Prototyping Cyber-Physical Systems

A hands-on approach to the Cyber- part

Sagar Behere

23 June 2015

Kungliga Tekniska Högskolan

Disclaimer

This presentation contains personal opinions

What does this program do?

```c
#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 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/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'){){nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{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'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),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

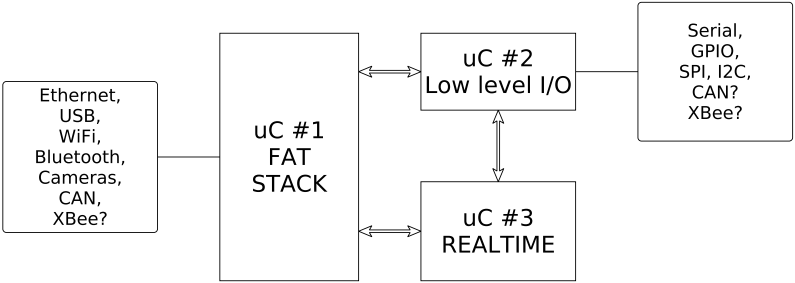
Proposition

Use the fattest stack possible
(and build up proficiency)

Use an operating system if at all possible

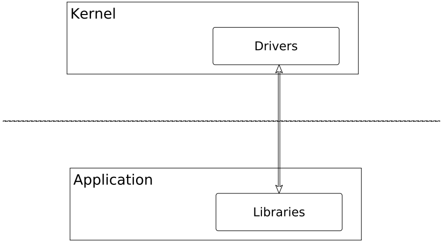
But think of i/o and realtime constraints

Suggested pattern



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

  $\Longrightarrow$ Industrial control, automotive, avionics, medical
- Soft realtime
  - Execute a task according to a desired time schedule on **average**
  - Best effort

  $\Longrightarrow$audio, video, VoIP

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

Temporal determinism

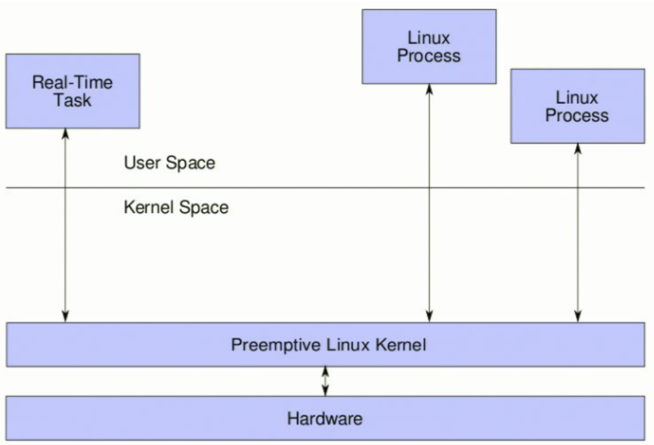
- Simple microcontrollers are temporally deterministic. Given an instruction sequence and the clock frequency, one can calculate the execution time.
- Modern CPUs are **not** deterministic in this sense. Innovations like caches, instruction scheduling, predictive execution, bus scheduling, etc. make 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?

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)

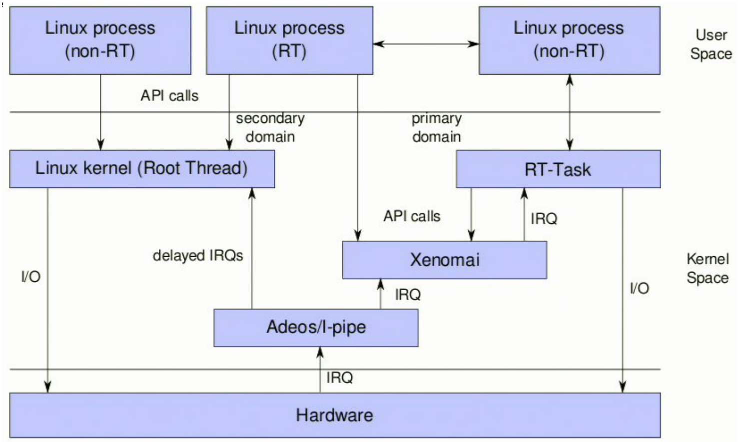
PREEMPT_VOLUNTARY
: explicit preemption points

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

Linux RT preempt

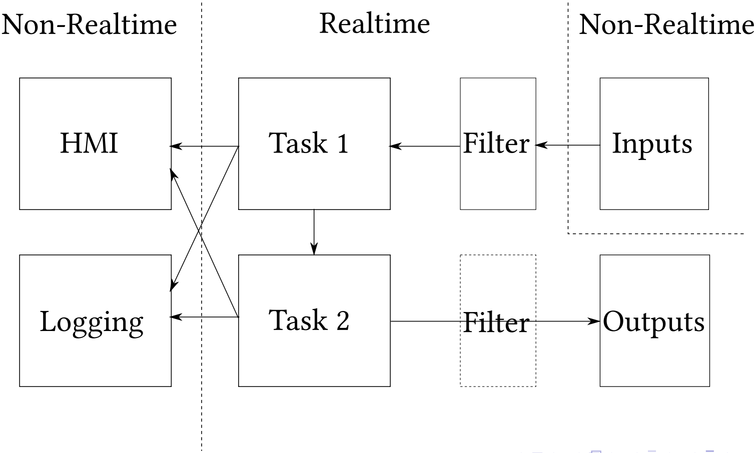
- Easy for the software developers as "real-time" attributes can be adjusted after the design by juggling priorities
- no need for separate drivers
- test suite must cover all kernel configurations (i.e. modules)
- x86 centric

Dual kernel approach

- Clear separation of RT and non-RT domains. This usually leads to cleaner designs. Good RT performance.
- separate drivers are needed
- small code base, maybe even certifiable
- supports also low-end architectures (Blackfin, ARM, etc.)

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

Application partitioning

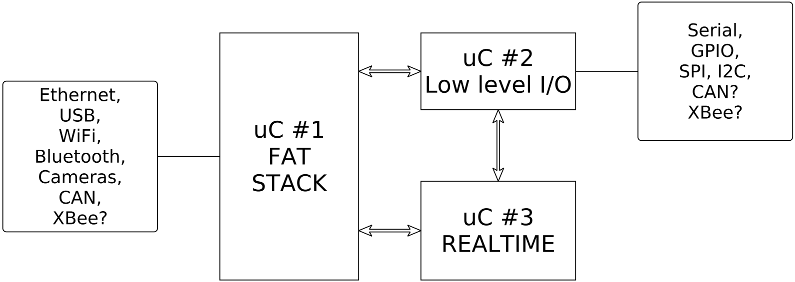


Simulink models

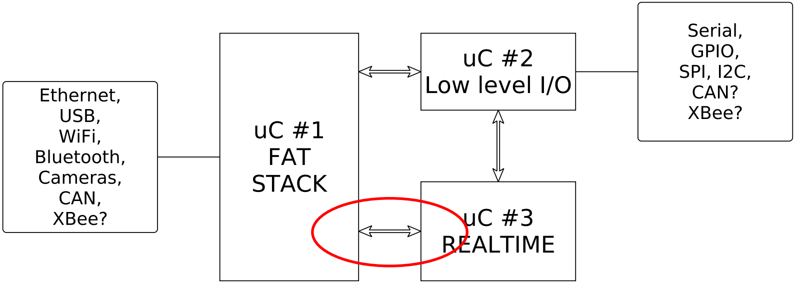
Don't ask the control engineer to write the controller in C++

- 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



But there is an annoyance...

Communication

How will you send this?

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

gcc's `__attribute__((__packed__))` ? Then never use -> or a pointer to the struct

or this?

```
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();
};
```

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

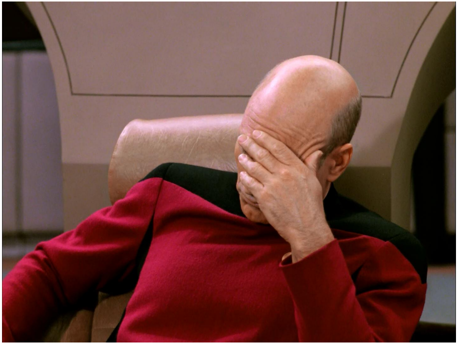
Simulink direct execution

- 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

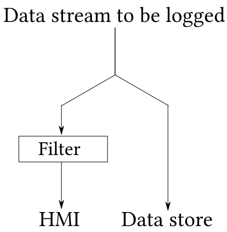
- If the hardware can run a proper linux distribution (e.g. emdebian)
  - You have access to a gadzillion libraries..
  - .. and a bazillion languages
- C, C++, Java, Python, Ruby, Scala, Haskell, Erlang, ...
- Don't be afraid to use multiple languages
  - Some language might have a library with the 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

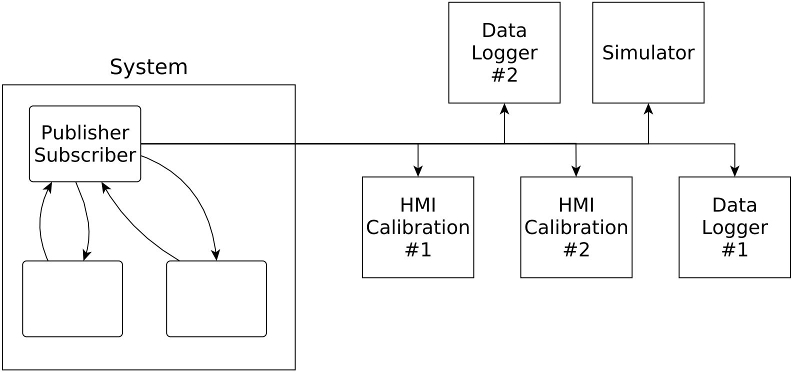
- Data logging is not realtime [unless it is ;-)]
  - Needs to be done from a non-realtime task
  - Or preferably, on a separate computer
- Typically, three things need to be logged
  - Timeseries data ← periodic
  - Error, exception and non-error messages ← event driven
  - Data associated with errors 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

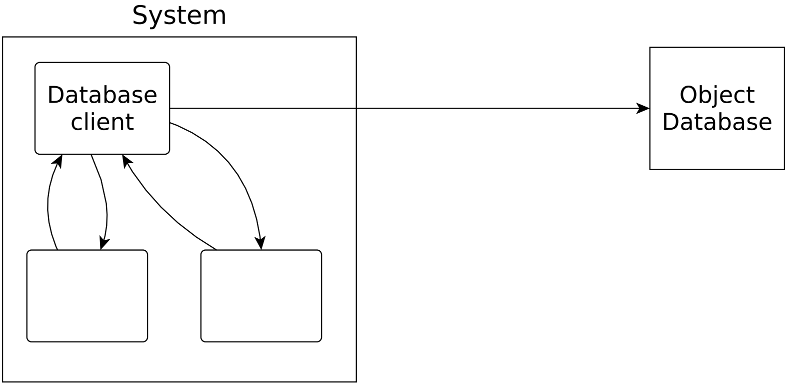


Another useful pattern



Concerns of data transfer and de/marshalling still valid

A logging workaround

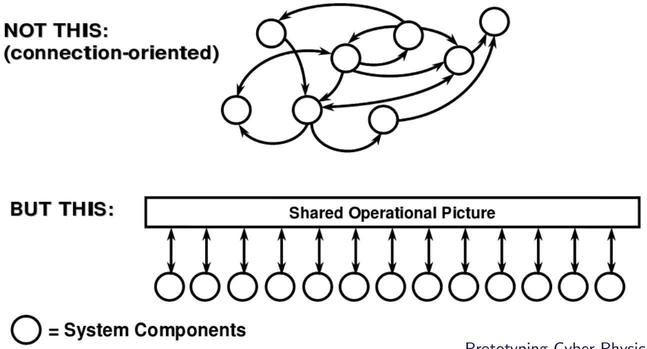


Communication: ZeroMQ

- 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

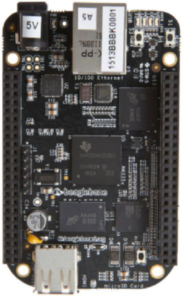
- Interoperable publish-subscribe with QoS
- Data transfer as well as packaging
- Fault tolerance (over unreliable media)
- http://www.opensplice.com , http://www.rti.com

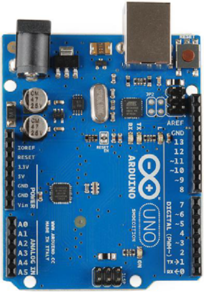


Clock synchronization

- 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

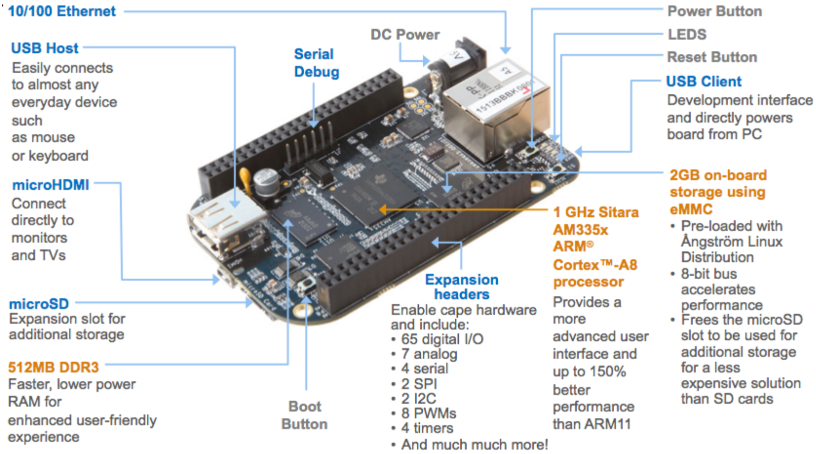
My three favorite platforms





Between them, they can take on practically anything

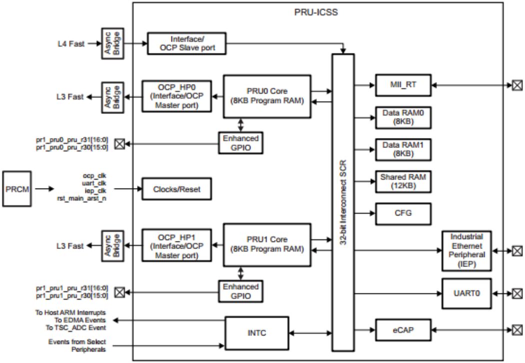
Beaglebone black (or white)



Beaglebone PRUs

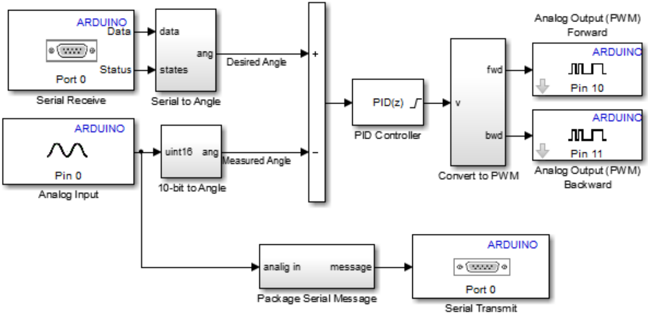
Separate realtime processors on the silicon of main chip

- 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

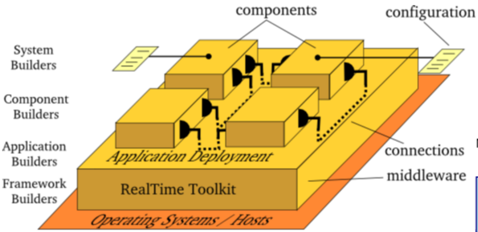


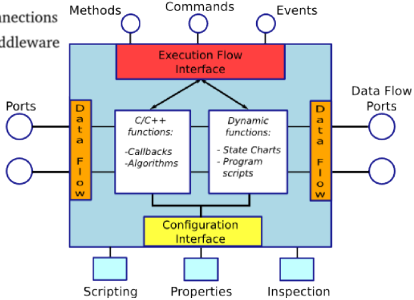
Arduino

- Easy, easy, easy
- Wide variety of devices
- Naturally realtime
- Matlab/Simulink integration makes it the poor man's dSpace

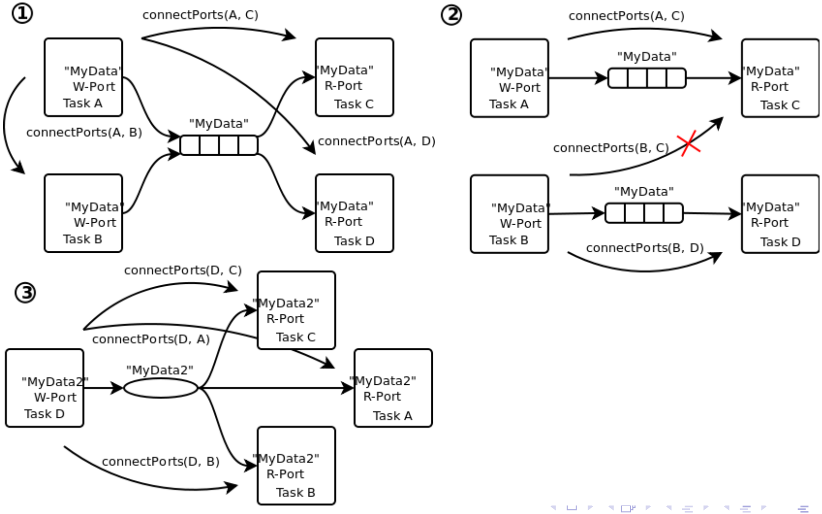


OROCOS

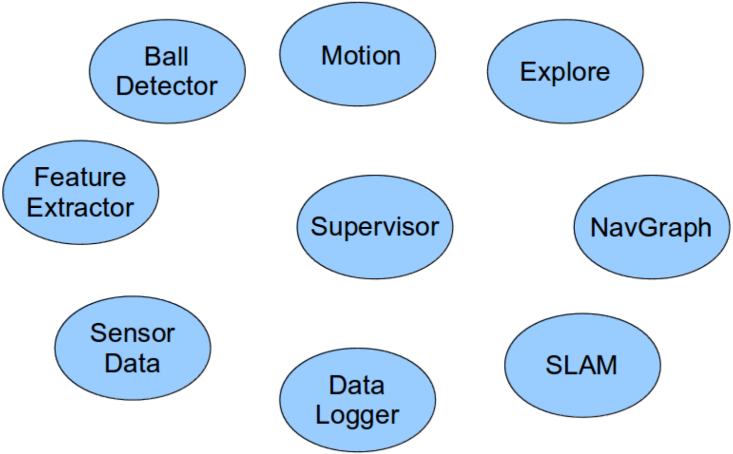

components
configuration
System Builders
Component Builders
Application Builders
Framework Builders
Application Deployment
connections
middleware
RealTime Toolkit
Operating Systems / Hosts


Methods
Commands
Events
Execution Flow Interface
Ports
Data Flow
C/C++ functions:
-Callbacks
-Algorithms
Dynamic functions:
- State Charts
- Program scripts
Data Flow
Data Flow Ports
Configuration Interface
Scripting
Properties
Inspection

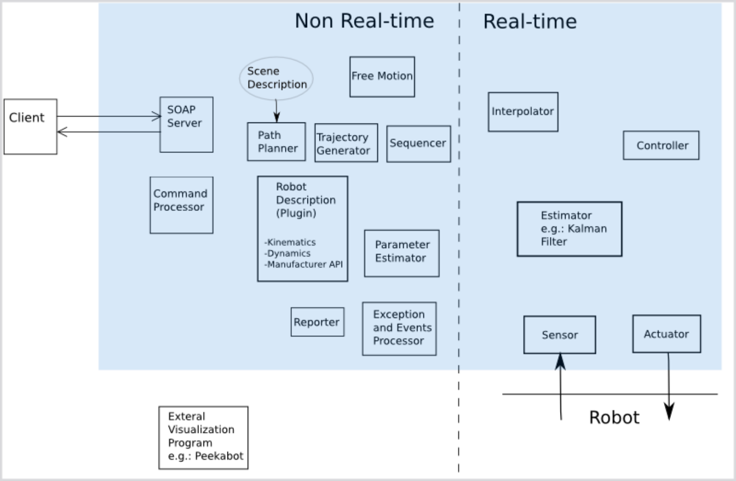
OROCOS dataflows



Example: Autonomous maze solving robot



Example: Robot motion control



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