Enhancing Embedded Development with Ruby

RubyConf 2007

Bill Bereza

Atomic Object

atomicobject.com

What?

Embedded C

8-bit PIC micro

How?

rake

argent

C unit tests

C mocks generated by Ruby

system testing in Ruby

continuous integration

Why?

In the beginning...

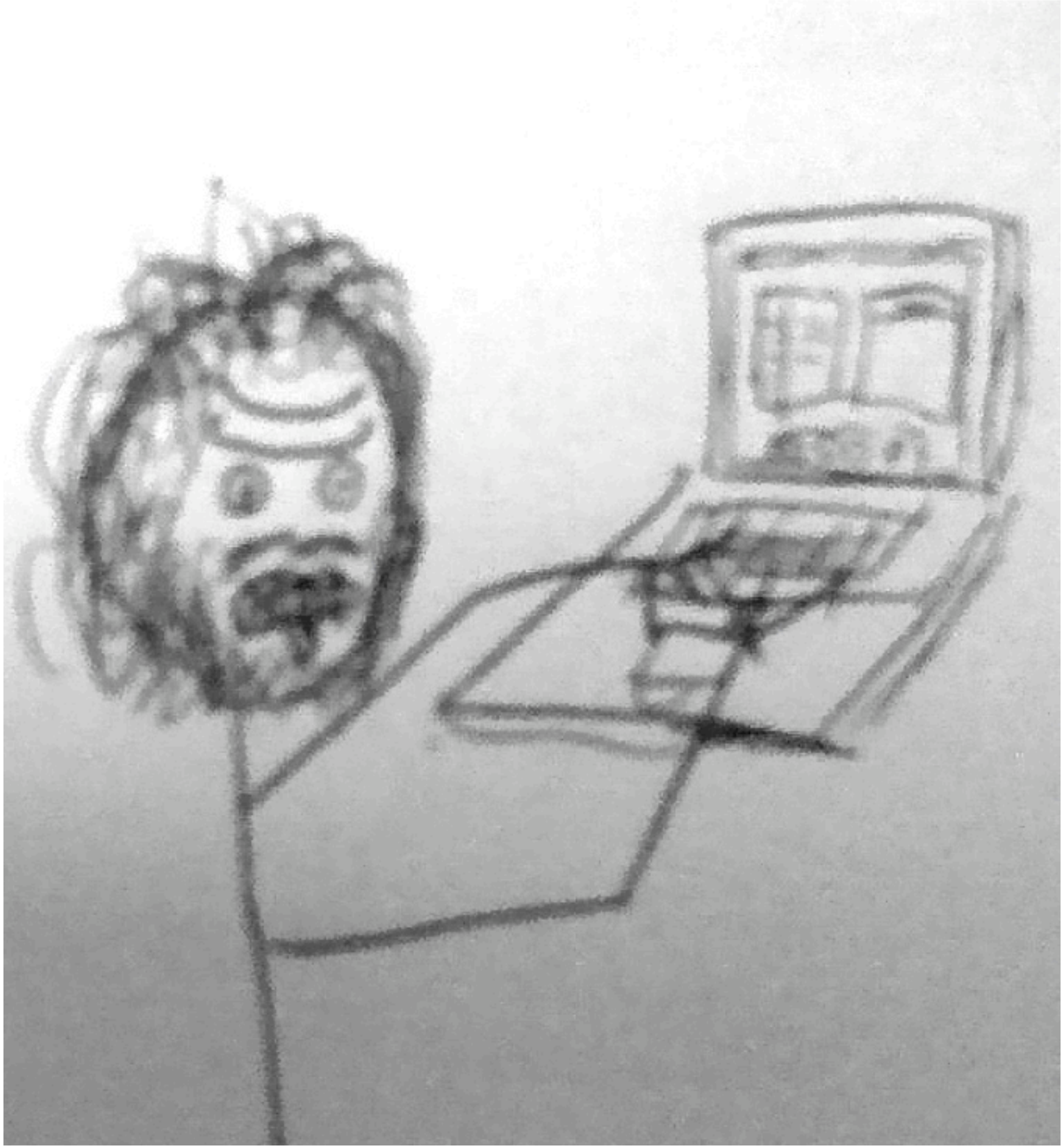
programming required intense knowledge of the machine

computers

not powerful

and tools were primitive

and there was much pain



along came

compilers

languages

libraries

which helped some

Today

automated tests

code generation

domain specific languages.

but

there's a problem.

embedded devices

are small

cheap

and they're everywhere.

The problem is

they are everywhere

they all have software

software contains bugs.

How it's usually done

write

flash

fiddle

debug

scope

try again.

Unrepeatable

This leads to

hope

anger

fear.

The Goal

Apply good practices

automation

unit tests

system tests

user stories

behavior driven

mocking

continuous integration.

Vendor tools suck.

IDEs oversimplify

and

also fail to do

what is needed,

like test

and automate.

Debugging tools

are plentiful,

testing support is rare.

Solution.

Ruby

& tools written in

ruby

provide glue

for

all good practices.

The Project

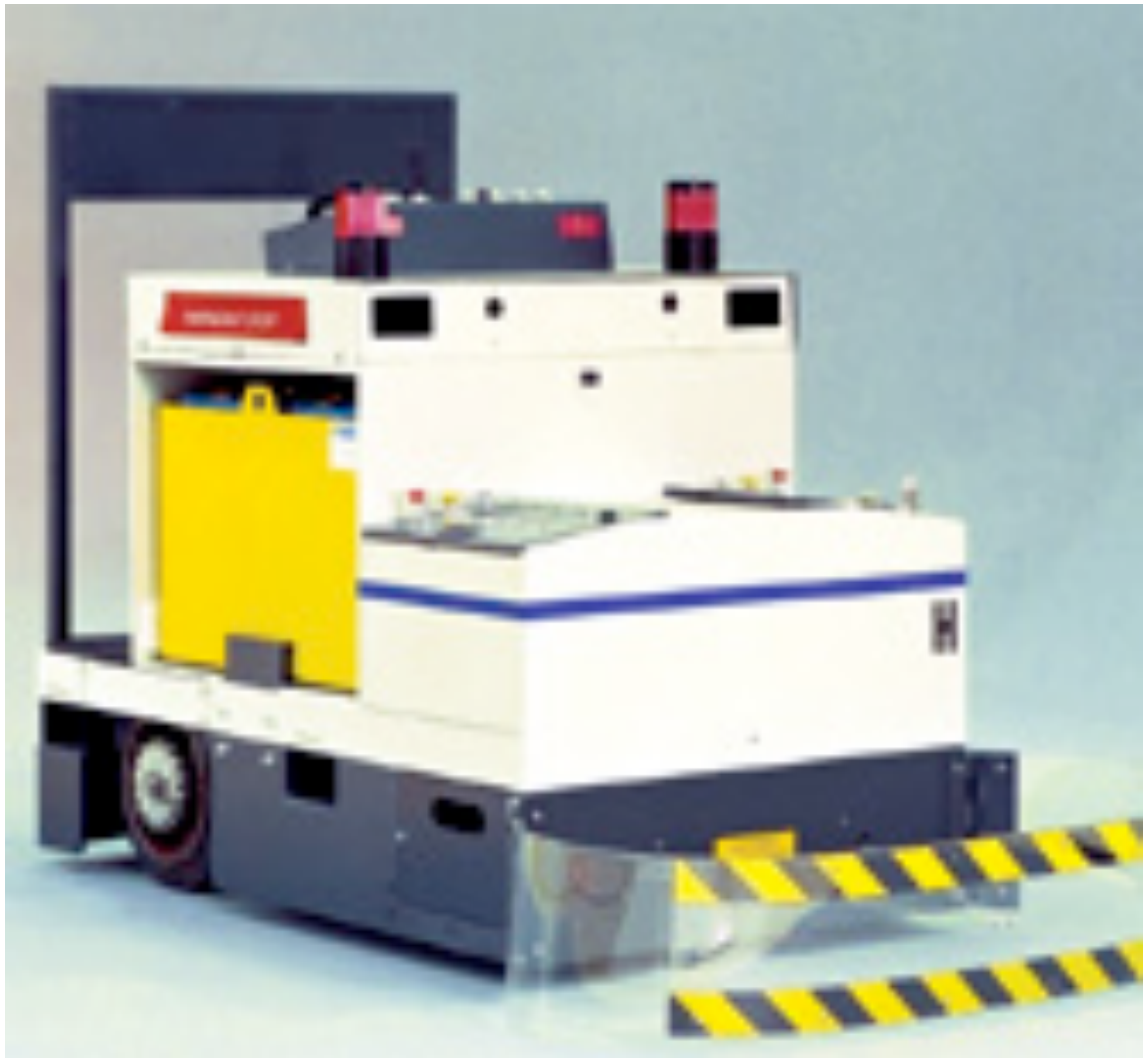
Automated guided vehicles

which are

self propelled

semi-autonomous

factory vehicles.


H

NO RIDING
5

4 basic types

forklifts

tow vehicles

unit load carriers (lift material on back) carts (items are placed on the back)

Shared hardware architecture

configurable boards

for

steering

speed

battery

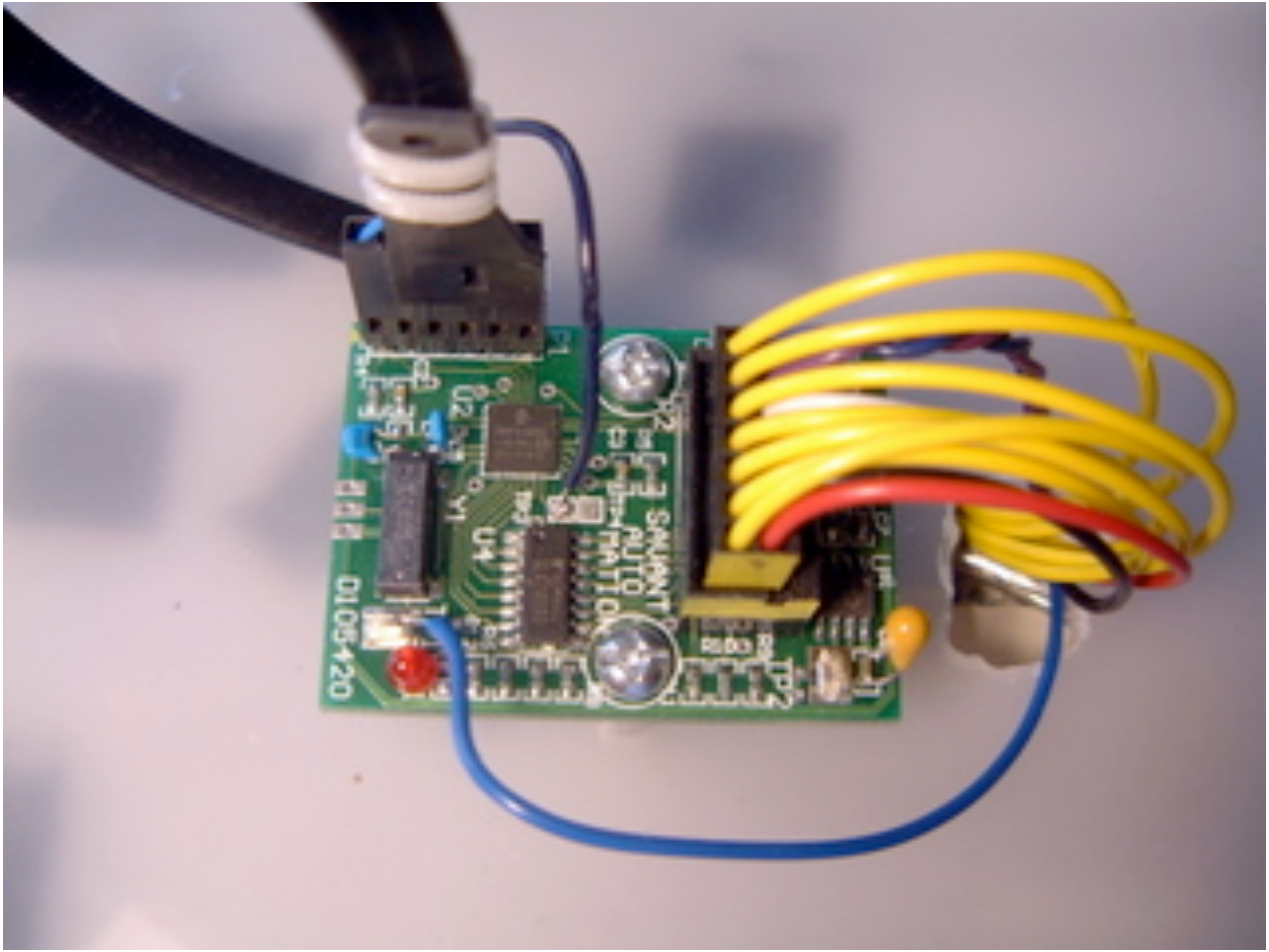
navigation.

Our first project

speed control

Our second project

battery monitor



Embedded C

Microchip PIC18F2480

16 kilobytes

FLASH

768 bytes

RAM

How Ruby helped

rake

```
rake test:system
```

Written in

Ruby.

Uses systir

system testing framework

An example:

```
proves "the instantaneous battery level reported via CAN is
correct."

set_battery_level_display_instantaneous
set_can_status_mode_to CAN_STATUS_NORMAL

set_battery_data 700, 24, FLOODED
set_battery_current_for_vehicle_consuming_charge_with_amps 40
set_battery_voltage 24

wait_for_battery_level_output 4

set_battery_voltage 25.80
verify_battery_level_output_is 9

set_battery_voltage 22.8
verify_battery_level_output_is 0
```

System tests

are function calls

in bm_driver.rb

```
def set_battery_voltage(voltage)
  @minilab.write_analog(
               VOLTAGE_OUTPUT_PIN,
               voltage /
               BATTERY_VOLTAGE_DIVIDER)
end
```

Ruby C extensions

minilab

Hardware

digital & analog I/O

Ruby class

provides

read_analog(channel)

write_analog(channel, volts)

read_digital(pin)

write_digital(pin)

The box

is wired

to the board.

pcan

Hardware

CAN bus

to USB

The vehicles

use CAN

internally

between each board.

We use CAN

in system tests

to

simulate the system.

Ruby class

provides

message methods.

receive_message (timeout)

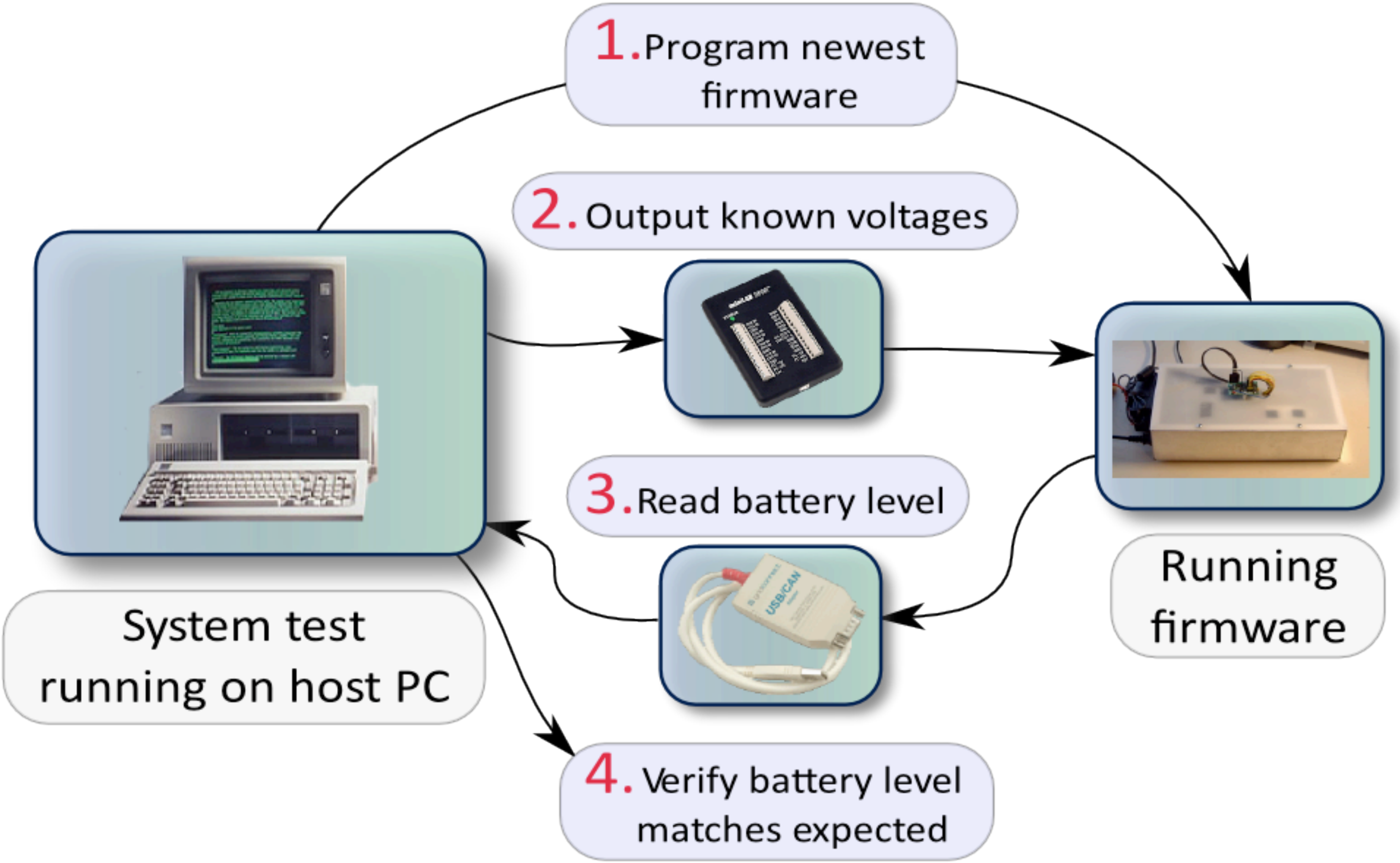
transmit_message
message

The Build Server

connected via

pcan and minilab

to the board.



1. Program newest firmware

2. Output known voltages

3. Read battery level

4. Verify battery level matches expected

System test running on host PC

Running firmware

```
rake test:units
```

written in C

unity

C unit test library

small enough

argent

per unit test file

```
//[[$argent require
'generate_unity.rb';
generate_unity();$]]
//[[$end$]]
```

parses *test*.c

generates and inserts

C code

main()

calls to unit tests

```
//[[$argent require
'generate_unity.rb';
generate_supermock
("Model,Utilities");$]]
//[[$end$]]
```

initializes mocks

for _all_ C files

except the ones passed in

Model,Utilities

CMock

quick Ruby script

```
FUNC_MAGIC =
/(\w*\s+)*(\w+)\s+(\w+)
\s*\(([^\)]*)\)/
```

Parses .h

for each function

generates

*function*_ExpectAndReturn(args)

and writes

Mock_*file*.c.

A unit test:

```
static void testCANConductorHandlesNewMessage_
WhenNewMessageAvailable(void)
{
  CANMessage message = {0};

  HardwareEvents_GetDoCANOutput_Return(FALSE);
  CAN_IsNewMessageAvailable_Return(TRUE);
  CAN_GetNewMessage_Return(message);
  Model_ProcessCANMessage_Expect(message);

  CANConductor_Run();
}
```

For unit tests

we link

the file under test

to the mocked files.

Built binary

run in a simulator.

One binary per

test file.

Continuous integration

written in Ruby

monitors subversion

gets each checkin

runs rake.

Unit tests run

in a simulator.

System tests run

on the build server.

Actual production build

runs on the board.

Benefits

C code is clean

DRY

and readable.

```
void CANConductor_Run(void)
{
  if(HardwareEvents_GetDoCANOutput())
  {
    CAN_SendMessage
(Model_GetCANStatusMessage());
    HardwareEvents_ClearDoCANOutput();
  }

  if(CAN_IsNewMessageAvailable() ==
TRUE)
  {
    Model_ProcessCANMessage
(CAN_GetNewMessage());
  }
}
```

Binary being tested

is actually

the binary deployed.

No magic test build.

Readable system tests.

Quick to write.

Unit tests

test behavior.

Unit and System

tests are easy

and fun to write.

Customer is happy.

Developers are happy.

Questions?

References:
atomicobject.com - papers, software