« Notes on using Eagle PCB

Simple driver code for Microchip MRF24J40 radio

Back in September 2011 I wrote about the rocket telemetry system I built using the Microchip MRF24J40MB radio module.

As I mentioned back then, I ended up re-writing Microchip's "MiWi P2P" stack to vastly simplify it for my application. A few people have asked for a copy of my simplified driver code, and today I'm posting it (after having cleaned up a few loose ends).

The radio supports IEEE 802.15.4 on the 2.4 GHz ISM band. The "-MB" is Microchip's long-range module – it has the radio, antenna, power amplifier (PA), and low noise amplifier (LNA), and it's good for ranges of 2500 meters or more (line-of-sight, outdoors). It comes pre-approved by the FCC for unlicensed use. You can read my old post for more info about the module.

File package

To get started, download the ZIP file from here.

This contains a whole buildable project (for MPLAB IDE v8.83) that works on my PIC32MX440-based platform. I've built it with C32 v1.11b and v2.02; it should be trivial to port it to other MCUs (see notes about that below).

The files include a very short demo program in main.c that shows how to use the driver to send and receive simple packets.

The driver itself consists of 3 files:

> *MRF24J40.c – Driver source code.*
> *MRF24J40.h – Headers and public function declarations.*
> *radioAddress.h – Sets the address for your radio.*

Now would be a good time to unzip the files and have a quick look at the source code.

Oh, and:

> ***Software rights**: I hereby grant everyone and everything in the universe permission – to the extent I have rights to grant – to use and modify this software for any purpose whatsoever. In exchange, you agree not to sue me about it. I make no promises. By using the design you agree that if you're unhappy the most I owe you is what you paid me (zip). That seems fair.*
>
> *Be aware that the original MiWi P2P v3.1.3 code this started from (of which there might not be any code left) was copyrighted by Microchip – they offer it free for use with their hardware (which is all it's useful for), and I*

Aerodynamics
Electronics
SMD
Funny
iPhone
Photos
Rockets
Sad
Software
Uncategorized
Video telecommunication
Web design

RECENT POSTS

- Simple driver code for Microchip MRF24J40 radio
- Notes on using Eagle PCB
- Mystery of the stainless steel pinhole
- Add console here
- Linux still isn't ready
- Last flights of rocket glider Autonomy
- MTK ($PMTK) GPS command summary
- Start rdesktop from a Gnome launcher

PAGES

- About me

doubt they'd want to sue their own customers over it,
but talk to them if you have concerns.

That having been said, if you work for Microchip (this means you, Yifeng) and find this code useful enough to refer it to customers, or if you want to supply it directly, you are very welcome to do so. I'd appreciate (but do not demand) in return (a) credit in the source code to this posting on NerdFever.com, and (b) a small token of your appreciation. A RealICE would be great. If that's too much, how about a Microchip T-shirt or coffee mug? (I already have a ICD3 and a Microchip bag; but swag is good. You know how to find me.)

General concept – Initialization

Call RadioInit() to initialize the radio.

General concept – Transmission

To transmit a packet, fill out the "Tx" structure to describe the packet, then call RadioTxPacket().

General concept – Reception

Call RadioRXPacket(). If there is at least one received packet the return value is non-zero (it returns the number of un-processed received packets) and the next packet is described by structure "Rx".

Do what you want with the packet, and then call RadioDiscardPacket() to throw away that packet. Now you can call RadioRXPacket() again to get the next one (if any).

API – Transmitting

void RadioTXRaw(void);

Low-level routine to transmit a single packet described by Tx.

The Tx structure must be completely setup before calling this routine. (Don't set the lqi or rssi fields; these are used only on received packets.)

It does no error checking – it assumes the transmitter is not busy, and does not do anything automatically (like incrementing the frame number or recovering from crashes). It does not block.

This routine is not normally used as an API, but it is available for transmitting unusual packets. For normal use, call RadioTXPacket() instead.

void RadioTXPacket(void);

High-level API to transmit a single packet.

Fill out the following fields of the "Tx" structure to describe your packet, then call this routine to transmit:

```
unsigned frameType // normally PACKET_TYPE_DATA
unsigned securityEnabled // must be 0 (security not
supported)
unsigned framePending // must be 0
unsigned ackRequest // usually 1
unsigned panIDcomp // usually 1
unsigned dstAddrMode // usually SHORT_ADDR_FIELD
unsigned frameVersion // must be 0
unsigned srcAddrMode // usually NO_ADDR_FIELD
UINT16 dstPANID // PAN ID of destination radio
UINT64 dstAddr; // address of destination radio
UINT8 payloadLength // length of payload field (bytes)
UINT8 * payload // points at payload start
```

You don't have to fill out the other fields; they'll be filled automatically. *(See the RadioInitP2P() function below; this will fill out most of this stuff for you.)*

Any fields that are the same as the previous packet don't need to be re-filled out either – they will keep the values they had before, unless you change them. For many applications this means the only things you need to change from one packet to the next are the "payloadLength" and the contents of the payload itself.

- GPS-guided Rocket Recovery Project

If the transmitter is busy when this is called, this routine will block until the transmitter is done with the previous packet. Typically this is less than 3 milliseconds. If you don't want to be blocked, do not call this routine until (RadioStatus.TX_BUSY == 0).

I've never seen it happen, but have read reports that (very rarely) the radio firmware can crash and the radio will need to be reset. If this happens, the driver will automatically reset the radio within 20 milliseconds; you don't have to do anything be but aware of the (very unlikely) possibility of being blocked for 20 ms. If this is a problem, use RadioTXRaw() and manage it yourself.

Note that if the radio doesn't get an acknowledgement from the far-end radio in a brief time (< 500 uS; see IEEE 802.15.4 for details), it will re-transmit the packet automatically. It will do this twice before giving up (for a total of 3 transmissions).

This improves the chance of the receiver getting the correct data, but it is not a guarantee. An acknowledgement means the receiver got the packet and computed a correct 16-bit CRC for it. It also releases the transmitter to send the next packet. Don't rely on the ACK as indicating more than a high probability that the data was received correctly; a 16 bit CRC is short enough that you will occasionally see an errored packet that has a correct CRC by chance. If you really need to be sure, use your own CRC (I recommend a 64 bit one).

UINT8 RadioTXResult(void);

Returns status of the most-recently transmitted packet. The possible return values are:

0	No result yet because TX is busy or far end hasn't had enough time to respond.
TX_SUCCESS (1)	Packet was received and acknowledged by far end (*1)
TX_FAILED (2)	Packet was not acknowledged by far end (*2)

*1 The number of re-transmissions used by the transmitter is in RadioStatus.TX_RETRIES (0, 1, or 2). Acknowledgement by the far-end is not a guarantee that the packet was delivered in all cases. In a multi-node network, it is possible in some configurations that the acknowledgment received was meant for a different transmitter (not you), because IEEE 802.15.4 acknowledgments are not addressed.

*2 It can happen that a packet was in fact received successfully at the far-end, but the acknowledgment was not received locally. In this case the local transmitter will attempt to re-send the packet up to 2 more times. If this happens, it is possible that the far-end receiver will get up to 3 copies of the same packet; it is up to the receiver to notice that they're duplicates and discard the extras. (Check for duplicate frame numbers.)

If you don't care whether the receiver got the packet, you don't need to call this.

UINT8 RadioWaitTXResult(void);

Same as RadioTXResult(), except this routine blocks for up to 19 milliseconds, and always returns either TX_SUCCESS or TX_FALIED. (It waits for the result.) Normally you'll get a result in < 3 ms, but it could be up to 19 milliseconds if the radio crashes.

Again, if you don't care whether the receiver got the packet, you don't need to call this.

API – Receiving

UINT8 RadioRXPacket(void);

This returns the count of received packets waiting to be processed, and puts the next packet to be processed (if any) into the structure "Rx". If there are no received packets, it returns 0.

Received packets are buffered in RAM until you finish processing them. The buffer can hold PACKET_BUFFERS packets (defined in the .h file; must be a power of 2).

If the receive buffer overruns (you don't process them fast enough), this will be reflected in RadioStatus.RXBufferOverruns (see the .h file).

If the return value is 0, there are no more un-processed packets in the buffer, and the "Rx" structure still describes the previous packet.

The next packet waiting to be processed is in Rx.

In most modes the radio hardware filters out received packets that aren't addressed to your radio. But in some modes it doesn't (useful for network monitoring, etc.) This routine gives you all received packets delivered by the radio hardware. If the radio

doesn't filter them, it is up to you to look at the address fields and determine if the packet was meant for you.

Also be aware that successive identical packets (same frame number) will be received if the far-end misses your acknowledgement (it will re-transmit). Check for that if you care.

void RadioDiscardPacket(void);

Discards the received packet in Rx, freeing up memory in the buffer for another packet.

Call this routine after you have processed each received packet.

API – General

BOOL RadioInit(void);

Call this once to initialize the radio. It will set the device address to MY_PAN_ID, MY_SHORT_ADDRESS, and MY_LONG_ADDRESS (all as setup in radioHeaders.h), and set the radio to channel 11.

You are free to change these at any time.

BOOL RadioSetChannel(UINT8 channel);

Tune radio to given channel. Returns true if it worked (if the channel number was a valid IEEE 802.15.4 channel number, usually in the range 11 to 25), false otherwise.

Note that this does not affect the channel used by the far-end radio(s).

void RadioSetAddress(UINT16 shortAddress, UINT64 longAddress, UINT16 panID);

Use this to change your own node address. If your address won't change and you initialized with RadioInit(), then you never need to use this.

void RadioSetSleep(UINT8 powerState);

If passed a 0, puts radio to sleep. It will draw 0.245 mA while sleeping.

If passed a 1, wakes up the radio. It will draw approximately 28.4 mA when in receive mode and a nominal peak current of 130 mA while transmitting (but an average of only 65.8 mA as fast as I can get it to transmit.)

I have found that in practice you can completely disconnect the radio from Vdd (using a MOSFET switch) for short periods (up to a few seconds, anyway) while it's in the "sleep" mode, without needing to re-initialize the radio when waking up (if you power it back up and then "wake" it, it works fine).

If you periodically cycle between power-off and "sleep" this way, you can effectively reduce the average sleep current to a small fraction of 0.245 mA.

UINT8 RadioEnergyDetect(void);

Does a single 128 microsecond energy detect on the current channel. Returns the RSSI for the channel.

This is mainly useful for protocols (like 802.15.4 and Zigbee) in which you listen before transmitting to be sure no other radio is transmitting on the channel at the same time. It also can be used to choose a channel with less noise.

General notes

All routines here do not block (they return immediately) unless noted otherwise.

There is code for interrupt-driven SPI transfers, but I never got it working (see thread on Microchip Forum about it; http://www.microchip.com/forums/m573732.aspx). The switch is SPI_INTERRUPTS; don't turn it on.

The hardware SPI transfer does work, and is significantly faster than bit-banging the SPI transfer (which also works).

Defining the HARDWARE_SPI switch turns on the hardware mode. Commenting it out uses bit-banging. The hardware mode is preferable because it's faster and requires less MCU cycles, but if your MCU doesn't support it, you can use bit-banging.

PACKET Tx, Rx;

These contain full descriptions of the packet to be transmitted or received.

MRF24J40_STATUS volatile RadioStatus;

Complete description of radio state.

Addressing

Each radio has 3 types of addresses:

- A “PAN ID” (16 bits)
- A “short address” (16 bits)
- A “long address” (64 bits)

At any given time, a radio is addressed by a combination of a PAN ID (shared by all other radios in the same network) and either a long or short address.

For the driver to receive a packet (in most modes), the incoming packet must have a destination address containing the PAN ID of your radio and the address of your radio (in either long or short form). It also must be transmitted on the channel that the receiver is tuned to.

These addresses are set during initialization in the RadioInit() function, based on the values of the macros MY_PAN_ID, MY_SHORT_ADDRESS, and MY_LONG_ADDRESS in radioHeaders.h.

Please choose your own values (don’t keep the ones in radioHeaders.h; otherwise everyone using this driver will be on the same PAN ID and address). You can change the values either by changing the macros in radioHeaders.h or by calling RadioSetAddress().

Demo code

The demo code gives a simple example of how to use the driver for communication between a pair of radios, both of which have the same address.

Files

The demo code adds the following source files to the driver:

hardware.c – Source for configuration of the hardware platform.
hardware.h – Pin definitions, etc. for the hardware platform.
debug.h – Debugging macros.
main.c – The demo program.

Platform customization

To get the demo to run on your hardware, you’ll have to modify hardware.c and hardware.h to match your platform. These files are written now to support a PIC32MX440 MCU on my own hardware. You shouldn’t have to modify any other files.

In hardware.h, make sure that macro BAUD_RATE matches your terminal baud rate, and the pin definitions (for radio control and SPI interface) match your platform.

In hardware.c, the function BoardInit() must initialize your platform, and trigger the radio ISR function (in MRF24J40.c) when the radio INT pin goes high. The ReadUART() function must return the ASCII code sent by the terminal, or 0 if there is no data from the terminal.

Running the demo

Run the demo program on two boards. They should be within radio range of each other.

Attach an ASCII terminal to the serial I/O of each MCU. (I used UART2.) A PC-based terminal program such as Hyperterminal or PuTTY works fine. The terminal should be set to 8 data bits, 1 stop bit, and no parity, and the baud rate to BAUD_RATE (I use 460,800 bps).

Type A, B, or C on either terminal. Each of the 3 keys should send a different message (in a single packet) to the other radio. The message is printed on the terminal of the receiver.

How the demo code works

The inner while() loop checks for received packets from the radio. If it finds one, it checks that the frame number is not duplicated from the previous packet (this can happen if the far-end missed our ACK and re-transmitted; usually only happens under very weak signal conditions). If it’s not a duplicate, it prints out the contents of the packet payload.

The switch() statement checks for keys from the terminal, and sends a packet containing a payload appropriate to each key.

Note that during initialization the function RadioInitP2P() is called. This initializes the Tx structure to send packets in a simple point-to-point mode. Each packet contains data, doesn’t use security (not implemented in the driver), requests an ack from the far-end

radio, and is addressed to the same address as the local radio (both radios have the same address). It uses a 16-bit 'short' address. Here's the code:

```
void RadioInitP2P(void)
{
Tx.frameType = PACKET_TYPE_DATA;
Tx.securityEnabled = 0;
Tx.framePending = 0;
Tx.ackRequest = 1;
Tx.panIDcomp = 1;
Tx.dstAddrMode = SHORT_ADDR_FIELD;
Tx.frameVersion = 0;
Tx.srcAddrMode = NO_ADDR_FIELD;
Tx.dstPANID = RadioStatus.MyPANID;
Tx.dstAddr = RadioStatus.MyShortAddress;
Tx.payload = txPayload;
}
```

Once you've called this to initialize the Tx structure, further packets can be sent just by changing the contents of the payload buffer (*Tx.payload), setting the payload length (Tx.payloadLength), and calling RadioTxPacket().

See the demo source code (main.c) to see how this is done.

Further reading

If you want to understand how the driver works in detail, before looking at the source code, skim over:

http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf

and

http://ww1.microchip.com/downloads/en/DeviceDoc/39776C.pdf

The former is the IEEE spec. The latter is Microchip's datasheet for the radio – look at sections 3.2 (Initialization), 3.11 (Reception), and 3.12 (Transmission). You can skip the rest.

Submit Comment

PAGES

About me

GPS-guided Rocket Recovery Project