XPLDevices

Generated by Doxygen 1.9.6

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Class Documentation

4.1 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 +/- ranges are adapted to cover +/-1.

4.1.1 Detailed Description

Class to encapsulate analog inputs.

Definition at line 13 of file AnalogIn.h.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 AnalogIn() [1/2]

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

Setup analog input.

Parameters

pin	Arduino pin number to use
type	unipolar (0..1) or bipolar (-1..1) range

Definition at line 7 of file AnalogIn.cpp.

4.1.2.2 AnalogIn() [2/2]

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

Setup analog input with low pass filter.

Parameters

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

Definition at line 27 of file AnalogIn.cpp.

4.1.3 Member Function Documentation

4.1.3.1 calibrate()

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

Perform calibration for bipolar input, current position gets center and +/- ranges are adapted to cover +/-1.

Definition at line 43 of file AnalogIn.cpp.

4.1.3.2 handle()

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

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

Definition at line 32 of file AnalogIn.cpp.

4.1.3.3 raw()

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

Return raw value.

Returns

Read raw analog input and compensate bipolta offset

Definition at line 38 of file AnalogIn.cpp.

4.1.3.4 value()

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

Return actual value.

Returns

Actual, filtered value as captured with handle()

Definition at line 32 of file AnalogIn.h.

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

- AnalogIn.h
- AnalogIn.cpp

4.2 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.
- 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

4.2.1 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 7 of file Button.h.

4.2.2 Member Enumeration Documentation

4.2.2.1 anonymous enum

```
anonymous enum  [protected]
```

Definition at line 60 of file Button.h.

4.2.3 Constructor & Destructor Documentation

4.2.3.1 Button() [1/2]

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

Constructor, set mux and pin number.

Parameters

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

Definition at line 10 of file Button.cpp.

4.2.3.2 Button() [2/2]

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

Constructor, set digital input without mux.

Parameters

pin	Arduino pin number

Definition at line 20 of file Button.h.

4.2.4 Member Function Documentation

4.2.4.1 engaged()

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

Evaluate status of Button.

Returns

true: Button is currently held down

Definition at line 46 of file Button.h.

4.2.4.2 getCommand()

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

Get XPLDirect command associated with Button.

Returns

Handle of the command

Definition at line 54 of file Button.h.

4.2.4.3 handle() [1/2]

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

Handle realtime. Read input and evaluate any transitions.

Definition at line 23 of file Button.h.

4.2.4.4 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 27 of file Button.h.

4.2.4.5 handleXP() `[1/2]`

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

Handle realtime and process XPLDirect commands.

Definition at line 30 of file Button.h.

4.2.4.6 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 34 of file Button.h.

4.2.4.7 pressed()

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

Evaluate and reset transition if button pressed down.

Returns

true: Button was pressed. Transition detected.

Definition at line 38 of file Button.h.

4.2.4.8 processCommand()

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

Process all transitions and active transitions to XPLDirect

Definition at line 40 of file Button.cpp.

4.2.4.9 released()

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

Evaluate and reset transition if button released.

Returns

true: Button was released. Transition detected.

Definition at line 42 of file Button.h.

4.2.4.10 setCommand()

```
void Button::setCommand (
             int cmdPush )  [inline]
```

Set XPLDirect command for Button events.

Parameters

cmdPush	Command handle as returned by XP.registerCommand()

Definition at line 50 of file Button.h.

4.2.5 Member Data Documentation

4.2.5.1 _cmdPush

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

Definition at line 70 of file Button.h.

4.2.5.2 _mux

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

Definition at line 66 of file Button.h.

4.2.5.3 _pin

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

Definition at line 67 of file Button.h.

4.2.5.4 _state

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

Definition at line 68 of file Button.h.

4.2.5.5 _transition

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

Definition at line 69 of file Button.h.

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

- Button.h
- Button.cpp

4.3 DigitalIn_ Class Reference

Class to encapsulate digital inputs from 74HC4067 and MCP23017 input multiplexers, used by all digital input devices. Scans all mux 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 mux, uint8_t muxpin)

    Get one bit from the mux or a digital input.
- void handle ()

    Read all mux inputs into process data input image.

4.3.1 Detailed Description

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

Definition at line 23 of file DigitalIn.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 DigitalIn_()

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

Class constructor.

Definition at line 7 of file DigitalIn.cpp.

4.3.3 Member Function Documentation

4.3.3.1 addMux()

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

Add one 74HC4067 multiplexer.

Parameters

pin	Data pin the mux is connected to

Returns

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

Definition at line 34 of file DigitalIn.cpp.

4.3.3.2 getBit()

```
bool DigitalIn_::getBit (
            uint8_t mux,
            uint8_t muxpin )
```

Get one bit from the mux or a digital input.

Parameters

mux	mux to read from. Use NOT_USED to access ardunio digital input without mux
muxpin	pin (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 69 of file DigitalIn.cpp.

4.3.3.3 handle()

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

Read all mux inputs into process data input image.

Definition at line 79 of file DigitalIn.cpp.

4.3.3.4 setMux()

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

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

Parameters

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

Definition at line 21 of file DigitalIn.cpp.

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

- DigitalIn.h
- DigitalIn.cpp

4.4 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 (int cmdUp, int cmdDown)

    Set XPLDirect commands for Encoder events without push function.
- 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.

4.4.1 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 21 of file Encoder.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Encoder() [1/2]

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

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

Parameters

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

Definition at line 10 of file Encoder.cpp.

4.4.2.2 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 37 of file Encoder.h.

4.4.3 Member Function Documentation

4.4.3.1 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 55 of file Encoder.h.

4.4.3.2 engaged()

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

Evaluate status of Encoder push function.

Returns

true: Button is currently held down

Definition at line 67 of file Encoder.h.

4.4.3.3 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 82 of file Encoder.cpp.

4.4.3.4 handle()

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

Handle realtime. Read input and evaluate any transitions.

Definition at line 32 of file Encoder.cpp.

4.4.3.5 handleXP()

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

Handle realtime and process XPLDirect commands.

Definition at line 43 of file Encoder.h.

4.4.3.6 pos()

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

Read current Encoder count.

Returns

Remaining Encoder count.

Definition at line 47 of file Encoder.h.

4.4.3.7 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 59 of file Encoder.h.

4.4.3.8 processCommand()

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

Check for Encoder events and process XPLDirect commands as appropriate.

Definition at line 101 of file Encoder.cpp.

4.4.3.9 released()

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

Evaluate and reset transition if Encoder released.

Returns

true: Button was released. Transition detected and reset.

Definition at line 63 of file Encoder.h.

4.4.3.10 setCommand() [1/2]

```
void Encoder::setCommand (
             int cmdUp,
             int cmdDown )  [inline]
```

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 78 of file Encoder.h.

4.4.3.11 setCommand() [2/2]

```
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.

4.4.3.12 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 51 of file Encoder.h.

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

- Encoder.h
- Encoder.cpp

4.5 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)
  Constructor, setup DM13A LED driver and set pins.
- void set (uint8_t pin, led_t mode)
  Set one LED to a display mode.
- void set_all (led_t mode)
  Set display mode for all 16 LEDs.
- void handle ()
  Real time handling, call cyclic in loop()

4.5.1 Detailed Description

Class to encapsulate a DM13A LED driver IC.

Definition at line 20 of file LedShift.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 LedShift()

```
LedShift::LedShift (
            uint8_t pin_DAI,
            uint8_t pin_DCK,
            uint8_t pin_LAT )
```

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

Definition at line 6 of file LedShift.cpp.

4.5.3 Member Function Documentation

4.5.3.1 handle()

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

Real time handling, call cyclic in loop()

Definition at line 71 of file LedShift.cpp.

4.5.3.2 set()

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

Set one LED to a display mode.

Parameters

pin	DM13A pin of the LED (0-15)
mode	LED display mode

Definition at line 56 of file LedShift.cpp.

4.5.3.3 set_all()

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

Set display mode for all 16 LEDs.

Parameters

mode	LED display mode

Definition at line 62 of file LedShift.cpp.

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

- LedShift.h
- LedShift.cpp

4.6 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.
- 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** }

4.6.1 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 76 of file Button.h.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 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 52 of file Button.cpp.

4.6.2.2 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 91 of file Button.h.

4.6.3 Member Function Documentation

4.6.3.1 handle() `[1/2]`

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

Handle realtime. Read input and evaluate any transitions.

Definition at line 94 of file Button.h.

4.6.3.2 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 98 of file Button.h.

4.6.3.3 handleXP() `[1/2]`

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

Handle realtime and process XPLDirect commands.

Definition at line 101 of file Button.h.

4.6.3.4 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 105 of file Button.h.

4.6.4 Member Data Documentation

4.6.4.1 _delay

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

Definition at line 108 of file Button.h.

4.6.4.2 _timer

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

Definition at line 109 of file Button.h.

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

- Button.h
- Button.cpp

4.7 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, int cmdOff)

  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.

4.7.1 Detailed Description

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

Definition at line 6 of file Switch.h.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 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)
muxpin	pin on the mux (0-15)

Definition at line 9 of file Switch.cpp.

4.7.2.2 Switch() [2/2]

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

Constructor, set digital input without mux.

Parameters

pin	Arduino pin number

Definition at line 16 of file Switch.h.

4.7.3 Member Function Documentation

4.7.3.1 getCommand()

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

Get XPLDirect command for last transition of Switch.

Returns

Handle of the last command

Definition at line 47 of file Switch.cpp.

4.7.3.2 handle()

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

Handle realtime. Read input and evaluate any transitions.

Definition at line 19 of file Switch.cpp.

4.7.3.3 handleXP()

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

Handle realtime and process XPLDirect commands.

Definition at line 22 of file Switch.h.

4.7.3.4 off()

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

Check whether Switch set to off.

Returns

true: Switch is off

Definition at line 30 of file Switch.h.

4.7.3.5 on()

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

Check whether Switch set to on.

Returns

true: Switch is on

Definition at line 26 of file Switch.h.

4.7.3.6 processCommand()

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

Process all transitions to XPLDirect.

Definition at line 63 of file Switch.cpp.

4.7.3.7 setCommand()

```
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 41 of file Switch.cpp.

4.7.3.8 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 48 of file Switch.h.

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

- Switch.h
- Switch.cpp

4.8 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 (int cmdOn1, int cmdOff, int cmdOn2)

  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.

4.8.1 Detailed Description

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

Definition at line 66 of file Switch.h.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 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)
pin1	on2 pin on the mux (0-15)

Definition at line 74 of file Switch.cpp.

4.8.2.2 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 78 of file Switch.h.

4.8.3 Member Function Documentation

4.8.3.1 getCommand()

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

Get XPLDirect command for last transition of Switch.

Returns

Handle of the last command

Definition at line 130 of file Switch.cpp.

4.8.3.2 handle()

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

Handle realtime. Read inputs and evaluate any transitions.

Definition at line 89 of file Switch.cpp.

4.8.3.3 handleXP()

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

Handle realtime and process XPLDirect commands.

Definition at line 84 of file Switch.h.

4.8.3.4 off()

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

Check whether Switch set to off.

Returns

true: Switch is off

Definition at line 88 of file Switch.h.

4.8.3.5 on1()

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

Check whether Switch set to on1.

Returns

true: Switch is on1

Definition at line 92 of file Switch.h.

4.8.3.6 on2()

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

Check whether Switch set to on2.

Returns

true: Switch is on2

Definition at line 96 of file Switch.h.

4.8.3.7 processCommand()

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

Process all transitions to XPLDirect.

Definition at line 169 of file Switch.cpp.

4.8.3.8 setCommand() `[1/2]`

```
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 123 of file Switch.cpp.

4.8.3.9 setCommand() [2/2]

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

Definition at line 116 of file Switch.cpp.

4.8.3.10 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 121 of file Switch.h.

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

- Switch.h
- Switch.cpp

4.9 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.

4.9.1 Detailed Description

Priovide a simple software driven timer for general purpose use.

Definition at line 5 of file Timer.h.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 Timer()

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

Setup timer.

Parameters

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

Definition at line 4 of file Timer.cpp.

4.9.3 Member Function Documentation

4.9.3.1 count()

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

Return cycle counter and reset to zero.

Returns

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

Definition at line 35 of file Timer.cpp.

4.9.3.2 elapsed()

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

Check if cyclic timer elapsed and reset if so.

Returns

true: timer elapsed and restarted, false: still running

Definition at line 15 of file Timer.cpp.

4.9.3.3 getTime()

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

Get measured time since and reset timer.

Returns

Elapsed time in ms

Definition at line 27 of file Timer.cpp.

4.9.3.4 setCycle()

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

Set or reset cycle time.

Parameters

cycle	Cycle time in ms

Definition at line 10 of file Timer.cpp.

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

- Timer.h
- Timer.cpp

4.10 XPLDirect Class Reference

Public Member Functions

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

4.10.1 Detailed Description

Definition at line 81 of file XPLDirect.h.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 XPLDirect()

```
XPLDirect::XPLDirect ( )
```

Definition at line 11 of file XPLDirect.cpp.

4.10.3 Member Function Documentation

4.10.3.1 allDataRefsRegistered()

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

Definition at line 417 of file XPLDirect.cpp.

4.10.3.2 begin()

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

Definition at line 18 of file XPLDirect.cpp.

4.10.3.3 commandEnd()

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

Definition at line 97 of file XPLDirect.cpp.

4.10.3.4 commandStart()

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

Definition at line 89 of file XPLDirect.cpp.

4.10.3.5 commandTrigger() [1/2]

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

Definition at line 73 of file XPLDirect.cpp.

4.10.3.6 commandTrigger() [2/2]

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

Definition at line 81 of file XPLDirect.cpp.

4.10.3.7 connectionStatus()

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

Definition at line 105 of file XPLDirect.cpp.

4.10.3.8 datarefsUpdated()

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

Definition at line 133 of file XPLDirect.cpp.

4.10.3.9 hasUpdated()

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

Definition at line 123 of file XPLDirect.cpp.

4.10.3.10 registerCommand()

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

Definition at line 525 of file XPLDirect.cpp.

4.10.3.11 registerDataRef() `[1/5]`

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

Definition at line 505 of file XPLDirect.cpp.

4.10.3.12 registerDataRef() `[2/5]`

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

Definition at line 464 of file XPLDirect.cpp.

4.10.3.13 registerDataRef() `[3/5]`

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

Definition at line 485 of file XPLDirect.cpp.

4.10.3.14 registerDataRef() `[4/5]`

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

Definition at line 422 of file XPLDirect.cpp.

4.10.3.15 registerDataRef() [5/5]

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

Definition at line 443 of file XPLDirect.cpp.

4.10.3.16 sendDebugMessage()

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

Definition at line 110 of file XPLDirect.cpp.

4.10.3.17 sendResetRequest()

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

Definition at line 159 of file XPLDirect.cpp.

4.10.3.18 sendSpeakMessage()

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

Definition at line 116 of file XPLDirect.cpp.

4.10.3.19 xloop()

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

Definition at line 28 of file XPLDirect.cpp.

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

- XPLDirect.h
- XPLDirect.cpp

Chapter 5

File Documentation

5.1 AnalogIn.h

```
#ifndef AnalogIn_h
#define AnalogIn_h

#define AD_RES 10

enum Analog_t
{
  unipolar,
  bipolar
};

class AnalogIn
{
public:
  AnalogIn(uint8_t pin, Analog_t type);

  AnalogIn(uint8_t pin, Analog_t type, float timeConst);

  void handle();

  float value() { return _value; };

  int raw();

  void calibrate();

private:
  float _filterConst;
  float _value;
  float _scale;
  float _scalePos;
  float _scaleNeg;
  int _offset;
  uint8_t _pin;
};

#endif
```

5.2 Button.h

```
#ifndef Button_h
#define Button_h
#include <DigitalIn.h>

class Button
{
private:
  void _handle(bool input);

public:
  Button(uint8_t mux, uint8_t muxpin);

  Button(uint8_t pin) : Button(NOT_USED, pin){};

  void handle()                 { _handle(true); };

  void handle(bool input)       { _handle(input); };

  void handleXP()               { _handle(true); processCommand(); };

  void handleXP(bool input)     { _handle(input); processCommand(); };

  bool pressed()                { return _transition == transPressed  ? (_transition = transNone, true) : false; };

  bool released()               { return _transition == transReleased ? (_transition = transNone, true) : false; };

  bool engaged()                { return _state > 0; };

  void setCommand(int cmdPush)  { _cmdPush = cmdPush; };

  int getCommand()              { return _cmdPush; };

  void processCommand();

protected:
  enum
  {
    transNone,
    transPressed,
    transReleased
  };
  uint8_t _mux;
  uint8_t _pin;
  uint8_t _state;
  uint8_t _transition;
  int _cmdPush;
};

class RepeatButton : public Button
{
private:
  void _handle(bool input);

public:
  RepeatButton(uint8_t mux, uint8_t muxpin, uint32_t delay);

  RepeatButton(uint8_t pin, uint32_t delay) : RepeatButton(NOT_USED, pin, delay){};

  void handle()                 { _handle(true); };

  void handle(bool input)       { _handle(input); };

  void handleXP()               { _handle(true); processCommand(); };

  void handleXP(bool input)     { _handle(input); processCommand(); };

protected:
  uint32_t _delay;
  uint32_t _timer;
};

#endif
```

5.3 DigitalIn.h

```
#ifndef Mux_h
#define Mux_h

#ifndef MUX_MAX_NUMBER
#define MUX_MAX_NUMBER 6
#endif

#ifndef MCP_MAX_NUMBER
#define MCP_MAX_NUMBER 0
#endif

// Include i2c lib only when needed
#if MCP_MAX_NUMBER > 0
#include <Adafruit_MCP23X17.h>
#endif

#define NOT_USED 255

class DigitalIn_
{
```

```
public:
  DigitalIn_();

  void setMux(uint8_t s0, uint8_t s1, uint8_t s2, uint8_t s3);

  bool addMux(uint8_t pin);

#if MCP_MAX_NUMBER > 0
  bool addMCP(uint8_t adress);
#endif

  bool getBit(uint8_t mux, uint8_t muxpin);

  void handle();
private:
  uint8_t _s0, _s1, _s2, _s3;
  uint8_t _numPins;
  uint8_t _pin[MUX_MAX_NUMBER + MCP_MAX_NUMBER];
  int16_t _data[MUX_MAX_NUMBER + MCP_MAX_NUMBER];
#if MCP_MAX_NUMBER > 0
  uint8_t _numMCP;
  Adafruit_MCP23X17 _mcp[MCP_MAX_NUMBER];
#endif
};

extern DigitalIn_ DigitalIn;

#endif
```

5.4 Encoder.h

```
#ifndef Encoder_h
#define Encoder_h
#include <DigitalIn.h>

enum EncCmd_t
{
  encCmdUp,
  encCmdDown,
  encCmdPush
};

enum EncPulse_t
{
  enc1Pulse = 1,
  enc2Pulse = 2,
  enc4Pulse = 4
};

class Encoder
{
public:
  Encoder(uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses);

  Encoder(uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses) : Encoder(NOT_USED, pin1, pin2,
pin3, pulses) {}

  void handle();

  void handleXP()   { handle(); processCommand(); };

  int16_t pos()     { return _count; };

  bool up()         { return _count >= _pulses ? (_count -= _pulses, true) : false; };

  bool down()       { return _count <= -_pulses ? (_count += _pulses, true) : false; };

  bool pressed()    { return _transition == transPressed  ? (_transition = transNone, true) : false;
};

  bool released()   { return _transition == transReleased ? (_transition = transNone, true) : false;
};

  bool engaged()    { return _state > 0; };

  void setCommand(int cmdUp, int cmdDown, int cmdPush);

  void setCommand(int cmdUp, int cmdDown) { setCommand(cmdUp, cmdDown, -1); };

  int getCommand(EncCmd_t cmd);

  void processCommand();
private:
  enum
  {
    transNone,
    transPressed,
    transReleased
  };
  uint8_t _mux;
  uint8_t _pin1, _pin2, _pin3;
  int8_t _count;
  uint8_t _pulses;
  uint8_t _state;
  uint8_t _debounce;
  uint8_t _transition;
  int _cmdUp;
  int _cmdDown;
  int _cmdPush;
};

#endif
```

5.5 LedShift.h

```
#ifndef Led_h
#define Led_h

enum led_t
{
  ledOff,
  ledSlow,
  ledMedium,
  ledFast,
  ledOn
};

class LedShift
{
public:
  LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT);

  void set(uint8_t pin, led_t mode);

  void set_all(led_t mode);

  void handle();

private:
  void _send();
  void _update(uint8_t pin);
  uint8_t _pin_DAI;
  uint8_t _pin_DCK;
  uint8_t _pin_LAT;
  uint16_t _state;
  led_t _mode[16];
  uint8_t _count;
  unsigned long _timer;
};

#endif
```

5.6 Switch.h

```
#ifndef Switch_h
#define Switch_h
#include <DigitalIn.h>

class Switch
{
public:
  Switch(uint8_t mux, uint8_t pin);

  Switch(uint8_t pin) : Switch (NOT_USED, pin) {};

  void handle();

  void handleXP() { handle(); processCommand(); };

  bool on()       { return _state == switchOn; };

  bool off()      { return _state == switchOff; };

  void setCommand(int cmdOn, int cmdOff);

  int getCommand();

  void processCommand();

  float value(float onValue, float offValue) { return on() ? onValue : offValue; };

private:
  enum SwState_t
  {
    switchOff,
    switchOn
  };
  uint8_t _mux;
  uint8_t _pin;
  uint8_t _debounce;
  uint8_t _state;
  bool _transition;
  int _cmdOff;
  int _cmdOn;
};

class Switch2
{
public:
  Switch2(uint8_t mux, uint8_t pin1, uint8_t pin2);

  Switch2(uint8_t pin1, uint8_t pin2) : Switch2(NOT_USED, pin1, pin2) {}

  void handle();

  void handleXP() { handle(); processCommand(); };

  bool off()      { return _state == switchOff; };

  bool on1()      { return _state == switchOn1; };

  bool on2()      { return _state == switchOn2; };

  void setCommand(int cmdUp, int cmdDown);

  void setCommand(int cmdOn1, int cmdOff, int cmdOn2);

  int getCommand();

  void processCommand();

  float value(float on1Value, float offValue, float on2value) { return (on1() ? on1Value : on2() ? on2value : offValue); };

private:
  enum SwState_t
  {
    switchOff,
    switchOn1,
    switchOn2
  };
  uint8_t _mux;
  uint8_t _pin1;
  uint8_t _pin2;
  uint8_t _lastState;
  uint8_t _debounce;
  uint8_t _state;
  bool _transition;
  int _cmdOff;
  int _cmdOn1;
  int _cmdOn2;
};

#endif
```

5.7 Timer.h

```
#ifndef SoftTimer_h
#define SoftTimer_h

class Timer
{
  public:
    Timer(float cycle = 0); // ms

    void setCycle(float cycle);

    bool elapsed();

    float getTime(); // ms

    long count();
  private:
    unsigned long _cycleTime;
    unsigned long _lastUpdateTime;
    long _count;
};

#endif
```

5.8 XPLDevices.h

```
#ifndef XPLDevices_h
#define XPLDevices_h

#include <XPLDirect.h>
#include <Button.h>
#include <Encoder.h>
#include <Switch.h>
#include <LedShift.h>
#include <Timer.h>
#include <DigitalIn.h>
#include <AnalogIn.h>

#endif
```

5.9 XPLDirect.h

```
/*
  XPLDirect.h - Library for serial interface to Xplane SDK.
  Created by Michael Gerlicher,  September 2020.
  To report problems, download updates and examples, suggest enhancements or get technical support,
please visit my patreon page:
     www.patreon.com/curiosityworkshop
  Stripped down to Minimal Version by mrusk, February 2023
*/

#ifndef XPLDirect_h
#define XPLDirect_h

#ifndef XPLDIRECT_MAXDATAREFS_ARDUINO
#define XPLDIRECT_MAXDATAREFS_ARDUINO 100 // This can be changed to suit your needs and capabilities
of your board.
#endif

#ifndef XPLDIRECT_MAXCOMMANDS_ARDUINO
#define XPLDIRECT_MAXCOMMANDS_ARDUINO 100  // Same here.
#endif

#define XPLDIRECT_RX_TIMEOUT 500 // after detecting a frame header, how long will we wait to receive
the rest of the frame.  (default 500)

#ifndef XPLMAX_PACKETSIZE
#define XPLMAX_PACKETSIZE 80  // Probably leave this alone. If you need a few extra bytes of RAM it
could be reduced, but it needs to
                              // be as long as the longest dataref name + 10.  If you are using
datarefs
                              // that transfer strings it needs to be big enough for those too.
(default 200)
#endif

#ifndef XPL_USE_PROGMEM
#define XPL_USE_PROGMEM 1
#endif

// STOP! Dont change any other defines in this header!

#ifdef XPL_USE_PROGMEM
// use Flash for strings, requires F() macro for strings in all registration calls
  typedef const __FlashStringHelper XPString_t;
#else
  typedef const char XPString_t;
#endif

```

```
#define XPLDIRECT_BAUDRATE 115200   // don't mess with this, it needs to match the plugin which won't change
#define XPLDIRECT_PACKETHEADER '<'  // ...or this
#define XPLDIRECT_PACKETTRAILER '>' // ...or this
#define XPLDIRECT_VERSION 2106171   // The plugin will start to verify that a compatible version is being used
#define XPLDIRECT_ID 0              // Used for relabled plugins to identify the company.  0 = normal distribution version

#define XPLERROR 'E'                    // %s         general error
#define XPLRESPONSE_NAME '0'
#define XPLRESPONSE_DATAREF '3'         // %3.3i%s    dataref handle, dataref name
#define XPLRESPONSE_COMMAND '4'         // %3.3i%s    command handle, command name
#define XPLRESPONSE_VERSION 'V'
#define XPLCMD_PRINTDEBUG '1'
#define XPLCMD_RESET '2'
#define XPLCMD_SPEAK 'S'                // speak string
#define XPLCMD_SENDNAME 'a'
#define XPLREQUEST_REGISTERDATAREF 'b'  // %1.1i%2.2i%5.5i%s RWMode, array index (0 for non array datarefs), divider to decrease resolution, dataref name
#define XPLREQUEST_REGISTERCOMMAND 'm'  // just the name of the command to register
#define XPLREQUEST_NOREQUESTS 'c'       // nothing to request
#define XPLREQUEST_REFRESH 'd'          // the plugin will call this once xplane is loaded in order to get fresh updates from arduino handles that write
#define XPLCMD_DUMPREGISTRATIONS 'Z'    // for debug purposes only (disabled)
#define XPLCMD_DATAREFUPDATE 'e'
#define XPLCMD_SENDREQUEST 'f'
#define XPLCMD_DEVICEREADY 'g'
#define XPLCMD_DEVICENOTREADY 'h'
#define XPLCMD_COMMANDSTART 'i'
#define XPLCMD_COMMANDEND 'j'
#define XPLCMD_COMMANDTRIGGER 'k' //  %3.3i%3.3i   command handle, number of triggers
#define XPLCMD_SENDVERSION 'v'    // we will respond with current build version
#define XPL_EXITING 'x'           // MG 03/14/2023: xplane sends this to the arduino device during normal shutdown of xplane.  It may not happen if xplane crashes.

#define XPL_READ 1
#define XPL_WRITE 2
#define XPL_READWRITE 3

#define XPL_DATATYPE_INT 1
#define XPL_DATATYPE_FLOAT 2
#define XPL_DATATYPE_STRING 3

class XPLDirect
{
public:
  XPLDirect();
  void begin(const char *devicename); // parameter is name of your device for reference
  int connectionStatus(void);
  int commandTrigger(int commandHandle);                    // triggers specified command 1 time;
  int commandTrigger(int commandHandle, int triggerCount);  // triggers specified command triggerCount times.
  int commandStart(int commandHandle);
  int commandEnd(int commandHandle);
  int datarefsUpdated();      // returns true if xplane has updated any datarefs since last call to datarefsUpdated()
  int hasUpdated(int handle); // returns true if xplane has updated this dataref since last call to hasUpdated()
  int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int *value);
  int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int *value, int index);
  int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float *value);
  int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float *value, int index);
  int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, char* value);
  int registerCommand(XPString_t *commandName);
  int sendDebugMessage(const char *msg);
  int sendSpeakMessage(const char* msg);
  int allDataRefsRegistered(void);
  void sendResetRequest(void);
  int xloop(void); // where the magic happens!
private:
  void _processSerial();
  void _processPacket();
  void _sendPacketInt(int command, int handle, long int value); // for ints
  void _sendPacketFloat(int command, int handle, float value);  // for floats
  void _sendPacketVoid(int command, int handle);                // just a command with a handle
  void _sendPacketString(int command, char *str);               // for a string
  void _transmitPacket();
  void _sendname();
  void _sendVersion();
  int _getHandleFromFrame();
  int _getPayloadFromFrame(long int *);
  int _getPayloadFromFrame(float *);
  int _getPayloadFromFrame(char *);

  Stream *streamPtr;
  char *_deviceName;
  char _receiveBuffer[XPLMAX_PACKETSIZE];
  int _receiveBufferBytesReceived;
  char _sendBuffer[XPLMAX_PACKETSIZE];
  int _connectionStatus;
  int _dataRefsCount;
  struct _dataRefStructure
  {
    int dataRefHandle;
    byte dataRefRWType;       // XPL_READ, XPL_WRITE, XPL_READWRITE
    byte dataRefVARType;      // XPL_DATATYPE_INT 1, XPL_DATATYPE_FLOAT  2   XPL_DATATYPE_STRING 3
    float divider;            // tell the host to reduce resolution by dividing then remultiplying by
this number to reduce traffic.   (ie .02, .1, 1, 5, 10, 100, 1000 etc)
    byte forceUpdate;         // in case xplane plugin asks for a refresh
    unsigned long updateRate; // maximum update rate in milliseconds, 0 = every change
    unsigned long lastUpdateTime;
    XPString_t *dataRefName;
    void *latestValue;
    union {
      long int lastSentIntValue;
      float lastSentFloatValue;
    };
    byte updatedFlag; //  True if xplane has updated this dataref.  Gets reset when we call hasUpdated
method.
    byte arrayIndex;  // for datarefs that speak in arrays
  } *_dataRefs[XPLDIRECT_MAXDATAREFS_ARDUINO];
  int _commandsCount;
  struct _commandStructure
  {
    int commandHandle;
    XPString_t *commandName;
  } *_commands[XPLDIRECT_MAXCOMMANDS_ARDUINO];
  byte _allDataRefsRegistered; // becomes true if all datarefs have been registered
  byte _datarefsUpdatedFlag;   // becomes true if any datarefs have been updated from xplane since
last call to datarefsUpdated()
};

extern XPLDirect XP;

#endif
```

5.10 AnalogIn.cpp

```
#include <Arduino.h>
#include "AnalogIn.h"

#define FULL_SCALE ((1 « AD_RES) - 1)
#define HALF_SCALE (1 « (AD_RES - 1))

AnalogIn::AnalogIn(uint8_t pin, Analog_t type)
{
  _pin = pin;
  _filterConst = 1.0;
  _scale = 1.0;
  pinMode(_pin, INPUT);
  if (type == bipolar)
  {
    _offset = HALF_SCALE;
    _scalePos = _scale / HALF_SCALE;
    _scaleNeg = _scale / HALF_SCALE;
  }
  else
  {
    _offset = 0;
    _scalePos = _scale / FULL_SCALE;
    _scaleNeg = 0.0;
  }
}

AnalogIn::AnalogIn(uint8_t pin, Analog_t type, float timeConst) : AnalogIn(pin, type)
{
  _filterConst = 1.0 / timeConst;
}

void AnalogIn::handle()
{
  int _raw = raw();
  _value = (_filterConst * _raw * (_raw >= 0 ? _scalePos : _scaleNeg)) + (1.0 - _filterConst) *
_value;
}
```

```

int AnalogIn::raw()
{
  return analogRead(_pin) - _offset;
}

void AnalogIn::calibrate()
{
  long sum = 0;
  for (int i = 0; i < 64; i++)
  {
    sum += analogRead(_pin);
  }
  _offset = (int)(sum / 64);
  _scalePos = (_offset < FULL_SCALE) ? _scale / (float)(FULL_SCALE - _offset) : 1.0;
  _scaleNeg = (_offset > 0)? _scale / (float)(_offset) : 1.0;
}
```

5.11 Button.cpp

```
#include <Arduino.h>
#include <XPLDirect.h>
#include "Button.h"

#ifndef DEBOUNCE_DELAY
#define DEBOUNCE_DELAY 20
#endif

// Buttons
Button::Button(uint8_t mux, uint8_t pin)
{
  _mux = mux;
  _pin = pin;
  _state = 0;
  _transition = 0;
  _cmdPush = -1;
  pinMode(_pin, INPUT_PULLUP);
}

// use additional bit for input masking
void Button::_handle(bool input)
{
  if (DigitalIn.getBit(_mux, _pin) && input)
  {
    if (_state == 0)
    {
      _state = DEBOUNCE_DELAY;
      _transition = transPressed;
    }
  }
  else if (_state > 0)
  {
    if (--_state == 0)
    {
      _transition = transReleased;
    }
  }
}

void Button::processCommand()
{
  if (pressed())
  {
    XP.commandStart(_cmdPush);
  }
  if (released())
  {
    XP.commandEnd(_cmdPush);
  }
}

RepeatButton::RepeatButton(uint8_t mux, uint8_t pin, uint32_t delay) : Button(mux, pin)
{
  _delay = delay;
  _timer = 0;
}

void RepeatButton::_handle(bool input)
{
  if (DigitalIn.getBit(_mux, _pin) && input)
  {
    if (_state == 0)
    {
      _state = DEBOUNCE_DELAY;
      _transition = transPressed;
      _timer = millis() + _delay;
    }
    else if (_delay > 0 && (millis() >= _timer))
    {
      _state = DEBOUNCE_DELAY;
      _transition = transPressed;
      _timer += _delay;
    }
  }
  else if (_state > 0)
  {
    if (--_state == 0)
    {
      _transition = transReleased;
    }
  }
}
```

5.12 DigitalIn.cpp

```
#include <Arduino.h>
#include "DigitalIn.h"

#define MCP_PIN 254

// constructor
DigitalIn_::DigitalIn_()
{
  _numPins = 0;
  for (uint8_t mux = 0; mux < MUX_MAX_NUMBER; mux++)
  {
    _pin[mux] = NOT_USED;
  }
  _s0 = NOT_USED;
  _s1 = NOT_USED;
  _s2 = NOT_USED;
  _s3 = NOT_USED;
}

// configure 74HC4067 adress pins S0-S3
void DigitalIn_::setMux(uint8_t s0, uint8_t s1, uint8_t s2, uint8_t s3)
{
  _s0 = s0;
  _s1 = s1;
  _s2 = s2;
  _s3 = s3;
  pinMode(_s0, OUTPUT);
  pinMode(_s1, OUTPUT);
  pinMode(_s2, OUTPUT);
  pinMode(_s3, OUTPUT);
}

// Add a 74HC4067
bool DigitalIn_::addMux(uint8_t pin)
{
  if (_numPins >= MUX_MAX_NUMBER)
  {
    return false;
  }
  _pin[_numPins++] = pin;
  pinMode(pin, INPUT);
  return true;
}

#if MCP_MAX_NUMBER > 0
// Add a MCP23017
bool DigitalIn_::addMCP(uint8_t adress)
{
  if (_numMCP >= MCP_MAX_NUMBER)
  {
    return false;
  }
  if (!_mcp[_numMCP].begin_I2C(adress, &Wire))
  {
    return false;
  }
  for (int i = 0; i < 16; i++)
  {
    // TODO: register write iodir = 0xffff, ipol = 0xffff, gppu = 0xffff
    _mcp[_numMCP].pinMode(i, INPUT_PULLUP);
  }
  _numMCP++;
  _pin[_numPins++] = MCP_PIN;
  return true;
}
#endif

// Gets specific pin from mux, number according to initialization order
bool DigitalIn_::getBit(uint8_t mux, uint8_t pin)
{
  if (mux == NOT_USED)
  {
    return !digitalRead(pin);
  }
  return bitRead(_data[mux], pin);
}

// read all inputs together -> base for board specific optimization by using byte read
void DigitalIn_::handle()
{
  // only if Mux Pins present
#if MCP_MAX_NUMBER > 0
  if (_numPins > _numMCP)
#else
  if (_numPins > 0)
#endif
  {
    for (uint8_t muxpin = 0; muxpin < 16; muxpin++)
    {
      digitalWrite(_s0, bitRead(muxpin, 0));
      digitalWrite(_s1, bitRead(muxpin, 1));
      digitalWrite(_s2, bitRead(muxpin, 2));
      digitalWrite(_s3, bitRead(muxpin, 3));
      for (uint8_t mux = 0; mux < _numPins; mux++)
      {
        if (_pin[mux] != MCP_PIN)
        {
          bitWrite(_data[mux], muxpin, !digitalRead(_pin[mux]));
        }
      }
    }
  }
#if MCP_MAX_NUMBER > 0
  int mcp = 0;
  for (uint8_t mux = 0; mux < _numPins; mux++)
  {
    if (_pin[mux] == MCP_PIN)
    {
      _data[mux] = ~_mcp[mcp++].readGPIOAB();
    }
  }
#endif
}

DigitalIn_ DigitalIn;
```

5.13 Encoder.cpp

```
#include <Arduino.h>
#include <XPLDirect.h>
#include "Encoder.h"

#ifndef DEBOUNCE_DELAY
#define DEBOUNCE_DELAY 20
#endif

// Encoder with button functionality on MUX
Encoder::Encoder(uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses)
{
  _mux = mux;
  _pin1 = pin1;
  _pin2 = pin2;
  _pin3 = pin3;
  _pulses = pulses;
  _count = 0;
  _state = 0;
  _transition = transNone;
  _cmdUp = -1;
  _cmdDown = -1;
  _cmdPush = -1;
  pinMode(_pin1, INPUT_PULLUP);
  pinMode(_pin2, INPUT_PULLUP);
  if (_pin3 != NOT_USED)
  {
    pinMode(_pin3, INPUT_PULLUP);
  }
}

// real time handling
void Encoder::handle()
{
  // collect new state
  _state = ((_state & 0x03) « 2) | (DigitalIn.getBit(_mux, _pin2) « 1) | (DigitalIn.getBit(_mux,
_pin1));
  // evaluate state change
  if (_state == 1 || _state == 7 || _state == 8 || _state == 14)
  {
    _count++;
  }
  if (_state == 2 || _state == 4 || _state == 11 || _state == 13)
  {
    _count--;
  }
  if (_state == 3 || _state == 12)
  {
    _count += 2;
  }
  if (_state == 6 || _state == 9)
  {
    _count -= 2;
  }

  // optional button functionality
  if (_pin3 != NOT_USED)
  {
    if (DigitalIn.getBit(_mux, _pin3))
    {
      if (_debounce == 0)
      {
        _debounce = DEBOUNCE_DELAY;
        _transition = transPressed;
      }
    }
    else if (_debounce > 0)
    {
      if (--_debounce == 0)
      {
        _transition = transReleased;
      }
    }
  }
}

void Encoder::setCommand(int cmdUp, int cmdDown, int cmdPush)
{
  _cmdUp = cmdUp;
  _cmdDown = cmdDown;
  _cmdPush = cmdPush;
}

int Encoder::getCommand(EncCmd_t cmd)
{
  switch (cmd)
  {
  case encCmdUp:
    return _cmdUp;
    break;
  case encCmdDown:
    return _cmdDown;
    break;
  case encCmdPush:
    return _cmdPush;
    break;
  default:
    return -1;
    break;
  }
}

void Encoder::processCommand()
{
  if (up())
  {
    XP.commandTrigger(_cmdUp);
  }
  if (down())
  {
    XP.commandTrigger(_cmdDown);
  }
  if (_cmdPush >= 0)
  {
    if (pressed())
    {
      XP.commandStart(_cmdPush);
    }
    if (released())
    {
      XP.commandEnd(_cmdPush);
    }
  }
}
```

5.14 LedShift.cpp

```
#include <Arduino.h>
#include "LedShift.h"

#define BLINK_DELAY 150

LedShift::LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT)
{
  _pin_DAI = pin_DAI;
  _pin_DCK = pin_DCK;
  _pin_LAT = pin_LAT;
  _count = 0;
  _state = 0;
  _timer = millis() + BLINK_DELAY;
  for (int pin = 0; pin < 16; pin++)
  {
    _mode[pin] = ledOff;
  }
  pinMode(_pin_DAI, OUTPUT);
  pinMode(_pin_DCK, OUTPUT);
  pinMode(_pin_LAT, OUTPUT);
  digitalWrite(_pin_DAI, LOW);
  digitalWrite(_pin_DCK, LOW);
  digitalWrite(_pin_LAT, LOW);
  _send();
}

void LedShift::_send()
{
  shiftOut(_pin_DAI, _pin_DCK, MSBFIRST, (_state & 0xFF00) » 8);
  shiftOut(_pin_DAI, _pin_DCK, MSBFIRST, (_state & 0x00FF));
  digitalWrite(_pin_LAT, HIGH);
  digitalWrite(_pin_LAT, LOW);
}

void LedShift::_update(uint8_t pin)
{
  switch (_mode[pin])
  {
  case ledOn:
    bitSet(_state, pin);
    break;
  case ledFast:
    bitWrite(_state, pin, bitRead(_count, 0));
    break;
  case ledMedium:
    bitWrite(_state, pin, bitRead(_count, 1));
    break;
  case ledSlow:
    bitWrite(_state, pin, bitRead(_count, 2));
    break;
  default:
    bitClear(_state, pin);
  }
}

void LedShift::set(uint8_t pin, led_t mode)
{
  _mode[pin] = mode;
  _update(pin);
}

void LedShift::set_all(led_t mode)
{
  for (int pin = 0; pin < 16; pin++)
  {
    _mode[pin] = mode;
    _update(pin);
  }
}

```

```
void LedShift::handle()
{
  if (millis() >= _timer)
  {
    _timer += BLINK_DELAY;
    _count = (_count + 1) % 8;
    for (int pin = 0; pin < 16; pin++)
    {
      _update(pin);
    }
  }
  _send();
}
```

5.15 Switch.cpp

```
#include <Arduino.h>
#include <XPLDirect.h>
#include "Switch.h"

#ifndef DEBOUNCE_DELAY
#define DEBOUNCE_DELAY 20
#endif

Switch::Switch(uint8_t mux, uint8_t pin)
{
  _mux = mux;
  _pin = pin;
  _state = switchOff;
  _cmdOff = -1;
  _cmdOn = -1;
  pinMode(_pin, INPUT_PULLUP);
}

void Switch::handle()
{
  if (_debounce > 0)
  {
    _debounce--;
  }
  else
  {
    SwState_t input = switchOff;
    if (DigitalIn.getBit(_mux, _pin))
    {
      input = switchOn;
    }
    if (input != _state)
    {
      _debounce = DEBOUNCE_DELAY;
      _state = input;
      _transition = true;
    }
  }
}

void Switch::setCommand(int cmdOn, int cmdOff)
{
  _cmdOn = cmdOn;
  _cmdOff = cmdOff;
}

int Switch::getCommand()
{
  switch (_state)
  {
  case switchOff:
    return _cmdOff;
    break;
  case switchOn:
    return _cmdOn;
    break;
  default:
    return -1;
    break;
  }
}

void Switch::processCommand()
{
  if (_transition)
  {
    XP.commandTrigger(getCommand());
    _transition = false;
  }
}

// Switch 2

Switch2::Switch2(uint8_t mux, uint8_t pin1, uint8_t pin2)
{
  _mux = mux;
  _pin1 = pin1;
  _pin2 = pin2;
  _state = switchOff;
  _cmdOff = -1;
  _cmdOn1 = -1;
  if (_mux == NOT_USED)
  {
    pinMode(_pin1, INPUT_PULLUP);
    pinMode(_pin2, INPUT_PULLUP);
  }
}

void Switch2::handle()
{
  if (_debounce > 0)
  {
    _debounce--;
  }
  else
  {
    SwState_t input = switchOff;
    if (DigitalIn.getBit(_mux, _pin1))
    {
      input = switchOn1;
    }
    else if (DigitalIn.getBit(_mux, _pin2))
    {
      input = switchOn2;
    }
    if (input != _state)
    {
      _debounce = DEBOUNCE_DELAY;
      _lastState = _state;
      _state = input;
      _transition = true;
    }
  }
}

void Switch2::setCommand(int cmdUp, int cmdDown)
{
  _cmdOn1 = cmdUp;
  _cmdOff = cmdDown;
  _cmdOn2 = -1;
}

void Switch2::setCommand(int cmdOn1, int cmdOff, int cmdOn2)
{
  _cmdOn1 = cmdOn1;
  _cmdOff = cmdOff;
  _cmdOn2 = cmdOn2;
}

int Switch2::getCommand()
{
  if (_cmdOn2 == -1)
  {
    if (_state == switchOn1)
    {
      return _cmdOn1;
    }
    if (_state == switchOff && _lastState == switchOn1)
    {
      return _cmdOff;
    }
    if (_state == switchOn2)
    {
      return _cmdOff;
    }
    if (_state == switchOff && _lastState == switchOn2)
    {
      return _cmdOn1;
    }
  }
  else
  {
    if (_state == switchOn1)
    {
      return _cmdOn1;
    }
    if (_state == switchOff)
    {
      return _cmdOff;
    }
    if (_state == switchOn2)
    {
      return _cmdOn2;
    }
  }
  return -1;
}

void Switch2::processCommand()
{
  if (_transition)
  {
    XP.commandTrigger(getCommand());
    _transition = false;
  }
}
```

5.16 Timer.cpp

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

Timer::Timer(float cycle)
{
  setCycle(cycle);
  _lastUpdateTime = micros();
}

void Timer::setCycle(float cycle)
{
  _cycleTime = (unsigned long)(cycle * 1000.0);
}

bool Timer::elapsed()
{
  _count++;
  unsigned long now = micros();
  if (now > _lastUpdateTime + _cycleTime)
  {
   _lastUpdateTime = now;
   return true;
  }
  return false;
}

float Timer::getTime()
{
  unsigned long now = micros();
  unsigned long cycle = now - _lastUpdateTime;
  _lastUpdateTime = now;
  return (float)cycle * 0.001;
}

long Timer::count()
{
  long ret = _count;
  _count = 0;
  return ret;
}
```

5.17 XPLDirect.cpp

```
/*
  XPLDirect.cpp
  Created by Michael Gerlicher, September 2020.
  Modified by mrusk, March 2023
*/

#include <arduino.h>
#include "XPLDirect.h"

// Methods
XPLDirect::XPLDirect()
```

```cpp
{
  Serial.begin(XPLDIRECT_BAUDRATE);
  streamPtr = &Serial;
  streamPtr->setTimeout(XPLDIRECT_RX_TIMEOUT);
}

void XPLDirect::begin(const char *devicename)
{
  _deviceName = (char *)devicename;
  _connectionStatus = 0;
  _dataRefsCount = 0;
  _commandsCount = 0;
  _allDataRefsRegistered = 0;
  _receiveBuffer[0] = 0;
}

int XPLDirect::xloop(void)
{
  _processSerial();
  if (!_allDataRefsRegistered)
  {
    return _connectionStatus;
  }
  // process datarefs to send
  for (int i = 0; i < _dataRefsCount; i++)
  {
    if (_dataRefs[i]->dataRefHandle >= 0 && (_dataRefs[i]->dataRefRWType == XPL_WRITE || _dataRefs[i]->dataRefRWType == XPL_READWRITE))
    {
      if ((millis() - _dataRefs[i]->lastUpdateTime > _dataRefs[i]->updateRate) || _dataRefs[i]->forceUpdate)
      {
        switch (_dataRefs[i]->dataRefVARType)
        {
        case XPL_DATATYPE_INT:
          if (*(long int *)_dataRefs[i]->latestValue != _dataRefs[i]->lastSentIntValue)
          {
            _sendPacketInt(XPLCMD_DATAREFUPDATE, _dataRefs[i]->dataRefHandle, *(long int *)_dataRefs[i]->latestValue);
            _dataRefs[i]->lastSentIntValue = *(long int *)_dataRefs[i]->latestValue;
            _dataRefs[i]->lastUpdateTime = millis();
            _dataRefs[i]->forceUpdate = 0;
          }
          break;
        case XPL_DATATYPE_FLOAT:
          if (_dataRefs[i]->divider > 0)
          {
            *(float *)_dataRefs[i]->latestValue = ((int)(*(float *)_dataRefs[i]->latestValue / _dataRefs[i]->divider) * _dataRefs[i]->divider);
          }
          if (*(float *)_dataRefs[i]->latestValue != _dataRefs[i]->lastSentFloatValue)
          {
            _sendPacketFloat(XPLCMD_DATAREFUPDATE, _dataRefs[i]->dataRefHandle, *(float *)_dataRefs[i]->latestValue);
            _dataRefs[i]->lastSentFloatValue = *(float *)_dataRefs[i]->latestValue;
            _dataRefs[i]->lastUpdateTime = millis();
            _dataRefs[i]->forceUpdate = 0;
          }
          break;
        }
      }
    }
  }
  return _connectionStatus;
}

int XPLDirect::commandTrigger(int commandHandle)
{
  if (!_commands[commandHandle])
    return -1; // inactive command
  _sendPacketInt(XPLCMD_COMMANDTRIGGER, _commands[commandHandle]->commandHandle, 1);
  return 0;
}

int XPLDirect::commandTrigger(int commandHandle, int triggerCount)
{
  if (!_commands[commandHandle])
    return -1; // inactive command
  _sendPacketInt(XPLCMD_COMMANDTRIGGER, _commands[commandHandle]->commandHandle, (long int)triggerCount);
  return 0;
}

int XPLDirect::commandStart(int commandHandle)
{
  if (!_commands[commandHandle])
    return -1; // inactive command
  _sendPacketVoid(XPLCMD_COMMANDSTART, _commands[commandHandle]->commandHandle);
  return 0;
}

int XPLDirect::commandEnd(int commandHandle)
{
  if (!_commands[commandHandle])
    return -1; // inactive command
  _sendPacketVoid(XPLCMD_COMMANDEND, _commands[commandHandle]->commandHandle);
  return 0;
}

int XPLDirect::connectionStatus()
{
  return _connectionStatus;
}

int XPLDirect::sendDebugMessage(const char* msg)
{
  _sendPacketString(XPLCMD_PRINTDEBUG, (char *)msg);
  return 1;
}

int XPLDirect::sendSpeakMessage(const char* msg)
{
  _sendPacketString(XPLCMD_SPEAK, (char *)msg);
  return 1;
}


int XPLDirect::hasUpdated(int handle)
{
  if (_dataRefs[handle]->updatedFlag)
  {
    _dataRefs[handle]->updatedFlag = false;
    return true;
  }
  return false;
}

int XPLDirect::datarefsUpdated()
{
  if (_datarefsUpdatedFlag)
  {
    _datarefsUpdatedFlag = false;
    return true;
  }
  return false;
}

void XPLDirect::_sendname()
{
  if (_deviceName != NULL)
  {
    _sendPacketString(XPLRESPONSE_NAME, _deviceName);
  }
}

void XPLDirect::_sendVersion()
{
  if (_deviceName != NULL)
  {
    _sendPacketInt(XPLRESPONSE_VERSION, XPLDIRECT_ID, XPLDIRECT_VERSION);
  }
}

void XPLDirect::sendResetRequest()
{
  if (_deviceName != NULL)
  {
    _sendPacketVoid(XPLCMD_RESET, 0);
  }
}

void XPLDirect::_processSerial()
{
  while (streamPtr->available() && _receiveBuffer[0] != XPLDIRECT_PACKETHEADER)
  {
    _receiveBuffer[0] = (char)streamPtr->read();
  }
  if (_receiveBuffer[0] != XPLDIRECT_PACKETHEADER)
  {
    return;
  }
  _receiveBufferBytesReceived = streamPtr->readBytesUntil(XPLDIRECT_PACKETTRAILER, (char
*)&_receiveBuffer[1], XPLMAX_PACKETSIZE - 1);
  if (_receiveBufferBytesReceived == 0)
  {
    _receiveBuffer[0] = 0;
    return;
  }
  _receiveBuffer[++_receiveBufferBytesReceived] = XPLDIRECT_PACKETTRAILER;
  _receiveBuffer[++_receiveBufferBytesReceived] = 0; // old habits die hard.
  _processPacket();
  _receiveBuffer[0] = 0;
}

void XPLDirect::_processPacket()
{
  int i;

  switch (_receiveBuffer[1])
  {
  case XPLCMD_RESET:
    _connectionStatus = false;
    break;

  case XPL_EXITING :          // MG 03/14/2023:  Added protocol code so the device will know if xplane has shut down normally.
    _connectionStatus = false;
    break;

  case XPLCMD_SENDNAME:
    _sendname();
    _connectionStatus = true;             // not considered active till you know my name
    for (i = 0; i < _dataRefsCount; i++) // also, if name was requested reset active datarefs and commands
    {
      _dataRefs[i]->dataRefHandle = -1; //  invalid again until assigned by Xplane
    }
    for (i = 0; i < _commandsCount; i++)
    {
      _commands[i]->commandHandle = -1;
    }
    break;

  case XPLCMD_SENDVERSION:
  {
    _sendVersion();
    break;
  }

  case XPLRESPONSE_DATAREF:
    for (int i = 0; i < _dataRefsCount; i++)
    {
      if (strncmp_PF((char *)&_receiveBuffer[5], (uint_farptr_t)_dataRefs[i]->dataRefName, strlen_PF((uint_farptr_t)_dataRefs[i]->dataRefName)) == 0 && _dataRefs[i]->dataRefHandle == -1)
      {
        _dataRefs[i]->dataRefHandle = _getHandleFromFrame(); // parse the refhandle
        _dataRefs[i]->updatedFlag = true;
        i = _dataRefsCount; // end checking
      }
    }
    break;

  case XPLRESPONSE_COMMAND:
    for (int i = 0; i < _commandsCount; i++)
    {
      if (strncmp_PF((char *)&_receiveBuffer[5], (uint_farptr_t)_commands[i]->commandName, strlen_PF((uint_farptr_t)_commands[i]->commandName)) == 0 && _commands[i]->commandHandle == -1)
      {
        _commands[i]->commandHandle = _getHandleFromFrame(); // parse the refhandle
        i = _commandsCount;                                 // end checking
      }
    }
    break;

  case XPLCMD_SENDREQUEST:
  {
    int packetSent = 0;
    int i = 0;
    while (!packetSent && i < _dataRefsCount && i < XPLDIRECT_MAXDATAREFS_ARDUINO) // send dataref registrations first
    {
      if (_dataRefs[i]->dataRefHandle == -1)
      { // some boards cant do sprintf with floats so this is a workaround
        sprintf(_sendBuffer, "%c%c%1.1i%2.2i%05i.%02i%S%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_REGISTERDATAREF, _dataRefs[i]->dataRefRWType, _dataRefs[i]->arrayIndex,
                (int)_dataRefs[i]->divider, (int)(_dataRefs[i]->divider * 100) % 100, (wchar_t *)_dataRefs[i]->dataRefName, XPLDIRECT_PACKETTRAILER);
        _transmitPacket();
        packetSent = 1;
      }
      i++;
    }
    i = 0;
    while (!packetSent && i < _commandsCount && i < XPLDIRECT_MAXCOMMANDS_ARDUINO) // now send command registrations
    {
      if (_commands[i]->commandHandle == -1)
      {
        sprintf(_sendBuffer, "%c%c%S%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_REGISTERCOMMAND, (wchar_t *)_commands[i]->commandName, XPLDIRECT_PACKETTRAILER);
        _transmitPacket();
        packetSent = 1;
      }
      i++;
    }
    if (!packetSent)
    {
      _allDataRefsRegistered = true;
      sprintf(_sendBuffer, "%c%c%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_NOREQUESTS, XPLDIRECT_PACKETTRAILER);
      _transmitPacket();
    }
    break;
  }

  case XPLCMD_DATAREFUPDATE:
  {
    int refhandle = _getHandleFromFrame();
    for (int i = 0; i < _dataRefsCount; i++)
    {
      if (_dataRefs[i]->dataRefHandle == refhandle && (_dataRefs[i]->dataRefRWType == XPL_READ || _dataRefs[i]->dataRefRWType == XPL_READWRITE))
      {
        if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_INT)
        {
          _getPayloadFromFrame((long int *)_dataRefs[i]->latestValue);
          _dataRefs[i]->lastSentIntValue = *(long int *)_dataRefs[i]->latestValue;
          _dataRefs[i]->updatedFlag = true;
          _datarefsUpdatedFlag = true;
        }
        if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_FLOAT)
        {
          _getPayloadFromFrame((float *)_dataRefs[i]->latestValue);
          _dataRefs[i]->lastSentFloatValue = *(float *)_dataRefs[i]->latestValue;
          _dataRefs[i]->updatedFlag = true;
          _datarefsUpdatedFlag = true;
        }
        if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_STRING)
        {
          _getPayloadFromFrame((char *)_dataRefs[i]->latestValue);
          _dataRefs[i]->updatedFlag = true;
          _datarefsUpdatedFlag = true;
        }
        i = _dataRefsCount; // skip the rest
      }
    }
    break;
  }
  case XPLREQUEST_REFRESH:
    for (int i = 0; i < _dataRefsCount; i++)
    {
      if (_dataRefs[i]->dataRefRWType == XPL_WRITE || _dataRefs[i]->dataRefRWType == XPL_READWRITE)
      {
        _dataRefs[i]->forceUpdate = 1; // bypass noise and timing filters
      }
    }
    break;

  default:
    break;
  }
}

void XPLDirect::_sendPacketInt(int command, int handle, long int value) // for ints
{
  if (handle >= 0)
  {
    sprintf(_sendBuffer, "%c%c%3.3i%ld%c", XPLDIRECT_PACKETHEADER, command, handle, value, XPLDIRECT_PACKETTRAILER);
    _transmitPacket();
  }
}

void XPLDirect::_sendPacketFloat(int command, int handle, float value) // for floats
{
  if (handle >= 0)
  {
    // some boards cant do sprintf with floats so this is a workaround.
    char tmp[16];
    dtostrf(value, 8, 6, tmp);
    sprintf(_sendBuffer, "%c%c%3.3i%s%c", XPLDIRECT_PACKETHEADER, command, handle, tmp,
XPLDIRECT_PACKETTRAILER);
    _transmitPacket();
  }
}

void XPLDirect::_sendPacketVoid(int command, int handle) // just a command with a handle
{
  if (handle >= 0)
  {
    sprintf(_sendBuffer, "%c%c%3.3i%c", XPLDIRECT_PACKETHEADER, command, handle,
XPLDIRECT_PACKETTRAILER);
    _transmitPacket();
  }
}

void XPLDirect::_sendPacketString(int command, char *str) // for a string
{
  sprintf(_sendBuffer, "%c%c%s%c", XPLDIRECT_PACKETHEADER, command, str, XPLDIRECT_PACKETTRAILER);
  _transmitPacket();
}

void XPLDirect::_transmitPacket(void)
{
  streamPtr->write(_sendBuffer);
  if (strlen(_sendBuffer) == 64)
  {
    streamPtr->print(" "); // apparently a bug on some boards when we transmit exactly 64 bytes
  }
}

int XPLDirect::_getHandleFromFrame() // Assuming receive buffer is holding a good frame
{
  char holdChar;
  int handleRet;
  holdChar = _receiveBuffer[5];
  _receiveBuffer[5] = 0;
  handleRet = atoi((char *)&_receiveBuffer[2]);
  _receiveBuffer[5] = holdChar;
  return handleRet;
}

int XPLDirect::_getPayloadFromFrame(long int *value) // Assuming receive buffer is holding a good
frame
{
  char holdChar;
  holdChar = _receiveBuffer[15];
  _receiveBuffer[15] = 0;
  *value = atol((char *)&_receiveBuffer[5]);
  _receiveBuffer[15] = holdChar;
  return 0;
}

int XPLDirect::_getPayloadFromFrame(float *value) // Assuming receive buffer is holding a good frame
{
  char holdChar;
  holdChar = _receiveBuffer[15];
  _receiveBuffer[15] = 0;
  *value = atof((char *)&_receiveBuffer[5]);
  _receiveBuffer[15] = holdChar;
  return 0;
}

int XPLDirect::_getPayloadFromFrame(char *value) // Assuming receive buffer is holding a good frame
{
  memcpy(value, (char *)&_receiveBuffer[5], _receiveBufferBytesReceived - 6);
  value[_receiveBufferBytesReceived - 6] = 0; // erase the packet trailer
  for (int i = 0; i < _receiveBufferBytesReceived - 6; i++)
  {
    if (value[i] == 7)
    {
      value[i] = XPLDIRECT_PACKETTRAILER; //  How I deal with the possibility of the packet trailer
being within a string
    }
  }
  return 0;
}

int XPLDirect::allDataRefsRegistered()
{
  return _allDataRefsRegistered;
}

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

```
{
  if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
  {
    return -1; // Error
  }
  _dataRefs[_dataRefsCount] = new _dataRefStructure;
  _dataRefs[_dataRefsCount]->dataRefName = datarefName; // added for F() macro
  _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
  _dataRefs[_dataRefsCount]->divider = divider;
  _dataRefs[_dataRefsCount]->updateRate = rate;
  _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_INT;
  _dataRefs[_dataRefsCount]->latestValue = (void *)value;
  _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
  _dataRefs[_dataRefsCount]->arrayIndex = 0;     // not used unless we are referencing an array
  _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
  _dataRefsCount++;
  _allDataRefsRegistered = 0;
  return (_dataRefsCount - 1);
}

int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int *value, int index)
{
  if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
  {
    return -1;
  }
  _dataRefs[_dataRefsCount] = new _dataRefStructure;
  _dataRefs[_dataRefsCount]->dataRefName = datarefName;
  _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
  _dataRefs[_dataRefsCount]->updateRate = rate;
  _dataRefs[_dataRefsCount]->divider = divider;
  _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_INT; // arrays are dealt with on the XPlane plugin side
  _dataRefs[_dataRefsCount]->latestValue = (void *)value;
  _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
  _dataRefs[_dataRefsCount]->arrayIndex = index; // not used unless we are referencing an array
  _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
  _dataRefsCount++;
  _allDataRefsRegistered = 0;
  return (_dataRefsCount - 1);
}

int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float *value)
{
  if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
  {
    return -1;
  }
  _dataRefs[_dataRefsCount] = new _dataRefStructure;
  _dataRefs[_dataRefsCount]->dataRefName = datarefName;
  _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
  _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_FLOAT;
  _dataRefs[_dataRefsCount]->latestValue = (void *)value;
  _dataRefs[_dataRefsCount]->lastSentFloatValue = -1; // force update on first loop
  _dataRefs[_dataRefsCount]->updateRate = rate;
  _dataRefs[_dataRefsCount]->divider = divider;
  _dataRefs[_dataRefsCount]->arrayIndex = 0;     // not used unless we are referencing an array
  _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
  _dataRefsCount++;
  _allDataRefsRegistered = 0;
  return (_dataRefsCount - 1);
}

int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float *value, int index)
{
  if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
  {
    return -1;
  }
  _dataRefs[_dataRefsCount] = new _dataRefStructure;
  _dataRefs[_dataRefsCount]->dataRefName = datarefName;
  _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
  _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_FLOAT; // arrays are dealt with on the Xplane plugin side
  _dataRefs[_dataRefsCount]->latestValue = (void *)value;
  _dataRefs[_dataRefsCount]->lastSentFloatValue = 0;
  _dataRefs[_dataRefsCount]->updateRate = rate;
  _dataRefs[_dataRefsCount]->arrayIndex = index; // not used unless we are referencing an array
  _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
  _dataRefsCount++;
  _allDataRefsRegistered = 0;
  return (_dataRefsCount - 1);
}

```

```
int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, char *value)
{
  if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
  {
    return -1;
  }
  _dataRefs[_dataRefsCount] = new _dataRefStructure;
  _dataRefs[_dataRefsCount]->dataRefName = datarefName;
  _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
  _dataRefs[_dataRefsCount]->updateRate = rate;
  _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_STRING;
  _dataRefs[_dataRefsCount]->latestValue = (void *)value;
  _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
  _dataRefs[_dataRefsCount]->arrayIndex = 0;      // not used unless we are referencing an array
  _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
  _dataRefsCount++;
  _allDataRefsRegistered = 0;
  return (_dataRefsCount - 1);
}

int XPLDirect::registerCommand(XPString_t *commandName) // user will trigger commands with
commandTrigger
{
  if (_commandsCount >= XPLDIRECT_MAXCOMMANDS_ARDUINO)
  {
    return -1;
  }
  _commands[_commandsCount] = new _commandStructure;
  _commands[_commandsCount]->commandName = commandName;
  _commands[_commandsCount]->commandHandle = -1; // invalid until assigned by xplane
  _commandsCount++;
  _allDataRefsRegistered = 0; // share this flag with the datarefs, true when everything is registered
with xplane.
  return (_commandsCount - 1);
}

XPLDirect XP;
```

Index