FREAKLABS

HOWTO:
Using the chibiArduino Wireless Protocol Stack
v0.51

Document Revision History

Date	*Description*
2010-11-01	v0.5 Document creation
2010-12-28	v0.51 Added new configuration parameter: CHIBI_PROMISCUOUS

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. As of this writing, the stack footprint is approximately 300 bytes of RAM and 3.8 kB of flash.

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. These channels are defined as channels 11 to 26 so the channel needs to be within this range.

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);
   }
}
```

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.

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.

CHB_RX_POLLING_MODE

This is an interesting parameter. When data arrives into the radio, the radio will inform the microcontroller via an interrupt. There are two ways to handle this. One is to move the data immediately via an interrupt service routine and the other is to flag the interrupt and move it at the next available time the MCU is free. Both of these methods are supported in this stack because there is a bit of debate within the Arduino community about what can be done inside an interrupt service routine. In some cases, like in a very busy system with many strict timing requirements, taking the time to move data inside an interrupt service routine might not be possible without violating the timing of some other driver. In other cases, the communication data is high priority and so it needs to be moved as quickly as possible before more data is received. Because of the wide application scenarios that the Arduino could be put in, both methods were implemented. The default is that the communications data is high priority and will be moved inside the interrupt service routine however this can be turned off by changing this parameter from 0 to 1.

In the interrupt based receiving mode, you get the data out of the radio as quickly as possible and store it into the microcontroller's memory where it can be retrieved later by the application. The reason you'd want to do this is if the data isn't moved off the radio before the next frame arrives, the next frame will overwrite the previous frame and you'll lose that data. If you set this parameter to 0 (default), the data will be moved inside the interrupt service routine which means that moving the data takes priority over all other tasks.

In the polling based receiving mode, once the radio issues an interrupt signaling data arrival, you flag the interrupt and service it at the next available opportunity. This means that moving the data occurs outside of the interrupt service routine and is not considered a high priority. When the chibiDataRcvd() function is called, it will poll the receive interrupt flag and if it happens to be set, it will access the radio and remove any data inside of it into memory. Setting this parameter to 1 enables this mode. The downside is that if another wireless data frame comes in before the MCU has a chance to retrieve the data from the radio, then the original data will be lost. Incidentally, if the Arduino Ethernet shield is used, then the CHB_RX_POLLING_MODE will need to be set to 1 due to a problem it has with other devices accessing the SPI bus from an interrupt. This can also be worked around with a code modification to the ethernet library.

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.

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_PORT
CHB_SLPTR_DDR

CHB_SLPTR_PIN

The SLP_TR pin on the radio controls the sleep mode of the radio. When the radio is not in use and power savings is desired, then the device can be put to sleep by bringing this pin low. This is handled in the chibiSleepRadio() function which relies on this pin definition of the SLP_TR pin.

CHB_RADIO_IRQ
CFG_CHB_INTP
CHB_IRQ_ENABLE/DISABLE

These parameters set up the interrupt pin for the radio. The RADIO_IRQ parameter defines which interrupt vector will be used and is based on the pin that the interrupt goes to. The CFG_CHB_INTP parameter contains the code needed to initialize the interrupt and put it in the proper mode of operation, ie: rising edge, falling edge, etc. The IRQ_ENABLE/DISABLE contains the code to enable or disable the IRQ and is only used if RX_POLLING_MODE is set to 1.

CHB_SPI_CS_PORT
CHB_SPI_CS_DDIR
CHB_SPI_CS_PIN

These parameters set up the SPI's chip select pin. This pin needs to map to the microcontroller pin connected to the radio's SPI's chip select. The port and direction register must also be specified in order for the radio to work properly.

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(unsigned int 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(byte 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(byte 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.

byte chibiRegRead(byte 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(byte addr, byte 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.

byte chibiTx(unsigned int address, byte data[], byte 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.

byte 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.");
}
```

byte chibiGetData(byte 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);
}
```

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.

byte 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);
```

int 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);
```

byte chibiSetChannel(byte 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);
```