

XPLDevices

Generated by Doxygen 1.9.6

1 Hierarchical Index	1
1.1 Class Hierarchy	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Class Documentation	7
4.1 AnalogIn Class Reference	7
4.1.1 Detailed Description	7
4.1.2 Constructor & Destructor Documentation	7
4.1.2.1 AnalogIn() [1/2]	7
4.1.2.2 AnalogIn() [2/2]	8
4.1.3 Member Function Documentation	8
4.1.3.1 calibrate()	8
4.1.3.2 handle()	8
4.1.3.3 raw()	9
4.1.3.4 value()	9
4.2 Button Class Reference	9
4.2.1 Detailed Description	10
4.2.2 Member Enumeration Documentation	11
4.2.2.1 anonymous enum	11
4.2.3 Constructor & Destructor Documentation	11
4.2.3.1 Button() [1/2]	11
4.2.3.2 Button() [2/2]	11
4.2.4 Member Function Documentation	12
4.2.4.1 engaged()	12
4.2.4.2 getCommand()	12
4.2.4.3 handle() [1/2]	12
4.2.4.4 handle() [2/2]	12
4.2.4.5 handleXP() [1/2]	13
4.2.4.6 handleXP() [2/2]	13
4.2.4.7 pressed()	13
4.2.4.8 processCommand()	14
4.2.4.9 released()	14
4.2.4.10 setCommand()	14
4.2.5 Member Data Documentation	14
4.2.5.1 _cmdPush	14
4.2.5.2 _mux	15
4.2.5.3 _pin	15
4.2.5.4 _state	15

4.2.5.5 _transition	15
4.3 DigitalIn_ Class Reference	15
4.3.1 Detailed Description	16
4.3.2 Constructor & Destructor Documentation	16
4.3.2.1 DigitalIn_()	16
4.3.3 Member Function Documentation	16
4.3.3.1 addMux()	16
4.3.3.2 getBit()	17
4.3.3.3 handle()	17
4.3.3.4 setMux()	17
4.4 Encoder Class Reference	18
4.4.1 Detailed Description	19
4.4.2 Constructor & Destructor Documentation	19
4.4.2.1 Encoder() [1/2]	19
4.4.2.2 Encoder() [2/2]	19
4.4.3 Member Function Documentation	20
4.4.3.1 down()	20
4.4.3.2 engaged()	20
4.4.3.3 getCommand()	20
4.4.3.4 handle()	21
4.4.3.5 handleXP()	21
4.4.3.6 pos()	21
4.4.3.7 pressed()	21
4.4.3.8 processCommand()	22
4.4.3.9 released()	22
4.4.3.10 setCommand() [1/2]	22
4.4.3.11 setCommand() [2/2]	22
4.4.3.12 up()	23
4.5 LedShift Class Reference	23
4.5.1 Detailed Description	24
4.5.2 Constructor & Destructor Documentation	24
4.5.2.1 LedShift()	24
4.5.3 Member Function Documentation	24
4.5.3.1 handle()	24
4.5.3.2 set()	24
4.5.3.3 set_all()	25
4.6 RepeatButton Class Reference	25
4.6.1 Detailed Description	27
4.6.2 Constructor & Destructor Documentation	27
4.6.2.1 RepeatButton() [1/2]	27
4.6.2.2 RepeatButton() [2/2]	27
4.6.3 Member Function Documentation	28

4.6.3.1 handle() [1/2]	28
4.6.3.2 handle() [2/2]	28
4.6.3.3 handleXP() [1/2]	28
4.6.3.4 handleXP() [2/2]	28
4.6.4 Member Data Documentation	29
4.6.4.1 _delay	29
4.6.4.2 _timer	29
4.7 Switch Class Reference	29
4.7.1 Detailed Description	30
4.7.2 Constructor & Destructor Documentation	30
4.7.2.1 Switch() [1/2]	30
4.7.2.2 Switch() [2/2]	31
4.7.3 Member Function Documentation	31
4.7.3.1 getCommand()	31
4.7.3.2 handle()	31
4.7.3.3 handleXP()	31
4.7.3.4 off()	32
4.7.3.5 on()	32
4.7.3.6 processCommand()	32
4.7.3.7 setCommand()	32
4.7.3.8 value()	33
4.8 Switch2 Class Reference	33
4.8.1 Detailed Description	34
4.8.2 Constructor & Destructor Documentation	34
4.8.2.1 Switch2() [1/2]	34
4.8.2.2 Switch2() [2/2]	34
4.8.3 Member Function Documentation	35
4.8.3.1 getCommand()	35
4.8.3.2 handle()	35
4.8.3.3 handleXP()	35
4.8.3.4 off()	36
4.8.3.5 on1()	36
4.8.3.6 on2()	36
4.8.3.7 processCommand()	36
4.8.3.8 setCommand() [1/2]	36
4.8.3.9 setCommand() [2/2]	37
4.8.3.10 value()	37
4.9 Timer Class Reference	38
4.9.1 Detailed Description	38
4.9.2 Constructor & Destructor Documentation	38
4.9.2.1 Timer()	38
4.9.3 Member Function Documentation	39

4.9.3.1 count()	39
4.9.3.2 elapsed()	39
4.9.3.3 getTime()	39
4.9.3.4 setCycle()	39
4.10 XPLDirect Class Reference	40
4.10.1 Detailed Description	40
4.10.2 Constructor & Destructor Documentation	40
4.10.2.1 XPLDirect()	41
4.10.3 Member Function Documentation	41
4.10.3.1 allDataRefsRegistered()	41
4.10.3.2 begin()	41
4.10.3.3 commandEnd()	41
4.10.3.4 commandStart()	41
4.10.3.5 commandTrigger() [1/2]	42
4.10.3.6 commandTrigger() [2/2]	42
4.10.3.7 connectionStatus()	42
4.10.3.8 datarefsUpdated()	42
4.10.3.9 hasUpdated()	42
4.10.3.10 registerCommand()	42
4.10.3.11 registerDataRef() [1/5]	43
4.10.3.12 registerDataRef() [2/5]	43
4.10.3.13 registerDataRef() [3/5]	43
4.10.3.14 registerDataRef() [4/5]	43
4.10.3.15 registerDataRef() [5/5]	44
4.10.3.16 sendDebugMessage()	44
4.10.3.17 sendResetRequest()	44
4.10.3.18 sendSpeakMessage()	44
4.10.3.19 xloop()	44
5 File Documentation	45
5.1 AnalogIn.h	45
5.2 Button.h	45
5.3 DigitalIn.h	46
5.4 Encoder.h	47
5.5 LedShift.h	48
5.6 Switch.h	48
5.7 Timer.h	49
5.8 XPLDevices.h	50
5.9 XPLDirect.h	50
5.10 AnalogIn.cpp	52
5.11 Button.cpp	53
5.12 DigitalIn.cpp	54

5.13 Encoder.cpp	55
5.14 LedShift.cpp	57
5.15 Switch.cpp	58
5.16 Timer.cpp	60
5.17 XPLDirect.cpp	60
Index	69

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

AnalogIn	7
Button	9
RepeatButton	25
DigitalIn_	15
Encoder	18
LedShift	23
Switch	29
Switch2	33
Timer	38
XPLDirect	40

Chapter 2

Class Index

2.1 Class List

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

AnalogIn	Class to encapsulate analog inputs	7
Button	Class for a simple pushbutton with debouncing and XPLDirect command handling. Supports start and end of commands so XPlane can show the current Button status	9
DigitalIn_	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	15
Encoder	Class for rotary encoders with optional push functionality. The number of counts per mechanical notch can be configured for the triggering of up/down events	18
LedShift	Class to encapsulate a DM13A LED driver IC	23
RepeatButton	Class for a simple pushbutton with debouncing and XPLDirect command handling, supports start and end of commands so XPlane can show the current Button status. When button is held down cyclic new pressed events are generated for auto repeat function	25
Switch	Class for a simple on/off switch with debouncing and XPLDirect command handling	29
Switch2	Class for an on/off/on switch with debouncing and XPLDirect command handling	33
Timer	Provide a simple software driven timer for general purpose use	38
XPLDirect	40

Chapter 3

File Index

3.1 File List

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

AnalogIn.h	45
Button.h	45
DigitalIn.h	46
Encoder.h	47
LedShift.h	48
Switch.h	48
Timer.h	49
XPLDevices.h	50
XPLDirect.h	50
AnalogIn.cpp	52
Button.cpp	53
DigitalIn.cpp	54
Encoder.cpp	55
LedShift.cpp	57
Switch.cpp	58
Timer.cpp	60
XPLDirect.cpp	60

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

<i>pin</i>	Arduino pin number to use
<i>type</i>	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

<i>pin</i>	Arduino pin number to use
<i>type</i>	unipolar (0..1) or bipolar (-1..1)
<i>timeConst</i>	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 bipolar 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

<i>mux</i>	mux number (from DigitalIn initialization order)
<i>muxpin</i>	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

<i>pin</i>	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

<i>input</i>	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

<i>input</i>	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

<i>cmdPush</i>	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

<i>pin</i>	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

<i>mux</i>	mux to read from. Use NOT_USED to access arduino digital input without mux
<i>muxpin</i>	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 address pins for 74HC4067 multiplexers. All mux share the same address pins.

Parameters

<i>s0</i>	Adress pin s0
<i>s1</i>	Adress pin s1
<i>s2</i>	Adress pin s2
<i>s3</i>	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 associated 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

<i>mux</i>	mux number (from DigitalIn initialization order)
<i>pin1</i>	pin for Encoder A track
<i>pin2</i>	pin for Encoder B track
<i>pin3</i>	pin for encoder push function (NOT_USED if not connected)
<i>pulses</i>	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

<i>pin1</i>	pin for Encoder A track
<i>pin2</i>	pin for Encoder B track
<i>pin3</i>	pin for encoder push function (NOT_USED if not connected)
<i>pulses</i>	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 associated with the selected event.

Parameters

<i>cmd</i>	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

<i>cmdUp</i>	Command handle for positive turn as returned by XP.registerCommand()
<i>cmdDown</i>	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

<i>cmdUp</i>	Command handle for positive turn as returned by XP.registerCommand()
<i>cmdDown</i>	Command handle for negative turn as returned by XP.registerCommand()
<i>cmdPush</i>	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

<i>pin_DAI</i>	DAI pin of DM13A
<i>pin_DCK</i>	DCL pin of DM13A
<i>pin_LAT</i>	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

<i>pin</i>	DM13A pin of the LED (0-15)
<i>mode</i>	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

<i>mode</i>	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

<i>mux</i>	mux number (from initialization order)
<i>muxpin</i>	pin on the mux (0-15)
<i>delay</i>	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

<i>pin</i>	Arduino pin number
<i>delay</i>	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

<i>input</i>	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

<i>input</i>	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

<i>mux</i>	mux number (from DigitalIn initialization order)
<i>multipin</i>	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

<i>pin</i>	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

<i>cmdOn</i>	Command handle for Switch moved to on as returned by XP.registerCommand()
<i>cmdOff</i>	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

<i>onValue</i>	Value to return when Switch is set to on
<i>offValue</i>	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

<i>mux</i>	mux number (from DigitalIn initialization order)
<i>pin1</i>	on1 pin on the mux (0-15)
<i>pin1</i>	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

<i>pin1</i>	on1 Arduino pin number
<i>pin2</i>	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

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

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

<i>on1Value</i>	Value to return when Switch is set to on1
<i>offValue</i>	Value to return when Switch is set to off
<i>on2Value</i>	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

Provide 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

Provide 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

<i>cycle</i>	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

<i>cycle</i>	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

```
00001 #ifndef AnalogIn_h
00002 #define AnalogIn_h
00003
00004 #define AD_RES 10
00005
00006 enum Analog_t
00007 {
00008     unipolar,
00009     bipolar
00010 };
00011
00013 class AnalogIn
00014 {
00015 public:
00019     AnalogIn(uint8_t pin, Analog_t type);
00020
00025     AnalogIn(uint8_t pin, Analog_t type, float timeConst);
00026
00028     void handle();
00029
00032     float value() { return _value; };
00033
00036     int raw();
00037
00039     void calibrate();
00040
00041 private:
00042     float _filterConst;
00043     float _value;
00044     float _scale;
00045     float _scalePos;
00046     float _scaleNeg;
00047     int _offset;
00048     uint8_t _pin;
00049 };
00050
00051 #endif
```

5.2 Button.h

```
00001 #ifndef Button_h
00002 #define Button_h
00003 #include <DigitalIn.h>
00004
00007 class Button
00008 {
00009 private:
00010     void _handle(bool input);
00011
00012 public:
00016     Button(uint8_t mux, uint8_t muxpin);
00017
00020     Button(uint8_t pin) : Button(NOT_USED, pin){};
00021
```

```

00023 void handle() { _handle(true); };
00024
00027 void handle(bool input) { _handle(input); };
00028
00030 void handleXP() { _handle(true); processCommand(); };
00031
00034 void handleXP(bool input) { _handle(input); processCommand(); };
00035
00038 bool pressed() { return _transition == transPressed ? (_transition = transNone,
true) : false; };
00039
00042 bool released() { return _transition == transReleased ? (_transition = transNone,
true) : false; };
00043
00046 bool engaged() { return _state > 0; };
00047
00050 void setCommand(int cmdPush) { _cmdPush = cmdPush; };
00051
00054 int getCommand() { return _cmdPush; };
00055
00057 void processCommand();
00058
00059 protected:
00060 enum
00061 {
00062     transNone,
00063     transPressed,
00064     transReleased
00065 };
00066 uint8_t _mux;
00067 uint8_t _pin;
00068 uint8_t _state;
00069 uint8_t _transition;
00070 int _cmdPush;
00071 };
00072
00076 class RepeatButton : public Button
00077 {
00078 private:
00079     void _handle(bool input);
00080
00081 public:
00086     RepeatButton(uint8_t mux, uint8_t muxpin, uint32_t delay);
00087
00091     RepeatButton(uint8_t pin, uint32_t delay) : RepeatButton(NOT_USED, pin, delay){};
00092
00094 void handle() { _handle(true); };
00095
00098 void handle(bool input) { _handle(input); };
00099
00101 void handleXP() { _handle(true); processCommand(); };
00102
00105 void handleXP(bool input) { _handle(input); processCommand(); };
00106
00107 protected:
00108     uint32_t _delay;
00109     uint32_t _timer;
00110 };
00111
00112 #endif

```

5.3 DigitalIn.h

```

00001 #ifndef Mux_h
00002 #define Mux_h
00003
00005 #ifndef MUX_MAX_NUMBER
00006 #define MUX_MAX_NUMBER 6
00007 #endif
00008
00010 #ifndef MCP_MAX_NUMBER
00011 #define MCP_MAX_NUMBER 0
00012 #endif
00013
00014 // Include i2c lib only when needed
00015 #if MCP_MAX_NUMBER > 0
00016 #include <Adafruit_MCP23X17.h>
00017 #endif
00018
00019 #define NOT_USED 255
00020
00023 class DigitalIn_
00024 {

```

```

00025 public:
00027     DigitalIn();
00028
00034     void setMux(uint8_t s0, uint8_t s1, uint8_t s2, uint8_t s3);
00035
00039     bool addMux(uint8_t pin);
00040
00041     #if MCP_MAX_NUMBER > 0
00045     bool addMCP(uint8_t address);
00046 #endif
00047
00052     bool getBit(uint8_t mux, uint8_t muxpin);
00053
00055     void handle();
00056 private:
00057     uint8_t _s0, _s1, _s2, _s3;
00058     uint8_t _numPins;
00059     uint8_t _pin[MUX_MAX_NUMBER + MCP_MAX_NUMBER];
00060     int16_t _data[MUX_MAX_NUMBER + MCP_MAX_NUMBER];
00061     #if MCP_MAX_NUMBER > 0
00062     uint8_t _numMCP;
00063     Adafruit_MCP23X17 _mcp[MCP_MAX_NUMBER];
00064 #endif
00065 };
00066
00068 extern DigitalIn_ DigitalIn;
00069
00070 #endif

```

5.4 Encoder.h

```

00001 #ifndef Encoder_h
00002 #define Encoder_h
00003 #include <DigitalIn.h>
00004
00005 enum EncCmd_t
00006 {
00007     encCmdUp,
00008     encCmdDown,
00009     encCmdPush
00010 };
00011
00012 enum EncPulse_t
00013 {
00014     enc1Pulse = 1,
00015     enc2Pulse = 2,
00016     enc4Pulse = 4
00017 };
00018
00021 class Encoder
00022 {
00023 public:
00030     Encoder(uint8_t mux, uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses);
00031
00037     Encoder(uint8_t pin1, uint8_t pin2, uint8_t pin3, EncPulse_t pulses) : Encoder(NOT_USED, pin1, pin2,
pin3, pulses) {}
00038
00040     void handle();
00041
00043     void handleXP() { handle(); processCommand(); };
00044
00047     int16_t pos() { return _count; };
00048
00051     bool up() { return _count >= _pulses ? (_count -= _pulses, true) : false; };
00052
00055     bool down() { return _count <= -_pulses ? (_count += _pulses, true) : false; };
00056
00059     bool pressed() { return _transition == transPressed ? (_transition = transNone, true) : false;
};
00060
00063     bool released() { return _transition == transReleased ? (_transition = transNone, true) : false;
};
00064
00067     bool engaged() { return _state > 0; };
00068
00073     void setCommand(int cmdUp, int cmdDown, int cmdPush);
00074
00078     void setCommand(int cmdUp, int cmdDown) { setCommand(cmdUp, cmdDown, -1); };
00079
00083     int getCommand(EncCmd_t cmd);
00084
00086     void processCommand();
00087 private:

```

```

00088     enum
00089     {
00090         transNone,
00091         transPressed,
00092         transReleased
00093     };
00094     uint8_t _mux;
00095     uint8_t _pin1, _pin2, _pin3;
00096     int8_t _count;
00097     uint8_t _pulses;
00098     uint8_t _state;
00099     uint8_t _debounce;
00100     uint8_t _transition;
00101     int _cmdUp;
00102     int _cmdDown;
00103     int _cmdPush;
00104 };
00105
00106 #endif

```

5.5 LedShift.h

```

00001 #ifndef Led_h
00002 #define Led_h
00003
00005 enum led_t
00006 {
00008     ledOff,
00010     ledSlow,
00012     ledMedium,
00014     ledFast,
00016     ledOn
00017 };
00018
00020 class LedShift
00021 {
00022 public:
00027     LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT);
00028
00032     void set(uint8_t pin, led_t mode);
00033
00036     void set_all(led_t mode);
00037
00039     void handle();
00040
00041 private:
00042     void _send();
00043     void _update(uint8_t pin);
00044     uint8_t _pin_DAI;
00045     uint8_t _pin_DCK;
00046     uint8_t _pin_LAT;
00047     uint16_t _state;
00048     led_t _mode[16];
00049     uint8_t _count;
00050     unsigned long _timer;
00051 };
00052
00053 #endif

```

5.6 Switch.h

```

00001 #ifndef Switch_h
00002 #define Switch_h
00003 #include <DigitalIn.h>
00004
00006 class Switch
00007 {
00008 public:
00012     Switch(uint8_t mux, uint8_t pin);
00013
00016     Switch(uint8_t pin) : Switch (NOT_USED, pin) {};
00017
00019     void handle();
00020
00022     void handleXP() { handle(); processCommand(); };
00023
00026     bool on()        { return _state == switchOn; };
00027
00030     bool off()       { return _state == switchOff; };

```



```

00031
00035 void setCommand(int cmdOn, int cmdOff);
00036
00039 int getCommand();
00040
00042 void processCommand();
00043
00048 float value(float onValue, float offValue) { return on() ? onValue : offValue; };
00049
00050 private:
00051     enum SwState_t
00052     {
00053         switchOff,
00054         switchOn
00055     };
00056     uint8_t _mux;
00057     uint8_t _pin;
00058     uint8_t _debounce;
00059     uint8_t _state;
00060     bool _transition;
00061     int _cmdOff;
00062     int _cmdOn;
00063 };
00064
00066 class Switch2
00067 {
00068 public:
00073     Switch2(uint8_t mux, uint8_t pin1, uint8_t pin2);
00074
00078     Switch2(uint8_t pin1, uint8_t pin2) : Switch2(NOT_USED, pin1, pin2) {}
00079
00081     void handle();
00082
00084     void handleXP() { handle(); processCommand(); };
00085
00088     bool off()      { return _state == switchOff; };
00089
00092     bool on1()      { return _state == switchOn1; };
00093
00096     bool on2()      { return _state == switchOn2; };
00097
00101     void setCommand(int cmdUp, int cmdDown);
00102
00107     void setCommand(int cmdOn1, int cmdOff, int cmdOn2);
00108
00111     int getCommand();
00112
00114     void processCommand();
00115
00121     float value(float on1Value, float offValue, float on2value) { return (on1() ? on1Value : on2() ?
on2value : offValue); };
00122
00123 private:
00124     enum SwState_t
00125     {
00126         switchOff,
00127         switchOn1,
00128         switchOn2
00129     };
00130     uint8_t _mux;
00131     uint8_t _pin1;
00132     uint8_t _pin2;
00133     uint8_t _lastState;
00134     uint8_t _debounce;
00135     uint8_t _state;
00136     bool _transition;
00137     int _cmdOff;
00138     int _cmdOn1;
00139     int _cmdOn2;
00140 };
00141
00142 #endif

```

5.7 Timer.h

```

00001 #ifndef SoftTimer_h
00002 #define SoftTimer_h
00003
00005 class Timer
00006 {
00007 public:
00010     Timer(float cycle = 0); // ms
00011

```

```

00014     void setCycle(float cycle);
00015
00018     bool elapsed();
00019
00022     float getTime(); // ms
00023
00026     long count();
00027 private:
00028     unsigned long _cycleTime;
00029     unsigned long _lastUpdateTime;
00030     long _count;
00031 };
00032
00033 #endif

```

5.8 XPLDevices.h

```

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

```

5.9 XPLDirect.h

```

00001 /*
00002  XPLDirect.h - Library for serial interface to Xplane SDK.
00003  Created by Michael Gerlicher, September 2020.
00004  To report problems, download updates and examples, suggest enhancements or get technical support,
00005  please visit my patreon page:
00006  www.patreon.com/curiosityworkshop
00007  Stripped down to Minimal Version by mrusk, February 2023
00008 */
00009 #ifndef XPLDirect_h
00010 #define XPLDirect_h
00011
00012 #ifndef XPLDIRECT_MAXDATAREFS_ARDUINO
00013 #define XPLDIRECT_MAXDATAREFS_ARDUINO 100 // This can be changed to suit your needs and capabilities
00014 // of your board.
00015 #endif
00016 #ifndef XPLDIRECT_MAXCOMMANDS_ARDUINO
00017 #define XPLDIRECT_MAXCOMMANDS_ARDUINO 100 // Same here.
00018 #endif
00019
00020 #define XPLDIRECT_RX_TIMEOUT 500 // after detecting a frame header, how long will we wait to receive
00021 // the rest of the frame. (default 500)
00022
00023 #ifndef XPLMAX_PACKETSIZE
00024 #define XPLMAX_PACKETSIZE 80 // Probably leave this alone. If you need a few extra bytes of RAM it
00025 // could be reduced, but it needs to
00026 // be as long as the longest dataref name + 10. If you are using
00027 // that transfer strings it needs to be big enough for those too.
00028 #define XPLMAX_PACKETSIZE 200
00029 #endif
00030
00031 #ifndef XPL_USE_PROGMEM
00032 #define XPL_USE_PROGMEM 1
00033 #endif
00034
00035 // STOP! Dont change any other defines in this header!
00036
00037 #ifndef XPL_USE_PROGMEM
00038 // use Flash for strings, requires F() macro for strings in all registration calls
00039 typedef const __FlashStringHelper XPString_t;
00040 #else
00041 typedef const char XPString_t;
00042 #endif

```

```

00043 #define XPLDIRECT_BAUDRATE 115200    // don't mess with this, it needs to match the plugin which won't
      change
00044 #define XPLDIRECT_PACKETHEADER '<'    // ...or this
00045 #define XPLDIRECT_PACKETTRAILER '>'    // ...or this
00046 #define XPLDIRECT_VERSION 2106171     // The plugin will start to verify that a compatible version is
      being used
00047 #define XPLDIRECT_ID 0                  // Used for relabled plugins to identify the company. 0 = normal
      distribution version
00048
00049 #define XPLERROR 'E'                    // %s          general error
00050 #define XPLRESPONSE_NAME '0'
00051 #define XPLRESPONSE_DATAREF '3'        // %3.3i%s      dataref handle, dataref name
00052 #define XPLRESPONSE_COMMAND '4'        // %3.3i%s      command handle, command name
00053 #define XPLRESPONSE_VERSION 'V'
00054 #define XPLCMD_PRINTDEBUG '1'
00055 #define XPLCMD_RESET '2'
00056 #define XPLCMD_SPEAK 'S'                // speak string
00057 #define XPLCMD_SENDNAME 'a'
00058 #define XPLREQUEST_REGISTERDATAREF 'b' // %1.1i%2.2i%5.5i RWMMode, array index (0 for non array
      datarefs), divider to decrease resolution, dataref name
00059 #define XPLREQUEST_REGISTERCOMMAND 'm' // just the name of the command to register
00060 #define XPLREQUEST_NOREQUESTS 'c'      // nothing to request
00061 #define XPLREQUEST_REFRESH 'd'         // the plugin will call this once xplane is loaded in order to
      get fresh updates from arduino handles that write
00062 #define XPLCMD_DUMPREGISTRATIONS 'Z'    // for debug purposes only (disabled)
00063 #define XPLCMD_DATAREFUPDATE 'e'
00064 #define XPLCMD_SENDREQUEST 'f'
00065 #define XPLCMD_DEVICEREADY 'g'
00066 #define XPLCMD_DEVICENOTREADY 'h'
00067 #define XPLCMD_COMMANDSTART 'i'
00068 #define XPLCMD_COMMANDEND 'j'
00069 #define XPLCMD_COMMANDTRIGGER 'k'       // %3.3i%3.3i  command handle, number of triggers
00070 #define XPLCMD_SENDVERSION 'v'          // we will respond with current build version
00071 #define XPL_EXITING 'x'                 // MG 03/14/2023: xplane sends this to the arduino device during
      normal shutdown of xplane. It may not happen if xplane crashes.
00072
00073 #define XPL_READ 1
00074 #define XPL_WRITE 2
00075 #define XPL_READWRITE 3
00076
00077 #define XPL_DATATYPE_INT 1
00078 #define XPL_DATATYPE_FLOAT 2
00079 #define XPL_DATATYPE_STRING 3
00080
00081 class XPLDirect
00082 {
00083 public:
00084     XPLDirect();
00085     void begin(const char *devicename); // parameter is name of your device for reference
00086     int connectionStatus(void);
00087     int commandTrigger(int commandHandle); // triggers specified command 1 time;
00088     int commandTrigger(int commandHandle, int triggerCount); // triggers specified command triggerCount
      times.
00089     int commandStart(int commandHandle);
00090     int commandEnd(int commandHandle);
00091     int datarefsUpdated(); // returns true if xplane has updated any datarefs since last call to
      datarefsUpdated()
00092     int hasUpdated(int handle); // returns true if xplane has updated this dataref since last call to
      hasUpdated()
00093     int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int
      *value);
00094     int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, long int
      *value, int index);
00095     int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float
      *value);
00096     int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider, float
      *value, int index);
00097     int registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, char* value);
00098     int registerCommand(XPString_t *commandName);
00099     int sendDebugMessage(const char *msg);
00100     int sendSpeakMessage(const char *msg);
00101     int allDataRefsRegistered(void);
00102     void sendResetRequest(void);
00103     int xloop(void); // where the magic happens!
00104 private:
00105     void _processSerial();
00106     void _processPacket();
00107     void _sendPacketInt(int command, int handle, long int value); // for ints
00108     void _sendPacketFloat(int command, int handle, float value); // for floats
00109     void _sendPacketVoid(int command, int handle); // just a command with a handle
00110     void _sendPacketString(int command, char *str); // for a string
00111     void _transmitPacket();
00112     void _sendname();
00113     void _sendVersion();
00114     int _getHandleFromFrame();
00115     int _getPayloadFromFrame(long int *);
00116     int _getPayloadFromFrame(float *);

```

```

00117 int _getPayloadFromFrame(char *);
00118
00119 Stream *streamPtr;
00120 char *_deviceName;
00121 char _receiveBuffer[XPLMAX_PACKETSIZE];
00122 int _receiveBufferBytesReceived;
00123 char _sendBuffer[XPLMAX_PACKETSIZE];
00124 int _connectionStatus;
00125 int _dataRefsCount;
00126 struct _dataRefStructure
00127 {
00128     int dataRefHandle;
00129     byte dataRefRWType; // XPL_READ, XPL_WRITE, XPL_READWRITE
00130     byte dataRefVARType; // XPL_DATATYPE_INT 1, XPL_DATATYPE_FLOAT 2 XPL_DATATYPE_STRING 3
00131     float divider; // tell the host to reduce resolution by dividing then remultiplying by
    this number to reduce traffic. (ie .02, .1, 1, 5, 10, 100, 1000 etc)
00132     byte forceUpdate; // in case xplane plugin asks for a refresh
00133     unsigned long updateRate; // maximum update rate in milliseconds, 0 = every change
00134     unsigned long lastUpdateTime;
00135     XPString_t *dataRefName;
00136     void *latestValue;
00137     union {
00138         long int lastSentIntValue;
00139         float lastSentFloatValue;
00140     };
00141     byte updatedFlag; // True if xplane has updated this dataref. Gets reset when we call hasUpdated
    method.
00142     byte arrayIndex; // for datarefs that speak in arrays
00143 } *dataRefs[XPLDIRECT_MAXDATAREFS_ARDUINO];
00144 int _commandsCount;
00145 struct _commandStructure
00146 {
00147     int commandHandle;
00148     XPString_t *commandName;
00149 } *commands[XPLDIRECT_MAXCOMMANDS_ARDUINO];
00150 byte _allDataRefsRegistered; // becomes true if all datarefs have been registered
00151 byte _dataRefsUpdatedFlag; // becomes true if any datarefs have been updated from xplane since
    last call to dataRefsUpdated()
00152 };
00153
00155 extern XPLDirect XP;
00156
00157 #endif

```

5.10 AnalogIn.cpp

```

00001 #include <Arduino.h>
00002 #include "AnalogIn.h"
00003
00004 #define FULL_SCALE ((1 << AD_RES) - 1)
00005 #define HALF_SCALE (1 << (AD_RES - 1))
00006
00007 AnalogIn::AnalogIn(uint8_t pin, Analog_t type)
00008 {
00009     _pin = pin;
00010     _filterConst = 1.0;
00011     _scale = 1.0;
00012     pinMode(_pin, INPUT);
00013     if (type == bipolar)
00014     {
00015         _offset = HALF_SCALE;
00016         _scalePos = _scale / HALF_SCALE;
00017         _scaleNeg = _scale / HALF_SCALE;
00018     }
00019     else
00020     {
00021         _offset = 0;
00022         _scalePos = _scale / FULL_SCALE;
00023         _scaleNeg = 0.0;
00024     }
00025 }
00026
00027 AnalogIn::AnalogIn(uint8_t pin, Analog_t type, float timeConst) : AnalogIn(pin, type)
00028 {
00029     _filterConst = 1.0 / timeConst;
00030 }
00031
00032 void AnalogIn::handle()
00033 {
00034     int _raw = raw();
00035     _value = (_filterConst * _raw * (_raw >= 0 ? _scalePos : _scaleNeg)) + (1.0 - _filterConst) *
        _value;
00036 }

```

```

00037
00038 int AnalogIn::raw()
00039 {
00040     return analogRead(_pin) - _offset;
00041 }
00042
00043 void AnalogIn::calibrate()
00044 {
00045     long sum = 0;
00046     for (int i = 0; i < 64; i++)
00047     {
00048         sum += analogRead(_pin);
00049     }
00050     _offset = (int)(sum / 64);
00051     _scalePos = (_offset < FULL_SCALE) ? _scale / (float)(FULL_SCALE - _offset) : 1.0;
00052     _scaleNeg = (_offset > 0) ? _scale / (float)(_offset) : 1.0;
00053 }

```

5.11 Button.cpp

```

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

```

```

00064     _state = DEBOUNCE_DELAY;
00065     _transition = transPressed;
00066     _timer = millis() + _delay;
00067 }
00068 else if (_delay > 0 && (millis() >= _timer))
00069 {
00070     _state = DEBOUNCE_DELAY;
00071     _transition = transPressed;
00072     _timer += _delay;
00073 }
00074 }
00075 else if (_state > 0)
00076 {
00077     if (--_state == 0)
00078     {
00079         _transition = transReleased;
00080     }
00081 }
00082 }

```

5.12 DigitalIn.cpp

```

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

```

```

00062     _numMCP++;
00063     _pin[_numPins++] = MCP_PIN;
00064     return true;
00065 }
00066 #endif
00067
00068 // Gets specific pin from mux, number according to initialization order
00069 bool DigitalIn_::getBit(uint8_t mux, uint8_t pin)
00070 {
00071     if (mux == NOT_USED)
00072     {
00073         return !digitalRead(pin);
00074     }
00075     return bitRead(_data[mux], pin);
00076 }
00077
00078 // read all inputs together -> base for board specific optimization by using byte read
00079 void DigitalIn_::handle()
00080 {
00081     // only if Mux Pins present
00082     #if MCP_MAX_NUMBER > 0
00083         if (_numPins > _numMCP)
00084         #else
00085             if (_numPins > 0)
00086         #endif
00087         {
00088             for (uint8_t muxpin = 0; muxpin < 16; muxpin++)
00089             {
00090                 digitalWrite(_s0, bitRead(muxpin, 0));
00091                 digitalWrite(_s1, bitRead(muxpin, 1));
00092                 digitalWrite(_s2, bitRead(muxpin, 2));
00093                 digitalWrite(_s3, bitRead(muxpin, 3));
00094                 for (uint8_t mux = 0; mux < _numPins; mux++)
00095                 {
00096                     if (_pin[mux] != MCP_PIN)
00097                     {
00098                         bitWrite(_data[mux], muxpin, !digitalRead(_pin[mux]));
00099                     }
00100                 }
00101             }
00102         }
00103         #if MCP_MAX_NUMBER > 0
00104         int mcp = 0;
00105         for (uint8_t mux = 0; mux < _numPins; mux++)
00106         {
00107             if (_pin[mux] == MCP_PIN)
00108             {
00109                 _data[mux] = ~_mcp[mcp++].readGPIOAB();
00110             }
00111         }
00112         #endif
00113     }
00114 }
00115 DigitalIn_ DigitalIn;

```

5.13 Encoder.cpp

```

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

```

```

00027     pinMode(_pin3, INPUT_PULLUP);
00028 }
00029 }
00030
00031 // real time handling
00032 void Encoder::handle()
00033 {
00034     // collect new state
00035     _state = ((_state & 0x03) << 2) | (DigitalIn.getBit(_mux, _pin2) << 1) | (DigitalIn.getBit(_mux,
    _pin1));
00036     // evaluate state change
00037     if (_state == 1 || _state == 7 || _state == 8 || _state == 14)
00038     {
00039         _count++;
00040     }
00041     if (_state == 2 || _state == 4 || _state == 11 || _state == 13)
00042     {
00043         _count--;
00044     }
00045     if (_state == 3 || _state == 12)
00046     {
00047         _count += 2;
00048     }
00049     if (_state == 6 || _state == 9)
00050     {
00051         _count -= 2;
00052     }
00053
00054     // optional button functionality
00055     if (_pin3 != NOT_USED)
00056     {
00057         if (DigitalIn.getBit(_mux, _pin3))
00058         {
00059             if (_debounce == 0)
00060             {
00061                 _debounce = DEBOUNCE_DELAY;
00062                 _transition = transPressed;
00063             }
00064         }
00065         else if (_debounce > 0)
00066         {
00067             if (--_debounce == 0)
00068             {
00069                 _transition = transReleased;
00070             }
00071         }
00072     }
00073 }
00074
00075 void Encoder::setCommand(int cmdUp, int cmdDown, int cmdPush)
00076 {
00077     _cmdUp = cmdUp;
00078     _cmdDown = cmdDown;
00079     _cmdPush = cmdPush;
00080 }
00081
00082 int Encoder::getCommand(EncCmd_t cmd)
00083 {
00084     switch (cmd)
00085     {
00086     case encCmdUp:
00087         return _cmdUp;
00088         break;
00089     case encCmdDown:
00090         return _cmdDown;
00091         break;
00092     case encCmdPush:
00093         return _cmdPush;
00094         break;
00095     default:
00096         return -1;
00097         break;
00098     }
00099 }
00100
00101 void Encoder::processCommand()
00102 {
00103     if (up())
00104     {
00105         XP.commandTrigger(_cmdUp);
00106     }
00107     if (down())
00108     {
00109         XP.commandTrigger(_cmdDown);
00110     }
00111     if (_cmdPush >= 0)
00112     {

```



```

00113     if (pressed())
00114     {
00115         XP.commandStart(_cmdPush);
00116     }
00117     if (released())
00118     {
00119         XP.commandEnd(_cmdPush);
00120     }
00121 }
00122 }

```

5.14 LedShift.cpp

```

00001 #include <Arduino.h>
00002 #include "LedShift.h"
00003
00004 #define BLINK_DELAY 150
00005
00006 LedShift::LedShift(uint8_t pin_DAI, uint8_t pin_DCK, uint8_t pin_LAT)
00007 {
00008     _pin_DAI = pin_DAI;
00009     _pin_DCK = pin_DCK;
00010     _pin_LAT = pin_LAT;
00011     _count = 0;
00012     _state = 0;
00013     _timer = millis() + BLINK_DELAY;
00014     for (int pin = 0; pin < 16; pin++)
00015     {
00016         _mode[pin] = ledOff;
00017     }
00018     pinMode(_pin_DAI, OUTPUT);
00019     pinMode(_pin_DCK, OUTPUT);
00020     pinMode(_pin_LAT, OUTPUT);
00021     digitalWrite(_pin_DAI, LOW);
00022     digitalWrite(_pin_DCK, LOW);
00023     digitalWrite(_pin_LAT, LOW);
00024     _send();
00025 }
00026
00027 void LedShift::_send()
00028 {
00029     shiftOut(_pin_DAI, _pin_DCK, MSBFIRST, (_state & 0xFF00) >> 8);
00030     shiftOut(_pin_DAI, _pin_DCK, MSBFIRST, (_state & 0x00FF));
00031     digitalWrite(_pin_LAT, HIGH);
00032     digitalWrite(_pin_LAT, LOW);
00033 }
00034
00035 void LedShift::_update(uint8_t pin)
00036 {
00037     switch (_mode[pin])
00038     {
00039     case ledOn:
00040         bitSet(_state, pin);
00041         break;
00042     case ledFast:
00043         bitWrite(_state, pin, bitRead(_count, 0));
00044         break;
00045     case ledMedium:
00046         bitWrite(_state, pin, bitRead(_count, 1));
00047         break;
00048     case ledSlow:
00049         bitWrite(_state, pin, bitRead(_count, 2));
00050         break;
00051     default:
00052         bitClear(_state, pin);
00053     }
00054 }
00055
00056 void LedShift::set(uint8_t pin, led_t mode)
00057 {
00058     _mode[pin] = mode;
00059     _update(pin);
00060 }
00061
00062 void LedShift::set_all(led_t mode)
00063 {
00064     for (int pin = 0; pin < 16; pin++)
00065     {
00066         _mode[pin] = mode;
00067         _update(pin);
00068     }
00069 }
00070

```

```

00071 void LedShift::handle()
00072 {
00073     if (millis() >= _timer)
00074     {
00075         _timer += BLINK_DELAY;
00076         _count = (_count + 1) % 8;
00077         for (int pin = 0; pin < 16; pin++)
00078         {
00079             _update(pin);
00080         }
00081     }
00082     _send();
00083 }

```

5.15 Switch.cpp

```

00001 #include <Arduino.h>
00002 #include <XPLDirect.h>
00003 #include "Switch.h"
00004
00005 #ifndef DEBOUNCE_DELAY
00006 #define DEBOUNCE_DELAY 20
00007 #endif
00008
00009 Switch::Switch(uint8_t mux, uint8_t pin)
00010 {
00011     _mux = mux;
00012     _pin = pin;
00013     _state = switchOff;
00014     _cmdOff = -1;
00015     _cmdOn = -1;
00016     pinMode(_pin, INPUT_PULLUP);
00017 }
00018
00019 void Switch::handle()
00020 {
00021     if (_debounce > 0)
00022     {
00023         _debounce--;
00024     }
00025     else
00026     {
00027         SwState_t input = switchOff;
00028         if (DigitalIn.getBit(_mux, _pin))
00029         {
00030             input = switchOn;
00031         }
00032         if (input != _state)
00033         {
00034             _debounce = DEBOUNCE_DELAY;
00035             _state = input;
00036             _transition = true;
00037         }
00038     }
00039 }
00040
00041 void Switch::setCommand(int cmdOn, int cmdOff)
00042 {
00043     _cmdOn = cmdOn;
00044     _cmdOff = cmdOff;
00045 }
00046
00047 int Switch::getCommand()
00048 {
00049     switch (_state)
00050     {
00051     case switchOff:
00052         return _cmdOff;
00053         break;
00054     case switchOn:
00055         return _cmdOn;
00056         break;
00057     default:
00058         return -1;
00059         break;
00060     }
00061 }
00062
00063 void Switch::processCommand()
00064 {
00065     if (_transition)
00066     {
00067         XP.commandTrigger(getCommand());

```

```

00068     _transition = false;
00069 }
00070 }
00071
00072 // Switch 2
00073
00074 Switch2::Switch2(uint8_t mux, uint8_t pin1, uint8_t pin2)
00075 {
00076     _mux = mux;
00077     _pin1 = pin1;
00078     _pin2 = pin2;
00079     _state = switchOff;
00080     _cmdOff = -1;
00081     _cmdOn1 = -1;
00082     if (_mux == NOT_USED)
00083     {
00084         pinMode(_pin1, INPUT_PULLUP);
00085         pinMode(_pin2, INPUT_PULLUP);
00086     }
00087 }
00088
00089 void Switch2::handle()
00090 {
00091     if (_debounce > 0)
00092     {
00093         _debounce--;
00094     }
00095     else
00096     {
00097         SwState_t input = switchOff;
00098         if (DigitalIn.getBit(_mux, _pin1))
00099         {
00100             input = switchOn1;
00101         }
00102         else if (DigitalIn.getBit(_mux, _pin2))
00103         {
00104             input = switchOn2;
00105         }
00106         if (input != _state)
00107         {
00108             _debounce = DEBOUNCE_DELAY;
00109             _lastState = _state;
00110             _state = input;
00111             _transition = true;
00112         }
00113     }
00114 }
00115
00116 void Switch2::setCommand(int cmdUp, int cmdDown)
00117 {
00118     _cmdOn1 = cmdUp;
00119     _cmdOff = cmdDown;
00120     _cmdOn2 = -1;
00121 }
00122
00123 void Switch2::setCommand(int cmdOn1, int cmdOff, int cmdOn2)
00124 {
00125     _cmdOn1 = cmdOn1;
00126     _cmdOff = cmdOff;
00127     _cmdOn2 = cmdOn2;
00128 }
00129
00130 int Switch2::getCommand()
00131 {
00132     if (_cmdOn2 == -1)
00133     {
00134         if (_state == switchOn1)
00135         {
00136             return _cmdOn1;
00137         }
00138         if (_state == switchOff && _lastState == switchOn1)
00139         {
00140             return _cmdOff;
00141         }
00142         if (_state == switchOn2)
00143         {
00144             return _cmdOff;
00145         }
00146         if (_state == switchOff && _lastState == switchOn2)
00147         {
00148             return _cmdOn1;
00149         }
00150     }
00151     else
00152     {
00153         if (_state == switchOn1)
00154         {

```

```

00155         return _cmdOn1;
00156     }
00157     if (_state == switchOff)
00158     {
00159         return _cmdOff;
00160     }
00161     if (_state == switchOn2)
00162     {
00163         return _cmdOn2;
00164     }
00165     }
00166     return -1;
00167 }
00168
00169 void Switch2::processCommand()
00170 {
00171     if (_transition)
00172     {
00173         XP.commandTrigger(getCommand());
00174         _transition = false;
00175     }
00176 }

```

5.16 Timer.cpp

```

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

```

5.17 XPLDirect.cpp

```

00001 /*
00002  XPLDirect.cpp
00003  Created by Michael Gerlicher, September 2020.
00004  Modified by mrusk, March 2023
00005 */
00006
00007 #include <arduino.h>
00008 #include "XPLDirect.h"
00009
00010 // Methods
00011 XPLDirect::XPLDirect()

```

```

00012 {
00013     Serial.begin(XPLDIRECT_BAUDRATE);
00014     streamPtr = &Serial;
00015     streamPtr->setTimeout(XPLDIRECT_RX_TIMEOUT);
00016 }
00017
00018 void XPLDirect::begin(const char *devicename)
00019 {
00020     _deviceName = (char *)devicename;
00021     _connectionStatus = 0;
00022     _dataRefsCount = 0;
00023     _commandsCount = 0;
00024     _allDataRefsRegistered = 0;
00025     _receiveBuffer[0] = 0;
00026 }
00027
00028 int XPLDirect::xloop(void)
00029 {
00030     _processSerial();
00031     if (!_allDataRefsRegistered)
00032     {
00033         return _connectionStatus;
00034     }
00035     // process datarefs to send
00036     for (int i = 0; i < _dataRefsCount; i++)
00037     {
00038         if (_dataRefs[i]->dataRefHandle >= 0 && (_dataRefs[i]->dataRefRWType == XPL_WRITE ||
00039 _dataRefs[i]->dataRefRWType == XPL_READWRITE))
00040         {
00041             if ((millis() - _dataRefs[i]->lastUpdateTime > _dataRefs[i]->updateRate) ||
00042 _dataRefs[i]->forceUpdate)
00043             {
00044                 switch (_dataRefs[i]->dataRefVARType)
00045                 {
00046                     case XPL_DATATYPE_INT:
00047                         if (*(long int *)_dataRefs[i]->latestValue != _dataRefs[i]->lastSentIntValue)
00048                         {
00049                             _sendPacketInt(XPLCMD_DATAREFUPDATE, _dataRefs[i]->dataRefHandle, *(long int
00050 *)_dataRefs[i]->latestValue);
00051                             _dataRefs[i]->lastSentIntValue = *(long int *)_dataRefs[i]->latestValue;
00052                             _dataRefs[i]->lastUpdateTime = millis();
00053                             _dataRefs[i]->forceUpdate = 0;
00054                         }
00055                         break;
00056                     case XPL_DATATYPE_FLOAT:
00057                         if (_dataRefs[i]->divider > 0)
00058                         {
00059                             *(float *)_dataRefs[i]->latestValue = ((int)(*(float *)_dataRefs[i]->latestValue /
00060 _dataRefs[i]->divider) * _dataRefs[i]->divider);
00061                         }
00062                         if (*(float *)_dataRefs[i]->latestValue != _dataRefs[i]->lastSentFloatValue)
00063                         {
00064                             _sendPacketFloat(XPLCMD_DATAREFUPDATE, _dataRefs[i]->dataRefHandle, *(float
00065 *)_dataRefs[i]->latestValue);
00066                             _dataRefs[i]->lastSentFloatValue = *(float *)_dataRefs[i]->latestValue;
00067                             _dataRefs[i]->lastUpdateTime = millis();
00068                             _dataRefs[i]->forceUpdate = 0;
00069                         }
00070                         break;
00071                     }
00072             }
00073         }
00074     }
00075     return _connectionStatus;
00076 }
00077
00078 int XPLDirect::commandTrigger(int commandHandle)
00079 {
00080     if (!_commands[commandHandle])
00081         return -1; // inactive command
00082     _sendPacketInt(XPLCMD_COMMANDTRIGGER, _commands[commandHandle]->commandHandle, 1);
00083     return 0;
00084 }
00085
00086 int XPLDirect::commandTrigger(int commandHandle, int triggerCount)
00087 {
00088     if (!_commands[commandHandle])
00089         return -1; // inactive command
00090     _sendPacketInt(XPLCMD_COMMANDTRIGGER, _commands[commandHandle]->commandHandle, (long
00091 int)triggerCount);
00092     return 0;
00093 }
00094
00095 int XPLDirect::commandStart(int commandHandle)
00096 {
00097     if (!_commands[commandHandle])
00098         return -1; // inactive command
00099 }

```

```

00093     _sendPacketVoid(XPLCMD_COMMANDSTART, _commands[commandHandle]->commandHandle);
00094     return 0;
00095 }
00096
00097 int XPLDirect::commandEnd(int commandHandle)
00098 {
00099     if (!_commands[commandHandle])
00100         return -1; // inactive command
00101     _sendPacketVoid(XPLCMD_COMMANDEND, _commands[commandHandle]->commandHandle);
00102     return 0;
00103 }
00104
00105 int XPLDirect::connectionStatus()
00106 {
00107     return _connectionStatus;
00108 }
00109
00110 int XPLDirect::sendDebugMessage(const char* msg)
00111 {
00112     _sendPacketString(XPLCMD_PRINTDEBUG, (char *)msg);
00113     return 1;
00114 }
00115
00116 int XPLDirect::sendSpeakMessage(const char* msg)
00117 {
00118     _sendPacketString(XPLCMD_SPEAK, (char *)msg);
00119     return 1;
00120 }
00121
00122
00123 int XPLDirect::hasUpdated(int handle)
00124 {
00125     if (_dataRefs[handle]->updatedFlag)
00126     {
00127         _dataRefs[handle]->updatedFlag = false;
00128         return true;
00129     }
00130     return false;
00131 }
00132
00133 int XPLDirect::datarefsUpdated()
00134 {
00135     if (_datarefsUpdatedFlag)
00136     {
00137         _datarefsUpdatedFlag = false;
00138         return true;
00139     }
00140     return false;
00141 }
00142
00143 void XPLDirect::_sendname()
00144 {
00145     if (_deviceName != NULL)
00146     {
00147         _sendPacketString(XPLRESPONSE_NAME, _deviceName);
00148     }
00149 }
00150
00151 void XPLDirect::_sendVersion()
00152 {
00153     if (_deviceName != NULL)
00154     {
00155         _sendPacketInt(XPLRESPONSE_VERSION, XPLDIRECT_ID, XPLDIRECT_VERSION);
00156     }
00157 }
00158
00159 void XPLDirect::sendResetRequest()
00160 {
00161     if (_deviceName != NULL)
00162     {
00163         _sendPacketVoid(XPLCMD_RESET, 0);
00164     }
00165 }
00166
00167 void XPLDirect::_processSerial()
00168 {
00169     while (streamPtr->available() && _receiveBuffer[0] != XPLDIRECT_PACKETHEADER)
00170     {
00171         _receiveBuffer[0] = (char)streamPtr->read();
00172     }
00173     if (_receiveBuffer[0] != XPLDIRECT_PACKETHEADER)
00174     {
00175         return;
00176     }
00177     _receiveBufferBytesReceived = streamPtr->readBytesUntil(XPLDIRECT_PACKETTRAILER, (char
*)&_receiveBuffer[1], XPLMAX_PACKETSIZE - 1);
00178     if (_receiveBufferBytesReceived == 0)

```

```

00179 {
00180     _receiveBuffer[0] = 0;
00181     return;
00182 }
00183 _receiveBuffer[++_receiveBufferBytesReceived] = XPLDIRECT_PACKETTRAILER;
00184 _receiveBuffer[++_receiveBufferBytesReceived] = 0; // old habits die hard.
00185 _processPacket();
00186 _receiveBuffer[0] = 0;
00187 }
00188
00189 void XPLDirect::_processPacket()
00190 {
00191     int i;
00192
00193     switch (_receiveBuffer[1])
00194     {
00195     case XPLCMD_RESET:
00196         _connectionStatus = false;
00197         break;
00198
00199     case XPL_EXITING :           // MG 03/14/2023: Added protocol code so the device will know if xplane
has shut down normally.
00200         _connectionStatus = false;
00201         break;
00202
00203     case XPLCMD_SENDNAME:
00204         _sendname();
00205         _connectionStatus = true;           // not considered active till you know my name
00206         for (i = 0; i < _dataRefsCount; i++) // also, if name was requested reset active datarefs and
commands
00207         {
00208             _dataRefs[i]->dataRefHandle = -1; // invalid again until assigned by Xplane
00209         }
00210         for (i = 0; i < _commandsCount; i++)
00211         {
00212             _commands[i]->commandHandle = -1;
00213         }
00214         break;
00215
00216     case XPLCMD_SENDVERSION:
00217     {
00218         _sendVersion();
00219         break;
00220     }
00221
00222     case XPLRESPONSE_DATAREF:
00223         for (int i = 0; i < _dataRefsCount; i++)
00224         {
00225             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)
00226             {
00227                 _dataRefs[i]->dataRefHandle = _getHandleFromFrame(); // parse the refhandle
00228                 _dataRefs[i]->updatedFlag = true;
00229                 i = _dataRefsCount; // end checking
00230             }
00231         }
00232         break;
00233
00234     case XPLRESPONSE_COMMAND:
00235         for (int i = 0; i < _commandsCount; i++)
00236         {
00237             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)
00238             {
00239                 _commands[i]->commandHandle = _getHandleFromFrame(); // parse the refhandle
00240                 i = _commandsCount; // end checking
00241             }
00242         }
00243         break;
00244
00245     case XPLCMD_SENDREQUEST:
00246     {
00247         int packetSent = 0;
00248         int i = 0;
00249         while (!packetSent && i < _dataRefsCount && i < XPLDIRECT_MAXDATAREFS_ARDUINO) // send dataref
registrations first
00250         {
00251             if (_dataRefs[i]->dataRefHandle == -1)
00252             { // some boards cant do sprintf with floats so this is a workaround
00253                 sprintf(_sendBuffer, "%c%c%1.1i%2.2i%05i.%02i%c", XPLDIRECT_PACKETHEADER,
XPLREQUEST_REGISTERDATAREF, _dataRefs[i]->dataRefRWType, _dataRefs[i]->arrayIndex,
00254                     (int)_dataRefs[i]->divider, (int)(_dataRefs[i]->divider * 100) % 100, (wchar_t
*)_dataRefs[i]->dataRefName, XPLDIRECT_PACKETTRAILER);
00255                 _transmitPacket();
00256                 packetSent = 1;
00257             }
00258             i++;

```

```

00259     }
00260     i = 0;
00261     while (!packetSent && i < _commandsCount && i < XPLDIRECT_MAXCOMMANDS_ARDUINO) // now send command
registrations
00262     {
00263         if (_commands[i]->commandHandle == -1)
00264         {
00265             sprintf(_sendBuffer, "%c%c%S%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_REGISTERCOMMAND, (wchar_t
*)_commands[i]->commandName, XPLDIRECT_PACKETTRAILER);
00266             _transmitPacket();
00267             packetSent = 1;
00268         }
00269         i++;
00270     }
00271     if (!packetSent)
00272     {
00273         _allDataRefsRegistered = true;
00274         sprintf(_sendBuffer, "%c%c%c", XPLDIRECT_PACKETHEADER, XPLREQUEST_NOREQUESTS,
XPLDIRECT_PACKETTRAILER);
00275         _transmitPacket();
00276     }
00277     break;
00278 }
00279
00280 case XPLCMD_DATAREFUPDATE:
00281 {
00282     int refhandle = _getHandleFromFrame();
00283     for (int i = 0; i < _dataRefsCount; i++)
00284     {
00285         if (_dataRefs[i]->dataRefHandle == refhandle && (_dataRefs[i]->dataRefRWType == XPL_READ ||
_dataRefs[i]->dataRefRWType == XPL_READWRITE))
00286         {
00287             if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_INT)
00288             {
00289                 _getPayloadFromFrame((long int *)_dataRefs[i]->latestValue);
00290                 _dataRefs[i]->lastSentIntValue = *(long int *)_dataRefs[i]->latestValue;
00291                 _dataRefs[i]->updatedFlag = true;
00292                 _dataRefsUpdatedFlag = true;
00293             }
00294             if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_FLOAT)
00295             {
00296                 _getPayloadFromFrame((float *)_dataRefs[i]->latestValue);
00297                 _dataRefs[i]->lastSentFloatValue = *(float *)_dataRefs[i]->latestValue;
00298                 _dataRefs[i]->updatedFlag = true;
00299                 _dataRefsUpdatedFlag = true;
00300             }
00301             if (_dataRefs[i]->dataRefVARType == XPL_DATATYPE_STRING)
00302             {
00303                 _getPayloadFromFrame((char *)_dataRefs[i]->latestValue);
00304                 _dataRefs[i]->updatedFlag = true;
00305                 _dataRefsUpdatedFlag = true;
00306             }
00307             i = _dataRefsCount; // skip the rest
00308         }
00309     }
00310     break;
00311 }
00312 case XPLREQUEST_REFRESH:
00313     for (int i = 0; i < _dataRefsCount; i++)
00314     {
00315         if (_dataRefs[i]->dataRefRWType == XPL_WRITE || _dataRefs[i]->dataRefRWType == XPL_READWRITE)
00316         {
00317             _dataRefs[i]->forceUpdate = 1; // bypass noise and timing filters
00318         }
00319     }
00320     break;
00321
00322 default:
00323     break;
00324 }
00325 }
00326
00327 void XPLDirect::_sendPacketInt(int command, int handle, long int value) // for ints
00328 {
00329     if (handle >= 0)
00330     {
00331         sprintf(_sendBuffer, "%c%c%3.3i%ld%c", XPLDIRECT_PACKETHEADER, command, handle, value,
XPLDIRECT_PACKETTRAILER);
00332         _transmitPacket();
00333     }
00334 }
00335
00336 void XPLDirect::_sendPacketFloat(int command, int handle, float value) // for floats
00337 {
00338     if (handle >= 0)
00339     {
00340         // some boards cant do sprintf with floats so this is a workaround.

```



```

00341     char tmp[16];
00342     dtostrf(value, 8, 6, tmp);
00343     sprintf(_sendBuffer, "%c%c%3.3i%s%c", XPLDIRECT_PACKETHEADER, command, handle, tmp,
XPLDIRECT_PACKETTRAILER);
00344     _transmitPacket();
00345 }
00346 }
00347
00348 void XPLDirect::_sendPacketVoid(int command, int handle) // just a command with a handle
00349 {
00350     if (handle >= 0)
00351     {
00352         sprintf(_sendBuffer, "%c%c%3.3i%c", XPLDIRECT_PACKETHEADER, command, handle,
XPLDIRECT_PACKETTRAILER);
00353         _transmitPacket();
00354     }
00355 }
00356
00357 void XPLDirect::_sendPacketString(int command, char *str) // for a string
00358 {
00359     sprintf(_sendBuffer, "%c%c%s%c", XPLDIRECT_PACKETHEADER, command, str, XPLDIRECT_PACKETTRAILER);
00360     _transmitPacket();
00361 }
00362
00363 void XPLDirect::_transmitPacket(void)
00364 {
00365     streamPtr->write(_sendBuffer);
00366     if (strlen(_sendBuffer) == 64)
00367     {
00368         streamPtr->print(" "); // apparantly a bug on some boards when we transmit exactly 64 bytes
00369     }
00370 }
00371
00372 int XPLDirect::_getHandleFromFrame() // Assuming receive buffer is holding a good frame
00373 {
00374     char holdChar;
00375     int handleRet;
00376     holdChar = _receiveBuffer[5];
00377     _receiveBuffer[5] = 0;
00378     handleRet = atoi((char *)&_receiveBuffer[2]);
00379     _receiveBuffer[5] = holdChar;
00380     return handleRet;
00381 }
00382
00383 int XPLDirect::_getPayloadFromFrame(long int *value) // Assuming receive buffer is holding a good
frame
00384 {
00385     char holdChar;
00386     holdChar = _receiveBuffer[15];
00387     _receiveBuffer[15] = 0;
00388     *value = atol((char *)&_receiveBuffer[5]);
00389     _receiveBuffer[15] = holdChar;
00390     return 0;
00391 }
00392
00393 int XPLDirect::_getPayloadFromFrame(float *value) // Assuming receive buffer is holding a good frame
00394 {
00395     char holdChar;
00396     holdChar = _receiveBuffer[15];
00397     _receiveBuffer[15] = 0;
00398     *value = atof((char *)&_receiveBuffer[5]);
00399     _receiveBuffer[15] = holdChar;
00400     return 0;
00401 }
00402
00403 int XPLDirect::_getPayloadFromFrame(char *value) // Assuming receive buffer is holding a good frame
00404 {
00405     memcpy(value, (char *)&_receiveBuffer[5], _receiveBufferBytesReceived - 6);
00406     value[_receiveBufferBytesReceived - 6] = 0; // erase the packet trailer
00407     for (int i = 0; i < _receiveBufferBytesReceived - 6; i++)
00408     {
00409         if (value[i] == 7)
00410         {
00411             value[i] = XPLDIRECT_PACKETTRAILER; // How I deal with the possibility of the packet trailer
being within a string
00412         }
00413     }
00414     return 0;
00415 }
00416
00417 int XPLDirect::allDataRefsRegistered()
00418 {
00419     return _allDataRefsRegistered;
00420 }
00421
00422 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
long int *value)

```

```

00423 {
00424     if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00425     {
00426         return -1; // Error
00427     }
00428     _dataRefs[_dataRefsCount] = new _dataRefStructure;
00429     _dataRefs[_dataRefsCount]->dataRefName = datarefName; // added for F() macro
00430     _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00431     _dataRefs[_dataRefsCount]->divider = divider;
00432     _dataRefs[_dataRefsCount]->updateRate = rate;
00433     _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_INT;
00434     _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00435     _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
00436     _dataRefs[_dataRefsCount]->arrayIndex = 0; // not used unless we are referencing an array
00437     _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00438     _dataRefsCount++;
00439     _allDataRefsRegistered = 0;
00440     return (_dataRefsCount - 1);
00441 }
00442
00443 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
long int *value, int index)
00444 {
00445     if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00446     {
00447         return -1;
00448     }
00449     _dataRefs[_dataRefsCount] = new _dataRefStructure;
00450     _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00451     _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00452     _dataRefs[_dataRefsCount]->updateRate = rate;
00453     _dataRefs[_dataRefsCount]->divider = divider;
00454     _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_INT; // arrays are dealt with on the XPlane
plugin side
00455     _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00456     _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
00457     _dataRefs[_dataRefsCount]->arrayIndex = index; // not used unless we are referencing an array
00458     _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00459     _dataRefsCount++;
00460     _allDataRefsRegistered = 0;
00461     return (_dataRefsCount - 1);
00462 }
00463
00464 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
float *value)
00465 {
00466     if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00467     {
00468         return -1;
00469     }
00470     _dataRefs[_dataRefsCount] = new _dataRefStructure;
00471     _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00472     _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00473     _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_FLOAT;
00474     _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00475     _dataRefs[_dataRefsCount]->lastSentFloatValue = -1; // force update on first loop
00476     _dataRefs[_dataRefsCount]->updateRate = rate;
00477     _dataRefs[_dataRefsCount]->divider = divider;
00478     _dataRefs[_dataRefsCount]->arrayIndex = 0; // not used unless we are referencing an array
00479     _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00480     _dataRefsCount++;
00481     _allDataRefsRegistered = 0;
00482     return (_dataRefsCount - 1);
00483 }
00484
00485 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, float divider,
float *value, int index)
00486 {
00487     if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00488     {
00489         return -1;
00490     }
00491     _dataRefs[_dataRefsCount] = new _dataRefStructure;
00492     _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00493     _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00494     _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_FLOAT; // arrays are dealt with on the
Xplane plugin side
00495     _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00496     _dataRefs[_dataRefsCount]->lastSentFloatValue = 0;
00497     _dataRefs[_dataRefsCount]->updateRate = rate;
00498     _dataRefs[_dataRefsCount]->arrayIndex = index; // not used unless we are referencing an array
00499     _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00500     _dataRefsCount++;
00501     _allDataRefsRegistered = 0;
00502     return (_dataRefsCount - 1);
00503 }
00504

```

```
00505 int XPLDirect::registerDataRef(XPString_t *datarefName, int rwmode, unsigned int rate, char *value)
00506 {
00507     if (_dataRefsCount >= XPLDIRECT_MAXDATAREFS_ARDUINO)
00508     {
00509         return -1;
00510     }
00511     _dataRefs[_dataRefsCount] = new _dataRefStructure;
00512     _dataRefs[_dataRefsCount]->dataRefName = datarefName;
00513     _dataRefs[_dataRefsCount]->dataRefRWType = rwmode;
00514     _dataRefs[_dataRefsCount]->updateRate = rate;
00515     _dataRefs[_dataRefsCount]->dataRefVARType = XPL_DATATYPE_STRING;
00516     _dataRefs[_dataRefsCount]->latestValue = (void *)value;
00517     _dataRefs[_dataRefsCount]->lastSentIntValue = 0;
00518     _dataRefs[_dataRefsCount]->arrayIndex = 0; // not used unless we are referencing an array
00519     _dataRefs[_dataRefsCount]->dataRefHandle = -1; // invalid until assigned by xplane
00520     _dataRefsCount++;
00521     _allDataRefsRegistered = 0;
00522     return (_dataRefsCount - 1);
00523 }
00524
00525 int XPLDirect::registerCommand(XPString_t *commandName) // user will trigger commands with
commandTrigger
00526 {
00527     if (_commandsCount >= XPLDIRECT_MAXCOMMANDS_ARDUINO)
00528     {
00529         return -1;
00530     }
00531     _commands[_commandsCount] = new _commandStructure;
00532     _commands[_commandsCount]->commandName = commandName;
00533     _commands[_commandsCount]->commandHandle = -1; // invalid until assigned by xplane
00534     _commandsCount++;
00535     _allDataRefsRegistered = 0; // share this flag with the datarefs, true when everything is registered
with xplane.
00536     return (_commandsCount - 1);
00537 }
00538
00539 XPLDirect XP;
```


Index

- [_cmdPush](#)
 - [Button, 14](#)
 - [_delay](#)
 - [RepeatButton, 29](#)
 - [_mux](#)
 - [Button, 14](#)
 - [_pin](#)
 - [Button, 15](#)
 - [_state](#)
 - [Button, 15](#)
 - [_timer](#)
 - [RepeatButton, 29](#)
 - [_transition](#)
 - [Button, 15](#)
- [addMux](#)
 - [DigitalIn_, 16](#)
- [allDataRefsRegistered](#)
 - [XPLDirect, 41](#)
- [AnalogIn, 7](#)
 - [AnalogIn, 7, 8](#)
 - [calibrate, 8](#)
 - [handle, 8](#)
 - [raw, 8](#)
 - [value, 9](#)
- [AnalogIn.cpp, 52](#)
- [AnalogIn.h, 45](#)
- [begin](#)
 - [XPLDirect, 41](#)
- [Button, 9](#)
 - [_cmdPush, 14](#)
 - [_mux, 14](#)
 - [_pin, 15](#)
 - [_state, 15](#)
 - [_transition, 15](#)
 - [Button, 11](#)
 - [engaged, 12](#)
 - [getCommand, 12](#)
 - [handle, 12](#)
 - [handleXP, 13](#)
 - [pressed, 13](#)
 - [processCommand, 13](#)
 - [released, 14](#)
 - [setCommand, 14](#)
- [Button.cpp, 53](#)
- [Button.h, 45](#)
- [calibrate](#)
 - [AnalogIn, 8](#)
- [commandEnd](#)
 - [XPLDirect, 41](#)
- [commandStart](#)
 - [XPLDirect, 41](#)
- [commandTrigger](#)
 - [XPLDirect, 41, 42](#)
- [connectionStatus](#)
 - [XPLDirect, 42](#)
- [count](#)
 - [Timer, 39](#)
- [datarefsUpdated](#)
 - [XPLDirect, 42](#)
- [DigitalIn.cpp, 54](#)
- [DigitalIn.h, 46](#)
- [DigitalIn_, 15](#)
 - [addMux, 16](#)
 - [DigitalIn_, 16](#)
 - [getBit, 17](#)
 - [handle, 17](#)
 - [setMux, 17](#)
- [down](#)
 - [Encoder, 20](#)
- [elapsed](#)
 - [Timer, 39](#)
- [Encoder, 18](#)
 - [down, 20](#)
 - [Encoder, 19](#)
 - [engaged, 20](#)
 - [getCommand, 20](#)
 - [handle, 21](#)
 - [handleXP, 21](#)
 - [pos, 21](#)
 - [pressed, 21](#)
 - [processCommand, 21](#)
 - [released, 22](#)
 - [setCommand, 22](#)
 - [up, 23](#)
- [Encoder.cpp, 55](#)
- [Encoder.h, 47](#)
- [engaged](#)
 - [Button, 12](#)
 - [Encoder, 20](#)
- [getBit](#)
 - [DigitalIn_, 17](#)
- [getCommand](#)
 - [Button, 12](#)
 - [Encoder, 20](#)

- Switch, 31
- Switch2, 35
- getTime
 - Timer, 39
- handle
 - AnalogIn, 8
 - Button, 12
 - DigitalIn_, 17
 - Encoder, 21
 - LedShift, 24
 - RepeatButton, 28
 - Switch, 31
 - Switch2, 35
- handleXP
 - Button, 13
 - Encoder, 21
 - RepeatButton, 28
 - Switch, 31
 - Switch2, 35
- hasUpdated
 - XPLDirect, 42
- LedShift, 23
 - handle, 24
 - LedShift, 24
 - set, 24
 - set_all, 25
- LedShift.cpp, 57
- LedShift.h, 48
- off
 - Switch, 31
 - Switch2, 35
- on
 - Switch, 32
- on1
 - Switch2, 36
- on2
 - Switch2, 36
- pos
 - Encoder, 21
- pressed
 - Button, 13
 - Encoder, 21
- processCommand
 - Button, 13
 - Encoder, 21
 - Switch, 32
 - Switch2, 36
- raw
 - AnalogIn, 8
- registerCommand
 - XPLDirect, 42
- registerDataRef
 - XPLDirect, 42, 43
- released
 - Button, 14
 - Encoder, 22
 - RepeatButton, 25
 - _delay, 29
 - _timer, 29
 - handle, 28
 - handleXP, 28
 - RepeatButton, 27
 - sendDebugMessage
 - XPLDirect, 44
 - sendResetRequest
 - XPLDirect, 44
 - sendSpeakMessage
 - XPLDirect, 44
 - set
 - LedShift, 24
 - set_all
 - LedShift, 25
 - setCommand
 - Button, 14
 - Encoder, 22
 - Switch, 32
 - Switch2, 36, 37
 - setCycle
 - Timer, 39
 - setMux
 - DigitalIn_, 17
 - Switch, 29
 - getCommand, 31
 - handle, 31
 - handleXP, 31
 - off, 31
 - on, 32
 - processCommand, 32
 - setCommand, 32
 - Switch, 30, 31
 - value, 33
 - Switch.cpp, 58
 - Switch.h, 48
 - Switch2, 33
 - getCommand, 35
 - handle, 35
 - handleXP, 35
 - off, 35
 - on1, 36
 - on2, 36
 - processCommand, 36
 - setCommand, 36, 37
 - Switch2, 34
 - value, 37
 - Timer, 38
 - count, 39
 - elapsed, 39
 - getTime, 39
 - setCycle, 39
 - Timer, 38
- Timer.cpp, 60

Timer.h, [49](#)

up

Encoder, [23](#)

value

AnalogIn, [9](#)

Switch, [33](#)

Switch2, [37](#)

xloop

XPLDirect, [44](#)

XPLDevices.h, [50](#)

XPLDirect, [40](#)

allDataRefsRegistered, [41](#)

begin, [41](#)

commandEnd, [41](#)

commandStart, [41](#)

commandTrigger, [41](#), [42](#)

connectionStatus, [42](#)

dataRefsUpdated, [42](#)

hasUpdated, [42](#)

registerCommand, [42](#)

registerDataRef, [42](#), [43](#)

sendDebugMessage, [44](#)

sendResetRequest, [44](#)

sendSpeakMessage, [44](#)

xloop, [44](#)

XPLDirect, [40](#)

XPLDirect.cpp, [60](#)

XPLDirect.h, [50](#)