

10/1/2009

WiFly GSX Case Study: Data logger

1. Overview

Dataloggers are used in a variety of industries, some examples are:

- Environmental sensors
- Motor RPM measurements
- Location tags using GPS
- Temperature sensors
- Light sensors
- Motion sensors

A data logger can be configured in a number of ways depending on the application. This case study covers the following application topics:

- **Point to point setup:** In this scenario the module is used to transmit sensor readings, but is not connected to the user's network. Typically used in remote areas such as maritime sensors, the user can periodically connect via adhoc to the data logger and download the sensor data that was collected.
- **Network setup:** When the module can be connected to an access point, the user has the ability to upload the data as frequently as needed from anywhere in the world through the user's network.

Within these two application setups there are a number of implementation options to consider:

- **Sleep/Wake:** To keep power consumption as low as possible, we often want to put the WiFi module into sleep mode, and only have it wake up when we want to upload data from the data logger.
- **Host side:** Depending on the application we can configure the host in two ways: Push and Pull. When using a Pull setup the host is responsible for opening the connection to the module to initiate an upload of data. Alternatively with a Push setup, the module opens a connection and the user's application program on the micro pushes the data to a server on the network. This is typically needed when working with large amounts of data that cannot be stored locally.

2. WiFly GSX Module Features

For use with a data logger the following features are important and should be considered when choosing a WiFi module:

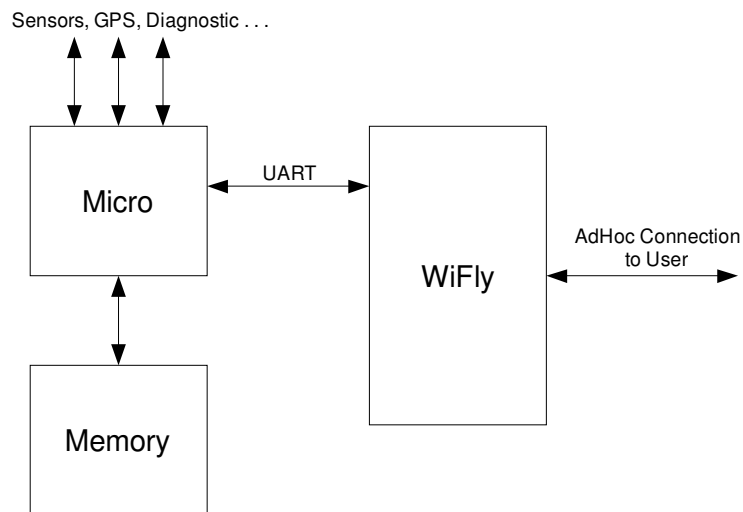
- Battery powered
- Low power consumption
- Enterprise and adhoc network
- Periodic wake-up capability
- Wake-up on sensor inputs
- Simplicity of setup and configuration
- Built in networking applications DHCP, UDP, DNS, ARP, ICMP

10/1/2009

3. Applications

3.1 Data logger - Point to Point

As shown in the block diagram below, the system consists of the user's microprocessor with connected memory and a simple UART interface to the WiFly module.

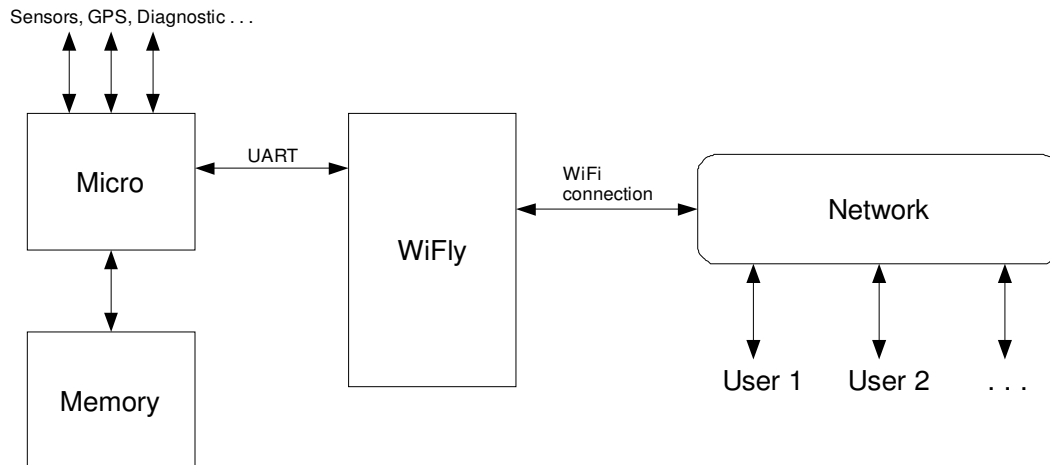


The WiFly module is configured to create an ad-hoc network to which the user can then connect. Once connected to the WiFly module the user can retrieve the data from the memory through the micro.

3.2 Data logger - Networked

When an access point is available, the module can be configured to associate with the network. The user can then connect to the module and download data over the network from anywhere. This setup is shown below.

10/1/2009



4. Implementation

In this section we go through the setup procedure for each of the cases in a step-by-step manner.

4.1 Entering Command Mode

The WiFly module can be configured both locally through a UART connection or over the air via a Telnet connection. The UART configuration is the simplest but has the disadvantage that it cannot be done remotely, and requires access to the UART pins on the module. For most users the following explanation on entering command mode should be enough, but for more detail on setting up a connection to the module, see the Evaluation board manual RN-131G_EVAL and the WiFly GSX user manual.

NOTE: We suggest using TeraTerm as your terminal emulator program. This is available for download from the Roving Networks website. <http://www.rovingnetworks.com/support/teraterm.zip>

Uart connection:

- Determine the COM port the module is connected on.
- Next open up a terminal emulation program specifying the COM port found in the previous step. If using TeraTerm, select **Serial** and choose the COM **Port** from the pull down list. Note: the default serial port setting is 9600, 8 bit, no parity.
- From within the terminal window, put the WiFly GSX module into command mode by typing **\$\$\$** in the terminal window. You should get **CMD** back confirming you are in command mode.

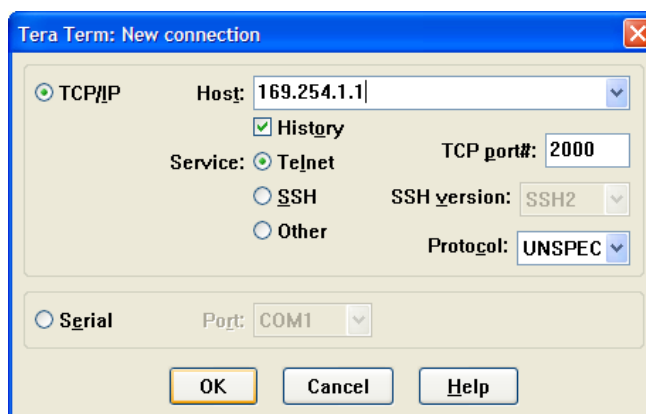
Telnet connection:

To enable adhoc mode via hardware: Set PIO9 high (3.3V) at power up. When the module powers up in adhoc

10/1/2009

mode the WiFly module creates an adhoc network with the following information:
 SSID: WiFly-GSX-XX where XX is the final two bytes of the device's MAC address
 IP address: 169.254.1.1

To start a Telnet session, start TeraTerm and set it up to connect to the module as follows:



The module will reply with the string ***HELLO*** indicating that the connection has been established. From within the terminal window, put the WiFly GSX module into command mode by typing **\$\$\$** in the terminal window. You should get **CMD** back confirming you are in command mode.

4.2 Data logger – Point to Point

Step 1: Set up the wlan properties

From command mode we configure the module for adhoc mode using the join command. We also set the name and channel of the adhoc network:

```
set wlan join 4           <Puts the module in adhoc Mode>
set wlan ssid my_adhoc_network <Sets the name of the adhoc network>
set wlan chan 1          <Sets the channel of the adhoc network>
```

Step2: Set up the ip properties

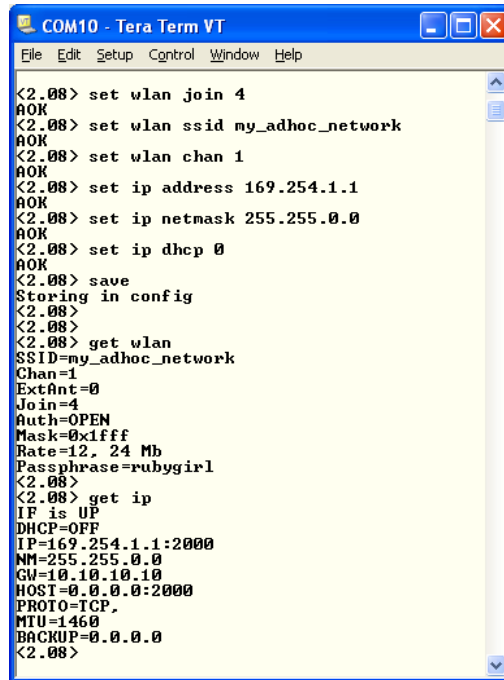
Next we turn off DHCP and set the IP address and netmask so other devices know where to connect to the WiFly GSX. Since auto IP fixes the first two bytes of the IP address you want to use the netmask of 255.255.0.0 so that other device connecting to the module can be reached. Alternatively you can set the netmask to a smaller subnet if the other device's IP addresses are being set statically to the same subnet as the adhoc device.

```
set ip address 169.254.1.1 <Set the module's IP address>
set ip netmask 255.255.0.0 <Set the module's IP netmask>
```

10/1/2009

set ip dhcp 0 <Turn off DHCP>

Be sure to save the configuration, then upon reboot the module will be in adhoc mode.

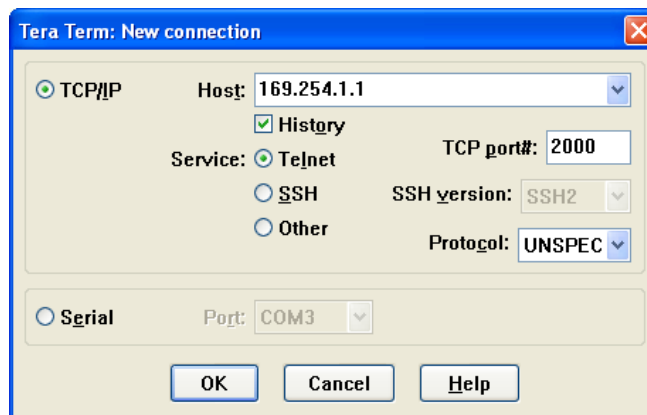


```
<2.08> set wlan join 4
AOK
<2.08> set wlan ssid my_adhoc_network
AOK
<2.08> set wlan chan 1
AOK
<2.08> set ip address 169.254.1.1
AOK
<2.08> set ip netmask 255.255.0.0
AOK
<2.08> set ip dhcp 0
AOK
<2.08> save
Storing in config
<2.08>
<2.08>
<2.08> get wlan
SSID=my_adhoc_network
Chan=1
ExtAnt=0
Join=4
Auth=OPEN
Mask=0xffff
Rate=12, 24 Mb
Passphrase=rubygirl
<2.08>
<2.08> get ip
IF is UP
DHCP=OFF
IP=169.254.1.1:2000
NM=255.255.0.0
GW=10.10.10.10
HOST=0.0.0.0:2000
PROTO=TCP
MTU=1460
BACKUP=0.0.0.0
<2.08>
```

Step 3: Connecting to the module over adhoc

When using Windows, it is very important to disable all existing network connections before attempting to associate to the adhoc network. Otherwise Windows won't use Auto-IP, and the IP address of the computer won't be set to 169.254.x.x

Open another Tera Term window and set it up as follows:



Tera Term: New connection

☒ **TCP/IP** Host: 169.254.1.1

☒ History TCP port#: 2000

Service: ☒ Telnet SSH version: SSH2

☐ SSH Protocol: UNSPEC

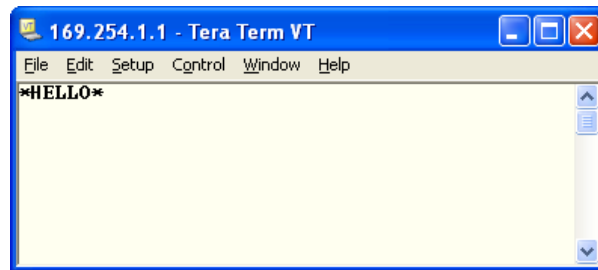
☐ Other

☐ **Serial** Port: COM3

OK Cancel Help

10/1/2009

Clicking on OK, you should get the following screen displaying ***HELLO***, which indicates that the connection is open.



Any characters typed in this window will now appear in the other Tera Term window.

4.3 Data logger – Networked

Step 1: Set up the wlan properties

After selecting the network that we want to connect to, we set up the module so that it will connect to that network automatically upon rebooting. In this example we want to connect to the wireless network TheLoft.

```
set wlan join 1          <Autojoin>
set wlan chan 0          <Scan all channels>
set wlan ssid TheLoft    <Network: TheLoft>
set wlan phrase rubygirl <Pass phrase for TheLoft>
save                    <Save changes>
```

The **join 1** setting ensures that when the module wakes up, it tries to join the access point that matches the stored SSID, passkey and channel. Channel can be set to 0 for scanning. (Default)

Note that TheLoft is a secure network, so that's why we need to provide the pass phrase. The same procedure can be used to connect to an open network without supplying the pass phrase. The **save** command is needed to save the configuration so that when the module wakes up, it connects to TheLoft again.

Step2: Set up the ip properties

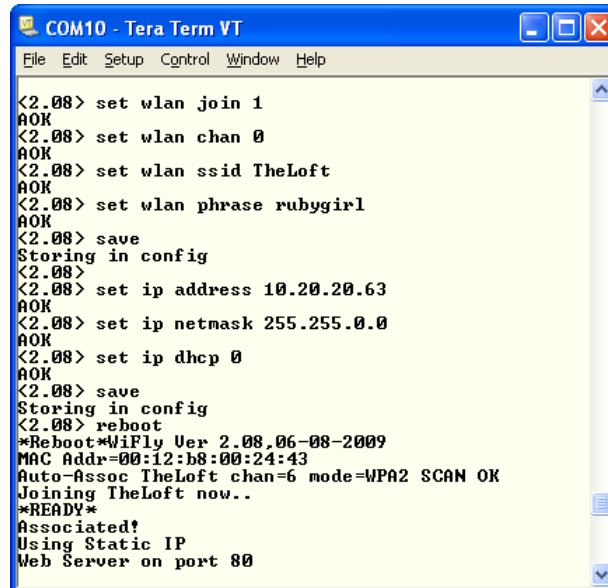
Next we turn off DHCP and set the IP address and netmask so other devices know where to connect to the WiFly GSX. Since auto IP fixes the first two bytes of the IP address you want to use the netmask of 255.255.0.0 so that other device connecting to the module can be reached. Alternatively you can set the netmask to a smaller subnet if the other device's IP addresses are being set statically to the same subnet as the adhoc device.

```
set ip address 10.20.20.63 <Set the module's IP address>
```

10/1/2009

```

set ip netmask 255.255.0.0    <Set the module's IP netmask>
set ip dhcp 0                <Turn off DHCP>
save                          <Save configuration>
  
```



```

COM10 - Tera Term VT
File Edit Setup Control Window Help

<2.08> set wlan join 1
AOK
<2.08> set wlan chan 0
AOK
<2.08> set wlan ssid TheLoft
AOK
<2.08> set wlan phrase rubygirl
AOK
<2.08> save
Storing in config
<2.08>
<2.08> set ip address 10.20.20.63
AOK
<2.08> set ip netmask 255.255.0.0
AOK
<2.08> set ip dhcp 0
AOK
<2.08> save
Storing in config
<2.08> reboot
*Reboot*WiFly Ver 2.08.06-08-2009
MAC Addr=00:12:b8:00:24:43
Auto-Assoc TheLoft chan=6 mode=WPA2 SCAN OK
Joining TheLoft now..
*READY*
Associated!
Using Static IP
Web Server on port 80
  
```

To check that the module has been set up correctly, use the **get wlan** and **get ip** commands. This will display the settings that were saved in the module's configuration memory.

Step 3: Set up the host configuration

Now we set up the host address and port number on the WiFly module, so that it connects to the host when it wakes up.

```

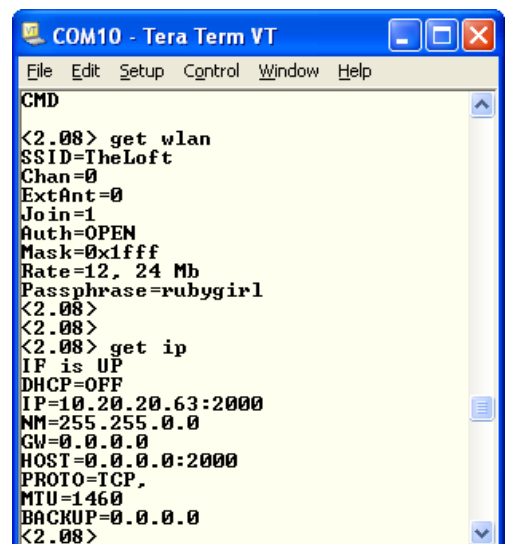
set ip host 10.20.20.75      <Set the host IP address>
set ip remote 3000           <Set the remote port>
  
```

These commands set the host IP address to 10.20.20.75, and the host port to 3000. Next we configure the module to connect to the stored remote host IP every 2 seconds.

```

set sys autoconn 2           <Connect to the host every 2 seconds>
save                          <Save configuration>
  
```

Note: If autoconn is set to once the module will only make one attempt to auto connect. 0 is the default and disables auto connect.



```

COM10 - Tera Term VT
File Edit Setup Control Window Help

CMD

<2.08> get wlan
SSID=TheLoft
Chan=0
ExtAnt=0
Join=1
Auth=OPEN
Mask=0xffff
Rate=12, 24 Mb
Passphrase=rubygirl
<2.08>
<2.08>
<2.08> get ip
IF is UP
DHCP=OFF
IP=10.20.20.63:2000
NM=255.255.0.0
GW=0.0.0.0
HOST=0.0.0.0:2000
PROTO=TCP,
MTU=1460
BACKUP=0.0.0.0
<2.08>
  
```

10/1/2009

Step 4: Connecting to the module over the network

The module has been set up to autoconnect, so upon rebooting it will try to connect to the host ip that was set in step 3. We can still use Tera Term to connect to the module, as in **section 4.2**, but if we want the module to connect upon rebooting we need a server that listens on port 3000. See **section 4.5 – Host side** for details on programming a host so that it listens on port 3000.

4.4 Data logger – Sleep/Wake

For many applications it is desired to keep the power consumption as low as possible. To achieve this we can put the WiFly module in the 10uWatt sleep mode. There are a couple of ways in which we can configure the module to wake up: The module can wake up periodically, on sensor input or on a UART input from the user's microprocessor.

4.4.1 Periodic wakeup

In this example we set up the module so that it will go to sleep after 30 seconds and wake up after sleeping for 10 seconds.

```
set sys wake 10      <Wake after 10 seconds>
set sys sleep 30     <Go to sleep after 30 seconds>
save                 <Save configuration>
```

After entering the “save” command, the module will stay on for 30 seconds and then go to sleep. NOTE: If not using Sensor pins to wake the module, be sure to set the wake timer before issuing the sleep timer or the module will not wake up.

4.4.2 Wake on Sensor Input

There are 4 inputs available to wake the module from sleep, SENS0-3. For this example we activate SENS2 as follows:

```
set sys trigger 4    <Activate wakeup on SENS2>
set sys sleep 30     <Go to sleep after 30 seconds>
save                 <Save configuration>
```

The trigger value is a bitmapped setting, so the value to enter is calculated as $2^{\text{pin\#}}$. The SENSE inputs do NOT have a resistor divider to allow pins to tolerate 3V logic, so a minimum of 24K in series with 10K to ground as a divider network should be used.

WARNING: Under no conditions should the voltage on any SENS0-7 input exceed 1.2VDC. Permanent damage to the module will result. The SENS0-3 inputs have a small current source that is activated in sleep mode. This source is approximately 100nA, and will cause the input to float up to about 1.2VDC. IF SENSE2 for example, is enabled, pulling the SENS1 pin to GROUND will wake the device. An open drain FET is a good device to tie to the pin. The threshold is about 500mV. Additional pullup to 1.2VDC may be needed if the circuit has an impedance (due to leakage current) of less than 5Mohms (500mv / 100nA). SENS1-4 pins that are not used should be left unconnected.

4.4.3 Wake on UART

10/1/2009

When the module is in Sleep mode, the UART itself is disabled. However, wake on UART can be accomplished by connecting the SENS1 pin to the UART RX or CTS pin using a minimum of 24K in series with 10K to ground as a divider network. To enable wakeup on the RX or CTS pin, use

```
set sys trigger 1      <Activate wakeup on SENS1>
set sys sleep 30      <Go to sleep after 30 seconds>
save                  <Save configuration>
```

When waking up on the RX pin it should be noted that the first (or possibly multiple) byte(s) sent into the module will likely be lost, so the designer should take care to send a preamble byte to wake up the module before sending valid data by. A better way to do this is to use the CTS input to wake the module, and wait until it is ready to accept data.

4.4.4 WiFly behavior upon wakeup

The behavior of the WiFly module when waking up is determined by the **join** and **autoconn** settings.

- If **join** is set to 4 as in section **4.2 Data logger – Point to Point** the module will wake up and create an adhoc network.
- If **join** is set to 1 as in section **4.3 Data logger – Networked**, the module will associate with the access point that matches the stored SSID, pass phrase and channel.
- If **autoconn** is set to 1 or higher, as in section **4.3 Data logger – Networked**, the module will wake up and connect to the host id that was specified. Note that it is necessary to have a server listening on the specified port so that the module can connect to it. The client side setup is covered in the next section.

4.4.5 The Idle timer

As long as the IP connection is open, the module will not go back to sleep. To ensure that the module goes to sleep after the data is uploaded to the server we can set the idle timer as follows:

```
set comm idle 5      <Disconnect 5 seconds after no data activity>
```

This causes a disconnect if no transmit or receive data is seen for 5 seconds.

4.5 Data logger – Host Side

For both network setups, the module can be used in either a Push or Pull mode. This section describes the typical setup for both scenarios.

Note: The code presented in this section is used to explain the actions the host program should take, it is not intended as a complete source for a server application and should not be used as such.

4.5.1 Push setup

In a typical Push setup, we want the WiFi module to wake up periodically and connect to the server upon waking up. The user's application code on the micro can then upload the data to the server. The communication process is as follows:

- The server is listening for a connection on a port, say port 3000

10/1/2009

- The module wakes up, and attempts to connect to the server
- The server accepts the connection and sends a string to the user's micro to let it know that the connection is open
- The server waits to receive incoming data
- The user's micro sends the data to the server
- The user's micro closes the connection once all the data has been uploaded

There are a number of ways to open a connection to the host server once the module is awake. As describes in section 4.3, the module can be configured to connect automatically once it wakes up. Alternatively the user's micro can enter command mode and use the **open** command to open a connection to the host.

To allow the module to connect, the server must be listening for a connection on port 3000. The following is a simple code example of a server that listens on port 3000, opens a stream socket and sends the string "Hello WiFi module" to the module once the connection has been established. The application on the user's micro can then send the collected data to the server.

```
#define PORT "3000"           // Port number users will be connecting to
#define BACKLOG 10           // Number of pending connections queue will hold
#define MAXRECVSIZE 100      // Max number of bytes we can receive at once

// This function is a handler for the child process, it's used later to clean
// up the processes that appear as the fork()ed child processes exit.
void sigchld_handler(int s)
{
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

// This functions returns a pointer to the address info in the given structure
void *get_in_addr(struct sockaddr *sa)
{
    return &(((struct sockaddr_in*)sa)->sin_addr);
}

int main(void)
{
    int sockfd, new_fd;        // File descriptors: listen on sock_fd,
                                // New connection on new_fd

    struct addrinfo hints, *servinfo, *p; // Structs to store the server's address info
    struct sockaddr_storage their_addr;    // Connector's address information
    socklen_t sin_size;
    struct sigaction sa;
    int yes = 1;
    char s[INET_ADDRSTRLEN];             // Space to hold the WiFi module's IP address string
    char recvbuf[MAXRECVSIZE];           // Stores the received data
    int rv;                               // A return value used in error checking

    // First we get the address info from this machine for use in filling out the
    // other structures
    memset(&hints, 0, sizeof hints);      // Clear the hints structure
    hints.ai_family = AF_INET;            // Use IPv4
    hints.ai_socktype = SOCK_STREAM;      // TCP stream sockets
    hints.ai_flags = AI_PASSIVE;          // Use my IP

    if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
```

10/1/2009

```
fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
return 1;
}

// servinfo now points to a linked list of 1 or more struct addrinfos
// We now want to get a socket descriptor and bind it to the port
// Loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
        p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }

// This code just allows the system to reuse the port, otherwise if the
// port has been used previously, bind() may fail, claiming "Address already in use."
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
        sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

// Bind to the port
    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        continue;
    }

    break;
}

// Display an error message if we couldn't bind:
if (p == NULL) {
    fprintf(stderr, "Server: failed to bind\n");
    return 2;
}

// Now that we have the socket set up, we don't need the server address info anymore, so
// we free that memory
freeaddrinfo(servinfo);

// Listen for incoming connections on the port
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen error");
    exit(1);
}

// This code cleans up the processes that appear as the fork()ed child processes exit.
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction error");
    exit(1);
}

// Wait for a connection and accept it once it comes in on the port
printf("Server: waiting for connections...\n");

while(1) {
```

10/1/2009

```

    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }

// Once we have accepted the connection, convert the IP address of the module
// to a string and print it out

    inet_ntop(their_addr.ss_family,
        get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);
    printf("Server: Connected to %s\n", s);

// Send the string "Hello WiFi module" so that the user's micro knows the connection
// is successful and get ready to receive data
    if (!fork()) { // This is the child process
        close(sockfd); // The child doesn't need the listener
        if (send(new_fd, "Hello WiFi module", 17, 0) == -1)
            perror("send");

        if ((numbytes = recv(new_fd, recvbuf, MAXRECVSIZE-1, 0)) == -1) {
            perror("recv");
            exit(1);
        }

        close(new_fd);
        exit(0);
    }
    close(new_fd); // The parent process doesn't need this

    return 0;
}

```

4.5.1 Pull setup

In a typical Pull setup, we want the WiFi module to wake up periodically and listen on a port so that the host can open a connection to the module and initiate the upload of data from the user's micro. The communication process is as follows:

- The module wakes up, and listens on a port, say port 2000
- The host attempts to open a connection to the module
- The module accepts the connection and sends a string to the user's micro to let it know that the connection is open
- The host sends a command to the user's micro to request the data
- The host receives the data from the user's micro
- The host closes the connection once all the data has been uploaded

To allow the host to connect to the module, the module must be listening on port 2000. The following is a simple code example of a host program that opens a stream socket on port 2000 and sends the string "Ready for data" to the module once the connection has been established. The application on the user's micro can then send the

10/1/2009

collected data to the host.

```

#define PORT "2000"           // The port to connect to the module
#define MAXRECVSIZE 100      // Max number of bytes we can receive at once

// This functions returns a pointer to the address info in the given structure
void *get_in_addr(struct sockaddr *sa)
{
    return &(((struct sockaddr_in*)sa)->sin_addr);
}

int main(void)
{
    int sockfd, numbytes;      // sockfd stores the socket file descriptor
    char recvbuf[MAXRECVSIZE]; // Stores the received data
    struct addrinfo hints, *servinfo, *p; // Structs to store the server's address info
    int rv;                   // A return value used in error checking
    char s[INET_ADDRSTRLEN];  // Space to hold the WiFi module's IP address string

    // First we get the address info from this machine for use in filling out the
    // other structures
    memset(&hints, 0, sizeof hints); // Clear the hints structure
    hints.ai_family = AF_INET;       // Use IPv4
    hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

    if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // servinfo now points to a linked list of 1 or more struct addrinfos
    // loop through all the results and connect to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                             p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("client: connect");
            continue;
        }

        break;
    }

    // Display an error message if we couldn't connect:
    if (p == NULL) {
        fprintf(stderr, "Failed to connect\n");
        return 2;
    }

    // Convert the IP address of the module to a string and print it out
    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
              s, sizeof s);
    printf("Connecting to %s\n", s);

```

10/1/2009

```
// Now that we have the connection set up, we don't need the server address info anymore, so
// we free that memory
freeaddrinfo(servinfo); // all done with this structure

// Receive the "*hello*" string from the module and send the request for data to the user's
// micro.
if ((numbytes = recv(sockfd, recvbuf, MAXRECVSIZE-1, 0)) == -1) {
    perror("recv");
    exit(1);
}

if (send(sockfd, "Ready for data", 14, 0) == -1)
    perror("send");

// Wait to receive the data from the user's micro

if ((numbytes = recv(sockfd, recvbuf, MAXRECVSIZE-1, 0)) == -1) {
    perror("recv");
    exit(1);
}

close(sockfd);
return 0;
}
```