

Volume

1

A-WIT TECHNOLOGIES INC.

... a passion for execution ...

Embedded Web Server with the C Stamp

Version 1.0

A-WIT TECHNOLOGIES INC.

Embedded Web Server with the C Stamp Reference Guide Manual

© A-WIT Technologies Inc.
Phone (800) 985-AWIT • Fax (800) 985-2948

Table of Contents

Registering Your C Stamp or C Stamp	
Related Product	1
Notices	1
Getting Support	2
Installing the Microchip MPLAB and C	
Compiler Software	2
Installing the A-WIT C Stamp Quick	
Programmer	4
Installing the USB Software	4
Setting Up the C Stamp Software	
Templates	5
Documentation	5
Materials	5
Configuration	5
Router Configuration	6
GPIO Test Configuration	6
Messenger Demo Configuration	8
C Stamp as a Serial Receiver	8
C Stamp as a Serial Transmitter	10
Final System Design	11
Verification of Functionality	25
Conclusion	29
Terms & Conditions	31

Introduction

This document describes how to enable C Stamp based projects with the ability to connect to the internet and provide information and control to remote users. The Implementation of an Embedded Web Server through use of the C Stamp interfaced with the Digi Connect ME through use of the CS492000 networking module is a multi-step process. The first step is the configuration of all parts involved, especially the router connecting the Connect ME to the internet and the Connect ME itself to function with the software designed for it. The next step is the development of code for the C Stamp to accept serial input from the Connect ME and respond accordingly so that the input can be verified. The subsequent step is to establish a communication between the C Stamp and the Connect ME such that the Connect ME can receive serial data transmitted from the C Stamp. The next step is the design and implementation of a final system, utilizing both serial sending and receiving to drive both an applet run off the Connect ME and code run off the C Stamp to work together to create a functional system. The final step is the testing of this system for errors that could cause it to function improperly, and once that is completed, the Embedded Web Server is implemented successfully.

Registering Your C Stamp or C Stamp Related Product

At A-WIT Technologies we respect your privacy; however, we do ask you to register your C Stamp or C Stamp related product, so you can receive free of charge product updates. The registration procedure is simple. Just send an e-mail to tech_support@awit.com with the word “REGISTRATION x” in the subject line, where “x” is the product number that you purchased. If you purchased more than one product, send an e-mail for each different product.

Notices

CSTAMP™ and CSTAMP™ Related Hardware Products, Software Products and Documentation are developed and distributed by A-WIT Technologies, Inc. All rights reserved by A-WIT Technologies, Inc. A-WIT

SOFTWARE OR FIRMWARE AND LITERATURE IS PROVIDED “AS IS,” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL A-WIT BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY ARISING OUT OF OR IN CONNECTION WITH THE SOFTWARE OR FIRMWARE OR THE USE OF OTHER DEALINGS IN THE SOFTWARE OR FIRMWARE.

MPLAB C-18 and MPLAB C-18 Users Guide is reproduced and distributed by A-WIT Technologies, Inc. under license from Microchip Technology Inc. All rights reserved by Microchip Technology Inc. MICROCHIP SOFTWARE OR FIRMWARE AND LITERATURE IS PROVIDED “AS IS,” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL MICROCHIP BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY ARISING OUT OF OR IN CONNECTION WITH THE SOFTWARE OR FIRMWARE OR THE USE OF OTHER DEALINGS IN THE SOFTWARE OR FIRMWARE.

Getting Support

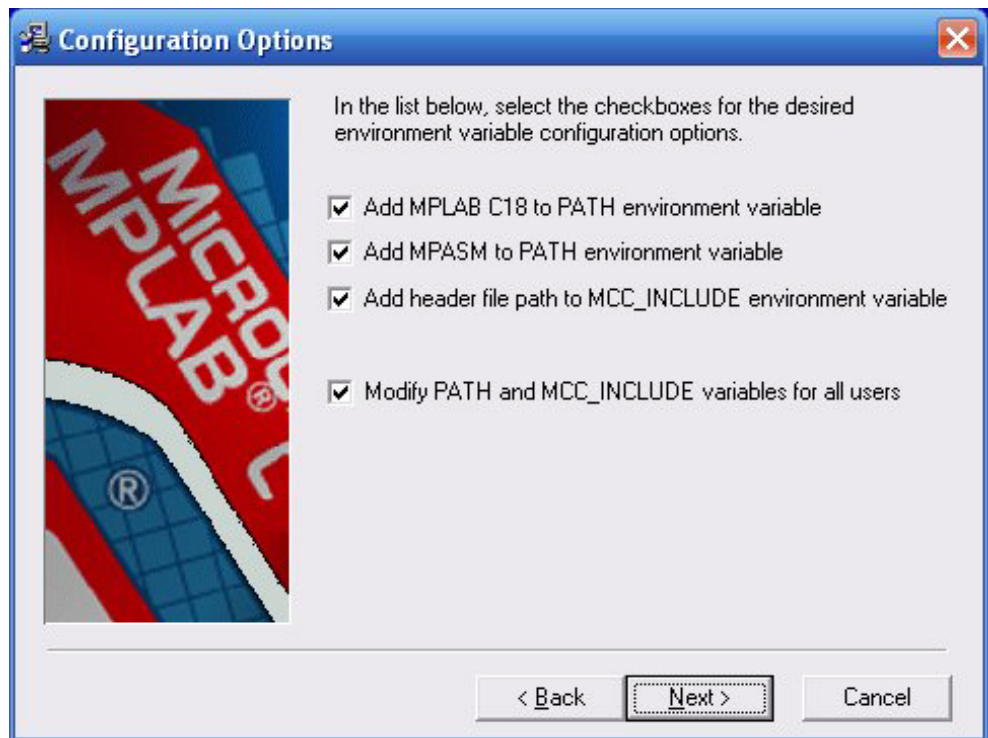
If possible, please check the C Stamp website www.C Stamp.com under SUPPORT for any updates to documentation, changes, or notices that may have become available since your Installation CD was produced. If you continue to have any issues for which a solution is not found in the aforementioned website, please e-mail tech_support@a-wit.com for help.

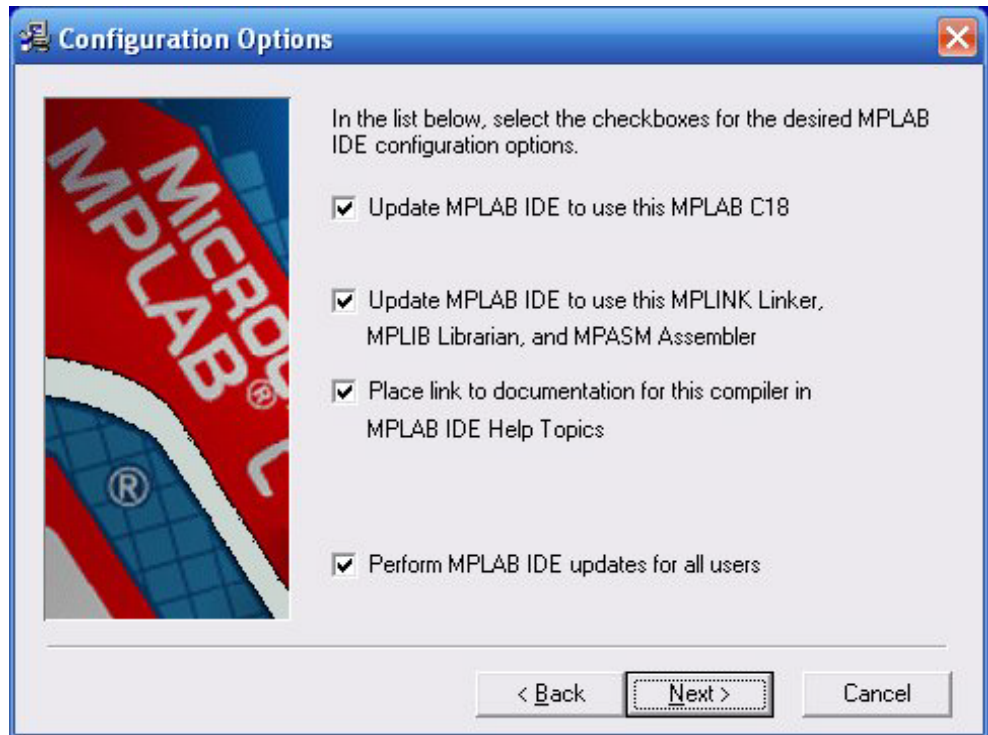
Installing the Microchip MPLAB and C Compiler Software

The first step is to install the Microchip MPLAB software that you will use to develop your programs.

Insert your A-WIT provided Installation CD in your CD drive. Go to the MPLAB directory in the CD and double click on the “MPLAB vX.XX Install” file in that directory. Follow the installation steps, prompts, and directions provided by the installer software, accepting all the default options.

After the MPLAB installation is complete, switch to the C18 directory in the CD, and double click on the file in that directory. Follow the installation steps, prompts, and directions provided by the installer software, accepting all the default options. The only exceptions to accepting all the default options is that on the 5th and 6th windows of the installation process for the C18 Compiler, you have to select everything as shown in the figures below. This will ensure that MPLAB is configured to use the C18 Compiler.





Installing the A-WIT C Stamp Quick Programmer

To install the A-WIT C Stamp Quick Programmer, switch to the CSTAMPQP directory in the CD using Windows Explorer, and double click on the file in that directory. Follow the installation steps, prompts, and directions provided by the installer software, accepting all the default options.

Installing the USB Software

If you purchased a product with a USB download cable, make sure that the A-WIT provided CD is in the CD drive of your PC and insert the USB cable in the USB port of your PC. Windows auto detects the new USB device. If Windows prompts you to install drivers for the USB cable device, follow the installation steps, prompts, and directions provided by the installer software, accepting all the default options.

After the USB adapter has been installed, open a Windows Explorer window from the Accessories sub-menu in the Start menu, and right click on My Computer. Proceed to select Properties, and then select the Hardware tab. Click on the Device Manger button, and expand the Ports (COM & LPT) branch. Make a note of the COM port that has been assigned to the USB-to-Serial adapter. This is the port that should be selected in the C Stamp programmer software.

Setting Up the C Stamp Software Templates

To set up the C Stamp Software Templates, switch to the CSTAMP_Template directory in the CD using Windows Explorer, and double click on the file in that directory. Follow the installation steps, prompts, and directions provided by the installer software, accepting all the default options.

Documentation

Copy the DOCS directory from the C Stamp Installation CD to your C:\A-WIT directory. This directory contains all the C Stamp related documentation in PDF format.

Materials

Table 1 - Items Used to Complete the Embedded Web Server Project.

Item	Quantity
A-WIT CS215001 BOL Starter Kit with C Stamp 48-Pin Module	1
A-WIT CS492000 Internet/Ethernet Network Connectivity Kit	1
Linksys WRT54G Wireless Router	1
Workstation with Eclipse for Java and MPLAB IDE	1
DM7404N Hex Inverting Gates	1
LED	2
1 K Ω Resistor	2

Configuration

In order to complete this project, it is necessary to configure the parts utilized. The first item that needs to be configured for use is the router, which is necessary to allow the Digi Connect ME to function as a remote web server. The C Stamp requires little to no configuration besides constructing the circuit for the final design and developing code to interface with the Digi Connect ME, which will be covered in the Serial Transmitter, Serial Receiver, and Final system sections. The Digi Connect ME can be utilized in a variety of configurations; one to test the GPIO pins, and one to function as a web server through use of the messenger demo applet. Once all the parts involved are configured, the system is ready for design and development.

Router Configuration

The Digi Connect ME is designed to interface with a DHCP router. For this purpose, a Linksys WRT54G router was utilized to provide this connection. Once the Digi Connect ME has been turned on and connected to the router through the use of a CAT5 Ethernet cable, the router must be accessed by a PC in order to determine the IP address of the Digi connect ME. For a WRT54G that has not been assigned a non-default administrator username and password, the PC must also be connected to the router by an Ethernet cable. After logging in to the router, there are two steps that must be completed. The first is that the IP address of the Connect ME must be determined, by viewing the router's DHCP Clients Table, found under the status heading for the WRT54G. This can be also be accomplished via running the Digi Device Discovery tool from the PC connected to the router either through an Ethernet cable or wirelessly. The IP address of the router should also be taken note of at this time, as it will be used to access the Connect ME remotely later. The next step is to enable port forwarding on the router, found under the Applications and Gaming heading for the WRT54G, to forward all traffic on ports 80, 2001, and 2101 to the IP address of the Connect ME. This allows for remote access of the Connect ME for computers outside the network. Once the router has been configured properly, the Connect ME can be accessed both locally and remotely.

GPIO Test Configuration

Once the router is properly configured, the next step is to access the Digi Connect ME and ensure it is properly connected with the C Stamp. The Documentation for the CS492000 Networking module provides specific instructions for this, including instructions for how to modify the Connect ME and code to run on the C Stamp. The Connect ME needs to be configured for this test by setting each of the GPIO pins to output (Figure 1). Once this is accomplished, modifying the pins' states in the system information menu (Figure 2) will drive the C Stamp to modify its output. The C Stamp continually accepts input for the pins the GPIO of the Connect ME outputs to, and modifies its LEDs accordingly (Figure 3). This test, once completed successfully, proved the C Stamp and Connect ME were interfaced properly and no hardware issues were present for the GPIO pins.

GPIO Configuration		
General Purpose Input/Output Pins		
	Mode	Initial Output State
Pin 1:	Out	De-asserted
Pin 2:	Out	De-asserted
Pin 3:	Out	De-asserted
Pin 4:	Out	De-asserted
Pin 5:	Out	De-asserted
Apply		

Figure 1 - Menu for Configuring GPIO Pins for the Connect ME

System Information		
▶ General		
▼ GPIO		
General Purpose I/O (GPIO) pins can be asserted or de-asserted if they are configured as output.		
	Asserted	De-asserted
Pin 1:	<input type="radio"/>	<input checked="" type="radio"/>
Pin 2:	<input type="radio"/>	<input checked="" type="radio"/>
Pin 3:	<input type="radio"/>	<input checked="" type="radio"/>
Pin 4:	<input type="radio"/>	<input checked="" type="radio"/>
Pin 5:	<input type="radio"/>	<input checked="" type="radio"/>
Set Pins Refresh		
▶ Serial		
▶ Network		

Figure 2 - Menu for Modifying the GPIO Output for the Connect Me

```
#include "CS110000.h"

void main(void){
    // CS4X2000 pin definitions
    // Outputs at Net Mod
    NIBBLE Pin1 = 15, Pin2 = 14, Pin3 = 13,
        Pin4 = 12, Pin5 = 11;
    BYTE LED1 = 46, LED2 =45, LED3 =44, LED4 =43,
    LED5 =42;
    while(TRUE){
        if(GTPIND(Pin1)) STPIND(LED1, HIGH);
        else STPIND(LED1, LOW);
        if(GTPIND(Pin2)) STPIND(LED2, HIGH);
        else STPIND(LED2, LOW);
    }
}
```

```

        if(GTPIND(Pin3)) STPIND(LED3, HIGH);
        else STPIND(LED3, LOW);
        if(GTPIND(Pin4)) STPIND(LED4, HIGH);
        else STPIND(LED4, LOW);
        if(GTPIND(Pin5)) STPIND(LED5, HIGH);
        else STPIND(LED5, LOW);
    }
}

```

Figure 3 - Code for the C Stamp to Check for GPIO Input from Connect ME

Messenger Demo Configuration

The next step in developing the embedded web server is to implement a demonstration system that already supported this task, so that it could be modified accordingly. For this purpose, the Messenger demo applet supplied with the Digi Connect ME is loaded onto the module and run to test its effectiveness. For this demo to run, the first step is to configure the Connect ME to have its serial ports run from TCP socket communication. This configuration has the Connect ME forward all data received within TCP packets to its serial Txd line, and for all serial data received on its Rxd line to be forwarded back to a Computer it has an open connection with. To test that the messenger demo applet worked properly, the serial subheading under the system information menu could be accessed to check how much data the system sent out. Once this is verified, the next step is to design a system for the C Stamp to receive and interpret this data.

C Stamp as a Serial Receiver

To have the C Stamp as a device for receiving serial communication from the connect ME, use a multiple-byte string following a waitOn character.

```

void display(BYTE buffer[], int length)
{
    int i;
    BYTE pins[8];
    /* These pins store the data received by the
       buffer. */
    pins[0]=39; pins[1]=40;
    pins[2]=41; pins[3]=42;
    pins[4]=43; pins[5]=44;
    pins[6]=45; pins[7]=46;

    for (i = 0; i < length; i++)
    {
        BYTEOUT(buffer[i], pins);
        PAUSE(1000);
    }
}

```

```

    }

    BYTEOUT(0xF0, pins);
    PAUSE(1000);
    BYTEOUT(0x00, pins);
}

```

Figure 4 - Code Snippet for Method Used to Output Received Serial Input

The display method (Figure 4) was utilized to test the input the C Stamp received. It functions by iterating through each byte in the buffer array, outputting its data utilizing the LEDs attached to the C Stamp BOL pins 39-46. The outputting of 0xF0 and 0x00 following the display of the bytes contained in the array serves as a message that the C Stamp is ready to receive more input.

```

    BYTEOUT(0xA5, pins);
    SERIN2(0, 0, baud, 8, 0, 0, buffer, 3, 23,
          waiton);
    Display(buffer, 3);

```

Figure 5 - Code Snippet Showing WaitOn for Many Bytes Approach of C Stamp Accepting Serial Input

The receive approach waits to receive a stream of bytes from the Connect ME (Figure 5). It allows the C Stamp to check that it was receiving well formatted data. The test string sent by the Messenger Demo Applet in this configuration is of the format “!x\n!x\n” and the output of the C Stamp would show whether the string was received properly by checking that the ‘x’ characters matched and that the ‘\n’ and ‘!’ characters were present in the right locations. This approach is close to what would be required for the Embedded Web Server to function properly, but some improvements could still be made with 38400 bps as the Baud Rate.

For the approach chosen that would wait for many bytes following the ‘!’ character, it was necessary to find a format for messages sent to the C Stamp that would allow it to check for accuracy of the data sent from the Connect ME. It was determined that increasing the number of waitOn characters in the messages sent to the C Stamp would increase the likelihood of accepting signals from the Connect ME, as would increasing the length of the signals sent. In order to provide an acceptable level of error checking and decrease the odds of signal deterioration, a 3-byte string was determined to be the ideal message length. The message format would therefore contain three bytes: the first and last would contain the instruction and be checked against each other, and the middle byte would be the waitOn character. To accomplish this, the Connect ME would send signals of the format “!x!x!x” and the C Stamp would wait to receive signals of the format ‘x!x’ following the ‘!’ character. The extra byte in the string sent from

the Connect ME would not be added to the buffer, so they would not adversely affect the functionality of the C Stamp.

C Stamp as a Serial Transmitter

The Embedded Web Server needs the C Stamp to function as a serial transmitter to allow for two-way communication with the Digi Connect ME. This functionality allows the system to function in multiple applications, such that it can inform remote users about its state. In order to accomplish this task, code needed to be developed for the C Stamp in order to properly send serial signals. The signals sent needed to be modified to function with the Messenger Demo Applet. After this was accomplished, two way serial communications were properly established, allowing for the implementation of the final system design.

A hardware solution needs to be implemented to allow the Connect ME to receive serial data from the C Stamp. Soldering a wire to the Connect ME's pin 7 is not an option, so it is necessary to wrap a wire around the pin to establish a connection to the C Stamp's output. Once the connection was established, code was run from the C Stamp to echo back input received from the Connect ME's transmission pin, pin 23 on the C Stamp, to the Connect ME's reception pin, pin 22 on the C Stamp (Figure 6).

```

BYTEOUT(0xA5, pins);
SERIN2(0, 0, baud, 8, 0, 0, buffer, 3, 23,
      waiton);
SEROUT2(0, 0, baud, 0, 8, 0, 0, buffer, 3,
      22);

```

Figure 6 - Code Snippet Showing Echoing Back of Input from Connect ME

With the proper connection established, the Connect ME showed on its system information menu that it was receiving the correct amount of data from the C Stamp. However, strings generated by the C Stamp were not being displayed in the Messenger Applet. The solution to this issue was implemented after careful inspection of the Messenger Applet code. The Messenger applet received data utilizing the `readLine()` method, which waits until a string contains the '\n' character for a new line before reading the information. Since the strings sent by the C Stamp at this point did not contain the new line character, they would never be received by the Messenger Applet. Simply adding the characters "\r\n," with '\r' corresponding to a carriage return, to the end of the byte string sent by the C Stamp resolved this issue. With both sending and receiving serial data established, the Embedded Web Server was ready to implement its final design.

designing C Stamp code specifically to carry it out and test it by simulating it with the Messenger Demo Applet.

The C Stamp code needed to utilize the sending and receiving commands established while supporting the handshaking method developed to ensure the commands are executed properly. The first step of this process is waiting for serial input from the Connect ME. The second step is for the C Stamp to ensure the signal is well formatted and response with a verification request. The next step is to wait for the confirmation request, and if the C Stamp receives another command signal instead, it sends another verification request. Once the C Stamp received a confirmation request, it sends a confirmation response and modifies its state accordingly. These steps (Figure 8) ensure the C Stamp handshakes properly.

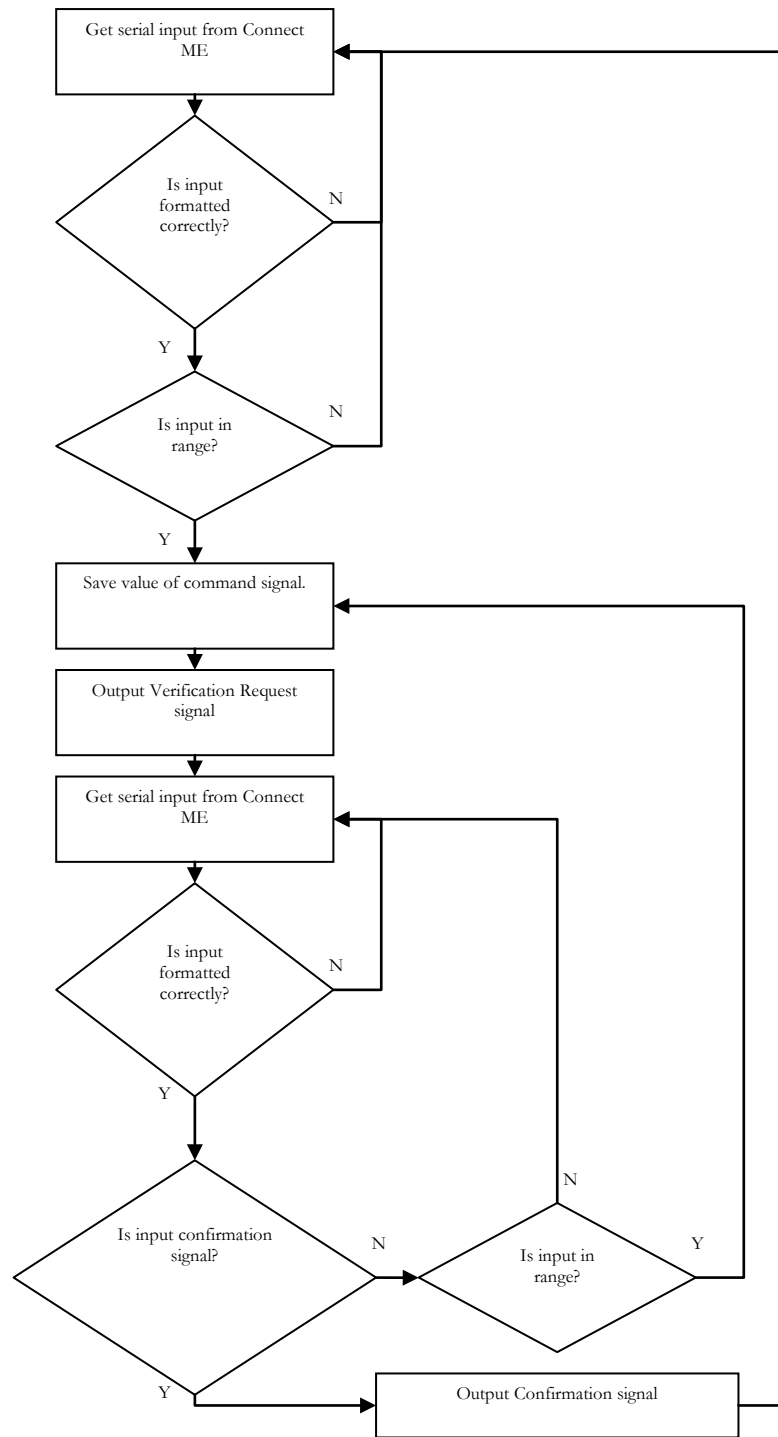


Figure 8 – Flow chart of C Stamp Execution

```

void main(void)
{

```

```

        BYTE buffer[] = "888\r\n";    // store input
                                        // from terminal

        STPIND(16,LOW);
        STPIND(17,LOW);

        while (1) input(buffer, 5);

        END();
    }

```

Figure 9 – Code Snippet showing main method of C Stamp program

The first portion of the program that needs to be established is the continuous loop that has the system wait for serial input after outputting a confirmation signal. This is accomplished in the main method of the C Stamp program (Figure 9). The buffer is established at this point, so that its length can be modified to ease changing the functionality of the program. The data contained in the first 3 bytes is unimportant, as it will be replaced before it is ever output, but the last two bytes ensure the line is read and are never changed. The setting of pins 16 and 17 to low turns off the status indicator LEDs, with one showing the C Stamp is waiting for a command and the other showing the C Stamp is waiting for confirmation. Following this, the input method is called, which receives data and responds accordingly.

```

void input(BYTE buffer[], int length)
{
    int validIn1 = 0;
    int validIn2 = 0;
    float baud = 38.4;
    BYTE selected = '8';
    BYTE modified[] = "mdf\r\n";
    BYTE waiton[] = "!";    //start filling
                            //buffer after '!'

    BYTE pins[8];
    pins[0]=39; pins[1]=40;
    pins[2]=41; pins[3]=42;
    pins[4]=43; pins[5]=44;
    pins[6]=45; pins[7]=46;

    do
    {
        // Insert Figure 11 here.
    } while (validIn1 == 0);
}

```

Figure 10 – Code Snippet showing first portion of input method of C Stamp program

The input method traverses a number of steps en route to its completion. First, it sets both loop control variables to 0, with the first loop checking for a serial command and the second checking for a confirmation request. Next, all the variables are specified: baud is the baud rate used, selected is the LED that is to be toggled, modified is the confirmation signal, waiton is the waitOn character used for serial input, and the pins correspond to the LEDs used. Once these variables are specified, the first loop is entered, checking for a command signal.

```
//Read input until it is of the correct format.
STPIND(16,HIGH);
SERIN2(0, 0, baud, 8, 0, 0, buffer, 3, 23, waiton);
STPIND(16,LOW);

//Input is formatted correctly
if (buffer[0] == buffer[2] && buffer[1] == '!')
{
    selected = buffer[0];
    //Input is in range
    if (inRange(selected))
    {
        //Send back input to verify it is
        //correct.
        SEROUT2(0, 0, baud, 0, 8, 0, 0, buffer,
            length, 22);

        do
        {
            // Insert Figure 12 here.
        } while (validIn2 == 0) ;
    }
}
}
```

Figure 11 – Code Snippet showing second portion of input method of C Stamp program

The second portion of the input method is responsible for accepting command signals from the Connect ME and interpreting them. First, it outputs to pin 16 to indicate its state, and then waits for serial input. Once it receives serial input, it turns pin 16 low and checks if the serial signal is well formatted, and then if it is in the range of LEDs that can be modified. If it satisfies both of these conditions, the method outputs a verification request and proceeds to wait for confirmation. Otherwise, the method returns to waiting for a command signal at its beginning.

```
//Read input until it is of the correct format.
STPIND(17,HIGH);
SERIN2(0, 0, baud, 8, 0, 0, buffer, 3, 23, waiton);
```

```

STPIND(17,LOW);

//Input is formatted correctly
if (buffer[0] == buffer[2] && buffer[1] == '!')
{
    //Input is confirmation signal.
    if (confirmed(buffer[0]))
    {
        //Exit the loop.
        validIn1 = 1;
        validIn2 = 1;
    }
//Input is in range; the previous input received
//was incorrect
else if (inRange(buffer[0]))
{
    //Inform applet C Stamp knows of the
    //discrepancy.
    selected = buffer[0];
    SEROUT2(0, 0, baud, 0, 8, 0, 0, buffer,
            length, 22);
}
}

```

Figure 12 – Code Snippet showing third portion of input method of C Stamp program

The third portion of the input method checks for a confirmation signal. The first step is to output a high signal to pin 17, lighting the status indicator LED. Following this, the C Stamp accepts serial input and outputs low to pin 17. If the input is formatted correctly and is a confirmation request, the C Stamp outputs a confirmation signal and it exits both loops. However, if the input is another command signal, the C Stamp outputs a new verification request and proceeds to wait for a confirmation request again.

```

//Toggle the selected LED.
TOGGLE(39+(selected&0x0F));
SEROUT2(0, 0, baud, 0, 8, 0, 0, modified,
        length, 22);
}

```

Figure 13 – Code Snippet showing final portion of input method of C Stamp program

The final portion of the input method is responsible for toggling the selected LED after the loops accepting input have been exited. It accomplishes this by first modifying the selected byte from its ASCII character to its real value, as for any digit 0-9, its ASCII representation is 0x30-0x39, so the 3 needs to be removed via a bitwise AND operation with the byte 0x0F. Since only one LED can be

modified at any time, this is accomplished by using the TOGGLE method of the C Stamp. The C Stamp then sends a confirmation signal and ends the input method, returning to the beginning via the loop in the main method.

```
int inRange(BYTE target)
{
    switch (target)
    {
        case '0': return 1;
        case '1': return 1;
        case '2': return 1;
        case '3': return 1;
        case '4': return 1;
        case '5': return 1;
        case '6': return 1;
        case '7': return 1;
    }
    return 0;
}
```

Figure 14 – Code Snippet showing inRange method of C Stamp program

The inRange method (Figure 14) is responsible for checking if a byte in a command signal is commanding the C Stamp to toggle one of its 8 LEDs. Although the applet will only modify 4 LEDs, this will not cause a problem as it would never send a confirmation signal if the C Stamp receives a command to toggle an LED between 4-7.

```
int confirmed(BYTE target)
{
    return (target == 'a');
}
```

Figure 15 – Code Snippet showing confirmed method of C Stamp program

The confirmed method (Figure 15) checks if a signal is a confirmation request, comparing the byte in question to 'a' with "ala" being the confirmation request signal. With this method, the C Stamp program is complete and capable of receiving command and confirmation request signals and interpreting them properly.

The Messenger demo applet provided a basis for serial communication between the Connect ME and the C Stamp which was modified to create the Lights applet. In creating the Lights applet, several files were only modified by replacing references to Messenger objects to those of Lights objects, and these include LightsApplet.java, LightsConfig.java, and LightsDisplay.java. The classes

LightsPanel and Lights were modified accordingly, and a new class, Wait, was added to the program, allowing threads to pause.

The MessengerPanel class needed to be modified slightly in order to create the LightsPanel class, which is responsible for creating the GUI of the Lights system. The main modification was the removal of the message entry text box and the Send button, replacing them with the four buttons to toggle LEDs. Different ActionListeners also needed to be implemented as a result of this change.

```
private JPanel lightsPanel = null;
private JLabel header = null;
private JTextArea messageArea = null;
private JButton[] LEDButtons = {null, null,
                                null, null};
```

Figure 16 – Code Snippet showing declaring of objects used in the LightsPanel class

The objects declared to be used in the LightsPanel class are the messageArea and the LEDButtons (Figure 16). The message area serves the same purpose as it did in the Messenger class, displaying messages sent to and received from the Digi Connect ME, so that the user is aware of what the applet is doing at any given time. The array of LEDButtons stores the buttons which toggle each of the 4 LEDs the applet is programmed to modify.

```
gbc.weightx = 25;
gbc.weighty = 25;
for (int x = 0; x < 4; x++)
{
    final int target = x;
    LEDButtons[x] = new JButton("LED" + (x+1));
    LEDButtons[x].addActionListener(
        new ActionListener() {
            public void
actionPerformed(ActionEvent e) {
                lights.handshake(target);
            }
        });
    addToGrid(lightsPanel, LEDButtons[x], gbc, x %
4, 7, 1, 1);
}
```

Figure 17 – Code Snippet showing initialization of LEDButtons

The only other modification made to the LightsPanel class was the removal of the instructions adding the Message entry window and the send button, and the addition of instructions to add the LEDButtons to the panel (Figure 17). First, it modifies the GridBagConstraints of the LightsPanel, setting each button to a

weight of 25 in both the x and y axes, versus 100 in both axes for the messageArea. Then, for each button, the method adds it to the panel with a different x coordinate for each LED and the same y coordinate. Following this, the method adds an ActionListener for each button which will initiate the handshake method for the LED it's linked to.

The Messenger Class needed to be modified heavily in order to create the Lights class, which houses the majority of the functionality of the Lights system. The portions of the class that needed to be modified were the local variables, of which some were added, and the receive method, as it gained new functionality. A few methods were added, specifically the handshake method which controls all handshaking functionality, the checkMsg method which determines if a received signal is what is expected, and the isFormatted method, which determines if a received signal is formatted properly. These modifications added a wealth of functionality to the Lights class so it could handle proper handshaking, performing operations based on signals sent and received instead of only sending and receiving them without consequence.

```

    private static final String SEND_PREFIX =
"send: ";
    private static final String RECEIVE_PREFIX =
"recv: ";
    private static final String INFO_PREFIX =
"info: ";
    private static final String ERROR_PREFIX =
"!error! ";
    private static final String CONFIRM_SIGNAL =
"!a!a!a";
    private static final String CONFIRM_RESPONSE =
"mdf";

    private String expectedResponse;
    private boolean correctResponse;
    private boolean confirmResponse;

    private boolean busy = false;
    private String host;
    private int port;
    private LightsDisplay msgDisplay = null;

```

Figure 18 – Code Snippet showing local variable declarations of the Lights class

The only changes in the declaration of local variable for the Lights class (Figure 18) were the additions of variables CONFIRM_SIGNAL, CONFIRM_RESPONSE, expectedResponse, correctResponse, confirmResponse, and busy. CONFIRM_SIGNAL is the signal the applet

instructs the Connect ME to output to the C Stamp after it was requested verification. CONFIRM_RESPONSE is the expected confirmation signal to be received from the C Stamp after sending the CONFIRM_SIGNAL. The String expectedResponse contains the value of the expected response to a command signal output by the Connect ME, and correctResponse and confirmResponse are both loop control variables. The Boolean busy is used as a form of mutual exclusion, as if the Applet tries to instruct the C Stamp to toggle two LEDs at the same time, a race condition could occur that may stall the system indefinitely. These changes in local variables allow for the implementation of the handshaking method, which is called every time one of the LEDButtons is selected in the LightsPanel.

```

public void handshake(int target)
{
    final String targetNum = Integer.toString(target);
    final String command = "!" + targetNum + "!" +
                           targetNum + "!" + targetNum;

    Thread shaker = new Thread()
    {
        public void run()
        {
            //Only perform one handshake at a time.
            if (busy == false)
            {
                busy = true;

                correctResponse = false;
                confirmResponse = true;
                expectedResponse = targetNum +
                                   "!" + targetNum;

                //Continuously send command until
                //the C Stamp has responded
                //properly.
                while (correctResponse == false)
                {
                    send(command);
                    Wait.manyMilSec(100);
                    //Continuously send
                    //confirmation until the C
                    //Stamp responds.
                    while (confirmResponse ==
                           false)
                    {

```

```

        send(CONFIRM_SIGNAL);
        Wait.manyMilSec(100);
    }
}
    busy = false;
}
};
shaker.start();
}

```

Figure 19 – Code Snippet showing handshake method of the Lights class

The handshake method is responsible for sending all data to the Digi Connect ME and therefore the C Stamp, as it calls the send method based on its state. The state of the Applet is modified in the checkMsg function which is called by the receive function. The first step of the handshake method is to convert the target integer to a final integer so that it can be accessed in a thread, and to generate the final String containing the command message to be sent over the network to the C Stamp. The method then creates a thread to perform the basis of the handshaking function, so that the other threads, specifically the ones controlling the send and receive functions, are allowed to continue operation.

The first step of the thread is to check the busy variable, supporting the mutual exclusion necessary to avoid race conditions. If another thread is performing a handshake, this thread performs no other process and terminates. Otherwise, the thread sets the busy variable to true and continues its operation. The loop control variable correctResponse is set to false so that the thread continually transmits the command signal, and the confirmResponse variable is set to true so that it does not transmit the confirmation signal. The transmission of the command is performed once every 100 mS, allowing the C Stamp time to process it and respond in most cases without being so slow as to cause an issue. When the verification request is received and it matches the expected response, the checkMsg method will modify the loop control variables so that the handshake method will start sending confirmation requests rather than command signals. This will continue at the same rate until the checkMsg method receives either an invalid verification request or a confirmation signal. In the event of an invalid verification request, the method returns to outputting the command signal, and in the event of a confirmation signal, the method releases the busy variable and completes its execution, allowing another button press to start a new thread.

```

private void receive() throws Exception
{
    displayMessage(INFO_PREFIX + "Connecting to " +
        host+":"+port+"...");
}

```

```

socket = new Socket(host, port);
displayMessage(INFO_PREFIX + "Connected.");

try
{
    BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    while (true)
    {
        String msg = in.readLine();
        if (msg != null && msg.length() > 0)
        {
            displayMessage(RECEIVE_PREFIX+msg);
            checkMsg(msg);
        }
    }
} catch (Exception ex) {
    displayMessage(INFO_PREFIX +
        "communications ended. (" +ex+"");
}
}

```

Figure 20 – Code Snippet showing receive method of the Lights class

The receive method was modified very slightly, so that in addition to displaying received messages, it also called the checkMsg method (Figure 20). This function call initiates changes in the state of the applet, allowing it to modify its output once input is received.

```

private void checkMsg(String msg)
{
    //Applet has not received correct response
    //from C Stamp yet
    //verifying signal was sent properly.
    if (!correctResponse && confirmResponse)
    {
        //If response is what is expected, then it is
        //correct and
        //the handshake method can continue.
        if (msg.equals(expectedResponse))
            confirmResponse = false;
    }
}
//Applet has not received confirmation response
//from C Stamp
//yet verifying confirmation signal was sent

```

```

//properly.

    else if (!confirmResponse)
    {
        //If response is confirmation response,
        //handshake can
        //continue and terminate
        if (msg.equals(CONFIRM_RESPONSE))
        {
            confirmResponse = true;
            correctResponse = true;
        }
        //If message is well formatted and not a
        //confirmation
        //response, it is likely the confirmation
        //signal was not
        //received properly, so the process needs to
        //start over.
        else if (isFormatted(msg) &&
                !msg.equals(expectedResponse))
            confirmResponse = true;
    }
}

```

Figure 21 – Code Snippet showing checkMsg method of the Lights class

The checkMsg is responsible for modifying the state of the applet based on the data received from the C Stamp, and is only called by the receive method. If the applet is in the state in which it continuously transmits the command signal, then the method checks if the message received is the appropriate verification request. If it is, the method changes the state of the applet so that it outputs the confirmation request. If the applet is in the state in which it transmits the confirmation signal continuously, it first checks if the message received is the confirmation response. If it is, the loop control variables are modified so that the handshake thread can terminate. If the message is the expected response, it is likely there was a delay and the C Stamp sent that response before receiving a confirmation signal, the applet does nothing. If the method is an invalid verification request, the applet returns to the state in which it outputs the command signal. The checkMsg method allows for smooth transitions between states based on input from the C Stamp.

```

private boolean isFormatted(String msg)
{
    if (msg.charAt(0) == msg.charAt(2) &&
        msg.charAt(1) == '!')
        return true;
}

```

```

    return false;
}

```

Figure 22 – Code Snippet showing isFormatted method of the Lights class

The isFormatted method is used to check verification request input signals to make sure they are well formatted, specifically to determine if the C Stamp received a command signal improperly. If the C Stamp sends a properly formatted signal that is not equal to the expected response, then it must be sent another command signal.

```

public class Wait
{
    public static void oneMilSec()
    {
        try
        {
            Thread.currentThread();
            Thread.sleep(1);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    public static void oneSec()
    {
        try
        {
            Thread.currentThread();
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    public static void manyMilSec(long s)
    {
        try
        {
            Thread.currentThread();
            Thread.sleep(s);
        }
    }
}

```

```

        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    public static void manySec(long s)
    {
        try
        {
            Thread.currentThread();
            Thread.sleep(s * 1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

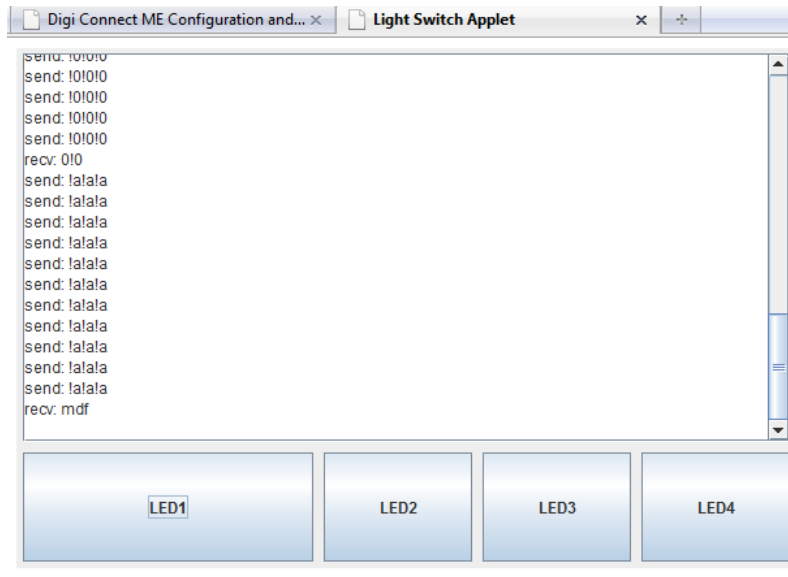
```

Figure 23 – Code Snippet showing Wait class and its methods

The Wait class (Figure 23) adds functionality to pause threads so as to not overload the C Stamp with the applet's output. It functions as a static class, as all of its methods are static and when called they affect the thread calling them. It contains four methods, oneMilSec, oneSec, manyMilSec, and manySec. The oneSec and oneMilSec methods have no input, and are responsible for delaying the calling thread for either a second or a millisecond. The manySec and manyMilSec functions accept a long input, and delay the thread for the number of seconds or milliseconds specified.

Verification of Functionality

Through the testing of the Embedded Web Server, proper demonstration of its functionality and error handling is shown in Figures 24-31, showing both the output shown to the user via the Lights Applet GUI and the output at the C Stamp as a result of the signals sent and received.



Light Switch 1.3

Figure 24 - Light Switch Applet instructing C Stamp to Toggle LED 1

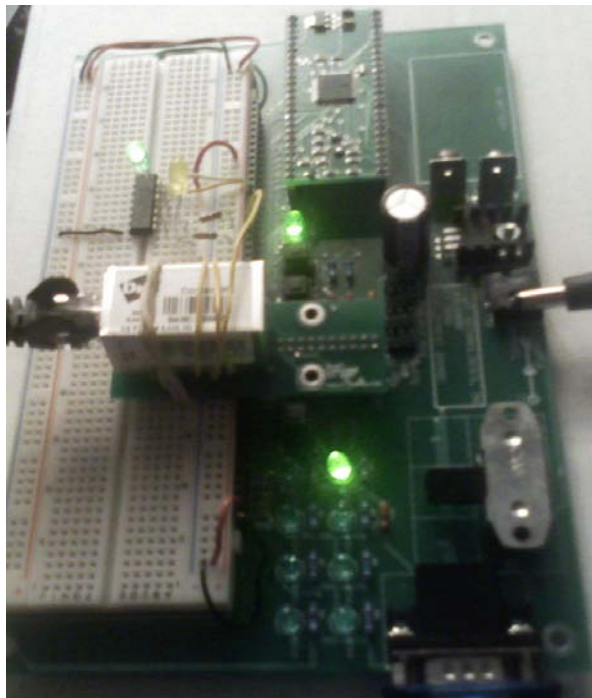
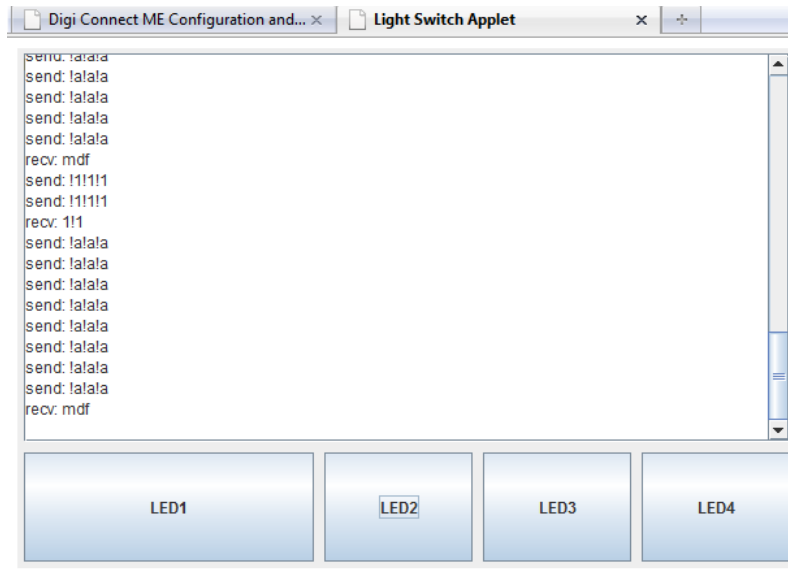


Figure 25 - C Stamp after toggling LED 1



Light Switch 1.3

Figure 26 - Light Switch Applet instructing C Stamp to Toggle LED 2

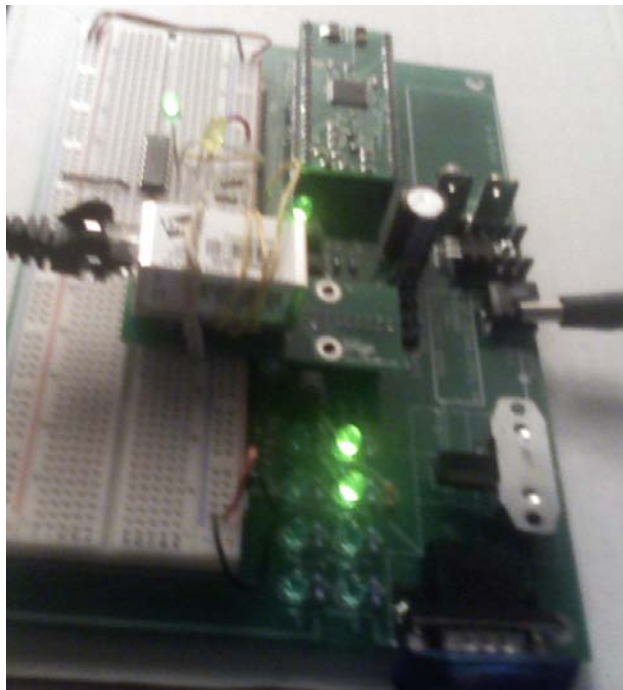
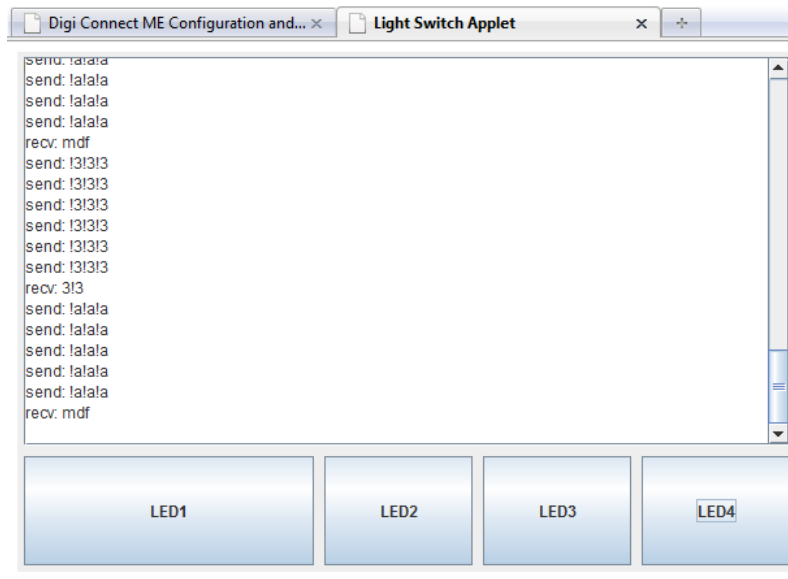


Figure 27 - C Stamp after toggling LED 2



Light Switch 1.3

Figure 28 - Light Switch Applet instructing C Stamp to Toggle LED 4

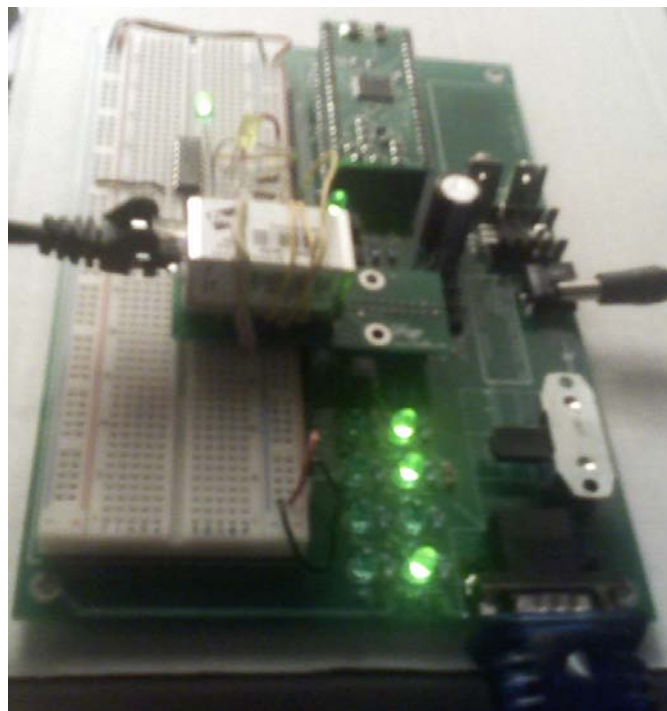
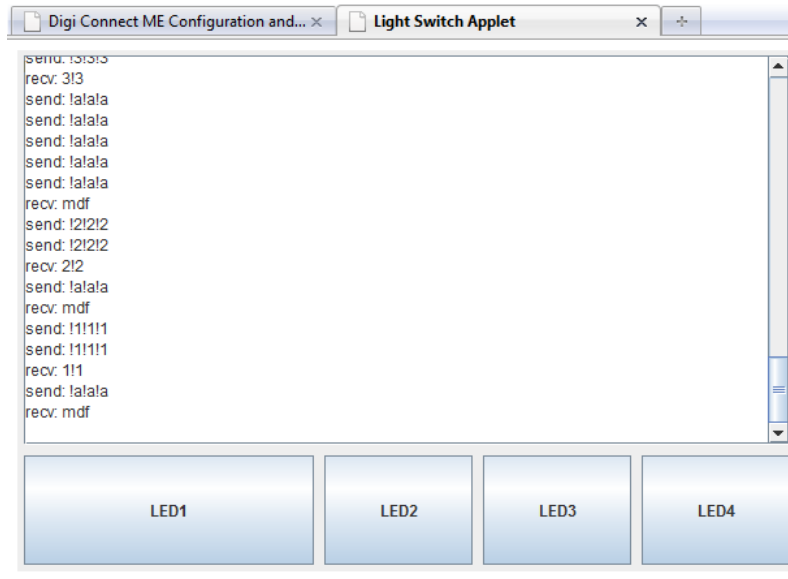


Figure 29 - C Stamp after toggling LED 4



Light Switch 1.3

Figure 30 - Light Switch Applet instructing C Stamp to Toggle LEDs 2 and 3

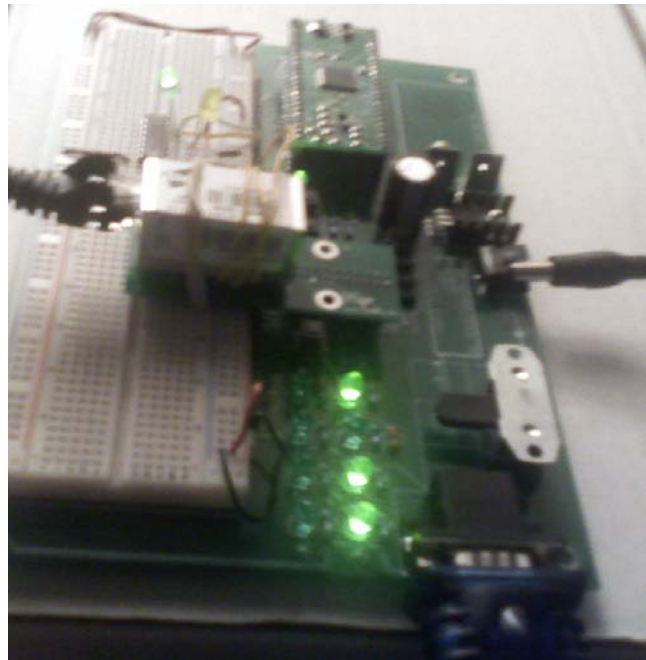


Figure 31 - C Stamp after toggling LEDs 2 and 3

The C Stamp performed as expected, modifying the specified LED after completing the handshake protocol.

Conclusion

The Implementation of an Embedded Web Server serves as a valuable milestone towards developing projects of your own. It is a true multi-step process, with numerous milestones that need to be reached before the project could advance further.

Terms & Conditions

Quality Assurance

A-WIT has stringent quality control procedures in place to insure the best quality products.

90-Day Limited Warranty

A-WIT Technologies, Inc warrants its products against defects in materials and workmanship for a period of 90 days. If you discover a defect, A-WIT Technologies, Inc. will, at its option, repair, replace, or refund the purchase price. After 90 days, products can still be sent in for repair or replacement, but there will be a \$10.00USD minimum inspection/labor/repair fee (not including return shipping and handling charges).

14-Day Money-Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a refund. A-WIT will refund the purchase price of the product in the form of a check, excluding shipping/handling costs, once the product is received. This refund does not apply if the product has been altered or damaged. If you decide to return the products after the 14-day evaluation period, a 20% restocking fee will be charged against a credit.

Disclaimer

Warranty does not apply if the product has been altered, modified, or damaged. A-WIT makes no other warranty of any kind, expressed or implied, including any warranty of merchantability, fitness of the product for any particular purpose even if that purpose is known to A-WIT, or any warranty relating to patents, trademarks, copyrights or other intellectual property. A-WIT shall not be liable for any injury, loss, damage, or loss of profits resulting from the handling or use of the product shipped.

How to Return a Product

When returning, you must first e-mail sales@a-wit.com for a Return Merchandise Authorization number. No packages will be accepted without the RMA number clearly marked on the outside of the package. After inspecting and testing, we will return your product, or its replacement using the same shipping method used to ship the product to A-WIT within 30 days. In your package, please include a daytime telephone number and a brief explanation of the problem.

Please contact our Sales Department at sales@a-wit.com if you have any questions regarding our warranty policy or if you are requesting an RMA number.

