Volume

1

A-WIT TECHNOLOGIES INC.

… a passion for execution …

C Stamp Syntax and Reference Guide Manual

Version 2.7

A-WIT TECHNOLOGIES INC.

C Stamp Syntax and Reference Manual

Table of Contents

Chapter 1

Introduction to the C Stamp

Welcome to the world of the C Stamp™ microcontroller module. C Stamp modules are microcontrollers (self-contained single chip computers) that are designed for use in a wide variety of applications. Projects that form an embedded system, meaning that they have some level of intelligence derived from hardware-software interaction, can use a C Stamp module as the controller.

Acknowledgements

A-WIT Technologies would like to thank and recognize Mr. Keith Kitowski, "Tinkerer Extraordinaire", for a very thorough review and feedback for this manual.

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 words "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, and analog functions. The C stamp is complemented by a set of library function commands for math, pin operations, and much more. C Stamp modules are capable of running many thousands of 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.

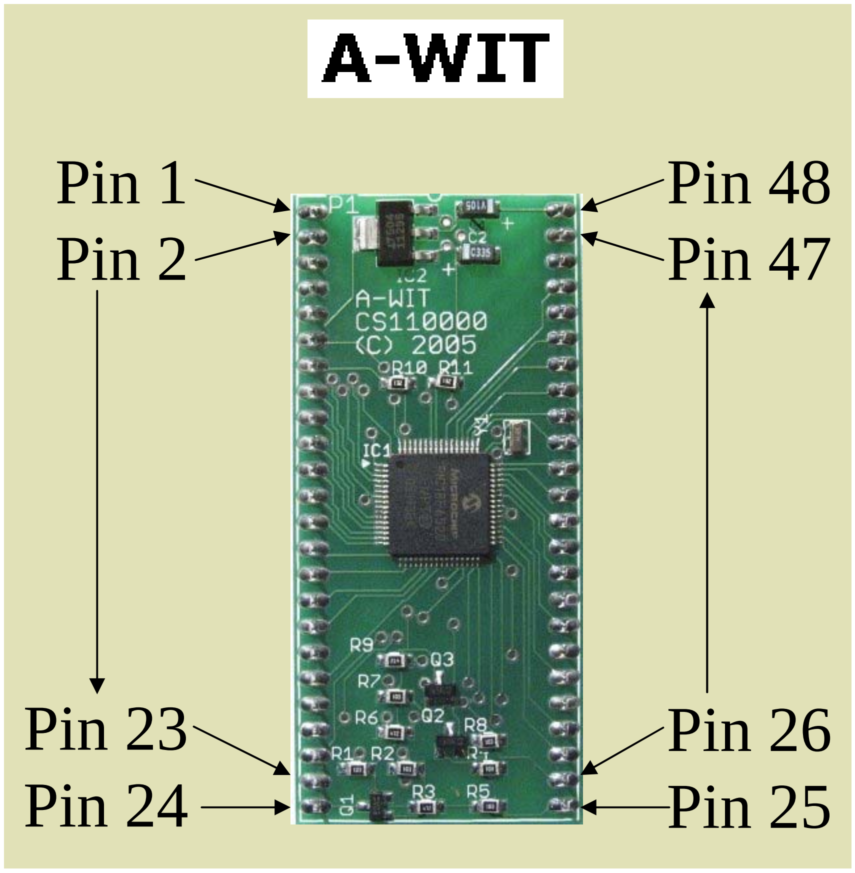
Features/Attributes	
PACKAGE	48-pin DIP
PACKAGE SIZE (L x W x H)	2.4" x 1.1" x 0.4"
PINS ATTACHMENT METHODOLOGY	Through Hole Strong Will not fall off
ENVIRONMENT	-40 to 85 deg. C (-40 to 185 deg. F)
MICROCONTROLLER	MICROCHIP PIC18F6520
PROCESSOR SPEED	40 MHz
PROGRAM EXECUTION SPEED	~10,000,000 instructions/sec.
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
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	MPLAB IDE (v7.22 and up)

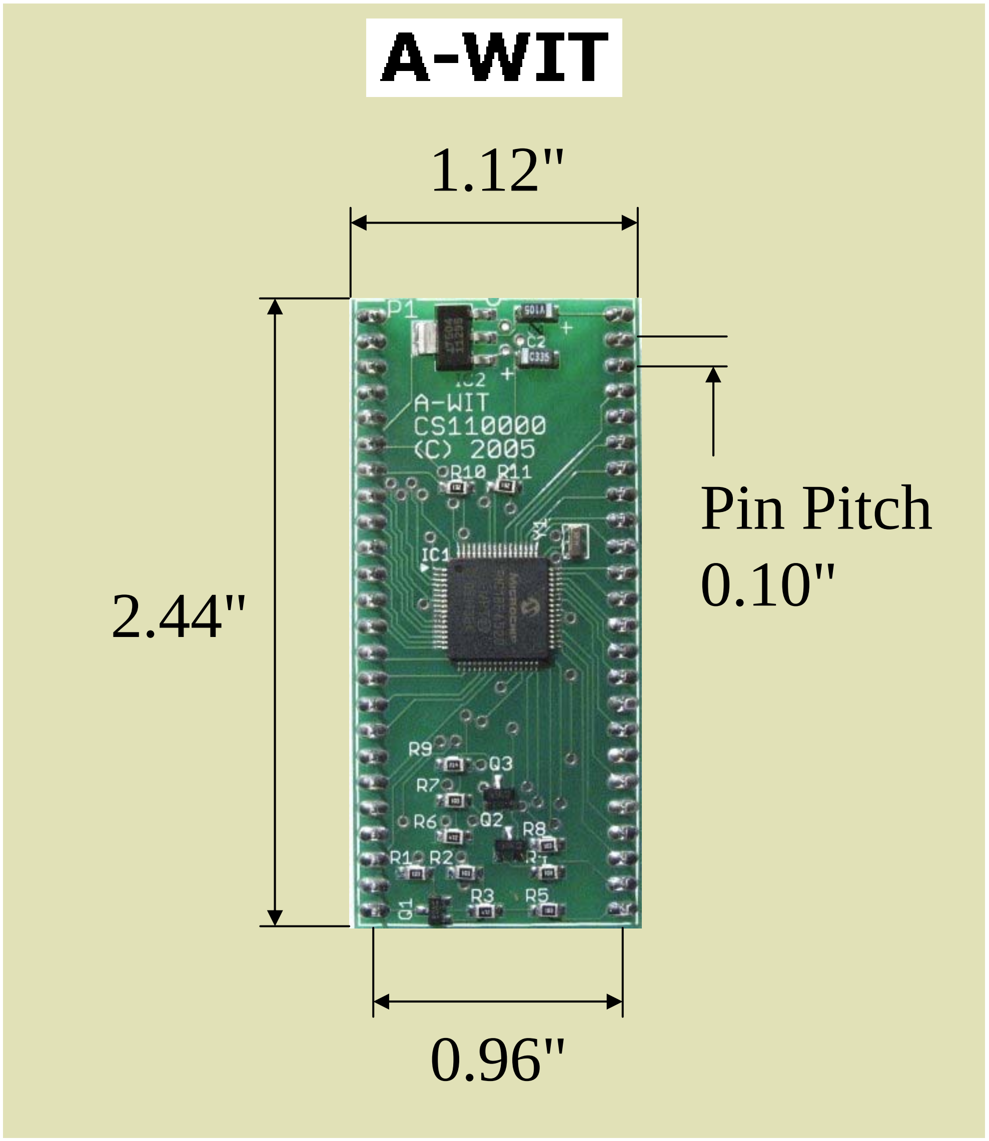
Features/Attributes	
ENVIRONMENT	
EEPROM (DATA) SIZE	1K Byte
INTERRUPTS	8
DIGITAL TO ANALOG CONVERTERS	2 channels (10 bits) single ended
OTHER COMMUNICATION INTERFACES	3-wire SPI™, I^{2}C Master and Slave
PARALLEL SLAVE PORT	8 bits
ANALOG TO DIGITAL CONVERSION	12 channels (10 bits) single ended
ANALOG COMPARATORS	2
WATCHDOG TIMER	Yes

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

NOTE: UNDER NO CIRCUMSTANCES SHOULD THE TOTAL CURRENT SUPPLIED BY ALL THE PINS OF THE CS110000 C STAMP MODULE 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; the diameter of the C Stamp pins is 0.018". The table after that provides the CS110000 C Stamp module Pin Descriptions.





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

Pin #	***Name***	***Type***	***Buffer***	***Description***
17	RA2/AN2/VREFM 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		Dedicated 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 I^{2}C 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 I^{2}C 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	 I/O	 TTL	 Digital I/O Port B, bit 7

Pin #	*Name*	*Type*	*Buffer*	*Description*
	KBI3	I	TTL	Interrupt-on-change pin 3 Built-in pull-up
32	RB6/KBI2 RB6 KBI2	 I/O I	 TTL TTL	 Digital I/O Port B, bit 6 Interrupt-on-change pin 2 Built-in pull-up
33	RB5/KBI1 RB5 KBI1	 I/O I	 TTL TTL	 Digital I/O Port B, bit 5 Interrupt-on-change pin 1 Built-in pull-up
34	RB4/KBI0 RB4 KBI0	 I/O I	 TTL TTL	 Digital I/O Port B, bit 4 Interrupt-on-change pin 0 Built-in pull-up
35	RB3/INT3 RB3 INT3	 I/O I	 TTL ST	 Digital I/O Port B, bit 3 External Interrupt 3 Built-in pull-up
36	RB2/INT2 RB2 INT2	 I/O I	 TTL ST	 Digital I/O Port B, bit 2 External Interrupt 2 Built-in pull-up
37	RB1/INT1 RB1 INT1	 I/O I	 TTL ST	 Digital I/O Port B, bit 1 External Interrupt 1 Built-in pull-up
38	RB0/INT0 RB0 INT0	 I/O I	 TTL ST	 Digital I/O Port B, bit 0 External Interrupt 0 Built-in pull-up
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			

Pin #	***Name***	***Type***	***Buffer***	***Description***
	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 PSP2	 I/O I/O	 ST TTL	 Digital I/O Port D, bit 2 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.

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

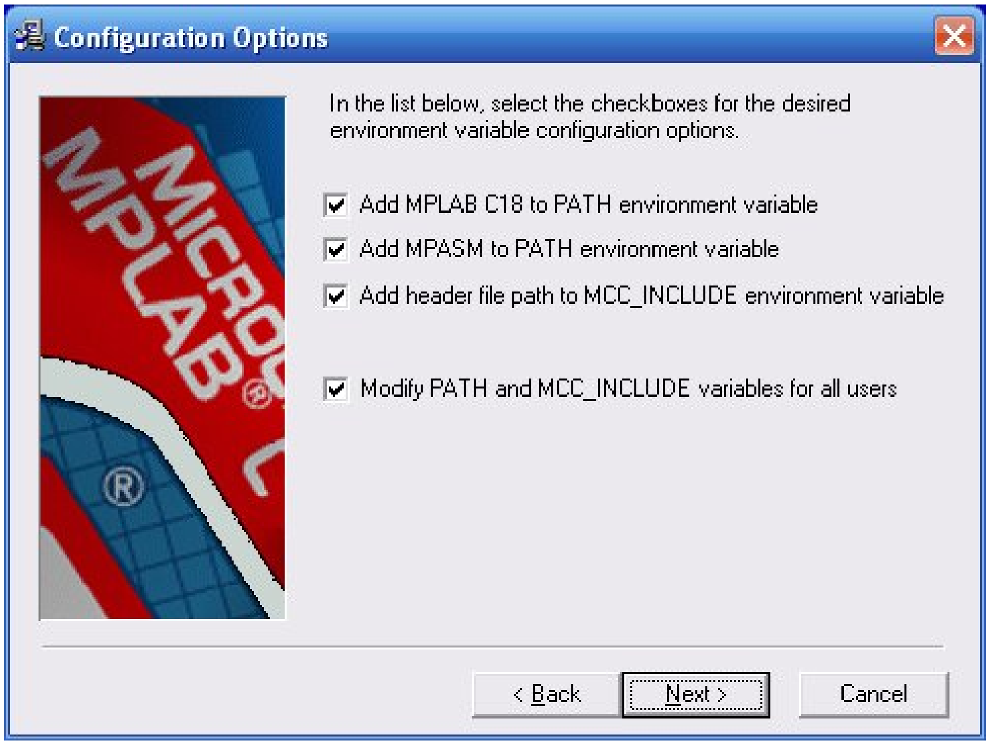
If possible, please check the C Stamp website www.c-stamp.com under SUPPORT for any updates to documentation, changes, or notices that may have become available since your Installation CD was produced. If you continue to have any issues for which a solution is not found in the aforementioned website,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

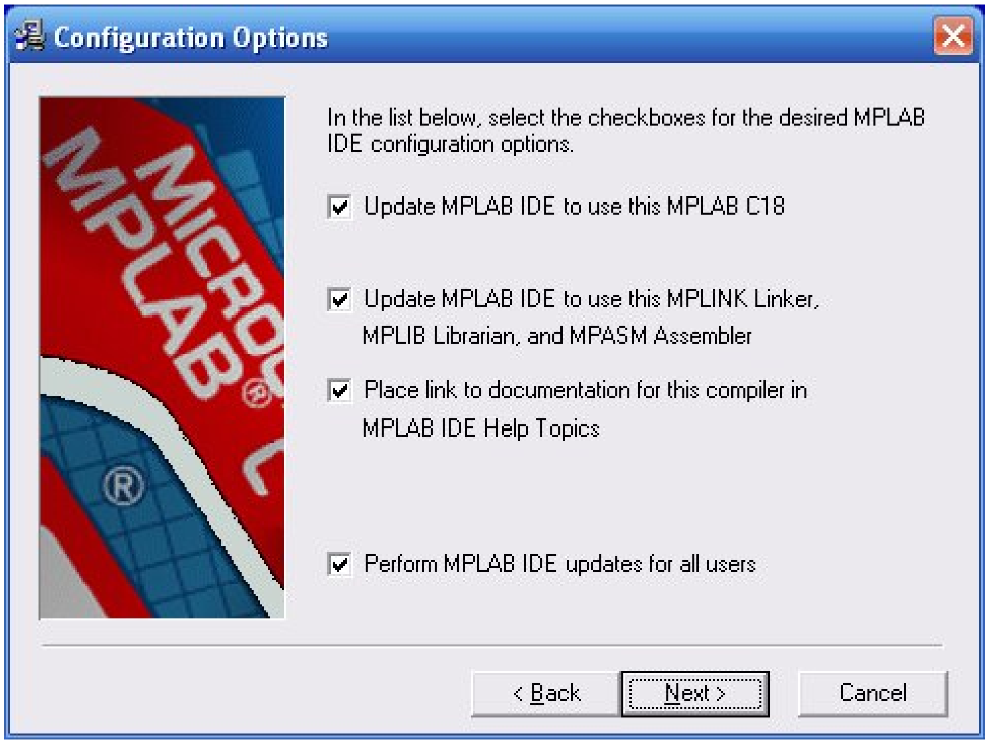
Insert your A-WIT provided Installation CD in your CD drive. Go to the MPLAB directory in the CD and double click on the “MPLAB vX.XX Install” file in that directory. Follow the installation steps, prompts, and directions provided by the installer software, accepting all the default options.

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.



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



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

Installing the A-WIT C Stamp Quick Programmer

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

Installing the USB Software

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

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

Setting Up the C Stamp Software Templates

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

Documentation

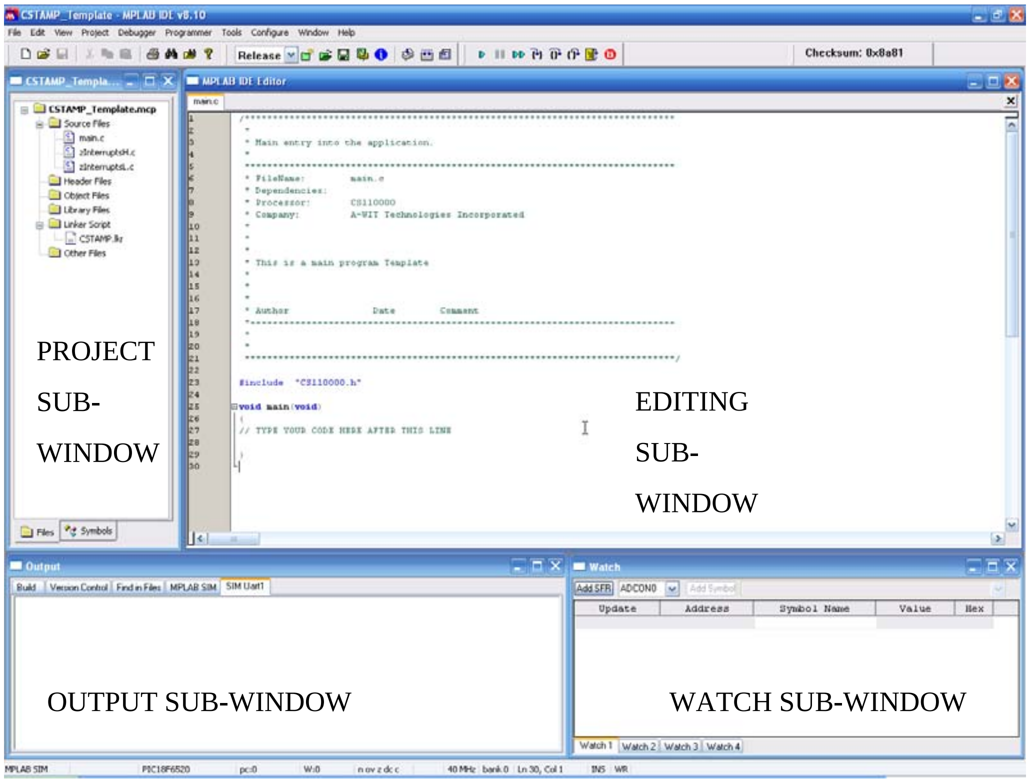
Copy the DOCS directory from the C Stamp Installation CD to your C:\A-WIT directory. This directory contains all the C Stamp related documentation in PDF format.

Creating your First C Stamp Program

Create a directory where you want to have 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 all the 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 in 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.


CSTAMP_Template - MPLAB IDE v8.10
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
Checksum: 0x8a81
CSTAMP_Templa...
CSTAMP_Template.mcp
Source Files
main.c
zInterruptsH.c
zInterruptsL.c
Header Files
Object Files
Library Files
Linker Script
CSTAMP.lkr
Other Files
Files
Symbols
MPLAB IDE Editor
main.c
PROJECT SUB-WINDOW
EDITING SUB-WINDOW
Output
Build Version Control Find in Files MPLAB SIM SIM Uart1
OUTPUT SUB-WINDOW
Watch
Add SFR ADCON0 Add Symbol
Update Address Symbol Name Value Hex
WATCH SUB-WINDOW
Watch 1 Watch 2 Watch 3 Watch 4
MPLAB SIM PIC18F6520 pc:0 W:0 n ov z dc c 40 MHz bank 0 Ln 30, Col 1 INS WR

```
/****************************************************************************
 *
 * Main entry into the application.
 *
 ****************************************************************************
 * FileName:        main.c
 * Dependencies:
 * Processor:       CS110000
 * Company:         A-WIT Technologies Incorporated
 *
 *
 *
 * This is a main program Template
 *
 *
 *
 * Author               Date        Comment
 *~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *
 *
 ****************************************************************************/

#include  "CS110000.h"

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

}
```

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 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), the program will run. This programs starts by lighting all 8 LEDs in your KIT and waits for the utility button at pin 37 to be pushed and let go (cycled). When you do this, the program blinks 4 of the 8 LEDs in your KIT. 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 and releasing 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);
  }
```

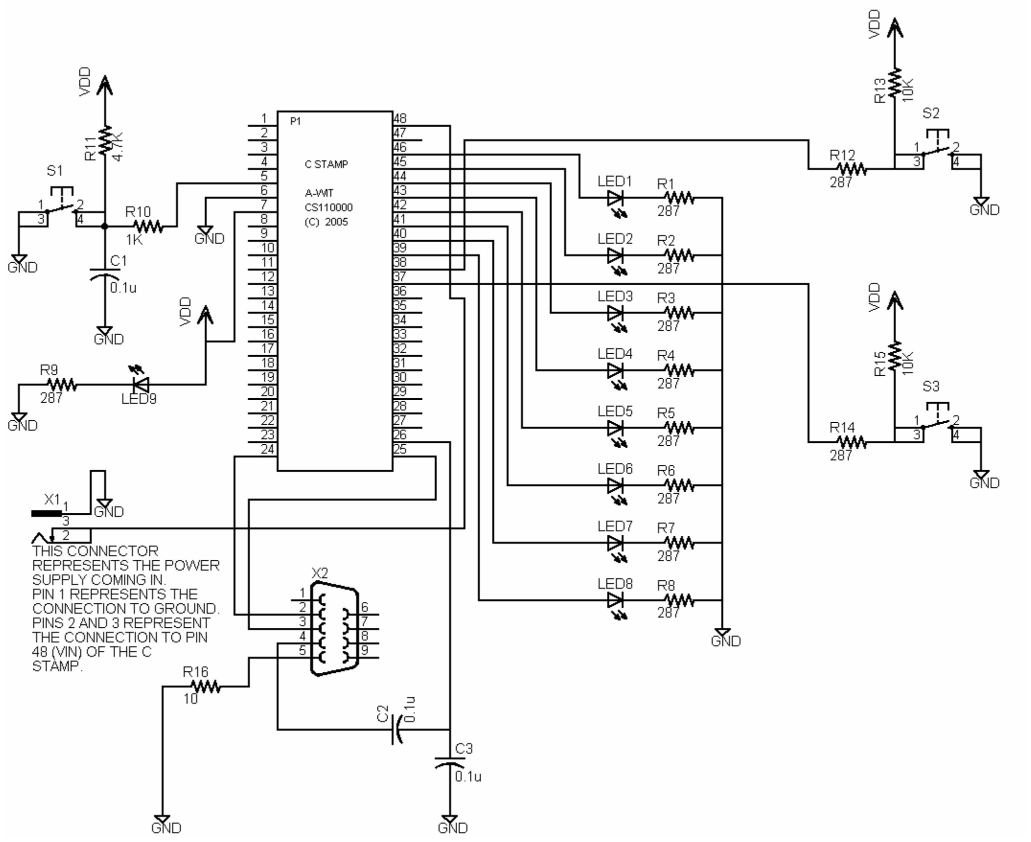
Save your program from the “File” menu or by clicking on the appropriate icon in the tool bar. Then “Build All” from the “Project” menu or from the tool bar.

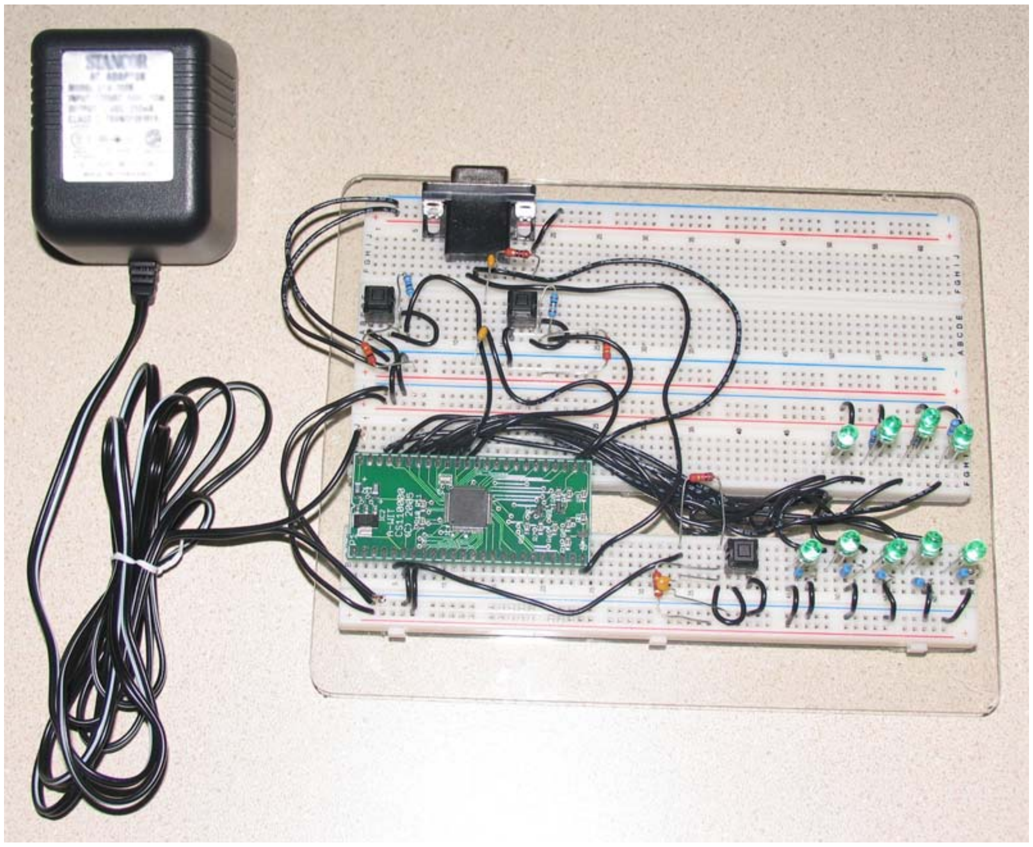
If the code was typed correctly, you will have a file in your program directory with the name of your program project and a .HEX extension. An example is FIRST_LED_APP.HEX. This is the file that you will download to the C Stamp, as explained up ahead.

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 errors. 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.

Assembling Your C Stamp Prototyping Circuit

The figures below show a schematic of the circuit that you will build if you have purchased the C Stamp Programming KIT and the finished product. In the schematic, when two wires cross, they do not connect unless there is a junction (dot) joining them. This kit will allow you to download your programs to the C-Stamp, run demo programs, and even add your own circuitry for your other own simple projects. After you have downloaded a program and from that point on, you can use the C Stamp in your other own circuits, in simple demos with the kit, or in other circuits that you put together onto the kit. All of this is done without the kit prototyping board having to be connected to the PC via the download cable (serial or USB). If you have experience with breadboards, it is easy to build the circuit. If not, just follow the detailed instructions below.





The overall goals of the prototyping kit is to allow you to download programs from the PC to the C Stamp, run demo programs, and serve as a starting platform for your own projects. The prototyping kit features a connector to the PC, power supply input, reset and start buttons, a utility button for generic inputs, and eight LED's for outputs.

To build your prototyping kit, start with the special C Stamp solder-less breadboard. Notice that this breadboard has two panels; one with a separation in the middle to accommodate the C Stamp (BOTTOM), and another without it (TOP). 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.

These are the step-by step instructions and comments about building your prototyping kit. The first step is to connect all the blue rows together that are going to supply your ground (node GND), and all the red rows together which are going to supply your positive supply as it is output from the C Stamp pin 7 (node VDD).

Then insert your C Stamp all the way to left in the BOTTOM panel as shown in the figure. Connect pin 6 of the C Stamp to ground (blue), and pin 7 to the positive supply (red). The C Stamp is a very sophisticated piece of electronics, so try not to touch its pins, so they do not get bent or damaged.

Connect the first four output LEDs in the lower corner of the breadboard (LEDs 1-4 in the schematic). Each LED has a limiting resistor connected to ground on one side. This resistor sets the current flowing through the LED to provide enough illumination (about 15 mA). Connect the resistors vertically. If leads of components have to be trimmed, a wire cutter or a set of shears can be used. Remember which were the short lead and the longer lead of the LED. Also, as you connect components, you can push wires out of the way. Then the short lead, or the one that was the short lead, of the LED (the Cathode) connects to the ungrounded side of the resistor, and the other lead of the LED (the Anode) connects to a C Stamp pin. Connect the right most LED to pin 46 of the C Stamp, and so on, so that the left most LED is connected to pin 43 of the C Stamp. For example, to start these connections, you can connect a resistor from B-63 to B-62 in the very lower-right corner of the breadboard, then connect a wire from A-63 to the grounding terminal (the nearest blue row), then connect the short LED lead to C-62 in that region and the other LED lead in the next insert C-61, and then connect a wire from the LED to the C Stamp pin 46 by connecting a wire from E-61 in the lowest right most region to next to the C Stamp pin 46, which is insert G-3 in the breadboard (by pin 46 of the C Stamp).

Repeat the above step for LEDs 5-8 in the schematic, placing these LEDs above the previous four. Connect the right most LED to pin 42 of the C Stamp, and so on, so that the left most LED is connected to pin 39 of the C Stamp.

The next step is to connect the power indication LED. This is the fifth right most LED in the most bottom row of LEDs, and it signals that the C Stamp is generating power correctly on pin 7. Connect this LED like the others, but connect the long lead of the LED to the nearest red row or the positive supply out of the C Stamp.

Next, connect the reset push-button S1 as shown in the schematic. The provided switches have four legs, but two electrical contacts. As you look at the switch with the button up and one of the smooth sides toward you, the two legs on your right is one contact tied together inside the switch, and the other two are the other electrical contact tied together inside the switch. After assembled, the reset circuit connects to the master reset (MCLRn) input of the C Stamp at pin 5. When the button is pushed, the MCLRn is LOW, and the C Stamp is doing nothing in reset mode. When the button is depressed, the RC circuit attached to the button allows for a smooth LOW-

HIGH transition on the MCLRn pin, and the C Stamp exits reset mode, and enters active mode.

The next steps are to connect push-buttons S2 and S3, as indicated in the schematic. S2 is the START button, which will be explained later, and S3 is a utility switch that you can use for the demo program and your own projects.

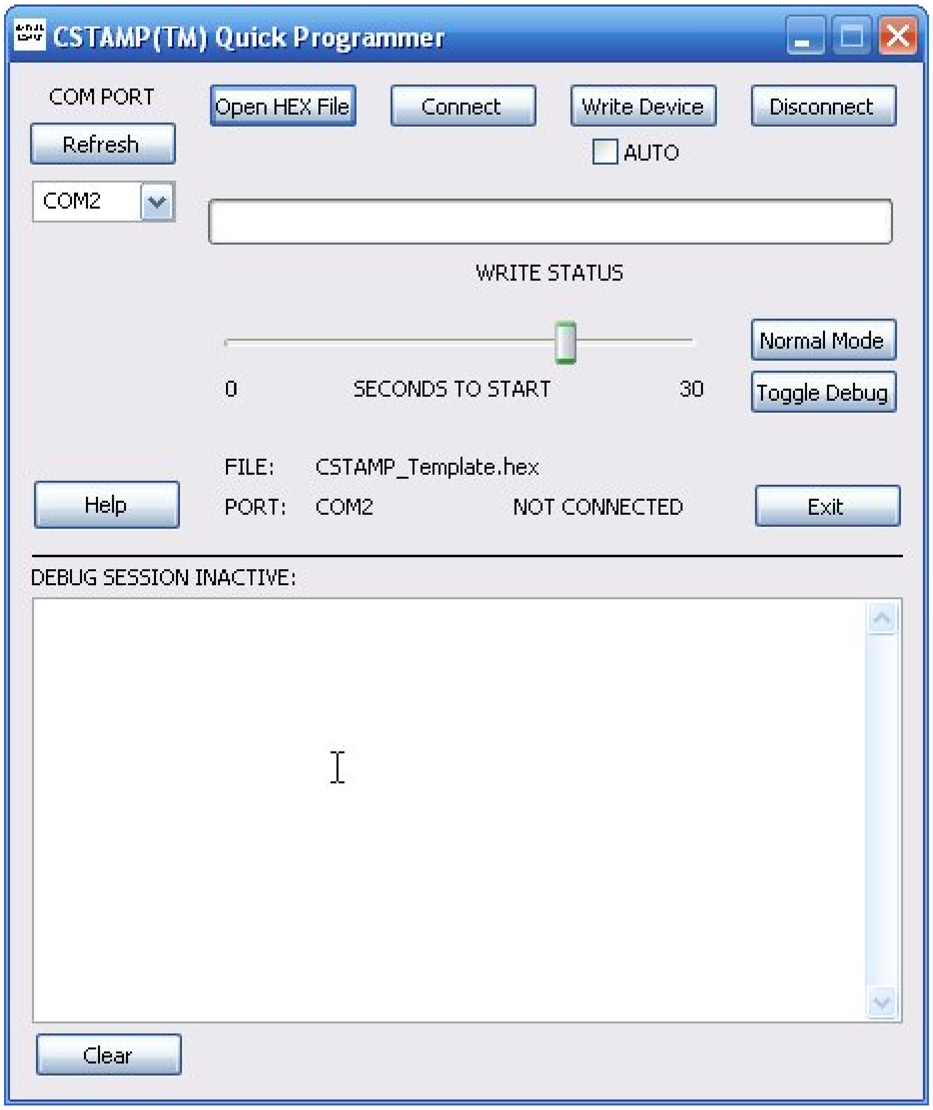
Next, connect the serial cable assembly that will connect to the breadboard, and provide a connector for the cable that will connect the breadboard to the PC. If you look at the connector in the assembly, you will see that the terminals (holes) are numbered. These same numbers are used in the schematic. Our recommendation is to use the area of the breadboard above switches S2 and S3 to connect the serial assembly. Connect terminals 1-5 of the assembly to inserts H-10 through H-14, and then connect a wire from F-11 of that area to pin 24 of the C Stamp. Connect terminal 3 of the assembly to pin 25 of the C Stamp. These two pins of the C Stamp are the ones used for the serial connection to the PC. The third connection is for terminal 4 of the assembly through capacitor C2 to pin 26 of the C Stamp; also connect capacitor C3 to pin 26 of the C Stamp. Finally, connect terminal 5 of the assembly to the ground (GND) plane through a 10Ω resistor.

The last step in putting together the kit is to connect the power supply. The positive terminal of the power supply is the wire with the white stripe, and the negative terminal of the power supply is the one with the solid colored wire (i.e. the wire does NOT have a white stripe). Connect the positive terminal to pin 48 of the C Stamp, and connect the negative terminal to any ground (GND) connection.

At this point your kit is ready to accept programs from the PC, and to run programs, which will be explained subsequently.

Downloading and Running Your Program

Power up your kit, and connect it 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.



FILE: CSTAMP_Template.hex
PORT: COM2 NOT CONNECTED

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 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 kit, 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. After the text below the progress bar indicates that writing has finished, you can click on "Disconnect" to disconnect the PC from the kit, disconnect the serial cable from both the PC and the kit, and start your program manually at the kit. 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 kit. Then you can disconnect the serial cable from the PC and the kit. 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 get 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 to bypass the START button sequence and start your program at power up or when the RESET button is pressed, the solution is simple. Tie pin 38 of the C Stamp permanently to ground through a low value resistor equal to or less than 287 Ω. Of course, this is after the C Stamp has been programmed. To program the C Stamp, pin 38 should be tied to a HIGH through the same type resistor (287 Ω or less). Although bypassing the RESET sequence costs a pin (i.e. you cannot use pin 38 for anything else in your project), the C Stamp has so many I/O pins that this should be an acceptable option.

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. To do this, your first step should be to thoroughly study Chapter 4 of this manual: C Stamp Language and Command Reference.

Chapter
3

Getting More from the Tools

This chapter addresses various techniques to get more information from the project development software tools, as well as advanced usage, and debugging with these tools. The software tools have many advanced features that will give you information about your C Stamp, and help you with the debugging of your programs. The more you get familiar with the tools, the easier project development will be. It is suggested that you explore all the menu items, and the Help facilities, and if you have any problems or questions you can always contact our support department by e-mailing tech_support@a-wit.com.

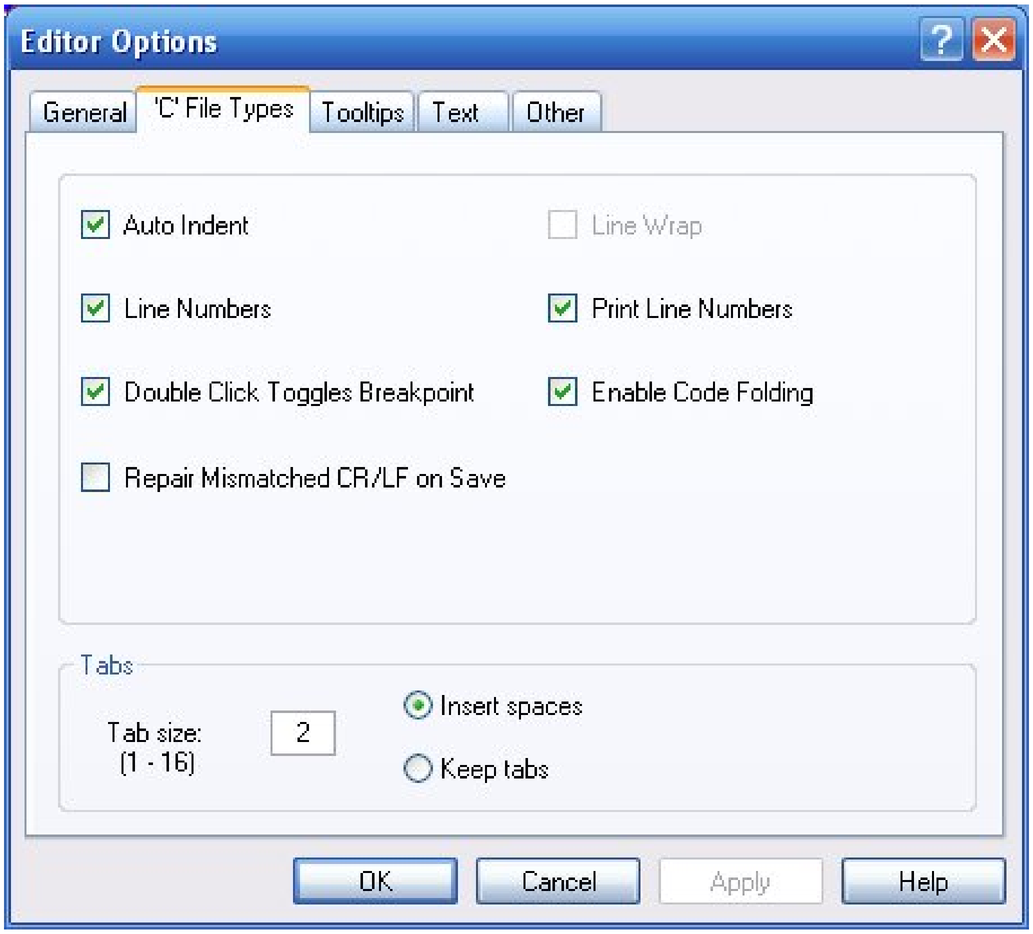
Recommended Editing Settings in the MPLAB IDE

The following figure shows the A-WIT recommended C editing setting in the IDE. These setting will enable:

- Automatic indenting
- Displaying line numbers
- Printing line numbers
- Code folding
- Spaces instead of tabs, where 2 spaces = 1 tab

Line numbers allow you to reference code more easily. Code folding is very handy, as it allows you to collapse code that you are not interested in examining or code portions you are done with; including functions. Then expand a code fragment when you want to look at it. It de-clutters your code file when you are looking at it.

To change these settings, go to the “Edit” menu in the IDE and select “Properties ...”. Then select the “'C' File Types” tab, and make the changes per the following image.



- [✓] Auto Indent
- [ ] Line Wrap
- [✓] Line Numbers
- [✓] Print Line Numbers
- [✓] Double Click Toggles Breakpoint
- [✓] Enable Code Folding
- [ ] Repair Mismatched CR/LF on Save

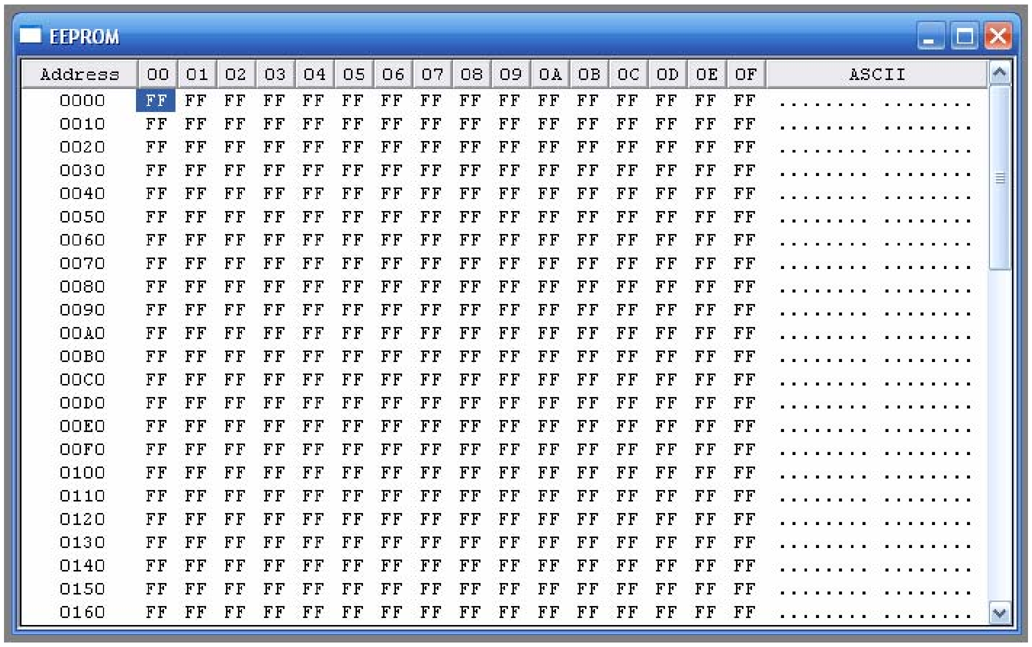
Tabs

Tab size: (1 - 16) 2

[✓ Insert spaces] Keep tabs

Exploring the Data EEPROM in the MPLAB IDE

Go to the "View" menu, and click on EEPROM. This will bring up a sub-window with the contents of the EEPROM. Such window is shown below.

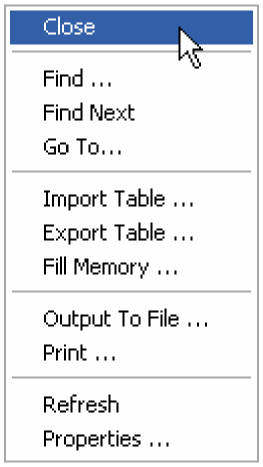


EEPROM

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
00E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
00F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0100	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0110	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0120	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0130	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0140	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0150	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0160	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

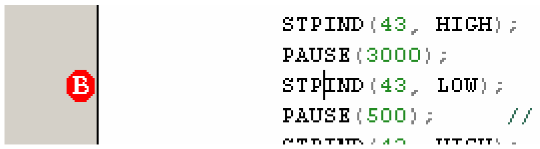
This sub-window presents the contents of the Data EEPROM in table format with 16 addresses in each row. The C Stamp has 1024 Data EEPROM locations of 1 BYTE each, so the Data EEPROM address locations run from 0 through 1024; or from 0000 through 03FF in hexadecimal (hex). The addresses in the "Address" column is the first address of a 16 BYTE block, but in hex format. The hex numbers in the header row of the next 16 columns (00 through 0F) are the added to the base block address in the "Address" column to form the actual address. This amounts to replacing the last digit of the base address, which is always 0, with the second digit (0 through F) of the hex numbers in the headers of the columns. This sub-window also presents each BYTE in the Data EEPROM as an ASCII character if the BYTE indeed converts to an ASCII equivalent character, which may not always be the case.

If you right-click on this sub-window, you will see a menu like the one shown below. The entries in this menu will allow you to Close the sub-window, find a specific value, go to a specific address, import and export the contents of the memory as a single column textual value format for any memory range, fill the memory with a specific value, output the complete table to as a text file or a portion of it by lines or addresses, print the table, refresh it, and change its properties. The properties that can be changed are the font and the font color for values that just changed, but this will apply to other sub-windows as well, so it recommended that these properties are NOT changed.



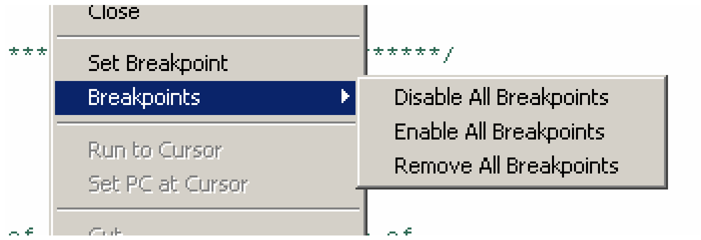
Breakpoints

Sometimes while performing a simulation you may wish to pause the simulation at a specific line of code. To do this you must add a Breakpoint at that particular line of code. Setting a Breakpoint can be done one of several ways. One way is to simply double click on the line at which you wish to pause. When you set a Breakpoint somewhere, the simulation will pause just prior to executing this command. Once you double click the line at which you wish to set a Breakpoint, and a red circle with a white letter "B" inside of it will appear to the left of the line of code as shown below.



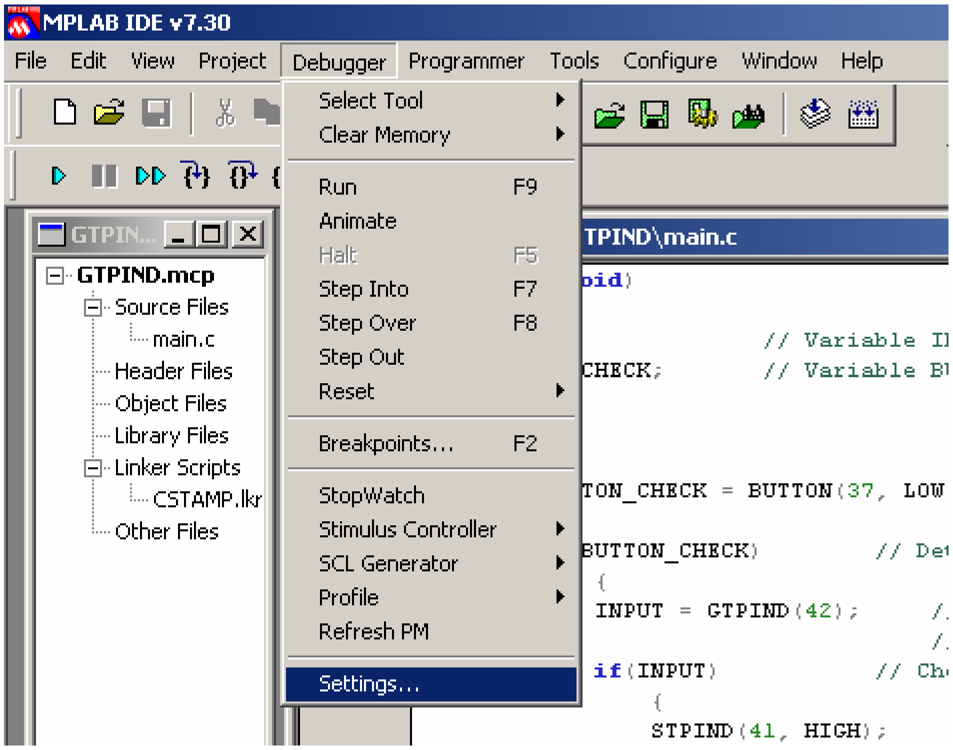
```
STPIND(43, HIGH);
PAUSE(3000);
STPIND(43, LOW);
PAUSE(500);     //
```

Another way to set a Breakpoint is to right click on the line of code and select the "Set Breakpoint" option in the menu that appears. The menu that appears also has a submenu titled "Breakpoints" in which you can remove all Breakpoints, temporarily disable all Breakpoints and restore all previously disabled Breakpoints. Part of this menu is shown below.



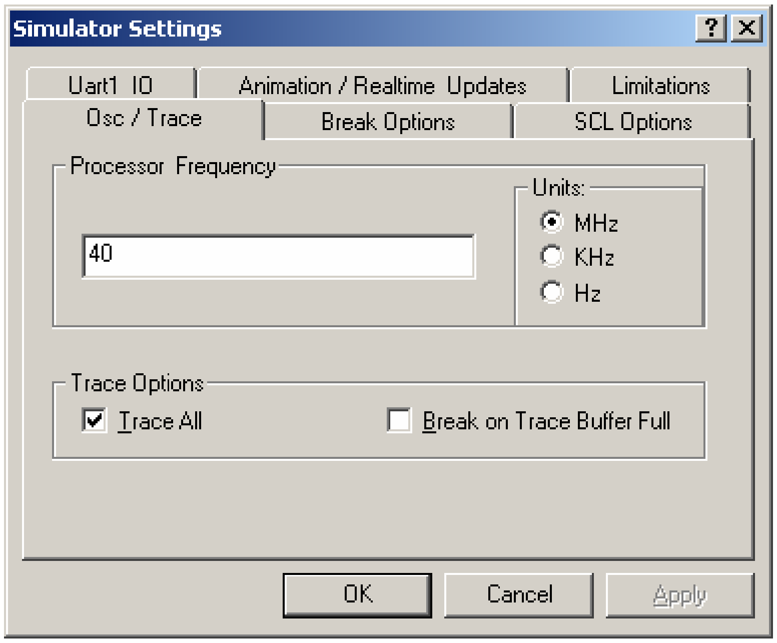
Timing – Stop Watch

Included in the tools, there is one named the "Stop Watch", which allows the user to determine the actual time spent executing a line or group of consecutive lines of code. The first step to utilize the Stop Watch tool is to ensure that the software development environment is properly configured to simulate the C Stamp. First select the "Debugger" menu, then under the Debugger menu select "Settings...", as shown below.



```
oid)
                          // Variable I
CHECK;                    // Variable B

TON_CHECK = BUTTON(37, LOW

BUTTON_CHECK)              // De
 {
 INPUT = GTPIND(42);        /
                            /
 if(INPUT)                 // Ch
     {
     STPIND(41, HIGH);
```

Once you have selected Settings, a dialog window will open. Select the tab "Osc / Trace" if it is not already selected. Under "Processor Frequency", enter the number "40". Under "Units", select the radial button next to "MHz". The dialog window should look as depicted below.

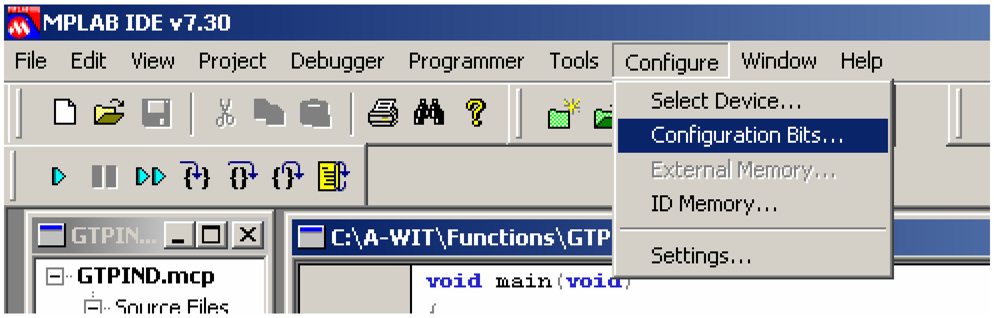


Processor Frequency: 40

Units: [✓ MHz] KHz Hz

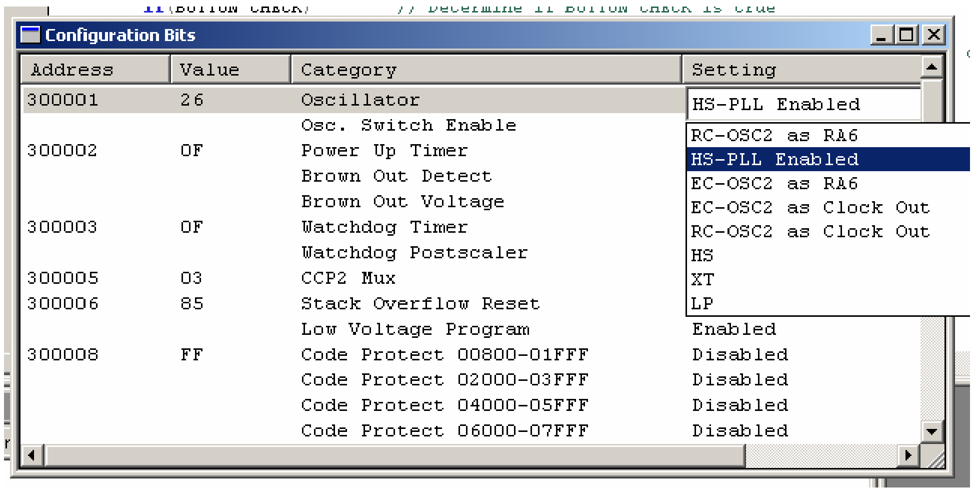
Trace Options: [✓ Trace All] Break on Trace Buffer Full

After the dialog window looks as shown, press OK and continue by going to the "Configure" menu and selecting the "Configuration Bits..." option.



```
void main(void)
```

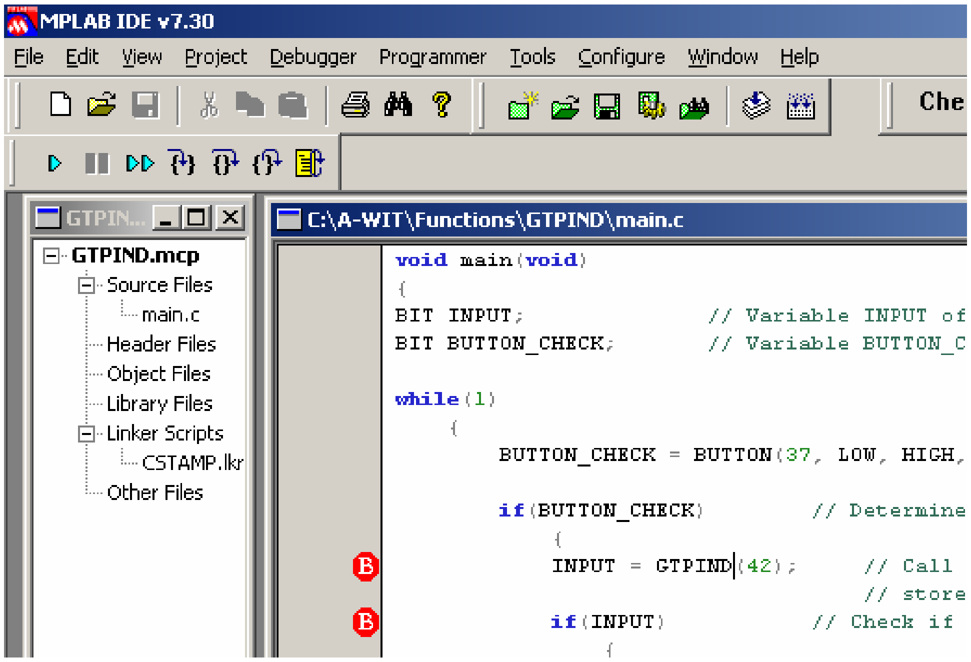
Once selected a window will appear which displays the options for the various configuration bits. To complete configuration to properly simulate the timing of the C Stamp the setting for the "Oscillator" must be "HS-PLL Enabled". Once this selection is made as shown below, simply close the window by clicking on the "X" for the Configuration Bits window.



Configuration Bits

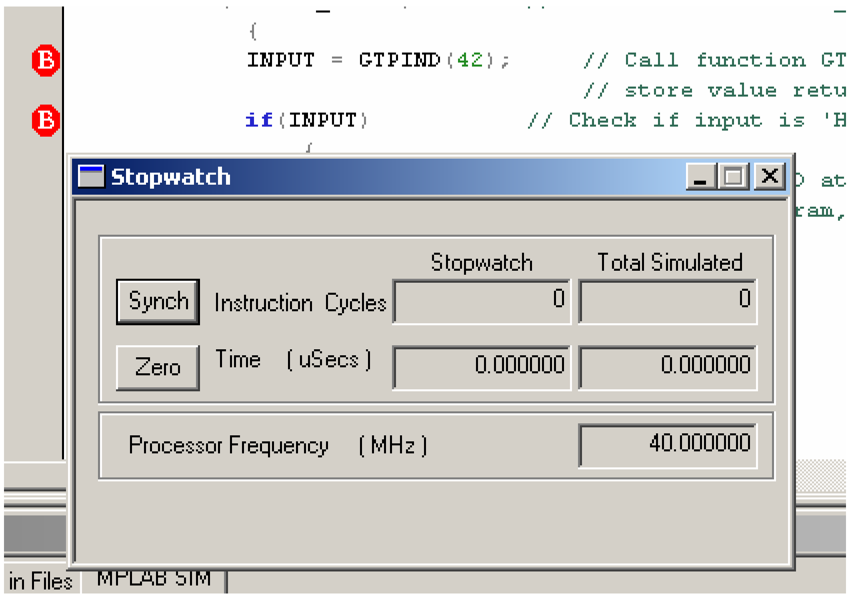
Address	Value	Category	Setting
300001	26	Oscillator	HS-PLL Enabled
		Osc. Switch Enable	
300002	0F	Power Up Timer	
		Brown Out Detect	
		Brown Out Voltage	
300003	0F	Watchdog Timer	
		Watchdog Postscaler	
300005	03	CCP2 Mux	
300006	85	Stack Overflow Reset	
		Low Voltage Program	Enabled
300008	FF	Code Protect 00800-01FFF	Disabled
		Code Protect 02000-03FFF	Disabled
		Code Protect 04000-05FFF	Disabled
		Code Protect 06000-07FFF	Disabled

Now we are finished with the setup to simulate timing events on the C Stamp. To use the Stop Watch we must first let the tool know where to pause simulation. For our purpose we will want to pause simulation just before and just after the code we're interested in timing. In this example we want to know how long it would take for the C Stamp to read the digital value at a pin and then store that value in a variable. First, we begin by double clicking on the line of code where we call the command to read the value of the pin and store it in a variable. When we do this, to the left beyond where text can be entered, a red circle with a white "B" inscribed can be seen. Next, we will double click the line of code just below the command and another red circle with inscribed white "B" will appear to the left of that line. It is important to note that if the line immediately following the line of interest does not contain code then you cannot place a "Breakpoint" there. The simulation will not be affected by the empty line in between so in such a case simply place the Breakpoint on the next line of code.



```
void main(void)
{
BIT INPUT;                    // Variable INPUT of
BIT BUTTON_CHECK;             // Variable BUTTON_C

while(1)
     {
          BUTTON_CHECK = BUTTON(37, LOW, HIGH,

          if(BUTTON_CHECK)          // Determine
               {
B                   INPUT = GTPIND(42);      // Call
                                             // store
B                   if(INPUT)          // Check if
                         {
```

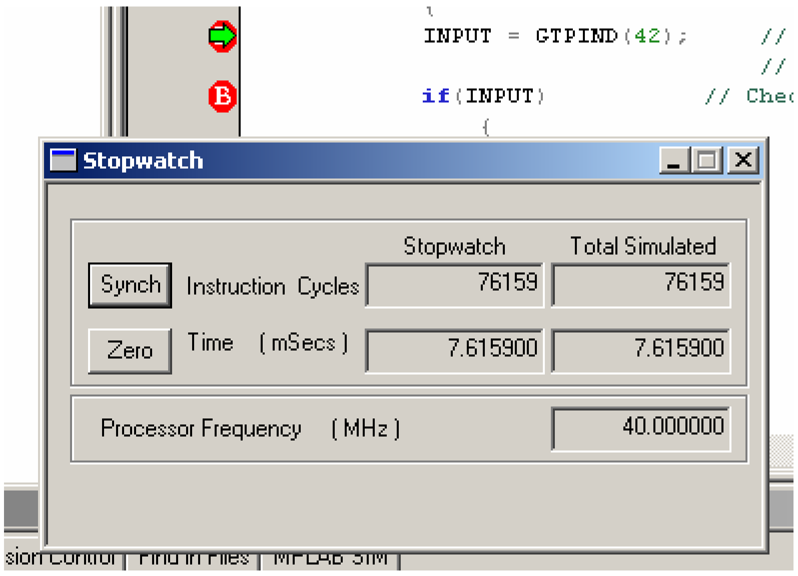
After selecting your Breakpoints select “Stop Watch” out of the Debugger menu and the Stop Watch tool window will appear. Once the window has been opened select the blue play button from the toolbar, or select the Run option in the Debugger menu, or press the F9 key.



```
               {
B                   INPUT = GTPIND(42);      // Call function GT
                                             // store value retu
B                   if(INPUT)          // Check if input is 'H
```

	Stopwatch	Total Simulated
Instruction Cycles	0	0
Time (uSecs)	0.000000	0.000000
Processor Frequency (MHz)		40.000000

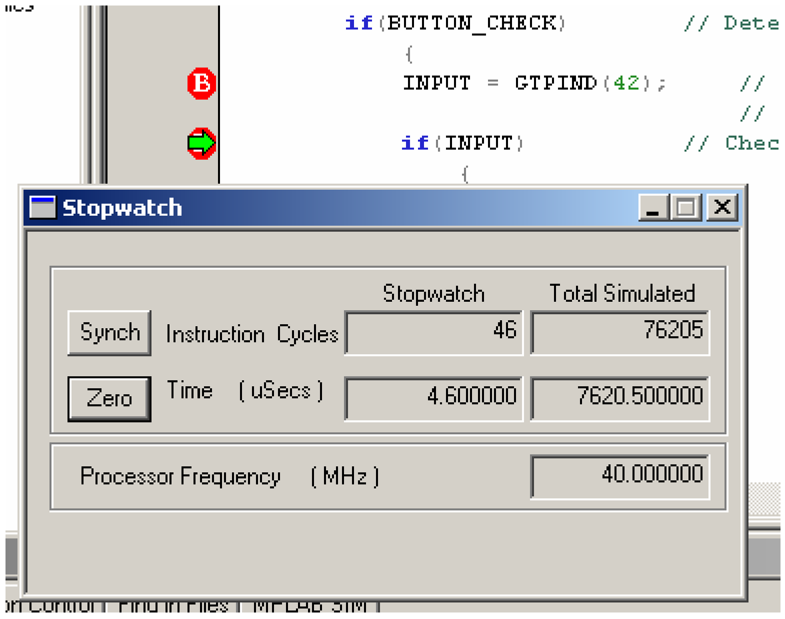
Clicking the play button from the toolbar starts the simulation. Once the simulation reaches the first Breakpoint it will stop and a green arrow will appear over top of the Breakpoint symbol found on the left. The Stop Watch window will display the number of instruction cycles as well as real time that has passed since the beginning of the program to the first Breakpoint.



```
INPUT = GTPIND(42);     //
                        //
if(INPUT)               // Chec
{
```

```
                         Stopwatch   Total Simulated
Instruction Cycles           76159             76159
Time    ( mSecs )         7.615900          7.615900
Processor Frequency  ( MHz )               40.000000
```

To determine how much time will pass in between the two Breakpoints click the "Zero" button on the Stop Watch tool window and hit the play button from the toolbar once more.



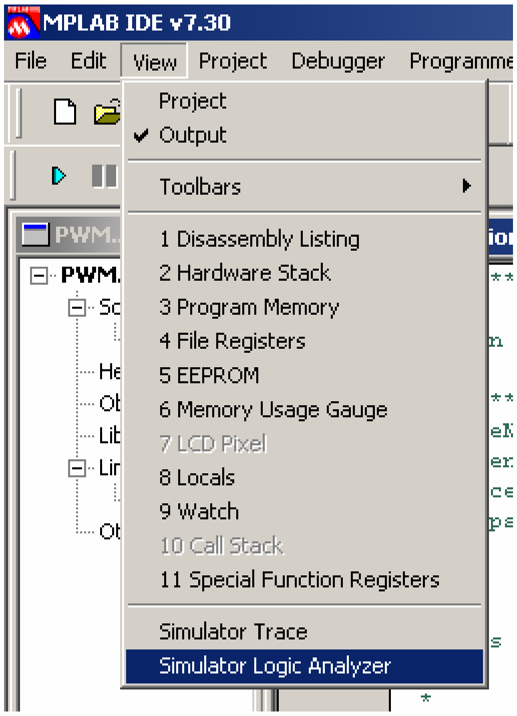
```
if(BUTTON_CHECK)           // Dete
    {
    INPUT = GTPIND(42);       //
                              //
    if(INPUT)              // Chec
        {
```

```
                         Stopwatch   Total Simulated
Instruction Cycles              46             76205
Time    ( uSecs )         4.600000       7620.500000
Processor Frequency  ( MHz )               40.000000
```

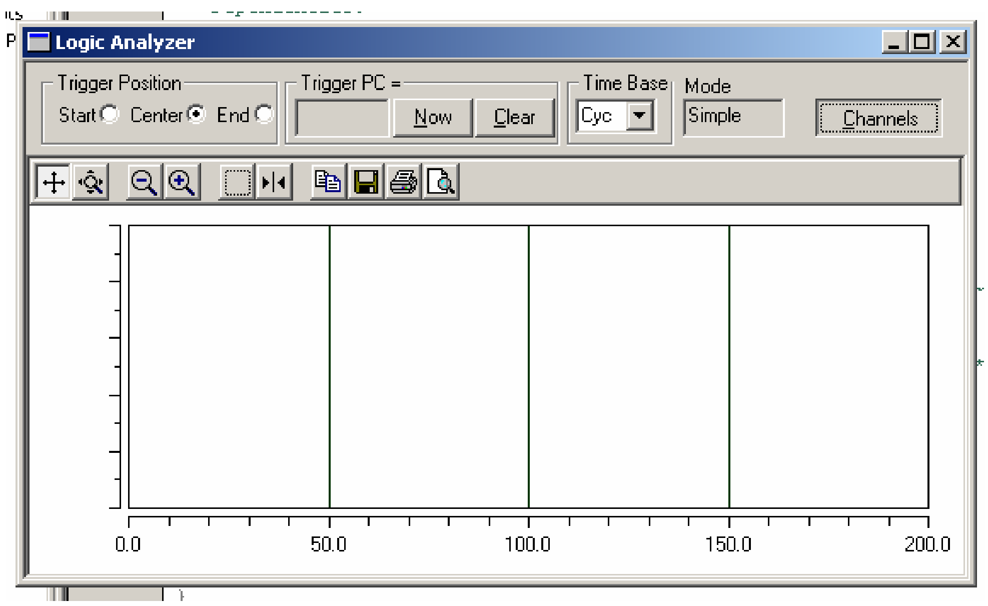
The Stop Watch tool will now show how many instruction cycles have been simulated since the last Breakpoint as well as the total amount simulated. This is also true of the real time which has passed in between Breakpoints and since the beginning of the program. In this case the Stop Watch tool shows that it would take the C Stamp 4.6 µs to find the value of a pin and store it in a variable. While this is only a simulation and experimental data may differ, the Stop Watch tool allows for very accurate predictions of how long events will take to execute within the C Stamp.

Logic Analyzer

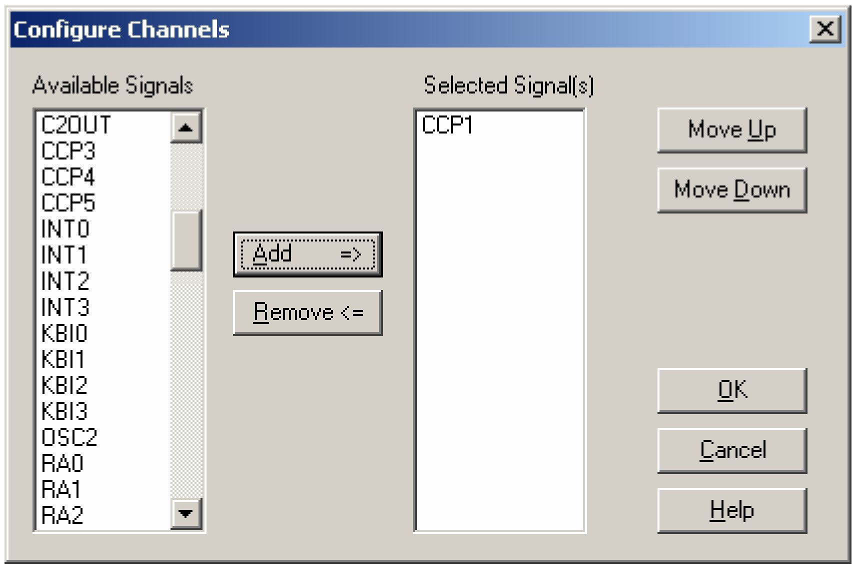
Included in the tools, there is one called the Simulator Logic Analyzer. This tool allows us to accurately simulate the waveform that will appear at a given pin over the execution of the program. To access this tool, select the "View" menu and select the option "Simulator Logic Analyzer" as shown in the picture below.



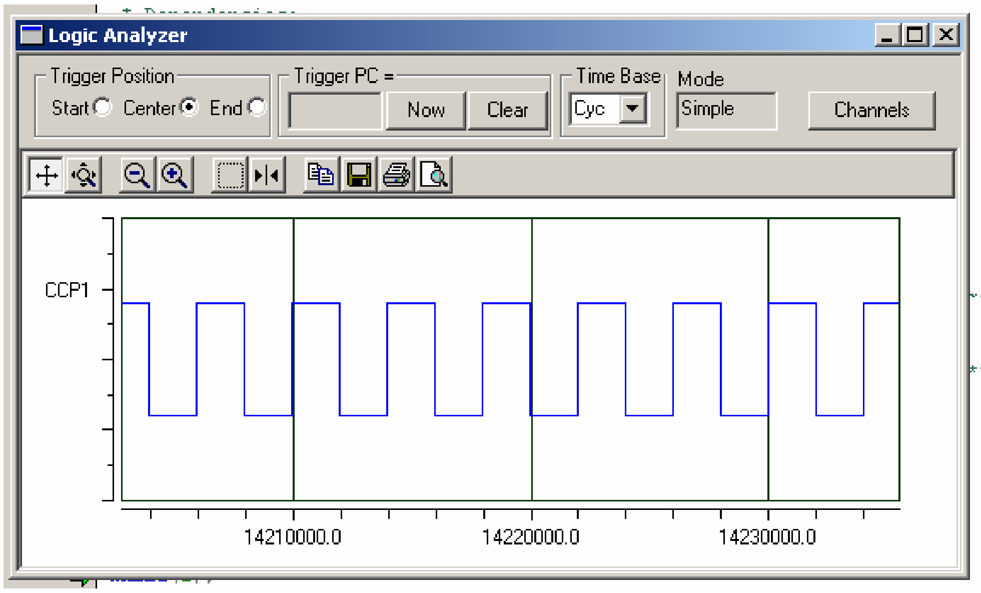
Once selected, the Simulator Logic Analyzer window will appear. To choose which pin you wish to analyze click the "Channels" button and a new dialog window will appear where you can make your selection.



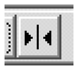
In the new dialog window select the channel you wish to view. You may have to reference the C Stamp pin out table in this manual to determine which "Available Signal" goes along with the pin you wish to analyze. In our case we will be looking at pin 27 which is CCP1.



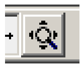
After selecting the proper signal click OK and the Logic Analyzer will be ready to graphically display the waveform which will appear on this pin. Simply start the simulation by either pressing the blue play button in the toolbar, choosing "Run" from the Debugger menu, or by pressing the F9 key. This starts the simulation, but in some cases, such as this one, if there is no termination written in the program we must end the simulation manually but pressing the blue pause button in the toolbar, choosing "Halt" from the Debugger menu or by pressing the F5 key. Once the simulation has been stopped the Simulation Logic Analyzer will display the desired waveform as seen below.



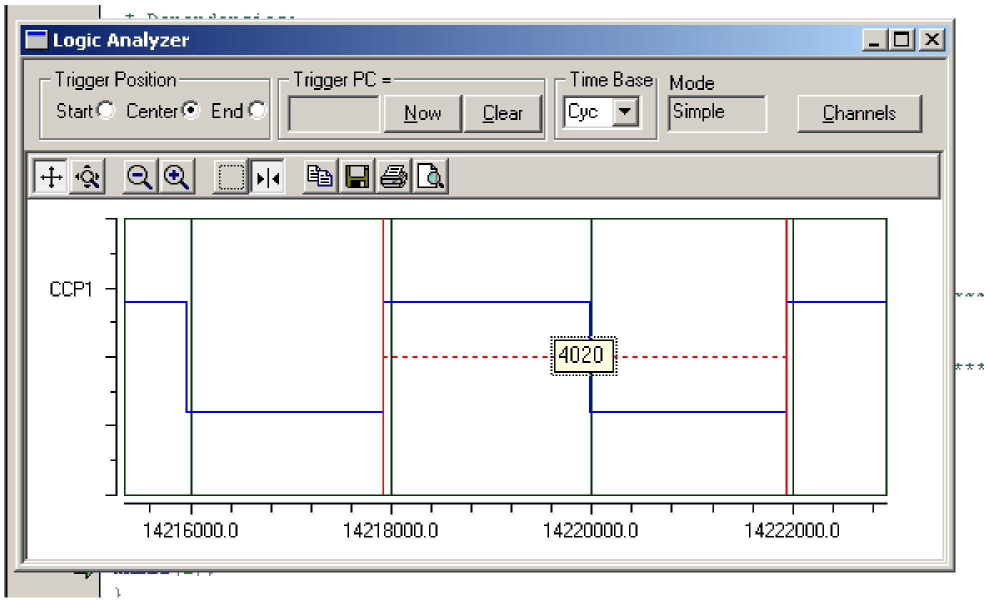
It is important to note that the time scale is in machine cycles and that one machine cycle is 0.1 μs. To measure simply select the button that has two arrows point inward at a line as seen below.



This will allow two red bars to appear on the screen which displays the number of machine cycles that separates the two. You may also zoom a specific axis, most commonly the time axis, by selecting the icon on the toolbar with a magnifying glass and four arrows pointing away as seen below.

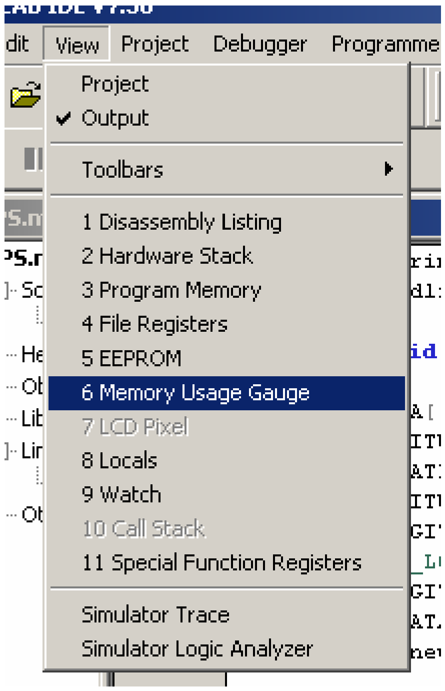


Once selected, simply click on the axis and drag left or right to zoom in or out until you find the appropriate magnification.

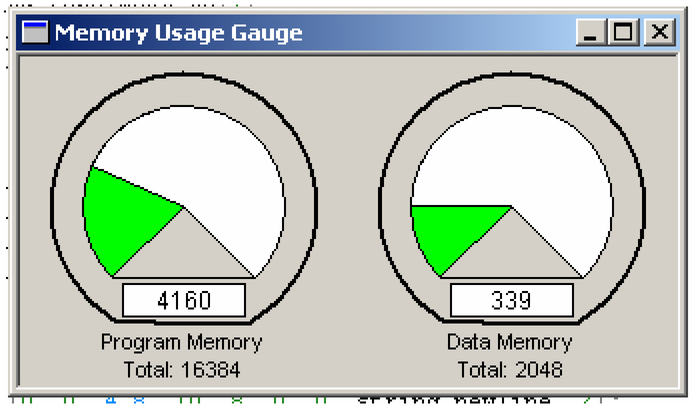


Memory Usage Gauge

The MPLAB IDE software provides a tool called the Memory Usage Gauge. The Memory Usage Gauge allows us to find out how much memory we are using for our programs and variables. With this tool we can find out the number of bytes being used as well as the total available. To open the Memory Usage Gauge, select "6 Memory Usage Gauge" from the "View" menu.



Once it has been selected from the menu, the Memory Usage Gauge will appear as shown below. The numbers inside of white rectangles are the number of bytes currently being used by the program (Program Memory) and for data storage (Data Memory). The gauge graphically displays the memory used as a percentage of the total memory available. The total memory available can be seen listed at the bottom.



Chapter

4

Interrupts

This chapter covers the Interrupts infrastructure in the C Stamp system. One the many features that set the C Stamp apart from all the other Stamp form-factor microcontrollers is the availability of Interrupt-drive event handling integrated seamlessly into the very user friendly software environment of the C Stamp. Thus, it was felt that a whole Chapter should be devoted to this topic. It is important to stress that it is **NOT** necessary to implement Interrupts into a project to be able to use the C Stamp. In fact, all projects supplied by A-WIT do not use Interrupts, and instead use the simpler technique of Polling-based event handling.

Introduction to Event Handling

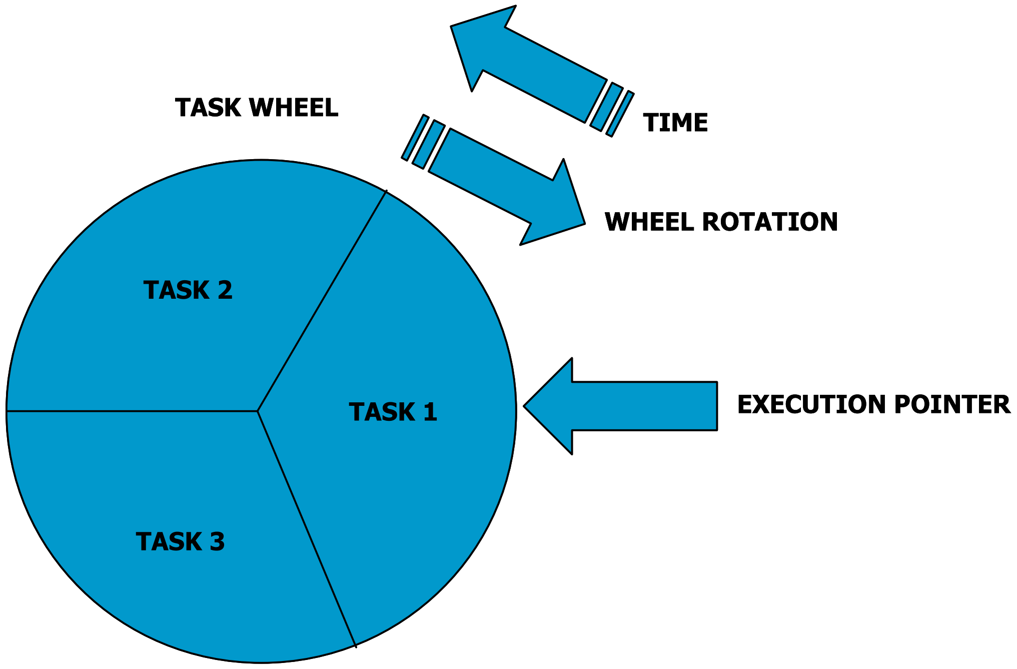
There are two general ways of handling events in Microcontroller-based systems. One is via polling the event sources, and the other is by servicing Interrupts originating from event sources. Since we encourage the handling of events via Polling, we also review both Polling and Interrupt techniques in this Chapter.

The dichotomy of event handling in Microcontroller-based systems can be seen in the table below.

Event Handling Approach	***Controller Software Architecture***	***Experience Level Required***
POLLING	TASK WHEEL	LOW
	FINITE STATE MACHINE	MEDIUM
INTERRUPT	INTERRUPT SERVICE ROUTINE	HIGH

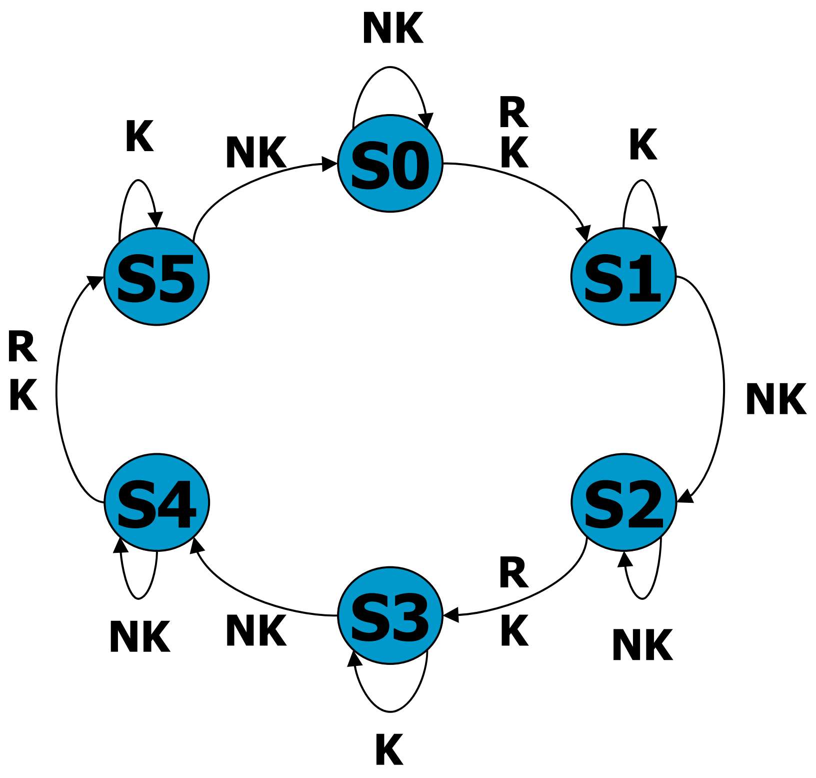
Event Handling with a Task Wheel

When using a Task Wheel, we start our program by resetting and initializing the system. Then we start the main infinite loop of our program, which essentially forms the Task Wheel. This approach entails 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. This wheel is defined by main non conditional processes, and conditional processes are said to be hidden within these non conditional tasks. Flags and counters are used for inter process communications; these are simply user defined variables that are visible across processes. A flag is a variable used to signal whether an event occurred, and a counter is a variable used to count the number of event occurrences. A visual depiction of a Task Wheel is shown in the figure below.



Event Handling with a Finite State Machine

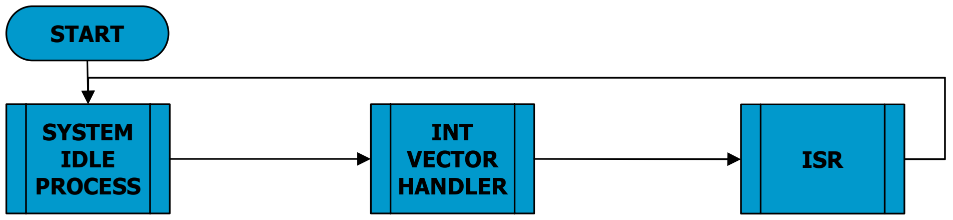
Although a Finite State Machine is usually associated with logic design, it can also be used in embedded systems software development. The software flow follows a State Diagram, where actions out of the system are taken depending on the Present State of the system, as given by a Present State Variable, and state transitions are taken based on the Present State and the inputs to the system that are polled during every execution cycle of the system. This execution cycle does not refer to the CPU execution cycle, but to the execution cycle of the main loop of the program. An example of a Sate Diagram is shown in the figure below with input actions results: NK, RK, and K; as a matter of example.



This state diagram has 6 states. We start at state S0, and as long as NK is TRUE, we remain in that state. Then if RK is TRUE, we transition state to S1, and we remain in that state until NK is TRUE. Then we transition to S2 so on until we get back to state S0 to start a new sequence.

Event Handling with Interrupts

When handling events with Interrupts, the main program consists of a System Idle Process, which can usually be an infinite loop where the processors is waiting for events to occur, and thus be interrupted out of the Idle Process to service the event in an Interrupt Service Routine (ISR). When an Interrupt occurs, the processor automatically goes to a specific Interrupt Vector Handler function depending on the priority level of the Interrupt event source, and from there the user can direct the processor to the appropriate ISR depending on the source and priority level of the interrupt. Once the Interrupt has been “serviced”, the processor returns to the System Idle Process to wait for other events to occur and be serviced. Interrupt sources can usually be given an individual level of Priority (High or Low), and they can be individually enabled or disabled. Process intercommunication is handled via global variables, which can take the form of data structures, such as semaphores, message queues, linked lists, buffers, etc. The flowchart in the following figure illustrates event handling via interrupts.



C Stamp Interrupts Infrastructure

The C Stamp provides 8 external interrupt source events as summarized in the table below. The **KBIx** group is sensitive to both edges (HIGH to LOW **and** LOW to HIGH transitions), and the **INTx** group is sensitive to a single edge (HIGH to LOW **or** LOW to HIGH transitions), which can be programmed. All interrupt sources, except INT0, which is always HIGH priority, can be assigned a priority level (HIGH or LOW). A high priority interrupt can interrupt a low priority one. In this case, the high priority interrupt would be serviced, and after this task is finished, control would return to the low priority interrupt already in progress. After the low priority interrupt finishes, control would resume back at the main program. At most two interrupts (a low priority and a high priority one) can be serviced at the same time, but on the same thread. When a low priority interrupt is being serviced, the low priority interrupts are disabled to prevent another low priority interrupt from interrupting the one being serviced. When a high priority interrupt is being serviced, all interrupts are disabled to prevent any interrupt from interrupting the high priority one being serviced.

Interrupt Name	***C Stamp Pin***	***Sensitivity***	***Priority***
KBI3	31	CHANGE	HIGH or LOW programmable
KBI2	32	CHANGE	HIGH or LOW programmable
KBI1	33	CHANGE	HIGH or LOW programmable

Interrupt Name	*C Stamp Pin*	*Sensitivity*	*Priority*
KBI0	34	CHANGE	HIGH or LOW programmable
INT3	35	EDGE programmable	HIGH or LOW programmable
INT2	36	EDGE programmable	HIGH or LOW programmable
INT1	37	EDGE programmable	HIGH or LOW programmable
INT0	38	EDGE programmable	valueless always HIGH

The interrupts subsystem in the C Stamp software support infrastructure consists of two Interrupt Vector Handler function templates in the C Stamp software template, and two functions to set up the interrupts and to find the sources of interrupts respectively. The **`zInterruptsH.c`** Interrupt Vector Handler function handles the HIGH priority interrupts, the **`zInterruptsL.c`** Interrupt Vector Handler function handles the LOW priority interrupts, the **`INTSET`** function sets up the interrupts, and the **`INTSOURCE`** queries the sources of interrupts.

The KBI interrupts were originally devised to interface to keyboards (hence the KB in KBI), therefore it is assumed that the user wants to read those pins to get the data. What this means is that the interrupt condition will not clear, even if **`INTSOURCE`**, which clears interrupts, is called until the pin is read. Therefore, you must read the pin at some point in your ISR.

Furthermore, the way the system is designed, if any interrupt occurred AND there were changes in any of the KBI pins AND the KBI pins have not been read (i.e. with **`GTPIND`**), the KBI interrupts as a group will get reported by **`INTSOURCE`**. The easiest thing to do is to mask the least significant nibble of the returned value by **`INTSOURCE`** if there is no interest in the KB interrupts. For example, if you are looking for INT1 (1000000 in binary), you would check with a statement like this:

```
if(INTSOURCE(some_priority) & 0xF0 == 0x40) ...
```

or if your are looking for INT2 (100000 in binary), you would check with a statement like this:

```
if(INTSOURCE(some_priority) & 0xF0 == 0x20) ...
```

To use a function that you defined in main.c in any of the interrupt files, you just restate its prototype in the interrupt file.

When using interrupts with the C Stamp, a user MUST follow the following steps:

1. Set up the appropriate global variables outside the **main** function in the **main.c** file in the Project.

2. Set up the interrupts using the **INTSET** function inside the **main** function in the **main.c** file in the Project.

3. Set up the System Idle Process infinite loop inside the **main** function in the **main.c** file in the Project.

4. If there are low priority interrupts:

    a. Restate the appropriate global variables with the **extern** data type modifier.

    b. Restate any functions that you defined in main.c that you are going to use while servicing the low priority interrupts.

    c. Query the interrupt sources using the **INTSOURCE** function inside the **InterruptHandlerLow** function in the **zInterruptsL.c** file in the Project.

    d. Setup the servicing calls for the low priority interrupts in the **InterruptHandlerLow** function in the **zInterruptsL.c** file in the Project.

    e. Restore the interrupts settings using the **INTSET** function before exiting the **InterruptHandlerLow** function in the **zInterruptsL.c** file in the Project. This is necessary, because the **INTSOURCE** function that was called previously disables interrupts, so that there are no conflicts when interrupts are being serviced.

5. If there are high priority interrupts:

    a. Restate the appropriate global variables with the **extern** data type modifier.

b. Restate any functions that you defined in main.c that you are going to use while servicing the high priority interrupts.

c. Query the interrupt sources using the **`INTSOURCE`** function inside the **`InterruptHandlerHigh`** function in the **`zInterruptsH.c`** file in the Project.

d. Setup the servicing calls for the high priority interrupts in the **`InterruptHandlerHigh`** function in the **`zInterruptsH.c`** file in the Project.

e. Restore the interrupts settings using the **`INTSET`** function before exiting the **`InterruptHandlerHigh`** function in the **`zInterruptsH.c`** file in the Project. This is necessary, because the **`INTSOURCE`** function that was called previously disables interrupts, so that there are no conflicts when interrupts are being serviced.

The two functions provided to support interrupts are described below:

```
void INTSET(BYTE interrupts, BYTE edge,
            BYTE priorities, NIBBLE calledfrom);
```

The **`INTSET`** function enables or disables interrupts, and sets the priority for each one of them.

`interrupts` is a variable/constant/expression (0 – 255) indicating whether each interrupt is going to be enabled or disabled. Each bit of this argument corresponds to an interrupt as shown in the following table. If the bit is **`ONE`**, the interrupt is enabled; if it is **`ZERO`**, the interrupt is disabled.

`edge` is a variable/constant/expression (0 – 255) indicating the edge of a particular INTx interrupt. Each bit (7 – 4) of this argument corresponds to the edge of an interrupt as shown in the following table. If the bit is **`ONE`**, the interrupt edge is a rising edge; if it is **`ZERO`**, the interrupt edge is a falling edge.

`priorities` is a variable/constant/expression (0 – 255) indicating the priority level of a particular interrupt. Each bit of this argument corresponds to the priority level of an interrupt as shown in the following table. When enabled, if the bit is **`ONE`**, the interrupt priority level is HIGH; if it is **`ZERO`**, the priority level is LOW.

`interrupts`*,* **`edge`***, and* **`priorities`** *BYTE BITS mapping to Interrupts*							
BIT 7	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**

`interrupts`*, *`edge`*, and *`priorities` ***BYTE BITS mapping to Interrupts***							
INT0	INT1	INT2	INT3	KBI0	KBI1	KBI2	KBI3

When a particular interrupt BIT is disabled, its corresponding **`edge`** and **`priorities`** BITs are meaningless.

BITs 3 - 0 of the **`edge`** BYTE can be anything, as the KBIx interrupts are sensitive to change, not edges.

BIT 7 of the **`priorities`** BYTE can be anything, as the priority of INT0 cannot be set. It is always a high priority interrupt.

If any of the KBIx interrupts have a high priority, then all of the KBIx interrupts will be treated as high priority.

For example, take the command

```
INTSET(0x40, 0x00, 0x40, 0x2);
```

You must convert the numbers into binary and look at the bit positions according to the table above: 0x40 = 01000000 so this means INT1 is enabled; 0x00 = 00000000 means the interrupt is set to falling edge; and 0x40 again is 01000000, meaning INT1 is high priority.

`calledfrom` is a variable/constant/expression (0 – 2) that signals the system from where is **`INTSET`** being called. The values of **`calledfrom`** are interpreted according to the table below.

Symbol	*Value*	*Meaning*
`INTLOW`	0	**`INTSET`** is being called from the low priority interrupt vector handler **`InterruptHandlerLow`**
`INTHIGH`	1	**`INTSET`** is being called from the high priority interrupt vector handler **`InterruptHandlerHigh`**
`INTOTHER`	2	**`INTSET`** is being called from somewhere else other than the two

Symbol	*Value*	*Meaning*
		cases above

`BYTE INTSOURCE(BIT calledfrom);`

The **`INTSOURCE`** function returns a **`BYTE`** that contains the sources of pending interrupts. When this information has been identified to be returned to the user, all interrupts will be disabled. Each bit of the returned **`BYTE`** corresponds to an interrupt as shown in the following table. If the bit is **`ONE`**, the interrupt is pending; if it is **`ZERO`**, the interrupt is not pending.

Pending interrupts returned BYTE BITS mapping to Interrupts							
BIT 7	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**
INT0	INT1	INT2	INT3	KBI0	KBI1	KBI2	KBI3

If any of the KBIx interrupts are pending, then all of them will be identified as pending.

`calledfrom` is a variable/constant/expression (0 or 1) indicating the priority level of the interrupt handler from which **`INTSOURCE`** is being called. If priority is **`ONE`**, this signals the system that **`INTSOURCE`** is being called from **`InterruptHandlerHigh`**; if priority is **`ZERO`**, the signal to the system is that **`INTSOURCE`** is being called from **`InterruptHandlerLow`**. **`INTHIGH`** and **`INTLOW`** can also be used respectively.

Event Handling Examples

In this Section, we will code the same program using the three event handling techniques described in this Chapter, so users can compare and contrast the differences between these approaches.

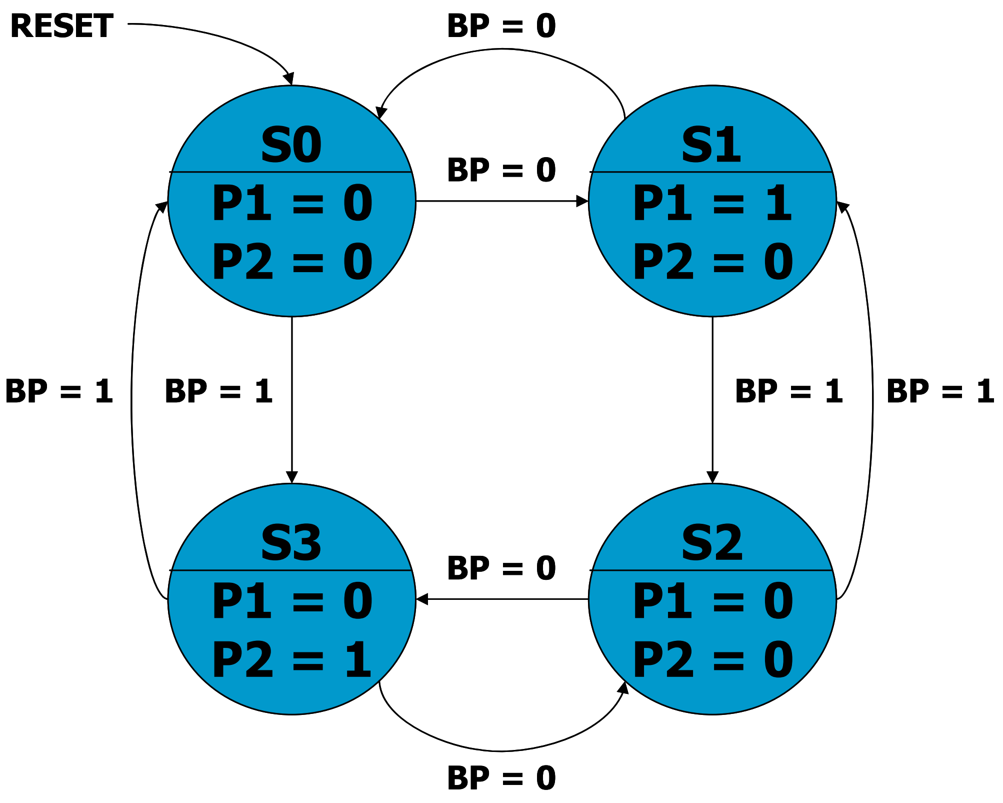
The problem to be coded is as follows. There are two LEDs properly biased and connected to pins 1 and 2 respectively. When the pins are HIGH, the LEDs are on, and when the pins are LOW, the LEDs are off. There is also a pushbutton properly biased and connected to Pin 38, such that when the Button is pushed, Pin 38 is LOW; otherwise, Pin 38 is HIGH. The functionality of the program needs to be such that the program starts by blinking the LED at Pin1 at a rate of 1 second (1 second on, 1 second off). Then, every time the Button is pushed, the LED that is blinking will get turned off and the other LED will start blinking at the same rate.

The following code implements this program using a Task Wheel technique. The comments in the code offer more clarifications.

```
#include  "CS110000.h"

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

  BYTE ON_PIN = 1;
  BYTE OFF_PIN = 2;
  BIT BUTTON_PUSHED;

// initialize LEDs
  STPIND(ON_PIN, HIGH);
  STPIND(OFF_PIN, LOW);

  while(TRUE){
// delay of 1 second built into BUTTON call
    BUTTON_PUSHED = BUTTON(38, LOW, LOW, 5);

// process button
    if(BUTTON_PUSHED){
      if(ON_PIN == 1){
        ON_PIN = 2;
        OFF_PIN = 1;
      }
      else{
        ON_PIN = 1;
        OFF_PIN = 2;
      }
    }

// set up LEDs
    STPIND(OFF_PIN, LOW);
    TOGGLE(ON_PIN);
  }
}
```

To implement the program using a State machine technique, we can use the State Diagram in the figure below. The solution is comprised of 4 states: 0, 1, 2, and 3 denoted as S0, S1, S2, and S3 respectively below. After RESET, the machine starts in

State 0, and at each state, pins 1 and 2 (P1 and P2 respectively) are set to the necessary value. The input to the state machine is the BP signal, which is 1 if the Button is pushed and 0 if it has not.



The following code implements this program using a State Machine technique. The comments in the code offer more clarifications.

```
#include  "CS110000.h"

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

  NIBBLE PRESENT_STATE = 0;
  NIBBLE NEXT_STATE = 0;
  BIT BP;

  while(TRUE){
// delay of 0.2 second built into BUTTON call
    BP = BUTTON(38, LOW, LOW, 1);
```

```
// next state logic
    switch(PRESENT_STATE){
      case 0:
        if(BP) NEXT_STATE = 3;
        else NEXT_STATE = 1;
        break;
      case 1:
        if(BP) NEXT_STATE = 2;
        else NEXT_STATE = 0;
        break;
      case 2:
        if(BP) NEXT_STATE = 1;
        else NEXT_STATE = 3;
        break;
      case 3:
        if(BP) NEXT_STATE = 0;
        else NEXT_STATE = 2;
        break;
    }

// output logic
    switch(PRESENT_STATE){
      case 0:
        STPIND(1, LOW);
        STPIND(2, LOW);
        break;
      case 1:
        STPIND(1, HIGH);
        STPIND(2, LOW);
        break;
      case 2:
        STPIND(1, LOW);
        STPIND(2, LOW);
        break;
      case 3:
        STPIND(1, LOW);
        STPIND(2, HIGH);
        break;
    }

// state transition
// this delay + 0.2 seconds from the BUTTON call make
// up the total 1 second delay of each machine cycle
    PAUSE(800);
    PRESENT_STATE = NEXT_STATE;
  }
}
```

The last example of this Section implements the program using interrupts. The pin that is used for the Button is also INT0, which is programmable edge sensitive and always high priority. We need to make INT0 sensitive to a falling edge, because the Button is LOW when pushed. The comments in the code offer more clarifications.

main.c file:

```
#include  "CS110000.h"

// global variables for inter-process communications
BYTE ON_PIN = 1;
BYTE OFF_PIN = 2;

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

// initialize LEDs
  STPIND(ON_PIN, HIGH);
  STPIND(OFF_PIN, LOW);

// enable INT0 for falling edges
  INTSET(0x80, 0x00, 0x80, INTOTHER);

  while(TRUE){
// set up LEDs and wait
    STPIND(OFF_PIN, LOW);
    TOGGLE(ON_PIN);
    PAUSE(1000);  // 1 second
  }
}
```

zInterruptsH.c file:

```
#include  "CS110000.h"

// RESTATE GLOBAL VARIABLES HERE
extern BYTE ON_PIN;
extern BYTE OFF_PIN;

#pragma code
```

```
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh(void)
{
// TYPE YOUR CODE HERE AFTER THIS LINE

// we use the following command, just to disable
// interrupts
// we do not need to find out explicitly the source of
// the interrupt in this case, because INT0 is the only
// one enabled
// if there were more than one interrupt enabled, we
// would receive the returned value from INTSOURCE into
// a local variable, and test the bits with and
// operations and masks
// then we would use those results in if constructs to
// execute the code that services each pending
// interrupt
  INTSOURCE(INTHIGH);

// process button
// in a larger system, the interrupt may get serviced
// by a function call at this point in the code that
// does that
// since this is a small example, we do it right here
  if(ON_PIN == 1){
    ON_PIN = 2;
    OFF_PIN = 1;
  }
  else{
    ON_PIN = 1;
    OFF_PIN = 2;
  }

// restore interrupts settings
// enable INT0 for falling edges
  INTSET(0x80, 0x00, 0x80, INTHIGH);
}
```

Chapter

5

C Stamp Language and Command Reference

This chapter describes the elements of WC. WC is the combination of the minimum subset of ANSI C necessary to program the C Stamp, and the functionality and commands provided by A-WIT Technologies, Inc. that make the C Stamp so versatile and easy to use. To generate a C Stamp program, you type your program in the Compiler Environment, and build it. Most lines in a WC program end with a semicolon " ; ". You can also put the equivalent of multiple lines of code in a single line of program text by separating them with a semicolon. You can use spaces and extra lines freely, as the compiler simply ignores all spaces and new-line markers when parsing the code. 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 language are in "**`bold`**". Generic syntax is in a "***`bold and italics fixed font`***".

WC Program Structure

The structure template of a WC program file is as follows. Although any comments are optional, beginning comments are encouraged, and comments can be anywhere in the program, as well as extra lines for clarity and readability of your program. The sections are: the beginning comments of the program, included files and other directives, function prototypes, global variable declarations, and the **`main`** function followed by any number of other functions. It is possible that **`main`** is the only function.

```
Beginning comments for program

Included files and other pre-processor directives

Function prototypes (except main)

Global variable declarations

Main function
  Beginning comments for main function
  Local variables for main function
  Code for main function

Other functions

Function 1
  Beginning comments for function 1
  Local variables for function 1
  Code for function 1

Function 2
  Beginning comments for function 2
  Local variables for function 2
  Code for function 2
.
.
.
Function N
  Beginning comments for function N
  Local variables for function N
  Code for function N
```

The file **`CS110000.h`** **MUST** always be included in the **`Included files and other pre-processor directives`** section of the program.

Program Comments

You should be generous with your programs comments, and strive toward self-documenting code. There are two types of comments. These are single line comments and multi line comments. The single line comments are denoted by a double slash "**`//`**". This instructs the compiler to ignore any text after the double slash to the end of the line, so it can be used to comment single lines in their entirety or the rest of a line after some valid code. The multi line comments starts with a slash and an asterisk "**`/*`**" and ends with an asterisk and a slash "**`*/`**". This instructs the compiler to ignore any text between these two pairs of characters, so it can be used to put comments in the middle of a line and to comment any number of lines, including a single line.

Defining and Using Variables

Before you can use a variable in a WC program you must declare it. "Declare" means letting the compiler know that you plan to use a variable, what you want to call it, and how big it is. Here is the syntax for a variable declaration:

```
BYTE VariableName;
```

-- or --

```
int VariableName;
```

where `VariableName` is the name by which you will refer to the variable, and the preceding keywords denote the data type.

A variable can be initialized by following its name with an equal sign and an initial value. For example, this declaration assigns `count` an initial value of 100:

```
int count = 100;
```

Data Types	*Size in Bits*	*Numerical Range*
BIT	1	0 to 1
NIBBLE	4	0 to 15
BYTE	8	0 to 255
WORD	16	0 to 65535
DWORD	32	0 to 4294967295
int	16	-32768 to 32767
long	32	-2147483648 to 2147483647
float	32	6 digits of precision
void		valueless

A variable should be assigned the smallest sized data type that will hold the largest value that the variable will ever hold. If you need a variable to hold the on/off status (1 or 0) of a switch, use a **BIT**. If you need a counter for a loop that will count from 1 to 100, use a **BYTE**; and so on. If you assign a value to a variable that exceeds its size, the excess bits will be lost. For example, suppose you use the **BYTE** variable dog, and write dog = 260 (100000100 binary). What will dog contain? It will hold only the lowest 8 bits of 260 (00000100 binary = 4 decimal. In the same way, if you assign a variable type of a given size to another of a smaller size, the excess bits will be lost. However, you can always assign a variable type or value of a smaller size to a variable type of a larger size without loss of data.

Predefined Values

WC has the following predefined values that can be used in logical expressions, to set pin values, and for the purpose of making easier mathematical calculations:

Value Name	*Value Type*	*Value*
TRUE	**BIT**	1
FALSE	**BIT**	0
ONE	**BIT**	1
ZERO	**BIT**	0
HIGH	**BIT**	1
LOW	**BIT**	0
PI	**float**	3.1416
TWOPI	**float**	6.2832
HALFPI	**float**	1.5708
NULL	**void**	valueless

Rules of Variable Names

Variable names are strings of letter or digits from one to several characters in length. A digit cannot begin a name. Variable names can be a maximum of 31 characters. The underscore may also be used as part of the variable name for clarity, as in `first_time`. WC is case sensitive for its keywords, commands, function names, and variable names. That means that uppercase and lowercase are different. For example `test` and `TEST` are two different variable names. Some programmers define all **`float`** type variables in lowercase and all others in uppercase.

Variable names cannot be reserved words that the language uses for data types, predefined values, keywords, commands, or function names.

Defining Arrays

You can also define multipart variables called arrays. An array is a group of variables of the same size, and sharing a single name, but broken up into numbered cells, called elements. You may declare arrays of any data type. This is the general form of a singly dimensioned array:

```
type var-name[size];
```

where ***size*** specifies the number of elements in the array. For example, to declare an integer array `x` of 100 elements, you would write:

```
int x[100];
```

This will create an array of integers that is 100 elements long, with the first element being 0 and the last one being 99. Numbering always starts at 0 and ends at ***size***-1.

```
x[3] = 57;
```

will set the fourth element of the `x` array to 57. The real power of arrays is that the index value can be a variable itself. For example:

```
BYTE myBytes[10]; // Define 10-byte array
NIBBLE idx; // Define 4-bit var
// Repeat with idx = 0, 1, 2...9
for (idx = 0; idx < 10; idx = idx + 1){
  // Write idx * 13 to each cell
  myBytes[idx] = idx * 13;
}
```

Multidimensional arrays are declared by placing the additional dimensions inside additional brackets. For example, you would write the following statement to declare a 10 by 20 integer array:

```
int x[10][20];
```

Multidimensional arrays are ordered in row-major format. This has the following implications:

- For two-dimensional array, the first index specifies the row and the second the column.
- Since elements are stored in a one-dimensional linear memory, all the column elements of a row are stored first, then the next row, and so on, as shown in the tables below.

Two-dimensional arrays can be initialized with multiple lines or in a single line, as long as you keep in mind the organizational order of the elements in the array.

For example, the 2-rows by 3-columns **BYTE** array y can be initialized as

```
BYTE y[2][3] = {1, 2, 3,
                4, 5, 6};
```

or as

```
BYTE y[2][3] = {1, 2, 3, 4, 5, 6};
```

These two initializations will have the same effect, because the line-break has no syntactical meaning and the row-major order is comprehended.

We can visualize this array as a matrix

y ***Array Elements***	***Column 1***	***Column 2***	***Column 3***
Row 1	1	2	3
Row 2	4	5	6

but in memory, the array would be organized linearly as

Memory Location	***Value***
N	1
N + 1	2
N + 2	3
N + 3	4
N + 4	5
N + 5	6

so if we wanted to access the element in the second row and first column, we would write

```
x = y[1][0];  // x = 4
```

A word of caution about arrays: All arrays indexes begin at 0 and WC provides no array bounds-checking. Such safety checks are your responsibility. Bounds are the lower and upper limits of the index or subscript variable of the array. For instance, in the example above, `myBytes` is a 10-cell array. Allowable index numbers are 0 through 9. If your program exceeds this range, the compiler will not respond with an error message. Instead, it will access the next memory location past the end of the array. If you are not careful about this, it can cause all sorts of bugs. It is up to the programmer (you!) to prevent bugs.

Constants and Number Representations

Variables of type **`const`** may not be changed by your program during execution. For example,

```
const int a;
```

will create an integer called `a` that may not be modified by your program. It can however be used in other types of expressions. A const variable will receive its value either from an explicit initialization or by some hardware-dependent means.

WC constants can be of any of the data types. The way each constant is represented depends upon its type.

In your programs, you may express a number in various ways, depending on how the number will be used and what makes sense to you. By default, the compiler recognizes numbers like 0, 99 or 62145 as being in our everyday decimal (base-10) system. However, you may also use hexadecimal (base-16; also called hex) or octal (base-8). Since the symbols used in decimal, hex and octal numbers overlap (e.g., 0 through 7 are used by all; 0 through 9 apply to both decimal and hex) the compiler needs prefixes to tell the numbering systems apart, as shown below:

```
99 // Decimal (no prefix)

0x1A6 // Hex (prefix ‘0x’ required)

012 // Octal (prefix ‘0’ required)
```

The compiler also automatically converts quoted text into ASCII codes. For example:

```
BYTE LetterA = ‘A’; // ASCII code for A (65)

BYTE Cheers = 3;

BYTE Hex128 = 0x80;

BYTE OctalNumber = 0101;
```

-- or --

```
const BYTE LetterA = ‘A’; // ASCII code for A (65)

const BYTE Cheers = 3;

const BYTE Hex128 = 0x80;

const BYTE OctalNumber = 0101;
```

Structures

A *structure* is a collection of variables that are grouped and referenced with one name. This is the general form of a structure declaration:

```
struct tag{
  element 1;
  element 2;
  .
  .
  .
} struct-var list;
```

The **tag** is essentially the type name of the structure.

For example, the following structure has two elements: **myval1**, a **BYTE** array, and **mybal2**, a floating-point number:

```
struct mystruct{
  BYTE myval1[80];
  float myval2;
} robot;
```

To reference individual structure elements, use the dot (.) operator. For example, this statement accesses the **myval2** element of **robot**:

```
robot.myval2
```

Unions

A *union* is a collection of variables that share the same memory space, and therefore that shared memory space can be referenced by the different variable names, according to the users needs. This is the general form of a union declaration:

```
union tag{
  element 1;
  element 2;
  .
  .
  .
} union-var list;
```

The ***tag*** is essentially the type name of the union.

For example, the following union has two elements: **myval1**, a **BYTE** array, and **mybal2**, a floating-point number:

```
union myunion{
  BYTE myval1[4];
  float myval2;
} mydata;
```

To reference individual union elements, use the dot (.) operator. For example, this statement accesses the **myval2** element of **mydata**:

```
mydata.myval2
```

Furthermore, mydata.myval2 and the four bytes of myval.myval1 share the same memory space as shown in the figure below. Remember that a floating point number is 32 bits or 4 bytes.

myval.myval1[0]	**myval.myval1[1]**	**myval.myval1[2]**	**myval.myval1[3]**
myval.myval2			

Being able to form unions of variables is useful, for example, when you want to decompose a floating point number into its bytes to send them serially or store them in the C Stamp EEPROM. Let us say that you have declared the union above and that you want to store a floating point number in the C Stamp EEPROM locations 0 through 3. This could be accomplished with the program below.

```
#include  "CS110000.h"

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

// floating point number
  float f = 1.2;
// counter
  NIBBLE i;

// declare union
  union myunion{
    BYTE myval1[4];
    float myval2;
  } mydata;

// store f in union
  mydata.myval2 = f;

// store f in the C Stamp EEPROM
  for(i=0; i<4; i=i+1) WRITE(i, mydata.myval1[i]);

  STOP();
}
```

Taking Advantage of the Large C Stamp RAM

By default, the compiler for the C Stamp tries to fit all the program data into a space of 120 bytes, and if you have a lot of variables, or large arrays like strings of characters in your program, you may get a "stack frame too large" error during compilation. This is easily solved by forcing the compiler to use the very large RAM space present in the C Stamp. To do this, the optional **RAM** data type modifier is used when declaring variables or arrays. For example:

```
RAM int a;
```

will force the compiler to put variable `a` anywhere in the 2K RAM present in the C Stamp, instead of trying to fit it in the preferred 256 bytes space.

The WC **RAM** modifier can be applied to any of the data types.

For example, if you do not have many variables in your program, you may declare the following variables like shown below, or you can use the **RAM** modifier and force the compiler to put them in RAM when the program is run.

```
BYTE LetterA = 'A'; // ASCII code for A (65)

BYTE Cheers = 3;

BYTE Hex128 = 0x80;

BYTE OctalNumber = 0101;
```

-- or --

```
RAM BYTE LetterA = ‘A’; // ASCII code for A (65)

RAM BYTE Cheers = 3;

RAM BYTE Hex128 = 0x80;

RAM BYTE OctalNumber = 0101;
```

The C Stamp Data RAM is abstracted into 9 groups as shown in the table below.

Group Name	***Start Address in hex***	***End Address in hex***	***Size in Bytes***	***Comments***
N/A	0x0	0x5F	96	Tool Priority
GPR0DATA	0x60	0xFF	160	
GPR1DATA	0x100	0x1FF	256	
GPR2DATA	0x200	0x2FF	256	
BIGDATA	0x300	0x4FF	512	
GPR5DATA	0x500	0x5FF	256	
N/A	0x600	0x6FF	256	Tool Priority
GPR7DATA	0x700	0x7F3	244	
N/A	0x7F4	0x7FF	12	Tool Priority

Normally a programmer needs not be concerned with the memory mapping described in the table above, as the software development tool makes all memory allocation automatically. However, these automatic memory allocations try to fit all data for each function in single groups. This practice works for most usages, but if you have large arrays that make the memory allocation spill over to another group, the software development tool will issue an error message to the effect of:

```
Error – section ‘.udata_xxx.o’ can not fit the section.
```

In that case, the memory group for large arrays has to be explicitly indicated using a **`#GROUP <Group Name>`** directive, as shown in the example below. The **`<Group`**

Name> is from the table above, groups that have "Tool Priority" should be avoided, and if an array is larger than 256 bytes, it should go in the group **BIGDATA**. Other two considerations that must be taken is that arrays that are explicitly assigned to memory groups must be declared as global data, and referenced via an index variable, and not with a constant (e.g. use `array[index]`, and not `array[2]`.

C-Stamp Example:

```
//  The following allocates big arrays totaling a
//  whopping 1200 bytes of data memory
#GROUP GPR1DATA
RAM BYTE myBuffer1[200];
#GROUP
#GROUP GPR2DATA
RAM BYTE myBuffer2[200];
#GROUP
#GROUP BIGDATA
RAM BYTE myBuffer3[400];
#GROUP
#GROUP GPR5DATA
RAM BYTE myBuffer4[200];
#GROUP
#GROUP GPR7DATA
RAM BYTE myBuffer5[200];
#GROUP
```

Taking Advantage of the Large C Stamp Program Memory

When working with large arrays of constant data strings, **ROM** is better. In this case, the data is stored in program memory, and you can take advantage of the large program memory (up to 32K BYTES including your code and your large constant data). To do this, the optional **ROM** data type modifier is used when declaring variables or arrays. For example,

```
ROM BYTE LetterA = 'A'; // ASCII code for A (65)
```

will force the compiler to put variable LetterA anywhere in the 32K Program Memory along with your code.

WC **ROM** modifier can be applied to any of the data types.

For example, if you do not have many constants in your program, you may declare the following variables like shown below, or you can use the **ROM** modifier and force the compiler to put them in Program Memory.

```
BYTE Cheers = 3;

BYTE Hex128 = 0x80;

BYTE OctalNumber = 0101;
```

-- or --

```
ROM BYTE Cheers = 3;

ROM BYTE Hex128 = 0x80;

ROM BYTE OctalNumber = 0101;
```

Functions

A WC program is a collection of one or more user-defined functions. One of the functions must be called **main** because it is at this function where execution will begin. Traditionally, **main()** is the first function in a program; however, it could go anywhere in the program. This is the general form of a WC function:

```
return_type function_name(parameter list)
{
  body of function
}
```

The parameter list is a comma-separated list of local variables (local to the function itself) that will receive any arguments passed to the function. If the function has no parameters, then no parameter declaration is needed. For example, this function has two integer parameters called **i** and **j**, and a float parameter called **count**:

```
void fn1(int i, int j, float count)
{  …
```

Notice that you must declare each parameter separately.

Functions terminate and return automatically to the calling procedure when the last brace is encountered. You may force a return prior to that by using the **return** statement.

All functions (except those declared as **void**) return a value. The type of the return value must match the type declaration of the function. The **void** type is used to explicitly declare functions that return no value. It is also used to explicitly denote that a function takes no arguments. For example the code fragment "**void** main(**void**)" means that the function main returns no value, and it also takes no arguments.

Prototypes

It is necessary to declare functions (other than main) prior to using it. This is done in the global definition area of the program, which is before any functions are coded. A general form of a prototype is shown here:

***type name*(*parameter list*);**

In essence, a prototype is simply the return type, name, and parameter list of a function definition, followed by a semicolon.

The example here shown how the function **fn()** is prototyped.

```
float fn(float x); /* prototype */

void main(void)
{
  .
  .
  .
  y = fn(0.23);
  .
  .
  .
}

float fn(float x)
{
  return x * 3.1416;
}
```

In addition to telling the compiler about the return type of a function, a function prototype also tells the compiler the number and type of function parameters.

The Scope and Lifetime of Variables

WC has two general classes of variables: *global* and *local*. A global variable is available for use by all functions in the program, while a local variable is known and used only by the function in which it is declared.

Math and Operators

For non-floating numbers, the C Stamp performs all math operations by the rules of positive integer math. That is, it handles only whole numbers, and drops any fractional portions from the results of computations. The C Stamp handles negative numbers using two's complement rules.

WC has the following arithmetic operators:

Operator	***Action***
-	Subtraction, unary minus
+	Addition
*****	Multiplication
/	Division
%	Modulus division

The **-**, **+**, *****, and **/** operators behave in the expected fashion. The **%** operator returns the remainder of an integer division.

These operators have the following order of precedence:

Precedence	***Operators***
Highest	**-** (unary minus)
	*** / %**
Lowest	**- +**

Operators on the same precedence level are evaluated left to right, so WC follows the rules of conventional algebra.

Operators that take two arguments are called "binary" operators, and those that take only one argument are called "unary" operators. Please note that the term "binary operator" has nothing to do with binary numbers; it's just an inconvenient coincidence that the same word, meaning 'involving two things' is used in both cases. The minus sign (-) is a bit of a hybrid. It can be used as a binary operator, as in 8-2 = 6, or it can be used as a unary operator to represent negative numbers, such as -4.

Relational and Logical Operators

The relational and logical operators are used to produce true/false results, and are often used together. In WC, *any* non-zero value is considered as true. The only value that is considered as false is 0. The relational and logical operators are listed in the following tables.

Relational Operators:

Operator	***Meaning***
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal
!=	Not Equal

Logical Operators:

Operator	***Meaning***
&&	AND
||	OR
!	NOT

The relational operators are used to compare two values. The logical operators are used to connect two values or, in the case of NOT, to reverse a value. The precedence of these operators is as follows:

Precedence	***Operators***

Precedence	***Operators***
Highest	**!**
	> >= < <=
	== !=
	&&
Lowest	**||**

The Bitwise Operators

WC provides operators that manipulate the actual bits inside a variable. The bitwise operators cannot be used in floating point variables. They are listed here:

Operator	***Meaning***
&	AND
|	OR
^	XOR
~	Complement
>>	Right shift by specified amount
<<	Left shift by specified amount

Assignment Operator

In WC, the assignment operator is the single equal sign.

Operator Precedence Summary

The table below lists the precedence of all WC operators. Please note that all operators, except the unary operators, associate from left to right. The unary operators associate from right to left.

Precedence	***Operators***
Highest	**`() []`**
	`! ~ -`
	`* / %`
	`+ -`
	`<< >>`
	`> >= < <=`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
	`||`
Lowest	**`=`**

break

break is used to exit from a **do**, **for**, or **while** loop, bypassing the normal loop condition. It is also used to exit from a **switch** statement.

An example of a **break** in a loop is shown here:

```
while (x < 100){
  x = x + 1;
  if (x > 50) break; /* will not let x reach 52,
                        terminate when x is 51 */
  process (x);
}
```

Here, if **x** is 51, the loop is terminated. The if statement will not let **x** reach 52.

A **break** always terminates the innermost **for**, **do**, **while**, or switch statement, regardless of the way these might be nested. In a **switch** statement, **break** effectively keeps program execution from falling through to the next **case**. (See "**switch**" for details.)

case

See "**switch**".

continue

continue is used to bypass the portions of code in a loop that follows it and force the conditional test to be performed. For example, the following **while** loop will simply **process** the values of **x** that are less than 51; however, x will reach a value of 100.

```
while (x < 100){
  x = x + 1;
  if (x > 50) continue; /* will only process values of
                           x that are less than 51 */
  process (x);
}
```

The call to **process()** will only occur when **x** is less than 51.

default

default is used in the **switch** statement to signal a default block of code to be executed if no **case** matches are found in the **switch**. (See "**switch**".)

do

The do loop is one of the three loop constructs available in WC. This is the general form of the do loop:

```
do {
  statement block
} while (condition);
```

If only one statement is repeated, the braces are not necessary, but they do add clarity to the statement.

The do loop is the only loop in WC that will always have at least one iteration, because the condition is tested at the bottom of the loop. The fragment shown here will store **ch** until **fp** is zero, but it will always store the first **ch** regardless of the value of **fp**.

```
do {
  ch = fp;
  store (ch);
} while (!fp);
```

else

See "**`if`**".

extern

`extern` is a data type modifier used to tell the compiler that a variable is declared elsewhere in the program. This is used in conjunction with the separately compiled interrupt file(s) that share(s) the same global data as the main file. In essence, it notifies the compiler about the existence of a variable without re-declaring it.

As an example, if `first` were declared in the main file as a global integer variable, and it was used in an interrupt file, then the following statement would have to be present at the beginning of the interrupt file:

```
extern int first;
```

for

The **`for`** loop allows automatic initialization and update of a counter variable. This is the general form:

```
for (initialization; condition; update) {
  statement block
}
```

If the ***`statement block`*** is only one statement, the braces are not necessary.

It is important to understand that if the ***`condition`*** is false to begin with, the body of the **`for`** will not execute even once.

The following code will store the squared values of 0 to 9 in an array.

```
for (t = 0; t < 10; t = t + 1) {
  x[t] = t * t;
}
```

if

This is the general form of the **`if`** statement:

```
if (condition) {
  statement block 1
}
else {
  statement block 2
}
```

If single statements are used, the braces are not needed. The **`else`** is optional.

The ***`condition`*** may be any expression. If that expression evaluates to any value other than 0, then ***`statement block 1`*** will be executed; otherwise, if it exists, ***`statement block 2`*** will be executed.

The following fragment checks for the letter **`q`**, which terminates the function.

```
if (ch == ‘q’) {
  return;
}
else {
  proceed ();
}
```

return

The **`return`** statement forces a return from a function and can be used to transfer a value back to the calling routine.

For example, the following function returns the product of its two integer arguments.

```
mul (int a, int b)
{
  return (a * b);
}
```

Keep in mind that as soon as **`return`** is encountered, the function will return, skipping any other code that may be in the function after it.

Remember also that a function can contain more than one **`return`** statement.

sizeof

The **`sizeof`** compile time operator return the length of the variable or type it precedes in bytes. The variable or type must be enclosed in parentheses.

For example, given the following:

```
NIBBLE a, b;
int i;
a = sizeof(int);
b = sizeof(i);
```

both `a` and `b` will have a value of 2.

`sizeof`'s principal use is to generate string lengths to be passed to **`SEROUT`** and **`SEROUT2`**. For example:

```
RAM BYTE prompt_1[] = "\n\rFile: ";
:
// later on
SEROUT(0, 0, 9.6, 0, 8, NOPAR, 0, prompt_1,
       sizeof(prompt_1));
// or
SEROUT2(0, 0, 9.6, 0, 8, NOPAR, 0, prompt_1,
        sizeof(prompt_1), pin);
```

struct

The **`struct`** keyword is used to create complex or conglomerate variables, called *structures*, which are made up of one or more elements. This is the general form of a structure:

```
struct tag{
  type element 1;
  type element 2;
  .
  .
  .
  type element n;
} struct-var list;
```

The *`tag`* is essentially the type name of the structure. The individual elements are referenced by using the dot (.) operator.

For example, the following structure contains a string called **`message`** and two bytes called **`length`** and **`severity`**. It also declares a variable called **`error_message_1`**.

```
struct error_message{
  BYTE message[16];
  BYTE length;
  BYTE severity;
} error_message_1;
```

switch

The **switch** statement is WC's multi-way branch statement. It is used to route execution one of several different ways. This is the general form of the **switch** statement:

```
switch (control_var) {
  case constant 1:
    statement sequence 1
    break;
  case constant 2:
    statement sequence 2
    break;
  .
  .
  .
  case constant n:
    statement sequence n
    break;
  default:
    default statements
}
```

Each statement sequence may be from one to several statements long. The **default** portion is optional.

switch works by checking the ***control_var*** against the constants. If a match is found, that sequence of statements is executed. If the statement sequence associated with the **case** that matches the value of ***control_var*** does not contain a **break**, execution will continue on into the next **case**. Put differently, from the point of the match, execution will continue until either a **break** statement is found or the **switch** ends. If no match is found and a **default** case exists, its statement sequence is executed. Otherwise, no action takes place. The following example processes a command selection.

```
ch = getcommand();

switch (ch) {
  case ‘e’:
    erase();
    break;
  case ‘r’:
    read();
    break;
  case 'w':
    write();
    break;
  default:
    return;
}
```

union

union is used to assign two or more variables to the same memory location. The form of the definition and the way the . (dot) operator reference an element are the same as for **struct**. This is the general form:

```
union tag{
  type element 1;
  type element 2;
  .
  .
  .
  type element n;
} union-var list;
```

The *tag* is essentially the type name of the union. For example, this creates a union between a float and a character string and creates one variable called my_var.

```
union my_union{
  BYTE message[15];
  float offset;
} my_var;
```

The four bytes of my_var.offset will share the same memory locations as the first four bytes of my_var.message.

void

The **void** type is used to explicitly declare functions that return no value. It is also used to explicitly denote that a function takes no arguments.

while

The **while** loop has the following general form:

```
while (condition) {
  statement block
}
```

If a single statement is the object of the **`while`**, then the braces may be omitted.

`while` tests its ***`condition`*** at the top of the loop. Therefore, if the ***`condition`*** is false to begin with, the loop will not execute even once. The ***`condition`*** may be any expression.

An example of a **`while`** loop is shown below. It will transfer 100 characters from an array to another array.

```
BYTE y[256];

t = 0;

while (t < 100) {
  y[t] = x[t];
  t = t + 1;
}
```

#include

The **`#include`** preprocessor directive instructs the compiler to read and compile another source file. It takes these general forms:

`#include “`*`filename`*`”`

`#include <`*`filename`*`>`

The source file to be read in must be enclosed in double quotes or angle brackets. If the ***`filename`*** is enclosed by angle brackets, the file is searched for in a manner defined by the creator of the compiler. This means searching some special directory set aside for include files. If the ***`filename`*** is enclosed in quotes, the file is looked for in the current working directory. If the file is not found, the search is repeated as if the ***`filename`*** had been enclosed in angle brackets.

`#include`’s may be nested within other included files.

For example,

```
#include “CS110000.h”
```

will instruct the compiler to read and compile the header for the C Stamp CS110000.

abs

int abs(int num);

The **abs** function returns the absolute value of the integer **num**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
// Declare two variables, one to evaluate in the abs
// function and one to compare with the result of the
// abs function
  int EVALUATE;
  int COMPARE;

// Set the variable EVALUATE to a negative integer
// ex: -1
  EVALUATE = -1;
// Set the variable COMPARE to the abs value of the
// integer previously used
  COMPARE = 1;

  while(1){  // Continuous loop
    if(BUTTON(38, LOW, HIGH, 5)){ // Check state of
                                  // utility button
// Find the absolute value of EVAULATE and then
// store the value in the variable EVAULATE
      EVALUATE = abs(EVALUATE);
// Test to see if COMPARE is equal to the absolute
// value of EVALUATE
      if(COMPARE == EVALUATE)
        STPIND(46, HIGH); // If COMPARE and the
// absolute value of EVALUATE are the same then light
// the LED attached to pin 46
      else
        STPIND(42, HIGH); // If COMPARE and the
// absolute value of EVALUATE are not the same, light
// the LED attached to pin 42
      break; // This breaks from the while loop
    }
  }
}
```

ANALOGIN

`float ANALOGIN(BYTE pin, NIBBLE tpins, NIBBLE refmode, float VREFM, float VREFP);`

The **`ANALOGIN`** function samples the indicated analog input **`pin`** (pins 9 through 20) and returns a floating-point digital value corresponding to the value of the analog voltage at the analog input **`pin`**. The BANDWIDTH of the sampled signal **MUST** be less than or equal to 3.4 KHz, and the signal **MUST** be bounded by **`VREFM`** and **`VREFP`**. If unsuccessful, the function returns -1.

`pin` is a variable/constant/expression that specifies the analog input pin to be sampled.

`tpins` is a variable/constant/expression that specifies the total number of analog input pins being used in the project. These pins will be set to input mode. Depending on the number of analog inputs required by your project, analog input pins (pins 9 through 20) **MUST** be used according to the table below (D = digital I/O. A = analog input). This also includes the case where one or two external voltage references are used.

Number of Analog Inputs Required	***C Stamp Pins Usage***											
`tpins`	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	D	D	D	D	D	D	D	D	D	D	D	D
1	D	D	D	D	D	D	D	D	D	D	A	D
2	D	D	D	D	D	D	D	D	D	A	A	D
3	D	D	D	D	D	D	D	D	A	A	A	D
4	D	D	D	D	D	D	D	A	A	A	A	D
5	D	D	D	D	D	D	D	A	A	A	A	A
6	D	D	D	D	D	D	A	A	A	A	A	A
7	D	D	D	D	D	A	A	A	A	A	A	A

Number of Analog Inputs Required **`tpins`**	***C Stamp Pins Usage***											
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
8	D	D	D	D	A	A	A	A	A	A	A	A
9	D	D	D	A	A	A	A	A	A	A	A	A
10	D	D	A	A	A	A	A	A	A	A	A	A
11	D	A	A	A	A	A	A	A	A	A	A	A
12	A	A	A	A	A	A	A	A	A	A	A	A

`refmode` is a variable/constant/expression that specifies the configuration of the voltage references for the analog-to-digital conversion according to the table below.

`refmode` ***Value***	**A/D VREFM**	**A/D VREFP**
0	Internal VREFM = 0 V	Internal VREFP = 5.0 V
1	Internal VREFM = 0 V	External VREFP = 3 V to 5.3 V
2	External VREFM = -0.3 V to 2 V	Internal VREFP = 5.0 V
3	External VREFM = -0.3 V to (VREFP – 3 V)	External VREFP = (VREFM + 3 V) to 5.3 V

VREFM is a variable/constant/expression that specifies the VREFM value if a low external voltage reference is used. If **refmode** is 0 or 1, this argument is ignored, but some argument still needs to be given to the function.

VREFP is a variable/constant/expression that specifies the VREFP value if a high external voltage reference is used. If **refmode** is 0 or 2, this argument is ignored, but some argument still needs to be given to the function.

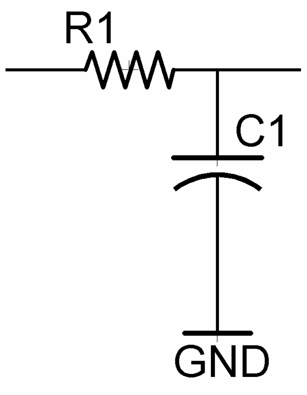
EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float value_pin_19; // Declare variable value_pin_19
                      // of type float to store value
                      // returned by ANALOGIN function
  float value_pin_18; // Declare variable value_pin_18
                      // of type float to store value
                      // returned by ANALOGIN function

// Call ANALOGIN function, store value returned in
// value_pin_19
  value_pin_19 = ANALOGIN(19, 2, 0, 0, 5);

// Call ANALOGIN function, store value returned in
// value_pin_18
  value_pin_18 = ANALOGIN(18, 2, 0, 0, 5);

// Check if value_pin_19 is 3.75 volts +/- .05 volts
  if((value_pin_19 <= 3.8) && (value_pin_19 >= 3.7)){
    STPIND(45, HIGH); // Light LED attached to pin 45
    PAUSE(2000);      // Pause for 2 seconds
    STPIND(45, LOW); // Turn off LED attached to pin 45
    PAUSE(2000);     // Pause for 2 seconds
  }

// Check if value_pin_18 is 1.25 volts +/- .05 volts
  if((value_pin_18 <= 1.3) && (value_pin_18 >= 1.2)){
    STPIND(45, HIGH); // Light LED attached to pin 45
    PAUSE(2000);      // Pause for 2 seconds
    STPIND(45, LOW); // Turn off LED attached to pin 45
    PAUSE(2000);     // Pause for 2 seconds
  }
  END(); // Stop program, CStamp enters low power mode
}
```

ANALOGOUT

`BIT ANALOGOUT(BYTE pin, float V);`

The **`ANALOGOUT`** function converts a floating point digital value to analog output via pulse-width modulation on one the PWM outputs (pin 3 or 27). The maximum real-time BANDWIDTH of the digital signal being converted to analog is 19 KHz. To actually do the Digital to Analog conversion, the PWM output pin being used needs to drive a Low-Pass-Filter (LPF). This LPF is shown in the following figure. The C Stamp PWM pin will be connected to the left of R1, and the analog voltage will be available to the right of R1 (i.e. at C1). To drive DC motors; however, it is not necessary to use an LPF, because the inertial dynamics that are inherent to DC motors act as a built-in LPF in the motor. The values for the LPF components are shown in the table below. If the values for the components of the LPF are not available, some other RC combination can be used, as long as the product of the component values R x C = 1.0 x 10^{-6}.



Low Pass Filter Components Values	
R	1 KΩ (or as close as possible)
C	0.001 µF (or as close as possible)

`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to output mode.

V is a variable/constant/expression (0 – 5 V) that specifies the analog output level (0+ to **HIGH** level).

If **V** is 0, then the PWM output is disabled, if it had been previously enabled, meaning that the PWM output will be 0 V.

If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT SUCCESS; // Declare variable SUCCESS of type BIT

// Call function ANALOGOUT, store value returned in
// SUCCESS
  SUCCESS = ANALOGOUT(3, 2);

// Check if ANALOGOUT executed successfully
  if(SUCCESS){
    STPIND(45, HIGH); // Light LED attached to pin 45
  }

  END(); // Stop program, C Stamp enters low power mode
}
```

atoW

WORD atoW(BYTE a[], BYTE radix);

The **atoW** function converts the null terminated **BYTE** array **a** into its numeric equivalent, and returns the result. The base of the input string is given by **radix**, which may be 2, 10, or 16.

Be sure to call **atoW** with an array **a** that represents a number no larger than what can be held in a **WORD**: [0 to 65535].

Note that if any character in the ASCII string is not numeric, **atoW** will return 0, as it checks for this, so you must validate the string before hand and make sure that all characters are numeric.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  WORD n;
  BYTE a[] = "0512";

  n = atoW(a, 10); // n would be equal to 512

  END(); // Stop program, C Stamp enters low power mode
}
```

BUTTON

`BIT BUTTON(BYTE pin, BIT downstate, BIT targetstate,`
`           WORD delay);`

The **BUTTON** function monitors and returns **TRUE** if the processing of a pushbutton input is successful resulting from the button being pushed and reaching the **targetstate**; otherwise it returns **FALSE**. Button circuits may be active-low or active-high.

pin is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to input mode.

downstate is a variable/constant/expression (0 or 1) that specifies which logical state occurs when the button is pressed.

targetstate is a variable/constant/expression (0 or 1) that specifies which state the button should be in for the function to return **TRUE**. (0=not pressed, 1=pressed)

delay is a variable/constant/expression (1 – 65535) that specifies for how long the button will be monitored. The unit of time for **delay** is 0.2 second.

If the button is not pressed, then the function returns **FALSE**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT CHECK_BUTTON; // Declare variable CHECK_BUTTON to
                    // store the utility button's state
  STPIND(43, HIGH); // Light LED attached to pin 43 to
                    // show program is running.

// Use the BUTTON function to check if the utility
// button has been pushed the value of the BUTTON
// function is the stored in the variable CHECK_BUTTON
// If after 10 seconds the utility button has not been
// pushed CHECK_BUTTON will have a value of FALSE (0)
  CHECK_BUTTON = BUTTON(37, LOW, LOW, 50);

  if(CHECK_BUTTON){ // Tests to see if CHECK_BUTTON is
// TRUE or FALSE (1 or 0 respectively)
    STPIND(46, HIGH); // If CHECK_BUTTON is TRUE light
                      // the LED attached to pin 46
  }
  else{
    STPIND(45, HIGH); // If CHECK_BUTTON is FALSE light
                      // the LED attached to pin 45
  }
  END(); // The END function stops the program from
         // continuously looping
}
```

BYTEIN

BYTE BYTEIN(BYTE pins[]);

The **BYTEIN** function sets the pins in the array **pins** to digital input mode and returns the BYTE value at those pins. **pin[0]** is the least significant bit (LSB) of the desired BYTE and **pin[7]** is the most significant bit (MSB). User **MUST** ensure that the **pins** array has at least 8 elements.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BYTE pins[8];   // Declare array pins of type BYTE
  BYTE BYTE_READ; // Declare variable BYTE_READ of type
                  // BYTE

/* Set array pins to the appropriate pins on which data
is placed, where 12 is the LSB and 19 is the MSB */
  pins[0] = 12; pins[1] = 13;
  pins[2] = 14; pins[3] = 15;
  pins[4] = 16; pins[5] = 17;
  pins[6] = 18; pins[7] = 19;

// Call function BYTEIN and store value in BYTE_READ
  BYTE_READ = BYTEIN(pins);

// Check if value returned by BYTEIN function is
// hexidecimal AA or binary 10101010
  if(BYTE_READ == 0xAA){
    STPIND(45, HIGH); // Light LED attached to pin 45
  }

// Stop program, CStamp enters low power mode
  END();
}
```

BYTEOUT

`BIT BYTEOUT(BYTE value, BYTE pins[]);`

The **`BYTEOUT`** function sets the pins in the array **`pins`** to digital output mode and to **`value`**. **`pin[0]`** corresponds to the LSB of **`value`** and **`pin[7]`** to the MSB. User **MUST** ensure that the **`pins`** array has at least 8 elements. If the operation is successful, it returns a value of **`TRUE`**; otherwise, it returns a value of **`FALSE`**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BYTE pins[8]; // Declare array pins of type BYTE with
                // 8 elements
  BIT SUCCESS;  // Declare variable SUCCESS of type BIT
                // to store value returned by BYTEOUT
                // function

/* The array pins is defined element by element so that
pin 8 is the LSB and pin 15 is the MSB */
  pins[0] = 8;  pins[1] = 9;
  pins[2] = 10; pins[3] = 11;
  pins[4] = 12; pins[5] = 13;
  pins[6] = 14; pins[7] = 15;
// Call function BYTEOUT, store value returned in
// SUCCESS
  SUCCESS = BYTEOUT(0xAA, pins);

// Check if function BYTEOUT executed successfully
  if(SUCCESS){
    STPIND(45, HIGH); // Light LED attached to pin 45
  }

// Stop program, CStamp enters low power mode
  END();
}
```

CLPINPU

`void CLPINPU(void);`

The **`CLPINPU`** function clears (disables) the built-in pull-up resistors available in the module (Pins 31 – 38).

COMPARE

`BIT COMPARE(NIBBLE comparator, NIBBLE mode);`

The **`COMPARE`** function enables or disables a comparator, compares voltages on its pins, and returns the comparison result.

`comparator` is a variable/constant/expression (1 or 2) that denotes on which comparator the function operates.

`mode` is a variable/constant/expression (0 – 2) that enables or disables the comparator and determines if the optional comparator output pin is enabled or not.

The table below shows an explanation of the mode values.

`mode` *Values*	*Function Behavior*
0	Disables the comparator
1	Enables the comparator with comparator output as result output If the voltage at the comparator (+) input is greater than the voltage at its (-) input, returns **`TRUE`** and outputs a **`HIGH`** at the comparator output pin; otherwise it returns **`FALSE`** and outputs

mode *Values*	*Function Behavior*
	a **LOW** at the comparator output pin
2	Enables the comparator without comparator output as result output If the voltage at the comparator (+) input is greater than the voltage at its (-) input, returns **TRUE**; otherwise it returns **FALSE**

If Comparator 1 is disabled, then Comparator 2 gets disabled too. If Comparator 2 is enabled, then Comparator 1 gets enabled too, although it does not have to be used. This means that if you need only one comparator, you should use Comparator 1 first and then use Comparator 2 if a second is necessary. If you need only one comparator, then you can use Comparator 1 and disable Comparator 2. The comparators operate independently of the execution speed of the module and will continue to run and update their outputs if **mode = 1**, even during sleep mode. To avoid additional current draw during sleep mode, disable the comparators before entering sleep mode.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT RESULT; // Variable RESULT of type BIT to store
              // value returned by function COMPARE

  STPIND(42, LOW); // Set LED attached to pin 42 LOW
  STPIND(46, LOW); // Set LED attached to pin 46 LOW

  RESULT = COMPARE(1, 1); // Call function COMPARE,
// activate comparator 1 on mode 1, store the logical
// value returned in the variable RESULT
  if(RESULT){ // Check to see if + or - pin is higher
// (TRUE or '1' for + pin FALSE or '0' for - pin)
    STPIND(42, HIGH); // If + pin higher, Light LED
                      // attached to pin 42
  }
  else{
    STPIND(46, HIGH); // If - pin higher, Light LED
                      // attached to pin 46
  }
  PAUSE(10); // Pause program for 10 milliseconds
             // before looping.
}
```

COUNT

WORD COUNT(BYTE pin, WORD duration);

The **COUNT** function counts the number of cycles (0-1-0 or 1-0-1) on the specified Capture **pin** during the **duration** time frame and returns that number.

pin is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to input mode.

duration is a variable/constant/expression (1 – 65535) specifying the time during which to count. The unit of time for **duration** is 1 millisecond.

If an error occurs or no pulse is detected, the function returns **FALSE**.

NOTE:

This function works ONLY on "Capture" pins. To count pulses on non-Capture pins, see **COUNT2()** below.

The Capture pins are names for a couple of pins; Capture1 and Capture3. They are defined in the pin table and referenced in the **COUNT** and **PULSIN** functions.

The functions that use Capture pins are using dedicated hardware in the C Stamp to "Capture" edges and count pulses or time. Therefore they have 2.75 times better resolution than their counterpart functions terminated in a "2". The functions that use Capture pins have a resolution of 0.8 µS and the others a resolution of 2.2 µS.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  WORD i; // Counter variable used in for loop

  WORD NUMBER_OF_CYCLES = 0; // Declare variable to
// store the number of cycles detected

  STPIND(40, HIGH); // Light LED attached to pin 40 to
                    // show program is running.
  NUMBER_OF_CYCLES = COUNT(3, 65000); // Call function
// COUNT on pin 3 for 65 seconds and store value
// returned in variable NUMBER_OF_CYCLES

// for loop which will blink the LED attached to pin 42
// once for each cycle
  for(i = 0; i < NUMBER_OF_CYCLES; i = i + 1){
    STPIND(42, HIGH); // Turn on LED connected to pin
                      // 42
    PAUSE(2000);  // PAUSE command stops program for
                  // 2000 milliseconds (2 seconds)
    STPIND(42, LOW); // Turn off LED connected to pin
                     // 42
    PAUSE(2000); // PAUSE command stops program for
                 // 2000 milliseconds (2 seconds)
  }
  STPIND(40, LOW); //Turn off LED attached to pin 40
  END(); //END function ends program and stops program
         // from looping
}
```

COUNT2

WORD COUNT2(BYTE pin, WORD duration);

The **COUNT2** function counts the number of cycles (0-1-0 or 1-0-1) on the specified **pin** during the **duration** time frame and returns that number.

pin is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to input mode.

duration is a variable/constant/expression (1 – 65535) specifying the time during which to count. The unit of time for **duration** is 1 millisecond.

If an error occurs or no pulse is detected, the function returns **FALSE**.

DCD

WORD DCD(NIBBLE num);

The **DCD** function is a 2^n-power decoder of a four-bit value. **DCD** accepts a value from 0 to 15, and returns a 16-bit number with the bit described by the value **num** set to 1, and all others to 0.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  WORD RESULT;

  RESULT = DCD(1); // Call function DCD to decode '1'
// store value returned in variable RESULT
  if(RESULT == 0x2){ // Check if RESULT is equal to
                   // hexadecimal '2' or binary '10'
    STPIND(42, HIGH); // Light LED attached to pin 42
    PAUSE(1000); // Pause for 1 second
    STPIND(42, LOW); // Turn off LED attached to pin 42
    PAUSE(1000); // Pause for 1 second
  }
  else{
    STPIND(46, HIGH); // Light LED attached to pin 46
    PAUSE(1000); // Pause for 1 second
    STPIND(46, LOW); // Turn off LED attached to pin 46
    PAUSE(1000); // Pause for 1 second
  }
  END(); // Stops the program from further execution,
         // C Stamp enters low power mode
}
```

DEBUG

`void DEBUG(format, ...);`

The **`DEBUG`** command writes to Pin 24 the arguments that comprise the argument list as specified by the string **`format`**.

The communication settings of the **`DEBUG`** command are 57600 baud, 8 data bits, 1 stop bit, no parity, no flow control.

The string **`format`** consists of two types of items. The first type is made up of characters that will be printed on the screen using HyperTerm. The second type contains format commands that define the way the arguments are displayed. A format command begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format commands, and the format commands and the arguments are matched in order. For example, this **`DEBUG`** call

```
DEBUG(“Hi %c %d there!”, ‘c’, 10);
```

displays over the serial port

Hi c 10 there!

If there are insufficient arguments, with which to match the format commands, the output is undefined. If there are more arguments than format commands, the remaining arguments are discarded. The format commands supported in the **DEBUG** command are shown in the table below:

Code	*Format Meaning*
%b	Binary value
%c	Character
%d	Signed decimal integer
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)

The format commands may have modifiers that specify the field width, the number of decimal places, and a left justification flag. An integer placed between the **%** sign and the format command acts as a minimum field width specifier. This pads the output with blanks or zeros to ensure that it is at least a certain minimum length. If the string or number is greater than that minimum, it will be printed in full even if it overruns the minimum. The default padding is done with spaces. If you wish to pad with zeros, place a 0 before the field width specifier. For example, **%05d** will pad a number of less than five digits with zeros so that its total length is five.

Special characters that can be used when specifying text while using **DEBUG** are shown in the following table.

Back-Slash Character or Code	*Meaning or Control Function*

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 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)

DEBUG usage examples:

```
DEBUG(“Hi there!”);
DEBUG("The value of the variable i is %d", i);

DEBUG("%u%s%u", number1, string1, number2);
```

DEBUGIN

`void DEBUGIN(format, ...);`

The **`DEBUGIN`** command accepts input from Pin 25, and stores the information in the variables supplied in its argument list as specified by the string **`format`**.

The communication settings of the **`DEBUGIN`** command are 57,600 baud, 8 data bits, 1 stop bit, no parity, no flow control.

The input format specifiers are preceded by a % sign and tell **`DEBUGIN`** which type of data to be read next. The **`DEBUGIN`** codes are matched in order, with the variables receiving the input in the argument list. For example, **`%s`** reads a string while **`%u`** reads a **`WORD`**. These codes are listed in the following table.

Code	*Format Meaning*
`%s`	String of characters
`%u`	Unsigned decimal integers

The **`format`** string is read left to right, and the format codes are matched, in order, with the arguments that comprise the argument list.

A white space character causes **`DEBUGIN`** to skip over one or more white space characters. A non white space character causes the function to read and discard a matching character.

All the variables used to receive values must either be a mono-array. For example, if you want to read a **`WORD`** into a variable `count`, you would use the following **`DEBUGIN`** function call.

```
BYTE count[1];

DEBUGIN("%u", count);
```

Strings will be read into **`BYTE`** arrays, and the array names, without any indexes, are the arguments. For example,

```
BYTE string1[10];
```

```
DEBUGIN("%s", string1);
```

When reading strings, it is important to note that the input will stop when the first white space character is not specified.

The input data must be separated by spaces, tabs, or new line characters. Punctuation marks such as periods or commas are not valid unless specified in the **format** string. This means that

```
DEBUGIN("%u%u", r, c);
```

will accept an input of "10 20" but fail with "10,20." However, the function call

```
DEBUGIN("%u,%u", r, c);
```

will accept an input of "10,20" but fail with "10 20".

DMX512OUT

```
BIT DMX512OUT(NIBBLE break, NIBBLE mab, NIBBLE mtbf,
              BYTE sc, NIBBLE mtbp, BYTE buffer[],
              WORD N, BYTE TXpin);
```

The **DMX512OUT** function transmits asynchronous serial DMX512 packet data via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25). A frame in the packet is defined as 1 **LOW** start bit, a channel byte, and 2 **HIGH** stop bits.

break is a variable/constant/expression that specifies the number of **LOW** bits (0 - 15) to append to the default 22 **LOW** bits break sequence.

mab is a variable/constant/expression that specifies the number of **HIGH** bits (0 - 15) to append to the default 1 **HIGH** bit "Mark-After-Break" sequence.

mtbf is a variable/constant/expression that specifies the number of **HIGH** bits (0 - 15) to send after each single data byte channel frames.

sc is a variable/constant/expression that specifies the "Start Code".

mtbp is a variable/constant/expression that specifies the number of **HIGH** bits (0 - 15) to append at the end of the packet.

buffer is the name of the array of channel **BYTES** that the function will send. This can be an array of length equals to 1 byte. The maximum length of the buffer is 512 bytes.

N is a variable/constant/expression that specifies the number of bytes (1 - 512) in **buffer** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **N**-1).

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

TXpin is a variable/constant/expression that specifies the I/O pin to be used as a transmitter. This pin will be set to output mode.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
DMXOUT_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
DMXOUT_BUFEMP	1	Function returned after transmitting all data in the buffer.

The bit time for **DMX512OUT** is 4 μS, and its detailed behavior for sending a packet is shown in the following step sequence:

From any transmit pin (**TXpin**) state:

1. Send 11 **HIGH** bits
2. Send (22 + **break**) **LOW** bits
3. Send (1 + **mab**) **HIGH** bits
4. Send **sc**
    a. Send 1 **LOW start** bit
    b. Send **sc byte** Least Significant Bit (LSB) first

   c. Send 2 **HIGH** **stop** bits

   d. Send **mtbf** **HIGH** bits

5. For each **channel bytes**

   a. Send 1 **LOW** **start** bit

   b. Send **channel byte** LSB first

   c. Send 2 **HIGH** **stop** bits

   d. Send **mtbf** **HIGH** bits

6. Send **mtbp** **HIGH** bits

7. Set **TXpin** **LOW**

DMX512OUT Usage Example:

The following shows an example of the usage of the new **DMX512OUT** function. In this case, the channel data will be sent on pin 1 of the C Stamp, with the default of zero extra bits in the packet fields, except for an extra **HIGH** bit in the **mab** field.

```
//  somewhere at the beginning of the program
BYTE mysc;
#GROUP BIGDATA
RAM BYTE myBuffer[512];
#GROUP
:
//  later on manipulate mysc and myBuffer
:
//  send channel data
DMX512OUT(0, 1, 0, mysc, 0, MyBuffer, 512, 1);
:
```

DTMFOUT

```
BIT DTMFOUT(BYTE pin, WORD ontime, WORD offtime,
            BYTE tone);
```

The **DTMFOUT** function generates dual-tone, multi-frequency tones (DTMF, i.e. telephone “touch” tones) on one of the PWM outputs (Pin 3 or 27).

If the operation is successful, the function returns **TRUE**; otherwise it returns **FALSE**.

`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to output mode.

`ontime` is a variable/constant/expression (0 – 65535) specifying a duration of the tone. The unit of time is 1 millisecond.

`offtime` is a variable/constant/expression (0 – 65535) specifying the length of silent pause after a tone for when consecutive DTMF commands are issued. The unit of time is 1 millisecond.

`tone` is a variable/constant/expression (0 – 19) specifying the DTMF tone to generate. Tones 0 through 11 correspond to the standard layout of the telephone keypad, 12 through 15 are the fourth column tones used by phone test equipment and in ham-radio applications, and 16 through 18 correspond to events. This is summarized in the following table:

`tone` *value*	*Meaning*	*Low frequency*	*High frequency*
0	1	697 Hz	1209 Hz
1	2	697 Hz	1336 Hz
2	3	697 Hz	1477 Hz
3	4	770 Hz	1209 Hz
4	5	770 Hz	1336 Hz
5	6	770 Hz	1477 Hz
6	7	882 Hz	1209 Hz
7	8	882 Hz	1336 Hz
8	9	882 Hz	1477 Hz
9	*	941 Hz	1209 Hz
10	0	941 Hz	1336 Hz
11	#	941 Hz	1477 Hz
12	A	697 Hz	1633 Hz
13	B	770 Hz	1633 Hz

tone *value*	*Meaning*	*Low frequency*	*High frequency*
14	C	882 Hz	1633 Hz
15	D	941 Hz	1633 Hz
16	busy signal	480 Hz	620 Hz
17	dial tone	350 Hz	440 Hz
18	ring-back (US)	440 Hz	480 Hz

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  STPIND(46, HIGH); // Light LED attached to pin 46

  while(1){ // Continuous loop
    DTMFOUT(3, 10000, 100, 2);  // DTMFOUT function
// called to output DTMF signal for the number '3' on
// pin 3
// On duration of 10 seconds - off of 100 milliseconds.
  }
}
```

END

void END(void);

The **END** function ends the program, placing the C Stamp into low-power mode indefinitely. This is required to have a program that does not loop continuously. If no **END** statement is present, program execution restarts at the beginning of the **main()** function once all the statement in it have been executed.

END puts the C Stamp into its inactive, low-power mode. In this mode, the Stamp's current draw (excluding loads driven by the I/O pins) is minimized. **END** keeps the C Stamp inactive until the reset line is activated or the power is cycled off and back on.

Pins will retain their input or output settings after the C Stamp is deactivated by **END**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT STARTBUTTON; // Declare variable STARTBUTTON of
                   // type BIT

  STARTBUTTON = BUTTON(37, LOW, HIGH, 5); // Use
// function BUTTON to see if the utility button at pin
// 37 has been pressed and store value returned in
// STARTBUTTON

  while(STARTBUTTON){ // Loop that begins when
                      // STARTBUTTON is TRUE (1)
    STPIND(42, HIGH); // Turns on the LED attached to
                      // pin 42

/* The END function stops the program from executing
any further and places the CSTAMP in a low power mode.
Nothing placed after an END statement will be executed
since the CSTAMP stops the program once it encounters
the END function.
When the CSTAMP is in the low power mode the output to
the pins do not change so the LED attached to
pin 42 remains lit */

    END(); // END function is executed, program stops,
           // CSTAMP enters low power mode
    STPIND(46, HIGH); // This statement will never get
                      // executed
    STPIND(42, LOW); // This statement will never get
                     // executed
  }
}
```

FLOOKDOWN

int FLOOKDOWN(float target, NIBBLE comparison_op,
float array[], int array_size);

The **FLOOKDOWN** function compares the value of the **float target** to a list of values in an **array** of **float**, and returns the index number of the first value that matches the **comparison_op** condition. If no value in the list matches or an error is encountered, the function returns -1.

`target` is a variable/constant/expression (floating point) to be compared to the values in the list.

`comparison_op` is a comparison operator, as described in the table below, to be used as the criteria when comparing values.

`comparison_op` *code*	*value*	***Meaning***
EQ	1	= Find the first value equal to **`target`**.
NE	2	!= Find the first value not equal to **`target`**.
GT	3	> Find the first value greater than **`target`**.
LT	4	< Find the first value less than **`target`**.
GE	5	>= Find the first value greater than or equal to **`target`**.
LE	6	<= Find the first value lees than or equal to **`target`**.

`array` is an array of floating point values to be compared to **`target`**.

`array_size` is the number of elements in **`array`** (1 to 32767).

The function returns the index value (0 for the first element to 32766 for the 32767th element).

EXAMPLE PROGRAM:

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

```
void main(void)
{
  int index_result;  // Variable index_result of type
// int to store the result from FLOOKDOWN function
  float test_array[6]; // Array test_array of type
// float to store values and compare against the
// target
  float target; // Variable target of type float to
                // store the target value

// Explicitly defining the array
  test_array[0] = 15.1;   // Defining index 0
  test_array[1] = 9.24;   // Defining index 1
  test_array[2] = 0.0;    // Defining index 2
  test_array[3] = 3.9;    // Defining index 3
  test_array[4] = 21.451; // Defining index 4
  test_array[5] = 3.1;    // Defining index 5

  target = 3.1; // Set target equal to 3.1

/*Example: Comparison_op = 1, find the index of the
first value equal to target (index 5)*/

  index_result = FLOOKDOWN(target, 1, test_array, 6);
// Call the function FLOOKDOWN to find the array index
// and store the result in the variable
// index_result

  END(); // Stop program execution, C Stamp enters low
         // power mode
}
```

FLOOKUP

float FLOOKUP(int index, float array[],
int array_size);

The **FLOOKUP** function returns the value at location **index** in an **array** of **float**. If **index** is not in the proper range (0 to **array_size** - 1), zero is returned.

index is a variable/constant/expression (0 for the first element to 32766 for the 32767th element) indicating the list item to retrieve.

array is an array of floating point numbers.

array_size is the number of elements in **array** (1 to 32767).

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float lookup; // Variable lookup of type float to
               // store the value returned by FLOOKUP
  float array[6]; // Array of type float

//Defining the array one element at a time
  array[0] = 3.1;
  array[1] = 9.24;
  array[2] = 0.0;
  array[3] = 3.9;
  array[4] = 21.451;
  array[5] = 15.1;

  lookup = FLOOKUP(3, array, 6); // Call the function
// FLOOKUP to find the value of the element in array at
// index 3 (4th element) and store it in lookup
  if(lookup == 3.9) // Check to see if the value
                    // returned was 3.9
    STPIND(42, HIGH); // Light LED attached to pin 42
  else
    STPIND(46, HIGH); // Light LED attached to pin 46
}
```

FREQOUT

```
BIT FREQOUT(BYTE pin, WORD duration, float freq1,
            float freq2);
```

The **FREQOUT** function generates one or two sine-wave tones for a specified **duration** on one of the PWM outputs.

pin is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to output mode.

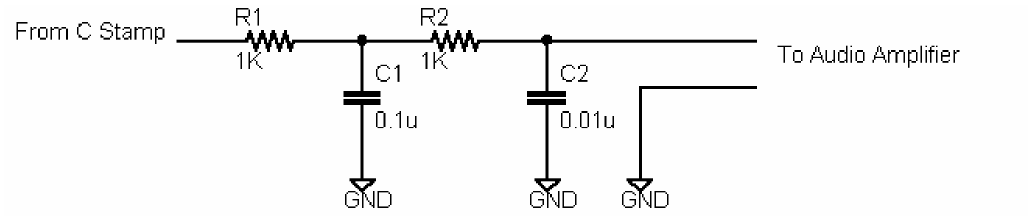
duration is a variable/constant/expression (0 - 65535) specifying the amount of time to generate the tone(s). The unit of time for **duration** is 1 millisecond.

`freq1` is a variable/constant/expression (0^+ – 2000) specifying the frequency of the first tone. The unit of **`freq1`** is 1 Hz. If **`freq1`** is zero, the function will PAUSE for the **`duration`**.

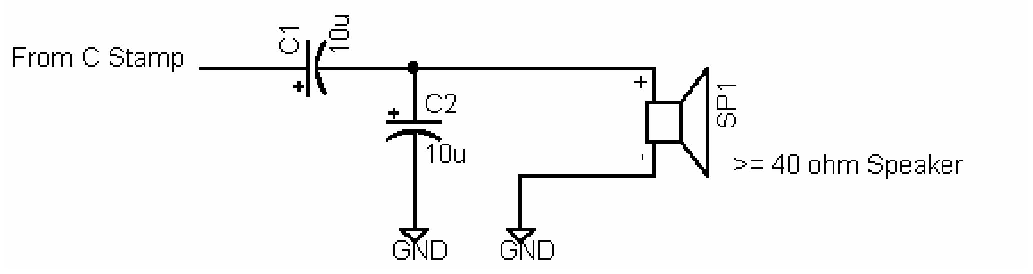
`freq2` is an argument exactly like **`freq1`**. When **`freq1`** and **`freq2`** are non-zero, two frequencies will be mixed together on the specified I/O pin.

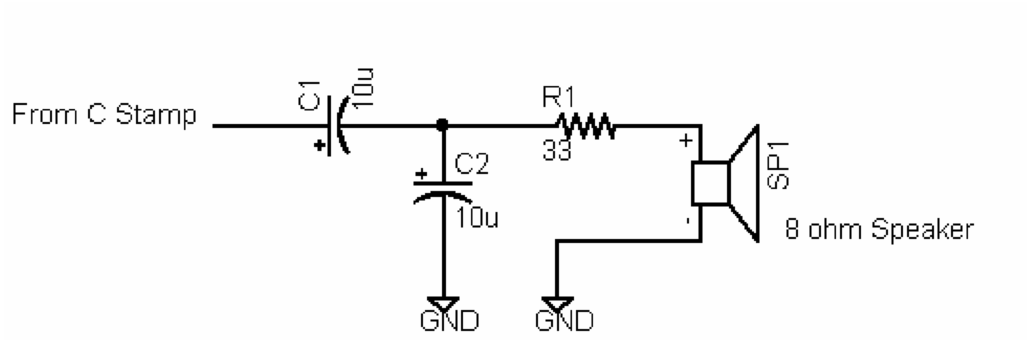
If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

To drive an audio amplifier, a filter like the one shown in the figure below should be used.



To drive a speaker directly, a filter like the one shown in the figure below should be used depending on the speaker being driven. A piezo speaker would fall in the category of a ≥ 40 Ω speaker. Even though the speakers below are shown with polarity, many speakers do not have polarity (+ or -), so those speakers can be connected either way.





EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  while(1){ // Continuous loop
    FREQOUT(3, 65000, 1000, 0); // Call function
// FREQOUT to generate a 1000 Hz signal via PWM on
// pin 3
  }
}
```

FREQOUT2

`BIT FREQOUT2(BYTE pin, WORD duration, float freq);`

The **`FREQOUT2`** function generates a square-wave signal for a specified **`duration`** on any Digital I/O pin.

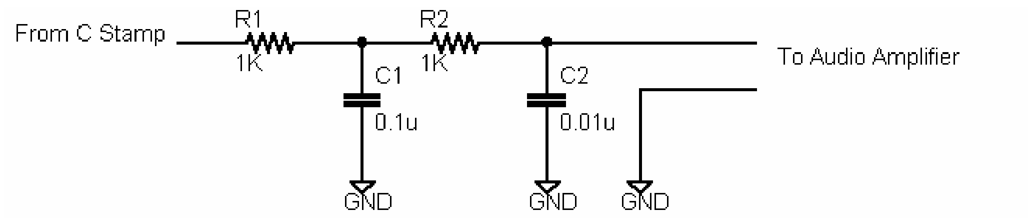
`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to output mode.

`duration` is a variable/constant/expression (0 - 65535) specifying the amount of time to generate the signal. The unit of time for **`duration`** is 1 millisecond.

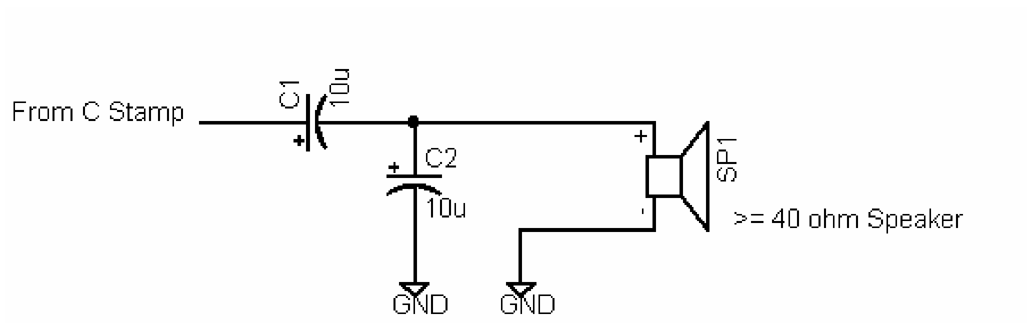
`freq` is a variable/constant/expression (4 - 227272) specifying the frequency of the signal. The unit of **`freq`** is 1 Hz. If **`freq`** is zero, the function will PAUSE for the **`duration`**.

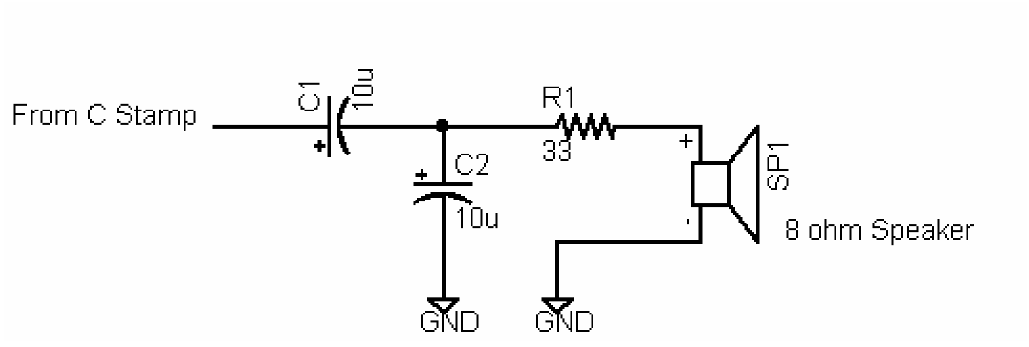
If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

To drive an audio amplifier, a filter like the one shown in the figure below should be used.



To drive a speaker directly, a filter like the one shown in the figure below should be used depending on the speaker being driven. A piezo speaker would fall in the category of a ≥ 40 Ω speaker. Even though the speakers below are shown with polarity, many speakers do not have polarity (+ or -), so those speakers can be connected either way.





GTPIND

`BIT GTPIND(BYTE pin);`

The **`GTPIND`** function sets **`pin`** to digital input mode and returns the value of **`pin`**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT INPUT; // Variable INPUT of type BIT to store the
// value returned by function GTPIND
  BIT BUTTON_CHECK; // Variable BUTTON_CHECK of type
// BIT to store the value returned by BUTTON function

  while(1){
    BUTTON_CHECK = BUTTON(37, LOW, HIGH, 5); // Check
// if utility button at pin 37 has been pressed and
// store value in variable BUTTON_CHECK
    if(BUTTON_CHECK){ // Determine if BUTTON_CHECK is
                      // true
      INPUT = GTPIND(42); // Call function GTPIND to
// find the digital value of pin 42
// store value returned in variable INPUT
      if(INPUT){ // Check if input is 'HIGH'
        STPIND(41, HIGH); // Light LED attached to pin
                          // 41
        END(); // End program, C Stamp enters low power
               // mode.
      }
    }
  }
}
```

HYP

`float HYP(float x, float y);`

The **HYP** function returns the hypotenuse of the (x, y) vector.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float hypotenuse; // Variable hypotenuse of type
// float to store value returned by HYP function
  float x; // Variable x of type float to store one
```

```
          // input for the HYP function
  float y; // Variable y of type float to store one
          // input for the HYP function
  float ans; // Variable ans of type float to store the
           // pre-calculated answer

//Example
  x = 5.1; // Set variable x to 5.1
  y = 6.8; // Set variable y to 6.8
  ans = 8.5; // Set variable ans to 8.5
  hypotenuse = HYP(x, y); // HYP functino returns the
// hypotenuse of <x, y> (or <5.1, 6.8>) and stores it
// in the variable hypotenuse
  if(hypotenuse == ans){ // Check if hypotenuse is
                         // equal to ans
    STPIND(41, HIGH); // If so, Turn on LED connected
                      // to pin 41
  }

  END();
}
```

I2CIN

```
BIT I2CIN(BYTE slaveID, NIBBLE addressmode,
          BYTE address, BYTE lowaddress,
          BYTE inputdata[], BYTE n);
```

The **I2CIN** function receives data in master mode from a device using the I^2C protocol on the C Stamp I^2C pins (Pin 28 for the I^2C Clock and Pin 29 for the I^2C Data). The I^2C protocol is a form of synchronous serial communication developed by Phillips Semiconductors. It only requires two I/O pins and both pins can be shared between multiple I^2C devices. Both I/O pins will be toggled between input and output modes during the **I2CIN** command and both will be set to input mode by the end of the **I2CIN** command. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**. This could mean that there was an error in the arguments of the function or some other problem.

slaveID is a variable/constant/expression (0 – 255) indicating the unique ID of the I^2C device sending data.

addressmode variable/constant/expression (0 – 2) that indicates the number of bytes used for the address within the I^2C device sending data. If **addressmode** is 0, this indicates that the I^2C device sending data does not require an address, and both **address** and **lowaddress** are ignored. If **addressmode** is 1, **address** is the byte address in the sender and **lowaddress** is ignored. If **addressmode** is 2, **address** is the high byte of the word address at the sender and **lowaddress** is the low byte. When **address** and/or **lowaddress** are ignored, they can be any value. However, a value for each of these arguments always has to be provided.

address is a variable/constant/expression (0 – 255) indicating the desired address within the I^{2}C device from which data is received. The **address** argument may be used with the **lowaddress** argument to indicate a word-sized address value when **addressmode** is 2.

lowaddress is a variable/constant/expression (0 – 255) indicating the low-byte of the word-sized address within the I^{2}C device from which data is received when **addressmode** is 2.

inputdata is a **BYTE** array to put the received data. The array fills up from the low address 0 to the high address **n**-1.

n is the number of bytes (1 - 255) to be placed in **inputdata**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **n**-1).

The number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

I2COUT

```
BIT I2COUT(BYTE slaveID, NIBBLE addressmode,
           BYTE address, BYTE lowaddress,
           BYTE outputdata[], BYTE n);
```

The **I2COUT** function sends data in master mode to a device using the I^{2}C protocol on the C Stamp I^{2}C pins (Pin 28 for the I^{2}C Clock and Pin 29 for the I^{2}C Data). The I^{2}C protocol is a form of synchronous serial communication developed by Phillips Semiconductors. It only requires two I/O pins and both pins can be shared between multiple I^{2}C devices. Both I/O pins will be toggled between input and output modes during the **I2COUT** command and both will be set to input mode by the end of the **I2COUT** command. If the function is successful, it returns **TRUE**; otherwise, it returns

FALSE. This could mean that there was an error in the arguments of the function or some other problem.

slaveID is a variable/constant/expression (0 – 255) indicating the unique ID of the I^2C device receiving data.

addressmode variable/constant/expression (0 – 2) that indicates the number of bytes used for the address within the I^2C device receiving data. If **addressmode** is 0, this indicates that the I^2C device receiving data does not require an address, and both **address** and **lowaddress** are ignored. If **addressmode** is 1, **address** is the byte address in the receiver and **lowaddress** is ignored. If **addressmode** is 2, **address** is the high byte of the word address at the receiver and **lowaddress** is the low byte. When **address** and/or **lowaddress** are ignored, they can be any value. However, a value for each of these arguments always has to be provided.

address is a variable/constant/expression (0 – 255) indicating the desired address within the I^2C device to which data is sent. The **address** argument may be used with the **lowaddress** argument to indicate a word-sized address value when **addressmode** is 2.

lowaddress is a variable/constant/expression (0 – 255) indicating the low-byte of the word-sized address within the I^2C device to which data is sent when **addressmode** is 2.

outputdata is a **BYTE** array with the data to be sent. The array gets sent from the low address 0 to the high address **n**-1.

n is the number of bytes (1 – 255) in **outputdata** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **n**-1).

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

INTSET

```
void INTSET(BYTE interrupts, BYTE edge,
           BYTE priorities, NIBBLE calledfrom);
```

The **INTSET** function enables or disables interrupts, and sets the priority for each one of them.

`interrupts` is a variable/constant/expression (0 – 255) indicating whether each interrupt is going to be enabled or disabled. Each bit of this argument corresponds to an interrupt as shown in the following table. If the bit is **ONE**, the interrupt is enabled; if it is **ZERO**, the interrupt is disabled.

edge is a variable/constant/expression (0 – 255) indicating the edge of a particular INTx interrupt. Each bit (7 – 4) of this argument corresponds to the edge of an interrupt as shown in the following table. If the bit is **ONE**, the interrupt edge is a rising edge; if it is **ZERO**, the interrupt edge is a falling edge.

`priorities` is a variable/constant/expression (0 – 255) indicating the priority level of a particular interrupt. Each bit of this argument corresponds to the priority level of an interrupt as shown in the following table. When enabled, if the bit is **ONE**, the interrupt priority level is HIGH; if it is **ZERO**, the priority level is LOW.

`interrupts`*, *`edge`*, and* `priorities` ***BYTE BITS mapping to Interrupts***							
BIT 7	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**
INT0	INT1	INT2	INT3	KBI0	KBI1	KBI2	KBI3

When a particular interrupt BIT is disabled, its corresponding **edge** and **`priorities`** BITs are meaningless.

BITs 3 - 0 of the **edge** BYTE can be anything, as the KBIx interrupts are sensitive to change, not edges.

BIT 7 of the **`priorities`** BYTE can be anything, as the priority of INT0 cannot be set. It is always a high priority interrupt.

If any of the KBIx interrupts have a high priority, then all of the KBIx interrupts will be treated as high priority.

For example, take the command

```
INTSET(0x40, 0x00, 0x40, 0x2);
```

You must convert the numbers into binary and look at the bit positions according to the table above: 0x40 = 01000000 so this means INT1 is enabled; 0x00 = 00000000 means the interrupt is set to falling edge; and 0x40 again is 01000000, meaning INT1 is high priority.

`calledfrom` is a variable/constant/expression (0 – 2) that signals the system from where is **`INTSET`** being called. The values of **`calledfrom`** are interpreted according to the table below.

Symbol	*Value*	*Meaning*
`INTLOW`	0	**`INTSET`** is being called from the low priority interrupt vector handler **`InterruptHandlerLow`**
`INTHIGH`	1	**`INTSET`** is being called from the high priority interrupt vector handler **`InterruptHandlerHigh`**
`INTOTHER`	2	**`INTSET`** is being called from somewhere else other than the two cases above

INTSOURCE

`BYTE INTSOURCE(BIT calledfrom);`

The **`INTSOURCE`** function returns a **`BYTE`** that contains the sources of pending interrupts. When this information has been identified to be returned to the user, all interrupts will be disabled. Each bit of the returned **`BYTE`** corresponds to an interrupt as shown in the following table. If the bit is **`ONE`**, the interrupt is pending; if it is **`ZERO`**, the interrupt is not pending.

Pending interrupts returned BYTE BITS mapping to Interrupts							
BIT 7	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**
INT0	INT1	INT2	INT3	KBI0	KBI1	KBI2	KBI3

If any of the KBIx interrupts are pending, then all of them will be identified as pending.

`calledfrom` is a variable/constant/expression (0 or 1) indicating the priority level of the interrupt handler from which **`INTSOURCE`** is being called. If priority is **`ONE`**, this signals the system that **`INTSOURCE`** is being called from **`InterruptHandlerHigh`**; if priority is **`ZERO`**, the signal to the system is that

`INTSOURCE` is being called from **`InterruptHandlerLow`**. **`INTHIGH`** and **`INTLOW`** can also be used respectively.

LOOKDOWN

```
int LOOKDOWN(int target, NIBBLE comparison_op,
             int array[], int array_size);
```

The **`LOOKDOWN`** function compares the value of the **`int target`** to a list of values in an **`array`** of **`int`**, and returns the index number of the first value that matches the **`comparison_op`** condition. If no value in the list matches or an error is encountered, the function returns -1.

`target` is a variable/constant/expression (-32768 to 32767) to be compared to the values in the list.

`comparison_op` is a comparison operator, as described in the table below, to be used as the criteria when comparing values.

`comparison_op` *code*	*value*	*Meaning*
EQ	1	= Find the first value equal to **`target`**.
NE	2	!= Find the first value not equal to **`target`**.
GT	3	> Find the first value greater than **`target`**.
LT	4	< Find the first value less than **`target`**.
GE	5	>= Find the first value greater than or equal to **`target`**.
LE	6	<= Find the first value lees than or equal to

comparison_op *code*	*value*	*Meaning*
		target.

array is an array of integers with values (-32768 to 32767) to be compared to **target**.

array_size is the number of elements in **array** (1 to 32767).

The function returns the index value (0 for the first element to 32766 for the 32767th element).

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  int index; // Variable index of type int, used to
// store the value returned by the LOOKDOWN function
  NIBBLE i; // Variable i of type NIBBLE, used as
// counter in the for loop which generates test_array
  int test_array[6]; // Array test_array of type int,
// used to store the values to be processed by the
// LOOKDOWN function

  for(i = 0; i < 6; i = i + 1){ // for loop to generate
                                // values in test_array
    test_array[i] = i * 2;  // Each index of test_array
// has a value of twice the index, i.e. index 4 has a
// value of 8.
  }

  index = LOOKDOWN(8, 1, test_array, 6); // Call
// function LOOKDOWN to find which index has a value
// equal to 8 (index 4)
// store the value returned in variable index

  END(); // Stop program from looping, C Stamp enters
         // low power mode
}
```

LOOKUP

`int LOOKUP(int index, int array[], int array_size);`

The **`LOOKUP`** function returns the value at location **`index`** in an **`array`** of **`int`**. If **`index`** is not in the proper range (0 to **`array_size`** - 1), zero is returned.

`index` is a variable/constant/expression (0 for the first element to 32766 for the 32767th element) indicating the list item to retrieve.

`array` is an array of integers with values (-32768 to 32767).

`array_size` is the number of elements in **`array`** (1 to 32767).

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  int VALUE; // Variable VALUE of type int to store the
             // value returned by the function LOOKUP
  NIBBLE i; // Variable i of type NIBBLE, used as
            // counter in for loop
  int test_array[10]; // Array test_array to store even
                      // numbers 0-20

  for(i = 0; i < 10; i = i + 1){ // for loop to
                                 // generate array
    test_array[i] = i * 2; // Sets each element as
                           // twice it's index value
  }

  VALUE = LOOKUP(4, test_array, 10); // Call function
// LOOKUP to determine value of test_array at element 4
// the value returned is then stored in VALUE

  if(VALUE == 8){ // Check to see if VALUE is 8
    STPIND(42, HIGH); // Turn LED attached to pin 42 on
  }
  else{
    STPIND(46, HIGH); // Turn LED attached to pin 46 on
  }
}
```

NAP

BIT NAP(NIBBLE value);

The **NAP** function enters sleep mode for a short period of time. Power consumption is reduced.

value is a variable/constant/expression (0 – 7) that specifies the duration of the reduced-power nap. The duration is (2^{value}) x 18 mS.

If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  while(1){ // Continuous loop
    STPIND(42, HIGH); // Light LED attached to pin 42
    NAP(5); // Call NAP function for a duration of
            // 576 mS
    STPIND(42, LOW); // Turn off LED attached to pin 42
    NAP(5); // Call NAP function for a duration of
            // 576 mS
  }
}
```

NCD

BYTE NCD(WORD num);

The Encoder function **NCD** is a "priority" encoder of a 16-bit value **num**. **NCD** takes a 16-bit value, finds the highest bit containing a 1 and returns the bit position plus one (1 through 16). If the input value is 0, **NCD** returns 0.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BYTE ENCODED; // Declare variable ENCODED of type
// BYTE to store value returned by function NCD
```

```
  ENCODED = NCD(0x1000); // Call function NCD to encode
// hexadecimal 1000 or binary 1 0000 0000 0000 store
// value returned in variable ENCODED

  if(ENCODED == 13){ // Check if ENCODED is 13
    STPIND(42, HIGH); // Light LED attached to pin 42
  }
  else{
    STPIND(46, HIGH);  // Light LED attached to pin 46
  }

/*
The inputs for DCD range from 0 to 15. The values
returned by DCD range from 1 to 8000 in hexadecimal.
The inputs for the function NCD range from 0 to 8000.
The outputs, however, range from 1 to 16 instead of 0
to 15. This means that if you were to call DCD(5) which
should return 10 hexadecimal and then NCD(0x10) you
would not get the value 5 returned. 10 hexadecimal is
10000 in binary, NCD finds the position of the bit
which is set to 1, in this case the 5th bit and then
adds one and so returns 6. To directly compare the two
you must take the value of one less than the value
returned by NCD.
*/

  ENCODED = NCD(DCD(5)); // Call function DCD to decode
// 5, the value returned is then encoded by the
// function NCD, the result is stored in ENCODED

  if((ENCODED - 1) == 5){ // Check if ENCODED - 1 is 5
    STPIND(41, HIGH); // Light LED attached to pin 41
  }
  else{
    STPIND(45, HIGH); // Light LED attached to pin 45
  }
}
```

PARINA

NIBBLE PARINA(WORD timeout, BYTE buffer[], BYTE N);

The **PARINA** function receives asynchronous parallel data via the built-in asynchronous 8-bit parallel slave port in the C Stamp. The connections of this port, from the C Stamp's perspective are shown in the following table.

Pin Name	*Pin Number*	*Pin Function*	*Pin Direction*
CSn	47	Chip select control	input
WRn	1	Write Control	input
RDn	2	Read control	input
PSP7 – PSP0	39 - 46	Data Bus	input

This built-in parallel port functions in the following manner in receive mode:

1. The external entity sending data to the C Stamp (sender) puts the data on the Data Bus.
2. The sender pulls the CSn line LOW.
3. The sender pulls the WRn line LOW for at least 50 nS.
4. The sender pulls the WRn line HIGH.
5. The sender waits for at least 25 nS.
6. The sender removes the data from the Data Bus if necessary. This is optional.
7. The sender waits for at least 25 nS.
8. The sender pulls the CSn line HIGH.

`timeout` is a variable/constant/expression (0 – 65535) that tells **`PARINA`** how long to wait in mS for incoming data. If data does not arrive in time or if the buffer gets full before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait until it receives enough data to fill the buffer before it returns. The usage of timeouts applies to each data unit being received.

`buffer` is the name of the array of **`BYTES`** that the function will use as storage to return the received data to the calling function. This can be an array of length equal to 1 byte. The maximum length of the buffer is 255 bytes.

`N` is the number of bytes (1 - 255) to be placed in **`buffer`**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **`N`**-1).

If you do not use the timeout feature, the commands wait forever to get the number of bytes that you requested. However, the number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
PARINA_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
PARINA_BUFULL	1	Function returned with the buffer full of data.
PARINA_TIMOUT	2	The function exited after a timeout condition occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the timeout occurred. If this location contains the value 0xFF, this indicates that no data was received.
PARINA_OVRERR	4	An overrun error occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, this indicates that no data was received.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
```

```
{
  NIBBLE RETURNED; // Declare variable RETURNED of type
                   // NIBBLE to store value returned by
                   // PARINA function
  BYTE DATA[1];    // Declare array DATA of type BYTE
                   // with 1 element to store data that
                   // will be received by PARINA
                   // function

// Light LED attached to pin 20 to show program is
// running
  STPIND(20, HIGH);

// Call function PARINA, store value returned in
// RETURNED and the data read in the array DATA
  RETURNED = PARINA(0, DATA, 1);

// Turn off LED attach to pin 20 to show data has been
// read
  STPIND(20, LOW);

// Check if PARINA function returned with a full buffer
// of data
  if(RETURNED == 1){
    STPIND(23, HIGH); // Light LED attached to pin 23
    PAUSE(1000);      // Pause for one second
    STPIND(23, LOW); // Turn off LED attached to pin 23
    PAUSE(1000);     // Pause for one second
  }

// Check if data read was hexadecimal AA or binary
// 10101010
  if(DATA[0] == 0xAA){
    STPIND(23, HIGH); // Light LED attached to pin 23
    PAUSE(1000);      // Pause for one second
    STPIND(23, LOW); // Turn off LED attached to pin 23
    PAUSE(1000);     // Pause for one second
  }
  else{
    STPIND(20, HIGH); // Light LED attached to pin 20
    PAUSE(1000);      // Pause for one second
    STPIND(20, LOW); // Turn off LED attached to pin 20
  }

// End program and stop C Stamp from looping
```

```
  END();
}
```

PAROUTA

`NIBBLE PAROUTA(WORD timeout, BYTE buffer[], BYTE N);`

The **`PAROUTA`** function sends asynchronous parallel data via the built-in asynchronous 8-bit parallel slave port in the C Stamp. The connections of this port, from the C Stamp's perspective, are shown in the following table.

Pin Name	*Pin Number*	*Pin Function*	*Pin Direction*
CSn	47	Chip select control	input
WRn	1	Write Control	input
RDn	2	Read control	input
PSP7 – PSP0	39 - 46	Data Bus	output

This built-in parallel port functions in the following manner in send mode:

1. The external entity receiving data from the C Stamp (receiver) pulls the CSn line LOW.
2. The receiver pulls the RDn LOW for at least 50 nS.
3. The C Stamp puts the data on the Data Bus.
4. The receiver reads the data.
5. The receiver pulls the RDn line HIGH.
6. The receiver waits for at least 125 nS.
7. The receiver pulls the CSn line HIGH.

`timeout` is a variable/constant/expression (0 – 65535) that tells **`PAROUTA`** how long to wait in mS for the receiver to read more data. If data is not read before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait indefinitely until the data is read. The usage of timeouts applies to each data unit being sent.

buffer is the name of the array of **BYTES** that the function will send. This can be an array of length equal to 1 byte. The maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) in **buffer** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **N**-1).

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
PAROUTA_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
PAROUTA_BUFEMP	1	Function returned after transmitting all data in the buffer.
PAROUTA_TIMOUT	2	The function exited after a timeout condition occurred. Some or all of the data was not sent.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE RETURNED; // Declare variable RETURNED of type
                   // NIBBLE to store value returned by
                   // PAROUTA function
  BYTE DATA[1];    // Declare array DATA of type BYTE
                   // with one element, used to store
                   // the data which will be placed on
                   // pins 39-46 by PAROUTA function

// Set first element of DATA to hexadecimal AA or
// binary 10101010
  DATA[0] = 0xAA;

// Light LED attached to pin 23 to show program is
// running
  STPIND(23, HIGH);

// Call function PAROUTA with no timeout to display the
// data in array DATA
  RETURNED = PAROUTA(0, DATA, 1);

// Turn off LED attached to pin 23
  STPIND(23, LOW);
  PAUSE(1000); // Pause for 1 second

// Check if PAROUTA function returned code for all data
// in buffer being transmitted
  if(RETURNED == 1){
    STPIND(23, HIGH); // Light LED attached to pin 23
    PAUSE(1000);      // Pause for 1 second
    STPIND(23, LOW);  // Turn off LED attached to
}                     // pin 23

// Stop program from further execution, C Stamp enters
// low power mode
  END();
}
```

PAUSE

`void PAUSE(WORD duration);`

The **`PAUSE`** function pauses the program (does nothing) for the specified **`duration`**. **`duration`** is a variable/constant/expression (0 – 65535) that specifies the duration of the pause. The unit of time for **`duration`** is 1 millisecond.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT ON; // Declare variable ON of size BIT to store
          // value of BUTTON function
  ON = BUTTON(37, LOW, LOW, 50); // Use BUTTON function
// to determine if the utility button connected to pin
// 37 has been pushed.

  if(ON){ // Check to see if the utility button was
          // pushed
    while(1){ // Continuous loop
      STPIND(43, HIGH); // Turn LED connected to pin 43
                        // on
      PAUSE(3000); // Wait 3000 milliseconds
                   // (3 seconds)
      STPIND(43, LOW); // Turn LED connected to pin 43
                       // off
      PAUSE(500); // Wait 500 milliseconds
                  // (0.5 seconds or half of a second)
      STPIND(43, HIGH); // Turn LED connected to pin 43
                        // on
      PAUSE(5000); // Wait 5000 milliseconds
                   // (5 seconds)
      STPIND(43, LOW); // Turn LED connected to pin 43
                       // off
      PAUSE(1000); // Wait 1000 milliseconds (1 second)
    }
  }
}
```

PAUSEUS

void PAUSEUS(WORD duration);

The **PAUSEUS** function pauses the program (does nothing) for the specified **duration**. **duration** is a variable/constant/expression (0 – 65535) that specifies the duration of the pause. The unit of time for **duration** is 2.2 microseconds.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT ON; // Declare variable ON of size BIT to store
          // value of BUTTON function

  ON = BUTTON(37, LOW, LOW, 50); // Use BUTTON function
// to determine if the utility button connected to pin
// 37 has been pushed.

  if(ON){ // Check to see if the utility button was
          // pushed
    STPIND(39, HIGH); // Set pin 39 HIGH
/* The following loop's speed is too quick to be
visible via LED. */
    while(1){ // Continuous loop
      PAUSEUS(100); // Pause for 220 microseconds
      TOGGLE(39); // Toggle pin 39
    }
  }
}
```

PULSIN

`WORD PULSIN(BYTE pin, BIT state, WORD delay);`

The **`PULSIN`** function measures the width of a pulse on a Capture **`pin`** described by **`state`** and returns the result.

`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to input mode.

`state` is a variable/constant/expression (0 – 1) that specifies whether the pulse to be measured is low (0) or high (1). A low pulse begins with a 1-to-0 transition and a high pulse begins with a 0-to-1 transition.

If an error occurs or no pulse is detected, the function returns **`FALSE`**; otherwise the width of the pulse is returned in units of is 0.8 µS.

`delay` is a variable/constant/expression (1 – 65535) that specifies for how long the pin will be monitored. The unit of time for **`delay`** is 1 millisecond.

NOTE:

This function works ONLY on "Capture" pins. To measure a pulse on non-Capture pins, see **`PULSIN2()`** below.

The Capture pins are names for a couple of pins; Capture1 and Capture3. They are defined in the pin table and referenced in the **`COUNT`** and **`PULSIN`** functions.

The functions that use Capture pins are using dedicated hardware in the C Stamp to "Capture" edges and count pulses or time. Therefore, they have 28.375 times better resolution than their counterpart functions terminated in a "2". The functions that use Capture pins have a resolution of 0.8 µS and the others a resolution of 22.7 µS.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT ON; // Declare variable ON of type BIT, used to
          // store value returned by BUTTON function
  WORD WIDTH; // Declare variable WIDTH of type WORD,
// used to store value returned by PULSIN function

  while(1){ // Continuous loop
    ON = BUTTON(37, LOW, HIGH, 5); // Check if utility
// button at pin 37 has been pressed and store value in
// variable ON
    if(ON){ // If ON is TRUE (1)
      WIDTH = PULSIN(3, 1, 65000); // Call function
// PULSIN to determine the width of the pulse on pin 3
// the value returned is then stored in the variable
// WIDTH
      if((WIDTH<=6500) && (WIDTH >=5500)){ // Check if
// WIDTH is in the correct range
        STPIND(46, LOW); // Turn off LED attached to
                         // pin 46
        STPIND(42, HIGH); // Light LED attached to
                          // pin 42
        END(); // Stop program from looping, C Stamp
               // enters low power mode
      }
      else
        STPIND(46, HIGH); // Light LED attached to
                          // pin 46
    }
  }
}
```

PULSIN2

`WORD PULSIN2(BYTE pin, BIT state, WORD delay);`

The **`PULSIN2`** function measures the width of a pulse on a **`pin`** described by **`state`** and returns the result.

`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to input mode.

`state` is a variable/constant/expression (0 – 1) that specifies whether the pulse to be measured is low (0) or high (1). A low pulse begins with a 1-to-0 transition and a high pulse begins with a 0-to-1 transition.

If an error occurs or no pulse is detected, the function returns **FALSE**; otherwise, the width of the pulse is returned in units of is 22.7 µS.

`delay` is a variable/constant/expression (1 – 65535) that specifies for how long the pin will be monitored. The unit of time for **`delay`** is 1 millisecond.

pin will be monitored. The unit of time for **`delay`** is 1 millisecond.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT ON;
  WORD WIDTH;

  PWM(3,512,250);  // This will create a pulse width of
                   // roughly 206 microseconds
  PAUSE(1000);

  while(1){
    ON = BUTTON(37, LOW, HIGH, 5);
    if (ON){  //Check for button pushed
      WIDTH = PULSIN2(17, 1, 65000);
// PULSIN2 to determine the width of the pulse on a
// desired pin
// the value returned is then stored in the variable
// WIDTH
      if((WIDTH <= 10) && (WIDTH >= 8)){
// Check if WIDTH is in the correct range
        STPIND(42, HIGH);  // Turn off LED attached to
                           // pin 46
        STPIND(46, LOW);   // Light LED attached to
                           // pin 42
        END();  // Stop program from looping, C Stamp
              // enters low power mode
      }
      else STPIND(46, HIGH);  // Light LED attached to
                              // pin 46
    }
  }
  END();  //C-Stamp enters low power mode
}
```

PULSOUT

`BIT PULSOUT(BYTE pin, WORD duration, BIT mode,`
`            BIT polarity);`

The **`PULSOUT`** function generates a pulse on **`pin`** with a width of **`duration`** and a given **`polarity`**.

`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to output mode.

`duration` is a variable/constant/expression (0 – 65535) that specifies the duration of the pulse. The unit of time for **`duration`** is 7.5 μS if **`mode`** is a variable/constant/expression that evaluates to 0. If **`mode`** evaluates to 1, then the unit of time for **`duration`** is 2.5 μS.

`polarity` is a variable/constant/expression (0 – 1) that specifies the polarity of the generated pulse. If **`polarity`** is 0, then a negative pulse is generated (1 – 0 – 1). If **`polarity`** is 1, then a positive pulse is generated (0 – 1 – 0).

The function returns **TRUE** if successful; **FALSE** otherwise.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  while(1){ // Continuous loop
    PULSOUT(3, 100, 0, 1); // Call function PULSOUT to
// generate a positive pulse for 750 microseconds on
// pin 3
    PAUSEUS(10); // Pause for 22 microseconds
  }
}
```

PWM

`BIT PWM(BYTE pin, WORD duty, BYTE period);`

The **`PWM`** function converts a digital value to analog output via pulse-width modulation on one the PWM outputs (Pin 3 and 27).

`pin` is a variable/constant/expression that specifies the I/O pin to use. This pin will be set to output mode.

`duty` is a variable/constant/expression (0 - 1023) that specifies the analog output level (0+ to **`HIGH`** level).

If **`duty`** is 0, then the PWM output is disabled, if it had been previously enabled.

There is a practical constraint on the **`duty`** parameter. The maximum value of this parameter is $2^{PRM} - 1$, where

$$PRM = \left\lfloor \frac{\log\left(\frac{40\ MHz}{F_{PWM}}\right)}{\log(2)} \right\rfloor$$

If $PRM > 10$, then $PRM = 10$.

F_{PWM} is the frequency of the PWM signal.

Consider the extreme case of the command:

```
PWM(X, 1, 1);  // X = 3 or 27
```

In this example, $F_{PWM} = 312.5\ KHz$, so $PRM = 7$, and the maximum value of **`duty`** is 127, so the minimum duty you can get is $\frac{1}{312.5\ KHz \times 128} = 0.025\ \mu S$.

Note that in general, the minimum duty is the period of the maximum global frequency signal available in the system: the 40 MHz main system clock. In addition, a period of 15 is as short as you can set it and still get 1024 resolution on the PWM duty cycle.

`period` is one less the integer part of the multiple of 1.6 µS time units that the real time period is desired to be. For example, if we wanted to synthesize an analog signal with a low-pass bandwidth of 40 KHz, the sampling rate would be at least 80 KHz.

Then the real time period is the reciprocal of the sampling rate, which is $\frac{1}{80\ KHz} = 12.5\ \mu S$. We take this value and divide it by 1.6 µS, which yields 7.8. We take the integer part and subtract 1. This is the **period** argument for the **PWM** command; 6 in this case.

If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

It is necessary to force the pin into low output mode by issuing a **STPIND** command only one time any time before you start the first **PWM** command. This can be accomplished with a command such as:

```
STPIND(pin, LOW);
```

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
/* PWM differs from functions like PULSOUT in that you
do not have to continue calling it.
Once the PWM function has been called, C Stamp
will continue to produce the pulse even after the PWM
function is not longer being called. */

  STPIND(3, LOW);
  PWM(3, 512, 250); // Set PWM on pin 3 to a 50% duty
// cycle for a duration of about 400 microseconds
  while(1); // Continuous loop, holds program looping
}
```

rand

WORD rand(void);

The **rand** function generates a sequence of random numbers. Each time it is called, a different random **WORD** value is returned. The range of the value returned is [0, 32,767].

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
```

```
{
  WORD RANDOM_WORD = 0; // Variable RANDOM_WORD of type
// WORD to store value returned by rand function
// initialize RANDOM_WORD to 0

  srand(0x5); // srand function sets starting point for
// random word generation at hexidecimal 5

  while(1){ // Continuous loop
    RANDOM_WORD = rand(); // Call rand function and
// store the value returned in RANDOM_WORD

    if(RANDOM_WORD == 0){ // Check that rand returned a
                          // new value
      STPIND(46, HIGH); // Light LED attached to pin 46
    }
    else{
      STPIND(42, HIGH); // Light LED attached to pin 42
    }

    PAUSE(100); // Pause program for 100 milliseconds
                // before looping
  }
}
```

READ

BYTE READ(WORD location);

The **READ** function reads the value at **location** in data EEPROM and returns the result.

location is a variable/constant/expression that specifies the data EEPROM address from which to read.

The user must ensure that the **location** value is within the bounds of the existing data EEPROM address space.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
/* This program should be used after first downloading
and running the WRITE program so that a value will
first be stored in the EEPROM. The EEPROM will not lose
the information stored there if the CSTAMP was turned
off before this program has been executed.*/

  BYTE EEPROM_VALUE; // Variable EEPROM_VALUE of type
// BYTE to store value returned from READ function

  EEPROM_VALUE = READ(0000); // Call READ function to
// retrieve value stored in EEPROM at location '0000'
// this value is then stored in the variable
// EEPROM_VALUE

  if(EEPROM_VALUE == 0x64){ // Check to see if value
// stored in EEPROM_VALUE is the hexadecimal value '64'
    STPIND(41, HIGH); // Turn on LED attached to pin 41
  }
  END();
}
```

REV

WORD REV(WORD value, BYTE num);

The **REV** function returns a reversed (mirrored) copy of a specified number of bits **num** of **value**, starting with the right-most bit (least significant bit or "LSB").

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BYTE FORWARD; // Variable FORWARD of type BYTE, used
// to store the bits in their forward direction
  BYTE REVERSED; // Variable REVERSED of type BYTE,
// used to store the bits in their reversed direction
  FORWARD =  0xA2; // Set FORWARD to 'A2' in
// hexadecimal, A2 in hex is equal to 10100010 in
// binary
  REVERSED = REV(FORWARD, 8); // Call function REV to
// reverse the bits in the variable FORWARD and
// store them in the variable REVERSED

/* 10100010 reversed becomes 01000101, the zero can be
ignored and this becomes 1000101 in binary which is
equal to 45 in hexadecimal. */

  if(REVERSED == 0x45){ // Check to see if REVERSED is
                        // equal to hexadecimal 45
    STPIND(42, HIGH); // Turn on LED attached to pin 42
  }
  else{
    STPIND(46, HIGH); // Turn on LED attached to pin 46
  }
}
```

SATH

`float SATH(float num, float limit);`

The **`SATH`** function saturates high or limits the value of **`num`** to specified high **`limit`**. It returns **`num`** if **`num`** is less than **`limit`**, or it returns **`limit`** if **`num`** is greater than or equal to **`limit`**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float LIMIT; // Variable LIMIT of type float to store
               // the upper saturation limit
  float NUMBER; // Variable NUMBER of type float to
// store the value to be compared with LIMIT
  float SATURATION; // Variable SATURATION of type
// float to store the value returned by SATH function

//Example
  LIMIT = 14.671; // Set LIMIT to 14.671
  NUMBER = PI * 10; // Set NUMBER to PI * 10 or 31.416
                  // (PI is defined as 3.1416)

  SATURATION = SATH(NUMBER, LIMIT); // Function SATH
// returns NUMBER or LIMIT which ever is least which is
// then stored in the variable SATURATION

  if (SATURATION == NUMBER){ // Check to see if
// SATURATION is equal to NUMBER
    STPIND(41, HIGH); // If so, light LED attached to
```

```
                    // pin 41
  }

  if (SATURATION == LIMIT){ // Check to see if
// SATURATION is equal to LIMIT
    STPIND(45, HIGH); // If so, light LED attached to
                    // pin 45
  }

  END(); //END function stops program and puts CSTAMP
         // into low power mode.
}
```

SATL

`float SATL(float num, float limit);`

The **`SATL`** function saturates low or limits the value of **`num`** to specified low **`limit`**. It returns **`num`** if **`num`** is greater than **`limit`**, or it returns **`limit`** if **`num`** is less than or equal to **`limit`**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float LIMIT; // Variable LIMIT of type float to store
             // the upper saturation limit
  float NUMBER; // Variable NUMBER of type float to
// store the value to be compared with LIMIT
  float SATURATION; // Variable SATURATION of type
// float to store the value returned by SATL function

//Example
  LIMIT = 5.1799; // Set LIMIT to 5.1799
  NUMBER = PI * 1.5; // Set NUMBER to PI * 1.5 or
                   // 4.7124
                   // (PI is defined as 3.1416)

  SATURATION = SATL(NUMBER, LIMIT); // Function SATH
// returns NUMBER or LIMIT which ever is greatest which
// is then stored in the variable SATURATION

  if (SATURATION == NUMBER){ // Check to see if
// SATURATION is equal to NUMBER
    STPIND(41, HIGH); // If so, light LED attached to
                      // pin 41
  }

  if (SATURATION == LIMIT){ // Check to see if
// SATURATION is equal to LIMIT
    STPIND(45, HIGH); // If so, light LED attached to
                      // pin 45
  }

  END(); //END function stops program and puts the
         // CSTAMP into low power mode.
}
```

SERIN

```
NIBBLE SERIN(BYTE fpin, BIT polarity, float baudrate,
             NIBBLE databits, NIBBLE parity,
             WORD timeout, BYTE buffer[], BYTE N,
             BYTE waiton[]);
```

The **SERIN** function receives asynchronous serial data (e.g., RS-232 data) via the built-in asynchronous serial receiver in the C Stamp (Pin 25). This is the same connection used to download programs to the C Stamp.

fpin is a variable/constant/expression that specifies the I/O pin on which to indicate flow control status. This pin will be set to output mode. If **fpin** is 0, no flow control will be used.

polarity is an argument that specifies how the **fpin** will be used in the cases that the user chooses to utilize flow control. The allowed values are **TRUE** and **FALSE**. If polarity is TRUE, a ONE or HIGH will be used as the value of **fpin** that will signal the sender to send more data, and a ZERO or LOW will be used as the value of **fpin** that will signal the sender not to send more data and wait for the value of **fpin** to be ONE or HIGH again. If polarity is FALSE, a ZERO or LOW will be used as the value of **fpin** that will signal the sender to send more data, and a ONE or HIGH will be used as the value of **fpin** that will signal the sender not to send more data and wait for the value of **fpin** to be ZERO or LOW again.

baudrate is a variable/constant/expression (4.8, 9.6, 19.2, 38.4, 57.6, 76.8, 96, 115.2, 300, 500, 625, 1250, or 2500) that specifies serial timing in 1000's of bits per second (Kbps).

`databits` is a variable/constant/expression (7 or 8) that specifies the length of each data unit ("byte") to be received.

`parity` is an argument that specifies which type of parity to check for each received byte. If parity is used, the parity bit is expected to be the Most Significant Bit (MSB) of the data unit. The allowed values for **`parity`** are in the table below.

`parity` *code*	*value*	*Meaning*
NOPAR	0	No parity
EVENPAR	1	Even parity
ODDPAR	2	Odd parity

`timeout` is a variable/constant/expression (0 – 65535) that tells **SERIN** how long to wait in mS for incoming data. If data does not arrive in time or if the buffer gets full before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait until it receives enough data to fill the buffer before it returns. The usage of timeouts applies to each data unit being received.

`buffer` is the name of the array of **BYTES** that the function will use as storage to return the received data to the calling function. This can be an array of length equal to 1 byte. The maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) to be placed in **`buffer`**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **N**-1).

If you do not use the timeout feature, the commands wait forever to get the number of bytes that you requested. However, the number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

`waiton` is a Null ('\0') terminated string to match before the function starts filling the **`buffer`**. If this argument is ZERO, the function will not do any matching, and will start filling the buffer immediately with received data. Note that if the matching string is not found in the incoming data stream and timeout is not used, your program will get stuck in this function forever, until power to the C Stamp is cycled, or until a RESET condition is applied.

One of the many possible uses of the **waiton** capability is to process incoming data from a GPS receiver peripheral, and only receive certain NMEA sentences.

An illustration of the usage of the **waiton** capability would be as follows:

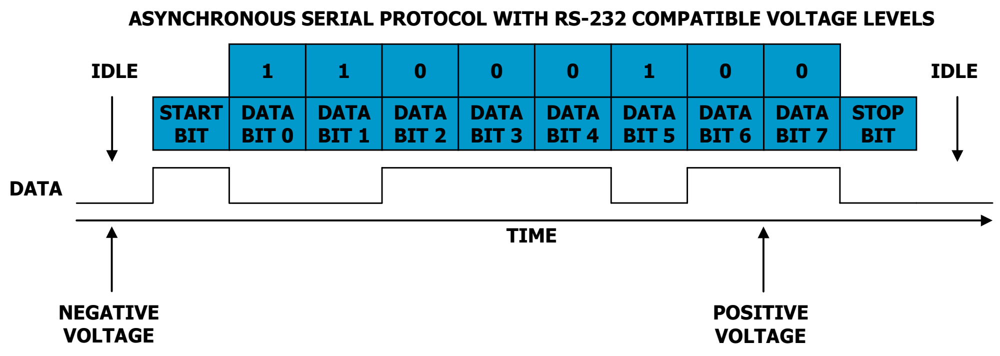
```
//  somewhere at the beginning of the program
NIBBLE result;
RAM BYTE buffer[255];
RAM BYTE waiton[] = “$GPRMC”;  //  no restriction on
                               //  the length
//  :
//  :
//  later on ...
result = SERIN(0, 0, 4.8, 8, 0, 2000, buffer, 255,
               waiton);
```

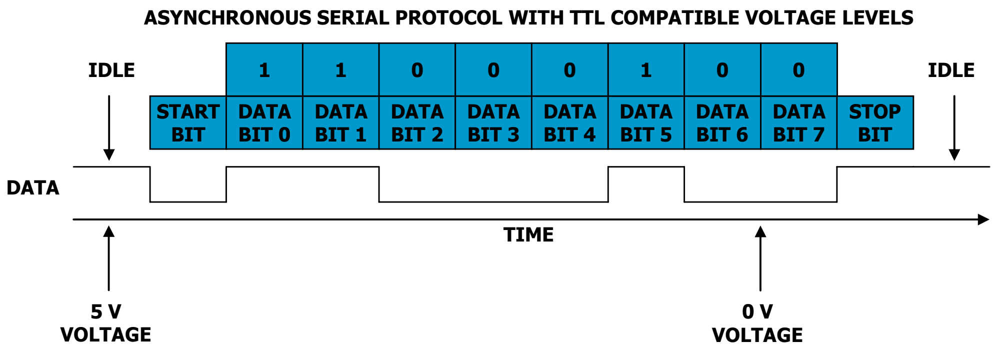
On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
SERIN_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
SERIN_BUFULL	1	Function returned with the buffer full of data.
SERIN_PARERR	2	A parity error occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, it indicates that no data was received.
SERIN_TIMOUT	3	The function exited after a timeout condition occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the timeout occurred. If this location contains the

return code	*value*	*Meaning*
		value 0xFF, it indicates that no data was received.
SERIN_OVRERR	4	An overrun error occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, it indicates that no data was received.

NOTE: SERIN/SEROUT are to be used with RS-232 type interfaces, such as the interface to a PC; while **SERIN2/SEROUT2** are to be used with non RS-232 interfaces. For an RS232 interface, a logic 1 is a low voltage and a logic 0 is a high voltage. For a non RS-232 interface also called TTL interface, a logic 1 is 5V and a logic 0 is 0V. So the voltage levels from **SERIN2/SEROUT2** will be inverted with respect to those from **SERIN/SEROUT**. RS-232 voltage levels are usually negative for the low voltage and positive for the high. However, some components use the RS-232 signaling with 0-5V, so 0V is a logic 1 and 5V is a logic 0. For such an interface, you could still use **SERIN2/SEROUT2** by inverting the input and output of the pins. If this is the case, a CS459020 chip can be used for that. The figures below clarify these points further.





EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE RETURNED; // Declare variable RETURNED of type
                   // NIBBLE to store value returned by
                   // SERIN function
  BYTE buffer[10]; // Declare array buffer of type BYTE
                   // to store input from terminal
// Declare array string1 of type BYTE to store the new
// line character (\n) followed by the carriage return
// (\r) character and the string "Hello "
  BYTE string1[] = "\n\rHello ";
// Declare array string2 of type BYTE to store the new
// line character followed by the carriage return
// character and then the string "Enter your name
// (using ten characters total)"
  BYTE string2[] =
  "\n\rEnter your name (using ten characters total): ";

// Call function SEROUT
  SEROUT(0, 0, 9.6, 0, 8, 0, 0, string2, 48);
  PAUSE(100); // Pause for 100 milliseconds
// Call function SERIN to get data from terminal, store
// value returned in RETURNED
  RETURNED = SERIN(0, 0, 9.6, 8, 0, 0, buffer, 10, 0);

// Check if SERIN function executed successfully
  if(RETURNED == 1){
    STPIND(42, HIGH); // Light LED attached to pin 42
  }
  PAUSE(100); // Pause for 100 milliseconds
// Call function SEROUT
  SEROUT(0, 0, 9.6, 0, 8, 0, 0, string1, 9);
  PAUSE(100); // Pause for 100 milliseconds
// Call function SEROUT
  SEROUT(0, 0, 9.6, 0, 8, 0, 0, buffer, 10);
  PAUSE(100); // Pause for 100 milliseconds

// Stop program, C Stamp enters low power mode
  END();
}
```

SEROUT

```
NIBBLE SEROUT(BYTE fpin, BIT polarity, float baudrate,
              WORD pace, NIBBLE databits,
              NIBBLE parity, WORD timeout,
              BYTE buffer[], BYTE N);
```

The **SEROUT** function transmits asynchronous serial data (e.g., RS-232 data) via the built-in asynchronous serial transmitter in the C Stamp (Pin 24). This is the same connection used to download programs to the C Stamp.

fpin is a variable/constant/expression that specifies the I/O pin on which to indicate flow control status. This pin will be set to input mode. If **fpin** is 0, no flow control will be used.

polarity is an argument that specifies how the **fpin** will be used in the cases that the user chooses to utilize flow control. The allowed values are **TRUE** and **FALSE**. If polarity is TRUE, a ONE or HIGH from the receiver will be interpreted as a request to send more data, and a ZERO or LOW will be interpreted as not to send more data and wait for the value of **fpin** to be ONE or HIGH again. If polarity is FALSE, a ZERO or LOW from the receiver will be interpreted as a request to send more data, and a ONE or HIGH will be interpreted as not to send more data and wait for the value of **fpin** to be ZERO or LOW again.

baudrate is a variable/constant/expression (4.8, 9.6, 19.2, 38.4, 57.6, 76.8, 96, 115.2, 300, 500, 625, 1250, or 2500) that specifies serial timing in 1000's of bits per second (Kbps).

pace is a variable/constant/expression (0 – 65535) that determines the length of the pause between transmitted data units ("bytes") in mS. If **pace** is 0, there is no pause between sending data units. NOTE: **pace** cannot be used simultaneously with **timeout** and **fpin**.

databits is a variable/constant/expression (7 or 8) that specifies the length of each data unit ("byte") to be sent. If 7 data bits are selected for transmission with no parity, a full byte will be sent with the MSB set to ZERO.

parity is an argument that specifies which type of parity to use for each sent byte. If parity is used, the parity bit is the MSB of the data unit. The allowed values for **parity** are in the table below.

parity *code*	*value*	*Meaning*
NOPAR	0	No parity
EVENPAR	1	Even parity
ODDPAR	2	Odd parity

timeout is a variable/constant/expression (0 – 65535) that tells **SEROUT** how long to wait in mS for the receiver to request more data via **fpin** if flow control is used. If data is not requested before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait indefinitely until it receives a data request from the receiver. The usage of timeouts applies to each data unit being sent.

buffer is the name of the array of **BYTES** that the function will send. This can be an array of length equal to 1 byte. The maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) in **buffer** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **N**-1). **N** can also have a special value of 0. During transmission, if **N** is 0, the command will exit after the first null character '\0' in the **buffer** is detected, unless a timeout condition is reached. The null character is also sent. This is useful when you have a **buffer** that has been initialized to a constant string, which are automatically terminated by a null. Then you can send it without having to count the number of characters in the **buffer**. For example, the code below illustrates how the string "Hello World!" can be sent using this feature without counting the number of characters in it.

```
BYTE hello[] = "Hello World!";
:
SEROUT(0, 0, 4.8, 0, 8, 0, 0, hello, 0);
```

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

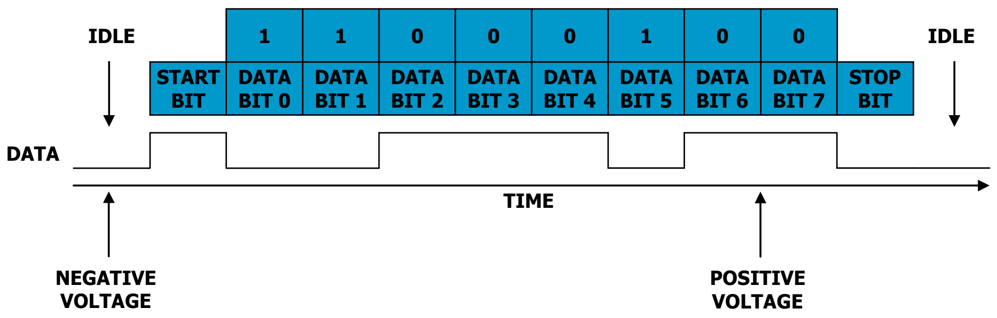
and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

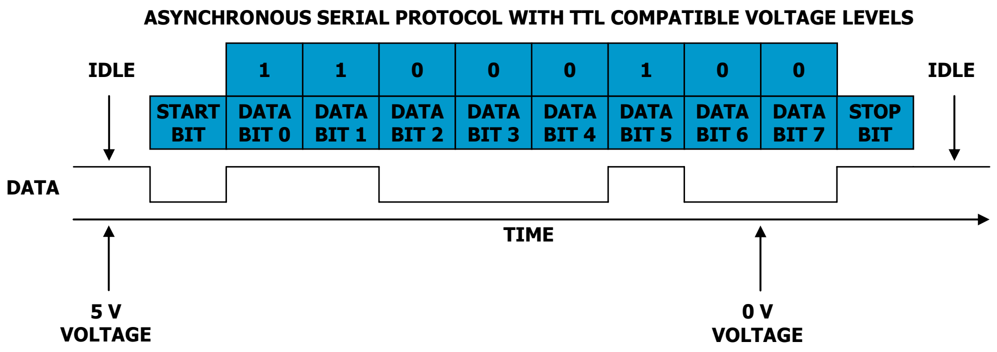
On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
SEROUT_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
SEROUT_BUFEMP	1	Function returned after transmitting all data in the buffer.
SEROUT_TIMOUT	2	The function exited after a timeout condition occurred. Not all data was sent.

NOTE: SERIN/SEROUT are to be used with RS-232 type interfaces, such as the interface to a PC; while **SERIN2/SEROUT2** are to be used with non RS-232 interfaces. For an RS232 interface, a logic 1 is a low voltage and a logic 0 is a high voltage. For a non RS-232 interface also called TTL interface, a logic 1 is 5V and a logic 0 is 0V. So the voltage levels from **SERIN2/SEROUT2** will be inverted with respect to those from **SERIN/SEROUT**. RS-232 voltage levels are usually negative for the low voltage and positive for the high. However, some components use the RS-232 signaling with 0-5V, so 0V is a logic 1 and 5V is a logic 0. For such an interface, you could still use **SERIN2/SEROUT2** by inverting the input and output of the pins. If this is the case, a CS459020 chip can be used for that. The figures below clarify these points further.

ASYNCHRONOUS SERIAL PROTOCOL WITH RS-232 COMPATIBLE VOLTAGE LEVELS





EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
// Array string1 of type BYTE to store the string "ABC"
// with a new line and carriage return character
// following
  BYTE string1[] = "ABC\n\r";
// Array string2 of type byte to store the string "DEF"
// with a new line and carriage return character
// following
  BYTE string2[] = "DEF\n\r";
// Variable RESULT of type NIBBLE to store value
// returned by SEROUT
  NIBBLE RESULT;

// Call function SEROUT and store value returned in
// variable RESULT
  RESULT = SEROUT(0, 0, 9.6, 0, 8, 0, 0, string1, 5);

// Check that SEROUT function executed successfully
  if(RESULT == 1){
    STPIND(42, HIGH); // Light LED attached to pin 42
  }

// Call function SEROUT and store value returned in
// variable RESULT
  RESULT = SEROUT(0, 0, 9.6, 0, 8, 0, 0, string2, 5);

// Check that SEROUT function executed successfully
```

```
  if(RESULT == 1){
    STPIND(46, HIGH); // Light LED attached to pin 46
  }

  END(); // End program, C Stamp enters low power mode
}
```

SERIN2

```
NIBBLE SERIN2(BYTE fpin, BIT polarity, float baudrate,
              NIBBLE databits, NIBBLE parity,
              WORD timeout, BYTE buffer[], BYTE N,
              BYTE RXpin, BYTE waiton[]);
```

The **SERIN2** function receives asynchronous serial data via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25).

fpin is a variable/constant/expression that specifies the I/O pin on which to indicate flow control status. This pin will be set to output mode. If **fpin** is 0, no flow control will be used.

polarity is an argument that specifies how the **fpin** will be used in the cases that the user chooses to utilize flow control. The allowed values are **TRUE** and **FALSE**. If polarity is TRUE, a ONE or HIGH will be used as the value of **fpin** that will signal the sender to send more data, and a ZERO or LOW will be used as the value of **fpin** that will signal the sender not to send more data and wait for the value of **fpin** to be ONE or HIGH again. If polarity is FALSE, a ZERO or LOW will be used as the value of **fpin** that will signal the sender to send more data, and a ONE or HIGH will be used as the value of **fpin** that will signal the sender not to send more data and wait for the value of **fpin** to be ZERO or LOW again.

baudrate is a variable/constant/expression (1.2, 2.4, 4.8, 9.6, 19.2, 38.4, 57.6, 76.8, 96, or 115.2) that specifies serial timing in 1000's of bits per second (Kbps).

databits is a variable/constant/expression (7 or 8) that specifies the length of each data unit ("byte") to be received.

parity is an argument that specifies which type of parity to check for each received byte. If parity is used, the parity bit is expected to be the Most Significant Bit (MSB) of the data unit. The allowed values for **parity** are in the table below.

parity *code*	*value*	*Meaning*
NOPAR	0	No parity

parity *code*	*value*	*Meaning*
EVENPAR	1	Even parity
ODDPAR	2	Odd parity

timeout is a variable/constant/expression (0 – 65535) that tells **SERIN2** how long to wait in mS for incoming data. If data does not arrive in time or if the buffer gets full before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait until it receives enough data to fill the buffer before it returns. The usage of timeouts applies to each data unit being received.

buffer is the name of the array of **BYTES** that the function will use as storage to return the received data to the calling function. This can be an array of length equal to 1 byte. The maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) to be placed in **buffer**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **N**-1).

If you do not use the timeout feature, the commands wait forever to get the number of bytes that you requested. However, the number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

RXpin is a variable/constant/expression that specifies the I/O pin to be used as a receiver. This pin will be set to input mode. **RXpin** cannot be equal to **fpin**.

waiton is a Null ('\0') terminated string to match before the function starts filling the **buffer**. If this argument is ZERO, the function will not do any matching, and will start filling the buffer immediately with received data. Note that if the matching string is not found in the incoming data stream and timeout is not used, your program will get stuck in this function forever, until power to the C Stamp is cycled, or until a RESET condition is applied.

One of the many possible uses of the **waiton** capability is to process incoming data from a GPS receiver peripheral, and only receive certain NMEA sentences.

An illustration of the usage of the **waiton** capability would be as follows:

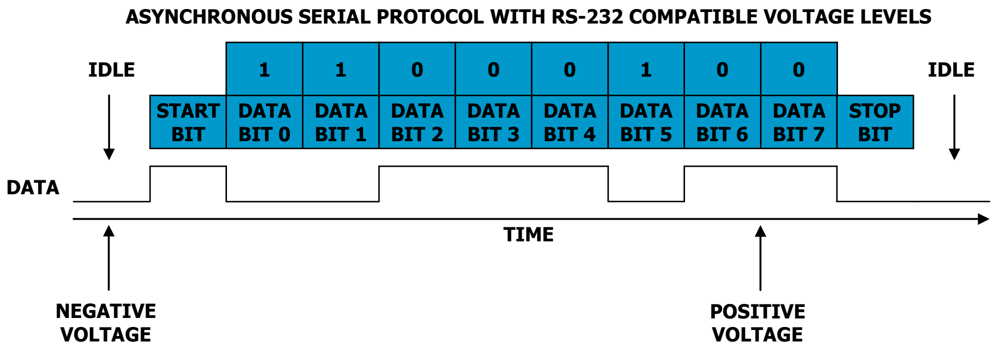
```
//  somewhere at the beginning of the program
NIBBLE result;
RAM BYTE buffer[255];
RAM BYTE waiton[] = "$GPRMC";  //  no restriction on
                               //  the length
//  :
//  :
//  later on ...
//  result = SERIN2(0, 0, 4.8, 8, 0, 2000, buffer, 255,
                    RXpin, waiton);  //  on some
                                     //  previously
                                     //  defined RXpin
```

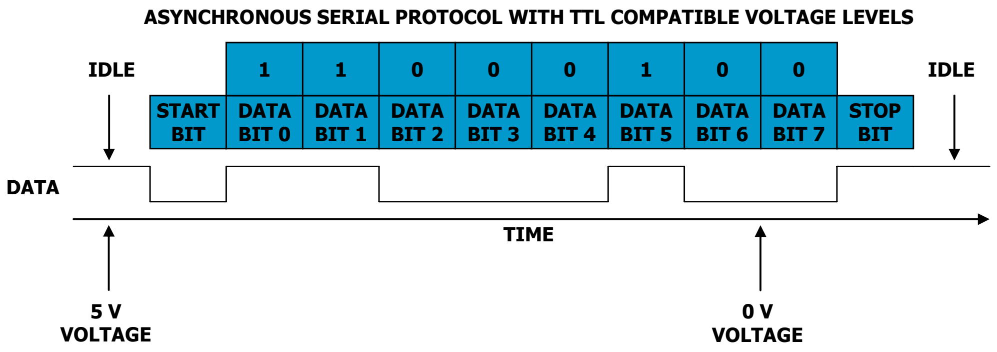
On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
SERIN_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
SERIN_BUFULL	1	Function returned with the buffer full of data.
SERIN_PARERR	2	A parity error occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, this indicates that no data was received.
SERIN_TIMOUT	3	The function exited after a timeout condition occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the timeout occurred. If this location contains the value 0xFF, this indicates that no data was received.
SERIN_OVRERR	4	An overrun error occurred.

return code	*value*	*Meaning*
		The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, this indicates that no data was received.

NOTE: **SERIN**/**SEROUT** are to be used with RS-232 type interfaces, such as the interface to a PC; while **SERIN2**/**SEROUT2** are to be used with non RS-232 interfaces. For an RS232 interface, a logic 1 is a low voltage and a logic 0 is a high voltage. For a non RS-232 interface also called TTL interface, a logic 1 is 5V and a logic 0 is 0V. So the voltage levels from **SERIN2**/**SEROUT2** will be inverted with respect to those from **SERIN**/**SEROUT**. RS-232 voltage levels are usually negative for the low voltage and positive for the high. However, some components use the RS-232 signaling with 0-5V, so 0V is a logic 1 and 5V is a logic 0. For such an interface, you could still use **SERIN2**/**SEROUT2** by inverting the input and output of the pins. If this is the case, a CS459020 chip can be used for that. The figures below clarify these points further.





EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE RETURNED;  // Declare variable RETURNED of
                    // type NIBBLE to store value
                    // returned by SERIN2 function
  int i;  //Loop counter

  BYTE buffer[8];  // Declare array buffer of type BYTE
                   // to store input from terminal
  BIT ON;
  BYTE pins[8];

/* The array pins is defined element by element so that
   pin 39 is the LSB and pin 46 is the MSB */
  pins[0]=39; pins[1]=40;
  pins[2]=41; pins[3]=42;
  pins[4]=43; pins[5]=44;
  pins[6]=45; pins[7]=46;

  for(i=0; i<8; i++) buffer[i]=0;

  ON = BUTTON(37, LOW, HIGH, 5);
  if (ON){
    while(1){
      SERIN2(0, 0, 4.8, 8, 0, 0, buffer, 8, 17, 0);
      for(i=0; i<8; i++){
        BYTEOUT(buffer[i], pins);
        PAUSE(1000);
      }
    }
  }
  END();
}
```

SEROUT2

```
NIBBLE SEROUT2(BYTE fpin, BIT polarity, float baudrate,
               WORD pace, NIBBLE databits,
               NIBBLE parity, WORD timeout,
                    BYTE buffer[], BYTE N, BYTE TXpin);
```

The **SEROUT2** function transmits asynchronous serial data via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25).

fpin is a variable/constant/expression that specifies the I/O pin on which to indicate flow control status. This pin will be set to input mode. If **fpin** is 0, no flow control will be used.

polarity is an argument that specifies how the **fpin** will be used in the cases that the user chooses to utilize flow control. The allowed values are **TRUE** and **FALSE**. If polarity is TRUE, a ONE or HIGH from the receiver will be interpreted as a request to send more data, and a ZERO or LOW will be interpreted as not to send more data and wait for the value of **fpin** to be ONE or HIGH again. If polarity is FALSE, a ZERO or LOW from the receiver will be interpreted as a request to send more data, and a ONE or HIGH will be interpreted as not to send more data and wait for the value of **fpin** to be ZERO or LOW again.

baudrate is a variable/constant/expression (1.2, 2.4, 4.8, 9.6, 19.2, 38.4, 57.6, 76.8, 96, or 115.2) that specifies serial timing in 1000's of bits per second (Kbps).

pace is a variable/constant/expression (0 – 65535) that determines the length of the pause between transmitted data units ("bytes") in mS. If **pace** is 0, there is no pause between sending data units. NOTE: **pace** cannot be used simultaneously with **timeout** and **fpin**.

databits is a variable/constant/expression (7 or 8) that specifies the length of each data unit ("byte") to be sent. If 7 data bits are selected for transmission with no parity, a full byte will be sent with the MSB set to ZERO.

parity is an argument that specifies which type of parity to use for each sent byte. If parity is used, the parity bit is the MSB of the data unit. The allowed values for **parity** are in the table below.

parity *code*	*value*	*Meaning*
NOPAR	0	No parity
EVENPAR	1	Even parity
ODDPAR	2	Odd parity

timeout is a variable/constant/expression (0 – 65535) that tells **SEROUT2** how long to wait in mS for the receiver to request more data via **fpin** if flow control is used. If data is not requested before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait indefinitely until it receives a data request from the receiver. The usage of timeouts applies to each data unit being sent.

buffer is the name of the array of **BYTES** that the function will send. This can be an array of length equals to 1 byte. The maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) in **buffer** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **N**-1). **N** can also have a special value of 0. During transmission, if **N** is 0, the command will exit after the first null character '\0' in the **buffer** is detected, unless a timeout condition is reached. The null character is also sent. This is useful when you have a **buffer** that has been initialized to a constant string, which are automatically terminated by a null. Then you can send it without having to count the number of characters in the **buffer**. For example, the code below illustrates how the string "Hello World!" can be sent using this feature without counting the number of characters in it.

```
BYTE hello[] = "Hello World!";
:
// Send on pin 1
SEROUT2(0, 0, 4.8, 0, 8, 0, 0, hello, 0, 1);
```

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

TXpin is a variable/constant/expression that specifies the I/O pin to be used as a transmitter. This pin will be set to output mode. **TXpin** cannot be equal to **fpin**.

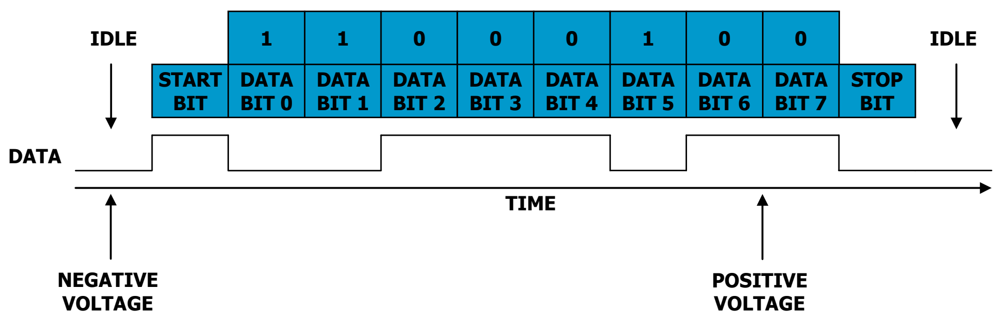
On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
SEROUT_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.

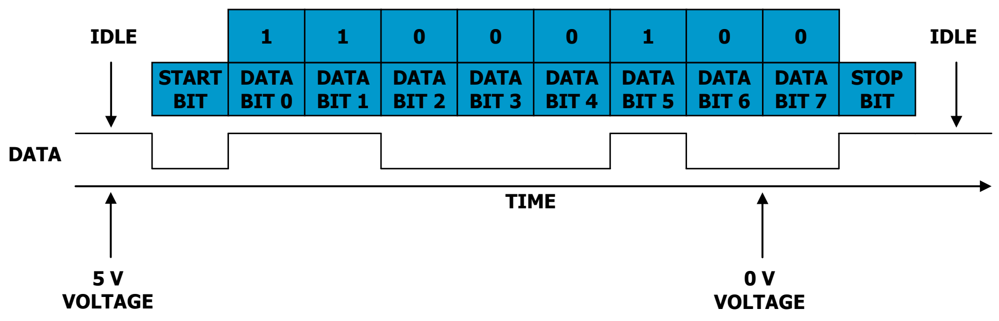
return code	*value*	*Meaning*
SEROUT_BUFEMP	1	Function returned after transmitting all data in the buffer.
SEROUT_TIMOUT	2	The function exited after a timeout condition occurred. Not all data was sent.

NOTE: SERIN/SEROUT are to be used with RS-232 type interfaces, such as the interface to a PC; while **SERIN2/SEROUT2** are to be used with non RS-232 interfaces. For an RS232 interface, a logic 1 is a low voltage and a logic 0 is a high voltage. For a non RS-232 interface also called TTL interface, a logic 1 is 5V and a logic 0 is 0V. So the voltage levels from **SERIN2/SEROUT2** will be inverted with respect to those from **SERIN/SEROUT**. RS-232 voltage levels are usually negative for the low voltage and positive for the high. However, some components use the RS-232 signaling with 0-5V, so 0V is a logic 1 and 5V is a logic 0. For such an interface, you could still use **SERIN2/SEROUT2** by inverting the input and output of the pins. If this is the case, a CS459020 chip can be used for that. The figures below clarify these points further.

ASYNCHRONOUS SERIAL PROTOCOL WITH RS-232 COMPATIBLE VOLTAGE LEVELS



ASYNCHRONOUS SERIAL PROTOCOL WITH TTL COMPATIBLE VOLTAGE LEVELS



EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
// Array string1 of type BYTE to store the string
// "AWIT"
  BYTE string1[] = "AWIT";
// Array string2 of type byte to store the string
// "TECH"
  BYTE string2[] = "TECH";

// Variable RESULT of type NIBBLE to store value
// returned by SEROUT2
  NIBBLE RESULT;

// Call function SEROUT2 and store value returned in
// variable RESULT
  RESULT =
    SEROUT2(0, 0, 9.6, 0, 8, 0, 0, string1, 4, 9);

// Check that SEROUT2 function executed successfully
  if(RESULT == 1) STPIND(42, HIGH);
// Light LED attached to pin 42

// Call function SEROUT2 and store value returned in
// variable RESULT
  RESULT =
    SEROUT2(0, 0, 9.6, 0, 8, 0, 0, string2, 4, 9);

// Check that SEROUT2 function executed successfully
  if(RESULT == 1)    STPIND(46, HIGH);
// Light LED attached to pin 46

  END();  // End program, C Stamp enters low power mode
}
```

SHIFTIN

```
WORD SHIFTIN(BYTE dpin, BYTE cpin, NIBBLE mode,
             BYTE bits);
```

The **SHIFTIN** function shifts data in from a synchronous serial device and returns the received data.

The transfer data rate is about 65 Kbits/S.

dpin is a variable/constant/expression that specifies the I/O pin that will be connected to the synchronous serial device's data output. This pin will be set to input mode.

cpin is a variable/constant/expression that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.

mode is a variable/constant/expression (0 – 3), or one of four predefined symbols, that tells **SHIFTIN** the order in which data bits are to be arranged and the relationship of clock pulses to valid data. The table below shows these relationships.

bits is a variable/constant/expression (1 – 16) specifying how many bits are to be input by **SHIFTIN**.

If the function is not successful, it returns **FALSE**.

Symbol	*Value*	*Meaning*
MSBPRE	0	Data is MSB-first; sample bits before clock pulse
LSBPRE	1	Data is LSB-first; sample bits before clock pulse
MSBPOST	2	Data is MSB-first; sample bits after clock pulse
LSBPOST	3	Data is LSB-first; sample bits after clock pulse

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  WORD GET_WORD = 0; // Variable GET_WORD of type WORD,
// used to store value returned by SHIFTIN function
// from a DM74165N 8-Bit Parallel In/Serial Output
// Shift Register chip

  STPIND(12, HIGH); // Set pin 12 high to enable clock
```

```
                  // inhibit
  STPIND(13, HIGH); // Set Load/Shift to Shift
  PAUSE(5);         // Wait 5 milliseconds
  STPIND(13, LOW);  // Set Load/Shift to Load
  PAUSE(1);         // Wait 1 millisecond
  STPIND(13, HIGH); // Set Load/Shift to Shift
  PAUSE(1);         // Wait 1 millisecond
  STPIND(12, LOW);  // Turn off clock inhibit
  PAUSE(1);

  GET_WORD = SHIFTIN(10, 11, 0, 8); // Call function
// SHIFTIN where the data is going to be received on
// pin 10, the clock is on pin 11 of eight bits long.
// Store the value in the variable GET_WORD

  if(GET_WORD == 0xAA){ // Check if GET_WORD is
// hexidecimal AA or binary 10101010
    STPIND(45, HIGH); // Turn on LED attached to pin 45
  }

  END(); // Stop program from looping, C Stamp enters
         // low power mode
}
```

SHIFTOUT

**BIT SHIFTOUT(BYTE dpin, BYTE cpin, BIT mode,
              WORD outputdata, BYTE bits);**

The **SHIFTOUT** function shifts data out to a synchronous serial device.

The transfer data rate is about 65 Kbits/S.

dpin is a variable/constant/expression that specifies the I/O pin that will be connected to the synchronous serial device's data input. This pin will be set to output mode.

cpin is a variable/constant/expression that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.

mode is a variable/constant/expression (0 – 1), or one of two predefined symbols, that tells **SHIFTOUT** the order in which data bits are to be arranged. The table below shows the values and symbols definitions.

`outputdata` is a variable/constant/expression containing the data to be sent.

`bits` is a variable/constant/expression (1 – 16) specifying how many bits are to be output by **`SHIFTOUT`**.

If the function is successful, it returns **`TRUE`**; otherwise it returns **`FALSE`**.

Symbol	*Value*	*Meaning*
LSBFIRST	0	Data is shifted out LSB-first
MSBFIRST	1	Data is shifted out MSB-first

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  while(1){ // Continuous loop
    SHIFTOUT (10, 11, 1, 0xABCD, 16); // Call SHIFTOUT
// function where pin 10 will be used as the data pin,
// pin 11 used as the clock pin, the data will be
// shifted 'Most Significant Bit' first, the data being
// hexadecimal ABCD or binary 1010101111001101, and the
// total number of bits being 16
  }
}
```

SHIFTOUT2

`BIT SHIFTOUT2(BYTE dpin, BYTE cpin, BIT mode,`
`               WORD outputdata, BYTE bits);`

The **`SHIFTOUT2`** function shifts data out to a synchronous serial device using high impedance as a **`HIGH`** level in the data line.

The transfer data rate is about 33 Kbits/S.

`dpin` is a variable/constant/expression that specifies the I/O pin that will be connected to the synchronous serial device's data input. This pin is restricted to pins from the following list: 28, 29, 31 – 38.

cpin is a variable/constant/expression that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.

mode is a variable/constant/expression (0 – 1), or one of two predefined symbols, that tells **SHIFTOUT2** the order in which data bits are to be arranged. The table below shows the values and symbols definitions.

outputdata is a variable/constant/expression containing the data to be sent.

bits is a variable/constant/expression (1 – 16) specifying how many bits are to be output by **SHIFTOUT2**.

If the function is successful, it returns **TRUE**; otherwise it returns **FALSE**.

Symbol	*Value*	*Meaning*
LSBFIRST	0	Data is shifted out LSB-first
MSBFIRST	1	Data is shifted out MSB-first

SLEEP

BIT SLEEP(WORD duration);

The **SLEEP** function puts the C Stamp into low-power mode for a specified time, and then resumes program execution.

duration is a variable/constant/expression (1 – 65535) that specifies the duration of sleep. The unit of time for **duration** is 1 second.

If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  STPIND(42, HIGH); // Light LED attached to pin 42
  PAUSE(10000);     // Pause for 10 seconds
  STPIND(42, LOW);  // Turn off LED attached to pin 42
  SLEEP(1);      // Call function SLEEP to enter CSTAMP
                 // into sleep mode for 1 second
  TOGGLE(43);       // Toggle pin 43
}
```

SPIIN

```
BIT SPIIN(NIBBLE speed, NIBBLE mode,
          BYTE inputdata[], BYTE n);
```

The **SPIIN** function receives data in master mode from a device using the SPI™ protocol on the C Stamp SPI pins (Pin 28 for the SPI Clock, Pin 29 for the SPI Data In, and Pin 30 for the SPI Data Out). The SPI protocol is a form of high speed synchronous serial communication developed by Motorola Corporation. The I/O pins can be shared between multiple SPI devices. The Data In pin will be set to input mode, and the Clock and Data Out pins will be set to output mode. The Clock Polarity is such that idle state for the clock is a low level. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**. This could mean that there was an error in the arguments of the function or some other problem.

speed is a variable/constant/expression (3 – 1) indicating the reception speed according to the table below.

Symbol	*Value*	*Meaning*
SPISH	3	Data is received at 10.0 Mbits/second
SPISM	2	Data is received at 2.5 Mbits/second
SPISL	1	Data is received at 625.0 Kbits/second

mode is a variable/constant/expression that determines the reception mode according to the table below.

Symbol	*Value*	*Meaning*
SPIM00	0	Input data sampled at middle of data output time Transmit occurs on transition from Idle to active clock state (low to high)
SPIM01	1	Input data sampled at middle of data output time Transmit occurs on transition from active to Idle clock state (high to low)

Symbol	*Value*	*Meaning*
SPIM10	2	Input data sampled at end of data output time Transmit occurs on transition from Idle to active clock state (low to high)
SPIM11	3	Input data sampled at end of data output time Transmit occurs on transition from active to Idle clock state (high to low)

`inputdata` is a **`BYTE`** array to put the received data. The array fills up from the low address 0 to the high address **`n`**-1.

`n` is the number of bytes (1 – 255) to be placed in **`inputdata`**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **`n`**-1).

If you do not use the timeout feature, the commands wait forever to get the number of bytes that you requested. However, the number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE speed = 1;
  NIBBLE mode = 0;
  BYTE inputdata[4];
  BYTE compare[] = "1234";  // Data that was sent for
                            // SPIOUT
  BYTE n = 4;
  BIT flag = 1;
  int i;

  STPIND(42, HIGH);

  while (flag == 1){
    SPIIN(speed, mode, inputdata, n);
//Recieve data on pin 29

    for(i = 0; i<4; i++){
//Check to see if elements of the array are equal
      if(inputdata[i] == compare[i]) flag = 0;
//If equal, do not set flag
      else{
        flag = 1;  //Otherwise, set flag
        break;
      }
    }

    PAUSE(1000);
  }

  STPIND(46, HIGH);
  STPIND(42, LOW);

  END();
}
```

SPIOUT

BIT SPIOUT(NIBBLE speed, NIBBLE mode,
BYTE outputdata[], BYTE n);

The **SPIOUT** function sends data in master mode from a device using the SPI™ protocol on the C Stamp SPI pins (Pin 28 for the SPI Clock, Pin 29 for the SPI Data In, and Pin 30 for the SPI Data Out). The SPI protocol is a form of high speed synchronous serial communication developed by Motorola Corporation. The I/O pins can be shared between multiple SPI devices. The Data In pin will be set to input mode, and the Clock and Data Out pins will be set to output mode. The Clock Polarity is such that idle state for the clock is a low level. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**. This could mean that there was an error in the arguments of the function or some other problem.

speed is a variable/constant/expression (3 – 1) indicating the reception speed according to the table below.

Symbol	*Value*	*Meaning*
SPISH	3	Data is sent at 10.0 Mbits/second
SPISM	2	Data is sent at 2.5 Mbits/second
SPISL	1	Data is sent at 625.0 Kbits/second

mode is a variable/constant/expression that determines the transmission mode according to the table below.

Symbol	*Value*	*Meaning*
SPIM00	0	Input data sampled at middle of data output time Transmit occurs on transition from Idle to active clock state (low to high)
SPIM01	1	Input data sampled at middle of data output time Transmit occurs on transition from active to Idle clock state (high to low)
SPIM10	2	Input data sampled at end of data output time Transmit occurs on transition from Idle to active clock state (low to high)
SPIM11	3	Input data sampled at end of data output time Transmit occurs on transition from active to Idle clock state (high to low)

outputdata is a **BYTE** array with the data to be sent. The array is sent from the low address 0 to the high address **n**-1.

n is the number of bytes (1 – 255) in **outputdata** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **n**-1).

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE speed = 1;
  NIBBLE mode = 0;
  BYTE outputdata[] = "1234";
  BYTE n = 4;

  SPIOUT(speed, mode, outputdata, n);

  END();
}
```

SPIINOUT

```
BIT SPIINOUT(NIBBLE speed, NIBBLE mode,
             BYTE inputdata[],BYTE outputdata[],
             BYTE n, BYTE filler);
```

The **SPIINOUT** function sends data in full-duplex master mode from a device using the SPI™ protocol on the C Stamp SPI pins (Pin 28 for the SPI Clock, Pin 29 for the SPI Data In, and Pin 30 for the SPI Data Out). This function can also be used for half-duplex communication. The SPI protocol is a form of high speed synchronous serial communication developed by Motorola Corporation. The I/O pins can be shared between multiple SPI devices. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**. This could mean that there was an error in the arguments of the function or some other problem.

speed is a variable/constant/expression (3 – 1) indicating the reception speed according to the table below.

Symbol	*Value*	*Meaning*
SPISH	3	Data is sent at 10.0 Mbits/second
SPISM	2	Data is sent at 2.5 Mbits/second

Symbol	*Value*	*Meaning*
`SPISL`	1	Data is sent at 625.0 Kbits/second

`mode` is a variable/constant/expression that determines the transmission mode according to the table below.

Symbol	*Value*	*Meaning*
`SPIM00`	0	Input data sampled at middle of data output time Transmit occurs on transition from Idle to active clock state (low to high)
`SPIM01`	1	Input data sampled at middle of data output time Transmit occurs on transition from active to Idle clock state (high to low)
`SPIM10`	2	Input data sampled at end of data output time Transmit occurs on transition from Idle to active clock state (low to high)
`SPIM11`	3	Input data sampled at end of data output time Transmit occurs on transition from active to Idle clock state (high to low)

`inputdata` is a **`BYTE`** array that will be used to stored the data received over the SPI bus. If **`inputdata`** is **`NULL`**, the received data will be thrown away. The array is filled from the low address 0 to the high address **`n`**-1.

`outputdata` is a **`BYTE`** array with the data to be sent. The array is sent from the low address 0 to the high address **`n`**-1. If **`outputdata`** is **`NULL`**, the value of **`filler`** will be transmitted repeatedly.

`n` is the number of bytes (1 – 255) to be transmitted and received. The buffers are processed from the low to high direction of its index address space (i.e. from 0 to **`n`**-1).

The number of bytes that you send or receive can be any number, as long as it is less than or equal to what you have allocated. For example, you can have

```
BYTE mybufferin[10];
BYTE mybufferout[10];
```

and then tell the command that you will send and receive only 4 bytes. Then the command will finish after the 4 bytes in `mybufferout[0]` through `mybufferout[3]` are sent, and the 4 bytes in `mybufferin[0]` through `mybufferin[3]` are received.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  NIBBLE speed = 1;
  NIBBLE mode = 0;
  BYTE inputdata[4];
  BYTE outputdata[] = "1234";
  BYTE n = 4;

  SPIINOUT(speed, mode, inputdata, outputdata, n, 0);

  END();
}
```

sprintf

`int sprintf(BYTE buf[], const BYTE format[], ...);`

The **`printf`** function writes to **`buf`** the arguments that comprise the argument list as specified by the string **`format`**. The string **`format`** consists of two types of items. The first type is made up of characters that will be written to **`buf`**. The second type contains format commands that define the way the arguments are written. A format command begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format commands, and the format commands and the arguments are matched in order. For example, this **`sprintf`** command

`sprintf(mybuf, “Hi %c %d %s”, ‘U’, 10, mystr);`

writes “Hi U 10 there!” to **`mybuf`** if **`mystr`**`[] = “there!”`.

If there are insufficient arguments with which to match the **`format`** commands, the contents of **`buf`** is undefined. If there are more arguments than format commands, then the remaining arguments are discarded. The format commands are shown here.

Code	*Format Meaning*
`%c`	Character
`%d`	Signed decimal integer
`%o`	Unsigned octal
`%s`	String of characters
`%u`	Unsigned decimal integers
`%x`	Unsigned hexadecimal (lowercase letters)
`%X`	Unsigned hexadecimal (uppercase letters)
`%%`	Writes a % sign

The **`sprintf`** function returns the number of characters actually written. This number includes a null terminator that **`sprintf`** always appends to its resulting string. A negative return value indicates that an error has taken place.

The **`format`** commands may have modifiers that specify the field width, the number of decimal places, and a left justification flag. An integer placed between the **`%`** sign and the format command acts as a minimum field width specification. This pads the output with blanks or zeros to ensure that it is at least a certain minimum length. If the string or number is greater than the minimum, it will be written in full even if it outruns the minimum. The default padding is done with spaces. If you wish to pad with zeros place a 0 before the field width specification. For example, **`%05d`** will pad a number of less than 5 digits with zeros so its total length is five.

You can also place a decimal point followed by the number after the field width specification. For example, **`%5.7s`** will write a string that will be at least five characters long and will not exceed seven characters. If the string is longer than the maximum field width, the excess characters will be truncated off of the end.

By default, all output is right-justified. If the field width is larger than all the data printed on the right edge of the field, the data will be placed on the right edge of the field. You can force the information to be left-justified by placing a minus sign directly after the **`%.`** For example, **`%-10d`** will left-justify an integer number in a ten character field.

If you precede the x format code with a #, the hexadecimal number will be written with a 0x prefix. If you precede the o format code with a #, the octal value will be written with a 0 prefix. The # cannot be applied to any other format specification.

The minimum field width and precision specifications may be provided by arguments to **`sprintf`** instead of by constants. To accomplish this, use an asterisk (**`*`**) as a placeholder. When the format string is scanned, **`sprintf`** will match the *s to the arguments in the order in which they occur.

srand

`void srand(WORD seed);`

The **`srand`** function is used to set a starting point for the random sequence generated by **`rand`**. The **`rand`** function will always return the same sequence of integers when identical seed values are used. If **`rand`** is called without **`srand`** having been called first, the sequence of numbers generated will be the same as if **`srand`** had been called with a seed value of 1.

SSCANF

`int SSCANF(BYTE buffer[], format, ...);`

The **`SSCANF`** function is a general-purpose input routine that reads the buffer and stores the information in the variables supplied in its argument list. It can read all of the built-in data types.

`buffer` is a Null (**`'\0'`**) terminated string to parse.

The control string **`format`** consists of three classifications of characters:

- Format specifiers
- White-space characters
- Non-white-space characters

The input format specifiers are preceded by a **`%`** sign and tell **`SSCANF`** which type of data to be read next. The **`SSCANF`** codes are matched in order, with the variables receiving the input in the argument list. For example, **`%s`** reads a string while **`%u`** reads a **`WORD`**.

The **`format`** string is read left to right, and the format codes are matched, in order, with the arguments that comprise the argument list.

A white-space character in the **format** strings causes **SSCANF** to skip over one or more white-space characters in the **buffer**. A white-space character is a space, a tab, or a new line character. In essence, one white-space character in the control string will cause **SSCANF** to read, but not store, any number (including zero) of white-space characters up to the first non-white-space character.

A non-white-space character in the **format** string causes **SSCANF** to read and discard a matching character. For example, **"%u,%u"** causes **SSCANF** to first read an **WORD**, then read and discard a comma, and finally read another **WORD**. If the specified character is not found, **SSCANF** will terminate.

All the variables used to receive values through **SSCANF** must be passed as a mono-array. This means that all arguments must be the array name of an array that contains a single element. For example, if you wanted to read a **WORD** into the variable **count**, you would use the following **SSCANF** call.

```
SSCANF(buffer, "%u", count);
```

Then, to use **count**, you would access it as **count[0]**.

Strings will be read into BYTE arrays, and the array name, without any index, is the argument. So, to read a string into the character array **address**, you would use

```
SSCANF(buffer, "%s", address);
```

When reading strings, it is important to remember that input will stop when the first white-space character if the length of the string is not specified.

The input data items must be separated by spaces, tabs, or new-line characters. Punctuation such as commas, semicolons, and the like do not count as separators. This means that

```
SSCANF(buffer, "%u%u", r, c);
```

will accept an input of 10 20, but will fail with 10,20.

When an ***** is placed after the **%** and before the format code, it will read the data of the specified type but suppress its assignment. Thus the following command

```
SSCANF(buffer, "%u%*1s%u", x, y);
```

given the input "10/20" will place the value 10 into **x[0]**, discard the divide by sign, and give **y[0]** the value of 20.

The format commands can specify a maximum field length modifier. This is an integer number placed between the **%** and the format command code that limits the number of characters read for any field. For example, if you wanted to place no more than 20 characters into address, then you would write

```
SSCANF(buffer, “%20s”, address);
```

Input for a field may terminate before the maximum field length is reached if the Null end of buffer is encountered.

Characters can also be used in the control string – including spaces, tabs, and new-line characters – will be used to match and discard characters from the buffer. Any character that matches is discarded. For example, given the input “10t20”,

```
SSCANF(buffer, “%ut%u”, x, y);
```

will place 10 into **`x[0]`** and 20 into **`y[0]`**. The **`‘t’`** is discarded because of the **`‘t’`** in the control string.

The **`SSCANF`** function returns a number equal to the number of fields that were successfully assigned values, including zero. This number will not include fields that were read but not assigned because of the **`*`** modifier was used to suppress the assignment. -1 is returned if an error occurs before the first field is assigned.

One of the many possible uses of the **`SSCANF`** command is to parse NMEA sentences from a GPS receiver peripheral.

Another illustration of the usage of the **`SSCANF`** would be as follows:

```
//  somewhere at the beginning of the program
RAM BYTE buffer[255];
RAM BYTE serString[1];
RAM WORD speed[1];
//  :
//  :
//  later on ...
SSCANF(buffer, “%*12s%1s%*26s%u”, serString, speed);
```

STOP

```
void STOP(void);
```

The **`STOP`** function stops program execution.

`STOP` prevents the C Stamp from executing any further instructions until it is reset. The following actions will reset the C Stamp: Reset line is activated or the power is cycled off and back on.

STOP does not put the C Stamp into low-power mode. The C Stamp draws just as much current as if it were actively running program instructions.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT STARTBUTTON; // Declare variable STARTBUTTON of
                   // type BIT

  STARTBUTTON = BUTTON(37, LOW, HIGH, 5); // Use
// function BUTTON to see if the utility button at pin
// 37 has been pressed and store value returned in
// STARTBUTTON

  while(STARTBUTTON){ // Loop that begins when
                      // STARTBUTTON is TRUE (1)
    STPIND(42, HIGH); // Turns on the LED attached to
                      // pin 42

/* The STOP function stops the program from executing
any further but the CSTAMP will draw just as much power
as if a program was running. Nothing placed after a
STOP statement will be executed since the CSTAMP stops
the program once it encounters the STOP function. */

    STOP(); // STOP function is executed, program
            // stops.
    STPIND(46, HIGH); // This statement will never get
                      // executed
    STPIND(42, LOW);  // This statement will never get
                      // executed
  }
}
```

STPIND

BIT STPIND(BYTE pin, BIT value);

The **STPIND** function sets **pin** to digital output mode and to **value**. If the operation is successful, it returns a value of **TRUE**; otherwise, it returns a value of **FALSE**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  while (1){  //Infinite Loop
    STPIND(45, HIGH);  //Set Pin High
    PAUSE(500);  //Pause for 500 ms
    STPIND(45, LOW);  //Set Pin Low
    PAUSE(500);  //Pause for 500 ms
  }
}
```

STPINPU

`void STPINPU(void);`

The **`STPINPU`** function sets (enables) the built-in pull-up resistors available in the module (Pins 31 – 38).

strcat

`void strcat(BYTE str1[], const BYTE str2[]);`

The **`strcat`** function concatenates a copy of **`str2`** to **`str1`** and terminates **`str1`** with a null. The null terminator originally ending **`str1`** is overwritten by the first character of **`str2`**. The string **`str2`** is untouched by the operation.

No bounds-checking takes place, so it is the programmer's responsibility to ensure that **`str1`** is large enough to hold both its original contents and those of **`str2`**.

strcmp

`int strcmp(BYTE str1[], const BYTE str2[]);`

The **`strcmp`** function lexicographically compares two strings and returns an integer based on the outcome as shown below:

- Less than 0 if **`str1`** is less than **`str2`**
- 0 if **`str1`** is equal to **`str2`**
- Greater than 0 if **`str1`** is greater than **`str2`**

As an example, if str1 and str2, and result are declared as

```
BYTE str1[] = {abc};
BYTE str2[] = {def};
int result;
```

The following calls will result in the following

```
// result will be a negative number
result = strcmp(str1, str2);

// result will be zero  -  if str2 is declared
// as BYTE str2[] = {abc}; result will be zero too
result = strcmp(str1, str1);

// result will be positive
result = strcmp(str2, str1);
```

strcpy

void strcpy(BYTE str1[], const BYTE str2[]);

The **strcpy** function is used to copy the contents of **str2** into **str1**. **str2** must be null terminated string.

strlen

BYTE strlen(BYTE str[]);

The **strlen** function returns the length of the null-terminated string **str**. The null is not counted.

strncat

```
void strncat(BYTE str1[], const BYTE str2[],
             BYTE count);
```

The **strncat** function concatenates no more than **count** characters of **str2** to the string **str1**, and terminates **str1** with a null. The null terminator originally ending **str1** is overwritten by the first character of **str2**. The string **str2** is untouched by the operation.

No bounds-checking takes place, so it is the programmer's responsibility to ensure that **str1** is large enough to hold both its original contents and those of **str2**.

strncpy

`void strncpy(BYTE str1[], const BYTE str2[],`
`             BYTE count);`

The **`strncpy`** function is used to copy up to **`count`** characters from **`str2`** into **`str1`**. **`str2`** must be null terminated string. If the string in **`str2`** has less than **`count`** characters, then nulls will be appended to the end of **`str1`** until **`count`** characters have been copied.

Alternately, if **`str2`** is longer than **`count`** characters, then the resultant string **`str1`** will not be null-terminated.

TOGGLE

`BIT TOGGLE(BYTE pin);`

The **`TOGGLE`** function inverts the state of an output pin.

`pin` is a variable/constant/expression that specifies on which I/O pin to switch logic state. This pin will be placed into output mode.

`TOGGLE` sets a pin to output mode and inverts the output state of the pin, changing 0 (**`LOW`**) to 1 (**`HIGH`**) and 1 (**`HIGH`**) to 0 (**`LOW`**).

If the function is successful, it returns **`TRUE`**; otherwise it returns **`FALSE`**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT ON; // Variable ON of type BIT to store value
// from utility button to determine if LED should be on
// or off.

  STPIND(43, HIGH); // Start with LED attached to pin
                    // 43 on
  while(1){
    ON = BUTTON(37, LOW, HIGH, 5); // Check if utility
// button at pin 37 has been pressed and store value in
// variable ON
    if(ON){ // If ON is TRUE (1)
      TOGGLE(43); // TOGGLE function toggles the value
// of pin 43 (HIGH becomes LOW, LOW becomes HIGH)
    }
  }
}
```

WCacos

`float WCacos(float num);`

The **`WCacos`** function returns the arc cosine of the floating point number **`num`**. The value of **`num`** must be in the range -1 to 1; otherwise a domain error will occur. The returned value is the arccosine in radians, and is between 0 and **`PI`**.

WCasin

`float WCasin(float num);`

The **`WCasin`** function returns the arc sine of the floating point number **`num`**. The value of **`num`** must be in the range -1 to 1; otherwise a domain error will occur. The returned value is the arcsine in radians, and is between -**`HALFPI`** and **`HALFPI`**.

WCatan2

`float WCatan2(float y, float x);`

The **`WCatan2`** function returns the arc tangent of **`y/x`**. It uses the signs of its arguments to compute the quadrant of the return value.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float y; // Declare variable y of type float for use
         // in WCatan2 function
  float x; // Declare variable x of type float for use
         // in WCatan2 function
  float arctangent; // Declare variable arctangent of
// type float to store value returned by the function
// WCatan2
  float ans; // Variable ans used to store hand
           // calculated answer for example
  float ans_high; // Variable used to store the upper
                // range for the answer
  float ans_low;  // Variable used to store the lower
                  // range for the answer

//Example
  y = -5.5;
  x = 2.75;
  arctangent = WCatan2(y, x); // function WCatan2 is
// used to evaluate the arctangent of -5.5/2.75. The
// value is then stored in the variable arctangent.

  ans = -1.10714; // An approximation of the answer
// computed by hand is stored in the variable ans3.
// Float variables can only hold six digits of
// precision
// this means that some approximations must be made

/* Since our answer computed by hand is an
approximation we will compare the value returned by the
function WCatan2 as being greater than 99.9% of our
hand calculated value and less than 100.1%. */

  ans_high = ans * 1.001; // ans_high stores 100.1% of
// our estimate of the arctangent
  ans_low =  ans * 0.999; // ans_low stores 99.9% of
// our estimate of the arctangent

/* Note: we want the value returned to be less than
ans_low  because  we  know  the  arctangent  will  be
negative. Had it been positive we would have wanted the
answer to be greater than ans_low.
Similarly, arctangent should be greater than ans_high
since arctangent is negative */

  if(arctangent < ans_low){ // Value returned by
// WCatan2 less then 99.9% of believed
    if(arctangent > ans_high){ // Value returned by
// WCatan2 greater then 100.1% of believed
      STPIND(41, HIGH); // If both are true turn on LED
                        // connected to pin 41
    }
  }

  END; // CSTAMP enters inactive, low power mode
       // Stops program from looping
}
```

WCatof

`float WCatof(BYTE buffer[]);`

The **`WCatof`** function converts the null terminated ASCII character string buffer into the numeric floating point number it represents, and returns this value.

The user must ensure that **`buffer`** truly represents a floating point number, and that is terminated with a null character '\0'.

If the function encounters any problem, it will return 0.

WCcos

`float WCcos(float num);`

The **`WCcos`** function returns the cosine of the floating point number **`num`**. The value of **`num`** must be given in radians.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float ANGLE; // Variable NUMBER of type float to
// store the angle who's cosine will be determined
  float COSINE; // Variable COSINE of type float to
// store the value returned by the WCcos function
  float ANS; // Variable ANS of type float to store the
// hand calculated value of the cosine of ANGLE
  float ANS_HIGH; // Variable ANS_HIGH of type float to
// store the high limit for comparison
  float ANS_LOW; // Variable ANS_LOW of type float to
// store the low limit for comparison

//Example
  ANGLE = 27.384;
  ANS = -0.629152;
  COSINE = WCcos(ANGLE); // WCcos function returns the
// cosine of ANGLE which is stored in the variable
// COSINE
  ANS_HIGH = ANS * 1.01; // 101% of the value stored in
                         // ANS is stored in ANS_HIGH
  ANS_LOW = ANS * 0.99;  // 99% of the value stored in
```

```
                         // ANS is stored in ANS_LOW

/* Since COSINE is not restricted to positive numbers
only the outer-most if-else pair is to determine if
COSINE is positive or negative. If it is positive it
goes through two more if-else pairs which test to make
sure that the expression ANS_LOW < COSINE < ANS_HIGH is
true. If COSINE is negative however, the two following
if-else pairs are used to determine if the expression
ANS_HIGH < COSINE < ANS_LOW is true. This is because a
number which is negative and with a larger magnitude
than another is considered to be the smaller number. */

  if(COSINE == WCfabs(COSINE)){ // If statement to test
// if COSINE is positive or negative COSINE is positive
    if(COSINE > ANS_LOW){ // If statement to test if
// the positive COSINE is greater than ANS_LOW
// ANS_LOW < COSINE
      if(COSINE < ANS_HIGH){ // If statement to test if
// the postive COSINE is less than ANS_HIGH
// COSINE in range (ANS_LOW < COSINE < ANS_HIGH)
        STPIND(39, HIGH); // Light LED attached to
                          // pin 39
      }
      else{
// COSINE out of range (ANS_HIGH < COSINE)
        STPIND(43, HIGH); // Light LED attached to
                          // pin 43
      }
    }
    else{
// COSINE out of range (COSINE < ANS_LOW)
      STPIND(43, HIGH); // Light LED attached to pin 43
    }
  }
  else{
// COSINE is negative
    if(COSINE < ANS_LOW){ // If statement to test if
// the negative COSINE is less than ANS_LOW
// COSINE < ANS_LOW
      if(COSINE > ANS_HIGH){ // If statement to test if
// the negative COSINE is greather than ANS_HIGH
// COSINE in range (ANS_HIGH < COSINE < ANS_LOW)
        STPIND(39, HIGH); // Light LED attached to
                          // pin 39
      }
      else{
// COSINE out of range (COSINE < ANS_HIGH)
        STPIND(43, HIGH); // Light LED attached to
                          // pin 43
      }
    }
    else{
// COSINE out of range (COSINE > ANS_LOW)
      STPIND(43, HIGH); // Light LED attached to pin 43
    }
  }

  END(); // END function stops the program and puts the
         // CSTAMP into low power mode.
}
```

WCfabs

float WCfabs(float num);

The **WCfabs** function returns the absolute value of the of the floating point number **num**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  //Declaring two variables to be tested with the
  //absolute value function
  float evaluate;
  float compare;

  evaluate = -7.676;  // Set evaluate to a negative
  compare = 7.676;  // Set compare to the correct
                    // absolute value of evaluate

  while(1){ // Continuous loop
    evaluate = WCfabs(evaluate);  // Sets evaluate to
                                  // its absolute
    if(evaluate == compare)  // Test to verify that the
                             // function works
                             // properly
```

```
      STPIND(46, HIGH);  // If compare and the absolute
                         // value of evaluate are
                         // equal, then light LED on
                         // pin 46
    else
      STPIND(42, HIGH);  // If compare and the absolute
                         // value of evaluate are not
                         // equal, then light LED
                         // on pin 42
  }
}
```

WCftoa

BYTE WCftoa(float num, BYTE buffer[]);

The **WCftoa** function converts the floating point number **num** into its ASCII string equivalent, and places the result in the **buffer**. The result is terminated with a null character '\0' (**BYTE** value 0).

The user must ensure that **buffer** has sufficient space or capacity in number of characters/**BYTES** to hold the converted result and the terminating null character. What this means is that you have to have some idea of the dynamic range of your variable. Let us say you are dealing with a value from the analog to digital converter that you know varies from 0 to 5V. The precision of the **float** type is about 6 decimal places, so you have one character for the integer part, the decimal point, six decimal places, and the null. That is 9 characters, so you would make your buffer say 11 bytes to be safe. If you have negative values, you have to account for the minus sign, and so on. It is better to have a large buffer. The C Stamp has a large RAM. In any case, you always know the exact number of characters that the conversion from **float** took. This value is returned by **WCftoa**.

The return value is equal to the number of characters actually placed into the buffer array excluding the null terminator.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BYTE letter[5];  // Array to store the result of the
                   // function
  BYTE compare[5];  // This variable is used to compare
                    // to the result of the function
                    // call
  float convert;  // Variable used to set a
                  // predetermined value
  int i;
  BIT flag = 0;  // Flag to signal an incorrect match
                 // in the arrays

  convert = 7.28;  // Convert is set to 7.28

  WCftoa(convert, letter);  // Function is called which
                            // converts float 7.28 to
                            // ASCII
  compare[0] = 0x37;  // These hexadecimal values are
  compare[1] = 0x2E;  // the ASCII equivalent of the
  compare[2] = 0x32;  // values in the letter array.
  compare[3] = 0x38;  // This will show whether or not
  compare[4] = 0x00;  // the function converted the
                      // float properly

  for(i = 0; i<4; i++){  // Check to see if elements of
                         // the array are equal
    if(letter[i] == compare[i]) flag = 0;
//If equal, do not set flag
    else flag = flag + 1;  //Otherwise, set flag
  }

  if (flag == 0){  // If letter equals compare then
                   // light the
    STPIND(46, HIGH);  // LED attached to pin 46
    STPIND(42, LOW);
  }
  else{
    STPIND(42, HIGH);  // If letter does not equal
                       // compare then light
    STPIND(46, LOW);   // the LED attached to pin 42
  }
  END();
}
```

WCpow

float WCpow(float base, int exp);

The **WCpow** function returns **base** raised to the **exp** power ($\mathbf{base}^{\mathbf{exp}}$). Both **base** and **exp** have to be greater than 0.

WCsin

`float WCsin(float num);`

The **`WCsin`** function returns the sine of the floating point number **`num`**. The value of **`num`** must be given in radians.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float ANGLE; // Variable NUMBER of type float to
// store the angle which sine will be determined
  float SINE; // Variable SINE of type float to store
// the value returned by the WCsin function
  float ANS; // Variable ANS of type float to store the
// hand calculated value of the sine of ANGLE
  float ANS_HIGH; // Variable ANS_HIGH of type float to
// store the high limit for comparison
  float ANS_LOW; // Variable ANS_LOW of type float to
// store the low limit for comparison

//Example
  ANGLE = 27.384;
  ANS = 0.777281;
  SINE = WCsin(ANGLE); // WCsin function returns the
// sine of ANGLE which is stored in the variable SINE
  ANS_HIGH = ANS * 1.0001; // 100.01% of the value
// stored in ANS is stored in ANS_HIGH
  ANS_LOW = ANS * 0.9999; // 99.99% of the value stored
// in ANS is stored in ANS_LOW

/* Since SINE is not restricted to positive numbers
only the outer-most if-else pair is to determine if
SINE is positive or negative. If it is positive it goes
through two more if-else pairs which test to make sure
that the expression ANS_LOW < SINE < ANS_HIGH is true.
If SINE is negative however, the two following if-else
pairs are used to determine if the expression
ANS_HIGH < SINE < ANS_LOW is true. This is because a
number which is negative and with a larger magnitude
than another is considered to be the smaller number. */
```

```
  if(SINE == WCfabs(SINE)){ // If statement to test if
// SINE is positive or negative
// SINE is positive
    if(SINE > ANS_LOW){ // If statement to test if the
// positive SINE is greater than ANS_LOW
// ANS_LOW < SINE
      if(SINE < ANS_HIGH){ // If statement to test if
// the postive SINE is less than ANS_HIGH
// SINE in range (ANS_LOW < SINE < ANS_HIGH)
        STPIND(39, HIGH); // Light LED attached to
                          // pin 39
      }
      else{
// SINE out of range (ANS_HIGH < SINE)
        STPIND(43, HIGH); // Light LED attached to
                          // pin 43
      }
    }
    else{
// SINE out of range (SINE < ANS_LOW)
      STPIND(43, HIGH); // Light LED attached to pin 43
    }
  }
  else{
// SINE is negative
    if(SINE < ANS_LOW){ // If statement to test if the
// negative SINE is less than ANS_LOW
// SINE < ANS_LOW
      if(SINE > ANS_HIGH){ // If statement to test if
// the negative SINE is greather than ANS_HIGH
// SINE in range (ANS_HIGH < SINE < ANS_LOW)
        STPIND(39, HIGH); // Light LED attached to
                          // pin 39
      }
      else{
// SINE out of range (SINE < ANS_HIGH)
        STPIND(43, HIGH); // Light LED attached to
                          // pin 43
      }
    }
    else{
// SINE out of range (SINE > ANS_LOW)
      STPIND(43, HIGH); // Light LED attached to pin 43
    }
  }
  END(); // END function stops the program and puts the
         // CSTAMP into low power mode.
}
```

WCsqrt

float WCsqrt(float num);

The **WCsqrt** function returns the square root of the floating point number **num**. The value of **num** must not be negative.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  float NUMBER; // Variable NUMBER of type float, used
// to store the number to be square rooted
  float SQRT; // Variable SQRT of type float, used to
// store the value returned by the WCsqrt function
  float ANS; // Variable ANS of type float,
             // Pre-calculated answer
  float ANS_HIGH; // Variable ANS_HIGH of type float,
                  // high limit of range
  float ANS_LOW;  // Variable ANS_LOW of type float,
                  // low limit of range

  NUMBER = 3 * PI / 4; // Set NUMBER to 3 * PI / 4
                       // where PI is defined as 3.1416
  ANS = 1.53499;
  SQRT = WCsqrt(NUMBER);
  ANS_HIGH = ANS * 1.000001; // ANS_HIGH is set to
                             // 100.0001% of ANS
  ANS_LOW = ANS * 0.999999;  // ANS_LOW is set to
                             // 99.9999% of ANS
  if(SQRT < ANS_HIGH){ // Check if SQRT is less than
                       // ANS_HIGH
    if(SQRT > ANS_LOW){ // Check if SQRT is greater
                        // than ANS_LOW
      STPIND(40, HIGH); // Turn on LED connected to
                        //pin 40
    }
    else{
      STPIND(44, HIGH); // Turn on LED connected to
                        // pin 44
    }
  }
  else{
    STPIND(44, HIGH); // Turn on LED connected to
                      // pin 44
  }
  END(); // Stops the program from looping, CSTAMP
         // enters low power mode
}
```

WCtan

`float WCtan(float num);`

The **`WCtan`** function returns the tangent of **`num`**. The value of **`num`** must be given in radians.

WCtanh

`float WCtanh(float num);`

The **`WCtanh`** function returns the hyperbolic tangent of **`num`**. The value of **`num`** must be given in radians.

WDT

`BIT WDT(BIT action);`

The **`WDT`** function handles the Watchdog Timer present in the C Stamp. The concept is simple. There is a timer in the C Stamp that runs independent of the CPU, and it times out at 896 mS. Sometime before the timer runs out, you have to clear it in your program. If you do not, this would be an indication that the program has gone errant, and the watchdog will reset the CPU. However, if the CPU is in sleep mode, the CPU will be woken up instead of reset if the watchdog times out.

If **`action`** is **`ZERO`**, the watchdog will be disabled, which is the default condition upon program start.

If **`action`** is **`ONE`**, the watchdog will be enabled and cleared or just cleared if it is already enabled.

In any case, **WDT** always returns an indication of whether a watchdog timeout occurred before **WDT** was invoked. A return value of **ZERO** means that a timeout did not occur, and a **ONE** means that a timeout occurred.

WRITE

BIT WRITE(WORD location, BYTE value);

The **WRITE** function writes **value** into **location** in data EEPROM.

The user must ensure that the **location** value is within the bounds of the existing data EEPROM address space (0 – 1023). If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BIT SUCCESSFUL; // Declare variable SUCCESSFUL of
                  // type bit

  SUCCESSFUL = WRITE(0000, 0x64); // Call function
// WRITE to store the hexadecimal value 64 in the
// EEPROM at location '0000'. WRITE returns TRUE if it
// is successful and FALSE if it fails, this is then
// stored in the variable SUCCESSFUL
  if(SUCCESSFUL){ // Check to see if the WRITE was
                  // successful
    STPIND(42, HIGH); // Light LED attached to pin 42
  }

  END(); // End program execution, C Stamp enters low
         // power mode
}
```

Wtoa

void Wtoa(WORD W, BYTE a[], BYTE radix);

The **Wtoa** function converts the **WORD W** into its string equivalent with leading zeros and capital letters, and places the result in the **BYTE** array **a**. The base of the output string is determined by **radix**, which may be 2, 10, or 16.

Be sure to call **`Wtoa`** with an array **`a`** of sufficient length to hold the converted result.

EXAMPLE PROGRAM:

```
#include  "CS110000.h"

void main(void)
{
  BYTE letter[];  // Array to store the result of the
                  // function
  BYTE compare = 36;  // This variable is used to
                      // compare to the result of the
                      // function call
  WORD convert;  // Variable used to set a
                 // predetermined value

  convert = 65; //Convert is set to 65

  Wtoa(convert, letter, 10);  // Function is called
                              // which converts decimal
                              // 65 to ASCII with the
                              // base of 10

  while (1){  //Continuous loop
    if (letter[0] == compare)  // If letter[0] equals
                               // 36 then light the
      STPIND(46, HIGH);  // LED attached to pin 46
    else
      STPIND(42, HIGH);  // If letter does not equal
                         // 36 then light the LED
                         // attached to pin 42
  }
}
```

Chapter

6

Accessory Specific Functions and Commands Reference

This chapter describes the functions and commands that are specific to the software support of different types of accessories that are available from A-WIT Technologies to complement the function and projects developed with the C Stamp. The user should consult the manual for a specific accessory for full information on connectivity and usage.

LCDCMD_CS410000

```
BIT LCDCMD_CS410000(BYTE command, BYTE B0, BYTE B1,
                    BYTE B2, BYTE B3, BYTE B4,
                    BYTE B5, BYTE B6, BYTE B7);
```

The **LCDCMD** function sends a command to the CS410000 LCD display according to the table below. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**. This could mean that there was an error in the arguments of the function or some other problem.

command is a variable/constant/expression (1 – 17) indicating the LCD command to send.

B0 – B7 are bytes variables/constants/expressions that are used by different commands. Not all bytes are used by all commands. If a byte is not used by a command, any value can be passed to the function (e.g. **ZERO**), and it will be ignored. However, something must be passed in all the arguments. If the usage of a byte is not specified in the table below, it means that byte is not used and is ignored by the function.

Symbol	*Value*	*Command*	*Description*

Symbol	*Value*	*Command*	*Description*
LCD_HOME	1	Cursor Home	Sets the cursor to the home position (top left).
LCD_SETC	2	Set Cursor (1 – 80)	Set cursor to a position specified by **B0**, where 1 is the top left and 80 is the bottom right.
LCD_SETCLC	3	Set Cursor (line, column)	Sets cursor using two bytes, where **B0** is the line and **B1** is the column.
LCD_HIDEC	4	Hide Cursor	Stops the position cursor from appearing on the display.
LCD_UNDLC	5	Show Underline Cursor	Changes the cursor to the underline type.
LCD_BLNKC	6	Show Blinking Cursor	Changes the cursor to the blinking type.
LCD_BACKS	7	Backspace	Deletes the preceding character from the current position on the display.
LCD_HTAB	8	Horizontal Tab (by Tab Set)	Moves the current position across by the tab space set by command **LCD_SETTA** (default tab space is 4).
LCD_SLF	9	Smart Line Feed	Moves the cursor down one line to the position beneath it in the same column.
LCD_VTAB	10	Vertical Tab	Moves the cursor up one line to the position above it in the same column.
LCD_CLRS	11	Clear Screen	Clears the screen and sets the cursor to the home position.
LCD_CR	12	Carriage Return	Moves the cursor to the start of the next line.
LCD_CLRC	13	Clear Column	Clears the contents of the current column and moves the cursor right by one column.

Symbol	*Value*	*Command*	*Description*
`LCD_SETTA`	14	Tab Set	Sets the tab size to **`B0`**. **`B0`** can be a size between 1 and 10.
`LCD_BLON`	15	Backlight On	Turns the backlight of the LCD on.
`LCD_BLOFF`	16	Backlight Off (default)	Turns the backlight of the LCD off.
`LCD_CCHAR`	17	Custom Character	Sends one of 8 possible custom characters to the LCD to be used later. **`B0 – B7`** define the custom character. See below.
`LCD_DCCHAR`	18	Display Custom Character	Displays a custom character at the current cursor position. **`B0`** has a value 0 – 7 that specifies which one of the custom characters is to be displayed.

Up to 8 custom characters can be stored in the LCD for subsequent usage by sending an 8 byte map in bytes **`B0 – B7`** with the LCD command **`LCD_CCHAR`**. The example below shows how to define a byte map for a custom character.

Byte	*Char #*	*Bits of* **`B0 – B7`**								*Byte in Binary*	*Byte in Hex*
		Char # in Binary			*Char Map*						
B0	2	0	1	0	0	0	0	0	0	0100 0000	0x40
B1	2	0	1	0	0	0	1	0	0	0100 0100	0x44
B2	2	0	1	0	0	1	1	1	0	0100 1110	0x4E
B3	2	0	1	0	1	0	1	0	1	0101 0101	0x55
B4	2	0	1	0	0	0	1	0	0	0100 0100	0x44
B5	2	0	1	0	0	0	1	0	0	0100 0100	0x44
B6	2	0	1	0	0	0	1	0	0	0100 0100	0x44

Byte	*Char #*	*Bits of* **B0 – B7**								*Byte in Binary*	*Byte in Hex*
		Char # in Binary			*Char Map*						
B7	2	0	1	0	0	0	0	0	0	0100 0000	0x40

The bytes **B0 – B7** defined above would store an up-arrow as custom character # 2. To store this character in the LCD, we would call **LCDCMD** as:

```
result = LCDCMD_CS410000(LCD_CCHAR, 0x40, 0x44, 0x4E,
                         0x55, 0x44, 0x44, 0x44,
                         0x40);
```

Checking the result of calling the function is optional, although it is a good programming practice. Also, decimal values could be sent for **B0 – B7**, instead of hexadecimal; so an alternative way of calling **LCDCMD** is:

```
LCDCMD_CS410000(LCD_CCHAR, 64, 68, 78, 85, 68, 68, 68,
                64);
```

Then to display this character at the current cursor position, we would call:

```
result = LCDCMD_CS410000(LCD_DCCHAR, 2, 0, 0, 0, 0, 0,
                          0, 0);
```

or

```
LCDCMD_CS410000(LCD_DCCHAR, 2, 0, 0, 0, 0, 0, 0, 0);
```

LCDIN_CS410000

```
NIBBLE LCDIN_CS410000(void);
```

The **LCDIN** function receives data from the CS410000 LCD display. This command supports the feature of the CS410000 LCD to host a standard 3 x 4 Matrix keypad, like the CS410001. **LCDIN** returns the number corresponding to the key that is being pressed in the keypad (0 – 9, 10 for the * key, and 11 for the # key). If no key is being pressed or there is an error, the function returns 12.

LCDOUT_CS410000

```
BIT LCDOUT_CS410000(BYTE string[], BYTE n);
```

The **LCDOUT** sends ASCII bytes to the CS410000 LCD for displaying. Displaying of these characters will start at the current cursor position. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**.

string is an array of the ASCII bytes to be displayed.

n is a variable/constant/expression that specifies how many bytes are in **string**.

VFDCMD_CS410100

```
BIT VFDCMD_CS410100(BYTE command, BYTE Data_Pin,
                    BYTE Reset_Pin,
                    BYTE Argument_Array[],
                    BYTE lArgument_Array);
```

The **VFDCMD** function sends a command to the CS410100 VFD display according to the table below. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**. This could mean that there was an error in the arguments of the function or some other problem.

command is a variable/constant/expression (1 – 14) indicating the VFD command to send.

Data_Pin is a variable/constant/expression indicating which CStamp pin, the VFD data wire is connected to. **Data_Pin** must be a valid I/O pin on the CStamp as listed in the CStamp manual. This excludes pins 5-7, 24-26, and 48.

Reset_Pin is a variable/constant/expression indicating which CStamp pin, the VFD reset wire is connected to. **Reset_Pin** must be a valid I/O pin on the C Stamp as listed in the C Stamp manual. This excludes pins 5 - 7, 24 - 26, 48, and **Data_Pin**.

Argument_Array is an array of type BYTE which contains the arguments specific to the command being sent to the VFD module. These arguments are specified by the command list in the table below. The number of elements in **Argument_Array** is also specific to the command being sent, refer to the table for the length used for a specific command.

lArgument_Array is a variable/constant/expression which gives the length of the array **Argument_Array**. The number of elements necessary is based upon which command is being issued, refer to the table below for details on the length of the array for each command. Although array indices begin at 0 in WC, **lArgument_Array** will be the number of elements in the array, i.e. if the **Argument_Array** as defined as:

```
BYTE TEST_ARRAY[10];
```

`lArgument_Array` should be assigned a value of 10. Commands which have an argument size of 0, as specified in the table below, do not require a true **`Argument_Array`** or **`lArgument_Array`**, for these commands **`FALSE`** may be used to replace both in the command, or any variable/constant/expression may be entered, because the value of both are not used in the function.

Symbol	*Value*	*Command*	*Description*
VFD_RESET	0	Reset	Resets the module to the default state. This function will interrupt any operation currently being performed by the VFD module. Argument Size: 0
VFD_HOME	1	Cursor Home	Sets the cursor to the home position (top left). Argument Size: 0
VFD_SETC	2	Set Cursor (1-32)	Set the cursor to a position specified by the value of the argument array at index 0, where 1 is the top left and 32 is the bottom right. Argument Size: 1
VFD_SETCLC	3	Set Cursor (line, column)	Sets cursor using two positions in the argument array, where index 0 is the line and index 1 is the column. Argument Size: 2
VFD_HIDEC	4	Hide Cursor	Stops the position cursor from appearing on the display. Argument Size: 0
VFD_SHOWC	5	Show Cursor	Changes the cursor to the underline type. Argument Size: 0
VFD_BACKS	6	Backspace	Deletes the preceding character from the current position on the display. Argument Size: 0
VFD_HTAB	7	Horizontal Tab	Moves the current position across by the space of one character. Argument Size: 0
VFD_SLF	8	Smart Line Feed	Moves the cursor down one line to the position beneath it in the same column. Argument Size: 0

Symbol	*Value*	*Command*	*Description*
VFD_VTAB	9	Vertical Tab	Moves the cursor up one line to the position above it in the same column. Argument Size: 0
VFD_CR	10	Carriage Return	Moves the cursor to the start of the next line. Argument Size: 0
VFD_CCHAR	11	Custom Character	Sends one of 16 possible custom characters to the VFD to be used later. **Argument_Array** defines the custom character. See table on custom characters below. Argument Size: User defined.
VFD_DCCHAR	12	Display Custom Character	Displays a custom character at the current cursor position. **Argument_Array** has a value 0 – 16, at index 0, which specifies the custom character to be displayed. Argument Size: 1
VFD_IMAGE	13	Display bit image	Displays a user defined image immediately once the command has been issued. Index 0 of **Argument_Array** gives the height of the image, in units of 8 pixels, and index 1 gives the width of the image, in units of 1 pixel. All subsequent indices contain the bit image data. Argument Size: User defined
VFD_MOVIE	14	Display bit image animation	Uses multiple bit images, or frames, to create an animated image. Index 0 of **Argument_Array** gives the height of the images, in units of 8 pixels, and index 1 gives the width of the images, in units of a single pixel. All frames must have the same dimensions. Index 2 gives the number of iterations the animation is to be played. Valid ranges are from 0-255, where 0 is an infinite loop. Index 3 specifies the speed of the animation 1-255 milliseconds delay inserted between

Symbol	*Value*	*Command*	*Description*
			each frame. The frames are placed in sequential order starting with index 4 holding the first byte of the first frame. See bitmap section below for details on creating the data corresponding to a single image. Argument Size: User defined.

Up to 16 custom characters can be stored in the VFD for subsequent use with the VFD command **`VFD_CCHAR`**. The command uses the first two elements of **`Argument_Array`** to select the character and its format; subsequent elements are used for the data which defines the character. The first table below shows how to define the custom character options, and the second table shows how to map the data for a character.

Custom Character Options	
Index of Argument_Array	**Option**
0	Select Character (1-16)
1	Select format; 5x7 (value of 0) or 7x8 (value of 1)

Custom Character Map					
	Indices 2 - 6 of `Argument_Array`				
	Char Map				
Bits	**2**	**3**	**4**	**5**	**6**
`Bit 7`	0	0	0	0	0
`Bit 6`	0	0	1	0	0
`Bit 5`	0	1	1	1	0

Custom Character Map					
	Indices 2 - 6 of **`Argument_Array`**				
	Char Map				
Bit 4	1	0	1	0	1
Bit 3	0	0	1	0	0
Bit 2	0	0	1	0	0
Bit 1	0	0	1	0	0
Bit 0	0	0	0	0	0

In this example the indices would have the following values given in binary, and in hexadecimal:

Index 2: binary 00010000 or hexadecimal 10
Index 3: binary 00100000 or hexadecimal 20
Index 4: binary 01111110 or hexadecimal 7E
Index 5: binary 00100000 or hexadecimal 20
Index 6: binary 00010000 or hexadecimal 10

These values are obtained by reading the bits down the index from bit 7 (MSB) to bit 0 (LSB).

In the 5x7 format, bit 0 is not eligible and should be kept at a value of 0 for each index. If the 7x8 format was selected bit 0 would be eligible for use, and the character map would occupy indices 2 - 8.

The resulting **`Argument_Array`** for this example might look like the following:

```
BYTE Example_Array[7] = {1, 0, 0x10, 0x20, 0x7E, 0x20,
     0x10};
```

The command would then look similar to the code fragment below:

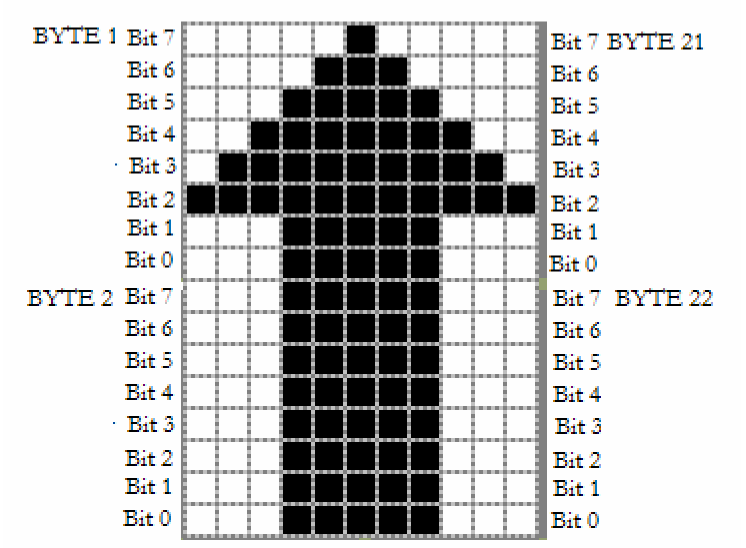
```
result = VFDCMD_CS410100(VFD_CCHAR, 18, 17,
                          Example_Array, 7);
```

Checking the result of calling the function is optional, although it is a good programming practice.

Then to display this character at the current cursor position, we would call the function as seen below:

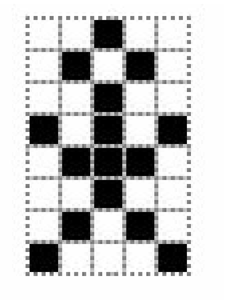
```
result = VFDCMD_CS410100(VFD_DCCHAR, 18, 17, 1, 1);
```

Bit image data is defined from top to bottom and left to right, where the most significant bit of the first byte of bit image data controls the top-left pixel, and the least significant bit of the last byte of bit image data controls the bottom-right pixel. The graphic below shows how to create a bitmap for an image. This image uses both rows of the display and 11 columns. The black spaces represent a '1' while the white spaces represent a '0'.

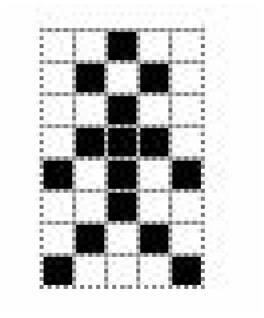


The bytes 3-20 run from left to right following the same pattern in which all odd numbered bytes control the top row and all even numbered bytes control the bottom row. This bitmap gives the resulting data which should be incorporated into **`Argument_Array`** as specified in the command table.

Animation on the VFD is accomplished through the use of multiple user defined bit images. The bitmap corresponding to each frame is listed in succession and looped through based on the arguments set by the user. Below is the first bitmap of a two frame animation.



As described previously, the bitmap is converted to the appropriate bytes in order from top to bottom, and left to right. In this example the five bytes for this image, given in hexadecimal format are 0x11, 0x4A, 0xBC, 0x4A and 0x11.



The second image is converted from bitmap to bytes in the same fashion. This bitmap corresponds to the following bytes in hexadecimal format, 0x09, 0x52, 0xBC, 0x52, and 0x09. To create the animation in which these two images are repeated indefinitely and with a delay of approximately 50ms between each frame, we define the following array, and call the VFDCMD function as seen below. Once the function is called the animation is displayed immediately on the VFD.

```
MOVIE_ARG[14] = {2, 5, 0, 50, 0x11, 0x4A, 0xBC, 0x4A,
0x11, 0x09, 0x52, 0xBC, 0x52, 0x09};
```

```
VFDCMD_CS410100(14, Data_Pin, Reset_Pin, MOVIE_ARG,
                14);
```

VFDOUT_CS410100

```
BIT VFDOUT_CS410100(BYTE Data_Pin, BYTE ASCII_Array[],
                    BYTE lASCII_Array);
```

The **VFDOUT** sends ASCII bytes to the CS410100 VFD for displaying. Displaying of these characters will start at the current cursor position. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**.

Data_Pin is a variable/constant/expression indicating which CStamp pin, the VFD data wire is connected to.

ASCII_Array is an array of the ASCII bytes to be displayed.

lASCII_Array is a variable/constant/expression that specifies how many bytes are in **ASCII_Array**.

COMPASSIN_CS420000

```
NIBBLE COMPASSIN_CS420000(BYTE Npin, BYTE Epin,
                           BYTE Spin, BYTE Wpin);
```

The **COMPASSIN** function returns a direction by querying a CS420000 Digital Compass Sensor that is connected to the C Stamp.

Npin is a variable/constant/expression that specifies the I/O pin that is connected to the North Signal of the compass. This pin will set to input mode.

Epin is a variable/constant/expression that specifies the I/O pin that is connected to the East Signal of the compass. This pin will set to input mode.

Spin is a variable/constant/expression that specifies the I/O pin that is connected to the South Signal of the compass. This pin will set to input mode.

Wpin is a variable/constant/expression that specifies the I/O pin that is connected to the West Signal of the compass. This pin will set to input mode.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
COMPASSIN_ERROR	0	There was at least one error while executing the function.
COMPASSIN_N	1	Direction is North.
COMPASSIN_NE	2	Direction is Northeast.
COMPASSIN_E	3	Direction is East.

return code	*value*	*Meaning*
COMPASSIN_SE	4	Direction is Southeast.
COMPASSIN_S	5	Direction is South.
COMPASSIN_SW	6	Direction is Southwest.
COMPASSIN_W	7	Direction is West.
COMPASSIN_NW	8	Direction is Northwest.

EXREAD_CS450000

`BYTE EXREAD_CS450000(WORD location, BYTE CSn);`

The **EXREAD** function reads the value at **location** in the CS450000 EEPROM and returns the result.

location is a variable/constant/expression that specifies the CS450000 EEPROM address to read from.

CSn is a variable/constant/expression that specifies the pin on the C Stamp that pin 1 (chip select) on the CS450000 EEPROM is connected to.

The user must ensure that the **location** value is within the bounds of the CS450000 EEPROM address space and that the chip is selected.

EXWRITE_CS450000

```
BIT EXWRITE_CS450000(WORD location, BYTE value,
                     BYTE CSn);
```

The **EXWRITE** function writes **value** into **location** in the CS450000 EEPROM using **CSn** as the C Stamp pin connected to the CSn pin of the EEPROM.

The user must ensure that the **location** value is within the bounds of the CS450000 EEPROM address space, and that the chip is selected and not write protected. If successful, the function returns **TRUE**; otherwise it returns **FALSE**.

TEMPSIN_CS452000

`float TEMPSIN_CS452000(NIBBLE A, NIBBLE scale);`

The **TEMPSIN** function returns the temperature read by the Temp Sensor. If the function is not successful, it returns the value of 500, which is outside the range of measurable temperature by the sensor in any of the available formats. This could mean that there was an error in the arguments of the function or some other problem.

A is a variable/constant/expression (0 – 7) indicating the binary address assigned to the Temp Sensor by connecting its $A_2 – A_0$ address inputs to appropriate logic levels.

scale is a variable/constant/expression that indicates the scale for the returned temperature value according to the table below.

Symbol	*Value*	*Temperature Scale*
TEMPS_C	1	Degrees Celsius
TEMPS_F	2	Degrees Fahrenheit
TEMPS_K	3	Degrees Kelvin

TEMPSST_CS452000

```
BIT TEMPSST_CS452000(NIBBLE A, float TH, float TL,
                      NIBBLE scale, BIT active_level,
                      BIT LP);
```

The **TEMPSST** sets up the thermostat alarm function of the Temp Sensor, or to set it to inactive low power mode. If the function is successful, it returns **TRUE**; otherwise, it returns **FALSE**.

A is a variable/constant/expression (0 – 7) indicating the binary address assigned to the Temp Sensor by connecting its $A_2 – A_0$ address inputs to appropriate logic levels.

TH is a variable/constant/expression that specifies the High Temperature value of the thermostat function.

TL is a variable/constant/expression that specifies the Low Temperature value of the thermostat function.

scale is a variable/constant/expression that indicates the temperature scale for both **TH** and **TL** according to the table below.

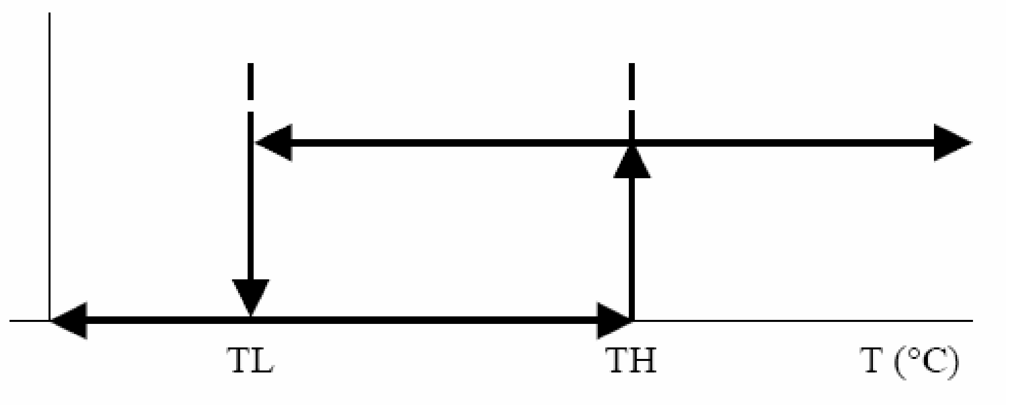
Symbol	*Value*	*Temperature Scale*

Symbol	*Value*	*Temperature Scale*
TEMPS_C	1	Degrees Celsius
TEMPS_F	2	Degrees Fahrenheit
TEMPS_K	3	Degrees Kelvin

active_level is a variable/constant/expression that specifies the active state for the output T_{OUT}. An active state may either be a logic "1" (VDD) or a logic "0" (0V). Typically, **active_level** is set **HIGH** for activating a cooling device at **TH** degrees and **LOW** for activating a heating device at **TL** degrees.

LP is a variable/constant/expression that sets the Temp Sensor in inactive low power mode. If **LP** is **TRUE**, all other arguments are ignored, and the Temp Sensor is set to inactive low power mode. If **LP** is **FALSE**, then the thermostat function is set using the rest of the arguments. The function always needs to receive a value for each argument, so if it is being used to set the Temp Sensor in inactive low power mode, **ZERO**'s or any other values should be passed for the non **LP** arguments.

Even though **TH** and **TL** can be specified in any of the three available temperature scales, these values are rounded internally to the nearest ½ °C. When the Temp Sensor's temperature meets or exceeds the **TH** value, the T_{OUT} output pin of the Temp Sensor becomes active and will stay active until the temperature falls below the temperature **TL**. In this way, any amount of hysteresis may be obtained. This is shown in the figure below for the case where both **TH** and **TL** are specified in °C's, and the active level of T_{OUT} is specified as **HIGH**.



DPOTSET_CS458000

`void DPOTSET_CS458000(float value, BYTE CSn);`

The **`DPOTSET`** function sets the resistance of a CS458000 Digital POT from terminal W to terminal B to **`value`**, or to the closest one supported by the device. The resistance from the W terminal to A, will then be: (total resistance of the POT – **`value`**).

`value` is a variable/constant/expression that specifies the resistance value.

If **`value`** is a negative number, the Digital POT will be put into a power-saving mode. In this mode, the A terminal is open-circuited and the B and W terminals are shorted together. Once the potentiometer has entered the shutdown mode, it will remain in this mode until a new valid **`value`** is written to the potentiometer.

`CSn` is a variable/constant/expression that specifies the C Stamp I/O pin connected to the CSn pin of the POT (Pin 1).

DPOTSET_CS458001

`void DPOTSET_CS458001(float value, BYTE CSn);`

The **`DPOTSET`** function sets the resistance of a CS458001 Digital POT from terminal W to terminal B to **`value`**, or to the closest one supported by the device. The resistance from the W terminal to A, will then be: (total resistance of the POT – **`value`**).

`value` is a variable/constant/expression that specifies the resistance value.

If **`value`** is a negative number, the Digital POT will be put into a power-saving mode. In this mode, the A terminal is open-circuited and the B and W terminals are shorted together. Once the potentiometer has entered the shutdown mode, it will remain in this mode until a new valid **`value`** is written to the potentiometer.

`CSn` is a variable/constant/expression that specifies the C Stamp I/O pin connected to the CSn pin of the POT (Pin 1).

RADIOCMD_CS470000

```
BYTE RADIOCMD_CS470000(NIBBLE command,
                       WORD Wparameter, BYTE RXpin,
                       BYTE TXpin);
```

The **RADIOCMD** function sends a command to the CS470000 RF Module according to the table below. If the function is successful, it returns a non zero number depending on the command; otherwise, it returns zero. This could mean that there was an error in the arguments of the function or some other problem.

command is a variable/constant/expression (1 – 3) indicating the RADIO command to send.

Wparameter is a word variable/constant/expression that is used by different commands. This parameter is not used by all the commands. If the parameter is not used by a command, any value can be passed to the function (e.g. **ZERO**), and it will be ignored. However; something always must be passed in all the arguments. If the usage of the parameter is not specified in the table below, then it is not used and ignored by the function.

RXpin is a variable/constant/expression that specifies the I/O pin to be used as a receiver. This pin will be set to input mode.

TXpin is a variable/constant/expression that specifies the I/O pin to be used as a transmitter. This pin will be set to output mode.

command *Symbol*	*Value*	*Command*	*Description*
RF_DB	1	Receive Signal Strength	Returns the signal strength (in decibels) of the last received packet. Range: 37 - 106
RF_DT	2	Destination Address	Sets the address that identifies the module's address and the destination of the RF packet. The address is specified by **Wparameter**, and it can be any value from 0x0 to 0xFFFF in hexadecimal, or from 0 to 65535 in decimal. The factory default for the RF Module address is 0x0 in hex. Only radio modems having matching addresses can communicate with each other. If successful, this command returns 200.
RF_MK	3	Address Mask	Sets the address mask to configure local and global address space. The address mask is specified by **Wparameter**,

command *Symbol*	*Value*	*Command*	*Description*
			and it can be any value from 0x0 to 0xFFFF in hexadecimal, or from 0 to 65535 in decimal. The factory default for the RF Module address mask is 0xFFFF in hex. If successful, this command returns 200.

Destination Addresses and Masks provide the means to set up global or local addresses for establishing module groups, subnets, etc. The Destination Address network layer provides for more granular isolation of radio modems. The Destination Addresses and Masks can be used to:

- Set up point-to-point and point-to-multipoint network configurations
- Provide greater flexibility in establishing module groups, subnets, etc.

Each radio modem in a network can be configured with a 16-bit Destination Address to establish selective communications within a network. This address is set to one of 65535 values using the **RF_DT** (Destination Address) Command. The default Destination Address is 0.

All radio modems with the same Destination Address can transmit and receive data among themselves. Radio modems having different Destination Addresses still detect and listen to the data (in order to maintain network synchronization); however, the data is discarded data rather than passing on through the DO pin.

CS470000 Radio Modems are packet based. This means that all data shifted into one module is packetized and sent out the antenna port. Because these RF modules use a peer-to-peer architecture, all modules will receive the packet and decide whether to pass it to the host or to throw it away. Each transmitted packet contains information about the transmitting module.

Any module that receives a packet will check the address values and decide what to do with the packet. The options are as follows:

- Receive the packet as a global packet
- Receive the packet as a local packet
- Discard the packet

The mask parameter can be used to allow a base module to receive data from a range of addresses. It may also be used to configure "subnets" of modules that communicate in a group together.

The RF Module uses the bit-wise "AND" operation to qualify the Destination Addresses and Address Masks. This operation is performed bit by bit on each of the 16 bits in the Destination Address and Address Mask parameters. The table below shows the Bit–Wise AND Truth Table.

Bit-Wise AND Operation

Operand 1	**AND Operand 2**	**= Result**
0	0	0
0	1	0
1	0	0
1	1	1

The Address Mask can be used as an additional method of facilitating communications between modules. The Address Mask can be set to one of 65535 possible values using **RF_MK** (Address Mask) Command. The default value of the **RF_MK** Parameter is 0xFFFF.

All transmitted data packets contain the Destination Address of the transmitting module. When a transmitted packet is received by a module, the Destination Address of the transmitter (contained in the packet) is logically "ANDed" (bitwise) with the Address Mask of the Receiver. If the resulting value matches the Destination Address of the Receiver, or if it matches the Receiver Address Mask, the packet is accepted. Otherwise, the packet is discarded.

Note: When performing this comparison, any "0" values in the Receiver Address Mask are treated as irrelevant and are ignored.

For example, suppose that two RF modules are setup as follows. RF Module 1 has a Destination Address of 0x1 and an Address Mask of 0xFFFF, and RF Module 2 has a Destination Address of 0x1 and an Address Mask of 0x000F. In this case RF Module 1 will always transmit to RF module 2, but it will receive from any module with any Destination Address. RF Module 2; however, will also transmit to RF Module 1, but it will only receive from modules with Destination Addresses from 0 through 15 (0x0 through 0xF in hex). This range, of course, includes RF Module 1.

RADIOIN_CS470000

```
NIBBLE RADIOIN_CS470000(WORD timeout, BYTE buffer[],
                        BYTE N, BYTE RXpin);
```

The **RADIOIN** function receives asynchronous serial data from the CS470000 RF Module via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25), and the ATN pin (Pin 26).

timeout is a variable/constant/expression (0 – 65535) that tells **RADIOIN** how long to wait in mS for incoming data. If data does not arrive in time or if the buffer gets full before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait until it receives enough data to fill the buffer before it returns. The usage of timeouts applies to each data byte being received.

buffer is the name of the array of **BYTES** that the function will use as storage to return the received data to the calling function. This can be an array of length equals to 1 byte, and the maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) to be placed in **buffer**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **N**-1).

If you do not use the timeout feature, the commands wait forever to get the number of bytes that you requested. However, the number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

RXpin is a variable/constant/expression that specifies the I/O pin to be used as a receiver. This pin will be set to input mode.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
RFIN_ARGERR	0	There was at least one error in the arguments of the function.

return code	*value*	*Meaning*
		Function did not execute.
RFIN_BUFULL	1	Function returned with the buffer full of data.
RFIN_TIMOUT	2	The function exited after a timeout condition occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the timeout occurred. If this location contains the value 0xFF, this indicates that no data was received.
RFIN_OVRERR	3	An overrun error occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, this indicates that no data was received.

RADIOOUT_CS470000

```
BIT RADIOOUT_CS470000(BYTE buffer[], BYTE N,
                      BYTE TXpin);
```

The **RADIOOUT** function transmits asynchronous serial data to the CS470000 RF Module via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25), and the ATN pin (Pin 26).

buffer is the name of the array of **BYTES** that the function will send. This can be an array of length equals to 1 byte, and the maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) in **buffer** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **N**-1).

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

TXpin is a variable/constant/expression that specifies the I/O pin to be used as a transmitter. This pin will be set to output mode.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
RFOUT_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
RFOUT_BUFEMP	1	Function returned after transmitting all data in the buffer.

BLUECMD_CS47100X

```
BIT BLUECMD_CS47100X(BYTE command, WORD Wparameters[],
                     BYTE RXpin, BYTE TXpin,
                     WORD Wresponses[],
                     BYTE Sparameter[]);
```

The **BLUECMD** function sends a command to the CS47100X Bluetooth Module according to the tables below. If the function is successful, it returns **TRUE**; otherwise, it returns **ZERO**. This could mean that there was an error in the arguments of the function or some other problem.

command is a variable/constant/expression (1 – 17) indicating the BLUETOOTH command to send.

Wparameters is a 4 WORD array that is used by different commands. This parameter is not used by all the commands. If the parameter is not used by a command, any value can be passed to the function (e.g. **ZERO**), and it will be ignored. However; something always must be passed in all the arguments. If the usage of the parameter is not specified in the tables below, then it is not used and ignored by the function.

RXpin is a variable/constant/expression that specifies the I/O pin to be used as a receiver. This pin will be set to input mode.

TXpin is a variable/constant/expression that specifies the I/O pin to be used as a transmitter. This pin will be set to output mode.

`Wresponses` is a 3 WORD array that is used by different commands. This array is not used by all the commands. If the array is not used by a command, any value can be passed to the function (e.g. **`ZERO`**), and it will be ignored. However; something always must be passed in all the arguments. If the usage of the array is not specified in the tables below, then it is not used and ignored by the function.

`Sparameter` is a string array of 16 alpha-numeric characters that is used by different commands. This parameter is not used by all the commands. If the parameter is not used by a command, any value can be passed to the function (e.g. **`ZERO`**), and it will be ignored. However; something always must be passed in all the arguments. If the usage of the parameter is not specified in the tables below, then it is not used and ignored by the function.

Command Categories			
Command Category		*Value*	`command` *Symbol*
RESET		1	**BT_SWRST**
		2	**BT_HWRST**
BLUETOOTH	Mode	3	**BT_MODE**
	Status	4	**BT_CANCEL**
		5	**BT_SCAN**
		6	**BT_SCANT**
		7	**BT_SCANA**
	Connection	8	**BT_CLAST**
		9	**BT_CA**
		10	**BT_REL**
	Information	11	**BT_ALAST**
		12	**BT_SIGSTR**

Command Categories			
Command Category		*Value*	**command** *Symbol*
	Security	13 14 15	**BT_KEY** **BT_STKEY** **BT_STAUTH**
	Miscellaneous	16 17	**BT_STLP** **BT_NAME**

command *Symbol*	*Value*	*Command*	*Description*
BT_SWRST	1	Software Reset	This has the same effects as power cycling the unit. This command disconnects any connected Bluetooth devices, and stops ongoing tasks. After rebooting, the status will be decided by the preset operation mode.
BT_HWRST	2	Hardware Reset	All parameters are initialized to factory defaults.
BT_MODE	3	Set Operating Mode	Operating Mode is set to the one given by **Wparameters[0]**. **Wparameters[0]** = 0: MODE 0, (Default) **Wparameters[0]** = 1: MODE 1 **Wparameters[0]** = 2:

command *Symbol*	*Value*	*Command*	*Description*
			MODE 2 **Wparameters[0]** = 3: MODE 3
BT_CANCEL	4	Terminate the Current Task	This terminates the current task being executed, such as Inquiry scan and Page scan.
BT_SCAN	5	Wait for Inquiry and Connection From Other Bluetooth Devices	This allows Inquiry and Connection from the other Bluetooth devices. When connection is made the BD address of the connected device is returned in the **Wresponses** array as follows: **Wresponses[0]** = Left most 4 digits of the BD Address **Wresponses[1]** = Middle 4 digits of the BD Address **Wresponses[2]** = Right most 4 digits of the BD Address
BT_SCANT	6	Wait for Inquiry and Connection From Other Bluetooth Devices for a Given Duration	The module waits for **Wparameters[1]** seconds for Inquiry and Connection from other Bluetooth devices. If the **Wparameters[1]** is 0, the module waits forever. **Wparameters[0]** = 1: Allows Inquiry Scan **Wparameters[0]** = 2:

command *Symbol*	*Value*	*Command*	*Description*
			Allows Page Scan **`Wparameters[0]`** = 3: Allows both Inquiry and Page Scan When connection is made the BD address of the connected device is returned in the **`Wresponses`** array as follows: **`Wresponses[0]`** = Left most 4 digits of the BD Address **`Wresponses[1]`** = Middle 4 digits of the BD Address **`Wresponses[2]`** = Right most 4 digits of the BD Address
BT_SCANA	7	Wait for Connection by the Bluetooth Device with the Given Bluetooth Device (BD) Address	The module waits for **`Wparameters[3]`** seconds to be connected to by the Bluetooth device with the given BD address. If **`Wparameters[3]`** is 0, the module waits forever. **`Wparameters[0]`** = Left most 4 digits of the BD Address **`Wparameters[1]`** = Middle 4 digits of the BD Address **`Wparameters[2]`** = Right most 4 digits of the BD Address

command *Symbol*	*Value*	*Command*	*Description*
BT_CLAST	8	Connect to the Last Connected Bluetooth Device	The CS47100X devices save the BD Address of the Bluetooth device that was last connected to it. When connection is made the BD address of the connected device is returned in the **Wresponses** array as follows: **Wresponses[0]** = Left most 4 digits of the BD Address **Wresponses[1]** = Middle 4 digits of the BD Address **Wresponses[2]** = Right most 4 digits of the BD Address
BT_CA	9	Connect to a Specific Bluetooth Device with a Given BD Address	The CS47100X attempts to connect to the Bluetooth device with the given BD address. To make a successful connection, the Bluetooth device must be in Page scan mode. This attempt continues for 5 minutes. **Wparameters[0]** = Left most 4 digits of the BD Address **Wparameters[1]** = Middle 4 digits of the BD Address **Wparameters[2]** = Right most 4 digits of the BD Address
BT_REL	10	Release the Current Connection	The current Bluetooth connection is disconnected. It takes about 30 seconds to detect an abnormal

command *Symbol*	*Value*	*Command*	*Description*
			disconnection such as power off or moving out of service range.
BT_ALAST	11	Returns the BD Address of the Last Connected Device	The CS47100X devices return the BD Address of the Bluetooth device that was last connected to it. This is returned in the **Wresponses** array. **Wresponses[0]** = Left most 4 digits of the BD Address **Wresponses[1]** = Middle 4 digits of the BD Address **Wresponses[2]** = Right most 4 digits of the BD Address
BT_SIGSTR	12	Test Signal Strength	When a Bluetooth connection is established, you can use this command to query the module for the signal strength. This is returned in the **Wresponses** array. **Wresponses[0]** = LinkQuality with range [0, 255]. Higher is better. **Wresponses[1]** = RSSI (interference) with range [0, 255]. Lower is better. If the LinkQuality is close to 255, and the RSSI is close to 0, it implies that the signal strength is good.
BT_KEY	13	Change Pin Code	Pin code is a string in the **Sparameter** array, which allows up to 16 alpha-numeric characters. Based

command *Symbol*	*Value*	*Command*	*Description*
			on this pin code, the CS47100X generates a link key which is used in the actual authentication process. The default code is “1234”.
BT_STKEY	14	Set Generation of Link Key Every Time a Connection Is Made	If **Wparameters[0]** is set to 1, the CS47100X asks for the pin code every time a connection is made. This can be used to increase security. If **Wparameters[0]** is set to 0, this feature is deactivated.
BT_STAUTH	15	Set Authentication and Data Encryption	Authentication is set with **Wparameters[0]**. **Wparameters[0]** = 0: Inactive (Default) **Wparameters[0]** = 1: Active Encryption is set with **Wparameters[1]**. **Wparameters[1]** = 0: Inactive (Default) **Wparameters[1]** = 1: Active
BT_STLP	16	Set Low Power Mode	If **Wparameters[0]** is set to 1, Low Power Mode is enabled. If **Wparameters[0]** is set to 0, this feature is deactivated. When activated, and while data is not

command *Symbol*	*Value*	*Command*	*Description*
			being transmitted or received, the CS47100X goes into low power mode to save power. It takes a few seconds to wake the CS47100X from low power mode.
BT_NAME	17	Change Device Name	The CS47100X can have a user friendly name for easy identification. The name is allowed to be up to 30 alpha-numeric characters, and is passed in the **Sparameter** character array. The default is "PSDv3b-445566".

BLUEIN_CS47100X

```
NIBBLE BLUEIN_CS47100X(WORD timeout, BYTE buffer[],
                          BYTE N, BYTE RXpin);
```

The **BLUEIN** function receives asynchronous serial data from the CS47100X Bluetooth Module via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25), and the ATN pin (Pin 26).

timeout is a variable/constant/expression (0 – 65535) that tells **BLUEIN** how long to wait in mS for incoming data. If data does not arrive in time or if the buffer gets full before the timeout condition occurs, the function will return with the appropriate return code. The value of 0 is special; it indicates that the function will wait until it receives enough data to fill the buffer before it returns. The usage of timeouts applies to each data byte being received.

buffer is the name of the array of **BYTES** that the function will use as storage to return the received data to the calling function. This can be an array of length equals to 1 byte, and the maximum length of the buffer is 255 bytes.

N is the number of bytes (1 - 255) to be placed in **buffer**. The buffer is filled from the low to high direction of its index address space (i.e. from 0 to **N**-1).

If you do not use the timeout feature, the commands wait forever to get the number of bytes that you requested. However, the number of bytes that you get can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will get only 4 bytes. Then the command will finish after the 4 bytes are received, and these will be in `mybuffer[0]` through `mybuffer[3]`. You do have to know how many bytes you expect each time you use the command. If you do not know, you have to get 1 byte at a time, and put it in a larger array.

RXpin is a variable/constant/expression that specifies the I/O pin to be used as a receiver. This pin will be set to input mode.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
BTIN_ARGERR	0	There was at least one error in the arguments of the function. Function did not execute.
BTIN_BUFULL	1	Function returned with the buffer full of data.
BTIN_TIMOUT	2	The function exited after a timeout condition occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the timeout occurred. If this location contains the value 0xFF, this indicates that no data was received.
BTIN_OVRERR	3	An overrun error occurred. The highest index of buffer (**N**-1) contains the index of the last byte that was received successfully before the error occurred. If this location contains the value 0xFF, this indicates that no data was received.

BLUEOUT_CS47100X

```
BIT BLUEOUT_CS47100X(BYTE buffer[], BYTE N,
                     BYTE TXpin);
```

The **`BLUEOUT`** function transmits asynchronous serial data to the CS47100X Bluetooth Module via any I/O pin, except the built-in asynchronous serial transmitter (Pin 24) and receiver (Pin 25), and the ATN pin (Pin 26).

`buffer` is the name of the array of **`BYTES`** that the function will send. This can be an array of length equals to 1 byte, and the maximum length of the buffer is 255 bytes.

`N` is the number of bytes (1 - 255) in **`buffer`** to be sent. The buffer is processed from the low to high direction of its index address space (i.e. from 0 to **`N`**-1).

The number of bytes that you send can be any number, as long as it is less than or equal to what you have allocated. In other words, you can have

```
BYTE mybuffer[10];
```

and then tell the command that you will send only 4 bytes. Then the command will finish after the 4 bytes in `mybuffer[0]` through `mybuffer[3]` are sent.

`TXpin` is a variable/constant/expression that specifies the I/O pin to be used as a transmitter. This pin will be set to output mode.

On exit, the function returns one of the following exit codes.

return code	*value*	*Meaning*
`BT_ARGERR`	0	There was at least one error in the arguments of the function. Function did not execute.
`BT_BUFEMP`	1	Function returned after transmitting all data in the buffer.

HUMSIN_CS490000

```
float HUMSIN_CS490000(BYTE Dpin, BYTE Cpin,
                      NIBBLE command, NIBBLE format);
```

The **`HUMSIN`** function returns a measurement of Relative Humidity, Temperature, Dew Point. If the function is not successful, it returns the value of 500, which is outside the range of any measurement by the sensor in any of the available formats. This could mean that there was an error in the arguments of the function or some other problem.

`Dpin` is a variable/constant/expression that specifies the I/O pin to be used as the DATA pin.

`Cpin` is a variable/constant/expression that specifies the I/O pin to be used as the SCK pin. This pin will be set to output mode.

`command` is a variable/constant/expression (1 – 3) indicating the measurement to get from the sensor.

***Command* Symbol**	***Value***	***Description***
`HUMS_REH`	1	Returns the Relative Humidity. Range: 0% – 100%
`HUMS_TMP`	2	Returns the Temperature in the scale given by **`format`**. Range: -40 – 123.8 °C, or Range: -40 – 254.9 °F, or Range: 233.15 – 396.95 °K
`HUMS_DPT`	3	Returns the Dew Point Temperature in the scale given by **`format`**. Range: -40 – 123.8 °C, or Range: -40 – 254.9 °F, or Range: 233.15 – 396.95 °K

`format` is a variable/constant/expression that indicates the format for the returned Temperature or Dew Point value according to the table below. This argument is not used if the measurement requested is Relative Humidity. In this case, any value can be passed to the function (e.g. **`ZERO`**), and it will be ignored. However, something always must be passed in all the arguments.

Symbol	***Value***	***Temperature Scale***
`TEMPS_C`	1	Degrees Celsius

Symbol	*Value*	*Temperature Scale*
`TEMPS_F`	2	Degrees Fahrenheit
`TEMPS_K`	3	Degrees Kelvin

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.