libmapper tutorial

May 27, 2011

Introduction

This tutorial introduces a new user to *libmapper*, providing steps to construct *devices* that are compatible with the *libmapper* network protocol. Programs that implement a *device* interface provide *signals*, which can be inputs or outputs, which are announced on the network. A compatible GUI program can then be used to *map* signals between devices.

Conceptually, a device typically consists only of outputs, for example in the case of a program that gets information from a human input device like a joystick, or only of inputs, for example a software-controlled synthesizer. For convenience, we will call output devices “controllers”, or “senders”, and input devices “synthesizers”, or “receivers”. This betrays the use case that was in mind when the *libmapper* system was conceived, but of course receivers could just as well be programs that control motors, lights, or anything else that might need control information. Similarly, senders could easily be programs that generate trajectory data based on algorithmic composition, or whatever you can imagine.

It is also possible to create devices which have inputs and outputs, and these can be mapped “in between” senders and receivers in order to perform some intermediate processing for example. However, this is a more advanced topic that won’t be covered in this tutorial.

First we will cover how to create a sender, and then how to create a receiver. This should be enough for you to try creating a connection and explore the possibilities afforded by dynamic mapping design.

Getting started

Since *libmapper* uses GNU autoconf, getting started with the library is the same as any other library on Linux; use `./configure` and then `make` to compile it. You’ll need `swig` available if you want to compile the Python bindings. On Mac OS X, we provide a precompiled Framework bundle for 32- and 64-bit Intel platforms, so using it with XCode should be a matter of including it in your project.

Overview of the API organization

If you take a look at the API documentation, there is a section called "modules". This is divided into the following sections:

- Signals
- Devices
- Admins
- Device database
- Signal database
- Connections database
- Links database
- Monitors

For this tutorial, the only sections to pay attention to are Devices and Signals. Admins is reserved for providing custom networking configurations, but in general you don't need to worry about it.

Monitor and the various database modules are used to keep track of what devices, signals and connections are on the network. Devices do not need to worry about this. It is used mainly for creating user interfaces for mapping design and will also not be covered here.

Functions and types from each module are prefixed with `mapper_<module>_`, in order to avoid namespace clashing. However, since this is a bit verbose, it is shortened to `mdev_` and `msig` for device and signal functions respectively.

Devices

Creating a device

To create a *libmapper* device, it is necessary to provide a few parameters to `mdev_new`:

```
mapper_device mdev_new( const char *name_prefix,
                        int initial_port,
                        mapper_admin admin )
```

Every device on the network needs a name and port. In fact the requested name and port are only “starting values”. There is an initialization period after a device is created where a unique ordinal is chosen to append to the device name. This allows multiple devices with the same name to exist on the network.

Similarly, each device gets a unique port to use for exchanging signal data. The provided port number is the “starting port”, but the allocation algorithm will possibly choose another port number close to it if someone else on the network is already using it. We usually use a port number of 9000, and let the system decide if this is appropriate.[1]

The third parameter of mdev_new is an optional admin instance. It is not necessary to provide this, but can be used to specify different networking parameters, such as specifying the name of the network interface to use.

An example of creating a device:

```
mapper_device my_device = mdev_new("test", 9000, 0);
```

Polling the device

The device lifecycle looks like this, in terrible ASCII diagram art:

```
mdev_new -> mdev_poll +-> mdev_free
              |       |
              +----<--+
```

In other words, after a device is created, it must be continuously polled during its lifetime, and then explicitly freed when it is no longer needed.

The polling is necessary for several reasons: to respond to requests on the admin bus; to check for incoming signals; to update outgoing signals. Therefore even a device that does not have signals must be polled. The user program must organize to have a timer or idle handler which can poll the device often enough. Polling interval is not extremely sensitive, but should be at least 100 ms or less. The faster it is polled, the faster it can handle incoming and outgoing signals.

The `mdev_poll` function can be blocking or non-blocking, depending on how you want your application to behave. It takes a number of milliseconds during which it should do some work, or 0 if it should check for any immediate actions and then return without waiting:

[1]Strictly this is only necessary for devices on the same computer, but port numbers are in abundance so we just allocate one per device to keep things consistant.

```
int mdev_poll( mapper_device md,
               int block_ms )
```

An example of calling it with non-blocking behaviour:

```
mdev_poll( my_device, 0 );
```

If your polling is in the middle of a processing function or in response to a GUI event for example, non-blocking behaviour is desired. On the other hand if you put it in the middle of a loop which reads incoming data at intervals or steps through a simulation for example, you can use `mdev_poll` as your "sleep" function, so that it will react to network activity while waiting.

It returns the number of messages handled, so optionally you could continue to call it until there are no more messages waiting. Of course, you should be careful doing that without limiting the time it will loop for, since if the incoming stream is fast enough you might never get anything else done!

Note that an important difference between blocking and non-blocking polling is that during the blocking period, messages will be handled immediately as they are received. On the other hand, if you use your own sleep, messages will be queued up until you can call poll(); stated differently, it will "time-quantize" the message handling. This is not necessarily bad, but you should be aware of this effect.

Since there is a delay before the device is completely initialized, it is sometimes useful to be able to determine this using `mdev_ready`. Only when `mdev_ready` returns non-zero is it valid to use the device's name.

Freeing the device

It is necessary to explicitly free the device at the end of your program. This not only frees memory, but also sends some messages to "politely" remove itself from the network.

An example of freeing a device:

```
mdev_free( my_device );
```

Signals

Now that we know how to create a device, poll it, and free it, we only need to know how to add signals in order to give our program some input/output functionality.

We'll start with creating a "sender", so we will first talk about how to update output signals.

Creating a signal

A signal requires a bit more information than a device, much of which is optional:

```
mapper_signal mdev_add_input( const char *name,
                              int length,
                              char type,
                              const char *unit,
                              void *minimum,
                              void *maximum,
                              mapper_signal_handler *handler,
                              void *user_data )

mapper_signal mdev_add_output( const char *name,
                               int length,
                               char type,
                               const char *unit,
                               void *minimum,
                               void *maximum )
```

The only *required* parameters here are the signal "length", its name, and data type. Signals are assumed to be vectors of values, so for usual single-valued signals, a length of 1 should be specified. A signal name should start with "/", as this is how it is represented in the OSC address. (One will be added if you forget to do this.) Finally, supported types are currently 'i' or 'f' (specified as characters in C, not strings), for `int` or `float` values, respectively.

The other parameters are not strictly required, but the more information you provide, the more the mapper can do some things automatically. For example, if `minimum` and `maximum` are provided, it will be possible to create linear-scaled connections very quickly. If `unit` is provided, the mapper will be able to similarly figure out a linear scaling based on unit conversion. (Centimeters to inches for example.)[2]

Notice that optional values are provided as `void*` pointers. This is because a signal can either be `float` or `int`, and your maximum and minimum values should correspond in type. So you should pass in a `float*` or `int*` by taking the address of a local variable.

Lastly, it is usually necessary to be informed when input signal values change. This is done by providing a function to be called whenever its value is modified by an incoming message. It is passed in the `handler` parameter, with context information to be passed to that function during callback in `user_data`.

[2]Currently automatic unit-based scaling is not a supported feature, but will be added in the future. You can take advantage of this future development by simply providing unit information whenever it is available. It is also helpful documentation for users.

An example of creating a “barebones” `int` scalar output signal with no unit, minimum, or maximum information:

```
mapper_signal outputA = mdev_add_output( dev, "/outA", 1, 'i', 0, 0, 0 );
```

An example of a `float` signal where some more information is provided:

```
float minimum = 0.0f;
float maximum = 5.0f;
mapper_signal sensor1_voltage = mdev_add_output( dev, "/sensor1", 1, 'f',
                                                 "V", &minimum, &maximum );
```

So far we know how to create a device and to specify an output signal for it. To recap, let’s review the code so far:

```
mapper_device my_sender = mdev_new("test_sender", 9000, 0);
mapper_signal sensor1_voltage = mdev_add_output( my_sender, "/sensor1",
                                                 1, 'f', "V",
                                                 &minimum, &maximum )

while (!done) {
    mdev_poll(my_sender, 50);
    ... do stuff ...
    ... update signals ...
}

mdev_free(my_sender);
```

Note that although you have a pointer to the mapper_signal structure, which was retuned by `mdev_add_output`, its memory “owned” by the device. In other words, you should not worry about freeing its memory—this will happen automatically when the device is destroyed. It is possible to retrieve a device’s inputs or outputs by name or by index at a later time using the functions `mdev_get_<input/output>_by_<name/index>`.

Updating signals

We can imagine the above program getting sensor information in a loop. It could be running on an network-enable ARM device and reading the ADC register directly, or it could be running on a computer and reading data from an Arduino over a USB serial port, or it could just be a mouse-controlled GUI slider. However it’s getting the data, it must provide it to *libmapper* so that it will be sent to other devices if that signal is mapped.

This is accomplished by the `msig_update` function:

```
void msig_update( mapper_signal sig, void *value )
```

As you can see, a `void*` pointer must be provided. This must point to a data structure identified by the signal's `length` and `type`. In other words, if the signal is a 10-vector of `int`, then `value` should point to the first item in a C array of 10 `int`s. If it is a scalar `float`, it should be provided with the address of a `float` variable.

A short-hand is provided for scalar signals of particular types:

```
void msig_update_int( mapper_signal sig, int value )

void msig_update_float( mapper_signal sig, float value )
```

So in the "sensor 1 voltage" example, assuming in "do stuff" we have some code which reads sensor 1's value into a float variable called `v1`, the loop becomes:

```
while (!done) {
    mdev_poll(my_device, 50);
    float v1 = read_sensor_1();
    msig_update_float( sensor1_voltage, v1 );
}
```

This is about all that is needed to expose sensor 1's voltage to the network as a mappable parameter. The *libmapper* GUI can now map this value to a receiver, where it could control a synthesizer parameter or change the brightness of an LED, or whatever else you want to do.

Signal conditioning

Most synthesizers of course will not know what to do with "voltage"—it is an electrical property that has nothing to do with sound or music. This is where *libmapper* really becomes useful.

Scaling or other signal conditioning can be taken care of *before* exposing the signal, or it can be performed as part of the mapping. Since the end user can demand any mathematical operation be performed on the signal, he can perform whatever mappings between signals as he wishes.

As a developer, it is therefore your job to provide information that will be useful to the end user.

For example, if sensor 1 is a position sensor, instead of publishing "voltage", you could convert it to centimeters or meters based on the known dimensions of the sensor, and publish a "/sensor1/position" signal instead, providing the unit information as well.

We call such signals “semantic”, because they provide information with more meaning than a relatively uninformative value based on the electrical properties of the sensing technqiue. Some sensors can benefit from low-pass filtering or other measures to reduce noise. Some sensors may need to be combined in order to derive physical meaning. What you choose to expose as outputs of your device is entirely application-dependent.

You can even publish both “/sensor1/position” and “/sensor1/voltage” if desired, in order to expose both processed and raw data. Keep in mind that these will not take up significant processing time, and *zero* network bandwidth, if they are not mapped.

Receiving signals

Now that we know how to create a sender, it would be useful to also know how to receive signals, so that we can create a sender-receiver pair to test out the provided mapping functionality.

As mentioned above, the `mdev_add_input` function takes an optional `handler` and `user_data`. This is a function that will be called whenever the value of that signal changes. To create a receiver for a synthesizer parameter “pulse width” (given as a ratio between 0 and 1), specify a handler when calling `mdev_add_input`. We’ll imagine there is some C++ synthesizer implemented as a class `Synthesizer` which has functions `setPulseWidth()` which sets the pulse width in a thread-safe manner, and `startAudioInBackground()` which sets up the audio thread.

Create the handler function, which is fairly simple,

```
void pulsewidth_handler ( mapper_signal msig,
                          void *v )
{
    mapper_db_signal props = msig_properties(msig);
    Synthesizer *s = (Synthesizer*) props->user_data;
    s->setPulseWidth(*(float*)v);
}
```

First, the pointer to the `Synthesizer` instance is extracted from the `user_data` pointer, then it is dereferenced to set the pulse width according to the value pointed to by `v`.

Then `main()` will look like,

```
void main()
{
    Synthesizer synth;
    synth.startAudioInBackground();

    float min_pw = 0.0f;
    float max_pw = 1.0f;

    mapper_device my_receiver = mdev_new("test_receiver", 9000, 0);

    mapper_signal synth_pulsewidth =
        mdev_add_input( my_receiver, "/synth/pulsewidth",
                        1, 'f', 0, &min_pw, &max_pw,
                        pulsewidth_handler, &synth );

    while (!done)
        mdev_poll(my_receiver, 50);

    mdev_free(my_receiver);
}
```

Publishing metadata

Things like device names, signal units, and ranges, are examples of metadata—information about the data you are exposing on the network.

libmapper also provides the ability to specify arbitrary extra metadata in the form of name-value pairs. These are not interpreted by *libmapper* in any way, but can be retrieved over the network. This can be used for instance to give a device X and Y information, or to perhaps give a signal some property like "reliability", or some category like "light", "motor", "shaker", etc.

Some GUI implementing a Monitor could then use this information to display information about the network in an intelligent manner.

Any time there may be extra knowledge about a signal or device, it is a good idea to represent it by adding such properties, which can be of any OSC-compatible type. (So, numbers and strings, etc.)

The property interface is through the functions,

```
void mdev_set_property( mapper_device dev,
                        const char *property,
                        lo_type type,
                        lo_arg *value );

void msig_set_property( mapper_signal sig,
                        const char *property,
                        lo_type type,
                    lo_arg *value );
```

As you can see, *libmapper* reuses the `lo_arg` union from the *liblo* OSC library, which can be used to hold any OSC-compatible value. The type of the `value` argument is specified by `type`, and can be any `lo_type` value; floats are `'f'` or `LO_FLOAT`, 32-bit integers are `'i'` or `LO_INT32`, and strings are `'s'` or `LO_STRING`, but you should consult the *liblo* documentation for more information.

For example, to store a `float` indicating the X position of a device, you can call it like this:

```
lo_arg x;
x.f = 12.5;
mdev_set_property( my_device, "x", 'f', &x);
```

In practice it is safe to cast to `lo_arg*`:

```
float x = 12.5;
mdev_set_property( my_device, "x", 'f', (lo_arg*)&x);
```

To specify strings, it is necessary to perform such a cast, since the `lo_arg*` you provide should actually point to the beginning of the string:

```
char *sensingMethod = "resistive";
msig_set_property( sensor1, "sensingMethod",
                   's', (lo_arg*)sensingMethod);
```

In general you can use any property name not already in use by the device or signal data structure. Reserved words for signals are:

```
device_name, direction, length, max, min, name, type, unit, user_data;
```

for devices, they are:

```
host, port, name, user_data.
```

By the way, if you query or set signal properties using these keywords, you will get or modify the same information that is available directly from the `mapper_db_signal` data structure. Therefore this can provide a unified string-based method for accessing any signal property:

```
mapper_db_signal *props = msig_properties( sensor1 );
lo_type type;
const lo_arg *value;
mapper_db_signal_property_lookup( props, "sensingMethod", &type, &value );
```

Primarily this is an interface meant for network monitors, but may come in useful for an application implementing a device.