XPLPro

Generated by Doxygen 1.9.6

Chapter 1

XPLPro

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:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

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 6 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 26 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 45 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 34 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 40 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 60 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 80 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 8 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 6 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 43 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 78 of file DigitalIn.cpp.

5.3.3.3 handle()

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

Read all mux inputs into process data input image.

Definition at line 92 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 20 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 8 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 5 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 72 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 63 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 51 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 3 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 63 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 54 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 42 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 7 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 3 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 34 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 14 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 26 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 9 of file Timer.cpp.

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

- Timer.h
- Timer.cpp

5.11 XPLPro Class Reference

Public Member Functions

- XPLPro (Stream *device)

  Constructor.
- void begin (const char *devicename, void(*initFunction)(void), void(*stopFunction)(void), void(*inboundHandler)(int))

  Register device and set callback functions.
- int connectionStatus ()

  Return connection status.
- int commandTrigger (int commandHandle)

  Trigger a command once.
- int commandTrigger (int commandHandle, int triggerCount)

  Trigger a command multiple times.
- int commandStart (int commandHandle)

  Start a command. All commandStart must be balanced with a commandEnd.
- int commandEnd (int commandHandle)

  End a command. All commandStart must be balanced with a commandEnd.
- void datarefWrite (int handle, long value)

  Write an integer DataRef.
- void datarefWrite (int handle, int value)

  Write an integer DataRef. Maps to long DataRefs.
- void datarefWrite (int handle, long value, int arrayElement)

  Write a Integer DataRef to an array element.
- void datarefWrite (int handle, int value, int arrayElement)

  Write a Integer DataRef to an array element. Maps to long DataRefs.
- void datarefWrite (int handle, float value)

  Write a float DataRef.
- void datarefWrite (int handle, float value, int arrayElement)

  Write a float DataRef to an array element.
- void requestUpdates (int handle, int rate, float precision)

  Request DataRef updates from the plugin.
- void requestUpdates (int handle, int rate, float precision, int element)

  Request DataRef updates from the plugin for an array DataRef.
- void setScaling (int handle, int inLow, int inHigh, int outLow, int outHigh)

  set scaling factor for a DataRef (offload mapping to the plugin)
- int registerDataRef (XPString_t *datarefName)

  Register a DataRef and obtain a handle.
- int registerCommand (XPString_t *commandName)

  Register a Command and obtain a handle.
- float datarefReadFloat ()

  Read the received float DataRef.
- long datarefReadInt ()

  Read the received integer DataRef.
- int datarefReadElement ()

Read the received array element.

- int sendDebugMessage (const char *msg)

  Send a debug message to the plugin.

- int sendSpeakMessage (const char *msg)

  Send a speech message to the plugin.

- void sendResetRequest (void)

  Request a reset from the plugin.

- int xloop ()

  Cyclic loop handler, must be called in idle task.

5.11.1 Detailed Description

Definition at line 98 of file XPLPro.h.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 XPLPro()

```
XPLPro::XPLPro (
            Stream * device )
```

Constructor.

Parameters

device	Device to use (should be &Serial)

Definition at line 5 of file XPLPro.cpp.

5.11.3 Member Function Documentation

5.11.3.1 begin()

```
void XPLPro::begin (
            const char * devicename,
            void(*)(void) initFunction,
            void(*)(void) stopFunction,
            void(*)(int) inboundHandler )
```

Register device and set callback functions.

Parameters

devicename	Device name
initFunction	Callback for DataRef and Command registration
stopFunction	Callback for XPlane shutdown or plane change
inboundHandler	Callback for incoming DataRefs

Definition at line 11 of file XPLPro.cpp.

5.11.3.2 commandEnd()

```
int XPLPro::commandEnd (
            int commandHandle )
```

End a command. All commandStart must be balanced with a commandEnd.

Parameters

commandHandle	Handle of the command to start

Returns

0: OK, -1: command was not registered

Definition at line 61 of file XPLPro.cpp.

5.11.3.3 commandStart()

```
int XPLPro::commandStart (
            int commandHandle )
```

Start a command. All commandStart must be balanced with a commandEnd.

Parameters

commandHandle	Handle of the command to start

Returns

0: OK, -1: command was not registered

Definition at line 51 of file XPLPro.cpp.

5.11.3.4 commandTrigger() [1/2]

```
int XPLPro::commandTrigger (
            int commandHandle )  [inline]
```

Trigger a command once.

Parameters

commandHandle	of the command to trigger

Returns

0: OK, -1: command was not registered

Definition at line 119 of file XPLPro.h.

5.11.3.5 commandTrigger() [2/2]

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

Trigger a command multiple times.

Parameters

commandHandle	Handle of the command to trigger
triggerCount	Number of times to trigger the command

Returns

0: OK, -1: command was not registered

Definition at line 40 of file XPLPro.cpp.

5.11.3.6 connectionStatus()

```
int XPLPro::connectionStatus ( )
```

Return connection status.

Returns

True if connection to XPlane established

Definition at line 71 of file XPLPro.cpp.

5.11.3.7 datarefReadElement()

```
int XPLPro::datarefReadElement ( )  [inline]
```

Read the received array element.

Returns

Received array element

Definition at line 205 of file XPLPro.h.

5.11.3.8 datarefReadFloat()

```
float XPLPro::datarefReadFloat ( )  [inline]
```

Read the received float DataRef.

Returns

Received value

Definition at line 197 of file XPLPro.h.

5.11.3.9 datarefReadInt()

```
long XPLPro::datarefReadInt ( )  [inline]
```

Read the received integer DataRef.

Returns

Received value

Definition at line 201 of file XPLPro.h.

5.11.3.10 datarefWrite() [1/6]

```
void XPLPro::datarefWrite (
            int handle,
            float value )
```

Write a float DataRef.

Parameters

handle	Handle of the DataRef to write
value	Value to write to the DataRef

Definition at line 130 of file XPLPro.cpp.

5.11.3.11 datarefWrite() [2/6]

```
void XPLPro::datarefWrite (
            int handle,
            float value,
            int arrayElement )
```

Write a float DataRef to an array element.

Parameters

handle	Handle of the DataRef to write
value	Value to write to the DataRef

Definition at line 147 of file XPLPro.cpp.

5.11.3.12 datarefWrite() [3/6]

```
void XPLPro::datarefWrite (
            int handle,
            int value )
```

Write an integer DataRef. Maps to long DataRefs.

Parameters

handle	Handle of the DataRef to write
value	Value to write to the DataRef

Definition at line 90 of file XPLPro.cpp.

5.11.3.13 datarefWrite() [4/6]

```
void XPLPro::datarefWrite (
            int handle,
```

```
            int value,
            int arrayElement )
```

Write a Integer DataRef to an array element. Maps to long DataRefs.

Parameters

handle	Handle of the DataRef to write
value	Value to write to the DataRef
arrayElement	Array element to write to

Definition at line 100 of file XPLPro.cpp.

5.11.3.14 datarefWrite() [5/6]

```
void XPLPro::datarefWrite (
            int handle,
            long value )
```

Write an integer DataRef.

Parameters

handle	Handle of the DataRef to write
value	Value to write to the DataRef

Definition at line 110 of file XPLPro.cpp.

5.11.3.15 datarefWrite() [6/6]

```
void XPLPro::datarefWrite (
            int handle,
            long value,
            int arrayElement )
```

Write a Integer DataRef to an array element.

Parameters

handle	Handle of the DataRef to write
value	Value to write to the DataRef
arrayElement	Array element to write to

Definition at line 120 of file XPLPro.cpp.

5.11.3.16 registerCommand()

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

Register a Command and obtain a handle.

Parameters

commandName	Name of the Command (or abbreviation)

Returns

Assigned handle for the Command, -1 if Command was not found

Definition at line 462 of file XPLPro.cpp.

5.11.3.17 registerDataRef()

```
int XPLPro::registerDataRef (
            XPString_t * datarefName )
```

Register a DataRef and obtain a handle.

Parameters

datarefName	Name of the DataRef (or abbreviation)

Returns

Assigned handle for the DataRef, -1 if DataRef was not found

Definition at line 437 of file XPLPro.cpp.

5.11.3.18 requestUpdates() [1/2]

```
void XPLPro::requestUpdates (
            int handle,
            int rate,
            float precision )
```

Request DataRef updates from the plugin.

Parameters

handle	Handle of the DataRef to subscribe to
rate	Maximum rate for updates to reduce traffic
precision	Floating point precision

Definition at line 479 of file XPLPro.cpp.

5.11.3.19 requestUpdates() [2/2]

```
void XPLPro::requestUpdates (
            int handle,
            int rate,
            float precision,
            int element )
```

Request DataRef updates from the plugin for an array DataRef.

Parameters

handle	Handle of the DataRef to subscribe to
rate	Maximum rate for updates to reduce traffic
precision	Floating point precision
arrayElement	Array element to subscribe to

Definition at line 493 of file XPLPro.cpp.

5.11.3.20 sendDebugMessage()

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

Send a debug message to the plugin.

Parameters

msg	Message to show as debug string

Returns

Definition at line 76 of file XPLPro.cpp.

5.11.3.21 sendResetRequest()

```
void XPLPro::sendResetRequest (
            void  )
```

Request a reset from the plugin.

Definition at line 174 of file XPLPro.cpp.

5.11.3.22 sendSpeakMessage()

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

Send a speech message to the plugin.

Parameters

msg	Message to speak

Returns

Definition at line 82 of file XPLPro.cpp.

5.11.3.23 setScaling()

```
void XPLPro::setScaling (
            int handle,
            int inLow,
            int inHigh,
            int outLow,
            int outHigh )
```

set scaling factor for a DataRef (offload mapping to the plugin)

Definition at line 508 of file XPLPro.cpp.

5.11.3.24 xloop()

```
int XPLPro::xloop (
            void  )
```

Cyclic loop handler, must be called in idle task.

Returns

Connection status

Definition at line 25 of file XPLPro.cpp.

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

- XPLPro.h
- XPLPro.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 <XPLPro.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 <XPLPro.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 <XPLPro.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 XPLPro.h

```
00001 //   XPLPro.h - Library for serial interface to Xplane SDK.
00002 //   Created by Curiosity Workshop, Michael Gerlicher and Martin Ruskowski,  2020-2023
00003 //   See readme.txt file for information on updates.
00004 //   To report problems, download updates and examples, suggest enhancements or get technical support,
      please visit:
00005 //      discord:  https://discord.gg/gzXetjEST4
00006 //      patreon:  www.patreon.com/curiosityworkshop
00007
00008 #ifndef XPLPro_h
00009 #define XPLPro_h
00010
00011 #include <Arduino.h>
00012
00014 // Parameters which can be overridden by command line defines
00016
00017 // Decimals of precision for floating point datarefs. More increases dataflow (default 4)
00018 #ifndef XPL_FLOATPRECISION
00019 #define XPL_FLOATPRECISION 4
00020 #endif
00021
00022 // Timeout after sending a registration request, how long will we wait for the response.
00023 // This is giant because sometimes xplane says the plane is loaded then does other stuff for a while.
      (default 90000 ms)
00024 #ifndef XPL_RESPONSE_TIMEOUT
00025 #define XPL_RESPONSE_TIMEOUT 90000
00026 #endif
00027
00028 // For boards with limited memory that can use PROGMEM to store strings.
00029 // You will need to wrap your dataref names with F() macro ie:
00030 // Xinterface.registerDataref(F("laminar/B738/annunciator/drive2"), XPL_READ, 100, 0, &drive2);
00031 // Disable for boards that have issues compiling: errors with strncmp_PF for instance.
00032 #ifndef XPL_USE_PROGMEM
00033 #ifdef __AVR_ARCH__
00034 // flash strings are default on on AVR architecture
00035 #define XPL_USE_PROGMEM 1
00036 #else
00037 // and off otherwise
00038 #define XPL_USE_PROGMEM 0
00039 #endif
00040 #endif
00041
00042 // Package buffer size for send and receive buffer each.
```

```
00043 // If you need a few extra bytes of RAM it could be reduced, but it needs to
00044 // be as long as the longest dataref name + 10.  If you are using datarefs
00045 // that transfer strings it needs to be big enough for those too.  (default 200)
00046 #ifndef XPLMAX_PACKETSIZE_TRANSMIT
00047 #define XPLMAX_PACKETSIZE_TRANSMIT 200
00048 #endif
00049
00050 #ifndef XPLMAX_PACKETSIZE_RECEIVE
00051 #define XPLMAX_PACKETSIZE_RECEIVE 200
00052 #endif
00053
00055 // All other defines in this header must not be modified
00057
00058 // define whether flash strings will be used
00059 #if XPL_USE_PROGMEM
00060 // use Flash for strings, requires F() macro for strings in all registration calls
00061   typedef const __FlashStringHelper XPString_t;
00062 #else
00063   typedef const char XPString_t;
00064 #endif
00065
00066 // Parameters around the interface
00067 #define XPL_BAUDRATE 115200   // Baudrate needed to match plugin
00068 #define XPL_RX_TIMEOUT 500    // Timeout for reception of one frame
00069 #define XPL_PACKETHEADER '['  // Frame start character
00070 #define XPL_PACKETTRAILER ']' // Frame end character
00071 #define XPL_HANDLE_INVALID -1 // invalid handle
00072
00073 // Items in caps generally come from XPlane. Items in lower case are generally sent from the arduino.
00074 #define XPLCMD_SENDNAME 'N'               // plugin request name from arduino
00075 #define XPLRESPONSE_NAME 'n'              // Arduino responds with device name as initialized in the
      "begin" function
00076 #define XPLCMD_SENDREQUEST 'Q'            // plugin sends this when it is ready to register bindings
00077 #define XPLREQUEST_REGISTERDATAREF 'b'    // Register a DataRef
00078 #define XPLREQUEST_REGISTERCOMMAND 'm'    // Register a command
00079 #define XPLRESPONSE_DATAREF 'D'           // Plugin responds with handle to dataref or - value if not
      found.  dataref handle, dataref name
00080 #define XPLRESPONSE_COMMAND 'C'           // Plugin responds with handle to command or - value if not
      found.  command handle, command name
00081 #define XPLCMD_PRINTDEBUG 'g'             // Plugin logs string sent from arduino
00082 #define XPLCMD_SPEAK 's'                  // plugin speaks string through xplane speech
00083 #define XPLREQUEST_REFRESH 'd'            // the plugin will call this once xplane is loaded in order
      to get fresh updates from arduino handles that write
00084 #define XPLREQUEST_UPDATES 'r'            // arduino is asking the plugin to update the specified
      dataref with rate and divider parameters
00085 #define XPLREQUEST_UPDATESARRAY 't'       // arduino is asking the plugin to update the specified
      array dataref with rate and divider parameters
00086 #define XPLREQUEST_SCALING 'u'            // arduino requests the plugin apply scaling to the dataref
      values
00087 #define XPLCMD_RESET 'z'                  // Request a reset and reregistration from the plugin
00088 #define XPLCMD_DATAREFUPDATEINT '1'       // Int DataRef update
00089 #define XPLCMD_DATAREFUPDATEFLOAT '2'     //  Float DataRef update
00090 #define XPLCMD_DATAREFUPDATEINTARRAY '3'  // Int array DataRef update
00091 #define XPLCMD_DATAREFUPDATEFLOATARRAY '4' // Float array DataRaéf Update
00092 #define XPLCMD_DATAREFUPDATESTRING '9'    // String DataRef update
00093 #define XPLCMD_COMMANDTRIGGER 'k'         // Trigger command n times
00094 #define XPLCMD_COMMANDSTART 'i'           // Begin command (Button pressed)
00095 #define XPLCMD_COMMANDEND 'j'             // End command (Button released)
00096 #define XPL_EXITING 'X'                   // XPlane sends this to the arduino device during normal
      shutdown of XPlane. It may not happen if xplane crashes.
00097
00098 class XPLPro
00099 {
00100 public:
00103   XPLPro(Stream *device);
00104
00110   void begin(const char *devicename, void (*initFunction)(void), void (*stopFunction)(void), void
      (*inboundHandler)(int));
00111
00114   int connectionStatus();
00115
00119   int commandTrigger(int commandHandle) { return commandTrigger(commandHandle, 1); };
00120
00125   int commandTrigger(int commandHandle, int triggerCount);
00126
00130   int commandStart(int commandHandle);
00131
00135   int commandEnd(int commandHandle);
00136
00140   void datarefWrite(int handle, long value);
00141
00145   void datarefWrite(int handle, int value);
00146
00151   void datarefWrite(int handle, long value, int arrayElement);
00152
00157   void datarefWrite(int handle, int value, int arrayElement);
00158
```

```
00162   void datarefWrite(int handle, float value);
00163
00167   void datarefWrite(int handle, float value, int arrayElement);
00168
00173   void requestUpdates(int handle, int rate, float precision);
00174
00180   void requestUpdates(int handle, int rate, float precision, int element);
00181
00183   void setScaling(int handle, int inLow, int inHigh, int outLow, int outHigh);
00184
00188   int registerDataRef(XPString_t *datarefName);
00189
00193   int registerCommand(XPString_t *commandName);
00194
00197   float datarefReadFloat() { return _readValueFloat; }
00198
00201   long datarefReadInt() { return _readValueLong; }
00202
00205   int datarefReadElement() { return _readValueElement; }
00206
00210   int sendDebugMessage(const char *msg);
00211
00215   int sendSpeakMessage(const char *msg);
00216
00218   void sendResetRequest(void);
00219
00222   int xloop();
00223
00224 private:
00225   void _processSerial();
00226   void _processPacket();
00227   void _transmitPacket();
00228   void _sendname();
00229   void _sendPacketVoid(int command, int handle);        // just a command with a handle
00230   void _sendPacketString(int command, const char *str); // send a string
00231   int _parseInt(int *outTarget, char *inBuffer, int parameter);
00232   int _parseInt(long *outTarget, char *inBuffer, int parameter);
00233   int _parseFloat(float *outTarget, char *inBuffer, int parameter);
00234   int _parseString(char *outBuffer, char *inBuffer, int parameter, int maxSize);
00235
00236   Stream *_streamPtr;
00237   const char *_deviceName;
00238   byte _registerFlag;
00239   byte _connectionStatus;
00240
00241   char _sendBuffer[XPLMAX_PACKETSIZE_TRANSMIT];
00242   char _receiveBuffer[XPLMAX_PACKETSIZE_RECEIVE];
00243   int _receiveBufferBytesReceived;
00244
00245   void (*_xplInitFunction)(void);  // this function will be called when the plugin is ready to receive
      binding requests
00246   void (*_xplStopFunction)(void);  // this function will be called with the plugin receives message or
      detects xplane flight model inactive
00247   void (*_xplInboundHandler)(int); // this function will be called when the plugin sends dataref
      values
00248
00249   int _handleAssignment;
00250   long _readValueLong;
00251   float _readValueFloat;
00252   int _readValueElement;
00253 };
00254
00255 #ifndef XPLPRO_STANDALONE
00257 extern XPLPro XP;
00258
00259 // include device libraries
00260 #include <DigitalIn.h>
00261 #include <Button.h>
00262 #include <Encoder.h>
00263 #include <Switch.h>
00264 #include <ShiftOut.h>
00265 #include <LedShift.h>
00266 #include <Timer.h>
00267 #include <AnalogIn.h>
00268 #endif
00269
00270 #endif
```

6.12 AnalogIn.cpp

```
00001 #include "AnalogIn.h"
00002
00003 #define FULL_SCALE ((1 « AD_RES) - 1)
```

```
00004 #define HALF_SCALE (1 « (AD_RES - 1))
00005
00006 AnalogIn::AnalogIn(uint8_t pin, Analog_t type)
00007 {
00008   _pin = pin;
00009   _filterConst = 1.0;
00010   _scale = 1.0;
00011   _min = 0;
00012   _max = FULL_SCALE;
00013   _type = type;
00014   pinMode(_pin, INPUT);
00015   if (_type == bipolar)
00016   {
00017     _offset = HALF_SCALE;
00018   }
00019   else
00020   {
00021     _offset = 0;
00022   }
00023   _calcScales();
00024 }
00025
00026 AnalogIn::AnalogIn(uint8_t pin, Analog_t type, float timeConst) : AnalogIn(pin, type)
00027 {
00028   if (timeConst > 0)
00029   {
00030     _filterConst = 1.0 / timeConst;
00031   }
00032 }
00033
00034 void AnalogIn::handle()
00035 {
00036   int _raw = raw();
00037   _value = (_filterConst * _raw * (_raw >= 0 ? _scalePos : _scaleNeg)) + (1.0 - _filterConst) *
      _value;
00038 }
00039
00040 int AnalogIn::raw()
00041 {
00042   return constrain(analogRead(_pin), (int16_t)_min, (int16_t)_max) - _offset;
00043 }
00044
00045 void AnalogIn::calibrate()
00046 {
00047   if (_type == unipolar)
00048   {
00049     return;
00050   }
00051   long sum = 0;
00052   for (int i = 0; i < 64; i++)
00053   {
00054     sum += analogRead(_pin);
00055   }
00056   _offset = (int)(sum / 64);
00057   _calcScales();
00058 }
00059
00060 void AnalogIn::setRange(uint16_t min, uint16_t max)
00061 {
00062   _min = min(min, max);
00063   _max = max(min, max);
00064   if (min == max)
00065   {
00066     _min = 0;
00067     _max = FULL_SCALE;
00068   }
00069   if (_type == unipolar)
00070   {
00071     _offset = _min;
00072   }
00073   else
00074   {
00075     _offset = (_max + _min) / 2;
00076   }
00077   _calcScales();
00078 }
00079
00080 void AnalogIn::setScale(float scale)
00081 {
00082   _scale = scale;
00083   _calcScales();
00084 }
00085
00086 void AnalogIn::_calcScales()
00087 {
00088   if (_type == unipolar)
00089   {
```

```
00090     _scalePos = _scale / (float)(_max - _min);
00091     _scaleNeg = 0;
00092   }
00093   else
00094   {
00095     _scalePos = (_offset == _max) ? 0 : _scale / (float)(_max - _offset);
00096     _scaleNeg = (_offset == _min) ? 0 : _scale / (float)(_offset - _min);
00097   }
00098 }
```

6.13 Button.cpp

```
00001 #include "Button.h"
00002
00003 #ifndef DEBOUNCE_DELAY
00004 #define DEBOUNCE_DELAY 20
00005 #endif
00006
00007 // Buttons
00008 Button::Button(uint8_t mux, uint8_t pin)
00009 {
00010   _mux = mux;
00011   _pin = pin;
00012   _state = 0;
00013   _transition = 0;
00014   _cmdPush = -1;
00015   if(mux == NOT_USED) {
00016     pinMode(_pin, INPUT_PULLUP);
00017   }
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 }
```

6.14 DigitalIn.cpp

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

```
00060     return false;
00061   }
00062   if (!_mcp[_numMCP].begin_I2C(adress, &Wire))
00063   {
00064     return false;
00065   }
00066   for (int i = 0; i < 16; i++)
00067   {
00068     // TODO: register write iodir = 0xffff, ipol = 0xffff, gppu = 0xffff
00069     _mcp[_numMCP].pinMode(i, INPUT_PULLUP);
00070   }
00071   _numMCP++;
00072   _pin[_numPins++] = MCP_PIN;
00073   return true;
00074 }
00075 #endif
00076
00077 // Gets specific channel from expander, number according to initialization order
00078 bool DigitalIn_::getBit(uint8_t expander, uint8_t channel)
00079 {
00080   if (expander == NOT_USED)
00081   {
00082   #ifdef ARDUINO_ARCH_AVR
00083     return (*portInputRegister(digitalPinToPort(channel)) & digitalPinToBitMask(channel)) ? false :
      true;
00084   #else
00085     return !digitalRead(channel);
00086   #endif
00087   }
00088   return bitRead(_data[expander], channel);
00089 }
00090
00091 // read all inputs together -> base for board specific optimization by using byte read
00092 void DigitalIn_::handle()
00093 {
00094   // only if Mux Pins present
00095 #if MCP_MAX_NUMBER > 0
00096   if (_numPins > _numMCP)
00097 #else
00098   if (_numPins > 0)
00099 #endif
00100   {
00101     for (uint8_t channel = 0; channel < 16; channel++)
00102     {
00103 #ifdef ARDUINO_ARCH_AVR
00104       uint8_t oldSREG = SREG;
00105       noInterrupts();
00106       bitRead(channel, 0) ? *portOutputRegister(_s0port) |= _s0mask : *portOutputRegister(_s0port) &=
      ~_s0mask;
00107       bitRead(channel, 1) ? *portOutputRegister(_s1port) |= _s1mask : *portOutputRegister(_s1port) &=
      ~_s1mask;
00108       bitRead(channel, 2) ? *portOutputRegister(_s2port) |= _s2mask : *portOutputRegister(_s2port) &=
      ~_s2mask;
00109       bitRead(channel, 3) ? *portOutputRegister(_s3port) |= _s3mask : *portOutputRegister(_s3port) &=
      ~_s3mask;
00110       SREG = oldSREG;
00111       delayMicroseconds(1);
00112 #else
00113       digitalWrite(_s0, bitRead(channel, 0));
00114       digitalWrite(_s1, bitRead(channel, 1));
00115       digitalWrite(_s2, bitRead(channel, 2));
00116       digitalWrite(_s3, bitRead(channel, 3));
00117 #endif
00118       for (uint8_t expander = 0; expander < _numPins; expander++)
00119       {
00120         if (_pin[expander] != MCP_PIN)
00121         {
00122 #ifdef ARDUINO_ARCH_AVR
00123           bitWrite(_data[expander], channel, (*portInputRegister(digitalPinToPort(_pin[expander])) &
      digitalPinToBitMask(_pin[expander])) ? false : true);
00124 #else
00125           bitWrite(_data[expander], channel, !digitalRead(_pin[expander]));
00126 #endif
00127         }
00128       }
00129     }
00130   }
00131 #if MCP_MAX_NUMBER > 0
00132   int mcp = 0;
00133   for (uint8_t expander = 0; expander < _numPins; expander++)
00134   {
00135     if (_pin[expander] == MCP_PIN)
00136     {
00137       _data[expander] = ~_mcp[mcp++].readGPIOAB();
00138     }
00139   }
00140 #endif
```

```
00141 }
00142
00143 DigitalIn_ DigitalIn;
```

6.15 Encoder.cpp

```
00001 #include "Encoder.h"
00002
00003 #ifndef DEBOUNCE_DELAY
00004 #define DEBOUNCE_DELAY 20
00005 #endif
00006
00007 // Encoder with button functionality on MUX
00008 Encoder::Encoder(uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses)
00009 {
00010   _mux = mux;
00011   _pin1 = pin1;
00012   _pin2 = pin2;
00013   _pin3 = pin3;
00014   _pulses = pulses;
00015   _count = 0;
00016   _state = 0;
00017   _transition = transNone;
00018   _cmdUp = -1;
00019   _cmdDown = -1;
00020   _cmdPush = -1;
00021   if(mux == NOT_USED) {
00022     pinMode(_pin1, INPUT_PULLUP);
00023     pinMode(_pin2, INPUT_PULLUP);
00024     if (_pin3 != NOT_USED)
00025     {
00026         pinMode(_pin3, INPUT_PULLUP);
00027     }
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 }
```

6.16 LedShift.cpp

```
00001 #include "LedShift.h"
00002
00003 #define BLINK_DELAY 150
00004
00005 LedShift::LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins)
00006 {
00007   _count = 0;
00008   _timer = millis() + BLINK_DELAY;
00009   _pin_DAI = pin_DAI;
00010   _pin_DCK = pin_DCK;
00011   _pin_LAT = pin_LAT;
00012   _pins = min(pins, 64);
00013   for (int pin = 0; pin < _pins; pin++)
```

```
00014   {
00015     _mode[pin] = ledOff;
00016   }
00017   pinMode(_pin_DAI, OUTPUT);
00018   pinMode(_pin_DCK, OUTPUT);
00019   pinMode(_pin_LAT, OUTPUT);
00020   digitalWrite(_pin_DAI, LOW);
00021   digitalWrite(_pin_DCK, LOW);
00022   digitalWrite(_pin_LAT, LOW);
00023   _send();
00024 }
00025
00026 // send data
00027 void LedShift::_send()
00028 {
00029   // get bit masks
00030   uint8_t dataPort = digitalPinToPort(_pin_DAI);
00031   uint8_t dataMask = digitalPinToBitMask(_pin_DAI);
00032   uint8_t clockPort = digitalPinToPort(_pin_DCK);
00033   uint8_t clockMask = digitalPinToBitMask(_pin_DCK);
00034   uint8_t oldSREG = SREG;
00035   noInterrupts();
00036   uint8_t val = _count | 0x08;
00037   for (uint8_t pin = _pins; pin-- > 0;)
00038   {
00039     (_mode[pin] & val) > 0 ? *portOutputRegister(dataPort) |= dataMask : *portOutputRegister(dataPort)
      &= ~dataMask;
00040     *portOutputRegister(clockPort) |= clockMask;
00041     *portOutputRegister(clockPort) &= ~clockMask;
00042   }
00043   // latch LAT signal
00044   clockPort = digitalPinToPort(_pin_LAT);
00045   clockMask = digitalPinToBitMask(_pin_LAT);
00046   *portOutputRegister(clockPort) |= clockMask;
00047   *portOutputRegister(clockPort) &= ~clockMask;
00048   SREG = oldSREG;
00049 }
00050
00051 void LedShift::setPin(uint8_t pin, led_t mode)
00052 {
00053   if (pin < _pins)
00054   {
00055     if (_mode[pin] != mode)
00056     {
00057       _mode[pin] = mode;
00058       _update = true;
00059     }
00060   }
00061 }
00062
00063 void LedShift::setAll(led_t mode)
00064 {
00065   for (int pin = 0; pin < _pins; pin++)
00066   {
00067     _mode[pin] = mode;
00068   }
00069   _update = true;
00070 }
00071
00072 void LedShift::handle()
00073 {
00074   if (millis() >= _timer)
00075   {
00076     _timer += BLINK_DELAY;
00077     _count = (_count + 1) & 0x07;
00078     _update = true;
00079   }
00080   if (_update)
00081   {
00082     _send();
00083     _update = false;
00084   }
00085 }
```

6.17 ShiftOut.cpp

```
00001 #include "ShiftOut.h"
00002
00003 ShiftOut::ShiftOut(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT, uint8_t pins)
00004 {
00005   _pin_DAI = pin_DAI;
00006   _pin_DCK = pin_DCK;
00007   _pin_LAT = pin_LAT;
```

```
00008   _pins = min(pins, 64);
00009   pinMode(_pin_DAI, OUTPUT);
00010   pinMode(_pin_DCK, OUTPUT);
00011   pinMode(_pin_LAT, OUTPUT);
00012   digitalWrite(_pin_DAI, LOW);
00013   digitalWrite(_pin_DCK, LOW);
00014   digitalWrite(_pin_LAT, LOW);
00015   _send();
00016 }
00017
00018 // send data
00019 void ShiftOut::_send()
00020 {
00021   // get bit masks
00022   uint8_t dataPort = digitalPinToPort(_pin_DAI);
00023   uint8_t dataMask = digitalPinToBitMask(_pin_DAI);
00024   uint8_t clockPort = digitalPinToPort(_pin_DCK);
00025   uint8_t clockMask = digitalPinToBitMask(_pin_DCK);
00026   uint8_t oldSREG = SREG;
00027   noInterrupts();
00028   for (uint8_t pin = _pins; pin-- > 0;)
00029   {
00030     bitRead(_state[pin » 3], pin & 0x07) ? *portOutputRegister(dataPort) |= dataMask :
      *portOutputRegister(dataPort) &= ~dataMask;
00031     *portOutputRegister(clockPort) |= clockMask;
00032     *portOutputRegister(clockPort) &= ~clockMask;
00033   }
00034   // latch LAT signal
00035   clockPort = digitalPinToPort(_pin_LAT);
00036   clockMask = digitalPinToBitMask(_pin_LAT);
00037   *portOutputRegister(clockPort) |= clockMask;
00038   *portOutputRegister(clockPort) &= ~clockMask;
00039   SREG = oldSREG;
00040 }
00041
00042 void ShiftOut::setPin(uint8_t pin, bool state)
00043 {
00044   if (pin < _pins)
00045   {
00046     if (state != bitRead(_state[pin » 3], pin & 0x07))
00047     {
00048       bitWrite(_state[pin » 3], pin & 0x07, state);
00049       _update = true;
00050     }
00051   }
00052 }
00053
00054 void ShiftOut::setAll(bool state)
00055 {
00056   for (int pin = 0; pin < _pins; pin++)
00057   {
00058     bitWrite(_state[pin » 3], pin & 0x07, state);
00059   }
00060   _update = true;
00061 }
00062
00063 void ShiftOut::handle()
00064 {
00065   if (_update)
00066   {
00067     _send();
00068     _update = false;
00069   }
00070 }
```

6.18 Switch.cpp

```
00001 #include "Switch.h"
00002
00003 #ifndef DEBOUNCE_DELAY
00004 #define DEBOUNCE_DELAY 20
00005 #endif
00006
00007 Switch::Switch(uint8_t mux, uint8_t pin)
00008 {
00009   _mux = mux;
00010   _pin = pin;
00011   _state = switchOff;
00012   _cmdOn = -1;
00013   _cmdOff = -1;
00014   if(mux == NOT_USED) {
00015     pinMode(_pin, INPUT_PULLUP);
00016   }
```

```
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 }
```

6.19 Timer.cpp

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

6.20 XPLPro.cpp

```
00001 // XPLPro.cpp
00002 // Created by Curiosity Workshop, Michael Gerlicher, 2023.
00003 #include "XPLPro.h"
00004
00005 XPLPro::XPLPro(Stream *device)
00006 {
00007   _streamPtr = device;
00008   _streamPtr->setTimeout(XPL_RX_TIMEOUT);
00009 }
00010
```

```
00011 void XPLPro::begin(const char *devicename, void (*initFunction)(void), void (*stopFunction)(void),
      void (*inboundHandler)(int))
00012 {
00013 #ifndef XPL_STANDALONE
00014   Serial.begin(XPL_BAUDRATE);
00015 #endif
00016   _deviceName = (char *)devicename;
00017   _connectionStatus = 0;
00018   _receiveBuffer[0] = 0;
00019   _registerFlag = 0;
00020   _xplInitFunction = initFunction;
00021   _xplStopFunction = stopFunction;
00022   _xplInboundHandler = inboundHandler;
00023 }
00024
00025 int XPLPro::xloop(void)
00026 {
00027   // handle incoming serial data
00028   _processSerial();
00029   // when device is registered, perform handle registrations
00030   if (_registerFlag)
00031   {
00032     _xplInitFunction();
00033     _registerFlag = 0;
00034   }
00035   // return status of connection
00036   return _connectionStatus;
00037 }
00038
00039 // TODO: is a return value necessary? These could also be void like for the datarefs
00040 int XPLPro::commandTrigger(int commandHandle, int triggerCount)
00041 {
00042   if (commandHandle < 0)
00043   {
00044     return XPL_HANDLE_INVALID;
00045   }
00046   sprintf(_sendBuffer, "%c%c,%i,%i%c", XPL_PACKETHEADER, XPLCMD_COMMANDTRIGGER, commandHandle,
      triggerCount, XPL_PACKETTRAILER);
00047   _transmitPacket();
00048   return 0;
00049 }
00050
00051 int XPLPro::commandStart(int commandHandle)
00052 {
00053   if (commandHandle < 0)
00054   {
00055     return XPL_HANDLE_INVALID;
00056   }
00057   _sendPacketVoid(XPLCMD_COMMANDSTART, commandHandle);
00058   return 0;
00059 }
00060
00061 int XPLPro::commandEnd(int commandHandle)
00062 {
00063   if (commandHandle < 0)
00064   {
00065     return XPL_HANDLE_INVALID;
00066   }
00067   _sendPacketVoid(XPLCMD_COMMANDEND, commandHandle);
00068   return 0;
00069 }
00070
00071 int XPLPro::connectionStatus()
00072 {
00073   return _connectionStatus;
00074 }
00075
00076 int XPLPro::sendDebugMessage(const char *msg)
00077 {
00078   _sendPacketString(XPLCMD_PRINTDEBUG, msg);
00079   return 1;
00080 }
00081
00082 int XPLPro::sendSpeakMessage(const char *msg)
00083 {
00084   _sendPacketString(XPLCMD_SPEAK, msg);
00085   return 1;
00086 }
00087
00088 // these could be done better:
00089
00090 void XPLPro::datarefWrite(int handle, int value)
00091 {
00092   if (handle < 0)
00093   {
00094     return;
00095   }
```

```
00096   sprintf(_sendBuffer, "%c%c,%i,%i%c", XPL_PACKETHEADER, XPLCMD_DATAREFUPDATEINT, handle, value,
      XPL_PACKETTRAILER);
00097   _transmitPacket();
00098 }
00099
00100 void XPLPro::datarefWrite(int handle, int value, int arrayElement)
00101 {
00102   if (handle < 0)
00103   {
00104     return;
00105   }
00106   sprintf(_sendBuffer, "%c%c,%i,%i,%i%c", XPL_PACKETHEADER, XPLCMD_DATAREFUPDATEINTARRAY, handle,
      value, arrayElement, XPL_PACKETTRAILER);
00107   _transmitPacket();
00108 }
00109
00110 void XPLPro::datarefWrite(int handle, long value)
00111 {
00112   if (handle < 0)
00113   {
00114     return;
00115   }
00116   sprintf(_sendBuffer, "%c%c,%i,%ld%c", XPL_PACKETHEADER, XPLCMD_DATAREFUPDATEINT, handle, value,
      XPL_PACKETTRAILER);
00117   _transmitPacket();
00118 }
00119
00120 void XPLPro::datarefWrite(int handle, long value, int arrayElement)
00121 {
00122   if (handle < 0)
00123   {
00124     return;
00125   }
00126   sprintf(_sendBuffer, "%c%c,%i,%ld,%i%c", XPL_PACKETHEADER, XPLCMD_DATAREFUPDATEINTARRAY, handle,
      value, arrayElement, XPL_PACKETTRAILER);
00127   _transmitPacket();
00128 }
00129
00130 void XPLPro::datarefWrite(int handle, float value)
00131 {
00132   if (handle < 0)
00133   {
00134     return;
00135   }
00136   char tBuf[20]; // todo:  rewrite to eliminate this buffer.  Write directly to _sendBuffer
00137   dtostrf(value, 0, XPL_FLOATPRECISION, tBuf);
00138   sprintf(_sendBuffer, "%c%c,%i,%s%c",
00139           XPL_PACKETHEADER,
00140           XPLCMD_DATAREFUPDATEFLOAT,
00141           handle,
00142           tBuf,
00143           XPL_PACKETTRAILER);
00144   _transmitPacket();
00145 }
00146
00147 void XPLPro::datarefWrite(int handle, float value, int arrayElement)
00148 {
00149   if (handle < 0)
00150   {
00151     return;
00152   }
00153   char tBuf[20]; // todo:  rewrite to eliminate this buffer.  Write directly to _sendBuffer
00154   dtostrf(value, 0, XPL_FLOATPRECISION, tBuf);
00155   sprintf(_sendBuffer, "%c%c,%i,%s,%i%c",
00156           XPL_PACKETHEADER,
00157           XPLCMD_DATAREFUPDATEFLOATARRAY,
00158           handle,
00159           tBuf,
00160           arrayElement,
00161           XPL_PACKETTRAILER);
00162   _transmitPacket();
00163 }
00164
00165 void XPLPro::_sendname()
00166 {
00167   // register device on request only when we have a valid name
00168   if (_deviceName != NULL)
00169   {
00170     _sendPacketString(XPLRESPONSE_NAME, _deviceName);
00171   }
00172 }
00173
00174 void XPLPro::sendResetRequest()
00175 {
00176   // request a reset only when we have a valid name
00177   if (_deviceName != NULL)
00178   {
```

```
00179     _sendPacketVoid(XPLCMD_RESET, 0);
00180   }
00181 }
00182
00183 void XPLPro::_processSerial()
00184 {
00185   // read until package header found or buffer empty
00186   while (_streamPtr->available() && _receiveBuffer[0] != XPL_PACKETHEADER)
00187   {
00188     _receiveBuffer[0] = (char)_streamPtr->read();
00189   }
00190   // return when buffer empty and header not found
00191   if (_receiveBuffer[0] != XPL_PACKETHEADER)
00192   {
00193     return;
00194   }
00195   // read rest of package until trailer
00196   _receiveBufferBytesReceived = _streamPtr->readBytesUntil(XPL_PACKETTRAILER, (char
      *)&_receiveBuffer[1], XPLMAX_PACKETSIZE_RECEIVE - 1);
00197   // if no further chars available, delete package
00198   if (_receiveBufferBytesReceived == 0)
00199   {
00200     _receiveBuffer[0] = 0;
00201     return;
00202   }
00203   // add package trailer and zero byte to frame
00204   _receiveBuffer[++_receiveBufferBytesReceived] = XPL_PACKETTRAILER;
00205   _receiveBuffer[++_receiveBufferBytesReceived] = 0; // old habits die hard.
00206   // at this point we should have a valid frame
00207   _processPacket();
00208 }
00209
00210 void XPLPro::_processPacket()
00211 {
00212   int tHandle;
00213   // check whether we have a valid frame
00214   if (_receiveBuffer[0] != XPL_PACKETHEADER)
00215   {
00216     return;
00217   }
00218   // branch on receiverd command
00219   switch (_receiveBuffer[1])
00220   {
00221   // plane unloaded or XP exiting
00222   case XPL_EXITING:
00223     _connectionStatus = false;
00224     _xplStopFunction();
00225     break;
00226
00227   // register device
00228   case XPLCMD_SENDNAME:
00229     _sendname();
00230     _connectionStatus = true; // not considered active till you know my name
00231     _registerFlag = 0;
00232     break;
00233
00234   // plugin is ready for registrations.
00235   case XPLCMD_SENDREQUEST:
00236     _registerFlag = 1; // use a flag to signal registration so recursion doesn't occur
00237     break;
00238
00239   // get handle from response to registered dataref
00240   case XPLRESPONSE_DATAREF:
00241     _parseInt(&_handleAssignment, _receiveBuffer, 2);
00242     break;
00243
00244   // get handle from response to registered command
00245   case XPLRESPONSE_COMMAND:
00246     _parseInt(&_handleAssignment, _receiveBuffer, 2);
00247     break;
00248
00249   // int dataref received
00250   case XPLCMD_DATAREFUPDATEINT:
00251     _parseInt(&tHandle, _receiveBuffer, 2);
00252     _parseInt(&_readValueLong, _receiveBuffer, 3);
00253     _readValueFloat = 0;
00254     _readValueElement = 0;
00255     _xplInboundHandler(tHandle);
00256     break;
00257
00258   // int array dataref received
00259   case XPLCMD_DATAREFUPDATEINTARRAY:
00260     _parseInt(&tHandle, _receiveBuffer, 2);
00261     _parseInt(&_readValueLong, _receiveBuffer, 3);
00262     _parseInt(&_readValueElement, _receiveBuffer, 4);
00263     _readValueFloat = 0;
00264     _xplInboundHandler(tHandle);
```

```
00265     break;
00266
00267   // float dataref received
00268   case XPLCMD_DATAREFUPDATEFLOAT:
00269     _parseInt(&tHandle, _receiveBuffer, 2);
00270     _parseFloat(&_readValueFloat, _receiveBuffer, 3);
00271     _readValueLong = 0;
00272     _readValueElement = 0;
00273     _xplInboundHandler(tHandle);
00274     break;
00275
00276   // float array dataref received
00277   case XPLCMD_DATAREFUPDATEFLOATARRAY:
00278     _parseInt(&tHandle, _receiveBuffer, 2);
00279     _parseFloat(&_readValueFloat, _receiveBuffer, 3);
00280     _parseInt(&_readValueElement, _receiveBuffer, 4);
00281     _readValueLong = 0;
00282     _xplInboundHandler(tHandle);
00283     break;
00284
00285   // obsolete?
00286   case XPLREQUEST_REFRESH:
00287     break;
00288
00289   default:
00290     break;
00291   }
00292   // empty receive buffer
00293   _receiveBuffer[0] = 0;
00294 }
00295
00296 void XPLPro::_sendPacketVoid(int command, int handle) // just a command with a handle
00297 {
00298   // check for valid handle
00299   if (handle < 0)
00300   {
00301     return;
00302   }
00303   sprintf(_sendBuffer, "%c%c,%i%c", XPL_PACKETHEADER, command, handle, XPL_PACKETTRAILER);
00304   _transmitPacket();
00305 }
00306
00307 void XPLPro::_sendPacketString(int command, const char *str) // for a string
00308 {
00309   sprintf(_sendBuffer, "%c%c,\"%s\"%c", XPL_PACKETHEADER, command, str, XPL_PACKETTRAILER);
00310   _transmitPacket();
00311 }
00312
00313 void XPLPro::_transmitPacket(void)
00314 {
00315   _streamPtr->write(_sendBuffer);
00316   if (strlen(_sendBuffer) == 64)
00317   {
00318     // apparently a bug in arduino with some boards when we transmit exactly 64 bytes. That took a
      while to track down...
00319     _streamPtr->print(" ");
00320   }
00321 }
00322
00323 int XPLPro::_parseString(char *outBuffer, char *inBuffer, int parameter, int maxSize)
00324 {
00325   int cBeg;
00326   int pos = 0;
00327   int len;
00328
00329   for (int i = 1; i < parameter; i++)
00330   {
00331     while (inBuffer[pos] != ',' && inBuffer[pos] != 0)
00332     {
00333       pos++;
00334     }
00335     pos++;
00336   }
00337
00338   while (inBuffer[pos] != '\"' && inBuffer[pos] != 0)
00339   {
00340     pos++;
00341   }
00342   cBeg = ++pos;
00343
00344   while (inBuffer[pos] != '\"' && inBuffer[pos] != 0)
00345   {
00346     pos++;
00347   }
00348   len = pos - cBeg;
00349   if (len > maxSize)
00350   {
```

```
00351     len = maxSize;
00352   }
00353   strncpy(outBuffer, (char *)&inBuffer[cBeg], len);
00354   outBuffer[len] = 0;
00355   // fprintf(errlog, "_parseString, pos: %i, cBeg: %i, deviceName: %s\n", pos, cBeg, target);
00356   return 0;
00357 }
00358
00359 int XPLPro::_parseInt(int *outTarget, char *inBuffer, int parameter)
00360 {
00361   int cBeg;
00362   int pos = 0;
00363   // search for the selected parameter
00364   for (int i = 1; i < parameter; i++)
00365   {
00366     while (inBuffer[pos] != ',' && inBuffer[pos] != 0)
00367     {
00368       pos++;
00369     }
00370     pos++;
00371   }
00372   // parameter starts here
00373   cBeg = pos;
00374   // search for end of parameter
00375   while (inBuffer[pos] != ',' && inBuffer[pos] != 0 && inBuffer[pos] != XPL_PACKETTRAILER)
00376   {
00377     pos++;
00378   }
00379   // temporarily make parameter null terminated
00380   char holdChar = inBuffer[pos];
00381   inBuffer[pos] = 0;
00382   // get integer value from string
00383   *outTarget = atoi((char *)&inBuffer[cBeg]);
00384   // restore buffer
00385   inBuffer[pos] = holdChar;
00386   return 0;
00387 }
00388
00389 int XPLPro::_parseInt(long *outTarget, char *inBuffer, int parameter)
00390 {
00391   int cBeg;
00392   int pos = 0;
00393   for (int i = 1; i < parameter; i++)
00394   {
00395     while (inBuffer[pos] != ',' && inBuffer[pos] != 0)
00396     {
00397       pos++;
00398     }
00399     pos++;
00400   }
00401   cBeg = pos;
00402   while (inBuffer[pos] != ',' && inBuffer[pos] != 0 && inBuffer[pos] != XPL_PACKETTRAILER)
00403   {
00404     pos++;
00405   }
00406   char holdChar = inBuffer[pos];
00407   inBuffer[pos] = 0;
00408   *outTarget = atoi((char *)&inBuffer[cBeg]);
00409   inBuffer[pos] = holdChar;
00410   return 0;
00411 }
00412
00413 int XPLPro::_parseFloat(float *outTarget, char *inBuffer, int parameter)
00414 {
00415   int cBeg;
00416   int pos = 0;
00417   for (int i = 1; i < parameter; i++)
00418   {
00419     while (inBuffer[pos] != ',' && inBuffer[pos] != 0)
00420     {
00421       pos++;
00422     }
00423     pos++;
00424   }
00425   cBeg = pos;
00426   while (inBuffer[pos] != ',' && inBuffer[pos] != 0 && inBuffer[pos] != XPL_PACKETTRAILER)
00427   {
00428     pos++;
00429   }
00430   char holdChar = inBuffer[pos];
00431   inBuffer[pos] = 0;
00432   *outTarget = atof((char *)&inBuffer[cBeg]);
00433   inBuffer[pos] = holdChar;
00434   return 0;
00435 }
00436
00437 int XPLPro::registerDataRef(XPString_t *datarefName)
```

```
00438 {
00439   long int startTime;
00440
00441   // registration only allowed in callback (TODO: is this limitation really necessary?)
00442   if (!_registerFlag)
00443   {
00444     return XPL_HANDLE_INVALID;
00445   }
00446 #if XPL_USE_PROGMEM
00447   sprintf(_sendBuffer, "%c%c,\"%S\"%c", XPL_PACKETHEADER, XPLREQUEST_REGISTERDATAREF, (wchar_t
      *)datarefName, XPL_PACKETTRAILER);
00448 #else
00449   sprintf(_sendBuffer, "%c%c,\"%s\"%c", XPL_PACKETHEADER, XPLREQUEST_REGISTERDATAREF, (char
      *)datarefName, XPL_PACKETTRAILER);
00450 #endif
00451   _transmitPacket();
00452
00453   _handleAssignment = XPL_HANDLE_INVALID;
00454   startTime = millis(); // for timeout function
00455
00456   while (millis() - startTime < XPL_RESPONSE_TIMEOUT && _handleAssignment < 0)
00457     _processSerial();
00458
00459   return _handleAssignment;
00460 }
00461
00462 int XPLPro::registerCommand(XPString_t *commandName)
00463 {
00464   long int startTime = millis(); // for timeout function
00465 #if XPL_USE_PROGMEM
00466   sprintf(_sendBuffer, "%c%c,\"%S\"%c", XPL_PACKETHEADER, XPLREQUEST_REGISTERCOMMAND, (wchar_t
      *)commandName, XPL_PACKETTRAILER);
00467 #else
00468   sprintf(_sendBuffer, "%c%c,\"%s\"%c", XPL_PACKETHEADER, XPLREQUEST_REGISTERCOMMAND, (char
      *)commandName, XPL_PACKETTRAILER);
00469 #endif
00470   _transmitPacket();
00471   _handleAssignment = XPL_HANDLE_INVALID;
00472   while (millis() - startTime < XPL_RESPONSE_TIMEOUT && _handleAssignment < 0)
00473   {
00474     _processSerial();
00475   }
00476   return _handleAssignment;
00477 }
00478
00479 void XPLPro::requestUpdates(int handle, int rate, float precision)
00480 {
00481   char tBuf[20]; // todo:  rewrite to eliminate this buffer.  Write directly to _sendBuffer?
00482   dtostrf(precision, 0, XPL_FLOATPRECISION, tBuf);
00483   sprintf(_sendBuffer, "%c%c,%i,%i,%s%c",
00484           XPL_PACKETHEADER,
00485           XPLREQUEST_UPDATES,
00486           handle,
00487           rate,
00488           tBuf,
00489           XPL_PACKETTRAILER);
00490   _transmitPacket();
00491 }
00492
00493 void XPLPro::requestUpdates(int handle, int rate, float precision, int element)
00494 {
00495   char tBuf[20]; // todo:  rewrite to eliminate this buffer.  Write directly to _sendBuffer?
00496   dtostrf(precision, 0, XPL_FLOATPRECISION, tBuf);
00497   sprintf(_sendBuffer, "%c%c,%i,%i,%s,%i%c",
00498           XPL_PACKETHEADER,
00499           XPLREQUEST_UPDATESARRAY,
00500           handle,
00501           rate,
00502           tBuf,
00503           element,
00504           XPL_PACKETTRAILER);
00505   _transmitPacket();
00506 }
00507
00508 void XPLPro::setScaling(int handle, int inLow, int inHigh, int outLow, int outHigh)
00509 {
00510   sprintf(_sendBuffer, "%c%c,%i,%i,%i,%i,%i%c",
00511           XPL_PACKETHEADER,
00512           XPLREQUEST_SCALING,
00513           handle,
00514           inLow,
00515           inHigh,
00516           outLow,
00517           outHigh,
00518           XPL_PACKETTRAILER);
00519   _transmitPacket();
00520 }
```

```
00521
00522 #ifndef XPL_STANDALONE
00523 XPLPro XP(&Serial);
00524 #endif
```

Index