PacketIO Documentation

Release 0.3.0

Eric Wieser

Jul 12, 2017

Contents

PacketIO is a C++ library for framing packets sent or received over an Arduino Stream, such as Serial. It is distributed as a PlatformIO library.

The key feature of this library over other framing implementations, such as PacketSerial, it that it operates on streams. This means that if your application layer is able to produce or consume streams, you can push these streams right the way through your program. Put simply, this means you can send arbitrarily large packets, without having to worry about allocating buffers.

The example below shows a contrived case when streams are needed - it duplicates whatever packets come in, but it works no matter how large the incoming packet is:

```
#include <Arduino.h>
#include <cobs/Stream.h>
#include <cobs/Print.h>
using namespace packetio;

COBSPrint cobs_out(Serial);
COBSStream cobs_in(Serial);

void setup() {
    Serial.begin();
}

void loop() {
    // send a packet
    cobs_out.print("Starting packet duplicator");
    cobs_out.end();

    // duplicate bytes of every packet read
    while(true) {
        int c = cobs_in.read();
        if(c == COBSStream::EOP) {
            // incoming packet ended - end ours too
            cobs_out.end();
            cobs_in.next();
        }
        else if(c != COBSStream::EOF) {
            // got a byte - duplicate it
            cobs_out.write(c);
            cobs_out.write(c);
        }
    }
}
```

CHAPTER 1

Contents

Basic interface

The framing methods are exposed through the *PacketPrint* and *PacketStream* interfaces.

class `packetio::PacketPrint` : **public** *Print*

Augments arduino's *Print* to add knowledge of packets.

Acts like a new *Print* for each packet. To stop printing to the current packet, and send new prints to the next one, call :func:end

This only concerns itself with the read parts of stream, and should not be used for writing.

Subclassed by *packetio::COBSPrint*, *packetio::EscapedPrint< EscapeCodes >*

Public Functions

virtual bool **`end`**`()` = 0

End the current packet.

Return true if successful

virtual void **`abort`**`()` = 0

Abort the current packet, preferably invalidating it if possible.

class `packetio::PacketStream` : **public** *Stream*

Augments arduino's *Stream* to add knowledge of packets.

Acts like a new stream for each packet, but instead of returning EOF when the packet is over, it returns EOP. EOF is returned when there is nothing to read, but the packet is not complete.

This only concerns itself with the read parts of stream, and should not be used for writing.

Subclassed by *packetio::COBSStream*, *packetio::EscapedStream< EscapeCodes >*

Public Functions

virtual int **read**() = 0
Read a single byte from the current packet.

Return The byte if available EOF if we have reached the end of the stream EOP if the current packet is complete

virtual int **peek**() = 0
Peek the next byte from the current packet.

Return The byte if available EOF if we have reached the end of the stream EOP if the current packet is complete

virtual int **available**() = 0

Return A lower bound on the number of bytes available. This count includes the EOP return value.

virtual void **next**() = 0
Advance to the next packet. If the current packet is not complete, then *read()* will return EOF until it is.

Public Static Attributes

const int **EOF** = -1
End Of File. This is already the value used by arduino.

const int **EOP** = -2
End Of Packet.

Framing Protocols

COBS

This is an implementation of Consistent-Overhead Byte Stuffing.

This uses a null byte as an end of packet marker, and uses a clever technique to encode null bytes within the packet with minimal overhead. See the link above for more information.

class `packetio::COBSPrint` : **public** packetio::*PacketPrint*

class `packetio::COBSStream` : **public** packetio::*PacketStream*

Escaped

This packet framer uses a special character to indicate end-of-frame, and an escape character to allow this to appear within a message. In the worst-case, this causes the packet to be twice the data size.

The choice of these special characters is parameterizable through template arguments. `encoded/codes.h` contains some example choices of values, including an implementation of SLIP. To define your own, you can use code like the following:

```
#include <escaped/Print.h>
#include <escaped/Stream.h>
#include <escaped/codes.h>

using namespace packetio;

// end, escape, escaped end, escaped escape
typedef EscapeCodes<'A','/','a','\\'> MyCodes;

EscapedPrint<MyCodes> printer(Serial);
EscapedStream<MyCodes> reader(Serial);
```

In this example, we use `A` to end a packet. So the packet `ABCD/EFGH` is encoded to `/aBCD/\EFGHA`. Here, the first *A* is replaced by the escape sequence `/a`, and the `/` is replaced with `/\`. Finally, an `A` is appended to end the packet.

template <uint8_t *pEND*, uint8_t *pESC*, uint8_t *pESC_END*, uint8_t *pESC_ESC*>
struct `packetio::`**`EscapeCodes`**

A trait type for indicating which markers to use in an *EscapedStream* or *EscapedPrint*. This type should be used as the type argument.

Template Parameters

- `pEND`: The byte indicating end of frame
- `pESC`: The byte indicating that the following byte is an escape code
- `pESC_END`: The escape code for encoding an in-data END value
- `pESC_ESC`: The escape code for encoding an in-data ESC value

template <typename *EscapeCodes*>
class `packetio::`**`EscapedPrint`** : **public** packetio::*PacketPrint*

template <typename *EscapeCodes*>
class `packetio::`**`EscapedStream`** : **public** packetio::*PacketStream*

SLIP

These is just a convenience aliases for escaped streams with the appropriate `EscapeCodes`.

typedef *EscapeCodes*<0xC0, 0xDB, 0xDC, 0xDD> `packetio::`**`SLIPEscapeCodes`**

typedef *EscapedPrint*<*SLIPEscapeCodes*> `packetio::`**`SLIPPrint`**

typedef *EscapedStream*<*SLIPEscapeCodes*> `packetio::`**`SLIPStream`**

Listening for packets

typedef *PacketListener_* `packetio::`**`PacketListener`**

Convenience typedef for a PacketListener with a sensible buffer size.

template <size_t *BufferSize* = 256>
class `packetio::`**`PacketListener_`**

Class for listening for packets on a *PacketStream*.

Template Parameters

- `BufferSize`: The size of the buffer to allocate for incoming packets

Public Types

enum `Error`
Types of error that can occur when receiving a packet.

Values:

`Overflow`
The end of the message could not be recieved because the buffer overflowed.

`Framing`
The message was framed incorrectly.

typedef LambdaPointer<void (uint8_t *, size_t, *Error*) > **`ErrorHandler`**

typedef LambdaPointer<void (uint8_t *, size_t) > **`MessageHandler`**

Public Functions

`PacketListener_` (*PacketStream &base*)
Construct a listener for the given packet stream.

void **`update`** ()
Read as much as possible from the underlying stream. If new packets are completed or errors occur, invoke the appropriate handler.

void **`onMessage`** (*MessageHandler handler*)
Set the handler to invoke when a message is recieved.

Note that because this is a LambdaPointer, the lambda function must have been allocated on the stack, such that its lifetime exceeds that of this object.

Parameters

- `handler`: The handler to invoke. This will be called with the pointer to the start of the message, and its length.

void **`onError`** (*ErrorHandler handler*)
Set the handler to invoke when an error occurs.

Note that because this is a LambdaPointer, the lambda function must have been allocated on the stack, such that its lifetime exceeds that of this object.

Parameters

- `handler`: The handler to invoke. This will be called with the pointer to the start of the message, and its length.

Example usage:

```
#include <packet_interface.h>
#include <PacketListener.h>
#include <cobs/Stream.h>
using namespace packetio;

void setup () {
    COBSStream cobs_serial_in(Serial);
```

```
    PacketListener handler(cobs_serial_in);

    int message_count = 0;
    auto message_handler = [&](const uint8_t* buffer, size_t len) {
        message_count++;
    };
    handler.onMessage(&message_handler);

    while(true) {
        handler.update();
        Serial.print("Message recieved: ");
        Serial.print(message_count);
        Serial.println();
    }
}
```

Internals

Callback functions

The `LambdaPointer` class is used to provide contextfull callback functions, using c++11 lambdas

template <typename Out, typename... *In*>
template<>
class `packetio::LambdaPointer`<Out (In...)>
A wrapper for storing a pointer to a lambda function, that works with captured variables.

Template Parameters

- `Out`: lambda return type
- `In`: lambda argument types

Public Functions

template <typename T>
`LambdaPointer`(T **lambda*)
Create a LambdaPointer from a lambda, possibly with captures.

Parameters

- `lambda`: A pointer to the lambda function. Because this a pointer, this must be allocated on the stack.

`LambdaPointer`(Out (**fptr*)) In...
Create a LambdaPointer from a raw function pointer.

Parameters

- `fptr`: The context-less function

Out **`operator()`**(In... *in*)
Invoke the underlying function.

operator bool()
Determine if the reference has been initialized.

Mirror of the Arduino interface

These classes, found in `_compat`, allow testing on the desktop, and potentially execution on non-Arduino platforms

class Print
Lightweight mirror of the (undocumented) arduino *Print* class, for use on other platforms.

Subclassed by *packetio::PacketPrint*, *Stream*

Public Functions

virtual size_t **write**(uint8_t) = 0

size_t **write**(**const** char *_str_)

virtual size_t **write**(**const** uint8_t *_buffer_, size_t _size_) = 0

size_t **write**(**const** char *_buffer_, size_t _size_)

class Stream : *Print*
Lightweight mirror of the arduino Stream class, for use on other platforms.

Subclassed by *packetio::PacketStream*

Public Functions

virtual int **available**() = 0

virtual int **read**() = 0

virtual int **peek**() = 0

virtual void **flush**() = 0

Stream()

size_t **readBytes**(char *_buffer_, size_t _length_)

size_t **readBytes**(uint8_t *_buffer_, size_t _length_)

Index

P

S