Volume

2

A-WIT TECHNOLOGIES INC.

… a passion for execution …

CS310XXX (μC 101) Reference Guide Manual

Version 1.4

A-WIT TECHNOLOGIES INC.

CS310XXX (µC 101) Reference Manual

Phone (800) 985-AWIT • Fax (800) 985-2948

Table of Contents

Chapter

1

Microcomputers 101 (µC 101)

This text covers the basics of microcomputers or microcontrollers by showing the audience how they can create several of their own microcomputer-based embedded systems using the A-WIT Technologies, Inc.'s C Stamp™ microcomputer module. The designs described in this text expose users to the fundamentals of microcomputer/microcontroller based embedded systems by using light, tactile feedback, sound, and motion. The activities in the text are projects that introduce a variety of concepts in embedded systems development and programming, electricity and electronics, mathematics, and physics. All activities are highly illustrated hands-on presentation of design practices used by engineers in the design of modern equipments.

Audience

This text is organized so that it can be used by pre-engineering students as well as independent learners. Comprehension and problem-solving skills can be tested with the questions, exercises, projects, and solutions provided at the end of each chapter.

Reference Material

The "C Stamp Syntax and Reference Guide Manual" is an essential reference for all C Stamp related projects and activities. It contains information on the C Stamp microcomputer module, software development tools, programming language, kits, Boards of Learning (BOLs), and accessories.

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@a-

wit.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.

Introduction to the C Stamp

Each C Stamp comes with a microcontroller chip that contains the C Stamp Operating System, internal memory (RAM, EEPROM, and Flash), a 5-volt regulator, a number of general-purpose I/O pins (TTL-level and Schmitt Trigger inputs, and 0-5 Volts outputs), communication and other peripherals, analog functions, and a set of library function commands for math, pin operations, and much more. C Stamp modules are capable of running many thousand instructions per second and are programmed with a subset of the C programming language, called WC. WC is a simple, easy to learn language, and it is highly optimized for embedded system. It includes many specialized functions. This manual includes an extensive section devoted to each of the available functions. The table below provides the specifications of the CS110000 C Stamp module.

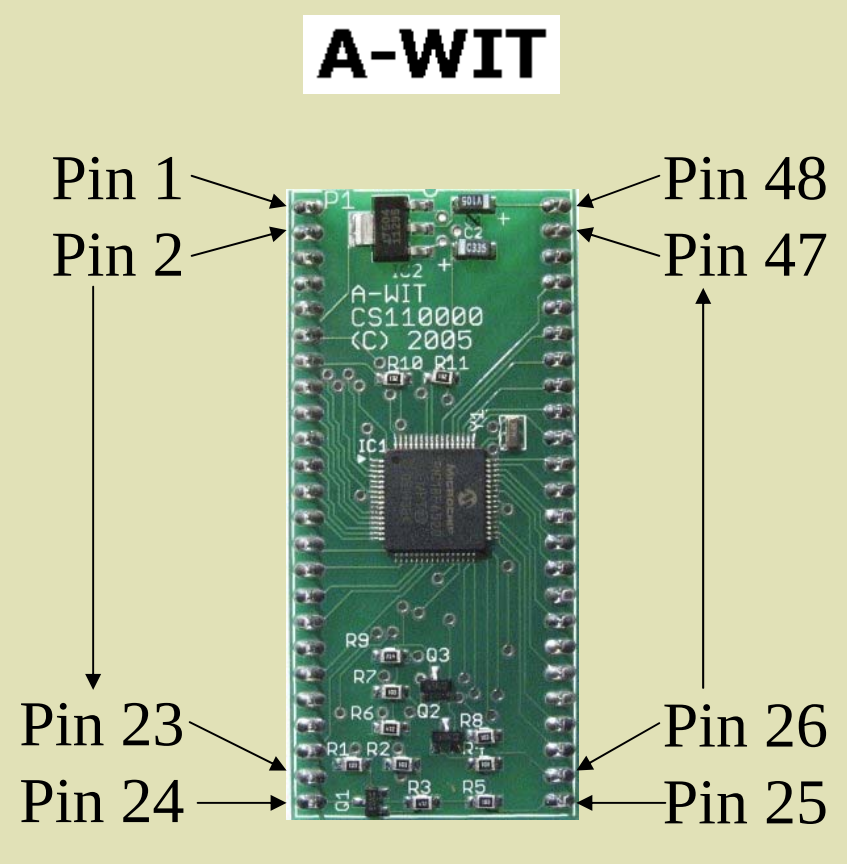
Features/Attributes	
PACKAGE	48-pin DIP
PACKAGE SIZE (L x W x H)	2.4" x 1.1" x 0.4"
PINS ATTACHMENT METHODOLOGY	Through Hole Strong Will not fall off
ENVIRONMENT	-40 to 85 deg. C (-40 to 185 deg. F)
MICROCONTROLLER	MICROCHIP PIC18F6520
PROCESSOR SPEED	40 MHz
PROGRAM EXECUTION SPEED	~10,000,000 instructions/sec.
RAM SIZE	2K Bytes
SCRATCH PAD RAM	2K Bytes
PROGRAM MEMORY SIZE	32K Bytes, ~16,000 inst.
NUMBER OF I/O PINS	41 + 2 Dedicated Serial
VOLTAGE REQUIREMENTS	5 - 24 V DC

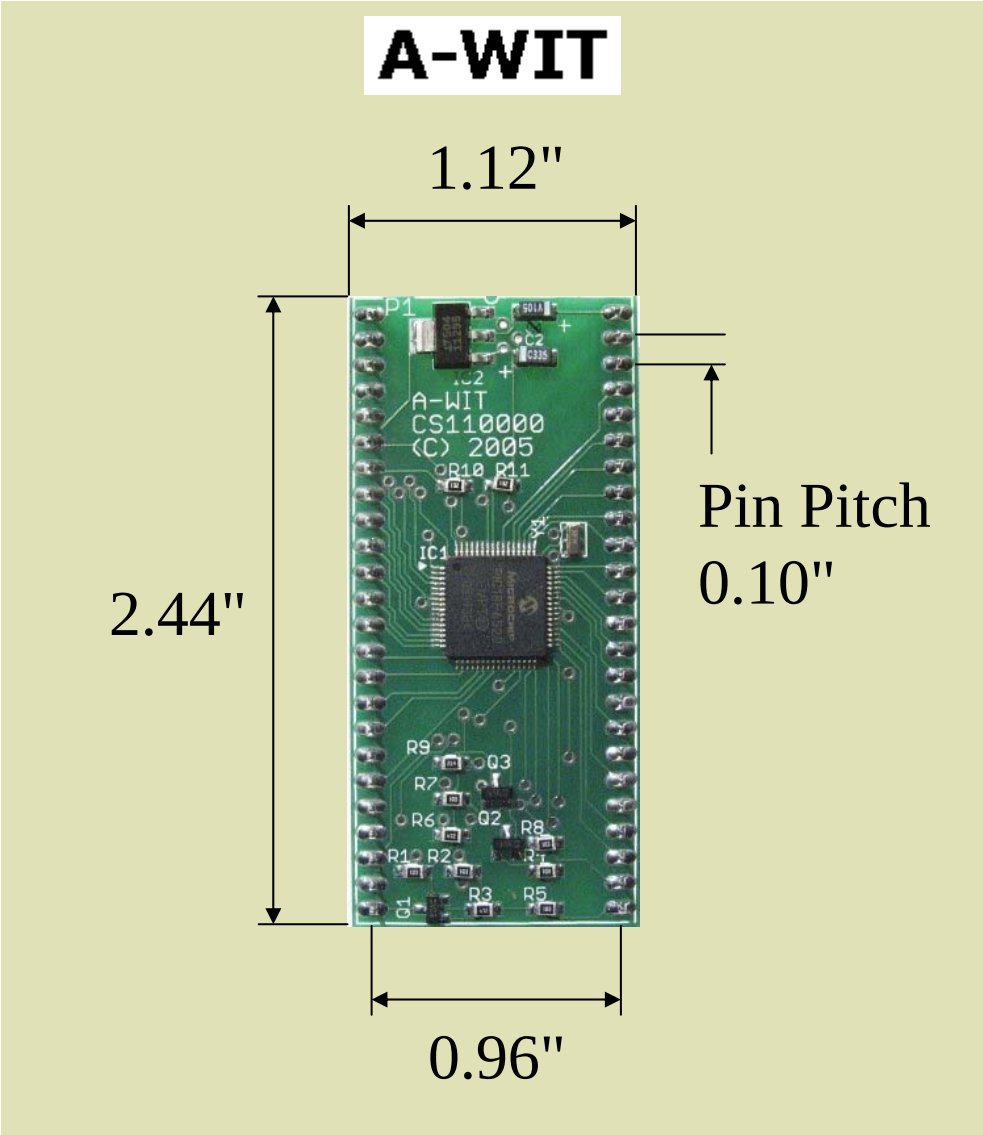
Features/Attributes	
CURRENT DRAW @ 5V	19 mA Run / 0.7 uA Sleep
SOURCE/SINK CURRENT PER I/O	25 mA / 25 mA
SOURCE/SINK CURRENT PER MODULE	100 mA / 100 mA per 4 I/O pins
PC PROGRAMMING INTERFACE	Serial (57600 baud)
C STAMP™ INTEGRATED PROGRAMMING ENVIRONMENT	MPLAB IDE (v7.22 and up)
EEPROM (DATA) SIZE	1K Byte
INTERRUPTS	8
DIGITAL TO ANALOG CONVERTERS	2 channels (10 bits)
OTHER COMMUNICATION INTERFACES	3-wire SPI™, I2C Master and Slave
PARALLEL SLAVE PORT	8 bits
ANALOG TO DIGITAL CONVERSION	12 channels (10 bits)
ANALOG COMPARATORS	2

All C Stamp models come in Industrial-rated versions, with an environmental temperature tolerance range of -40°C to +85°C.

NOTE: UNDER NO CIRCUNSTANCES, THE TOTAL CURRENT SUPPLIED BY ALL THE PINS OF THE CS110000 C STAMP MODULE SHOULD BE GREATER THAN 200 mA.

The figures below shows how the pins of the CS110000 C Stamp module are numbered and the dimensions associated with the C Stamp; and the table after that provides the CS110000 C Stamp module Pin Descriptions.





Pin #	*Name*	*Type*	*Buffer*	*Description*
1	RE1/WRn RE1 WRn	 I/O I	 ST TTL	 Digital I/O Port E, bit 1 Write control for PSP
2	RE0/RDn RE0 RDn	 I/O I	 ST TTL	 Digital I/O Port E, bit 0 Read control for PSP
3	RG0/CCP3 RG0 CCP3	 I/O I/O	 ST ST	 Digital I/O Port G, bit 0 Capture3 input /

Pin #	*Name*	*Type*	*Buffer*	*Description*
				Compare3 output / PWM3 output
4	RG1	I/O	ST	Digital I/O Port G, bit 1
5	MCLRn	I	ST	Master Clear (Reset) input
6	VSS	P		Ground reference
7	VDD	P		Positive supply
8	RF7/SSn			
	RF7	I/O	ST	Digital I/O Port F, bit 7
	SSn	I	TTL	SPI slave select input
9	RF6/AN11			
	RF6	I/O	ST	Digital I/O Port F, bit 6
	AN11	I	Analog	Analog input 11
	C1INM	I	Analog	Comparator 1 input -
10	RF5/AN10/OVREF			
	RF5	I/O	ST	Digital I/O Port F, bit 5
	AN10	I	Analog	Analog input 10
	C1INP	I	Analog	Comparator 1 input +
	OVREF	O	Analog	Voltage reference output
11	RF4/AN9			
	RF4	I/O	ST	Digital I/O Port F, bit 4
	AN9	I	Analog	Analog input 9
	C2INM	I	Analog	Comparator 2 input -
12	RF3/AN8			
	RF3	I/O	ST	Digital I/O Port F, bit 3
	AN8	I	Analog	Analog input 8
	C2INP	I	Analog	Comparator 2 input +
13	RF2/AN7/C1OUT			
	RF2	I/O	ST	Digital I/O Port F, bit 2
	AN7	I	Analog	Analog input 7
	C1OUT	O		Comparator 1 output
14	RF1/AN6/C2OUT			
	RF1	I/O	ST	Digital I/O Port F, bit 1
	AN6	I	Analog	Analog input 6
	C2OUT	O		Comparator 2 output
15	RF0/AN5			
	RF0	I/O	ST	Digital I/O Port F, bit 0
	AN5	I	Analog	Analog input 5
16	RA3/AN3/VREFP			
	RA3	I/O	TTL	Digital I/O Port A, bit 3
	AN3	I	Analog	Analog input 3
	VREFP	I	Analog	A/D reference voltage (High) input
17	RA2/AN2/VREFM			

Pin #	*Name*	*Type*	*Buffer*	*Description*
	RA2 AN2 VREFM	I/O I I	TTL Analog Analog	Digital I/O Port A, bit 2 Analog input 2 A/D reference voltage (Low) input
18	RA1/AN1 RA1 AN1	 I/O I	 TTL Analog	 Digital I/O Port A, bit 1 Analog input 1
19	RA0/AN0 RA0 AN0	 I/O I	 TTL Analog	 Digital I/O Port A, bit 0 Analog input 0
20	RA5/AN4 RA5 AN4	 I/O I	 TTL Analog	 Digital I/O Port A, bit 5 Analog input 4
21	RA4	I/O	ST/OD	Digital I/O Port A, bit 4
22	RC1	I/O	ST	Digital I/O Port C, bit 1
23	RC0	I/O	ST	Digital I/O Port C, bit 0
24	TX1	O		Dedicated USART asynchronous transmit
25	RX1	I	ST	Dedicated USART asynchronous receive
26	ATN	I		Serial (USART asynchronous) DTR
27	RC2/CCP1 RC2 CCP1	 I/O I/O	 ST ST	 Digital I/O Port C, bit 2 Capture1 input / Compare1 output / PWM1 output
28	RC3/SCK/SCL RC3 SCK SCL	 I/O I/O I/O	 ST ST ST	 Digital I/O Port C, bit 3 Synchronous serial clock input/output for SPI mode Synchronous serial clock input/output for I2C mode
29	RC4/SDI/SDA RC4 SDI SDA	 I/O I I/O	 ST ST ST	 Digital I/O Port C, bit 4 SPI data in I2C data I/O
30	RC5/SDO RC5 SDO	 I/O O	 ST	 Digital I/O Port C, bit 5 SPI data out
31	RB7/KBI3 RB7 KBI3	 I/O I	 TTL TTL	 Digital I/O Port B, bit 7 Interrupt-on-change pin 3

Pin #	*Name*	*Type*	*Buffer*	*Description*
32	RB6/KBI2 RB6 KBI2	 I/O I	 TTL TTL	 Digital I/O Port B, bit 6 Interrupt-on-change pin 2
33	RB5/KBI1 RB5 KBI1	 I/O I	 TTL TTL	 Digital I/O Port B, bit 5 Interrupt-on-change pin 1
34	RB4/KBI0 RB4 KBI0	 I/O I	 TTL TTL	 Digital I/O Port B, bit 4 Interrupt-on-change pin 0
35	RB3/INT3 RB3 INT3	 I/O I	 TTL ST	 Digital I/O Port B, bit 3 External Interrupt 3
36	RB2/INT2 RB2 INT2	 I/O I	 TTL ST	 Digital I/O Port B, bit 2 External Interrupt 2
37	RB1/INT1 RB1 INT1	 I/O I	 TTL ST	 Digital I/O Port B, bit 1 External Interrupt 1
38	RB0/INT0 RB0 INT0	 I/O I	 TTL ST	 Digital I/O Port B, bit 0 External Interrupt 0
39	RD7/PSP7 RD7 PSP7	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 7 Parallel Slave Port Data, bit 7
40	RD6/PSP6 RD6 PSP6	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 6 Parallel Slave Port Data, bit 6
41	RD5/PSP5 RD5 PSP5	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 5 Parallel Slave Port Data, bit 5
42	RD4/PSP4 RD4 PSP4	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 4 Parallel Slave Port Data, bit 4
43	RD3/PSP3 RD3 PSP3	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 3 Parallel Slave Port Data, bit 3
44	RD2/PSP2 RD2	 I/O	 ST	 Digital I/O Port D, bit 2

Pin #	*Name*	*Type*	*Buffer*	*Description*
	PSP2	I/O	TTL	Parallel Slave Port Data, bit 2
45	RD1/PSP1 RD1 PSP1	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 1 Parallel Slave Port Data, bit 1
46	RD0/PSP0 RD0 PSP0	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 0 Parallel Slave Port Data, bit 0
47	RE2/CSn RE2 CSn	 I/O I	 ST TTL	 Digital I/O Port E, bit 2 Chip select control for PSP
48	VIN	P		Input voltage into module

Legend: TTL = TTL compatible input

ST = Schmitt Trigger input with CMOS levels

Analog = Analog input

I = Input

O = Output

P = Power

OD = Open-Drain

For the Schmitt Trigger inputs, the hysteresis levels are 1 V and 4 V, except for pins 28 and 29. For these two pins, the hysteresis levels are 1.5 V and 3.5 V.

Pin 3 is CCP3 and Pin 27 is CCP1. There is no CCP2.

Pin 21 is an Open Drain output, so it needs an external pull-up resistor to VDD. A 10 KΩ will do.

Pin 26 (ATN) is not available to the user. It is there for future enhancements of the C Stamp programmer software.

Pin 7 (VDD) is a 5-volt DC input/output. Unregulated voltage applied to the VIN pin (Pin 48) will output 5 volts on VDD. Regulated voltage between 4.2V and 5.5V should be applied to VDD if no voltage is applied to VIN.

Introduction to the CS31X000 (µC 101) Microcontroller Fundamentals Board of Learning (BOL)

The Microcontroller Fundamentals Board of Learning (BOL) is a complete, high performance, low cost development platform designed for those interested in learning and using A-WIT's C Stamp module. Its size, rich feature set, and low price make it an ideal tool for the student and educator. The µC101-BOL is a great tool with which to get started with A-WIT's C Stamp modules. For educators, the µC101-BOL provides a clean and efficient platform to teach students the basics of microcontrollers, where they can easily plug in parts, and perform A-WIT provided projects or educator developed C Stamp based curriculum. This manual includes an extensive section devoted to projects. The list below provides the specifications of the CS31X000 BOL.

- 2.0 mm center-positive plug that accepts different outer diameters for 5 to 24 V DC power supplies and 9 V battery connections; electronically interlocked to prevent dual connections, while both the power supply and battery can be present on the board at the same time. With the BOL having the breadboard toward you, if the switch is to the left, then the battery will be supplying the power. If the switch is to the right, then the external power supply will be supplying power.

- Independent 1 A high current power (VHC and VDDHC, which are the same connection) generated on board for servos or other components that require more current than what the C Stamp can provide in large projects.

- No jumpers required for configuration.

- Serial (DB-9) connector for C Stamp IC programming and serial communications during runtime.

- C Stamp I/O pins, VIN, VHC, and GND connections brought adjacent to spacious full size 6.50" x 2.14" breadboard area.

- Female 10-pin dual row connector for additional connections.

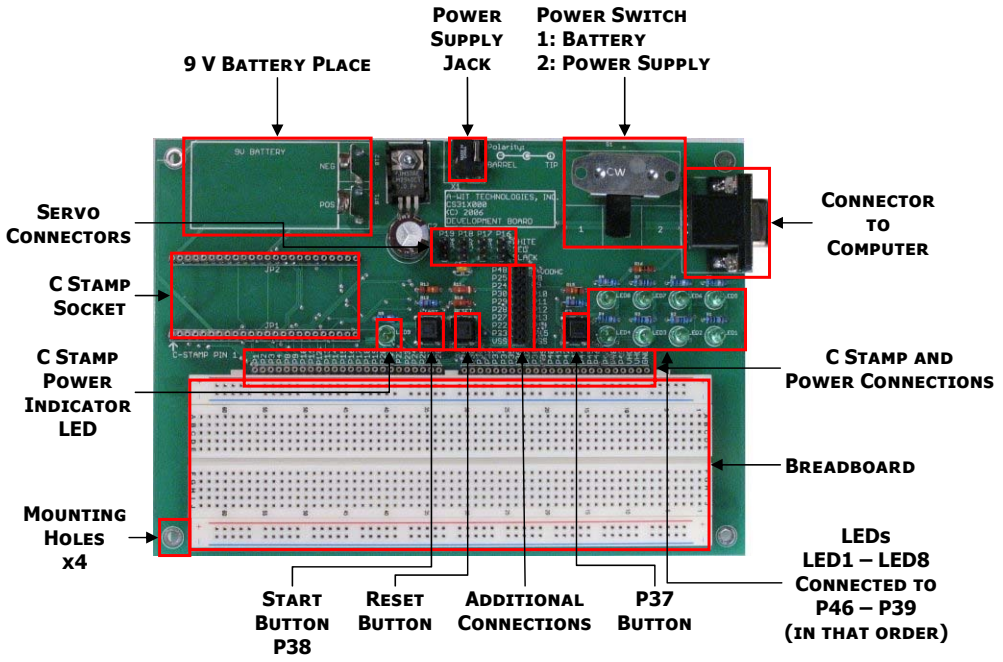
- Four servo connectors pre-connected to VHC, GND, and C Stamp I/O pins for control.

- RESET button with high tactile feedback.

- START button with high tactile feedback. START button can be used as a utility button during run time.

- Additional utility button with high tactile feedback.
- Eight utility green, high intensity, general purpose Light Emitting Diodes (LEDs).
- C Stamp power indicator LED.
- Mechanical Dimensions:
    - PCB: 7.50" x 5.28"
    - MOUNTING HOLES: 7.00" x 4.78"

The figure below shows the CS310X00 BOL with all features highlighted.



The START Button is pre-connected to Pin 38. When the Button is pushed, Pin 38 is pulled LOW; otherwise, Pin 38 is HIGH. For the Utility Button pre-connected to Pin 37, when the Button is pushed, Pin 37 is pulled LOW; otherwise, Pin 37 is HIGH. Each one of the LED1 - LED8 LED group is pre-connected to Pins 46 - 39 respectively. When a pin is set HIGH, the corresponding LED lights up, and when the pin is et LOW, the corresponding LED turns off. If the user wants to use any of the pre-connected pins for a different purpose, any of the pre-connected components can be ignored.

The C Stamp should be inserted in its corresponding socket with the pins 1 of the C Stamp and the socket aligned. Pin 1 is at the bottom-left corner of the socket in the figure above. Since the C Stamp has so many pins, enough pressure needs to be applied to the C Stamp to push it into the socket until it snaps into place. Press at several points of the C Stamp, but be gentle and careful at the same time. If you ever need to remove the C Stamp from the socket, first make sure that the C Stamp is powered off. Then, use a dull prying or lever like object to consecutively pry the C Stamp incrementally from both sides (left and right in the figure above) until the C Stamp is released from the socket. Since the C Stamp has so many pins, prying all the way from one side only will likely bend some pins on the opposite side from where the prying is being done.

Chapter

2

Getting Started

This chapter is a quick start guide to connecting the C Stamp to the PC and programming it. Without even knowing how the C Stamp functions, you should be able to complete the tutorial in this chapter and complete and run your first C Stamp program. The tutorial assumes you have a C Stamp and an appropriate connection kit or development board. You will also need a programming cable, power supply, PC running Windows® 2000/XP/Media/Vista, with a quantity of RAM recommended for the OS, sufficient free hard disk drive space for the software installations, CD-ROM drive, Internet access (recommended only), and available port compatible with your programming cable.

Notices

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

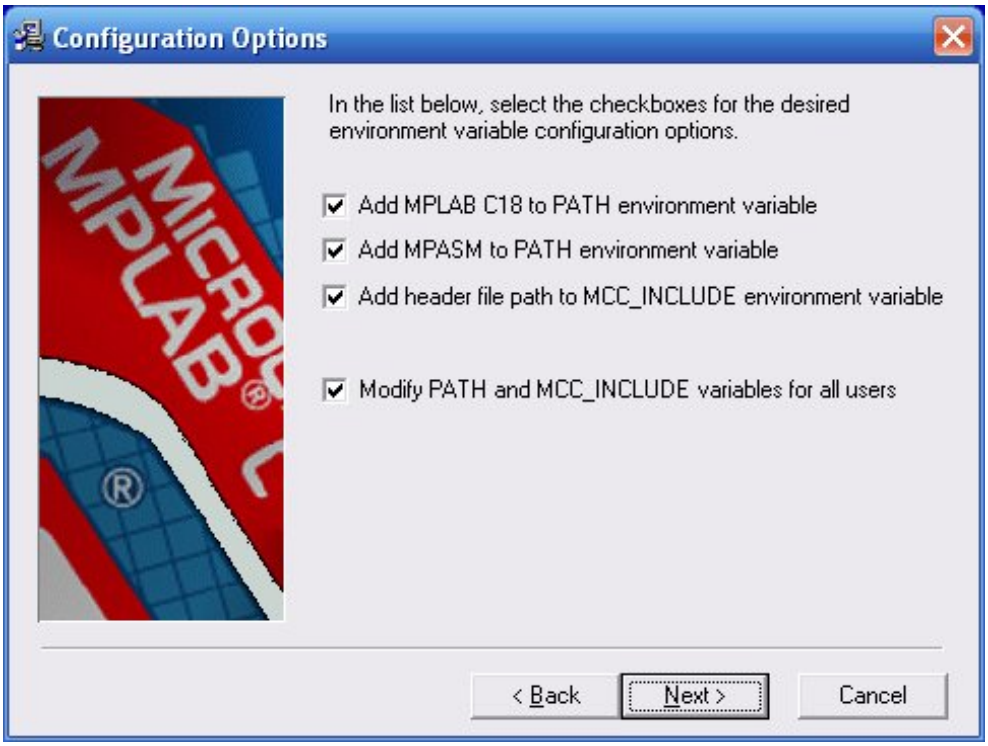
If possible, please check the C Stamp website www.c-stamp.com under SUPPORT for any updates to documentation, changes, or notices that may have become available since your Installation CD was produced. If you continue to have any issues for which a solution is not found in the aforementioned website,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

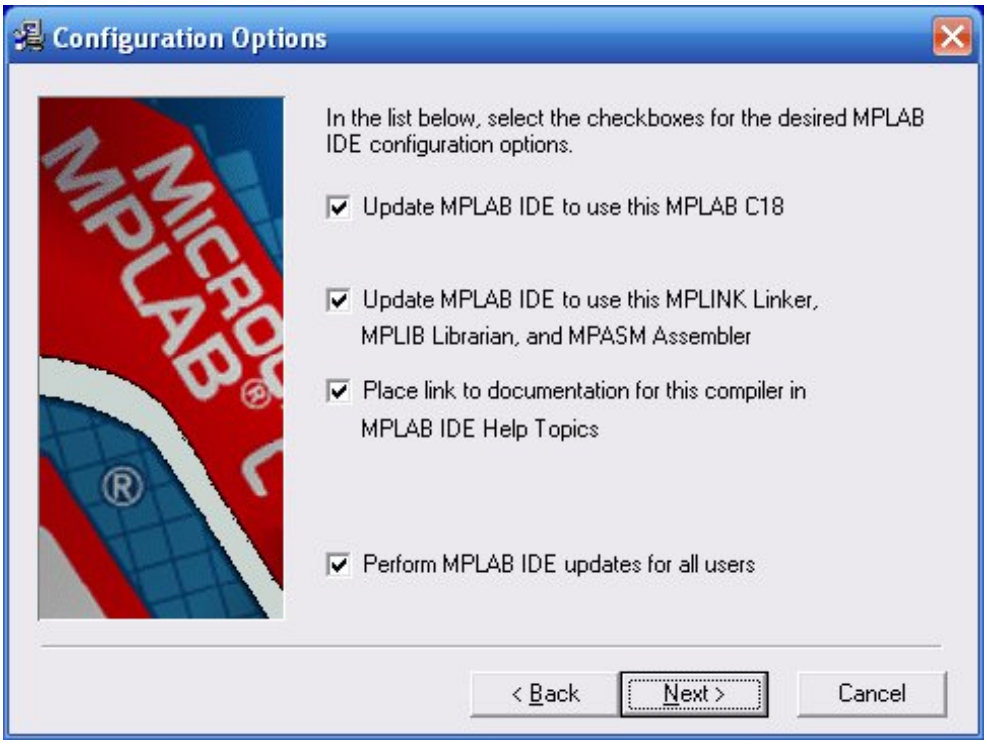
Installing the Microchip MPLAB and C Compiler Software

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.


Configuration Options
In the list below, select the checkboxes for the desired environment variable configuration options.
Add MPLAB C18 to PATH environment variable
Add MPASM to PATH environment variable
Add header file path to MCC_INCLUDE environment variable
Modify PATH and MCC_INCLUDE variables for all users
< Back
Next >
Cancel


Configuration Options
In the list below, select the checkboxes for the desired MPLAB IDE configuration options.
Update MPLAB IDE to use this MPLAB C18
Update MPLAB IDE to use this MPLINK Linker, MPLIB Librarian, and MPASM Assembler
Place link to documentation for this compiler in MPLAB IDE Help Topics
Perform MPLAB IDE updates for all users
< Back
Next >
Cancel

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 detect 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.

Setting Up the C Stamp Software Templates

Make a directory in your C:\ drive named A-WIT, and copy the CSTAMP_Template directory from the C Stamp Installation CD to your C:\A-WIT directory.

Documentation

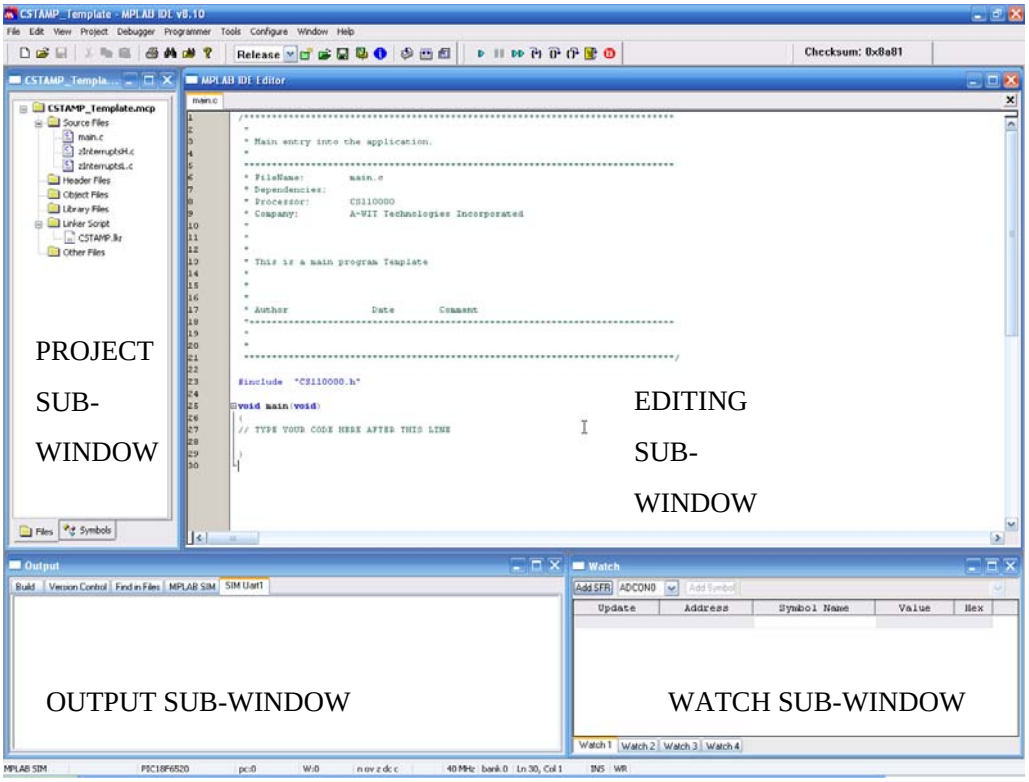
Copy the DOCS directory from the C Stamp Installation CD to your C:\A-WIT directory. This directory contains all the C Stamp related documentation in PDF format.

Creating your First C Stamp Program

Create a directory where you want to have all the files for your program; for example, FIRST_LED_APP. We recommend making this directory under your C:\A-WIT directory, so you can have all your CSTAMP related files in one place.

Copy the all files in your C Stamp Software Templates directory C:\A-WIT\CSTAMP_Template to the directory you just made.

Open the Microchip MPLAB IDE application. As shown the following figure, the IDE has several sub-windows. Depending on the resolution of your screen, your sub-windows may have a different layout. However; you can move and resize these into the position that you want to fit your screen, and your layout for that particular project will get saved upon answering yes to the prompt of saving the workspace when you exit the software development environment.



Go to the "File" menu to "Open Workspace…". Then navigate to your program directory, and open CSTAMP_Template.mcw.

Right click on CSTAMP_Template.mcp in the "Project" sub-window, and "Save as…" the name of your program project after you have navigated to your program

directory. For example, your program project could be named "FIRST_LED_APP". Now when you open the Microchip MPLAB IDE (Integrated Development Environment), and go to your program directory to open the workspace for your program, you will see a .mws file with the name of your program preceding it. This is the file that you should open any time you want to work on your program.

Double click on the main.c source and type the following code fragment where it is indicated. You can omit the comments for brevity, as they are written here to offer clarifications of what the code does. Do pay attention; however, to the indentation of the code blocks between curly brackets for loops, if statements, etc. Although indenting the code is not a requirement for the compiler to parse your code (i.e. any blank spaces are ignored by the compiler), it does help tremendously to make your code much more readable, and consequently, it makes finding any errors easier. Keywords and function names in the code fragment below are bolded.

After you START the C Stamp in user mode as explained in the "Downloading and Running Your Program" section (this will not be the RESET/BOOT/DOWNLOAD mode). This programs starts by lighting all 8 LEDs in your BOL, and waits for the utility button at pin 37 to be pushed and let go (cycled). When you do this, then the program blinks 4 of the 8 LEDs in your BOL. Any time you cycle the utility button, it switches to blinking the other four LEDs alternating the blinking of the LEDs between each bank of 4 LEDs. The program executes indefinitely until you restart it by pushing the RESET button while holding the START button and then letting go of the latter.

```
// Declare some necessary variables
  BIT button_pushed;
  BIT half;

// Initialize variable
  half = 0;       // Denotes which half (4 LEDs) of the
                  // 8 LEDs blinks

// Light LED's connected to the following pins
  STPIND(46, HIGH); STPIND(45, HIGH);
  STPIND(44, HIGH); STPIND(43, HIGH);
  STPIND(42, HIGH); STPIND(41, HIGH);
  STPIND(40, HIGH); STPIND(39, HIGH);

// Wait for button connected to pin 37 to be pushed
  button_pushed = FALSE;
  while(!button_pushed){
    button_pushed = BUTTON(37, LOW, HIGH, 5);
  }
```

```
  button_pushed = FALSE;

  while(1){
// Check if button is pushed every second
    button_pushed = BUTTON(37, LOW, HIGH, 5);
// If button was pushed, reset the variable that keeps
// track of that event, and switch the half of the LEDs
// that is lit
    if (button_pushed){
      button_pushed = FALSE;
      if (half == 0) half = 1;
      else half = 0;
    }
// Set one half of the LEDs to LOW, and toggle the
// other
    if (half == 0){
      STPIND(42, 0); STPIND(41, 0);
      STPIND(40, 0); STPIND(39, 0);
      TOGGLE(46); TOGGLE(45); TOGGLE(44); TOGGLE(43);
    }else{
// Set the other half of the LEDs to LOW, and toggle
// the other
      STPIND(46, 0); STPIND(45, 0);
      STPIND(44, 0); STPIND(43, 0);
      TOGGLE(42); TOGGLE(41); TOGGLE(40); TOGGLE(39);
    }
    PAUSE(250);
  }
```

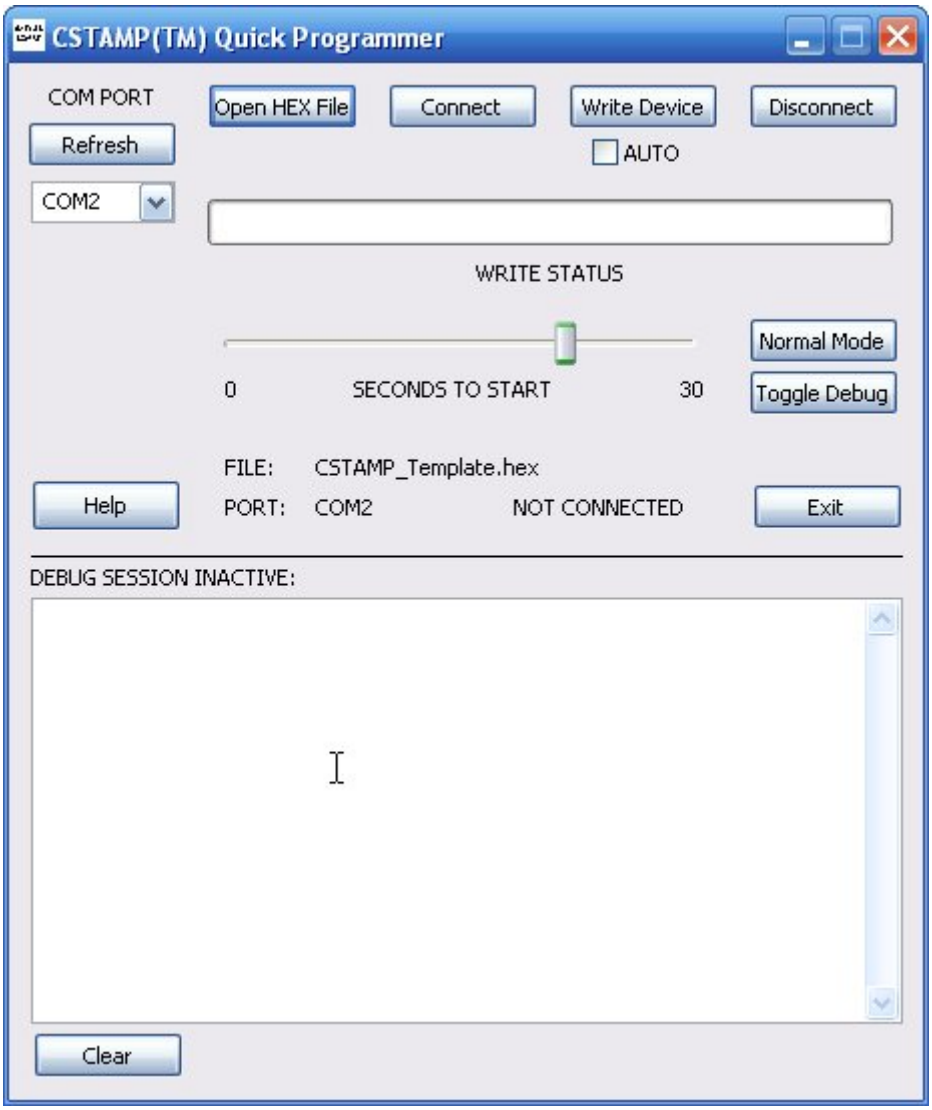
Save your program from the "File" menu or by clicking on the appropriate icon in the tool bar. Then "Build All" from the "Project" menu or from the tool bar.

If the code was typed correctly, you will have a file in your program directory with the name of your program project and a .HEX extension. An example is FIRST_LED_APP.HEX. This is the file that you will download to the C Stamp, as explained up ahead.

If you get an error message or an indication that your program did not build successfully in the "Output" sub-window of the IDE, you probably have one or more syntax error(s). Double click on the line of the "Output" sub-window that mentions the error, and the program line that most likely contains the error will be indicated in the sub-window where you were editing your program. Correct as necessary, and "Build All" again until you get a successful .HEX file output.

Downloading and Running Your Program

Insert the C Stamp in the BOL, power up your BOL, and connect the BOL to the PC with the provided cable. Upon power up, the C Stamp will be in RESET/BOOT/DOWNLOAD mode. To go back to this mode at any time, just push and let go of the RESET button. Then open the A-WIT C Stamp Quick Programmer application shown in the next figure.



The first step is to choose the serial port that you are using from the drop-down menu. Then click on "Refresh", so that the program registers your selection. Your selection should show in the status area of the program next to "PORT:". Then click on "Open

HEX File" and load/select the HEX file that you had previously created during the development of your program. The status are should indicate that the file has been loaded successfully. This is what will be downloaded to the C Stamp. Then click on "Connect", and the PC will be connected to the BOL, and the status area should indicate so. To download the HEX file to the C Stamp, just click on "Write Device", and you should see the progress bar after a few seconds, as the HEX file is downloaded. At this point, you can click on "Disconnect" to disconnect the PC from the BOL, disconnect the serial cable from both the PC and the BOL, and start your program manually at the BOL. To do this just push and let go of the RESET button while pushing the START button. Then you can let go of the START button. Alternatively, you can click on "Normal Mode" to start your program from the PC. This will also disconnect the program/PC from the BOL. Then you can disconnect the serial cable from the PC and the BOL. You can also instruct the CSTAMP™ Quick Programmer to wait several seconds before starting your program from the PC and disconnecting by adjusting the "SECONDS TO START" slide. This feature is useful in case you want to keep the PC connected with the serial cable, but need time to manually set up something in a circuit that you have built. If this is not the case it can just be left at the default of "0", and your program will started from the PC right away. After you click "Normal Mode" and your program is started, the CSTAMP™ Quick Programmer will not be communicating with the C Stamp any longer, so if you want to reconnect, you must click on "Connect" again.

Bypassing the START/RESET Sequence

If you want the C Stamp enter your program at power up bypassing the START button sequence, the solution is very simple. Just tie pin 38 of the C Stamp permanently to ground through a low value resistor less than 287 Ω. This will cost you that pin (i.e. you cannot use that pin in your project), but since the C Stamp has so many pins, this should be an acceptable option.

Using the BOL Breadboard

To build circuits that require additional components, the μC101 BOL provides a spacious full size solderless breadboard. Breadboards are simply a matrix of inserts that allow you to connect electrical circuits a simple manner with the supplied jumper wires without the need of solder. At the top and bottom of each panel there are rows of connections that are used for power. These are denoted with a solid red or blue line, and each row corresponding to a color has all its inserts connected together. The rest of the inserts are labeled in rows (A-J) and columns (1-63). For each column, each group of rows (A-E) or (F-J) are connected together inside the breadboard, and are used to connect different components by inserting terminals into the inserts of that group. If you run out of inserts in a group, then jump a connection to an unused group and keep connecting components.

Developing Your Own Programs and Projects

Now that you have successfully developed and run your program, it is easy to move on to more complex and elaborate projects and circuits of your own or the ones described in Chapter 3 of this manual: CS310XXX (μC 101) Reference Manual. To do this, your first step should be to thoroughly study Chapter 5 of the C Stamp Syntax and Reference Guide Manual.

Chapter

3

Design Projects and Activities

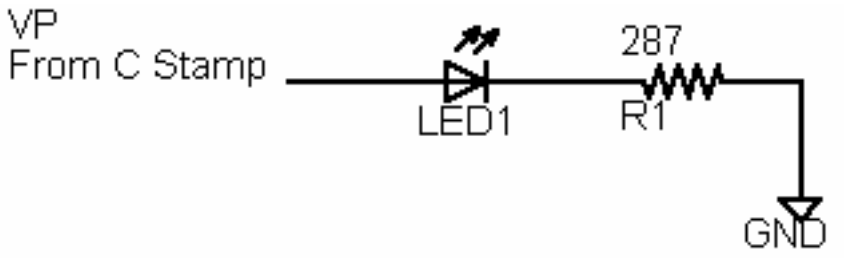
This chapter describes the projects and activities that can be done with the C Stamp and the μC101 BOL It is advisable that, to learn the most and get the most out of your C Stamp, the activities are attempted first without reading on to the solutions. The material in this chapter is written at a very fundamental level, so depending on the present level of expertise, not all the material will have to be reviewed by all readers. While this manual is generally written in a variable size font, code and its comments are written with a "`fixed font`". Additionally, keywords, functions, and commands of the WC language are in "**`bold`**". Generic syntax is in a "***`bold and italics fixed font`***".

Project 1: Using Light Emitting Diodes (LEDs)

In this project, you will light a light emitting diode (LED), and show that it lights 60 times in one minute using an I/O pin of the C Stamp.

Project 1: Solution

The first step is to define the topology for a circuit that implements a driven LED by the C Stamp voltage from one of its pins. We will denote this voltage as VP that can be either 0 V or 5 V as controlled by your C Stamp program. The circuit that accomplishes this function is shown in the figure below.



This circuit is comprised by an LED LED1 and a limiting resistor R1. In general, diodes are electronic devices that (conduct) allow current to flow) in one direction from a high voltage point in the circuit to a low voltage point, while dropping a constant voltage across it. Ground or GND is always at 0 V. The symbol similar to an arrow denotes the direction of the flow of current, and when the diode is conducting current, we say that it is forward biased. The left side of the symbol as drawn above is called the anode, and the right side the cathode. In the physical package of LEDs like the ones offered by A-WIT, there is a flat side that denotes the cathode. In other words, the lead closest to the flat side is the cathode of the LED, and the other is the anode.

To forward bias a diode, the anode must be at a higher voltage then the cathode by the constant voltage that the diode drops across it. When forward biased, the voltage across a diode is constant. LEDs are a special type of diode that emits light when forward biased. Depending on the semiconductor material used to fabricate the LED, the color (wave length) of the light emitted is different, so any given LED will emit light of a given color. For example, the LEDs provided in the μC101 BOL are green, and they will always emit green light. Manufacturers of LEDs provide data to users of the LEDs that relate how much current the LED should be biased with to achieve a particular level of brightness, and what voltage will be dropped across the LED. For the LEDs in the μC101 BOL this current ID is 10 mA for a good brightness, and the voltage dropped across the diode VD is 2.1 V.

The other component in the circuit is R1. Resistors are electronic components that allow certain amount of current to flow through them depending on the voltage across it, and the value of its resistance. The resistance value depends on the material and how much of this material is used to fabricate the resistor. The unit of resistance is ohms, and it is symbolized by the capital Greek letter omega (Ω). In many resistors, the resistance is color coded with three, four, or five color stripes. If the last stripe is either gold or silver, it represents the tolerance of the resistance (i.e. how much the value of the resistance varies, since no component will have an exact value over all possible manufactured pieces). A gold band means a ±5% tolerance, silver means ±10%, and the absence of a gold or silver band means ±20%. For the other color bands, the first stripe represents the first digit, the second the second digit, and the third a power of 10 multiplier. If an additional non gold or silver band is present, the third stripe represents the third digit and the fourth the power of 10 multiplier. The numerical meaning of the non gold or silver color bands are shown in the table below. For example, a resistor with color bands yellow, violet, and brown has a resistance of 470 Ω. 4 for the first digit, 7 for the second and a 1 power of ten multiplier $10^1 = 10$ (47 x 10 = 470).

Color		***Meaning***
	BLACK	0
	BROWN	1
	RED	2
	ORANGE	3

Color		*Meaning*
	YELLOW	4
	GREEN	5
	BLUE	6
	VIOLET	7
	GRAY	8
	WHITE	9

The equation that governs how much current a resistor allows through it (I) given its resistance value (R), and a given voltage V across it is

$$I = \frac{V}{R}.$$

This is called Ohm's Law.

With this equation, the knowledge we have gained about the LED, and the diode data from the manufacturer, we are able to proceed to calculate the necessary value of the resistor that will set the current that we want through the LED to achieve a given brightness when the C Stamp pin is at 5 V. We know that we need to set the C Stamp pin at 5 V (HIGH) to light the LED, because the alternative (LOW) will be 0 V, and then there will not be any voltage difference between the C Stamp pin and GND to force any current through the LED; so VP = 5 V.

The value of R1 will be given by the voltage across it VR and the current through it, which will be the same current that will be conducted by the diode ID. We can express this relationship mathematically by writing Ohm's Law as

$$R1 = \frac{VR}{ID}.$$

But since we know that the LED will have a constant voltage across it, and the total voltage available as supplied by the C Stamp pin VP is 5 V to light the LED, then

$$VR = VP - VD,$$

and

$$R1 = \frac{VP - VD}{ID}.$$

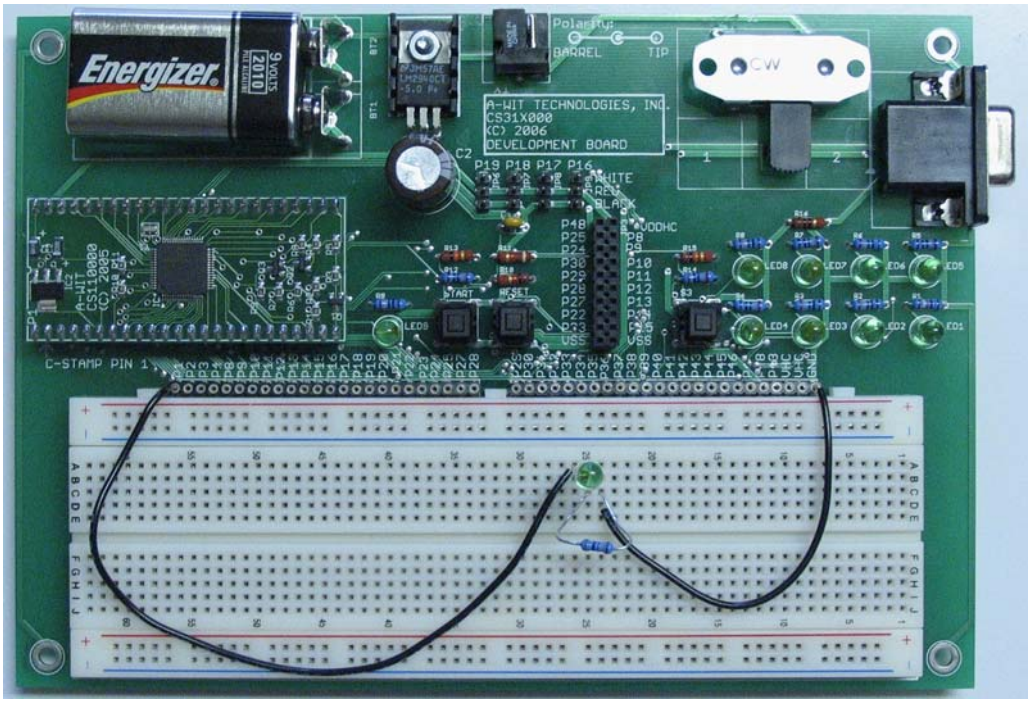
Since we know all the quantities in the right hand side of the equation, we can easily calculate the value of R1.

$$R1 = \frac{5\,V - 2.1\,V}{10\,mA}$$

$$R1 = 290\ \Omega$$

We can choose a standard value for R1 of 287 Ω. The lower and closest value to the calculated one will insure that at least we get the current we want. Remember that if the resistance is increased, the current will decrease, according to Ohm's Law. Remember the Law! - Ohm's Law that is.

Now that we know all the component values, the circuit can be built in the μC101 BOL breadboard using the C Stamp PIN 1 to provide VP, and the program can be developed. The built circuit is shown in the figure below.



The programming problem at hand is to turn the LED on and off 60 times in one minute or every second. The program below will accomplish this task. As is the case in programming, there are multiple solutions to a programming problem. The code below is our solution, and the comments embedded in the code offer more clarifications.

```
#include   "CS110000.h"

void main(void)
```

```
{
  STPIND(1, LOW);  // start with the pin low
  while(TRUE){     // do this forever
    PAUSE(1000);   // wait 1 second
    TOGGLE(1);     // toggle pin 1
  }
}
```

The operation of the program is as follows. We start by setting PIN 1 **LOW**. Then in an infinite loop, we wait for 1 second, and then change the value of PIN 1 from 0 V to 5 V if the pin is at 0 V; and from 5 V to 0 V if the pin is at 5 V. This is accomplished by the **TOGGLE** command. Notice that there is no need to end the program with an **END** command, because the program will stay executing what is inside the **while** loop as long as there is power applied to the C Stamp or until the C Stamp is reset by pressing the μC101 BOL RESET button.

Project 2: Using a 7 Segment LED Digit Display

Now that we know how to bias LEDs, we can proceed to use more complex display techniques using a 7 Segment LED Digit Display. In this project, you are to write a program that will display all the numbers and letters that are possible using a 7 Segment Display. The numbers should light up first in the sequence 0 to 9 then followed by the letters A to Z. The requirement is for the display to stay on for 5 seconds.

Project 2: Solution

The circuit for this project is a 7 Segment Display properly biased and connected to the C Stamp in the μC101 BOL. The A-WIT Part# CS480000 is a suitable 7 Segment LED Digit Display that can be used for this project. This display is also provided with some A-WIT KITS.

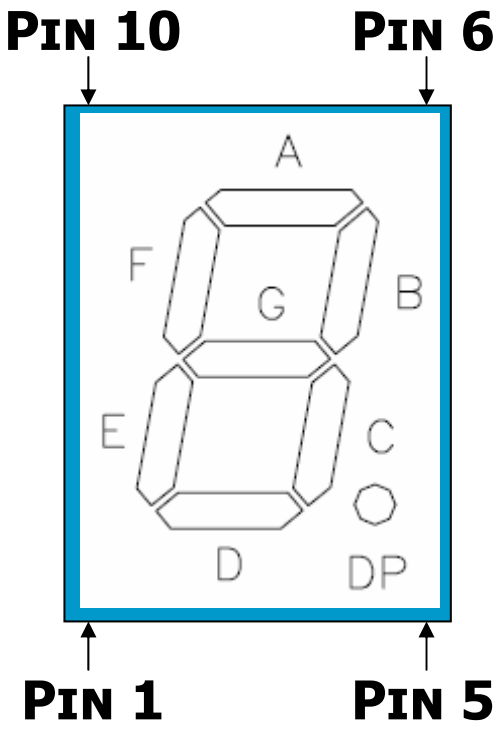
The CS480000 is a 0.56 inch (14.22 mm) digit height single-digit display. This device uses GREEN LED chips. The display has gray face and green segments. Driving the segments is made easy with A-WIT's supplied software command STPIND. This simple one command interface is all that is required to drive each segment in your project. Just connect one pin of the display through a current limiting resistor to a C Stamp pin, and the common anode to VHC, and execute a software command.

Technical Specifications

- 0.56 inch (14.22 mm) digit height
- Excellent segment uniformity
- Low power requirement

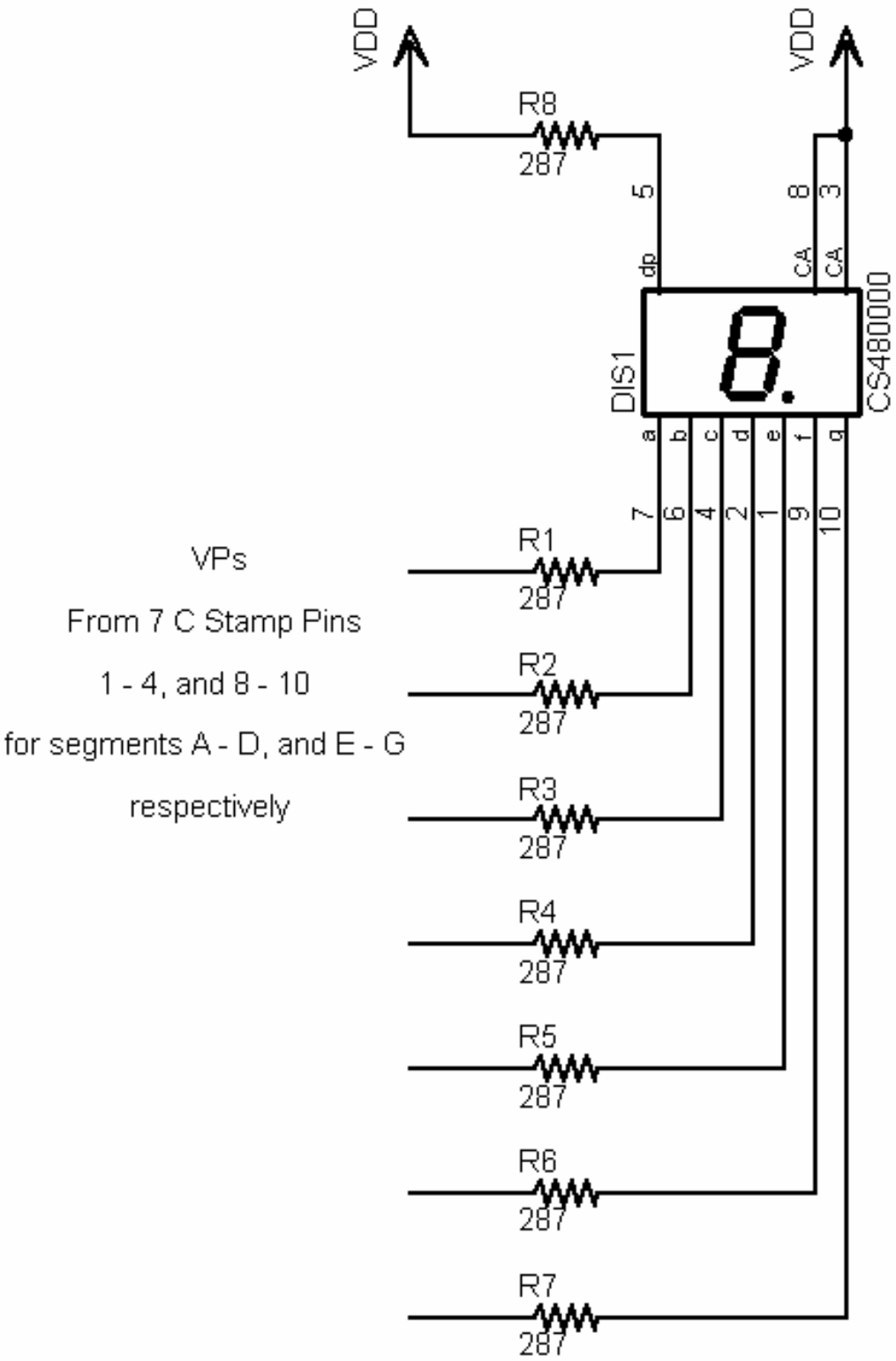
- High brightness and high contrast
- Wide viewing angle
- Solid state reliability
- High luminous intensity
- Common anode
- Current limiting resistor per segment RL = 287 Ω
- Additional right decimal point
- Breadboard friendly

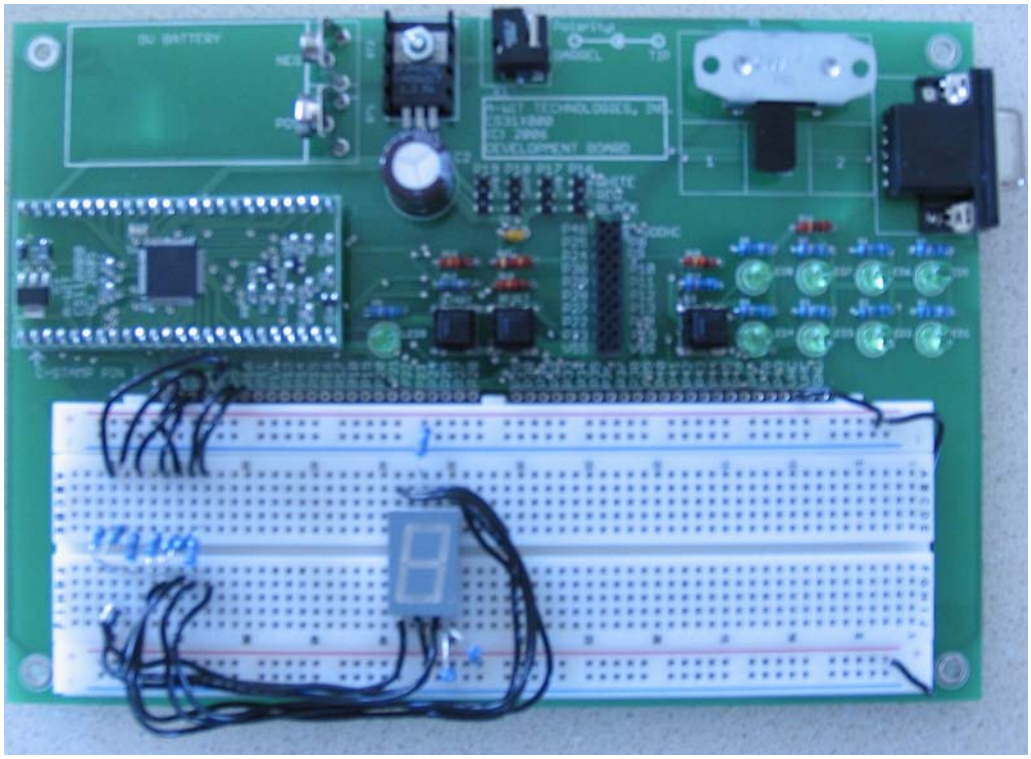
The figure below shows the pins orientation of the display and the segment notations; and the following table shows the pin to segment mapping.



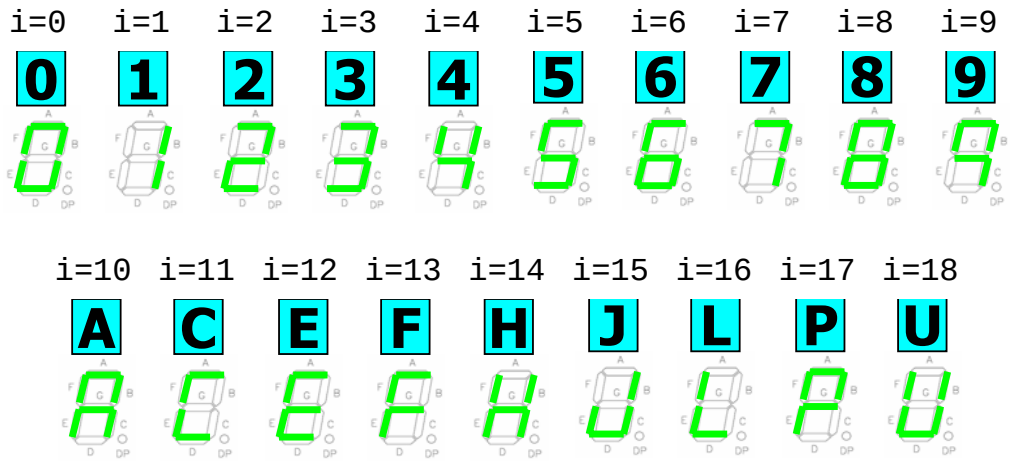
Pin Number	*Connection*
1	Cathode E
2	Cathode D
3	Common Anode
4	Cathode C
5	Cathode DP
6	Cathode B
7	Cathode A
8	Common Anode
9	Cathode F
10	Cathode G

Since the display has a common anode, which in LEDs has to be connected to a higher voltage than the cathode for the LED to be lit, we know that we have to connect Pin 3 or Pin 8 (or both) to VHC. Then we control whether each segment is lit by providing a voltage VP from the C Stamp. A **`LOW`** will light a segment, and a **`HIGH`** will keep the segment off. From the specifications of the display, we also know that each segment needs a current limiting resistor RL = 287 Ω. Additionally, since the decimal point (DP) segment will not be used, we connect it to VHC through a limiting resistor. The circuit schematic and the wired circuit in the μC101 BOL are shown in the following two figures. We use C Stamp pins 1 – 4, and 8 – 10.


VDD
VDD
R8
287
5
8
3
dp
CA
CA
DIS1
8.
CS480000
a
b
c
d
e
f
g
7
6
4
2
1
9
10
VPs
From 7 C Stamp Pins
1 - 4, and 8 - 10
for segments A - D, and E - G
respectively
R1
287
R2
287
R3
287
R4
287
R5
287
R6
287
R7
287



The programming task is more challenging. The first step is to determine which characters (numbers 0 – 9, and letters A – Z) can be displayed without ambiguity, and what are the segments that need to be lit for each character. The figure below shows this.



The programming task thus becomes to light up each segment according to the character that needs to be displayed, and having the delay between displaying each full character. A brute force approach would be to have a group of 7 **STPIND** commands for each character for a total of 19 groups. However; we can be more clever, and look at the problem as having 7 groups of either turning each segment on or off according to the character that is being displayed taking advantage of the commonality that each segment is turned on or off for different characters. The code below accomplishes this task, and the comments explain how the code works.

```
#include  "CS110000.h"

void main(void)
{
  BYTE i; // loop counter
// implicitly i = 0 – 18 corresponds to 0 – 9 and A – U

// turn off all segments
  STPIND(1, HIGH); STPIND(2, HIGH); STPIND(3, HIGH);
  STPIND(4, HIGH); STPIND(8, HIGH); STPIND(9, HIGH);
  STPIND(10, HIGH);

  for(i=0; i<19; i=i+1){ // loop through the characters
    PAUSE(5000); // wait 5 seconds

// turn on segment A for these characters
    if(i==0 || i==2 || i==3 || (i>4 && i<14) || i==17)
      STPIND(1, LOW);
    else STPIND(1, HIGH);

// turn on segment B for these characters
    if(i<5 || (i>6 && i<11) || i==14 || i==15 ||
       i==17 || i==18)
         STPIND(2, LOW);
    else STPIND(2, HIGH);

// turn on segment C for these characters
    if(i==0 || i==1 || (i>2 && i<11) || i==14 ||
       i==15 || i==18)
         STPIND(3, LOW);
    else STPIND(3, HIGH);

// turn on segment D for these characters
    if(i==0 || i==2 || i==3 || i==5 || i==6 || i==8 ||
```

```
       i==11 || i==12 || i==15 || i==16 || i==18)
         STPIND(4, LOW);
    else STPIND(4, HIGH);

// turn on segment E for these characters
    if(i==0 || i==2 || i==6 || i==8 || i>9)
      STPIND(8, LOW);
    else STPIND(8, HIGH);

// turn on segment F for these characters
    if(i==0 || (i>3 && i<7) || (i>7 && i<15) || i>15)
      STPIND(9, LOW);
    else STPIND(9, HIGH);

// turn on segment G for these characters
    if((i>1 && i<7) || (i>7 && i<11) ||
        (i>11 && i<15) || i==17)
         STPIND(10, LOW);
    else STPIND(10, HIGH);
  }

  PAUSE(5000); // wait 5 seconds

// turn off all segments
  STPIND(1, HIGH); STPIND(2, HIGH); STPIND(3, HIGH);
  STPIND(4, HIGH); STPIND(8, HIGH); STPIND(9, HIGH);
  STPIND(10, HIGH);

  END();
}
```

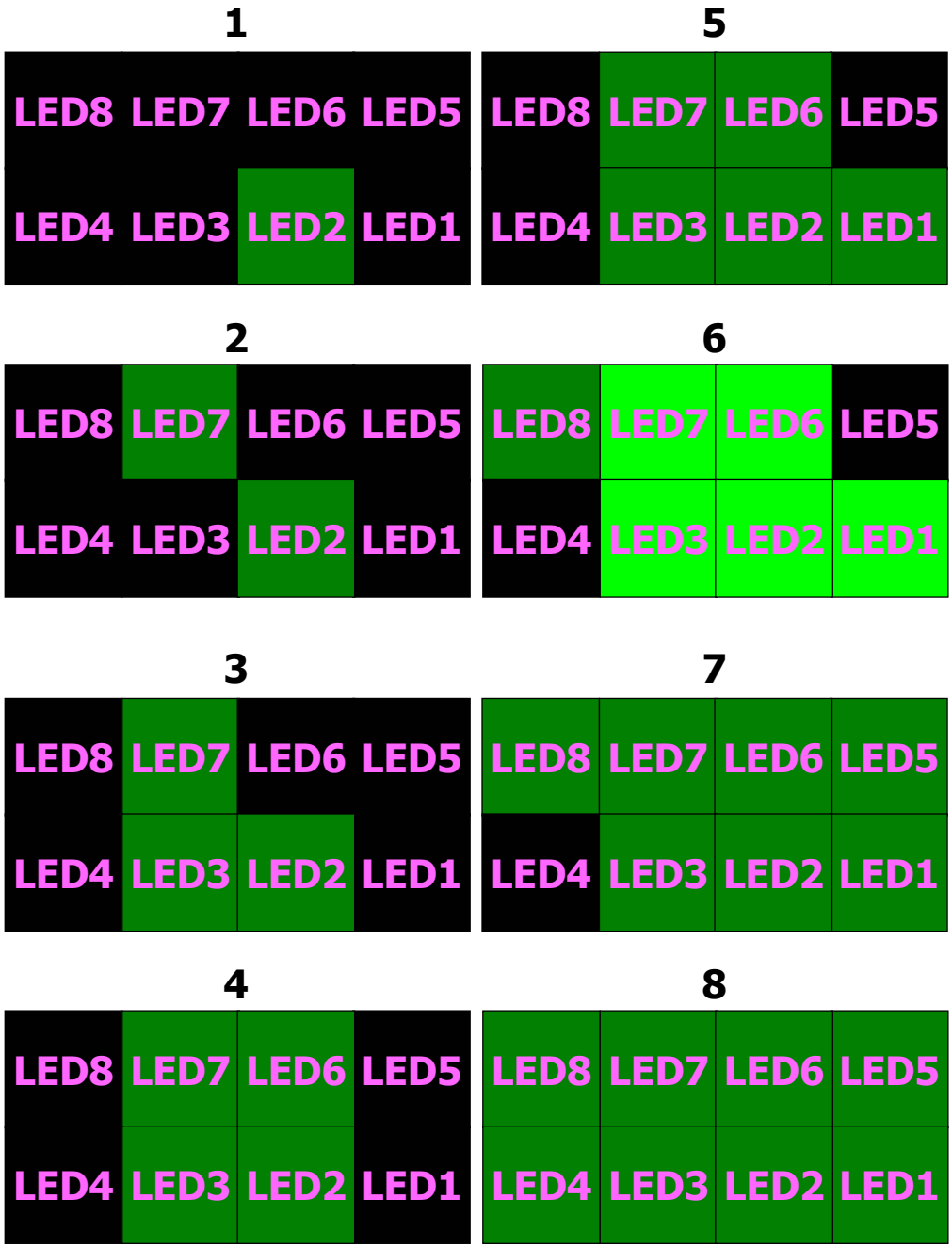
The operation of the program is as follows. We start by turning off all the segments. Then we loop through the loop counter `i`, which implicitly represents our character sequence. For each character, we examine whether each segment should be on or off for that character and act accordingly. In the loop, we also wait for 5 seconds at the beginning of the loop to have the required delay between characters. When we finish looping through all the character, we wait another 5 seconds to display the last character, and then turn all the segments off, and end the program. The program can be restarted by restarting the μC101 BOL. That is resetting the BOL while holding down the START button, and then letting go of it.

Project 3: The Board Game Dice

How many times has this happened to you? It is a rainy Sunday afternoon, there is nothing on TV, and you decide to play a board game with the family. You go to the

closet and get out the Monopoly® game, only to find that the die is missing (note: die is the singular for dice). Wouldn't it be great to have a simple electronic device to replace all the dice that have gone missing? Well, after you complete this project; that is exactly what you will have.

In this project, you will code and test a Board Game Dice program that randomly displays a number between one and eight in the patterns that we will specify. The patterns that will represent the numbers 1 through 8 using the eight LEDs of the BOL are shown in the figure below. Black means that the LED is off and green means that the LED is on. The program will work, such that after the start sequence is applied to the BOL, the program will wait for button S3 in the BOL to be pushed. Then it will flash ten random numbers at a rate of 100 mS to mimic the dice rolling. Finally, the program will display the random dice value, and go to wait for S3 to be pressed again to go through the sequence of action all over again.



Project 3: Solution

The code below accomplishes the task we have set forth in this project and the comments in the code offer more clarifications.

```
#include  "CS110000.h"

void main(void)
{
// TYPE YOUR CODE HERE AFTER THIS LINE
// define LED pins
  BYTE LEDS[8] = {46, 45, 44, 43, 42, 41, 40, 39};

// button pin
  BYTE B = 37;

// some counter variables
  NIBBLE i;
  WORD w;

// number variable
  NIBBLE N;

// LEDs byte patterns
  BYTE LB[8] =
    {0x02, 0x42, 0x46, 0x66, 0x67, 0xE7, 0xF7, 0xFF};

// roll dice every time button is pushed
  while(TRUE){
// wait for button to be pushed
    w = 0;
    do{
      w = w + 1;
    }while(GTPIND(B));

    srand(w);

    for(i=0; i<10; i=i+1){
      N = (rand() / 32768.0) * 8;
      BYTEOUT(LB[N], LEDS);
      PAUSE(50);
      BYTEOUT(0, LEDS);
      PAUSE(50);
    }

    N = (rand() / 32768.0) * 8;
    BYTEOUT(LB[N], LEDS);
  }
}
```

Functional Test: Now that the coding of the Board Game Dice is complete, test the device to determine if it is working properly.

First, start the program. Next, press and hold switch S3 in the BOL. This should cause all of the LED to flash. Release S3 and watch as the LEDs flash and eventually stops. If everything is working as expected, you have completed the programming of the Board Game Dice.

Trouble Shooting: Given that the Board Game Dice is a proven design (i.e., we know that it works if programmed correctly), its malfunction can only be caused by a bug in the code. Make sure your code matches the program above.

Conclusions:

1. If you were to use your Board Game Dice to play Monopoly® during a family game night, you would want the numbers rolled to be random and evenly distributed (i.e., the likelihood of rolling a one is the same as two, is the same as three, etc). How fair is your Board Game Dice? To measure this, use the data table shown below to tally each number displayed as the roll button S3 is pressed one hundred times. After you complete the tally, calculate the total count for each number and its distribution.

Number Rolled	***Tally Count***	***Total Count***	***Distribution***
1			
2			
3			
4			
5			
6			
7			
8			

How evenly distributed were the numbers for your Board Game Dice? If your game was perfect, then each number should have come up approximately

12.5% of the time. Do you think your Board Game Dice is fair? Why or why not?

2. Another word for troubleshooting is debugging. Do a little research (Google, of course) to determine the origin of the term debug and who was the person to coin the phrase.

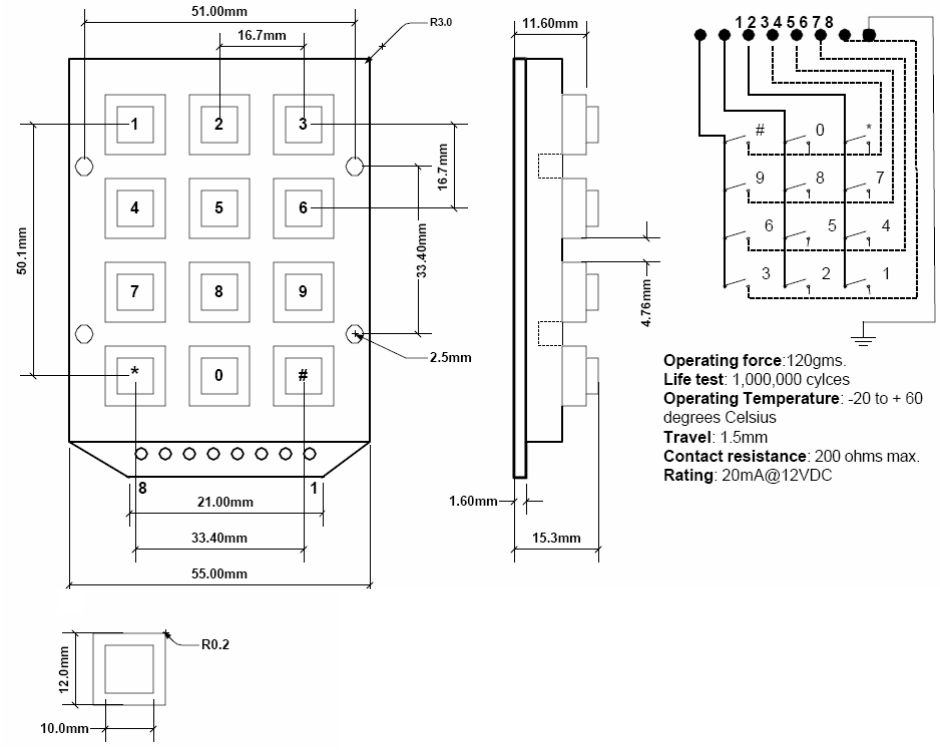
Project 4: Interfacing and Using a Keypad

In this project, we will interface and use a keypad, and write the program for it. The program should display all the numbers when you use a 4x3-matrix keypad as the input. The numbers to be displayed are 0 – 9. The characters will be displayed on a 7-segment LED display like the one discussed in Project 2. The characters should be inputted in any order, i.e. any random selection should be displayed. The requirements are for the display to stay on for as long as the key is being pressed and for 1 second after the key is released. The C Stamp is to signal the user that another key may be depressed by remaining dark for 3 seconds. The user should be able to terminate the program by pressing the '#' key.

Project 4: Solution

The circuit for this project is a 7 Segment Display properly biased and connected to the C Stamp in the μC101 BOL like discussed in Project 2, and a keypad connected to the C Stamp as well. The A-WIT Part# CS480000 and CS410001 are suitable 7 Segment LED Digit Display and keypad respectively, that can be used for this project. The display and keypad are also provided with some A-WIT KITS.

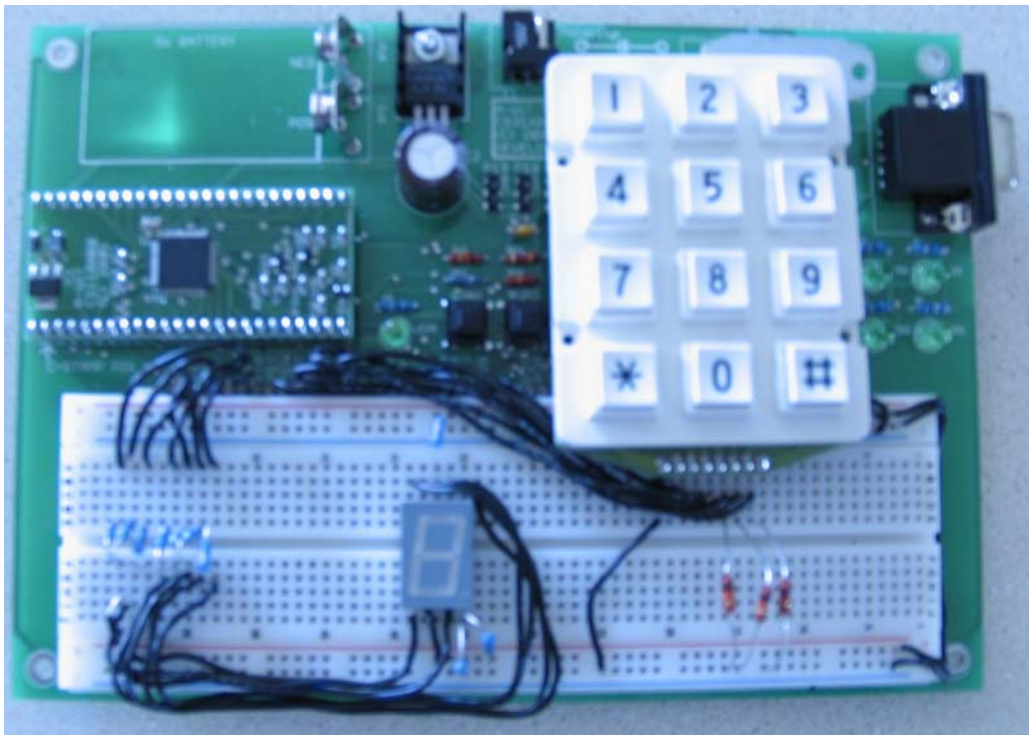
The following figures describe the CS410001 Keypad that can be interfaced with the C Stamp, and its pin-out connectivity.



Keypad Pin-Out Connectivity	
Pin	***Description***
1	Column 3
2	Column 2
3	Column 1
4	Row 4
5	Row 3
6	Row 2
7	Row 1
8	GND (optional connection)

To interface the keypad to the C Stamp, we simply connect all the pins that must be connected to C Stamp pins. The first step in devising a keypad sensing scheme is to decide which level we are going to use as our active level (**HIGH** or **LOW**). We will use **HIGH** as our active level in this solution. This means that a **HIGH** will mean that a key is active. To sense inputs from the keypad, we then set each row **HIGH** one at a time with the C Stamp pins, and "listen" for which row(s) are active through C Stamp pins. This will tell us which key(s) have been pressed. It is more efficient from a code writing perspective to listen as little as possible, so we have decided to set the rows and listen for columns, since the keypad being used for this solution has more rows than columns.

The connections for the display will be the same as those used in Project 2. For the keypad, we will use C Stamp pins 11 – 13 for Columns 3, 2, and 1; and pins 14 – 17 for Rows 4, 3, 2, and 1. All connections will be done in the order previously mentioned. 10 KΩ resistors should also be connected from the column pins to ground to pull down the pins when they are not being connected to a row pin by the pressing of a key. The wired circuit in the μC101 BOL is shown in the following figure.



The code below accomplishes the tasks we have set forth in this project, and the comments explain how the code works. In this project, we will simply ignore, as far displaying is concerned, the "*" and "#", since they cannot be displayed. However; we

will still recognize the '#' key for the purpose of the user signaling the program to terminate. We will denote these two keys as 10 for "*" and 11 for "#".

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE i, ii; // loop counters
  BYTE row_pins[]={17, 16, 15, 14}; // row pins
  BYTE column_pins[]={13, 12, 11};  // column pins
  NIBBLE key1, last_key1; // key variables
// flags
  BIT key_event, first_key_pressed;

// initialize variables
  last_key1 = 12; // impossible key

// initialize flags
  first_key_pressed = FALSE; // no first key yet

// turn off rows
  for(i=0; i<4; i=i+1) STPIND(row_pins[i], LOW);

// turn off all segments of the display
  STPIND(1, HIGH); STPIND(2, HIGH); STPIND(3, HIGH);
  STPIND(4, HIGH); STPIND(8, HIGH); STPIND(9, HIGH);
  STPIND(10, HIGH);

  while(TRUE){ // infinite loop
    key1 = 12; // reset key variable
    for(i=1; i<5; i=i+1){ // set up row
      STPIND(row_pins[i-1], HIGH);
      for(ii=1; ii<4; ii=ii+1){ // listen for columns
        key_event = GTPIND(column_pins[ii-1]);
        if(key_event){
          if(i==1 && ii==1) key1=1;
          if(i==1 && ii==2) key1=2;
          if(i==1 && ii==3) key1=3;
          if(i==2 && ii==1) key1=4;
          if(i==2 && ii==2) key1=5;
          if(i==2 && ii==3) key1=6;
          if(i==3 && ii==1) key1=7;
          if(i==3 && ii==2) key1=8;
          if(i==3 && ii==3) key1=9;
          if(i==4 && ii==1) key1=10;
```

```
          if(i==4 && ii==2) key1=0;
          if(i==4 && ii==3) key1=11;
        }
      }
      STPIND(row_pins[i-1], LOW);
    }
// check for program termination condition
    if(key1 == 11) break;

// check for no key pressed condition
    if(key1==12){
      if(first_key_pressed){
        first_key_pressed = FALSE;
        PAUSE(1000); // keep the display for 1 second
// turn off all segments of the display
        STPIND(1, HIGH); STPIND(2, HIGH);
        STPIND(3, HIGH); STPIND(4, HIGH);
        STPIND(8, HIGH); STPIND(9, HIGH);
        STPIND(10, HIGH);
        PAUSE(3000); // wait 3 seconds before next key
      }
    }

// check for one key pressed condition and display
    if(key1<10){
      if(!first_key_pressed) first_key_pressed = TRUE;
      if(key1 != last_key1){
        last_key1 = key1;
        i = key1; // use i for key1 to save on typing

// turn on segment A for these characters
        if(i==0 || i==2 || i==3 || i>4) STPIND(1, LOW);
        else STPIND(1, HIGH);

// turn on segment B for these characters
        if(i<5 || i>6) STPIND(2, LOW);
        else STPIND(2, HIGH);

// turn on segment C for these characters
        if(i==0 || i==1 || i>2) STPIND(3, LOW);
        else STPIND(3, HIGH);

// turn on segment D for these characters
        if(i==0 || i==2 || i==3 || i==5 || i==6 ||
           i==8) STPIND(4, LOW);
```

```
        else STPIND(4, HIGH);

// turn on segment E for these characters
        if(i==0 || i==2 || i==6 || i==8)
          STPIND(8, LOW);
        else STPIND(8, HIGH);

// turn on segment F for these characters
        if(i==0 || (i>3 && i<7) || i>7) STPIND(9, LOW);
        else STPIND(9, HIGH);

// turn on segment G for these characters
        if((i>1 && i<7) || i>7) STPIND(10, LOW);
        else STPIND(10, HIGH);
      }
    }
  }

// turn off all segments
  STPIND(1, HIGH); STPIND(2, HIGH); STPIND(3, HIGH);
  STPIND(4, HIGH); STPIND(8, HIGH); STPIND(9, HIGH);
  STPIND(10, HIGH);

  END();
}
```

The operation of the program is as follows. We start by turning off all the row signals and all segments of the display. Then, in an infinite loop we scan for a key value, and act accordingly depending on the type of key event that took place. We also keep track if a key has been displayed already, and we are waiting for a new key event to happen with the variable `first_key_pressed`. If a '#' key is pressed, we exit out of the infinite loop with the **`break`** command. If a valid key was just displayed, then we keep the display active for 1 second, turn the display off, and wait 3 seconds. If a valid key was pressed, we flag that a new key was just displayed. This takes place even if the new key was the same as the last one. This behavior allows repeated keys to be displayed. However; for actually displaying the character, we keep track of what was just displayed with the variable `last_key1`. This allows to keep the display on in the case a valid key is continuously pressed. When we are ready to display a character, we use the same techniques used in Project 2 to manipulate the 7 segment LED display. When a program termination condition is encountered, we simply turn off the display, and end.

This project is a good subject to demonstrate the use of functions. Instead of a single long in-line program, we could have used functions that are called from the main function to abstract some repetitious tasks. Three such tasks can be identified: turning off the display, scanning for a key, and displaying a character. The following version of

the program works exactly as the previous one, but these three procedural tasks have been abstracted as functions. Notice how the code is better partitioned, easier to understand, and overall more efficient.

```
#include  "CS110000.h"

void turn_off_display(void);
NIBBLE scan_for_key(void);
void display_character(NIBBLE i);

void main(void)
{
  NIBBLE i; // loop counter
  BYTE row_pins[]={17, 16, 15, 14}; // row pins
  NIBBLE key1, last_key1; // key variables
// flags
  BIT first_key_pressed;

// initialize variables
  last_key1 = 12; // impossible key

// initialize flags
  first_key_pressed = FALSE; // no first key yet

// turn off rows
  for(i=0; i<4; i=i+1) STPIND(row_pins[i], LOW);

// turn off all segments of the display
  turn_off_display();


  while(TRUE){ // infinite loop
    key1 = scan_for_key();

// check for program termination condition
    if(key1 == 11) break;

// check for no key pressed condition
    if(key1==12){
      if(first_key_pressed){
        first_key_pressed = FALSE;
        PAUSE(1000); // keep the display for 1 second
// turn off all segments of the display
        turn_off_display();
        PAUSE(3000); // wait 3 seconds before next key
```

```
      }
    }

// check for one key pressed condition and display
    if(key1<10){
      if(!first_key_pressed) first_key_pressed = TRUE;
      if(key1 != last_key1){
        last_key1 = key1;
        display_character(key1);
      }
    }
  }

// turn off all segments
  turn_off_display();

  END();
}

void turn_off_display(void)
{
// turn off all segments of the display
  STPIND(1, HIGH); STPIND(2, HIGH); STPIND(3, HIGH);
  STPIND(4, HIGH); STPIND(8, HIGH); STPIND(9, HIGH);
  STPIND(10, HIGH);
}

NIBBLE scan_for_key(void)
{
  BYTE row_pins[]={17, 16, 15, 14}; // row pins
  BYTE column_pins[]={13, 12, 11};  // column pins
  NIBBLE key, i, ii;
  BIT key_event;

  key = 12; // reset key variable
  for(i=1; i<5; i=i+1){ // set up row
    STPIND(row_pins[i-1], HIGH);
    for(ii=1; ii<4; ii=ii+1){ // listen for columns
    key_event = GTPIND(column_pins[ii-1]);
      if(key_event){
        if(i==1 && ii==1) key=1;
        if(i==1 && ii==2) key=2;
        if(i==1 && ii==3) key=3;
        if(i==2 && ii==1) key=4;
        if(i==2 && ii==2) key=5;
```

```
        if(i==2 && ii==3) key=6;
        if(i==3 && ii==1) key=7;
        if(i==3 && ii==2) key=8;
        if(i==3 && ii==3) key=9;
        if(i==4 && ii==1) key=10;
        if(i==4 && ii==2) key=0;
        if(i==4 && ii==3) key=11;
      }
    }
    STPIND(row_pins[i-1], LOW);
  }
  return key;
}

void display_character(NIBBLE i)
{
// turn on segment A for these characters
  if(i==0 || i==2 || i==3 || i>4) STPIND(1, LOW);
  else STPIND(1, HIGH);

// turn on segment B for these characters
  if(i<5 || i>6) STPIND(2, LOW);
  else STPIND(2, HIGH);

// turn on segment C for these characters
  if(i==0 || i==1 || i>2) STPIND(3, LOW);
  else STPIND(3, HIGH);

// turn on segment D for these characters
  if(i==0 || i==2 || i==3 || i==5 || i==6 || i==8)
    STPIND(4, LOW);
  else STPIND(4, HIGH);

// turn on segment E for these characters
  if(i==0 || i==2 || i==6 || i==8) STPIND(8, LOW);
  else STPIND(8, HIGH);

// turn on segment F for these characters
  if(i==0 || (i>3 && i<7) || i>7) STPIND(9, LOW);
  else STPIND(9, HIGH);

// turn on segment G for these characters
  if((i>1 && i<7) || i>7) STPIND(10, LOW);
  else STPIND(10, HIGH);
}
```

These functions could have been made more generic by passing arrays that define the pins connected to the display or the keypad. A more generic version that uses this generic connectivity follows. The functionality of the program is still the same as the previous two, but now just by changing the arrays that define the connectivity of the display and keypad, these components can be connected to any set of C Stamp pins.

```
#include  "CS110000.h"

void turn_off_display(BYTE pins2segs[]);
NIBBLE scan_for_key(BYTE row_pins [],
                    BYTE column_pins []);
void display_character(NIBBLE i, BYTE pins2segs[]);

void main(void)
{
  NIBBLE i; // loop counter
  BYTE pins2segments[] = {1, 2, 3, 4, 8, 9, 10}; // A-G
  BYTE pins2rows[]={17, 16, 15, 14}; // row pins 1-4
  BYTE pins2columns[]={13, 12, 11};  // column pins 1-3
  NIBBLE key1, last_key1; // key variables
// flags
  BIT first_key_pressed;

// initialize variables
  last_key1 = 12; // impossible key

// initialize flags
  first_key_pressed = FALSE; // no first key yet

// turn off rows
  for(i=0; i<4; i=i+1) STPIND(pins2rows[i], LOW);

// turn off all segments of the display
  turn_off_display(pins2segments);


  while(TRUE){ // infinite loop
    key1 = scan_for_key(pins2rows, pins2columns);

// check for program termination condition
    if(key1 == 11) break;

// check for no key pressed condition
    if(key1==12){
      if(first_key_pressed){
```

```
        first_key_pressed = FALSE;
        PAUSE(1000); // keep the display for 1 second
// turn off all segments of the display
        turn_off_display(pins2segments);
        PAUSE(3000); // wait 3 seconds before next key
      }
    }

// check for one key pressed condition and display
    if(key1<10){
      if(!first_key_pressed) first_key_pressed = TRUE;
      if(key1 != last_key1){
        last_key1 = key1;
        display_character(key1, pins2segments);
      }
    }
  }

// turn off all segments
  turn_off_display(pins2segments);

  END();
}

void turn_off_display(BYTE pins2segs[])
{
// turn off all segments of the display
  STPIND(pins2segs[0], HIGH);
  STPIND(pins2segs[1], HIGH);
  STPIND(pins2segs[2], HIGH);
  STPIND(pins2segs[3], HIGH);
  STPIND(pins2segs[4], HIGH);
  STPIND(pins2segs[5], HIGH);
  STPIND(pins2segs[6], HIGH);
}

NIBBLE scan_for_key(BYTE row_pins [],
                    BYTE column_pins [])
{
  NIBBLE key, i, ii;
  BIT key_event;

  key = 12; // reset key variable
  for(i=1; i<5; i=i+1){ // set up row
    STPIND(row_pins[i-1], HIGH);
```

```
    for(ii=1; ii<4; ii=ii+1){ // listen for columns
    key_event = GTPIND(column_pins[ii-1]);
      if(key_event){
        if(i==1 && ii==1) key=1;
        if(i==1 && ii==2) key=2;
        if(i==1 && ii==3) key=3;
        if(i==2 && ii==1) key=4;
        if(i==2 && ii==2) key=5;
        if(i==2 && ii==3) key=6;
        if(i==3 && ii==1) key=7;
        if(i==3 && ii==2) key=8;
        if(i==3 && ii==3) key=9;
        if(i==4 && ii==1) key=10;
        if(i==4 && ii==2) key=0;
        if(i==4 && ii==3) key=11;
      }
    }
    STPIND(row_pins[i-1], LOW);
  }
  return key;
}

void display_character(NIBBLE i, BYTE pins2segs[])
{
// turn on segment A for these characters
  if(i==0 || i==2 || i==3 || i>4)
    STPIND(pins2segs[0], LOW);
  else STPIND(pins2segs[0], HIGH);

// turn on segment B for these characters
  if(i<5 || i>6) STPIND(pins2segs[1], LOW);
  else STPIND(pins2segs[1], HIGH);

// turn on segment C for these characters
  if(i==0 || i==1 || i>2) STPIND(pins2segs[2], LOW);
  else STPIND(pins2segs[2], HIGH);

// turn on segment D for these characters
  if(i==0 || i==2 || i==3 || i==5 || i==6 || i==8)
    STPIND(pins2segs[3], LOW);
  else STPIND(pins2segs[3], HIGH);

// turn on segment E for these characters
  if(i==0 || i==2 || i==6 || i==8)
    STPIND(pins2segs[4], LOW);
```

```
  else STPIND(pins2segs[4], HIGH);

// turn on segment F for these characters
  if(i==0 || (i>3 && i<7) || i>7)
    STPIND(pins2segs[5], LOW);
  else STPIND(pins2segs[5], HIGH);

// turn on segment G for these characters
  if((i>1 && i<7) || i>7) STPIND(pins2segs[6], LOW);
  else STPIND(pins2segs[6], HIGH);
}
```

We will use these functions in subsequent projects.

Project 5: Home Security System

In this project, we will design and implement a home security system using the C Stamp, the μC101 BOL, and several accessories. The system has the following specifications.

1. The system will be designed to secure an average home.
2. There will be a minimum of 4 openings. They are either doors or windows.
3. The system will be enabled and disabled by inputting a 3 digit code sequentially.
4. There will be a 15 second delay upon entering the code and the enabling of the system. This will allow the occupant to exit the premises without setting off the alarm.
5. There will be a similar delay upon entering the premises allowing the occupant to enter the house without setting off the alarm.
6. There will be two distinct outputs for the system. 1) A signal to turn on a siren or bell attached to the house that will sound for bursts of 1 second. 2) A second signal that is delayed 30 seconds from the first one to enable a signal to a local monitoring station.
7. The signal to the outside siren will also have a silent feature attached to it.
8. The system will also indicate which opening was entered when enabled and will retain that information until the system is reset.
9. The inputs are the sensor outputs (4), the code to enable and disable the system, the silent feature, and the reset signal.

10. The outputs are the 2 system outputs discussed in (6) above and the status of each opening.

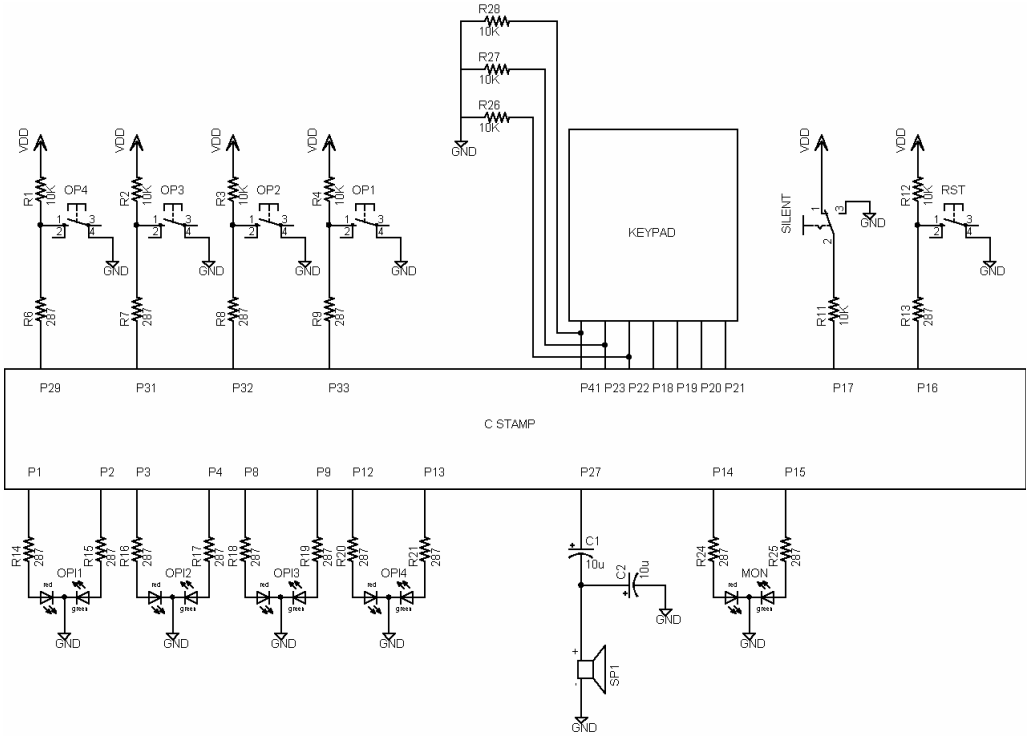
11. To activate or deactivate the silent feature, the user needs to enter the code first and then he or she will have 15 seconds to do this.

12. To reset the system, the user needs to enter the code first and then he or she will have 15 seconds to do this.

13. The system comes with a pre-programmed code of "000".

Project 5: Solution

As systems become more complex, we find that constructing a block diagram of the hardware that will support a given system is a useful abstraction tool to describe the system. A block diagram for this project is shown in the figure below.



For this project, we simulate the four openings being monitored with four push-button switches OP1 through OP4, and their associated resistors. This represents a monitoring scheme with a normally open contact and pulled **HIGH**. This means that when an opening is closed, there is a contactor that is open, and when the opening closes, this contactor closes and the C Stamp pin monitoring the opening is pulled

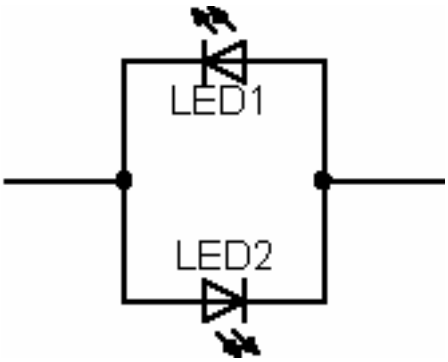
LOW. Any combination of normally open or closed contacts, and normally pulled HIGH or LOW can be used. What we have to know is what voltage level is the C Stamp pin receiving when the opening is closed and when it is open. In this case, if the opening is closed, the C Stamp pin will be receiving a **HIGH** voltage level, and if it is open, the C Stamp pin will be receiving a **LOW** voltage level.

The connections for the keypad will be similar as those used in Project 3. We will use C Stamp pins 41, 23, and 22 for Columns 3, 2, and 1; and pins 18 – 21 for Rows 4, 3, 2, and 1. All connections will be done in the order previously mentioned.

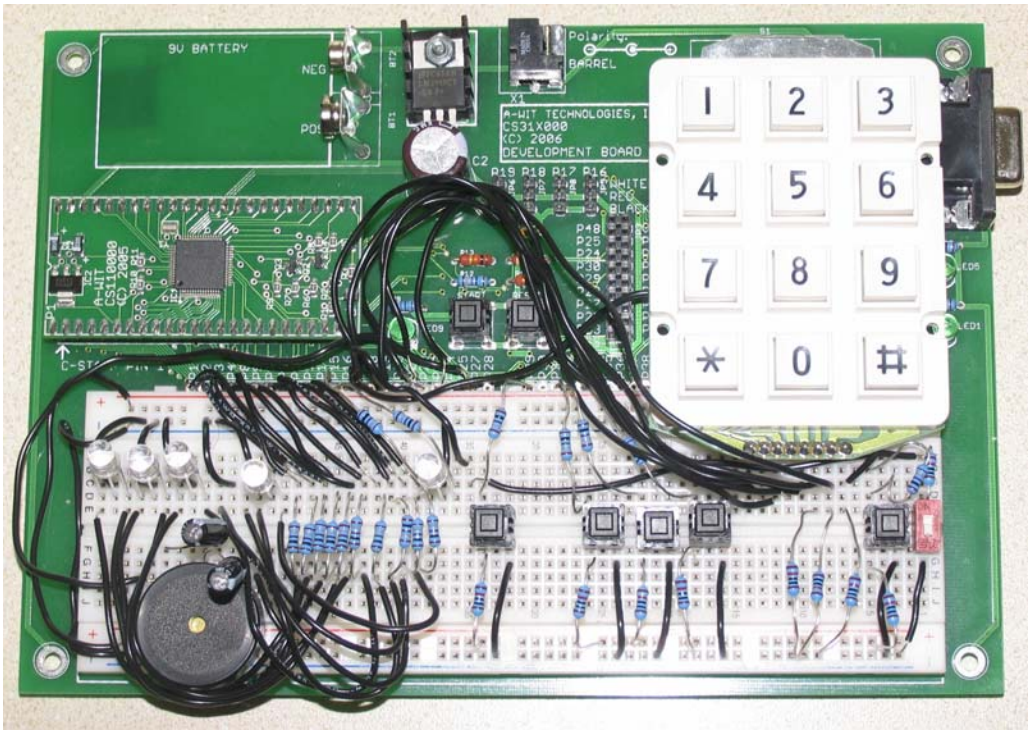
The SILENT switch is a constant switch (not a push-button) to activate the silent feature of the alarm. If the switch connects C Stamp pin 17 to VDD, then the security system will be in silent mode, and if it connects to ground, the security system will be in audible mode. The RST switch is used to reset the system to its initial state, except that any newly programmed access code will be retained.

The indicators toward the bottom-left corner of the diagram OPI1 – OPI4 are used to indicate the status of an opening. If the opening is open when the alarm system is armed, the indicator will be red, and if it is closed, the indicator will be green. This is accomplished with bi-color (green and red) LEDs properly biased like explained in Project 1. Bi-color LEDs contain two LEDs in one package. This type, like the CS511000, has a common cathode connection and two separate anode connections; one for each color. Therefore, the CS511000 that we use in this project has 3 leads. The shortest lead is the green anode, the next to longest lead is the red anode, and the longest lead is the common cathode. The cathode should be connected to ground, and the anodes should be connected to the C Stamp pins through the biasing resistors to activate the different colors by setting those pins **HIGH**. Setting the pins **LOW** will turn off the LED. With this LED, we could turn both colors at the same time, but we will not be using this feature for this project.

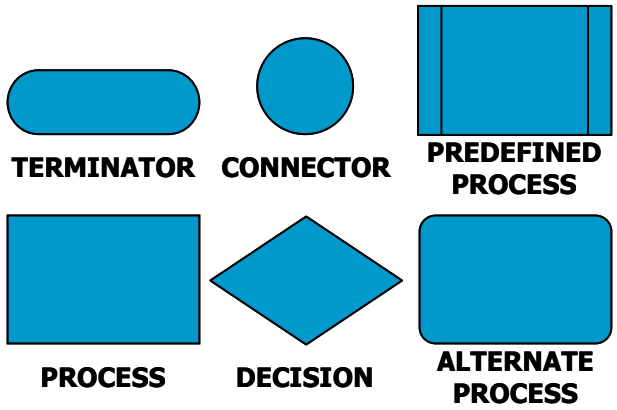
Other bi-color LEDs are like the one shown in the figure below. This type has two different colors LEDs in the same package connected back-to-back. This type would get connected to two C Stamp pins with a current limiting resistor whose value is calculated like in Project 1. Then setting a pin to a **HIGH** and the other to a **LOW** would light one color, and reversing the levels on both pins would light the other color. This type of LED does not allow for both colors to be on at the same time.



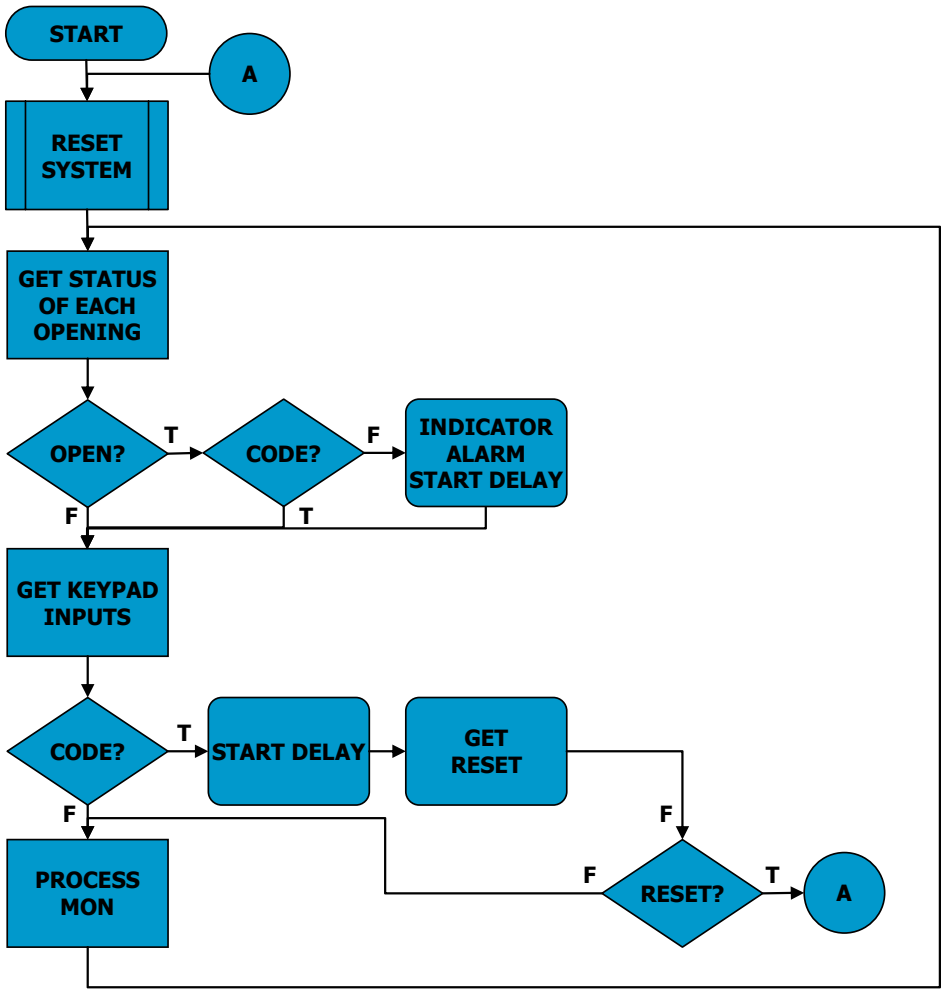
The audible alarm is connected to C Stamp pin 27 following the circuit indicated in the description of the **FREQOUT** command in the C Stamp Syntax and Reference Guide Manual for a piezo speaker. Such speaker could be the A-WIT CS430000 part number, which is used in our solution to this project. Finally, the last indicator toward the bottom right corner of the block diagram is used to represent the remote monitoring signal. If this indicator is green, then we are not signaling an intrusion, and if it is red we are. The wired circuit in the μC101 BOL is shown in the following figure.



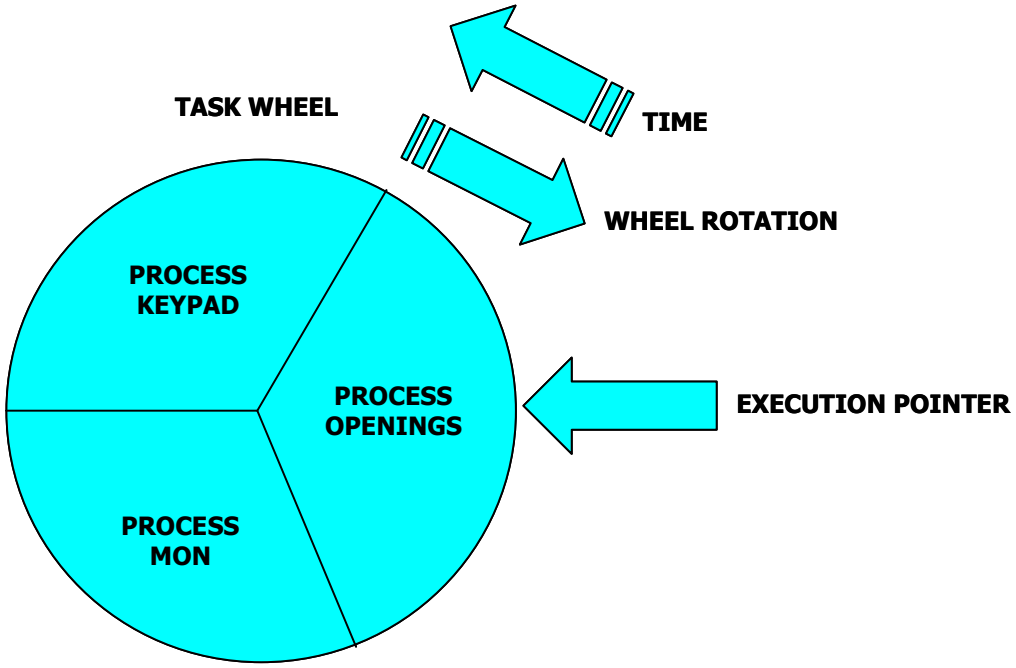
Once software systems start growing in complexity, it is helpful to avail ourselves of tools that allow us visualize and abstract the flow of our software. One such tool that we will use in this solution is the flowchart. The figure below shows the fundamental flowcharting symbols and their meanings. These are connected using arrows, so that the flowchart represents the flow of our program



Going from top left to bottom right in the figure above, the **TERMINATOR** symbol is used to show a starting or ending point of you program. The **CONNECTOR** is used to connect different parts of the flowchart without using arrows that may otherwise clutter your drawing. The **PREDEFINED PROCESS** encapsulates a set of well defined actions. The **PROCESS** is used to denote a generic process or set of actions that occur in your program at some point unconditionally. The **DECISION** symbol is used to show a test being made in your software flow, and then some conditional action or decision being made based upon the results of the test. These conditional actions or processes are represented by the **ALTERNATE PROCESS** symbol. There are many more flowcharting symbols, but these are the most commonly used ones, and are the ones used in the flowchart for our software system. This flowchart is shown in the figure below. The T's (TRUE) and F's (FALSE) by the DECISION symbols represent the possible outcomes of the test on which the decision is based. Flowcharts can be built at any level of detail or abstraction, and this depends in the level of comfort of the user. In our case we have chosen to build the flowchart at a fairly high level to show its usefulness without cluttering it with a lot of low level details that will be explained later. The flowchart does offer, however, a good idea and visual depiction of the main elements and flow of our planned software infrastructure. Always remember that the flowchart is a tool to help you develop and visualize the software system, and some details of the software itself will become more clear and apparent when you are actually coding.



We start our program by resetting the system. This entails making sure that all opening status indicator LEDs are green (OK condition), turning off the audible alarm, and turning off the remote monitoring signal. Then we start the main loop of our program. This approach of having one main, large, outer loop to do or check on all the tasks that our program needs to do by polling different components of our overall system is called a task wheel. This wheel is defined by the main three non conditional processes, and the conditional processes are said to be hidden within these non conditional tasks. Flags and counters are used for inter process communications. A visual depiction of our task wheel is shown in the figure below. Note how it reflects at a high level what our program keeps doing or repeating over and over again.



The first task is to check for the status of each opening. If any opening is open, we check if the right code has been entered. If that is the case, then we check whether the delay allotment that gets started by putting in the code has expired, so that the user can enter or exit without setting off the alarm. If the delay has expired, or no code or an incorrect code has been entered, we set the indicators and speaker alarm right away, and start a delay counter to signal when to set off the remote monitor alarm. After we process the openings, we collect inputs from the keypad. If the code is correct, we start the entrance/exit delay and also the delay for the user to push the reset button. If the reset button is pushed we reset the system and restart the task wheel. After we process the keypad, we check whether an intrusion condition is active and whether the delay for the remote monitoring signal has expired. If that is the case we set the remote monitoring signal, and restart the task wheel. Armed with our specifications of the system, our software flowchart, and a good understanding of our task wheel and what our code is supposed to do, we are ready to start code development. The code below is our solution for this project, and the comments embedded within the code offer more clarifications.

```
#include   "CS110000.h"

void main(void)
{
// define connections to C Stamp pins
  RAM BYTE OP_pins[]={33, 32, 31, 30, 29}; // sensors
  RAM BYTE row_pins[]={21, 20, 19, 18};  // row pins
```

```
  RAM BYTE column_pins[]={22, 23, 41}; // row pins
  RAM BYTE silent_pin=17;
  RAM BYTE RST_pin=16;
  RAM BYTE OPIred_pins[]={1, 3, 8, 12}; // indicators
  RAM BYTE OPIgreen_pins[]={2, 4, 9, 13}; // indicators
  RAM BYTE SP_pin=27;
  RAM BYTE MONred_pin=14;
  RAM BYTE MONgreen_pin=15;

  RAM BYTE timebase=250; // system time base is 250 mS
  RAM BYTE sec15d = 60;  // 15 seconds delay
  RAM BYTE sec30d = 120; // 30 seconds delay
  RAM BYTE sec60d = 240; // 60 seconds delay
  RAM NIBBLE i; // loop counter
  RAM NIBBLE i1, ii; // loop counters
  RAM NIBBLE first_set;

  RAM BIT key_event; // key event
  RAM NIBBLE key; // key input
  RAM NIBBLE code1, code2, code3; // code numbers
  RAM NIBBLE pkey_state, nkey_state;
  RAM BYTE code_counter;
  RAM BYTE exit_counter;
  RAM BYTE entry_counter;

// flags
  RAM BIT RST_flag=TRUE;
  RAM BIT code_flag=FALSE;
  RAM BIT silent_flag=FALSE;
  RAM BIT code_counter_flag=FALSE;
  RAM BIT exit_counter_flag=FALSE;
  RAM BIT entry_counter_flag=FALSE;
  RAM BIT intrusion_flag=FALSE;
  RAM BIT alarm_set_flag = FALSE;

// initialize variables
  pkey_state=0; nkey_state=0;
  code_counter=0;
  exit_counter=0;
  entry_counter=0;
  first_set=0;

// Initilaize the system LEDs
  for(i=0; i<4; i=i+1) STPIND(OPIred_pins[i], LOW);
  for(i=0; i<4; i=i+1) STPIND(OPIgreen_pins[i], HIGH);
```

```
  STPIND(SP_pin, LOW);
  STPIND(MONred_pin, HIGH);
  STPIND(MONgreen_pin, LOW);

// turn off rows and columns
  for(i=0; i<4; i=i+1) STPIND(row_pins[i], LOW);
  for(i=0; i<3; i=i+1) STPIND(column_pins[i], LOW);


  while(TRUE){ // infinite loop
// reset system
    if(RST_flag){
//reset flags
      RST_flag=FALSE;
      code_flag=FALSE;
      silent_flag=FALSE;
      code_counter_flag=FALSE;
      exit_counter_flag=FALSE;
      entry_counter_flag=FALSE;
      intrusion_flag=FALSE;
      alarm_set_flag=FALSE;
      code_counter=0;
      exit_counter=0;
      entry_counter=0;
      first_set=0;
      STPIND(MONred_pin, LOW);
      STPIND(MONgreen_pin, HIGH);
      for(i=0; i<4; i=i+1) STPIND(OPIred_pins[i], LOW);
      for(i=0; i<4; i=i+1)
        STPIND(OPIgreen_pins[i], HIGH);
      STPIND(SP_pin, LOW);
// set up code to 000 for the first time
      if(READ(3)) for(i=0; i<4; i=i+1) WRITE(i, 0);
      code1 = READ(0);
      code2 = READ(1);
      code3 = READ(2);
    }

// check openings
    for(i=0; i<4; i=i+1){
      if(!GTPIND(OP_pins[i]) && alarm_set_flag){
        STPIND(OPIgreen_pins[i], LOW);
        STPIND(OPIred_pins[i], HIGH);
        if(!exit_counter_flag && !entry_counter_flag){
          if(!silent_flag)
```

```
            FREQOUT(SP_pin, 1000, 500, 250);
          if(!intrusion_flag) intrusion_flag=TRUE;
        }
      }
    }

// get keypad inputs
    key = 12; // reset key variable
    for(i1=1; i1<5; i1=i1+i1){ // set up row
      STPIND(row_pins[i1-1], HIGH);
      for(ii=1; ii<4; ii=ii+ii){ // listen for columns
        key_event = GTPIND(column_pins[ii-1]);
        if(key_event){
          if(i1==1 && ii==1) key=1;
          if(i1==1 && ii==2) key=2;
          if(i1==1 && ii==3) key=3;
          if(i1==2 && ii==1) key=4;
          if(i1==2 && ii==2) key=5;
          if(i1==2 && ii==3) key=6;
          if(i1==3 && ii==1) key=7;
          if(i1==3 && ii==2) key=8;
          if(i1==3 && ii==3) key=9;
          if(i1==4 && ii==1) key=10;
          if(i1==4 && ii==2) key=0;
          if(i1==4 && ii==3) key=11;
        }
      }
      STPIND(row_pins[i1-1], LOW);
// increment correct code counter
// reset the counter if any incorrect key is pressed
      if(key==0) code_counter=code_counter+1;
      else if (key>0 && key<12) code_counter=0;

// activate system when code is initially set
      if(code_counter==3 && first_set==0){
        first_set=1;
        code_counter=0;
        STPIND(MONred_pin, LOW);
        STPIND(MONgreen_pin, HIGH);
        PAUSE(1000);
        STPIND(MONred_pin, HIGH);
        STPIND(MONgreen_pin, LOW);
        alarm_set_flag=TRUE;
      }
```

```
// set code flag only after the alarm is set
      if(code_counter==3 && alarm_set_flag){
        code_flag=TRUE;
        code_counter=0;
      }

// set monitor LED when a correct code is entered
      if(code_flag){
        STPIND(MONred_pin, LOW);
        STPIND(MONgreen_pin, HIGH);
      }

      PAUSE(timebase);
    }

// get reset input
    if(!GTPIND(RST_pin) && code_flag) RST_flag = TRUE;

// get silent input
    if(!GTPIND(silent_pin) && code_flag)
      silent_flag = TRUE;
    else silent_flag = FALSE;

// advance system time
    PAUSE(timebase);
    if(code_counter_flag)
      code_counter = code_counter + 1;
    if(code_counter > sec15d){
      code_counter = 0;
      code_counter_flag = FALSE;
    }
    if(exit_counter_flag)
      exit_counter = exit_counter + 1;
    if(exit_counter > sec60d){
      exit_counter = 0;
      exit_counter_flag = FALSE;
    }
    if(entry_counter_flag)
      entry_counter = entry_counter + 1;
    if(entry_counter > sec60d){
      entry_counter = 0;
      entry_counter_flag = FALSE;
    }
  }
}
```

Project 6: A First Console I/O Program

In this project, we will write a simple PC console I/O program for the C Stamp. We will use the DEBUG pane in the Quick Programmer software as a console. We can also use the standard HyperTerminal application of the Windows Operating System for this, and those figures are shown along with those where the DEBUG pane is used. In essence, the PC screen and keyboard will become I/O devices of the C Stamp, and the C Stamp will be the CPU of the overall system. The concept of using the DEBUG pane of the Quick Programmer or the HyperTerminal can be useful to get information in and out of the C Stamp for any actual application or for debugging a program in development. For this activity, we will be using the C Stamp and the μC101 BOL. Upon program START, the C Stamp will output the following:

```
My First C Stamp I/O Console Program
C Stamp Programmer
Press Enter key to continue...
```

After the user presses any key, the program will display:

```
Program completion succeeded
```

At this point, the program will end.

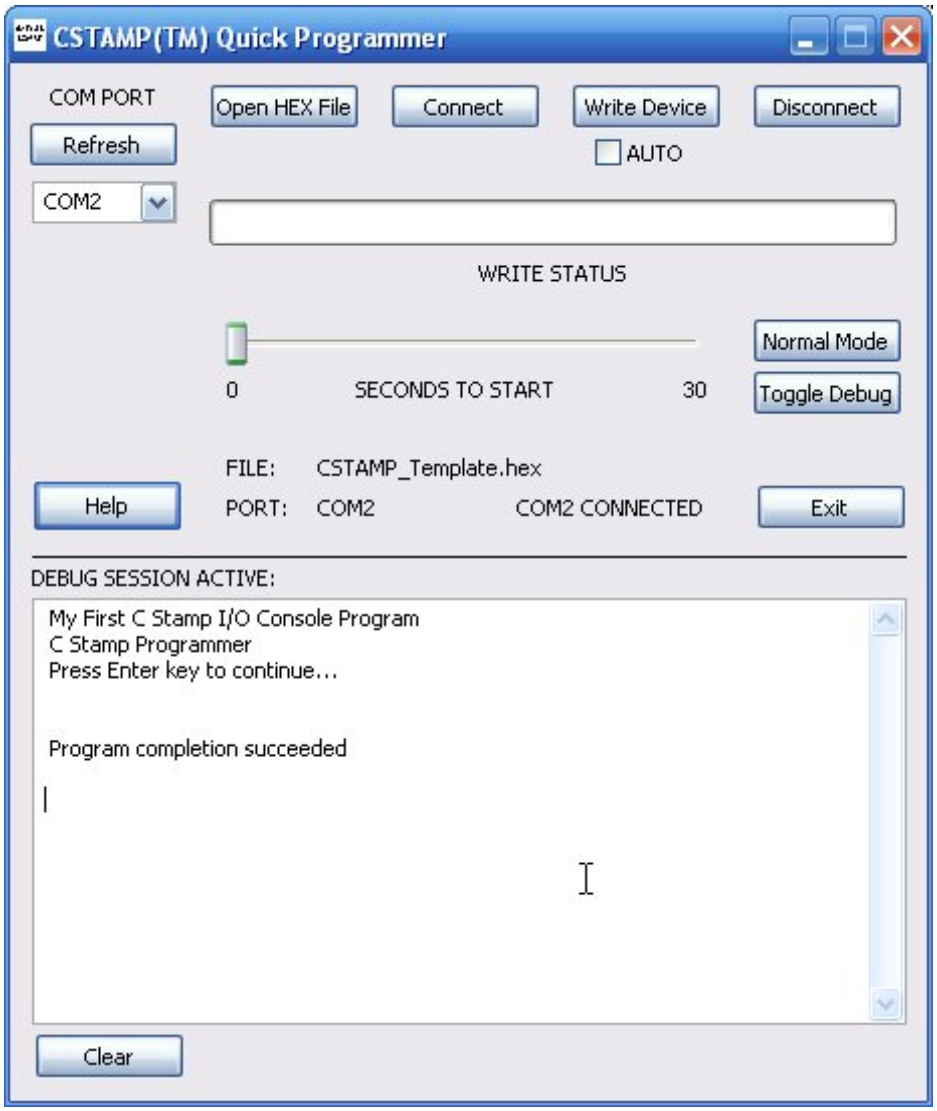
Project 6: Solution

The code below is our solution for this project, and the comments embedded within the code offer more clarifications.

```
#include  "CS110000.h"

void main(void)
{
// define dummy key
  BYTE key[1];

// start displaying
  DEBUG(" My First C Stamp I/O Console Program\r\n");
// You can put your name, instead of C Stamp Programmer
// in the line below
  DEBUG(" C Stamp Programmer\r\n");
  DEBUG(" Press Enter key to continue... \r\n");

// wait for key
  DEBUGIN("%s", key);
```

```
// print good bye message
  DEBUG(" Program completion succeeded\r\n\r\n");

  END();
}
```

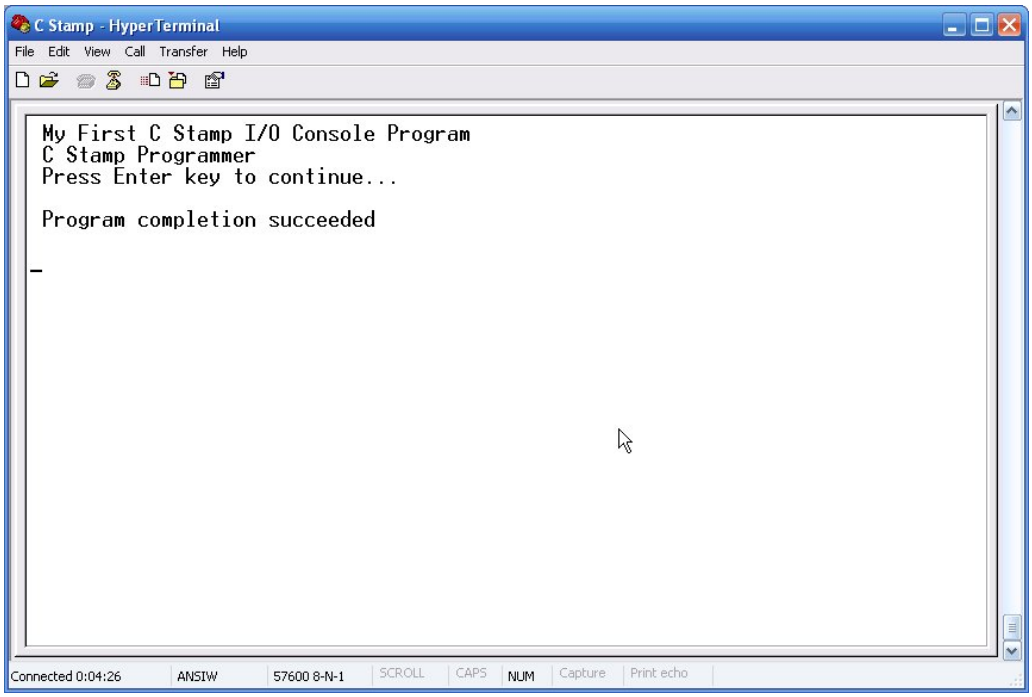
In the program above, we simply use the **DEBUG** command to put information, and to wait for the Enter key. The figure below shows the results with the Programmer after you activate the debug session by clicking on the "Toggle Debug" button.



To use the **DEBUG** command with the HyperTerminal, the communication speed of the PC HyperTerm must be 57,600. To run this program, first bring up a

HyperTerminal session by going to your PC "Start" menu → "All Programs" → "Accessories" → "Communications", and running "HyperTerminal". You can name this "HyperTerminal" session "C Stamp". Then choose the COM port that you are using to connect with your BOL from the drop-down list nest to the "Connect using:" prompt, and set up the "Port Settings" to be the same that your C Stamp program is using (i.e. 57,600 Bits per seconds, 8 Data bits, No Parity, 1 Stop bit, and No Flow control). The C Stamp always uses one Stop bit during its serial communications. Then click on the "OK" button.

Assuming that the program has been downloaded to your C Stamp, and that the powered up BOL and C Stamp combination is connected to your PC, START and run your program. After you have run your program, the "HyperTerminal" window should look something like the figure below.

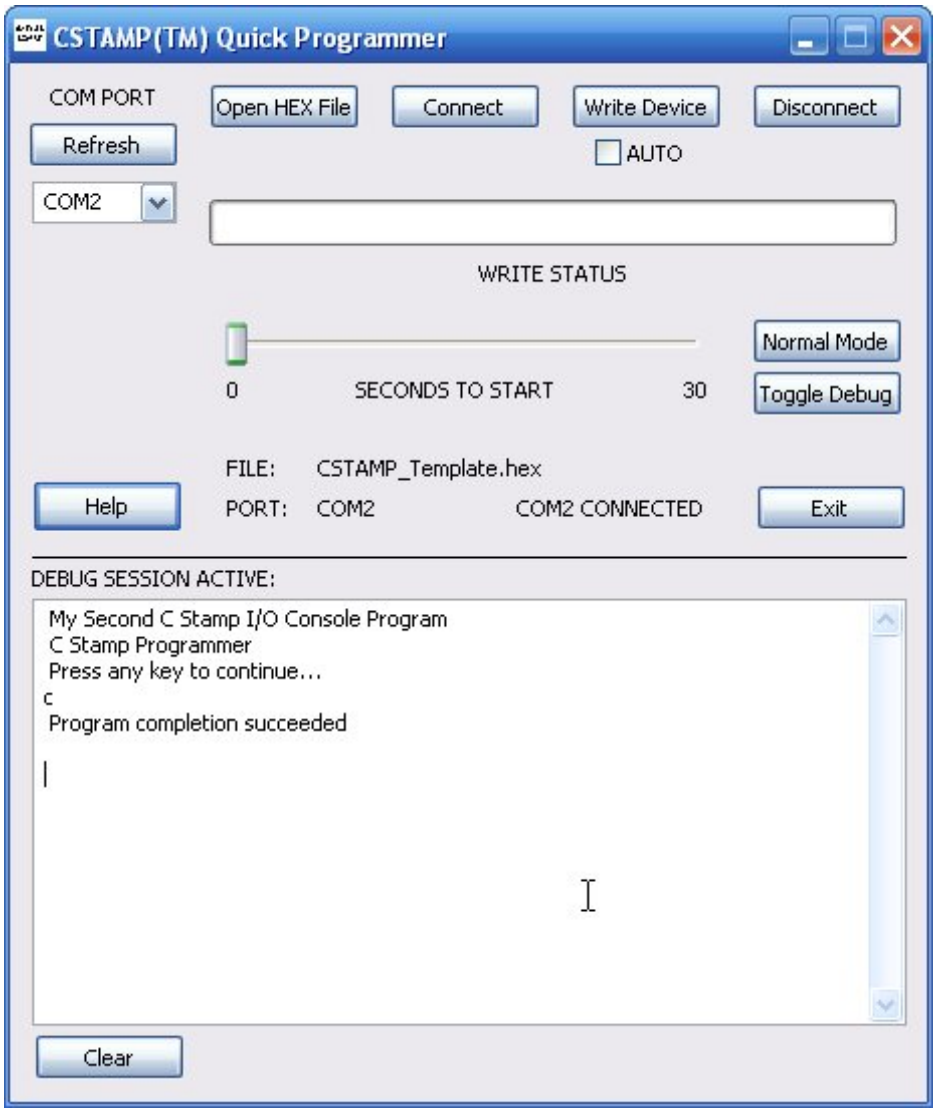


Instead of the Enter key, you could just accept any key. This is accomplished by using the **SERIN** command, instead of the **DEBUGIN** command. The code would change to the one shown below. The lines that change are underlined.

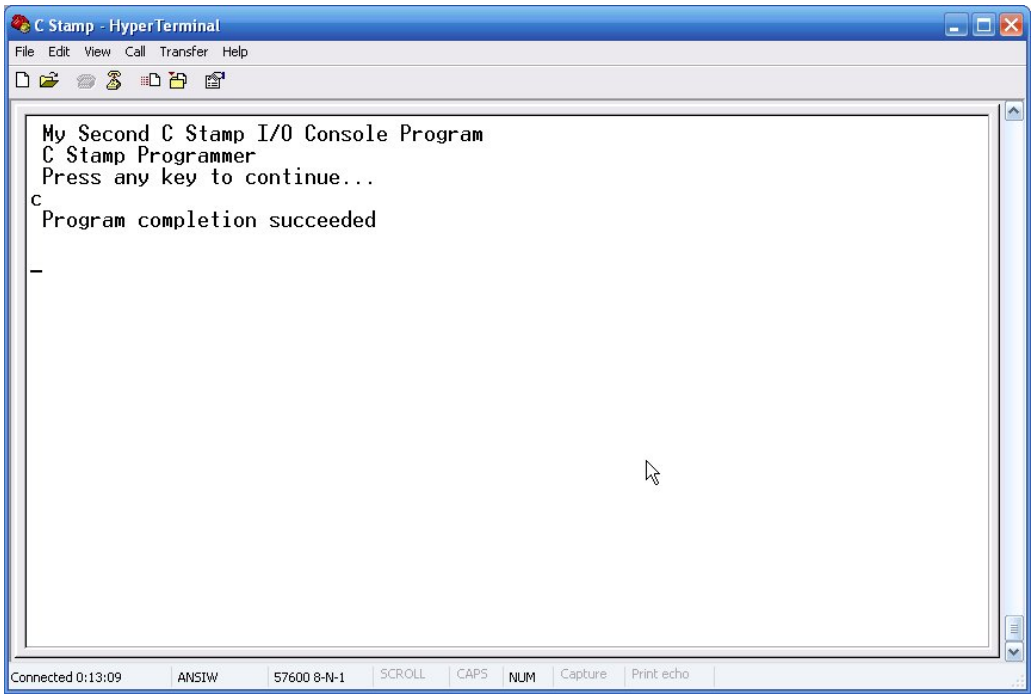
```
#include   "CS110000.h"

void main(void)
{
// define dummy key
```

```
  BYTE key[1];

// start displaying
  DEBUG(" My Second C Stamp I/O Console Program\r\n");
// You can put your name, instead of C Stamp Programmer
// in the line below
  DEBUG(" C Stamp Programmer\r\n");
  DEBUG(" Press any key to continue...\r\n");

// wait for key
  SERIN(0, 0, 57.6, 8, 0, 0, key, 1, 0);

// print good bye message
  DEBUG("\n\r Program completion succeeded\r\n\r\n");

  END();
}
```

If you are using the Programmer for a console, after you run this program, your Programmer screen will look like the figure below.



If you are using the HyperTerm, after you run this program, your HyperTerm screen will look like the figure below.



Project 7: Asynchronous Serial Communication

In this project, we will design a user interface for the C Stamp using asynchronous serial communication with the PC. We will use the standard HyperTerminal application of the Windows Operating System to communicate with the C Stamp. In essence, the PC screen and keyboard will become I/O devices of the C Stamp, and the C Stamp will be the CPU of the overall system. The concept of using the HyperTerminal can be useful to get information in and out of the C Stamp for any actual application or for debugging a program in development. For this activity, we will be using the C Stamp and the μC101 BOL. The system has the following specifications.

1. Upon program START, the system will turn on the LED's of the BOL.
2. The system will display a greeting message.
3. The system will display a prompting message asking the user to enter his or her name.
4. The system will capture the name entered by the user.
5. The system will display a greeting message using the name entered by the user, and tell the user to press one of the number keys (1 through 8) to toggle to

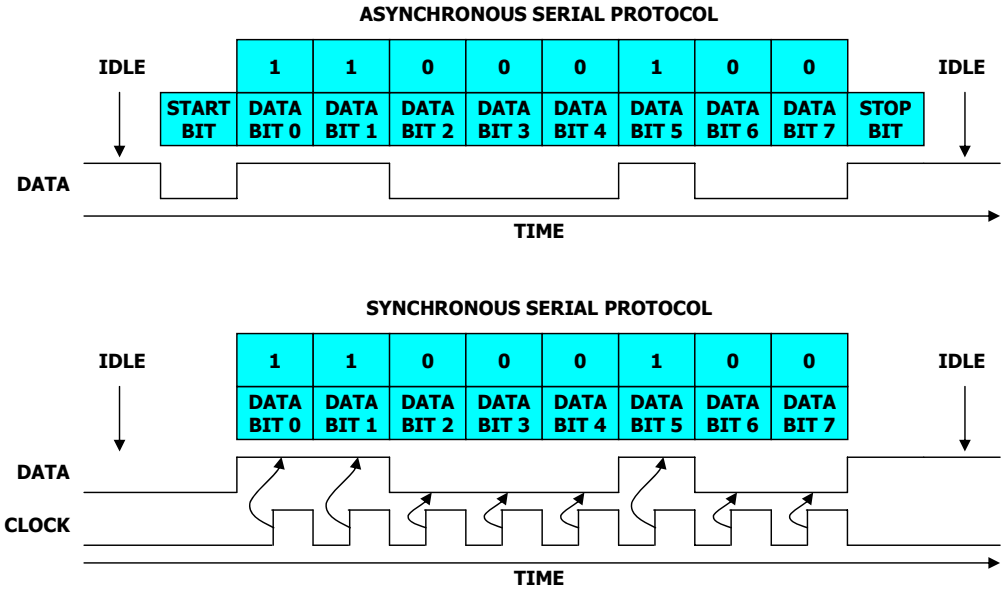
state of one of the LED's. Key 1 corresponds to toggling LED1, key 2 corresponds to toggling LED2, etc.

6. The system will monitor user input for a valid key to be entered, and will toggle the appropriate LED. If a non valid key is entered, the system will simply ignore that entry.

7. The system will display a message in the HyperTerminal telling the user which LED was toggled last.

8. After the user enters the first toggling key directive, the program will just continuously monitor user input to continue toggling LED's and updating the message that informs the user which LED was toggled last. The program will do this until the system is restarted or RESET.

In the dichotomy of digital data communications, ways of transmitting and receiving data can be divided into two main groups: parallel and serial communications, although hybrid systems exist. Parallel communication protocols transmit and receive digital data in multiple bits at a time, while serial communication protocols do the same one bit at a time. Furthermore, serial communications can, in turn, be sub-divided into two modes: synchronous and asynchronous. Synchronous protocols use an additional clock line to synchronize the transmitter and the receiver when data is valid. While synchronous communications are simpler to implement and at a given speed can achieve higher data throughputs than asynchronous communications, this modality reaches a speed ceiling that is much lower than asynchronous communications. The reason for this is that as speeds go up synchronizing the transmitter and the receiver proves to be a challenging task due to physical travel distances of electrical signals, system noise, and signal degradation. Asynchronous communications does not use a synchronizing clock line; instead, it relies on handshake signaling to establish transmitter-receiver communications. While this imposes complexity and overhead in these types of protocols, it allows the speeds to be much higher. This is the reason that the interface from a disk drive to the mother board of PC's used to be synchronous (ATA protocol), but as the data transfer rates of disk drives have increased, PC manufactures have started to use asynchronous serial interfaces to connect disk drives to the motherboard. (SATA protocol – Serial ATA protocol). In this project, we will employ a very widely used asynchronous communication protocol to interface the C Stamp and the PC. This is commonly known as Serial or RS-232. There are many other asynchronous serial protocols such as RS-485, USB, and IEEE 1394. The figure below shows our RS-232 serial protocol and also a synchronous communication protocol.



The synchronous protocol above is similar to that used by the **SHIFTIN** and **SHIFTOUT** functions provided by the C Stamp software system. It can be seen that it takes 8 bits to communicate 8 bits of DATA; thus the efficiency of the protocol is 100% or we can say that the protocol has 0 overhead. The synchronous protocol requires a CLOCK line to signal when the DATA is valid. This is on an edge of the CLOCK; in this case the DATA is sampled on the rising edge of the CLOCK. The simplest form of the RS-232 protocol is also shown above. It requires a START bit to signal that communication is going to start, it has 8 DATA bits, although in some modes the number of DATA bits can be 9, and it has a STOP bit to signal that communication is stopping for that DATA word. Some modes of the protocol allow for multiple STOP bits to separate the communicated data words. We can see that in the form of the RS-232 protocol depicted above, it takes us 10 bits to communicate 8 bits of DATA. Thus the efficiency of the protocol is 8/10 = 0.8 or 80%. We can also say that the protocol has 10/8 - 1 = 0.25 or 25% overhead. This means that is going to take us 25% more bits (and time) to communicate any number of bits. That is why at a given speed, the synchronous protocol is faster, but an asynchronous protocol can run at a faster rate than a synchronous one, because we do not have to be concerned about synchronizing the DATA and CLOCK lines. The efficiency and overhead of the RS-232 protocol depends on the number of DATA bits and STOP bits used for any given mode of the protocol.

We have seen that, because of signaling overhead, in asynchronous communications that data transfer rate, the data throughput, is less than the speed of the bus. The speed of the bus is termed "baud rate", and it refers to how many raw bits (START, DATA, and STOP bits) are communicated per second. When establishing RS-232

communications between two entities, the settings for the baud rate, the number of DATA bits, and the number of STOP bits must match. This is such that the receiving entity samples and interprets the data correctly.

Other optional settings used in RS-232 communications are termed "parity" and "flow control". Parity is an extra bit that is transmitted to help the receiver determine whether there was a single bit error during transmission due to electrical noise, etc. Parity can be "even" or "odd". For even parity, if the number of 1's in the data word is odd, the parity bit is set to a 1 to make the total number of bits (DATA + PARITY) even; if the number of data bits is even, then the parity bit is set to a 0. For odd parity, the parity bit is set to a 1 if the numbers of 1's in the data word is even to make the total number of bits odd, and it is set to a 0 if the number of ones in the data word is odd to keep the total number of bits odd. If flow control is used, an extra communication line is employed for the receiver to signal the transmitter that is ready to receive data. Both the parity and the flow control settings must also match between transmitter and receiver for successful RS-232 communications. For example, the receiver must know what type of parity, if parity is being used, is being sent by the transmitter to check for one bit errors. More than one bit errors cannot be detected by the parity scheme, but the probability of multiple bit errors is much smaller than of a single bit error. In fact the probability of single bit errors is very small in normal RS-232 communications, and in many instances parity is not used. If Parity was used there must exist a mechanism at the higher levels of the application for the receiver to communicate to the transmitter that there was a communication error, and that data should be retransmitted.

In the both cases of the figure above, the 8-bit binary data being transmitted is "0 0 1 0 0 0 1 1". Bits in digital systems have no inherent meaning. This meaning comes from the users or the application. For example, these bits could be the integer number 35 if we interpret them as numerical data or the character '#' if we interpret these bits as ASCII code. ASCII stands for American Standard Code for Information Interchange, and it is an agreed upon code to represent characters. The table below shows a partial set of ASCII codes for the characters. Codes for special control functions and an Extended ASCII set also exist. In the table below, the first column represents the first four bits in hexadecimal, and the first row represents the least significant four bits in hexadecimal as well. Code 0x20 or in binary "00100000" represents a SPACE character, code 0x7F is a DELETE character, and code 0x4B a 'K'. Note how the upper case letters have smaller numeric codes than the lower case ones, and that the alphabet runs numerically from lower value codes to higher values. The people that devised the code were ingenious. The arranged the letters in this manner, so that words could be alphabetized in the memory of a digital system by comparing the ASCII codes numerically. Also note that each lower case letters are exactly 0x20 numerically ahead of their upper case counterparts. This allows converting to characters to lower case or upper case by adding or subtracting 0x20 to their ASCII code. Having numerical algorithms for alphabetizing words and changing the case of characters is much more convenient and efficient than using look up table, which is what would

have to be done if the codes were not numerically related. The numbers are also conveniently assigned codes of 0x30 through 0x39, so we can get the numeric value of a number character by simply subtracting 0x30 to the ASCII code.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	←

Project 7: Solution

With the background covered in the previous discussion, we are now ready to present our solution to this project. For the communications settings we will use a 4800 baud rate, 8 data bits, no parity, and no flow control. The code below is our solution for this project, and the comments embedded within the code offer more clarifications.

```
#include  "CS110000.h"

void main(void)
{
// define the connections of LED1-8
  BYTE LED_pins[] = {46, 45, 44, 43, 42, 41, 40, 39};
// greeting message and length
  RAM BYTE greeting[] =
    "\fHello from your C Stamp!\r\n\r\n";
  BYTE greeting_l = 29;
// prompting message and length
  RAM BYTE prompt[] = "Please enter your name: ";
  BYTE prompt_l = 24;
// second greeting lines and lengths
  RAM BYTE greeting2[] = "\r\n\r\nWelcome ";
  BYTE greeting2_l = 12;
  RAM BYTE greeting3[] =
    ".\r\n\r\nPlease hit a number key (1 to 8) to
toggle an LED.\r\n\r\n";
  BYTE greeting3_l = 59;
// info lines and length
  RAM BYTE info[] = "LED last toggled: ";
  BYTE info_l = 18;
// return character
  RAM BYTE returnch[] = "\r";
```

```
// delete characters
  RAM BYTE delchs1[] = " \b";
  RAM BYTE delchs2[] = "\b \b";
// name can be up to 50 characters
  RAM BYTE namech[1];
  RAM BYTE name[50];
  BYTE name_l;

// declare other necessary variables
  NIBBLE i;
  BYTE LED[1]; // LED to toggle
  float baudr = 4.8; // baud rate

// turn on all LEDs
  for(i=0; i<8; i=i+1) STPIND(LED_pins[i], HIGH);

// display first greeting and prompt for name
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting,
         greeting_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, prompt, prompt_l);

// get name
  name_l = 0;
  do{
    SERIN(0, 0, baudr, 8, 0, 0, namech, 1, 0);
    if(namech[0] == '\r' || namech[0] == '\n') break;
    else{
      if(namech[0] == '\b'){
        SEROUT(0, 0, baudr, 0, 8, 0, 0, delchs1, 2);
        if(name_l > 0) name_l = name_l - 1;
      }
      else{
        name[name_l] = namech[0];
        name_l = name_l + 1;
      }
    }
  }while(TRUE);

// display second greeting lines
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting2,
         greeting2_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, name, name_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting3,
         greeting3_l);
```

```
  while(TRUE){ // infinite loop
// get LED to toggle
    SERIN(0, 0, baudr, 8, 0, 0, LED, 1, 0);
    SEROUT(0, 0, baudr, 0, 8, 0, 0, delchs2, 3);
    if(LED[0] > 0x30 && LED[0] < 0x39){
      TOGGLE(LED_pins[LED[0] - 0x31]);
      SEROUT(0, 0, baudr, 0, 8, 0, 0, returnch, 1);
      SEROUT(0, 0, baudr, 0, 8, 0, 0, info, info_l);
      SEROUT(0, 0, baudr, 0, 8, 0, 0, LED, 1);
    }
  }
}
```

The program above works as follows. We use the **SERIN** and **SEROUT** functions with the parameters that we have discussed to handle in input/output functionality of our C Stamp with the terminal. We have forced the arrays that contain character strings to RAM, since they contain a lot of data. It is a good idea to always do this for string array variables. After we define our various character string greetings, their lengths, and other necessary variables, we simply proceed to turn all the LEDS on, and display the first greeting line and the prompt for the name. Note that throughout this programs when we have defined character strings, we have used various characters proceeded by a back-slash (i.e. \). In WC, these back-slash characters represent a single character or BYTE, and they have certain special meaning or control function. All the back-slash characters used in WC are shown in the table below.

Back-Slash Character or Code	***Meaning or Control Function***
\b	Backspace (Moves the cursor back one space in the line)
\f	Clear Screen or Form Feed
\n	New Line
\r	Carriage Return (Moves cursor to beginning of the line)
\t	Horizontal Tab (Usually 4 or 8 spaces)
\"	Double Quote (So it is displayed, and not confused with the double quotes enclosing the definition of a string)
\'	Single Quote (So it is displayed, and not confused with the single quotes

Back-Slash Character or Code	*Meaning or Control Function*
	enclosing the definition of a character)
`\0`	Null Character
`\\`	Back-Slash (So it is displayed, and not confused with the back-slash used to define these codes)
`\v`	Vertical Tab (Usually 4 or 8 lines)
`\a`	Alert or Bell (Sounds the terminal bell)
`\N`	Octal Constant (Where N is an octal constant to directly put ASCII codes into a string being defined)
`\xN`	Hexadecimal Constant (Where N is a hexadecimal constant to directly put ASCII codes into a string being defined)

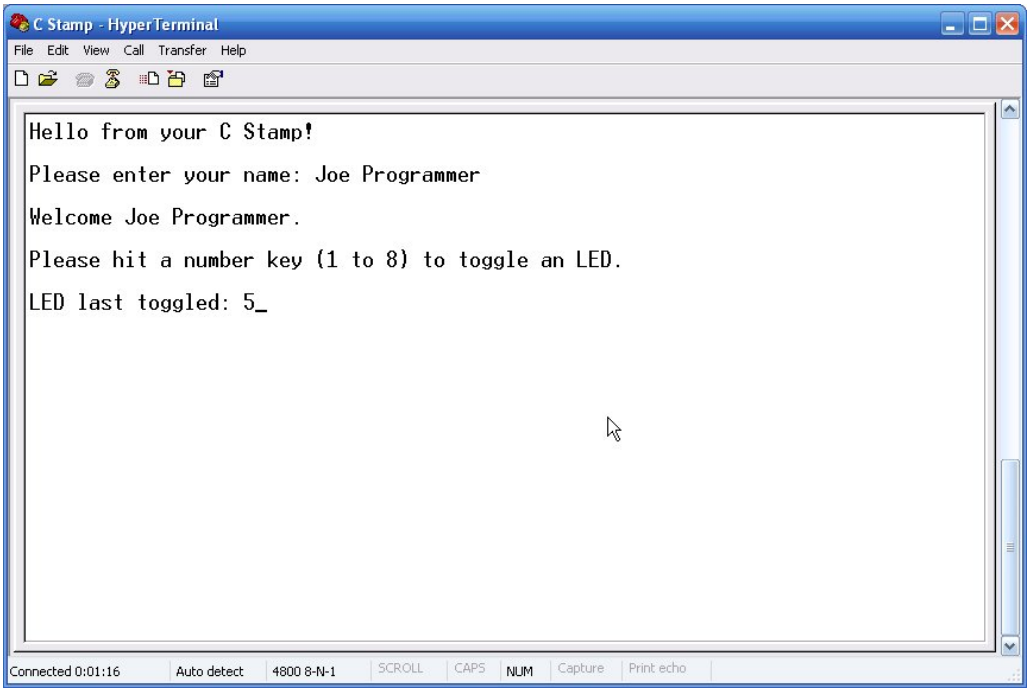
The name is collected in the **`do-while`** loop by storing the characters sent by the user in the string **`"name"`** until the user hits the "Enter" key. Note that the terminal is a "dumb" one. It can send the C Stamp the "back-space" character, but it cannot erase. Our program is performing that function, by detecting a "back-space". If one is detected, our program writes a "space" at that position and then moves back again. At this low level of programming, we have a great deal of control, but also a great deal of responsibility. We have to care of the user by letting the user have the illusion that a character has been erased. The sequence that accomplishes this is **`"delchs1"`**. When we detect a "back-space", we also rewind the index of where the characters received go in the string **`"name"`**, so that those do not become part of the name collected.

After we have the name, then we can display the second greeting, and ask the user to enter a number to toggle an LED. We monitor for this number continuously, and toggle the appropriate LED accordingly. When the user types a number, we immediately erase it so that it is not displayed. We use a sequence similar to **`"delchs1"`**. This sequence is **`"delchs2"`**. Since this sequence is sent to the terminal without the user inputting the first "back-space", it has an extra one at the beginning when compared with the used before. After the user presses the first number key, we display the **`"info"`** message with what LED was just toggled, and keep repeating the sequence of getting a number, erasing it, toggling the appropriate

LED, and displaying the `"info"` message. Note that to know which pin needs to be toggled; we simply subtract 0x31 from the ASCII code to get a number from 0 to 7 that corresponds to the index of the LED pins in the `"LED_pins"` array. Also if we get a key that is not a number from 1 to 8 (e.g. ASCII code has to be from 0x31 to 0x38), we simply ignore it and do nothing. Finally, to keep displaying the `"info"` message in the same line, we use the `"returnch"` string to move to the beginning of the line.

To run this program, first bring up a HyperTerminal session by going to your PC "Start" menu → "All Programs" → "Accessories" → "Communications", and running "HyperTerminal". You can name this "HyperTerminal" session "C Stamp". Then choose the COM port that you are using to connect with your BOL from the drop-down list nest to the "Connect using:" prompt, and set up the "Port Settings" to be the same that your C Stamp program is using (i.e. 4800 Bits per seconds, 8 Data bits, No Parity, 1 Stop bit, and No Flow control). The C Stamp always uses one Stop bit during its serial communications. Then click on the "OK" button.

Assuming that the program has been downloaded to your C Stamp, and that the powered up BOL and C Stamp combination is connected to your PC, START and run your program. After you have run your program, the "HyperTerminal" window should look something like the figure below.



Project 8: Using an Optical Switch to Build a Hand-Eye Coordination Meter

In this project, we will build a Hand-Eye Coordination Meter. For this activity, we will be using the C Stamp, the μC101 BOL, Optical Switches, and a user interface using the HyperTerminal. The Optical Switch concepts introduced in this project can be applied to the construction of an optical encoder to measure the speed of a wheel in vehicles, assert distance traveled, etc. The principles of light emission and detection are also the same used in optical communications, where signals are transmitted across great distances using light beams trapped in optical fibers. Light transmission tends to be faster and less subject to deterioration than when using electrical signals; although it is more expensive to implement for short distances like in printed circuit boards (PCBs). As we will see, light emitters and receiver have to be properly set up for operation on a signal-by-signal basis. The system in this project has the following specifications.

1. Upon program START, the system will display a greeting message.

2. The system will display a prompting message asking the user to enter his or her name.

3. The system will capture the name entered by the user.

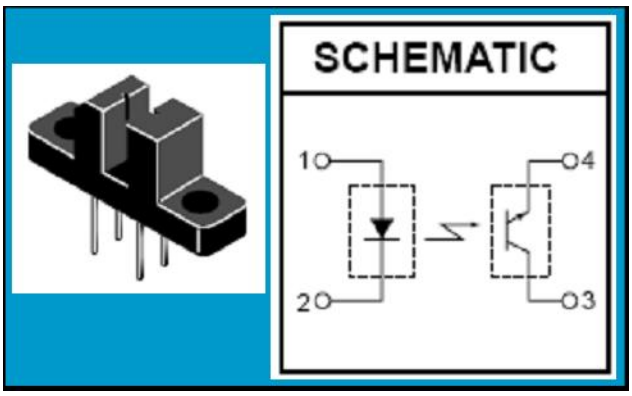
4. The system will display a greeting message using the name entered by the user, and tell users to insert a piece of light cardboard, such as a business card, in the left optical switch and then insert the piece of cardboard in the right optical switch as fast as they can.

5. The system will count the milliseconds that it takes the user to do the exercise. From the time that the user takes the cardboard out of the first switch to the time that the cardboard is inserted in the second switch.

6. Then the system will display the time of flight that users take to move their hand from one optical switch to the other in seconds, and the speed of travel in m/s.

In this project, most notably, we will be using a Phototransistor Optical Interrupter Switch, such as the A-WIT Part # CS440000. We will also reuse the knowledge gained in previous activities. The CS440000 is a gallium arsenide semiconductor infrared emitting diode coupled with a silicon phototransistor in a plastic housing. The packaging system is designed to optimize the mechanical resolution, coupling efficiency, ambient light rejection, cost and reliability. The gap in the housing provides a means of interrupting the signal with an opaque material, switching the output from an "ON" to an "OFF" state. The package form factor can be used in breadboards or printed circuit boards. The device is compatible with the C Stamp microcomputer's supplies and signal levels. Reading the status of the switch is made easy with A-WIT's

supplied software commands GTPIND, PULSIN, and COUNT. The figure below shows the optical switch, as well as its schematic.

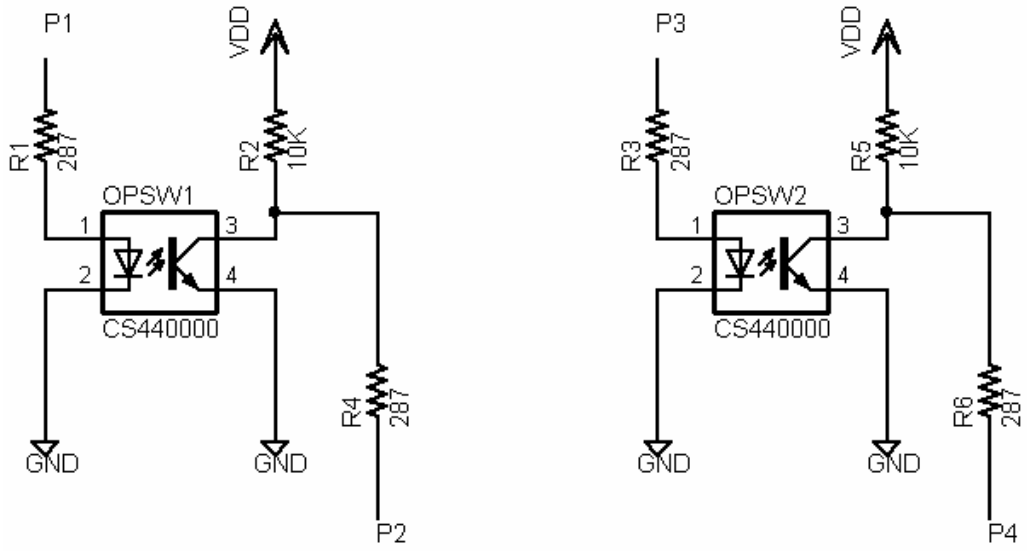


Looking at the top of the casing (opposite to the pins) of the switch, Pin 1 is denoted with a '+' on top of the 'E', Pin 2 is the 'E', Pin 3 is the '+' below the 'D', and Pin 4 is the 'D'. The principle of operation of the LED housed in this switch is not much different that the LEDs that we have seen in previous projects. The most notable difference is that the light emitted by this diode is not in the visible spectrum; it is infrared. As we had done before, this LED will have to be properly biased with a current limiting resistor. From the Data Sheet of the switch, we know that the forward voltage V_F of the LED is 1.7 V, and that a forward current appropriate for operation is I_F = 10 mA. As done before, with these values we can calculate the value of the limiting resistor.

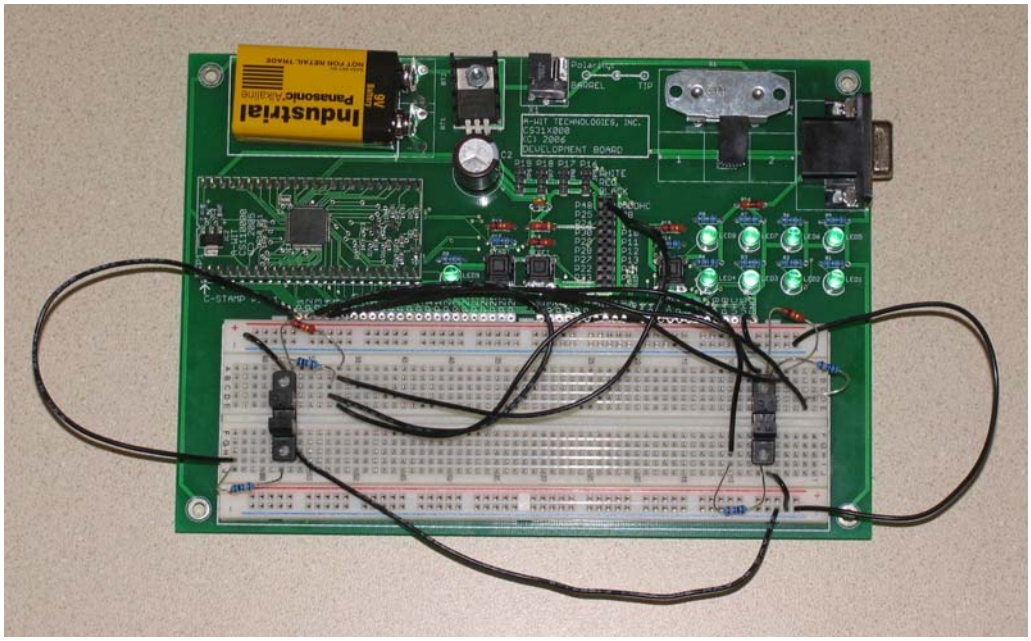
When the LED is emitting infrared light, this light beam activates the phototransistor. The phototransistor acts as a light controlled switch. When light is shone on the "Base" of the transistor, current is conducted from the "Collector" (Pin 3) to the "Emitter" (Pin 4). Therefore, we can bias the transistor part of the switch with the same type of resistor divider network that we have used previously to interface mechanical switches.

Project 8: Solution

The schematic for the interface of the optical switches in our system is shown in the following figure. Also, for the communications settings we will use the same ones that we used to interface with the PC HyperTerminal in the previous activity (i.e. 4800 baud rate, 8 data bits, no parity, and no flow control).



The wired circuit in the μC101 BOL is shown in the following figure, and the code below is our solution for this project. The comments embedded within the code offer more clarifications.



```
#include   "CS110000.h"
```

```
void main(void)
{
// define the connections of LEDs
  BYTE LED1 = 1;
  BYTE LED2 = 3;
// define the connections of transistors
  BYTE TRX1 = 2;
  BYTE TRX2 = 4;

// greeting message and length
  RAM BYTE greeting[] =
    "\fHello from your C Stamp!\r\n\r\n";
  BYTE greeting_l = 29;
// prompting message and length
  RAM BYTE prompt[] = "Please enter your name: ";
  BYTE prompt_l = 24;
// second greeting lines and lengths
  RAM BYTE greeting2[] = "\r\n\r\nWelcome ";
  BYTE greeting2_l = 12;
  RAM BYTE greeting3[] =
    ".\r\n\r\nPlease insert a piece of cardboard in the
left optical switch,\r\n";
  BYTE greeting3_l = 69;
  RAM BYTE greeting4[] =
    " and then in the right as fast as you
can.\r\n\r\n";
  BYTE greeting4_l = 46;
// info lines and length
  RAM BYTE info1[] = "Time of flight (in s): ";
  BYTE info1_l = 23;
  RAM BYTE info2[] = "Speed of hand (in m/s): ";
  BYTE info2_l = 24;
// new line, carriage return characters
  RAM BYTE nlcrch[] = "\n\r";
// delete characters
  RAM BYTE delchs1[] = " \b";
  RAM BYTE delchs2[] = "\b \b";
// name can be up to 50 characters
  RAM BYTE ch[1];
  RAM BYTE name[50];
  BYTE name_l;

// declare other necessary variables
  BYTE i;
```

```
  int t = 0;
  float tf;
  RAM BYTE tfs[50];
  float distance = 0.1651; // meters (6.5 inches)
  float speed;
  RAM BYTE speeds[50];
  float baudr = 4.8; // baud rate

// turn on all LEDs
  STPIND(LED1, HIGH);
  STPIND(LED2, HIGH);

// display first greeting and prompt for name
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting,
         greeting_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, prompt, prompt_l);

// get name
  name_l = 0;
  do{
    SERIN(0, 0, baudr, 8, 0, 0, ch, 1, 0);
    if(ch[0] == '\r' || ch[0] == '\n') break;
    else{
      if(ch[0] == '\b'){
        SEROUT(0, 0, baudr, 0, 8, 0, 0, delchs1, 2);
        if(name_l > 0) name_l = name_l - 1;
      }
      else{
        name[name_l] = ch[0];
        name_l = name_l + 1;
      }
    }
  }while(TRUE);

// display second greeting lines
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting2,
         greeting2_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, name, name_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting3,
         greeting3_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, greeting4,
         greeting4_l);

// wait for first switch to be activated
  while(TRUE){
```

```
    if(GTPIND(TRX1)){
      while(GTPIND(TRX1));
      break;
    }
  }

// wait for second switch to be activated
  while(TRUE){
    PAUSE(1);
    t = t + 1;
    if(GTPIND(TRX2)) break;
  }

// calculate time of flight
  tf = t;
  tf = tf/1000;

// convert tf to a string
  i = WCftoa(tf, tfs);

// display time of flight
  SEROUT(0, 0, baudr, 0, 8, 0, 0, info1, info1_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, tfs, i);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, nlcrch, 2);

// calculate speed of hand
  speed = distance/tf;

// convert speed to a string
  i = WCftoa(speed, speeds);

// display speed of hand
  SEROUT(0, 0, baudr, 0, 8, 0, 0, info2, info2_l);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, speeds, i);
  SEROUT(0, 0, baudr, 0, 8, 0, 0, nlcrch, 2);

  END();
}
```

The program above works as follows. First, we define the pin connections of the driving LEDs and transistors of the optical switches. Then all the different text messages that will be sent to the console are defined, as are defined their respective character lengths. Other necessary variables are also defined, such as a generic counter variable `i`, the time counter `t`, its floating point equivalent `tf`, and its ASCII representation `tfs`. We also define the `distance` between the two optical switches

to calculate the "Speed of hand" value. The `speed` and the ASCII version of this value `speeds` are also defined, so it can be displayed at the console. Finally, we also have a variable for the communication baud rate that we use `baudr`.

The actual operation of the program starts by turning on the two optical switches LEDs, and then displaying the greeting messages, and collecting the name of the user. After the user has been instructed to start the meter by putting the card in the first switch, the program waits for this action to take place. This means that the light beam in the first switch will be interrupted, and will remain interrupted until the card is taken out. The effect of interrupting the beam in a switch will be that the transistor will not conduct current, and a **`HIGH`** level will be read at the transistor pin. Since no current will flow through the transistor, all the voltage referenced to ground will be dropped across the transistor. After the card is retired from the first switch, the LED light will shine on the base of the transistor, and this will conduct, thus developing no voltage across it, and a **`LOW`** will be read at the transistor pin. When the card is taken out, the time count starts until the beam is interrupted at the second optical switch. Then the time count stops and we are ready to display the results for the user.

The first step in displaying the results is to convert the measured milliseconds to seconds. Then this value is converted to ASCII for displaying with the function **`WCftoa`**. Since we do not know how many characters the floating point number converted to ASCII will be, we have allocated up to 50 characters for the result floating numbers, but these numbers will be much shorter, and terminated with a null character '\0', and the length will be returned to the variable `i`. Then we display the value with the **`SEROUT`** function followed by a two character new line – carriage return combination. Finally to display the speed, we calculate it by dividing the distance between the optical switches in meters by the time in seconds, and repeat the ASCII conversion and display process. Then the program is ended. In this setup, to restart the program the BOL must be restarted.

Project 9: Interfacing a Servo

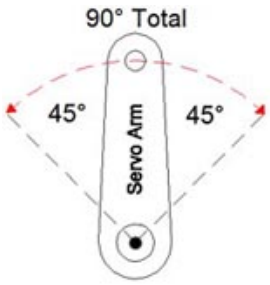
In this project, we will discuss how to interface a servo to the C Stamp μC101 BOL. The servo will simply move back and forth, as if it was waving. We will use the A-WIT Standard Servo (Part Number CS610000). The system in this project has the following specifications.

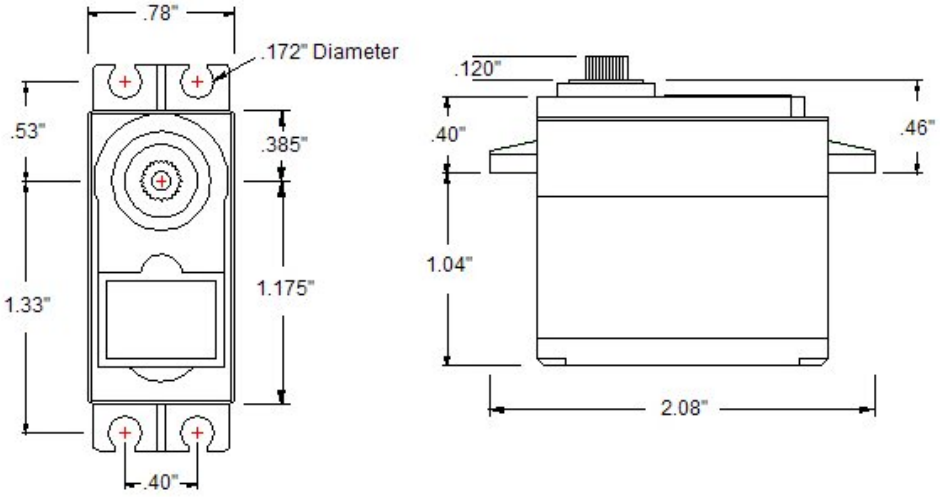
1. Upon program START, the system will control the servo to move it back and forth, as in a waving motion.

2. If the servo is moving, and a button is pushed, then the servo will stop.

3. If the servo is stopped, and the same button used above is pushed, the servo will resume motion.

In this project, most notably, we will be using the A-WIT Standard Servo (Part # CS610000). We will also reuse the knowledge gained in previous activities. The CS610000 servo provides high performance and reliability. Combined with precise resin gears and SMT circuitry, the CS610000 represents a remarkable value. One piece circuit board and tight mesh gears ensures great durability. It comes with a number of various servo horns and arms with mounting hardware. The servo is compatible with the C Stamp microcomputer's supplies and signal levels. Controlling the servo is made easy with A-WIT's supplied software command PULSOUT. This simple one command interface is all that is required to control the servo. The following figures show the servo, the associated hardware, the moving angle, and the dimensions.



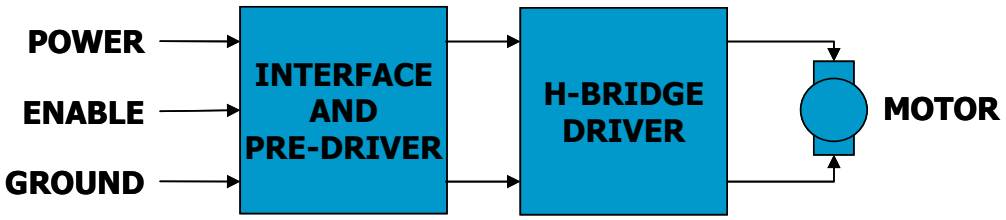


The A-WIT Standard Servo has the following Technical Specifications:

- Control System: +Pulse Width Control 1500 μS Neutral
- Required Pulse: 3-5 Volt Peak to Peak Square Wave
- Operating Voltage: 4.8-6.0 Volts
- Operating Temperature Range: -20 to +60 °C
- Operating Speed (4.8V): 0.19 S/60° at no load
- Operating Speed (6.0V): 0.15 S/60° at no load
- Stall Torque (4.8V): 42 oz-in (3.0 Kg-cm)

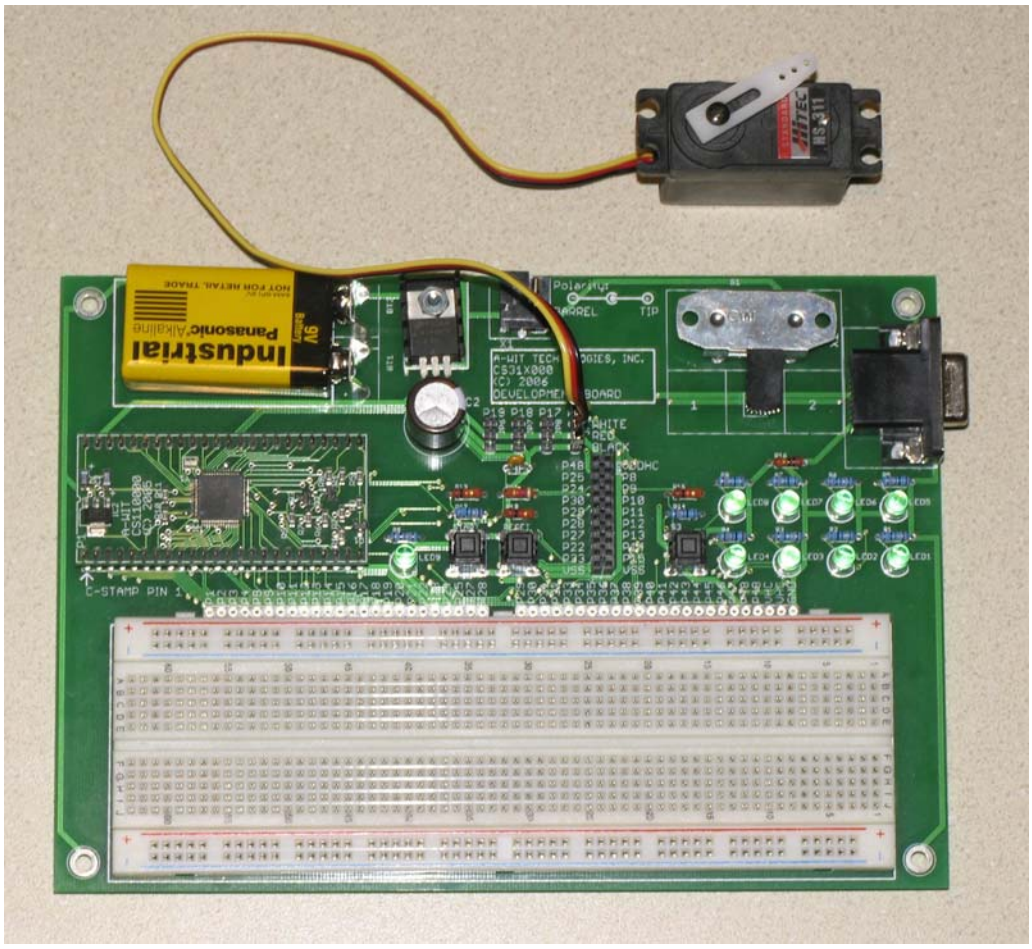
- Stall Torque (6.0V): 49 oz-in (4.5 Kg/cm)
- Current Drain (4.8V): 7.4 mA idle, 160mA no load operating
- Current Drain (6.0V): 7.7 mA idle, 180mA no load operating
- Dead Band Width: 5 μS
- Operating Angle: ±45°, 90° total, one side pulse traveling 450 μS
- Direction: Clockwise/Pulse Traveling 1500 to 1900 μS
- Motor Type: 3 Bole Ferrite, Cored Metal Brush
- Potentiometer Drive: 4 Slider/Direct Drive
- Bearing Type: Top/Resin/Nylon Bushing
- Gear Type: Nylon
- 360 Modifiable: Yes
- Connector Wire Length: 11.81" (300 mm)
- Size: 1.6 x 0.8 x 1.4" (40 x 20 x 37 mm)
- Weight: 1.51 oz (43 g)

A servo is different than a motor in that a servo contains a motor as part of its system, so a motor is a simpler system than a servo. The following figure shows a block diagram of a standard servo, such as the CS610000. The interface, beside the POWER and GROUND connections, is a simple enable command. The WC PULSOUT function can be used to enable the servo for a specified period of time and achieve a desired movement. Internal to the servo, there is a pre-driver circuit that generates the control signals for the motor. These signals are passed through an H-Bridge circuit that serves two functions. First, it provides current amplification to drive the windings of the motor, and second, it allows commutation of the motor current. A motor is a current device that generates rotary motion using the energy from a magnetic field that is developed from the current flowing through its windings. The commutation of the current through the motor is simply the ability to switch the direction of the current from one direction to the other to allow the motor to rotate in the two possible directions: clockwise and counterclockwise.



Project 9: Solution

Our circuit for the interfacing of the servo is simple. We connect the servo to the left most servo connector in the BOL (assuming you are facing the breadboard area), which is connected to Pin 16 of the C Stamp. We connect the Black servo wire to the Black labeled pin of the pin header connector, the Red wire to the Red pin, and the Yellow to the White pin. For the control button, we choose the utility button in the BOL, which is connected to Pin 37 of the C Stamp. The wired circuit in the μC101 BOL is shown in the following figure, and the code below is our solution for this project. The code essentially applies a series of pulses to the servo; either increasingly wider or increasingly narrower. Progressively widening the pulse width makes the servo move clockwise, and progressively narrowing the applied pulses make the servo move counter clockwise. When the trend of pulse width changes (progressively wider to progressively narrower or progressively narrower to progressively wider) changes, the servo stops, and reverses direction. The bounds of pulse widths control how much the servo will move in each direction. The wait time between pulses controls the speed of the servo. The comments embedded within the code offer more clarifications.




```
#include  "CS110000.h"

void main(void)
{
// Declare some necessary variables
  BIT button_pushed;
  BIT servo_on;
  BYTE Servo_pin = 16;
  WORD i, iH, iL;

// Initialize variables
  servo_on = TRUE; // Denotes whether the servo is on
// Controls how much the servo moves clockwise
  iH = 960;
// Controls how much the servo moves counter clockwise
  iL = 660;
```

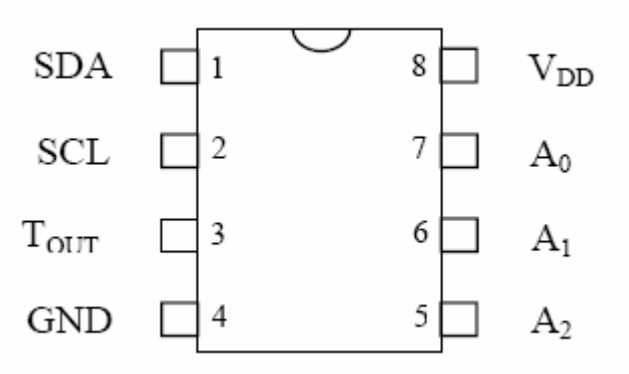
```
  STPIND(Servo_pin, LOW);
  PAUSE(50);

  while(TRUE){
    if(servo_on){
// This loop applies a series of pulses increasingly
// wider that make the servo move clockwise
      for(i = iL; i <= iH; i = i + 1){
        PULSOUT(Servo_pin, i, 1, 1);
// Controls the speed of the servo in the clockwise
// direction
        PAUSE(5);
      }
// This loop applies a series of pulses increasingly
// narrower that make the servo move counter clockwise
      for(i = iH; i >= iL; i = i - 1){
        PULSOUT(Servo_pin, i, 1, 1);
// Controls the speed of the servo in the counter
// clockwise direction
        PAUSE(5);
      }
    }
// Check if button is pushed every second
    button_pushed = BUTTON(37, LOW, HIGH, 5);
// If button was pushed, reset the variable that keeps
// track of that event, and switch the ON/OFF state of
// the servo
    if (button_pushed){
      button_pushed = FALSE;
      if (servo_on == FALSE) servo_on = TRUE;
      else servo_on = FALSE;
      PAUSE(1000);
    }
  }
}
```

Project 10: Digital Thermometer

In this project, we will design a Digital Thermometer. The design will use a temperature sensor (Part Number CS452000) to measure the temperature in Degrees Fahrenheit every second, and display it in Binary Coded Decimal (BCD) using the LED8 – LED1 LED's present in the C Stamp μC101 BOL. We will assume that the temperature range is from 0 to 99 °F inclusive. Then, if the BOL breadboard is at the bottom, LED8 – LED5 will be the most significant digit, and LED4 – LED1 will be

the least significant one. For each of those nibbles, the left most LED will be the Most Significant Bit (MSB), and the right most LED will be the Least Significant Bit (LSB). An LED that is on would have a value of "1", and if the LED is off, this would indicate a '0'.

The figure below shows the pins arrangement of the CS452000, and the following table shown the Detailed Pin Connections. Pins $A_2 - A_0$ can be set to any digital levels, as long as the combination represents a unique binary number (0 – 7) in your system. This allows the users to have up to 8 Temp Sensors in a single system connected to the same 2-wire serial bus (pins 28 and 29 of the C Stamp). We will use address 0 for the Temperature Sensor in this project.



Temp Sensor Pin-Out Connectivity		
Pin	***Symbol***	***Description***
1	SDA	Data input/output pin for 2-wire serial communication port. Connect to C Stamp pin 29.
2	SCL	Clock input/output pin for 2-wire serial communication port. Connect to C Stamp pin 28.
3	T_{OUT}	Thermostat output. Active when temperature exceeds TH; will reset when temperature falls below TL. Leave unconnected.
4	GND	Ground pin. Connect to GND in the BOL.

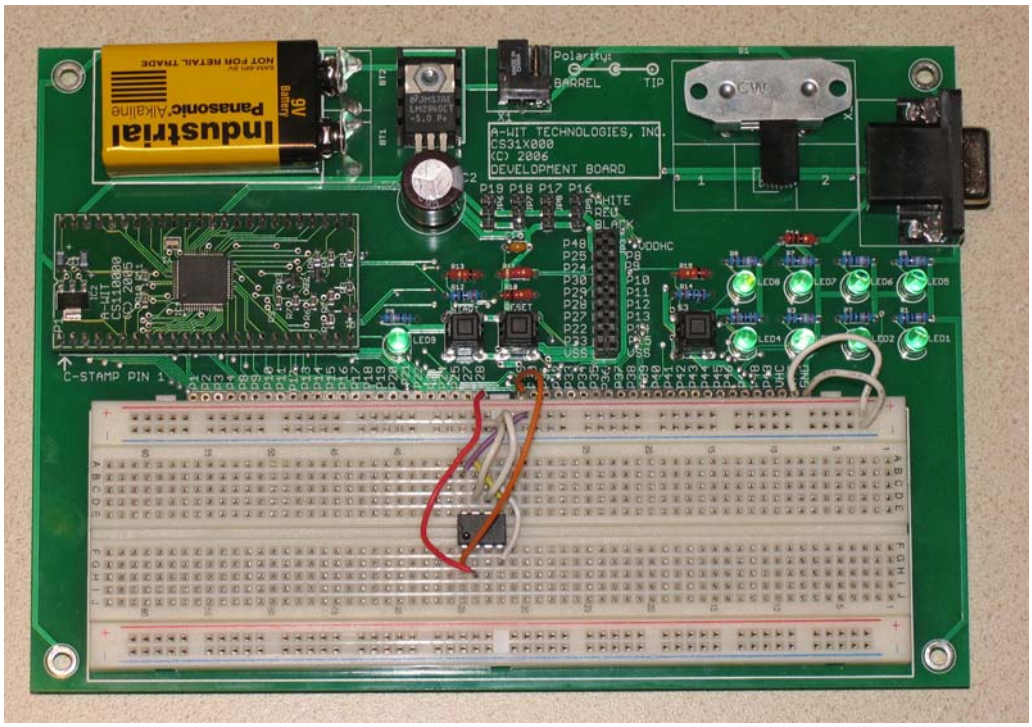
Temp Sensor Pin-Out Connectivity		
Pin	***Symbol***	***Description***
5	A2	Address input pin. Connect to GND in the BOL.
6	A1	Address input pin. Connect to GND in the BOL.
7	A0	Address input pin. Connect to GND in the BOL.
8	V_{DD}	Supply voltage input power pin. (2.7V to 5.5V) Connect to VHC in the BOL.

The CS452000 has the following Technical Specifications:

- Temperature measurements require no external components.
- Measures temperatures from -55°C to +125°C in 0.5°C increments. Fahrenheit equivalent is -67°F to 257°F in 0.9°F increments.
- Wide power supply range (2.7V to 5.5V).
- Temperature acquisition in less than 1 second.
- Thermostatic settings are user definable and nonvolatile.
- Data is read from/written via a 2-wire serial interface (open drain I/O lines) that allows to connect up to 8 Temp Sensors on the same two lines.
- Applications include thermostatic controls, industrial systems, consumer products, thermometers, or any thermal sensitive system.
- 8-pin DIP package.
- Compact and breadboard-friendly.
- Compatible with C Stamp microcomputer.

Project 10: Solution

Our circuit for the interfacing of the Temperature Sensor is simple. We just connect the sensor per the table above in the BOL. The wired circuit in the μC101 BOL is shown in the following figure, and the code below is our solution for this project. The comments embedded within the code offer more clarifications.




```
#include   "CS110000.h"

void main(void)
{
// TYPE YOUR CODE HERE AFTER THIS LINE

// declare variables
  float T, temp;
  BYTE TB, MSD;
  BYTE LEDpins[] = {46, 45, 44, 43, 42, 41, 40, 39};

  while(TRUE){  // infinite loop
// get floating point temperature in Degrees Fahrenheit
    T = TEMPSIN_CS452000(0, TEMPS_F);
```

```
// convert to BCD
// round floating point temperature and convert to BYTE
// sized positive integer
    TB = T + 0.5;
// round floating point temperature and store Most
// Significant Digit
    temp = (T + 0.5) / 10.0;
    MSD = temp;
// get Least Significant Digit
    TB = TB - (MSD * 10);
// construct BCD BYTE
    MSD = MSD << 4;
    TB = MSD | TB;

// display temperature
    BYTEOUT(TB, LEDpins);

// wait 1 second
    PAUSE(1000);
  }
}
```

The program above works as follows. First, we define some necessary variables and the pin connections of the LED8 – LED1 LED's in the BOL. Then we acquire the temperature as a floating point number, convert to a BCD BYTE, and display it. Before taking another measurement, we wait for one second, which gives us the 1 second update rate in side the infinite loop of temperature monitoring.

The following is a detailed example of the floating point to BDC conversion, assuming that the floating point number returned by the **TEMPSIN_CS452000** function is T = 75.6 °F.

1. Round floating point temperature and convert to BYTE sized positive integer

   TB = T + 0.5 = 75.6 + 0.5 = 76.1 = 76 = 01001100 (in binary)

2. Round floating point temperature and store Most Significant Digit

   temp = (T + 0.5) / 10.0 = (75.6 + 0.5) / 10.0 = 76.1 / 10.0 = 7.61

   MSD = temp = 7.61 = 7 = 00000111 (in binary)

3. Get Least Significant Digit

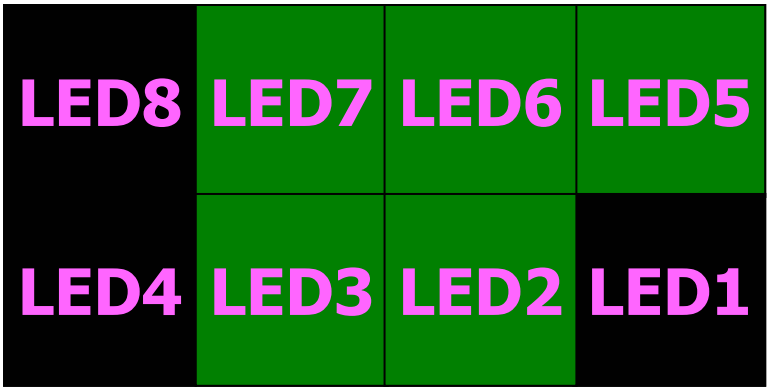
   TB = TB - (MSD * 10) = 76 – (7 * 10) = 76 – 70 = 6 = 00000110 (in binary)

4. Construct BCD BYTE

   `MSD` = `MSD` << 4 = 00000111 << 4 = 01110000 (in binary)

   `TB` = `MSD` | `TB` = 01110000 | 00000110 = 01110110 (in binary)

Notice that our rounded integer temperature should be 76 °F, because the floating point measurement was 75.6 °F. Thus the upper 4 bits of the TB BCD BYTE are 0111, which is 7 in binary; and the lower 4 bits of the TB BCD BYTE are 0110, which is 6 in binary. Then, our LED's would light up as per the following figure.



Terms and 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.