Study text

Arduino WSN

Lukáš Němec

Contents

1 Introduction

The purpose of this text is to provide a general introduction to Arduino and applications which can be made using Arduino platform. Focus of a second and third part will be at wireless communication and making wireless sensor networks with Arduino based nodes. All used source codes can be found at Edu-Hoc project home, `https://github.com/crocs-muni/Edu-hoc`; at official Arduino website, `www.arduino.cc`; or at JeeLib library website, `http://jeelabs.net/projects/jeelib/wiki`.

2 Arduino - general view

Arduino is an open-source platform and also the phenomenon of last few years. It offers hardware itself and also software for programming ATMega microcontrollers. Thre are not only official Arduino boards but also a large number of clones and derivatives. These are motivated either by cheaper price or by added functionality compared to official ones. The first category typically includes Chinese clones like Funduino or others, while the second category consists usually of specialized boards like JeeLink and JeeNode USB, which we are going to use.

From all containing Arduino platform, we will use only development environment because our focus will be on ad-hoc networks, we will use specialized boards with a build-in radio module as was mentioned earlier in the text. Nevertheless, we will start with absolute basics of working with Arduino.

2.1 Arduino IDE

Arduino IDE is a very simple environment for development programs, from a practical point of view, it is just a little bit tweaked text editor with added support for Arduino. It supports all major OS, so you can run it on Linux, Windows or OS X. It can run without installation, but this option does not support communication with boards over a serial port. So this option is useful for programming, but if you want to send a program to board itself or communicate with it, then it is highly recommended to install Arduino IDE on your computer.

You can benefit from this environment also if you decide to use your favorite text editor for actual coding, then Arduino IDE could be used just for compiling the code or because of other tools it offers.

Installation Installation is not difficult at all. Many Linux distributions will ease the process because Arduino IDE package is usually present in the system repositories. The only thing you should care about is the version of such package, ideally, it should be 1.5 or higher.

In case of Windows OS, recommended approach is to download installation files directly from official web page `http://arduino.cc/en/Main/Software` where you will find

executable files, which will either install Arduino IDE on your machine, or just run Arduino IDE itself (in case you do not have access to administrator account on machine you are working).

If for whatever reason you need to install Arduino IDE some other way (e.g. directly from source code), then all the information you should need is again accessible on official web page `http://arduino.cc/en/Main/Software`.

Tools

Verification One of the most important tools which environment offers is verifier, which sole purpose is to check for syntax errors of your code. This option is unfortunately not real time, so it is more than recommended to check your syntax once a while.

Compilation The second tool is a compiler, which also takes care of upload to the board. Fist you are required to select your version of Arduino board, in some cases board which is the most similar to the one you have (especially important if you have clones or various derivates). Next you should check the port setting and set the port which is the board connected to. In a case of only one board connected, IDE usually selects the right port autonomously, but it is recommended to check it. It is also possible to connect multiple boards at once and then you need to select the right one.

In a case of issues with detection of the board, it is recommended to restart the whole IDE and try again. Also, it is good practice to connect the board before you start the IDE.

Serial monitor Serial monitor is last of the important tools which are useful to know. Although its functionality can be substituted bu any program, which allows communication over a serial port (e.g. PySerial), on the other hand, Serial Monitor is the most accessible tool you have.

After start, you are required to set up communication frequency (same value as in source code) and then you can read information send from the board or send instructions to the board.

If you can see characters, which make no sense, it is more than likely, that it is an issue with different setup of frequency on board and Serial Monitor.

2.2 Arduino Code basics

Arduino can be seen as special case of C language. Almost everything is the same, only thing changed is `main` function, which is divided into two parts `setup` and `loop`. Therefore, you are able to write cycles same as in C language, same for conditions, functions and everything else you might ever need. In addition to this, Arduino adds it's own functions and a large amount of libraries, which are usually targeted to accessory hardware. You can also find many other libraries which make writing the code in C less painful.

`setup` As the name suggests, this function is mainly for setup. It is run after the board is powered on and only once. It takes care of initialization of everything you might need. Thus, it is ideal for setup of serial port communication, pin setup and initialization of libraries.

`loop` A function which contains the main part of the program and this function can be seen as an infinite cycle. It is called after `setup` function, so we can count on all being initialized. This function will loop until the board is disconnected from power or restarted. It is good practice to write code in this function with respect to infinite loop and also if you need regular check of some event (button pressed) it is recommended to either have code so fast, that checking once a cycle is short enough time frame or run check several times in one run of cycle.[1]

Serial One of the most important libraries you will ever encounter. It manages serial port on the side of the board. First you have to initialize it in `setup` function, together with frequency setting. Then you can start using it, even immediately after initialization in `setup` function. All functions from this library has to be called as `Serial.read()` not just `read()`.

`print` Basic function for sending data, debugging values and text over a serial port. It is possible to print value itself of format as it is more convenient (e.g. hexadecimal values). Also, it is possible to use `println`, which in addition sends new line character. When using this function you have to remember, that this function is asynchronous, therefore it sends a return value before it starts to send characters over a serial port.

You can also use `write` (for writing binary data) or `flush` (waits for completion of writing).

`read` Reads data from serial port and returns read value. It is convenient to use together with `available` function, that returns a number of available bytes on a serial port. You can use this function for sending commands to the board or initial setup, which cannot be hard coded.

3 WSN node, JeeLib

Now we look away from general purpose Arduino and aim our focus on wireless communication. We can connect radio module to Arduino board and then we get one node of a wireless network. Or we can take a derivate of Arduino which has radio module already

[1] some boards have dedicated pins for interrupts, which can step into `loop` function and provide immediate event handling. You can connect special function to the interrupt pin, which takes care of interrupt service

integrated. The radio is the most important part, we do not care much about large amount of pins for other additional components.

One node has not much of a use, it's strength is in large quantities. Nevertheless, it is recommended to get familiarized with just two node network and advanced applications with large amounts of nodes leave for later. With one node you can try communication with computer, use functions from common Arduino libraries and experiment with serial port communication for initial setup.

With two nodes you can experiment with basic radio communication, send, receive messages, and experiment with other functions from JeeLib library.

3.1 JeeNode HW

JeeNode US and JeeLink are Arduino clones; in addition they come with radio module. In contrast to the most common Arduino boards; which are 5V powered; these run on only 3.3V and their components layout is quite a bit different, thus you cannot connect majority of common Arduino accessories (i.e. expansion boards). On top of that, JeeLink is protected with plastic cover, so you cannot really connect anything.

The difference between JeeNode USB and JeeLink is in already mentioned plastic cover and in used USB connector. While JeeNode USB uses mini USB connector, JeeLink uses much more convenient USB type A, which you can connect directly to your computer.

Both versions are based on Arduino mini with ATmega328P processor.[2] JeeLink also contains 16 Mbit of flash memmory, witch can be used for storing application data.

Radio on the board can communicate on three different frequencies, these are 433, 868 or 915 MHz. Frequency is important mainly because of antenna, which can be adjusted for the best performance by selecting the right length for selected frequency, as they are dependent. Although it is important, for most applications you do not need to worry much about it, it will be fine as it is. JeeLink has frequency set during manufacturing, you can recognize these from color marking on the radio chip. Yellow color means 868 MHz and green color means 433 MHz.[3]

Pins These are contacts, which can be used to connect additional components to the Arduino. It can be anything from simple LED up to light or gravity sensor. These can be connected only to the JeeNode USB.

3.2 JeeLib

JeeLib is an Arduino library aimed specifically for JeeLink and JeeNode USB nodes and other compatible hardware. It allows us to control radio and few other modules, more information about it can be found directly on project website or from documentation for the library, which you can obtain in various ways, either generate it yourselves directly

[2]This is the board you have to select in the Arduino IDE, if you wish to upload code

[3]HW used for lab testbed has frequency set to 868 MHz

from source code you are using with Doxygen tool, or use official documentation available on JeeLabs website `http://jeelabs.net/pub/docs/jeelib/`.

From the content of JeeLib is radio communication the most important part and we can find it under RF12 controller `http://jeelabs.net/pub/docs/jeelib/md_intro_rf12.html`. It contains all necessary from initial node setup, message sending and receiving, up to the simple encryption mode.

How to add to IDE JeeLib requires Arduino IDE version 1.5 and higher, it is not compatible with older versions. It this condition is satisfied, the addition to the IDE is few simple clicks. Fist we have to obtain the library itself, ideally directly from a webpage of JeeLabs `http://jeelabs.net/projects/jeelib/wiki`.

In the menu we can then find **Sketch** option, Under it is option to **Import Library** and then we select **Manage Libraries** (in newer versions of IDE), or **Add Library** (for older versions). Then it is just a simple task of finding a folder with our downloaded library and its addition. Library can be also added directly in the form of ZIP archive.

Header format The header is first 8 bits of each message. It contains a type of the message and its sender or recipient. First, three bits determine the type of the message, it can either mean if an acknowledgment is required and then distinguishment between unicast and broadcast message. Remaining five bits determine ID of the node, it can either be a recipient, if a node is sending this message, or sender, if node received this message (the change is done by radio driver in the moment of sending the message).

We have only 5 bits for ID of the node, therefore, we can assign only 32 different node IDs. We assign them in the range 0 to 31 and border values are special dedicated ID. ID 0 is universal for sending out broadcast and a node with ID 31 will receive all messages in the network, without distinguishing its recipient.

Send unicast message In order to send the message, you fist need to make sure that no other node is transmitting since only one node can transmit a message at given time. JeeLib library makes this relatively easy because it itself deals with access control. It is sufficient to call `rf12_canSend()` and result is boolean variable, if we can begin trasmission or not.

After that we can send message with function `rf12_sendStart()` with its parameters (header, payload and payload length). More details to both functions can be found in the documentation. In case of simple application, where we expect small ammount of traffic, we can use `rf12_canSend()` in cycle, until the result is positive. Moreover we can simplify this with function `rf12_sendNow()` which does exactly the same. Thus it contains waiting cycle and then it sends the message. Parameters are the same as for `rf12_sendStart()`.

Receive message In order to receive a message, we have to periodically (in short intervals) ask with function `rf12_recvDone()`, which returns a positive answer, in a case of an incoming message. However, if a node receives two or more messages in the interval, then

the node is able to process only the last one because message properties are stored in the global variables.

If the message is received, it is recomended to check for required acknowledgment, and if it is present, then the ideal solution is following condition:

Example condition for ACK checking

```
1           if(RF12_WANTS_ACK){
2               f12_sendStart(RF12_ACK_REPLY,0,0);
3           }
```

Network with two motes With two motes we have to periodically check, whether any message was received and periodically send own messages, or wait for external command (e.g. mote can send a message as a reaction to serial port command). As an example, we can use application from JeeLib library, in particular, RF12demo, which allows control almost all node and radio functions (including message send) with serial port commands.

It is sufficient to connect two motes and to bost of them upload RF12demo application and then use Serial Monitor to post commands to bost of the motes. Ideally, it is a good idea to have both connection open at the same time so that you can see the output of both nodes at the same time.

Mote setup example as used in RF12demo

```
537         void setup () {
538             ...
539
540             Serial.begin(SERIAL_BAUD);
541             Serial.println();
542             displayVersion();
543
544             if (rf12_configSilent()) {
545                 loadConfig();
546             } else {
547                 memset(&config, 0, sizeof config);
548                 config.nodeId = 0x81;       // 868 MHz, node 1
549                 config.group = 0xD4;        // default group 212
550                 config.frequency_offset = 1600;
551                 config.quiet_mode = true;   // Default flags, quiet on
552                 saveConfig();
553                 rf12_configSilent();
554             }
555
556             rf12_configDump();
557             df_initialize();
558             ...
559         }
```

In the example of `setup` function you can see initial setup of a serial port, where we can set up the frequency to predefined constant. Then we check for saved configuration

and if there is such thing saved, then we load configuration from memory. If the values had not been saved yet, then they are set up manually and then saved to memory. This is not necessary for a small amount of nodes (which are easy to set up manually each time), but for larger networks it is recommended to save configurations and then speed up later changes in the network.

Simplified example of message receive in RF12demo

```
582         if (rf12_recvDone()) {
583             byte n = rf12_len;
584             if (rf12_crc == 0)
585                 showString(PSTR("OK"));
586             else {
587                 if (config.quiet_mode)
588                     return;
589                 showString(PSTR(" ?"));
590                 if (n > 20) // print at most 20 bytes if crc is wrong
591                     n = 20;
592             }
593             ...
594             printOneChar(' ');
595             showByte(rf12_hdr);
596             for (byte i = 0; i < n; ++i) {
597                 printOneChar(' ');
598                 showByte(rf12_data[i]);
599             }
600 
601             Serial.println();
602             ...
603         }
```

In the example, we can see how the message receive is actually done in the application. First, the condition than message was received has to be satisfied. After that we check the CRC value, to eliminate any errors in the transmission, if the CRC check passes, then we can continue in message processing, otherwise, we indicate an error and write out first 20 bytes for debugging purposes. In the case of correct message, we write out all contents of the message. In the example is left out hexadecimal output and signalization with LED. If you happen to be interested in the complete code, you can find it directly in the application code (line numbers should match so that it is easier to find).

Simplified example of message send in RF12demo

```
641         if (cmd && rf12_canSend()) {
642             activityLed(1);
643 
644             showString(PSTR(" -> "));
645             Serial.print((word) sendLen);
646             showString(PSTR(" b\n"));
647             byte header = cmd == 'a' ? RF12_HDR_ACK : 0;
648             if (dest)
649                 header |= RF12_HDR_DST | dest;
650             rf12_sendStart(header, stack, sendLen);
651             cmd = 0;
652 
653             activityLed(0);
654         }
```

In the example, we can see initial condition (only one mote can transmit at the time). Follows the indication with LED and sending the message itself, including writing out the information about a message over a serial port.

These two examples show almost all basic functionality of the application (contents of the `loop` function), without any description are additional functions which deal with parsing received commands. These are basically plain C language, or very similar to it, so there is no need to explain it here. Also, most of the advanced functions are left without any description since their functionality is not intended for this introductory study text. In the case of interest, you can always find more in the official documentation for JeeLib library `http://jeelabs.net/pub/docs/jeelib/`.

4 WSN network

Wireless sensor network is composed of a large number of small and autonomous devices, which communicate one with another over a radio. Every device, called sensor mote, is battery powered and equipped with all the necessary sensors for monitoring the environment. A typical application of such network is a random placement of thousands of motes and following a collection of data from sensors. As an example of fields, where such approach is applicable, we can mention military, agriculture and health applications and much more.

4.1 Sample applications

We have two sample applications, which can be used to run such network. First, one periodically sends data (counter) from each node, while the other does not communicate at all, but has a rather special purpose. It is a sniffer, which allows to one mote to receive all messages in the network, which is useful for debugging own applications and also for basic recognition of network we know nothing about.

Alive Application with static network topology, every node has fixed ID and parent ID. Each message is sent to the parent node and in this manner is message routed through the whole network to the central node (base station). With such simple fixed routing tree we can achieve multihop communication, therefore, cover much longer distances compared to the radio range of the single node.

For correct operation of this simple routing, we have to take care, that every node has its parent within radio range, otherwise, we could not route some of the messages to their destination. On the other hand, we have to design the routing tree balanced[4], because we do not want overload some motes more than others.

The application itself is very simple, important parts of the code are described in the following example of `loop` function.

`loop` function in Alive application

```
26                  void loop () {
27                    //if incoming message received
28
29                    if(rf12_recvDone()){
30                      if(RF12_WANTS_ACK){
31                        rf12_sendStart(RF12_ACK_REPLY,0,0);
32                      }
33
34                      if(rf12_crc == 0){ //packet checksum is correct
35                        //propagate to parent
36                        byte header = B00000000;
37                        //fill header using radioUtils
38                        ru.resetAck(&header);
39                        ru.setID(&header, parent);
40                        rf12_sendNow(header, (const void*)rf12_data, rf12_len);
41                      }
42                    }
43                    delay(10);
44                    counter++;
45
46                    if(counter%100 == 0){
47                      msgCounter++;
48                      //send still alive msg
49                      byte header;
50                      //fill header using radioUtils
51                      ru.resetAck(&header);
52                      ru.setID(&header, parent);
53                      rf12_sendNow(header, (const void*) &msgCounter, sizeof(msgCounter));
54                      counter = 0;
55                    }
56
57                  }
```

[4]It is not always possible to make the tree balanced, but we should try to make it as balanced as we can

In the `loop` function we can see first see the ACK check so that the sender has information about correct transmission. Then there is CRC check and in a case of correct CRC check, there is the imediate reaction to the message, which means forwarding such message to the node parent.

Follows short delay, so that the node is overloaded, and incrementation of the counter. The counter makes sure, that the loop function goes around much more times (to check received messages) but only once per 100 revolutions it sends message itself. Ideal is, when mote can receive message anytime, but sending the message should be slowed down, so we need to perform it only once per certain number of cycles.

Sending the message itself is simple task, we use radioUtils library for setting up the recipient of the message and all other necessary parameters. After then the message is send using `rf12_sendNow` function. In case of this application we can afford active waiting while the message is being send, because we do not expect heavy load on the node or network. In different case we should combine this waiting with checks on received messages, because we do not want to block mote from any other activity (using sensors, receiving messages ...) just because it is waiting to send a message.

Sniffer Second application is capable of capturing any traffic in the network. It will be of a great help, if we need to debug our own application, when we need to monitor what is which mote transmitting and when. The same application can be without any modification used in the role of attacker and very easily passively eavesdrop any communication in the network. Only limitation is the radio range. Application uses little bit more advanced approach because we suppose usage for heavy load and large networks. In the `setup` function we set the ID of mote to 31 (mote which can receive all messages in the network). This is the most important part, other is just message processing and serial output. Sniffer itself is special application, because it does not send any messages.

`loop` function in the Sniffer application

```
32          void loop () {
33
34              if (rf12_recvDone()) {
35                  // quickly save a copy of all volatile data saveLen = rf12_len;
36                  saveCrc = rf12_crc;
37                  saveHdr = rf12_hdr;
38                  memcpy(saveData, (const void*) rf12_data, sizeof saveData);
39                  rf12_recvDone();
40                  // release lock on info for next reception
41                  if (saveCrc != 0) {
42                      su.print("CRC error #", output);
43                      su.println(saveLen, DEC, output);
44                  }
45                  else {
46                      su.print("OK (", output);
47                      String header = "";
48                      for (byte i = 0; i < 8; ++i) {
49                          header.concat(bitRead(saveHdr, 7-i)); // read bytes of header
50                      }
51                      su.print(header, output);
52                      su.print("b) ", output);
53                      su.print(saveHdr & RF12_HDR_ACK ? "REQ " : " ", output);
54                      su.print(saveHdr & RF12_HDR_CTL ? "ACK " : " ", output);
55                      su.print(saveHdr & RF12_HDR_DST ? "DST:" : "SRC:", output);
56                      su.print(saveHdr & RF12_HDR_MASK, output);
57                      su.print(" #", output);
58                      su.println(saveLen,DEC, output);
59                      // print out all data bytes, wrapping long lines byte pos = 0;
60                      int pos = 0;
61                      for (byte i = 0; i < saveLen; ++i) {
62                          su.print(' ', output);
63                          su.print(saveData[i],HEX, output);
64                          pos += 2;
65                          if (saveData[i] >= 16) ++pos;
66                          if (pos > 75) {
67                              su.println();
68                              pos = 0;
69                          }
70                      }
71                      if (pos > 0) su.println();
72                  }
73              }
74          }
```

We can again see check after the message is received, but because this message was not meant for this node, we do not send any acknowledgments. Because we care about the fast processing of received messages, we first copy the contents of the message to local variables and we allow the radio to receive a new message as soon as possible. After that, we can check CRC value (cyclic redundancy check) and then we process the message itself.

We can see, that the header is processed bit after bit so that we can see all 8 bits individually. After we process all other parts of the message. We can see, that we do not use a usual function for serial output, but our own from the serialUtils library, because it allows us to set the priority for individual outputs and for example leave out debugging outputs when we run the application later.