OpenRTDynamics — A framework for the implementation of real-time controllers

Christian Klauer[1]
[1]Control Systems Group, Technische Universität Berlin
Kontakt: klauer@control.tu-berlin.de

May 2014

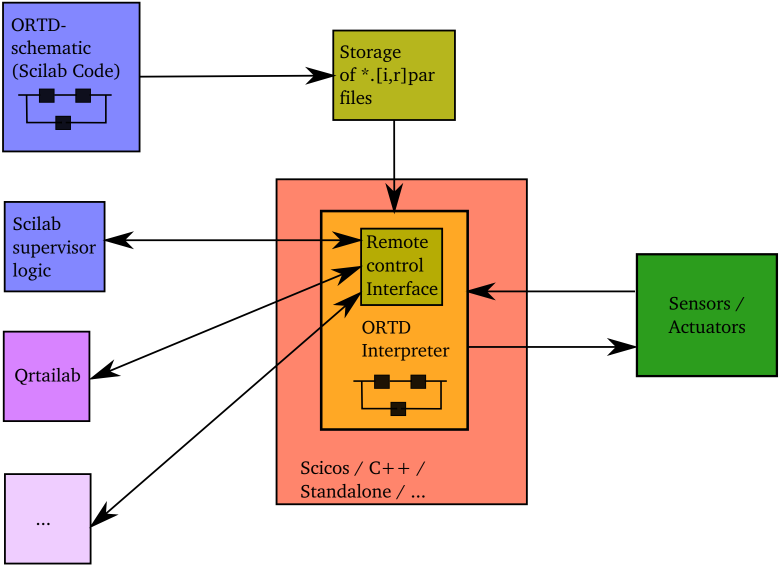
OpenRTDynamics contains

- A simulator/interpreter for discrete-time dynamical systems
- A Scilab-toolbox for describing these systems in an block/signal based way.
- Many plugins (e.g. state machines, threads (multiple main loops), UDP-communication, remote control via GUIs, ...)

Compared to other systems it features

- A new concept of defining schematics that enables well structured and easy maintainable code as projects get bigger.
- Nesting schematics of infinite depth: e.g. ability to reset and/or switch between sub-schematics (state machines).
- Specially defined parts of the real-time implementation can be exchanged on-line.

General Overview

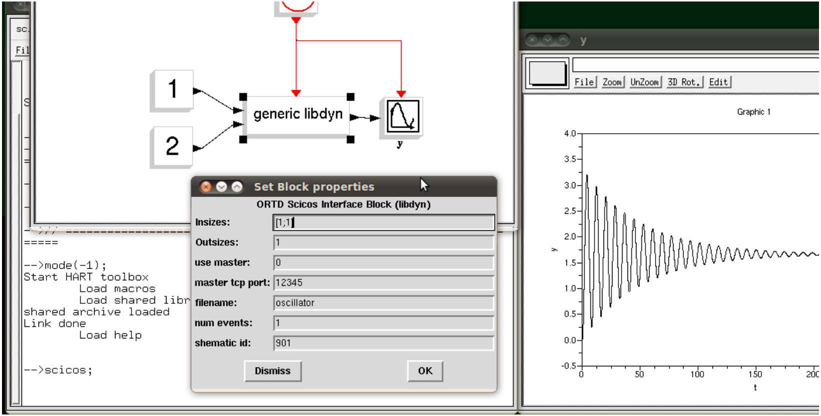


Target systems so far:

- Linux incl. realtime preemption
- Linux on ARM-based systems (Beaglebone, Raspberry Pi, ...)
- Android
- Easily portable if an implementation of Posix Threads is available.

Possible modes of operation

- Standalone applications using the `ortd`-command
- Embedded into a Scicos-block
- API for C++/C projects available `libortd.so`



- Specification of the in- and output port sizes along the name of the schematic to load.
- Multiple interface blocks within one Scicos diagram are possible

demo : bash

File Edit View Bookmarks Settings Help

```
chr@tamora:~/demo$ libdyn_generic_exec --baserate=100 -s oscillator_remote -i 901 -l
 0 -- 0 --master_tcpport 10000
Baserate set to 100
fnames ipar = oscillator_remote.ipar
fnames rpar = oscillator_remote.rpar
Using schematic_id 901
sched_setscheduler failed: Operation not permitted
Running without RT-Preemption
Initialising remote control interface on port 10000
...................................
.. Setting up new simulation  ......
...................................
Plugins are disabled in RTAI-compatible mode
libdyn: successfully compiled schematic
ringbuffer: allocated 10000 elements of size 8. 80000 bytes in total
filewriter: open logfile result.dat
x =  [0.000000, ].
x =  [0.000000, ].
x =  [1.000000, ].
x =  [2.990000, ].
x =  [5.954100, ].
x =  [9.870619, ].
x =  [14.712248, ].
x =  [20.446237, ].
x =  [27.034613, ].
x =  [34.434428, ].
x =  [42.598037, ].
x =  [51.473403, ].
x =  [61.004427, ].
x =  [71.131301, ].
x =  [81.790879, ].
```

demo : bash

- Simulation mode or real-time execution with RT-Preempt scheduling or using soft RT.

How schematics are defined:

- **Signals** are represented by a special Scilab variables.
- **Blocks** are defined by calls to special Scilab functions (`ld_`-prefix). They may take input signal variables and may return new signal variables.

An Example:

- A linear combination of two signals ($y = u_1 - u_2$) is implemented:

```
[sim, y] = ld_add(sim, defaultevents, list(u1, u2), [ 1, -1 ] );
```

- A time-discrete transfer function is implemented like:

```
[sim, y] = ld_ztf(sim, defaultevents, u, (1-0.2)/(z-0.2) );
```

Please Note:

- For all calculations the toolbox functions must be used. **Not possible:** `y = u1 - u2`, whereby `u1` and `u2` are signal variables.

Some more explanation:

```
[sim, y] = ld_add(sim, defaultevents, list(u1, u2), [ 1, -1 ] );
```

- The variable `sim` is technically used to emulate object-orientated behaviour in Scilab.
- `defaultevents` must be zero.

Help:

- A list of available blocks may be browsed using the Scilab-help functionality, e.g. try `help ld_add`.

Definition: Within Scilab by writing a function that describes the blocks and connections:

```
// This is the main top level schematic
function [sim, outlist]=schematic_fn(sim, inlist)
   u1 = inlist(1); // Simulation input #1
   u2 = inlist(2); // Simulation input #2

  // sum up two inputs
  [sim,out] = ld_add(sim, defaultevents, list(u1, u2), [1, 1] );

  // save result to file
  [sim, save0] = ld_dumptoiofile(sim, defaultevents, ...
       "result.dat", out);

  // output of schematic
  outlist = list(out); // Simulation output #1
endfunction
```

- It takes the simulation object `sim` as well as a list of in- and outputs.

Generation: A set of function calls trigger evaluation of the functions describing the schematic.

```
defaultevents = [0]; // main event

// set-up schematic by calling the user defined
// function "schematic_fn"
insizes = [1,1]; outsizes=[1];
[sim_container_irpar, sim]=libdyn_setup_schematic(schematic_fn, ...
                            insizes, outsizes);

// Initialise a new parameter set
parlist = new_irparam_set();

// pack simulations into irpar container with id = 901
parlist = new_irparam_container(parlist, sim_container_irpar, 901);

// irparam set is complete convert to vectors
par = combine_irparam(parlist);

// save vectors to a file
save_irparam(par, 'simple_demo.ipar', 'simple_demo.rpar');
```

- The schematic is saved to disk by `save_irparam`.

Execution:

- This Scilab-Script will generate two files `simple_demo.ipar` and `simple_demo.rpar`, which contain a encoded definition of the whole schematic.
- These file are then loaded by the provided interpreter library and executed.

Superblocks are introduced by writing a new Scilab function.

```
function [sim, y]=ld_mute(sim, ev, u, cntrl, mutewhengreaterzero)
    [sim, zero] = ld_const(sim, ev, 0);

    if (mutewhengreaterzero == %T) then // parametrised functionality
      [sim,y] = ld_switch2to1(sim, ev, cntrl, zero, u);
    else
      [sim,y] = ld_switch2to1(sim, ev, cntrl, u, zero);
    end
endfunction
```

- This example describes a superblock, which has two inputs `u` and `cntrl` and one output `y`.
- `mutewhengreaterzero` describes a parameter.
- **NOTE**: With the if / else construction a superblock can have different behaviour depending on a parameter! (This enables great possibilities for creating reusable code)

Superblocks: Usage

Once defined, the superblock can be used like any other ORTD-Block:

```
[sim, y] = ld_mute( sim, ev, u=input, cntrl=csig, ...
                    mutewhengreaterzero=%T )
```

How to implement feedback?

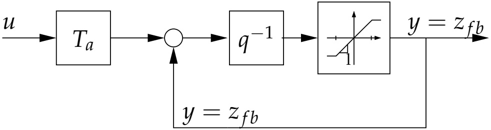
- A dummy signal is required, which can be used to connect a real block:

```
[sim, feedback] = libdyn_new_feedback(sim);
```

- Later in the ongoing code, the loop is closed via `libdyn_close_loop`, which means `feedback` is assigned to a real signal `y`:

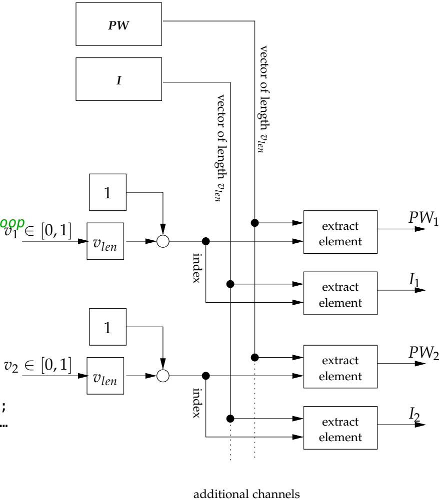
```
[sim] = libdyn_close_loop(sim, y, feedback);
```

```
function [sim, y]=limited_int(sim, ev, u, min__, max__, Ta)
  // Implements a time discrete integrator with saturation
  // of the output between min__ and max__
  //
  // u * - input
  // y * - output
  //
  // y(k+1) = sat( y(k) + Ta*u , min__, max__ )

    [sim, u__] = ld_gain(sim, ev, u, Ta);

    // create z_fb, because it is not available by now
    [sim,z_fb] = libdyn_new_feedback(sim);

    // do something with z_fb
    [sim, sum_] = ld_sum(sim, ev, list(u__, z_fb), 1, 1);
    [sim, tmp] = ld_ztf(sim, ev, sum_, 1/z);

    // Now y becomes available
    [sim, y] = ld_sat(sim, ev, tmp, min__, max__);

    // assign z_fb = y
    [sim] = libdyn_close_loop(sim, y, z_fb);
endfunction
```



Example: Lookup table for multiple channels

```
function [sim, I_list, PW_list]=ld_charge_cntrl_multich(sim, ev, v_list, Nch)
//
// Charge control for multiple channels using the same lookup table
//

   // Get table data and their lengths
   tabPW_ = CHCNTL.tabPW;
   tabI_ = CHCNTL.tabI;
   vlen = length(tabI_);

   // The vectors containing the tables
   [sim,TABI] = ld_constvec(sim, ev, tabI_ );
   [sim,TABPW] = ld_constvec(sim, ev, tabPW_ );

   // init output lists
   I_list = list(); PW_list = list();

   // loop
   for i=1:Nch // Create blocks for each channel by a for loop

      // extract normalised stimulation for channel i
      v = v_list(i);

      // calc index
      [sim, index] = ld_gain(sim, ev, v, vlen);
      [sim, index] = ld_add_ofs(sim, ev, index, 1);

      // look up the values
      [sim, I] = ld_extract_element(sim, ev, TABI, index, …
                                    vecsize=vlen );
      [sim, PW] = ld_extract_element(sim, ev, invec=TABPW, …
                         pointer=index, vecsize=vlen );

      // store signals
      I_list($+1) = I; PW_list($+1) = PW;
   end
endfunction
```



Principle:

- Each state is represented by a one sub-schematic.
- Indicating the current state, only one schematic is active at once. State related computations are performed while the state is active.
- The active sub-schematic may cause the transition to another state at any time.

Extra features:

- When a state is left, the corresponding schematic is reset.
- Possibility to share variables among states; to e.g. realise counters, ...

State machines: implementation

One superblock-function is evaluate for each state. Differentiation among states may be realised by using a `select` statement:

```
function [sim, outlist, active_state, x_global_kp1, userdata]=state_mainfn(sim, ...
                                          inlist, x_global, state, statename, userdata)
  printf("defining state %s (#%d) ... userdata(1)=%s\ n", statename, state, userdata(1) );

  // define names for the first event in the simulation
  events = 0;

  // demultiplex x_global
  [sim, x_global] = ld_demux(sim, events, vecsize=4, invec=x_global);

  // sample data fot output
  [sim, outdata1] = ld_constvec(sim, events, vec=[1200]);

  select state
    case 1 // state 1
      // wait 10 simulation steps and then switch to state 2
      [sim, active_state] = ld_steps(sim, events, activation_simsteps=[10], values=[-1,2]);
      [sim, x_global(1)] = ld_add_ofs(sim, events, x_global(1), 1); // increase counter 1 by 1
    case 2 // state 2
      // wait 10 simulation steps and then switch to state 3
      [sim, active_state] = ld_steps(sim, events, activation_simsteps=[10], values=[-1,3]);
      [sim, x_global(2)] = ld_add_ofs(sim, events, x_global(2), 1); // increase counter 2 by 1
    case 3 // state 3
      // wait 10 simulation steps and then switch to state 1
      [sim, active_state] = ld_steps(sim, events, activation_simsteps=[10], values=[-1,1]);
      [sim, x_global(3)] = ld_add_ofs(sim, events, x_global(3), 1); // increase counter 3 by 1
  end

  // multiplex the new global states
  [sim, x_global_kp1] = ld_mux(sim, events, vecsize=4, inlist=x_global);

  // the user defined output signals of this nested simulation
  outlist = list(outdata1);
endfunction
```

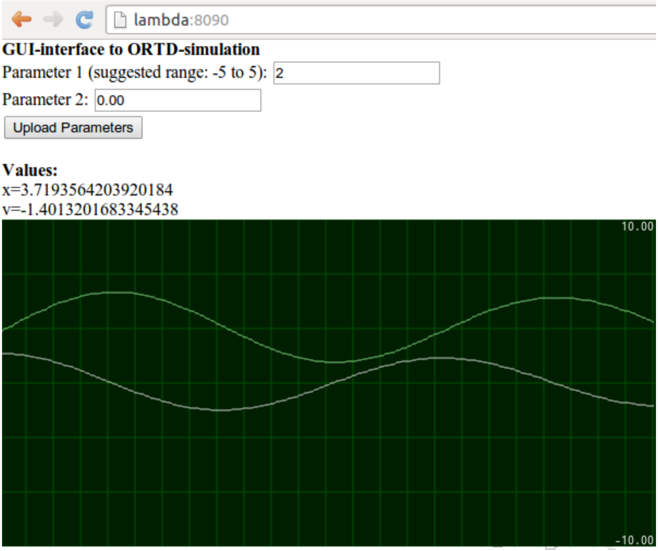
```
// The simulation running in a thread
function [sim, outlist, userdata]=Thread_MainRT(sim, inlist, userdata)
  [sim, Tpause] = ld_const(sim, 0, 1/27);  // The sampling time that is constant at 27 Hz
  [sim, out] = ld_ClockSync(sim, 0, in=Tpause); // synchronise this simulation

  // print the time interval
  [sim] = ld_printf(sim, 0, Tpause, "Time interval [s]", 1);

  // save the absolute time into a file
  [sim, time] = ld_clock(sim, 0);
  [sim] = ld_savefile(sim, 0, fname="AbsoluteTime.dat", source=time, vlen=1);

  outlist = list();
endfunction

// Start a thread
ThreadPrioStruct.prio1=ORTD.ORTD_RT_NORMALTASK; // or  ORTD.ORTD_RT_REALTIMETASK
ThreadPrioStruct.prio2=0;  // for ORTD.ORTD_RT_REALTIMETASK: 1-99 (man sched_setscheduler)
                           // for ORTD.ORTD_RT_NORMALTASK this is the unix nice-value
ThreadPrioStruct.cpu = -1; // The CPU on which the thread will run; -1 dynamically assigns to a CPU,
                           // counting of the CPUs starts at 0

[sim, StartThread] = ld_initimpuls(sim, 0); // triggers the computation only once
[sim, outlist, computation_finished] = ld_async_simulation(sim, 0, ...
                              inlist=list(), ...
                              insizes=[], outsizes=[], ...
                              intypes=[], outtypes=[], ...
                              nested_fn = Thread_MainRT, ...
                              TriggerSignal=StartThread, name="MainRealtimeThread", ...
                              ThreadPrioStruct, userdata=list() );
```

Remote Control Interface – I

- Send and receive data/streams via packet-based communication channels, e.g. UDP

```
// The configuration of the remote communication interface
Configuration.UnderlyingProtocol = "UDP";
Configuration.DestHost = "127.0.0.1";
Configuration.DestPort = 20000;
Configuration.LocalSocketHost = "127.0.0.1";
Configuration.LocalSocketPort = 20001;
[sim, PacketFramework] = ld_PF_InitInstance(sim, InstanceName="RemoteControl", Configuration)

// Add a parameter for controlling the oscillator
[sim, PacketFramework, Input]=ld_PF_Parameter(sim, PacketFramework, NValues=1, ...
                        datatype=ORTD.DATATYPE_FLOAT, ParameterName="Oscillator input");

// The system to control
[sim, x,v] = damped_oscillator(sim, Input);

// Stream the data
[sim, PacketFramework]=ld_SendPacket(sim, PacketFramework, Signal=x, NValues_send=1, ...
                                     datatype=ORTD.DATATYPE_FLOAT, SourceName="X")

// finalise the communication interface
// and create a configuration file describing the protocol
[sim,PacketFramework] = ld_PF_Finalise(sim,PacketFramework);
ld_PF_Export_js(PacketFramework, fname="ProtocolConfig.json");
```

- Use e.g. a node.js program to build a web-interface to visualise data and to edit parameters.



- Allows to easily implement automated calibration procedures. The calib. routine may also be implemented using normal Scilab code.

```
function [sim, finished, outlist, userdata]=experiment(sim, ev, inlist, userdata)
    // Do the experiment; collect e.g. data to a shared memory
    AccGyro = inlist(1);
    [sim] = ld_printf(sim, 0, AccGyro, "Collecting data ... ", 6);
    // ...
    outlist=list(out);
endfunction

function [sim, outlist, userdata]=whileComputing(sim, ev, inlist, userdata)
    // While the computation is running this is called regularly
    [sim, out] = ld_const(sim, ev, 0);
    outlist=list(out);
endfunction

function [sim, outlist, userdata]=whileIdle(sim, ev, inlist, userdata)
    // When no calibration or computation is active
    AccGyro = inlist(1);
    [sim, out] = ld_const(sim, ev, 0);
    outlist=list(out);
endfunction

function [sim, CalibrationOk, userdata]=evaluation(sim, userdata)
    // Will run in a thread in background execution mode. Only one time step is executed here.
    // ...
    // Embedded e.g. a Scilab script that will be called once to perform the calibration
    [sim, Calibration] = ld_scilab2(sim, 0, in=CombinedData, comp_fn=scilab_comp_fn, include_scilab_fns=list(),
                                    scilab_path="BUILDIN_PATH");
    // ...
    // Tell ld_AutoExperiment that the calibration was successful
    [sim, oneint32] = ld_constvecInt32(sim, 0, vec=1)
    CalibrationOk = oneint32;
endfunction

[sim, finished, outlist] = ld_AutoExperiment(sim, ev, inlist=list(AccGyro, Ts), insizes=[6,1], outsizes=[1], ...
                                   intypes=[ORTD.DATATYPE_FLOAT,ORTD.DATATYPE_FLOAT], ...
                                   outtypes=[ORTD.DATATYPE_FLOAT], ...
                                   ThreadPrioStruct, experiment, whileComputing, evaluation, whileIdle);
```

Examples for advanced features like

- Online replacement of sub-controllers (`modules/nested`)
- State machines (`modules/nested`)
- Simulations running in threads (`modules/nested`)
- Shared memory, circular buffers, sending events to threads, ...
- Mathematical formula parsing (`modules/muparser`)
- Vector/matrix operations (`modules/basic_ldblocks`)
- Embeding Scilab-code (`modules/scilab`)
- Starting, I/O to external processes (`modules/ext_process`)
- Variable sampling rates (`modules/synchronisation`)
- Scicos to ORTD block wrapper (`modules/scicos_blocks`)
- UDP-communication & remote control interface (`modules/udp_communication`)
- ...

can be found within the directories `modules/*/demo`. Most of them are ready to run.