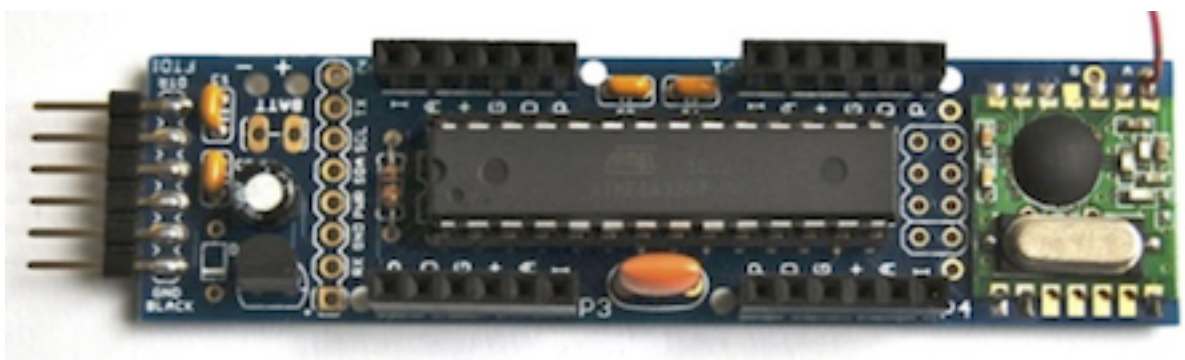
RFM69 on ATmega

In Book on May 27, 2015 at 00:01

Now that we have the RFM69 working on Raspberry Pi and Odroid C1, we've got all the pieces to create a Wireless Sensor Network for home monitoring, automation, IoT, etc.

But I absolutely don't want to leave the current range of JeeNodes behind. Moving to newer hardware should *not* be about making existing hardware obsolete, in my book!



The JeeNode v6 with its on-board RFM12 wireless radio module, Arduino and IDE compatibility, JeePorts, and ultra-low power consumption has been serving me well for many years, and continues to do so – with some two dozen nodes installed here at JeeLabs, each monitoring power consumption, house temperatures, room occupancy, and more. It has spawned numerous other products and DIY installations, and the open-source JeeLib library code has opened up the world of low-cost wireless signalling for many years. There are many thousands of JeeNodes out there by now.

There's no point breaking what works. The world wastes enough technology as it is.

Which is why, long ago a special RF69-based "compatibility mode" driver was added to JeeLib, allowing the *newer* RFM69 modules to interoperate with the *older* RFM12B's. All you have to do, is to add the following line of code to your Arduino Sketches:

```
#define RF69_COMPAT 1
```

… and the RFM69 will automagically behave like a (less featureful) RFM12.

This week is about doing the same, but *in reverse*: adapting JeeLib's existing RF12 driver, which uses a specific packet format, to make an RFM12 work as if it were an RFM69:

- Classic vs native packets – Wed
- RF compatibility options – Thu
- RF69 native on ATmega's – Fri
- Using RFM12's with RFM69 native – Sat

As I've said, I really don't like to break what works well. These articles will show you that there is no need. You can continue to use the RFM12 modules, and you can mix them with RFM69 modules. You can continue to use and add Arduino-compatible JeeNodes, etc. in your setup, without limiting your options to explore some of the new ARM-based designs.

Let me be clear: there *are* incompatibilites, and they *do* matter at times. Some flashy new features will not be available on older hardware. I don't plan to implement everything on every combination, in fact I've been focusing more and more on ARM µC's with RFM69 wireless, and will most likely continue to do so, simply to manage my limited time.

Long live forward compatibility, i.e. letting old hardware inter-operate with the new…

BEFORE
RFM69 on Raspberry Pi
May 20, 2015

AFTER
Introducing the LPC824
Jun 3, 2015

Classic vs native packets

There are so *many* different wireless radio modules and different packet protocols, that it's easy to get lost. In the context of JeeLabs, the radio modules currently in use are the **RFM12** and **RFM69**, both produced by HopeRF.

At some point, a rumour started echoing on internet that the RFM12 was being abandoned and replaced by the RFM69, but after over a year it is now clear that both will continue to be produced for a long time to come – if only because there is plenty of demand.

The use cases for both types are quite similar, i.e. being able to send and receive short packets on the 433, 868, or 915 MHz bands, but there are also some major differences:

- the RFM12 module can only store 1..2 bytes in its FIFO, meaning that data must be read and written at a fairly high rate (within 200 µs with the default JeeLib settings)
- the RFM69 can store an entire packet, so there are no hard real-time timing requirements, other than to clear the FIFO for the next reception or transmission
- the RFM69 is more sophisticated, in that many settings are more detailed – and it supports more modulation-, synchronisation-, and packet-formatting modes
- the RFM69 supports *hardware encryption*, based on 128-bit AES, and *data whitening*, which helps better recover the data rate clock in some edge cases
- the RFM69's receiver is more sensitive, and its transmitter can be more powerful, leading to a potentially much larger range with the same current consumption

All in all, the RFM69 can indeed be considered a next-generation design – although the RFM12 continues to be just fine for lots of home-monitoring and -automation scenarios. But there is one *tricky* difference: the on-air packet format / protocol of the RF12 driver (implemented by software in JeeLib) is not the same as the format in the RFM69 hardware-based packet engine:

The "**CLASSIC**" packet layout used by the RF12 driver is:

- preamble, SYNC bytes, *header* byte, packet length, payload, CRC bytes

The "**NATIVE**" packet layout used by the RFM69 hardware is:

- preamble, SYNC bytes, packet length, payload, CRC bytes

We can easily designate the first byte of the payload to act as header byte, but we *cannot* change the order of the length and header bytes. Furthermore, the length differs: on the RFM12, it does not include the header byte, whereas on the RFM69 it does.

There are other differences in specific bits and in how the CRC is calculated, but the main difference preventing trivial inter-operability is really that header-/length-byte order.

Obviously, senders and receivers need to agree on the protocol used.

So far, the code in JeeLib for both RFM12 and RFM69 has been aimed at implementing the *classic* packet format everywhere. By adding the following define into our code:

```
#define RF69_COMPAT 1
```

… we have the option to use an RFM69 module, while keeping the classic packet layout. This effectively makes the RFM69 *backwards compatible*: it acts like an older RFM12.

But there is a price to pay, because we can't use some of the newer features of the RFM69 this way, in particular full-packet buffering and hardware-based AES encryption.

On the ARM platform, we've been using a new native "RF69" driver (in rf69.h) which is considerably simpler (less code), doesn't need fast interrupt support, and offers optional encryption. But while doing so, we've also introduced a serious incompatibility.

Now seems like a good time to re-investigate all the different options and combinations…

[Back to article index]

RF compatibility options

Let's define the playing field first, and all the associated terminology:

- **Classic** packets are the ones understood by the RF12 driver in JeeLib
- **Native** packets are built into RFM69 hardware and the RF69 driver in Embello
- **RFM12** is a HopeRF wireless module, with limited FIFO & logic (one B variant)
- **RFM69** is the newer HopeRF module (there are W, CW, and HW variants)
- **RF12** is the driver in JeeLib for Arduino IDE use, supporting classic packets
- **RF69** is a slightly different API supporting the RFM69 in native mode
- **Arduino** is the name of the IDE used to build/upload JeeLib-based code
- **Make** is the Unix-style mechanism to build/upload Embello-based code
- **AVR** is the architecture of ATmega and ATtiny chips made by Atmel
- **ARM** is the architecture of NXP's LPC8xx series and many other vendors

And to elaborate a bit more on this:

- Classic vs Native refers to the packet format traveling through the air
- RFM12 vs RFM69 refers to different wireless radio hardware modules
- RF12 vs RF69 refers to different software and their slightly different API's
- Arduino vs Make refers to the cross-build + upload environments
- AVR vs ARM refers to the different µC architectures (and 8- vs 32-bit)

That's *five* pairs of variations, for a theoretical 32 different build and usage combinations! Not all of them make sense, and fewer still need to be implemented for inter-operability.

The most widely used setup is currently (because it was the first available on JeeNodes):

- Classic + RFM12 + RF12 + Arduino + AVR

While the most recent developments at JeeLabs have focused on this configuration:

- Native + RFM69 + RF69 + Make + ARM

In other words: *it couldn't be more different* and these two cannot inter-operate as is.

Using RF69_COMPAT

This configuration uses "`#define RF69_COMPAT 1`" and can be characterised as:

- Classic + RFM69 + RF12 + Arduino + AVR

It's what allows an RFM69-based node to play nice in an RFM12/RF12-type network.

On Raspberry Pi

Recent developments have made it possible to introduce a *third* platform, i.e. Linux running on Raspberry Pi's (any model) or the mostly-header-compatible Odroid C1.

Though omitted from the above list of terms to avoid confusion, it could be described as:

- Native + RFM69 + RF69 + Linux-make + RasPi

This implementation is designed to be used with the RasPi RF board.

Long term trend

Given some of the new features offered by the RFM69 in native mode, it's safe to assume that *in the long run*, the following configurations will become mainstream at JeeLabs:

- Native + RFM69 + RF69 + some-build-system + some-architecture

What this means, is that "on the air", the *NATIVE* packet format will at some point start to dominate. And while we could easily set up our environment to support native and classic packets alongside each other – using a different frequency or net group or both, with two central nodes running in parallel – this wouldn't be the most convenient setup, long term.

A much better strategy will be to just bite the bullet and make everything work in native packet format. Then we could mix and match, old and new, RFM12's and RFM69's, and still be able to treat the entire wireless network as a single one.

This is precisely what the upcoming two articles intend to describe.

Using RFM69 w/ Arduino

Luckily, the new RF69 driver is extremely portable. It started out on LPC8xx ARM chips, but has been extended to run under Linux as well – using a simple "RasPi RF" board. The next article in this series will show that it can also be used in the Arduino environment:

- Native + RFM69 + RF69 + Arduino + AVR

A new "RF12_COMPAT" option

But the most interesting option perhaps, is to bring the RFM12 into the future with a modification to the RF12 driver to make it work with native packets (keeping its API):

- Native + RFM12 + RF12 + Arduino + AVR

This could in fact be called "forward compatibility": an RFM12 can be made to act as if it were a more recently produced RFM69. Because of this, an installed base of RFM12's need not prevent progress and the use of the more advanced RFM69's in the rest of the network.

There *are* trade-offs, of course. Perhaps the most important one is that the RFM69's AES encryption engine is not available on the RFM12. But even this is not a hard restriction: in principle, AES could be implemented in software in a future extension of the RF12 driver.

Stay tuned for these two exciting new options…

[Back to article index]

RF69 native on ATmega's

The RF69 driver on GitHub is highly portable, due to the use of C++ templates to keep all platform-specific details nicely separate. All we need, to use this driver in the Arduino IDE for use with JeeNodes, is to implement an SPI class with the proper API (see spi.h):

```
template< int N>
class SpiDev {
  static uint8_t spiTransferByte (uint8_t out) {
    SPDR = out;
    while ((SPSR & (1<<SPIF)) == 0)
      ;
    return SPDR;
  }

public:
  static void master (int div) {
    digitalWrite(N, 1);
    pinMode(N, OUTPUT);

    pinMode(10, OUTPUT);
    pinMode(11, OUTPUT);
    pinMode(12, INPUT);
    pinMode(13, OUTPUT);

    SPCR = _BV(SPE) | _BV(MSTR);
    SPSR |= _BV(SPI2X);
  }

  static uint8_t rwReg (uint8_t cmd, uint8_t val) {
    digitalWrite(N, 0);
    spiTransferByte(cmd);
    uint8_t in = spiTransferByte(val);
    digitalWrite(N, 1);
    return in;
  }
};

typedef SpiDev<10> SpiDev10;
```

The template argument "N" is the pin we are going to use as master SPI select, i.e. 10.

Note that there is also an "SPI" class defined in the newer Arduino IDE releases, which you could use as basis for this SpiDev definition. It would make things more compatible if you intend to use multiple SPI devices – but the above definition will be fine for simple uses.

Using the new driver takes a little setup, since the Arduino IDE expects files to be in a specific place and the sketch directory to have a specific name, whereas this driver lives in the embello respository, which has a different layout (and is more general than the IDE). See the README on Github for how to set things up in your Arduino IDE environment.

The actual demo is very similar to the ones for the LPC8xx and the Raspberry Pi, both using this same RF69 driver. And not surprisingly, so is its output:

```
[rf69demo]
OK 80180801020304050607 (130+38:3)
OK 801809010203040506070 8 (130+6:4)
 > #1, 1b
OK 80180A01020304050607080 9 (132+18:3)
OK 80180B0102030405060708090A (130+22:4)
 > #2, 2b
OK 80180C0102030405060708090A0B (128+6:4)
OK 80180D0102030405060708090A0B0C (130+20:4)
```

The numbers in parentheses are `(-2*RSSI ± AFC : LNA)`.

The API of this new RF69 driver is slightly different from the RF12 driver API:

```
#include "spi.h"
#include "rf69.h"
...
RF69<SpiDev10> rf;
...
rf.init(28, 42, 8686); // node 28, group 42, 868.6 MHz
...
rf.send(0, txBuf, txLen);
...
int len = rf.receive(rxBuf, sizeof rxBuf);
if (len > 0) ...
```

Some notes (see rf69demo.ino for the complete example):

- the `spi.h` header has to be included before the `rf69.h` header
- you need to declare an RF69 instance to use this driver
- initialisation takes a node ID (RF69 supports 1..60), group, and frequency
- the frequency is specified in MHz, but you can add as many decimals as you want
- sending is essentially the same as with the RF12 driver
- the `rf.receive()` caller must supply the buffer to hold the incoming packet
- the return value is the actual packet size, which may exceed the data stored in rxBuf if that buffer is too small to hold the entire packet
- the first two bytes returned in rxBuf are a header byte and an origin byte – for a maximum-size packet, the buffer has to be at least 64 bytes

Several conventions have changed slightly with this new RF69 native driver:

- more node ID's, also: 61 is for send-only nodes, 62 is reserved, 63 is receive-all
- the payload length can be 0..62 bytes (compared to max 66 for the RF12)
- the header byte has the destination ID in bits 0..5 (or 0 if this was a broadcast)
- the origin byte has some flags in bits 6..7, and the origin node ID in bits 0..5

Last but not least, there's no need to frequently poll (as with `rf12_recvDone`) – you can wait to read out an incoming packet when your code is ready for it, the RFM69 will keep the entire packet in its FIFO buffer until that time (or until you re-use it for sending).

Encryption is easy with the new RF69 driver, just define a 1..16-char encryption key:

```
rf.encrypt("mysecret");
```

Note that encryption will be applied to all nodes in the same group, since the receiver cannot change its encryption mode for packets coming from different node IDs. If you need to disable encryption again, call "`rf.encrypt(0)`".

To reduce the transmit power (0 is lowest, 31 is highest and the default), use this:

```
rf.txPower(15); // -3 dBm
```

This can be useful to limit power consumption and for close-range communication.

So there you have it: a simple new RF69 driver which can be used on all RFM69-based JeeNodes and JeeLinks to run these wireless radio modules in native packet mode.

[Back to article index]

Using RFM12's with RFM69 native

So far so good – we now have the RFM69 running in native packet mode using the RF69 driver, for LPC8xx ARM µC's, ATmega328 JeeNodes, and Raspberry Pi's w/ RasPi RF.

But let's not leave the RFM12 modules behind, and the many Arduino IDE projects using the RF12 driver in JeeLib. If only it could send and receive *native* RF69-type packets!

Well, it turns out that the RFM12B wireless module *can* be tricked into doing just this. There are several issues involved:

- the frequency, FSK swing, and bandwidth can be chosen to match the RFM69
- likewise, the preamble and sync bytes can be chosen to work with both modules
- the RFM69's CRC can be computed in software (it's not the same as in RF12!)
- the data whitening used in the RFM69 can also be emulated in software
- and lastly, differences in packet / header layouts can all be handled in software

Quite a few differences, but by carefully choosing the parameters on *both* types of modules, we can indeed get packets across in both directions. An updated RF12.cpp driver has been committed to GitHub with all the necessary changes.

To enable this "native + RFM12 + RF12 + Arduino + AVR" mode, change one define in RF12.h to set `RF12_COMPAT` to 1 (not to be confused with `RF69_COMPAT` mode!):

```
#define RF12_COMPAT 1
```

Note that this change needs to be made in JeeLib itself, you cannot simply add such a define to your own sketch. The reason for this is that the changes are much more pervasive.

The result is an "RF12 driver" with a virtually unmodified "RF12 API", i.e. you can poll using `rf12_recvDone()`, etc – just as you would with a classic setup.

But there are some important differences when using RF12_COMPAT mode:

- the packet layout is different (`rf12_len` and `rf12_hdr` are defined differently)
- there's a new `rf12_dst` field, the lower 6 bits contain the destination node ID (or 0)
- the lower 6 bits of `rf12_hdr` always contain the node ID of the sending node
- `RF12_HDR_ACK` is defined as bit 6 (this is bit 5 in classic mode packets)
- `RF12_HDR_DST` is no longer present, since there is now a `rf12_dst` field

The `RF12_ACK_REPLY` test is not working right now, due to the above flag bit changes.

For an example, see the new rf12compat.ino sketch:

```
#include <JeeLib.h>

MilliTimer timer;

void setup() {
  Serial.begin(57600);
  Serial.println("\n[rf12compat]");
  rf12_initialize(63, RF12_868MHZ, 42, 1720); // 868.6 MHz for testing
}

void loop() {
  if (rf12_recvDone()) {
    Serial.print(rf12_crc == 0 ? "OK" : " ?");
    Serial.print(" dst: ");
    Serial.print(rf12_dst, HEX);
    Serial.print(" hdr: ");
    Serial.print(rf12_hdr, HEX);
    Serial.print(' ');
    for (int i = 0; i < rf12_len && i < 66; ++i) {
      Serial.print(rf12_data[i] >> 4, HEX);
      Serial.print(rf12_data[i] & 0xF, HEX);
    }
    Serial.println();
  }
  if (timer.poll(1000))
    rf12_sendNow(0, "abc", 3);
}
```

Here is some sample output, receiving packets from a Micro Power Snitch:

```
[rf12compat]
OK dst: 80 hdr: 3D 1EBFCFEF01
OK dst: 80 hdr: 3D 1EBFCFEF41
OK dst: 80 hdr: 3D 1EBFCFEF81
OK dst: 80 hdr: 3D 1EBFCFEFC1
OK dst: 80 hdr: 3D 1EBFCFEF01
```

The destination is `0x80 & 0x3F => 0`, i.e. this is a broadcast.
The sending node ID is `0x3D & 0x3F => 61`, i.e. the sender's node ID is 61.
The rest is the 5-byte payload (4 h/w ID bytes, and type 1 = MPS with 2-bit sequence).

Some code refinements are still needed – the exact settings have not yet been optimised for both modules to inter-operate as well as possible. This results in some packets still being missed (IOW, the CRC not matching up). Also, as usual with the RF12 driver, there are occasional noise packets coming in (again with non-matching CRC, and easily flagged).

But as you can see, the basic mechanism is working: *RFM12B-based nodes can successfully participate in a network designed for RFM69's, all running in native packet mode!*

In summary, the RF12 driver in JeeLib can now be used in three different ways:

- as is, the "traditional" mode: classic + RFM12 + RF12 + Arduino + AVR
- in `RF69_COMPAT` mode: classic + RFM69 + RF12 + Arduino + AVR
- in `RF12_COMPAT` mode: native + RFM12 + RF12 + Arduino + AVR

The two compatibility modes are a compromise compared with an all-RFM12 or all-RFM69 network, simply because these modules are being pushed in some very unusual ways using various software tricks, but these modes should nevertheless come in handy for existing networks, where you don't have the option to choose 100% uniform hardware.

[Back to article index]

JeeLib "Classic" Packet

Preamble			SYN		Header		Payload (n)	CRC	
0xAA	0xAA	0xAA	0x2D	Grp	Hdr	Len			

Hdr byte: Bits 7 - 5: flags, bits 4 - 0: source.

This CRC is not the same as the RFM69 CRC

Hope RFM69 Variable Length Packet

D.C. free data encoding

Checksum

Encryption

Preamble			SYN					CRC	
0xAA	0xAA	0xAA	0x2D	Grp	Len	Addr	Payload (n)		

JeeLib "RFM69 Native" Packet

D.C. free data encoding

Checksum

Encryption

Preamble			SYN		Header				CRC	
0xAA	0xAA	0xAA	0x2D	Grp	Len	Hdr	Src	Payload (n)		

Hdr byte: Bit 7: flag, bit 6: ack, bits 5 - 0: destination.
Src byte: Bits 7 & 6: flags, bits 5 - 0: source.

LowPowerLab RFM69 Packet

Preamble			SYN		Header				Payload (n)	CRC	
0xAA	0xAA	0xAA	0x2D	NetId	Len	Dest	Src	Ctl			