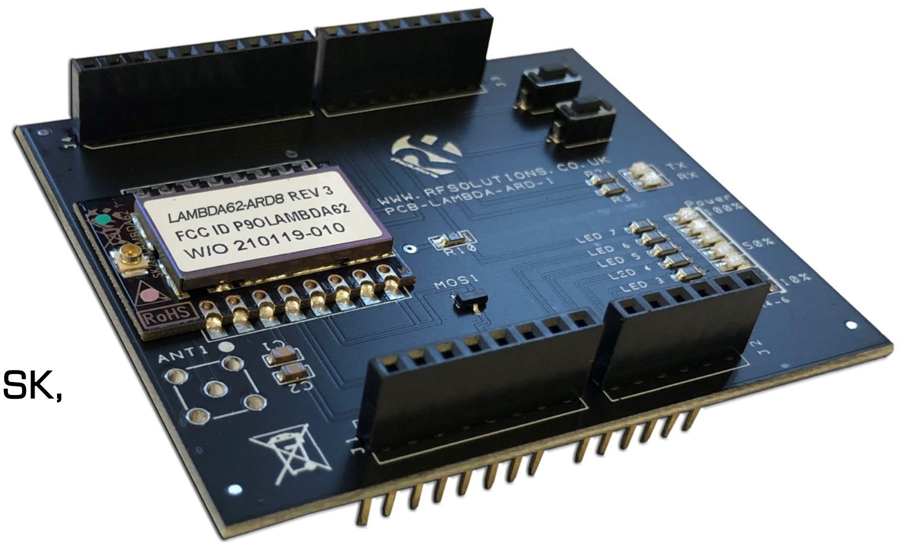
Arduino Uno Lambda62 Shield

Features

- LoRa™ Semtech SX1262
- Up to 20Km Range
- TX +22 dBm
- High Rx sensitivity: down to -148dBm
- Built in RF Switch
- 868/915 MHz Versions
- 100mm PCB antenna
- LoRa, FSK, GFSK, MSK, GMSK, OOK modulation
- Preamble detection
- CE Compliant
- Stackable shield design
- Example Code Available



Description

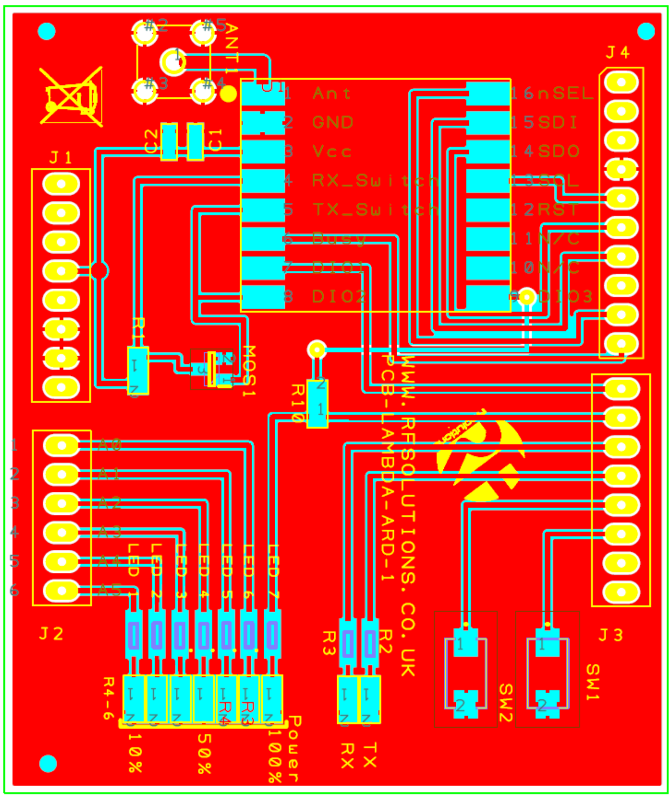
An Arduino Uno RF shield based on the LAMBD62 RF Module.
This Shield allows for the easy addition of RF to your Arduino project. This shield allows for long transmission distances, with a wide range of encoding options.
There are 2 onboard user configurable switches and the option of an LED array to display the RF signal strength.

Ordering Information

Part Number	Description
LAMBDA62-ARD8	Arduino UNO Development shield, 868MHz

Lambda62 Arduino Shield

Overview



Arduino Pin	Shield Pin	Connector/Pin
0	N/C	J3-8
1	N/C	J3-7
2 (Input)	SW1	J3-6
3 (Input)	SW2	J3-5
4 (Output)	LED TX	J3-4
5 (Output)	LED RX	J3-3
6 (Output/Input)	LED 7 / DIO3	J3-2
7	DIO1	J3-1
8	Busy	J4-10
9	nSEL	J4-9
10	N/C	J4-8
11	SDI	J4-7
12	SDO	J4-6
13	SCL	J4-5
A0 (Output)	LED 6 / N/C	J2-1
A1 (Output)	LED 5 / N/C	J2-2
A2 (Output)	LED 4 / N/C	J2-3
A3 (Output)	LED 3 / N/C	J2-4
A4 (Output)	LED 2 / N/C	J2-5
A5 (Output)	LED 1 / N/C	J2-6

Available Resources

- Lambda62 Datasheet
- SX1262 Datasheet

A Build Pack is also available containing;

- Schematic
- Gerber's
- DesignSpark Project
- Example Code

This is available at www.rfsolutions.co.uk/downloads/Arduino.php

Application Notes

Transmitter and Receiver Example with two Lambda62 Shields

In this example two Lambda62 shields were used, one as a transmitter and one a receiver. The frequency used was 868 MHz, but is configurable to 915 MHz. The modulation used was FSK, however, they can also operate in LoRa mode. More information about different operating modes can be found in the SX1262 Datasheet. The Lambda62 modules communicate with the Arduino's via SPI. This communication allows for the Arduino to configure the Lambda62 module and transfer the data to be transmitted.

Step 1: Define Inputs and Outputs

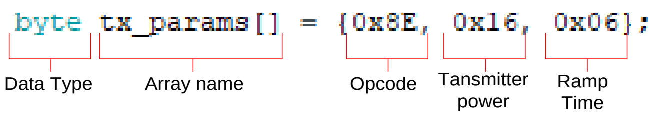
The first step is to define each of the pins connected from the Arduino to the Lambda62 module. The pin connections can be seen under the table above.

All the SPI pin connections are defined within the SPI.h file which needs to be included. Next the pins need to be set as either Inputs or outputs. For the on-board switches the internal pullup resistor needs to be activated, this is achieved by using:

```
pinMode(SW_1, INPUT_PULLUP);
```

Step 2: Define Lambda configuration settings

In order to define the configuration settings, first the structure of the command needs to be understood. From the statement below it can be seen that the command is stored as an array in byte format. The array can be broken down into two sections, the Opcode and the settings. This Opcode tells the Lambda62 module what the information it is about to receive relates to. For a full list of Opcodes and the settings associated with them please refer to the SX1262 datasheet.



```
byte tx_params[] = {0x8E, 0x16, 0x06};
```

For both transmitter and receiver the following settings need to be defined:

Standby mode: `byte standby[] = {0x80, 0x01};`

FSK Mode: `byte fsk[] = {0x8A, 0x00};`

Modulation Parameters: `byte modulation[] = {0x8B, 0x01, 0xA0, 0xAA, 0x00, 0x1A, 0x00, 0x7A, 0xE1};`

Packet parameters: `byte packet_params[] = {0x8C, 0x00, 0x20, 0x05, 0x10, 0x00, 0x00, 0x0D, 0x00, 0x00};`

Regulator Parameters: `byte regulator[] = {0x96, 0x00};`

Frequency Parameters: `byte frequency[] = {0x86, 0x36, 0x57, 0xE9, 0xBE};`

Image Calibration Parameters: `byte calib_image[] = {0x98, 0xD7, 0xDB};`

Power Amplifier Parameters: `byte pa_config[] = {0x95, 0x04, 0x07, 0x00, 0x01};`

DIO2 as RF Switch: `byte dio2[] = {0x9D, 0x01};`

Step 3: Define Transmitter Settings

To setup the transmitter the following settings/commands need to be defined for every transmission:

Transmitter parameters: `byte tx_params[] = {0x8E, 0x16, 0x06};`

Power Amplifier Parameters: `byte pa_config[] = {0x95, 0x04, 0x07, 0x00, 0x01};`

OCP Parameters: `byte ocp[] = {0x08, 0xE7, 0x38};`

Regulator Parameters: `byte regulator[] = {0x96, 0x00};`

Clear Device Errors command: `byte clear_deviceErrors[] = {0x07, 0x00};`

Clear IRQ command: `byte clear_IRQ[] = {0x02, 0xFF, 0xFF};`

Buffer Parameters: `byte buffer_base[] = {0x8F, 0x80, 0x00};`

Sync byte parameters: `byte set_sync[] = {0x0D, 0x06, 0xC0, 0x2D, 0xD4};`

IRQ Parameters: `byte irq_param[] = {0x08, 0x00, 0x02, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00};`

Step 4: Define Receiver Settings

To setup the receiver the following settings/commands need to be defined every time it is moved from standby to receive mode:

Clear Device Errors command: `byte clear_deviceErrors[] = {0x07, 0x00};`

Clear IRQ command: `byte clear_IRQ[] = {0x02, 0xFF, 0xFF};`

Buffer Parameters: `byte buffer_base[] = {0x8F, 0x80, 0x00};`

Sync byte parameters: `byte set_sync[] = {0x0D, 0x06, 0xC0, 0x2D, 0xD4};`

Receiver Gain: `byte rx_gain[] = {0x0D, 0x08, 0xAC, 0x96};`

IRQ Parameters: `byte irq_param[] = {0x08, 0x00, 0x02, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00};`

Set to Receiver Mode: `byte rx_mode[] = {0x82, 0x00, 0x00, 0x00};`

Step 5: Send configuration settings to Lambda62

These settings can then be sent to the Lambda62 module. To do this we first check to make sure the Lambda62 module is not busy. This is achieved by getting the state of the busy pin using the command “digitalRead(busy_pin)”. If the device is busy it will have a value of 1, if it is not busy then it will be 0.

To ensure that a command is not sent whilst the Lambda62 module is busy a while loop was used.

```
while (digitalRead(busy_pin) == 1) {
}
```

When the Lambda62 is no longer busy the Opcode and settings can be sent. This can be achieved by setting the chip select pin low and using a for loop as shown below:

```
digitalWrite(Chip_Select, LOW);
for (int i = 0; i < Size; i++) {
  SPI.transfer(Array[i]);
}
digitalWrite(Chip_Select, HIGH);
```

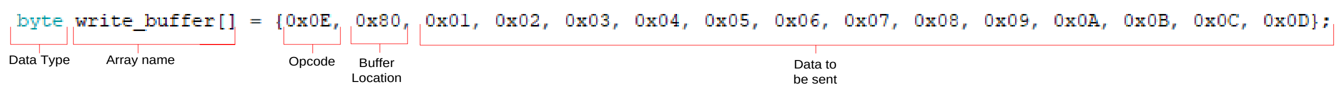
The for loop will send each byte of the array one after the other, for the length of the array. Once the entire array has been sent the chip select pin is set to high. This indicates to the Lambda62 module that there is no more information for that array. This needs to be repeated for each array in the order they have been shown previously.

Step 6: Transmit/Receive

Now that both the transmitter and receiver are setup and in the correct operating mode a packet can be sent/ received.

Transmitter:

To send a packet you first need to write the data to the buffer. This can be achieved by using the following array:



```
byte write_buffer[] = {0x0E, 0x80, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D};
```

The amount of bytes that you write to the buffer can be adjusted by changing the packet parameters for both receiver and transmitter, and adjusting the write to buffer packet to match. For this example it is sending 13 bytes containing 1-13 in decimal.

Once your data has been written to the buffer to lambda module needs to be set to Transmit mode. This is achieved by sending the array:

```
byte txmode[] = {0x83, 0x00, 0x00, 0x00};
```

This will send all the data stored within the buffer. Once complete the lambda module needs to be set into standby mode by sending the standby array.

```
byte standby[] = {0x80, 0x01};
```

Receiver:

When receiving the sent packet first we need for a valid packet to be detected. This can be achieved by monitoring DIO3. When this pin goes high it indicates a valid packet has been received.

Once a valid packet has been detected the buffer can be read. To do this the following section of code was used:

```
digitalWrite(Chip_Select, LOW);
SPI.transfer(read_buff);
for (int i = 0; i < 15; i++) {
  byte1 = SPI.transfer(0x00);
}
digitalWrite(Chip_Select, HIGH);
```

The first step is to send the chip_ select pin low to indicate to the lambda module you are about to send a command. Next the command "read_buff" is sent. This is the Opcode for reading the buffer, which is "0x1E".

To read the data in the buffer you then need to send a "blank" byte and store the returned value in a variable. For this example we need to read 15 bytes from the buffer as this will also read the preamble which is 2 bytes long.

Once you have finished reading the buffer the chip select pin needs to be set low, to indicate you have finished.

The Lambda62 now needs to be put back to standby mode using the same array as before.