ESP Gen 2
Detailed Software PDR

Revision 1.1

Derived from Outline in
/ESP/gen2/software/plan/softdesign.rtf

Brent Roman

December 2, 2004

Contents & Goals

- Introduce Processing Hardware
- Firmware & Software Components
- Explain their interfaces:
  - To each other &
  - To peripheral microcontrollers
    - Via the PC/104 bus
    - Via serial busses: USB, I2C, Ethernet & RS-232

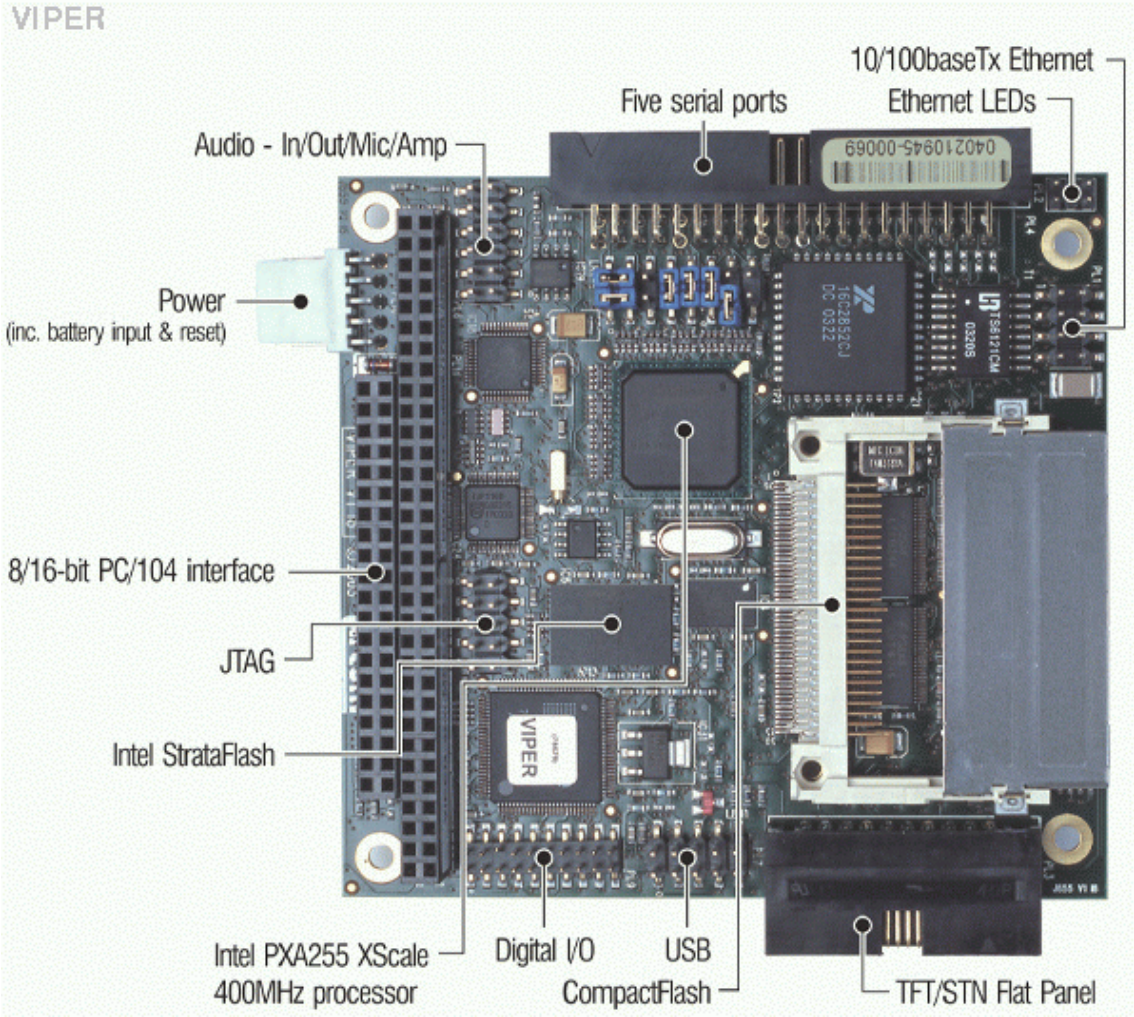
Embedded Host CPU Requirements

- >= 16MB RAM, 16MB 'flash' memory
- Battery Backup-up Real-Time clock
- Removable Compact Flash disk storage
- 10 Mbit Ethernet
- USB v1 host (for CCD Camera)
- I^2C interface for “dwarf” microcontrollers
- “3-wire” RS-232 port for console & debug
- PC/104 bus interface to custom motherboard

Arcom's “Viper” PC/104 CPU

- Intel Xscale @400 Mhz, 2W @+5Vdc
  - Successor to StrongARM in our SideARM
- 64MB DRAM + 32MB flash
- Real-Time Clock with 5yr battery
- 100/10 Mbit Ethernet
- 5 serial ports
- Linux Support (Kernel version 2.4)
- Limited PC/104 bus support
- USB (v1) host interface

Arcom's “Viper” PC/104 CPU



Custom PC/104 Motherboard

- 7 RS-232 ports with limited modem control
  - For contextual sensors, analytical modules, sampler
- 1 RS-232 port with full modem control
  - For RF modem or hardwire interface to mooring
- Sleep controller restores power after:
  - A preset number of minutes
  - "Wakeup" packet from RF modem port
  - External magnetic contact closure

Embedded Host CPU System Software

- OS Linux version >= 2.4.9
- No graphical interface
- Bash or equivalent shell
- Minimal set of Unix utilities
  - Administration commands
  - May use BusyBox for fast rebooting
- Ruby Object Oriented scripting language
  - Version 1.6.8 -- because it's stable!

Embedded Host Networking

- TCP/IP with DHCP support for Ethernet
- PPP on the RF Modem/mooring serial port
- Telnet
- FTP
- HTTP (basic web server for status displays)
- NFS client (for sharing files in the lab)

Embedded Networking Configuration

- Automatic if network supports DHCP
- If DHCP fails, fall back on preset defaults
- User may log in via dedicated RS-232 port
  - To set network presets with an ASCII editor
  - To diagnose network problems
    - Using standard Unix/Linux admin commands

PC Development Host

- Any Linux platform
  - Had planned to support Windows, but why?!
- Same Ruby v1.6.8 Scripting Language
- X-windows graphics for:
  - Ruby's object class browser
  - text editors, CVS gui, etc.
  - Later for ESP's own GUI
- PC replaces embedded Host CPU for lab use

TI MSP430 Servo Controllers aka “Dwarves”

- 60KB of Flash, 2KB of RAM @~4Mhz
  - DCO servoed to a watch crystal in firmware!
- Programmed in ANSI 'C' + some assembly
- Bare metal, No real-time kernel
- All run exactly the same code:
  - Using ~40KB of code, 1.5KB of RAM
  - Multi-master I^2C @ 100kbps
    - I^2C bus address set by a tiny on board dip switch
  - Debug RS-232 port @ 38.4kbps

Why use I^2C?

- Very low power
- Adequate data rate (as high as 400kbs)
- Supported by new MSP430 variants
  - Many sensor chips also connect directly
- Supports Peer-to-Peer networking
  - Defines flow control, collision detection
- Supports up to 126 nodes (~40 electrically)
- Unlimited packet length
  - Not as amenable to hard real time as CAN

MSP430 Sleep Mode Controller aka “Sleepy”

- Marks time while the system is shut down
- Can access motherboard's 8 serial ports
- Wakes host and other micros when:
  - Preset delay expires, or
  - External event detected by contextual sensor, or
  - Wake-up packet received on the radio modem
- Serves as the host's gateway to the I^2C bus
- Runs at ~8Mhz to cope with high data rates

I²C Network Gateway

- Sleepy's main function
- Complete I²C bus access via an RS-232 port
  - Works for any I²C bus peripheral, not just ours
  - Flow-thru design reduces latency to 2 byte times
- Supports our I²C protocol extensions
  - But does not require them
- Stresses the poor, slow MSP430 to its limits
  - Peak interrupt loads of one every 30µs
  - TI's I²C controller design is flawed/buggy

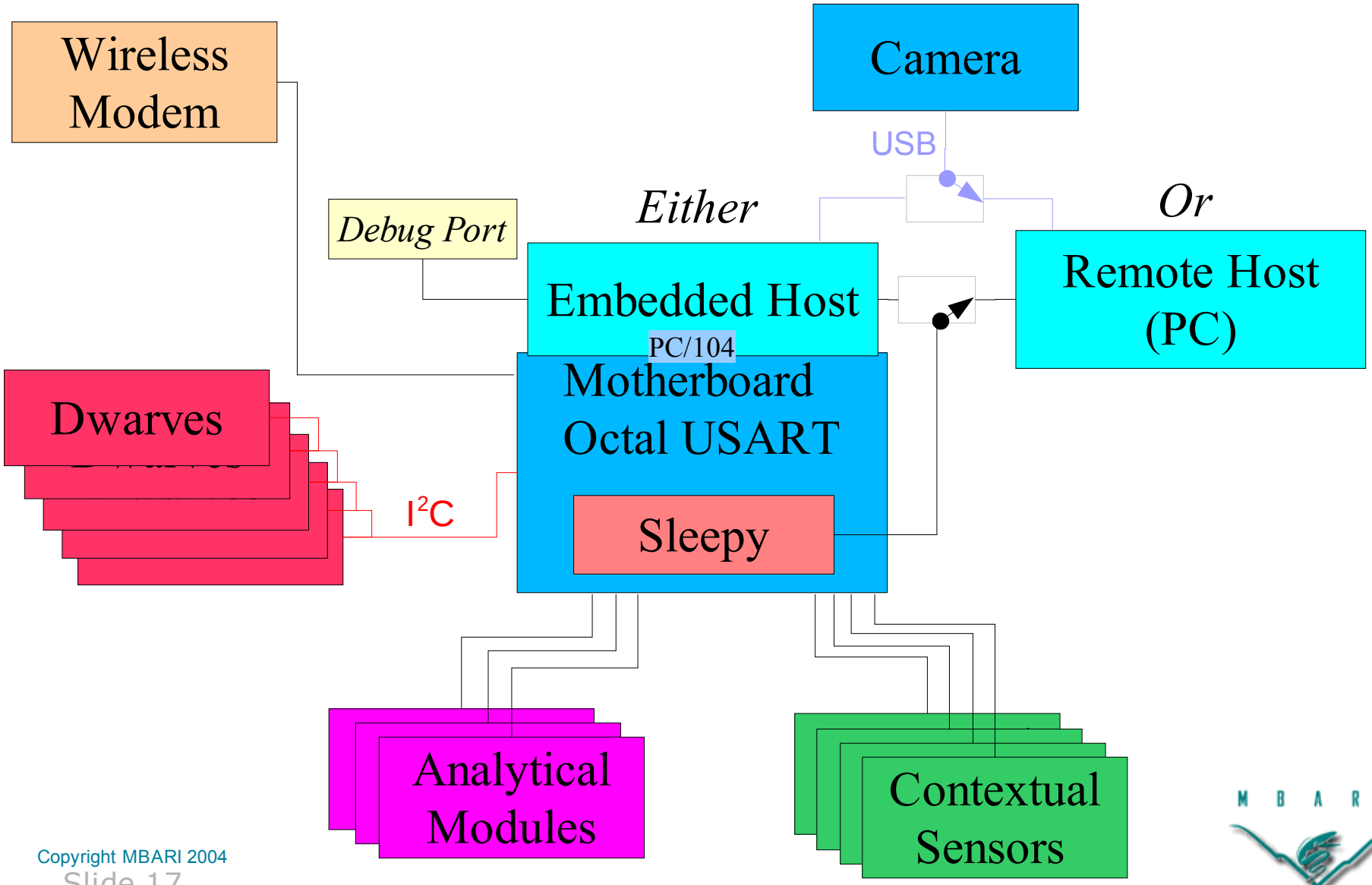
Our I^2C Protocol Extensions

- Standard I^2C adds 1 bit ACK for every byte
  - But no CRC check or message received ACK
  - So, lacks an error detection/recovery mechanism
- We add a single "Message ACK" bit
  - And a one byte CRC to each message
  - Also used for message level flow control
- This will not confuse "third-parties" on bus
  - Standard msgs may still be sent to them
- Already proven its worth in the lab

The Gateway Concept

- All high-level control multplexed onto a single 115kbps serial port
- Embedded host connects via this port
  - Uses PC/104 only for:
    - faster access to Octal USART
    - Access to optional PC/104 cards
- Development Host connects via same port
- Remote control possible via terminal server
  - Or even a freewave modem!

System Comms Overview



"Dwarf" Microcontroller Firmware Concepts

- Dwarves Close all real-time servo loops
  - So the host can contemplate its cybernetic navel
- Identical microcontrollers configured by host
  - They may be moved from one subsystem to another by changing cables and a network address dip switch
  - Less firmware development, fewer spares, easier config cntrl
- No Message Polling!
  - Dwarves master the I^2C bus when events warrant it
- All supported operations can be done in parallel
  - Heating while moving servo motors, etc.

Dwarf Microcontroller Messages

- Three message types: *[from doc/DwarfFunctions.rtf]*
  - Commands have no Reply
    - eg. Download configuration, stop servo, etc.
    - No error/status return possible
  - Requests have at least one Reply
    - eg. Move to specified location, Get current status...
    - A unique tag number is associated with each
      - to support multi-threading in the originating host
    - One request may be answered by many replies
      - eg. Get status every second until canceled
  - Replies simply answer a previous Request
    - Always include the relevant Request's tag #

Host Application Software

- Written entirely in Ruby
  - Aside from 1 (tiny) custom 'C' extension
- Supports three distinct operational modes:
  - With Actual hardware (normal operation)
  - Simulated hardware in simulated time
  - Simulated hardware in real time
- Runs protocol scripts as OS commands
- Or, takes them from its own interactive shell
  - Modified Interactive Ruby (aka 'irb')

Core System Modules

- Schedule Module
  - Queues and Dispatches events
- Message Module
  - Routes messages to network targets
  - Dwarves or Models thereof for simulation
- Logging Module
  - Records dumps, & queries log files
  - Logs time, user commands & comments
    - messages, exceptions, & comments

Driver/Model Modules

- Interface to real (or simulated) world
- Convert raw device messages into objects
  - And visa-versa
  - Do not convert user friendly units
- Also model the device to enable simulation
  - messages to/from model instead of the device
  - Even crude models are useful
- Drivers & corresponding Model in one place
  - To ease maintenance (they must rev in sync)

Driver/Model Module Examples

- I2C::Shaft -- Absolute Shaft positioning
  - Rotary Valves
- I2C::Solenoid -- Solenoid actuation
  - Burkert Valves
- I2C::Thermal -- Thermal control
  - Process and Collection puck heater/cooler(later)
- I2C::Servo -- Linear servo positioning
  - Syringes, Clamps, Grippers, Shuttle, Carousel

Library Modules

- Each builds atop one or more drivers
  - Provide simplified access
  - Convert to user friendly units
- Entirely unaware of operational mode
- Provide the basic set of end-user commands
- Enforce safety interlocks
  - Engineer type users can bang directly on the drivers if they are bent on breaking things

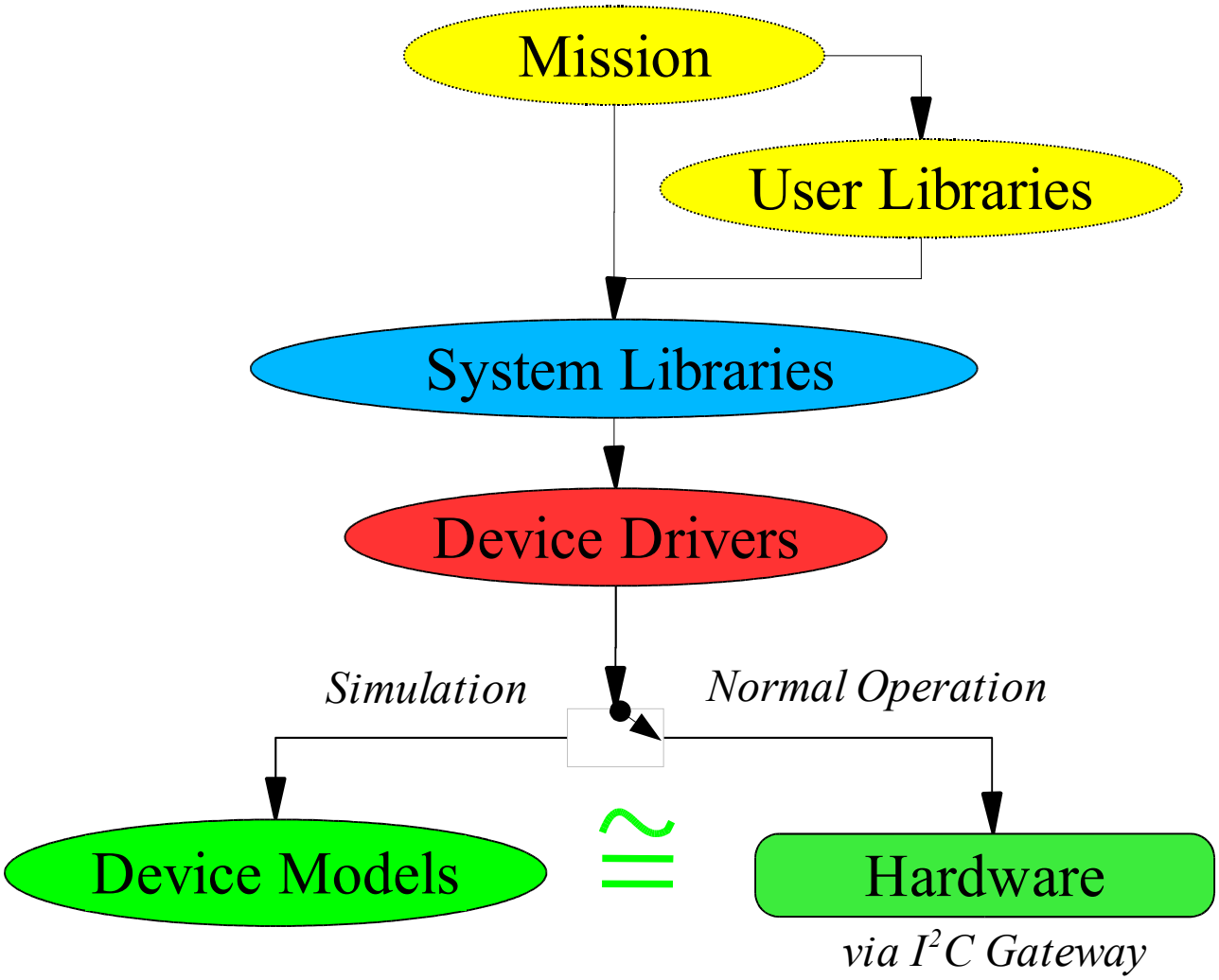
Example Library Modules

- Gripper
  - gripper.release; gripper.grip
- Delay
  - Delay (seconds); delayUntil (Time)
- Syringe
  - .empty; .fill; .to(volume); .volume, ...
- Shaft – control of rotary valves
  - .dial :positionName; .dialBetween :pos1, :pos2
- Thermal
  - .seek (temperature), .hold(temp), .temperature
- Valve – burkert valves and manifolds
  - .open, .close, .to (positionName), .state

User Library Modules

- Mission phase scripts
  - Will likely be unique to each deployment
  - Need not be a named module, just a script
- Mission/User Libraries
  - Named libraries of common user code
  - Maintained by researchers and techs
  - Common sequences in missions migrate into User Libraries

Application Module Hierarchy



Mission Phases & Sequencing

- A mission is a sequence of “phases”
  - Each phase is typically a single Ruby script file
    - But it may be any OS command
  - Mission schedule determines when phases run
  - The ESP is powered off between phases
  - Phases may also be invoked from OS prompt
    - For testing in the lab

Mission Schedule

- Mission schedule defines when phases run
  - May be edited as the mission progresses
  - Is simply an association of Times with phases
    - Example ruby script defining a mission schedule:

```
Mission = schedule "UTC
   1/13/04  7:52",  "esp sample 2",     #sample puck in tube 2
  "         9:14",  "esp archive 4",    #archive puck in tube 4
  "1/15    22:10",  "esp process",      #1/15/04
  "         3:19",  "date",             #1/16/04-any OS command
  "2/03    12:19pm","esp sandhyb 6",    #2/03/04
  "from 4:19pm until 2/5 every +6:23", #checkContext on the tide
                   "esp checkContext" #...until 4:19pm 2/5

4.times {Mission.add "Monday 15:14:52pst", "esp phoneHome"}
```

Mission Sequencing Notes

- Few objects are preserved across phases
  - They must be written to disk before shutdown
  - User may easily register objects to save/restore
- The schedule editor
  - Displays phases already executed & remaining
  - Allows remaining phase times to be changed
- Reordering phases by changing start times?
  - Beware, pucks can't be reordered in carousel!

Mission Suspension

- Unhanded exceptions in main thread suspend the mission
- ESP remains powered on for a preset number of minutes
- Calls out for for “Help” -- someone to manually intervene
  - User logs in, inspects state and may clear the error
  - ESP stays up as log as some user is logged in
  - User may resume the mission
  - Otherwise, ESP shuts down N minutes after logout

Mission Abort

- ESP powers down if no one intervenes while suspended
- Current mission phase is aborted
- ESP does a controlled shutdown to enter a safe state
- Remains powered down until an external wakeup received
- Remaining mission schedule and state are preserved
- Users will try to restart operations from the next phase

Simulation Notes

- Simulate on embedded or development host
  - Simulating on the target insures all files present
  - Verifies reagents & pucks properly loaded, or
  - Tells users how to configure pucks & reagents
- Catches logic errors
  - Including mechanical interference between axes
- As fast as can be computed or in “real-time”
  - Next year's effort to include real-time animation

Simulation of Core Hardware

- Event messages redirected to simulated microcontrollers
  - Actual I2C bus traffic is simulated and logged
- Motor controller models mimic closed-loop behaviour
  - Axis models may coupled to detect mechanical interference in future
- Valve controller models are coupled to syringe axes
  - Must determine fluid source from manifold config.
  - Err when pushing large displacements against a closed valve
    - (excessive pressure)
- Environmental Sensors are simply preset

Simulation of Internal Imager

- USB CCD camera simulated very crudely
  - Actual USB bus traffic is not simulated
  - Simple Finite State model of camera states
  - Simulated delays model exposure times
  - Discrete time dynamic model of camera cooling
- No images are produced
  - Just log entries denoting camera use

Parallel Threads of Execution

- Ruby supports multi-threading
- Threads help in optimising multi-axis movements
- Extensions of Ruby's built-in multi-threading:
  - Synchronization “Locks” with synchronous signalling
    - Easier than Ruby's Mutexes & Condition Variables
  - Thread “Families”
    - Similar to Unix process trees
    - Simplify error handling
- Careful programming still needed to avoid deadlocks!
  - Sim. will catch almost all such programming errors

ESP Thread Families

- Built-in Ruby threads lack this concept
  - Each ruby thread is, by default, completely independent
- ESP threads are a more like Unix processes
- Unhanded exceptions in ESP threads:
  - Raise “ChildDied” exception in parent thread(s)
  - Raise “ParentDied” exception in children
    - Exceptions avoid trying to sync with dead thread!
  - Threads explicitly orphan children and disown parents
  - Otherwise, all threads in family die if any member does

What's done

- User-Friendly, Object Oriented Control of:
  - Valves (Rotary & Solenoid), Manifolds
  - Syringes (Collecter & Sampler)
  - Thermal (just Collection Puck heat for now)
  - Multiple threads
- Logging with Timestamps of:
  - Executed High-Level Commands
  - Low-level Dwarf Messages sent and received
  - Unhandled Exceptions (Errors)
  - User Comments

Simulation

- Operational modes:
  - Real-Time with Real Hardware in the loop
  - Real-Time with Simulated hardware
  - Simulated Time with Simulated hardware
- Dynamic Simulation of:
  - Valves, Syringes and Puck Heating & Cooling
- Framework for adding more...

MSP430 Firmware

- Robust I^2C networking
- Complete Servo Motor control
  - 2 Channels, Trapezoidal Trajectory
  - Dynamic drive voltage compensation
  - Overcurrent Sensing
- Temperature Servo control
  - Some ongoing work
- Control of Rotary & Solenoid Valves
  - done

Embedded Host

- Tested Arcom Viper PXA-250 PC/104 CPU
  - Runs Ruby in ~4MB of RAM
  - Lots of room to spare
  - Boots to application in <30secs
  - USB host works in general
- USB camera tested
  - Works under Windows and Linux
  - Need to test Camera with Arcom board

Development Tasks in 2005

- Implement firmware for “sleepy” MSP430
  - A servo dwarf is I^2C gateway in our lab system
- Contextual Sensors & Analytical Modules
- Enhance Simulation
  - Track Fluid & Power use
  - Detect mechanical interference
- Sequence Mission “Phases”
  - Boot, run next script, save state, shutdown
- Port Everything to the embedded target
  - Not as hard as it sounds...

GUI Development

- We didn't get to it this year
  - All aspects of firmware development ran long
  - Firmware (I^2C) development was worst
    - 'Experimental' MSP430 variants are properly labeled
    - Worked around and documented numerous I^2C bugs
      - TI says “Thank you very much for your feedback”!
    - Planned one month task became 4+ months
    - But, our firmware workarounds are very stable
      - No significant performance hit
      - Haven't changed this code in 3 months!

GUI Plans

- Cartoon depiction of ESP state
  - Dynamically updated
  - Simulated or real hardware
  - Also depict resource use: reagents, pucks, etc.
- Commands in text window change state
- Build using National Instruments LabView
  - Already well established among our target users
  - OS agnostic (Windows, Linux, MacOS, Unix)
- Add input icons as time allows
  - ESP script in LabView dataflow diagrams?