microblx Documentation

Markus Klotzbuecher et al

Apr 01, 2020

Contents:

CHAPTER 1

Getting started

1.1 Overview

Microblx is a very lightweight function block model and implementation.

- **block**: the basic building block of microblx. Is defined by filling in a `ubx_block_t` type and registering it with a microblx `ubx_node_t`. Blocks *have* configuration, ports and operations.

  Each block is part of a module and becomes available once the module is loaded in a node.

  There are two types of blocks: **computation blocks** ("cblocks", `BLOCK_TYPE_COMPUTATION`) encapsulate "functionality" such as drivers and controllers. **interaction blocks** ("iblocks", `BLOCK_TYPE_INTERACTION`) are used to implement communication or interaction between blocks. This manual focuses on how to build cblocks, since this is what most application builders need to do.

  – **configuration**: defines static properties of blocks, such as control parameters, device file names etc.

  – **port**: define which data flows in and out of blocks.

- **type**: types of data sent through ports or of configuration must be registered with microblx.
- **node**: an administrative entity which keeps track of blocks and types. Typically one per process is used, but there's no constraint whatsoever.
- **module**: a module contains one or more blocks or types that are registered/deregistered with a node when the module is loaded/unloaded.

1.2 Installation

1.2.1 Dependencies

- uthash (apt: `uthash-dev`)
- **luajit (>=v2.0.0) (apt: `luajit` and `libluajit-5.1-dev`) (not** strictly required, but recommended)

- `uutils` Lua utilities git
- `liblfds` lock free data structures (v6.1.1) git
- autotools etc. (apt: `automake`, `libtool`, `pkg-config`, `make`)

Only for microblx development:

- `lua-unit` (apt: `lua-unit`, git) (to run the tests)
- `cproto` (apt: `cproto`) to generate C prototype header file

1.2.2 Building and setting up

Using yocto

The best way to use microblx on an embedded system is by using the meta-microblx yocto layer. Please see the README in that repository for further steps.

Building manually

Building to run locally on a PC.

Before building microblx, liblfds611 needs to be built and installed. There is a set of patches in the microblx repository to clean up the packaging of liblfds. Follow the instructions below:

Clone the code:

```
$ git clone https://github.com/liblfds/liblfds6.1.1.git
$ git clone https://github.com/kmarkus/microblx.git
$ git clone https://github.com/kmarkus/uutils.git
```

First build lfds:

```
$ cd liblfds6.1.1
$ git am ../microblx/liblfds/*.patch
$ ./bootstrap
$ ./configure --prefix=/usr
$ make
$ sudo make install
```

Then install `uutils`:

```
$ cd ../uutils
$ sudo make install
```

Now build microblx:

```
$ cd ../microblx
$ ./bootstrap
$ ./configure
$ make
$ sudo make install
```

1.3 Quickstart

NOTE: the following assume microblx was installed in the default locations under `/usr/local/`. If you installed it in a different location you will need to adopt the path to the examples.

1.4 Run the random block example

This (silly) example creates a random number generator block. It's output is hexdump'ed (using the `hexdump` interaction block) and also logged using a `file_logger` block.

Before launching the composition, it is advisable to run the logging client to see potential errors:

```
$ ubx_log
```

and then in another terminal:

```
$ ubx_ilaunch -webif -c /usr/local/share/ubx/examples/systemmodels/trig_rnd_hexdump.
↪usc
```

Browse to http://localhost:8888

Explore:

1. clicking on the node graph will show the connections
2. clicking on blocks will show their interface
3. start the `file_log1` block to enable logging
4. start the `ptrig1` block to start the system.

CHAPTER 2

Developing microblx blocks

2.1 Overview

Generally, building a block entails the following:

1. declaring configuration: what is the static configuration of a block
2. declaring ports: what is the input/output of a block
3. declaring types: which data types are communicated
4. declaring block meta-data: provide further information about a block
5. declaring and implementing hook functions: how is the block initialized, started, run, stopped and cleaned up?
    1. reading configuration values: how to access configuration from inside the block
    2. reading and writing from ports: how to read and write from ports
6. declaring the block: how to put everything together
7. registration of blocks and types: make block prototypes and types known to the system

The following describes these steps in detail and is based on the (heavily) documented random number generator block (`std_blocks/random/`).

Note: Instead of manually implementing the above, a tool `ubx_genblock` is available which can generate blocks including interfaces from a simple description. See *Block code-generation*.

2.2 Declaring configuration

Configuration is described with a `NULL` terminated array of `ubx_config_t` types:

```
ubx_config_t rnd_config[] = {
    { .name="min_max_config", .type_name = "struct random_config" },
```

(continues on next page)

(continued from previous page)

```
    { NULL },
};
```

The above defines a single configuration called “min_max_config” of the type “struct random_config”.

Note:: custom types like `struct random_config` must be registered with the system. (see section “declaring types”)

To reduce boilerplate validation code in blocks, `min` and `max` attributes can be used to define permitted length of configuration values. For example:

```
ubx_config_t rnd_config[] = {
    { .name="min_max_config", .type_name = "struct random_config", .min=1, .max=1 },
    { NULL },
};
```

Would require that this block must be configured with exactly one `struct random_config` value. Checking will take place before the transition to *inactive* (i.e. before `init`).

In fewer cases, configuration takes place in state `inactive` and must be checked before the transition to `active`. That can be achieved by defining the config attribute `CONFIG_ATTR_CHECKLATE`.

Legal values of `min` and `max` are summarized below:

min	max	result
0	0	no checking (disabled)
0	1	optional config
1	1	mandatory config
0	CONFIG_LEN_MAX	zero to many
0	undefined	zero to many
N	M	must be between N and M

2.3 Declaring ports

Like configurations, ports are described with a `NULL` terminated array of ubx_config_t types:

```
ubx_port_t rnd_ports[] = {
    { .name="seed", .in_type_name="unsigned int" },
    { .name="rnd", .out_type_name="unsigned int" },
    { NULL },
};
```

Depending on whether an `in_type_name`, an `out_type_name` or both are defined, the port will be an in-, out- or a bidirectional port.

2.4 Declaring block meta-data

```
char rnd_meta[] =
    "{ doc='A random number generator function block',"
    "  realtime=true,"
    "}";
```

Additional meta-data can be defined as shown above. The following keys are supported so far:

- `doc:` short descriptive documentation of the block
- `realtime`: is the block real-time safe, i.e. there are is no memory allocation / deallocation and other non deterministic function calls in the `step` function.

2.5 Declaring/implementing block hook functions

The following block operations can be implemented to realize the blocks behavior. All are optional.

```
int rnd_init(ubx_block_t *b);
int rnd_start(ubx_block_t *b);
void rnd_stop(ubx_block_t *b);
void rnd_cleanup(ubx_block_t *b);
void rnd_step(ubx_block_t *b);
```

These functions can be called according to the microblx block life-cycle finite state machine:

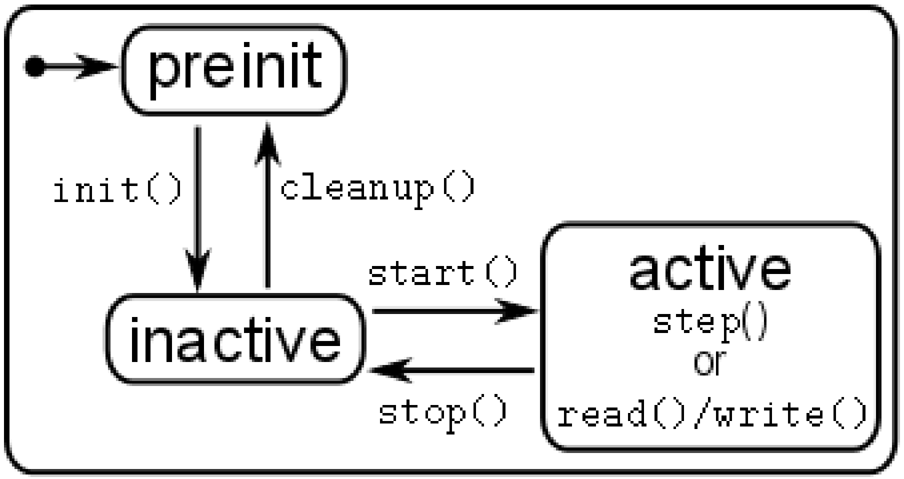

Fig. 1: Block lifecycle FSM

They are typically used for the following:

- `init`: initialize the block, allocate memory, drivers: check if the device is there and return non-zero if not.
- `start`: become operational, open device, last checks. Cache pointers to ports, read configuration.
- `step`: read from ports, compute, write to ports
- `stop`: stop/close device. (often not used).
- `cleanup`: free all memory, release all resources.

2.5.1 Storing block local state

As multiple instances of a block may exists, **NO** global variables may be used to store the state of a block. Instead, the `ubx_block_t` defines a `void* private_data` pointer which can be used to store local information. Allocate this in the `init` hook:

```
if ((b->private_data = calloc(1, sizeof(struct random_info)))==NULL) {
    ubx_err(b, "Failed to alloc memory");
    goto out_err;
}
```

and retrieve it in the other hooks:

```
struct block_info inf*;

inf = (struct random_info*) b->private_data;
```

2.5.2 Reading configuration values

The following example from the `random` block shows how to retrieve a struct configuration called `min_max_config`:

```
long int len;
struct random_config* rndconf;

/*...*/

if((len = ubx_config_get_data_ptr(b, "min_max_config", &rndconf)) < 0)
    goto err;

if(len==0)
    /* set a default or fail */
```

`ubx_config_get_data_ptr` returns the pointer to the actual data. `len` will be set to the array lenghth: 0 if unconfigured, >0 if configured and <0 in case of error.

For basic types there are several predefined and somewhat type safe convenience functions `cfg_getptr_*`. For example, to retrieve a scalar `uint32_t` and to use a default 47 if unconfigured:

```
long int len;
uint32_t *value;

if ((len = cfg_getptr_int(b, "myconfig", &value)) < 0)
    goto out_err;

value = (len > 0) ? *value : 47;
```

When to read configuration: init vs start?

It depends: if needed for initalization (e.g. a char array describing which device file to open), then read in `init`. If it's not needed in `init` (e.g. like the random min-max values in the random block example), then read it in start.

This choice affects reconfiguration: in the first case the block has to be reconfigured by a `stop`, `cleanup`, `init`, `start` sequence, while in the latter case only a `stop`, `start` sequence is necessary.

Reading from and writing to ports

The following helper macros are available to support

```
def_read_fun(read_uint, unsigned int)
def_write_fun(write_uint, unsigned int)
```

2.5.3 Declaring the block

The block aggregates all of the previous declarations into a single data-structure that can then be registered in a microblx module:

```
ubx_block_t random_comp = {
    .name = "random/random",
    .type = BLOCK_TYPE_COMPUTATION,
    .meta_data = rnd_meta,
    .configs = rnd_config,
    .ports = rnd_ports,

    /* ops */
    .init = rnd_init,
    .start = rnd_start,
    .step = rnd_step,
    .cleanup = rnd_cleanup,
};
```

2.5.4 Declaring types

All types used in configurations and ports must be declared and registered. This is necessary because microblx needs to know the size of the transported data. Moreover, it enables type reflection which is used by logging or the webinterface.

In the random block example, we used a `struct random_config`, that is defined in `types/random_config.h`:

```
struct random_config {
    int min;
    int max;
};
```

It can be declared as follows:

```
#include "types/random_config.h"
#include "types/random_config.h.hexarr"
ubx_type_t random_config_type = def_struct_type(struct random_config, &random_config_
↪h);
```

This fills in a `ubx_type_t` data structure called `random_config_type`, which stores information on types. Using this type declaration the `struct random_config` can then be registered with a node (see "Block and type registration" below).

What is this .hexarr file?

The file `types/random_config.h.hexarr` contains the contents of the file `types/random_config.h` converted to an array `const char random_config_h []` using the tool `tools/ubx_tocarr`. This char array is stored in the `ubx_type_t private_data` field (the third argument to the `def_struct_type` macro). At runtime, this type model is loaded into the luajit ffi, thereby enabling type reflection features such as logging or changing configuration values via the webinterface. The conversion from `.h` to `.hexarray` is done via a simple Makefile rule.

This feature is optional. If no type reflection is needed, don't include the .hexarr file and pass NULL as a third argument to def_struct_type.

2.5.5 Block and type registration

So far we have *declared* blocks and types. To make them known to the system, these need to be *registered* when the respective *module* is loaded in a microblx node. This is done in the module init function, which is called when a module is loaded:

```
1: static int rnd_module_init(ubx_node_info_t* ni)
2: {
3:         ubx_type_register(ni, &random_config_type);
4:         return ubx_block_register(ni, &random_comp);
5: }
6: UBX_MODULE_INIT(rnd_module_init)
```

Line 3 and 4 register the type and block respectively. Line 6 tells microblx that rnd_module_init is the module's init function.

Likewise, the module's cleanup function should deregister all types and blocks registered in init:

```
static void rnd_module_cleanup(ubx_node_info_t *ni)
{
    ubx_type_unregister(ni, "struct random_config");
    ubx_block_unregister(ni, "random/random");
}
UBX_MODULE_CLEANUP(rnd_module_cleanup)
```

2.5.6 Using real-time logging

Microblx provides logging infrastructure with loglevels similar to the Linux Kernel. Loglevel can be set on the (global) node level (e.g. by passing it -loglevel N to ubx_launch or be overridden on a per block basis. To do the latter, a block must define and configure a loglevel config of type int. If it is left unconfigured, again the node loglevel will be used.

The following loglevels are supported:

- UBX_LOGLEVEL_EMERG (0) (system unusable)
- UBX_LOGLEVEL_ALERT (1) (immediate action required)
- UBX_LOGLEVEL_CRIT (2) (critical)
- UBX_LOGLEVEL_ERROR (3) (error)
- UBX_LOGLEVEL_WARN (4) (warning conditions)
- UBX_LOGLEVEL_NOTICE (5) (normal but significant)
- UBX_LOGLEVEL_INFO (6) (info message)
- UBX_LOGLEVEL_DEBUG (7) (debug messages)

The following macros are available for logging from within blocks:

```
ubx_emerg(b, fmt, ...)
ubx_alert(b, fmt, ...)
ubx_crit(b, fmt, ...)
```

(continues on next page)

(continued from previous page)

```
ubx_err(b, fmt, ...)
ubx_warn(b, fmt, ...)
ubx_notice(b, fmt, ...)
ubx_info(b, fmt, ...)
ubx_debug(b, fmt, ...)
```

Note that `ubx_debug` will only be logged if `UBX_DEBUG` is defined in the respective block and otherwise compiled out without any overhead.

To view the log messages, you need to run the `ubx_log` tool in a separate window.

Important: The maximum total log message length (including is by default set to 80 by default), so make sure to keep log message short and sweet (or increase the lenghth for your build).

Note that the old (non-rt) macros `ERR`, `ERR2`, `MSG` and `DBG` are deprecated and shall not be used anymore.

Outside of the block context, (e.g. in `module_init` or `module_cleanup`, you can log with the lowlevel function

```
ubx_log(int level,
        ubx_node_info_t *ni,
        const char* src,
        const char* fmt, ...)

/* for example */
ubx_log(UBX_LOGLEVEL_ERROR, ni, __FUNCTION__, "error %u", x);
```

e.g.

The ubx core uses the same logger, but mechanism, but uses the `log_info` resp `logf_info` variants. See `libubx/ubx.c` for examples.

2.5.7 SPDX License Identifier

Microblx uses a macro to define module licenses in a form that is both machine readable and available at runtime:

```
UBX_MODULE_LICENSE_SPDX(MPL-2.0)
```

To dual-license a block, write:

```
UBX_MODULE_LICENSE_SPDX(MPL-2.0 BSD-3-Clause)
```

Is is strongly recommended to use this macro. The list of licenses can be found on http://spdx.org/licenses

2.5.8 Block code-generation

The `ubx_genblock` tool generates a microblx block including a Makefile. After this, only the hook functions need to be implemented in the `.c` file:

Example: generate stubs for a `myblock` block (see `examples/block_model_example.lua` for the block generator model).

```
$ ubx_genblock -d myblock -c /usr/local/share/ubx/examples/blockmodels/block_model_
↪example.lua
    generating myblock/bootstrap
    generating myblock/configure.ac
```

(continues on next page)

(continued from previous page)

```
    generating myblock/Makefile.am
    generating myblock/myblock.h
    generating myblock/myblock.c
    generating myblock/myblock.usc
    generating myblock/types/vector.h
    generating myblock/types/robot_data.h
```

Run `ubx_genblock -h` for full options.

The following files are generated:

- `bootstrap` autoconf bootstrap script
- `configure.ac` autoconf input file
- `Makefile.am` automake input file
- `myblock.h` block interface and module registration code (don't edit)
- `myblock.c` module body (edit and implement functions)
- `myblock.usc` simple microblx system composition file, see below (can be extended)
- `types/vector.h` sample type (edit and fill in struct body)
- `robot_data.h` sample type (edit and fill in struct body)

If the command is run again, only the `.c` file will NOT be regenerated. This can be overridden using the `-force` option.

2.5.9 Compile the block

```
$ cd myblock/
$ ./bootstrap
$ ./configure
$ make
$ make install
```

2.5.10 Launch block using ubx_launch

```
$ ubx_ilaunch -webif -c myblock.usc
```

Run `ubx_launch -h` for full options.

Browse to http://localhost:8888

2.6 Tips and Tricks

2.6.1 Using C++

See `std_blocks/cppdemo`. If the designated initializer (the struct initalization used in this manual) are used, the block must be compiled with `clang`, because g++ does not support designated initializers (yet).

2.6.2 Avoiding Lua scripting

It is possible to avoid the Lua scripting layer entirely. A small example can be found in `examples/c-only.c`. See also the tutorial for a more complete example.

2.6.3 Speeding up port writing

To speed up port writing, the pointers to ports can be cached in the block info structure. The `ubx_genblock` script automatically takes care of this.

2.6.4 What the difference between block types and instances?

First: to create a block instance, it is cloned from an existing block and the `block->prototype` char pointer set to a newly allocated string holding the protoblocks name.

There's very little difference between prototypes and instances:

- a block type's `prototype` (char) ptr is `NULL`, while an instance's points to a (copy) of the prototype's name.
- Only block instances can be deregistered and freed (`ubx_block_rm`), prototypes must be deregistered (and freed if necessary) by the module's cleanup function.

2.6.5 Module visibility

The default Makefile defines `-fvisibility=hidden`, so there's no need to prepend functions and global variables with `static`

CHAPTER 3

Composing microblx systems

Building a microblx application typically involves instantiating blocks, configuring and interconnecting these and finally starting these up. The recommended way to do this is by specifying the system using the microblx composition DSL.

3.1 Microblx System Composition DSL (usc files)

`usc` are declarative descriptions of microblx systems that can be validated and instantiated using the `ubx_launch` tool. A usc model describes one microblx **system**, as illustrated by the following minimal example:

```
local bd = require("blockdiagram")

return bd.system
{
   -- import microblx modules
   imports = {
      "stdtypes", "ptrig", "lfds_cyclic", "myblocks",
   },

   -- describe which blocks to instantiate
   blocks = {
      { name="x1", type="myblocks/x" },
      { name="y1", type="myblocks/y" },
      { name="ptrig1", type="std_triggers/ptrig" },
      ...
   },

   -- connect blocks
   connections = {
      { src="x1.out", tgt="y1.in",   },
      { src="y1.out", tgt="x1.in",   },
   },
```

(continues on next page)

(continued from previous page)

```
   -- configure blocks
   configurations = {
      { name="x1", config = { cfg1="foo", cfg2=33.4 } },
      { name="y1", config = { cfgA={ p=1,z=22.3 }, cfg2=33.4 } },

      -- configure a trigger
      { name="trig1", config = { period = {sec=0, usec=100000 },
                                  sched_policy="SCHED_OTHER",
                                  sched_priority=0,
                                  trig_blocks={
                                     -- the #<blockname> directive will
                                     -- be resolved to an actual
                                     -- reference to the respective
                                     -- block once instantiated
                                     { b="#x1", num_steps=1, measure=0 },
                                     { b="#y1", num_steps=1, measure=0 } } } }
   },
}
```

3.1.1 Launching

usc files like the above example can be launched using `ubx_launch` tool. Run with `-h` for further information on the options. The following simple example:

```
$ ubx_launch -webif -c examples/trig_rnd_hexdump.usc
```

this will launch the given system composition and in addition create and configure a web server block to allow the system to be introspected via a browser.

Unless the `-nostart` option is provided, all blocks will be initialized, configured and started. `ubx_launch` takes care to do this is safe way by starting up active blocks and triggers after all other blocks (In earlier versions, there was `start` directive to list these types of blocks, today this attribute is defined active and trigger blocks)

3.1.2 Node configs

Node configs allow to assign the same configuration to multiple blocks. This is useful for global configuration values which are the same for all blocks.

For this, a new top level section `node_configurations` is introduced, which allows to defined named configurations.

```
node_configurations = {
    global_rnd_conf = {
        type = "struct random_config",
        config = { min=333, max=999 },
    }
}
```

These configurations can then be assigned to multiple blocks:

```
{ name="b1", config = { min_max_config = "&global_rnd_conf"} },
{ name="b2", config = { min_max_config = "&global_rnd_conf"} },
```

Please refer to `examples/systemmodels/node_config_demo.usc` for a full example.

3.2 Alternatives

Although using `usc` model is the preferred approach, there are others way to launch a microblx application:

1. by writing a Lua called "deployment script" (e.g. see `examples/trig_rnd_to_hexdump.lua`). This is not recommended under normal circumstances, but can be useful in specific cases such as for building dedicated test tools.
2. by assembling everything in C/C++. Possible, but somewhat painful to do by hand. This would be better solved by introducing a usc-compiler tool. Please ask on the mailing list.

CHAPTER 4

Tutorial: close loop control of a robotic platform

4.1 Goal

This is a walk-trough that shows how to:

1. write, compile and install two blocks:
   - the plant (a two DoF robot that accepts velocity as input, and give the relative position), and
   - The controller, that given as a property the set-point and the gain, computes the desired velocity to be set to the robot.
2. instantiate the blocks via the *ubx_launch* tool, and
3. instantiate the blocks with a c-written program.

All the files can be found in the examples/platform folder.

4.2 Introductory steps

First of all, we need need to define the interface of and between our two components.

The plant and controller has two ports, that exchange position and velocity, each of dimension two, and some properties (initial position and velocity limits for the plant, gain and setpoint for the controller). These properties are described it two lua files:

The plant, "platform_2dof.lua"

```
return block
{
      name="platform_2dof",
      license="MIT",
      meta_data="",
      port_cache=true,
```

(continues on next page)

(continued from previous page)

```
      configurations= {
     { name="joint_velocity_limits", type_name="double", len=2 },
     { name="initial_position", type_name="double", len=2 },
      },

      ports = {
     { name="pos", out_type_name="double", out_data_len=2, doc="measured position [m]
↪" },
     { name="desired_vel", in_type_name="double", in_data_len=2, doc="desired␣
↪velocity [m/s]" }
      },

      operations = { start=true, stop=true, step=true }
}
```

The controller, “platform_2dof_control.lua”

```
return block
{
      name="platform_2dof_control",
      license="MIT",
      meta_data="",
      port_cache=true,

      configurations= {
     { name="gain", type_name="double" },
     { name="target_pos", type_name="double" ,len=2 },
      },

      ports = {
     { name="measured_pos", in_type_name="double", in_data_len=2, doc="measured␣
↪position [m]" },
     { name="commanded_vel", out_type_name="double", out_data_len=2, doc="desired␣
↪velocity [m/s]" },
      },

      operations = { start=true, stop=true, step=true }
}
```

Let us have these file in a folder (*e.g.* microblx_tutorial). from these file we can generate the two blocks using the the following bash commands

```
cd microblx_tutorial
ubx_genblock -c platform_2dof.lua -d platform_2dof
ubx_genblock -c platform_2dof_control.lua -d platform_2dof_control
```

Each command generates a folder with the name specified after the -d with five files. For the plant, we will have:

- configure.ac
- Makefile.am
- platform_2dof.h
- platform_2dof.c
- platform_2dof.usc

The only files we will modify are **platform_2dof/platform_2dof.c** and **platform_2dof_control/platform_2dof_control.c**. ## Code of the blocks The file that is auto-generated gives already some hints on how to approach the programming.

```
#include "platform_2dof.h"

/* define a structure for holding the block local state. By assigning an
 * instance of this struct to the block private_data pointer (see init), this
 * information becomes accessible within the hook functions.
 */
struct platform_2dof_info
{
    /* add custom block local data here */

    /* this is to have fast access to ports for reading and writing, without
     * needing a hash table lookup */
    struct platform_2dof_port_cache ports;
};

/* init */
int platform_2dof_init(ubx_block_t *b)
{
    int ret = -1;
    struct platform_2dof_info *inf;

    /* allocate memory for the block local state */
    if ((inf = (struct platform_2dof_info*)calloc(1, sizeof(struct platform_2dof_
↪info)))==NULL) {
        ERR("platform_2dof: failed to alloc memory");
        ret=EOUTOFMEM;
        goto out;
    }
    b->private_data=inf;
    update_port_cache(b, &inf->ports);
    ret=0;
out:
    return ret;
}

/* start */
int platform_2dof_start(ubx_block_t *b)
{
    /* struct platform_2dof_info *inf = (struct platform_2dof_info*) b->private_data;␣
↪*/
  ubx_info(b, "platform_2dof start");
    int ret = 0;
    return ret;
}

/* stop */
void platform_2dof_stop(ubx_block_t *b)
{
    /* struct platform_2dof_info *inf = (struct platform_2dof_info*) b->private_data;␣
↪*/
  ubx_info(b, "platform_2dof stop");
}

/* cleanup */
```

(continues on next page)

(continued from previous page)

```
void platform_2dof_cleanup(ubx_block_t *b)
{
    /* struct platform_2dof_info *inf = (struct platform_2dof_info*) b->private_data;␣
↪*/
  ubx_info(b, "platform_2dof cleanup");
    free(b->private_data);
}

/* step */
void platform_2dof_step(ubx_block_t *b)
{
    /* struct platform_2dof_info *inf = (struct platform_2dof_info*) b->private_data;␣
↪*/
    ubx_info(b, "platform_2dof step");
}
```

We will need then to insert the code indicated by the comments. ### Step 1: insert the state of the robot, and other helper functions At the beginning of the file we insert the following code, to save the state of the robot:

```
struct robot_state {
  double pos[2];
  double vel[2];
  double vel_limit[2];
};
double sign(double x)
{
  if(x > 0) return 1;
  if(x < 0) return -1;
  return 0;
}

struct platform_2dof_info
{
  /* add custom block local data here */
  struct robot_state r_state;
  struct ubx_timespec last_time;
  /* this is to have fast access to ports for reading and writing, without
         * needing a hash table lookup */
  struct platform_2dof_port_cache ports;
};
```

The “last_time” variable is needed to compute the time passed between two calls of
the “platform_2dof_step” function.

4.2.1 Step 2: Initialization and Start functions

This is the function called when the block is initialized; it allocates memory for the info structure, caches the ports, and initializes the state given the configuration values (these values are given in the “.usc” or main application file).

```
int platform_2dof_init(ubx_block_t *b)
{
  int ret = -1;
  long int len;
```

(continues on next page)

(continued from previous page)

```
  struct platform_2dof_info *inf;
  double *pos_vec;

  /* allocate memory for the block local state */
  if ((inf = (struct platform_2dof_info*)calloc(1, sizeof(struct platform_2dof_
↪info)))==NULL) {
      ubx_err(b,"platform_2dof: failed to alloc memory");
      ret=EOUTOFMEM;
      goto out;
    }
  b->private_data=inf;
  update_port_cache(b, &inf->ports);

  //read configuration - initial position
  if ((len = ubx_config_get_data_ptr(b, "initial_position",(void **)&pos_vec)) < 0) {
      ubx_err(b,"platform_2dof: failed to load initial_position");
      goto out;
    }
  inf->r_state.pos[0]=pos_vec[0];
  inf->r_state.pos[1]=pos_vec[1];
  if ((len = ubx_config_get_data_ptr(b, "joint_velocity_limits",(void **)&pos_vec)) <␣
↪0) {
      ubx_err(b,"platform_2dof: failed to load joint_velocity_limits");
      goto out;
    }
  //read configuration - max velocity
  inf->r_state.vel_limit[0]=pos_vec[0];
  inf->r_state.vel_limit[1]=pos_vec[1];

  inf->r_state.vel[0]=inf->r_state.vel[1]=0.0;
  ret=0;
out:
  return ret;
}
```

The function `ubx_config_get_data_ptr(ubx_block_t *b, const char *name, void **ptr)` returns in the pointer passed by reference the address of the required configuration. In this case the function will return "2", success, the length of the data or "-1", failure.

For the start function, we only need to initialize the internal timer

```
int platform_2dof_start(ubx_block_t *b)
{
  struct platform_2dof_info *inf = (struct platform_2dof_info*) b->private_data;
  ubx_info(b, "platform_2dof start");
  ubx_gettime(&(inf->last_time));
  int ret = 0;
  return ret;
}
```

4.2.2 Step 3: Step function

In the step function, we compute the time since last iteration, read the commanded velocity, integrate to position, and then write position.

```
void platform_2dof_step(ubx_block_t *b)
{
  int32_t ret;
  double velocity[2];
  struct ubx_timespec current_time, difference;
  struct platform_2dof_info *inf = (struct platform_2dof_info*) b->private_data;
  //compute time from last call
  ubx_gettime(&current_time);
  ubx_ts_sub(&current_time,&(inf->last_time),&difference);
  inf->last_time=current_time;
  double time_passed= ubx_ts_to_double(&difference);

  //read velocity from port
  ret = read_desired_vel_2(inf->ports.desired_vel, &velocity);
  if (ret<=0){ //nodata
      velocity[0]=velocity[1]=0.0;
      ubx_err(b,"no velocity data");
    }

  for (int i=0;i<2;i++){// saturate and integrate velocity
      velocity[i]=fabs(velocity[i])> inf->r_state.vel_limit[i]?␣
↪sign(velocity[i])*(inf->r_state.vel_limit[i]):velocity[i];
      inf->r_state.pos[i]+=velocity[i]*time_passed;}
  //write position in the port
  write_pos_2(inf->ports.pos,&(inf->r_state.pos));

}
```

In case there is no value in the port ,an error is signaled, and nominal velocity is set to zero. This will always happens in the first interaction, since the controller did step yet, thus no velocity command is available.

4.2.3 Step 4: Stop and clean-up functions

These functions are OK as they are generated, since the only thing we want to take care of is that memory is freed.

4.2.4 Final listings of the block

The plant is, *mutatis mutandis*, built following the same rationale, and will be not detailed here. the final code of the plant and the controller can be retrieved here **TODO add link to the code**

4.2.5 building of the blocks

In order to build and install the blocks, you must execute the following bash commands in each of the two directories:

```
 autoreconf --install
./configure
make
sudo make install
```

see also the quickstart about these. ## Deployment via the usc (microblx system composition) file. The `ubx_genblock` commands generates two sample files to run independently each block. We want to run and compose them together, and save the results in a logger file. The composition file **platform_2dof_and_control.usc** is quite self explanatory: It indicates

- which libraries are imported,
- which block (name, type) are created,
- the configuration values of properties.

The code is the following.

```
plat_report_conf = [[{
{ blockname='plat1', portname="pos"},
{ blockname='control1', portname="commanded_vel"}
}]] -- this is a multiline string


return bd.system
{
   imports = {
      "stdtypes",
      "stattypes",
      "ptrig",
      "lfds_cyclic",
      "logger",
      "platform_2dof",
      "platform_2dof_control",
   },

   blocks = {
      { name="plat1", type="platform_2dof" },
      { name="control1", type="platform_2dof_control" },
      { name="logger_time", type="logging/file_logger" },
      { name="fifo_pos", type="lfds_buffers/cyclic" },
      { name="fifo_vel", type="lfds_buffers/cyclic" },
      { name="ptrig1", type="std_triggers/ptrig" },
   },
   connections = {
     { src="plat1.pos", tgt="fifo_pos" },
     { src="fifo_pos",tgt="control1.measured_pos" },
     { src="control1.commanded_vel",tgt="fifo_vel" },
     { src="fifo_vel",  tgt="plat1.desired_vel" },

   },
   configurations = {
      { name="plat1", config = {  initial_position={1.1,1}, joint_velocity_limits={0.
↪5,0.5} } },
      { name="control1", config = {  gain=0.1, target_pos={4.5,4.5} } },
      { name="fifo_pos", config = { type_name="double", data_len=2, buffer_len=1 } },
      { name="fifo_vel", config = { type_name="double", data_len=2, buffer_len=1 } },
      { name="logger_time", config = { filename="/tmp/platform_time.log",
                                        separator=",",
                                        report_conf = timer_report_conf, } },
      { name="ptrig1", config = { period = {sec=1, usec=0 },
                                  sched_policy="SCHED_OTHER",
                                  sched_priority=0,
                                  trig_blocks={ { b="#plat1", num_steps=1, measure=1␣
↪},
                        { b="#control1", num_steps=1, measure=1 },
                        { b="#logger_time", num_steps=1, measure=0 }  } } }
   }
```

(continues on next page)

(continued from previous page)

```
}
```

It is worth noticing that configuration types can be arrays (*e.g.* `target_pos`), strings (`file_name` and `report_conf`) and structures (`period`) and vector of structures (`trig_blocks`). Types for each property should be checked in source files. Alternatively, the web server can provide insight of the types.

The file is launched with the command

```
ubx_ilaunch -c platform_2dof_and_control.usc
```

or

```
ubx_ilaunch -webif -c platform_2dof_and_control.usc
```

to enable the *web interface* at localhost:8888 . In order to visualize the data saved by the logger in the *:raw-latex:'tmp'* folder, consider kst or any other program that can visualize a comma-separated-value file. ###Some considerations about the fifos

First of all, consider that each (iblock) fifo can be connected with multiple input and multiple output ports. Consider also, that if multiple out are connected, if one read one data, that data will be consumed and not available for a second port.

The more common use-case is that each inport has is own fifo. If the data that is produced by one outport is needed to be read by two oe more inports, a fifo per inport is connected to the the outport. **If you use the DSL, this is automatically done, so you do not have to worry to explicitly instantiate the iblocks. This also happens when adding ports to the logger**.

TODO insert picture

##Deployment via c program This example is an extension of the *"c-only.c"*. It will be clear that using the above method is far easier, but in case for some reason we want to eliminate the dependency from *lua*, this example show that is possible, even if a little burdensome.

First of all, we need to make a package to enable the building. This can be done looking at the structure of the rest of packages.

we will create a folder called *platform_launch* that contains the following files:

- *main.c*
- *Makefile.am*
- *configure.am* ##setup the build system starting from the build part: *configure.am*:

```
m4_define([package_version_major],[0])
m4_define([package_version_minor],[0])
m4_define([package_version_micro],[0])

AC_INIT([platform_launch], [package_version_major.package_version_minor.package_
↪version_micro])
AM_INIT_AUTOMAKE([foreign -Wall])

# compilers
AC_PROG_CC

PKG_PROG_PKG_CONFIG
PKG_INSTALLDIR

AC_CONFIG_HEADERS([config.h])
```

(continues on next page)

(continued from previous page)

```
AC_CONFIG_MACRO_DIR([m4])

# Check if the `install` program is present
AC_PROG_INSTALL

m4_ifdef([AM_PROG_AR], [AM_PROG_AR])
LT_INIT(disable-static)

PKG_CHECK_MODULES(UBX, ubx0 >= 0.6.0)

PKG_CHECK_VAR([UBX_MODDIR], [ubx0], [UBX_MODDIR])
  AC_MSG_CHECKING([ubx module directory])
  AS_IF([test "x$UBX_MODDIR" = "x"], [
  AC_MSG_FAILURE([Unable to identify ubx module path.])
])
AC_MSG_RESULT([$UBX_MODDIR])

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Makefile.am

```
ubxmoddir = ${UBX_MODDIR}

ACLOCAL_AMFLAGS= -I m4
ubxmod_PROGRAMS = platform_main
platform_main_SOURCES = $(top_srcdir)/libubx/ubx.h  main.c
platform_main_CFLAGS = -I${top_srcdir}/libubx  @UBX_CFLAGS@
platform_main_LDFLAGS = -module -avoid-version -shared -export-dynamic  @UBX_LIBS@ -
↪ldl
```

Here, we specify that the name of the executable is *platform_main* It might be possible that, if some custom types are used in the configuration, but are not installed, they must be added to the *CFLAGS*:

```
platform_main_CFLAGS = -I${top_srcdir}/libubx -I path/to/other/headers  @UBX_CFLAGS@
```

In order to compile, we will use the same commands as before (we do not need to install).

```
autoreconf --install
./configure
make
```

4.3 The program

The main follows the same structure of the .usc file. ### Logging Microblx uses realtime safe functions for logging. For logging from the scope of a block the functions `ubx_info`, `ubx_info`, *etc* are used. In the main we have to use the functions, `ubx_log`, *e.g.*

```
ubx_log(UBX_LOGLEVEL_ERR, &ni,__FUNCTION__,  "failed to init control1");
```

More info on logging can be found in the Using real-time logging section. ### Libraries It start with some include (struct that are needed in configuration) and loading the libraries

```
#include <ubx.h>

#define WEBIF_PORT "8810"
#define DOUBLE_STR "double"
#include "ptrig_config.h"
#include "ptrig_period.h"
#define LEN_VEC(a) (sizeof(a)/sizeof(a[0]))
int main()
{
  int len, ret=EXIT_FAILURE;
  ubx_node_info_t ni;
  ubx_block_t *plat1, *control1, *logger1, *ptrig1, *webif, *fifo_vel, *fifo_pos;
  ubx_data_t *d;

  /* initalize the node */
  ubx_node_init(&ni, "platform_and_control");
const char *modules[8];
modules[0]= "/usr/local/lib/ubx/0.6/stdtypes.so";
modules[1]= "/usr/local/lib/ubx/0.6/stattypes.so";
modules[2]= "/usr/local/lib/ubx/0.6/ptrig.so";
modules[3]= "/usr/local/lib/ubx/0.6/platform_2dof.so";
modules[4]= "/usr/local/lib/ubx/0.6/platform_2dof_control.so";
modules[5]= "/usr/local/lib/ubx/0.6/webif.so";
modules[6]= "/usr/local/lib/ubx/0.6/logger.so";
modules[7]= "/usr/local/lib/ubx/0.6/lfds_cyclic.so";
  /* load modules */
for (int i=0; i<LEN_VEC(modules);i++)
  if(ubx_module_load(&ni, modules[i]) != 0){
      printf("fail to load %s",modules[i]);
      goto out;
    }
```

4.3.1 Block instantiation

Then, we instantiate blocks (code for only one, for sake of brevity):

```
if((plat1 = ubx_block_create(&ni, "platform_2dof", "plat1"))==NULL){
      printf("fail to create platform_2dof");
      goto out;
    }
```

4.3.2 Property configuration

Now we have the more tedious part, that is the configuration. it is necessary to allocate memory with the function `ubx_data_resize`, that takes as argument the data pointer, and the new length. #### String property:

```
d = ubx_config_get_data(webif, "port");
len = strlen(WEBIF_PORT)+1;
/* resize the char array as necessary and copy the port string */
ubx_data_resize(d, len);
strncpy((char *)d->data, WEBIF_PORT, len);
```

Here the sting is declared with a `#define`, it can be written directly, or with a variable, *e.g.* `char filename[]="/tmp/platform_time.log";`

Double property:

```
d = ubx_config_get_data(control1, "gain");
ubx_data_resize(d, 1);
*((double*)d->data)=0.12;
```

In this case, memory allocation is done for a scalar (i.e. size 1) . The second line says: consider `d->data` as a pointer to double, and assign to the pointed memory area the value `0.12`. #### Fixed size array of double:

```
d = ubx_config_get_data(control1, "target_pos");
ubx_data_resize(d, 2);
((double*)d->data)[0]=4.5;
((double*)d->data)[1]=4.52;
```

Same as before, but being a vector, of two elements, the memory allocation is changed accordingly, and data writings needs the index. #### Structure property: To assign the values to the structure, one option is to make/allocate a local instance of the structure, and then copy it.

```
struct ptrig_period p;
p.sec=1;
p.usec=14;
d=ubx_config_get_data(ptrig1, "period");
ubx_data_resize(d, 1);
*((struct ptrig_period*)d->data)=p;
```

In alternative, we can directly work on the fields of the structure

```
d=ubx_config_get_data(ptrig1, "period");
ubx_data_resize(d, 1);
((struct ptrig_period*)d->data)->sec=1;
((struct ptrig_period*)d->data)->usec=14;
```

Array of structures:

It combines what we saw for arrays and structures. In the case of the trigger block, we have to configure the order of blocks,

```
d=ubx_config_get_data(ptrig1, "trig_blocks");
len= 3;
ubx_data_resize(d, len);
printf("data size trig blocks: %li\n",d->type->size);
((struct ptrig_config*)d->data)[0].b = plat1;//ubx_block_get(&ni, "plat1")
((struct ptrig_config*)d->data)[0].num_steps = 1;
((struct ptrig_config*)d->data)[0].measure = 0;
((struct ptrig_config*)d->data)[1].b = control1;
((struct ptrig_config*)d->data)[1].num_steps = 1;
((struct ptrig_config*)d->data)[1].measure = 0;
((struct ptrig_config*)d->data)[2].b = logger1;
((struct ptrig_config*)d->data)[2].num_steps = 1;
((struct ptrig_config*)d->data)[2].measure = 0;
```

4.3.3 Port connection

To connect we have first to retrieve the ports, and then connect to an a **iblock**, the fifos. In the following, we have two inputs and two output ports, that are connected via two fifos:

```
ubx_port_t* plat1_pos=ubx_port_get(plat1,"pos");
ubx_port_t* control1_measured_pos=ubx_port_get(control1,"measured_pos");
ubx_port_t* control1_commanded_vel=ubx_port_get(control1,"commanded_vel");
ubx_port_t* plat1_desired_vel=ubx_port_get(plat1,"desired_vel");

ubx_port_connect_out(plat1_pos,fifo_pos);
ubx_port_connect_in(control1_measured_pos,fifo_pos);
ubx_port_connect_out(control1_commanded_vel,fifo_vel);
ubx_port_connect_in(plat1_desired_vel,fifo_vel);
```

4.3.4 Init and Start the blocks

Lastly, we need to init and start all the blocks. For example, for the `control1` iblock:

```
if(ubx_block_init(control1) != 0) {
        ubx_log(UBX_LOGLEVEL_ERR, &ni,__FUNCTION__,  "failed to init control1");
        goto out;
}
if(ubx_block_start(control1) != 0) {
    ubx_log(UBX_LOGLEVEL_ERR, &ni,__FUNCTION__,  "failed to start control1");
    goto out;
}
```

The same applies for all the block.

Once all the block are running, the `trigger` block will call all the blocks in the given order, so long the main does not terminate. To prevent the main process to terminate, we can insert either a blocking call to terminal input:

```
getchar();
```

or using the *signal.h* library, wait until *CTRL+C* is pressed:

```
sigset_t set;
int sig;

sigemptyset(&set);
sigaddset(&set, SIGINT);
pthread_sigmask(SIG_BLOCK, &set, NULL);
sigwait(&set, &sig);
```

Note that we have to link against pthread library, so the *Makefile.am* has to be modified accordingly:

```
...
platform_main_LDFLAGS = -module -avoid-version -shared -export-dynamic  @UBX_LIBS@ -
↪ldl -lpthread
```

CHAPTER 5

Frequently asked questions

5.1 fix "error object is not a string"

Note that this has been fixed in commit `be63f6408bd4d`.

This is most of the time happens when the `strict` module being loaded (also indirectly, e.g. via ubx.lua) in a luablock. It is caused by the C code looking up a non-existing global hook function. Solution: either define all hooks or disable the strict module for the luablock.

5.2 `blockXY`.so or `liblfds611.so.0`: cannot open shared object file: No such file or directory

There seems to be a bug in some versions of libtool which leads to the ld cache not being updated. You can manually fix this by running

```
$ sudo ldconfig
```

Often this means that the location of the shared object file is not in the library search path. If you installed to a non-standard location, try adding it to `LD_LIBRARY_PATH`, e.g.

```
$ export LD_LIBRARY_PATH=/usr/local/lib/
```

It would be better to install stuff in a standard location such as `/usr/local/`.

5.3 Real-time priorities

To run with realtime priorities, give the luajit binary `cap_sys_nice` capabilities, e.g:

```
$ sudo setcap cap_sys_nice+ep /usr/local/bin/luajit-2.0.2
```

5.4 My script immedately crashes/finishes

This can have several reasons:

- You forgot the `-i` option to `luajit`: in that case the script is executed and once completed will immedately exit. The system will be shut down / cleaned up rather rougly.
- You ran the wrong Lua executable (e.g. a standard Lua instead of `luajit`).

Typically the best way to debug crashes is using gdb and the core dump file:

```
# enable core dumps
$ ulimit -c unlimited
$ gdb luajit
...
(gdb) core-file core
...
(gdb) bt
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search