XPLDevices

Generated by Doxygen 1.9.6

1 XPLDevices 1

2 Hierarchical Index 3

2.1 Class Hierarchy . . . 3

3 Class Index 5

3.1 Class List . . . 5

4 File Index 7

4.1 File List . . . 7

5 Class Documentation 9

5.1 AnalogIn Class Reference . . . 9

5.1.1 Detailed Description . . . 9

5.1.2 Constructor & Destructor Documentation . . . 10

5.1.2.1 AnalogIn() [1/2] . . . 10

5.1.2.2 AnalogIn() [2/2] . . . 10

5.1.3 Member Function Documentation . . . 10

5.1.3.1 calibrate() . . . 10

5.1.3.2 handle() . . . 11

5.1.3.3 raw() . . . 11

5.1.3.4 setRange() . . . 11

5.1.3.5 setScale() . . . 12

5.1.3.6 value() . . . 12

5.2 Button Class Reference . . . 12

5.2.1 Detailed Description . . . 13

5.2.2 Member Enumeration Documentation . . . 14

5.2.2.1 anonymous enum . . . 14

5.2.3 Constructor & Destructor Documentation . . . 14

5.2.3.1 Button() [1/2] . . . 14

5.2.3.2 Button() [2/2] . . . 14

5.2.4 Member Function Documentation . . . 15

5.2.4.1 engaged() . . . 15

5.2.4.2 getCommand() . . . 15

5.2.4.3 handle() [1/2] . . . 15

5.2.4.4 handle() [2/2] . . . 15

5.2.4.5 handleXP() [1/2] . . . 16

5.2.4.6 handleXP() [2/2] . . . 16

5.2.4.7 pressed() . . . 16

5.2.4.8 processCommand() . . . 17

5.2.4.9 released() . . . 17

5.2.4.10 setCommand() [1/2] . . . 17

5.2.4.11 setCommand() [2/2] . . . 17

5.2.5 Member Data Documentation . . . 18

5.2.5.1 _cmdPush . . . 18
5.2.5.2 _mux . . . 18
5.2.5.3 _pin . . . 18
5.2.5.4 _state . . . 18
5.2.5.5 _transition . . . 18
5.3 DigitalIn_ Class Reference . . . 19
5.3.1 Detailed Description . . . 19
5.3.2 Constructor & Destructor Documentation . . . 19
5.3.2.1 DigitalIn_() . . . 19
5.3.3 Member Function Documentation . . . 19
5.3.3.1 addMux() . . . 19
5.3.3.2 getBit() . . . 20
5.3.3.3 handle() . . . 20
5.3.3.4 setMux() . . . 20
5.4 Encoder Class Reference . . . 21
5.4.1 Detailed Description . . . 22
5.4.2 Constructor & Destructor Documentation . . . 22
5.4.2.1 Encoder() [1/2] . . . 22
5.4.2.2 Encoder() [2/2] . . . 22
5.4.3 Member Function Documentation . . . 23
5.4.3.1 down() . . . 23
5.4.3.2 engaged() . . . 23
5.4.3.3 getCommand() . . . 23
5.4.3.4 handle() . . . 24
5.4.3.5 handleXP() . . . 24
5.4.3.6 pos() . . . 24
5.4.3.7 pressed() . . . 24
5.4.3.8 processCommand() . . . 25
5.4.3.9 released() . . . 25
5.4.3.10 setCommand() [1/4] . . . 25
5.4.3.11 setCommand() [2/4] . . . 25
5.4.3.12 setCommand() [3/4] . . . 26
5.4.3.13 setCommand() [4/4] . . . 26
5.4.3.14 up() . . . 26
5.5 LedShift Class Reference . . . 27
5.5.1 Detailed Description . . . 27
5.5.2 Constructor & Destructor Documentation . . . 27
5.5.2.1 LedShift() . . . 27
5.5.3 Member Function Documentation . . . 28
5.5.3.1 handle() . . . 28
5.5.3.2 set() . . . 28
5.5.3.3 set_all() . . . 28

5.5.3.4 setAll() . . . 28
5.5.3.5 setPin() . . . 29
5.6 RepeatButton Class Reference . . . 29
5.6.1 Detailed Description . . . 31
5.6.2 Constructor & Destructor Documentation . . . 31
5.6.2.1 RepeatButton() [1/2] . . . 31
5.6.2.2 RepeatButton() [2/2] . . . 31
5.6.3 Member Function Documentation . . . 32
5.6.3.1 handle() [1/2] . . . 32
5.6.3.2 handle() [2/2] . . . 32
5.6.3.3 handleXP() [1/2] . . . 32
5.6.3.4 handleXP() [2/2] . . . 32
5.6.4 Member Data Documentation . . . 33
5.6.4.1 _delay . . . 33
5.6.4.2 _timer . . . 33
5.7 ShiftOut Class Reference . . . 33
5.7.1 Detailed Description . . . 34
5.7.2 Constructor & Destructor Documentation . . . 34
5.7.2.1 ShiftOut() . . . 34
5.7.3 Member Function Documentation . . . 34
5.7.3.1 handle() . . . 34
5.7.3.2 setAll() . . . 34
5.7.3.3 setPin() . . . 35
5.8 Switch Class Reference . . . 35
5.8.1 Detailed Description . . . 36
5.8.2 Constructor & Destructor Documentation . . . 36
5.8.2.1 Switch() [1/2] . . . 36
5.8.2.2 Switch() [2/2] . . . 36
5.8.3 Member Function Documentation . . . 37
5.8.3.1 getCommand() . . . 37
5.8.3.2 handle() . . . 37
5.8.3.3 handleXP() . . . 37
5.8.3.4 off() . . . 38
5.8.3.5 on() . . . 38
5.8.3.6 processCommand() . . . 38
5.8.3.7 setCommand() [1/4] . . . 38
5.8.3.8 setCommand() [2/4] . . . 39
5.8.3.9 setCommand() [3/4] . . . 39
5.8.3.10 setCommand() [4/4] . . . 39
5.8.3.11 value() . . . 40
5.9 Switch2 Class Reference . . . 40
5.9.1 Detailed Description . . . 41

5.9.2 Constructor & Destructor Documentation . . . 41
5.9.2.1 Switch2() [1/2] . . . 41
5.9.2.2 Switch2() [2/2] . . . 42
5.9.3 Member Function Documentation . . . 42
5.9.3.1 getCommand() . . . 42
5.9.3.2 handle() . . . 42
5.9.3.3 handleXP() . . . 42
5.9.3.4 off() . . . 43
5.9.3.5 on1() . . . 43
5.9.3.6 on2() . . . 43
5.9.3.7 processCommand() . . . 43
5.9.3.8 setCommand() [1/4] . . . 43
5.9.3.9 setCommand() [2/4] . . . 44
5.9.3.10 setCommand() [3/4] . . . 44
5.9.3.11 setCommand() [4/4] . . . 45
5.9.3.12 value() . . . 45
5.10 Timer Class Reference . . . 45
5.10.1 Detailed Description . . . 46
5.10.2 Constructor & Destructor Documentation . . . 46
5.10.2.1 Timer() . . . 46
5.10.3 Member Function Documentation . . . 46
5.10.3.1 count() . . . 46
5.10.3.2 elapsed() . . . 47
5.10.3.3 getTime() . . . 47
5.10.3.4 setCycle() . . . 47
5.11 XPLDirect Class Reference . . . 48
5.11.1 Detailed Description . . . 48
5.11.2 Constructor & Destructor Documentation . . . 48
5.11.2.1 XPLDirect() . . . 48
5.11.3 Member Function Documentation . . . 48
5.11.3.1 allDataRefsRegistered() . . . 49
5.11.3.2 begin() . . . 49
5.11.3.3 commandEnd() . . . 49
5.11.3.4 commandStart() . . . 49
5.11.3.5 commandTrigger() [1/2] . . . 49
5.11.3.6 commandTrigger() [2/2] . . . 50
5.11.3.7 connectionStatus() . . . 50
5.11.3.8 datarefsUpdated() . . . 50
5.11.3.9 hasUpdated() . . . 50
5.11.3.10 registerCommand() . . . 50
5.11.3.11 registerDataRef() [1/5] . . . 51
5.11.3.12 registerDataRef() [2/5] . . . 51

5.11.3.13 registerDataRef() [3/5] . . . 51
5.11.3.14 registerDataRef() [4/5] . . . 51
5.11.3.15 registerDataRef() [5/5] . . . 52
5.11.3.16 sendDebugMessage() . . . 52
5.11.3.17 sendResetRequest() . . . 52
5.11.3.18 sendSpeakMessage() . . . 52
5.11.3.19 xloop() . . . 52

6 File Documentation **53**

6.1 Direct inputs/main.cpp . . . 53
6.2 MUX inputs/main.cpp . . . 54
6.3 AnalogIn.h . . . 54
6.4 Button.h . . . 55
6.5 DigitalIn.h . . . 56
6.6 Encoder.h . . . 57
6.7 LedShift.h . . . 58
6.8 ShiftOut.h . . . 58
6.9 Switch.h . . . 58
6.10 Timer.h . . . 60
6.11 XPLDevices.h . . . 60
6.12 XPLDirect.h . . . 60
6.13 AnalogIn.cpp . . . 62
6.14 Button.cpp . . . 64
6.15 DigitalIn.cpp . . . 65
6.16 Encoder.cpp . . . 67
6.17 LedShift.cpp . . . 68
6.18 ShiftOut.cpp . . . 69
6.19 Switch.cpp . . . 70
6.20 Timer.cpp . . . 73
6.21 XPLDirect.cpp . . . 73

Index **81**

Chapter 1

XPLDevices

This Repository hosts the enhanced XPLDevices library built on top of XPLDirect by Curiosity Workshop. Please visit our Discord: https://discord.gg/gzXetjEST4

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

AnalogIn 9
Button 12
    RepeatButton 29
DigitalIn_ 19
Encoder 21
LedShift 27
ShiftOut 33
Switch 35
Switch2 40
Timer 45
XPLDirect 48

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

AnalogIn
: Class to encapsulate analog inputs 9

Button
: Class for a simple pushbutton with debouncing and XPLDirect command handling. Supports start and end of commands so XPlane can show the current Button status 12

DigitalIn_
: Class to encapsulate digital inputs from 74HC4067 and MCP23017 input multiplexers, used by all digital input devices. Scans all expander inputs into internal process data image 19

Encoder
: Class for rotary encoders with optional push functionality. The number of counts per mechanical notch can be configured for the triggering of up/down events 21

LedShift
: Class to encapsulate a DM13A LED driver IC 27

RepeatButton
: Class for a simple pushbutton with debouncing and XPLDirect command handling, supports start and end of commands so XPlane can show the current Button status. When button is held down cyclic new pressed events are generated for auto repeat function 29

ShiftOut
: Class to encapsulate a DM13A LED driver IC 33

Switch
: Class for a simple on/off switch with debouncing and XPLDirect command handling 35

Switch2
: Class for an on/off/on switch with debouncing and XPLDirect command handling 40

Timer
: Priovide a simple software driven timer for general purpose use 45

XPLDirect 48

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Direct inputs/main.cpp . . . 53
MUX inputs/main.cpp . . . 54
AnalogIn.h . . . 54
Button.h . . . 55
DigitalIn.h . . . 56
Encoder.h . . . 57
LedShift.h . . . 58
ShiftOut.h . . . 58
Switch.h . . . 58
Timer.h . . . 60
XPLDevices.h . . . 60
XPLDirect.h . . . 60
AnalogIn.cpp . . . 62
Button.cpp . . . 64
DigitalIn.cpp . . . 65
Encoder.cpp . . . 67
LedShift.cpp . . . 68
ShiftOut.cpp . . . 69
Switch.cpp . . . 70
Timer.cpp . . . 73
XPLDirect.cpp . . . 73

Chapter 5

Class Documentation

5.1 AnalogIn Class Reference

Class to encapsulate analog inputs.

`#include <AnalogIn.h>`

Public Member Functions

- AnalogIn (uint8_t pin, Analog_t type)

  Setup analog input.
- AnalogIn (uint8_t pin, Analog_t type, float timeConst)

  Setup analog input with low pass filter.
- void handle ()

  Read analog input, scale value and perform filtering, call once per sample loop.
- float value ()

  Return actual value.
- int raw ()

  Return raw value.
- void calibrate ()

  Perform calibration for bipolar input, current position gets center and min/max ranges are adapted to cover +/- scale. Usage is only sensible for small deviations like for joysticks.
- void setRange (uint16_t min, uint16_t max)

  Set subrange for mechanically limited potentiometers and limit output value to this range. for bipolar applications the offset is set to the center value of this range.
- void setScale (float scale)

  Set output scale for max input range. Default scale is 1.0.

5.1.1 Detailed Description

Class to encapsulate analog inputs.

Definition at line 14 of file AnalogIn.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 AnalogIn() [1/2]

```
AnalogIn::AnalogIn (
            uint8_t pin,
            Analog_t type )
```

Setup analog input.

Parameters

pin	Arduino pin number to use
type	unipolar (0..scale) or bipolar (-scale..scale) range.

Definition at line 7 of file AnalogIn.cpp.

5.1.2.2 AnalogIn() [2/2]

```
AnalogIn::AnalogIn (
            uint8_t pin,
            Analog_t type,
            float timeConst )
```

Setup analog input with low pass filter.

Parameters

pin	Arduino pin number to use
type	unipolar (0..1) or bipolar (-1..1)
timeConst	Filter time constant (t_filter/t_sample)

Definition at line 27 of file AnalogIn.cpp.

5.1.3 Member Function Documentation

5.1.3.1 calibrate()

```
void AnalogIn::calibrate ( )
```

Perform calibration for bipolar input, current position gets center and min/max ranges are adapted to cover +/- scale. Usage is only sensible for small deviations like for joysticks.

Definition at line 46 of file AnalogIn.cpp.

5.1.3.2 handle()

```
void AnalogIn::handle ( )
```

Read analog input, scale value and perform filtering, call once per sample loop.

Definition at line 35 of file AnalogIn.cpp.

5.1.3.3 raw()

```
int AnalogIn::raw ( )
```

Return raw value.

Returns

Read raw analog input and compensate bipolta offset

Definition at line 41 of file AnalogIn.cpp.

5.1.3.4 setRange()

```
void AnalogIn::setRange (
            uint16_t min,
            uint16_t max )
```

Set subrange for mechanically limited potentiometers and limit output value to this range. for bipolar applications the offset is set to the center value of this range.

Parameters

min	Minimum value in raw digits (maps to Zero)
max	Maximum value in raw digits (maps to Scale)

Definition at line 61 of file AnalogIn.cpp.

5.1.3.5 setScale()

```
void AnalogIn::setScale (
            float scale )
```

Set output scale for max input range. Default scale is 1.0.

Parameters

scale	Scale of output value for maximum range

Definition at line 81 of file AnalogIn.cpp.

5.1.3.6 value()

```
float AnalogIn::value ( )  [inline]
```

Return actual value.

Returns

Actual, filtered value as captured with handle()

Definition at line 33 of file AnalogIn.h.

The documentation for this class was generated from the following files:

- AnalogIn.h
- AnalogIn.cpp

5.2 Button Class Reference

Class for a simple pushbutton with debouncing and XPLDirect command handling. Supports start and end of commands so XPlane can show the current Button status.

```
#include <Button.h>
```

Public Member Functions

- Button (uint8_t mux, uint8_t muxpin)

  Constructor, set mux and pin number.
- Button (uint8_t pin)

  Constructor, set digital input without mux.
- void handle ()

  Handle realtime. Read input and evaluate any transitions.
- void handle (bool input)

  Handle realtime. Read input and evaluate any transitions.
- void handleXP ()

  Handle realtime and process XPLDirect commands.
- void handleXP (bool input)

  Handle realtime and process XPLDirect commands.
- bool pressed ()

  Evaluate and reset transition if button pressed down.
- bool released ()

  Evaluate and reset transition if button released.
- bool engaged ()

  Evaluate status of Button.
- void setCommand (int cmdPush)

  Set XPLDirect command for Button events.
- void setCommand (XPString_t *cmdNamePush)

  Set XPLDirect command for Button events.
- int getCommand ()

  Get XPLDirect command associated with Button.
- void processCommand ()

  Process all transitions and active transitions to XPLDirect

Protected Types

- enum { **transNone** , **transPressed** , **transReleased** }

Protected Attributes

- uint8_t _mux
- uint8_t _pin
- uint8_t _state
- uint8_t _transition
- int _cmdPush

5.2.1 Detailed Description

Class for a simple pushbutton with debouncing and XPLDirect command handling. Supports start and end of commands so XPlane can show the current Button status.

Definition at line 8 of file Button.h.

5.2.2 Member Enumeration Documentation

5.2.2.1 anonymous enum

```
anonymous enum  [protected]
```

Definition at line 65 of file Button.h.

5.2.3 Constructor & Destructor Documentation

5.2.3.1 Button() [1/2]

```
Button::Button (
            uint8_t mux,
            uint8_t muxpin )
```

Constructor, set mux and pin number.

Parameters

mux	mux number (from DigitalIn initialization order)
muxpin	pin on the mux (0-15)

Definition at line 10 of file Button.cpp.

5.2.3.2 Button() [2/2]

```
Button::Button (
            uint8_t pin )  [inline]
```

Constructor, set digital input without mux.

Parameters

pin	Arduino pin number

Definition at line 21 of file Button.h.

5.2.4 Member Function Documentation

5.2.4.1 engaged()

```
bool Button::engaged ( )  [inline]
```

Evaluate status of Button.

Returns

true: Button is currently held down

Definition at line 47 of file Button.h.

5.2.4.2 getCommand()

```
int Button::getCommand ( )  [inline]
```

Get XPLDirect command associated with Button.

Returns

Handle of the command

Definition at line 59 of file Button.h.

5.2.4.3 handle() [1/2]

```
void Button::handle ( )  [inline]
```

Handle realtime. Read input and evaluate any transitions.

Definition at line 24 of file Button.h.

5.2.4.4 handle() [2/2]

```
void Button::handle (
            bool input )  [inline]
```

Handle realtime. Read input and evaluate any transitions.

Parameters

input	Additional mask bit. AND connected with physical input.

Definition at line 28 of file Button.h.

5.2.4.5 handleXP() [1/2]

```
void Button::handleXP ( )  [inline]
```

Handle realtime and process XPLDirect commands.

Definition at line 31 of file Button.h.

5.2.4.6 handleXP() [2/2]

```
void Button::handleXP (
            bool input )  [inline]
```

Handle realtime and process XPLDirect commands.

Parameters

input	Additional mask bit. AND tied with physical input.

Definition at line 35 of file Button.h.

5.2.4.7 pressed()

```
bool Button::pressed ( )  [inline]
```

Evaluate and reset transition if button pressed down.

Returns

true: Button was pressed. Transition detected.

Definition at line 39 of file Button.h.

5.2.4.8 processCommand()

```
void Button::processCommand ( )
```

Process all transitions and active transitions to XPLDirect

Definition at line 50 of file Button.cpp.

5.2.4.9 released()

```
bool Button::released ( )  [inline]
```

Evaluate and reset transition if button released.

Returns

true: Button was released. Transition detected.

Definition at line 43 of file Button.h.

5.2.4.10 setCommand() [1/2]

```
void Button::setCommand (
            int cmdPush )
```

Set XPLDirect command for Button events.

Parameters

cmdPush	Command handle as returned by XP.registerCommand()

Definition at line 40 of file Button.cpp.

5.2.4.11 setCommand() [2/2]

```
void Button::setCommand (
            XPString_t * cmdNamePush )
```

Set XPLDirect command for Button events.

Parameters

cmdNamePush	Command name to register

Definition at line 45 of file Button.cpp.

5.2.5 Member Data Documentation

5.2.5.1 _cmdPush

```
int Button::_cmdPush  [protected]
```

Definition at line 75 of file Button.h.

5.2.5.2 _mux

```
uint8_t Button::_mux  [protected]
```

Definition at line 71 of file Button.h.

5.2.5.3 _pin

```
uint8_t Button::_pin  [protected]
```

Definition at line 72 of file Button.h.

5.2.5.4 _state

```
uint8_t Button::_state  [protected]
```

Definition at line 73 of file Button.h.

5.2.5.5 _transition

```
uint8_t Button::_transition  [protected]
```

Definition at line 74 of file Button.h.

The documentation for this class was generated from the following files:

- Button.h
- Button.cpp

5.3 DigitalIn_ Class Reference

Class to encapsulate digital inputs from 74HC4067 and MCP23017 input multiplexers, used by all digital input devices. Scans all expander inputs into internal process data image.

`#include <DigitalIn.h>`

Public Member Functions

- DigitalIn_ ()
  Class constructor.
- void setMux (uint8_t s0, uint8_t s1, uint8_t s2, uint8_t s3)
  Set adress pins for 74HC4067 multiplexers. All mux share the same adress pins.
- bool addMux (uint8_t pin)
  Add one 74HC4067 multiplexer.
- bool getBit (uint8_t expander, uint8_t channel)
  Get one bit from the mux or a digital input.
- void handle ()
  Read all mux inputs into process data input image.

5.3.1 Detailed Description

Class to encapsulate digital inputs from 74HC4067 and MCP23017 input multiplexers, used by all digital input devices. Scans all expander inputs into internal process data image.

Definition at line 24 of file DigitalIn.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 DigitalIn_()

```
DigitalIn_::DigitalIn_ ( )
```

Class constructor.

Definition at line 7 of file DigitalIn.cpp.

5.3.3 Member Function Documentation

5.3.3.1 addMux()

```
bool DigitalIn_::addMux (
            uint8_t pin )
```

Add one 74HC4067 multiplexer.

Parameters

pin	Data pin the multiplexer is connected to

Returns

true when successful, false when all expanders have been used up (increase MUX_MAX_NUMBER)

Definition at line 44 of file DigitalIn.cpp.

5.3.3.2 getBit()

```
bool DigitalIn_::getBit (
            uint8_t expander,
            uint8_t channel )
```

Get one bit from the mux or a digital input.

Parameters

expander	Expander (mux or mcp) to read from. Use NOT_USED to access directly ardunio digital input
channel	Channel (0-15) on the mux or Arduino pin when mux = NOT_USED

Returns

Status of the input (inverted, true = GND, false = +5V)

Definition at line 79 of file DigitalIn.cpp.

5.3.3.3 handle()

```
void DigitalIn_::handle ( )
```

Read all mux inputs into process data input image.

Definition at line 93 of file DigitalIn.cpp.

5.3.3.4 setMux()

```
void DigitalIn_::setMux (
            uint8_t s0,
            uint8_t s1,
            uint8_t s2,
            uint8_t s3 )
```

Set adress pins for 74HC4067 multiplexers. All mux share the same adress pins.

Parameters

s0	Adress pin s0
s1	Adress pin s1
s2	Adress pin s2
s3	Adress pin s3

Definition at line 21 of file DigitalIn.cpp.

The documentation for this class was generated from the following files:

- DigitalIn.h
- DigitalIn.cpp

5.4 Encoder Class Reference

Class for rotary encoders with optional push functionality. The number of counts per mechanical notch can be configured for the triggering of up/down events.

`#include <Encoder.h>`

Public Member Functions

- Encoder (uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses)
  Constructor. Sets connected pins and number of counts per notch.
- Encoder (uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses)
  Constructor. Sets connected pins and number of counts per notch.
- void handle ()
  Handle realtime. Read input and evaluate any transitions.
- void handleXP ()
  Handle realtime and process XPLDirect commands.
- int16_t pos ()
  Read current Encoder count.
- bool up ()
  Evaluate Encoder up one notch (positive turn) and consume event.
- bool down ()
  Evaluate Encoder up down notch (negative turn) and consume event.
- bool pressed ()
  Evaluate and reset transition if Encoder pressed down.
- bool released ()
  Evaluate and reset transition if Encoder released.
- bool engaged ()
  Evaluate status of Encoder push function.
- void setCommand (int cmdUp, int cmdDown, int cmdPush)
  Set XPLDirect commands for Encoder events.
- void setCommand (XPString_t *cmdNameUp, XPString_t *cmdNameDown, XPString_t *cmdNamePush)
  Set XPLDirect commands for Encoder events.
- void setCommand (int cmdUp, int cmdDown)
  Set XPLDirect commands for Encoder events without push function.
- void setCommand (XPString_t *cmdNameUp, XPString_t *cmdNameDown)
  Set XPLDirect commands for Encoder events.
- int getCommand (EncCmd_t cmd)
  Get XPLDirect command assiciated with the selected event.
- void processCommand ()
  Check for Encoder events and process XPLDirect commands as appropriate.

5.4.1 Detailed Description

Class for rotary encoders with optional push functionality. The number of counts per mechanical notch can be configured for the triggering of up/down events.

Definition at line 22 of file Encoder.h.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Encoder() [1/2]

```
Encoder::Encoder (
            uint8_t mux,
            uint8_t pin1,
            uint8_t pin2,
            uint8_t pin3,
            EncPulse_t pulses )
```

Constructor. Sets connected pins and number of counts per notch.

Parameters

mux	mux number (from DigitalIn initialization order)
pin1	pin for Encoder A track
pin2	pin for Encoder B track
pin3	pin for encoder push function (NOT_USED if not connected)
pulses	Number of counts per mechanical notch

Definition at line 10 of file Encoder.cpp.

5.4.2.2 Encoder() [2/2]

```
Encoder::Encoder (
            uint8_t pin1,
            uint8_t pin2,
            uint8_t pin3,
            EncPulse_t pulses )  [inline]
```

Constructor. Sets connected pins and number of counts per notch.

Parameters

pin1	pin for Encoder A track
pin2	pin for Encoder B track
pin3	pin for encoder push function (NOT_USED if not connected)
pulses	Number of counts per mechanical notch

Definition at line 38 of file Encoder.h.

5.4.3 Member Function Documentation

5.4.3.1 down()

```
bool Encoder::down ( )  [inline]
```

Evaluate Encoder up down notch (negative turn) and consume event.

Returns

true: up event available and transition reset.

Definition at line 56 of file Encoder.h.

5.4.3.2 engaged()

```
bool Encoder::engaged ( )  [inline]
```

Evaluate status of Encoder push function.

Returns

true: Button is currently held down

Definition at line 68 of file Encoder.h.

5.4.3.3 getCommand()

```
int Encoder::getCommand (
            EncCmd_t cmd )
```

Get XPLDirect command assiciated with the selected event.

Parameters

cmd	Event to read out (encCmdUp, encCmdDown, encCmdPush)

Returns

Handle of the command, -1 = no command

Definition at line 103 of file Encoder.cpp.

5.4.3.4 handle()

```
void Encoder::handle ( )
```

Handle realtime. Read input and evaluate any transitions.

Definition at line 32 of file Encoder.cpp.

5.4.3.5 handleXP()

```
void Encoder::handleXP ( )  [inline]
```

Handle realtime and process XPLDirect commands.

Definition at line 44 of file Encoder.h.

5.4.3.6 pos()

```
int16_t Encoder::pos ( )  [inline]
```

Read current Encoder count.

Returns

Remaining Encoder count.

Definition at line 48 of file Encoder.h.

5.4.3.7 pressed()

```
bool Encoder::pressed ( )  [inline]
```

Evaluate and reset transition if Encoder pressed down.

Returns

true: Button was pressed. Transition detected and reset.

Definition at line 60 of file Encoder.h.

5.4.3.8 processCommand()

```
void Encoder::processCommand ( )
```

Check for Encoder events and process XPLDirect commands as appropriate.

Definition at line 122 of file Encoder.cpp.

5.4.3.9 released()

```
bool Encoder::released ( )  [inline]
```

Evaluate and reset transition if Encoder released.

Returns

true: Button was released. Transition detected and reset.

Definition at line 64 of file Encoder.h.

5.4.3.10 setCommand() [1/4]

```
void Encoder::setCommand (
            int cmdUp,
            int cmdDown )
```

Set XPLDirect commands for Encoder events without push function.

Parameters

cmdUp	Command handle for positive turn as returned by XP.registerCommand()
cmdDown	Command handle for negative turn as returned by XP.registerCommand()

Definition at line 89 of file Encoder.cpp.

5.4.3.11 setCommand() [2/4]

```
void Encoder::setCommand (
            int cmdUp,
            int cmdDown,
            int cmdPush )
```

Set XPLDirect commands for Encoder events.

Parameters

cmdUp	Command handle for positive turn as returned by XP.registerCommand()
cmdDown	Command handle for negative turn as returned by XP.registerCommand()
cmdPush	Command handle for push as returned by XP.registerCommand()

Definition at line 75 of file Encoder.cpp.

5.4.3.12 setCommand() [3/4]

```
void Encoder::setCommand (
            XPString_t * cmdNameUp,
            XPString_t * cmdNameDown )
```

Set XPLDirect commands for Encoder events.

Parameters

cmdNameUp	Command for positive turn
cmdNameDown	Command for negative turn

Definition at line 96 of file Encoder.cpp.

5.4.3.13 setCommand() [4/4]

```
void Encoder::setCommand (
            XPString_t * cmdNameUp,
            XPString_t * cmdNameDown,
            XPString_t * cmdNamePush )
```

Set XPLDirect commands for Encoder events.

Parameters

cmdNameUp	Command for positive turn
cmdNameDown	Command for negative turn
cmdNamePush	Command for push

Definition at line 82 of file Encoder.cpp.

5.4.3.14 up()

```
bool Encoder::up ( )  [inline]
```

Evaluate Encoder up one notch (positive turn) and consume event.

Returns

true: up event available and transition reset.

Definition at line 52 of file Encoder.h.

The documentation for this class was generated from the following files:

- Encoder.h
- Encoder.cpp

5.5 LedShift Class Reference

Class to encapsulate a DM13A LED driver IC.

`#include <LedShift.h>`

Public Member Functions

- LedShift (uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins=16)
  Constructor, setup DM13A LED driver and set pins.
- void setPin (uint8_t pin, led_t mode)
  Set one LED to a display mode.
- void set (uint8_t pin, led_t mode)
- void setAll (led_t mode)
  Set display mode for all LEDs.
- void set_all (led_t mode)
- void handle ()
  Real time handling, call cyclic in loop()

5.5.1 Detailed Description

Class to encapsulate a DM13A LED driver IC.

Definition at line 21 of file LedShift.h.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 LedShift()

```
LedShift::LedShift (
            uint8_t pin_DAI,
            uint8_t pin_DCK,
            uint8_t pin_LAT,
            uint8_t pins = 16 )
```

Constructor, setup DM13A LED driver and set pins.

Parameters

pin_DAI	DAI pin of DM13A
pin_DCK	DCL pin of DM13A
pin_LAT	LAT pin of DM13A
pins	Number of LED pins for cascaded LED drivers (max 64)

Definition at line 6 of file LedShift.cpp.

5.5.3 Member Function Documentation

5.5.3.1 handle()

```
void LedShift::handle ( )
```

Real time handling, call cyclic in loop()

Definition at line 73 of file LedShift.cpp.

5.5.3.2 set()

```
void LedShift::set (
            uint8_t pin,
            led_t mode )  [inline]
```

Definition at line 35 of file LedShift.h.

5.5.3.3 set_all()

```
void LedShift::set_all (
            led_t mode )  [inline]
```

Definition at line 40 of file LedShift.h.

5.5.3.4 setAll()

```
void LedShift::setAll (
            led_t mode )
```

Set display mode for all LEDs.

Parameters

mode	LED display mode (ledOff, ledFast, ledMedium, ledSlow, ledOn)

Definition at line 64 of file LedShift.cpp.

5.5.3.5 setPin()

```
void LedShift::setPin (
            uint8_t pin,
            led_t mode )
```

Set one LED to a display mode.

Parameters

pin	DM13A pin of the LED (0-64)
mode	LED display mode (ledOff, ledFast, ledMedium, ledSlow, ledOn)

Definition at line 52 of file LedShift.cpp.

The documentation for this class was generated from the following files:

- LedShift.h
- LedShift.cpp

5.6 RepeatButton Class Reference

Class for a simple pushbutton with debouncing and XPLDirect command handling, supports start and end of commands so XPlane can show the current Button status. When button is held down cyclic new pressed events are generated for auto repeat function.

```
#include <Button.h>
```

Public Member Functions

- RepeatButton (uint8_t mux, uint8_t muxpin, uint32_t delay)

  Constructor, set mux and pin number.
- RepeatButton (uint8_t pin, uint32_t delay)

  Constructor, set digital input without mux.
- void handle ()

  Handle realtime. Read input and evaluate any transitions.
- void handle (bool input)

  Handle realtime. Read input and evaluate any transitions.
- void handleXP ()

  Handle realtime and process XPLDirect commands.
- void handleXP (bool input)

  Handle realtime and process XPLDirect commands.

Public Member Functions inherited from Button

- Button (uint8_t mux, uint8_t muxpin)

  Constructor, set mux and pin number.
- Button (uint8_t pin)

  Constructor, set digital input without mux.
- void handle ()

  Handle realtime. Read input and evaluate any transitions.
- void handle (bool input)

  Handle realtime. Read input and evaluate any transitions.
- void handleXP ()

  Handle realtime and process XPLDirect commands.
- void handleXP (bool input)

  Handle realtime and process XPLDirect commands.
- bool pressed ()

  Evaluate and reset transition if button pressed down.
- bool released ()

  Evaluate and reset transition if button released.
- bool engaged ()

  Evaluate status of Button.
- void setCommand (int cmdPush)

  Set XPLDirect command for Button events.
- void setCommand (XPString_t *cmdNamePush)

  Set XPLDirect command for Button events.
- int getCommand ()

  Get XPLDirect command associated with Button.
- void processCommand ()

  Process all transitions and active transitions to XPLDirect

Protected Attributes

- uint32_t _delay
- uint32_t _timer

Protected Attributes inherited from Button

- uint8_t _mux
- uint8_t _pin
- uint8_t _state
- uint8_t _transition
- int _cmdPush

Additional Inherited Members

Protected Types inherited from Button

- enum { **transNone** , **transPressed** , **transReleased** }

5.6.1 Detailed Description

Class for a simple pushbutton with debouncing and XPLDirect command handling, supports start and end of commands so XPlane can show the current Button status. When button is held down cyclic new pressed events are generated for auto repeat function.

Definition at line 81 of file Button.h.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 RepeatButton() [1/2]

```
RepeatButton::RepeatButton (
            uint8_t mux,
            uint8_t muxpin,
            uint32_t delay )
```

Constructor, set mux and pin number.

Parameters

mux	mux number (from initialization order)
muxpin	pin on the mux (0-15)
delay	Cyclic delay for repeat function

Definition at line 62 of file Button.cpp.

5.6.2.2 RepeatButton() [2/2]

```
RepeatButton::RepeatButton (
            uint8_t pin,
            uint32_t delay )  [inline]
```

Constructor, set digital input without mux.

Parameters

pin	Arduino pin number
delay	Cyclic delay for repeat function

Definition at line 96 of file Button.h.

5.6.3 Member Function Documentation

5.6.3.1 handle() [1/2]

```
void RepeatButton::handle ( )  [inline]
```

Handle realtime. Read input and evaluate any transitions.

Definition at line 99 of file Button.h.

5.6.3.2 handle() [2/2]

```
void RepeatButton::handle (
            bool input )  [inline]
```

Handle realtime. Read input and evaluate any transitions.

Parameters

input	Additional mask bit. AND connected with physical input.

Definition at line 103 of file Button.h.

5.6.3.3 handleXP() [1/2]

```
void RepeatButton::handleXP ( )  [inline]
```

Handle realtime and process XPLDirect commands.

Definition at line 106 of file Button.h.

5.6.3.4 handleXP() [2/2]

```
void RepeatButton::handleXP (
            bool input )  [inline]
```

Handle realtime and process XPLDirect commands.

Parameters

input	Additional mask bit. AND tied with physical input.

Definition at line 110 of file Button.h.

5.6.4 Member Data Documentation

5.6.4.1 _delay

```
uint32_t RepeatButton::_delay  [protected]
```

Definition at line 113 of file Button.h.

5.6.4.2 _timer

```
uint32_t RepeatButton::_timer  [protected]
```

Definition at line 114 of file Button.h.

The documentation for this class was generated from the following files:

- Button.h
- Button.cpp

5.7 ShiftOut Class Reference

Class to encapsulate a DM13A LED driver IC.

```
#include <ShiftOut.h>
```

Public Member Functions

- ShiftOut (uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins=16)

  Constructor, setup shift register and set pins.
- void setPin (uint8_t pin, bool state)

  Set one outpot to a display mode.
- void setAll (bool state)

  Set state for all outputs.
- void handle ()

  Real time handling, call cyclic in loop()

5.7.1 Detailed Description

Class to encapsulate a DM13A LED driver IC.

Definition at line 6 of file ShiftOut.h.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 ShiftOut()

```
ShiftOut::ShiftOut (
            uint8_t pin_DAI,
            uint8_t pin_DCK,
            uint8_t pin_LAT,
            uint8_t pins = 16 )
```

Constructor, setup shift register and set pins.

Parameters

pin_DAI	DAI pin (data)
pin_DCK	DCL pin (clock)
pin_LAT	LAT pin (latch)
pins	Number of pins for cascaded shift registers (max 64)

Definition at line 4 of file ShiftOut.cpp.

5.7.3 Member Function Documentation

5.7.3.1 handle()

```
void ShiftOut::handle ( )
```

Real time handling, call cyclic in loop()

Definition at line 64 of file ShiftOut.cpp.

5.7.3.2 setAll()

```
void ShiftOut::setAll (
            bool state )
```

Set state for all outputs.

Parameters

state	State to set (HIGH/LOW)

Definition at line 55 of file ShiftOut.cpp.

5.7.3.3 setPin()

```
void ShiftOut::setPin (
            uint8_t pin,
            bool state )
```

Set one outpot to a display mode.

Parameters

pin	Pin to set (0-64)
state	State to set (HIGH/LOW)

Definition at line 43 of file ShiftOut.cpp.

The documentation for this class was generated from the following files:

- ShiftOut.h
- ShiftOut.cpp

5.8 Switch Class Reference

Class for a simple on/off switch with debouncing and XPLDirect command handling.

```
#include <Switch.h>
```

Public Member Functions

- Switch (uint8_t mux, uint8_t pin)

  Constructor. Connect the switch to a pin on a mux.
- Switch (uint8_t pin)

  Constructor, set digital input without mux.
- void handle ()

  Handle realtime. Read input and evaluate any transitions.
- void handleXP ()

  Handle realtime and process XPLDirect commands.
- bool on ()

  Check whether Switch set to on.
- bool off ()

Check whether Switch set to off.

- void setCommand (int cmdOn)

  Set XPLDirect commands for Switch events (command only for on position)
- void setCommand (XPString_t *cmdNameOn)

  Set XPLDirect commands for Switch events (command only for on position)
- void setCommand (int cmdOn, int cmdOff)

  Set XPLDirect commands for Switch events.
- void setCommand (XPString_t *cmdNameOn, XPString_t *cmdNameOff)

  Set XPLDirect commands for Switch events.
- int getCommand ()

  Get XPLDirect command for last transition of Switch.
- void processCommand ()

  Process all transitions to XPLDirect.
- float value (float onValue, float offValue)

  Check Status of Switch and translate to float value.

5.8.1 Detailed Description

Class for a simple on/off switch with debouncing and XPLDirect command handling.

Definition at line 7 of file Switch.h.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Switch() [1/2]

```
Switch::Switch (
            uint8_t mux,
            uint8_t pin )
```

Constructor. Connect the switch to a pin on a mux.

Parameters

mux	mux number (from DigitalIn initialization order)
pin	pin on the mux (0-15)

Definition at line 9 of file Switch.cpp.

5.8.2.2 Switch() [2/2]

```
Switch::Switch (
            uint8_t pin )  [inline]
```

Constructor, set digital input without mux.

Parameters

pin	Arduino pin number

Definition at line 17 of file Switch.h.

5.8.3 Member Function Documentation

5.8.3.1 getCommand()

```
int Switch::getCommand ( )
```

Get XPLDirect command for last transition of Switch.

Returns

Handle of the last command

Definition at line 65 of file Switch.cpp.

5.8.3.2 handle()

```
void Switch::handle ( )
```

Handle realtime. Read input and evaluate any transitions.

Definition at line 19 of file Switch.cpp.

5.8.3.3 handleXP()

```
void Switch::handleXP ( )  [inline]
```

Handle realtime and process XPLDirect commands.

Definition at line 23 of file Switch.h.

5.8.3.4 off()

```
bool Switch::off ( )  [inline]
```

Check whether Switch set to off.

Returns

true: Switch is off

Definition at line 31 of file Switch.h.

5.8.3.5 on()

```
bool Switch::on ( )  [inline]
```

Check whether Switch set to on.

Returns

true: Switch is on

Definition at line 27 of file Switch.h.

5.8.3.6 processCommand()

```
void Switch::processCommand ( )
```

Process all transitions to XPLDirect.

Definition at line 81 of file Switch.cpp.

5.8.3.7 setCommand() [1/4]

```
void Switch::setCommand (
            int cmdOn )
```

Set XPLDirect commands for Switch events (command only for on position)

Parameters

cmdOn	Command handle for Switch moved to on as returned by XP.registerCommand()

Definition at line 41 of file Switch.cpp.

5.8.3.8 setCommand() [2/4]

```
void Switch::setCommand (
            int cmdOn,
            int cmdOff )
```

Set XPLDirect commands for Switch events.

Parameters

cmdOn	Command handle for Switch moved to on as returned by XP.registerCommand()
cmdOff	Command handle for Switch moved to off as returned by XP.registerCommand()

Definition at line 53 of file Switch.cpp.

5.8.3.9 setCommand() [3/4]

```
void Switch::setCommand (
            XPString_t * cmdNameOn )
```

Set XPLDirect commands for Switch events (command only for on position)

Parameters

cmdNameOn	Command for Switch moved to on

Definition at line 47 of file Switch.cpp.

5.8.3.10 setCommand() [4/4]

```
void Switch::setCommand (
            XPString_t * cmdNameOn,
            XPString_t * cmdNameOff )
```

Set XPLDirect commands for Switch events.

Parameters

cmdNameOn	Command for Switch moved to on
cmdNameOff	Command for Switch moved to off

Definition at line 59 of file Switch.cpp.

5.8.3.11 value()

```
float Switch::value (
            float onValue,
            float offValue )  [inline]
```

Check Status of Switch and translate to float value.

Parameters

onValue	Value to return when Switch is set to on
offValue	Value to return when Switch is set to off

Returns

Returned value

Definition at line 62 of file Switch.h.

The documentation for this class was generated from the following files:

- Switch.h
- Switch.cpp

5.9 Switch2 Class Reference

Class for an on/off/on switch with debouncing and XPLDirect command handling.

```
#include <Switch.h>
```

Public Member Functions

- Switch2 (uint8_t mux, uint8_t pin1, uint8_t pin2)

  Constructor. Connect the switch to pins on a mux.
- Switch2 (uint8_t pin1, uint8_t pin2)

  Constructor, set digital input pins without mux.
- void handle ()

  Handle realtime. Read inputs and evaluate any transitions.
- void handleXP ()

  Handle realtime and process XPLDirect commands.
- bool off ()

  Check whether Switch set to off.
- bool on1 ()

  Check whether Switch set to on1.

- bool on2 ()

  Check whether Switch set to on2.
- void setCommand (int cmdUp, int cmdDown)

  Set XPLDirect commands for Switch events in cases only up/down commands are to be used.
- void setCommand (XPString_t ∗cmdNameUp, XPString_t ∗cmdNameDown)

  Set XPLDirect commands for Switch events in cases only up/down commands are to be used.
- void setCommand (int cmdOn1, int cmdOff, int cmdOn2)

  Set XPLDirect commands for Switch events in cases separate events for on1/off/on2 are to be used.
- void setCommand (XPString_t ∗cmdNameOn1, XPString_t ∗cmdNameOff, XPString_t ∗cmdNameOn2)

  Set XPLDirect commands for Switch events in cases separate events for on1/off/on2 are to be used.
- int getCommand ()

  Get XPLDirect command for last transition of Switch.
- void processCommand ()

  Process all transitions to XPLDirect.
- float value (float on1Value, float offValue, float on2Value)

  Check Status of Switch and translate to float value.

5.9.1 Detailed Description

Class for an on/off/on switch with debouncing and XPLDirect command handling.

Definition at line 80 of file Switch.h.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 Switch2() [1/2]

```
Switch2::Switch2 (
            uint8_t mux,
            uint8_t pin1,
            uint8_t pin2 )
```

Constructor. Connect the switch to pins on a mux.

Parameters

mux	mux number (from DigitalIn initialization order)
pin1	on1 pin on the mux (0-15)
pin2	on2 pin on the mux (0-15)

Definition at line 96 of file Switch.cpp.

5.9.2.2 Switch2() [2/2]

```
Switch2::Switch2 (
            uint8_t pin1,
            uint8_t pin2 )  [inline]
```

Constructor, set digital input pins without mux.

Parameters

pin1	on1 Arduino pin number
pin2	on2 Arduino pin number

Definition at line 92 of file Switch.h.

5.9.3 Member Function Documentation

5.9.3.1 getCommand()

```
int Switch2::getCommand ( )
```

Get XPLDirect command for last transition of Switch.

Returns

Handle of the last command

Definition at line 167 of file Switch.cpp.

5.9.3.2 handle()

```
void Switch2::handle ( )
```

Handle realtime. Read inputs and evaluate any transitions.

Definition at line 112 of file Switch.cpp.

5.9.3.3 handleXP()

```
void Switch2::handleXP ( )  [inline]
```

Handle realtime and process XPLDirect commands.

Definition at line 98 of file Switch.h.

5.9.3.4 off()

```
bool Switch2::off ( )  [inline]
```

Check whether Switch set to off.

Returns

true: Switch is off

Definition at line 102 of file Switch.h.

5.9.3.5 on1()

```
bool Switch2::on1 ( )  [inline]
```

Check whether Switch set to on1.

Returns

true: Switch is on1

Definition at line 106 of file Switch.h.

5.9.3.6 on2()

```
bool Switch2::on2 ( )  [inline]
```

Check whether Switch set to on2.

Returns

true: Switch is on2

Definition at line 110 of file Switch.h.

5.9.3.7 processCommand()

```
void Switch2::processCommand ( )
```

Process all transitions to XPLDirect.

Definition at line 206 of file Switch.cpp.

5.9.3.8 setCommand() [1/4]

```
void Switch2::setCommand (
            int cmdOn1,
            int cmdOff,
            int cmdOn2 )
```

Set XPLDirect commands for Switch events in cases separate events for on1/off/on2 are to be used.

Parameters

cmdOn1	Command handle for Switch moved to on1 position as returned by XP.registerCommand()
cmdOff	Command handle for Switch moved to off position as returned by XP.registerCommand()
cmdOn2	Command handle for Switch moved to on2 position as returned by XP.registerCommand()

Definition at line 153 of file Switch.cpp.

5.9.3.9 setCommand() [2/4]

```
void Switch2::setCommand (
            int cmdUp,
            int cmdDown )
```

Set XPLDirect commands for Switch events in cases only up/down commands are to be used.

Parameters

cmdUp	Command handle for Switch moved from on1 to off or from off to on2 as returned by XP.registerCommand()
cmdDown	Command handle for Switch moved from on2 to off or from off to on1 as returned by XP.registerCommand()

Definition at line 139 of file Switch.cpp.

5.9.3.10 setCommand() [3/4]

```
void Switch2::setCommand (
            XPString_t * cmdNameOn1,
            XPString_t * cmdNameOff,
            XPString_t * cmdNameOn2 )
```

Set XPLDirect commands for Switch events in cases separate events for on1/off/on2 are to be used.

Parameters

cmdNameOn1	Command for Switch moved to on1 position
cmdNameOff	Command for Switch moved to off position
cmdNameOn2	Command for Switch moved to on2 position

Definition at line 160 of file Switch.cpp.

5.9.3.11 setCommand() [4/4]

```
void Switch2::setCommand (
            XPString_t * cmdNameUp,
            XPString_t * cmdNameDown )
```

Set XPLDirect commands for Switch events in cases only up/down commands are to be used.

Parameters

cmdNameUp	Command for Switch moved from on1 to off or from off to on2 on
cmdNameDown	Command for Switch moved from on2 to off or from off to on1

Definition at line 146 of file Switch.cpp.

5.9.3.12 value()

```
float Switch2::value (
            float on1Value,
            float offValue,
            float on2Value )  [inline]
```

Check Status of Switch and translate to float value.

Parameters

on1Value	Value to return when Switch is set to on1
offValue	Value to return when Switch is set to off
on2Value	Value to return when Switch is set to on2

Returns

Returned value

Definition at line 146 of file Switch.h.

The documentation for this class was generated from the following files:

- Switch.h
- Switch.cpp

5.10 Timer Class Reference

Priovide a simple software driven timer for general purpose use.

```
#include <Timer.h>
```

Public Member Functions

- Timer (float cycle=0)

  Setup timer.
- void setCycle (float cycle)

  Set or reset cycle time.
- bool elapsed ()

  Check if cyclic timer elapsed and reset if so.
- float getTime ()

  Get measured time since and reset timer.
- long count ()

  Return cycle counter and reset to zero.

5.10.1 Detailed Description

Priovide a simple software driven timer for general purpose use.

Definition at line 6 of file Timer.h.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 Timer()

```
Timer::Timer (
            float cycle = 0 )
```

Setup timer.

Parameters

cycle	Cycle time for elapsing timer in ms. 0 means no cycle, just for measurement.

Definition at line 4 of file Timer.cpp.

5.10.3 Member Function Documentation

5.10.3.1 count()

```
long Timer::count ( )
```

Return cycle counter and reset to zero.

Returns

Number of calls to elapsed() since last call of count()

Definition at line 35 of file Timer.cpp.

5.10.3.2 elapsed()

```
bool Timer::elapsed ( )
```

Check if cyclic timer elapsed and reset if so.

Returns

true: timer elapsed and restarted, false: still running

Definition at line 15 of file Timer.cpp.

5.10.3.3 getTime()

```
float Timer::getTime ( )
```

Get measured time since and reset timer.

Returns

Elapsed time in ms

Definition at line 27 of file Timer.cpp.

5.10.3.4 setCycle()

```
void Timer::setCycle (
            float cycle )
```

Set or reset cycle time.

Parameters

cycle	Cycle time in ms

Definition at line 10 of file Timer.cpp.

The documentation for this class was generated from the following files:

- Timer.h
- Timer.cpp

5.11 XPLDirect Class Reference

Public Member Functions

- XPLDirect (Stream ∗)
- void begin (const char ∗devicename)
- int connectionStatus (void)
- int commandTrigger (int commandHandle)
- int commandTrigger (int commandHandle, int triggerCount)
- int commandStart (int commandHandle)
- int commandEnd (int commandHandle)
- int datarefsUpdated ()
- int hasUpdated (int handle)
- int registerDataRef (XPString_t ∗datarefName, int rwmode, unsigned int rate, float divider, long int ∗value)
- int registerDataRef (XPString_t ∗datarefName, int rwmode, unsigned int rate, float divider, long int ∗value, int index)
- int registerDataRef (XPString_t ∗datarefName, int rwmode, unsigned int rate, float divider, float ∗value)
- int registerDataRef (XPString_t ∗datarefName, int rwmode, unsigned int rate, float divider, float ∗value, int index)
- int registerDataRef (XPString_t ∗datarefName, int rwmode, unsigned int rate, char ∗value)
- int registerCommand (XPString_t ∗commandName)
- int sendDebugMessage (const char ∗msg)
- int sendSpeakMessage (const char ∗msg)
- int allDataRefsRegistered (void)
- void sendResetRequest (void)
- int xloop (void)

5.11.1 Detailed Description

Definition at line 81 of file XPLDirect.h.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 XPLDirect()

```
XPLDirect::XPLDirect (
            Stream * device )
```

Definition at line 11 of file XPLDirect.cpp.

5.11.3 Member Function Documentation

5.11.3.1 allDataRefsRegistered()

```
int XPLDirect::allDataRefsRegistered (
            void  )
```

Definition at line 458 of file XPLDirect.cpp.

5.11.3.2 begin()

```
void XPLDirect::begin (
            const char * devicename )
```

Definition at line 17 of file XPLDirect.cpp.

5.11.3.3 commandEnd()

```
int XPLDirect::commandEnd (
            int commandHandle )
```

Definition at line 129 of file XPLDirect.cpp.

5.11.3.4 commandStart()

```
int XPLDirect::commandStart (
            int commandHandle )
```

Definition at line 111 of file XPLDirect.cpp.

5.11.3.5 commandTrigger() [1/2]

```
int XPLDirect::commandTrigger (
            int commandHandle )
```

Definition at line 72 of file XPLDirect.cpp.

5.11.3.6 commandTrigger() [2/2]

```
int XPLDirect::commandTrigger (
            int commandHandle,
            int triggerCount )
```

Definition at line 90 of file XPLDirect.cpp.

5.11.3.7 connectionStatus()

```
int XPLDirect::connectionStatus (
            void  )
```

Definition at line 147 of file XPLDirect.cpp.

5.11.3.8 datarefsUpdated()

```
int XPLDirect::datarefsUpdated ( )
```

Definition at line 174 of file XPLDirect.cpp.

5.11.3.9 hasUpdated()

```
int XPLDirect::hasUpdated (
            int handle )
```

Definition at line 164 of file XPLDirect.cpp.

5.11.3.10 registerCommand()

```
int XPLDirect::registerCommand (
            XPString_t * commandName )
```

Definition at line 566 of file XPLDirect.cpp.

5.11.3.11 registerDataRef() [1/5]

```
int XPLDirect::registerDataRef (
            XPString_t * datarefName,
            int rwmode,
            unsigned int rate,
            char * value )
```

Definition at line 546 of file XPLDirect.cpp.

5.11.3.12 registerDataRef() [2/5]

```
int XPLDirect::registerDataRef (
            XPString_t * datarefName,
            int rwmode,
            unsigned int rate,
            float divider,
            float * value )
```

Definition at line 505 of file XPLDirect.cpp.

5.11.3.13 registerDataRef() [3/5]

```
int XPLDirect::registerDataRef (
            XPString_t * datarefName,
            int rwmode,
            unsigned int rate,
            float divider,
            float * value,
            int index )
```

Definition at line 526 of file XPLDirect.cpp.

5.11.3.14 registerDataRef() [4/5]

```
int XPLDirect::registerDataRef (
            XPString_t * datarefName,
            int rwmode,
            unsigned int rate,
            float divider,
            long int * value )
```

Definition at line 463 of file XPLDirect.cpp.

5.11.3.15 registerDataRef() [5/5]

```
int XPLDirect::registerDataRef (
            XPString_t * datarefName,
            int rwmode,
            unsigned int rate,
            float divider,
            long int * value,
            int index )
```

Definition at line 484 of file XPLDirect.cpp.

5.11.3.16 sendDebugMessage()

```
int XPLDirect::sendDebugMessage (
            const char * msg )
```

Definition at line 152 of file XPLDirect.cpp.

5.11.3.17 sendResetRequest()

```
void XPLDirect::sendResetRequest (
            void  )
```

Definition at line 200 of file XPLDirect.cpp.

5.11.3.18 sendSpeakMessage()

```
int XPLDirect::sendSpeakMessage (
            const char * msg )
```

Definition at line 158 of file XPLDirect.cpp.

5.11.3.19 xloop()

```
int XPLDirect::xloop (
            void  )
```

Definition at line 27 of file XPLDirect.cpp.

The documentation for this class was generated from the following files:

- XPLDirect.h
- XPLDirect.cpp

Chapter 6

File Documentation

6.1 Direct inputs/main.cpp

```
00001 #include <Arduino.h>
00002 #include <XPLDevices.h>
00003
00004 // The XPLDirect library is automatically installed by PlatformIO with XPLDevices
00005 // Optional defines for XPLDirect can be set in platformio.ini
00006 // This sample contains all the important defines. Modify or remove as needed
00007
00008 // A simple Pushbutton on Arduino pin 2
00009 Button btnStart(2);
00010
00011 // An Encoder with push functionality. 3&4 are the encoder pins, 5 the push pin.
00012 // configured for an Encoder with 4 counts per mechanical notch, which is the standard
00013 Encoder encHeading(3, 4, 5, enc4Pulse);
00014
00015 // A simple On/Off switch on pin 6
00016 Switch swStrobe(6);
00017
00018 // A Variable to be connected to a DataRef
00019 long strobe;
00020
00021 // Arduino setup function, called once
00022 void setup() {
00023   // setup interface
00024   Serial.begin(XPLDIRECT_BAUDRATE);
00025   XP.begin("Sample");
00026
00027   // Register Command for the Button
00028   btnStart.setCommand(F("sim/starters/engage_starter_1"));
00029
00030   // Register Commands for Encoder Up/Down/Push function.
00031   encHeading.setCommand(F("sim/autopilot/heading_up"),
00032                         F("sim/autopilot/heading_down"),
00033                         F("sim/autopilot/heading_sync"));
00034
00035   // Gegister Commands for Switch On and Off transitions. Commands are sent when Switch is moved
00036   swStrobe.setCommand(F("sim/lights/strobe_lights_on"),
00037                       F("sim/lights/strobe_lights_off"));
00038
00039   // Register a DataRef for the strobe light. Read only from XP, 100ms minimum Cycle time, no divider
00040   XP.registerDataRef(F("sim/cockpit/electrical/strobe_lights_on"),
00041                      XPL_READ, 100, 0, &strobe);
00042 }
00043
00044 // Arduino loop function, called cyclic
00045 void loop() {
00046   // Handle XPlane interface
00047   XP.xloop();
00048
00049   // handle all devices and automatically process commandsin background
00050   btnStart.handleXP();
00051   encHeading.handleXP();
00052   swStrobe.handleXP();
00053
00054   // Show the status of the Strobe on the internal LED
00055   digitalWrite(LED_BUILTIN, (strobe > 0));
00056 }
```

6.2 MUX inputs/main.cpp

```
00001 #include <Arduino.h>
00002 #include <XPLDevices.h>
00003
00004 // The XPLDirect library is automatically installed by PlatformIO with XPLDevices
00005 // Optional defines for XPLDirect can be set in platformio.ini
00006 // This sample contains all the important defines. Modify or remove as needed
00007
00008 // This sample shows how to use 74HC4067 Multiplexers for the inputs as commonly used by SimVim
00009
00010 // A simple Pushbutton on MUX0 pin 0
00011 Button btnStart(0, 0);
00012
00013 // An Encoder with push functionality. MUX1 pin 8&9 are the encoder pins, 10 the push pin.
00014 // configured for an Encoder with 4 counts per mechanical notch, which is the standard
00015 Encoder encHeading(1, 8, 9, 10, enc4Pulse);
00016
00017 // A simple On/Off switch on MUX0, pin 15
00018 Switch swStrobe(0, 15);
00019
00020 // A Variable to be connected to a DataRef
00021 long strobe;
00022
00023 // Arduino setup function, called once
00024 void setup() {
00025   // setup interface
00026   Serial.begin(XPLDIRECT_BAUDRATE);
00027   XP.begin("Sample");
00028
00029   // Connect MUX adress pins to Pin 22-25 (SimVim Pins)
00030   DigitalIn.setMux(22, 23, 24, 25);
00031   // Logical MUX0 on Pin 38
00032   DigitalIn.addMux(38);
00033   // Logical MUX1 on Pin 39
00034   DigitalIn.addMux(39);
00035
00036   // Register Command for the Button
00037   btnStart.setCommand(F("sim/starters/engage_starter_1"));
00038
00039   // Register Commands for Encoder Up/Down/Push function.
00040   encHeading.setCommand(F("sim/autopilot/heading_up"),
00041                         F("sim/autopilot/heading_down"),
00042                         F("sim/autopilot/heading_sync"));
00043
00044   // Gegister Commands for Switch On and Off transitions. Commands are sent when Switch is moved
00045   swStrobe.setCommand(F("sim/lights/strobe_lights_on"),
00046                       F("sim/lights/strobe_lights_off"));
00047
00048   // Register a DataRef for the strobe light. Read only from XP, 100ms minimum Cycle time, no divider
00049   XP.registerDataRef(F("sim/cockpit/electrical/strobe_lights_on"),
00050                      XPL_READ, 100, 0, &strobe);
00051 }
00052
00053 // Arduino loop function, called cyclic
00054 void loop() {
00055   // Handle XPlane interface
00056   XP.xloop();
00057
00058   // handle all devices and automatically process commandsin background
00059   btnStart.handleXP();
00060   encHeading.handleXP();
00061   swStrobe.handleXP();
00062
00063   // Show the status of the Strobe on the internal LED
00064   digitalWrite(LED_BUILTIN, (strobe > 0));
00065 }
```

6.3 AnalogIn.h

```
00001 #ifndef AnalogIn_h
00002 #define AnalogIn_h
00003 #include <Arduino.h>
00004
00005 #define AD_RES 10
00006
00007 enum Analog_t
00008 {
00009   unipolar,
00010   bipolar
00011 };
00012
00014 class AnalogIn
```

```
00015 {
00016 public:
00020   AnalogIn(uint8_t pin, Analog_t type);
00021
00026   AnalogIn(uint8_t pin, Analog_t type, float timeConst);
00027
00029   void handle();
00030
00033   float value() { return _value; };
00034
00037   int raw();
00038
00041   void calibrate();
00042
00047   void setRange(uint16_t min, uint16_t max);
00048
00051   void setScale(float scale);
00052
00053 private:
00054   void _calcScales();
00055   float _value;
00056   float _filterConst;
00057   float _scale;
00058   float _scalePos;
00059   float _scaleNeg;
00060   uint16_t _offset;
00061   uint16_t _min;
00062   uint16_t _max;
00063   uint8_t _pin;
00064   Analog_t _type;
00065 };
00066
00067 #endif
```

6.4 Button.h

```
00001 #ifndef Button_h
00002 #define Button_h
00003 #include <Arduino.h>
00004 #include <DigitalIn.h>
00005
00008 class Button
00009 {
00010 private:
00011   void _handle(bool input);
00012
00013 public:
00017   Button(uint8_t mux, uint8_t muxpin);
00018
00021   Button(uint8_t pin) : Button(NOT_USED, pin){};
00022
00024   void handle()                 { _handle(true); };
00025
00028   void handle(bool input)       { _handle(input); };
00029
00031   void handleXP()               { _handle(true); processCommand(); };
00032
00035   void handleXP(bool input)     { _handle(input); processCommand(); };
00036
00039   bool pressed()                { return _transition == transPressed  ? (_transition = transNone,
      true) : false; };
00040
00043   bool released()               { return _transition == transReleased ? (_transition = transNone,
      true) : false; };
00044
00047   bool engaged()                { return _state > 0; };
00048
00051   void setCommand(int cmdPush);
00052
00055   void setCommand(XPString_t *cmdNamePush);
00056
00059   int getCommand()              { return _cmdPush; };
00060
00062   void processCommand();
00063
00064 protected:
00065   enum
00066   {
00067     transNone,
00068     transPressed,
00069     transReleased
00070   };
00071   uint8_t _mux;
```

```
00072   uint8_t _pin;
00073   uint8_t _state;
00074   uint8_t _transition;
00075   int _cmdPush;
00076 };
00077
00081 class RepeatButton : public Button
00082 {
00083 private:
00084   void _handle(bool input);
00085
00086 public:
00091   RepeatButton(uint8_t mux, uint8_t muxpin, uint32_t delay);
00092
00096   RepeatButton(uint8_t pin, uint32_t delay) : RepeatButton(NOT_USED, pin, delay){};
00097
00099   void handle()                 { _handle(true); };
00100
00103   void handle(bool input)       { _handle(input); };
00104
00106   void handleXP()               { _handle(true); processCommand(); };
00107
00110   void handleXP(bool input)     { _handle(input); processCommand(); };
00111
00112 protected:
00113   uint32_t _delay;
00114   uint32_t _timer;
00115 };
00116
00117 #endif
```

6.5 DigitalIn.h

```
00001 #ifndef DigitalIn_h
00002 #define DigitalIn_h
00003 #include <Arduino.h>
00004
00006 #ifndef MUX_MAX_NUMBER
00007 #define MUX_MAX_NUMBER 6
00008 #endif
00009
00011 #ifndef MCP_MAX_NUMBER
00012 #define MCP_MAX_NUMBER 0
00013 #endif
00014
00015 // Include i2c lib only when needed
00016 #if MCP_MAX_NUMBER > 0
00017 #include <Adafruit_MCP23X17.h>
00018 #endif
00019
00020 #define NOT_USED 255
00021
00024 class DigitalIn_
00025 {
00026 public:
00028   DigitalIn_();
00029
00035   void setMux(uint8_t s0, uint8_t s1, uint8_t s2, uint8_t s3);
00036
00040   bool addMux(uint8_t pin);
00041
00042 #if MCP_MAX_NUMBER > 0
00046   bool addMCP(uint8_t adress);
00047 #endif
00048
00053   bool getBit(uint8_t expander, uint8_t channel);
00054
00056   void handle();
00057 private:
00058   uint8_t _s0, _s1, _s2, _s3;
00059 #ifdef ARDUINO_ARCH_AVR
00060   uint8_t _s0port, _s1port, _s2port, _s3port;
00061   uint8_t _s0mask, _s1mask, _s2mask, _s3mask;
00062 #endif
00063   uint8_t _numPins;
00064   uint8_t _pin[MUX_MAX_NUMBER + MCP_MAX_NUMBER];
00065   int16_t _data[MUX_MAX_NUMBER + MCP_MAX_NUMBER];
00066 #if MCP_MAX_NUMBER > 0
00067   uint8_t _numMCP;
00068   Adafruit_MCP23X17 _mcp[MCP_MAX_NUMBER];
00069 #endif
00070 };
00071
```

```
00073 extern DigitalIn_ DigitalIn;
00074
00075 #endif
```

6.6 Encoder.h

```
00001 #ifndef Encoder_h
00002 #define Encoder_h
00003 #include <Arduino.h>
00004 #include <DigitalIn.h>
00005
00006 enum EncCmd_t
00007 {
00008   encCmdUp,
00009   encCmdDown,
00010   encCmdPush
00011 };
00012
00013 enum EncPulse_t
00014 {
00015   enc1Pulse = 1,
00016   enc2Pulse = 2,
00017   enc4Pulse = 4
00018 };
00019
00022 class Encoder
00023 {
00024 public:
00031   Encoder(uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses);
00032
00038   Encoder(uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses) : Encoder(NOT_USED, pin1, pin2,
      pin3, pulses) {}
00039
00041   void handle();
00042
00044   void handleXP()    { handle(); processCommand(); };
00045
00048   int16_t pos()      { return _count; };
00049
00052   bool up()          { return _count >= _pulses ? (_count -= _pulses, true) : false; };
00053
00056   bool down()        { return _count <= -_pulses ? (_count += _pulses, true) : false; };
00057
00060   bool pressed()     { return _transition == transPressed  ? (_transition = transNone, true) : false;
      };
00061
00064   bool released()    { return _transition == transReleased ? (_transition = transNone, true) : false;
      };
00065
00068   bool engaged()     { return _state > 0; };
00069
00074   void setCommand(int cmdUp, int cmdDown, int cmdPush);
00075
00080   void setCommand(XPString_t *cmdNameUp, XPString_t *cmdNameDown, XPString_t *cmdNamePush);
00081
00085   void setCommand(int cmdUp, int cmdDown);
00086
00090   void setCommand(XPString_t *cmdNameUp, XPString_t *cmdNameDown);
00091
00095   int getCommand(EncCmd_t cmd);
00096
00098   void processCommand();
00099 private:
00100   enum
00101   {
00102     transNone,
00103     transPressed,
00104     transReleased
00105   };
00106   uint8_t _mux;
00107   uint8_t _pin1, _pin2, _pin3;
00108   int8_t _count;
00109   uint8_t _pulses;
00110   uint8_t _state;
00111   uint8_t _debounce;
00112   uint8_t _transition;
00113   int _cmdUp;
00114   int _cmdDown;
00115   int _cmdPush;
00116 };
00117
00118 #endif
```

6.7 LedShift.h

```
00001 #ifndef LedShift_h
00002 #define LedShift_h
00003 #include <Arduino.h>
00004
00006 enum led_t
00007 {
00009   ledOff = 0x00,
00011   ledFast = 0x01,
00013   ledMedium = 0x02,
00015   ledSlow = 0x04,
00017   ledOn = 0x08
00018 };
00019
00021 class LedShift
00022 {
00023 public:
00029   LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins = 16);
00030
00034   void setPin(uint8_t pin, led_t mode);
00035   void set(uint8_t pin, led_t mode) { setPin(pin, mode); }; // obsolete
00036
00039   void setAll(led_t mode);
00040   void set_all(led_t mode) { setAll(mode); }; // obsolete
00041
00043   void handle();
00044
00045 private:
00046   void _send();
00047   uint8_t _pin_DAI;
00048   uint8_t _pin_DCK;
00049   uint8_t _pin_LAT;
00050   uint8_t _pins;
00051   led_t _mode[64];
00052   uint8_t _count;
00053   unsigned long _timer;
00054   bool _update;
00055 };
00056
00057 #endif
```

6.8 ShiftOut.h

```
00001 #ifndef ShiftOut_h
00002 #define ShiftOut_h
00003 #include <Arduino.h>
00004
00006 class ShiftOut
00007 {
00008 public:
00014   ShiftOut(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins = 16);
00015
00019   void setPin(uint8_t pin, bool state);
00020
00023   void setAll(bool state);
00024
00026   void handle();
00027
00028 private:
00029   void _send();
00030   uint8_t _pin_DAI;
00031   uint8_t _pin_DCK;
00032   uint8_t _pin_LAT;
00033   uint8_t _pins;
00034   uint8_t _state[8];
00035   bool _update;
00036 };
00037
00038 #endif
```

6.9 Switch.h

```
00001 #ifndef Switch_h
00002 #define Switch_h
00003 #include <Arduino.h>
00004 #include <DigitalIn.h>
00005
00007 class Switch
```

```
00008 {
00009 public:
00013   Switch(uint8_t mux, uint8_t pin);
00014
00017   Switch(uint8_t pin) : Switch (NOT_USED, pin) {};
00018
00020   void handle();
00021
00023   void handleXP() { handle(); processCommand(); };
00024
00027   bool on()       { return _state == switchOn; };
00028
00031   bool off()      { return _state == switchOff; };
00032
00035   void setCommand(int cmdOn);
00036
00039   void setCommand(XPString_t *cmdNameOn);
00040
00044   void setCommand(int cmdOn, int cmdOff);
00045
00049   void setCommand(XPString_t *cmdNameOn, XPString_t *cmdNameOff);
00050
00053   int getCommand();
00054
00056   void processCommand();
00057
00062   float value(float onValue, float offValue) { return on() ? onValue : offValue; };
00063
00064 private:
00065   enum SwState_t
00066   {
00067     switchOff,
00068     switchOn
00069   };
00070   uint8_t _mux;
00071   uint8_t _pin;
00072   uint8_t _debounce;
00073   uint8_t _state;
00074   bool _transition;
00075   int _cmdOff;
00076   int _cmdOn;
00077 };
00078
00080 class Switch2
00081 {
00082 public:
00087   Switch2(uint8_t mux, uint8_t pin1, uint8_t pin2);
00088
00092   Switch2(uint8_t pin1, uint8_t pin2) : Switch2(NOT_USED, pin1, pin2) {}
00093
00095   void handle();
00096
00098   void handleXP() { handle(); processCommand(); };
00099
00102   bool off()      { return _state == switchOff; };
00103
00106   bool on1()      { return _state == switchOn1; };
00107
00110   bool on2()      { return _state == switchOn2; };
00111
00115   void setCommand(int cmdUp, int cmdDown);
00116
00120   void setCommand(XPString_t *cmdNameUp, XPString_t *cmdNameDown);
00121
00126   void setCommand(int cmdOn1, int cmdOff, int cmdOn2);
00127
00132   void setCommand(XPString_t *cmdNameOn1, XPString_t *cmdNameOff, XPString_t *cmdNameOn2);
00133
00136   int getCommand();
00137
00139   void processCommand();
00140
00146   float value(float on1Value, float offValue, float on2Value) { return (on1() ? on1Value : on2() ?
      on2Value : offValue); };
00147
00148 private:
00149   enum SwState_t
00150   {
00151     switchOff,
00152     switchOn1,
00153     switchOn2
00154   };
00155   uint8_t _mux;
00156   uint8_t _pin1;
00157   uint8_t _pin2;
00158   uint8_t _lastState;
00159   uint8_t _debounce;
```

```
00160   uint8_t _state;
00161   bool _transition;
00162   int _cmdOff;
00163   int _cmdOn1;
00164   int _cmdOn2;
00165 };
00166
00167 #endif
```

6.10 Timer.h

```
00001 #ifndef SoftTimer_h
00002 #define SoftTimer_h
00003 #include <Arduino.h>
00004
00006 class Timer
00007 {
00008   public:
00011     Timer(float cycle = 0); // ms
00012
00015     void setCycle(float cycle);
00016
00019     bool elapsed();
00020
00023     float getTime(); // ms
00024
00027     long count();
00028   private:
00029     unsigned long _cycleTime;
00030     unsigned long _lastUpdateTime;
00031     long _count;
00032 };
00033
00034 #endif
```

6.11 XPLDevices.h

```
00001 #ifndef XPLDevices_h
00002 #define XPLDevices_h
00003
00004 #include <XPLDirect.h>
00005 #include <Button.h>
00006 #include <Encoder.h>
00007 #include <Switch.h>
00008 #include <ShiftOut.h>
00009 #include <LedShift.h>
00010 #include <Timer.h>
00011 #include <DigitalIn.h>
00012 #include <AnalogIn.h>
00013
00014 #endif
```

6.12 XPLDirect.h

```
00001 /*
00002   XPLDirect.h - Library for serial interface to Xplane SDK.
00003   Created by Michael Gerlicher,  September 2020.
00004   To report problems, download updates and examples, suggest enhancements or get technical support,
      please visit my patreon page:
00005       www.patreon.com/curiosityworkshop
00006   Stripped down to Minimal Version by mrusk, February 2023
00007 */
00008 #ifndef XPLDirect_h
00009 #define XPLDirect_h
00010 #include <Arduino.h>
00011
00012 #ifndef XPLDIRECT_MAXDATAREFS_ARDUINO
00013 #define XPLDIRECT_MAXDATAREFS_ARDUINO 100 // This can be changed to suit your needs and capabilities
      of your board.
00014 #endif
00015
00016 #ifndef XPLDIRECT_MAXCOMMANDS_ARDUINO
00017 #define XPLDIRECT_MAXCOMMANDS_ARDUINO 100  // Same here.
00018 #endif
00019
```

```
00020 #define XPLDIRECT_RX_TIMEOUT 500 // after detecting a frame header, how long will we wait to receive
      the rest of the frame.  (default 500)
00021
00022 #ifndef XPLMAX_PACKETSIZE
00023 #define XPLMAX_PACKETSIZE 80  // Probably leave this alone. If you need a few extra bytes of RAM it
      could be reduced, but it needs to
00024                               // be as long as the longest dataref name + 10.  If you are using
      datarefs
00025                               // that transfer strings it needs to be big enough for those too.
      (default 200)
00026 #endif
00027
00028 #ifndef XPL_USE_PROGMEM
00029 #define XPL_USE_PROGMEM 1
00030 #endif
00031
00033 // STOP! Dont change any other defines in this header!
00035
00036 #ifdef XPL_USE_PROGMEM
00037 // use Flash for strings, requires F() macro for strings in all registration calls
00038   typedef const __FlashStringHelper XPString_t;
00039 #else
00040   typedef const char XPString_t;
00041 #endif
00042
00043 #define XPLDIRECT_BAUDRATE 115200   // don't mess with this, it needs to match the plugin which won't
      change
00044 #define XPLDIRECT_PACKETHEADER '<'  // ...or this
00045 #define XPLDIRECT_PACKETTRAILER '>' // ...or this
00046 #define XPLDIRECT_VERSION 2106171   // The plugin will start to verify that a compatible version is
      being used
00047 #define XPLDIRECT_ID 0              // Used for relabled plugins to identify the company.  0 = normal
      distribution version
00048
00049 #define XPLERROR 'E'                    // %s         general error
00050 #define XPLRESPONSE_NAME '0'
00051 #define XPLRESPONSE_DATAREF '3'         // %3.3i%s    dataref handle, dataref name
00052 #define XPLRESPONSE_COMMAND '4'         // %3.3i%s    command handle, command name
00053 #define XPLRESPONSE_VERSION 'V'
00054 #define XPLCMD_PRINTDEBUG '1'
00055 #define XPLCMD_RESET '2'
00056 #define XPLCMD_SPEAK 'S'                // speak string
00057 #define XPLCMD_SENDNAME 'a'
00058 #define XPLREQUEST_REGISTERDATAREF 'b'  // %1.1i%2.2i%5.5i%s RWMode, array index (0 for non array
      datarefs), divider to decrease resolution, dataref name
00059 #define XPLREQUEST_REGISTERCOMMAND 'm'  // just the name of the command to register
00060 #define XPLREQUEST_NOREQUESTS 'c'       // nothing to request
00061 #define XPLREQUEST_REFRESH 'd'          // the plugin will call this once xplane is loaded in order to
      get fresh updates from arduino handles that write
00062 #define XPLCMD_DUMPREGISTRATIONS 'Z'    // for debug purposes only (disabled)
00063 #define XPLCMD_DATAREFUPDATE 'e'
00064 #define XPLCMD_SENDREQUEST 'f'
00065 #define XPLCMD_DEVICEREADY 'g'
00066 #define XPLCMD_DEVICENOTREADY 'h'
00067 #define XPLCMD_COMMANDSTART 'i'
00068 #define XPLCMD_COMMANDEND 'j'
00069 #define XPLCMD_COMMANDTRIGGER 'k' //  %3.3i%3.3i   command handle, number of triggers
00070 #define XPLCMD_SENDVERSION 'v'    // we will respond with current build version
00071 #define XPL_EXITING 'x'           // MG 03/14/2023: xplane sends this to the arduino device during
      normal shutdown of xplane.  It may not happen if xplane crashes.
00072
00073 #define XPL_READ 1
00074 #define XPL_WRITE 2
00075 #define XPL_READWRITE 3
00076
00077 #define XPL_DATATYPE_INT 1
00078 #define XPL_DATATYPE_FLOAT 2
00079 #define XPL_DATATYPE_STRING 3
00080
00081 class XPLDirect
00082 {
00083 public:
00084   XPLDirect(Stream*);
00085   void begin(const char *devicename); // parameter is name of your device for reference
00086   int connectionStatus(void);
00087   int commandTrigger(int commandHandle);                    // triggers specified command 1 time;
00088   int commandTrigger(int commandHandle, int triggerCount);  // triggers specified command triggerCount
      times.
00089   int commandStart(int commandHandle);
00090   int commandEnd(int commandHandle);
00091   int datarefsUpdated();      // returns true if xplane has updated any datarefs since last call to
      datarefsUpdated()
00092   int hasUpdated(int handle); // returns true if xplane has updated this dataref since last call to
      hasUpdated()
00093   int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int
      *value);
00094   int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int
```

```
      *value, int index);
00095   int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float
      *value);
00096   int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float
      *value, int index);
00097   int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, char* value);
00098   int registerCommand(XPString_t *commandName);
00099   int sendDebugMessage(const char *msg);
00100   int sendSpeakMessage(const char* msg);
00101   int allDataRefsRegistered(void);
00102   void sendResetRequest(void);
00103   int xloop(void); // where the magic happens!
00104 private:
00105   void _processSerial();
00106   void _processPacket();
00107   void _sendPacketInt(int command, int handle, long int value); // for ints
00108   void _sendPacketFloat(int command, int handle, float value);  // for floats
00109   void _sendPacketVoid(int command, int handle);                // just a command with a handle
00110   void _sendPacketString(int command, char *str);               // for a string
00111   void _transmitPacket();
00112   void _sendname();
00113   void _sendVersion();
00114   int _getHandleFromFrame();
00115   int _getPayloadFromFrame(long int *);
00116   int _getPayloadFromFrame(float *);
00117   int _getPayloadFromFrame(char *);
00118
00119   Stream *streamPtr;
00120   char *_deviceName;
00121   char _receiveBuffer[XPLMAX_PACKETSIZE];
00122   int _receiveBufferBytesReceived;
00123   char _sendBuffer[XPLMAX_PACKETSIZE];
00124   int _connectionStatus;
00125   int _dataRefsCount;
00126   struct _dataRefStructure
00127   {
00128     int dataRefHandle;
00129     byte dataRefRWType;       // XPL_READ, XPL_WRITE, XPL_READWRITE
00130     byte dataRefVARType;      // XPL_DATATYPE_INT 1, XPL_DATATYPE_FLOAT  2   XPL_DATATYPE_STRING 3
00131     float divider;            // tell the host to reduce resolution by dividing then remultiplying by
      this number to reduce traffic.   (ie .02, .1, 1, 5, 10, 100, 1000 etc)
00132     byte forceUpdate;         // in case xplane plugin asks for a refresh
00133     unsigned long updateRate; // maximum update rate in milliseconds, 0 = every change
00134     unsigned long lastUpdateTime;
00135     XPString_t *dataRefName;
00136     void *latestValue;
00137     union {
00138       long int lastSentIntValue;
00139       float lastSentFloatValue;
00140     };
00141     byte updatedFlag; //  True if xplane has updated this dataref.  Gets reset when we call hasUpdated
      method.
00142     byte arrayIndex;  // for datarefs that speak in arrays
00143   } *_dataRefs[XPLDIRECT_MAXDATAREFS_ARDUINO];
00144   int _commandsCount;
00145   struct _commandStructure
00146   {
00147     int commandHandle;
00148     XPString_t *commandName;
00149   } *_commands[XPLDIRECT_MAXCOMMANDS_ARDUINO];
00150   byte _allDataRefsRegistered; // becomes true if all datarefs have been registered
00151   byte _datarefsUpdatedFlag;   // becomes true if any datarefs have been updated from xplane since
      last call to datarefsUpdated()
00152 };
00153
00155 extern XPLDirect XP;
00156
00157 #endif
```

6.13 AnalogIn.cpp

```
00001 #include <Arduino.h>
00002 #include "AnalogIn.h"
00003
00004 #define FULL_SCALE ((1 « AD_RES) - 1)
00005 #define HALF_SCALE (1 « (AD_RES - 1))
00006
00007 AnalogIn::AnalogIn(uint8_t pin, Analog_t type)
00008 {
00009   _pin = pin;
00010   _filterConst = 1.0;
00011   _scale = 1.0;
00012   _min = 0;
```

```
00013   _max = FULL_SCALE;
00014   _type = type;
00015   pinMode(_pin, INPUT);
00016   if (_type == bipolar)
00017   {
00018     _offset = HALF_SCALE;
00019   }
00020   else
00021   {
00022     _offset = 0;
00023   }
00024   _calcScales();
00025 }
00026
00027 AnalogIn::AnalogIn(uint8_t pin, Analog_t type, float timeConst) : AnalogIn(pin, type)
00028 {
00029   if (timeConst > 0)
00030   {
00031     _filterConst = 1.0 / timeConst;
00032   }
00033 }
00034
00035 void AnalogIn::handle()
00036 {
00037   int _raw = raw();
00038   _value = (_filterConst * _raw * (_raw >= 0 ? _scalePos : _scaleNeg)) + (1.0 - _filterConst) *
      _value;
00039 }
00040
00041 int AnalogIn::raw()
00042 {
00043   return constrain(analogRead(_pin), (int16_t)_min, (int16_t)_max) - _offset;
00044 }
00045
00046 void AnalogIn::calibrate()
00047 {
00048   if (_type == unipolar)
00049   {
00050     return;
00051   }
00052   long sum = 0;
00053   for (int i = 0; i < 64; i++)
00054   {
00055     sum += analogRead(_pin);
00056   }
00057   _offset = (int)(sum / 64);
00058   _calcScales();
00059 }
00060
00061 void AnalogIn::setRange(uint16_t min, uint16_t max)
00062 {
00063   _min = min(min, max);
00064   _max = max(min, max);
00065   if (min == max)
00066   {
00067     _min = 0;
00068     _max = FULL_SCALE;
00069   }
00070   if (_type == unipolar)
00071   {
00072     _offset = _min;
00073   }
00074   else
00075   {
00076     _offset = (_max + _min) / 2;
00077   }
00078   _calcScales();
00079 }
00080
00081 void AnalogIn::setScale(float scale)
00082 {
00083   _scale = scale;
00084   _calcScales();
00085 }
00086
00087 void AnalogIn::_calcScales()
00088 {
00089   if (_type == unipolar)
00090   {
00091     _scalePos = _scale / (float)(_max - _min);
00092     _scaleNeg = 0;
00093   }
00094   else
00095   {
00096     _scalePos = (_offset == _max) ? 0 : _scale / (float)(_max - _offset);
00097     _scaleNeg = (_offset == _min) ? 0 : _scale / (float)(_offset - _min);
00098   }
```

```
00099 }
```

6.14 Button.cpp

```
00001 #include <Arduino.h>
00002 #include <XPLDirect.h>
00003 #include "Button.h"
00004
00005 #ifndef DEBOUNCE_DELAY
00006 #define DEBOUNCE_DELAY 20
00007 #endif
00008
00009 // Buttons
00010 Button::Button(uint8_t mux, uint8_t pin)
00011 {
00012   _mux = mux;
00013   _pin = pin;
00014   _state = 0;
00015   _transition = 0;
00016   _cmdPush = -1;
00017   pinMode(_pin, INPUT_PULLUP);
00018 }
00019
00020 // use additional bit for input masking
00021 void Button::_handle(bool input)
00022 {
00023   if (DigitalIn.getBit(_mux, _pin) && input)
00024   {
00025     if (_state == 0)
00026     {
00027       _state = DEBOUNCE_DELAY;
00028       _transition = transPressed;
00029     }
00030   }
00031   else if (_state > 0)
00032   {
00033     if (--_state == 0)
00034     {
00035       _transition = transReleased;
00036     }
00037   }
00038 }
00039
00040 void Button::setCommand(int cmdPush)
00041 {
00042   _cmdPush = cmdPush;
00043 }
00044
00045 void Button::setCommand(XPString_t *cmdNamePush)
00046 {
00047   _cmdPush = XP.registerCommand(cmdNamePush);
00048 }
00049
00050 void Button::processCommand()
00051 {
00052   if (pressed())
00053   {
00054     XP.commandStart(_cmdPush);
00055   }
00056   if (released())
00057   {
00058     XP.commandEnd(_cmdPush);
00059   }
00060 }
00061
00062 RepeatButton::RepeatButton(uint8_t mux, uint8_t pin, uint32_t delay) : Button(mux, pin)
00063 {
00064   _delay = delay;
00065   _timer = 0;
00066 }
00067
00068 void RepeatButton::_handle(bool input)
00069 {
00070   if (DigitalIn.getBit(_mux, _pin) && input)
00071   {
00072     if (_state == 0)
00073     {
00074       _state = DEBOUNCE_DELAY;
00075       _transition = transPressed;
00076       _timer = millis() + _delay;
00077     }
00078     else if (_delay > 0 && (millis() >= _timer))
00079     {
```

```
00080       _state = DEBOUNCE_DELAY;
00081       _transition = transPressed;
00082       _timer += _delay;
00083     }
00084   }
00085   else if (_state > 0)
00086   {
00087     if (--_state == 0)
00088     {
00089       _transition = transReleased;
00090     }
00091   }
00092 }
```

6.15 DigitalIn.cpp

```
00001 #include <Arduino.h>
00002 #include "DigitalIn.h"
00003
00004 #define MCP_PIN 254
00005
00006 // constructor
00007 DigitalIn_::DigitalIn_()
00008 {
00009   _numPins = 0;
00010   for (uint8_t expander = 0; expander < MUX_MAX_NUMBER; expander++)
00011   {
00012     _pin[expander] = NOT_USED;
00013   }
00014   _s0 = NOT_USED;
00015   _s1 = NOT_USED;
00016   _s2 = NOT_USED;
00017   _s3 = NOT_USED;
00018 }
00019
00020 // configure 74HC4067 adress pins S0-S3
00021 void DigitalIn_::setMux(uint8_t s0, uint8_t s1, uint8_t s2, uint8_t s3)
00022 {
00023   _s0 = s0;
00024   _s1 = s1;
00025   _s2 = s2;
00026   _s3 = s3;
00027   pinMode(_s0, OUTPUT);
00028   pinMode(_s1, OUTPUT);
00029   pinMode(_s2, OUTPUT);
00030   pinMode(_s3, OUTPUT);
00031   #ifdef ARDUINO_ARCH_AVR
00032   _s0port = digitalPinToPort(_s0);
00033   _s1port = digitalPinToPort(_s1);
00034   _s2port = digitalPinToPort(_s2);
00035   _s3port = digitalPinToPort(_s3);
00036   _s0mask = digitalPinToBitMask(_s0);
00037   _s1mask = digitalPinToBitMask(_s1);
00038   _s2mask = digitalPinToBitMask(_s2);
00039   _s3mask = digitalPinToBitMask(_s3);
00040   #endif
00041 }
00042
00043 // Add a 74HC4067
00044 bool DigitalIn_::addMux(uint8_t pin)
00045 {
00046   if (_numPins >= MUX_MAX_NUMBER)
00047   {
00048     return false;
00049   }
00050   _pin[_numPins++] = pin;
00051   pinMode(pin, INPUT);
00052   return true;
00053 }
00054
00055 #if MCP_MAX_NUMBER > 0
00056 // Add a MCP23017
00057 bool DigitalIn_::addMCP(uint8_t adress)
00058 {
00059   if (_numMCP >= MCP_MAX_NUMBER)
00060   {
00061     return false;
00062   }
00063   if (!_mcp[_numMCP].begin_I2C(adress, &Wire))
00064   {
00065     return false;
00066   }
00067   for (int i = 0; i < 16; i++)
```

```
00068   {
00069     // TODO: register write iodir = 0xffff, ipol = 0xffff, gppu = 0xffff
00070     _mcp[_numMCP].pinMode(i, INPUT_PULLUP);
00071   }
00072   _numMCP++;
00073   _pin[_numPins++] = MCP_PIN;
00074   return true;
00075 }
00076 #endif
00077
00078 // Gets specific channel from expander, number according to initialization order
00079 bool DigitalIn_::getBit(uint8_t expander, uint8_t channel)
00080 {
00081   if (expander == NOT_USED)
00082   {
00083   #ifdef ARDUINO_ARCH_AVR
00084     return (*portInputRegister(digitalPinToPort(channel)) & digitalPinToBitMask(channel)) ? false :
      true;
00085   #else
00086     return !digitalRead(channel);
00087   #endif
00088   }
00089   return bitRead(_data[expander], channel);
00090 }
00091
00092 // read all inputs together -> base for board specific optimization by using byte read
00093 void DigitalIn_::handle()
00094 {
00095   // only if Mux Pins present
00096 #if MCP_MAX_NUMBER > 0
00097   if (_numPins > _numMCP)
00098 #else
00099   if (_numPins > 0)
00100 #endif
00101   {
00102     for (uint8_t channel = 0; channel < 16; channel++)
00103     {
00104 #ifdef ARDUINO_ARCH_AVR
00105       uint8_t oldSREG = SREG;
00106       noInterrupts();
00107       bitRead(channel, 0) ? *portOutputRegister(_s0port) |= _s0mask : *portOutputRegister(_s0port) &=
      ~_s0mask;
00108       bitRead(channel, 1) ? *portOutputRegister(_s1port) |= _s1mask : *portOutputRegister(_s1port) &=
      ~_s1mask;
00109       bitRead(channel, 2) ? *portOutputRegister(_s2port) |= _s2mask : *portOutputRegister(_s2port) &=
      ~_s2mask;
00110       bitRead(channel, 3) ? *portOutputRegister(_s3port) |= _s3mask : *portOutputRegister(_s3port) &=
      ~_s3mask;
00111       SREG = oldSREG;
00112       delayMicroseconds(1);
00113 #else
00114       digitalWrite(_s0, bitRead(channel, 0));
00115       digitalWrite(_s1, bitRead(channel, 1));
00116       digitalWrite(_s2, bitRead(channel, 2));
00117       digitalWrite(_s3, bitRead(channel, 3));
00118 #endif
00119       for (uint8_t expander = 0; expander < _numPins; expander++)
00120       {
00121         if (_pin[expander] != MCP_PIN)
00122         {
00123 #ifdef ARDUINO_ARCH_AVR
00124           bitWrite(_data[expander], channel, (*portInputRegister(digitalPinToPort(_pin[expander])) &
      digitalPinToBitMask(_pin[expander])) ? false : true);
00125 #else
00126           bitWrite(_data[expander], channel, !digitalRead(_pin[expander]));
00127 #endif
00128         }
00129       }
00130     }
00131   }
00132 #if MCP_MAX_NUMBER > 0
00133   int mcp = 0;
00134   for (uint8_t expander = 0; expander < _numPins; expander++)
00135   {
00136     if (_pin[expander] == MCP_PIN)
00137     {
00138       _data[expander] = ~_mcp[mcp++].readGPIOAB();
00139     }
00140   }
00141 #endif
00142 }
00143
00144 DigitalIn_ DigitalIn;
```

6.16 Encoder.cpp

```
00001 #include <Arduino.h>
00002 #include <XPLDirect.h>
00003 #include "Encoder.h"
00004
00005 #ifndef DEBOUNCE_DELAY
00006 #define DEBOUNCE_DELAY 20
00007 #endif
00008
00009 // Encoder with button functionality on MUX
00010 Encoder::Encoder(uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses)
00011 {
00012   _mux = mux;
00013   _pin1 = pin1;
00014   _pin2 = pin2;
00015   _pin3 = pin3;
00016   _pulses = pulses;
00017   _count = 0;
00018   _state = 0;
00019   _transition = transNone;
00020   _cmdUp = -1;
00021   _cmdDown = -1;
00022   _cmdPush = -1;
00023   pinMode(_pin1, INPUT_PULLUP);
00024   pinMode(_pin2, INPUT_PULLUP);
00025   if (_pin3 != NOT_USED)
00026   {
00027     pinMode(_pin3, INPUT_PULLUP);
00028   }
00029 }
00030
00031 // real time handling
00032 void Encoder::handle()
00033 {
00034   // collect new state
00035   _state = ((_state & 0x03) << 2) | (DigitalIn.getBit(_mux, _pin2) << 1) | (DigitalIn.getBit(_mux,
      _pin1));
00036   // evaluate state change
00037   if (_state == 1 || _state == 7 || _state == 8 || _state == 14)
00038   {
00039     _count++;
00040   }
00041   if (_state == 2 || _state == 4 || _state == 11 || _state == 13)
00042   {
00043     _count--;
00044   }
00045   if (_state == 3 || _state == 12)
00046   {
00047     _count += 2;
00048   }
00049   if (_state == 6 || _state == 9)
00050   {
00051     _count -= 2;
00052   }
00053
00054   // optional button functionality
00055   if (_pin3 != NOT_USED)
00056   {
00057     if (DigitalIn.getBit(_mux, _pin3))
00058     {
00059       if (_debounce == 0)
00060       {
00061         _debounce = DEBOUNCE_DELAY;
00062         _transition = transPressed;
00063       }
00064     }
00065     else if (_debounce > 0)
00066     {
00067       if (--_debounce == 0)
00068       {
00069         _transition = transReleased;
00070       }
00071     }
00072   }
00073 }
00074
00075 void Encoder::setCommand(int cmdUp, int cmdDown, int cmdPush)
00076 {
00077   _cmdUp = cmdUp;
00078   _cmdDown = cmdDown;
00079   _cmdPush = cmdPush;
00080 }
00081
00082 void Encoder::setCommand(XPString_t *cmdNameUp, XPString_t *cmdNameDown, XPString_t *cmdNamePush)
00083 {
00084   _cmdUp = XP.registerCommand(cmdNameUp);
```

```
00085   _cmdDown = XP.registerCommand(cmdNameDown);
00086   _cmdPush = XP.registerCommand(cmdNamePush);
00087 }
00088
00089 void Encoder::setCommand(int cmdUp, int cmdDown)
00090 {
00091   _cmdUp = cmdUp;
00092   _cmdDown = cmdDown;
00093   _cmdPush = -1;
00094 }
00095
00096 void Encoder::setCommand(XPString_t *cmdNameUp, XPString_t *cmdNameDown)
00097 {
00098   _cmdUp = XP.registerCommand(cmdNameUp);
00099   _cmdDown = XP.registerCommand(cmdNameDown);
00100   _cmdPush = -1;
00101 }
00102
00103 int Encoder::getCommand(EncCmd_t cmd)
00104 {
00105   switch (cmd)
00106   {
00107   case encCmdUp:
00108     return _cmdUp;
00109     break;
00110   case encCmdDown:
00111     return _cmdDown;
00112     break;
00113   case encCmdPush:
00114     return _cmdPush;
00115     break;
00116   default:
00117     return -1;
00118     break;
00119   }
00120 }
00121
00122 void Encoder::processCommand()
00123 {
00124   if (up())
00125   {
00126     XP.commandTrigger(_cmdUp);
00127   }
00128   if (down())
00129   {
00130     XP.commandTrigger(_cmdDown);
00131   }
00132   if (_cmdPush >= 0)
00133   {
00134     if (pressed())
00135     {
00136       XP.commandStart(_cmdPush);
00137     }
00138     if (released())
00139     {
00140       XP.commandEnd(_cmdPush);
00141     }
00142   }
00143 }
```

6.17 LedShift.cpp

```
00001 #include <Arduino.h>
00002 #include "LedShift.h"
00003
00004 #define BLINK_DELAY 150
00005
00006 LedShift::LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins)
00007 {
00008   _count = 0;
00009   _timer = millis() + BLINK_DELAY;
00010   _pin_DAI = pin_DAI;
00011   _pin_DCK = pin_DCK;
00012   _pin_LAT = pin_LAT;
00013   _pins = min(pins, 64);
00014   for (int pin = 0; pin < _pins; pin++)
00015   {
00016    _mode[pin] = ledOff;
00017   }
00018   pinMode(_pin_DAI, OUTPUT);
00019   pinMode(_pin_DCK, OUTPUT);
00020   pinMode(_pin_LAT, OUTPUT);
00021   digitalWrite(_pin_DAI, LOW);
```

```
00022   digitalWrite(_pin_DCK, LOW);
00023   digitalWrite(_pin_LAT, LOW);
00024   _send();
00025 }
00026
00027 // send data
00028 void LedShift::_send()
00029 {
00030   // get bit masks
00031   uint8_t dataPort = digitalPinToPort(_pin_DAI);
00032   uint8_t dataMask = digitalPinToBitMask(_pin_DAI);
00033   uint8_t clockPort = digitalPinToPort(_pin_DCK);
00034   uint8_t clockMask = digitalPinToBitMask(_pin_DCK);
00035   uint8_t oldSREG = SREG;
00036   noInterrupts();
00037   uint8_t val = _count | 0x08;
00038   for (uint8_t pin = _pins; pin-- > 0;)
00039   {
00040     (_mode[pin] & val) > 0 ? *portOutputRegister(dataPort) |= dataMask : *portOutputRegister(dataPort)
      &= ~dataMask;
00041     *portOutputRegister(clockPort) |= clockMask;
00042     *portOutputRegister(clockPort) &= ~clockMask;
00043   }
00044   // latch LAT signal
00045   clockPort = digitalPinToPort(_pin_LAT);
00046   clockMask = digitalPinToBitMask(_pin_LAT);
00047   *portOutputRegister(clockPort) |= clockMask;
00048   *portOutputRegister(clockPort) &= ~clockMask;
00049   SREG = oldSREG;
00050 }
00051
00052 void LedShift::setPin(uint8_t pin, led_t mode)
00053 {
00054   if (pin < _pins)
00055   {
00056     if (_mode[pin] != mode)
00057     {
00058       _mode[pin] = mode;
00059       _update = true;
00060     }
00061   }
00062 }
00063
00064 void LedShift::setAll(led_t mode)
00065 {
00066   for (int pin = 0; pin < _pins; pin++)
00067   {
00068     _mode[pin] = mode;
00069   }
00070   _update = true;
00071 }
00072
00073 void LedShift::handle()
00074 {
00075   if (millis() >= _timer)
00076   {
00077     _timer += BLINK_DELAY;
00078     _count = (_count + 1) & 0x07;
00079     _update = true;
00080   }
00081   if (_update)
00082   {
00083     _send();
00084     _update = false;
00085   }
00086 }
```

6.18 ShiftOut.cpp

```
00001 #include <Arduino.h>
00002 #include "ShiftOut.h"
00003
00004 ShiftOut::ShiftOut(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins)
00005 {
00006   _pin_DAI = pin_DAI;
00007   _pin_DCK = pin_DCK;
00008   _pin_LAT = pin_LAT;
00009   _pins = min(pins, 64);
00010   pinMode(_pin_DAI, OUTPUT);
00011   pinMode(_pin_DCK, OUTPUT);
00012   pinMode(_pin_LAT, OUTPUT);
00013   digitalWrite(_pin_DAI, LOW);
00014   digitalWrite(_pin_DCK, LOW);
```

```
00015   digitalWrite(_pin_LAT, LOW);
00016   _send();
00017 }
00018
00019 // send data
00020 void ShiftOut::_send()
00021 {
00022   // get bit masks
00023   uint8_t dataPort = digitalPinToPort(_pin_DAI);
00024   uint8_t dataMask = digitalPinToBitMask(_pin_DAI);
00025   uint8_t clockPort = digitalPinToPort(_pin_DCK);
00026   uint8_t clockMask = digitalPinToBitMask(_pin_DCK);
00027   uint8_t oldSREG = SREG;
00028   noInterrupts();
00029   for (uint8_t pin = _pins; pin-- > 0;)
00030   {
00031     bitRead(_state[pin » 3], pin & 0x07) ? *portOutputRegister(dataPort) |= dataMask :
      *portOutputRegister(dataPort) &= ~dataMask;
00032     *portOutputRegister(clockPort) |= clockMask;
00033     *portOutputRegister(clockPort) &= ~clockMask;
00034   }
00035   // latch LAT signal
00036   clockPort = digitalPinToPort(_pin_LAT);
00037   clockMask = digitalPinToBitMask(_pin_LAT);
00038   *portOutputRegister(clockPort) |= clockMask;
00039   *portOutputRegister(clockPort) &= ~clockMask;
00040   SREG = oldSREG;
00041 }
00042
00043 void ShiftOut::setPin(uint8_t pin, bool state)
00044 {
00045   if (pin < _pins)
00046   {
00047     if (state != bitRead(_state[pin » 3], pin & 0x07))
00048     {
00049       bitWrite(_state[pin » 3], pin & 0x07, state);
00050       _update = true;
00051     }
00052   }
00053 }
00054
00055 void ShiftOut::setAll(bool state)
00056 {
00057   for (int pin = 0; pin < _pins; pin++)
00058   {
00059     bitWrite(_state[pin » 3], pin & 0x07, state);
00060   }
00061   _update = true;
00062 }
00063
00064 void ShiftOut::handle()
00065 {
00066   if (_update)
00067   {
00068     _send();
00069     _update = false;
00070   }
00071 }
```

6.19 Switch.cpp

```
00001 #include <Arduino.h>
00002 #include <XPLDirect.h>
00003 #include "Switch.h"
00004
00005 #ifndef DEBOUNCE_DELAY
00006 #define DEBOUNCE_DELAY 20
00007 #endif
00008
00009 Switch::Switch(uint8_t mux, uint8_t pin)
00010 {
00011   _mux = mux;
00012   _pin = pin;
00013   _state = switchOff;
00014   _cmdOn = -1;
00015   _cmdOff = -1;
00016   pinMode(_pin, INPUT_PULLUP);
00017 }
00018
00019 void Switch::handle()
00020 {
00021   if (_debounce > 0)
00022   {
```

```
00023     _debounce--;
00024   }
00025   else
00026   {
00027     SwState_t input = switchOff;
00028     if (DigitalIn.getBit(_mux, _pin))
00029     {
00030       input = switchOn;
00031     }
00032     if (input != _state)
00033     {
00034       _debounce = DEBOUNCE_DELAY;
00035       _state = input;
00036       _transition = true;
00037     }
00038   }
00039 }
00040
00041 void Switch::setCommand(int cmdOn)
00042 {
00043   _cmdOn = cmdOn;
00044   _cmdOff = -1;
00045 }
00046
00047 void Switch::setCommand(XPString_t *cmdNameOn)
00048 {
00049   _cmdOn = XP.registerCommand(cmdNameOn);
00050   _cmdOff = -1;
00051 }
00052
00053 void Switch::setCommand(int cmdOn, int cmdOff)
00054 {
00055   _cmdOn = cmdOn;
00056   _cmdOff = cmdOff;
00057 }
00058
00059 void Switch::setCommand(XPString_t *cmdNameOn, XPString_t *cmdNameOff)
00060 {
00061   _cmdOn = XP.registerCommand(cmdNameOn);
00062   _cmdOff = XP.registerCommand(cmdNameOff);
00063 }
00064
00065 int Switch::getCommand()
00066 {
00067   switch (_state)
00068   {
00069   case switchOff:
00070     return _cmdOff;
00071     break;
00072   case switchOn:
00073     return _cmdOn;
00074     break;
00075   default:
00076     return -1;
00077     break;
00078   }
00079 }
00080
00081 void Switch::processCommand()
00082 {
00083   if (_transition)
00084   {
00085     int cmd = getCommand();
00086     if (cmd >= 0)
00087     {
00088       XP.commandTrigger(getCommand());
00089     }
00090     _transition = false;
00091   }
00092 }
00093
00094 // Switch 2
00095
00096 Switch2::Switch2(uint8_t mux, uint8_t pin1, uint8_t pin2)
00097 {
00098   _mux = mux;
00099   _pin1 = pin1;
00100   _pin2 = pin2;
00101   _state = switchOff;
00102   _cmdOff = -1;
00103   _cmdOn1 = -1;
00104   _cmdOn2 = -1;
00105   if (_mux == NOT_USED)
00106   {
00107     pinMode(_pin1, INPUT_PULLUP);
00108     pinMode(_pin2, INPUT_PULLUP);
00109   }
```

```
00110 }
00111
00112 void Switch2::handle()
00113 {
00114   if (_debounce > 0)
00115   {
00116     _debounce--;
00117   }
00118   else
00119   {
00120     SwState_t input = switchOff;
00121     if (DigitalIn.getBit(_mux, _pin1))
00122     {
00123       input = switchOn1;
00124     }
00125     else if (DigitalIn.getBit(_mux, _pin2))
00126     {
00127       input = switchOn2;
00128     }
00129     if (input != _state)
00130     {
00131       _debounce = DEBOUNCE_DELAY;
00132       _lastState = _state;
00133       _state = input;
00134       _transition = true;
00135     }
00136   }
00137 }
00138
00139 void Switch2::setCommand(int cmdUp, int cmdDown)
00140 {
00141   _cmdOn1 = cmdUp;
00142   _cmdOff = cmdDown;
00143   _cmdOn2 = -1;
00144 }
00145
00146 void Switch2::setCommand(XPString_t *cmdNameUp, XPString_t *cmdNameDown)
00147 {
00148   _cmdOn1 = XP.registerCommand(cmdNameUp);
00149   _cmdOff = XP.registerCommand(cmdNameDown);
00150   _cmdOn2 = -1;
00151 }
00152
00153 void Switch2::setCommand(int cmdOn1, int cmdOff, int cmdOn2)
00154 {
00155   _cmdOn1 = cmdOn1;
00156   _cmdOff = cmdOff;
00157   _cmdOn2 = cmdOn2;
00158 }
00159
00160 void Switch2::setCommand(XPString_t *cmdNameOn1, XPString_t *cmdNameOff, XPString_t *cmdNameOn2)
00161 {
00162   _cmdOn1 = XP.registerCommand(cmdNameOn1);
00163   _cmdOff = XP.registerCommand(cmdNameOff);
00164   _cmdOn2 = XP.registerCommand(cmdNameOn2);
00165 }
00166
00167 int Switch2::getCommand()
00168 {
00169   if (_cmdOn2 == -1)
00170   {
00171     if (_state == switchOn1)
00172     {
00173       return _cmdOn1;
00174     }
00175     if (_state == switchOff && _lastState == switchOn1)
00176     {
00177       return _cmdOff;
00178     }
00179     if (_state == switchOn2)
00180     {
00181       return _cmdOff;
00182     }
00183     if (_state == switchOff && _lastState == switchOn2)
00184     {
00185       return _cmdOn1;
00186     }
00187   }
00188   else
00189   {
00190     if (_state == switchOn1)
00191     {
00192       return _cmdOn1;
00193     }
00194     if (_state == switchOff)
00195     {
00196       return _cmdOff;
```

```
00197     }
00198     if (_state == switchOn2)
00199     {
00200       return _cmdOn2;
00201     }
00202   }
00203   return -1;
00204 }
00205
00206 void Switch2::processCommand()
00207 {
00208   if (_transition)
00209   {
00210     XP.commandTrigger(getCommand());
00211     _transition = false;
00212   }
00213 }
```

6.20 Timer.cpp

```
00001 #include <Arduino.h>
00002 #include "Timer.h"
00003
00004 Timer::Timer(float cycle)
00005 {
00006   setCycle(cycle);
00007   _lastUpdateTime = micros();
00008 }
00009
00010 void Timer::setCycle(float cycle)
00011 {
00012   _cycleTime = (unsigned long)(cycle * 1000.0);
00013 }
00014
00015 bool Timer::elapsed()
00016 {
00017   _count++;
00018   unsigned long now = micros();
00019   if (now > _lastUpdateTime + _cycleTime)
00020   {
00021    _lastUpdateTime = now;
00022    return true;
00023   }
00024   return false;
00025 }
00026
00027 float Timer::getTime()
00028 {
00029   unsigned long now = micros();
00030   unsigned long cycle = now - _lastUpdateTime;
00031   _lastUpdateTime = now;
00032   return (float)cycle * 0.001;
00033 }
00034
00035 long Timer::count()
00036 {
00037   long ret = _count;
00038   _count = 0;
00039   return ret;
00040 }
```

6.21 XPLDirect.cpp

```
00001 /*
00002   XPLDirect.cpp
00003   Created by Michael Gerlicher, September 2020.
00004   Modified by mrusk, March 2023
00005 */
00006
00007 #include <Arduino.h>
00008 #include "XPLDirect.h"
00009
00010 // Methods
00011 XPLDirect::XPLDirect(Stream* device)
00012 {
00013   streamPtr = device;
00014   streamPtr->setTimeout(XPLDIRECT_RX_TIMEOUT);
00015 }
00016
```

```
00017 void XPLDirect::begin(const char *devicename)
00018 {
00019   _deviceName = (char *)devicename;
00020   _connectionStatus = 0;
00021   _dataRefsCount = 0;
00022   _commandsCount = 0;
00023   _allDataRefsRegistered = 0;
00024   _receiveBuffer[0] = 0;
00025 }
00026
00027 int XPLDirect::xloop(void)
00028 {
00029   _processSerial();
00030   if (!_allDataRefsRegistered)
00031   {
00032     return _connectionStatus;
00033   }
00034   // process datarefs to send
00035   for (int i = 0; i < _dataRefsCount; i++)
00036   {
00037     if (_dataRefs[i]->dataRefHandle >= 0 && (_dataRefs[i]->dataRefRWType == XPL_WRITE ||
      _dataRefs[i]->dataRefRWType == XPL_READWRITE))
00038     {
00039       if ((millis() - _dataRefs[i]->lastUpdateTime > _dataRefs[i]->updateRate) ||
      _dataRefs[i]->forceUpdate)
00040       {
00041         switch (_dataRefs[i]->dataRefVARType)
00042         {
00043         case XPL_DATATYPE_INT:
00044           if (*(long int *)_dataRefs[i]->latestValue != _dataRefs[i]->lastSentIntValue)
00045           {
00046             _sendPacketInt(XPLCMD_DATAREFUPDATE, _dataRefs[i]->dataRefHandle, *(long int
      *)_dataRefs[i]->latestValue);
00047             _dataRefs[i]->lastSentIntValue = *(long int *)_dataRefs[i]->latestValue;
00048             _dataRefs[i]->lastUpdateTime = millis();
00049             _dataRefs[i]->forceUpdate = 0;
00050           }
00051           break;
00052         case XPL_DATATYPE_FLOAT:
00053           if (_dataRefs[i]->divider > 0)
00054           {
00055             *(float *)_dataRefs[i]->latestValue = ((int)(*(float *)_dataRefs[i]->latestValue /
      _dataRefs[i]->divider) * _dataRefs[i]->divider);
00056           }
00057           if (*(float *)_dataRefs[i]->latestValue != _dataRefs[i]->lastSentFloatValue)
00058           {
00059             _sendPacketFloat(XPLCMD_DATAREFUPDATE, _dataRefs[i]->dataRefHandle, *(float
      *)_dataRefs[i]->latestValue);
00060             _dataRefs[i]->lastSentFloatValue = *(float *)_dataRefs[i]->latestValue;
00061             _dataRefs[i]->lastUpdateTime = millis();
00062             _dataRefs[i]->forceUpdate = 0;
00063           }
00064           break;
00065         }
00066       }
00067     }
00068   }
00069   return _connectionStatus;
00070 }
00071
00072 int XPLDirect::commandTrigger(int commandHandle)
00073 {
00074   if (commandHandle < 0 || commandHandle >= _commandsCount)
00075   { // invalid handle
00076     return -1;
00077   }
00078   if (!_commands[commandHandle])
00079   { // inactive command
00080     return -1;
00081   }
00082 #if XPL_DEBUG
00083   Serial.print("Command Trigger: ");
00084   Serial.println(_commands[commandHandle]->commandName);
00085 #endif
00086   _sendPacketInt(XPLCMD_COMMANDTRIGGER, _commands[commandHandle]->commandHandle, 1);
00087   return 0;
00088 }
00089
00090 int XPLDirect::commandTrigger(int commandHandle, int triggerCount)
00091 {
00092   if (commandHandle < 0 || commandHandle >= _commandsCount)
00093   { // invalid handle
00094     return -1;
00095   }
00096   if (!_commands[commandHandle])
00097   { // inactive command
00098     return -1;
```

```
00099   }
00100 #if XPL_DEBUG
00101   Serial.print("Command Trigger: ");
00102   Serial.print(_commands[commandHandle]->commandName);
00103   Serial.print(" ");
00104   Serial.print(triggerCount);
00105   Serial.println(" times");
00106 #endif
00107   _sendPacketInt(XPLCMD_COMMANDTRIGGER, _commands[commandHandle]->commandHandle, (long
      int)triggerCount);
00108   return 0;
00109 }
00110
00111 int XPLDirect::commandStart(int commandHandle)
00112 {
00113   if (commandHandle < 0 || commandHandle >= _commandsCount)
00114   { // invalid handle
00115     return -1;
00116   }
00117   if (!_commands[commandHandle])
00118   { // inactive command
00119     return -1;
00120   }
00121 #if XPL_DEBUG
00122   Serial.print("Command Start  : ");
00123   Serial.println(_commands[commandHandle]->commandName);
00124 #endif
00125   _sendPacketVoid(XPLCMD_COMMANDSTART, _commands[commandHandle]->commandHandle);
00126   return 0;
00127 }
00128
00129 int XPLDirect::commandEnd(int commandHandle)
00130 {
00131   if (commandHandle < 0 || commandHandle >= _commandsCount)
00132   { // invalid handle
00133     return -1;
00134   }
00135   if (!_commands[commandHandle])
00136   { // inactive command
00137     return -1;
00138   }
00139 #if XPL_DEBUG
00140   Serial.print("Command End    : ");
00141   Serial.println(_commands[commandHandle]->commandName);
00142 #endif
00143   _sendPacketVoid(XPLCMD_COMMANDEND, _commands[commandHandle]->commandHandle);
00144   return 0;
00145 }
00146
00147 int XPLDirect::connectionStatus()
00148 {
00149   return _connectionStatus;
00150 }
00151
00152 int XPLDirect::sendDebugMessage(const char* msg)
00153 {
00154   _sendPacketString(XPLCMD_PRINTDEBUG, (char *)msg);
00155   return 1;
00156 }
00157
00158 int XPLDirect::sendSpeakMessage(const char* msg)
00159 {
00160   _sendPacketString(XPLCMD_SPEAK, (char *)msg);
00161   return 1;
00162 }
00163
00164 int XPLDirect::hasUpdated(int handle)
00165 {
00166   if (_dataRefs[handle]->updatedFlag)
00167   {
00168     _dataRefs[handle]->updatedFlag = false;
00169     return true;
00170   }
00171   return false;
00172 }
00173
00174 int XPLDirect::datarefsUpdated()
00175 {
00176   if (_datarefsUpdatedFlag)
00177   {
00178     _datarefsUpdatedFlag = false;
00179     return true;
00180   }
00181   return false;
00182 }
00183
00184 void XPLDirect::_sendname()
```

```
00185 {
00186   if (_deviceName != NULL)
00187   {
00188     _sendPacketString(XPLRESPONSE_NAME, _deviceName);
00189   }
00190 }
00191
00192 void XPLDirect::_sendVersion()
00193 {
00194   if (_deviceName != NULL)
00195   {
00196     _sendPacketInt(XPLRESPONSE_VERSION, XPLDIRECT_ID, XPLDIRECT_VERSION);
00197   }
00198 }
00199
00200 void XPLDirect::sendResetRequest()
00201 {
00202   if (_deviceName != NULL)
00203   {
00204     _sendPacketVoid(XPLCMD_RESET, 0);
00205   }
00206 }
00207
00208 void XPLDirect::_processSerial()
00209 {
00210   while (streamPtr->available() && _receiveBuffer[0] != XPLDIRECT_PACKETHEADER)
00211   {
00212     _receiveBuffer[0] = (char)streamPtr->read();
00213   }
00214   if (_receiveBuffer[0] != XPLDIRECT_PACKETHEADER)
00215   {
00216     return;
00217   }
00218   _receiveBufferBytesReceived = streamPtr->readBytesUntil(XPLDIRECT_PACKETTRAILER, (char
      *)&_receiveBuffer[1], XPLMAX_PACKETSIZE - 1);
00219   if (_receiveBufferBytesReceived == 0)
00220   {
00221     _receiveBuffer[0] = 0;
00222     return;
00223   }
00224   _receiveBuffer[++_receiveBufferBytesReceived] = XPLDIRECT_PACKETTRAILER;
00225   _receiveBuffer[++_receiveBufferBytesReceived] = 0; // old habits die hard.
00226   _processPacket();
00227   _receiveBuffer[0] = 0;
00228 }
00229
00230 void XPLDirect::_processPacket()
00231 {
00232   int i;
00233
00234   switch (_receiveBuffer[1])
00235   {
00236   case XPLCMD_RESET:
00237     _connectionStatus = false;
00238     break;
00239
00240   case XPL_EXITING :         // MG 03/14/2023:  Added protocol code so the device will know if xplane
      has shut down normally.
00241     _connectionStatus = false;
00242     break;
00243
00244   case XPLCMD_SENDNAME:
00245     _sendname();
00246     _connectionStatus = true;            // not considered active till you know my name
00247     for (i = 0; i < _dataRefsCount; i++) // also, if name was requested reset active datarefs and
      commands
00248     {
00249       _dataRefs[i]->dataRefHandle = -1; //  invalid again until assigned by Xplane
00250     }
00251     for (i = 0; i < _commandsCount; i++)
00252     {
00253       _commands[i]->commandHandle = -1;
00254     }
00255     break;
00256
00257   case XPLCMD_SENDVERSION:
00258   {
00259     _sendVersion();
00260     break;
00261   }
00262
00263   case XPLRESPONSE_DATAREF:
00264     for (int i = 0; i < _dataRefsCount; i++)
00265     {
00266       if (strncmp_PF((char *)&_receiveBuffer[5], (uint_farptr_t)_dataRefs[i]->dataRefName,
      strlen_PF((uint_farptr_t)_dataRefs[i]->dataRefName)) == 0 && _dataRefs[i]->dataRefHandle == -1)
00267       {
```

```
00268         _dataRefs[i]->dataRefHandle = _getHandleFromFrame(); // parse the refhandle
00269         _dataRefs[i]->updatedFlag = true;
00270         i = _dataRefsCount; // end checking
00271       }
00272     }
00273     break;
00274
00275   case XPLRESPONSE_COMMAND:
00276     for (int i = 0; i < _commandsCount; i++)
00277     {
00278       if (strncmp_PF((char *)&_receiveBuffer[5], (uint_farptr_t)_commands[i]->commandName,
      strlen_PF((uint_farptr_t)_commands[i]->commandName)) == 0 && _commands[i]->commandHandle == -1)
00279       {
00280         _commands[i]->commandHandle = _getHandleFromFrame(); // parse the refhandle
00281         i = _commandsCount;                                  // end checking
00282       }
00283     }
00284     break;
00285
00286   case XPLCMD_SENDREQUEST:
00287   {
00288     int packetSent = 0;
00289     int i = 0;
00290     while (!packetSent && i < _dataRefsCount && i < XPLDIRECT_MAXDATAREFS_ARDUINO) // send dataref
      registrations first
00291     {
00292       if (_dataRefs[i]->dataRefHandle == -1)
00293       { // some boards cant do sprintf with floats so this is a workaround
00294         sprintf(_sendBuffer, "%c%c%1.1i%2.2i%05i.%02i%S%c", XPLDIRECT_PACKETHEADER,
      XPLREQUEST_REGISTERDATAREF, _dataRefs[i]->dataRefRWType, _dataRefs[i]->arrayIndex,
00295                 (int)_dataRefs[i]->divider, (int)(_dataRefs[i]->divider * 100) % 100, (wchar_t
      *)_dataRefs[i]->dataRefName, XPLDIRECT_PACKETTRAILER);
00296         _transmitPacket();
00297         packetSent = 1;
00298       }
00299       i++;
00300     }
00301     i = 0;
00302     while (!packetSent && i < _commandsCount && i < XPLDIRECT_MAXCOMMANDS_ARDUINO) // now send command
      registrations
00303     {
00304       if (_commands[i]->commandHandle == -1)
00305       {
00306         sprintf(_sendBuffer, "%c%c%S%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_REGISTERCOMMAND, (wchar_t
      *)_commands[i]->commandName, XPLDIRECT_PACKETTRAILER);
00307         _transmitPacket();
00308         packetSent = 1;
00309       }
00310       i++;
00311     }
00312     if (!packetSent)
00313     {
00314       _allDataRefsRegistered = true;
00315       sprintf(_sendBuffer, "%c%c%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_NOREQUESTS,
      XPLDIRECT_PACKETTRAILER);
00316       _transmitPacket();
00317     }
00318     break;
00319   }
00320
00321   case XPLCMD_DATAREFUPDATE:
00322   {
00323     int refhandle = _getHandleFromFrame();
00324     for (int i = 0; i < _dataRefsCount; i++)
00325     {
00326       if (_dataRefs[i]->dataRefHandle == refhandle && (_dataRefs[i]->dataRefRWType == XPL_READ ||
      _dataRefs[i]->dataRefRWType == XPL_READWRITE))
00327       {
00328         if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_INT)
00329         {
00330           _getPayloadFromFrame((long int *)_dataRefs[i]->latestValue);
00331           _dataRefs[i]->lastSentIntValue = *(long int *)_dataRefs[i]->latestValue;
00332           _dataRefs[i]->updatedFlag = true;
00333           _datarefsUpdatedFlag = true;
00334         }
00335         if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_FLOAT)
00336         {
00337           _getPayloadFromFrame((float *)_dataRefs[i]->latestValue);
00338           _dataRefs[i]->lastSentFloatValue = *(float *)_dataRefs[i]->latestValue;
00339           _dataRefs[i]->updatedFlag = true;
00340           _datarefsUpdatedFlag = true;
00341         }
00342         if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_STRING)
00343         {
00344           _getPayloadFromFrame((char *)_dataRefs[i]->latestValue);
00345           _dataRefs[i]->updatedFlag = true;
00346           _datarefsUpdatedFlag = true;
```

```
00347         }
00348         i = _dataRefsCount; // skip the rest
00349       }
00350     }
00351     break;
00352   }
00353   case XPLREQUEST_REFRESH:
00354     for (int i = 0; i < _dataRefsCount; i++)
00355     {
00356       if (_dataRefs[i]->dataRefRWType == XPL_WRITE || _dataRefs[i]->dataRefRWType == XPL_READWRITE)
00357       {
00358         _dataRefs[i]->forceUpdate = 1; // bypass noise and timing filters
00359       }
00360     }
00361     break;
00362
00363   default:
00364     break;
00365   }
00366 }
00367
00368 void XPLDirect::_sendPacketInt(int command, int handle, long int value) // for ints
00369 {
00370   if (handle >= 0)
00371   {
00372     sprintf(_sendBuffer, "%c%c%3.3i%ld%c", XPLDIRECT_PACKETHEADER, command, handle, value,
      XPLDIRECT_PACKETTRAILER);
00373     _transmitPacket();
00374   }
00375 }
00376
00377 void XPLDirect::_sendPacketFloat(int command, int handle, float value) // for floats
00378 {
00379   if (handle >= 0)
00380   {
00381     // some boards cant do sprintf with floats so this is a workaround.
00382     char tmp[16];
00383     dtostrf(value, 8, 6, tmp);
00384     sprintf(_sendBuffer, "%c%c%3.3i%s%c", XPLDIRECT_PACKETHEADER, command, handle, tmp,
      XPLDIRECT_PACKETTRAILER);
00385     _transmitPacket();
00386   }
00387 }
00388
00389 void XPLDirect::_sendPacketVoid(int command, int handle) // just a command with a handle
00390 {
00391   if (handle >= 0)
00392   {
00393     sprintf(_sendBuffer, "%c%c%3.3i%c", XPLDIRECT_PACKETHEADER, command, handle,
      XPLDIRECT_PACKETTRAILER);
00394     _transmitPacket();
00395   }
00396 }
00397
00398 void XPLDirect::_sendPacketString(int command, char *str) // for a string
00399 {
00400   sprintf(_sendBuffer, "%c%c%s%c", XPLDIRECT_PACKETHEADER, command, str, XPLDIRECT_PACKETTRAILER);
00401   _transmitPacket();
00402 }
00403
00404 void XPLDirect::_transmitPacket(void)
00405 {
00406   streamPtr->write(_sendBuffer);
00407   if (strlen(_sendBuffer) == 64)
00408   {
00409     streamPtr->print(" "); // apparantly a bug on some boards when we transmit exactly 64 bytes
00410   }
00411 }
00412
00413 int XPLDirect::_getHandleFromFrame() // Assuming receive buffer is holding a good frame
00414 {
00415   char holdChar;
00416   int handleRet;
00417   holdChar = _receiveBuffer[5];
00418   _receiveBuffer[5] = 0;
00419   handleRet = atoi((char *)&_receiveBuffer[2]);
00420   _receiveBuffer[5] = holdChar;
00421   return handleRet;
00422 }
00423
00424 int XPLDirect::_getPayloadFromFrame(long int *value) // Assuming receive buffer is holding a good
      frame
00425 {
00426   char holdChar;
00427   holdChar = _receiveBuffer[15];
00428   _receiveBuffer[15] = 0;
00429   *value = atol((char *)&_receiveBuffer[5]);
```

```
00430   _receiveBuffer[15] = holdChar;
00431   return 0;
00432 }
00433
00434 int XPLDirect::_getPayloadFromFrame(float *value) // Assuming receive buffer is holding a good frame
00435 {
00436   char holdChar;
00437   holdChar = _receiveBuffer[15];
00438   _receiveBuffer[15] = 0;
00439   *value = atof((char *)&_receiveBuffer[5]);
00440   _receiveBuffer[15] = holdChar;
00441   return 0;
00442 }
00443
00444 int XPLDirect::_getPayloadFromFrame(char *value) // Assuming receive buffer is holding a good frame
00445 {
00446   memcpy(value, (char *)&_receiveBuffer[5], _receiveBufferBytesReceived - 6);
00447   value[_receiveBufferBytesReceived - 6] = 0; // erase the packet trailer
00448   for (int i = 0; i < _receiveBufferBytesReceived - 6; i++)
00449   {
00450     if (value[i] == 7)
00451     {
00452       value[i] = XPLDIRECT_PACKETTRAILER; //  How I deal with the possibility of the packet trailer
      being within a string
00453     }
00454   }
00455   return 0;
00456 }
00457
00458 int XPLDirect::allDataRefsRegistered()
00459 {
00460   return _allDataRefsRegistered;
00461 }
00462
00463 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
      long int *value)
00464 {
00465   if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00466   {
00467     return -1; // Error
00468   }
00469   _dataRefs[_dataRefsCount] = new _dataRefStructure;
00470   _dataRefs[_dataRefsCount]->dataRefName = datarefName; // added for F() macro
00471   _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00472   _dataRefs[_dataRefsCount]->divider = divider;
00473   _dataRefs[_dataRefsCount]->updateRate = rate;
00474   _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_INT;
00475   _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00476   _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
00477   _dataRefs[_dataRefsCount]->arrayIndex = 0;     // not used unless we are referencing an array
00478   _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00479   _dataRefsCount++;
00480   _allDataRefsRegistered = 0;
00481   return (_dataRefsCount - 1);
00482 }
00483
00484 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
      long int *value, int index)
00485 {
00486   if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00487   {
00488     return -1;
00489   }
00490   _dataRefs[_dataRefsCount] = new _dataRefStructure;
00491   _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00492   _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00493   _dataRefs[_dataRefsCount]->updateRate = rate;
00494   _dataRefs[_dataRefsCount]->divider = divider;
00495   _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_INT; // arrays are dealt with on the XPlane
      plugin side
00496   _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00497   _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
00498   _dataRefs[_dataRefsCount]->arrayIndex = index; // not used unless we are referencing an array
00499   _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00500   _dataRefsCount++;
00501   _allDataRefsRegistered = 0;
00502   return (_dataRefsCount - 1);
00503 }
00504
00505 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
      float *value)
00506 {
00507   if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00508   {
00509     return -1;
00510   }
00511   _dataRefs[_dataRefsCount] = new _dataRefStructure;
```

```
00512   _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00513   _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00514   _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_FLOAT;
00515   _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00516   _dataRefs[_dataRefsCount]->lastSentFloatValue = -1; // force update on first loop
00517   _dataRefs[_dataRefsCount]->updateRate = rate;
00518   _dataRefs[_dataRefsCount]->divider = divider;
00519   _dataRefs[_dataRefsCount]->arrayIndex = 0;     // not used unless we are referencing an array
00520   _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00521   _dataRefsCount++;
00522   _allDataRefsRegistered = 0;
00523   return (_dataRefsCount - 1);
00524 }
00525
00526 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
      float *value, int index)
00527 {
00528   if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00529   {
00530     return -1;
00531   }
00532   _dataRefs[_dataRefsCount] = new _dataRefStructure;
00533   _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00534   _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00535   _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_FLOAT; // arrays are dealt with on the
      Xplane plugin side
00536   _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00537   _dataRefs[_dataRefsCount]->lastSentFloatValue = 0;
00538   _dataRefs[_dataRefsCount]->updateRate = rate;
00539   _dataRefs[_dataRefsCount]->arrayIndex = index; // not used unless we are referencing an array
00540   _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00541   _dataRefsCount++;
00542   _allDataRefsRegistered = 0;
00543   return (_dataRefsCount - 1);
00544 }
00545
00546 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, char *value)
00547 {
00548   if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00549   {
00550     return -1;
00551   }
00552   _dataRefs[_dataRefsCount] = new _dataRefStructure;
00553   _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00554   _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00555   _dataRefs[_dataRefsCount]->updateRate = rate;
00556   _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_STRING;
00557   _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00558   _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
00559   _dataRefs[_dataRefsCount]->arrayIndex = 0;     // not used unless we are referencing an array
00560   _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00561   _dataRefsCount++;
00562   _allDataRefsRegistered = 0;
00563   return (_dataRefsCount - 1);
00564 }
00565
00566 int XPLDirect::registerCommand(XPString_t *commandName) // user will trigger commands with
      commandTrigger
00567 {
00568   if (_commandsCount >= XPLDIRECT_MAXCOMMANDS_ARDUINO)
00569   {
00570     return -1;
00571   }
00572   _commands[_commandsCount] = new _commandStructure;
00573   _commands[_commandsCount]->commandName = commandName;
00574   _commands[_commandsCount]->commandHandle = -1; // invalid until assigned by xplane
00575   _commandsCount++;
00576   _allDataRefsRegistered = 0; // share this flag with the datarefs, true when everything is registered
      with xplane.
00577   return (_commandsCount - 1);
00578 }
00579
00580 // The central instance for the application
00581 XPLDirect XP(&Serial);
```

Index

_cmdPush
    Button, 18
_delay
    RepeatButton, 33
_mux
    Button, 18
_pin
    Button, 18
_state
    Button, 18
_timer
    RepeatButton, 33
_transition
    Button, 18

addMux
    DigitalIn_, 19
allDataRefsRegistered
    XPLDirect, 48
AnalogIn, 9
    AnalogIn, 10
    calibrate, 10
    handle, 11
    raw, 11
    setRange, 11
    setScale, 11
    value, 12
AnalogIn.cpp, 62
AnalogIn.h, 54

begin
    XPLDirect, 49
Button, 12
    _cmdPush, 18
    _mux, 18
    _pin, 18
    _state, 18
    _transition, 18
    Button, 14
    engaged, 15
    getCommand, 15
    handle, 15
    handleXP, 16
    pressed, 16
    processCommand, 16
    released, 17
    setCommand, 17
Button.cpp, 64
Button.h, 55

calibrate
    AnalogIn, 10
commandEnd
    XPLDirect, 49
commandStart
    XPLDirect, 49
commandTrigger
    XPLDirect, 49
connectionStatus
    XPLDirect, 50
count
    Timer, 46

datarefsUpdated
    XPLDirect, 50
DigitalIn.cpp, 65
DigitalIn.h, 56
DigitalIn_, 19
    addMux, 19
    DigitalIn_, 19
    getBit, 20
    handle, 20
    setMux, 20
down
    Encoder, 23

elapsed
    Timer, 47
Encoder, 21
    down, 23
    Encoder, 22
    engaged, 23
    getCommand, 23
    handle, 24
    handleXP, 24
    pos, 24
    pressed, 24
    processCommand, 24
    released, 25
    setCommand, 25, 26
    up, 26
Encoder.cpp, 67
Encoder.h, 57
engaged
    Button, 15
    Encoder, 23

getBit
    DigitalIn_, 20
getCommand

Button, 15
Encoder, 23
Switch, 37
Switch2, 42
getTime
Timer, 47

handle
AnalogIn, 11
Button, 15
DigitalIn_, 20
Encoder, 24
LedShift, 28
RepeatButton, 32
ShiftOut, 34
Switch, 37
Switch2, 42
handleXP
Button, 16
Encoder, 24
RepeatButton, 32
Switch, 37
Switch2, 42
hasUpdated
XPLDirect, 50

LedShift, 27
handle, 28
LedShift, 27
set, 28
set_all, 28
setAll, 28
setPin, 29
LedShift.cpp, 68
LedShift.h, 58

main.cpp, 53, 54

off
Switch, 37
Switch2, 42
on
Switch, 38
on1
Switch2, 43
on2
Switch2, 43

pos
Encoder, 24
pressed
Button, 16
Encoder, 24
processCommand
Button, 16
Encoder, 24
Switch, 38
Switch2, 43

raw
AnalogIn, 11
registerCommand
XPLDirect, 50
registerDataRef
XPLDirect, 50, 51
released
Button, 17
Encoder, 25
RepeatButton, 29
_delay, 33
_timer, 33
handle, 32
handleXP, 32
RepeatButton, 31

sendDebugMessage
XPLDirect, 52
sendResetRequest
XPLDirect, 52
sendSpeakMessage
XPLDirect, 52
set
LedShift, 28
set_all
LedShift, 28
setAll
LedShift, 28
ShiftOut, 34
setCommand
Button, 17
Encoder, 25, 26
Switch, 38, 39
Switch2, 43, 44
setCycle
Timer, 47
setMux
DigitalIn_, 20
setPin
LedShift, 29
ShiftOut, 35
setRange
AnalogIn, 11
setScale
AnalogIn, 11
ShiftOut, 33
handle, 34
setAll, 34
setPin, 35
ShiftOut, 34
ShiftOut.cpp, 69
ShiftOut.h, 58
Switch, 35
getCommand, 37
handle, 37
handleXP, 37
off, 37
on, 38
processCommand, 38
setCommand, 38, 39

Switch, 36
value, 40
Switch.cpp, 70
Switch.h, 58
Switch2, 40
getCommand, 42
handle, 42
handleXP, 42
off, 42
on1, 43
on2, 43
processCommand, 43
setCommand, 43, 44
Switch2, 41
value, 45

Timer, 45
count, 46
elapsed, 47
getTime, 47
setCycle, 47
Timer, 46
Timer.cpp, 73
Timer.h, 60

up
Encoder, 26

value
AnalogIn, 12
Switch, 40
Switch2, 45

xloop
XPLDirect, 52
XPLDevices.h, 60
XPLDirect, 48
allDataRefsRegistered, 48
begin, 49
commandEnd, 49
commandStart, 49
commandTrigger, 49
connectionStatus, 50
datarefsUpdated, 50
hasUpdated, 50
registerCommand, 50
registerDataRef, 50, 51
sendDebugMessage, 52
sendResetRequest, 52
sendSpeakMessage, 52
xloop, 52
XPLDirect, 48
XPLDirect.cpp, 73
XPLDirect.h, 60