FREAKLABS

HOWTO:
Using the chibiArduino Wireless Protocol Stack
v1.06

Document Revision History

Date	Description
2010-11-01	v0.5 Document creation
2010-12-28	v0.51 Added new configuration parameter: CHIBI_PROMISCUOUS
2013-10-06	v1.03 A badly needed update
2016-01-05	v1.05 Added newly supported PHY INIT modes for high data rate OQPSK (2 Mbps, 1Mbps, 0.5 Mbps) for 2.4 GHz and (1Mbps, 0.5Mbps) and BPSK at 20 kbps for long range applications.
2016-01-24	v1.06 Fixed typos in chibiSetMode and chibiSetDataRate commands

Introduction

The chibiArduino protocol stack is a port of the Chibi 802.15.4 wireless stack to the Arduino platform. The stack was designed to be simple, easy to use, and have a very small memory footprint.

Chibi was created as a tool to introduce people to wireless communications and sensor networking. There are a number of protocols for wireless communications and sensor networking but most of them are complex with sophisticated features. In contrast, many people in the hobbyist and hacker communities just want something to send and receive small amounts of data without having to understand a lot of protocol complexity.

For simplicity, chibi relies mainly on three functions: init, send, and receive. There are also other functions available for management or configuration, however with just the basic initialization function, the default settings can allow people to start communicating wirelessly.

Traditionally, wireless communication protocol stacks also require an operating system, operating system-like services, or a complex state machine to manage connections. However since the radio hardware in many wireless ICs now include functionality to handle things like automatic retries and timeouts, these can be offloaded to hardware and simple communications can be performed without he need for an operating system or a complex state machine. This greatly simplifies the protocol stack implementation and also decreases the amount of system resources such as flash and RAM used to implement the communications.

The chibiArduino software also integrates a command line which can be optionally used inside an Arduino sketch. User commands can be integrated into the command line to call user functions interactively from a terminal. This is useful for configuring the node such as setting the address or channel. It's also a useful tool for controlling the data that is being sent and seeing the data that's being received from the node.

Operation

chibiArduino usage is fairly straightforward and just consists of initializing/configuring the node and setting up when the sending and receiving will take place.

Initialization

The first thing that needs to be done is to initialize the node. In a user sketch, it looks something like this:

```
void setup()
{
    chbInit();
}
```

This would set up the node with the default address, default network ID, and default channel.

More sophisticated configuration can also be performed:

```
void setup()
{
    chbInit();
    chibiSetShortAddr(0xAAAA);
    chibiSetChannel(15);
}
```

This would set the node up, change the network address to 0xAAAA (the address used to uniquely identify the node), and set the channel to 15. Incidentally, chibiArduino uses the IEEE 802.15.4 specification which allocates 16 channels in the 2.4 GHz spectrum of 11 channels in the 868/900 MHz spectrum. These channels are defined as channels 11 to 26 at 2.4 GHz and 0 to 10 in 868/900 MHz with channel 0 being 868 MHz. The channels need to be within the proper ranges for the selected device.

Transmitting

Transmitting data requires three things: a destination address to transmit the data to, the actual data, and the length of the data. Performing a simple "hello world" transmission would go something like this:

```
void loop()
{
    byte msg[] = "Hello World";
    // 1 byte added to length for terminating character
    chibiTx(0xAAAA, msg, 12);
}
```

You'll notice that the length of the Hello World message is 11 bytes but an extra byte was added to the transmission length. This is because strings contain a trailing NULL character to terminate the string. It's just one of those quirky things in programming.

In the example above, the node would transmit "Hello World" to the node whose address is 0xAAAA. The length informs the receiving end of how much data to expect. It's also important to keep the messages under the maximum payload size for an 802.15.4 frame. For the chibiArduino stack, the maximum payload size is 116 bytes but its set to 100 bytes in the user configuration parameters just to maintain a margin.

Another example of wireless transmission would be something like this:

```
void loop()
{
   byte data[6];
   readAccelerometer(data);
   chibiTx(CHIBI_BROADCAST_ADDR, data, 6);
   delay(500);
}
```

This code reads data from an accelerometer, broadcasts it to all nodes in the same network, and then waits for 500 msec before repeating the process. CHIBI_BROADCAST_ADDR is defined inside the chibiArduino stack and contains the value of 0xFFFF. This is the reserved address defined by the 802.15.4 specification for broadcast transmissions and all nodes on the same network will receive transmissions to this address.

Receiving

Receiving data is a little bit more complicated than transmitting data and that's because there is little control over when data is received. Data can come in at any time and the main loop needs to check if any new data arrives and handle it. Here's an example of how to do it:

```
void loop()
{
byte data[100];

   if (chibiDataRcvd() == true)
   {
     chibiGetData(data);
   }
}
```

In the above example, a byte array was created to store the received data. The "chibiDataRcvd()" function is used and will return true if new data has arrived. If there is data available, then the chibiGetData() function will retrieve the data and store it in the byte array.

A better way to handle receiving data would be like this:

```
void loop()
{
   byte len, data[CHIBI_MAX_PAYLOAD];
   if (chibiDataRcvd() == true)
   {
      len = chibiGetData(data);
      if (len == 0) return;
   }
}
```

In this case, there were two variables created. The "len" variable is used to store the length of the received data in bytes. When the chibiGetData() function is called, it will store the received data in the specified array and also return the length. The length can then be used to loop through the data array, check it for validity, or whatever else the application might require. If the returned length is 0, then the data is duplicate data and should be discarded. Duplicate data usually happens at reception boundaries where an ACK is unheard triggering a retry of the previous transmission.

Command Line Operation

The cmdArduino library has been integrated into the chibiArduino communications stack because it makes many things very convenient. Having an interactive command line makes things like setting different network addresses for each node very simple. It also gives the user control over when to send data and provides visual feedback on what kind of data arrived. Since the command line is configurable to call user functions, anything you might want to do such as reading the radio registers, MCU registers, toggling an I/O, or whatever else might want to be done is also possible interactively.

For a detailed tutorial on how to use the command line, please ***refer to the cmdArduino tutorial on the FreakLabs website***. All the functions are the same except they are prefixed with "chibi". For example, "cmdInit()" in the tutorial becomes "chibiCmdInit()" when accessed through the chibiArduino stack.

Configuration Parameters

The configuration parameters for the chibiArduino stack are contained in the "chibiUsrCfg.h" file in the main Chibi directory. There are detailed descriptions of each parameter in the file, but a few of the parameters are listed here for further description of their functionality.

CHIBI_PROMISCUOUS

This mode allows the chibiArduino stack to operate in promiscuous mode. In this mode, the stack will receive all frames within listening range regardless of address, duplicate frames, or any type of CRC error. The frame will not be processed and have the header stripped. All frame information is retained. This mode should only be used when the application requires raw 802.15.4 data frames such as when using the chibiArduino stack as a packet sniffer in conjunction with a tool like Wireshark. This is also useful for capturing raw data traffic and then parsing for analysis.

CHB_MAX_PAYLOAD

As mentioned before, the maximum payload size for a data frame using the chibiArduino stack is 116 bytes. By default, the CHB_MAX_PAYLOAD is set to 100 which allows for some extra overhead. However its possible to set this to the max payload size as well.

CHB_EEPROM_IEEE/SHORT_ADDR

These parameters are used to set the addresses in EEPROM where the IEEE or short addresses will be stored. Each device should have unique addresses stored in the EEPROM so its important to set this address so that it won't get overwritten by any other code that writes to the EEPROM.

CHB_SLPTR_*
CHB_SPI_CS_*
CHB_RADIO_IRQ
CFG_CHB_INTP
CHB_IRQ_ENABLE/DISABLE

The configuration of these registers and pins are automatically handled by the stack and the board settings. The board setting is chosen inside the Arduino IDE and should correspond to the board that will be used.

CHB_2_4GHZ_DEFAULT_CHANNEL

This is the default channel for the 2.4 GHz radios. The channels start from 11 (0x0B) and go up to 26 (0x1A). On startup, this parameter is used to set the default channel for the board.

CHB_900MHZ_DEFAULT_CHANNEL

This is the default channel for the 900 MHz radios. The channels start from 0 (868.3 MHz) and 1 through 10 being evenly spaced from 906 to 924 MHz.

CHB_2_4GHZ_INIT_MODE

This is the default initial mode for the 2.4 GHz radios. The available choices are:

- OQPSK_2000 : 2 Mbps mode with OQPSK modulation. This mode is proprietary to Atmel.
- OQPSK_1000 : 1 Mbps mode with OQPSK modulation. This mode is proprietary to Atmel.
  OQPSK_500 : 500 kbps mode with OQPSK modulation. This mode is proprietary to Atmel.
- OQPSK_250 : 2Mbps mode with OQPSK modulation. This mode is the 802.15.4-2006 standard.

CHB_900MHZ_INIT_MODE

This is the default initial mode for the 900 MHz radios. The available choices are:

- OQPSK_SINRC_100: 100 kbps w/OQPSK modulation and SIN-RC pulse shaping. Compatible with 802.15.4
- OQPSK_SIN_250: 250 kbps w/OQPSK & SIN pulse shaping. Compatible with 802.15.4
- OQPSK_RC_250 : 250 kbps w/OQPSK & RC pulse shaping. Compatible with 802.15.4c (China)
- OQPSK_SIN_500 : 500 kbps w/OQPSK. Proprietary to Atmel.
- OQPSK_SIN_1000 : 1 Mbps w/OQPSK. Proprietary to Atmel.
- BPSK_40 : 40 kbps w/BPSK. Low data rate but much longer range. Compatible with 802.15.4.
- BPSK_20 : 20 kbps w/BPSK. Lowest data rate, longest range. Compatible with 802.15.4.

There are other modes that the stack and radio supports but the discussion of the tradeoffs and implications are beyond the scope of this document.

CHB_PAN_ID

The default network ID for the 802.15.4 network. All devices that are to communicate with each other must have the same PAN ID. To have more than one network that are within listening range of each other, the PAN IDs must be different.

CHB_2_4GHZ_TX_PWR

CHB_900MHZ_TX_PWR

This is the default power settings for the radios. The corresponding register values can be found in the comments.

CHB_MAX_FRAME_RETRIES

The number of times a frame will be retried if no ACK is received and a timeout occurs. After the max number of retries, the radio will give up and return the CHB_NO_ACK status to the user

CHB_MAX_CSMA_RETRIES

The maximum number of times a transmission will be attempted if channel is busy. After each attempt, a backoff occurs for a random amount of time and then another transmission is attempted. After the max number of attempts, the radio will give up and return the CHB_CHANNEL_ACCESS_FAILURE status.

chibiArduino API List

This is a list of the default functions available in the chibiArduino wireless stack.

void chibiInit()

Usage: This is the initialization function for the chibiArduino stack and needs to be in the setup() area of the arduino code.

```
void setup()
{
    chibiInit();
}
```

void chibiSetShortAddr(uint16_t addr)

Usage: This function sets the short address of the wireless node. It writes the short address into the EEPROM at the CHB_EEPROM_SHORTADDR position and also writes it into the radio's RAM. The radio will then automatically filter frames and only accept ones sent to this address or the broadcast address. On initialization, the short address is automatically loaded from the EEPROM to the radio.

```
chibiSetShortAddr(0x1234);
```

Sets the 16-bit short address of the node to 0x1234.

void chibiGetShortAddr()

Usage: Gets the short address of the node from the EEPROM.

```
unsigned int addr;
addr = chibiGetShortAddr();
```

Retrieves the short address from the EEPROM.

void chibiSetIEEEAddr(uint8_t addr[])

Usage: Sets the 64-bit IEEE address of the node in EEPROM and in the radio. The IEEE address is not used in the chibiArduino stack but may be used by an application. The IEEE address is a unique identifier different for all nodes in existence. Ideally, a block of IEEE addresses should be purchased from the IEEE to take advantage of the uniqueness. However this address can also be used to store 8 character messages that describe the node.

```
byte addr[8]="HUMIDTY";
chibiSetIEEEADDR(addr);
```

Sets the IEEE address to the byte array representing HUMIDTY.

void chibiGetIEEEAddr(uint8_t addr[])

Usage: Retrieves the 64-bit IEEE address of the node from EEPROM. A byte array needs to be passed in and will store the address.

```
byte addr[8];
chibiGetIEEEAddr(addr);
```

Gets the IEEE address and stores it in the byte array.

uint8_t chibiRegRead(uint8_t addr)

Usage: Reads the radio's register at the given address. In normal operation, this should not be needed, however it may be useful for debugging purposes or if the user wants to check the register settings in the radio. An enumerated list of all the register addresses in the radio can be found in the chb_drvr.h file.

```
byte version = chibiRegRead(VERSION_NUM);
```

Reads the version number of the radio IC.

void chibiRegWrite(uint8_t addr, uint8_t data)

Usage: Writes the specified data to the radio's register at the address specified. In normal operation, this will probably not be used but can be used if the user wants to experiment with different register settings for the radio.

```
byte data = 0xFF;
chibiRegWrite(VERSION_NUM, data);
```

Writes 0xFF to the VERSION_NUM register in the radio. This will have no effect since the version register is read-only.

uint8_t chibiTx(uint16_t address, uint8_t data[], uint8_t len)

Usage: This is the main function to send data. The first argument is the destination address of the node the data will be going to. To send to all nodes, the reserved broadcast address 0xFFFF can be used and the data will go out to all nodes in the same network within listening distance. The second argument is a byte array containing the data to be sent. The third argument is the length of the data in the byte array. The function returns the status of the transmission and an

enumerated list of the return codes can be found in src/chb_drvr.h.

```
byte status, data[CHB_MAX_PAYLOAD];
for (int i=0; i<CHB_MAX_PAYLOAD; i++)
{
    data[i] = i;
}
status = chibiTx(0xFFFF, data, CHB_MAX_PAYLOAD);
```

This example fills a byte array with sequential numbers up to the max payload size. It then broadcasts it to all nodes and returns the status of the transmission.

uint8_t chibiDataRcvd()

Usage: This function returns TRUE if new data was received and is ready to be picked up. If CHB_RX_POLLING_MODE is 0, the data has already been moved to a temporary storage area in memory so that new data won't overwrite it. If CHB_RX_POLLING_MODE is 1, the data is still in the radio and calling this function will retrieve the data from the radio and move it into temporary storage in memory..

```
if (chibiDataRcvd() == true)
{
    Serial.println("New data has arrived.");
}
```

uint8_t chibiGetData(uint8_t data[])

Usage: This function retrieves the data from the temporary storage in memory (receive buffer) and puts it into the specified byte array. It returns the length of the data that was written into the array.

```
if (chibiDataRcvd() == true)
{
    byte len, data[CHB_MAX_PAYLOAD];
    len = chibiGetData(data);
    if (len == 0) return;
}
```

This example checks to see if data was received. If new data has arrived, then it will retrieve the data, store it in the specified byte array, and store the length of the data. If a duplicate data is received, it will return a length of 0 and the data should be discarded.

uint8_t chibiGetRSSI()

Usage: Gets the signal strength of the last received data frame. The range of the signal strength is 84 (-7 dB) to 0 (-91 dB) for the AT86RF230 with an accuracy of +/- 5 dB.

```
byte rssi = chibiGetRSSI();
Serial.print("RSSI = "); Serial.println(rssi, HEX);
```

uint16_t chibiGetSrcAddr()

Usage: Gets the source address for the most recently received frame. This is useful to figure out who originated the frame so that the data could be processed accordingly.

```
int src_addr = chibiGetSrcAddr();
Serial.print("Source Address = 0x");
Serial.println(src_addr, HEX);
```

uint8_t chibiSetChannel(uint8_t channel)

Usage: This sets the channel of the radio. The default channel the radio is initialized to is configured in chibiUsrCfg.h. If the channel needs to be dynamically changed, this function can be used. According to the IEEE 802.15.4 specification, there are 16 channels in the 2.4 GHz band from channel 11 to channel 26. When specifying the channel to change to, the values must be within this range. The function returns the status of the channel change.

```
chibiSetChannel(15);
```

Changes the channel the radio uses to channel 15. Incidentally, this channel will not interfere with any Wi-Fi channels.

void chibiSleepRadio(byte enable)

Usage: Put the radio into sleep mode. When in sleep mode, the radio will not be able to send or receive data. This is typically used to save power when the radio is not needed. A non-zero argument will enable sleep mode. An argument of 0 (false) will disable it and put it into receive mode.

```
chibiSleepRadio(true);
```

uint8_t chibiGetPartID()

Usage: This retrieves the ID of the radio for identification purposes. It's usually used to distinguish between the AT86RF230 (2.4 GHz), AT86RF231 (2.4 GHz), and AT86RF212 (900

MHz) radios.

```
uint8_t id = chibiGetPartID();
if (id == CHB_AT86RF212) Serial.println("AT86RF212");
```

void chibiSetDataRate(uint8_t rate)

Usage: This is used to set the datarate for the radios. For the 2.4 GHz radios, the maximum datarate is 2 Mbps. For the 900 MHz radios, the maximum datarate is 1 Mbps. The arguments are:

CHB_RATE_250KBPS
CHB_RATE_500KBPS
CHB_RATE_1000KBPS
CHB_RATE_2000KBPS

```
chibiSetDataRate(CHB_RATE_1000KBPS);
```

void chibiSetMode(uint8_t mode)

Usage: Used to set the transmission modulation of the 900 MHz radio. For 900 MHz, the OQPSK modulation allows higher data rates up to 1 Mbps. The BPSK modulation allows greater range at the expense of higher data rates. The valid modes are:

OQPSK_SIN
BPSK_40

```
chibiSetMode(BPSK_40);
```

void chibiHighGainModeEnable()

Usage: his is only for devices that have an RF front end and support long range operation. By default, high gain mode is enabled which enables the amplifier on the receiver. This improves the signal reception by about 10x although it also increases the noise. Real improvement is spec'd to be around 6 dB or about 4x.

void chibiHighGainModeDisable()

Usage: This is only for devices that have an RF front end and support long range operation. This function will disable the receiver amplifier. This is useful for power savings while the radio is still on.

```
chibiHighGainModeEnable();
chibiHighGainModeDisable();
```