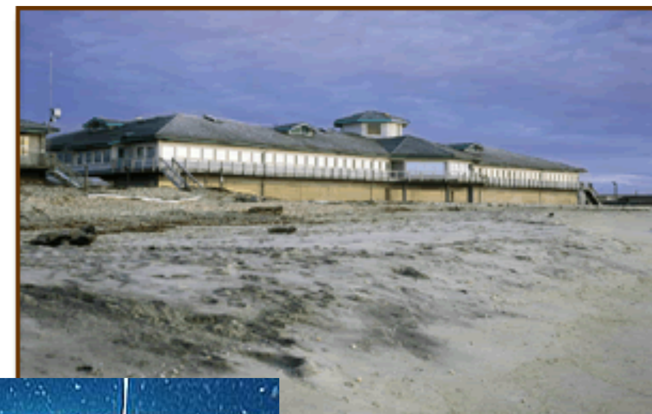
Making Linux do Hard (Real-)Time

2/20/13 Brent Roman brent@mbari.org

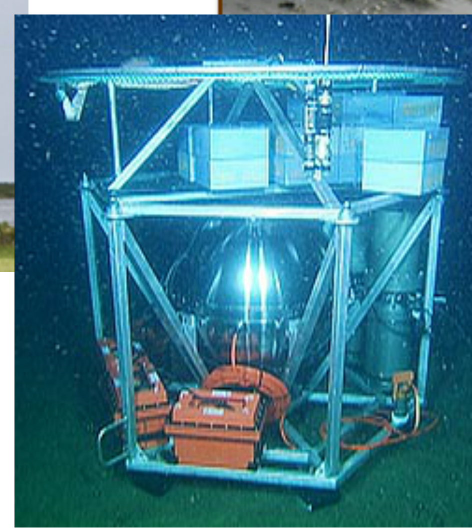
MBARI

The Monterey Bay Aquarium Research Institute is a
Non-Profit Research Center Founded in 1987 by Packard Foundation
Furthering marine research through the peer efforts of scientists and engineers
220 Employees (1/3 Science, 1/3 Engineering, 1/3 Administration)
approx. $40 M/yr annual operating budget
Located in Moss Landing, California
Operates 2 full-time research ships plus numerous ROVs and AUVs
Including the swath vessel “Western Flyer” for longer missions further afield



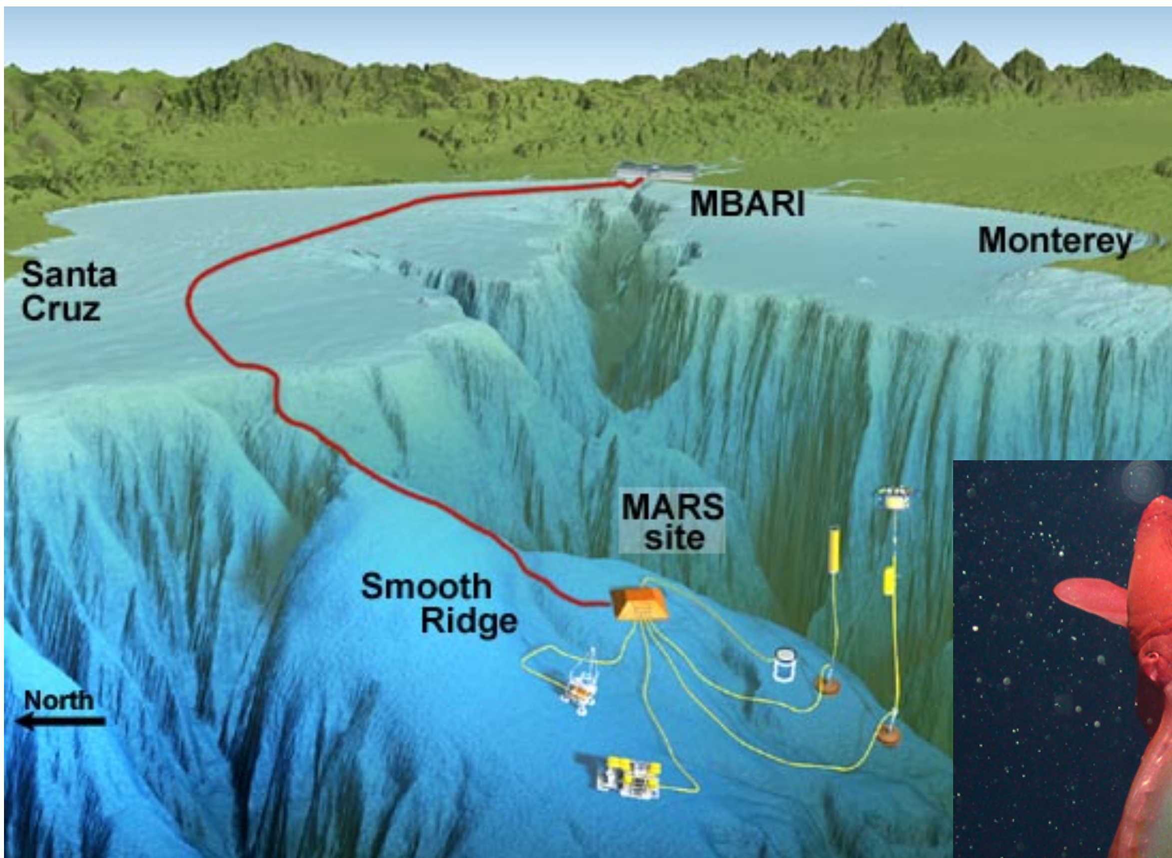


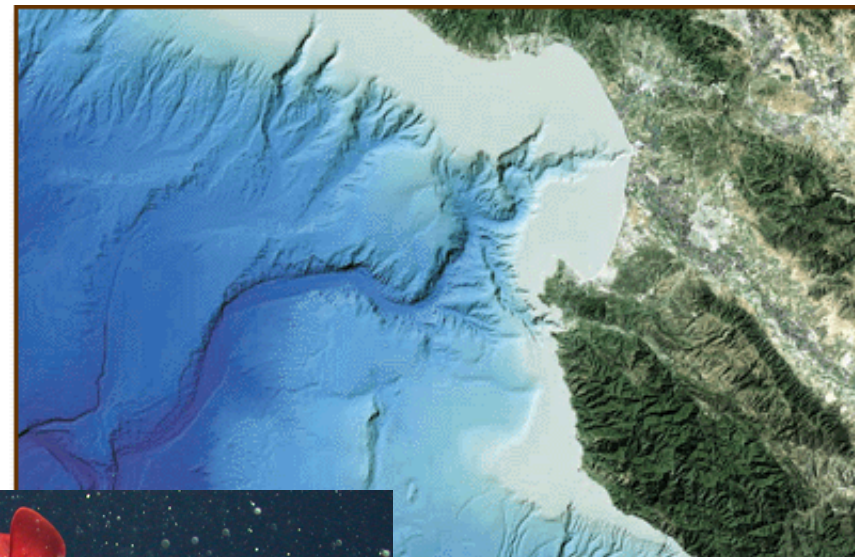




Monterey Bay Submarine Canyon

Extends 95 miles from Moss Landing, California
Maximum Depth is 3600 meters, reachable by day boats.
Canyon Sides are > 1600 meters -- deeper than the Grand Canyon
Much is classified as a National Marine Sanctuary
New species are discovered on a regular basis





Vampyroteuthis infernalis

Simulated Time

Simulated Time systems calculate time like any other quantity.

They incorporate model virtual worlds.

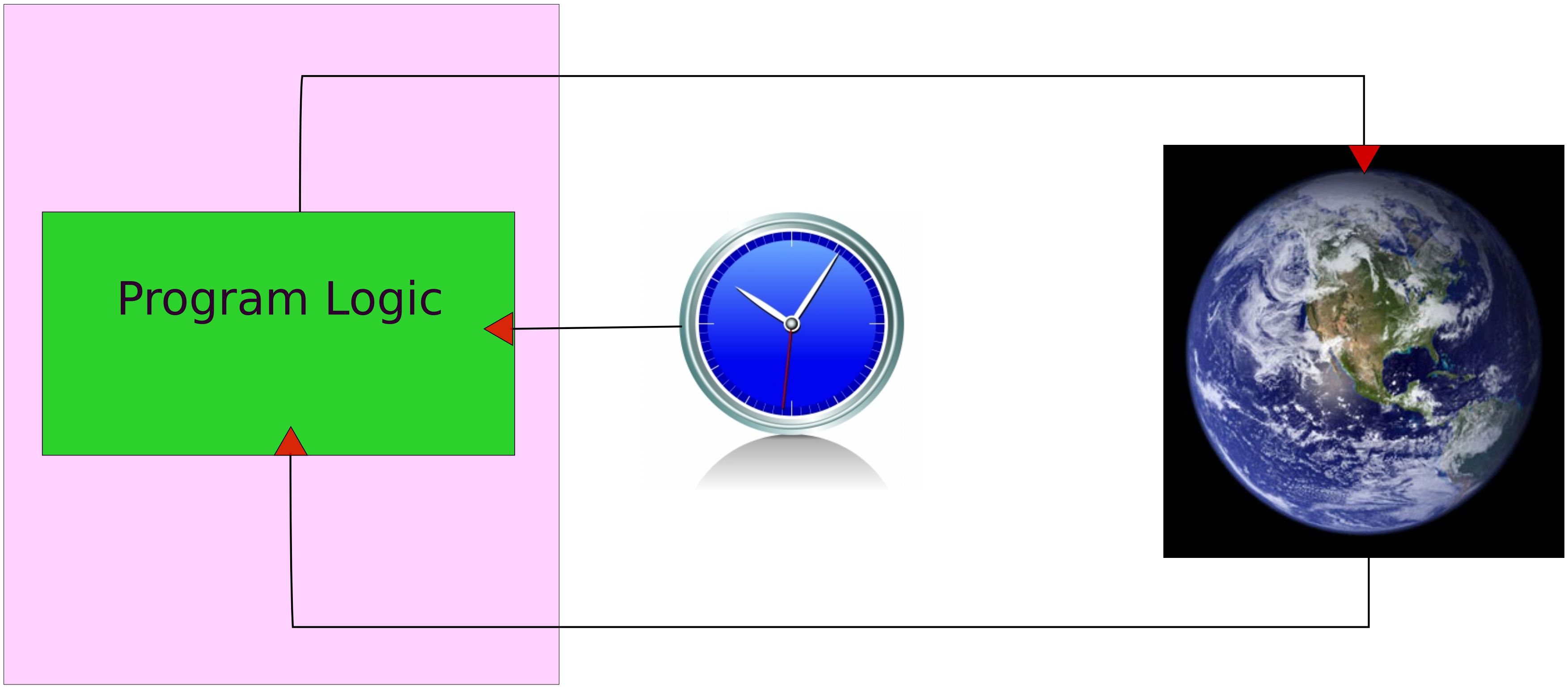
Useful for predicting the future or explaining the past...

but not much use for influencing the present!

Real-Time

Any system that interacts with the real, physical world

Time is an external input



Systems that interact with the real world must synchronize with it.

Deadlines

Real-Time deadlines can be

hard: where missed deadline means system failure
characteristic of interactions with physical world

firm: occasional missed deadlines are tolerable
characteristic of interactions with other computers

soft: preceived "quality" of system degrades as deadlines are missed
characteristic of interactions with humans

Hard, firm and soft are subjective generalizations
Most systems have multiple deadline types, each with unique qualities.

A system is denoted as ***hard***, ***firm*** or ***soft*** Real-Time depending on its most challenging deadlines.

"Hard-realtime" systems may have firm and/or soft deadlines as well

Epiphany

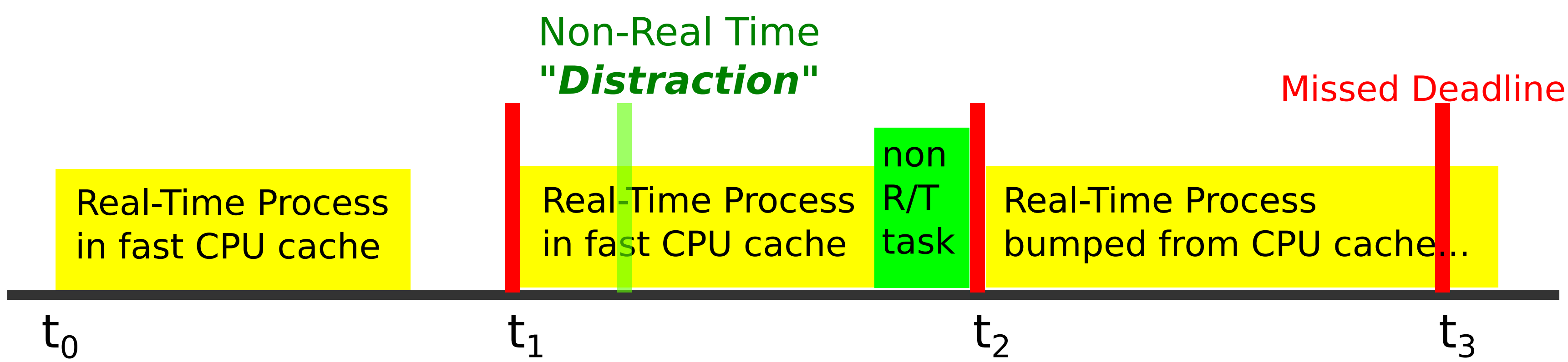
Computers are fast relative to most real-time constraints
Embeddeded Linux is everywhere!
  It is inexpensive, robust, easy to program,
  hosting a huge number of languages and libraries

Use Linux and dedicate sufficient computing resources to ensure hard real-time deadlines are always met.

Throughput vs. Determinism

Linux CPUs typically utilize large, multi-layer memory caches
Optimized for throughput rather than determinism
Caches make CPUs run like a hare
   but, in real-time systems, the tortoise wins!

CPU memory caching prevents Hard Real-Time processes from safely utilizing more than a small fraction of the available time.



One generally cannot lock real-time processes into CPU caches
Sometimes, one *can* reserve a core exclusively for R/T processes

Trouble in Kernel Space

Linux was designed to be open, flexible, fair, and fast.
It was never intended to meet hard timing deadlines.

Long running Linux kernel operations could not be interrupted.
Device Drivers would occasionally disable interrupts for many milliseconds.

These issues were scattered throughout the kernel sources!

Until recently...

Does PREEMPT_RT Spell Redeemtion?

Device drivers have been steadily improved

The PREEMPT_RT patch dramatically reduces the kernel's max. latency
--> a truely amazing feat of software engineering!!

But, the RT patch is still not in the kernel mainline, because:
It lowers aggregate throughput

Some low-end platforms lack the hardware support to implement RT well.

Trouble in User Space

PREEMPT_RT does not address User Space latency.

Modern, popular programming environments and languages
- Often sacrifice determinism for ease of use
- May "automagically" invoke time-consuming algorithms.

Software Libraries are black boxes by design
- APIs specify inputs and outputs, but rarely compute time.

Applications with challenging hard timing deadlines are often forced to utilize low-level programming and to reimplement existing libraries.

Even carefully written User Space code, running at "Real-Time" priority, my find itself contending with other user space processes for commonly accessed resources.

M B A R I

Biological Inspiration

Our cerebral cortex shares many qualities of a typical Linux computer
  It is very complex, flexible, and, sometimes, even fair.
Humans are blissfully unaware of firing of individual muscles for
    walking, talking, eating, digestion, etc.
Routine activities are controlled by our peripheral nervous system.
Our cerebrum focuses on analyzing and responding to unusual stimuli
  at a high level.
Our cerebellum, or "little brain", coordinates stimulation with motion
  It is our center for real-time control and perception

Interestingly, humans can function without their cerebellum, but:

```
the resulting quality of life is significantly compromised
with clumsiness, ...,
slowing of various cognitive perceptual processes, and
impaired fine motor and ocular-motor coordination.
```

http://jcn.sagepub.com/content/17/1/1.abstract

Partition the Problem

Identify what event-response loops have the most demanding deadlines

Factor only these critical loops into a separate, streamlined executable(s)

This is your real-time application's "Cerebellum"
- Insulate your main application logic from timing constraints!
- Implement it in a system programming language (like 'C' or C++)
- Minimize use of 3rd-party libraries

Connect to the non-time critical parts application parts via queues,
- Real-time parts must block, waiting to communicate results

Now you are are ready to...

Distribute Control (Virtually)

Run your real-time event response loops on reserved computing resources
Initally, try using virtual computing resources

Linux processes with Real-Time priority
- Most convenient option
- But it is not very effective without an RT-patched kernel
- If you can, dedicate a core to RT processes!
- Use shared memory to communicate with main app
- Complete access to Linux kernel and user space
- But you risk priority inversion

Real-time tasks running with Linux in a hypervisor environment
- Less convenient
- Works quite well even without an RT-patched kernel
- Hypervisor specific IPC mechanisms for comms with main app
- No easy access to Linux kernel and user space
- No danger of priority inversion
- Still vulnerable to trashing CPU caches

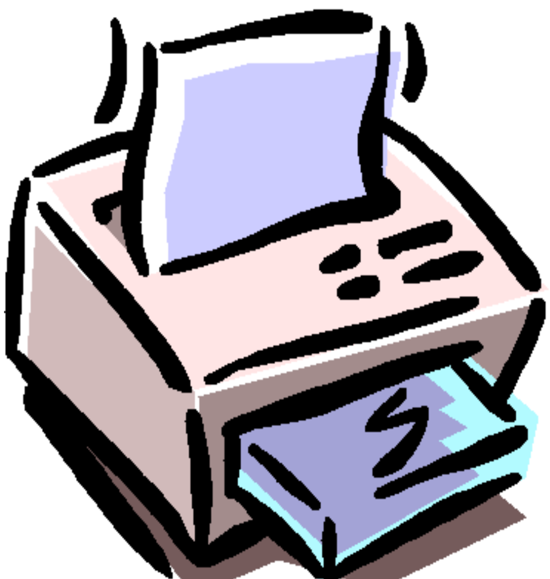
http://wiki.ok-labs.com/

Distribute Control (Physically)

Dedicate microcontrollers to your critical event-response loops

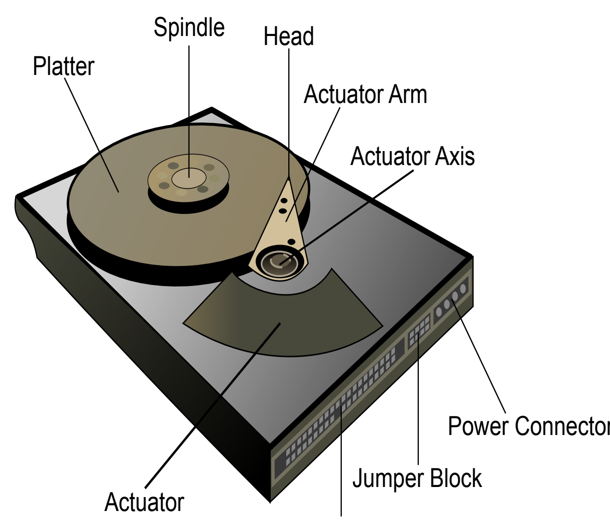
The least "convenient" option, but offering:
- No need for a RT-patched Linux kernel
- *Much more deterministic response times*
- No possibility of thrashing CPU caches
- Fewer resource contention issues
- *Much lower power consumption*
- ***Ability to safely limp or shutdown if host computer crashes***

But, you must program on "the bare metal" or small Real-Time OS



I don't think so

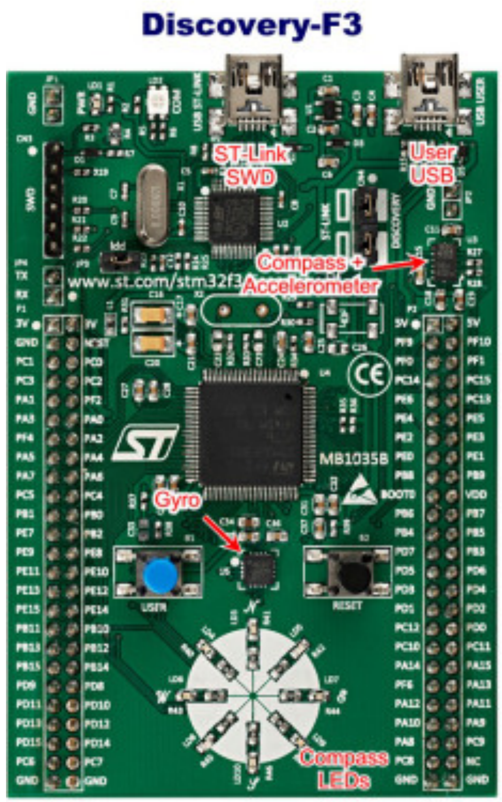
Linux Inside!



Dedicated DSP(s)

Microcontrollers Close the Loop

Microcontrollers are cheap and many use the GNU Compiler Collection
- Support for remote target debugging



~~Caches~~

For Example:

$15 USD/each

256kB Flash
48kB Fast Static RAM
72 MIPS
Analog I/O
High Resolution Timers
Eclipse Based IDE

http://hackaday.com/2012/11/15/in-depth-comparison-at-stm32-f3-and-f4-discovery-boards/

Disadvantages:
- Custom hardware design
- Generally, a lot slower than most x86 systems
- No shared memory with Linux host possible
  - Must use some form of physical networking

Smart I/O

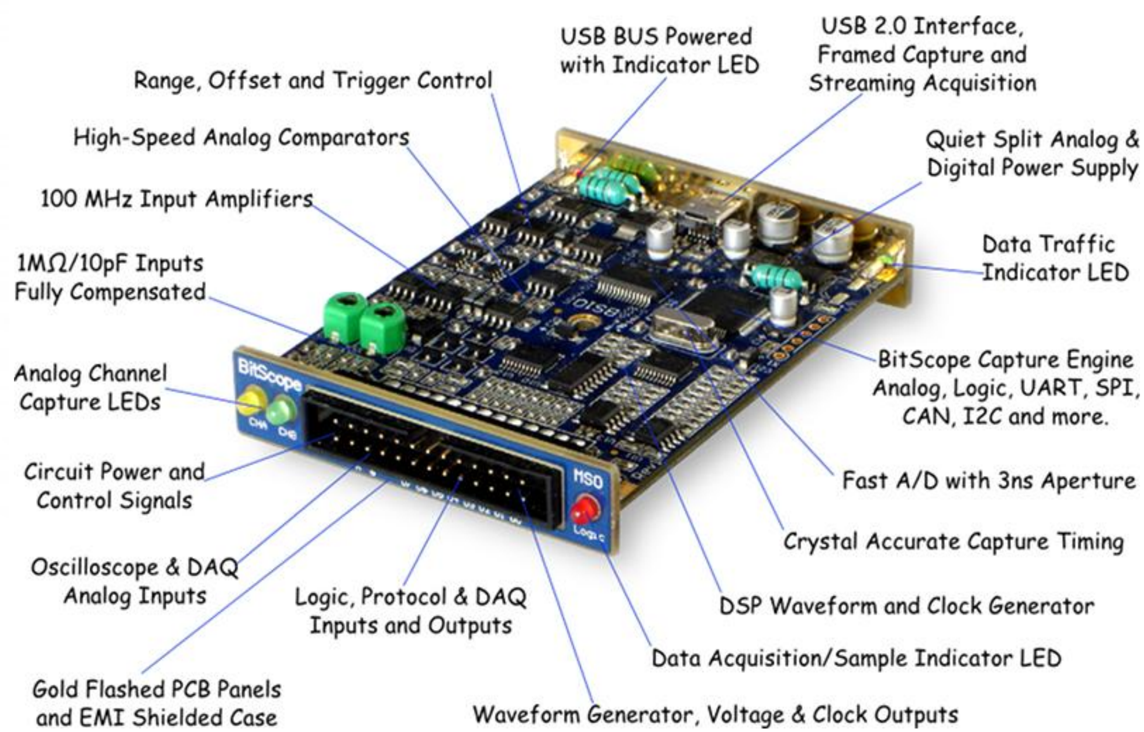
If you need specialized I/O...

You can likely find a microcontroller that already incorporates it.

Many USB I/O extenders are just such microcontrollers

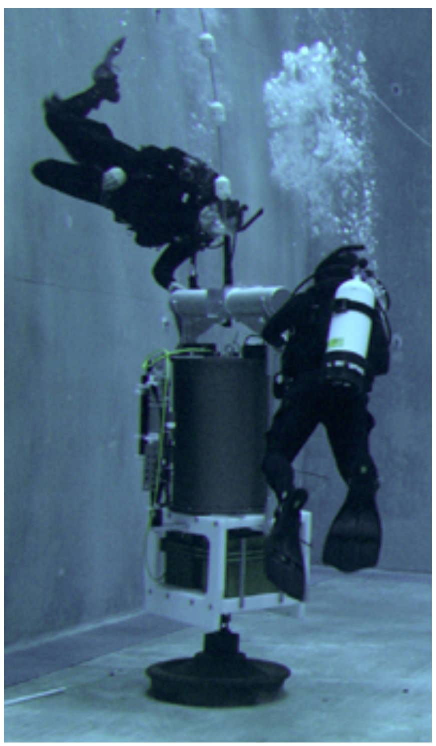
Programmed to provide bit-level access to their built-in peripherals

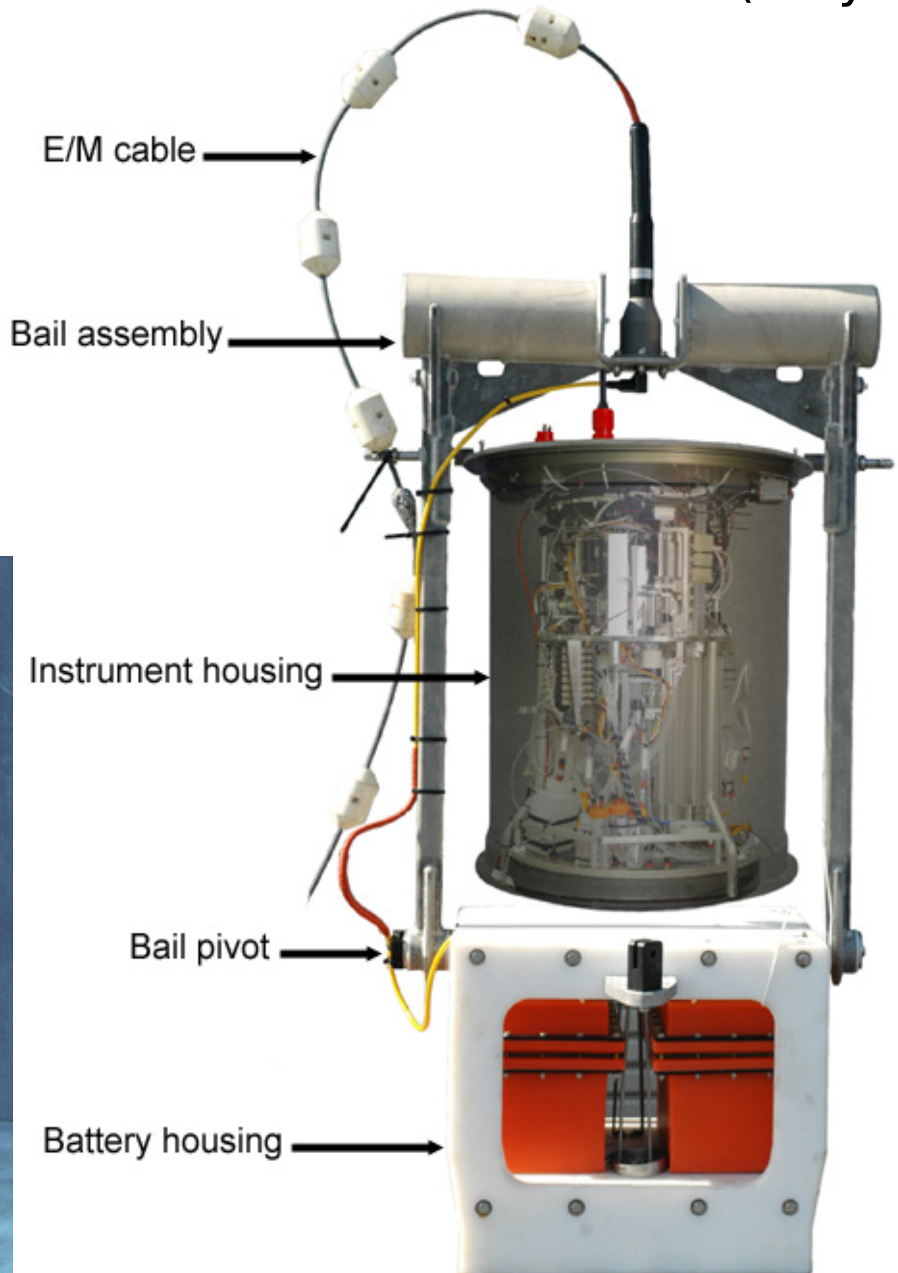
But, with custom programming, they can do much more!



Environmental Sample Processor

A (very complicated) Water Sampler



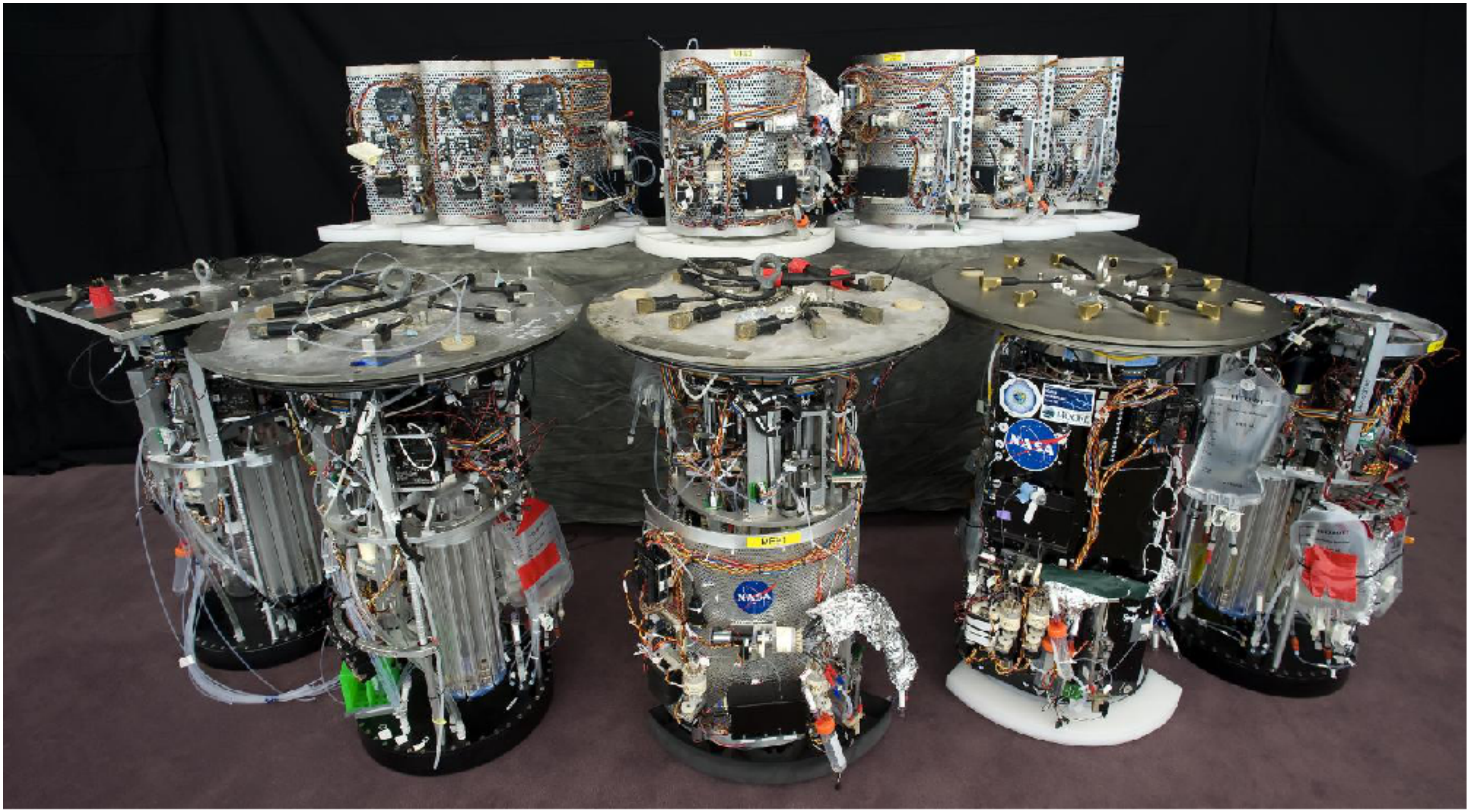


Filters 1 to 4 liters of water
Ruptures cells it collects
Extracts DNA and RNA
Identifies Species
Detects Algal Toxins
Radios results in hours

A robotic,
molecular biology
"Lab in a Can"

Distributed Control Case Study

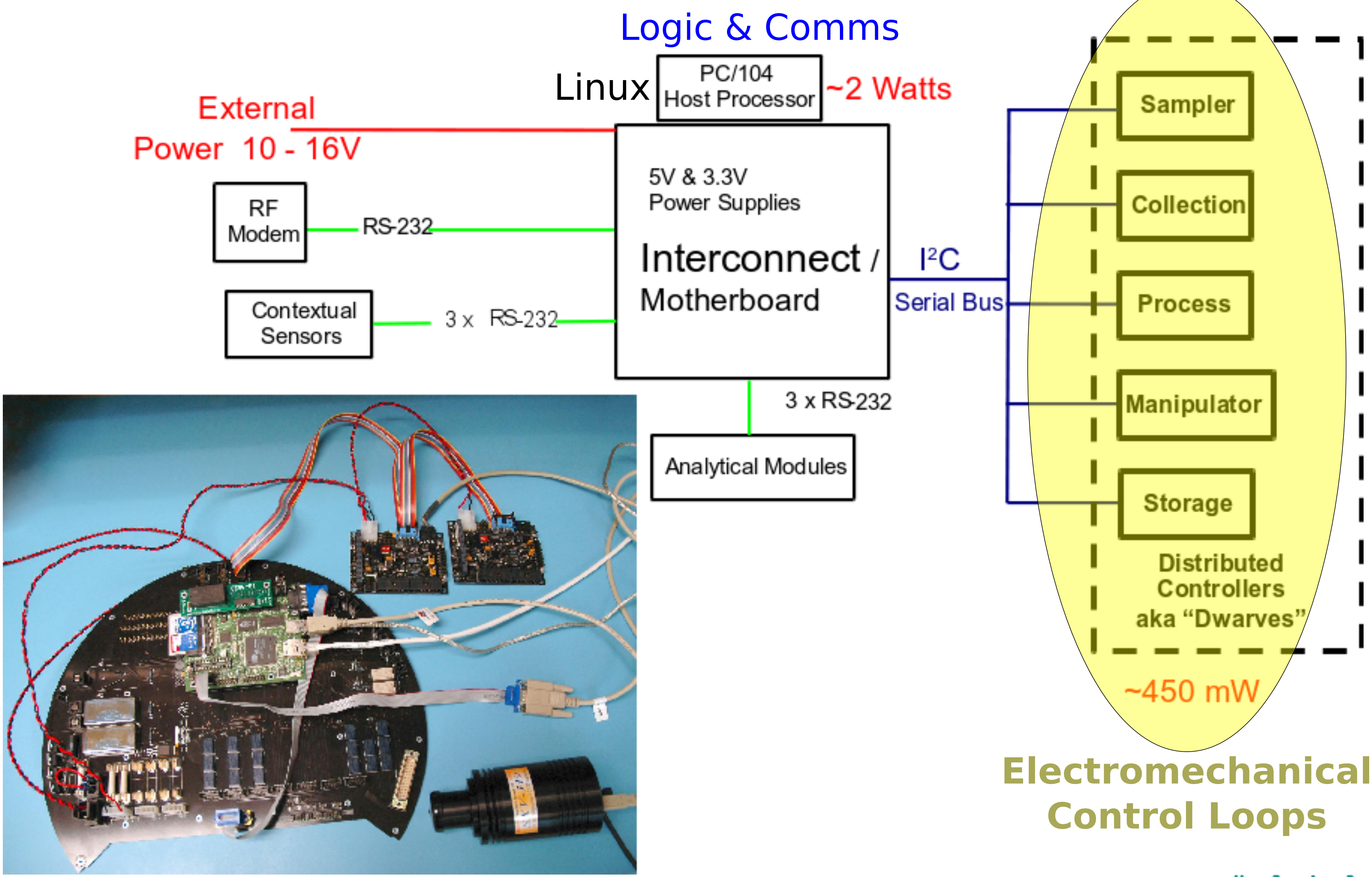
Ten year old hardware design, ARM9 @200Mhz, ~90 Bogomips
Linux 2.4 kernel (no RT patch, Big Kernel Lock, IDE disables interrupts)
Host application written almost entirely Ruby 1.8 scripting language!!



TI MSP430 microcontrollers networked to the Linux host via I^2C
control heaters and a dozen or so servo motors updated at 64hz.

M B A R I

Control System Electronics



Real-Time Rx

Partition your problem into Real-Time and non Real-Time tasks

Decouple different time domains (with queues)

Dedicate computing resources to Real-Time tasks

Consider dedicated CPUs optimized for deterministic response

Linux will sometimes be only part of the solution