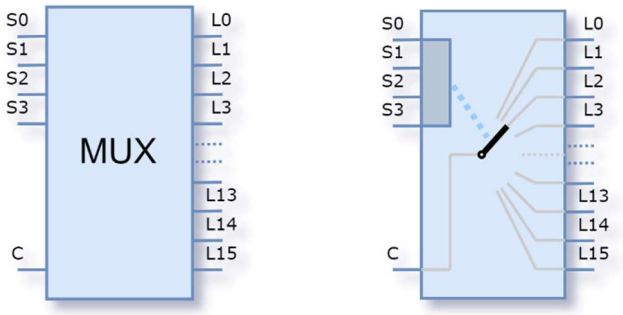
Generics about multiplexers (MUX) in MobiFlight

What MUXes are

- One ***common*** terminal, connected to one of 8/16[1] ***switched*** terminals
- Four ***selector*** lines determine the ***position*** (or ***channel***), i.e. which one of the switched terminals is connected to the common.
  All other, unselected, switched terminals remain unconnected.
- NO POLARITY: terminals can be both inputs and outputs. If the common terminal is an input, then switched terminals are outputs, and vice-versa[2].
- Think of the MUX *exactly* as "electrically controlled" rotary switches.



How MUXes can be used: direct inputs (technical notes)

- Most straight application: **multiplier** of **direct inputs**.
- Switched terminals are directly connected to input lines, while the common terminal is connected to an MCU pin configured as input.
- Inputs can be **digital** or **analog**[3].
- One single MCU pin can read 16 inputs
- This configuration is comparable in function (not in operation) to a shift register: get 16 inputs sequentially into a single pin.
  The advantage of MUXes in this capacity is that channels are selectable in any order and e.g. one of them can be left connected for as long as required, while in a SR only a "snapshot" of the status is taken and all inputs must be shifted in in sequential order.
- It has to be noted that the symmetrical application, as a multiplier of **outputs**, is NOT possible. Switched terminals remain connected to the common only for the brief time in which they are selected (which is all that is needed for reading an input), but otherwise they are disconnected, therefore an output could not be sustained.[4]

[1] *From now on, we will always assume 16 for simplicity*.
[2] The direction is basically determined by the common terminal, and is always the same. To be precise, it is possible to manage different directions (and signal types) for different positions, but – in connection with an MCU – the [MCU pin connected to the] common terminal should be reconfigured for every position, which, depending on the application, is often practically unfeasible.
[3] *From now on, we will assume digital inputs for simplicity*.
[4] It is however possible to use MUXes for output signals that e.g. drive peripherals; see further down in this document.

In this case, the "corresponding" version using a shift register has the advantage that… it works; once the status of the outputs is set, it is latched and therefore maintained.

How MUXes can be used: signal routing (technical notes)

- An evolution of the simple application is the routing of lines used to drive a "complex" component, like e.g. a display driver, but also a shift register (input or output).
- The MUX becomes in this case a multiplier of **components** (or **peripherals**)
- Some requirements:
    - the attached components should have the same, or at least compatible, electrical interface types (i.e. they can be driven with the same set of signals)[5]
    - it should be possible to use the same lines to drive all attached components, except for just one of the signals, for which there will be a different line for each component, routed through the multiplexer[6].
      For instance, for a set of LED drivers (each requiring a set of three signals CLK, LAT and DTA) there might be two common signal lines (e.g. CLK and DTA) common to all components, and the individual DTAn lines could be routed by the MUX to a single DTA_OUT on the MCU. Several choices of common lines are normally possible[7].
- Furthermore, we could even add devices of different kinds to the same multiplexer, as long as they have the same direction (i.e. inputs or outputs).

[5] For instance, it would be possible to attach some MAX7219 display drivers together with some output shift registers to a same MUX. The CLK, LAT and DTA signals have the same configuration and meaning for both component types; the protocol for sending the respective data is different, but to the hardware this is irrelevant and will be taken care of by the software drivers. More on this later.

[6] Important note regarding Mobiflight: *this implies that signal lines for different devices of the same / compatible types can be made common, because either pre-assigned or at least sharable.*

[7] This is a factor that must necessarily be taken into account if compatibility with other systems is desired. For instance, the *SimVim/RealSimControl* system – which relies heavily on multiplexers – shares the LAT and DTA (out) signals, and it routes the CLK signal to the individual CLK inputs of the components.
This is different from the most usual choice, which is to route the LAT signal instead.

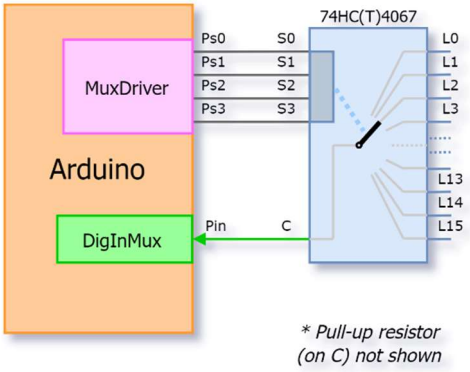
Implementation and practical applications

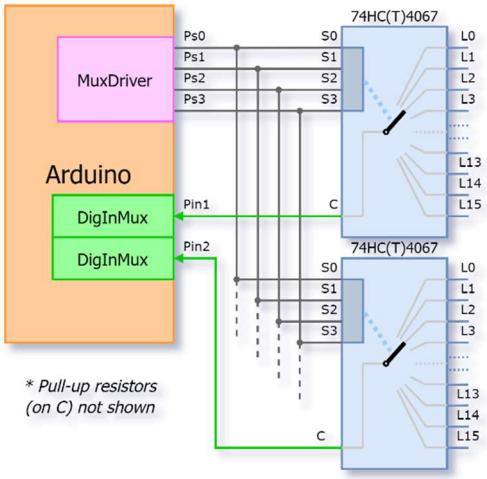
Digital input blocks

- The implementation is as follows:
    - a device (**MuxDriver**) that handles the four selector (output) lines and the single (input) data line
    - a device (**DigInMux**) that "collects" the input values presented to the Arduino pin and manages them as buttons

  and of course a component such as a 74HCT4067 multiplexer.

- The purpose of the **MuxDriver** is to setup the selector lines according to the value requested by any other device. The corresponding configuration in the Connector UI is particular, also because *the MuxDriver would only exist in a single instance*[8].
- When polled, the **DigInMux** device would sequentially span the 16 values for the selector lines and, after each value, read the input pin, collecting the values to form the state word (and triggering the appropriate events according to the changes).
- When several muxes are used, the set of selector lines is connected to all MUXes in parallel.





- The only difference in the firmware code from a "regular" device would be that, in order to select a channel, instead of calling a method of the same object the *DigInMux* device would call an external function (passed as a callback), which would in turn make use of the *MuxDriver*.
- It can be observed that this device is very similar in function / purpose to e.g. a digital input Shift Register block. In fact, except for the input polling procedure, most of the operation is exactly the same.
  Other similarities with Input Shift Registers:
    - on the Connector side, the DigInMux is very easy to manage, because it just sends Button-like events; the "major" effort is simply in adding configuration panels and internal registration, much like any other device.

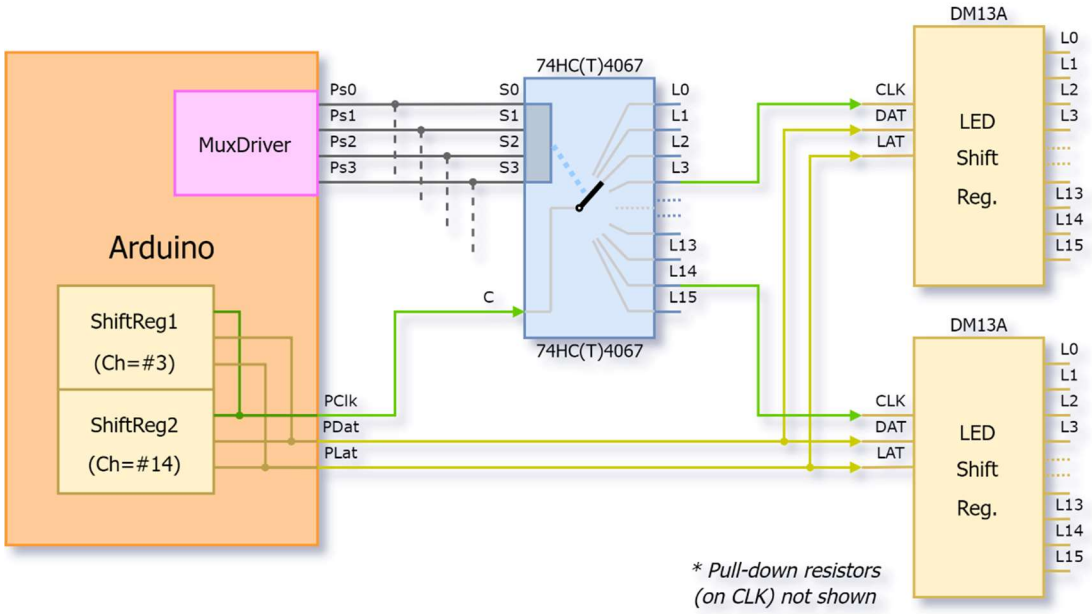
[8] Even if more than one instance could theoretically exist, this would bring no practical advantage whatsoever.

- given the sequential, time-scheduled nature of data polling, ***multiplexers – unless specially managed - are NOT suitable for fast or time-sensitive inputs like encoders***.

- An **analog** input block could be also implemented in almost identical way.

Multi-device routing

- Once a common MuxDriver object is in place, it can be easily used by other existing devices, making them MUX-compatible with very little effort and without changing the way they operate.
- For instance, let's imagine a single LED driver (an input shift register would do as well) connected to pin N. If we connect it by placing it behind a multiplexer, e.g. on channel M, the device and its configuration would look almost *exactly* the same, except we just specify that it corresponds to channel M.
- This way, just by adding the multiplexer component on the hardware part we are able to connect not just one, but up to 16 LED drivers, still using the same one pin, with almost no difference in configuration.



Important considerations about multiplexed devices

- In the previous schematic, it is apparent that – for a reasonably pin-efficient architecture – the devices to be multiplexed that have a common interface must be able to share driver pins (all but the one which is routed by the Mux). In Mobiflight, this implies that the **issue of having sharable lines** would have to be resolved beforehand.