Prototyping Cyber-Physical Systems
A hands-on approach to the Cyber-part

Sagar Behere
04 July 2014
Kungliga Tekniska Högskolan
Disclaimer

This presentation contains personal opinions
#include <stdio.h>

main(t, _, a)
char *a;
{ return !0 < t ? t < 3 ? main(−79, −13, a) + main(−87, 1 − _,
main(−86, 0, a + 1) + a)) : 1, t < __ main(t + 1, _, a) : 3, main(−94, −27 + t, a)
) & & t == 2 ? __ main(2, __ + 1, "%s %d %d\n") : 9:16: t < 0 ? t < −72 ? main(_,
'@n' + #'/*/{}w+/w#cdnr/+{r/°*%e}+/*/{s+,/w#q#n+//#{!,+/n{n+n+
,+/#n+,/#:q#n+,/+k#;*+,/'r:'d*3,}{w+K w'K:+'e#';dq#'l q#'d'K#/ /=+k#;q#'r}eKK#{w'r}eKK{nl]'/#;#q#n'}{#}w'}{c{nl]'/+#n';d}qw' i:# }{n\n'l!}/n{n#: ]; r{#w'r nc{n]}'/#{l,'+K {rw' iK{[{{nl]'/w#q#\n'wk nw' iwK{KK{nl]'/w{%'l#=#w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \\
;:{nl'−{r}w}'/+,,{#'}*#nc,.;#nw']/+kd'+e;+;
#'rdq#w' nr'/' ') }+}{rl'#{n'}#'}'+}##(!/"))
:t<−50? _==*a putcha r(a[31]): main(−65,__,a+1): main(*(a == '/') + t, __, a\n+1) : 0 < t main(2, 2, "%s") * a == '/' || main(0, main(−61, __, "!ek;dc \n@bK'(q)−[w]*%n+r3#l, {}/nuwloca−O;m .vpbks,fxtndCeghiry"), a + 1);}
Which systems are we talking about?

- Prototypes!!
  - Validation of concepts
- Your hobby projects
- Projects you'll be involved in as researchers
  - E.g.: EU FP7 projects in robotics
- Anything where it is not necessary to trim the system down to the leanest possible
  - in terms of hardware and software
Which systems are we talking about?

- Low quantities (not mass production) or one off designs
- Professional, certified tools not always available/used
- Professional software shops not utilized
- Multiple domain experts working on the project
  - Most are not good up-to-date programmers
- No concerns about conformance to industrial safety standards or product certification
Hardware scale

- **Individual microcontrollers**
  - 8, 16, 32 bit
  - PIC, AVR,...
- **Starter kits for above**
  - Typically with some peripherals on-board
  - LEDs, keypads, pots, LCD display, ...
- **Medium**
  - Typically based on ARM
  - Beaglebone, Raspberry Pi, ...
  - USB, ETH, WiFi,...
- **Big league**
  - "Proper" Intel processors
  - Core i7 etc.
  - Small form factor, SSDs
Software scale

- Bare metal
- Tiny OSes
  - Typically compiled into the application
  - e.g. FreeRTOS, Erika Enterprise
- Big league
  - Linux, Windows
Proposition

Use the fattest stack possible
(and build up proficiency)

Use an operating system if at all possible

But think of i/o and realtime constraints
Suggested pattern

- uC #1: FAT STACK
  - Ethernet, USB, WiFi, Bluetooth, Cameras, CAN, XBee?

- uC #2: Low level I/O
  - Serial, GPIO, SPI, I2C, CAN?, XBee?

- uC #3: REALTIME
Why not low level i/o with Linux?

- Kernel space programming is hard
different
- Need to write drivers + user
libraries
  - Think: Concurrency, blocking,
  reentrancy,...
- Mistakes can crash entire system
- Debugging kernel more difficult

Situation different if you have good drivers available
Hard vs Soft Realtime

- **Hard realtime**
  - strict determinism
  - bounded latencies
  - guaranteed worst case timing
    - → Industrial control, automotive, avionics, medical

- **Soft realtime**
  - Execute a task according to a desired time schedule on average
  - Best effort
    - → audio, video, VoIP

[source: Detlev Zundel's CC-BY-SA licensed presentation 'The Xenomai Real-Time Development Framework']
Temporal determinism

- Simple microcontrollers are temporally deterministic. Given an instruction sequence and the clock frequency, one can calculate the execution time.

- Modern CPUs are not deterministic in this sense. Innovations like caches, instruction scheduling, predictive execution, bus scheduling, etc. make it impossible to calculate execution times even of small instruction sequences. A paper at RTLWS11 showed that such execution timings pass standard randomness tests! Although peak performance increased by a factor of 20000 in the last 30 years, worst case execution time decreased only by a factor of 200.

[source: adapted from Detlev Zundel's CC-BY-SA licensed presentation 'The Xenomai Real-Time Development Framework']
Is realtime needed?

- What deadlines does the system have?
- Does the system have to meet each and every deadline?
- Can the system be split into a realtime and non-realtime part?
- Can the realtime constraints on software be eliminated by using suitable hardware?

[source: adapted from Detlev Zundel’s CC-BY-SA licensed presentation ‘The Xenomai Real-Time Development Framework’]
A fully preemptive kernel

[source: adapted from Detlev Zundel’s CC-BY-SA licensed presentation ‘The Xenomai Real-Time Development Framework’]
Degrees of preemption

Linux can be configured with different preemption models (in order of increasing preemption and decreasing performance):

**PREEMPT_NONE**

- no preemption, i.e. standard Unix behaviour (server configuration)

**PREEMPT_VOLUNTARY**

- explicit preemption points

**PREEMPT**

- implicit preemption points

**PREEMPT_RT**

- complete preemption (needs external patch)

[source: adapted from Detlev Zundel's CC-BY-SA licensed presentation 'The Xenomai Real-Time Development Framework']
Xenomai Adeos/I-Pipe architecture

[Source: adapted from Detlev Zundel's CC-BY-SA licensed presentation 'The Xenomai Real-Time Development Framework']
## PREEMPT_RT vs Xenomai

<table>
<thead>
<tr>
<th>Linux RT preempt</th>
<th>Dual kernel approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy for the software developers as “real-time” attributes can be adjusted after the design by juggling priorities</td>
<td>Clear separation of RT and non-RT domains. This usually leads to cleaner designs. Good RT performance.</td>
</tr>
<tr>
<td>no need for separate drivers</td>
<td>separate drivers are needed</td>
</tr>
<tr>
<td>test suite must cover all kernel configurations (i.e. modules)</td>
<td>small code base, maybe even certifiable</td>
</tr>
<tr>
<td>x86 centric</td>
<td>supports also low-end architectures (Blackfin, ARM, etc.)</td>
</tr>
</tbody>
</table>

[source: adapted from Detlev Zundel’s CC-BY-SA licensed presentation ‘The Xenomai Real-Time Development Framework’]
Application partitioning

Non-Realtime

HMI

Logging

Realtime

Task 1

Task 2

Filter

Realtime

Inputs

Filter

Outputs

Non-Realtime
Simulink models

Don’t ask the control engineer to write the controller in C++

- Code generation
  - Hand "massaging" almost always needed
  - Execution timing/jitter guarantees need to be assured ← tough!

- Direct execution
  - dSpace
  - xPC target
  - Arduino
  - Beagleboard (not realtime!)
Therefore the suggested pattern
Therefore the suggested pattern

Ethernet, USB, WiFi, Bluetooth, Cameras, CAN, XBees?

uC #1 FAT STACK

uC #2 Low level I/O

Serial, GPIO, SPI, I2C, CAN? XBees?

uC #3 REALTIME

But there is an annoyance...
Communication

How will you send this?

```c
struct {
    uint8_t fix;
    int32_t lat;
    int32_t lon;
    int32_t alt;
} t_gpsDataPayload;
```

or this?

```cpp
class gpsData {
    private:
        uint8_t fix;
        int32_t lat;
        int32_t lon;
        int32_t alt;
    public:
        uint8_t getfix();
        int32_t getlat();
        int32_t getlon();
        int32_t getalt();
};
```

 GCC’s `__attribute__((__packed__))` ?

Then never use -> or a pointer to the struct
Two aspects of communication

- Data transfer - protocols/mechanisms
  - TCP, UDP
  - Client/server, publish/subscribe, N-to-M, pipeline, ...

- Data packaging
  - serialization/deserialization a.k.a. marshalling/demarshalling
  - wire protocols
Communication solutions

There are solutions that do both transfer and de/marshalling

- CORBA, DDS
- Typically big and heavy
- Good luck running them on a small microcontroller

Solutions for transfer only

- Transfer a binary blob of data. Don’t care what’s inside it.
- Sender & Receiver need to know the actual data structure
- TCP/UDP client server is the traditional way BUT
- ZeroMQ is a modern way

Solutions for de/marshalling

- Google protocol buffers
- XML, JSON, BSON
- Boost serialization containers
Simulink direct execution

- Guess which modern communication methods are supported by Simulink?
Simulink direct execution

- Guess which modern communication methods are supported by Simulink?

- NONE!
- You are left banging bits together
Simulink direct execution

- Simulink supports UDP/TCP
  - UDP → packet fragmentation. Data MUST be less than packet size.
  - TCP → Non deterministic

- You need a simple protocol
  - First 4 bytes → Message type
  - Make sure to get endian-ness right
  - Check padding of data structures
  - Tip: Do the hard work in Simulink. At other side, use memcpy() to copy into struct buffer

- Maybe you could use the CAN bus
  - Message frames usually restricted to 8 bytes
    - If your data is uint64_t ...
Maximizing the fat stack

- If the hardware can run a proper Linux distribution (e.g. emdebian)
  - You have access to a gadzillion libraries..
  - .. and a bazillion languages
- C, C++, Java, Python, Ruby, Scala, Haskell, Erlang, ...
- Don’t be afraid to use multiple languages
  - Some language might have a library with the exact functionality you need
  - Switching from a procedural to functional language may solve a sub-problem elegantly
  - Some things are simply easier in high level languages (text processing in C? Eeeek!)

- Learn Inter-Process Communication (IPC)
  - Pipes, FIFOs, sockets, shared memory, mailboxes, queues
Data logging

- Data logging is not realtime [unless it is ;-) ]
  - Needs to be done from a non-realtime task
  - Or preferably, on a separate computer
- Typically, three things need to be logged
  - Timeseries data ← periodic
  - Error, exception and non-error messages ← event driven
  - Data associated with errors and exceptions ← event driven
- Periodic timeseries data size usually known in advance
- Event driven messages and associated data may have unknown size
- Tip: Log data in open and interoperable formats
  - Logs can be viewed in general purpose data analysis tools
  - Formats like csv, netCDF, HDF5 are desirable
  - Analyse in Matlab, GNU Octave, kst, Qtiplot or your own program
HMI and Calibration

- GUI **must** run in a separate thread, or better, in an independent process
  - Receives data via IPC, typically sockets
  - So HMI and calibration can run on different computer
- Make sure that received calibration data is sanitized!
- A useful pattern for displaying data in HMI

![Data stream diagram]

- Data stream to be logged
  - Filter
  - HMI, Data store
Another useful pattern

System

Concerns of data transfer and de/marshalling still valid
A logging workaround
Communication: ZeroMQ

- Data transfer independent of platform and language
- Carries messages across inproc, IPC, TCP, TPIC, multicast
- Smart patterns like pub-sub, push-pull, and router-dealer
- High-speed asynchronous I/O engines
- Excellent documentation [which begins with the phrase, "Fixing the World" ;-) ]
- Open source (LGPL with static linking exception), active community
- http://www.zeromq.org
Communication: DDS

- Interoperable publish-subscribe with QoS
- Data transfer as well as packaging
- Fault tolerance (over unreliable media)

**NOT THIS:**
(connection-oriented)

**BUT THIS:**

![Diagram of Shared Operational Picture with System Components]

○ = System Components
Clock synchronization

- If you have multiple computers in the system, the clocks often need to be synchronized.
  - But try to avoid this as far as possible, via smart architecture choices.
- For simple microcontrollers, possible to use global clock signal.
- `ntpd` can (theoretically) sync clocks within 232 picoseconds.
- You can even sync to GPS time, if your system uses a GPS.
  - But the GPS device should have a PPS signal.
My three favorite platforms

Between them, they can take on practically anything
Beaglebone black (or white)

- **10/100 Ethernet**
- **USB Host**
  - Easily connects to almost any everyday device such as mouse or keyboard
- **microHDMI**
  - Connect directly to monitors and TVs
- **microSD**
  - Expansion slot for additional storage
- **512MB DDR3**
  - Faster, lower power RAM for enhanced user-friendly experience
- **Serial Debug**
- **DC Power**
- **Expansion headers**
  - Enable cape hardware and include:
  - 65 digital I/O
  - 7 analog
  - 4 serial
  - 2 SPI
  - 2 I2C
  - 8 PWMs
  - 4 timers
  - And much much more!
- **1 GHz Sitara AM335x ARM® Cortex™-A8 processor**
  - Provides a more advanced user interface and up to 150% better performance than ARM11
- **Power Button**
- **LEDS**
- **Reset Button**
- **USB Client**
  - Development interface and directly powers board from PC
- **2GB on-board storage using eMMC**
  - Pre-loaded with Ångström Linux Distribution
  - 8-bit bus accelerates performance
  - Frees the microSD slot to be used for additional storage for a less expensive solution than SD cards
Beaglebone PRUs

Separate realtime processors on the silicon of main chip

- Dual 32-bit RISC cores, shared data, instruction memories and an interrupt controller (INTC)
- 8KB data memory and 8KB instruction memory
- 12KB shared RAM
- A small, deterministic instruction set
Arduino

- Easy, easy, easy
- Wide variety of devices
- Naturally realtime
- Matlab/Simulink integration makes it the poor man’s dSpace
OROCOS dataflows
Example: Autonomous maze solving robot

- Ball Detector
- Motion
- Explore
- Feature Extractor
- Supervisor
- NavGraph
- Sensor Data
- Data Logger
- SLAM
Example: Robot motion control

Non Real-time

- SOAP Server
- Command Processor
- Path Planner
- Trajectory Generator
- Sequencer
- Scene Description
- Robot Description (Plugin)
  - Kinematics
  - Dynamics
  - Manufacturer API
- Reporter

Real-time

- Interpolator
- Controller
- Estimator e.g.: Kalman Filter
- Parameter Estimator
- Exception and Events Processor
- Sensor
- Actuator

External Visualization Program e.g.: Peekabot

Prototyping Cyber-Physical Systems
Some resources

- "How fast is fast enough? Choosing between Xenomai and Linux for realtime applications" - Brown and Martin
- "The Xenomai real-time development framework: Recent and future developments" - Detlev Zundel
- "Middleware trends and market leaders 2011" - Dworak et al
- ZeroMQ guide
- "DDS - Advanced Tutorial using QoS to solve real world problems" - Gordon Hunt, OMG Real-Time & Embedded Workshop July 9-12, Arlington, VA
- OROCOS component builders manual
Recap: What have we seen?

- A pattern for system partitioning
- Two ways of achieving realtime with Linux and their pros/cons
- Data communication - transfer and packaging
- Data logging
- Clock synchronization
- Some useful platforms
- OROCOS Middleware
Questions?

behere@kth.se