From Github Wiki

Release

Giovanni Blu Mitolo

Mar 06, 2017

Contents

CHAPTER 1

Documentation

PJON is designed to be as user friendly and minimally technical as possible, so it can be quickly mastered by new users. Visit the dedicated sections to get detailed info:

Feel free to write me personally gioscarab@gmail.com

CHAPTER 2

Interfacing

PJON is designed to be as user friendly and less technical as possible to be fastly mastered also by new users. Visit the architecture dedicated sections to get detailed info on your setup:

ATtiny Interfacing

AVR ATtiny microcontroller family is a really interesting and compact platform supported by PJON Arduino compatible implementation and soon also by PJON_ASK wireless implementation.

####How to program ATtiny 45/85 You physically need at least one ATtiny microcontroller, a breadboard, some jumpers and an Arduino duemilanove / Uno used as an Arduino ISP programmer. Follow High-Low Tech tutorial by David Mellis and get the last version of the attiny repository.

####Use PJON with ATtiny45/85 PJON Arduino compatible implementation works smoothly on ATtiny45/85 8Mhz (internal) with pin 2 and 3.

####Use PJON with ATtiny45/85 with external oscillator Because of the internal clock's lack of precision, with some ATtiny85 in particular, low communication performance can be detected; extended tests proven the ATtiny internal clock to be extremely inaccurate (timing inconsistency between two identical ATtiny85 can be detected). Here is an example how it works with external 16 MHz oscillator.

This is the sketch for Arduino UNO:

```
#include <PJON.h>
PJON<SoftwareBitBang> bus(1); // <Strategy name> bus(selected device id)
#define LED 13
void setup() {
  pinModeFast(LED, OUTPUT);
  digitalWriteFast(LED, LOW);
  bus.strategy.set_pin(12);
  bus.include_sender_info(false);
  bus.begin();
}
void loop() {
```

```
  digitalWrite(LED, HIGH);
  delay(30);
  digitalWrite(LED, LOW);
  bus.send_packet_blocking(2, "B",1);
  delay(30);
};
```

This is the sketch for ATtiny85:

```
#include <PJON.h>
PJON<SoftwareBitBang> bus(2); // <Strategy name> bus(selected device id)
#define LED 0
void setup() {
  pinModeFast(LED, OUTPUT);
  digitalWriteFast(LED, LOW);
  bus.strategy.set_pin(2);
  bus.include_sender_info(false);
  bus.begin();
  bus.set_receiver(receiver_function);
};
void receiver_function(uint8_t *payload, uint8_t length, const PacketInfo &packet_
↪info) {
  digitalWrite(LED, HIGH);
  delay(30);
  digitalWrite(LED, LOW);
}
void loop() {
  bus.receive(1000);
};
```

ESP8266 Interfacing

If you are creating a 3.3v only PJON bus the wiring needed is really simple:

If you are connecting an ESP8266 to a 5v bus you need to add some complexity to the standard PJON wiring. Logic level converter usage is strongly suggested otherwise you risk to fry ESP8266 pins and to have communication problems due to logic 1 treshold of 5v boards. Here you can find a fritzing project made by aperepel: http://fritzing.org/projects/pjon-with-uno-and-esp8266nodemcu

Note:

- Add 10KOhm pulldown resistors on Arduino Duemilanove / Uno / Nano side.
- Add 10KOhm pulldown resistors on ESP side (converter without integrated pullup resistors).
- Add 1-2KOhm pulldown resistors on ESP side (converter with integrated pullup resistors).

Here's the diagram from the project.

Deal with interference

In some situations, interference can affect the communication medium; understanding its source and how to mitigate its effects is a key element to master understanding in electronics and digital communication. Interference affecting a PJON bus can be divided in three different groups:

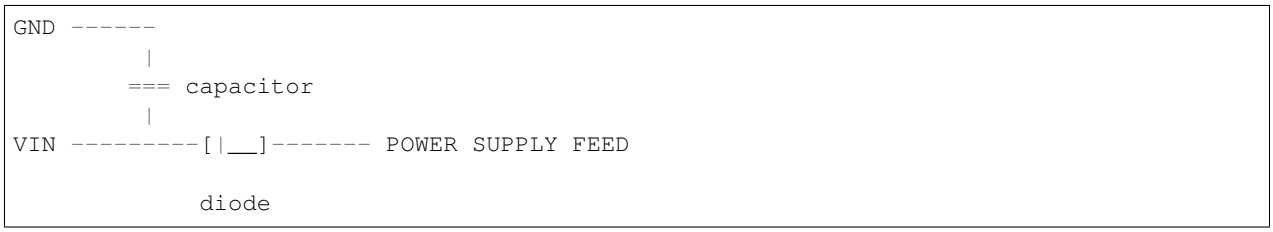
- Generated by hardware connected to the bus.
- Generated by third-party hardware not connected to the bus or device's power supply.
- Unknown.

The third category describes interference that can occur because of unknown sources, the extremely complicated and interrelated behavior of radiation waves often brings impossible to determine the cause (or the sum of causes) of a disturbing signal.

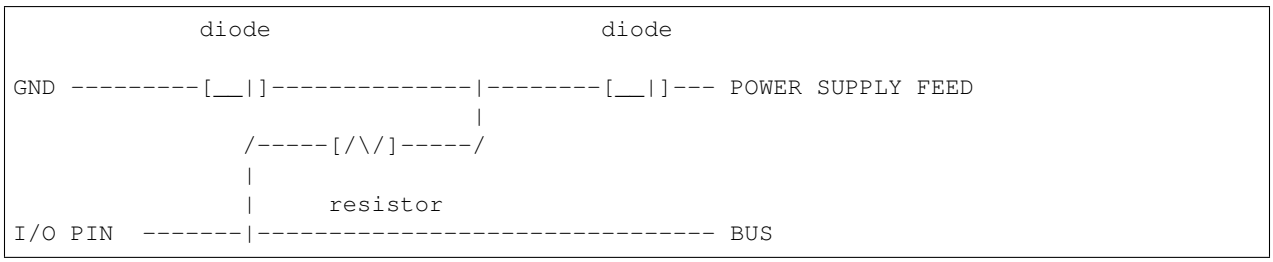
Connected hardware related interference

The most experienced sort of interference is a high chance to find the channel `BUSY`. This happens because the pin is "floating" from a logic state to the other one. This often happens because the bus, and the devices connected to it are acting as a capacitor, often reaching the `HIGH` logical state also in a quiescent state. A fast solution to this problem is the use of a pull-down resistor to discharge the unwanted capacitance build-up in the channel. The correct resistor value to be applied is very variable, depending on the topology and medium used, but is often near to the mega ohm order.

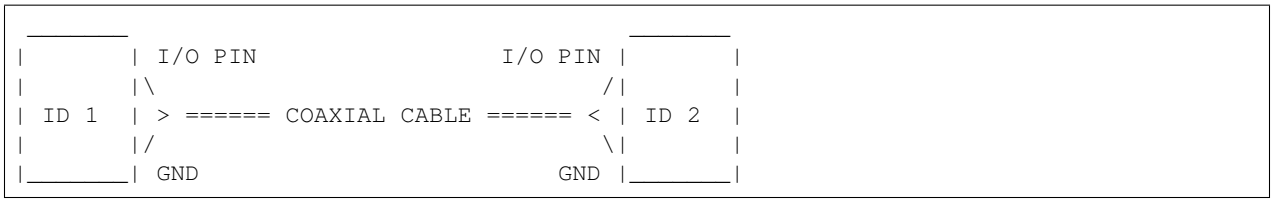
Bandwidth loss related to the system's power consumption is the sign of power supply interference. Never feed devices with shared power supply with high-power demanding appliances like servos, motors or actuators. If this is not possible at least position a diode connected to `VIN`, able to block inverse current from the `VIN` pin back to the wire and use a capacitor to fill temporary feeding shortages.



The above solution can reduce but not eliminate transient voltage spikes. A more decisive way to deal with transient voltage is to use diodes in opposition. This technique reduces the short duration electrical spikes can occur because of power shortages, power transitions in other large equipment on the same power line or lightning strikes.



A serious step toward reliability is to use a well insulated wire (mil standard for radio communication / avionics surplus), but this can only be applied for home-made / non-serial production; a more standard approach is the use of a simple coaxial cable in one of its forms, from earbuds wire to super-expensive gold-plated coaxial. Thanks to the "sock" ground shielding the use of this sort of cable highly reduces interference and is also really comfortable, connecting also the ground with only one wire.



Third-party hardware related interference

Electromagnetic fields can temporary charge the bus and provoke series of burst-errors. This is often provoked by powerful rotating, magnetic motors, welders, tasers and other devices able to burst a mix of high-power radio waves and magnetic fields. A strong palliative is the use of ferrite beads. Many devices where digital communication is used are equipped with it (see Sony Playstation old wired controller).

Troubleshooting

Also if PJON is designed to be a really stable, interference and error aware communication Standard, noise can variate enormously because of the environment, the setup and medium you are using as communication channel. PJON single bidirectional medium is not a balanced pair, for this reason long distances and interference sources can affect communication reliability and data throughput.

If you have a Saleae Logic Analyzer you will be amazed by saleae-pjon-protocol-analyzer crafted by the user aperepel, it offers a complete analysis suite aware of the Standard functional procedures and symbols.

Pull from PJON's desired implementation repository the master and run the example `NetworkAnalysis`, `SpeedTest` and `ErrorTest`. These 3 sketches are designed to execute a test and respond through the Serial monitor with a benchmark on communication channel performance and reliability:

- Absolute communication speed
- Practical bandwidth or channel throughput
- Number of packets sent in the test window
- How many errors detected with CRC
- How many times the receive function ended with no reception
- How many times the channel is found busy
- Accuracy (packets sent / packets received with mistakes ratio)

If you detect absent or slow communication speed, a lot of CRC detected mistakes and / or channel often busy, here you can find a list of really common issues that can lead to this problem:

- If necessary (i.e. wire / conductive medium) use common ground for every device.
- Pin configuration in your code.
- Physical wiring to the pin.

- Device ID configuration in your code.
- Other tasks are occupying all the available loop time.
- Uncorrect packet length passed to `send()` function
- Forgot the `update()` or `receive()` function in loop ;)

If you haven't identified the problem or you are a more advanced user, porting a new device / architecture, here you can find a list of common causes of this problem:

####Range

- **Long distance between devices.**
- Many failed receptions.
- Many mistakes detected by CRC.

You are probably near the maximum distance range of your system. The most straight-forward solution is to higher transmission power or to use a small capacitor to filter 1s received as 0s because of distance weakened voltage.

####Interference

- **Channel detected busy many times.**
- Many mistakes detected by CRC.
- Low or absent communication speed.

Device avoids to transmit over noise to ensure correct communication, when the medium is affected by noise, data throughput and communication reliability drops. Because of interference are also detected mistakes by CRC. See the dedicated wiki page Deal with interference.

####Timing

- **Many failed receptions.**
- Many mistakes detected by CRC.
- Low or absent communication speed.

Bad syncronization or timing configuration. If you are porting a new device or architecture try to tweak `BIT_WIDTH`, `BIT_SPACER`, `READ_DELAY` and `ACCEPTANCE` in `PJON.h` and consider that every architecture will execute code with a different timing.

####Execution time

- **Low quality of communication also after tuning / timing tweek.**
- Many failed receptions.
- Many CRC detected mistakes.
- Low or absent communication speed.

Every architecture needs a different time to execute the same PJON's code, so at the point where it should start to read the first bit of a byte, after initial padding bits, it can be a little shifted in time relatively to the transmitter, and so not able to read correctly the bit sequence. `READ_DELAY` constant in `PJON.h` regulates the correct positioning in every bit of the 8 readings in time. Take in consideration that a still not implemented architecture / device may not be fast enough to run PJON, try using a faster clock or optimize digital I/O (see `digitalWriteFast.h`).

Feel free to write me personally gioscarab@gmail.com

CHAPTER 3

Strategies

A Strategy its a class containing a set of functions able to physically communicate data through the medium used, abstracting the physical layer from PJON procedure and codebase.

```
boolean can_start(uint8_t input_pin, uint8_t output_pin)
```

Should Return `true` if the medium is free for use and `false` if the medium is in use by some other device.

```
void send_byte(uint8_t input_pin, uint8_t output_pin)
```

Sends a byte on a pin

```
uint16_t receive_byte(uint8_t input_pin, uint8_t output_pin)
```

Receives a byte from a pin

```
void send_response(uint8_t response, uint8_t input_pin, uint8_t output_pin)
```

Send a response to the packet's transmitter

```
uint16_t receive_response(uint8_t input_pin, uint8_t output_pin)
```

Receives a response from the packet's receiver

You can define your own set of functions to use PJON with your personal strategy on the medium you prefer. As can see two pins are passed to the methods, enabling twisted pair, serial or radio transceiver physical layer strategies. If you need other functions, variables or constants, those can be defined in your personal Strategy class. Other communication protocols could be used inside those functions to transmit data.

```
// Simple Serial physical layer example
static inline __attribute__((always_inline))
void send_byte(uint8_t b, uint8_t input_pin, uint8_t output_pin) {
  Serial.print(b);
}
```

####How to define a new strategy To define your new strategy you have only to create a new folder named for example `YourStrategyName` in `strategies` directory and write the necessary file `YourStrategyName.h`:

```
class YourStrategyName {
  public:
    boolean can_start() { ... };
    uint16_t receive_byte(uint8_t input_pin, uint8_t output_pin) { ... };
    void send_byte(uint8_t b, uint8_t input_pin, uint8_t output_pin) { ... };
    uint16_t receive_response(uint8_t input_pin, uint8_t output_pin) { ... };
    void send_response(uint8_t response, input_pin, uint8_t output_pin) { ... };
}
```

Simply add your code in the functions declaration shown above and instantiate PJON using the strategy type you have created: `PJON<YourStrategyName> bus();`.

#Compliant Tools

Here a list of the compliant tools:

Feel free to write me personally gioscarab@gmail.com

CHAPTER 4

Press

PJON has been released in the public domain through the Arduino forum and www.gioblu.com in the far 2010. Only in 2015, thanks to the first publication of Michael Teew and then soon Hackernews, the PJON's repository started receiving a lot of traffic and new users. Since that day PJON has been reviewed by some of the most visited websites and blogs: