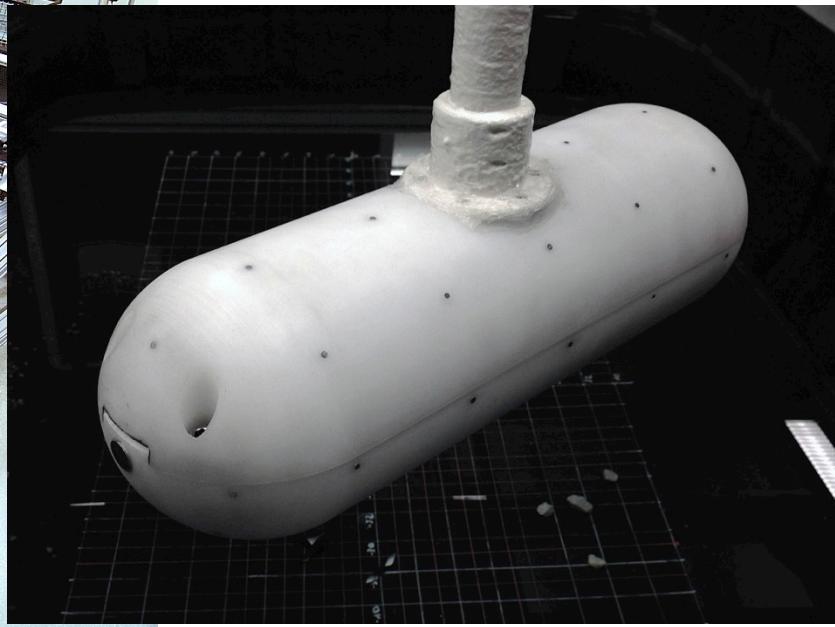
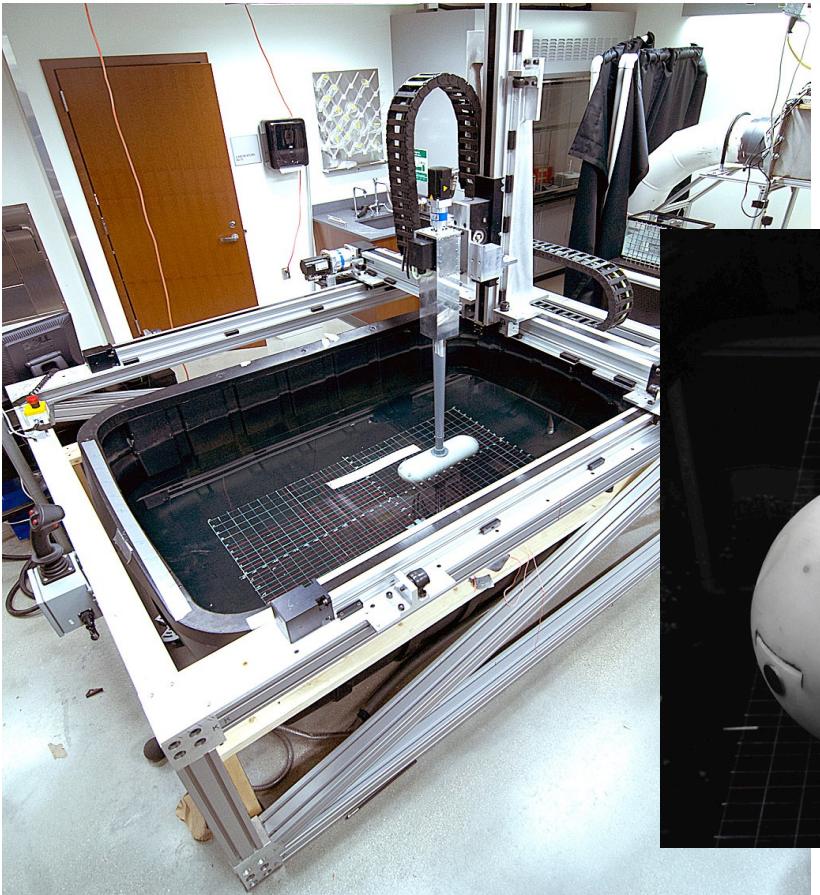
autorun.lua

```
local state = {}  
local start_time  
local stimer = tmr.SYS_TIMER  
  
cycletime = 10000  
  
function control()  
end  
  
handle = rpc.listen(uart.CDC,0)
```

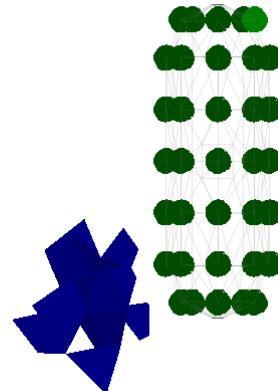
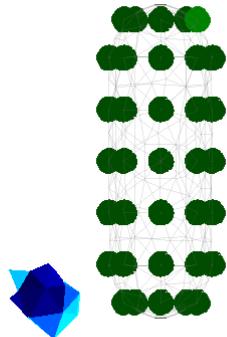
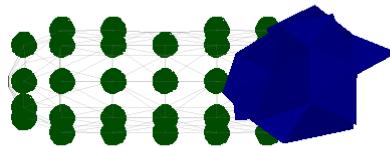
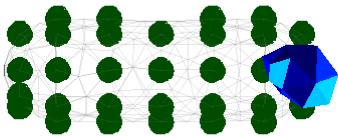
```
while true do  
    -- get cycle start time  
    start_time = tmr.read(stimer)  
    -- Check for pending RPC request  
    if rpc.peek( handle ) then  
        rpc.dispatch( handle )  
    end  
    -- run control function  
    control()  
    -- wait until cycle time elapsed  
    while tmr.getdiffnow( stimer, start_time ) < cycletime  
do end  
end
```



Imaging with Electricity



Simulated
Insulating
Object

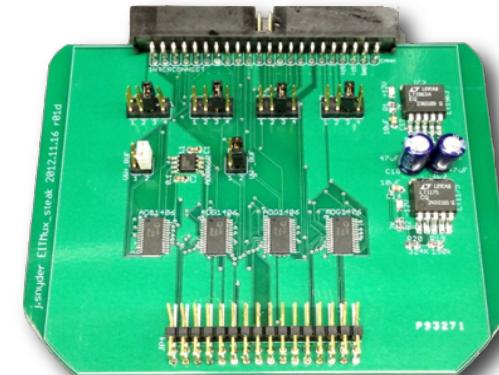


Reconstructed
Location/
Impedance



EITMux Head

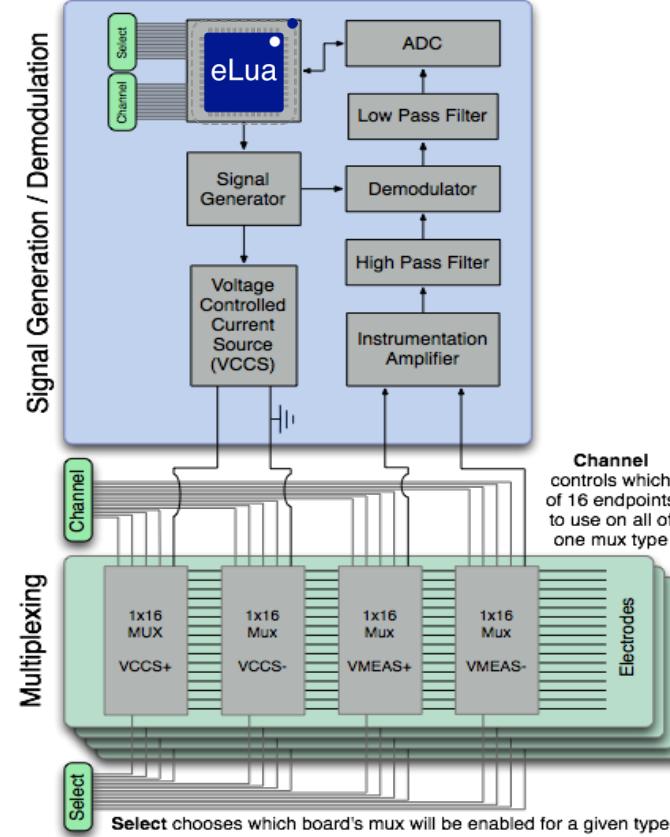
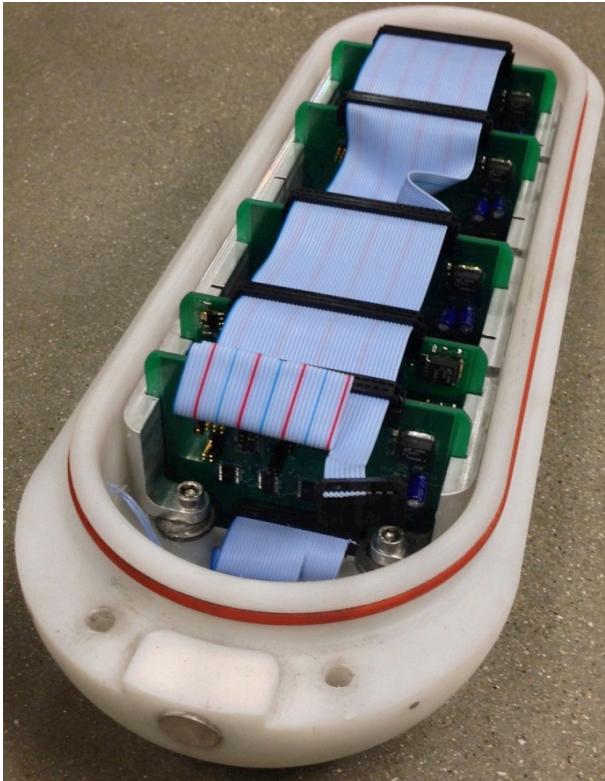
- STM32F4 running eLua with CAN interface
- Sinusoid Function Generator
- Voltage Controlled Current Source (VCCS)
- Lock-in amplifier / ADC



EITMux Steak

- 4x 1x16 multiplexers

x4



Lessons Learned / Implementation

- No custom C modules, drivers for ADC & waveform generator implemented in Lua
- All Lua code non-blocking
- Deterministic timing by using ADC clock to drive system (400 Hz)
- Updated GPIO multiplexing & forwarded ADC samples over CAN

Code Structure

autorun.lua (stripped down)

```
function mux_pattern()
    while true do
        -- generate mux pattern
        coroutine.yield(brd_select, chan_select)
    end
end

co_mux_pattern = coroutine.create( mux_pattern )

spi.setup( 0, spi.MASTER, 1e6, 1, 1, 16 )
can.setup(0,1000000)
adc_setup()
function_generator_setup()

function gpio_nededge_handler( resnum )
    cpu.cli( cpu.INT_GPIO_NEGEDGE, SPI_MISO )

    status, brd_select, chan_select =
    coroutine.resume( co_mux_pattern )

    if( chan_select == nil ) then
        pio.port.setval( chan_select, pio.PE )
        pio.port.setval( brd_select, pio.PD )
    end

    adc_value = spi.readwrite( 0, 0x58FF, 0xFFFF )
    cpu.sei( cpu.INT_GPIO_NEGEDGE, SPI_MISO )
end

-- continues on next slide...
```



Code Structure

```
while true do
    can, canidtype, message = can.recv()
    if canid ~= nil then
        if canid == CAN_ID_SET_FREQUENCY then
            -- update frequency
        elseif canid == CAN_ID_REQUEST_PARAM then
            -- send back value for parameter
        elseif canid == CAN_ID_SET_PARAM then
            -- update parameter
        elseif canid == CAN_ID_REQUEST_UPDATE then
            -- return current data
        else
            if DEBUG then io.write( string.format("unknown msg id: %d\n", canid) ) end
        end
    end
end
```



Scriptable Satellite Tracking

GSatMicro



- Compact Iridium tracker
- GPS, accelerometer,
magnetometer
- RS-232, USB & Bluetooth Low
Energy interfaces
- <http://www.gsatmicro.com>

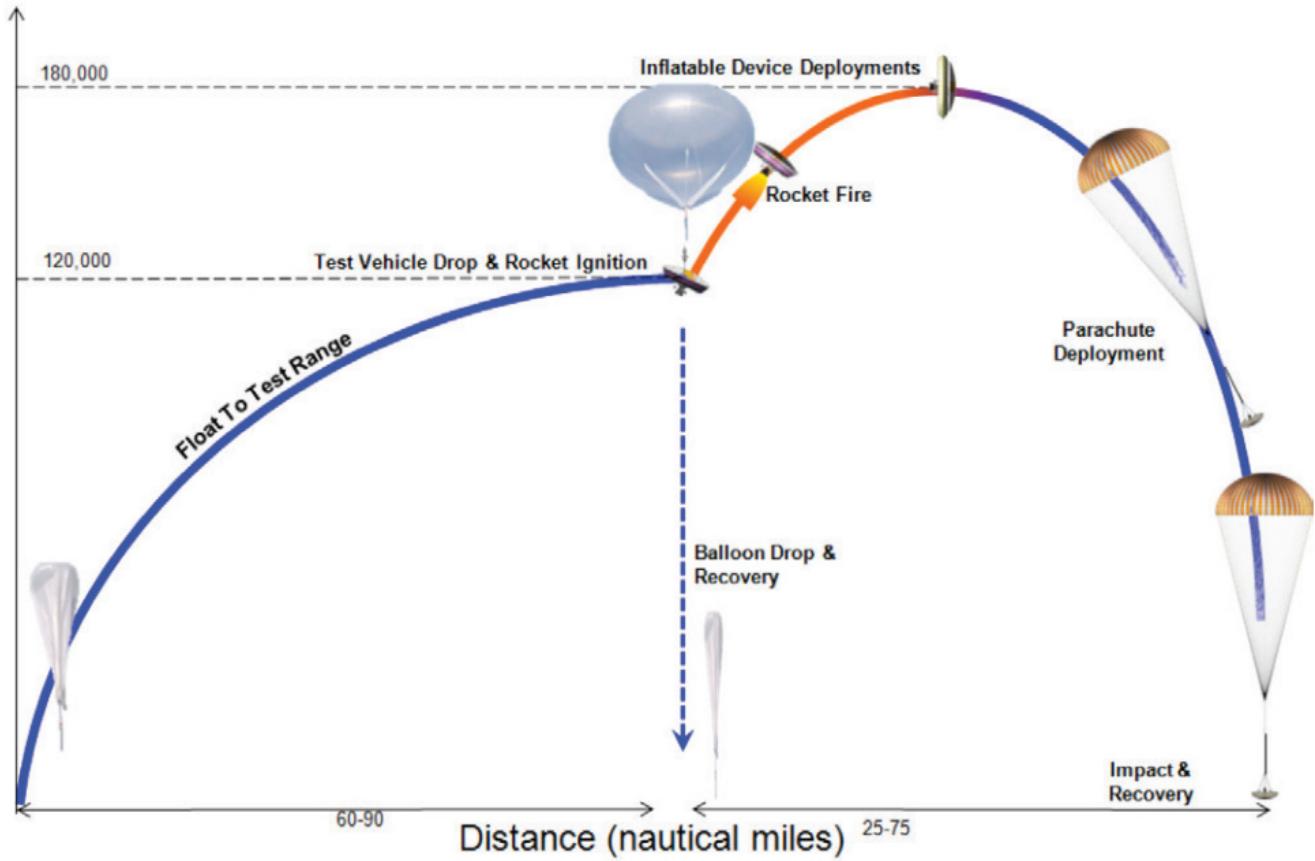




Image credit: NASA/JPL-Caltech

Lessons Learned / Implementation

- Modified project to allow mixing open/closed source code
- 32kB of SRAM adequate for ~5-6kB scripts with minimal table usage (~10kB used by Lua dynamic allocations)
- Extended eLua interrupts to provide event hooks
- Turnkey script with remote commands/settings or customized scripts
- Added watchdog to ensure VM is always running while awake

GSatMicro Code Structure

autorun.lua (simplified)

```
function satellite_signal ()
    -- handle satellite signal
end

function satellite_timeout ()
    -- handle no signal
end

function satellite_tx_ok ()
    -- handle successful transmit
end

function satellite_tx_fail ()
    -- handle failed transmit
end
```

```
function gps_valid ()
    -- handle valid gps
end

function gps_timeout ()
    -- handle no gps signal
end

cpu.set_int_handler( cpu.INT_SATELLITE_SIGNAL,
satellite_signal)
cpu.set_int_handler( cpu.INT_SATELLITE_TX_OK, satellite_tx_ok)
cpu.set_int_handler( cpu.INT_SATELLITE_TX_FAIL,
satellite_tx_fail)
cpu.set_int_handler( cpu.INT_SATELLITE_TIMEOUT,
iridium_timeout)
cpu.set_int_handler( cpu.INT_GPS_VALID, gps_valid)
cpu.set_int_handler( cpu.INT_GPS_TIMEOUT, gps_timeout)
```

Project Future

- Looking at Lua 5.3
 - Emergency garbage collection built-in (added in 5.2)
 - Bit manipulation built-in (added in 5.2), extended in 5.3
 - Supports 64-bit/32-bit int and float
- API updates to make event-driven programming easier

Questions?

- Project, Documentation, Resources:
 - <http://www.eluaproject.net/>
- Talk Materials
 - <http://github.com/jsnyder/elua-escbos>
- Contact

Bogdan Marinescu:

bogdanm@eluaproject.net

Dado Sutter:

dadosutter@eluaproject.net

James Snyder:

jsnyder@eluaproject.net

eLua Forks

- NodeMCU Firmware
 - Reworking and extension of APIs around event-driven model
 - Port to ESP8266
 - <http://github.com/nodemcu/nodemcu-firmware>
- Alcor6L
 - Adds Lisp & C support
 - <http://github.com/simplemachines-italy/Alcor6L>



eLua APIs

```
-- GPIO
WAV0_CS = pio.PC_0
-- pio.pin.setdir( direction, pin )
pio.pin.setdir( pio.OUTPUT, WAV0_CS )
-- pio.pin.sethigh( pin )
pio.pin.sethigh( WAV0_CS )
pio.pin.setlow( WAV0_CS )
val = pio.pin.read( WAV0_CS )

-- SPI
-- spi.setup( id, type, clock, cpol, cpha, databits )
spi.setup( 0, spi.MASTER, 1e6, 1, 0, 16 )
-- spi.write( id, data1, data2, data3 )
spi.write( 0, 0x23 )
status = spi.readwrite( 0, 0xFF )
```



eLua APIs

```
-- PWM
-- pwm.setup( id, frequency, duty )
pwm.setup( 0, 5000000, 50 )
-- pwm.start( id )
pwm.start( 0 )

-- ADC
-- adc.setclock( id, clock, timer_id )
-- adc.setclock( id, 0 ) -- no clock
adc.setclock( 0, 0 )
-- adc.setclock( id, length ) -- moving average filter
adc.setsmoothing( 0, 16 )
-- adc.sample( id, count )
adc.sample( 0, 1 )
-- adc.getsample( id, count )
sample = adc.getsample( 0 )
```

Documentation:

http://www.eluaproject.net/doc/master/en_refman_gen.html