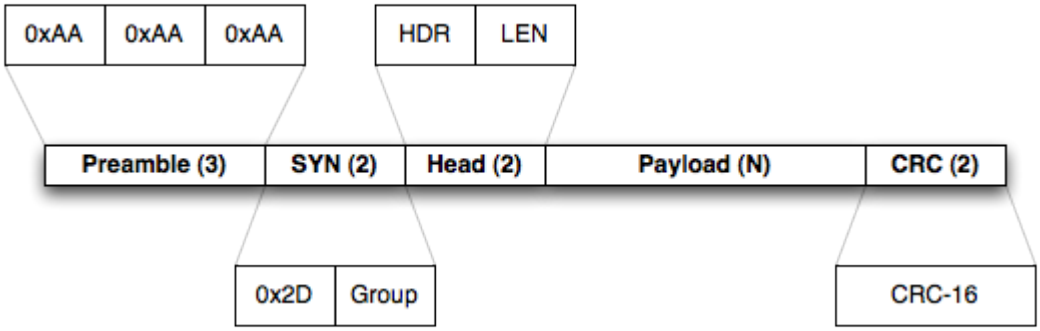
RF12 packet format and design

In Software on **Jun 9, 2011** at **00:01**

The RF12 library contains the code to support the RFM12B wireless module in an Arduino-like environment. It's used for JeeNodes but also in several projects by others.

Here's the general structure of a packet, as supported by the RF12 driver:



I made quite a few design decisions in the RF12 driver. One of the goals was to make the communication work in the background, so the driver is fully interrupt-driven. An important choice was to limit the packet size to 66 bytes of payload, to keep RAM use low and still allow just over 64 bytes of payload, enough to send data in 64-byte chunks with a *teeny* bit of additional info.

Another major design decision was to support absolutely minimal packet sizes. This directly affects the power consumption, because longer packets take more time, and the longer the receiver and transmitter are on, the more precious battery power they will consume. For this same reason, the transmission bit rate is set fairly high (about 50 kbits/sec) – a higher rate means the same message can be sent in less time. A higher rate also makes it harder for the receiver to still pick up a good packet, so I didn't want to push this to the limit.

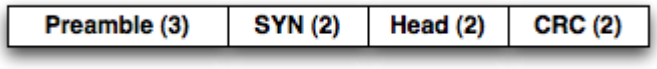
This is how the RF12 driver can support really short packets:

- the network group is one byte and also doubles as second SYN byte
- the node ID is small enough (5 bits) to allow a few more header bits in the same byte
- there are only three header bits, as described in more detail in tomorrow's post
- there is only room for *either* the source node ID *or* the destination node ID

That last decision is a bit unusual. It means an incoming packet can *only* inform the receiver where it came from, *or* define which receiver the packet is intended for – not both.

This may seem like a severe limitation, but it really isn't: just add the missing info in the payload and agree on a convention so that the receiver can pick it up. All the RF12 does is enforce a truly minimal design, you can add any info you like as payload.

As a result, a minimal packet has the following format:



That's 9 bytes, i.e. 72 bits – which means that a complete (but empty) packet can be sent out in less than 1.5 ms.

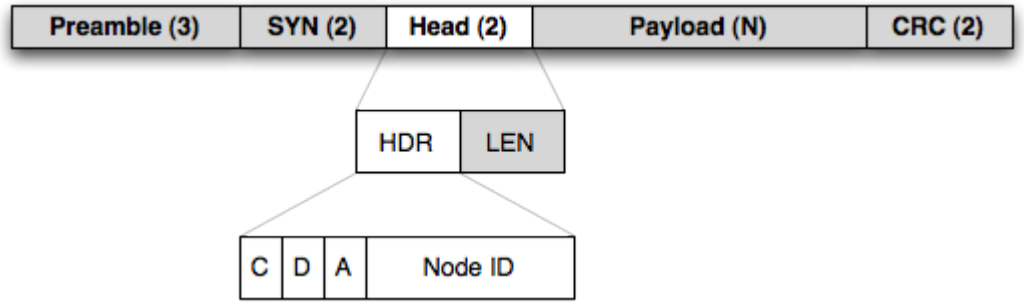
Tomorrow, I'll describe the exact logic used in the HDR byte, and how to use broadcasts and ACKs.

In yesterday's post, the general design of the RF12 driver was presented, and the format of the packets it supports.

The driver also support broadcasting, i.e. sending packets to all interested nodes, and ACKs, i.e. sending a short "acknowledge" packet from receiver to transmitter to let the latter know that the packet was properly received.

Broadcasting and ACKs can be combined, with some care: only one node should send back the ACK, so the usefulness of ACKs with broadcasts is limited if the goal was to reliably get a packet across to multiple listeners.

Broadcasts and ACKs use the HDR byte in each packet:



There are three bits: C = CTL, D = DST, and A = ACK, and there is a 5-bit node ID. Node ID 0 and 31 are special, so there can be 30 different nodes in the same net group.

The A bit (ACK) indicates whether this packet wants to *get* an ACK back. The C bit needs to be zero in this case (the name is somewhat confusing).

The D bit (DST) indicates whether the node ID specifies the destination node or the source node. For packets sent to a specific node, DST = 1. For broadcasts, DST = 0, in which case the node ID refers to the originating node.

The C bit (CTL) is used to send ACKs, and in turn must be combined with the A bit set to zero.

To summarize, the following combinations are used:

- normal packet, no ACK requested: CTL = 0, ACK = 0
- normal packet, wants ACK: CTL = 0, ACK = 1
- ACK reply packet: CTL = 1, ACK = 0
- the CTL = 1, ACK = 1 combination is not currently used

In each of these cases, the DST bit can be either 0 or 1. When packets are received with DST set to 1, then the receiving node has no other way to send ACKs back than using broadcasts. This is not really a problem, because the node receiving the ACK can check that it was sent by the proper node. Also, since ACKs are always sent immediately, each node can easily ignore an incoming ACK if it didn't send a packet shortly before.

Note that both outgoing packets and ACKs can contain payload data, although ACKs are often sent without any further data. Another point to make, is that broadcasts are essentially free: every node will

get every packet (in the same group) anyway – it's just that the driver filters out the ones not intended for it. A recent RF12 driver change: node 31 is now special, in that it will see packets sent to *any* node ID's, not just its own.

It turns out that for Wireless Sensor Networks, broadcasts are quite useful. You just kick packets into the air, in the hope that someone will pick them up. Often, the remote nodes don't really care *who* picked them up. For important events, a remote node can choose to request an ACK. In that case, one central node should always be listening and send ACKs back when requested. An older design of the Room Node sketch failed to deal with the case where the central node would be missing, off, or out of range, and would retry very often – quickly draining its own battery as a result. The latest code reduces the rate at which it resends an ACK, and stops asking for ACKs after 8 attempts. The next time an important event needs to be sent again, this process then repeats.

Nodes, Addresses, and Interference

In Software on **Jan 14, 2011** at **00:01**

The RF12 driver used for the RFM12B module on JeeNodes makes a bunch of assumptions and has a number of fixed design decisions built-in.

Here are a couple of obvious ones:

- nodes can only talk to each other if they use the same "net group" (1..250)
- nodes normally each have a unique ID in that netgroup (1..31)
- packets must be 0..66 bytes long
- packets need an extra 9 bytes of overhead, including the preamble
- data is sent at approximately 50,000 baud
- each byte takes ≈ 160 µs, i.e. a max-size packet can be sent in 12 milliseconds

So in the limiting case you could have up to 7,500 different nodes, as long as you keep in mind that they have to share the *same* frequency and therefore should never transmit at the same time.

For simple signaling purposes that's plenty, but it's obvious that you can't keep a serious high-speed datastream going this way, let alone multiple data streams, audio, or video.

On the 433 or 868 MHz bands, the situation is often worse than that – sometimes *much* worse, because simple OOK (which is a simple version of ASK) transmitters tend to completely monopolize those same frequency bands, and more often than not, they don't even wait for their turn so they also disturb transmissions which are *already in progress!* Add to that the fact that OOK transmitters often operate at 1000 baud or less, and tend to repeat their packets a number of times, and you can see how that "cheap" sensor you just installed could mess up everything!

So if you've got a bunch of wireless weather sensors, alarm sensors, or remotely controlled switches, chances are that your RF12-based transmissions will frequently fail to reach their intended destination.

Which is why "ACKs" are so important. These make it possible to *detect* when packets get damaged or fail to arrive altogehter. An ACK is just what the name says: an *acknowledgement* that the receiver got a proper packet. No more no less. And the implementation is equally simple, at least in concept: an ACK is nothing but a little packet, sent the other way, i.e. back from the receiver to the original transmitter.

With ACKs, transmitters have a way to find out whether their packet arrived properly. What they do is

send out the packet, and then wait for a valid reply packet. Such an "ACK packet" need not contain any payload data – it just needs to be verifiably correct (using a checksum), and the transmitter must somehow be able to tell that the ACK indeed refers to *its* original packet.

And this is where the RF12 driver starts to make a number of not-so-obvious (and in some cases even unconventional) design decisions.

I have to point out that wireless communication is a bit different from its wired counterpart. For one, *everyone can listen in*. Radio waves don't *aim*, they reach all nodes (unless the nodes are at the limit of the RF range). So in fact, each transmission is a *broadcast*. Whether a receiver picks up a transmitted packet is only a matter of whether it decides to let it through.

This is reflected in the design of the RF12 driver. At the time, I was trying to address both cases: broadcasts, aimed at anyone who cares to listen, and directed transmissions which target a specific node. The former is accomplished by sending to pseudo node ID zero, the latter requires passing the "destination" node ID as first argument to rf12_sendStart().

For the ACK, we need to send a packet the other way. The usual way to do this, is to include both source and destination node ID's in the packet. The receiver then swaps those fields and voilá… a packet ready to go the other way!

But that's in fact overkill. All we really need is a *single bit*, saying the packet is an ACK packet. And in the simplest case, we could avoid even that one bit by using the convention that data packets *must* have one or more bytes of data, whereas ACKs *may not* contain any data.

This is a bit restrictive though, so instead I chose to re-use a single field for either source or destination ID, plus a bit indicating which of those it is, plus a bit indicating that the packet is an ACK.

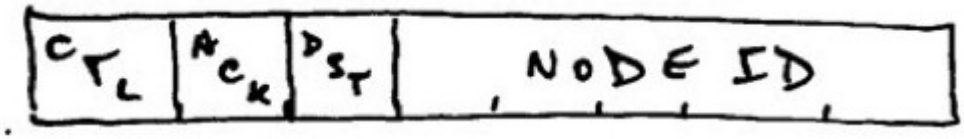
With node ID's in the range 1..31, we can encode the address as 5 bits. Plus the src-vs-dest bit, plus the ACK bit. Makes seven bits.

Why this extreme frugality and trying to save bits? Well, keep in mind that the main use of these nodes is for battery-powered Wireless Sensor Networks (WSN), so reducing power usage is normally one of the most important design goals. It may not seem like much, but one byte less to send in an (empty) ACK packet reduces the packet length by 10%. Since the transmitter is a power hog, that translates to 10% less power needed to send an ACK. *Yes, every little bit helps – literally!*

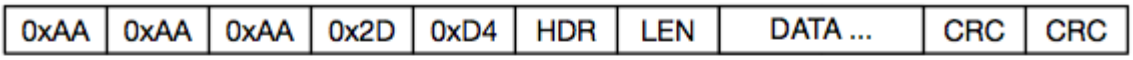
That leaves one unused bit in the header, BTW. Whee! :)

I'm not using that spare bit right now, but it will become important in the future to help filter out duplicate packets (a 1-bit sequence "number").

So here is the format of the "header byte" included in each RF12 packet:



And for completeness, here is the complete set of bytes sent out:



- 0xAA = preamble
- 0x2D = sync
- 0xD4 = net group (RFM12: fixed, RFM12B: adjustable)
- HDR = src/dst/ack packet header
- LEN = 0 .. 66
- DATA = up to 66 bytes of payload
- CRC = CRC16, little-endian

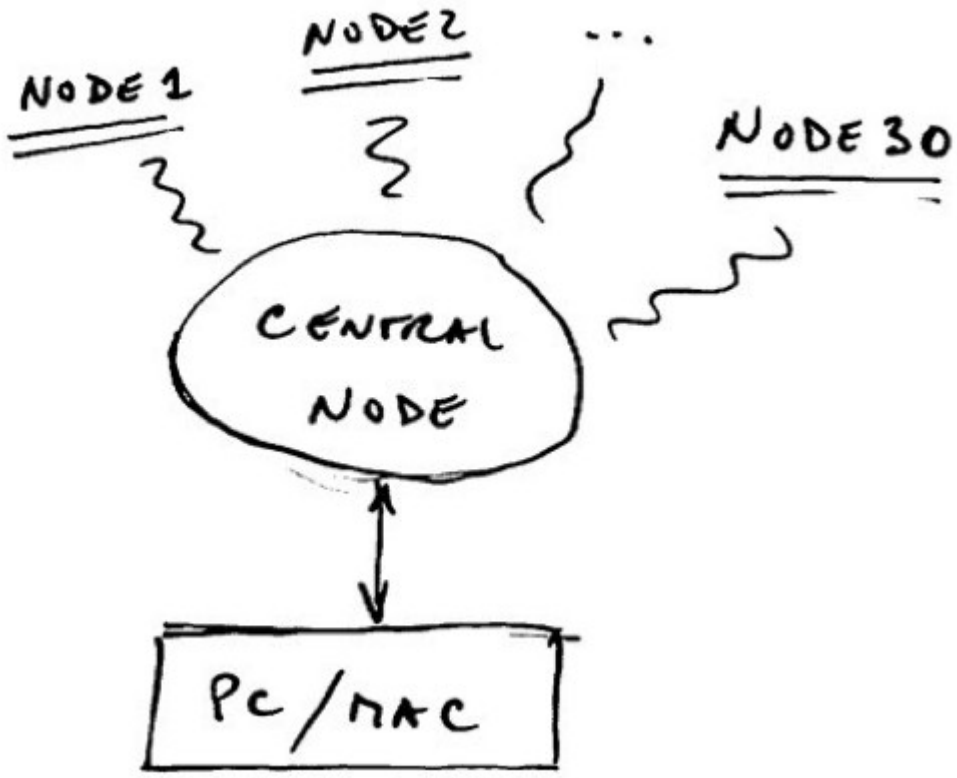
So what are the implications of not having both source and destination address in each packet?

One advantage of using a broadcast model, is that you don't have to know where to send your packet to. This can be pretty convenient for sensor nodes which don't really care *who* picks up their readings. In some cases, you don't even care whether the data arrived, because new readings are periodically being sent anyway. This is the case for the Room Nodes, when they send out temperature / humidity / light-level readings. Lost one? Who cares, another one will come in soon enough.

With the PIR motion detector on Room Nodes, we *do* want to get immediate reporting, especially if it's the first time that motion is being detected. So in this case, the Room Node code is set up to send out a packet *and* request an ACK. If one doesn't come in very soon, the packet is sent again, and so on. This repeats a few times, so that motion detection packets reach their destination as quickly as possible. Of course, this being wireless, there are no guarantees: someone could be *jamming* the RF frequency band, for example. But at least we now have a node which tries very hard to quickly overcome an occasional lost packet.

All we need for broadcasts to work with ACKs, is that exactly *one* node in the same netgroup acts as receiver and sends out an ACK when it gets a packet which asks to get an ACK back. We do not want more than one node doing so, because then ACKs would come from different nodes at the same time and interfere with each other.

So normally, a WSN based on RFM12B's looks like this:



The central node is the one sending back ACKs when requested. The other nodes should just ignore everything not intended for them, including broadcasts.

Note that it is possible to use more than one receiving node. The trick is to still use only a single one to produce the ACKs. If you're using the RF12demo sketch as central receiver, then there is a convenient (but badly-named) "collect" option to disable ACK replies. Just give "1c" as command to the second node, and it'll stop automatically sending out ACKs ("0c" re-enables normal ACK behavior). In such a "lurking" mode, you can have as many extra nodes listening in on the same netgroup as you like.

To get back to netgroups: these really act as a way to partition the network into different groups of nodes. Nodes only communicate with other nodes *in the same netgroup*. Nodes in other netgroups are unreachable, and data from those other nodes cannot be received (unless you set up a relay, as described a few days ago). If you want to have say hundreds of nodes all reporting to one central server, then one way to do it with RF12 is to set up a number of separate netgroups, each with one central receiving node (taking care of ACKs for that netgroup), and then collect the data coming from all the "central nodes", either via USB, Ethernet, or whatever other mechanism you choose. This ought to provide plenty of leeway for home-based WSN's and home-automation, which is what the RF12 was designed for.

So there you have it. There is a lot more to say about ACKs, payloads, and addressing… some other time.

Another topic worth a separate post, is using (slightly) different frequencies to allow multiple transmissions to take place at the same time. *Lots of things still left to explore, yummie!*

Binary packet decoding

In AVR, Software on **Dec 7, 2010** at **00:01**

The RF12 library used with the RFM12B wireless radio on JeeNodes is based on the principle of sending individual "packets" of data. I've described the reasons for this design choice in a number of posts.

Let me summarize what's going on with wireless:

- RFM12B-based nodes can send *binary* packets of 0..66 bytes
- these packets can contain any type of data you want
- a checksum detects transmission errors to let you ignore bad packets
- dealing with packet loss requires an ACK + re-transmission mechanism

Packets have the nice property that they either arrive intact *as a whole* or not at all. You won't get garbled or inter-mixed packets when multiple nodes happen to send at (nearly) the same time. Compare this to some other solutions where all the characters sent end up in one big "soup" if the sending happens (nearly) simultaneously.

But first: what's a packet?

Well, *loosely speaking*, you could say that a packet is like one line of text. In fact, that's exactly what you end up with when using the RF12demo sketch as central receiver: a line of text on the serial/USB connection for each received packet. Packets with valid checksums will be shown as lines starting with "OK", e.g.:

```
OK 3 128 192 1 0
OK 23 79 103 190 0
OK 3 129 192 1 0
OK 2 25 99 200 0
OK 3 130 192 1 0
OK 24 2 121 163 0
OK 5 86 97 201 0
OK 3 131 192 1 0
```

Let's examine how that corresponds with the actual *data* sent by the node.

- All the numbers are byte values, shown as numbers in the range 0..255.
- The first byte is a *header byte*, which usually includes the node ID of the sender, plus some extra info such as whether the sender expects an ACK back.
- The remaining data bytes are an exact copy of what was sent.

There appears to be some confusion about how to deal with the binary data in such packets, so let me go into it all in a bit more detail.

Let's start with a simple example – sending one byte:

```
byte measurement = 123;
rf12_sendData(0, &measurement, 1);
```

I'm leaving out tons of details, such as calling rf12_recvDone() and rf12_canSend() at the appropriate moments. This code is simply broadcasting one value as a packet for *anyone* who cares to listen (on the same frequency band and net group). Let's also assume this sender's node ID is 1.

Here's how RF12demo reports reception of this packet:

```
OK 1 123
```

Trivial, right? Now let's extend this a bit:

```
int measurement = 12345;
rf12_sendData(0, &measurement, sizeof measurement);
```

Two things changed:

- we're now sending a larger *int*, i.e. a 2-byte value
- instead of passing length 2, the compiler calculates it for us with the C "sizeof" keyword

Now, the incoming packet will be reported as:

```
OK 1 57 48
```

No "1", "2", "3", "4", or "5" in sight! What happened?

Welcome to the world of multi-byte values, as computers deal with them:

- a C "int" requires 2 bytes to represent
- bytes can only contain values 0..255
- 12345 will be "encoded" in two bytes as "12345 divided by 256″ and "12345 modulo 256″
- 12345 / 256 is 48 – this is the "upper" value (the top 8 bits)
- 12345 % 256 is 57 – this is the "lower" value (the low 8 bits)
- an ATmega stores values in little-endian format, i.e. lowest-range bytes come first
- hence, as *bytes*, the int "12345″ is represent as first 57 and then 48
- and sure enough, that's exactly what we got back from RF12demo

Yeah, ok, but why should we care about such details?

Indeed, on normal PC's (desktop and mobile) we rarely need to. We just think in terms of our numbering system and let the computer do the conversions to and from text for us. That's exactly what "Serial.print(12345)" does under the hood, even on an Arduino or a JeeNode. Keep in mind that "12345″ is also a specific representation of the abstract quantity it stands for (and "0×3039″ would be another one).

So we *could* have converted the number 12345 to the string "12345″, placed it into a packet as 5 bytes, and then we'd have gotten this message:

```
OK 1 31 32 33 34 35
```

Hm. Still not quite what we were looking for. Because now we're dealing with ASCII text, which itself is *also* an encoding!

But we *could* build a modified version of RF12demo which converts that ASCII-encoded result back to something like this:

```
OKSTR 1 12345
```

There are however a few reasons why this is not necessarily a good idea:

- sending would take 5 bytes instead of 2
- string manipulation uses more RAM, which is scarce on an ATmega
- lots of such little inefficiencies will add up, once more data is involved

There is an enormous gap in performance and availability of resources between a modern CPU (even on the simplest mobile phones) and this 8-bit few-bucks-chip we call an "ATmega". *Mega, hah!*

But you probably didn't really want to hear any of this. *You just want your data back, right?*

One way to accomplish this, is to keep RF12demo just as it is, and perform the proper transformation on the receiving PC. Given variables "a" = 57, and "b" = 48, you can get the int value back with this calculation:

```
byte a = 57, b = 48;
int result = a + 256 * b;
```

Sure enough, 57 + 48 * 256 is… 12345 – *hurray!*

It's obviously not hard to implement such a transformation in PHP, Python, C#, Delphi, VBasic, Java, Tcl… whatever your language of choice is.

But there's more to it, alas (hints: negative values, floating point, structs, bitfields).

Stay tuned… more options to deal with these representation details tomorrow!

Update – Silly mistake: the "rf12_sendData()" call doesn't exist – it should be "rf12_sendStart()".

Binary packet decoding – part 2

In AVR, Software on **Dec 8, 2010** at **00:01**

Yesterday's post showed how to get a 2-byte integer back out of a packet when reported as separate bytes:

```
byte a = 57, b = 48;
int result = a + 256 * b;
```

Unfortunately, all is not well yet. Without going into details, the above may fail on 32-bit and 64-bit machines when sending a negative value such as -12345. And it's not so convenient with other types of data. For example, here's how you would have to reconstruct a 4-byte long containing 123456789, reported as 4 bytes:

```
byte a = 21, b = 205, c = 91, d = 7;
long result = a + 256L * (b + 256L * (c + 256L * d));
```

And what about floating point values and C structs? The trouble with these, is that the receiving party doing the conversion needs to know exactly what the internal byte representation of the ATmega is.

Here is an even more complex example, as used in the roomNode.pde sketch:

```
struct {
    byte light;     // light sensor: 0..255
    byte moved :1;  // motion detector: 0..1
    byte humi  :7;  // humidity: 0..100
    int temp   :10; // temperature: -500..+500 (tenths)
    byte lobat :1;  // supply voltage dropped under 3.1V: 0..1
} payload;
```

This combines different measurement values into a 4-byte C struct using bit fields. Note how the

“temp” value crosses two bytes, but only uses specific bits in them.

Fortunately, there is a fairly simple way to deal with all this. The trick is to decode the values back into meaningful values *by the receiving ATmega* instead of an attached PC. When doing so, we can re-use the same definition of the information. By using the same hardware and the same C/C++ compiler on both sides, i.e. the Arduino IDE, all internal byte representation details can be left to the compiler.

Let’s start with this 2-byte example again:

```
int measurement = 12345;
rf12_sendData(0, &measurement, sizeof measurement);
```

I’m going to rewrite it slightly, as:

```
typedef int Payload;
Payload measurement;

measurement = 12345;
rf12_sendData(0, measurement, sizeof measurement);
```

No big deal. This sends out exactly the same packet. But now, we can rewrite the *receiving sketch* as follows:

```
typedef int Payload;
Payload measurement;

if (rf12_recvDone() && rf12_crc == 0 &&
            rf12_len == sizeof (Payload)) {
    measurement = *(Payload*) rf12_data;
    Serial.print("MEAS ");
    Serial.println(measurement);
}
```

The effect will be to send the following line to the serial / USB connection:

```
MEAS 12345
```

The magic incantation is this line:

```
    measurement = *(Payload*) rf12_data;
```

It uses a C typecast to force the interpretation of the bytes in the receive buffer into the “Payload” type. Which happens be the same as the one used by the sending node.

The benefit of doing it this way, is that the same approach can be used to transfer any type of data as a packet. Here is an example how a Room Node code sends out a 4-byte struct with various measurement results:

```
typedef struct {
    byte light;     // light sensor: 0..255
    byte moved :1;  // motion detector: 0..1
    byte humi  :7;  // humidity: 0..100
    int temp   :10; // temperature: -500..+500 (tenths)
    byte lobat :1;  // supply voltage dropped under 3.1V: 0..1
} Payload;
Payload measurement;

measurement.light = 123;
measurement.moved = 1;
measurement.humi = 78;
measurement.temp = -15;
measurement.lobat = 0;

rf12_sendData(0, measurement, sizeof measurement);
```

And here's how the receiving node can convert the bytes in the packet back to the proper values:

```
typedef struct {
    byte light;     // light sensor: 0..255
    byte moved :1;  // motion detector: 0..1
    byte humi  :7;  // humidity: 0..100
    int temp   :10; // temperature: -500..+500 (tenths)
    byte lobat :1;  // supply voltage dropped under 3.1V: 0..1
} Payload;
Payload measurement;

if (rf12_recvDone() && rf12_crc == 0 &&
            rf12_len == sizeof (Payload)) {
    measurement = *(Payload*) rf12_data;
    Serial.print("ROOM ");
    Serial.print(measurement.light, DEC);
    Serial.print(' ');
    Serial.print(measurement.moved, DEC);
    Serial.print(' ');
    Serial.print(measurement.humi, DEC);
    Serial.print(' ');
    Serial.print(measurement.temp, DEC);
    Serial.print(' ');
    Serial.print(measurement.lobat, DEC);
    Serial.println();
}
```

The output will look like:

```
ROOM 123 1 78 -15 0
```

Nice and tidy. Exactly the values we were after!

It looks like a lot of work, but it's all very straightforward to implement. Most importantly, the correspondence between what happens in the sender and the receiver should now be obvious. It would be trivial to include more data. Or to change some field into a long or a float, or to use more or fewer bits for any of the bit fields. Note also that we don't even need to know how large the packet is that gets sent, nor what all the individual bytes contain. Whatever the sender does to map values into a packet, will be reversed by the receiver.

This works, as long as the two struct definitions match. One way to make sure they match, is to place the payload definition in a separate *header* file, say "payload.h" and then include that file in both sketches using this line:

```
#include <payload.h>
```

The price to pay for this flexibility and "representation independence", is that you have to write your own receiving sketch. The generic RF12demo sketch cannot be used as is, since it does not have knowledge of the packet structures used by the sending nodes.

This can become a problem if different nodes use different packets sizes and structures. One way to simplify this, is to place all nodes using the same packet layout into a single net group, and then have one receiver per net group, each implemented in the way described above. Another option is to have a single receiver which knows about the different types of packets, and which switches into the proper decoding mode depending on who sent the packet.

Enough for now. Hopefully this will help you implement your own custom WSN to match exactly what you need.

Update – Silly mistake: the "rf12_sendData()" call doesn't exist – it should be "rf12_sendStart()".