techFX Zigbee rev A

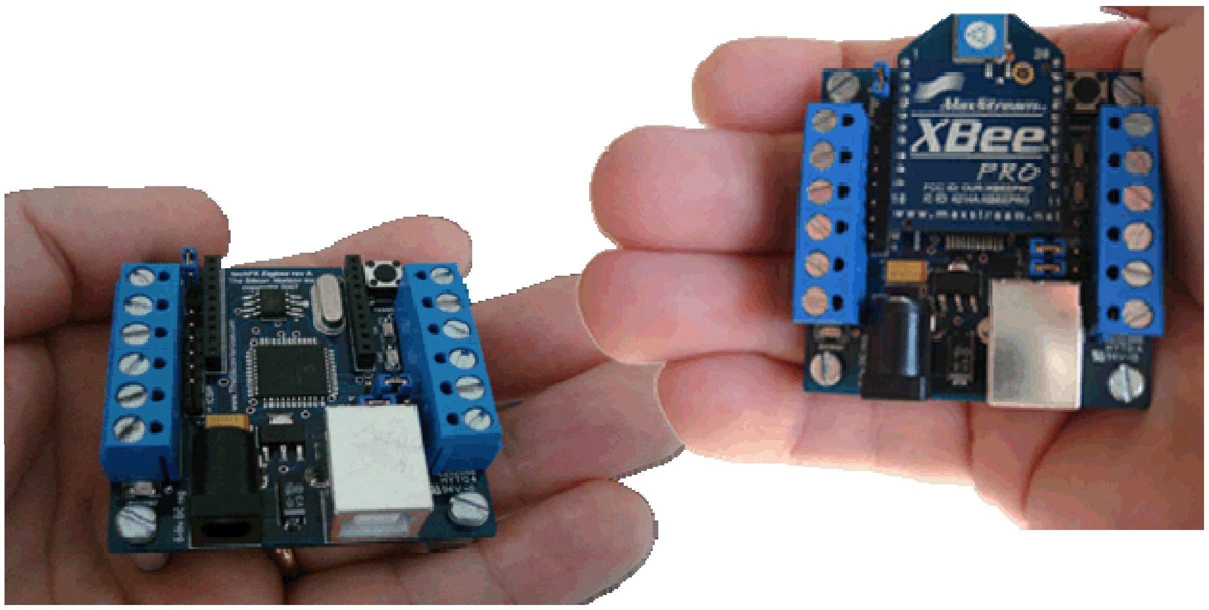
techFX Zigbee Tools v 1.0

by

The Silicon Horizon Inc.

www.TheSiliconHorizon.com

Email: support@TheSiliconHorizon.com



Contents

End-user License and agreement of liability

The End-user (you) agrees to the terms of liability and license by the use of this product. The product is the techFX Zigbee embedded controllers, and the software which is used on it including: The windows terminal program, boot loader, example applications, and all firmware revisions that may be released at a later date including windows updates.

The End-user is hereby licensed to use these products, programs, source code, and firmware in a non-commercial manner pertaining to the parts of code that are Intellectual property of The Silicon Horizon Inc. The portions include all of the applications, source code, and firmware except the USB framework which is licensed directly to you specifically for use on Microchip MCU based products. All commercial ventures should contact The Silicon Horizon Inc. for a commercial Intellectual property license.

By using these products, you agree to not hold The Silicon Horizon Inc. responsible for any damages which might occur to your equipment, household, or personal being. These products are manufactured to all legal specs, and carry FCC certification as set forth in following sections. We will not be responsible for misuse of the equipment, and negligence on the part of the End-user.

FCC Device Compliance

The techFX Zigbee controller is designed to operate with an RF module from Digi corp. named the Xbee series 1 and series 2. These modules contain all the RF components that are associated with the FCC compliance and are certified under the name of Digi corp. (formerly Maxstream corp.). The certifications can be found on their website and exist in different forms for different countries.

Since the techFX Zigbee contains no further RF components on our embedded controller, no further FCC compliances are necessary. It is an FCC requirement that the End-user affix the FCC label to the enclosure of the product when it is installed.

Computer System requirements:

Operating system

You must have NET 2.0 framework distribution installed to run this software. You can do a web search for "dotnetfx.exe" or use the following link:

http://www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en

TechFX tools has been tested to work in Microsoft Windows XP SP2 and Windows Vista. To run the software in Vista with UAC on, you must run the program in Administrator rights mode. You also must right click the program, and run it in “windows XP SP2 compatibility mode”. You should also go to the control panel and find the “PIC 18f4550 family device” driver and go to the power management tab, and disable the power management feature of Vista for that device driver. If you change the USB port, the power feature will turn itself on again and you must disable it again.

Cables and power supply

You will need a free USB 2.0 port, cable and an AD/DC wall adapter with 5-9v DC output, 2.1 mm jack Center positive, regulated output. The board will not run on USB power alone. You must have the wall adapter plugged in to operate the controller.

It is strongly recommended that you use a switching (regulated) power supply for this device. Switching power supplies offer true rated voltages at small loads while cheap power supplies might output 14 volts when rated a 9 volt wall adapter. The excessive voltage will be given off as heat, as a LDO linear voltage regulator is used in this application.

Technical features

The techFX Zigbee controller has a wide variety of features that can be used in a variety of applications. The list below is subject to change as the configuration of the controller can add better functionality in later revisions.

Application specific

Networking

- Zigbee wireless mesh networking including devices such as coordinators, routers, and end devices. Mesh networking creates an adaptive network that will get the data to its destination as parts of the network become unreachable due to power concerns or inference. The network can adapt to many outside uncontrollable changes.
- 802.15.4 wireless networking allows devices to be configures as coordinators or end devices. These networks can be beacon or non beacon in nature (although currently no firmware exists for the beacon specific networks).
- Point to point networking.
- Point to multipoint networking (broadcast mode).
- Peer to Peer networking.

Sensors and output devices

- Specialized sensors such as a compass, accelerometer, GPS, or CCD camera can be attached to the I2C port. Many other I2C sensors exist.
- Specialized output devices can be used on the I2C port such as a Graphical LCD, or regular LCD. Some models allow a keypad to be hooked directly to the LCD, and the keypad can be sampled through the I2C port with regular communication methods.
- Classical sensors such as analog and digital can be used on the IO ports, and can be configured in almost any manner necessary (i.e.: 8 analog, or 4 analog and 4 digital).
- Classical output devices such as LED's, and Solid state relays can be used to drive the outside devices that you wish to control.

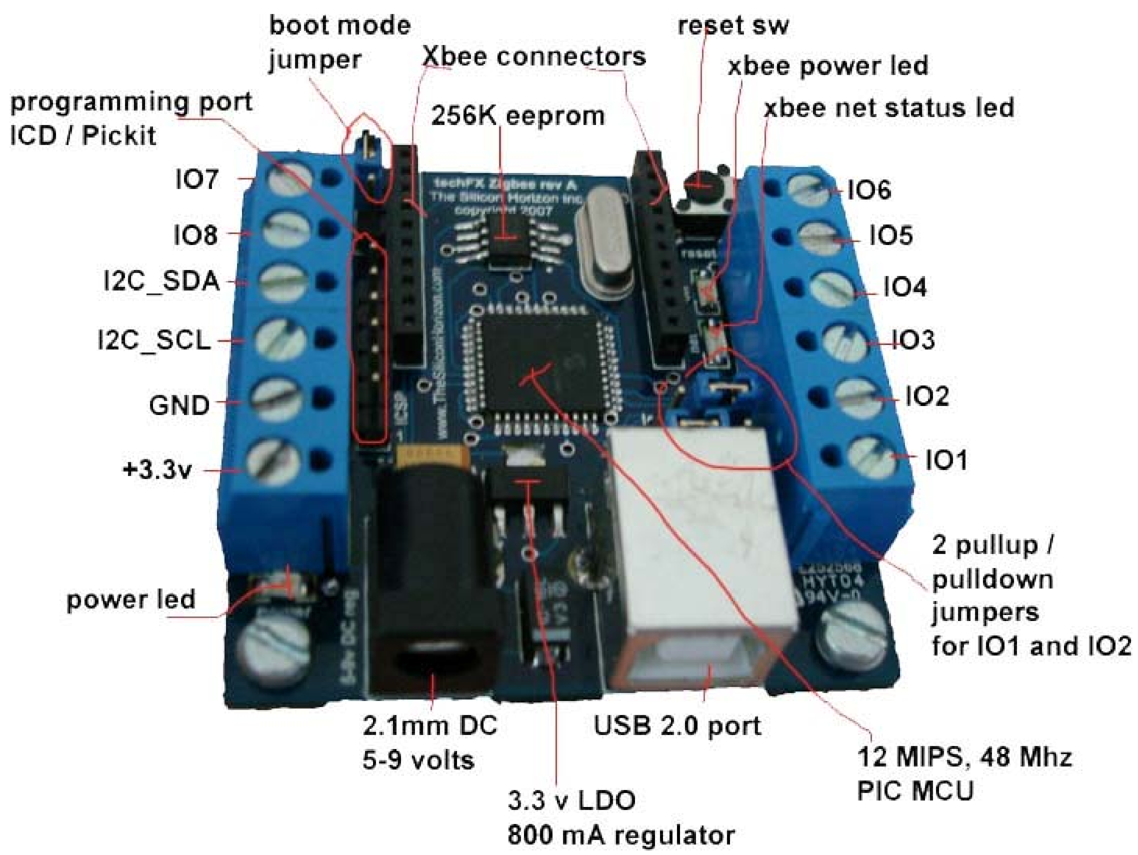
Software control, and data acquisition

- Use our terminal program to quickly setup modules with network parameters. Also diagnose network problems and query all nodes on the network with one command on the terminal.
- Develop your own application for data acquisition using our data acquisition example written using Microsoft Visual C#. Have the sensor data logged at regular intervals dictated by periodic functions in the firmware of each device (i.e.: have a sensor read every second or minute then transmitted). With our firmware framework, these features can be setup in minutes as can the configuration of IO ports and periodic functions.
- Use our USB boot loader application to upload new custom firmware that you edit to the device without having to purchase additional equipment (except for Xbee module firmware updates).

Hardware specific

- Connector for Xbee series 1 and series 2 RF modem for Zigbee and 802.15.4 wireless networking
- 12 MIPS 48 MHz 18F4550 Microchip PIC MCU
- FS USB 2.0 communications
- USART serial port for Xbee up to 200 Kbaud/second
- 1 MHz I2C port on screw terminals
- 256K on I2C eeprom, 32K on MCU
- 8 IO on screw terminals configurable as analog or digital IO
- Programming port for ICD, Pickit (or clone) programmers / debuggers
- 3.3 volt 800 mA LDO linear voltage regulator
- 2.1 mm CP DC power jack
- 5-9 volt DC input
- 2 Pullup / Pulldown jumpers on IO1 and IO2
- Boot loader jumper for software USB boot loader
- Reset switch can reset just Xbee module or Both MCU and module
- Power LED for controller board
- Power LED for Xbee module
- Network indication LED for Xbee module
- Small 2”x2” size (excluding Xbee module and antenna)
- Standoffs included
- Low power consumption with a variety of power saving methods that depend upon application and implementation.

Hardware diagram



Hardware notes, misc.

Reset switch operation:

The reset switch will reset the Xbee module if you hold it down for one second, and both the controller and the Xbee module if you hold it down for three seconds.

Power LED:

The power LED will stay on as long as power is applied. For low power applications, you must de-solder the power LED to save on current usage.

Xbee power LED:

The Xbee power LED is on whenever the Xbee is on and not is a sleep mode. While in sleep mode this LED will go off.

Xbee net LED (association LED):

The Xbee net LED shows the device association according to its setup parameter A2 and A1. The following table illustrates the modes according to 802.15.4, for Zigbee modes refer to the Digi Xbee manual.

Coordinator: (CE=1, A2=xx, A1=0) on series 1 devices or coordinator firmware for series 2

LED blinks once per second.

End device / Router: (CE=0, A2=0, A1=non zero) on series 1 devices or end/router firmware for series 2.

LED will be solid if failed association to a coordinator or could not find a coordinator.

LED will blink twice per second if associated to a coordinator/router.

End device: (CE=0, A2=0, A1=0) on series 1 devices , N/A for series 2 devices.

LED will blink twice per second. Association is not attempted because A1=0;

Wireless networking considerations

The question weather to choose the Xbee series 1 (802.15.4) modules or the series 2 (Zigbee) modules will present itself when designing your wireless network to your target application. The following information may help you choose which implementation to carry out. Note that Series 1 modules are not compatible with Series 2 modules in the same network.

802.15.4 networks

These networks are the easiest to setup as Zigbee networks have many more options and planning to carry them out. You have to first think how the devices will talk to each other in what order. In 802.15.4 networks we can either talk to all devices at once, or only one at a time. Therefore you have to plan how End-devices will communicate with each other. Will they simply send their data to its Coordinator? Or will it consist of only End-devices in a peer to peer configuration with no coordinator.

A coordinator is a device that manages what network id (PAN), operating channel (CH) and other options a network will operate under. When end devices are set to associate to a coordinator (set A1 association bit to hex 0x0f) then the end device will be configured according to the coordinators decisions (as the coordinator scans for an empty channel on startup and an empty PAN ID). This can be thought of as a dynamic configuration, since if there is interference on that channel, the coordinator will automatically switch to another channel and eventually the end-devices will get that configuration information and follow suit. The A2 parameter must be set on the coordinator to say which parameters will be scanned for on startup. Therefore, on a coordinator the A2 parameter is always nonzero, and the A1 parameter is always zero. A1 parameters are for end-devices.

End devices are devices that either set their own network settings or take orders from the coordinator. End devices have their A2 parameter set to zero, and their A1 parameter set to nonzero if getting configuration information from a coordinator. If setting its own PAN ID, and CH channel then the A1 parameter should be set to zero. Please consult the Xbee’s manual for more information on the associated network parameters and settings.

In 802.15.4 networks, data can only be directly delivered to its destination. Indirect delivery is in the form of a buffer, and waits for the device to poll the coordinator.

Zigbee networks

Zigbee networks can be much more difficult to setup, there are many more parameters and considerations in how many nodes and how many repeaters we may wish to add. Zigbee devices can be a coordinator, end device, or router. Since we have two different firmware's for the Xbee modules based on which device we want it to be, we must consider how to program those devices as we do not currently have a boot loader application for them (we are working on one now!). Therefore, it is currently necessary to obtain an addition Xbee interface board to update the firmware to the specific device function we wish to have (coordinator, router/end device).

Coordinators function similarly to the devices previously mentioned for 802.15.4 networks as Zigbee networks are based on 802.15.4 networks.

Routers can route data and assign network parameters to end devices. Data can take any path in the network to reach its destination as there is an adaptive algorithm to find the best route for data to take. Therefore, we have true indirect messaging capability which differs from 802.15.4 networks.

Zigbee networks are much more adaptive, and can be much bigger in size than 802.15.4 networks.

Please consult the Xbee's manuals for detailed information and comparisons of which networking solution is right for your application.

Xbee module firmware

For Xbee series 1 modules, you generally do not need to update those firmware versions unless you purchased the modules at different times and they are significantly different firmware versions. Both coordinator and end device are on 1 firmware. You may also update the firmware to add new functionality as they come out on www.Digi.com .To update Xbee series 1 firmware, you will need a separate interface board from Digi that can accomplish that function.

For Xbee series 2 modules, each device type (coordinator, router /end device) exist on separate firmwares, and thus the modules require programming to get them to work together as you have planned for your target networking application. We are working on a software boot loader to accomplish that task, however it is not available at this point in time. Therefore, you will need an interface board from www.Digi.com to upgrade the modules firmware to what you want it to be for series 2 modules.

Currently the router/end device firmware exists on one firmware, while the coordinator firmware is separate. The default firmware when you purchase them is the end device/router firmware. You distinguish from an end device/router by setting a sleep mode (SM not 0) for an end device, and (SM=0) for a router.

techFX Zigbee firmware

Using Microchip MPLAB and MCC18 C compiler to edit firmware

You should have the latest version of MPLAB IDE and MCC18 C compiler (both are available as free downloads from www.Microchip.com.

The first thing you should do is copy the firmware you want to a fresh directory.

Then click on the .MCW file to open the workspace.

Immediately go to 'project' then 'build options' to set the directories to your current directories.

Close any C file windows that may be open (because the old directory will show of that file in the title bar and we want to edit the new location not the old ones).

Open up the files you wish to edit and make any changes.

When finished, go to 'project' then 'build all' to compile the project.

Your new *.hex file will be in the new directory location / output folder.

Is using our USB boot loader, load that hex file in techFX tools USB boot loader and program the controller. (remember to move the boot jumper to the on position.

Reset the controller to run the newly programmed code. Do not forgot to move the boot jumper back to the off position.

Firmware selection: which one to use?

Currently, the auto configuration options are geared towards 802.15.4 network options in all of our firmware and can be modified easily reflecting options in the www.Digi.com Xbee manual for Zigbee networks; it is a matter of changing 4-8 values in one initialization function.

TechFX Zigbee controller firmware comes in a variety of flavors and will require some configuration to get your network the way you want it to be. Currently the following firmware configurations are included on the CD:

- **Automatic configuration firmware**
  This firmware contains configuration information to automatically setup the modules in several configurations such as with a coordinator, or strictly end devices.
  This firmware will configure the module EVERY time the controller is started or reset. This is ideal firmware for Coordinators or end devices that are not battery powered. (coordinators should be mains powered)
  This firmware has USB enabled and no power saving cyclic features or idle modes.

- **Manual configuration firmware**
  This firmware does not contain the configuration routines for the Xbee module and therefore, the Xbee modules must be setup using the included Terminal utility program and AT commands. The AT commands are listed in the Xbee manual and can be typed in after command mode entry through the Terminal program. This is ideal firmware for Coordinators or end devices that are not battery powered. (coordinators should be mains powered)
  This firmware has USB enabled and no power saving cyclic features or idle modes.

- **Low power consumption automatic configuration firmware**
  This firmware has the same configuration as the automatic above, however it implements several power saving techniques to lengthen battery life. This is ideal firmware for end devices.
  This firmware has USB disabled and basic power saving cyclic features or idle modes.
  The Xbee unit is operated in pin cyclic mode with wakeup according to the xbee_sleep pin.
  The MCU is operated in primary idle mode and wakes up on a device interrupt.
  This situation is best for a “send only” device that does not need to receive data.
  For a device to receive also, simply change the cyclic sleep mode to one where the xbee wakes itself up, but also uses the MCU pin to sleep and wakeup. Currently this firmware does not do a clock slowdown, however it may be added in the future.

Common firmware features and functions overview

- USB boot loader in firmware
  The USB boot loader is a program that resides on the MCU and will allow you to upload firmware to the controller with no additional hardware needed. The reset and interrupt vectors are remapped accordingly to use this USB boot loader feature.
- auto baud rate detection and configuration
  Every time the controller starts up it will check for an Xbee module response at 200 Kbaud and 9600 baud (new modules come with this default configuration). If it fails at both then the controller will execute the no Xbee module While(1) loop. In this loop you can run functions as a general controller with no Xbee module installed. If the auto baud rate check fails at 200Kbaud but gets a response at 9600 baud, then it will reprogram the module for 200 Kbaud and continue on with Xbee module configuration. If you wish to use a lower baud rate for operation, then you must first execute a change baud rate command in the Terminal utility to 9600 baud and save changes to non-volatile memory. Then you can change the start-up values in the firmware to whatever you like, and change the changebaudrate() function to the new baud rate that you want to have, however make sure you leave the check for the 9600 Baud rate as this will be the check to change the baud rate to the new one you want.
- configuration routines
  Auto configuration routines for Xbee network and sleep configuration depending on the firmware you use. These routines need to be changed to reflect the settings you want your Xbee module to have. These routines run every time the controller starts, and configures the Xbee module. If you do not use these routines (manual configuration) then each module must be programmed with AT commands using command mode in the Terminal utility program.
- circular buffers
  Complete receive circular buffer that is written to by interrupt. You can parse this to check for your custom network commands that you want the End device to act on. The transmit circular buffer is not on an interrupt, but is used solely for communicating with the terminal program. Therefore, transmission is accomplished “on the fly” directly with the putrsusart() function.
- USB communications
  Easy to create your own USB commands and talk to a Visual C# application that you make using our example as a guideline. Microsoft Visual C# Express edition is free.
- Periodic functions available on a timer
  Periodic functions hit every 1 second, 1 minute, 1 hour, and day. You can make something happen each of those times by adding simple code i.e.: such as reading an input and then transmitting it out.
- easy IO configuration in seconds
  As easy as specifying a few equates to what you want each port to be.
- easy network setup in seconds
  As easy as changing network parameters for auto configured firmware.
- very well commented sour code for easy learning
- free version of MPLAB IDE and MCC18 C compiler available from Microchip (student edition)
  Use those free versions to edit and compile the firmware framework after modifications.

Specific firmware features and functions

USB boot loader and remapped vectors

The USB boot loader is a program that resides on the MCU and will allow you to upload firmware to the controller with no additional hardware needed. The reset and interrupt vectors are remapped accordingly to use this USB boot loader feature. The windows portion of the boot loader is located within TechFX tools under the device menu and help for it can be found within that program. The device portion must be programmed using the programming port (ICD, Pickit) but already comes programmed on your chip. If for some reason you accidentally delete this boot loader from the chip, the USB boot loader function will cease to work and the chip must be reprogrammed with a programmer on the programmer port.

The Following code is the vector declarations if you are using the USB boot loader (in “main.c”):

**This is the default firmware setting, if you remove this, then you must use a programmer.

```
#pragma code _RESET_INTERRUPT_VECTOR = 0x000800
void _reset (void)
{
        _asm goto _start-up _endasm
}
#pragma code

#pragma code _HIGH_INTERRUPT_VECTOR = 0x000808
void _high_ISR (void)
{
        _asm goto interrupt1 _endasm
}
#pragma code


#pragma code _LOW_INTERRUPT_VECTOR = 0x000818
void _low_ISR (void)
{
        ;
}
#pragma code
```

The following is the vector declarations if you are not using the USB boot loader (in “main.c”):

```
///
/// this code is the vector mapping without the USB boot loader
///
#pragma code _HIGH_INTERRUPT_VECTOR = 0x000008
void _high_ISR (void)
{
        _asm goto interrupt1 _endasm
}
#pragma code

#pragma code _LOW_INTERRUPT_VECTOR = 0x000018
void _low_ISR (void)
{
        _asm goto interrupt1 _endasm
}
#pragma code
///
/// end remap
///
```

Once you remove the remapped vectors for the USB boot loader, you will have to use a device programmer and that will overwrite the USB boot loader residing on the chip. You will no longer be able to use techFX tools to upload your firmware to the controller.

Auto baud rate detection and change function

The controller firmware default is to set the baud rate and reprogram new modules (9600 baud default) to 200 Kbaud. If you want a lower baud rate for some reason, then that can be changed; see below.

Every time the controller starts up it will check for an Xbee module response at 200 Kbaud and 9600 baud (new modules come with this default configuration). If it fails at both then the controller will execute the no Xbee module While(1) loop. In this loop you can run functions as a general controller with no Xbee module installed. If the auto baud rate check fails at 200Kbaud but gets a response at 9600 baud, then it will reprogram the module for 200 Kbaud and continue on with Xbee module configuration. If you wish to use a lower baud rate for operation, then you must first execute a change baud rate command in the Terminal utility to the baud rate you want and save changes to non-volatile memory. Then you can change the start-up values in the firmware to whatever you like, and change the changebaudrate() function to the new baud rate that you want to have, however make sure you leave the check for the 9600 Baud rate as this will be the check to change new modules to the baud rate you want.

The following is a step by step program flow:

```
-check for a Baud rate of 200 Kbaud

    -(failed)
     Check for factor default Baud rate of 9600 baud.

        -(failed)
         Controller enters into "no module found" While(1) loop and continually executes code
         there.

        -(success)
         Controller has found module at 9600 baud and executes the Changebaudrate()
         function to reprogram controller to 200K baud rate.

    -(success)
     module found at 200K continue on with configuration and other tasks.
```

You can change the detection baud rates and reprogram the module to a custom baud rate.

This is done by 1st reprogramming the module's baud rate through the Terminal utility program. 1st, you enter command mode in the Terminal window by pressing the F1 key 3 times. After receiving an "OK" response in red, you are now in command mode. You need to refer to the Xbee manual for a list of baud rates or you can use a custom hexadecimal value i.e.: "1C200" is 115200 baud. Typing "ATBD1C200<enter>" , then to write changed to non-volatile memory we do a "ATWR<enter>". We should receive an “OK” response to each command. If you do not, then reissue the command again until you do. It may become necessary to enter command mode multiple times to accomplish all of the commands. If you do not write to non-volatile memory then the changes will be lost on the next power up.

Now since we have changed the Xbee to the new baud rate we want, we now go into the firmware and change the start-up values to what we want, and also the changebaudrate function to what we want.

The following is the code to call the start-up function (in “main.c”) to initialize the techFX ZIgbee controller at that baud rate to conduct the baud rate test, there are 2 of these to change but do not change the (9600) baud rate one. Note that the value is in decimal not hex.

```
//auto baud rate detection
//
//start the USART module at 200Kbaud baud rate
//
        startUsart(200000);
//
//
```

After changing 2 of those function calls, we now change the changebaudrate() function found in “user.c” ; the change that needs to be made is in red, and it is the new baud rate you want in hexadecimal format:

```
void changebaudrate(void)
{

//wait the guard time of 1 second before command mode entry
        for (i=0;i<1060;i++)
        {
                Delay1KTCYx(13);
        }

//put in command mode
        putrsuart( "+++" );
        getsuart(temp_buffa,3);

//set baud rate to 200 Kbaud command this value is in hexadecimal not decimal
        putrsuart("ATBD30d40\r");
//get rcv_buffa also fulfills guard time for command mode 1 second after wait time
        getsuart(temp_buffa,3);
```

```
                putrsuart("ATWR\r");
                getsuart(temp_buffa,3);

                putrsuart("ATCN\r");
                getsuart(temp_buffa,3);
        }
```

If you restart and the controller doesn’t respond to the command mode entry (F1 key 3 times) in the Terminal utility, then you must go back into the firmware and try lowering the “startUsart(200000);” value by 2%. Sometimes the Xbee firmware will set a lower baud rate value in the module for custom baud rates, and this has been noted to be lower by about 2% but you can experiment to find out until it is working in the Terminal utility.

Auto configuration routines

The auto configuration routines are only found in the auto configuration versions of the firmware. They consist of an Xbee initialization function and an Xbee sleep parameter initialization function. You can tailor these functions to meet that nodes setup needs. It is suggested that you do not write to non-volatile memory when doing these changes, and instead execute the “ATAC\r” command to apply all changes to the module. That way, if a value makes the module unstable a simple reset will bring back the original values from memory and the module will not be ruined. The following is the function calls in “main.c” to the functions in “user.c”:

```
initxbee();

sleep_config_xbee();
```

The following are examples of the 2 function above which can be found in the “user.c” file:

```
//
// init xbee function
//
// refer to maxstream module for AT commands to configure the xbee module //every
time the controller starts up
//
// the AT commands are follow by a "\r" which is a carriage return
// follow the format below
//
//  DONT PUT SLEEP INSTRUCTIONS IN HERE!!!!
//  PUT THEM IN THE XBEE_SLEEP_CONFIG FUNCTION BELOW
//

void initxbee(void)
{
        //this is a guard time delay that is required for command mode entry
        // do not remove
        for (i=0;i<1060;i++)
        {
                Delay1KTCYx(30);
        }

//this is command mode entry command do not remove
putrsuart( "+++" );
getsuart(temp_buffa,3);


// set auto associate to off for end device to connect to a coordinator
putrsuart("ATA100\r");
getsuart(temp_buffa,3);

// set association commands for coordinator, and scan settings
putrsuart("ATA20\r");
getsuart(temp_buffa,3);

// set channel to C
putrsuart("ATCH0C\r");
getsuart(temp_buffa,3);

//panid
putrsuart("ATID33\r");
getsuart(temp_buffa,3);

//destination addy destination high
putrsuart("ATDH0\r");
getsuart(temp_buffa,3);

//destination addy low value
putrsuart("ATDL1\r");
getsuart(temp_buffa,3);

//my addy
putrsuart("ATMY3\r");
getsuart(temp_buffa,3);

//node identifier for this node
putrsuart("ATNISilicon3\r");
getsuart(temp_buffa,3);

//coordinator disable (end device for this node)
putrsuart("ATCE0\r");
getsuart(temp_buffa,3);

// initialize with changes
putrsuart("ATAC\r");
getsuart(temp_buffa,3);

//exit command mode
putrsuart("ATCN\r");
getsuart(temp_buffa,3);



}

//
```

```
// END xbee init function
//



//
// xbee sleep configuration
// doesnt save to non volatile memory
// so when power is reset, easier to enter command mode again
//
void sleep_config_xbee(void)
{
        //this is a guard time delay that is required for command mode entry
        // do not remove
//sleep mode sucks

        for (i=0;i<1060;i++)
        {
                Delay1KTCYx(30);
        }

//this is command mode entry command do not remove
putrsuart( "+++" );
getsuart(temp_buffa,3);

// turn off sleep mode
putrsuart("ATSM0\r");
getsuart(temp_buffa,3);

// time before sleep x 1 mS curently 50 mS
putrsuart("ATST0004\r");
getsuart(temp_buffa,3);

// sleep period x 10 ms  //10 second wakeup (10 sec = 0x003e8) (1 sec=0x64)
putrsuart("ATSP0002\r");
getsuart(temp_buffa,3);

// initialize with changes
putrsuart("ATAC\r");
getsuart(temp_buffa,3);

//exit command mode
putrsuart("ATCN\r");
getsuart(temp_buffa,3);


}
```

```
//
// END xbee sleep config function
//
```

Note that all command instructions are followed by a “\r” which is a carriage return. Also note that we have the getsuart() function to get the response from the controller. This must be executed after every command we send. We dot not do “ATWR” commands here since we don’t want these values written to non-volatile memory. Instead we do the “ATAC” command and “ATCN” command to apply changes and exit the command mode. This way we wont ruin any modules with a bad setup, instead we simply restart and we have the old values from memory again and we can make corrections to the firmware.

The above example will initialize an end device with no auto association to a coordinator and will turn sleep mode off. It also sets the node identifier and destination and my (source node) address. The PAN id is the network id.

Please refer to the Digi Xbee manual for a complete list of commands and descriptions on how to use them correctly.

IO configuration

The loadconfigfriendly() function is responsible for setting up the controller configuration options and the input / output pins accordingly. As you can see from the code comments, 1st you must change the ADCON1 register accordingly to the table above it. This defines how many digital and analog pins you want. Then after that you define the TRIS settings by setting each pin to an input or output based on the ADCON1 configuration that you just did. Then we set the default “start-up” value (also called latch value) in the next section. Please note that the I2C bus speed configuration setting is found in “i2cstuff.c” and is commented on the values to use for 100Khz, 400 Khz and 1 Mhz (default value). The function appears in “user.c” as follows:

```
//
//  LOADCONFIGFRIENDLY function
//
//          PIN EQUATES AND PIN BEHAVIOR SETUP IN HERE
//          FOR PINS ON SCREW TERMINALS IN THIS DESIGN
//
//  change from input to output to analog
//
//  and default values on start-up (high or low)
//
//          these will only work
//  if you haven't changed the register values in loadconfig()
//
void loadconfigfriendly( void )
{

        //
        // set # of digital and analog pins here
        //
        // if you set something as analog, you must make it an ANALOG_PIN below
        //also!!!
        //
        // ________________________________________
        //   0x0F = ALL io pins digital
        //   0x0E = IO1 analog , All else digital
        //   0x0D = IO1,IO2 analog, All else digital
        //   0x0C = IO1,IO2,IO3 analog, All else digital
        //   0x0B = IO1,IO2,IO3,IO4 analog, All else digital
        //   0x0A = IO1,IO2,IO3,IO4,IO5 analog, All else digital
         / / 0x09 =         IO1,IO2,IO3,IO4,IO5,IO6 analog, All else digital
         //   0x08 = IO1,IO2,IO3,IO4,IO5,IO6,IO7 analog, All else digital
         //   0x07 = IO1,IO2,IO3,IO4,IO5,IO6,IO7,IO8 ALL analog lines.
        //
```

```
//

            ADCON1=0x0F;


//
// TRIS register equates
//
// change these values to the following for each pin below
//
// INPUT_PIN   = makes the pin an input
// OUTPUT_PIN  = makes the pin an output
// ANALOG_PIN  = makes the pin an analog pin
//

            IO1_TRIS = OUTPUT_PIN;
            IO2_TRIS = OUTPUT_PIN;
            IO3_TRIS = OUTPUT_PIN;
            IO4_TRIS = OUTPUT_PIN;
            IO5_TRIS = OUTPUT_PIN;
            IO6_TRIS = OUTPUT_PIN;
            IO7_TRIS = OUTPUT_PIN;
            IO8_TRIS = OUTPUT_PIN;

//
// default pin values on bootup
//
//          settings are as follows:
//
//          1: output is a high (+5v)
//  0: output is a low (ground)
//
//  put a 0 for INPUT_PIN and ANALOG_PIN
//

            IO1 = 0;
            IO2= 0;
            IO3 = 0;
            IO4= 0;
            IO5 = 0;
            IO6= 0;
            IO7 = 0;
            IO8= 0;



//
// I2C bus speed
//
```

```
        // default is 1 Mhz.
        //
        // please goto i2cstuff.c to change this setting.
        //

}

//
// END loadconfigfriendly function
//
```

Periodic functions

Periodic functions hit every 1 second, 1 minute, 1 hour, and day. You can make something happen each of those times by adding simple code i.e.: such as reading an input and then transmitting it out. The periodic functions are several different ones all located in the same section, and each function name is dependant on its execution time frame. In the following example, the code executed once per minute is the transmit functions putrsuart() which transmits a string on the Xbee and putsuart() which transmits variables and ports. Note we are sending a “!” as the 1st character, that is interpreted in our Visual C# data logger as a command code to replace that character with the current timestamp. This is a smart design since we don’t have to keep current accurate time on the controller, we let the computer do that job and thus all of our controllers are synced since data is recorded when it is received by the data logger application. So the output seen in the Data logger application is “[timestamp] [NODE2] [IO1 pin current state 0 for low and 1 for high] <carriage return>”. The following is the example function code as seen in “user.c” near the end. Note that you should leave a semicolon “;” in a function that is empty:

```
//
//INSERT YOUR PERIODIC FUNCTIONS IN HERE inside the correspond time function that
you want
//
// functions are executed according to the time in the function name
//

void function25mS(void)
{
        ;
}
void function100mS(void)
{
        ;
}
void function1Sec(void)
{
        ;
}
void function1min(void)
{
        putrsuart("![NODE2]  ");    // use this function when we use quotes
        data= port_to_ascii(IO1);  //convert the hex port value to an ascii character
        Writeuart_1byte(data);   // writes that 1 ascii character to transmitter
        putrsuart("\r");    //put a carriage return to transmitter
        ;
}
void function1hour(void)
{
        ;
}
void function1day(void)
{
        ;
}
//
// END OF PERIODIC FUNCTIONS
//
```

Circular buffers, received data, and transmitted data

There is a 128 byte circular receive buffer, and a 32 byte circular transmit buffer. These buffers are used by the Terminal utility and data logging example through the USB port to send and receive data. Therefore, if you are using it as part of your command received parser or user the transmit buffer to send data as part of your application, then you should do so when the USB cable is not connected as it will interfere with data and communications with the device.

The receive circular buffer is written to on interrupt from the USART module in the order that data is received. The buffer will discard data when the buffer is full, if more is received and it is not acted on. Therefore, if you are created a command interpreter it should check the receive buffer periodically such as in the Timer1 interrupt loop and parse the commands from the receive buffer and act on them. One such way to do that is you 1st have to create your own serial protocol i.e.: like a '#' character starts a command; therefore, we would just check for that character and parse the following characters in order and then run the command when we have read the command sequence.

The transmit buffer is written to on USB writes from the USB data that is sent. It is read from to be transmitted in the Timer 1 interrupt handler. If you are not using the buffer, then nothing will be transmitted as it is empty. To send data instantaneously you do not have to use the buffer, simply use the putrsuart() and Write_1byte() functions to transmit data without using the buffer. When the transmit buffer is full additional data will not be written to it and will be discarded until the buffer is not full.

USB communications and commands

The USB service routines are handled by polling, however they run continuously in the main While(1) loop for normal Xbee operations in “main.c” as follows:

```
USBTasks();
ProcessIO();
```

New commands can be created by adding the new command to the enumeration in “user.h” file. As you can see, we have added a new command below called “NEW_ONE” which occurs when we receive the hex byte 0xB0.

```
enum
{
                bootmodecheck       = 0x44,
                READ_VERSION        = 0x00,
                getrcv_buffa        = 0xA0,
                RESET               = 0xFF
               NEW_ONE              =0xB0
}CMD;
```

Now we head over to “user.c” and look at the ServiceRequests() function. We want to receive an 0xB0 from our Visual C# application that we are making, then send out 2 variables back to the Visual C# application when that happens. So we have added the following code to the “switch” statement in that function:

```
switch(dataPacket.CMD)
    {
                case New_ONE:
                        dataPacket.byte[1]=data;
                        dataPacket.byte[2]=data2;
                        counter=0x03;
                        break;

                case bootmodecheck:
                        dataPacket._byte[1]=0x05;
                        counter=0x02;
                        break;
```

The code in red is executed when the 0xB0 byte is received from the computer through USB. Note that we did not write to the array dataPacket.byte[0]. Note that the dataPacket array is used for sending and receiving in USB and therefore byte 0 still has the 0xB0 command in it we have received. So we assign byte 1 to the variable data and byte 2 to the variable data2 then set our counter. The counter is in hex of how many bytes we are sending back to the computer. Since the last byte we assigned to was byte 2 of dataPacket, then we have 0,1,2 which is 3 bytes total. So counter=0x03.

That's it for the firmware side! All you have to do is create the code in Visual C# that sends 0xB0 and receives 3 bytes and does something with them and you have created your own USB command.

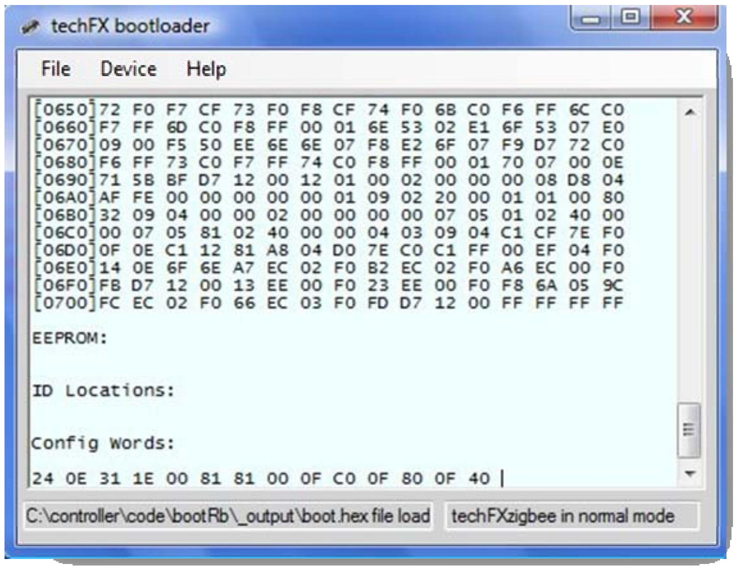
So the command flow goes like the following:

Computer sends this:	<0xb0>		
Controller sends this:	<0xb0>	<data>	<data2>

On the Visual C# side (using our example framework), we will be setting the send_buffer[0] byte to our command which is 0xB0. Then we will set the send length to '1', and the received length to 3. You could pass them in an array to the calling function in your windows application.

TechFX Zigbee Tools

Using the USB boot loader



```
[0650]72 F0 F7 CF 73 F0 F8 CF 74 F0 6B C0 F6 FF 6C C0
[0660]F7 FF 6D C0 F8 FF 00 01 6E 53 02 E1 6F 53 07 E0
[0670]09 00 F5 50 EE 6E 6E 07 F8 E2 6F 07 F9 D7 72 C0
[0680]F6 FF 73 C0 F7 FF 74 C0 F8 FF 00 01 70 07 00 0E
[0690]71 5B BF D7 12 00 12 01 00 02 00 00 00 08 D8 04
[06A0]AF FE 00 00 00 00 00 01 09 02 20 00 01 01 00 80
[06B0]32 09 04 00 00 02 00 00 00 00 07 05 01 02 40 00
[06C0]00 07 05 81 02 40 00 00 04 03 09 04 C1 CF 7E F0
[06D0]0F 0E C1 12 81 A8 04 D0 7E C0 C1 FF 00 EF 04 F0
[06E0]14 0E 6F 6E A7 EC 02 F0 B2 EC 02 F0 A6 EC 00 F0
[06F0]FB D7 12 00 13 EE 00 F0 23 EE 00 F0 F8 6A 05 9C
[0700]FC EC 02 F0 66 EC 03 F0 FD D7 12 00 FF FF FF FF

EEPROM:

ID Locations:

Config Words:

24 0E 31 1E 00 81 81 00 0F C0 0F 80 0F 40
```

C:\controller\code\bootRb_output\boot.hex file load | techFXzigbee in normal mode

The USB boot loader is used in conjunction with the firmware boot loader which is pre-programmed onto your controller when you receive it. If you should accidentally overwrite the USB boot loader, then your controller will have to be reprogrammed with the USB boot loader firmware through the ICD programming port on your controller.

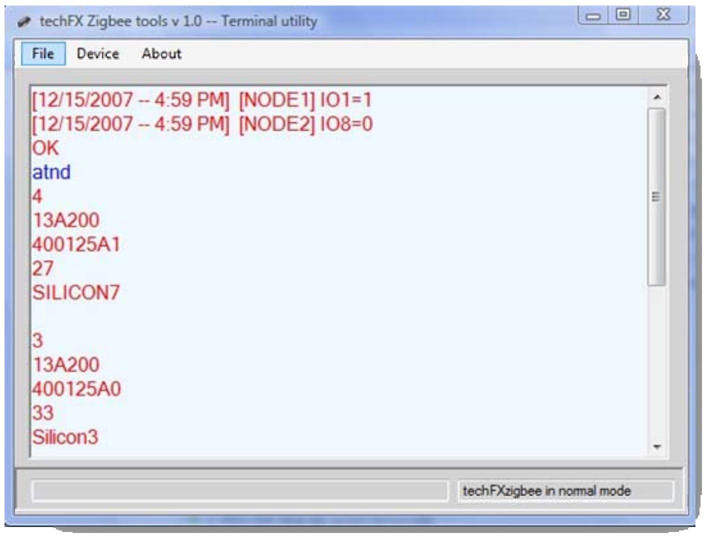
For detailed instructions on how to use the USB boot loader windows application, please refer to the ‘help’ menu.

The boot loader can upload *.hex files only.

Using the Xbee firmware boot loader

This feature will be added in the future for Xbee series 2 modules only. Xbee series 1 modules will not have support for this feature as both coordinator and end devices exist on the same firmware version for series 1 devices.

Using the Terminal utility



```
[12/15/2007 -- 4:59 PM]  [NODE1] IO1=1
[12/15/2007 -- 4:59 PM]  [NODE2] IO8=0
OK
atnd
4
13A200
400125A1
27
SILICON7

3
13A200
400125A0
33
Silicon3
```

techFXzigbee in normal mode

How to interface to your techFX Zigbee board with a Xbee module installed

1. install the module triangle facing outward of the board in the techFX controller Xbee sockets.

2. connect the module to a USB port, and install driver--after powering device on--from the installation CDROM.

3. It may take about 15 seconds for the program to connect the techFX controller since the initialization occurs 1st for the Xbee module that is installed.

4. If your bootload jumper is in off position, 'techFX in normal mode' is displayed in the status bar. If in the 'on' mode then boot mode is entered instead. You want to be in “normal” mode to use the Terminal.

5. To enter command mode, press F1 three times, this may have to be repeated. Make sure you wait 2 seconds before and after pressing three times while not pressing any other keys. <wait 2 seconds> <press F1 three times> <wait 2 seconds more for response>

6. You should get an OK response from the Xbee unit that signals a good command.

7. Now you may manually configure your module (by sending AT commands in command mode) only if you are using the Manual configuration firmware. Using the auto configuration firmware, you must edit the values in the firmware files and compile them using Microchip MCC18 C compiler, then program the firmware to the controller.

8. Whatever is typed into the console while not in command mode, and with a techFX Zigbee with Xbee in normal mode, will be send out over the

network according to the Xbee modules network paramters (broadcast mode or DH, DL paramteres). Data received will be displayed in RED, and data sent will be displayed in BLUE.

NOTE: above you can see some data received from 2 different NODE’s. Those are periodic functions occuring every minute.

After that we have a very useful command “ATND” which does a node scan on the channel and PAN the controller is on. All nodes on that channel and PAN will respond with 5 lines of information including the following:

Line1:	“MY” address
Line2:	serial # high
Line3:	serial # low
Line4:	Signal strength
Line5:	“NI” Node identifier name

This command can be used to verify correct configuration of other nodes and to communicate with them.

AT commands to use in command mode

For a complete reference and list of commands, please refer to the Xbee manual from www.digi.com.

ATND<enter>

Will do a node scan and show all Xbee modems on the current channel and PAN.

ATCN<enter>

Exit command mode

ATWR<enter>

Write values to non-volatile memory

ATAC<enter>

Apply changes and restart********

***only on series 1 devices

Example configurations for 802.15.4 networks

Simple 2 Node network static config

Both nodes as end-devices with a static configuration.

Node1:

```
ATA100
ATA200
ATCH0C
ATNINODE1
ATID01
ATMY01
ATDH00
ATDL02
ATCE0
```

Node2:

```
ATA100
ATA200
ATCH0C
ATNINODE2
ATID01
ATMY02
ATDH00
ATDL01
ATCE0
```

3 Node network (1 coordinator) dynamic config

1 coordinator with dynamic configuration hooked up to USB port.

2 end devices which get configuration from coordinator, both end devices send data to coordinator

Coord:

```
ATA100
ATA206
ATCH0C
ATNICOORD
ATID01
ATMY01
ATDH00
ATDL03
ATCE1
```

Node1:

```
ATA10F
ATA200
ATCH0C
ATNINODE1
ATID01
ATMY01
ATDH00
ATDL01
ATCE0
```

Node2:

```
ATA10F
ATA200
ATCH0C
ATNINODE2
ATID01
ATMY02
ATDH00
ATDL01
ATCE0
```

Example configuration for Zigbee networks

4 Node network (1 coordinator, 1 router, 2 end devices)

1 coordinator which has 2 nodes talking to it and 1 router

1 router for future expansion, currently doesn't have child devices

2 nodes which connect to parent coordinator.

USB HOST Coordinator NODE 1:

```
ATID33
ATNINODE1
```

***uses coordinator firmware

End device NODE2:

```
ATD61
ATD71
ATDL0
ATDH0
ATID33
ATNINODE2
ATSM1
```

**uses end device/router firmware with SM not 0 defines an end device.

End device NODE3: (talks to coordinator)

```
ATD61
ATD71
ATDL0
ATDH0
ATID33
ATNINODE3
ATSM1
```

**uses end device/router firmware with SM not 0 defines an end device.

Router ROUTER1:   (possibly for future nodes, talks to coordinator)

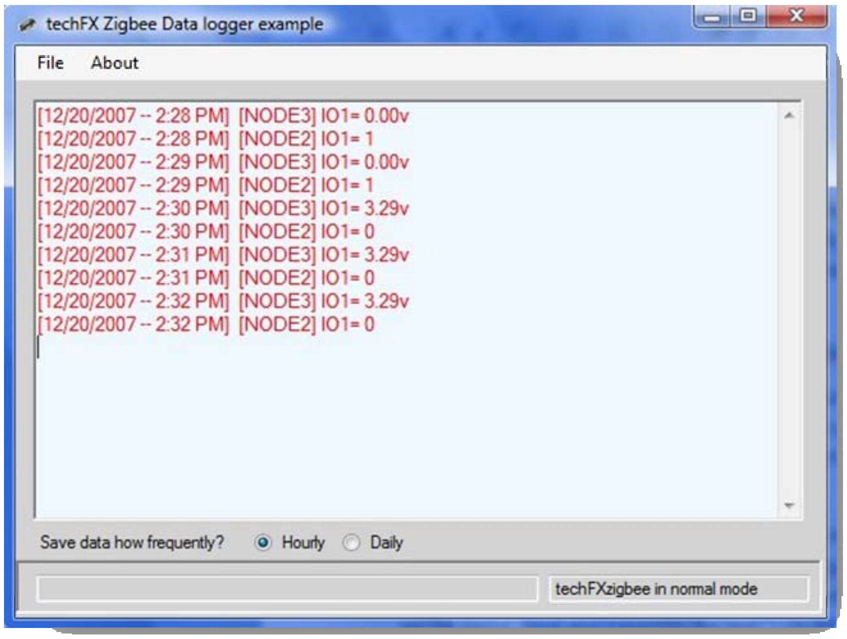
```
ATDL0
```

```
ATDH0
ATID33
ATNIROUTER1
ATSM0
```

***uses end device/router firmware with SM=0 defines a router

Example applications

Data logger application (Visual C# and firmware)



```
[12/20/2007 -- 2:28 PM]  [NODE3] IO1= 0.00v
[12/20/2007 -- 2:28 PM]  [NODE2] IO1= 1
[12/20/2007 -- 2:29 PM]  [NODE3] IO1= 0.00v
[12/20/2007 -- 2:29 PM]  [NODE2] IO1= 1
[12/20/2007 -- 2:30 PM]  [NODE3] IO1= 3.29v
[12/20/2007 -- 2:30 PM]  [NODE2] IO1= 0
[12/20/2007 -- 2:31 PM]  [NODE3] IO1= 3.29v
[12/20/2007 -- 2:31 PM]  [NODE2] IO1= 0
[12/20/2007 -- 2:32 PM]  [NODE3] IO1= 3.29v
[12/20/2007 -- 2:32 PM]  [NODE2] IO1= 0
```

Save data how frequently? [✓ Hourly] Daily

techFXzigbee in normal mode

The data logger application gives an excellent example of a three node network. While the example does not use a coordinator to dynamically configure the network (for series 1 devices), it shows you how to statically configure it using all End device elements. Even the USB-host element which connects to the computer is an end device. For the series 2 firmware, it uses a coordinator to configure the network.

The USB host (node 1) is an autoconfiguration in firmware, and regular power settings.

The other two end devices (node 2 and node 3) use the Low power autoconfigure firmware, which has both Xbee units going to sleep on a pin wakeup (SM mode 2 for series 1 and SM mode 1 for series 2) where the controller tells the Xbee when to sleep and when to wake up. The controller MCU itself goes into low power idle mode and shuts down its USB to save power. Node 2, and Node 3 both wakeup the Xbee module once a minute to execute the one minute periodic function, although the processor is coming out of idle mode approximately every 5 mS to increment the counters and check for data. Note that the 2 firmwares for the series 1 and series 2 examples are identical except for their Initialization and Sleep settings in those 2 functions.

Looking at the above output, you can see the two end devices (node 2, node3) are sending information once per minute (the Xbee's of node 2, and node 3 sleep until they wakeup at every minute to send data—then they go back to sleep. Node 2 reads the value of IO1, which is set for digital input and transmits the text ”! [NODEx] IO1= x”. You can see in our datalogger Visual C# source code that our program interprets the “!” and translates it into the current computer time. This saves us a lot of transmitting and the chore of keeping a current system time on the controllers. To setup the pin for digital IO, we configured the values in the loadconfigfriendly() function. For Node 3, we have it configured as an analog input and changed the ADCON1 value accordingly in the same function on that controller. Node 3 reads the analog value and runs the function to convert it to a string then sends it to the Xbee unit for transmission once per minute.

Remember IO1 and IO2 are on pullup / pulldown jumpers. So we don’t need to put anything on the input to change the value of the input pin IO1; we simply mode the jumper to high or low and observe the change in the datalogger output window. This will verify that the unit is actually sending the correct data and which unit is which (in case you lose track). This firmware can easily be modified to extend to 50 end devices if needed, and it can be done very quickly. You can also move the IO1 jumper for NODE 3 as it just will give you the full value of approximately 3.28 – 3.3 volts or 0.00 for ground (low setting on the jumper).

How does the controller go into idle mode to save power

The controller goes into idle mode by executing 1 command in the main.c normal run While(1) loop. 1st you set the OSCCON register IDLEN but to 1 as follows to signify that you want idle mode and not sleep mode:

```
OSCCONbits.IDLEN=1;
```

Next we execute the sleep command which will cause the processor to go into idle mode. The only thing that will wake it up is the interrupts of Timer 1 which is running for our periodic functions, and the Xbee module sending data to the processor (the RX side of the USART module interrupt):

```
Sleep();
```

We use the following code to make sure the processor external crystal clock is back up to speed if we are doing a clock slowdown to save power (it will loop until it is ready, we do this in the beginning of our periodic function):

```
while (!OSCCONbits.OSTS);
```

How do you put the Xbee modules to sleep

The Xbee module is put to sleep depending on the SM mode you set it to in the configuration. For this example we are using the SM=2 mode (for series 1) and SM=1 (for series 2) which is a pin sleep mode. We toggle our MCU pin by doing the following to put the Xbee to sleep:

```
xbee_sleep=1;
```

To wake up a module, we do the following:

```
xbee_sleep=0;
```

There is a wakeup time, the timer it takes for an Xbee module to be able to perform commands and transmit data after waking up. Therefore in our periodic functions we do the following to wait for the Xbee module to signal that it is ready to receive data (we watch the CTS line):

```
xbee_sleep=0;  //turn sleep mode off to wakeup xbee module
while(xbee_cts);  //wait until CTS line goes to 0 signaling xbee ready
```

And the Xbee goes back to sleep since we put the xbee_sleep=1 in the main While(1) loop as we did the Sleep() function.

How do you transmit a string or a digital port value from the Nodes?

The nodes transmit using the timer1 interrupt which triggers the periodic functions. We put the following code under the one minute periodic function:

```
while (!OSCCONbits.OSTS);   //wakeup code for a low power node
xbee_sleep=0;               //wakeup code for a low power node
while(xbee_cts);            //wakeup code for a low power node
putrsuart("![NODE3] IO1= ");
data= port_to_ascii(IO1);
Writeuart_1byte(data);
putrsuart("\r");
```

Upon timer 1 hit matching to 1 minute incement (every minute), it waits for the processor clock to become stable. Then it sets the xbee_sleep pin to zero which wakes up the xbee module. The while(xbee_cts) waits for the xbee to signal a clear to send which is bringing the xbee_cts line low to zero. Once that happens we send the data with the putrsuart() function. The next line reads the IO1 pin and converts the resultant value to an ascii value. The Writeuart_1byte() function writes a single data byte to be transmitted. We then send a carriage return to help format our text in the data logger since we want one message per line.

How do you transmit an analog port value as a string?

We have put together several functions to make this task easy. In the above datalogger example you can see Node 3 sending the IO1 pin value by moving the jumper high or low and observing the output on the logger on the next minute transmission.

The first thing to do is change your configuration in loadconfigfriendly() function. Remember we first set the ADCON1 value to the chart in the comments above it to how many analog and digital lines we want. We have to go by the table, that is the only way it works. Next we go below that and set each pin as an input pin, output pin, or analog pin. Then we set our default load up value (0 for analog or input).

The following code is executed (in addition to the standard wakeup code for this example as indicated by the red comments). We execute the a2dinputread() function sending it the IO pin number we want from 1 through 8. We then run the a2dstring()function passing the same int (interger) that was returned from the previous function. This will write the converted value to the "temp_buffa" as x.xx for a total of 4 ascii characters to transmit. We then do our formatting and send out the temp_buffa array, and our "v" + carriage return.

```
while (!OSCCONbits.OSTS);    //wakeup code for a low power node
xbee_sleep=0;                //wakeup code for a low power node
while(xbee_cts);             //wakeup code for a low power node
d= a2dinputread(1);  //do a analog input read of IO1
a2d2string(d);   //converts result to a string in temp_buffa
putrsuart("![NODE3] IO1= ");
 putsuart(temp_buffa);
putrsuart("v\r");
```

The screen output is shown above in the picture.

How does the data logger save the data?

The data logger saves the data every hour or every day depending on the radio button you check below the rich text box. The data is saved into a file with the name of the file being the current file time code from the system time, and residing in the same directory that the application is run from. The data file is saved in a rich text format the same way it is displayed in the rich text box in the application. It can easily be imported into an excel spreadsheet by using space delimiters, or you can change the data is sent to include a comma for comma delimited columns.

Example command interpreter (sending commands out to nodes)

This example will be available soon.

Example for low resolution CCD camera

This example will be available in the future.

Example for low grade audio

This example will be available in the future.

Low power modes and power saving tips

Getting your battery powered nodes to consume the least amount of power is the design goal for a lot of Zigbee / 802.15.4 networks. Below are some of the ideas and suggestions to get you to achieve the lowest possible power level while maintaining the highest amount of functionality:

- MCU idle mode lets the main processor run at the full 12 MIPS / 48 MHz so you can still run a full speed Timer1 and periodic functions, and enjoy power saving features. If you turn off the USB unit also you can save about 10 mA of current draw.
- MCU sleep mode is the best saving feature, it stops all the processor clocks except timer 1 can run off the interal oscillator if you configure it correctly. You can also run off the watchdog timer and wake off of that to count increments for periodic functions. You can save almost 20 mA by using the sleep mode. This presents the best savings from the processor point of view.
- Xbee sleep mode using pin wakeup is best used when you want the processor to control when to send data to your USB host or coordinator. This system is best suited for a send only system, or you can make a command interpreter to regularly send a check command from the node when it is awake, then this will cause the USB host or coordinator to send the data is has to toggle items or send commands to that particular node; however it will only do this when the processor wakes up the xbee unit (SM mode 2).
- Xbee sleep mode using cyclic sleep periods allow the Xbee unit to dictate when it wakes up to receive and send data, and can be adjusted with the sleep mode config function or via the terminal utility if using manual configuration. In this mode, the Xbee module wakes up regularly to poll the coordinator or check to see if there is any data to be received. This mode can consume lots of power if you don’t set the sleep length to a high value. However, setting the sleep value to a high value will cause you to lose data also. For this reason, a polling ideology with pin wakeup using a command interpreter might be ideal and save the most amount of power.
- Xbee sleep mode using cyclic sleep and pin wakeup allows both the Xbee unit to wake up to check the coordinator and the processor to wake up the xbee unit. You may encounter a problem with the Xbee unit triggering the USART RX interrupt as it requires a serial break to wakeup if the processor is in sleep or idle mode. The solution to this would be to send a zero as the 1st bit and the rising edge should cause the MCU to catch it as an interrupt although this method has not been tested as of yet.
- Disconnecting the power LED will cause you to save a little power, as a power LED may have no use in nodes. You can observe the unit is on when the modules wakeup the Xbee module LEDs will light up accordingly. The Xbee module power LED can be disconnected also by desoldering the LED.
- Clock frequencies during Idle mode. If you are using the receive interrupt it is not suggested that you down clock the device or switchover to an internal oscillator to save power, this is because the baud rate is determined at that clock rate. Therefore, is you are only using periodic functions then it is possible to do a clock switchover and still run pheripherals, namely timer 1, and then

once the interrupt hits you switch back over to your main external oscillator and run full speed and restart the USART module at 200 Kbaud to send the data you need. This method will allow you to approach the same power savings as sleep mode without stopping all oscillators (system clocks).

Electrical characteristics

Maximum current draw for MCU	200 mA
Maximum current draw per MCU pin	25 mA
Maximum controller current draw	800 mA
Maximum DC input	9 Volts
Minimum DC input (as specified In LD1117 datasheet)	4.1 Volts
Low power mode in sleep with Xbee sleep	~ approx 5 mA
Low power mode in idle with Xbee sleep	~approx 11 mA

*** low power modes depends on a variety of factors, and have lots of options.