Sierra Radio Systems

Getting Started with the

HamStack

C Development Environment

Rev 1.07

Introduction

The Hamstack C library provides a set of interrupt driven functions designed to make it straightforward to develop microcontroller based ham radio projects.

This manual documents those functions and simple examples that use them. In addition, it describes an electronic keyer project which uses a number of the features that might be useful in any number of projects. It demonstrates techniques for displaying and selecting parameters on an LCD display, and implementing timing sensitive operations with timer interrupts.

We have many ideas for additional functions, and hope to add them to the library in the coming months.

After loading the tools, we recommend compiling and trying out each of the example programs. The files are all included in the zip file containing the Hamstack project directories. If these do not compile correctly, something is wrong with the tool setup. There are some hints on correcting common tool setup problems in the Appendix at the end of this manual.

John, KJ6K
George, KJ6VU

Version changes

V1.07

1. The millisecond timer interrupt ran slow by 11 instruction cycles times (0.1% for the 40MHz 18F4620) due to a sign error in compensation for the time required to updated the counter within the interrupt. This has been fixed, so that the timer error is due only to the crystal clock frequency error.
2. Functions have been added to read and write array variables to the internal EEPROM: eeprom_readarr() and eeprom_writearr().
3. The brownout reset has been enabled for the 18F46K22 processor in the config.c file, with voltage threshold set at 2.5V. This will likely be expanded to be the default for all processors in the next release. It provides for cleaner initialization and shutdown in the presence of unstable power situations.
4. There was a serious bug in the cwsend() and cwsendr() functions which caused the wrong memory locations to be written when loading the cw buffer. This is fixed in this version.
5. The user_code_cwid.c example has been added. It provides a CW ID program which also demonstrates a number of programming techniques.

Introduction to the Tools

Microchip MPLab IDE

Microchip's MPLab integrated development environment is a free tool that is great for programming any of Microchip's products. It includes an editor, fully featured macro assembler, and built-in support for all of Microchip's programming and debugging tools. The Microchip C compilers, and a number of 3rd party compilers integrate into MPLab, so that you can use the same environment for editing, compiling, programming and debugging your entire project.

Microchip C18 Compiler

There are many available high level language tools available for programming the Microchip 18F series of microcontrollers. At Sierra Radio Systems, we have used several of these extensively. We have had a good experience with the Microchip C18 compiler. It has very few (in any) bugs, and generates good quality code. It runs from within MPLab, it is easy to mix C and assembly code, and works with MPLab source level debugger. There are two versions of the compiler, one free, and one that is $395. The free version is complete and has no restrictions on code size or use of memory. Some of the advanced optimizations that will reduce code size by 30 to 40% are only available in the paid version of the compiler

Hamstack Library

We have created a library of commonly used functions for ham projects, which can easily be incorporated into your projects. In addition, we have constructed a project outline and set of C files which makes is very easy to get started on even fairly sophisticated projects by making simple edits to one or two small files.

Some of the functions in the library are:

- Unlimited number of millisecond level, interrupt driven timers
- Sine wave tone generation, running in the background
- Interrupt driven serial port send and receive; it runs in the background while your program is doing other things
- Interrupt driven CW generation
- Interrupt driven DTMF generation
- 2 line x 16 character LCD display support
- Shift register digital i/o expansion
- A/D converter support
- Dallas 18B20 one-wire digital temperature chip support
- Interrupt driven rotary encoder support, for user controls
- CW keyer functions

As of this writing, both the 18F4620 and 18F46K22 processors are supported. Support is automatically configured for either 40MHz or 64MHz clock rates, and can easily be modified to support other clock rates.

Loading the Tools

The Microchip tools run under the Windows environment. We have successfully used them on both Windows XP and Windows 7.

MPLab

- Go to www.microchip.com, and click on the link to MPLab IDE that is on the front page of the site.
- Download and run the MPLab IDE install file that is near the bottom of the page.
- Use the default installation, and allow it to set your environment variables to point to the tools.

You may see a dialog like this at the end of the installation. If so, click on the "NO".

If you have purchased a Microchip programmer/debugger, such as the ICD3 or PicKit 3 (highly recommended),, do NOT plug this into your USB port before installing MPLab. It will mess up the driver installation. These devices come with a CD that includes MPLab. It is best to download the latest version directly from the Microchip web site and install that version.

Any other Pic programmer can also be used, and will work fine. They just may not have hardware debugging capability.

C18 (after installing MPLab)

- Go to www.microchip.com, click on "development tools", then navigate to "compilers" and select "MPLAB C Compiler for PIC18 MCUs".
- Download and run the "Standard-Eval Version", from the list near the bottom of the page.

If it asks if you should use the linker with C18 or with MPLab, choose C18.

Hamstack Project Libraries

- Create a directory where you want to work on your programs.
- Copy the hamstackc_nnn.zip file into this directory, and unzip it into its own subdirectory.
- Copy this subdirectory and rename it to create the working directory for your first project.

By creating a new copy of the entire directory each time you work on a new project, you can always access the original files.

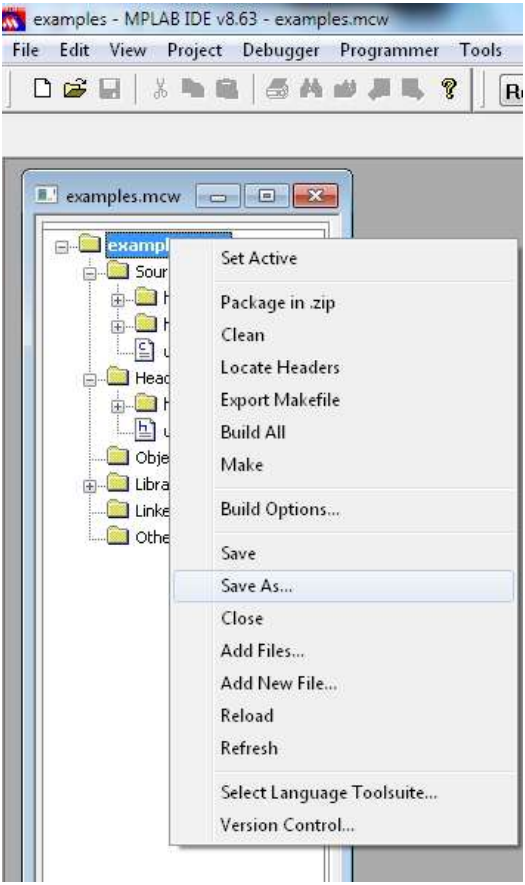
Loading the Tools (cont'd)

After setting up the renamed copy of the hamstackc directory, you can click on the examples.mcw file to open MPLab with the examples project loaded.

Creating a new program

We always start a new program by copying the entire directory for an existing program and renaming it, as shown above. To rename the project file and program hex file that it generates:

- Start MPLab with the existing project. Find the project window in MPLab. It's the one with a window title of *projectname*.mcw, and a tree structure below with *projectname*.mcp at the top.
- Right click on projectname.mcp.
- Click Save As.. Call it what you want, making sure that it is saved to the same directory in which you are working (it might not do this by default). This creates *newname*.mcw and *newname*.mcp files. The old .mcw and .mcp files can be deleted after this, if desired.



Example Programs

The example programs in the following pages mirror the Basic examples in the Hamstack introductory manual. Following that are a few examples to illustrate C library functions which are not yet supported in the Basic library. Finally, we show an electronic keyer project, which is a more substantial example that illustrates programming techniques which enable highly responsive programs.

For the first examples, refer to the introductory manual for the hardware connections required to run the demonstration:

Digital output – the blinking LED
Digital input – adding a push button
Controlling a relay
Analog input – read a DC voltage
RS-232 serial output
Expand digital outputs with a 74595 output shift register
Expand digital intputs with a 74165 shift register
Add an alphanumeric LCD display
Measure the temperature using a DS18B20

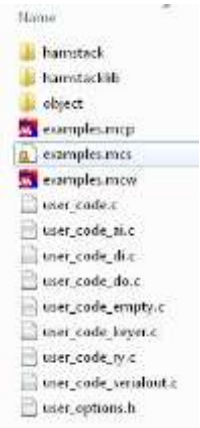
The next few examples illustrate some extra functionality which is included with the C library:

Interrupt driven serial input and output
Tone generation
DTMF generation
CW generation

The final example is more substantial:

CW keyer

Compiling the Examples

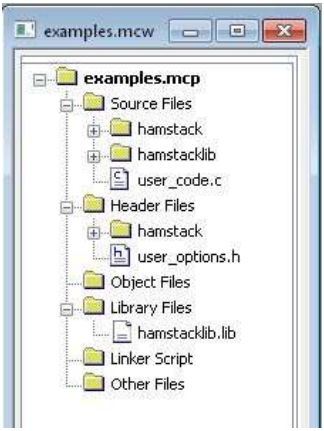


The source code for the examples is in the project directory, along with subdirectories which contain the Hamstack library code.

Open the project workspace in MPLab by either clicking on *examples.mcw*, or use the File/Open Workspace menu from within MPLab once it is already opened.

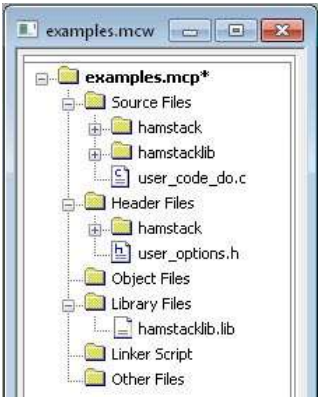
The project window defines the components which get compiled into your final program.

There are two files that must be edited to create a program. These are *user_code.c* and *user_options.h*. The C source code for a program is in *user_code.c*. Options that tell the compiler which library interrupt functions are needed, and what to put into initialization code are specified in *user_options.h.* The default *user_options.h* file has everything enabled for most of the examples (exception are the CW, keyer, and encoder examples), so it does not need to be changed to run those examples as long as the default pin assignments are used to attach the hardware associated with each example. Unused modules can be commented out with //, or enable by removing the //.



The default examples project window, shown to the left, includes the *user_code.c* file in the program. To compile one of the specific examples, substitute the corresponding .c file for *user_code.*c.

First, right-click on *user_code.c*, and select remove. Then, right-click on Source Files, select Add Files, and then select the user code file that is desired. The result is shown to the right.



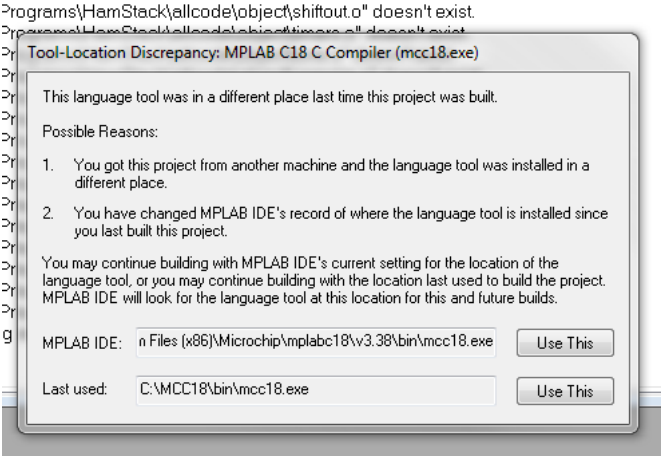
The Hamstack library pieces are hidden underneath the *hamstack* and *hamstacklib* folders. They can be expanded to see and modify what is inside of them. For most applications, they do not need to be touched.

Compiling the Examples (cont'd)

To compile:

- Select the processor using the Confgure/Select device menu. Hamstack library support is provided for the 18F4620 (default) and the 18F46K22.
- Select the corresponding Hamstack library. In the hamstalib directory, copy either the hamstacklib_18F4620.lib or the hamstacklib_18F46K22.lib and rename the copy as hamstacklib.lib. The Hamstack directory provided has already done this for the 18F4620. You will need to replace it with the 18F46K22 version if you are using that processor. The linker will return with an error message complaining that "processor types do not agree across all input files" if you are linking to the wrong library.
- Click on the Build All icon.

If you see a box like the message below, click on the first option. This happens when the project is saved on a machine with the C compiler installed in a different directory than on your machine.



Description

This example differs slightly from the Basic example. You do not need to wire up an LED. It drives the status LED that is on the Hamstack CPU board, which is connected to PORT C, pin 0.

When the CPU C0 pin is driven to a “1” state, it puts 5V on the anode of the LED, causing it to light up.

C Code

```
#include "hamstack.h"

void user_init(void)
{
  STATUSLEDDIR = DIGOUT;
}

void user_code(void)
{
  if (STATUSLEDOUT == 0)
    STATUSLEDOUT = 1;
  else
    STATUSLEDOUT = 0;
  waitlong(500);  // 0.5 s
}
```

The *hamstack.h* file contains all of the overhead glop that must be included in all programs. Details are selected by items added to the *user_options.h* file.

The user_init() function gets called once, at the beginning of the program, before any interrupts are turned on.

STATUSLEDDIR and DIGOUT are defined in *pins.h*, which is included by *hamstack.h.* The statement in user_init() is used to make the C0 pin connected to the status LED an output.

The user_code() function gets called repeatedly, forever, by the main program function, main(), which is part of the Hamstack library.

The waitlong() function is a Hamstack C library function which loops the processor long enough to use up the number of milliseconds given in its argument.

Notes

Description

This example differs slightly from the Basic example. You do not need to wire up a switch. It recognizes presses of either the mode switch on the CPU board, which is connected to PORT A, pin 4, or a switch wired to Port B, pin 1, as shown in the Basic example.

C Code

```
#include "hamstack.h"

void user_init(void)
{
  MODESWDIR = 1;
  B1DIR = 1;
  STATUSLEDDIR = 0;
}

void user_code(void)
{
  if (!MODESWIN || !B1IN)
    STATUSLEDOUT=1;
  else
    STATUSLEDOUT = 0;
}
```

The *pins.h* file defines MODESWDIR, B1DIR, and the other words that are in all caps. Making constant or macro definitions that are defined in one of the *.h* files all capital letters is a convention in C, but is not required. The compiler is case sensitive, so MODESWDIR would refer to something different than modeswdir.

All statements in C are terminated in a semicolon. The end of a line doesn't mean anything. It is equivalent to a space.

Groups of statements are bounded by matching curly braces.

The conditional in an if statement must be surrounded by matching parenthesis. The stuff that gets executed if the condition is true is either a single statement, or a group of statements inside curly brackets.

! is a logical not operator. || is a logical or operator. Basic would use the words, *not*, and *or* to mean the same thing. 5V on an input pin, reads as a 1, which is logically true. 0V reads as a 0, which is logically false.

Notes

Description

This example behaves identically to the Basic example.

C Code

```
#include "hamstack.h"

void user_init(void)
{
  STATUSLEDDIR = 0;
  B3DIR = 0;
}

// This runs over and over
void user_code(void)
{
  if (STATUSLEDOUT) {
    STATUSLEDOUT=0;
    B3OUT = 0;
  } else {
    STATUSLEDOUT = 1;
    B3OUT = 1;
  }
  waitlong(500); // 0.5s
}
```

Comments in C are separated from the executing code using ether a matching /* and */ pair, for by //.

Everything between /* and the following */ is a comment, which is ignored by the compiler.

Everything after the character pair, //, until the end of the line is also a comment.

As a matter of convention, we prefer to use the /* and */ only at the top of a source code file to write a description of what is in the file. Other comments are all delineated with //. This makes it possible to use /* and */ in the middle of the file to comment out sections of code during debugging.

/* and */ cannot be nested.

Notes

RS-232 Output user_code_serialout.c

Description

This example behaves identically to the Basic example.

C Code

```
#include "hamstack.h"
#include <stdio.h>

void user_code(void)
{
  static char s[50];
  static unsigned int i = 0;

  i++;
  sprintf(s,
    "The count is = %d\r\n",
    i);
  serial_puts(s);
  waitlong(500);
}
```

stdio.h is part of the standard C library, and is needed in order to sue the sprint() function that formats the output string.

In the standard C library, the puts() function is used to send a string to the standard output, which is the screen in a command line interface. The Hamstack serial_puts() function sends a string to the serial port.

The SERIAL_INT_ENABLED constant must be defined in the user_options.h file.

Notes

Description

This example behaves identically to the Basic example.

C Code

```
#include "hamstack.h"

void user_code(void)
{
  static char s[50];

  if
   (serial_readln(s,sizeof(s)-
   1)) serial_puts(s);
}
```

serial_readln() searches for a <CR> or <LF> character in the intput buffer. If one is present, it copies the buffer up to that point into s and returns 1. If none is found, it returns 0. If there isn't enough room for the incoming string in s, it copies what it can fit, and returns 3.

This code will properly echo what is received on a line-by-line basis, even if the incoming lines are longer than the string s has room for.

The SERIAL_INT_ENABLED constant must be defined in the user_options.h file.

Notes

Description

This example behaves identically to the Basic example.

C Code

```
#include "hamstack.h"

void app_once(void)
{
 lcd_putrs("Hello, World");
 lcd_line2();
 lcd_putrs("2nd line");
 serial_putrs("Hello, serial
   port\r\n");
}
```

Writing to the LCD is similar to the simplest functions to write to the serial port. Additional functions position the cursor at different locations on the LCD display.stdio.h is part of the standard C library, and is needed in order to sue the sprint() function that formats the output string.

The SERIAL_INT_ENABLED and LCD_ENABLED constants must be defined in the user_options.h file.

Notes

Description

This example behaves identically to the Basic example.

C Code

```
#include "hamstack.h"
#include <stdio.h>

void user_once(void)
{
  serial_putrs("\r\n");
}

void user_code(void)
{
  static char smv[20];
  static char s[50];
  static int oldadval = 0;
  int adval;

  adval = adc_readmv(0);
  if (adval != oldadval) {
    oldadval = adval;
    sprintmv(smv,adval);
    sprintf(s,
      "DC Volts = %s \r\n",
      smv);
    serial_puts(s);
  }
}
```

stdio.h is part of the standard C library, and is needed in order to sue the sprint() function that formats the output string.

In the standard C library, the puts() function is used to send a string to the standard output, which is the screen in a command line interface. The Hamstack serial_puts() function sends a string to the serial port. The serial _putrs() function does the same thing, but with strings that are in ROM memory.

The 18F46K22 and 18F46K80 processors have a built in voltage reference for the A/D converter. The 18F4620 uses the 5V supply as the reference. The 18F46K80 has a 12 bit resolution counter. The others are 10 bit. Higher accuracy voltage measurements are therefore possible with the 18F46K80.

The sprintmv() function converts the integer result of the adc_readmv() function into a decimal string format, with 3 digits to the right of the decimal point, rounded to the nearest millivolt.

The SERIAL_INT_ENABLED and ADC_ENABLED constants must be defined in the user_options.h file.

Notes

Multiple Analog Input user_code_ai4.c

Description

Scan the 4 analog inputs AN0..3, and display the values on a two line LCD display.

C Code

```
#include "hamstack.h"
#include <stdio.h>
#define AVGS 128

void user_once(void) {
 lcd_clear();
}

void user_code(void) {
 static char smv[20];
 static char s[50];
 static int olda[4]={0,0,0,0};
 static long adv[4]={0L};
 int newval;
 unsigned char i;
 static unsigned int avgs=0;

 avgs++;
 for (i=0; i<4; i++) {
  newval = adc_readmv(i);
  adv[i] += newval;
  if (avgs>=AVGS) {
   adv[i]=(adv[i]+AVGS/2)/AVGS;
  }
  if ((avgs>=AVGS)&&(adv[i]
      != olda[i])) {
   olda[i] = adv[i];
   sprintmv(smv,olda[i]);
   sprintf(s,"%8s",smv);
   lcd_cursorpos(i/2,
     (LCD_WIDTH/2)*(i&1));
   lcd_puts(s);
  }
 }
 if (avgs>=AVGS) {
  avgs=0;
  for (i=0;i<4;i++) adv[i]=0;
 }
}
```

The version here is compactly formatted to fit on the page. See the source code file for an easier to read version.

All 4 ADC inputs are read, and values are added to elements of the adv array. The idea is to display an average of 128 read values to demonstrate averaging to eliminate noise.

ADC inputs where are left unconnected will read poorly defined values which partly reflect the voltage on the pins which are connected. If the signal to be measure has an impedance of more than a few kilohms, an op amps buffer should be used to drive the ADC inputs.

After the 128 values are added, the array values are divided by 128. Integer division truncates any remainder, so we add ½ of the divisor before dividing to properly round the result.

The results are displayed in the four corners of a two line display. The display is only updated when the average value changes to minimize blinking of the display as it is updated.

The LCD_ENABLED and ADC_ENABLED constants must be defined in the *user_options.h* file. If something other than a 2 line by 16 character display is used, then the LCD_LINES and LCD_WIDTH parameters should also be defined in *user_options.h*.

Measuring Temperature user_code_ds1820.c

Description

Measure temperature with a Dallas DS18B20 one-wire temperature sensor. The connection is as described in the Basic example. This program will not work with more than one sensor attached. See user_code_ds1820s.c for an example using multiple sensors.

C Code

```
#include "hamstack.h"
#include <stdio.h>

void user_init(void) {
  ds1820_init(12);
}

void user_code(void) {
  int t, ti, tf;
  static char s[30];

  ds1820_convert();
  waitlong(1000);
  t = ds1820_readtf();
  ti = t/16;
  tf = ((t - ti*16)*10+8)/16;
  sprintf(s,"%d.%d F      \r",
   ti, tf);
  serial_puts(s);
  lcd_home();
  lcd_puts(s);
}
```

The one-wire interface is not interrupt driven, and interrupts are disabled for up to 60 us during critical timing portions of the one-wire communication with the temperature sensor. This can cause small, audible glitches in the tone generator if it is active.

The DS18B20 module is not initialized in the default code, so the ds1820_init() function must be called in user_init().

The parameter in the ds1820_init() function specifies the number of bits of resolution captured by the sensor. With 12 bit resolution, the conversion takes 750ms, so a 1 second wait is included before reading the temperature.

The sensor returns the temperature as an integer in degrees multiplied by 16. The messy math converts it to a decimal string for display.

The SERIAL_INT_ENABLED, LCD_ENABLED, and ONEWIRE_ENABLED constants must be defined in the user_options.h file.

Notes

Measuring Temperature user_code_ds1820s.c

Description

Measure temperature with a Dallas DS18B20 one-wire temperature sensor. This version is for multiple sensors connected at the same time. The sensors are all connected in parallel. Other types of one-wire devices may also be connected, and will be ignored by this program.

C Code

```
#include "hamstack.h"
#include <stdio.h>

#define NTPROBES 5
unsigned char devs[NTPROBES][8];
unsigned char nfound;

void user_init(void) {
  nfound=ow_search(devs,NTPROBES);
  ds1820_init(12);
}

void user_once(void) {
  char i, j;
  static char s[20];

  sprintf(s,"%d     ",nfound);
  lcd_home();
  lcd_puts(s);
  waitlong(2000);
  for (i=0; i<nfound; i++) {
    lcd_line2();
    for (j=7; j>=0; j--) {
      sprintf(s,"%02x",
        devs[i][j]);
      lcd_puts(s);
    }
    waitlong(2000);
  }
}

void user_code(void) {
  int t, ti, tf;
  static char s[30];
  unsigned char i;

  ds1820_convert();
  waitlong(1000);
  lcd_home();
  for (i=0; i<nfound; i++) {
    if (devices[i][0] == 0x28) {
      t = ds1820_readtfn(devs[i]);
      ti = t/16;
      tf = ((t - ti*16)*10+8)/16;
      sprintf(s,"%d.%d ",ti,tf);
      serial_puts(s);
      lcd_puts(s);
    }
  }
  serial_putrs("F\r");
  lcd_putrs("F");
}
```

`User_init()` is executed once at the beginning of the program, before any interrupts are started. In this program, the `ow_search()` function is used to search the one wire bus for all of the attached devices, capturing their unique ID codes to a sequence of 8 byte arrays, `devs`.

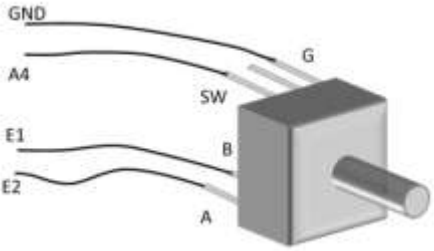
`User_once()` is executed once, after interrupts are started. This allows it to be used for outputting information to the serial port. In this case, the ID's of the devices on the one wire bus are output to both the LCD and serial port.

`User_code()` captures and displays the temperature for each of the temperature probes. Other devices are ignored by selecting only devices with a first ID byte of 0x28 (unique to the DS18B20).

Rotary Encoder user_code_encoder.c

Description

Implement controls using a mechanical rotary encoder. The Panasonic EVE-JBBF2020B is one model that works well. It provides 20 detented positions per turn of the shaft. The encoder support is interrupt driven and keeps track of the count of detents, even while the program is doing something else. The encoder should be connected to the processor pins as shown to the right. 10 kΩ pullup resistors need to be added from pin E1 to +5V and from E2 to +5V. A pullup is already included on pin A4 on the CPU board.



C Code

```
#include "hamstack.h"
#include "stdio.h"

void user_code(void) {
  int cnt, oldcnt;
  static char s[30];

  cnt = encoder_get(0);
  if (cnt<-100) {
    cnt = -100;
    encoder_reset(-100);
  } else if (cnt>100) {
    cnt = 100;
    encoder_reset(100);
  }
  if (cnt != oldcnt) {
    oldcnt = cnt;
    sprintf(s, "cnt = %d    ",
     cnt);
    lcd_home();
    lcd_puts(s);
  }
}
```

The encoder is monitored by the 1 ms interval timer interrupt. It counts up 1 for each detent in the clockwise direction and down 1 for each detent in the counter-clockwise direction.

The example shows that the encoder count can be reset to a specified value at any time. This can be used to limit the range of a parameter, as shown; or to initialize the count after selecting a new parameter to adjust.

The encoder module can support multiple encoders. The default initialization sets it up for one encoder (designated as encoder 0), attached to processor pins as shown above

The push switch is not used in this example code. It is used extensively in the keyer example.

The ENCODER_INT_ENABLED and LCD_ENABLED constants must be defined in the user_options.h file.

Sending CW user_code_cw.c

Description

Generate CW. Both tone and keyed outputs are available. The keyed output drives output pin B4 high for key down. The tone output is provided on the PWM1 output pin (C2). The project board provides a low pass filter on this pin to convert the pulse width modulated output to low distortion audio. See the project board documentation for further details on the tone output.

C Code

```
#include "hamstack.h"
#include <stdio.h>

void user_once(void) {
  cw_setspeed(18);
}

void user_code(void) {
  static int i;
  static char s[10];

  sprintf(s,"%d ", i++);
  cw_send(s)
  cw_sendr("This repeats
   until you turn it off.
   ");
}
```

The default CW speed is 18 WPM. The value in the `user_once()` function can be changed to be any reasonable value. It can be changed dynamically while CW is being sent.

The CW_INT_ENABLED and TONE_INT_ENABLED constants must be defined in the user_options.h file.

Notes

Sending DTMF user_code_dtmf.c

Description

Generate a sequence of DTMF tones on the PWM1 output pin (C2). The project board provides a low pass filter to turn the pulse width modulated output into low distortion audio. See the project board documentation for details on the tone output.

C Code

```
#include "hamstack.h"

void user_once(void) {
  lcd_putrs("DTMF test");
  dtmf_char('T');
  waitms(2000);
  dtmf_off();
  waitms(1000);
  dtmf_putrs("12345678");
}}
```

DTMF can be sent by either turning on and off a single character, or by sending an entire string. The valid characters are 0..9, A..D, *, #, T (dial tone), and Z (busy tone).

The DTMF_INT_ENABLED, TONE_INT_ENABLED and LCD_ENABLED constants must be defined in the user_options.h file.

Notes

Compass direction user_code_compass.c

Description

Read magnetic compass direction from a Honywell HMC6352 i2c magnetic compass. Send the result via the serial port and display on an LCD.

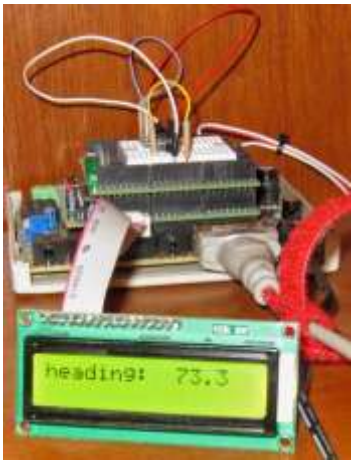
The full source code also include a calibration routine, triggered by the mode switch (not shown below).

C Code

```
#include "hamstack.h"
#include "stdio.h"

void user_once(void) {
 lcd_clear();
 lcd_home();
 waitlong(1000);
}

void user_code(void) {
 int cnt, oldcnt;
 static char s[30];

 cnt = compass_read();
 if (cnt < 0) {
  lcd_line2();
  sprintf(s,
        "i2c error: %d", -cnt);
  lcd_puts(s);
  waitlong(5000);
 } else {
  oldcnt = cnt;
  sprintf(s,
       "heading: %3d.%0d ",
        cnt/10, cnt%10);
  lcd_home();
  lcd_puts(s);
  sprintf(s, "%3d.%0d\r\n",
        cnt/10, cnt%10);
  serial_puts(s);
 }
 waitlong(100);
}
```

The 1 second delay here allows the sensor enough time to complete power-on initialization.

The only significant code here is to read the sensor. Everything else is just to display the result.



Notes

CW Keyer Example

This simple keyer demonstrates a number of the features of the Hamstack project board and many of the software modules described in the simple examples above.

In addition to the descriptions here, there is a separate document, *Keyer Project Prototype*, with photos showing the packaging of the keyer in the project board enclosure.

Keyer Hardware

The LCD is used to display the code speed, and also to display and modify various parameters of keyer operation. A convenient connector is provide on the project board to connect the LCD to the default output pins.

The encoder is used to change code speed and other parameters.

Side tone output is provided via the PWM1 output. The project board provides a low pass filter to convert this to sine wave output.

The project board has sockets for two types of opto-isolated switches, and an isolated RCA type connector output. This output is connected to CPU pin B4.

The project board also has a 1/8" stereo input jack connected to pins B0 and B1, for the keyer paddle input. 10 kΩ pullups to +5V and 0.1 µF bypass capacitors to ground are provided on these pins.

Keyer Features

- Adjustable CW speed in 1 WPM increments, with LCD display of speed
- Plain keyer, IAMBIC A and IAMBIC B modes
- Right or left handed key operation
- Adjustable weighting (in milliseconds)
- Adjustable sidetone frequency
- Optional automatic character spacing (not so good experimental idea)
- Last settings stored in EEPROM, and recalled on power up

There are many more features which can be added: memories, CW decoding, etc.

Keyer Software

The primary functions implementing the keyer are described here and listed below. For complete details, please consult the source code.

user_options.h

The following modules need to be enabled by removing any // comments designators from the corresponding #define: LCD_ENABLED, TONE_INT_ENABLED, CW_INT_ENABLED, TIMERS_INT_ENABLED, USER_TIMER_INT_ENABLED, ENCODER_INT_ENABLED, and KEYER_INT_ENABLED.

user_code_keyer.c

This file is the one that replaces user_code.c in the project window. All of the initialization and user interface code is in this module. A more detailed description of the functions in this file follows on the next few pages. Consult the source code to figure out exactly how it works. Don't hesitate to modify it and customize the keyer.

keyerdash.c, keyerdot.c, keyerinit.c, and keyerint.c, in hamstacklib

Although these files are in the hamstacklib subdirectory, they are not compiled into the Hamstack library at this time. They will likely get modified many times, so it make more sense to compile them with the rest of the project, until sometime in the future.

The *keyerint.c*, file contains the core keyer functionality, which is called by the 1 millisecond timer interrupt.

Keyer Software (cont'd)

Initialization sets things up before interrupts are turned on. The other Hamstack modules are initialized by Hamstack library code before this is called.

```
void user_init(void)
{
  unsigned char testval;

  keyermodeinit();  // initialize variable used for selection
  cwspeed=18;       // 18 WPM default
  cwweight=0;       // weight is signed milliseconds; + heavier

  testval = eeprom_read(1);// read the code speed from EEPROM
  if ((testval>80) ||
      (testval == 0)) {    // if ridiculous, EEPROM not set yet
    eeprom_write(0,keyertype);    // address 0 gets IAMBIC or not
    eeprom_write(1,cwspeed);      // address 1 gets speed in WPM
    eeprom_write(2,cwweight);     // address 2 gets weight in ms
    eeprom_writeword(3,cwtonefreq); // address 3,4 get tone Hz
  } else {                 // EEPROM previously set, so load it
    keyertype = eeprom_read(0);
    cwspeed = eeprom_read(1);
    cwweight = eeprom_read(2);
    cwtonefreq = eeprom_readword(3);
  }
}
```

Some stuff gets done only once, but after interrupts are already running, so that timers all work.

```
void user_once(void)
{
  // Set the code speed, initialize the encoder to the code
  // speed, and show it on the display
  encoder_reset(0,cwspeed);
  sprintf(s,"%d WPM      ",cwspeed);
  if ((cwspeed>0) && (cwspeed<80)) cw_setspeed(cwspeed);
  if (cwtonefreq==0) cwcont &= ~(CWSND1EN|CWSND2EN);
  else {
    if (!(cwcont&(CWSND1EN|CWSND2EN))) cwcont |= CWSND1EN;
    cw_setfreq(cwtonefreq,cwamp);
  }
  lcd_home();
  lcd_putrs("Hamstack keyer");
  lcd_line2();
  lcd_puts(s);
}
```

The main program code handles the user interface to display and change keyer paremeters. It is called repeatedly by the Hamstack library main() function. The keyer function itself, is called by the 1 ms timer interrupt and runs in the background.

```
void user_code(void)
{
  static unsigned char usermode = 0;     // current parameter
  int count;                             // encoder output value
  static unsigned char activity = 0;     // flag set on changes
  static unsigned char newstate = 0;     // flag, next parameter

  switch (usermode) {
    case CWSPEEDM:
      if (newstate) {             // initialize state on entry
        encoder_reset(0,cwspeed); // encoder gets current value
        newstate = 0;             // do this just once
      }
      if (modepressed) {   // go to next parameter
        modepressed = 0;   // handshake with button handler
        activity = 1;      // flag to update display
        newstate = 1;      // flag for next parameter init
        usermode = SETKEYERM;     // keyer mode is next
      } else {
        count = encoder_get(0);   // Check knob position
        if (count != cwspeed) {   // if it changed, change speed
          if (count <= 0 || count > 80) // verify in range
            encoder_reset(0,cwspeed);
          else {              // if so, then
            activity = 1;     // set flag to update the display
            cwspeed = count; // set the speed to the new value
            cw_setspeed(cwspeed); // update the CW generator
          }
        }
      }
      break;
    case SETKEYERM:               // Same idea, for IAMBIC mode
      if (newstate) {
        encoder_reset(0,keyermode);
        newstate = 0;
      }
      if (modepressed) {
        modepressed = 0;
        activity = 1;
        newstate = 1;
        usermode = SETLEFTYM;
      } else {
        activity=0;
        count = encoder_get(0);
        if (count>0) count = count % 4;
        else if (count<0) count = 4 - (-count)%4;
```

```
    if (count!=keyermode) {
      activity = 1;
      keyermode = count;
      switch (keyermode) {
        case 0: keyertype = keyertype &
                  ~(IAMBICA|IAMBICB|IAMBICB2);
          break;
        case 1: keyertype = (keyertype &
                  ~(IAMBICA|IAMBICB|IAMBICB2)) | IAMBICA;
          break;
        case 2: keyertype = (keyertype &
                  ~(IAMBICA|IAMBICB|IAMBICB2)) | IAMBICB;
          break;
        case 3: keyertype = (keyertype &
                  ~(IAMBICA|IAMBICB|IAMBICB2)) | IAMBICB2;
          break;
      }
      activity = 1;
    }
  }
  break;
case SETLEFTYM:        // Same idea for paddle reversal
  if (newstate) {
    if (keyertype&LEFTHANDEDKEY) encoder_reset(0,1);
    else encoder_reset(0,0);
    newstate = 0;
  }
  if (modepressed) {
    modepressed = 0;
    activity = 1;
    newstate = 1;
    usermode = SETWEIGHT;
  } else {
    count = encoder_get(0);
    count &= 1;
    if ((!count) != (!(keyertype&LEFTHANDEDKEY))) {
      keyertype ^= LEFTHANDEDKEY;
      activity = 1;
    }
  }
  break;
case SETWEIGHT:        // Same idea for setting weight
  if (newstate) {
    encoder_reset(0,cwweight);
    newstate = 0;
  }
  if (modepressed) {
    modepressed = 0;
    activity = 1;
    newstate = 1;
    usermode = SETTONEF;
```

```
  } else {
    count = encoder_get(0);
    if (count != cwweight) {
      if (count < -100 || count > 100)
        encoder_reset(0,cwweight);
      else {
        activity = 1;
        cwweight = count;
        cw_setspeed(cwspeed);
      }
    }
  }
  break;
case SETTONEF:          // Same idea for tone frequency
  if (newstate) {
    // handle out of range values
    if (cwtonefreq<0) cwtonefreq=0;
    else if (cwtonefreq>1500) cwtonefreq=1500;
    if (cwtonefreq==0 || cwtonefreq==1500)
      cw_setfreq(cwtonefreq,cwamp);
    encoder_reset(0,cwtonefreq/10);
    newstate = 0;
  }
  if (modepressed) {
    modepressed = 0;
    activity = 1;
    newstate = 1;
    usermode = SETAUTOSPM;
  } else {
    count = encoder_get(0);
    if (count*10 != cwtonefreq) {
      if (count < 0 || count > 150)
        encoder_reset(0,cwtonefreq/10);
      else {
        activity = 1;
        cwtonefreq = count*10;
        if (cwtonefreq==0) cwcont &= ~(CWSND1EN|CWSND2EN);
        else {
          if (!(cwcont&(CWSND1EN|CWSND2EN)))
            cwcont |= CWSND1EN;
          cw_setfreq(cwtonefreq,cwamp);
        }
      }
    }
  }
  break;
case SETAUTOSPM:        // Same idea for auto character space
  if (newstate) {
    if (keyertype&AUTOCHARSPACE) encoder_reset(0,1);
    else encoder_reset(0,0);
    newstate = 0;
```

```
      }
      if (modepressed) {
        modepressed = 0;
        activity = 1;
        newstate = 1;
        usermode = CWSPEEDM;
      } else {
        count = encoder_get(0);
        count &= 1;
        if ((!count) != (!(keyertype&AUTOCHARSPACE))) {
          keyertype ^= AUTOCHARSPACE;
          activity = 1;
        }
      }
      break;
  }  // encoder switch
  if (activity) {          // if change, then update display
    switch (usermode) {
      case CWSPEEDM: showcwspeed(); break;
      case SETKEYERM: showkeyermode(); break;
      case SETLEFTYM: showkeyerlefty(); break;
      case SETAUTOSPM: showkeyerautosp(); break;
      case SETWEIGHT: showkeyerweight(); break;
      case SETTONEF: showkeyerfreq(); break;
    }
  }
  // 5 second timer to revert to CW speed if no activity
  if (activity) setusrtos(MODETIMEOUT,50);
  activity = 0;
  if (!getusrtos(MODETIMEOUT)) {  // times out at 0
    if (usermode != CWSPEEDM) {   // if not speed, change it
      modepressed = 0;
      activity = 1;
      newstate = 1;
      usermode = CWSPEEDM;
    }
    // check the current values against the eeprom, and update
    // if different.
    if (eeprom_read(0) != keyertype) eeprom_write(0,keyertype);
    if (eeprom_read(1) != cwspeed) eeprom_write(1,cwspeed);
    if (eeprom_read(2) != cwweight) eeprom_write(2,cwweight);
    if (eeprom_readword(3) != cwtonefreq)
      eeprom_writeword(3,cwtonefreq);
  }
}
```

The core functionality of the keyer is contained in the `keyer_int()` function in the *keyer_init.c* file in hamstacklib. This gets called at precise 1 millisecond intervals.

```
void keyer_int(void)
{
  static int keyerdelay=0;        // timer that times everything
  static unsigned char state = 0;
  static struct {
    unsigned char dash : 1;
    unsigned char dot : 1;
  } pending = {0,0};

  if (keyerdelay) keyerdelay--;  // decrement the timers
  dashlever = keyer_dashlever();  // check paddle only once
  dotlever = keyer_dotlever();
  switch (state) {
    case 0:          // is sitting idle
      pending.dash = 0;
      pending.dot = 0;
      if (dashlever) {
        state = 1;
        keyerdelay = cw3el;
        keydown();
        charbits |= charbitptr;  // for reading sent code
        charbitptr <<= 1;        // for reading sent code
      } else if (dotlever) {
        state = 2;
        keyerdelay = cw1elw;
        keydown();
        charbitptr <<= 1;
      }
      break;
    case 1:          // dash key down
      if ((keyertype & (IAMBICA|IAMBICB|IAMBICB2)) && dotlever)
        pending.dot = 1;
      if ((keyertype & IAMBICB) && dotlever && dashlever)
        pending.dash = 1;
    case 2:          // dot key down
      if ((state!=1) && (keyertype & (IAMBICA|IAMBICB|IAMBICB2))
          && dashlever) pending.dash = 1;
      if ((state!=1) && (keyertype & IAMBICB) && dotlever
          && dashlever) pending.dot = 1;
      if (!keyerdelay) {
        keyup();
        keyerdelay = cw1el;
        state += 2;
      }
      break;
```

```
case 3:         // space after dash
  if ((keyertype & (IAMBICB|IAMBICB2)) && dashlever
      && dotlever) pending.dash = 1;
  if ((keyertype & (IAMBICA|IAMBICB|IAMBICB2)) && dotlever)
    pending.dot = 1;
  if (!keyerdelay) {
    if (dotlever || pending.dot) {
      state = 2;
      keyerdelay = cw1elw;
      keydown();
      pending.dot = 0;
      charbitptr <<= 1;
    } else if (dashlever || pending.dash) {
      state = 1;
      keyerdelay = cw3el;
      keydown();
      pending.dash = 0;
      charbits |= charbitptr;
      charbitptr <<= 1;
    } else {
      state = 5;  // between characters
      keyerdelay = cw2el;
      charbits |= charbitptr;
      keyer_charbitsout = charbits;
      charbits = 0;
      charbitptr = 1;
    }
  }
  break;
case 4:         // space after dot
  if ((keyertype & (IAMBICB|IAMBICB2)) && dotlever
      && dashlever) pending.dot = 1;
  if ((keyertype & (IAMBICA|IAMBICB|IAMBICB2))
      && dashlever) pending.dash = 1;
  if (!keyerdelay) {
    if (dashlever || pending.dash) {
      state = 1;
      keyerdelay = cw3el;
      keydown();
      pending.dash = 0;
      charbits |= charbitptr;
      charbitptr <<= 1;
    } else if (dotlever || pending.dot) {
      state = 2;
      keyerdelay = cw1elw;
      keydown();
      pending.dot = 0;
      charbitptr <<= 1;
    } else {
      state = 5;  // between characters
      keyerdelay = cw2el;
```

```
          charbits |= charbitptr;
          keyer_charbitsout = charbits;
          charbits = 0;
          charbitptr = 1;
        }
      }
      break;
    case 5:         // space after character
      if (keyertype & (IAMBICB|IAMBICB2)) {
        if (dashlever) pending.dash = 1;
        else if (dotlever) pending.dot = 1;
      } else if ((!pending.dot) && dashlever) pending.dash = 1;
      else if ((!pending.dash) && dotlever) pending.dot = 1;
      if ((!keyerdelay) || (!(keyertype&AUTOCHARSPACE))) {
        if (pending.dash) {
          state = 1;
          keyerdelay = cw3el;
          keydown();
          pending.dash = 0;
          charbits |= charbitptr;
          charbitptr <<= 1;
        } else if (pending.dot) {
          state = 2;
          keyerdelay = cw1elw;
          keydown();
          pending.dot = 0;
          charbitptr <<= 1;
        } else if (!keyerdelay) {
          state = 0;
          keyer_charbitsout = 1;  // space
        }
      }
      break;
  }  // end switch
}
```

Hamstack C Library Reference

The Hamstack C library provides support for a number of the Microchip C18 hardware peripherals and high level functions to perform some of the tasks that are often used in real projects. Where appropriate, the support uses hardware interrupts to enable high performance operation and excellent responsiveness to user input.

Modules in the library include:

Interrupt based functions that run in the background:

- Large number of millisecond resolution timers
- Serial port communications
- Sine wave tone generation
- CW generation
- DTMF sequence generation
- Rotary encoder support

Other functions:

- LCD character display output
- PWM analog output
- Shift register digital i/o expansion
- DS1820 one-wire temperature measurement

Program structure

Project and Directory structure
A project, and everything needed to compile that project are included in a single directory. When making new versions of a program, then entire directory should be copied. This insures that old working versions of a program can still be compiled exactly as when they were first created, even after Hamstack and programmer specific library functions have been updated or replaced.

It is intended that the primary source files for a given project reside in the outermost level of the project directory, along with the *projectname.mcw* and *projectname.mcp* files used by MPLab to define the project. At least 3 subdirectories exist under the main project directory; *hamstack*, *hamstacklib*, and *object*.

The *hamstack* subdirectory contains source code for portions of the hamstack library which must be recompiled with each program, so accommodate changes in hardware pin assignments, clock frequency, etc. It also contains the *.h* files that are included in project source files when library modules are used.

The *hamstacklib* subdirectory contains pre-compiled portions of the hamstack library. There are versions for each of the processors that are supported. Source code is only supplied for a select set of the modules, which the application programmer may want to modify or enhance.

The *object* subdirectory contains the intermediate object files generated by the compiler.

The program
All C programs begin execution with the `main()` function. In a Hamstack program, the application programmer does not normally create or edit this function. It is defined in the *main.c* file in the *hamstack* subdirectory and is automatically compiled and linked into the program whenever a project is compiled.

The Hamstack `main()` function does several things:

- It does any hardware initialization required by Hamstack library functions that have been enabled by the programmer.
- It calls the user defined function, `user_init()` to do any initialization that is additionally required by the functions written by the application programmer.
- It starts interrupts running – then enables hardware timers, serial port, and LCD functions, and any other interrupt driven functions.
- It does any further initialization required by Hamstack library functions (none at the time of this writing).
- It call the user defined function, `user_once()` to do anything that only needs to be done once, at the start of the program. This could include further

initialization which requires use of the interrupt driven timers, or may just send a welcome greeting out the serial port.

- It runs an infinite loop that repeats over and over. This first calls any library function operations that must take place periodically, but are not part of an interrupt handler. An example is the watchdog timer reset. It next calls the user defined function, `user_code()` where the main part of the user application resides.

Read the source code if you want to further understand what `main()` does.

Options

The *user_options.h* file is used to define which Hamstack library modules are initialized and are included in the Hamstack interrupt handlers. It is also used to set compile time values for some parameters such as the number of timers and serial port buffer sizes.

Most other default parameter values are specified in the *.h* file for that particular module.

Initialization for most modules enabled in the *user_options.h* file is performed by the default `hamstack_init()` function. The application programmer does not need to explicitly call the initialization function for those modules in the application code. A few of the modules have default pin assignments that are in conflict with other modules. Initialization for those modules must be explicitly called in the `user_init()` function. This is called out in the reference that follows.

Functions and variables need to be declared before they are used in a C program. There is a *.h* file associated with each of the Hamstack modules to provide those declarations. The name of that file is listed at the top of the module descriptions on the following pages. If *hamstack.h* is included at the top of each .c source file, the appropriate .h files will be automatically included, based on the definitions in *user_options.h.*

Code space can be saved by commenting out the #define statement for modules that are not used. For example, if the serial port is not used, the line in *user_options.h*:

```
#define SERIAL_INT_ENABLED
```

can be commented out with a pair of slash characters:

```
//#define SERIAL_INT_ENABLED
```

Interrupt management

Interrupts handlers are set up and managed by the functions in *interrupts.c* in the *hamstack* subdirectory. It is not necessary to edit this file to use the Hamstack library interrupt modules; the appropriate pieces are included in the code based upon the definitions in *user_options.h*.

Constants are defined in the *pins.h* file that can be used to refer to Hamstack CPU card pins, as they are labeled on the board silk screen.

```
#define A4DIR TRISAbits.TRISA4
#define A4OUT LATAbits.LATA4
#define A4IN  PORTAbits.RA4
#define MODESWDIR A4DIR
#define MODESWOUT A4OUT
#define MODESWIN  A4IN

#define A5DIR TRISAbits.TRISA5
#define A5OUT LATAbits.LATA5
#define A5IN  PORTAbits.RA5

#define B0DIR TRISBbits.TRISB0
#define B1DIR TRISBbits.TRISB1
#define B2DIR TRISBbits.TRISB2
#define B3DIR TRISBbits.TRISB3
#define B4DIR TRISBbits.TRISB4

#define B0OUT LATBbits.LATB0
#define B1OUT LATBbits.LATB1
#define B2OUT LATBbits.LATB2
#define B3OUT LATBbits.LATB3
#define B4OUT LATBbits.LATB4

#define B0IN PORTBbits.RB0
#define B1IN PORTBbits.RB1
#define B2IN PORTBbits.RB2
#define B3IN PORTBbits.RB3
#define B4IN PORTBbits.RB4

#define C0DIR TRISCbits.TRISC0
#define C1DIR TRISCbits.TRISC1
#define C2DIR TRISCbits.TRISC2
#define C3DIR TRISCbits.TRISC3
#define C4DIR TRISCbits.TRISC4
#define C5DIR TRISCbits.TRISC5

#define C0OUT LATCbits.LATC0
#define C1OUT LATCbits.LATC1
#define C2OUT LATCbits.LATC2
#define C3OUT LATCbits.LATC3
#define C4OUT LATCbits.LATC4
#define C5OUT LATCbits.LATC5
```

```
#define C0IN PORTCbits.RC0
#define C1IN PORTCbits.RC1
#define C2IN PORTCbits.RC2
#define C3IN PORTCbits.RC3
#define C4IN PORTCbits.RC4
#define C5IN PORTCbits.RC5

#define STATUSLEDDIR C0DIR
#define STATUSLEDOUT C0OUT
#define STATUSLEDIN C0IN

#define D0DIR TRISDbits.TRISD0
#define D1DIR TRISDbits.TRISD1
#define D2DIR TRISDbits.TRISD2
#define D3DIR TRISDbits.TRISD3
#define D4DIR TRISDbits.TRISD4
#define D5DIR TRISDbits.TRISD5
#define D6DIR TRISDbits.TRISD6
#define D7DIR TRISDbits.TRISD7

#define D0OUT LATDbits.LATD0
#define D1OUT LATDbits.LATD1
#define D2OUT LATDbits.LATD2
#define D3OUT LATDbits.LATD3
#define D4OUT LATDbits.LATD4
#define D5OUT LATDbits.LATD5
#define D6OUT LATDbits.LATD6
#define D7OUT LATDbits.LATD7

#define D0IN PORTDbits.RD0
#define D1IN PORTDbits.RD1
#define D2IN PORTDbits.RD2
#define D3IN PORTDbits.RD3
#define D4IN PORTDbits.RD4
#define D5IN PORTDbits.RD5
#define D6IN PORTDbits.RD6
#define D7IN PORTDbits.RD7

#define E0DIR TRISEbits.TRISE0
#define E1DIR TRISEbits.TRISE1
#define E2DIR TRISEbits.TRISE2
```

```
#define E0OUT LATEbits.LATE0
#define E1OUT LATEbits.LATE1
#define E2OUT LATEbits.LATE2

#define E0IN PORTEbits.RE0
#define E1IN PORTEbits.RE1
#define E2IN PORTEbits.RE2
```

```
// TRIS bit definitions
#define DIGOUT 0
#define DIGIT
```

TIMER_INT_ENABLED should be defined in *user_options.h* to use these functions.

The Timer 1 hardware timer is configured by the Hamstack library to interrupt the processor at a 1000 Hz rate. Many time critical events can be controlled using this interrupt. The Hamstack library implements multiple countdown timers using this interrupt that decrement in the background as the application program is doing other tasks. These timers stop when they get to zero. The number of timers is configurable at compile time by editing the *user_options.h* file. The following sets of timers are available:

```
volatile unsigned int usrmstmrs[NUSRMSTMRS]; // 1 ms interval
volatile unsigned char usrtos[NUSRTOS];      // 0.1 s interval
volatile unsigned long usrseconds[NUSRSECONDS]; // 1s interval
```

The default number of timers, defined by *NUSRMSTMRS*, *NUSRTOS*, and *NUSRSECONDS* in *user_options.h*, is 1 for all three timer types. Edit *user_options.h* to allocate additional timers.

The following commands should be used to access these timers:

```
void setusrmstmr(unsigned char timern, unsigned int value)
unsigned int getusrmstmr(unsigned char timern)
void setusrtos(unsigned char timern, unsigned char value)
unsigned char getusrtos(unsigned char timern)
void setusrseconds(unsigned char timern, unsigned long value)
unsigned long getusrseconds(timern)
```

Do not directly access the timer variables. They must be read and written with the timer interrupt temporarily turned off.

In each function, `timern` is the number of the timer in that particular timer array. The first `timern` is 0. To avoid conflicts within a large program spanning multiple source files, it is good practice to edit the *user_option.h* file to add #define statements to assign each timer that is used.

For example, in an application that is designed to blink two lights independently, in *user_option.h*, edit `NUSRMSTMRS` and add definitions to use in manipulating the two timers:

```
#define NUSRMSTMRS 2
#define LIGHT1TMR 0
#define LIGHT2TMR 1
```

In the application code within the user_code() function:

```
if (!getusrmstmr(LIGHT1TMR)) {
  if (light1) light1=0;
  else light1=1;
  setusrmstmr(LIGHT1TMR,500);  // 500 ms on/off times
}
if (!getusrmstmr(LIGHT2TMR)) {
  if (light2) light2=0;
  else light2=1;
  setusrmstmr(LIGHT2TMR,1200); // 1.2 second on/off times
}
```

The `light1` and `light2` variables must be defined elsewhere in the program to control the port output bits to which the lights are attached. This could be with a #define, such as:

```
#define light1 (LATCbits.LATC0)
```

There is an additional long integer timer which just counts seconds. It is read with using:

```
unsigned long getseconds()
```

The seconds timer can be synchronized by initializing the timer interrupt internal counters used for 0.1 and 1 second intervals (ticksms=100 and ticks=10). This is done, along with zeroing the seconds timer using:

```
void clearseconds()
```

TIMER_INT_ENABLED should be defined in *user_options.h* to use these functions.

The millisecond interval interrupt calls a user defined function. This can be used to add code that must run in the background of the main program. The function is declared as follows:

```
void user_timer_int(void)
```

This function is included in the interrupt handler code if `USER_TIMER_INT_ENABLED` is defined in *user_options.h*. There is a dummy `user_timer_int()` function defined in the Hamstack library, which will be linked into the program code if `USER_TIMER_INT_ENABLED` is defined and no function is explicitly defined in the program.

WATCHDOG_ENABLED should be defined in *user_options.h* to use this facility.

The Microchip 18F processors include a built-in hardware watchdog timer, which will reset the processor if the watchdog timer is not reset at a periodic interval.

The watchdog reset is performed by issuing the processor **clrwdt** instruction. The C compiler library provides a built-in function, **ClrWdt()**, which issues this instruction.

The Hamstack *config.c* file sets up the watchdog timer to timeout after a 32 second interval. The timer may be enabled and disabled through software control, and is initially disabled as the program starts execution.

If `#define WATCHDOG_ENABLED` is active in *`user_options.h`*, then

1) *watchdog.h* is included by *hamstack.h*.
2) The watchdog timer is enabled just before entering the main loop that repeatedly calls `user_code()`.
3) The watchdog timer is reset each time through the main loop, before `user_code()` is called.

When enabled, if there is a hang-up in `user_code()` due to either a hardware or software problem, then the processor internal hardware watchdog timer will reset the processor and start the entire program over again.

The functions below are used to enable, disable, and reset the watchdog timer. As these are called by the `main()` function, they are not normally needed by user code.

```
#define watchdog_reset() ClrWdt()
#define watchdog_enable() {ClrWdt(); WDTCONbits.SWDTEN=1;}
#define watchdog_disable() {WDTCONbits.SWDTEN=0;}
```

If the watchdog timer is enabled, the programmer must insure that the `user_code()` function will ALWAYS complete in less than the watchdog timer timeout interval (32 seconds). The programmer should avoid long wait loop within the code. If a long wait is required, a timer should be enabled and checked each time through the `user_code()` function, so that the `user_code()` function always completes very quickly.

If `user_init()` or `user_once()` should be protected against a hang-up by the watchdog timer, then `watchdog_enabled()` should be called in the first routine needing protection. Be cautious; if there is a long delay (>30 seconds) in these routines due to either a software wait or while waiting for some hardware to initialize, then the watchdog timer can create an infinite loop of processor resets.

SERIAL_INT_ENABLED should be defined in *user_options.h* to use these functions.

Refer to *serial.h* for a complete list of serial port functions.

Serial port buffer sizes are defined using the RXBUFSZ and TXBUFSZ constants in *user_options.h*. Default values are 100 character each.

`void` **`serial_init`**`()`
Intialization, called by `hamstack_init()` by default.

`void` **`serial_disable`**`()`
Undoes the serial port initialization to free up the processor pins for something else.

`void` **`serial_bufinit`**`()`
`serial_abort``()`
Clears and initializes the serial port buffer. Both functions are identical.

`void` **`serial_inton`**`()`
`void` **`serial_intoff`**`()`
Enables or disables the serial port interrupt. The interrupts are turned on by the Hamstack initialization if `SERIAL_INT_ENABLED` is defined in *user_options.h*, so these functions are not normally needed in an application program.

`void` **`serial_setbaud`**`(unsigned long baud)`
Sets the serial port baud rate. The default is 9600 baud. Although any value can be specified, standard values are 300, 600, 1200, 2400, 4800, 9600, 19200, 28800, 57600, and 115200. All will work successfully with a 40 or 64 MHz clock rate.

`void` **`serial_pushtxbuf`**`(unsigned char c)`
Push one character onto the serial port transmit buffer

`unsigned char` **`serial_txfull`**`()`
Returns true if the serial port transmit buffer is full.

`unsigned char` **`serial_txbusy`**`()`
Returns 1 if the serial tx buffer is not empty, and something is being sent.

`void` **`serial_putc`**`(unsigned char c)`
Sends one character to the serial port. If the serial port transmit buffer is full, this function will wait until it is not full, and then add the character to the buffer.

If that wait is unacceptable, `serial_txfull()` should be called first to determine if there is space in the buffer.

```
unsigned char serial_write(char *s)
unsigned char serial_writepgm(const rom char *s)
```

Writes the zero terminated string, `s`, to the serial port. The `serialwritepgm()` version is used for sending string constants to the serial port. This function will not complete until enough space clears in the serial port output buffer to hold all of the output string. If this behavior is unacceptable, then either the buffer should be checked for adequate room before executing this command, or one of the non-blocking serial write commands should be used. Both functions return 1 if successful, 255 on error. These functions can be used to send strings that are longer than the serial port buffer size.

For most situations in which the serial port baud rate is reasonably high, these functions will take a most 10 or 20 milliseconds to execute. If the program performance requirements can withstand this length of time, then these are the preferred serial output routines to use.

```
unsigned char serial_writenb(char *s)
unsigned char serial_writenbpgm(const rom char *s)
```

Non-blocking writes to the serial port. These functions first check the serial port buffer to determine if there is enough space free to hold the entire string to be sent. If there is, the string is sent, and the function returns 1. If there is not, then the function does nothing and returns 0. These functions will always return in less than 1 millisecond. Use caution with this routine; if the string is longer than the buffer size, then it will never return 1.

```
unsigned char serial_writenbp(char *s, char **nextchar)
unsigned char serial_writenbppgm(const rom char *s, const rom
   char **nextchar)
```

Alternate non-blocking writes to the serial port. These functions copy the string to the serial port output buffer until either the string is complete, or the buffer becomes full. If the write is complete, the function returns 0. If the write does not complete, the function returns 0 and nextchar receives a pointer to the next unsent character in the string. These functions will always return in less than 1 millisecond. These functions can be used to send strings that are longer than the serial port buffer size.

```
unsigned char serial_writebin(unsigned char *s, unsigned char len);
unsigned char serial_writebinnb(unsigned char *s, unsigned char len);
unsigned char serial_writebinnbp(unsigned char *s, unsigned char *len,
   unsigned char **nextchar)
```

Serial port write routines that can send arbitrary bytes. The number of characters specified in `len` are sent.

```
#define serial_puts(s) (serialwrite(s))
#define serial_putrs(s) (serialwritepgm(s))
```
Identical functions to `serialwrite()` and `serialwritepgm()`.

```
unsigned char serial_read(unsigned char *s, unsigned len)
unsigned char serial_gets(unsigned char * s,unsigned len)
```
Read whatever is in the serial input buffer, up to a maximum of len characters. Return the number of characters read, up to len.

```
unsigned char serial_readln(char *s, unsigned char len)
```
Returns a CR or LF terminated string up to len characters long. Does nothing and returns 0 if CR, LF, or CRLF is not in the input buffer. Return 1 if successfully receives CR,LF, or CRLF terminated input. Returns 3 if terminated because len character limit reached.

```
unsigned char serialrx_procchar(unsigned char c)
```
Optional user written function to handle incoming characters as they are received by the interrupt routine. This needs to return 1 if the character is handled by the function and should be removed from the serial input buffer. It should return 0 if the character should remain in the serial port input buffer, to be extracted with one of the other serial_read functions later. In most cases, the serial_read() function can handle special character handling requirements and should be used instead of this using this mechanism.

```
unsigned char serial_txbufspace()
```
Return the number of free characters in the transmit buffer. Using this before formatting a and writing a string to the serial port buffer, provided better performance and use of memory than using the non-blocking serial write commands without checking first.

TONE_INT_ENABLED and, optionally, TONE2_INT_ENABLED, should be defined in *user_options.h* to use these functions.

Tone generation functions are provided for the processor PWM 1 and PWM 2 outputs. A low pass filter is provided on the Hamstack Project board for the PWM 1 output. There are two independent sine wave generator for each output. For PWM1, these are tonea and toneb. For PWM2, these are tonec and toned. For most applications, only the PWM 1 output should be used for tone generation.

High level functions are documented here. The tone generator can also be controlled by direct manipulation of the tone generator control variables. See *tone.h* and *toneint.h* for details.

```
void tone_init()
void tone2_init()
```
Initialize the tone generator. Automatically included in `hamstack_init()`.

```
void toneon (unsigned int freq, unsigned char amp)
void tone2on (unsigned int freq, unsigned char amp)
```
Turn on one tone generator only for the associated PWM port.

```
void toneaon (unsigned int freq, unsigned char amp)
void tonebon (unsigned int freq, unsigned char amp)
void tonecon (unsigned int freq, unsigned char amp)
void tonedon (unsigned int freq, unsigned char amp)
```
Turn on the associated tone generator. The other tone generator on the same PWM port is not affected by this function.

```
void toneoff()
void tone2off()
```
Turn off both tone generators in the associated PWM port.

```
void toneaoff()
void toneboff()
void tonecoff()
void tonedoff()
```
Turn off the specified tone generator. This will not disable other processes, such as the CW or DTMR generator, which may turn the generator back on.

```
unsigned char tonebusy()
unsigned char tone2busy()
```
Returns 1 if either of the associated PWM tone generators is on. This will return busy if other processes driving the tone generator, such as the CW or DTMF generator is active.

```
unsigned char toneabusy()
unsigned char tonebbusy()
unsigned char tonecbusy()
unsigned char tonedbusy()
```

Returns 1 if the associated tone generator is on. This will return busy if other processes driving the tone generator, such as the CW or DTMF generator is active.

CW_INT_ENABLED should be defined in *user_options.h* to use these functions.

Default values for the CW key output pins, and tone generator PWM output, frequency, and amplitude are defined in *cw.h.* These do not need to be explicitly set in the application program if they are properly defined in this file.

CW behavioral parameters

`unsigned char` **`cwamp`**	0..255 CW sidetone amplitude
`char` **`cwweight`**	signed dot or dash weight incease in ms
`char` **`farnsworth`**	minimum character speed in WPM

CW module functions

`void` **`cw_init`**`()`

Initialize the CW module. Automatically called by *hamstack_init()*

`unsigned char` **`cw_setspeed`** `(unsigned char wpm)`

Set the CW speed. This must be explicitly called if the cwweight parameter is changed.

`char` **`cw_setpin`**`(unsigned char *latchport, unsigned char bit, unsigned char inverted)`

Define the keyer output pin. This call with the *cw.h* default values in *hamstack_init()*.

`void` **`cw_setfreq`** `(unsigned int freq, unsigned char amp)`
`void` **`cw_setport`**`(char pwmport)`

Set the CW sidetone PWM port, frequency, and amplitude. The toneb or toned generators are used, so other tones can be generator over the CW sidetone at the same time via tonea or toneb.

`void` **`cw_send`** `(char *s)`
`void` **`cw_sendr`** `(const rom char *s)`

Send a null terminated string. These return immediately; the CW send is handled in the background by the interrupt routine. `cwbusy()` should be tested before calling this or overwriting the string used in the previous call of these functions to avoid conflicts.

`void` **`cw_off`**`()`

Abort actively sending CW

`unsigned char` **`cw_busy`**`()`

Returns 1 if the CW generator is actively sending a string.

ENCODER_INT_ENABLED should be defined in *user_options.h* to use these functions.

The encoder module supports simultaneous tracking of multiple mechanical rotary encoders. It was designed for and tested with the Panasonic EVE-JBBF2020B encoder. The encoder count increments or decrements by one with each rotation by one detent position.

One encoder is defined and initialized in `hamstack_init()` if the encoder module is enabled. The default input pins for this encoder is defined in *encoder.h*. Edit *encoder.h* and *encoder.c* to support additional encoders.

`void` **`encoder_init`**`()`
Initialize the encoders. Called automatically by `hamstack_init()`

`void` **`encoder_reset`**`(unsigned char encoder, int initcount)`
Reset an encoder to a specific count.

`int` **`encoder_get`**`(unsigned char encoder)`
Get the current value of one encoder

ADC_ENABLED should be defined in *user_options.h* to use these functions.

The sample and hold capacitor internal to the A to D converter module in the processor chip is driven directly by the signal on the pin – there is no intervening buffer. This means that the source of the signal needs to be fairly low impedance to quickly drive the sampling capacitor to the correct signal level. If the signal source has more than a few kilohms input impedance, it should be buffered with an op amp.

`void` **`adc_init`**`()`

Initialize the adc's. Channels 0..3 are configured as ADC inputs in `hamstack_init()`. Edit that function to configure a different number of ADC channels.

`void` **`adc_refset`**`(unsigned int refmv)`

Set the ADC reference voltage. The only valid value for the 18F4620 is 5000 (5V). The valid values for 18F46K22 are 1024, 2048, and 4096. The valid values for the 18F46K80 are 2048 and 4096. Invalid values are not ignored. It is assumed that the ADC is configured for an external reference in that case and the provided value is used.

`int` **`adc_read`**`(unsigned char channel)`

Read the ADC raw value. It is a 10 bit integer for the 18F4620 and 18F46K22, and a 12 bit integer for the 18F46K80.

`int` **`adc_readmv`**`(unsigned char channel)`

Read the ADC value as an integer, in millivolts. The conversion is based on the reference voltage.

`char *` **`sprintmv`**`(char *s, int val)`

Format an integer as a decimal with 3 digits after the decimal point.

These are always available.

```
unsigned char eeprom_read (unsigned int adr)
void eeprom_write (unsigned int adr, unsigned char data)
unsigned int eeprom_readword(unsigned int adr)
void eeprom_writeword(unsigned int adr, unsigned int data)
void eeprom_readarr(unsigned int adr, unsigned char *d, unsigned
   char n);
void eeprom_writearr(unsigned int adr, unsigned char *data,
   unsigned char n);
```

ONEWIRE_ENABLED should be defined in *user_options.h* to use these functions.

Note that these functions are not interrupt driven. There are critical timing elements in the one-wire interface, which require brief suspension of interrupts during the one-wire communication. This causes a small glitch in sine wave tone generation.

DS1820 functions

`unsigned char` **`ds1820_init`**`(unsigned char resbits)`

This is not called called by `hamstack_init()`. The programmer must explicity call this function in `user_init()`.

`unsigned char` **`ds1820_convert`**`()`

Initiate a conversion. At full 12 bit resolution (resbits==12, the default), it will take about 1 second after calling this function before the value is ready to be read by the functions below.

`int` **`ds1820_readt`**`()`
`int` **`ds1820_readtf`**`()`
`int` **`ds1820_readtn`**`(unsigned char *device)`
`int` **`ds1820_readtfn`**`(unsigned char *device)`

Read the resulting temperature in C or F. The integer returned is 16X the temperature in degrees. The “n” versions of the routines address and read from one specific device on the one-wire bus, which is specified by an 8 byte unsigned character array, `unsigned char device[8]`. The other routines broadcast to any device on the one-wire bus, and work if there is only one temperature device attached to the bus.

One-wire interface functions

`unsigned char` **`ow_reset`**`()`
`void` **`ow_writebyte`**`(unsigned char dat)`
`unsigned char` **`ow_readbyte`**`()`

`unsigned char` **`ow_search`**`(unsigned char devices[][8], unsigned char ndevices);`

This function will search the one-wire bus for all attached devices and determine their unique 8 byte identification numbers, which will be returned in the `devices` array. A maximum of ndevices will be detected. Refer to user_code_ds1820s.c for an example of how to use this function. This function returns the number of devices found.

LCD_ENABLED should be defined in *user_options.h* to use these functions. The display configuration should be identified by defining the LCD_LINES and LCD_WIDTH paratmers in *user_options.h*. A 2 line by 16 character display is assumed if these are not defined.

These functions work with LCD character displays controlled using a single Hitachi HD44780 display controller. This includes 1, 2 and 4 line displays (note that 4 line displays with more than 20 characters per line require two of the controller chips – lcd.c would need to be modified to support this type of display).

All of the support code is in the lcd.c and lcd.h files in the hamstack directory (none of it is hidden in the precompiled hamstack library).

`void` **`lcd_init`**`()`

Initialize the LCD module. The is called by `hamstack_init()` if the module is enabled. Change the default port pin assignments by editing *lcd.h*.

`#define` **`lcd_home`**`()`

Move the cursor and next character position to the upper left display corner.

`#define` **`lcd_line2`**`()`

Move the cursor and next character position to the start of the 2nd line of the display.

`void lcd_cursorpos(unsigned char line, unsigned char pos)`

Sets the cursor to the specified line and column position on the display. The first line and column positions are specified by values of 0.

`#define` **`lcd_clear`**`()`

Clear the display and move the cursor to the upper left display corner.

`#define` **`lcd_cursorunderline`**`()`
`#define` **`lcd_cursorblink`**`()`
`#define` **`lcd_cursoroff`**`()`

Set the cursor behavior.

`#define` **`lcd_left`**`()`
`#define` **`lcd_right`**`()`

Move the cursor one character to the left or write

`void` **`lcd_setcgadr`**`(unsigned char adr)`

Sets the character generator address

```
void lcd_setdataadr(unsigned char adr)
```
Sets the display data address

```
void lcd_writecmd(unsigned char cmd)
```
Writes a command to the LCD

```
void lcd_writedata(char c)
void lcd_putc(char c)
```
Writes a data byte to the LCD

```
void lcd_puts(char *s)
void lcd_putrs(const rom char *s)
```
Writes a string of characters to the LCD

I2C_ENABLED should be defined in *user_options.h* to use these functions. Some processor support a second i2c interface. To use functions supporting the second i2c interface, **I2C2_ENABLED** should also be defined.

I2c is a two wire serial interface commonly used to communicate with small memory and sensor chips. Support for i2c EEPROM's is included in i2c.h. Other chips are supported in separate modules.

The low level i2c functions are declared in i2c.h and are not documented here. Please contact Sierra Radio Systems for further information if you want to use these to write your own drivers for other i2c chips.

These functions work with the 24LCxxx series of serial EEPROMs. The device address parameter, `devadr`, is in the range of 0..7 and is set via the adr pins on the 8 pin EEPROM devices.

```
unsigned char i2c_eeprom_wrbyte(unsigned char devadr, unsigned
   int adr, unsigned char data);
```
Write one byte at address `adr` in the eeprom.

```
unsigned char i2c_eeprom_rdbyte(unsigned char devadr, unsigned
   int adr, unsigned char *data);
```
Read one byte at address `adr` in the eeprom.

```
unsigned char i2c_eeprom_rdbytes(unsigned char devadr, unsigned
   int adr, unsigned char len, unsigned char *data);
```
Read an array of bytes starting at address `adr` in the eeprom.

```
unsigned char i2c_eeprom_wrbytes(unsigned char devadr, unsigned
   int adr, unsigned int len, unsigned char *data);
```
Copy data from an array in RAM to the eeprom

```
unsigned char i2c_eeprom_wrbytespgm(unsigned char devadr,
   unsigned int adr unsigned int len, rom unsigned char *data);
```
Copy data from an array in program ROM to the eeprom

```
unsigned char i2c_eeprom_setbytes(unsigned char devadr, unsigned
   int adr, unsigned int len, unsigned char data);
```
Copy from one location in the eeprom to another

```
unsigned char i2c_eeprom_cpy(unsigned char devadr, unsigned int
   dest, unsigned int src, unsigned int len);
```
Copy from one location in the eeprom to another

I2C_ENABLED should be defined in *user_options.h* to use these functions.

The Honeywell HMC6352 is an i2c interface magnetic compass sensor. It directly reads out magnetic compass heading in tenths of a degree. A subset of the functions developed to interact with this sensor are documented here. See the hmc6352.h file and the HMC6352 datasheet for more complete details.

No initialization is required.

```
int compass_read(void);
```
Read the compass heading, in tenths of a degree.

```
unsigned char compass_calstart(void);
unsigned char compass_calend(void);
```
These are used to do a compass calibration. This automatically compensates for differences in offset and gain of the two internal magnetic field sensors used to determine direction. To do a calibration, issue the start command, rotate the sensor through at least 360 degrees, then issue the end command. The sensor must be held level during this operation.

SHIFTI_ENABLED and/or **SHIFTO_ENABLED** should be defined in *user_options.h* to use these functions.

`void` **`shifti_init`**`()`
`void` **`shifto_init`**`()`
Initialization, automatically called by `hamstack_init()` if the modules are enabled. Change the default port pin assignments by editing *shift.h*.

`void` **`shiftin`**`(unsigned char *dat, unsigned char nbytes)`
`void` **`shiftout`**`(unsigned char *dat, unsigned char nbytes)`
Shift multiple bytes from or to a daisy-chained string of shift registers. In both cases, dat[0] refers to the shift register closest to the processor.

There is no #define switch in user_options.h associated with this module. These functions are incompatible with simultaneous use of the tone generators on the same PWM port. The PWM timer initialization is done in hamstack_init() for both tone generation and for these functions.

`void` **`pwm1_init`**`(void)`
`void` **`pwm2_init`**`(void)`

Initialize the PWM pins for PWM output. These functions are not called by `hamstack_init()`, and must be included in `user_init()`.

`void` **`pwm1`**`(int outval)`
`void` **`pwm2`**`(int outval)`

Set the PWM output to the desired output value, in the range from 0 to 1023. If the Hamstack project board is used, there is an oboard low pass filter attached to PWM1 with an appropriate rolloff for use with the default PWM pulse rates.

DTMF_INT_ENABLED should be defined in *user_options.h* to use these functions.

`void` **`dtmf_init`**`(void)`
Initialize the DTMF output module. This function is automatically called by `hamstack_init()` if the module is enabled.

`unsigned char` **`dtmf_busy`**`(void)`
Returns 1 if the DTMF module is busy sending a string of DTMF characters. Calling `dtmf_puts()` or `dtmf_putrs()` before this is clear will return 0 and do nothing.

`void` **`dtmf_off`**`(void)`
Halts any ongoing DTMF output.

`void` **`dtmf_char`**`(char c)`
Turns on the tone generator with the character, c. It stays on until it is turned off with `dtmf_off()`. Valid characters are 0..9, A..D, T (dial tone), and Z (busy tone).

`unsigned char` **`dtmf_puts`**`(char *s)`
`unsigned char` **`dtmf_putrs`**`(const rom char *s);`
Write a string of characters to DTMF output.

The following are variables used to adjust the behavior of the DTMF functions. They can be directly changed by the application program at any time.

`unsigned char` **`dtmfontime`**
Time in milliseconds that each character is turned on. The default is 125 ms.

`unsigned char` **`dtmfofftime`**
Time in milliseconds between characters. The default is 75 ms.

`unsigned char` **`dtmfamp`**
Amplitude of each DTMF tone in the tone pair. Valid values are 0..127. Values greater than 127 will result in severe distortion. The default is 127.

`unsigned char` **`dtmfpwmoutput`**
Flag to tell which PWM port to be used for the tone output. Valid values are 1 and 2. The default is 1.

In addition, the DTMF output buffer size is defined by **DTMFBUFFERSIZE** in *dtmf.h*. Character strings output with `dtmf_puts()` or `dtmf_putrs()` must be shorter than this value. The default is 30.

Notes and Troubleshooting

Failure to Compile Examples

If the examples do not successfully compile, the likely reason is a problem with the compiler installation. When the C18 compiler is updated from an older installation, the pointers to directories where various files needed to build the program may not be updated correctly. We have not had a problem with a fresh installation of C18 v3.38. This problem has occurred when upgrading an earlier installation to v3.38.

Earlier versions of the compiler were installed to the location c:\mcc18. Newer versions (starting at v3.37), are, by default, installed in a version specific location in C:\Program Files or C:\Program Files (x86). For example, version 3.38 is installed to C:\Program Files (x86)\Microchip\mplabc\v3.38. In doing an update from a C:\mcc18 to a c:\Program Files located installation, pointers to some locations may not get updated correctly. This can cause various error messages during compilation. Some are missing file messages. In other cases, an old version of the processor specific .h file is included and a message appears because one of the bit definitions for a 18F4620 register is only defined in the newer versions of the file.

Here are the things to look for:

1. MCC_INCLUDE environment variable.

   To find the Windows environment variable settings:
   a. Go to Start, then Control Panel
   b. Then click either System, or Performance and Maintenance then System, or System and Security then System, depending on which version of Windows you have.
   c. Then click Advanced or Advanced System Settings
   d. Then click Environment Variables

   The environment variable MCC_INCLUDE must point to the location of the include directory for your installation of C18. For the old compiler location, that is C:\mcc18\h. For the default v3.38 installation, that is C:\Programs Files (x86)\Microchip\mplabc\v3.38\h. The (x86) is not present for Windows XP. There may be two instances of this variable, under both the user and system lists of environment variables.

2. Default Search Paths and Directories

   Default search paths can be set by traversing the following menu tree within MPLab: /Project/Set Language Tool Locations/Microchip C18 Toolsuite. The values for each of the entries should be as follows:

a. Output Directory – blank
b. Intermediates Director – blank
c. Include Search Path – blank
d. Library Search Path – set to library location for compiler installation.
   Eg. C:\Program Files\Microchip\mplabc\v3.38\lib
e. Linker-Script Search Path – blank

3. Executable Locations

These are set by traversing the following menu tree withing MPLab: /Project/Set Language Tool Suite/MPLab C18 Compiler. Verify that these are ALL set to the compiler installation directory. For the default v3.38 installation location that is C:\Program Files\Microchip\mplabc18\v3.38\bin:

a. C:\Program Files\Microchip\mplabc18\v3.38\mpasm\mpasmwin.exe
b. C:\Program Files\Microchip\mplabc18\v3.38\bin\mplink.exe
c. C:\Program Files\Microchip\mplabc18\v3.38\bin\mcc18.exe
d. C:\Program Files\Microchip\mplabc18\v3.38\bin\mplib.exe

If your installation uses a different directory than the installation we used to create the project, then a message may pop up asking which of a couple of versions of the executable you want to use. You should select the first of the two options, which is the location correct for your installation (as opposed to the installation on the machine where the project was first created).

4. Project Build Directories

These options set project specific exceptions to the defaults. The values in the Hamstack example project files should not need to be changed. Check these by navigating the menu tree to /Project/Build Options/Project, then select the Directories tab. The "Assemble/Compile/Link in the project directory box should be checked. Directory entries should be as follows:

a. Output Directory – blank – puts the output hex file in the project directory.
b. Intermediary Directory
   .\object
c. Include Search Path
   .\hamstack
d. Library Search Path – blank
e. Linker-Script Search Path – blank

We could not get the programs to link properly under MPLab 8.63 with C18 v3.37.01 unless the Project/Build Options/Project Directories tab was modified to add an explicit entry under Library Search Path to the compiler lib directory at C:\program files\Microchip\mplabc\v3.37.01/lib. We recommend installing MPLab 8.70 and C18 v3.38 or later.

Linker error message about inconsistent target processor

The libraries have been compiled for multiple processors (18F4620 and 18F46K22 at the time of this writing). The processor is designated in the name of the .lib file in the hamstacklib subdirectory. The hamstacklib.lib file needs to be a copy of the appropriate file for the processor you are compiling the rest of the program for. The default examples directory hamstacklib.lib is a copy of hamstacklib_18F4620.lib. There will be an error message during compile time if the target device is different than that of the library. To complie for the 18F46K22, change the processor in the /Configure/Select Device menu, and copy the hamstacklib_18F46K22.lib file to hamstacklib.lib.

Opening a file from the wrong directory

When using the File/Open or Add Files menus in MPLab, it opens in the same directory in which is was last used – not necessarily the same directory as the current project. This can cause huge problems when it is not noticed, and different versions of the same files, with the same file names are present in different directories. Be careful to check this whenever adding new files to a project.

Multiple warning messages during compilation about one of the optimization features not being available

Some of the advanced compiler optimizations which generate smaller and faster code are not available in the free version of the C18 compiler. If the project options are set to enable these when you are using the free compiler version, then warnings will be generated. An example is:

WARNING: This version of MPLAB C18 does not support procedural abstraction. Procedural abstraction will not be run.

This will not prevent the compiler from generating code and the program from working. You can eliminate the message by turning off these optimizations. Starting with the Project menu item at the top of MPLab, you can do this by going to: Project/Build Options/Project, click on the MPLab C18 tab, select Optimization under Categories, and check the Disable box.