ECE 486 DSP Configuration for the STM32L476G-Discovery Demonstration Board

Don Hummels

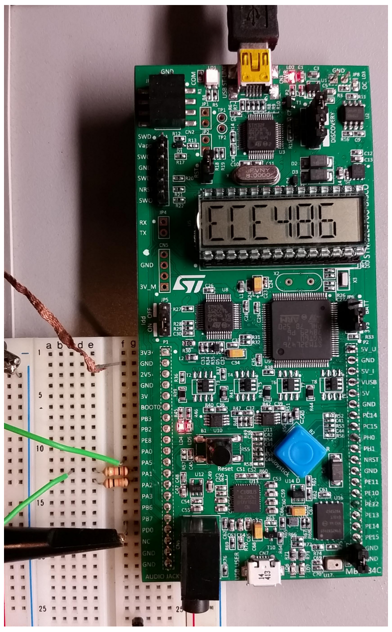
Electrical and Computer Engineering
University of Maine

December 28, 2017

Contents

Abstract

This document describes the use of the STM32L476-Discovery evaluation board to support labs in ECE 486 Digital Signal Processing.



Caution

To prevent damage to the board, it is recommend that all external function-generator connections (PA1 and PA2) be made through a 10 kΩ resistor, limiting currents to a few milliamps (even for inadvertently large function generator settings) !!!

Figure 1: STM32L476G-Discovery Board connected for test. The P1 connector provide the functionality needed for ECE-486.

1 Background

STMicroelectronics has produced a low-cost evaluation board to showcase their STM32L4xx line of processors. The STM32L476G-Discovery board features an STM32L476VG microcontroller, which includes 1MB of Flash memory, 128 kB of user RAM and a floating point co-processor. The microcontroller includes multiple ADC and DAC peripherals, as well as a rich host of other peripherals (DMAs, Timers, USART, and support for I^2C, SPI, USB-OTG, and a host of other acronyms). An external USB interface allows debugging and programming. The development board includes user-controlled LEDs, a small LCD display, a reset pushbutton and joystick, a digital MEMs microphone, gyroscope, compass, , and an external audio output DAC.

In ECE 486, the board is configured to allow students to perform signal processing experiments in real-time. Two 12-bit ADCs are configured to sample an input waveform. The sampled data is passed to a user for processing. The calculated output waveform is then streamed to a pair of output DACs so that the results may be viewed on an oscilloscope. All timing and DMA data transfers are handled via an ECE 486 library. This document provides the hardware and software interface necessary to use this library.

2 Connecting to the Board

The ECE 486 functionality is obtained through the 20 pins P1 extension connector on the STM32L476G-Discovery board. The connector may be plugged directly into a prototyping board to allow easy access to the analog and digital inputs and outputs. Figure 1 shows a photo of correctly configured board under test. Pin assignments for the 20-PIN P1 connector are discussed further in Section 6.

2.1 How to not break the board

The board may be safely powered by connecting the discovery board to a PC through the "type A to mini-B" USB connector CN1. (This connector is also used to program and debug software on the board.)

Care should be taken when applying external signals to any of the pins of the board, since there is limited protection on the development board. Signals outside of the 0-3 V power supply range can potentially damage the processor. All GPIO pins being used do contain clamping diodes on the processor, but protection diodes typically tolerate only small amounts of current. Function generator connections are particularly problematic, since it's easy to leave the generator on when power is removed—and it's easy to accidentally apply large (or negative) voltages. *A* $10\ k\Omega$ *protection resistor is recommended for all analog input signals*.

2.2 Suggested board modification

The default configuration of the STM32L476G-Discovery board is not optimal for the DSP labs for this course. Significant improvements in performance is possible by reconfiguring a few jumpers.

2.2.1 Improving DAC and ADC performance

For ECE 486, pins PA1 and PA2 are used to interface to the on-chip ADCs, and PA5 and PA3 are used to access the on-chip DACs. (Technically, PA3 is actually the output of an opamp, which is being used to buffer the DAC channel 1 output to allow access from the connector.)

Unfortunately, pins PA1, PA2, PA3, and PA5 are also connected to the Joystick "left", "right", "up", and "down" buttons. A large 100 nF capacitor is attached to these lines on the discovery board to act as a hardware de-bounce circuit. These capacitors will severly limit the bandwidth of the analog signals on the required pins.

The Joystick and de-bounce capacitors may be isolated from the input/output pins by removing the 0Ω resistors R52, R54, R56, and R58 on the discovery board. Removing these resistors will restore the bandwidth for the PA1, PA2, PA3 and PA5 GPIO lines, but will disconnect the directional inputs from the joystick. (The joystic "center" input, connected to PA0 will continue to operate, and is used in the software library as a user pushbutton input.)

The options are:

- *Remove R52, R54, R56, and R58*: With the Joystick disconnected, you'll be able to use the available bandwidth for both the ADCs and DACs. You will no longer have access to the directional inputs from the Joystic.
- *Do Nothing: Leave R52, R54, R56, and R58 populated*: Your board will operate, and you can access the joystick directional inputs if desired. You can still develop and debug code for this class, but the bandwidth of the analog signals you measure will be severly limited by the joystic capacitors. If the ADC inputs are connected through 10 kΩ resistors, your ADC inputs will be passed through a lowpass filter with a bandwidth of about 160 Hz. The DACs will be unable to drive the output pins at speeds much above 1 ksample/sec.

2.2.2 Improving clock accuracy and stability

By defaut, the STM32L476G processor on the discovery board does not have the crystal reference oscillator populated. To use the processor "out-of-the-box", the processor's internal RC oscillator (MSI) can be used to provide a reference clock source. However, the resulting clock signals will not be terribly accurate, and will drift over time. The drift will become most notable for labs in which multiple (non-synchronized) development boards are sharing signals with each other.

The discovery board does, however, have a crystal reference oscillator that is used to support the *other* ST-LINK MCU for communication with a host computer. By changing a few jumpers, this 8 MHz reference signal may be shared with the STM32L476G processor, significantly improving the clock accuracy.

Here are some configuration options:

- To use the 8 MHz crystal reference signal from the ST-LINK MCU STM32F103CBT6:
  - Discovery modifications: SB18 closed, SB22 opened, R89 not fitted.
  - Specify `HSE_EXTERNAL_8MHz` in the `initialize_ece486()` call.
- To use the internal RC Oscillator:
  - Leave Discovery default configuration: SB18 opened, SB21 and SB22 closed.
  - Specify `MSI_INTERNAL_RC` in the `initialize_ece486()` call.

- To use an externally generated clock:
  - Leave Discovery default configuration: SB18 opened, SB22 closed, R89 not fitted.
  - Drive an 8 MHz external clock signal on PH0 (pin 9) through the P2 header.
  - Specify `HSE_EXTERNAL_8MHz` in the `initialize_ece486()` call.

3 Writing code

For ECE 486, a gcc cross-compiler is used to compile C code for the board. Writing code for this board is no different from writing any other C code under Linux. Bring up your favorite editor (e.g. kate, vi, nedit, gedit, nano, etc.), and just start writing code! No specific integrated development environment (IDE) is required.

4 Software interface

Programs for real-time signal processing should initialize the processor using a single call to the `initialize_ece486()` function before any other processing. This call configures the GPIO terminals of the processor, and then waits until the blue USER push-button is pressed (Joystic "center") while the red and green LEDs blink. (This idle state is returned to when the black RESET button is pressed, allowing the board to be reprogrammed without error, we hope.)

When the USER push-button is pressed, the `initialize_ece486()` function configures the on-chip ADC and DAC so that blocks of memory are continually transferred from the ADC and written to the DAC (at a desired sample rate) while the processor is working on other tasks (such as manipulating the signals). In a typical signal processing application, one block of data is requested from the ADC, processed by the processor, and the resulting output waveform is transferred back to DAC output buffer. When a new block of data is requested from the ADC, the processor remains idle until the ADC interface competes filling the requested block and the returns the data for further processing.

Two interface functions `getblock()` and `putblock()` are provided to enable a user to easily transfer blocks of data to and from the DACs and ADCs. The user must allocate any required memory to store the ADC input samples or the DAC output samples. (Similar stereo input/output routines are provided by `getblockstereo()` and `putblockstereo`.)

Most real-time signal processing programs will access the following functions:

void setblocksize(int n_samples) :
: Optionally, the user may call `setblocksize()` to determine the size of the data block that will be used in later calls to `getblock()` or `putblock`. If `setblocksize()` is not called, a default value "`DEFAULT_BLOCKSIZE`" will be used. Using a larger block size may result in more efficient code, while using a smaller block size will reduce the latency of the system.

void initialize_ece486(sample_rate_select, input_mode, output_mode, clk_ref) :
: The `initialize_ece486()` function is called once, at the beginning of program execution to configure the ADCs, DACs, DMAs, processor clocks, etc.

Valid values for `sample_rate_select` are defined by including `ece486.h` and include `FS_48K`, `FS_24K`, and `FS_8K` to request sample rates of 48, 24, and 8 ksps respectively. (Similar constants are available for 2, 4, 5, 8, 10, 16, 20, 24, 25, 32, 40, 48, 50, 64, 80, 96, 100, 200, 400, and 500 ksps.)

Similarly, valid values of `input_mode` are `MONO_IN` and `STEREO_IN` to configure whether a single ADC (PA1) or both ADCs (PA1 and PA2) are used. Valid values of `output_mode` are `MONO_OUT` or `STEREO_OUT` to configure whether a single DAC (PA5) or both DACs (PA5 and PA3) are used.

Valid values for `clk_ref` include `MSI_INTERNAL_RC` and `HSE_EXTERNAL_8MHz`. An "out-of-the-box" discovery board is configured to use the internal MSI RC reference oscillator (which can be unstable, and should be expected to drift over time). Specifying `MSI_INTERNAL_RC` selects this option. By modifying a few jumpers, you can utilize the 8 MHz crystal reference clock that is used by the ST-LINK MCU that is also on the development board for communication with the host. If your board is modified, specify `HSE_EXTERNAL_8MHz` to use this more accurate clock. See section 2.2.2 for the required modifications.

int getblocksize(void) :
: `getblocksize()` returns the number of samples that `getblock()` will produce, and the number of samples that must be provided to `putblock()`. This is useful for allocating the any arrays needed to store or

manipulate the waveforms (especially when `setblocksize` was not called). In the interface description below, a working array (named "`working`") is used to store data provided by getblock() and pass output data to putblock(). Storage for this working array *must be allocated by the interface user*.

int getsamplingfrequency(void) :

`getsamplingfrequency()` returns the best guess at the actual sampling rate being implemented (in samples/second). The actual sample rate may be slightly different from the requested rate, depending upon the capabilities of the on-board timers that generate the clocks. Note that the time reference being used for the processor is not crystal reference — it is a factory calibrated on-chip RC time reference. Frequency errors on the order of a percent or so would not be unexpected.

void getblock(float *working) :

`getblock()` waits until the ADC (connected to PA1) has complete acquiring a block of input samples. These samples are then transferred into the caller's `working` array and returned. The caller must allocate the memory required for the returned result array.

Returned samples are stored as single-precision (type float) values ranging from -1.0 (corresponding to input voltages near 0 V) to $+1.0$ (corresponding to the maximum input voltage, near 3 V). The actual input voltage corresponding to the i^{th} sample is approximately $v(iT_s) \approx 1.5 + (working[i] * 1.5)$.

void putblock(float *working) :

putblock() transfers the caller's working array for output via the DAC (connected to PA5).

Output samples should range from -1.0 (for a minimum DAC output, near 0 V) to $+1.0$ (for the maximum output value near 3.0 V).

In practice, `initialize_ece486()` and `getblocksize()` are called once, memory is allocated, and then the `getblock()` and `putblock()` functions are repeatedly called as blocks of input samples are processed to form the output waveform.

If the board has been configured to use stereo inputs or outputs, modified data transfer functions are provided with additional arrays for the two ADCs/DACs. The two analog input pins are PA1 and PA2, and the DAC outputs are accessed through PA5 and PA3.

void getblockstereo(float *input1, float *input2)

void putblockstereo(float *output1, float *output2)

A few other utility functions are provided which may prove useful to debug or measure program performance. A GPIO digital io pin provides one method of indicating program status, or timing events (via an oscilloscope) within the program flow. The pin is configured as PD0 on the discovery board.

DIGITAL_IO_SET();

DIGITAL_IO_RESET();

DIGITAL_IO_TOGGLE();

The 6-character LCD display is also initalized, and output may be written to the display using the STM HAL interface library:

BSP_LCD_GLASS_DisplayString("HELLO");

BSP_LCD_GLASS_ScrollSentence("This is a test",SCROLL_SPEED_MEDIUM);

(SCROLL_SPEED_LOW and SCROLL_SPEED_HIGH are also supported.)

LEDs are also controled using the HAL library. The board has two user-programmable LEDs: Red (LED4), and Green (LED5). The `initialize_ece486()` function resets the red LED, and sets the green LED on startup to indicate normal running conditions. Error conditions will set the red LED. To change the state of the green LED, use:

BSP_LED_Toggle(LED5);

BSP_LED_On(LED5);

BSP_LED_Off(LED5);

5 Compiling and Running

5.1 Makefile

Cross-compiling code with multiple libraries requires relatively complex gcc commands, so a Makefile is provided that handles telling gcc how it should compile code, and where to find libraries. Edit the Makefile to include your source code (following the comments in the Makefile). Running “make” should create two output files ”myexe” and “myexe.bin”, where “myexe” is the executable name that you provide in the Makefile.

When the board is connected to a host computer via the USB port, the host should detect a mass-storage device, and a serial port. The board is “flashed” by dragging the “myexe.bin” file to mass-storage device. Alternatively (and less reliable?), reset the discovery board to disable the DMA activity. Then typing “make flash” should ”flash” the board by writing the ”myexe.bin” binary image to the development board.

Any “printf()” output generated by code on the discovery board may be accessed through the serial port accessed through the USB connection. Connect a serial terminal (115200 8N1) on the host to the serial port in order to access the output.

5.2 Flashing and running image

1. Connect a USB cable from the host to the CN1 connector on the Discovery board.
2. Reset the STM32L476 board (hit the black push-button)
3. On the host, run:

   ```
   make flash
   ```

4. Reset the STM32L476 board to restart the program back to the idle state. The green and red LEDs should be flashing.
5. Continue program execution beyond the “`initialize_ece486()`” function by pressing the blue USER push-button. The green LED should stay lit, indicating that the image is executing. Error conditions are indicated by lighting the red LED.

5.3 Running a debugger

A symbolic debugger may be used to trace program execution, set breakpoints, and examine variable values. The “gdb” debugger is supported as follows:

1. Recompile the code with debugging options, and flash the compiled image to the processor.

   ```
   make clean
   make debug
   make flash
   ```

2. In a separate window on the host, run the `st-util` to establish the communications link between the development board and the host (Port :4242).

   ```
   st-util
   ```

3. You may run gdb directly using the ELF file (called “myexe” in Section 5.1). After running the ARM compiled gdb command (at the “(gdb)” prompt) specify the target development board port provided by `st-util`.

   ```
   arm-none-eabi-gdb myexe
   >>>>> (Output lines from gdb... wait for the (gdb) prompt) <<<<<
   (gdb) target extended :4242
   ```

   You may now continue using gdb commands to trace program execution.

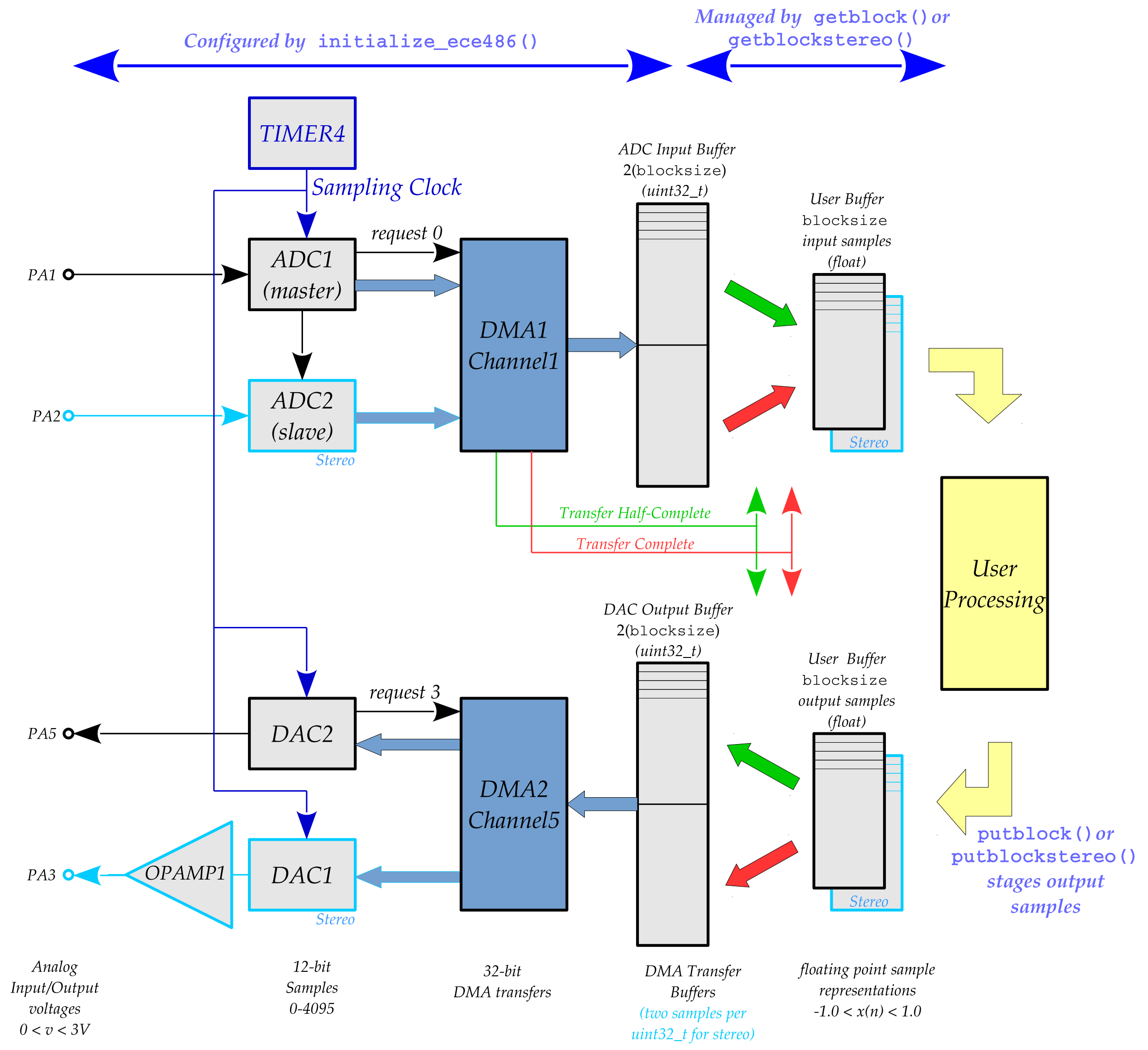


Figure 2: Peripheral configuration used to stream analog inputs to the user program, and user data output samples back to the DACs for analog outputs.

4. As an alternative to using the raw gdb commands, the “ddd” debugger provides a graphical front-end to the gdb debugger. To use ddd, you must specify the path to the ARM-compiled gdb debugger on the command line, and then specify the target on the “(gdb)” prompt within the ddd window:

```
ddd --debugger arm-none-eabi-gdb myexe
>>>>>> Graphical debugger should start                    <<<<<
>>>>>> Look for the gdb command window with the (gdb) prompt <<<<<
(gdb) target extended :4242
```

6 Hardware Interface

Figure 2 provides an overview of the STM32L476G peripherals used to stream analog input samples from the ADCs and to the DACs.

Pin assignments for the 20-pins accessed through the ECE 486 connector on the Discovery board P1 connector are summarized in Table 1.

Table 1: Discovery board P1 Pin Assignments

P1 Pin	P1 Label	ECE 486 Function
1	3V3	Power (3.3 V)
2	GND	Ground
3	2V5	Power (2.5 V)
4	GND	Ground
5	3V	Power (3 V)
6	BOOT0	
7	PB3	3V3_REG_ON: Tied to 3.3V Regulator via 4.7 kΩ
8	PB2	Red LED
9	PE8	Green LED
10	PA0	GPIO Input: Connected to Joystic "center"
11	PA5	Primary DAC Output(MONO and STEREO)
12	PA1	Primary ADC Input (MONO and STEREO)
13	PA2	Second ADC Input (STEREO only)
14	PA3	Second DAC Output via opamp (STEREO only)
15	PB6	I2C1_SCL
16	PB7	I2C1_SDA
17	PD0	EXT_RST: GPIO Digital Output
18	NC	
19	GND	Ground
20	GND	Ground

6.1 Analog Inputs/Outputs

The STM32L47607VGT6 microcontroller includes two on-chip 12-bit ADCs, and two on-chip 12-bit DACs which are accessed through the PA1, PA2, PA5, and PA3 terminals. In MONO modes, only the PA1 or PA5 terminals are active. The analog inputs and outputs should be in the $0 - 3$ V range. These analog inputs and outputs are clamped to the supply rails within the microcontroller using protection diodes, but currents in or out of these terminals must be limited to no more than 5 mA. Inadvertent damage to the microcontroller may be avoided by making all function generator connections to the PA1 or PA2 terminals through a 10 kΩ resistor.

6.2 Use of GPIO Pins

A digital output is configured on pin PD0. Users can set the values of these output using statements such as:

```
DIGITAL_IO_SET();
DIGITAL_IO_RESET();
DIGITAL_IO_TOGGLE();
```

The pins may be set/cleared to enable timing of segments of code using an oscilloscope.

6.3 Accessing the USER Pushbutton

The Blue USER pushbutton (Joystick "center") is also configured by the `initialize_ece486()` function. Button presses may be detected in software by periodically monitoring the global variable `KeyPressed`. Normally `KeyPressed` has a value of "`RESET`". The value changes to "`SET`" on every button press, and remains at this value until reset by the user's software. A typical code segment is given below:

```
#include "ece486.h"
...
if (KeyPressed) {
  ...  // Take some action on button presses
  UserButtonPressed = RESET;   // Allows detection of the next press
}
...
```

7 Sample Program

```c
/*
 * Example program to illustrate the use of the ECE 486 interface.
 *
 * An input waveform is copied to the output DAC.  The waveform is also
 * squared and streamed to the second DAC output.  Each
 * USER button press inverts the signal on the original DAC.
 *
 * The use of printf()
 */

#include "stm32l4xx_hal.h"
#include "stm32l476g_discovery.h"

#include "ece486.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

extern FlagStatus KeyPressed;   // Use to detect button presses

int main(void)
{
  int nsamp, i;
  float *input, *output1, *output2;
  static float sign=1.0;
  static int button_count = 0;

  char lcd_str[8];

  /*
   * Set up ADCs, DACs, GPIO, Clocks, DMAs, and Timer
   */
  initialize_ece486(FS_50K, MONO_IN, STEREO_OUT, HSE_EXTERNAL_8MHz);

  /*
   * Allocate Required Memory
   */
  nsamp = getblocksize();

  input = (float *)malloc(sizeof(float)*nsamp);
  output1 = (float *)malloc(sizeof(float)*nsamp);
  output2 = (float *)malloc(sizeof(float)*nsamp);

  if (input==NULL || output1==NULL || output2==NULL) {
    flagerror(MEMORY_ALLOCATION_ERROR);
    while(1);
  }

  /*
   * Normally we avoid printf()... expecially once we start actually
   * processing streaming samples.  This is here to illustrate the
   * use of printf for debugging programs.
   *
   * To see the printf output, connect to the ST-Link serial port.
   * Use: 115200 8N1
   */
  printf("Starting execution using %d samples per input block.\n",nsamp);

  /*
   * Infinite Loop to process the data stream, "nsamp" samples at a time
   */
  while(1){
    /*
     * Ask for a block of ADC samples to be put into the working buffer
     *   getblock() will wait here until the input buffer is filled...  On return
     *   we work on the new data buffer... then return here to wait for
     *   the next block
     */
    getblock(input);

    /*
     * signal processing code to calculate the required output buffers
     */

    DIGITAL_IO_SET();    // Use a scope on PD0 to measure execution time
    for (i=0; i<nsamp; i++) {
      output1[i] = sign * input[i];
      output2[i] = input[i]*input[i];
    }
    DIGITAL_IO_RESET(); // (falling edge....  done processing data )

    /*
     * pass the processed working buffer back for DAC output
     */
    putblockstereo(output1, output2);

    if (KeyPressed) {
      KeyPressed = RESET;
      sign *= -1.0;              // Invert output1

      /*
       * On each press, modify the LCD display, and toggle an LED
       * (LED4=red, LED5=green) (Red is used to show error conditions)
       *
       * Don't be surprised when these cause a Sample Overrun error,
       * depending on your sample rate.
       */
      button_count++;
      sprintf(lcd_str, "BTN_%2d", button_count);
      BSP_LCD_GLASS_DisplayString(lcd_str);
      BSP_LED_Toggle(LED5);
    }
  }
}
```