Development Kit For the PIC® MCU

Exercise Book

Wireless - Ember ZigBee™ Edition

March 2010

Custom Computer Services, Inc.
Brookfield, Wisconsin, USA
262-522-6500

Recognized Microchip Third-Party Tool Provider

Custom Computer Services, Inc. proudly supports the Microchip brand with highly optimized C compilers and embedded software development tools.

1 UNPACKING AND INSTALLATION

Inventory

- ❑ Use of this kit requires a PC with Windows 95, 98, ME, NT, 2000 or XP. The PC must have a spare 9-Pin Serial or USB port, a CD-ROM drive and 75 MB of disk space.
- ❑ The diagram on the following page shows each component in the Wireless - Ember ZigBee™ Edition kit. Ensure every item is present.

Software

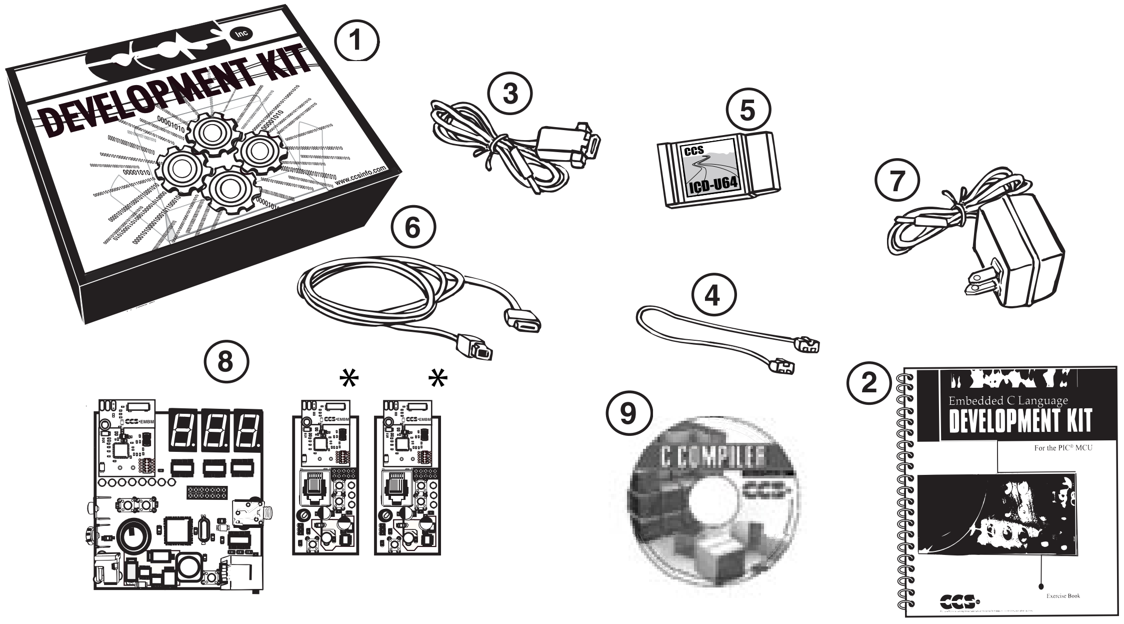
- ❑ Insert the CD into the computer and wait for the installation program to start. If your computer is not set up to auto-run CDs, then select **My Computer** and double-click on the CD drive.
- ❑ Click on **Install** and use the default settings for all subsequent prompts by clicking NEXT, OK, CONTINUE…as required.
- ❑ Identify a directory to be used for the programs in this booklet. The install program will have created an empty directory **c:\program files\picc\projects** that may be used for this purpose.
- ❑ Select the compiler icon on the desktop. In the PCW IDE, click **Help>About** and verify a version number is shown for the IDE, PCH, and PCM to ensure the software was installed properly. Exit the software.

Hardware

- ❑ Connect the PC to the ICD(5) using the USB cable.(1) Connect the prototyping board (8) to the ICD using the modular cable. Plug in the DC adaptor (7) to the power socket and plug it into the prototyping board (8). The first time the ICD-U is connected to the PC, Windows will detect new hardware. Install the USB driver from the CD or website using the new hardware wizard. The driver needs to be installed properly before the device can be used.
- ❑ The LED should be red(2) on the ICD-U to indicate the unit is connected properly.
- ❑ Run the Programmer Control Software by clicking on the CCSLOAD icon on the desktop. Use CCSLOAD Help File for assistance.
- ❑ The software will auto-detect the programmer and target board and the LED should be illuminated green. If any errors are detected, go to Diagnostic tab. If all tests pass, the hardware is installed properly.
- ❑ Disconnect the hardware until you are ready for Chapter 3. Always disconnect the power to the Prototyping board before connecting/disconnecting the ICD or changing the jumper wires to the Prototyping board.

(1) ICD-S40 can also be used in place of ICD-U. Connect it to an available serial port on the PC using the 9 pin serial cable. There is no driver required for S40.

(2) ICD-U40 units will be dimly illuminated green and may blink while connecting.



① Storage box
② Exercise booklet
③ Serial PC to Prototype board cable
④ Modular cable (ICD to Prototyping board)
⑤ ICD unit for programming and debugging
⑥ Serial (or USB) PC to ICD cable
⑦ AC Adaptor (9VDC)
⑧ Two Sensor boards with PIC16F886 processor chips and EM260 modules, and one Base station with a PIC18LF4620 processor chip and EM260 module
⑨ CD-ROM of C compiler (optional) and Ember Documentation

*Note: 9V battery required or can be powered with Ember Insight Adapter (included are 2 9V batteries)

2 USING THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Editor

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>Open>Source File**. Select the file: **c:\program files\picc\examples\ex_stwt.c**
- ❑ Scroll down to the bottom of this file. Notice the editor shows comments, preprocessor directives and C keywords in different colors.
- ❑ Move the cursor over the **Set_timer0** and click. Press the F1 key. Notice a Help file description for set_timer0 appears. The cursor may be placed on any keyword or built-in function and F1 will find help for the item.
- ❑ Review the editor special functions by clicking on **Edit**. The IDE allows various standard cut, paste and copy functions.
- ❑ Review the editor option settings by clicking on **Options>Editor Properties**. The IDE allows selection of the tab size, editor colors, fonts, and many more. Click on **Options>Toolbar** to select which icons appear on the toolbars.

Compiler

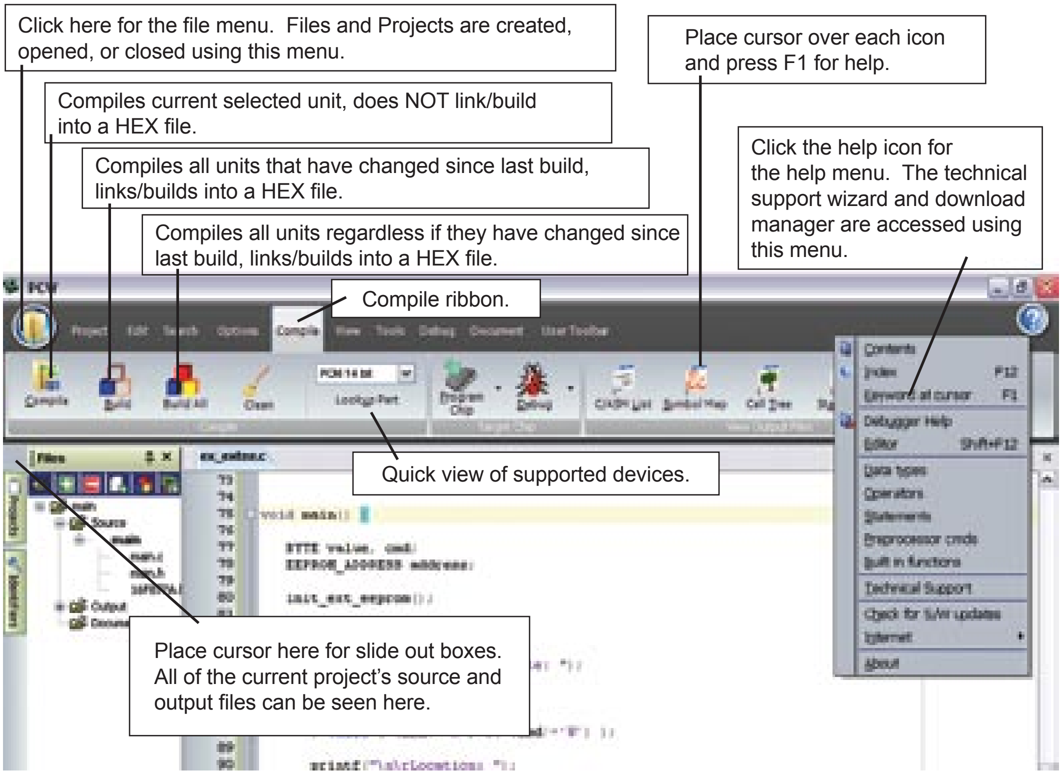
- ❑ Use the drop-down box under **Compile** to select the compiler. CCS offers different compilers for each family of Microchip parts. The Base station board has a PIC18LF4620 chip. Select **PCH 16 bit** in the drop down menu under the compile tab when you are compiling code for the Base station board. The Sensor boards have the PIC16F886 chip. Select **PCM 14 bit** from the drop-down menu under the Compile tab when you are compiling code for the sensor boards.
- ❑ The main program compiled is always shown in the bottom of the IDE. If this is not the file you want to compile, then click on the tab of the file you want to compile. Right click into editor and select **Make file project**.
- ❑ Click **Options>Project Options>Include Files…** and review the list of directories the compiler uses to search for included files. The install program should have put two directories in this list: devices and drivers.
- ❑ Normally the file formats need not be changed and global defines are not used in these exercises. To review these settings, click **Options>Project Options>Output Files** and **Options>Project Options>Global Defines**.
- ❑ Click the compile icon to compile. Notice the compilation box shows the files created and the amount of ROM and RAM used by this program. Press any key to remove the compilation box.

Viewer

- ❑ Click **Compile>Symbol Map**. This file shows how the RAM in the microcontroller is used. Identifiers that start with @ are compiler generated variables. Notice some locations are used by more than one item. This is because those variables are not active at the same time.
- ❑ Click **Compile>C/ASM List.** This file shows the original C code and the assembly code generated for the C. Scroll down to the line:

```
int_count=INTS_PER_SECOND;
```

- ❑ Notice there are two assembly instructions generated. The first loads 4C into the W register. INTS_PER_SECOND is #defined in the file to 76. 4C hex is 76 decimal. The second instruction moves W into memory. Switch to the Symbol Map to find the memory location where int_count is located.
- ❑ Click **View>Data Sheet**, then **View.** This brings up the Microchip data sheet for the microprocessor being used in the current project.


Click here for the file menu. Files and Projects are created, opened, or closed using this menu.

Compiles current selected unit, does NOT link/build into a HEX file.

Compiles all units that have changed since last build, links/builds into a HEX file.

Compiles all units regardless if they have changed since last build, links/builds into a HEX file.

Place cursor over each icon and press F1 for help.

Click the help icon for the help menu. The technical support wizard and download manager are accessed using this menu.

Compile ribbon.

Quick view of supported devices.

Place cursor here for slide out boxes. All of the current project's source and output files can be seen here.

3 COMPILING AND RUNNING A PROGRAM

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>New>Source File** and enter the filename **EX3.C**
- ❑ Type in the following program and **Compile.**

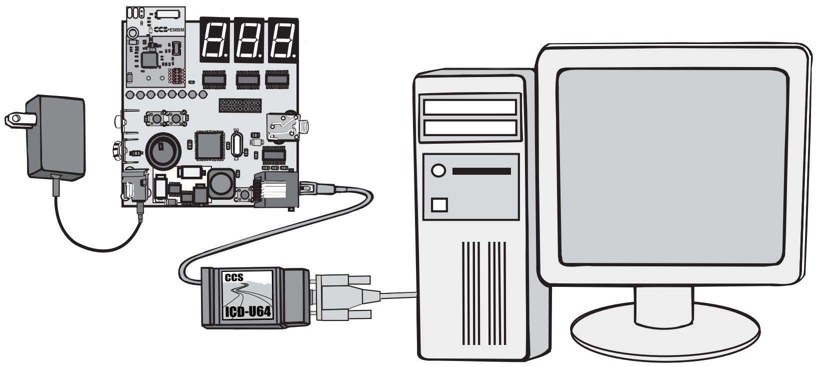
```
#include <18f4620.h>
#device ICD=TRUE
#fuses  HS,NOLVP,NOWDT
#use delay (clock=10000000)

#define GREEN_LED PIN_D5

void main () {
       while (TRUE) {
              output_low (GREEN_LED);
              delay_ms (1000);
              output_high (GREEN_LED);
              delay_ms (1000);
       }
}
```

NOTES

- The first four lines of this program define the basic hardware environment. The chip being used is the PIC18LF4620, running at 10MHz with the ICD debugger.
- The #define is used to enhance readability by referring to GREEN_LED in the program instead of PIN_D5.
- The “while (TRUE)” is a simple way to create a loop that never stops.
- Note that the “output_low” turns the LED on because the other end of the LED is +5V. This is done because the chip can tolerate more current when a pin is low than when it is high.
- The “delay_ms(1000)” is a one second delay (1000 milliseconds).



❑ Connect the ICD to the Base station board using the modular cable, and connect the ICD to the PC. Power up the Base station board.

❑ Click **Debug>Enable Debugger** and wait for the program to load.

❑ If you are using the ICD-U40 and the debugger cannot communicate to the ICD unit go to the debug configure tab and make sure ICD-USB from the list box is selected.

❑ Click the green go icon:

❑ Expect the debugger window status block to turn yellow indicating the program is running.

❑ The green LED on the Base station board should be flashing. One second on and one second off.

❑ The program can be stopped by clicking on the stop icon:

FURTHER STUDY

A *Modify the program to light the D5 LED for 5 seconds, then the D4 for 1 second and the D3 for 5 seconds.*

B *Add to the program a #define macro called "delay_seconds" so the delay_ms(1000) can be replaced with : delay_seconds(1); and delay_ms(5000) can be: delay_seconds(5);.*

Note: *Name these new programs EX3A.c and EX3B.c and follow the same naming convention throughout this booklet.*

4 ZIGBEE™ OVERVIEW

- ZigBee™ is a specified protocol for creating a wireless personal area network (WPAN). The two primary goals of the ZigBee™ organization was to create a WPAN that works using the least amount of power and the least amount of resources. Low power and low complexity make ZigBee™ a very attractive wireless networking solution for embedded systems.
- The physical and MAC layer used by ZigBee™ is IEEE 802.15.4, and will be discussed in the next chapter. 802.15.4 operates in two frequency bands, 900MHz or 2.4GHz.
- There are three different types of ZigBee™ devices: Full Function Devices (FFD), Coordinator and Reduced Function Devices (RFD). A FFD can route information between two nodes, or create entirely new networks. A Coordinator is an FFD that has taken the task of creating and administrating the network. An RFD is the simplest form of device, and can only talk to an FFD – it cannot create a network or relay between two nodes.
- ZigBee™ Coordinators can create three different kind of networks; Star, Cluster Tree or Mesh networks (figure 4.1):

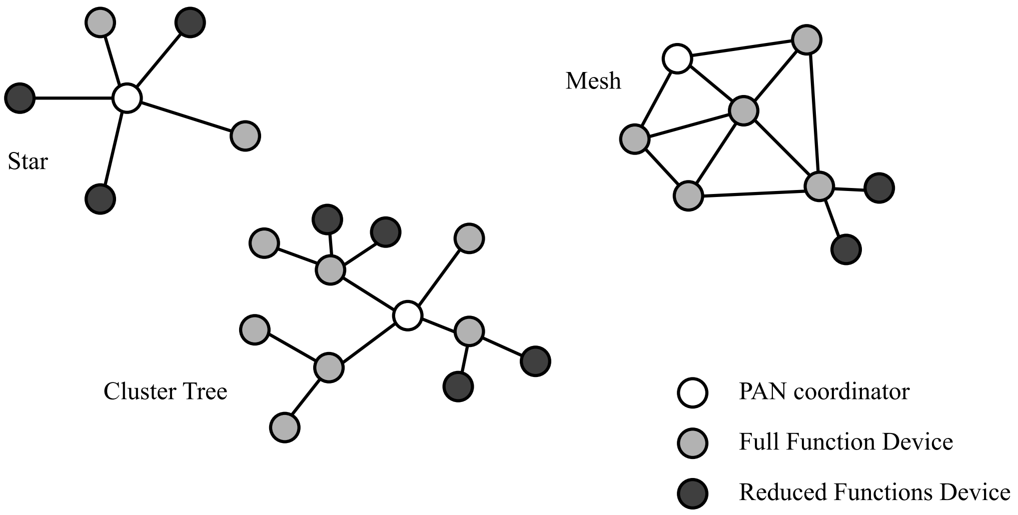


Figure 4.1

- The ability for a node to be a Coordinator, router or end node is dependent on each unit's capabilities. A RFD can only be an end node. Mesh and Cluster Tree networks provide for more stability and longer range than Star networks; sacrificing network/node complexity and larger power requirements.

5 802.15.4 OVERVIEW

- The physical/MAC layer of ZigBee™ is 802.15.4, which is a wireless standard aimed for low power consumption. Low power is achieved in two ways: low data rates and using a beacon to synchronize when nodes can go to sleep.

- The current 802.15.4 specification uses three frequency bands:

Frequency Band (MHz)	# Channels	Region	Modulation	Max Data Throughput (Kbit/s)
868.0 – 868.6	1	Europe	BPSK	20
902 – 928	10	North America, Japan	BPSK	40
2400 – 2483.5	16	Worldwide	O-QPSK	250

Figure 5.1 – 802.15.4 Frequency Bands

- 802.15.4 also uses Carrier Sense, Multiple Access (CSMA) to prevent two nodes from attempting to talk at the same time. Beacons and acknowledge messages do not use CSMA.

- Below is an overview of the data sent by the 802.15.4 physical layer:

Bytes	4	1	1	x
	Preamble	SOF	Length	MAC Sublayer (MPDU)

Figure 5.2 – 802.15.4 PHY Layer

- The preamble and start-of-field marker are used to distinguish a start of 802.15.4. Length is the size of the MPDU.

- Below is an overview of the data sent in the MAC Sublayer (MPDU):

Bytes	2	1	4 to 20	n	2
	Frame Control	Seq Num	Address Field	Data Payload (MSDU)	FCS

Figure 5.3 – 802.15.4 MAC Layer

❑ Addressing and frame control is specified by the two Frame Control bytes:

Bit	Description
0:2	Frame Type (0b000=Beacon, 0b001=Data, 0b010 = ACK, 0b011=Command)
3	Security Enabled
4	Frame Pending
5	Acknowledge Requested
6	IntraPan
7:9	RESERVED
10:11	Destination Addressing Mode (0b00=NONE, 0b10=16bit, 0b11=64bit)
12:13	RESERVED
14:15	Source Addressing Mode (0b00=NONE, 0b10=16bit, 0b11=64bit)

Figure 5.4 – 802.15.4 MAC Frame Control

❑ The size and content of the address field of the MAC datagram is dependent on the Frame Type and address mode specified in the Frame Control word.

❑ An ACK frame contains no addressing, a beacon frame may contain 4 to 10 bytes of addressing, and a data/cmd frame may contain 4-20 bytes of addressing.

❑ A beacon frame is originated by the PAN coordinator and is used to synchronize all nodes in the network. A command frame is used by frames to help create the networks shown in Figure 4.1, and will not be covered in this tutorial. Data and ACK frames will be covered extensively in the next few chapters.

6 OVERVIEW OF THE CCS WIRELESS EMBER EDITION DEVELOPMENT KIT

- The Wireless-Ember ZigBee™ Edition Development kit, with EM260, consists of one Base station board and two Sensor boards. The Base station board has a PIC18LF4620 running at 3.3V and the Sensor boards have a PIC16F886 running at 3.3V.
- The Base station board can be powered using a 9VDC power supply and the Sensor boards with a 9V battery or Insight Adapter for both boards.
- The Base station board has two pushbuttons connected to Pins RA4 and RA5 and a potentiometer connected to pin RA0. There are eight LEDs connected to the pins on Port D and there is an I/O header to give access to the other pins on the PIC18LF4620. For RS232 serial I/O, an RS232 driver is connected to RC6 and RC7. Three 7-segment LEDs are also available, which can be controlled using pins RB4, RB5 and RC5 (example code will be given later for the 7-segment LEDs).
- The Sensor boards have two pushbuttons connected to pins RA4 and RA7. They have three LEDs connected to pins RB4, RB5 and RC0. A light detector is connected to Pin RA1 and a thermistor is connected to Pin RA0. The other pins on the PIC16F886 are accessed through the I/O header.
- The Sensor boards are powered through the MAX1672 step up/down converter. The board powers up when the pushbutton near this chip is pressed. **Hold down this pushbutton when programming the chip.** Pin RA2 must be set to high at the start of the code to ensure the chip remains powered up.
- The Ember EM260 is a ZigBee™-compliant (802.15.4 PHY/MAC) radio that is used is used in this development kit and dicussed in this tutorial. The EM260 ZigBee™ network co-processor, the EM260 is a 2.4 GHz 802.15.4 radio with a Flash-based processor on-chip. The EM260 runs the EmberZNet ZigBee stack, exposing the top level API over an SPI interface. The microcontroller which is the HOST, runs the application and makes appropriate calls to the EmberZNet stack over the SPI interface
- This kit is intended to develop a 802.15.4 ZigBee™ network.

7 THE EM260 NETWORK COPROCESSOR AND THE EMBERZNET STACK

❑ The EM260 is a 2.4 GHz 802.15.4 radio transceiver with the following characteristics:
- Supports 16 channels which are spaced 5 MHz apart (2.405, 2.410, ...2.480 GHz)
- RX sensitivity of -97dBm (typical)
- +3 dBm TX output power (+5 dBm in boost mode.)
- Low supply voltage (2.1V to 3.6V)
- Minimal current drawn (35 mA in RX, 35 mA in TX; VCC at 3.0V)
- Integrated TX/RX switch, minimizing external components

❑ The EM260 integrates an IEEE 802.15.4 transceiver with a network processor core. It provides a hardware MAC that supports automatic acknowledgment generation and reception, and automatic filtering of received packets. The EM260 MAC utilizes a DMA interface to RAM memory to further reduce the overall microcontroller interaction while transmitting and receiving packets.

❑ The EM260 integrates the XAP2b microprocessor developed by Cambridge Consultants Inc. The XAP2b is a 16-bit Harvard architecture processor with separate program and data memory spaces. The word width is 16-bits for both program and data memory.

❑ The EmberZNet^{TM} is a self-organizing, self-healing, mesh networking protocol stack. The stack provides complete networking services from the physical layer up to a reliable transport layer. EmberZNet^{TM} provides a common application programming interface (API) that utilizes the underlying layers. This API provides support for the following layers of the OSI model:
- PHY: Radio control
- MAC: Medium access
- LINK: Route discovery
- NETWORK: Routing, association
- TRANSPORT: Data delivery

❑ The stack has the following features:
- PHY and MAC layers comply with the IEEE 802.15.4 standard
- LINK and NETWORK layers comply with the ZigBee specifications
- Transport layer incorporates many of the features previously provided by Ember's EmberZNet^{TM} networking solutions.

❑ The EM260 chip runs the EmberZNet^{TM} stack and the host PIC® microcontroller can access the commands of this API using an SPI interface called the EZSP (EmberZNet Serial Protocol) interface which is described in the following chapter.

❑ The EM260 chip is preprogrammed with the EmberZNet ZigBee^{TM} stack version 3.x, also known as the Zigbee^{TM} Pro stack. The following chapters explain in detail, the communication protocol between the PIC® MCU (HOST) and the EM260.

8 THE EM260 EZSP INTERFACE

❑ The EM260 is a transceiver with a Flash-based microprocessor running the EmberZNet ZigBee™ stack. The Microchip device uses the SPI interface to talk to the top level API of the EmberZNet stack.

❑ The microprocessor or the HOST uses the EZSP (EmberZNet Serial Protocol) interface to send commands and get responses from the EM260.

❑ CCS provides a library for communication with the EM260 over the SPI interface that allows for sending and receiving all the standard EZSP frames. An EZSP frame can be from 5 bytes to 136 bytes long and will have the following format:

EZSP Frame Format

SPI byte	Length or Error	Sequence byte	EZSP Frame (Variable length)	Frame Terminator

- The SPI byte is set to 0xFE for an EZSP frame.
- The length can take values from 3 to 133. The Length byte defines the length of just the EZSP frame and is only included if there is information in the EZSP frame. In case of an error in communication, the length byte is replaced by the Error byte that provides more information about the error.
- The EZSP frame consists of a frame control byte and a command byte followed by data which is optional.
- Every frame must end with a frame terminator byte which is set to 0xA7.

❑ The following example will initialize the SPI interface and ask the EM260 for the SPI protocol version and verify the SPI status. It will then perform the first EZSP transaction, an ezspVersion command.

❑ Type in the following example and name it EX8.c. Compile and run it on the Base Station board.

```
#include <18F4620.h>
#fuses HS, NOWDT, NOLVP
#use delay(clock = 10000000)
#use rs232(baud = 9600, xmit=Pin_C6, rcv=Pin_C7)

#define COORDINATOR 1

#include<EM260.h>

#include "ember _ utilities.c"

void main(void)
{
   int8 status;
   int8 i, version_no;
   EmberEUI64 MyEUI64;
   int8 stacktype;
   int16 stackVersion;

   SPIInit();
   EM260Reset();
   status = ask_SPI_version();
   if(status == 0)
   {
      printf("\n\r SPI Version OK");
      status = verify_SPI_status();
      if(status == 0)
         printf("\n\r SPI Status OK");
      else
         printf("\n\r SPI Bad status");
      version_no = ezspVersion(0x02, *stacktype, *stackversion);
      ezspNop();
      ezspGetEUI64(&MyEUI64[0]);
      printf("\n\r Local EUI64: ");
      for(i=0;i<8;i++)
      printf("%x", MyEUI64[i]);
   }
   else
   printf("\n\r SPI error");
   while(TRUE);
}
```

8 THE EM260 EZSP INTERFACE (CONT.)

- ❑ This code will initialize the SPI module which is used to communicate with the EM260. It will reset the EM260 and check the SPI version and the SPI status. Once the SPI communication is verified, perform the first EZSP (EmberZNet Serial Protocol) transaction, an ezspVersion(). EmberZNet 3.x requires the ezspVersion command to be the first command sent to the EM260 after it has reset. This is done to ensure that the HOST and the EM260 agree on the protocol format. The command packet has the following seven bytes -
  - 0xFE – SPI Byte indication an EZSP frame
  - 0x04 – Length Byte showing the EZSP frame is 2 bytes long.
  - 0x00 – Sequence Byte – This value varies based on previous sequence bytes.
  - 0x00 – EZSP Frame Control Byte indicating a command with no sleeping.
  - 0x00 – EZSP Frame Type Byte indicating the ezspVersion command.
  - 0x02 – Value of desired protocol version for EmberZNet Stack 3.x
  - 0xA7 – Frame Terminator Byte

- ❑ The ezspGetEUI64() command will return the local 64-bit identifier for the device. Each radio module is assigned a unique IEEE 64-bit identifier (EUI-64) and the host can ask the EM260 for this number using the ezspGetEUI() function.
- ❑ Connect the 9-pin serial cable to the PC and the other end to the Base station prototyping board and execute the above code. The code will verify the SPI status and then print out the local 64-bit EUI address of the device. Use **Tools>Serial Port Monitor** to view the output of the program.

9 FORMING A NETWORK

- ❑ An EmberZNet network is formed by a coordinator device. Each Personal Area Network (PAN) must be started by a node acting as a coordinator. After forming the network it will act as a router and it can then accept requests from other devices that wish to join the network. Depending on the stack and application profiles used, the coordinator might also perform additional duties after network formation.

- ❑ Network Formation Steps:
  - Initialize the EM260 by calling EmberIntitialization() - This will verify the SPI version, SPI status and the Ember Znet Serial Protocol version.
  - The EmberConfig() function will configure the EM260 stack for specific network settings.
  - The EmberInitializeBinding() will clear all the binding entries. More information on bindings in chapter 11.
  - The ezspGetEUI64() command will return the local 64-bit identifier for the device. Each radio module is assigned a unique IEEE 64-bit identifier (EUI-64) and the host can ask the EM260 for this number using the ezspGetEUI64() function.
  - Add an endpoint – Use the EmberAddEndpoint() function sets up an endpoint for the node. Endpoints are logical extensions defined by the application that can be thought of as devices accessible through a single ZigBee^{TM} radio. The EmberAddEndpoint() function calls the ezspAddEndpoint() command which tells the EM260 to add an endpoint with default settings.
  - EmberFormNetwork() - This function will test to see if the node is already a part of the network. If this test fails, it will assign its node type as coordinator and form a network using the PAN ID, Extended PAN ID, channel and radio power defined in the header file EM260.h.

- ❑ Since the network is not up, the ezspNetworkInt() function will fail and the EmberFormNetwork() function will be called for a ZigBee^{TM} network using the default network parameters. Once the network is formed, the EM260 will inform the host microcontroller that the network initialization has taken place.

- ❑ The following example includes the header file EM260.h, which is included in the compiler. This header file will include the ezsp API and the functions related to this API. Following is a brief description of each file included by the EM260.h.
  - EM260.h - This header file contains defines related to the EM260 chip hardware and the EmberZNet Stack software. It also contains defines useful for the development kit examples.
  - ember_types.h – This file defines the various structures and types used by the EM260 chip.
  - ezsp_function.c – This is the low level API which uses SPI commands to talk to the EM260.
  - ezsp_commands.h – Contains a list of EmberZNet Stack commands.
  - callback_handlers.c – This file defines unused callbacks. If the EM260 has some data, it notifies the host using callbacks. The CCS API handles the basic callback commands and the user can write routines to handle the lesser used callbacks.
  - EM260API.h – This API implements the ezsp commands as documented in the EM260 datasheet. It also contains a few useful helper functions.
  - ember_utilities.c – Contains callback implementations and helper functions specific to this exercise book.
- ❑ Helper function and application specific functions will begin with the word ‘Ember’, while the API functions will begin with ‘ezsp’.
- ❑ The Base station board is used as a coordinator node in this development kit. Type in the following program (**EX9.c**), compile and run it on the Base station board.

```
#include <18F4620.h>
#fuses HS,NOWDT,NOLVP,PUT,NOBROWNOUT
#use delay(clock=10000000)
#use rs232(xmit=PIN_C6, rcv=PIN_C7, baud=9600)

#define COORDINATOR 1

#include “em260.h”
#include “ember_utilities.c”

void main()
{
   EmberEUI64 emLocalEui64;
   EmberStatus Status;

   printf(“\n\r RESET....”);

  if(EmberInitialization())
   {
      EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEUI64[0]);
      EmberAddEndpoint();

      EmberFormNetwork();
      Status=ezspPermitJoining(0xFF);

      while(TRUE)
      {
         EM260Tick();
         sinkApplicationTick();
      }

   }
}
```

9 FORMING A NETWORK (CONT.)

- ❑ The EM260 will start the network with the default network parameters. Once the network is up, the sinkApplication() function keeps a rack of the number of nodes on the network. The EM260tick() uses ezspCallback function to query the EM260 for data. The EM260 will respond with data or with a noCallback response when it does not have any data for the host. The EmberTick function should be called as often as possible to preserve the EM260 timing and 802.15.5 MAC compliance. At a minimum, this should be once per iteration of the application main while {} loop.
- ❑ The ezspPermitJoining() function allows other nodes to join to the network. The valid parameters for this function are:
  - 0x00 - Disables joining.
  - 0xFF - Enables joining for an unlimited amount of time.
  - 0x01 - 0xFE specifies how many seconds to allow joining.
- ❑ The above example uses default network parameters to form a network. The user can also use the API command, EmberScanAndFormNetwork() to form a network. The user can decide which of the channels the radio should scan from the available channels 11-26. After scanning the desired channels, this function will then choose a channel with the least average energy. It will pick a PAN ID that does not appear in the active scan, it will use the extended PAN ID that was passed to the function, or it will randomly choose one if an extended PAN ID of all zeros was passed to it, and then it will form a network using the chosen channel, PAN ID and extended PAN ID.

  The API uses a 32-bit channel mask to identify the channel which is to be scanned, where bit 11 to bit 26 of this 32-bit mask specify the channel number. The following bit mask will be used to scan all the channels:

```
#define EMBER_ALL_802_15_4_CHANNELS_MASK 0x07FFF800
```

  For scanning a specific channel, for example 15, use:

```
#define Channel_15     0x00004000
```

  The Sink board is acting as coordinator and has started the network. It has also given permission to other devices to join the network. The next chapter, will configure end devices and join them to this network. Note that the D0 LED should light up indicating that the board is now the coordinator.

10 JOINING A NETWORK

- ❑ After the network is formed, router and end devices can join the network, provided the network contains one or more nodes that permit joining. A coordinator has been setup in the previous chapter using the Base Station device to form a network with APP_PANID. The function emberPermitJoining(int seconds) must be called in the coordinator to ensure that other devices can join the network.

- ❑ Routers and End Devices can join the network by calling the ezspJoinNetwork() function which with the default set of network parameters. Apart from the network parameters, the node type of the device that wishes to join the network must be specified. The node type is extremely important, as it determines what role that particular device plays in the network. A device can have the following node types:

  - EMBER_UNKNOWN_DEVICE - Device has not joined network.
  - EMBER_COORDINATOR (FFD) - It is responsible for forming a network. Afternetwork formation, it will act as router and act as parents to other nodes.
  - EMBER_ROUTER (FFD) - They provide routing services for the network devices and act as parents to other nodes.
  - EMBER_END_DEVICE (RFD) - A non-sleepy end device that will only communicate with its parent and will not relay messages.
  - EMBER_SLEEPY_END_DEVICE (RFD)
  - EMBER_MOBILE_END_DEVICE (RFD)

  The node type EMBER_COORDINATOR can only be used while forming a network, therefore, one of the remaining node types can be selected for the device.

- ❑ The following example will use the EmberJoinNetwork() function to join the network as a EMBER_SLEEPY_END_DEVICE. The Sensor boards, with PIC16F886 devices, are configured to use their Internal Oscillator at 4 Mhz using the #fuses directive. The device gets power from the 3V regulator- MAX1672. The regulator can be turned ON either by pressing down the pushbutton, or in software by setting the POWER_UP_PIN high. Once the code starts running, the chip needs to set the POWER_UP_PIN high, so that the regulator can continuously supply power to the chip.

- ❑ Type in the following code (**EX10.c**) and compile. Hold down the power pushbutton while downloading the code onto the chip. Once the code starts running, set the POWER_UP_PIN high at the start of main, so the chip will stay powered up. The user can release the power pushbutton once the ICD performs the programming and verification cycles.

JOINING A NETWORK (CONT.)

```
#include <16F886.h>
#fuses INTRC,NOWDT, NOLVP, BORV21
#device *=16
#use delay(clock = 4000000)
#define SLEEPY_SENSOR 1

#include “em260.h”
#include “ember_utilities.c”

void main(void)
{
   EmberStatus Status;
   EmberEUI64 emLocalEUI64;

   output_high(POWER_UP_PIN);

   if (EmberInitialization())
  {
      EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEUI64[0]);
      EmberAddEndpoint();
      EmberJoinNetwork(EMBER_SLEEPY_END_DEVICE);

      while(TRUE)
      {

          sensorApplicationTick();
          EM260Tick();
          if(!input(PUSH_BUTTON1))
          {
             ezspLeaveNetwork();
             while(input(PUSH_BUTTON2));
          }

      }
   }
}
```

- ❑ Remove the ICD cable after programming the board. The code should start running once the programming cycle has ended. The battery powered device will attempt to join the network as EMBER_SLEEPY_END_DEVICES. The coordinator node must permit these nodes to join the network.

  In the previous example, the coordinator executed the ezspPermitJoining command, so that the new nodes can join the network.

- ❑ Once the EM260 initializes and joins the network using the default parameters as defined in EM260.h, one more LED should light up on the coordinator node.
- ❑ Download this code on the second Sensor board device. Remove the ICD cable after programming the board. Another LED on the coordinator node will light up, indicating that there is a third device on the network.
- ❑ On one of the Sensor boards, press pushbutton 1 to execute the ezspLeaveNetwork() command. This node will leave the network. Notice that one LED on the coordinator is turned OFF.
- ❑ Press pushbutton 2 to rejoin the network. The node will again execute the sensorApplicationTick() function in the while loop which attempts to join the network. The sensorApplicationTick() function keeps a track of the network state and attempts to rejoin the network if connection is lost.
- ❑ The ezspLeaveNetwork() is an important API call as it allows change to the node functionality in the network, for example changing a EMBER_SLEEPY_END_DEVICE to a EMBER_ROUTER.
- ❑ The above example uses the default network parameters to join a network. The user can also use the API command, EmberScanAndJoinNetwork() to join a network. The user can decide which of the channels the radio should scan from the available channels 11-26. The command will cause the device to try and join the first network it finds that permits joining, matches it's stack profile, and matches the extended PAN ID passes to it by the function, or any extended PAN ID if all zeros was passed for the extended PAN ID.

11 MESSAGING TYPES AND BINDINGS

❑ A message is a single packet that is sent over the radio. The EmberZNet Serial Protocol (EZSP) provides three basic functions for sending messages between nodes.

- ezspSendUnicast(options);
- ezspSendBroadCast(options);
- ezspSendMulticast(options);

❑ **Unicast messages:** This is used to send messages to a specific node ID based on an address table entry. The ezspSendUnicast() command has the following parameters:

- EmberOutgoingMessage Type: this defines the type of message that is being sent by the EM260. The options EMBER_OUTGOING_DIRECT will be used for messages sent to a specific device whose node ID is known. EMBER_OUTGOING_VIA_BINDING is used when an index in the binding table is specified.
- EmberNodeID indexOrDestination: Destination node ID or index in the binding table.
- EmberApsFrame* apsFrame: APS frame (Application Support Layer)
- int8 messageTag : An 8 bit message tag.
- int8 messageLength: The length of the message
- int8 * messageContents: The payload.

❑ **Broadcast and Multicast messages**: These message commands have similar options to the Unicast command but do not specify the EmberOutgoingMessage type. These messages also define the radius or hops which defines the delivery radius of the messages. EMBER_MAX_HOPS controls this radius value.

❑ Before a message can be sent, it has to be constructed. These three messaging commands need an APS frame and this frame must be constructed by the user before the message can be delivered over the network. The EmberApsFrame struct has the following members:

- rofileID: The application profile ID (16 bits)
- clusterID: The cluster ID (16 bits)
- sourceEndpoint: Endpoint of the source (8 bits)
- destinationEndpoint: Endpoint of the destination (8 bits)
- EmberApsOption Options: Options to use while sending a message (16 bits)
- groupID: A Group ID used for multicast messages (16 bits) – This value is valid only for incoming broadcasts and multicasts.
- sequence: A sequence number (8 bits) - This value is valid only for incoming messages. For outgoing messages, the stack ignores the value supplied by the application.

❑ Sending Direct Unicast Messages: The coordinator node ID is always 0x0000. Once a transmitting device knows the destination node ID (0x0000 in this case), it can start sending messages using the OutgoingMessage type: EMBER_OUTGOING_DIRECT. The battery powered end devices can talk to the coordinator node.

In the next chapter the ezspSendUnicast() command will be used to transmit messages from the end devices to the coordinator.

❑ Sending Unicast Messages Via Binding: Bindings are not required for basic ZigBee^{TM} communication, but they are useful for tracking information about frequently used communication paths. If we have a simple network with one coordinator (Full Function Device) and two end devices, Node A and Node B, then Node A can transmit messages to Node B in one of the 2 following ways:

❑ Send Direct Unicast Messages: This requires the knowledge on the Node Id of Node B

❑ Send Unicast Messages via Binding: A binding needs to be setup between Node A and Node B. If Node A wants to send messages to Node B via binding, it needs to setup an entry in its binding table and needs the following information about Node B:

- 64 bit Address
- 8 bit endpoint
- 16 bit cluster ID

Similarly, if Node B wants to transmit messages to Node A, it can do the same using one of the above two methods.

12 TRANSMITTING AND RECEIVING UNICAST MESSAGES

- ❑ Any node on the Network can transmit Unicast messages to another node as long as it knows the 16-bit node ID of the desination. In previous examples the ZigBee™ network was formed using the Base station board as the coordinator node. The following example will use a network that has a similar structure with the Base Station configured as the coordinator node and the battery powered Sensor boards acting as end device nodes.

  The node ID of a coordinator in a ZigBee™ network is alwas 0x0000. This example will transmit Unicast messages from the end devices to the coordinator using the ezspSendUnicast() API call. These messages are received by the coordinator and the EM260 informs the host, using callback commands, that it has received messages requiring attention. The user must write callback handler routines to handle this incoming message. The header file, EM260API.h contains the basic Incoming message handler routine that grabs the incoming data. The user application can act upon this data based on his application requirements. A sample function is implemented in the included file ember_utilities.h. The following example displays the incoming message onto the 7-segment LED display on the Base station board.

- ❑ Type in the following code into a new source file and save it as digit.h.

```
#define EXP_OUT_ENABLE PIN_C1
#define EXP_OUT_CLOCK PIN_B4
#define EXP_OUT_DO   PIN_B5

#define NUMBER_OF_74595 3
#include<74595.c>
const char digit_format[10]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82,
0xF8,0x80, 0x90};

void lcd_clear(void){
   int8 digits[3]={0xFF, 0xFF,0xFF};
   write_expanded_outputs(&digits[0]);
   output_low(EXP_OUT_ENABLE);
}

void lcd_putd(int16 num){
   int8 digits[3];
   if(num>999){num=999;}

(continued...)
(continued...)

   digits[0]=num/100;
   digits[1]=(num%100)/10;
   digits[2]=num%10;

   digits[0]=digit_format[digits[0]];
   digits[1]=digit_format[digits[1]];
   digits[2]=digit_format[digits[2]];

   write_expanded_outputs(&digits[0]);
   output_low(EXP_OUT_ENABLE);

}
```

- ❑ Type in the following code in a new file (EX12A.c), compile it and download it onto the Base station board. This code will setup the board as an EMBER_COORDINATOR with a node ID of 0x0000 and the incomingmessagehandler will receive the data and display it on the 7-segment LED display.

```
#include <18F4620.h>
#fuses HS,NOWDT,NOLVP,PUT,NOBROWNOUT
#use delay(clock=10000000)
#use rs232(xmit=PIN_C6, rcv=PIN_C7, baud=9600)

#define COORDINATOR 1

#include “em260.h”

#include “ember_utilities.c”
#include<digit.h>

void main()
{
   EmberEUI64 emLocalEui64;
   EmberStatus Status;

(continued...)
```

TRANSMITTING AND RECEIVING UNICAST MESSAGES (CONT.)

```
(continued...)

   printf("\n\r RESET....");

   lcd_clear();
   if(EmberInitialization())
   {
      EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEui64[0]);
      EmberAddEndpoint();

      EmberFormNetwork();
      Status=ezspPermitJoining(0xFF);

      while(TRUE)
      {
         EM260Tick();
         sinkApplicationTick();
         lcd_putd(unicast_data);
      }

   }
}
```

❑ Type in the following code(**EX12B.c**), compile it and download it onto one of the Sensor boards.

```
#include <16F886.h>
#fuses INTRC,NOWDT, NOLVP, BORV21
#device *=16
#use delay(clock = 4000000)

#define SLEEPY_SENSOR 1

#include "em260.h"

#include "ember_utilities.c"


(continued...)
```

```
(continued...)

void main(void)
{
   int8 i;
   EmberStatus status;
   EmberEUI64 emLocalEui64;
   EmberApsFrame apsframe;
   int8 messagetag=0x01;
   int8 messagedata[1];
   int8 messageLength;
   int16 indexOrDestination;

   output_high(POWER_UP_PIN);


   if(EmberInitialization())
   {

      EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEui64[0]);
      EmberAddEndpoint();

      EmberJoinNetwork(EMBER_SLEEPY_END_DEVICE);

      indexOrDestination = 0x0000;
      apsframe.profileID=0xC00F;
      apsframe.clusterID=0x0001;
      apsframe.sourceEndpoint=0x01;
      apsframe.destinationEndpoint=0x01;
      apsframe.options=0x0000;
      apsframe.groupID=0x0000;
      apsframe.sequence=0x00;
      messageLength=1;

      while(TRUE)
      {
         sensorApplicationTick();
         EM260Tick();
         if(!input(PUSH_BUTTON1))
         {

(continued...)
```

TRANSMITTING AND RECEIVING UNICAST MESSAGES (CONT.)

```
(continued...

            i++;
            messagedata[0]=i;
            ezspSendUnicast(EMBER_OUTGOING_DIRECT,indexOrDestination,&ap
sframe,messagetag,messageLength,&messagedata[0]);
            delay_ms(200);
            }

         if(!input(PUSH_BUTTON2))
         {
            i--;
            messagedata[0]=i;
            ezspSendUnicast(EMBER_OUTGOING_DIRECT,indexOrDestination,&ap
sframe,messagetag,messageLength,&messagedata[0]);
            delay_ms(200);
         }

      }
   }
}
```

The above code will initialize the Sensor board to be a EMBER_SLEEPY_END_DEVICE and join the network. Once this code starts running on the end devices, press either pushbutton 1 or 2 to transmit Unicast messages to the coordinator node. The coordinator is already setup to receive these messages and display them on the 7-segment display. Hold down pushbutton 1 on the end device and notice the LED display being incremented. Press pushbutton 2 to decrement the values on the 7-segment display. This example transmits only one byte in each packet. The maximium payload for an APS layer message can be set by using the ezspMaximumPayloadLength() API call.

- ❑ The IncomingMessageHandler is an important function and the user should modify this based on the application requirements. The IncomingMessageHandler callback returns the following parameters:
    - **EmberIncomingMessageType** - Defines the type of the incoming message, for example Unicast or Datagram.
    - **EmberApsFrame** - The APS frame from the incoming message.
    - **lastHopLqi** - The link quality from the last node that relayed the message.
    - **lasthopRssi** - The energy level (in units dBm) observed during reception.
    - **EmberNodeID sender** - the 16-bit node ID of the sender.
    - **binding index** - Index of the binding that matches the message or 0xFF if there is not matching binding.
    - **DatagramReplyTag** - If the incoming message is a datagram and the host wishes to send a reply, this value must be supplied to the ezspSendReply() command.
    - **Messagelength** - The length of the message contents parameter in bytes.
    - **Messagecontents** - The incoming message.
- ❑ Whenever any node receives a message, it can obtain the 16-bit nodeID of the sender from the IncomingMessageHandler command. It can then use this information to transmit Unicast messages to the sender.

13 USING BINDINGS FOR BI-DIRECTIONAL COMMUNICATION

- ❑ For using Transport layer messages, devices must maintain a binding table with a configurable number of entries. The device can directly send messages to any of these entries using Unicast messages via binding.
- ❑ The following example describes how to setup a binding between two end devices, and then use binding to transfer data between the two nodes. The two EMBER_SLEEPY_END_DEVICES will be referred as Node A and Node B. If Node A wants to send data to Node B, it can send transport layer messages by creating a binding to Node B. If node A wishes to setup a binding of node B, then node A should know the 64-bit EUI address of that node. This example will configure each end device to transmit an ezspBroadcast message that contains it 64-bit EUI address. This broadcast message can be transmitted by any node that is on the network. The coordinator node has already been setup in the previous chapter with default network parameters defined in EM260.h. This coordinator will act as parent to these two nodes (A and B) and it will manually route the broadcast messages from one end device to another. If Node B transmits a broadcast message containing its EUI64, then Node A can hear it and then bind to Node B. Now, Node A can send transport layer messages to node B. Similarly, if Node B wants to send transport layer messages to Node A, Node A must broadcast its 64-bit EUI address so Node B, or any other Node wishing to talk to Node A can setup a binding.
- ❑ Type in the following code(**EX13.c**), compile it and download onto both the Sensor boards.

```
#include <16F886.h>
#fuses INTRC,NOWDT, NOLVP, BORV21
#device *=16
#use delay(clock = 4000000)

#define SLEEPY_SENSOR 1

#include "em260.h"

#include "ember_utilities.c"


int8 seconds;
int8 int_count;

#int_TIMER1
void clock_isr()
{

(continued...)
```

```
(continued...

  if(--int_count==0)
      {
      ++seconds;
      int_count=INTS_PER_SECOND;
      }
}

void main(void)
{

   Emberstatus status;
   int8 messagelength,messagetag;
   EmberEUI64 emLocalEui64;

   int8 message[10];
   int16 nodeID;
   int16 Time,milliseconds;
   EmberApsFrame apsframe1;
   EmberNodeID destination;


   output_high(POWER_UP_PIN);
   if(EmberInitialization())
   {
      EmberConfig();
      EmberInitializeBinding();
      EmberAddEndpoint();

      EmberJoinNetwork(EMBER_SLEEPY_END_DEVICE);

      setup_timer_1(T1_INTERNAL|T1_DIV_BY_1);
      enable_interrupts(int_TIMER1);
      enable_interrupts(GLOBAL);
      while(TRUE)
      {
         sensorApplicationTick();
         ezspCallBack();
         if(!input(PUSH_BUTTON1))
         {

(continued...)
```

USING BINDINGS FOR BI-DIRECTIONAL COMMUNICATION (CONT.)

```
(continued...)

            apsframe1.profileID=0xC00F;
            apsframe1.clusterID=MSG_SENSOR_ADVERTISE;
            apsframe1.sourceEndpoint=0x01;
            apsframe1.destinationEndpoint=0x01;
            apsframe1.options=EMBER_APS_OPTION_NONE;
            apsframe1.groupID=0x0000;
            apsframe1.sequence=0x00;

            messagetag=0x01;

            ezspGetEui64(&emLocalEui64[0]);
            nodeID=ezspGetNodeID();
            messagelength=EUI64_SIZE+2;
            memcpy(&message[0],&emLocalEui64[0],EUI64_SIZE);
            memcpy(&message[EUI64_SIZE],&nodeID,2);
            destination = 0xFFFC;
            status=ezspSendBroadcast(destination, &apsframe1 ,EMBER_MAX_
HOPS,messagetag,messageLength,&message[0]);

            delay_ms(500);
         }
         if(!input(PUSH_BUTTON2))
         {
            set_timer1(0);
            int_count=INTS_PER_SECOND;
            seconds=0;
            while(!input(PUSH_BUTTON2));
            time=get_timer1();
            milliseconds=get_milliseconds(time,int_count,seconds);
            messagelength=2;
            messagetag=0x02;
            memcpy(&message[0],&milliseconds,messagelength);

            apsframe1.profileID=0xC00F;
            apsframe1.clusterID= SensorClusterID;  //CHANGE
            apsframe1.sourceEndpoint=0x01;
            apsframe1.destinationEndpoint=0x01;
            apsframe1.options= (EMBER_APS_OPTION_RETRY |
                              EMBER_APS_OPTION_ENABLE_ROUTE_DISCOVERY |
(continued...)
(continued...)
                             EMBER_APS_OPTION_ENABLE_ADDRESS_DISCOVERY |
                             EMBER_APS_OPTION_DESTINATION_EUI64);



            apsframe1.groupID=0x0000;
            apsframe1.sequence=0x00;

            status=ezspSendUnicast(EMBER_OUTGOING_VIA_BINDING,sensor_
binding_index1,&apsframe1,messagetag,messageLength,&message[0]);

            delay_ms(500);
         }
      }
   }
}
```

- ❑ Both Sensor boards will run the same program. Transmit broadcast messages from one node by pressing pushbutton 1. Call this Node A. When Node A prepares the broadcast message, it assigns the cluster ID of the APS frame to be MSG_SENSOR_ADVERTISE. This information will be used by the receiving node to distinguish this message from other messages. Node A will transmit a broadcast message with the 64-bit EUI address of the device. The other board, Node B will hear this message and can setup a binding with Node A. The setting up of bindings is handled by the IncomingMessageHandler function which calls the handleBindingRequest() function. The clusterID of this message will be MSG_SENSOR_ADVERTISE, else this message is ignored. Node B will bind to the request and the LEDs on Node B will flash when a successful binding occurs. Now, node B can send data to node A using the Unicast message via binding table. This example uses Timer1 interrupt to transmit timing data between the boards. Hold down pushbutton 2 on Node B for 1 second. A LED on Node A should flash for 1 second. The LED on Node A will turn ON for the amount of time the pushbutton 2 is pressed on Node B.
- ❑ Now press pushbutton 1 on Node B. This will transmit a broadcast message with the 64-bit EUI address of the device. Now, Node A can setup a binding to Node B and can transmit timing data using pushbutton 2. The LED on Node B will turn ON for the amount of time the pushbutton 2 is pressed on Node A. A bidirectional link between the two SLEEPY_END_DEVICES is set up. These messages are routed through the coordinator. The end devices call the ezspPollforData() API function to query the coordinator for the message. It is the parent nodes responsibility to store the messages it needs to transmit to its children (end devices). The EmberZNet stack takes care of storing and delivering these messages automatically.

14 TRANSMITTING AND RECEIVING MULTICAST MESSAGES

- ❑ Multicast messages is any easy way to transmit the same message to large number of nodes that are using the same group ID within a specified radius of the sending node. Any node on the Network can transmit a multicast message using the ezspSendMulticast() function.
- ❑ The following example describes how to setup a node to receive and transmit multicast messages. There are two ways to setup a node to receive multicast messages, the first method, that the following example uses, is to create a binding table entry of type EMBER_MULTICAST_BINDING with the multicast group ID set as the EUI64 ID. This is done by calling the SetMulticastBinding() function, which creates the binding with a multicast group ID of 0x1111. The second method is to use the ezspSetMulticastTable Entry() function to store the multicast ID and end point the ID is associated with in the multicast table. The node will then receive any multicast message whose APS frame's group ID matches the EUI64 ID stored in the binding table, or the multicast ID that is stored in the multicast table.

  The ezspSendMulticast() function is used to transmit multicast messages. The first parameter passed to the function is the APS frame, which is very similar to the unicast APS frame with the exception that the apsframe.groupID needs to be set to the multicast group ID that the message is being sent to. For this example the apsframe.groupID will be set to 0x1111. The second parameter is the number of hops away from the sender the message will be delivered, this value can be 1 to EMBER_MAX_HOPS, a zero will be automatically set to EMBER_MAX_HOPS. The third parameter is the number of hops the message will be forwarded by non members of the group, a value of 7 or greater is treated as infinite. The fourth, fifth, and sixth parameters are the messageTag, messageLength, and messageContents which are the same as used in the ezspSendUnicast() function.
- ❑ When a multicast message is transmitted on the network the coordinator and routers receive the message and forward it on to the coordinator or routers they are connected to. However, they don't forward the message onto their end devices unless they originally sent the multicast message. In order for the end device to receive a multicast message that wasn't transmitted by it's parent node the coordinator or router needs to be a part of the multicast group the message was intended for, and then forward the message to it's end devices at the application level. Currently the IncomingMessageHandler() in the ember_utilities.c for the coordinator and router is written to forward multicast messages they receive to their end devices with a unicast message. This means that the end devices will receive the message regardless of whether it was part of the multicast group that the message was originally intended for. A more complex message forwarding scheme can be written if desired.
- ❑ Type in the following code (**EX14A.c**), compile it and download it onto the base station board.

```
#include <18F4620.h>
#fuses HS,NOWDT,NOLVP,PUT,NOBROWNOUT
#use delay(clock=10M)
#use rs232(baud=9600,UART1)

#define COORDINATOR 1

#include <em260.h>
#include <ember_utilities.c>
#include "digit.h"

void main()
{
   int8 i=1;
   EmberEUI64 emLocalEui64;
   EmberStatus Status;

   EmberApsFrame apsframe;
   int8 hops=10;
   int8 nonMemberRadius=7;
   int8 messagetag=0x01;
   int8 messagedata[1];
   int8 messageLength;
   int16 indexOrDestination;

   printf("\n\r RESET....");

   lcd_clear();

   indexOrDestination=0x0000;
   apsframe.profileID=0xC00F;
   apsframe.clusterID=0x0001;
   apsframe.sourceEndpoint=0x01;
   apsframe.destinationEndpoint=0x01;
   apsframe.options=0x0000;
   apsframe.groupID=MULTICAST_ID;   //sets groupID that multicast mes-
sages are sent to
   apsframe.sequence=0x00;
   messageLength=1;

   if(EmberInitialization())
   {
```
(continued...)

TRANSMITTING AND RECEIVING MULTICAST MESSAGES (CONT.)

(...continued)

```
       EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEui64[0]);
      EmberAddEndpoint();

      EmberFormNetwork();

      SetMulticastBinding();     //creates multicast binding in binding table

      Status=ezspPermitJoining(0xFF);

      while(TRUE)
      {
         EM260Tick();
         sinkApplicationTick();
         lcd_putd(unicast_data);

        if(!input(PIN_A4))           //Pressing Switch A4 will cause the
        {                                      //coordinator to disable current network.
            ezspLeaveNetwork();
            lcd_clear();                     //Pressing Reset will cause the
                                                   //coordinator to reform network.
            while(TRUE)
            {
               output_d(0x00);
               delay_ms(100);
               output_d(0xFF);
               delay_ms(100);
            }
        }

         if(!input(PIN_A5))             //Pressing switch A5 cause the coordinator
         {                                          //to send out a multicast message
              messagedata[0]=i;
              ezspSendMulticast(&apsframe,hops,nonMemberRadius,
      EmberConfig();
            messagedata[0]=i;
            ezspSendMulticast(&apsframe,hops,nonMemberRadius,messagetag, message-
Length, messagedata);
            if(i>=10)
               i=1;
            else
               i++;
            delay_ms(200);
         }
      }
   }
}
```

- [ ] Type in the following code (EX14B.c), compile it and down it onto both of the Sensor boards.

```
#include <16F886.h>
#fuses INTRC,NOWDT,NOLVP,BORV21
#device *=16
#use delay(clock=4000000)

#define SLEEPY_SENSOR 1

#include <em260.h>
#include <ember_utilities.c>

void main()
{
   int8 i,j=1;
   EmberEUI64 emLocalEui64;

   EmberApsFrame apsframe;
   int8 messagetag=0x01;
   int8 messagedata[1];
   int8 messageLength;
   int16 indexOrDestination;
   int8 hops=10;
   int8 nonMemberRadius=7;

   output_high(POWER_UP_PIN);

   if(EmberInitialization())
   {
      EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEui64[0]);
      EmberAddEndpoint();

      EmberJoinNetwork(EMBER_SLEEPY_END_DEVICE);

      SetMulticastBinding();     //creates multicast binding in binding table

      SensorData[0]=0;

      indexOrDestination=0x0000;
      apsframe.profileID=0xC00F;
      apsframe.clusterID=0x0001;
      apsframe.sourceEndpoint=0x01;
      apsframe.destinationEndpoint=0x01;
      apsframe.options=0x0000;
```

(continued...)

14 TRANSMITTING AND RECEIVING MULTICAST MESSAGES (CONT.)

(...continued)

```
      apsframe.options=0x0000;
      apsframe.groupID=MULTICAST_ID;    //sets groupID that multicast mes-
sages are sent to
      apsframe.sequence=0x00;
      messageLength=1;

      while(TRUE)
      {
         sensorApplicationTick();
         EM260Tick();

        if(!input(PUSH_BUTTON2))      //pressing push button two will cause Sleepy
        {                                     //sensor to send out a multicast message
            messagedata[0]=j;
            ezspSendMulticast(&apsframe,hops,nonMemberRadius,messagetag,
                                         messageLength,messagedata);
            if(j>=5)
               j=1;
            else
               j++;
            delay_ms(200);
         }

         if(!input(PUSH_BUTTON1))         //pressing push button one will cause
        {                                       //Sleepy sensor to leave Network
            ezspLeaveNetwork();

          while(input(PUSH_BUTTON2))       //pressing push button two will cause
          {                                        //Sleepy sensor to rejoin Network
            output_toggle(PIN_LED3);
               delay_ms(250);
            }
            output_high(PIN_LED3);
         }

        if(SensorData[0]!=0)               //when sleepy sensor receives new
        {                                       //data it flashes the Red LED
           for(i=0;i<SensorData[0];i++)
            {
               output_low(PIN_LED2);
               delay_ms(100);
               output_high(PIN_LED2);
               delay_ms(100);
            }
            SensorData[0]=0;
         }
      }
   }
}
```

- [ ] Pressing switch A5 on the base station board will cause a number of 1 to ten to be displayed on the base station’s 7 segment displays, and the red LED on the sensor boards to flash that number of times. Also by pressing push button 2 on either of the sensor boards will cause a number of 1 to 5 to be display on the base station’s 7 segment displays, and the red LED on the sensor boards to flash that number of times. Since the base station and sensor boards are part of the multicast group they also receive the multicast messages that they send, that is why when the switch A5 is pressed the number is display on the 7 segment displays, and when push button 2 on the sensor board is pressed it’s red LED is flashed.

15 CONSERVING POWER FOR ZIGBEE™ SLEEP END DEVICES

- ❑ When the sensor boards are configured as sleepy end devices they automatically shut off their radio when they are not transmitting or receiving data. This is done to conserve power since sleepy end devices are usually battery operated. When transmitting or receiving date the EM260 board uses approximately 27 mA of current and when idol the EM260 board uses approximately 5 mA of current, about 1/5 as much as when transmitting and receive data.
- ❑ However, the power requirements for the EM260 board can be reduced further by putting the EM260 chip to sleep when it's not transmitting or receiving data. There are two different sleep states that the EM260 chip can be placed in, both of which only consume approximately 1 uA of current. The two states that the EM260 can enter are Deep Sleep and Power Down. The difference between the two states are that when in Deep Sleep the EM260 chip can be waken by either an external handshake or an internal timer, and when in Power Down the EM260 chip can only be waken by an external handshake.
- ❑ The following example will use the Power Down state of the EM260 chip and the external handshake to put the EM260 board to sleep in between transmissions. This example will also further conserve power by also putting the PIC16F886 chip to sleep in between transmissions, using the watch dog timer to wake up the PIC.
- ❑ Load **EX12A.c** into the coordinator board.
- ❑ Type in the following code (**EX15.c**), compile it and download it onto one of the Sensor boards.

```
#include <16F886.h>
#fuses INTRC,NOWDT,NOLVP,BORV21
#device *=16
#use delay(clock=4000000)

#define SLEEPY_SENSOR 1

#include “em260.h”
#include “ember_utilities.c”

void main()
{
   int8 i=0;
   EmberStatus status;
   EmberEUI64 emLocalEui64;
   EmberApsFrame apsframe;
   int8 messagetag=0x01;
   int8 messagedata[1];
   int8 messageLength;
   int16 indexOrDestination;

   output_high(POWER_UP_PIN);

   if(EmberInitialization())
   {
```

(continued...)

(...continued)

```
      EmberConfig();
      EmberInitializeBinding();
      ezspGetEUI64(&emLocalEui64[0]);
      EmberAddEndpoint();

      EmberJoinNetwork(EMBER_SLEEPY_END_DEVICE);

      indexOrDestination=0x0000;
      apsframe.profileID=0xC00F;
      apsframe.clusterID=0x0001;
      apsframe.sourceEndpoint=0x01;
      apsframe.destinationEndpoint=0x01;
      apsframe.options=0x0000;
      apsframe.groupID=0x0000;
      apsframe.sequence=0x00;
      messageLength=1;

      setup_wdt(WDT_ON | WDT_2304MS | WDT_TIMES_1);  //sets up wdt to wake up
                                                  //PIC every 2.3 seconds
      status=ezspNetworkState();
      while(status!=EMBER_JOINED_NETWORK)
      {
         sensorApplicationTick();
         status=ezspNetworkState();
      }

      while(TRUE)
      {
         sensorApplicationTick();
         EM260Tick();

         messagedata[0]=i;
         ezspSendUnicast(EMBER_OUTGOING_DIRECT,indexOrDestination,&apsfr
ame,messagetag,
                    messageLength,&messagedata[0]);
         i++;

         EM260Sleep();     //puts EM260 chip to sleep

         #asm
         SLEEP             //puts PIC16F886 chip to sleep
         #endasm

         EM260Wake();      //wakes up EM260 chip
      }
   }
}
```

❑ The above code will initialize the Sensor board to be an EMBER_SLEEPY_END_DEVICE and join the network. The code will them transmit Unicast messages to the coordinator and then put the EM260 chip and the PIC to sleep until the next transmission time. The coordinator is already setup to receive these messages and display them on the 7-segment display.

16 MIGRATING TO YOUR OWN HARDWARE

- ❑ The EM260 radio module used in CCS's Wireless – Ember ZigBee Development Kit can be easily migrated your own hardware with any PIC. All that is required is for the PIC to have the hardware SPI1 peripheral available along with four additional I/O pins. The four additional I/O pins that are used by the EM260 module are defined in the em260.h as follows:

```
#define PIN_EZSP_INT        PIN_B0
#define PIN_EZSP_WAKE       PIN_B1
#define PIN_EZSP_RESET      PIN_B2
#define PIN_SSEL_INT        PIN_B3
```

  These pin defines can be changed by adding the previous defines with the desired pins just before the #include "em260.h" line in your main program

- ❑ The EM260 radio module is connected to CCS's development board via 2x5 pin connector and can be connected to any board by the same method. Please refer to the schematics at the end of this booklet for the correct wiring of the connector.

References

Ember EM260 References:

- EM260 Fact Sheet: http://www.ember.com/pdf/ember-EM260.pdf
- EM260 Data Sheet: http://www.ember.com/pdf/EM260/EM260_Datasheet.pdf
- Ember Applications Developer Guide: http://www.ember.com/pdf/EM260/120-0066-000J_appDevGuide.pdf
- EmberZNet API Guide: http://www.ember.com/pdf/EM250/120-3006-000O_EmberZNet_API/index.htm

Ember Software References:

- Insight Desktop Quick Start Video: http://www.ember.com/isd/ISD_quickstart.html
- Insight Desktop 30-day evaluation: http://www.ember.com/download_file.html?f=InSight_Desktop_1.2.1b42.exe
- Insight Desktop Users Guide: http://www.ember.com/pdf/120-4005-001_InSightDesktop.pdf
- Getting Started with Insight Desktop: http://www.ember.com/pdf/120-4011-000_ISDQuickStart.pdf

Customers can download the ZigBee specification at:
http://www.zigbee.org/en/spec_download/download_request.asp

On The Web

Comprehensive list of PIC® MCU Development tools and information	www.mcuspace.com
Microchip Home Page	www.microchip.com
CCS Compiler/Tools Home Page	www.ccsinfo.com
CCS Compiler/Tools Software Update Page	www.ccsinfo.com click: Support → Downloads
C Compiler User Message Exchange	www.ccsinfo.com/forum
Device Datasheets List	www.ccsinfo.com click: Support → Device Datasheets
C Compiler Technical Support	support@ccsinfo.com
MCU Space	www.mcuspace.com

Other Development Tools

EMULATORS
The ICD used in this booklet uses two I/O pins on the chip to communicate with a small debug program in the chip. This is a basic debug tool that takes up some of the chip's resources (I/O pins and memory). An emulator replaces the chip with a special connector that connects to a unit that emulates the chip. The debugging works in a simulator manner except that the chip has all of its normal resources, the debugger runs faster and there are more debug features. For example an emulator typically will allow any number of breakpoints. Some of the emulators can break on an external event like some signal on the target board changing. Some emulators can break on an external event like some that were executed before a breakpoint was reached. Emulators cost between $500 and $3000 depending on the chips they cover and the features.

DEVICE PROGRAMMERS
The ICD can be used to program FLASH chips as was done in these exercises. A stand alone device programmer may be used to program all the chips. These programmers will use the .HEX file output from the compiler to do the programming. Many standard EEPROM programmers do know how to program the Microchip parts. There are a large number of Microchip only device programmers in the $100-$200 price range. Note that some chips can be programmed once (OTP) and some parts need to be erased under a UV light before they can be re-programmed (Windowed). CCS offers the Mach X which is a stand-alone programmer and can be used as an in-circuit debugger.

PROTOTYPING BOARDS
There are a large number of Prototyping boards available from a number of sources. Some have an ICD interface and others simply have a socket for a chip that is externally programmed. Some boards have some advanced functionality on the board to help design complex software. For example, CCS has a Prototyping board with a full 56K modem on board and a TCP/IP stack chip ready to run internet applications such as an e-mail sending program or a mini web server. Another Prototyping board from CCS has a USB interface chip, making it easy to start developing USB application programs.

SIMULATORS
A simulator is a program that runs on the PC and pretends to be a microcontroller chip. A simulator offers all the normal debug capability such as single stepping and looking at variables, however there is no interaction with real hardware. This works well if you want to test a math function but not so good if you want to test an interface to another chip. With the availability of low cost tools, such as the ICD in this kit, there is less interest in simulators. Microchip offers a free simulator that can be downloaded from their web site. Some other vendors offer simulators as a part of their development packages.

CCS Programmer Control Software

The CCSLOAD software will work for all the CCS device programmers and replaces the older ICD.EXE and MACHX.EXE software. The CCSLOAD software is stand-alone and does not require any other software on the PC. CCSLOAD supports ICD-Sxx, ICD-Uxx, Mach X, Load-n-Go, and PRIME8.

Powerful Command Line Options in Windows and Linux

· Specify operational settings at the execution level
· Set-up software to perform, tasks like save, set target Vdd
· Preset with operational or control settings for user

Easy to use Production Interface

· Simply point, click and program
· Additions to HEX file organization include associating comments or a graphic image to a file to better ensure proper file selection for programming
· Hands-Free mode auto programs each time a new target is connected to the programmer
· PC audio cues indicate success and fail

Extensive Diagnostics

· Each target pin connection can be individually tested
· Programming and debugging is tested with known good programs
· Various PC driver tests to identify specific driver installation problems

Enhanced Security Options

· Erase chips that failed programming
· Verify protected code cannot be read after programming
· File wide CRC checking

Automatic Serial Numbering Options

· Program memory or Data EEPROM
· Incremented, from a file list or by user prompt
· Binary, ASCII string or UNICODE string

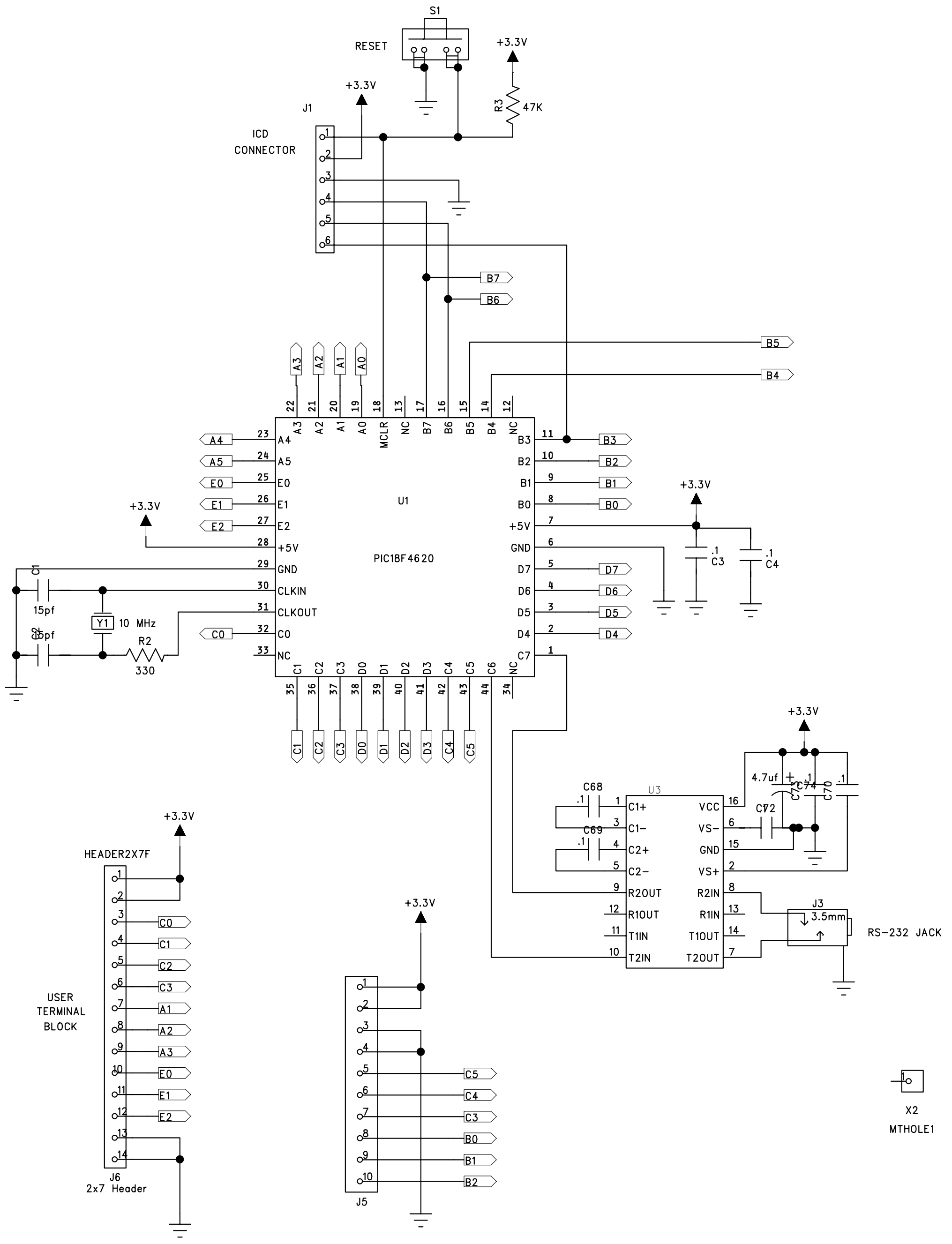
CCS IDE owners can use the CCSLOAD program with:

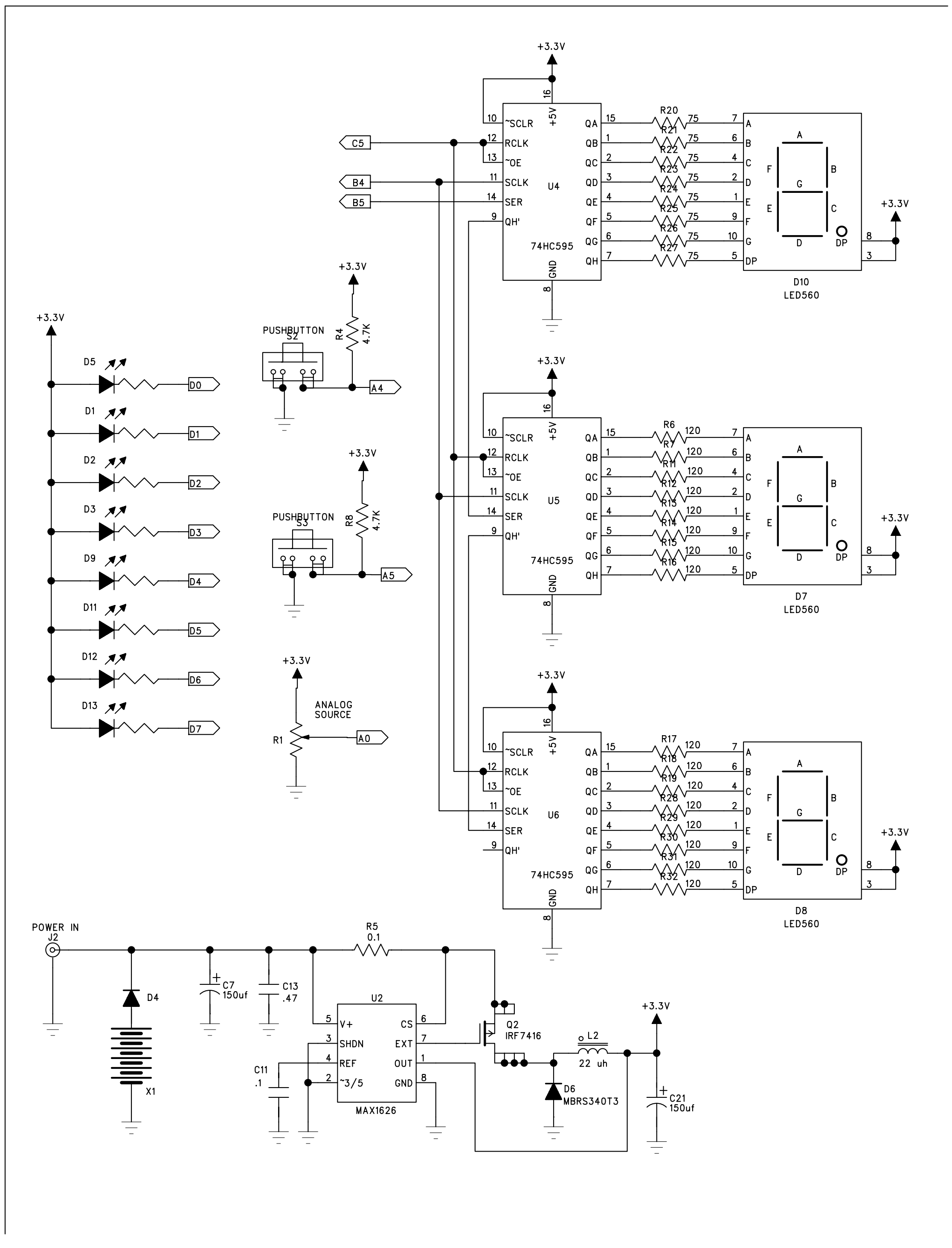
· MPLAB®ICD 2/ICD 3
· MPLAB®REAL ICE™
· All CCS programmers and debuggers

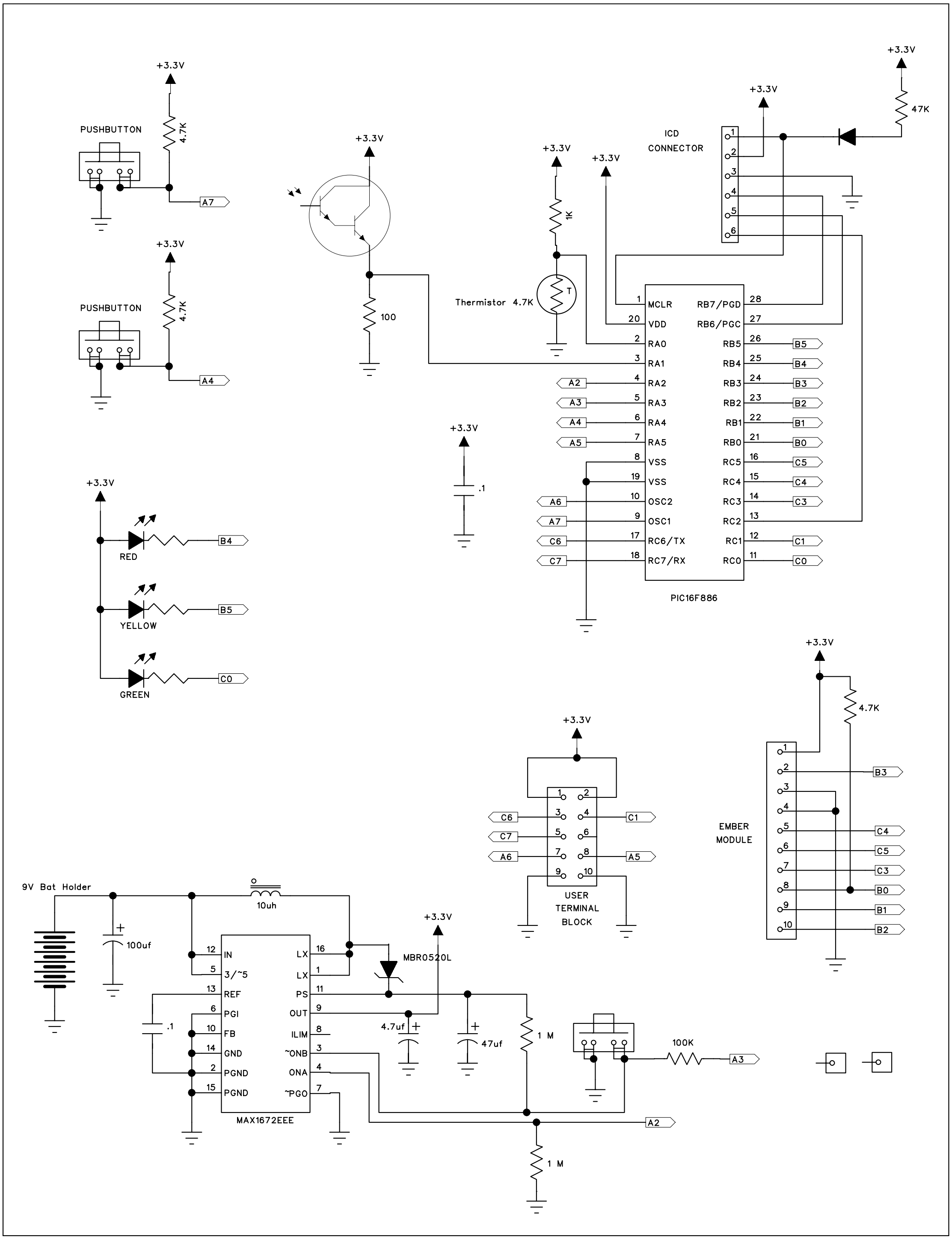
How to Get Started:

Step 1: *Connect Programmer to PC and target board. Software will auto-detect the programmer and device.*
Step 2: *Select Hex File for target board.*
Step 3: *Select Test Target. Status bar will show current progress of the operation.*
Step 4: *Click "Write to Chip" to program the device.*

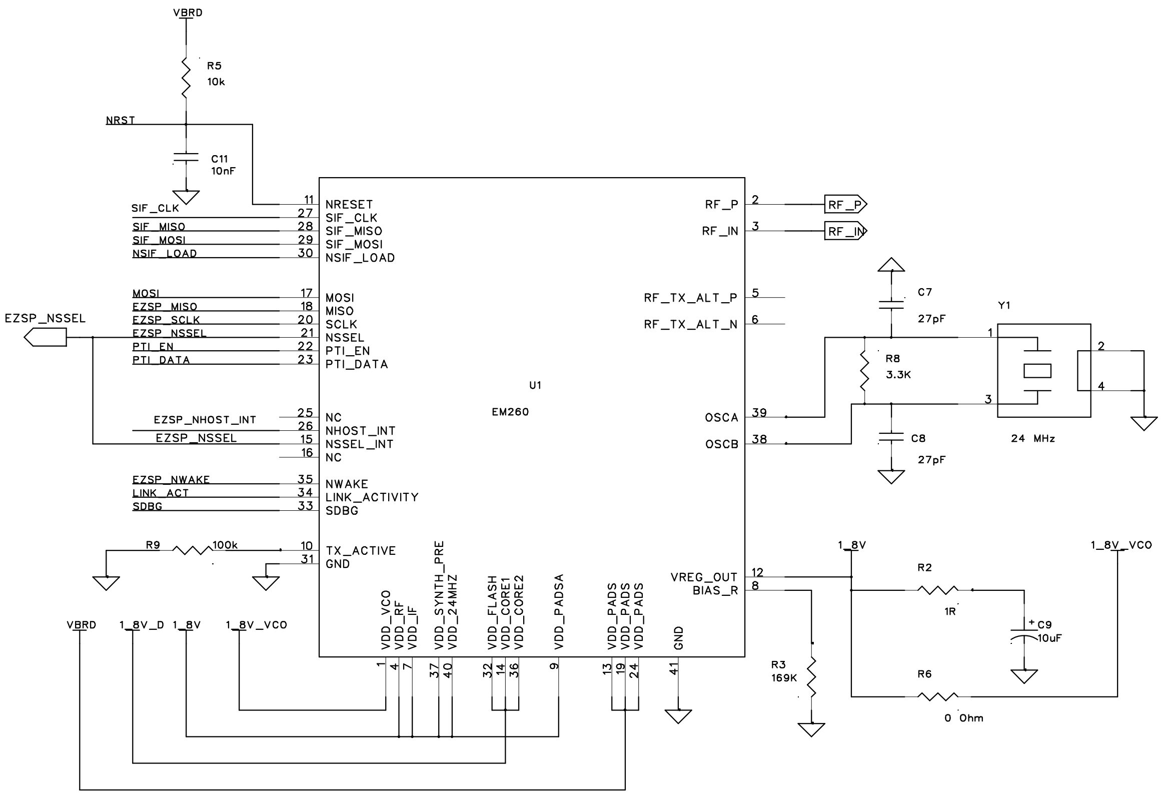
Use the Diagnostics tab for troubleshooting or the ccsload.chm help file for additional assistance.


S1
RESET
+3.3V
R3
47K
J1
ICD
CONNECTOR
B7
B6
B5
B4
A3
A2
A1
A0
MCLR
NC
U1
PIC18F4620
A4
A5
E0
E1
E2
+5V
GND
CLKIN
CLKOUT
C0
B3
B2
B1
B0
D7
D6
D5
D4
C7
C1
C2
C3
D0
D1
D2
D3
C4
C5
C6
C1
15pf
C2
15pf
Y1
10 MHz
R2
330
.1
C3
.1
C4
C68
C69
U3
C1+
C1−
C2+
C2−
R2OUT
R1OUT
T1IN
T2IN
VCC
VS−
GND
VS+
R2IN
R1IN
T1OUT
T2OUT
4.7uf
C71
C74
C70
C72
J3
3.5mm
RS-232 JACK
HEADER2X7F
USER
TERMINAL
BLOCK
J6
2x7 Header
J5
X2
MTHOLE1

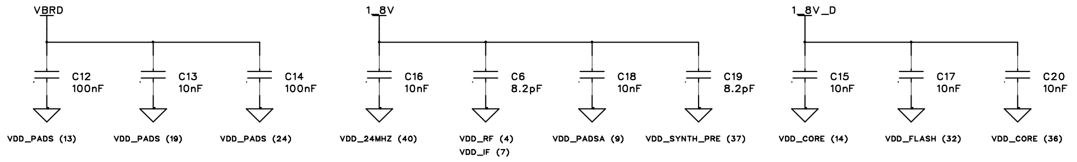

+3.3V
U4
74HC595
~SCLR
RCLK
~OE
SCLK
SER
QH'
+5V
GND
QA
QB
QC
QD
QE
QF
QG
QH
C5
B4
B5
R20 75
R21 75
R22 75
R23 75
R24 75
R25 75
R26 75
R27 75
D10
LED560
U5
74HC595
R6 120
R7 120
R11 120
R12 120
R13 120
R14 120
R15 120
R16 120
D7
LED560
U6
74HC595
R17 120
R18 120
R19 120
R28 120
R29 120
R30 120
R31 120
R32 120
D8
LED560
A
B
C
D
E
F
G
DP
PUSHBUTTON
S2
R4 4.7K
A4
PUSHBUTTON
S3
R8 4.7K
A5
ANALOG SOURCE
R1
A0
D5
D1
D2
D3
D9
D11
D12
D13
D0
D1
D2
D3
D4
D5
D6
D7
POWER IN
J2
D4
X1
C7 150uf
C13 .47
R5 0.1
U2
MAX1626
V+
SHDN
REF
~3/5
CS
EXT
OUT
GND
C11 .1
Q2 IRF7416
L2 22 uh
D6 MBRS340T3
C21 150uf

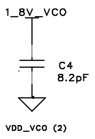


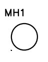
+3.3V
PUSHBUTTON
4.7K
A7
PUSHBUTTON
4.7K
A4
100
Thermistor 4.7K
1K
T
ICD
CONNECTOR
47K
MCLR
VDD
RA0
RA1
RA2
RA3
RA4
RA5
VSS
VSS
OSC2
OSC1
RC6/TX
RC7/RX
RB7/PGD
RB6/PGC
RB5
RB4
RB3
RB2
RB1
RB0
RC5
RC4
RC3
RC2
RC1
RC0
PIC16F886
A2
A3
A4
A5
A6
A7
C6
C7
B5
B4
B3
B2
B1
B0
C5
C4
C3
C1
C0
.1
RED
YELLOW
GREEN
B4
B5
C0
EMBER
MODULE
4.7K
USER
TERMINAL
BLOCK
9V Bat Holder
100uf
10uh
MBR0520L
IN
3/~5
REF
PGI
FB
GND
PGND
PGND
LX
LX
PS
OUT
ILIM
~ONB
ONA
~PGO
MAX1672EEE
4.7uf
47uf
1 M
100K
A3
A2
1 M



Decoupling Capacitors







Mounting Hole

Disclaimer

Design used with the permision of

Ember Corporation
343 Congress St 5th floor
Boston MA 02210

The information in this document is the proprieetary and confidential property of Ember Corporation and is priviliged. No part of the drawing or information may be duplicated or otherwise used without the express permission of Ember.

Title	EMBM Ember Radio Module	
Size B	Document EMBM	Rev 1
Date: Feb 26, 2007	Sheet 1 of 2	

Balun Circuit with Harmonic Filter

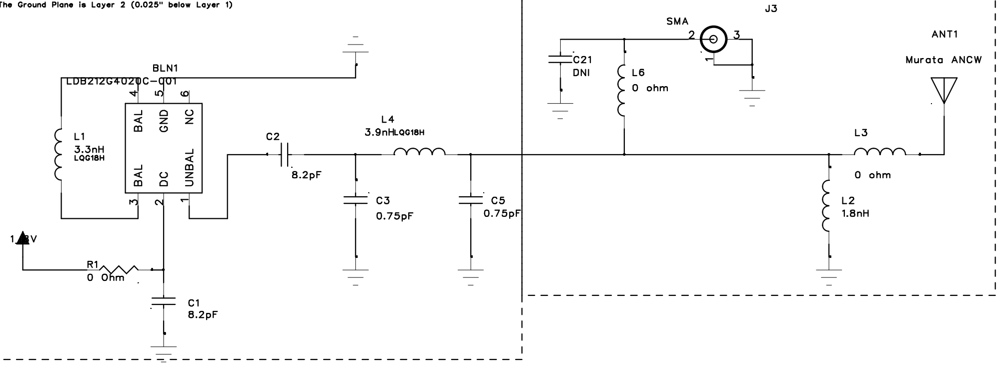
The Ceramic Balun is a 50 / 200 Ohm conversion which stabilizes the load observed by the PA across the 2.4 GHz ISM Band. L1 should be an 0603 Mulitlayer which provides a higher Q at a lower cost.

L4, C3 and C5 make up a 0.1dB Ripple, Cheby Harmonic filter which maximizes the conducted 2nd Harmonic margin. The Ground Plane is Layer 2 (0.025" below Layer 1)

Test Connector and Antenna

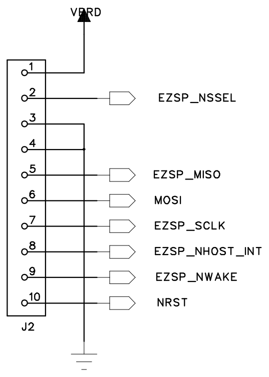
In order to allow for connection to an external antenna, the footprints for J3 and L6 will be placed, but not installed. C21 can be used along with L6 to tune the external antenna to 50 Ohms.

L2 and L3 are recommended by Murata to match the ANCW antenna to 50 Ohms



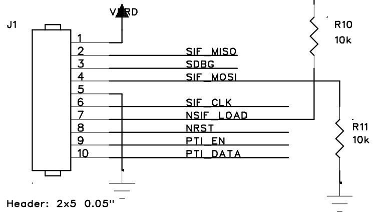
5

EZSP Interface (SPI Only)

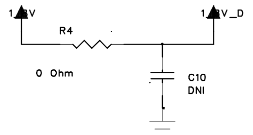


D

Debug Header



Header: 2x5 0.05"

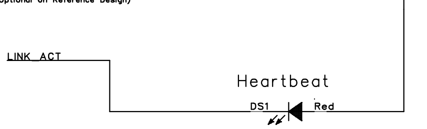


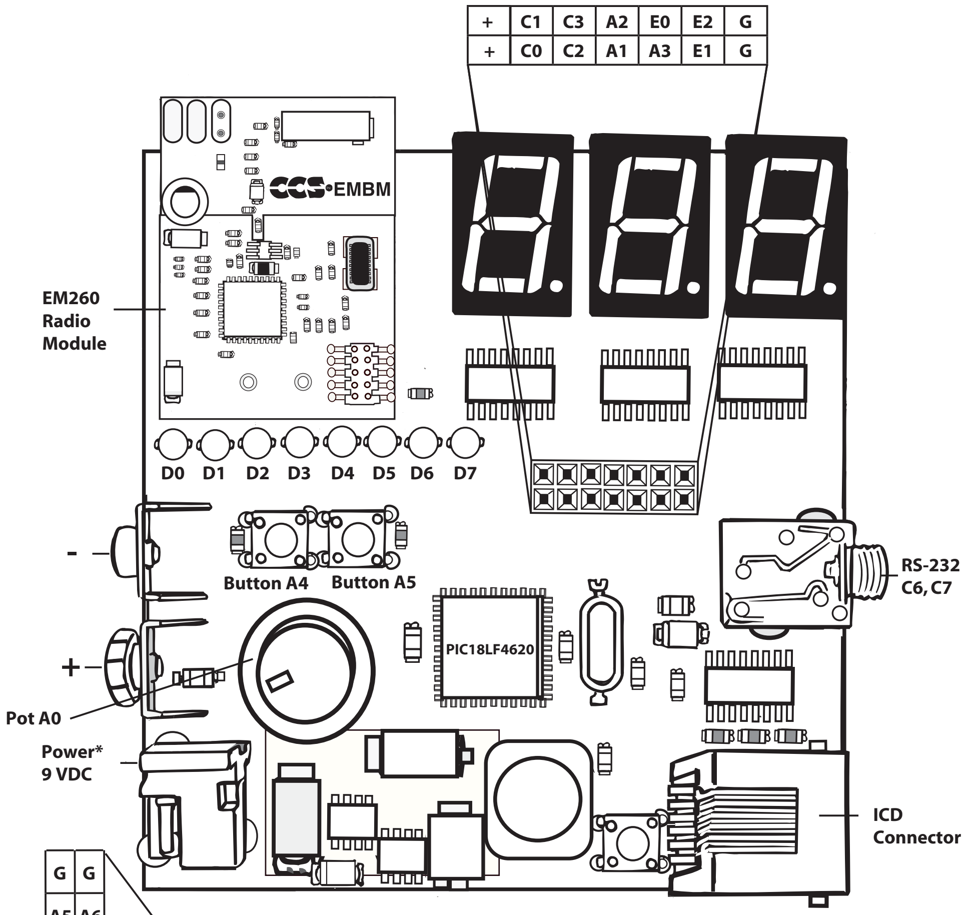
Digital Power Filter

(Optional on Reference Design)

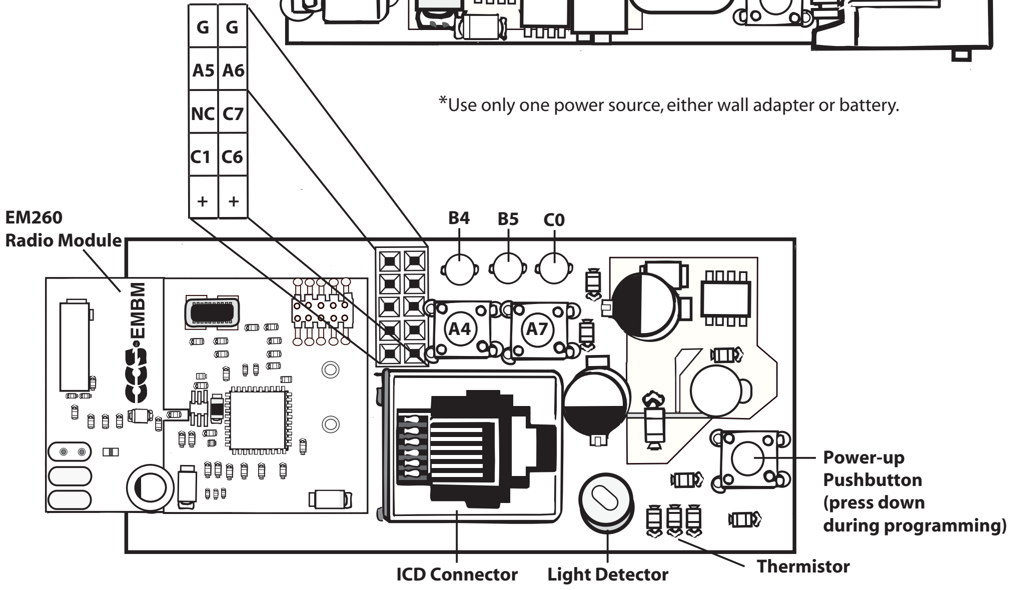
EmberZnet Stack Indicator

(Optional on Reference Design)





*Use only one power source, either wall adapter or battery.


G
G
A5
A6
NC
C7
C1
C6
+
+
EM260 Radio Module
CCS•EMBM
B4
B5
C0
A4
A7
Power-up Pushbutton (press down during programming)
ICD Connector
Light Detector
Thermistor