VirtualWire

Mike McCauley

Documentation for the VirtualWire 1.6 communications library for Arduino.

1.0 Introduction

Arduino is a low cost microcontroller with Open Source hardware, see http://www.arduino.cc. VirtualWire is a communications library for Arduino that allows multiple Arduino's to communicate using low-cost RF transmitters and receivers.

The document describes the VirtualWire library and how to install and use it.

2.0 Overview

VirtualWire is an Arduino library that provides features to send short messages, without addressing, retransmit or acknowledgment, a bit like UDP over wireless, using ASK (amplitude shift keying). Supports a number of inexpensive radio transmitters and receivers. All that is required is transmit data, receive data and (for transmitters, optionally) a PTT transmitter enable.

It is intended to be compatible with the RF Monolithics (www.rfm.com) Virtual Wire protocol, but this has not been tested.

Does not use the Arduino UART. Messages are sent with a training preamble, message length and checksum. Messages are sent with 4-to-6 bit encoding for good DC balance, and a CRC checksum for message integrity.

Why not just use the Arduino UART connected directly to the transmitter/receiver? As discussed in the RFM documentation, ASK receivers require a burst of training pulses to synchronize the transmitter and receiver, and also requires good balance between 0s and 1s in the message stream in order to maintain the DC balance of the message.

UARTs do not provide these. They work a bit with ASK wireless, but not as well as this code.

2.1 Supported hardware.

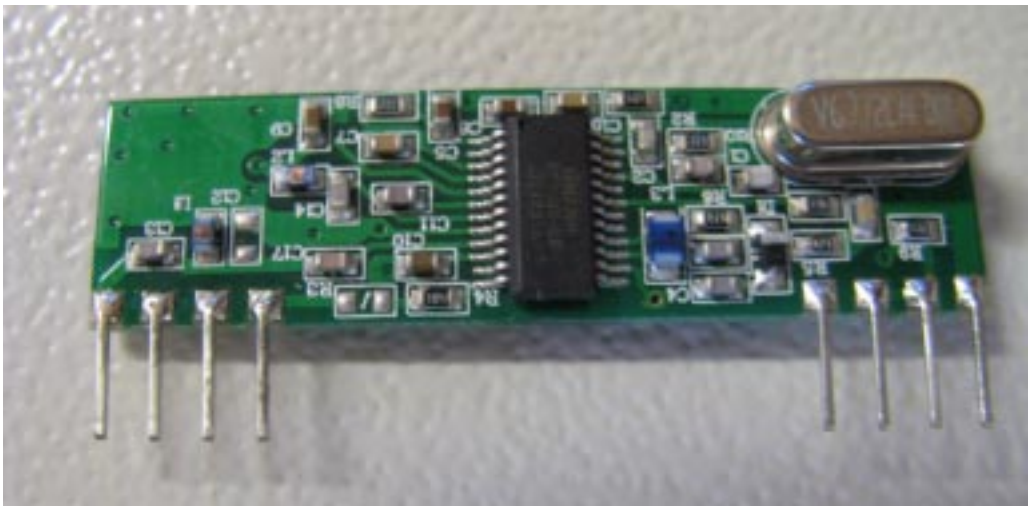
A range of communications hardware is supported. The ones listed blow are available in common retail outlets in Australian and other countries for under $10 per unit. Many other modules may also work with this software.

Runs on ATmega8/168 (Arduino Diecimila etc) and ATmega328 and possibly others.

2.2 Receivers

- RX-B1 (433.92MHz) (also known as ST-RX04-ASK)

FIGURE 1. RX-B1

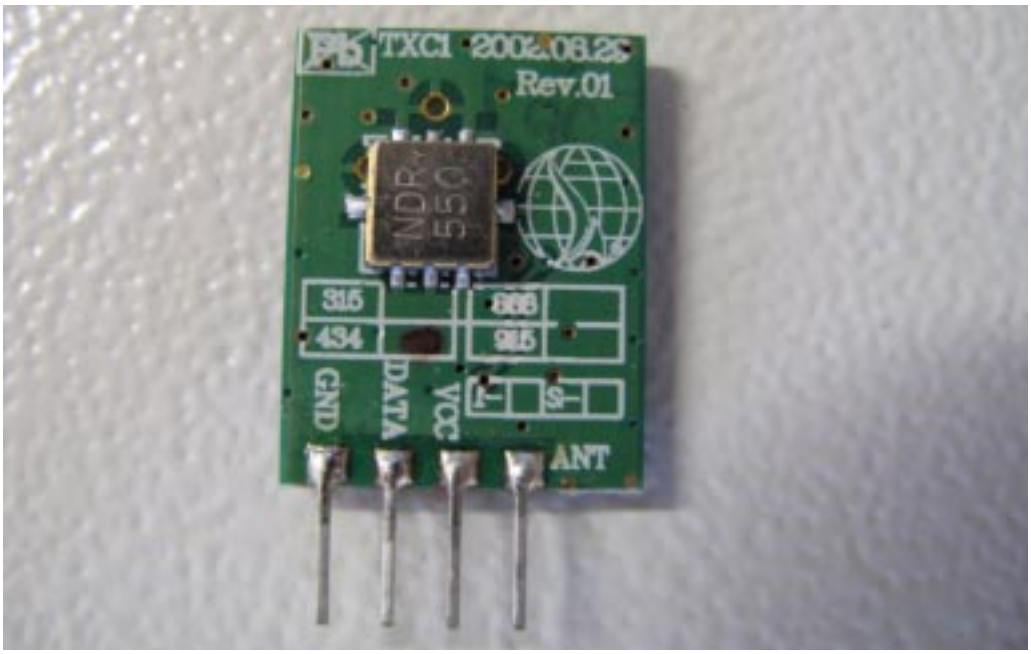


Details at http://www.summitek.com.tw/ST_SPEC/ST-RX04-ASK.pdf

2.3 Transmitters:

- TX-C1 (433.92MHz)

FIGURE 2. TX-C1



Details at http://www.tato.ind.br/files/TX-C1.pdf

2.4 Transceivers:

- DR3100 (433.92MHz)

FIGURE 3. DR3100

Details at http://www.rfmonolithics.com/products/data/dr3100.pdf

3.0 Downloading and installation

The latest version of this document is available from

```
http://www.open.com.au/mikem/arduino/VirtualWire.pdf
```

Download the VirtualWire distribution from

```
http://www.open.com.au/mikem/arduino/VirtualWire-1.5.zip
```

To install, unzip the library into the `libraries` sub-directory of your Arduino application directory. Then launch the Arduino environment; you should see the library in the Sketch->Import Library menu, and example code in File->Sketchbook->Examples->VirtualWire menu.

4.0 Function calls

To use the VirtualWire library, you must have

```
#include <VirtualWire.h>
```

At the top of your sketch.

4.1 vw_set_tx_pin

```
extern void vw_set_tx_pin(uint8_t pin);
```

Set the digital IO pin to use for transmit data. Defaults to 12.

4.2 vw_set_rx_pin

```
extern void vw_set_rx_pin(uint8_t pin);
```

Set the digital IO pin to use for receive data. Defaults to 11.

4.3 vw_set_ptt_pin

```
extern void vw_set_ptt_pin(uint8_t pin);
```

Set the digital IO pin to use to enable the transmitter (press to talk). Defaults to 10. Not all transmitters require PTT. The DR3100 does, but the TX-B1 does not.

4.4 vw_set_ptt_inverted

```
extern void vw_set_ptt_inverted(uint8_t inverted);
```

By default the PTT pin goes high when the transmitter is enabled. This flag forces it low when the transmitter is enabled. Required for the DR3100.

4.5 vw_setup

```
extern void vw_setup(uint16_t speed);
```

Initialise the VirtualWire software, to operate at *speed* bits per second. Call this once in your setup() after any vw_set_* calls. You must call vw_rx_start() after this before you will get any messages.

4.6 vw_rx_start

```
extern void vw_rx_start();
```

Start the receiver. You must do this before you can receive any messages. When a message is available (good checksum or not), vw_have_message() will return true.

4.7 vw_rx_stop

```
extern void vw_rx_stop();
```

Stop the receiver. No messages will be received until vw_rx_start() is called again. Saves interrupt processing cycles when you know there will be no messages.

4.8 vx_tx_active

```
extern uint8_t vx_tx_active();
```

Return true if the transmitter is active

4.9 vw_wait_tx

```
extern void vw_wait_tx();
```

Block and wait until the transmitter is idle

4.10 vw_wait_rx

```
xtern void vw_wait_rx();
```

Block and wait until a message is available from the receiver.

4.11 vw_wait_rx_max

```
extern uint8_t vw_wait_rx_max(unsigned long milliseconds);
```

Wait at most *milliseconds* ms for a message to be received. Return true if a message is available.

4.12 vw_send

```
extern uint8_t vw_send(uint8_t* buf, uint8_t len);
```

Send a message with the given length. Returns almost immediately, and the message will be sent at the right timing by interrupts. Returns true if the message was accepted for transmission. Returns false if the message is too long (>VW_MAX_PAYLOAD).

4.13 vw_have_message

```
extern uint8_t vw_have_message();
```

Returns true if an unread message is available from the receiver.

4.14 vw_get_message

```
extern uint8_t vw_get_message(uint8_t* buf, uint8_t* len);
```

If a message is available (good checksum or not), copies up to *len octets to buf. Returns true if there was a message *and* the checksum was good.

5.0 Sample code

The following samples are available as examples in the VirtualWire distribution.

5.1 transmitter

A simplex (one-way) transmitter. Sends a short message every 400 ms. Test this with the receiver below.

```
#include <VirtualWire.h>
void setup()
{
    vw_setup(2000); // Bits per sec
}

void loop()
{
    const char *msg = "hello";
    vw_send((uint8_t *)msg, strlen(msg));
    delay(400);
}
```

5.2 receiver

A simplex (one-way) receiver. Waits for a message and dumps it contents. Test this with transmitter above.

```
#include <VirtualWire.h>
void setup()
{
    Serial.begin(9600);
    Serial.println("setup");
    vw_setup(2000); // Bits per sec
    vw_rx_start();    // Start the receiver PLL running
}
void loop()
{
    uint8_t buf[VW_MAX_MESSAGE_LEN];
    uint8_t buflen = VW_MAX_MESSAGE_LEN;
    if (vw_get_message(buf, &buflen)) // Non-blocking
    {
          int i;
          // Message with a good checksum received, dump HEX
          Serial.print("Got: ");
          for (i = 0; i < buflen; i++)
          {
              Serial.print(buf[i], HEX);
              Serial.print(" ");
          }
          Serial.println("");
   }
}
```

5.3 client

Implements a simple wireless client for DR3100. Sends a message to another Arduino running the server code below and waits for a reply.

```
#include <VirtualWire.h>
void setup()
{
    Serial.begin(9600);// Debugging only
    Serial.println("setup");
    vw_set_ptt_inverted(true); // Required for DR3100
```

```
    vw_setup(2000); // Bits per sec
    vw_rx_start();       // Start the receiver PLL running
}
void loop()
{
    const char *msg = "hello";
    uint8_t buf[VW_MAX_MESSAGE_LEN];
    uint8_t buflen = VW_MAX_MESSAGE_LEN;
    vw_send((uint8_t *)msg, strlen(msg));
    vw_wait_tx(); // Wait until the whole message is gone
    Serial.println("Sent");
   // Wait at most 400ms for a reply
   if (vw_wait_rx_max(400))
   {
          if (vw_get_message(buf, &buflen)) // Non-blocking
          {
              int i;
              // Message with a good checksum received, dump it.
              Serial.print("Got reply: ");
              for (i = 0; i < buflen; i++)
              {
                 Serial.print(buf[i], HEX);
                 Serial.print(" ");
              }
              Serial.println("");
          }
    }
    else
          Serial.println("Timout");
}
```

5.4 server

Implements a simple wireless server for DR3100. Waits for a message from another Arduino running the client code above and sends a reply.

```
#include <VirtualWire.h>
void setup()
{
    Serial.begin(9600);// Debugging only
    Serial.println("setup");
    vw_set_ptt_inverted(true); // Required for DR3100
    vw_setup(2000); // Bits per sec
    vw_rx_start();       // Start the receiver PLL running
}
void loop()
{
    const char *msg = "hello";
    uint8_t buf[VW_MAX_MESSAGE_LEN];
    uint8_t buflen = VW_MAX_MESSAGE_LEN;

    // Wait for a message
    vw_wait_rx();
    if (vw_get_message(buf, &buflen)) // Non-blocking
```

```
    {
          int i;
          const char *msg = "goodbye";
          // Message with a good checksum received, dump it.
          Serial.print("Got: ");
          for (i = 0; i < buflen; i++)
          {
              Serial.print(buf[i], HEX);
              Serial.print(" ");
          }
          Serial.println("");
          // Send a reply
          vw_send((uint8_t *)msg, strlen(msg));
    }
}
```

6.0 Implementation Details

Messages of up to VW_MAX_PAYLOAD (27) bytes can be sent

Each message is transmitted as:

- 36 bit training preamble consisting of 0-1 bit pairs
- 12 bit start symbol 0xb38
- 1 byte of message length byte count (4 to 30), count includes byte count and FCS bytes
- n message bytes
- 2 bytes FCS, sent low byte-hi byte

Everything after the start symbol is encoded 4 to 6 bits, Therefore a byte in the message is encoded as 2x6 bit symbols, sent hi nybble, low nybble. Each symbol is sent LSBit first.

The Arduino Diecimila clock rate is 16MHz => 62.5ns/cycle.

For an RF bit rate of 2000 bps, need 500microsec bit period.

The ramp requires 8 samples per bit period, so need 62.5microsec per sample => interrupt tick is 62.5microsec.

The maximum message length consists of

(6 + 1 + VW_MAX_MESSAGE_LEN) * 6 = 222 bits = 0.11 secs (at 2000 bps).

The code consists of an ISR interrupt handler. Most of the work is done in the interrupt handler for both transmit and receive, but some is done from the user level. Expensive functions like CRC computations are always done in the user level.

7.0 Performance

Unit tests show that the receiver PLL can stand up to 1 sample in 11 being inverted by noise without ill effect.

Testing with TX-C1, RX-B1, 5 byte message, 17cm antenna, no ground plane, 1m above ground, free space

At 10000 bps the transmitter does not operate correctly (ISR running too frequently at 80000/sec?)

At 9000 bps, asymmetries in the receiver prevent reliable communications at any distance

At 7000bps, Range about 90m

At 5000bps, Range about 100m

At 2000bps, Range over 150m

At 1000bps, Range over 150m

As suggested by RFM documentation, near the limits of range, reception is strongly influenced by the presence of a human body in the signal line, and by module orientation.

Throughout the range there are nulls and strong points due to multipath reflection. So... your mileage may vary.

Similar performance figures were found for DR3100. 9000bps worked.

Arduino and TX-C1 transmitter draws 27mA at 9V.

Arduino and RX-B1 receiver draws 31mA at 9V.

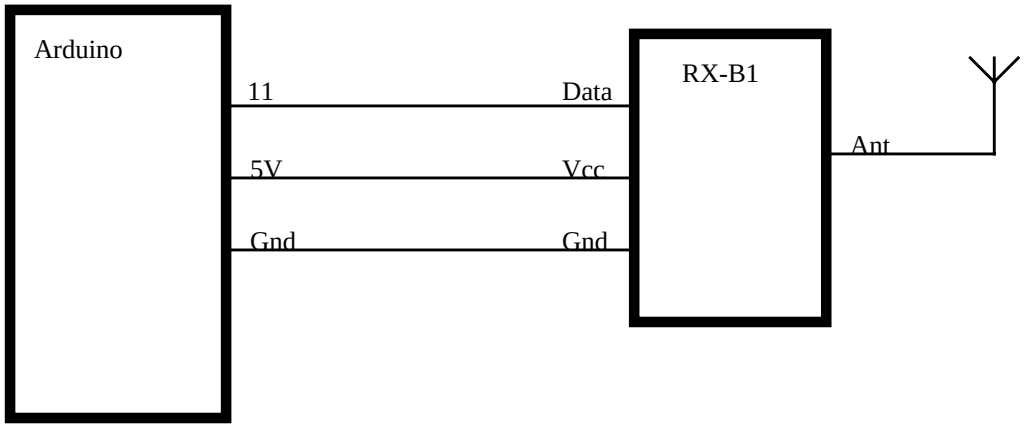
Arduino and DR3100 receiver draws 28mA at 9V.

8.0 Connections

Note that the IO pins can be changed from their defaults (as shown here) to any suitable IO pins using the vw_set_*_pin() calls.

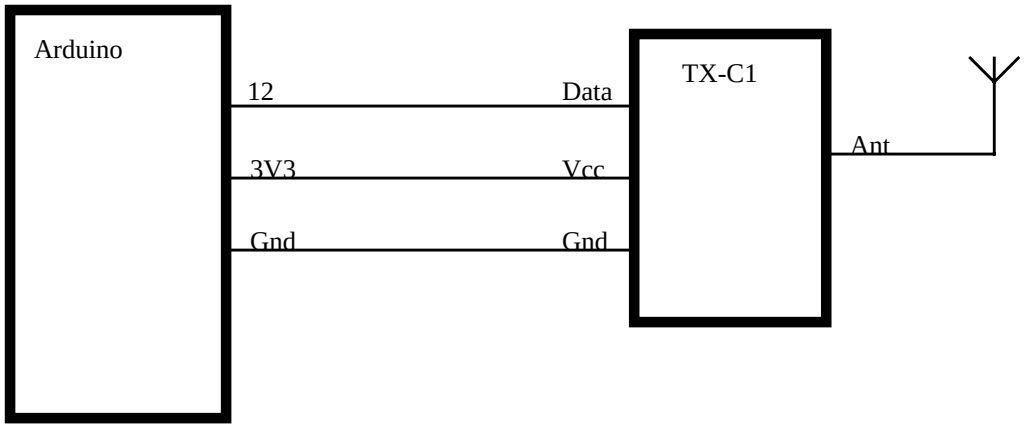
8.1 RX-B1 receiver

FIGURE 4. Wiring for RX-B1 receiver



8.2 TX-C1 transmitter

FIGURE 5. Wiring for TX-C1 transmitter



8.3 DR3100 transceiver

FIGURE 6. Wiring for DR3100

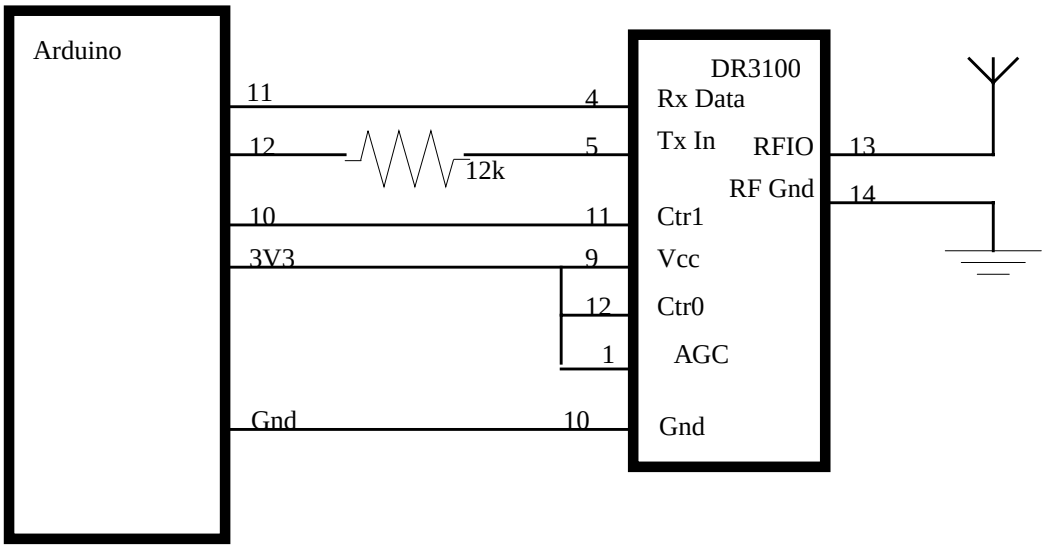
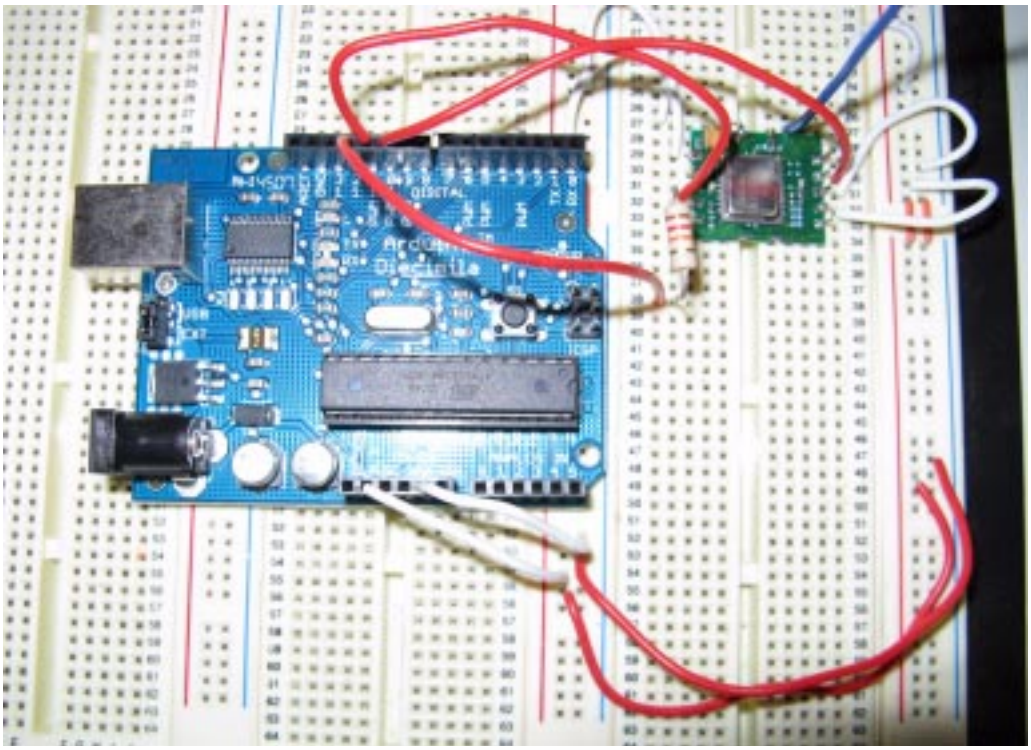


FIGURE 7. DR3100 connections on breadboard



Connections shown for no AGC, 1mW power out, 2400bps. Note the 12k resistor in series with Tx In to control the power output If you want to use higher data rates, need a resistor from pin 8 of the DR3100 to ground (see RFM documentation for more details).

If you want to use AGC, need a capacitor from pin 1 of the DR3100 to ground (see RFM documentation for more details).

The DR3100 module is supplied without any connection pins, only surface mount style pads. You will need to solder pins onto the module if you wish to use it in a breadboard.

9.0 Copyright and License

9.1 Open Source Licensing GPL V2

This is the appropriate option if you want to share the source code of your application with everyone you distribute it to, and you also want to give them the right to share who uses it. If you wish to use this software under Open Source Licensing, you must contribute all your source code to the open source community in accordance with the GPL Version 2 when your application is distributed. See http://www.gnu.org/copyleft/gpl.html

9.2 Commercial Licensing

This is the appropriate option if you are creating proprietary applications and you are not prepared to distribute and share the source code of your application. Contact info@open.com.au for details.