microblx Documentation

Markus Klotzbuecher et al

Jan 03, 2020

Contents:

CHAPTER 1

Installing

1.1 Building from source

1.1.1 Dependencies

Make sure to install the following dependencies

- uthash (apt: `uthash-dev`)
- autotools etc. (apt: `automake`, `libtool`, `pkg-config`, `make`)
- luajit (>=v2.0.0) (apt: `luajit` and `libluajit-5.1-dev`)
- lfs: lua-filesystem (apt: `lua-filesystem`)

The following must be installed from source (see instructions below):

- `uutils` Lua utilities uutils git
- `liblfds` lock free data structures (v6.1.1) liblfds6.1.1 git

Optionally, to run the tests:

- `lua-unit` (apt: `lua-unit`, git) (to run the tests)

1.1.2 Building

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-6.1.1*:

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

Then install *uutils*:

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

Now build *microblx*:

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

Note: it might be necessary to build with

```
$ make CXXFLAGS="-std=c++11"
```

1.2 Using yocto

If you are developing for an embedded system, the recommended way is use the meta-microblx yocto layer. Please see the README in that repository for further instructions.

CHAPTER 2

Getting started

2.1 Microblx in a nutshell

Microblx is a lightweight framework to build function block based systems. It is designed around a canonical component model with *ports* for data exchange, *configs* for configuration and a *state machine* for the "block" life cycle. Trigger blocks orchestrate the execution of the components functionality.

Building a microblx application typically involves two steps:

2.1.1 Implement the required blocks

define the **block API**

- *configs*: what is (statically) configurable
- *ports*: which data flows in and out of the block

and implement the required block **hooks**

- For example, `init` is typically used to initialize, allocate memory and/or validate configuration.
- the `step` hook implements the "main" functionality and is executed when the block is **triggered**.
- `cleanup` is to "undo" `init` (i.e. free resources etc.)

Take a look at a simple threshold checking demo block.

Note: You can examine a block interface using `ubx-modinfo`, e.g. run `$ ubx-modinfo show threshold`.

2.1.2 Define the application using a usc file

This involves specifying

- which *block instances* to create

- the *configuration* of for each block
- the *connections* to create between ports
- the *triggering* of blocks (i.e. the schedule of when to trigger the step functions of each block)

A small, ready to run usc demo using the *threshold* block is available here

usc applications can be launched using the *ubx-launch* tool, as shown in the following example.

2.2 Run the threshold example

In this small example, a ramp feeds a sine generator whose output is checked whether it exceeds a threshold. The threshold block outputs the current state (1: *above* or 0: *below*) as well as *events* upon passing the threshold. The events are connected to a mqueue block, where they can be logged using the `ubx-mq` tool. The actual composition looks as follows

```
/------\    /-----\    /-----\    /----\
| ramp |--->| sin |--->|thres|--->| mq |
\------/    \-----/    \-----/    \----/
   ^           ^          ^
   .           . #2       .
#1 .           .          .
   .        /------\      . #3
   .........| trig |.......
            \------/

---> depicts data flow
...> means "triggers"
```

Before launching, start a ubx logger client in a separate terminal:

```
$ ubx-log
waiting for rtlog.logshm to appear
```

Note: The following assumes microblx was installed in the default locations under `/usr/local/`. If you installed it in a different location you will need to adapt the path.

Then in a new terminal:

```
$ ubx-launch -loglevel 7 -c /usr/local/share/ubx/examples/usc/threshold.usc
core_prefix: /usr/local
prefixes:    /usr, /usr/local
```

We increase the loglevel to 7 (DEBUG) so that debug messages will be visible. In the log window you should now see "threshold passed" messages.

As the events output by the `thres` block are made available via a `mqueue`, these can easily be dumped to stdout using `ubx-mq`:

```
$ ubx-mq read threshold_events -p threshold
{ts={nsec=135724534,sec=287814},dir=1}
{ts={nsec=321029297,sec=287814},dir=0}
{ts={nsec=448964856,sec=287815},dir=1}
```

To stop the application again, just type `Ctrl-c` in the `ubx-lauch` window.

2.3 Run the PID controller block

This more complex example demonstrates how multiple, modular `usc` files can be *composed* into an application and how configuration can be *overlayed*. The use-case is a robot controller composition which shall be used in a test mode (extra mqueue ouputs, no real-time priorities) and in regular mode (real-time priorities, no debug outputs).

Before launching, run `ubx-log` as above to see potential errors.

Then:

```
$ cd /usr/local/share/ubx/examples/usc/pid/
$ ubx-launch -webif -c pid_test.usc,ptrig_nrt.usc
merging ptrig_nrt.usc into pid_test.usc
core_prefix: /usr/local
prefixes:    /usr, /usr/local
starting up webinterface block (http://localhost:8888)
loaded request_handler()
```

The *ubx-log* window will show a number messages from the instantiation of the application. The last lines will be about the blocks that were started.

2.3.1 Use the webif block

The cmdline arg `-webif` instructed `ubx-launch` to create a web interface block. This block is useful for debugging and introspecting the application. Browser to http://localhost:8888 and 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.

2.3.2 Examining data-flow

The `pid_test.usc` creates several mqueue blocks in order to export internal signals for debugging. They can be accessed using the `ubx-mq` tool:

```
$ ubx-mq list
243b40de92698defa93a145ace0616d2  1    trig_1-tstats
e8cd7da078a86726031ad64f35f5a6c0  10   ramp_des-out
e8cd7da078a86726031ad64f35f5a6c0  10   ramp_msr-out
e8cd7da078a86726031ad64f35f5a6c0  10   controller_pid-out
```

For example to print the `controller_pid-out` signal:

```
ubx-mq  read controller_pid-out
{1775781.9200001,1775781.9200001,1775781.9200001,1775781.9200001,1775781.9200001,
↪1775781.9200001,1775781.9200001,1775781.9200001,1775781.9200001,1775781.9200001}
{1776377.9200001,1776377.9200001,1776377.9200001,1776377.9200001,1776377.9200001,
↪1776377.9200001,1776377.9200001,1776377.9200001,1776377.9200001,1776377.9200001}
{1776974.0200001,1776974.0200001,1776974.0200001,1776974.0200001,1776974.0200001,
↪1776974.0200001,1776974.0200001,1776974.0200001,1776974.0200001,1776974.0200001}
{1777570.2200001,1777570.2200001,1777570.2200001,1777570.2200001,1777570.2200001,
↪1777570.2200001,1777570.2200001,1777570.2200001,1777570.2200001,1777570.2200001}
...
```

2.4 Important concepts

The following concepts are important to know:

- **modules** are shared libraries that contain blocks or custom types and are loaded when the application is launched.
- a **node** is a run-time container into which *modules* are loaded and which keeps track of blocks etc.
- **types**: microblx essentially uses the C type system (primitive types, structs and arrays of both) for *configs* and data sent via *ports*. To be supported by tools (that is in *usc* files or by tools like *ubx-mq*), custom types must be registered with microblx. The *stdtypes* module contains a large number of common types like *int*, *double*, stdints (*int32_t*) or time handling *ubx_tstat*.
- **cblocks** vs **iblocks**: there are two types of blocks: *cblocks* (computation blocks) are the "regular" functional blocks with a *step* hooks. In contrast *iblocks* (interaction blocks) are used to implement communication between blocks and implement *read* and *write* hooks. For most applications the available iblocks are sufficient, but sometimes creating a custom one can be useful.
- **triggers**: *triggers* are really just cblocks with a configuration for specifying a schedule and other properties such as period, thread priority, etc. *ptrig* is the most commonly used trigger which implements a periodic, POSIX pthread based trigger. Sometimes it is useful to implement custom triggers that trigger based on external events. The *trig_utils* functions (see *./libubx/trig_utils.h*) make this straightforward.
- **dynamic block interface**: sometimes the type or length of the port data is not static but depends on configuration values themselves. This is almost always the case for iblocks

CHAPTER 3

Developing microblx blocks

3.1 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 / used as configuration
4. declaring block meta-data: providing further oinformation about a block
5. declaring and implementing hook functions: how is the block initialized, started, run, stopped and cleaned up?
    1. reading configuration values: retrieving and using configuration from inside the block
    2. reading and writing data from resp. to ports
6. declaring the block: how to put everything together
7. registration of blocks and types in module functions: 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 *Generating blocks with ubx_genblock*.

3.2 Declaring configuration

Note: Since microblx v0.9, static block definitions must use the "proto" types `ubx_proto_config_t`, `ubx_proto_port_t` and `ubx_proto_block_t` to define prototype blocks. At runtime (i.e. in hooks etc) the non-*_proto_* versions are used as before

Configuration is described with a `{ 0 }` terminated array of `ubx_proto_config_t` types:

```
ubx_proto_config_t rnd_config[] = {
    { .name="min_max_config", .type_name = "struct random_config" },
    { 0 },
};
```

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*.) Primitives (*int*, *float*, *uint32_t*, ...) are available from the `stdtypes` module.

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

```
ubx_config_t rnd_config[] = {
    { .name="min_max_config", .type_name = "struct random_config", .min=1, .max=1 },
    { 0 },
};
```

These specifiers 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

3.3 Declaring ports

Like configurations, ports are described with a `{ 0 }` terminated array of `ubx_proto_port_t` types:

```
ubx_proto_port_t rnd_ports[] = {
    { .name="seed", .in_type_name="unsigned int" },
    { .name="rnd", .out_type_name="unsigned int" },
    { 0 },
};
```

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.

3.4 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 commonly used so far:

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

3.5 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 will 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 exists. Return zero if OK, non-zero otherwise.
- `start`: become operational, open/enable device, carry out last checks. Cache pointers to ports, apply configurations.
- `step`: read from ports, compute, write to ports
- `stop`: stop/close device. stop is often not used.
- `cleanup`: free all memory, release all resources.

3.5.1 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:

```
b->private_data = calloc(1, sizeof(struct random_info))

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

Retrieve and use it in the other hooks:

```
struct block_info *inf;

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

3.5.2 Reading configuration values

Configurations can be accessed in a type safe manner using the `cfg_getptr_<TYPE>` familiy of functions, which are available for all basic types. For example, the following snippet retrieves a scalar `uint32_t` config and uses a default `47` if unconfigured:

```
long len;
uint32_t *value;

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

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

Defining type safe configuration accessors for custom types can be achieved using the macros described in section *Declaring type safe accessors*.

The following example from the random (`std_blocks/ubx/random.c`) block shows how this is done for `struct min_max_config`:

```
def_cfg_getptr_fun(cfg_getptr_random_config, struct random_config)

int rnd_start(ubx_block_t *b)
{
     long len;
     const struct random_config* rndconf;

     /*...*/

     /* get and store min_max_config */
     len = cfg_getptr_random_config(b, "min_max_config", &rndconf);

     if (len < 0) {
             ubx_err(b, "failed to retrieve min_max_config");
             return -1;
     } else if (len == 0) {
             /* set a default */
             inf->min = 0;
             inf->max = INT_MAX;
     } else {
             inf->min = rndconf->min;
             inf->max = rndconf->max;
     }
}
```

Like with the first example, the the generated accessor `cfg_getptr_random_config` returns <0 in case of error, 0 if unconfigured, or the array length (>0) if configured. If >0 `rndconf` will be set to point to the actual configuration data.

Copy configs or use pointer directly?

In the above example, the configuration values are copied to the internal info struct. This is done to be able to assign defaults should no configuration have been given by the user. If this is not required (e.g. for mandatory configurations), it is perfectly OK to use the pointers retrieved via *cfg_getptr...* functions directly. The following table summarizes the permitted changes in each block state:

block state	allowed config changes
`preinit`	resizing and changing values
`inactive`	changing values
`active`	no changes allowed

Due to possible resizing in *preinit*, config ptr and length should be re-retreived in *init*.

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.

3.5.3 Reading from and writing to ports

Writing to ports can be done using the `write_<TYPE>` or `write_<TYPE>_array` functions. For example:

```
/* writing to a port */
unsigned int val = 1;
write_uint(my_outport, &val);

/* reading from a port */
long len;
int val;

len = read_int(my_inport, &val);

if (len < 0)
        ubx_err(b, "port read failed");
        return -1;
else if (len == 0) {
        /* no data on port */
        return 0;
} else {
        ubx_info(b, "new data: %i", val);
}

...
```

For more see `std_blocks/ramp/ramp.c`.

Type safe read/write functions are defined for all basic types and availale via the `<ubx.h>` header. Defining similar functions for custom types can be done using the macros described in *Declaring type safe accessors*.

3.6 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_proto_block_t random_comp = {
    .name = "myblocks/random",
    .type = BLOCK_TYPE_COMPUTATION,
    .meta_data = rnd_meta,
    .configs = rnd_config,
    .ports = rnd_ports,

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

3.7 Declaring types

All types used for configurations or 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).

3.7.1 Declaring type safe accessors

The following macros are available to define type safe accessors for accessing configuration and reading/writing from ports:

```
def_type_accessors(SUFFIX, TYPENAME)

/* will define the following functions */
long read_SUFFIX(const ubx_port_t* p, TYPENAME* val);
int write_SUFFIX(const ubx_port_t *p, const TYPENAME *val);
long read_SUFFIX_array(const ubx_port_t* p, TYPENAME* val, const int len);
```

(continues on next page)

(continued from previous page)

```
int write_SUFFIX_array(const ubx_port_t* p, const TYPENAME* val, const int len);
long cfg_getptr_SUFFIX(const ubx_block_t *b, const char *cfg_name, const TYPENAME␣
↪**valptr);
```

Using these is strongly recommended for most blocks.

Variants:

- `def_port_accessors(SUFFIX, TYPENAME)` will define the port but not the config accessors.
- `def_cfg_getptr_fun(FUNCNAME, TYPENAME)` will only define the config accessor
- `def_port_writers(FUNCNAME, TYPENAME)` and `def_port_readers(FUNCNAME, TYPENAME)` will only define the port write or read accessors respectively.

3.7.2 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 very useful but optional. If no type reflection is needed, don't include the `.hexarr` file and pass `NULL` as a third argument to `def_struct_type`.

3.8 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_t* ni)
2: {
3:         ubx_type_register(nd, &random_config_type);
4:         return ubx_block_register(nd, &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_t *nd)
{
    ubx_type_unregister(nd, "struct random_config");
    ubx_block_unregister(nd, "ubx/random");
}
UBX_MODULE_CLEANUP(rnd_module_cleanup)
```

3.9 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, ...)
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 120 by default), so make sure to keep log message short and sweet (or increase the length 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_t *nd, const char* src, const char* fmt, ...)

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

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

3.10 SPDX License Identifiers

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

3.11 Generating blocks with ubx_genblock

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
    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.

3.11.1 Compile the block

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

3.11.2 Launch block using ubx-launch

```
$ ubx-ilaunch -webif -c myblock.usc
```

Run `ubx-launch -h` for full options.

Browse to http://localhost:8888

3.12 Block Interface Guidelines

- use `long` (signed) for ubx type related lengths and sizes. This is sufficently large and errors can be returned as negative values (example: `cfg_getptr_uint32`).
- (i)blocks that allow configuring *type* and *length* of data to be handled should use the canonical config names `type_name` and `data_len`.

CHAPTER 4

Composing microblx systems

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

4.1 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="ubx/ptrig" },
      ...
   },

   -- connect blocks
   connections = {
      { src="x1.out", tgt="y1.in" },
      { src="y1.out", tgt="x1.in", buffer_len=16 },
   },
```

(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,
                                 chain0={
                                    -- 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 } } } }
   },
}
```

4.1.1 Launching

usc files like the above example can be launched using `ubx-launch` tool. Run with `-h` for further information. The following example

```
$ cd /usr/local/share/ubx/examples/usc/pid/
$ ubx-launch -webif -c pid_test.usc,ptrig_nrt.usc
...
```

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

Unless the `-nostart` option is provided, all blocks will be initialized, configured and started. `ubx-launch` handles this in safe way by starting up active blocks after all other blocks (In earlier versions, there was `start` directive to list the blocks to be started, however now this information is obtained by means of the block attributes `BLOCK_ATTR_ACTIVE` and `BLOCK_ATTR_TRIGGER`.)

4.1.2 Node configs

Node configs allow to assign the same configuration to multiple blocks. This is useful to avoid repeating global configuration values that are identical for multiple blocks.

The `node_configurations` keyword allows to define one or more named node 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.

4.1.3 Connections

The powerful `connections` keyword supports connecting blocks in multiple ways:

- cblocks to cblocks
- cblocks to iblocks
- cblocks to non-existing iblocks (the latter are created on the fly)

The syntax for these variants is discussed below.

cblock to cblock connections

The following example shows how to create ports among cblock ports:

```
{ src="blkA.portX", tgt="blkB.portY", type="lfds_cyclic", config = { ... }
```

- both `src` and `tgt` are of the form `CBLOCK.PORT`. Both blocks and ports must exist.
- `type` specifies the type of iblock to create for the connection. If unset it defaults to `ubx/lfds_cyclic`
- `config` is the optional configuration to apply to the newly created iblock. The configs `type_name` and `data_len` are set automatically unless specified.

cblock to iblock

The following examples illustrates creating connections to/from an *existing* iblock `myMq`:

```
{ src="blkX.portZ", tgt="myMQ" }

-- or

{ src="myMQ", tgt="blkX.portZ" }
```

- the iblock must exist and be of the form `IBLOCK` (i.e. no port).
- the cblock must exist and be of the form `CBLOCK.PORT`
- `type` and `config` must not be set (they will be ignored with a warning).

cblock to non-existing iblock

The following example creates a new mqueue with an automatic, unique name, configures it with `config` and connect `blkX.portZ` to it:

```
{ src="blkX.portZ", type="ubx/mqueue", config={ buffer_len=32 } }
```

- `type` must be set to desired iblock type and one of `src` or `tgt` must be unset
- `type_name`, `data_len` and `buffer_len` are set automatically unless defined in `config`.
- for type `ubx/mqueue`: if no `mq_id` is set in `config`, then `mq_id` is set to the corresponding peer "BLOCK.PORT", e.g. to `blkX.portZ` in the example above.

This form is useful to create one-line connections via mqueues or similar.

4.2 Hierarchical compositions

Using hierarchical composition[1] an application can be composed from other compositions. The motivation is to permit reuse of the individual compositions.

The `subsystems` keyword accepts a list of namespace-subsystem entries:

```
return bd.system {
    import = ...
    subsystems = {
        subsys1 = bd.load("subsys1.usc"),
        subsys2 = bd.load("subsys1.usc"),
    }
}
```

Subsystem elements like *configs* can be accessed by higher levels by adding the subsystem namespace. For example, the following lines override a configuration value of the `blk` block in subsystems `sub11` and `sub11/sub21`:

```
configurations = {
    { name="sub11/blk",        config = { cfgA=1, cfgB=2 } },
    { name="sub11/sub21/blk", config = { cfgA=5, cfgB=6 } },
}
```

Note how the subsystem namespaces prevent name collisions of the two identically names blocks. Similar to configurations, connections can be added among subsystems blocks:

```
connections = {
    { src="sub11/sub21/blk.portX", tgt="sub11/blk.portY" },
},
```

When launched, a hierarchical system is instantiated in a similar way to a non-hierarchical one, however:

- modules are only imported once
- blocks from all hierarchy levels are instantiated, configured and started together, i.e. the hierarchy has no implications on the startup sequence.
- microblx block names use the fully qualified name including the namespace. Therefore, the #blockname syntax for resolving block pointers works just the same.
- if multiple configs for the same block exist, only the highest one in the hierarchy will be applied.
- node configs are always global, hence no prefix is required. In case of multiple identically named node configs, the one at the highest level will be selected.

4.2.1 Merging subsystems

It is possible to add a subsystem without a namespace, as shown by the following snippet:

```
return bd.system {
    subsystems = {
        bd.load("subsys1.usc"),
    }
}
```

[1] This feature was introduced in the context of the COCORF RobMoSys Integrated Technical Project. Please see docs/dev/001-blockdiagram-composition.md for background information.

In this case, the `subsys1.usc` system will be merged directly into the parent system. Note that entries of the parent system take precedence, so in case of conflicts elements of the subsystem will be skipped.

This feature is useful to avoid an extra hierarchy level.

4.3 Model mixins

To obtain a reusable composition, it is important to avoid introducing platform specifics such as `ptrig` blocks and their configurations. Instead, passive `trig` blocks can be used to encapsulate the trigger schedule. `ptrig` or similar active blocks can then be added at *launch time* by merging them (encapsulated in an usc file) into the primary model by specifying both on the `ubx-launch` command line.

For example, consider the example in `examples/systemmodels/composition`:

```
ubx-launch -webif -c deep_composition.usc,ptrig.usc
```

Note: unlike merging from within the usc using an unnamed `subsystems` entry (see *Merging subsystems*), models merged on the command line will *override* existing entries.

4.4 Alternatives

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

4.4.1 Launching in C

It is possible to avoid the Lua scripting layer entirely and launch an application in C/C++. A small self-contained example `c-launch.c` is available under `examples/C/` (see the `README` for further details).

For a more complete example, checkout the respective tutorial section *Deployment via C program*. Please note that such launching code is a likely candidate for code generation and there are plans for a *usc-to-C* compiler. Please ask on the mailing if you are interested.

4.4.2 Lua scripts

One can write a Lua "deployment script" similar to the `ubx-launch`. Checkout the scripts in the `tools` section. This approach not recommended under normally, but can be useful in specific cases such as for building dedicated test tools.

CHAPTER 5

Tutorial: close loop control of a robotic platform

5.1 Goal

This walk-through that shows how to:

1. write, compile and install two blocks:
   - the *plant* (a two DoF robot that accepts velocity as input, and gives 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 program.

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

5.2 Introductory steps

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

The plant and controller have 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", min=2, max=2 },
      { name="initial_position", type_name="double", min=2, max=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, 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", min=1, max=1 },
      { name="target_pos", type_name="double", min=2, max=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 = { 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
generating platform_2dof/bootstrap
...
$ ubx-genblock -c platform_2dof_control.lua -d platform_2dof_control
generating platform_2dof_control/bootstrap
...
```

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

- `boostrap`
- `configure.ac`
- `Makefile.am`

- platform_2dof.h
- platform_2dof.c
- platform_2dof.usc

The only files we will modify are the C files platform_2dof.c and platform_2dof_control.c.

5.3 Code of the blocks

The auto-generated files already give 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 = 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);
        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, "%s", __func__);
        int ret = 0;
        return ret;
}

/* 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, "%s", __func__);
        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, "%s", __func__);
}
```

We will need then to insert the code indicated by the comments.

5.3.1 Step 1: add block state and helpers

First add variables for storing the robot state, and implement the 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.

5.3.2 Step 2: Initialization and start functions

The `init` function is 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 specified in the `.usc` or main application file).

```
int platform_2dof_init(ubx_block_t *b)
{
        long len;
        struct platform_2dof_info *inf;
        const double *pos_vec;

        /* allocate memory for the block local state */
        if ((inf = calloc(1, sizeof(struct platform_2dof_info)))==NULL) {
                ubx_err(b,"platform_2dof: failed to alloc memory");
                return EOUTOFMEM;
        }

        b->private_data=inf;
        update_port_cache(b, &inf->ports);

        /* read configuration - initial position */
        len = cfg_getptr_double(b, "initial_position", &pos_vec);

        /* this will never assert unless we made an error
         * (e.g. mistyped the configuration name), since min/max
         * checking will catch misconfigurations before we get
         * here. */
        assert(len==2);

        inf->r_state.pos[0] = pos_vec[0];
        inf->r_state.pos[1] = pos_vec[1];

        /* read configuration - max velocity */
        len = cfg_getptr_double(b, "joint_velocity_limits", &pos_vec);
        assert(len==2);

        inf->r_state.vel_limit[0] = pos_vec[0];
        inf->r_state.vel_limit[1] = pos_vec[1];
        inf->r_state.vel[0] = 0.0;
        inf->r_state.vel[1] = 0.0;

        return 0;
}
```

The function

```
long cfg_getptr_double(ubx_block_t *b, const char *name, const double **ptr)
```

returns the address of the `double` configuration in the pointer `ptr`. In this case the return value will be `2` (the length of the data) or `-1` (failure, e.g. mistyped configuration name). Because we set `min` and `max` in the configuration declaration, we can be sure that at this point the array length is not anything but 2.

In 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);
        return 0;
}
```

5.3.3 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_double_array(inf->ports.desired_vel, velocity, 2);
        assert(ret>=0);

        if (ret == 0) { /* nodata */
                ubx_notice(b,"no velocity setpoint");
                velocity[0] = velocity[1] = 0.0;
        }

        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 */
        ubx_debug(b, "writing pos [%f, %f]",
                  inf->r_state.pos[0], inf->r_state.pos[1]);
        write_double_array(inf->ports.pos, inf->r_state.pos, 2);
}
```

In case there is no value on the port, a notice is logged and the nominal velocity is set to zero. This will always happen for the first trigger, since the controller did step yet and thus has not produced a velocity command yet.

5.3.4 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.

5.3.5 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:

- `platform_2dof.c`
- `platform_2dof_control.c`

5.3.6 Compiling the blocks

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

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

See also the quickstart.

5.4 Deployment via the usc (microblx system composition) file

`ubx-genblock` generated sample `.usc` files to run each block independently. We want to run and compose them together and make the resulting signals available using message queues. The composition file **platform_2dof_and_control.usc** is quite self explanatory: It contains

- the libraries to be imported,
- which blocks (name, type) to create,
- the configuration values of blocks.
- the connections among ports

The file `platform_2dof_and_control.usc` is shown below:

```
-- -*- mode: lua; -*-

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

   blocks = {
      { name="plat1", type="platform_2dof" },
      { name="control1", type="platform_2dof_control" },
      { name="ptrig1", type="ubx/ptrig" },
   },

   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} } },
```

(continues on next page)

(continued from previous page)

```
      { name="ptrig1", config = { period = {sec=0, usec=100000 },
                                  sched_policy="SCHED_OTHER",
                                  sched_priority=0,
                                  chain0={
                                     { b="#plat1" },
                                     { b="#control1" } } } },
   },

   connections = {
      { src="plat1.pos", tgt="control1.measured_pos" },
      { src="control1.commanded_vel",tgt="plat1.desired_vel" },
      { src="plat1.pos", type="ubx/mqueue" },
      { src="control1.commanded_vel", type="ubx/mqueue" },
   },
}
```

It is worth noting that configuration types can be arrays (*e.g.* `target_pos`), strings (`file_name` and `report_conf`) and structures (`period`) and vector of structures (`chain0`). Types can be checked using `ubx-modinfo`:

```
$ ubx-modinfo show platform_2dof
module platform_2dof
  license: MIT

  blocks:
    platform_2dof [state: preinit, steps: 0] (type: cblock, prototype: false, attrs: )
       configs:
           joint_velocity_limits [double] nil //
           initial_position [double] nil //
       ports:
           pos  [out: double[2] #conn: 0] // measured position [m]
           desired_vel [in: double[2] #conn: 0]  // desired velocity [m/s]
```

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 .

To show the position and velocity signal, use the `ubx-mq` tool:

```
$ ubx-mq list
e8cd7da078a86726031ad64f35f5a6c0  2    vel_cmd
e8cd7da078a86726031ad64f35f5a6c0  2    pos_msr

$ ubx-mq read pos_msr
{1.1,1}
{1.13403850806,1.03503964065}
{1.1679003576875,1.0698974270313}
{1.2012522276799,1.1042302343764}
{1.2342907518755,1.1382404798718}
...
```

5.4.1 Some considerations about the fifos

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

The more common use-case is that each outport is connected to an inport with it's 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**.

5.5 Deployment via C program

> **Warning:** the following example is to illustrate the possibility of C only lauching, however generally, the usc DSL should be preferred. Furthermore, an *usc* compiler that can automatically and safely generate the code below is planned. If interested, please ask on the mailing list.

This example is an extension of the example `examples/C/c-launch.c`. It will be clear that using the above DSL based method is somewhat easier, but if for some reason we want to eliminate the dependency from *Lua*, this example show that is possible.

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`

5.5.1 Setup the build system starting from the build part

configure.ac

```
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])
AC_CONFIG_MACRO_DIR([m4])
```

(continues on next page)

(continued from previous page)

```
# 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 >= 0.9.0)

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 = main.c
platform_main_CFLAGS = @UBX_CFLAGS@ \
                       -I${top_srcdir}/../../../std_blocks/trig/types/

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

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
```

5.6 The program

The main follows the same structure of the `.usc` file.

5.6.1 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, __func__,  "failed to init control1");
```

More info on logging can be found in the *Real-time logging*.

5.6.2 Libraries

It starts with some includes (structs that are needed in configuration) and loading of the libraries

```
#include <ubx/ubx.h>
#include <ubx/trig_utils.h>

#define WEBIF_PORT "8810"

#include "ptrig_period.h"
#include "signal.h"

def_cfg_set_fun(cfg_set_ptrig_period, struct ptrig_period);
def_cfg_set_fun(cfg_set_ubx_triggee, struct ubx_triggee);

static const char* modules[] = {
        "/usr/local/lib/ubx/0.9/stdtypes.so",
        "/usr/local/lib/ubx/0.9/ptrig.so",
        "/usr/local/lib/ubx/0.9/platform_2dof.so",
        "/usr/local/lib/ubx/0.9/platform_2dof_control.so",
        "/usr/local/lib/ubx/0.9/webif.so",
        "/usr/local/lib/ubx/0.9/lfds_cyclic.so",
};

int main()
{
        int ret = EXIT_FAILURE;
        ubx_node_t nd;
        ubx_block_t *plat1, *control1, *ptrig1, *webif, *fifo_vel, *fifo_pos;

        /* initalize the node */
        nd.loglevel = 7;
        ubx_node_init(&nd, "platform_and_control", 0);

        /* load modules */
        for (unsigned int i=0; i<ARRAY_SIZE(modules); i++) {
                if(ubx_module_load(&nd, modules[i]) != 0){
                        ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "fail to load ␣
↪module %s %i",modules[i], i);
                        goto out;
                }
        }
```

5.6.3 Block instantiation

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

```
        if((plat1 = ubx_block_create(&nd, "platform_2dof", "plat1"))==NULL){
                ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "fail to create plat1");
                goto out;
        }
```

5.6.4 Property configuration

Now we have the more tedious part, that is the configuration. We use the type safe helper functions, for example

```
int cfg_set_double(const ubx_block_t *b, const char *cfg_name, const double *valptr,␣
↪const long len);
```

where

- `b` is the block
- `cfg_name` the name of the config to set
- `valptr` is a pointer to the value to assign to the config
- `len` is the array size of `valptr`

String property

```
        /* webif port config */
        if (cfg_set_char(webif, "port", WEBIF_PORT, strlen(WEBIF_PORT))) {
                ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "failed to configure port_vel
↪");
                goto out;
        }
```

The string can be passed as a `static const char[]` or using a `#define`.

Double property

```
        /* gain */
        double gain = 0.12;

        if (cfg_set_double(control1, "gain", &gain, 1)) {
                ubx_log(UBX_LOGLEVEL_ERR, &nd, __func__,  "failed to configure gain");
                goto out;
        }
```

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

```
        /* joint_velocity_limits */
        const double joint_velocity_limits[2] = { 0.5, 0.5 };

        if (cfg_set_double(plat1, "joint_velocity_limits", joint_velocity_limits, 2))
↪{
                ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "failed to configure joint_
↪velocity_limits");
                goto out;
        }
```

Almost the same as before, but being an array of two elements, we don't need to take the reference `&` here.

Structure property

Same thing for `struct` types:

```
        /* ptrig config */
        const struct ptrig_period period = { .sec=1, .usec=14 };

        if (cfg_set_ptrig_period(ptrig1, "period", &period, 1)) {
                ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "failed to configure ptrig_
↪period");
                goto out;
        }
```

Note that for custom types, it is necessary to define the accessor using a typemacro, e.g:

```
def_cfg_set_fun(cfg_set_ptrig_period, struct ptrig_period);
```

Array of structures:

```
        /* chain0 */
        const struct ubx_triggee chain0[] = {
                { .b = plat1, .num_steps = 1 },
                { .b = control1, .num_steps = 1 },
        };

        if (cfg_set_ubx_triggee(ptrig1, "chain0", chain0, ARRAY_SIZE(chain0))) {
                ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "failed to configure chain0
↪");
                goto out;
        }
```

5.6.5 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:

```
        /* connections setup - first ports must be retrived and then connected */
        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_ports_connect(plat1_pos, control1_measured_pos, fifo_pos);
        ubx_ports_connect(control1_commanded_vel, plat1_desired_vel, fifo_vel);
```

5.6.6 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, &nd,__func__,  "failed to init control1");
                goto out;
```

(continues on next page)

(continued from previous page)

```
        }

        if(ubx_block_start(control1) != 0) {
                ubx_log(UBX_LOGLEVEL_ERR, &nd,__func__,  "failed to start control1");
                goto out;
        }
```

The same applies to all other blocks.

Once all the blocks are running, the `ptrig1` block will `step` all the blocks in the configured order. To prevent the main process to terminate, we can use `ubx_wait_sigint` to wait for the user to type `ctrl-c`:

```
        ubx_wait_sigint(UINT_MAX);
        printf("shutting down\n");
```

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
```

5.7 Next steps

Some suggestions for next steps:

- it can be necessary to make the *array size* of data sent and received via ports configurable. Checkout the saturation block for a simple example.
- sometimes a block shall support multiple types. This can be done at
  - *compile time*: (example ramp block block)
  - *run-time*: (examples: most iblocks, e.g. lfds_cyclic)

CHAPTER 6

Frequently asked questions

6.1 Developing blocks

6.1.1 How to use C++ for blocks

Checkout the example `std_blocks/cppdemo`.

Note: *designated initializers*, which are used to initialize `ubx_proto_` structures are only supported by g++ versions 8 and newer!

6.1.2 What the difference between block types and instances?

There are very few differences. A prototype block is added by module init functions using `ubx_block_register` and must also be removed by the corresponding module cleanup hook using `ubx_block_unregister`. A prototype blocks `prototype` ptr is NULL.

Block instances are cloned from existing blocks using `ubx_block_create` and the instances `block->prototype` pointer is set to the block is was cloned from. Normally blocks are cloned from prototype blocks, but it is possible to clone any block (a warning is issued currently).

That said, the above are *internals* and you should not rely on them. Instead, use the predicates `blk_is_proto` and `blk_is_instance` if you need to determine what is what.

6.1.3 Why do you cache port pointers in the block info structure?

For two reasons:

- it's simpler to cache it once and then just use the pointer directly
- to a lesser degree: for performance. It avoids a repeated hash table lookups in `step`.

Note that the `ubx-genblock` script automatically takes care caching ports of this.

6.1.4 Avoiding `static`

You can avoid cluttering block functions and globals with `static`, by adding `-fvisibility=hidden` to `CFLAGS`.

6.2 Running microblx

6.2.1 `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/`.

6.2.2 luablock: "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.

6.2.3 Running with real-time priorities

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

```
$ sudo setcap cap_sys_nice+ep `which luajit`
```

Note that this will grant these capabilities to luajit binary in *PATH*. If you don't want to do this system-wide, you can do this for a local `luajit` binary instead.

In addition, you should pass the command-line option `-mlockall` to `ubx-launch`, to ensure memory is locked. For scripts, pass the `ND_MLOCK_ALL` node attribute to `ubx_node_init`.

6.2.4 I'm not getting core dumps when running with real-time priorities

This is not a bug, but a safety mechanism to prevent potential leaking of priviledged data from a `setcap` process. To override this, pass the `ubx-launch` command-line arg `-dumpable` or the for scripts the node attribute `ND_DUMPABLE`.

6.2.5 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 immediately.
- You ran the wrong Lua executable (e.g. a standard Lua instead of `luajit`).

If none of this works, see the following topic.

6.3 Debugging

6.3.1 Debugging segfaults

One of the best ways 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
```

Sometimes, running gdb directly on the processes produces better results than post-mortem coredumps. For example, to run the pid example with gdb attached:

```
$ cd /usr/local/share/ubx/examples/usc/pid
$ gdb luajit --args luajit `which ubx-launch` -c pid_test.usc,ptrig_nrt.usc
GNU gdb (Debian 9.1-2) 9.1
...
Reading symbols from luajit...
(No debugging symbols found in luajit)
(gdb) run
Starting program: /usr/bin/luajit /usr/local/bin/ubx-launch -c pid_test.usc,ptrig_nrt.
↪usc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
merging ptrig_nrt.usc into pid_test.usc
core_prefix: /usr/local
prefixes:    /usr, /usr/local
[New Thread 0x7ffff7871700 (LWP 2831757)]
...
```

6.3.2 Running valgrind

Valgrind is very useful to track down memory leaks or sporadic segfaults. To run it on dynamically loaded modules, the `UBX_CONFIG_VALGRIND` flag must be enabled in `ubx.h`. This flag will pass the `RTLD_NODELETE` flag to `dlopen(3)`, which causes modules not really to be unloaded. This is essential for valgrind to print meaningful traces in module code.

After that, you can run valgrind as follows on an *usc* file:

```
valgrind --leak-check=full \
         --track-origins=yes \
         luajit `which ubx-launch` -t 3 -c examples/usc/threshold.usc
...
```

This will run the demo for 3 seconds and then exit. Valgrind may print warnings related to luajit like `Conditional jump or move depends on uninitialised value`, which can be ignored (or silenced by building luajit with valgrind support , see `-DLUAJIT_USE_VALGRIND`)

Running a script can be done likewise:

```
$ valgrind --leak-check=full \
           --track-origins=yes \
           luajit tests/test_ptrig.lua
...
```

6.4 meta-microblx

6.4.1 building luajit fails

`luajit` fails with the following message:

```
arm-poky-linux-gnueabi-gcc  -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a8 -fstack-
↪protector-strong  -D_FORTIFY_SOURCE=2 -Wformat -Wformat-security -Werror=format-
↪security --sysroot=/build/bbblack-zeus/build/tmp/work/cortexa8hf-neon-poky-linux-
↪gnueabi/luajit/2.0.5+gitAUTOINC+02b521981a-r0/recipe-sysroot -fPIC   -Wall   -D_
↪FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -U_FORTIFY_SOURCE  -DLUA_ROOT=\"/usr\" -
↪DLUA_MULTILIB=\"lib\" -fno-stack-protector  -O2 -pipe -g -feliminate-unused-debug-
↪types -fmacro-prefix-map=/build/bbblack-zeus/build/tmp/work/cortexa8hf-neon-poky-
↪linux-gnueabi/luajit/2.0.5+gitAUTOINC+02b521981a-r0=/usr/src/debug/luajit/2.0.
↪5+gitAUTOINC+02b521981a-r0                       -fdebug-prefix-map=/build/bbblack-
↪zeus/build/tmp/work/cortexa8hf-neon-poky-linux-gnueabi/luajit/2.0.
↪5+gitAUTOINC+02b521981a-r0=/usr/src/debug/luajit/2.0.5+gitAUTOINC+02b521981a-r0
↪                 -fdebug-prefix-map=/build/bbblack-zeus/build/tmp/work/cortexa8hf-
↪neon-poky-linux-gnueabi/luajit/2.0.5+gitAUTOINC+02b521981a-r0/recipe-sysroot=
↪               -fdebug-prefix-map=/build/bbblack-zeus/build/tmp/work/cortexa8hf-
↪neon-poky-linux-gnueabi/luajit/2.0.5+gitAUTOINC+02b521981a-r0/recipe-sysroot-
↪native=  -c -o lj_obj_dyn.o lj_obj.c
In file included from /usr/include/bits/errno.h:26,
                 from /usr/include/errno.h:28,
                 from host/buildvm.h:13,
                 from host/buildvm_fold.c:6:
/usr/include/linux/errno.h:1:10: fatal error: asm/errno.h: No such file or directory
    1 | #include <asm/errno.h>
      |          ^~~~~~~~~~~~~
compilation terminated.
```

This solution is to install `gcc-multilib` on the build host.

CHAPTER 7

Microblx Module Index

7.1 Module trig

7.1.1 Block ubx/trig

Type: cblock
Attributes: trigger
Meta-data: { doc='simple, activity-less trigger', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
num_chains	`int`	number of trigger chains. def: 1
tstats_mode	`int`	0: off (def), 1: global only, 2: per block
tstats_profile_path	`char`	directory to write the timing stats file to
tstats_output_rate	`double`	throttle output on tstats port
tstats_skip_first	`int`	skip N steps before acquiring stats
loglevel	`int`	

Ports

name	out type	out len	in type	in len	doc
active_chain			`int`	1	switch the active trigger chain
tstats	`struct ubx_tstat`	1			timing statistics (if enabled)

7.2 Module ptrig

7.2.1 Block ubx/ptrig

Type: cblock
Attributes: trigger, active
Meta-data: { doc='pthread based trigger', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
period	`struct ptrig_period`	trigger period in { sec, ns }
stacksize	`size_t`	stacksize as per pthread_attr_setstacksize(3)
sched_priority	`int`	pthread priority
sched_policy	`char`	pthread scheduling policy
affinity	`int`	list of CPUs to set the pthread CPU affinity to
thread_name	`char`	thread name (for dbg), default is block name
autostop_steps	`int64_t`	if set and > 0, block stops itself after X steps
num_chains	`int`	number of trigger chains (def: 1)
tstats_mode	`int`	enable timing statistics over all blocks
tstats_profile_path	`char`	directory to write the timing stats file to
tstats_output_rate	`double`	throttle output on tstats port
tstats_skip_first	`int`	skip N steps before acquiring stats
loglevel	`int`	

Ports

name	out type	out len	in type	in len	doc
active_chain			`int`	1	switch the active trigger chain
tstats	`struct ubx_tstat`	1			out port for timing statistics
shutdown			`int`	1	input port for stopping ptrig

7.2.2 Types

Table 1: Types

type name	type class	size [B]
`struct ptrig_period`	struct	16

7.3 Module math_double

7.3.1 Block ubx/math_double

Type: cblock

Attributes:
Meta-data: { doc='math functions from math.h', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
func	`char`	math function to compute
data_len	`long`	length of output data (def: 1)
mul	`double`	optional factor to multiply with y (def: 1)
add	`double`	optional offset to add to y after mul (def: 0)

Ports

name	out type	out len	in type	in len	doc
x			`double`	1	math input
y	`double`	1			math output

7.4 Module rand_double

7.4.1 Block ubx/rand_double

Type: cblock
Attributes:
Meta-data: { doc='double random number generator block', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
seed	`long`	seed to initialize with

Ports

name	out type	out len	in type	in len	doc
out	`double`	1			rand generator output

7.5 Module ramp_double

7.5.1 Block ubx/ramp_double

Type: cblock

Attributes:
Meta-data: { doc='Ramp generator block', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
start	`double`	ramp starting value (def 0)
slope	`double`	rate of change (def: 1)
data_len	`long`	length of output data (def: 1)

Ports

name	out type	out len	in type	in len	doc
out	`double`	1			ramp generator output

7.6 Module pid

7.6.1 Block ubx/pid

Type: cblock
Attributes:
Meta-data: { doc=''', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
Kp	`double`	P-gain (def: 0)
Ki	`double`	I-gain (def: 0)
Kd	`double`	D-gain (def: 0)
data_len	`long`	length of signal array (def: 1)

Ports

name	out type	out len	in type	in len	doc
msr			`double`	1	measured input signal
des			`double`	1	desired input signal
out	`double`	1			controller output

7.7 Module saturation_double

7.7.1 Block ubx/saturation_double

Type: cblock
Attributes:
Meta-data: double saturation block
License: BSD-3-Clause

Configs

name	type	doc
data_len	`long`	data array length
lower_limits	`double`	saturation lower limits
upper_limits	`double`	saturation upper limits

Ports

name	out type	out len	in type	in len	doc
in			`double`	1	input signal to saturate
out	`double`	1			saturated output signal

7.8 Module luablock

7.8.1 Block ubx/luablock

Type: cblock
Attributes:
Meta-data: { doc='A generic luajit based block', realtime=false,}
License: BSD-3-Clause

Configs

name	type	doc
lua_file	`char`	
lua_str	`char`	
loglevel	`int`	

Ports

name	out type	out len	in type	in len	doc
exec_str	`int`	1	`char`	1	

7.9 Module cconst

7.9.1 Block ubx/cconst

Type: cblock
Attributes:
Meta-data: { doc='const value c-block', realtime=true }
License: BSD-3-Clause

Configs

name	type	doc
type_name	`char`	ubx type name of the value to output
data_len	`long`	data length of the value to output

7.10 Module iconst

7.10.1 Block ubx/iconst

Type: iblock
Attributes:
Meta-data: { doc='const value i-block', realtime=true }
License: BSD-3-Clause

Configs

name	type	doc
type_name	`char`	ubx type name of the value to output
data_len	`long`	data length of the value to output

7.11 Module lfds_cyclic

7.11.1 Block ubx/lfds_cyclic

Type: iblock
Attributes:
Meta-data: { doc='High performance scalable, lock-free cyclic, buffered in process communication description=[[This version is stongly typed and should be preferred This microblx iblock is based on based on liblfds ringbuffer (v0.6.1.1) (www.liblfds.org)]], version=0.01, hard_real_time=true,}
License: BSD-3-Clause

Configs

name	type	doc
type_name	`char`	name of registered microblx type to transport
data_len	`uint32_t`	array length (multiplier) of data (default: 1)
buffer_len	`uint32_t`	max number of data elements the buffer shall hold
allow_partial	`int`	allow msgs with len<data_len. def: 0 (no)
loglevel_overruns	`int`	loglevel for reporting overflows (default: NOTICE, -1 to disable)

Ports

name	out type	out len	in type	in len	doc
over-runs	`unsigned long`	1			Number of buffer overruns. Value is output only upon change.

7.12 Module mqueue

7.12.1 Block ubx/mqueue

Type: iblock
Attributes:
Meta-data: { doc='POSIX mqueue interaction', realtime=true,}
License: BSD-3-Clause

Configs

name	type	doc
mq_id	`char`	mqueue base id
type_name	`char`	name of registered microblx type to transport
data_len	`long`	array length (multiplier) of data (default: 1)
buffer_len	`long`	max number of data elements the buffer shall hold
blocking	`uint32_t`	enable blocking mode (def: 0)
unlink	`uint32_t`	call mq_unlink in cleanup (def: 1 (yes)

7.13 Module hexdump

7.13.1 Block ubx/hexdump

Type: iblock
Attributes:
Meta-data: { doc='hexdump interaction', realtime=false,}
License: BSD-3-Clause

CHAPTER 8

Indices and tables

- genindex
- modindex
- search