DEVELOPER SUMMIT 2025

PX4 Device Manifest

Nov 2025 | Niklas Hauser & Alexander Lerach | Auterion AG

Who We Are

Niklas Hauser

'10

'13

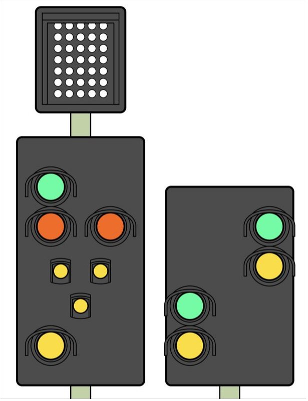
roboterclub.rwth-aachen.de

modm.io

'15

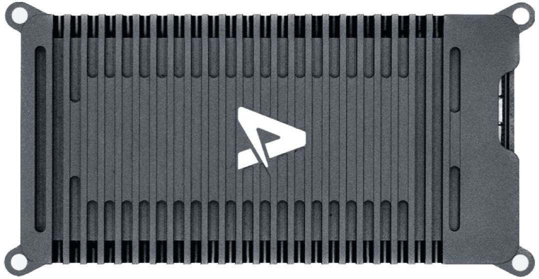
uVisor

'18



salkinium.com/elva

'23



PX4 Autopilot

Alexander Lerach

Vita

- VECTOR Safe RTOS (Cortex-M, PPC, TriCore, ...)
- Embedded security
- Auterion PX4 embedded, manufacturing

PX4

- Everything low level (adding boards, drivers, FLASH/CPU usage optimization)
- Debugging/fixing NuttX (H7 UART TX DMA getting stuck, ...)
- Occasionally mavlink / uXRCE-DDS client

Motivation

The Problem

Good case

Mavlink Console

Mavlink Console provides a connection to the vehicle's system shell.

```
bmm350 status
INFO  [SPI_I2C] Running on I2C Bus 4, Address 0x14
bmm350: reset: 1 events
bmm350: bad read: 0 events
bmm350: self test failed: 0 events
```

Driver starts normally:

- Device responded and successfully configured.
- uORB topic is published.
- Commander is happy.

⇒ Two `sensor_mag` (internal / external) topics.

Bad case

Mavlink Console

Mavlink Console provides a connection to the vehicle's system shell.

```
bmm350 status
INFO  [SPI_I2C] Not running
```

Driver does **not** start:

- I2C interference: driver not robust.
- Power issues: cabling not robust.
- Component failures: sensor not robust.

⇒ Fallback to internal compass, thus **silent failure!**

The Cause

Opportunistic quiet driver starting

```
hmc5883 -T -X -q start
iis2mdc -X -q start
ist8308 -X -q start
ist8310 -X -q start
if ! lis3mdl -X -q start
then
	lis3mdl -X -q -a 0x1c start
fi
qmc5883l -X -q start
qmc5883p -X -q start
rm3100 -X -q start
bmm350 -X -q start
iis2mdc -X -q start
```

One-time or fixed-count probing

```
int BMM350::probe()
{
	for (int i = 0; i < 3; i++) {
		uint8_t chip_id;

		if (PX4_OK == RegisterRead(Register::CHIP_ID, &chip_id)) {
			PX4_DEBUG("CHIP_ID: 0x%02hhX", chip_id);

			if (chip_id == chip_identification_number) {
				return PX4_OK;
			}
		}
	}

	return PX4_ERROR;
}
```

Solution Requirements

Ease of use

- Developers need to access the manifest data via CLI
- Integrators need to setup their airframe via the file system
- Pilots need to manage flight configuration via QGC

Configurable

- Need to encode different types of data for different drivers.
- Starting multiple drivers must allow for multiple instances of the same parameter type.

Lightweight

- Low resource usage: binary size and CPU utilization

Backward compatible

- Preserve as much of the existing user configuration as possible

Basic Idea

Let the user to state which drivers to start using a configuration system:

- Specify common communication settings: which I2C/SPI/UART bus id.
- Specify device specific settings: I2C address, rotation, sensor ranges, calibration.
- Specify multiple instances of settings when using multiple devices.
- Store these settings in non-volatile memory.

Can we use PX4 parameters for this?

- Already supported by MAVLink and DroneCAN transport protocols.
- GUI support in QGC, AMC, DroneCAN, and CLI support in NSH and airframe files.
- Widely used and known for storing setup specific configuration settings.
- BUT: inefficient use of metadata and storage, cannot instantiate multiple, limited types.

⇒ Autostart drivers based on instanced parameters and supervise their health!

Implementation

Parameter Structures

INA238 description:

```
uint4 p_version
Bus bus
uint10 current
float16 shunt
```

General bus description (`Bus`):

```
@union
I2c i2c
Spi spi
```

General I2C description (`I2c`):

```
uint4 p_version
uint4 bus_id
uint7 address
```

Much more powerful parameter structure:

- Allow more types than int32, float, bitmask.
- Parameters can have any length.
- DSDL allows for reusable standard blocks.
- Encode a version for easier translation support.

User experience is improved:

- User configures attached hardware in QGC or airframe files.
- PX4 now knows which drivers start.
- Arming depends on all expected drivers working.

Parameter Serialization

Scalar fields serialization example

Byte index	0								1								2								3							
Bit index	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Bit position	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Encoded bit values	1	1	0	1	1	0	1	0	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Value index	Value to encode	Target bit length	Binary representation before truncation	
0	0xBEDA	12	1011'1110 1101'1010	4 most significant bits are truncated; the value is then converted to little endian representation
1	-1	3	111	Two's complement
2	-5	4	1011	Two's complement
3	-1	2	11	Two's complement
4	0x88	4	1000'1000	4 most significant bits are truncated
N/A	Alignment	N/A	All zero	The encoded message must be byte aligned

Use existing technology developed by DroneCAN:

- Already have a DSDL, do not reinvent the wheel.
- Saves storage using packed bitfields.
- Allows to store complex composite data.
- Order of fields is preserved allowing prepending new fields for easier translation.

Using libcanard is much more efficient than libuavcan:

- Only need a small subset of the actual functionality offered by libuavcan.
- Need to patch only one function (descattering).
- Smaller binary size due to not using C++ templates for every type.

Parameter Instances

How to start two drivers using the same parameter?
We must encode different I2C busses, addresses, configuration twice somewhere.

`INA238#0` is a parameter of instance 0
`INA238#1` is a parameter of instance 1

`INA238_SHARED` is shared between all instances

Create an instance: `param add INA238#0`
Remove an instance: `param rm  INA238#0`

The same parameters need to be defined multiple times to start multiple drivers.

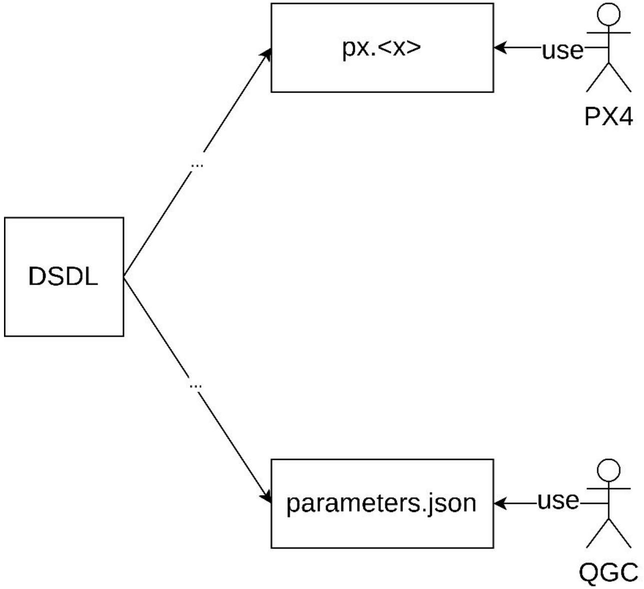
Instance is a suffix to the parameter name:

- Parameter instances can be added and deleted.
- Instance numbers are stable.

Drivers specify maximum number of instances:

- Allows reserving space in static memory.
- QGCs know how many instances to expect.

Parameter Architecture




```c
struct px_Ina238 {
#if defined(__cplusplus) && defined(DRONECAN_CXX_WRAPPERS)
    using cxx_iface = px_Ina238_cxx_iface;
#endif
    uint8_t p_version;
    struct px_Bus bus;
    uint16_t current;
    float shunt;
};
```

```json
{
        "category": "Standard",
        "default": 0,
        "group": "Sensors",
        "longDesc": "For systems a INA238 Power Monitor, this should be
        "name": "INA238",
        "rebootRequired": true,
        "shortDesc": "Enable INA238 Power Monitor",
        "type": "Int32",
        "max_instances": 4,
        "fields": [
                { "name": "p_version", "type": "uint4" },
                { "name": "Bus Tag", "type": "uint3" },
                { "name": "p_version", "type": "uint4" },
                { "name": "Bus Id", "type": "uint4" },
                { "name": "Address", "type": "uint7" },
                { "name": "Current", "type": "uint10" },
                { "name": "Shunt", "type": "float16" }
        ]
},
```

Using Parameters in Code

Reading parameters:

```
struct px_Ina238 ina238_data;
int ret = load_and_decode_param<px_Ina238>(px4::params::INA238, 0, ina238_data);
```

Writing parameters:

```
int ret = store_and_encode_param<px_Ina238>(px4::params::INA238, 0, ina238_data);
```

Using the generated structs:

```
PX4_INFO("bus_type: %d, address: %d",
               ina238_data.bus.union_tag, ina238_data.bus.i2c.address);
```

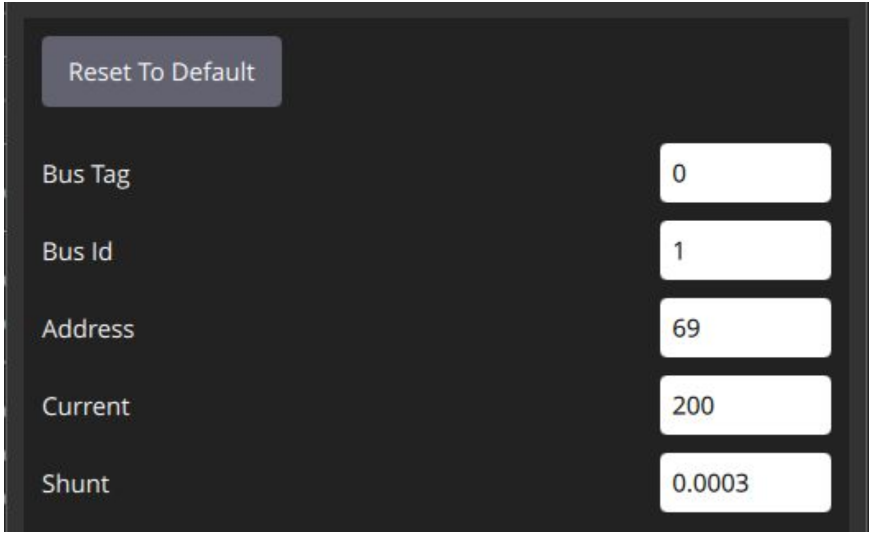
Using Parameters in Airframe Files

Have an index based access via the CLI

```
param add INA238#0
param set-default INA238#0[3] 1
param set-default INA238#0[4] 0x45
param set-default INA238#0[5] 200
param set-default INA238#0[6] 0.0003
```

- To save FLASH a member name based access is not implemented!
- User does not need to set versions, done automatically by the generated code.

Using Parameters in QGC



Access parameters using metadata in parameters.json:

- Can detect encoded parameters as they contain additional field description.
- Actual encoding/decoding can be done the same way as in PX4.

Platform-independent by using the DroneCAN serialization rules!

Autostarting Drivers

```
ina226 auto
ina228 auto
ina238 auto
```

finds `INA238#1`, `INA238#2`

```
for (auto config :
     param_find_instances(params::INA238)) {
  cli.i2c_address = config.bus.i2c.address;
  cli.requested_bus = config.bus.i2c.bus_id;
  cli.keep_running = true;
  cli.param = config;
  ThisDriver::module_start(cli, iterator);
}
```

Compile-time changes:

- Every driver specifies in CMakeLists.txt if they support an autostart and in what order.
- Build system generates a startup script that just calls the drivers with `auto` command.

Runtime changes:

- Driver main function reads the instance parameters and translates into the drivers starting.
- Driver gets parameter instance and reads further config from it directly.
- Updating the parameter instance at runtime can be read by the driver directly.

Monitoring Drivers

```
Health Driver::health() {
    if (running && errors == 0)
        return Health::Nominal;
    return Health::Critical;
}


if (i2c_readout() != PX4_OK) {
    perf_count(_bad_transfer_perf);
}
```

Arming checks should be delegated to drivers:

- Each driver registers themselves with the commander during startup.
- The commander can query them at any time for their status.
- Less spaghetti code in Commander!

Health monitoring is mostly implemented:

- Every driver implements perf counters.
- But: Do not always deliver useful information.
- But: perf counters are only streamed to ulog before and after arming. Not helpful in a crash.

⇒ Cleanup perf counters and stream to ulog.

The Future

“Backward Compatible” Parameters

```
INA238#1.bus_type     -> INA238_1_BUS_TYPE
INA238#1.i2c.bus_id   -> INA238_1_I2C_BUS_ID
INA238#1.i2c.address  -> INA238_1_I2C_ADDRESS
```

Mavlink limits parameter names to 16-chars:

```
INA238_1_BUS_TYP
INA238_1_I2C_BUS
INA238_1_I2C_ADD
```

Generate unique short handle from index:

```
INA238#1.bus_type     -> INA238_1A
INA238#1.i2c.bus_id   -> INA238_1B
INA238#1.i2c.address  -> INA238_1C
```

Destructure the subfields into separate parameters:

- Concatenate instance and subfield name.
- Map subfields to native types.
  - Integers → int32
  - floating points → float32.
  - booleans → bitmask.

This works for DroneCAN, but not for MAVLink:

- MAVLink has a parameter name limit of 16 chars.
- Precision can be lost: float16 vs float32.
- MAVLink can only send 32-bits per parameter.
- float64 and >int32 unsupported.

⇒ Add index of subfield to instance as letter.
⇒ Limit subfields to ≤32-bit values.

Next Steps

Non-breaking preparation:

- Introduce DSDL for structured parameters, add runtime API, and CLI tools.
- Implement automatic driver starting and health monitoring.
- Add structured parameter support to QGC and AMC.
- Update documentation and add upgrade path guide.

Breaking roll out:

- Update small set of drivers after internal dogfooding. **Parameters need to be updated!**
- Update more drivers carefully incorporating user feedback.

Limitations:

- Subfields must be limited ≤32bit for backward compatibility on Mavlink.
- There will be **no more auto detection of external sensors** by default!

Thanks for your attention! Questions?

PX4 autopilot DEVELOPER SUMMIT 2025