MicroModem Quickstart Guide

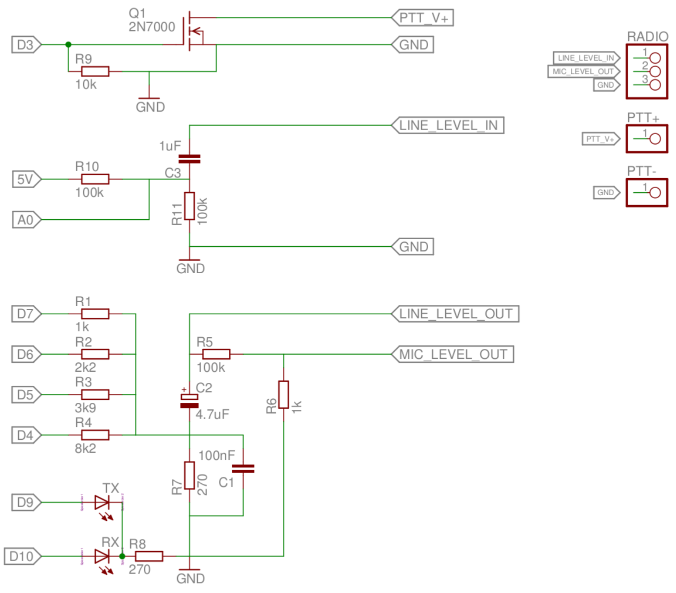
This guide provides a few pointers on getting started with MicroModem, how to build the circuit, what options to set in the software, and how to compile and flash the firmware. The USB-serial connection to the modem is 9600 baud, 8N1, no flow control.

Notes on the circuit

The circuit for MicroModem is really simple, and only needs a few common components. Even so, you might want to modify a few things depending on what medium you want to use MicroModem over. The default configuration of the circuit works well for sending the modulated waveform over hanheld ham-radio transceivers. This means a few things for the circuit:

- Modulated output audio from the modem is brought down to *microphone* level voltages before leaving the modem. This is around 20mV peak-to-peak.

- The PTT triggering part of the circuit is designed with the assumption of a Kenwood HT style PTT switch, namely: The ground pin on the **microphone** jack of the radio supplies about 3v, which will trigger PTT if it is connected to the ground pin of the **speaker** jack. The circuit accomplishes this by using a N-channel MOSFET to operate as a switch between the +3v and GND for PTT. Don't use a bipolar junction transistor for the PTT triggering: I have found them to be way to susceptible to picking up the RF radiation from the radio, and keeping the PTT triggered! You need a MOSFET.



Have a look at the schematic here. The circuit is divided into three logical parts.

- The topmost part is the PTT triggering circuit.

- The middle part is the input stage. This part AC-couples the input from the speaker jack of the radio and provides a 2.5v DC-bias, so we will be able to sample the audio with the ATmega's ADC. **R10** and **R11** constitutes a voltage divider that provides a 2.5v bias source from the ATmega's regulated 5V output.

- The bottom part is the output stage. This part is basically a very simple "soundcard". **R1** through **R4** constitutes a 4-bit string-resistor DAC. **C2** AC-couples the output, and **R5** and **R6** divides the output voltage down to around 20mV peak-to-peak. Most radios wont be able to accept the line-level output generated directly by the ATMega, without distorting it to a point where it will be impossible to correctly demodulate again. **But, if you want to test the modem over a direct 3-wire connection from modem to modem, you MUST take the modulated output before the R5/R6 voltage divider. The ATmega ADC is not sensitive enough to demodulate audio in the 20mV range!** *(Well, maybe it is, if you supply a tightly regulated 20mV reference for the ADC, but that is beyond the scope of this document)*

Putting it together

I'd suggest putting together a test-version on some breadboard or other prototyping setup first, and making sure everything works for your particular setup. If you are having problems, try to leave out the PTT part first, and make sure the input and output stages are working, and that you can successfully transmit directly over a 3-wire connection from one modem to another. Connect the output port of one modem to the input port of the other, the input port of the first modem to the output port of the other, and also connect GND together as well. If this works you should be good to also transmit over radio. You might need to adjust the values for the output voltage divider, if your radio expects another input voltage. As noted earlier, you also might need to modify the PTT triggering part to accommodate the way your radio's PTT work. If you are in doubt, you're welcome to leave me a comment and I will try to help.

Configuration

There's a few easy choices you can make in the source-code depending on how you want to use the modem. The default configuration provides a setup that will send any data received on the serial port out through the modem, and print any data received back out over serial. The default options include:

- Max frame length of 264 bytes, giving a max data length of **262 bytes per frame**, because of 1-byte header and 1-byte checksum.

- Automatic compression enabled. The modem will compress packets, if the compression actually saves space. If the compression doesn't reduce the packet size, the packet is sent uncompressed.

- 12,8 Hamming code forward error correction. This can correct up to two bits in error per byte.

- 12-byte block interleaving. By interleaving data in blocks of 12-bytes, we can better correct burst errors. The 12-byte interleaver makes it possible to correct errors in up to 8 consecutive bits.

- Automatic transmit with no serial framing. The data received over serial is sent out when a delay in the serial data is detected. The default configuration waits 2 milliseconds after each serial byte has been read, and if no more data is coming in, the buffer is transmitted. The buffer is also transmitted if the max data size has been reached.

- No TX queue. The modem will not queue packets for transmission. Due to memory constraints, it's not possible to implement both a TX queue and compression at the same time. The queue is needed for TCP/IP though, since we need to be able to accept the minimum TCP MSS of 536 bytes.

- No CSMA. The modem will not use Carrier Sense Multiple Access. This means that a transmit command will make the modem transmit no matter if there's already channel activity. You can enable the P-persistent CSMA if you want to have the modem try to avoid packet collisions. You will need to tune the parameters for your particular radio setup.

If you would like to change any of these behaviours, they can be configured in the “configuration.h” and “mp1.h” files. All of the necessary directives are in the beginnings of these two files.

Compiling and flashing

If you are fine with the default options, you can just skip ahead to flashing the firmware. The git repo includes a prebuilt .hex firmware file you can upload to the modem. If you want to compile your own version, you need some basic packages on your system: an AVR GCC toolchain, AVR-libc and make utilities. On Ubuntu/Debian this can be installed with:

```
sudo apt-get install build-essential gcc-avr avr-libc binutils-avr avrdude
```

Similar packages should exist for other distros. I have no clue how to setup an AVR toolchain on windows, so I'm unfortunately no help there! Sorry! On OS X it should be pretty simple to compile the AVR toolchain from source, but I don't know if prebuilt packages exist.

Once you have a working toolchain installed, you can simply open a terminal, `cd` into your `MicroModem` directory and execute `make`.

```
cd MicroModem
make
```

If all went well, you should see something like:

```
Building revision 1754
Modem: Compiling bertos/cpu/avr/drv/ser_avr.c (C)
Modem: Compiling bertos/cpu/avr/drv/ser_mega.c (C)
Modem: Compiling bertos/cpu/avr/drv/timer_avr.c (C)
Modem: Compiling bertos/cpu/avr/drv/timer_mega.c (C)
Modem: Compiling bertos/drv/ser.c (C)
Modem: Compiling bertos/drv/timer.c (C)
Modem: Compiling bertos/io/kfile.c (C)
Modem: Compiling bertos/mware/event.c (C)
Modem: Compiling bertos/mware/formatwr.c (C)
Modem: Compiling bertos/mware/hex.c (C)
Modem: Compiling bertos/struct/heap.c (C)
Modem: Compiling Modem/main.c (C)
Modem: Compiling Modem/hardware.c (C)
Modem: Compiling Modem/afsk.c (C)
Modem: Compiling Modem/protocol/mp1.c (C)
Modem: Compiling Modem/compression/heatshrink_decoder.c (C)
Modem: Compiling Modem/compression/heatshrink_encoder.c (C)
Modem: Compiling bertos/mware/formatwr.c (PROGMEM)
Modem: Compiling bertos/drv/kdebug.c (PROGMEM)
Modem: Linking images/Modem.elf
"/usr/bin/avr-"objcopy"" -O srec images/Modem.elf images/Modem.s19
"/usr/bin/avr-"objcopy"" -O ihex images/Modem.elf images/Modem.hex
"/usr/bin/avr-"objcopy"" -O binary images/Modem.elf images/Modem.bin
```

Your compiled firmware can now be found in **`MicroModem/images/Modem.hex`**.

To flash the firmware to the modem, you can use `avrdude`. Here's an example of how to use it:

```
avrdude -p m328p -c arduino -P /dev/tty$1 -b 115200 -F -U flash:w:images/Modem.hex
```

For convenience, I have included a small shell scripts that makes flashing a little easier. If your serial port is on /dev/ttyUSB0, you can just execute:

```
./flash USB0
```

Which will then call `avrdude` for you :) At this point you should be ready to transmit some data! Yay!

Connecting to the modem

The USB serial connection to the modem is **9600 baud, 8N1, no flow control**. This seems a little counterintuitive since the modem itself is only 1200 baud, but having a faster connection for sending and receiving data to the host makes sense, since the processor will spend less time in serial interrupts. I also found SLIP unwilling to correctly change serial port baud to anything else than 9600 baud on the systems I used it on, so I decided to stay with 9600 baud to preserve compatibility with SLIP. You can change the serial baudrate in the source if you would like to use a higher or lower rate.

You can connect any serial terminal program to the serial port of the modem and simply send and receive text like that. You can also write your own host program to send and receive data with the modem. If you want rudimentary TCP/IP, you can use SLIP (Serial Line IP). This gives you a point-to-point IP connection through the modem. If you want to use SLIP, check out this blog post for details on attaching a SLIP interface to the modem:
http://patrickst.blogspot.dk/2011/11/tcpip-over-slip-on-gnulinux-ubuntu.html

If using SLIP, remember to set the correct configuration options in the header files! You will need to tune the CSMA parameters for your particular radios for TCP/IP to work well. You probably wont be able to completely avoid packet collisions, but you do need to get it to a relatively sensible level for TCP to work well.