Appendix

B

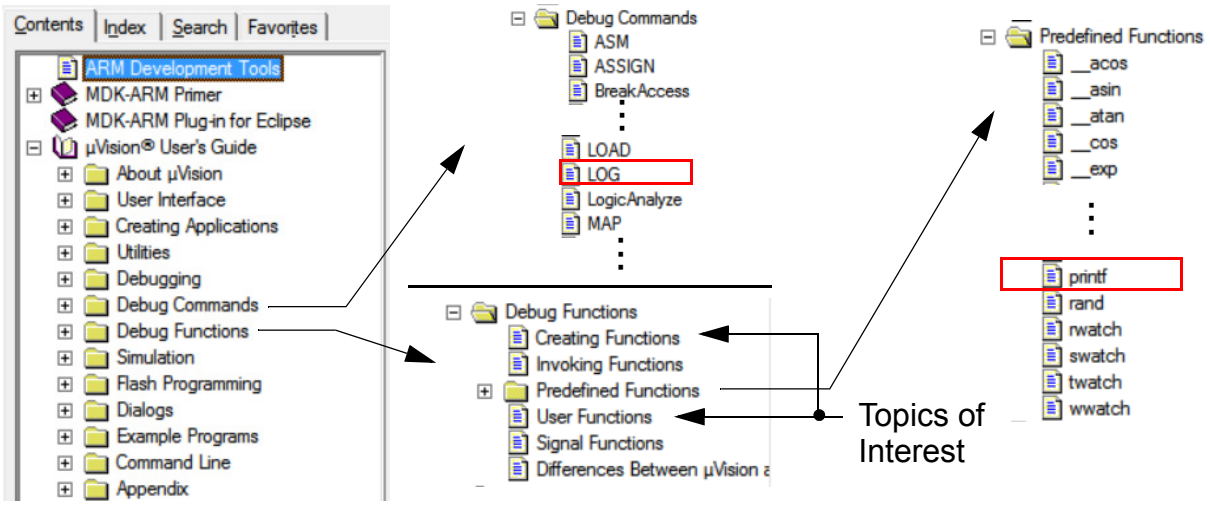
Using the Keil Debugger Command Line

Introduction

Much can be said about the command line capabilities of the Keil debugger. In this appendix I will focus on writing user functions and using the log capability to log what is seen in the command output display to a text file. Many other capabilities exist and may at some point be described in this appendix.

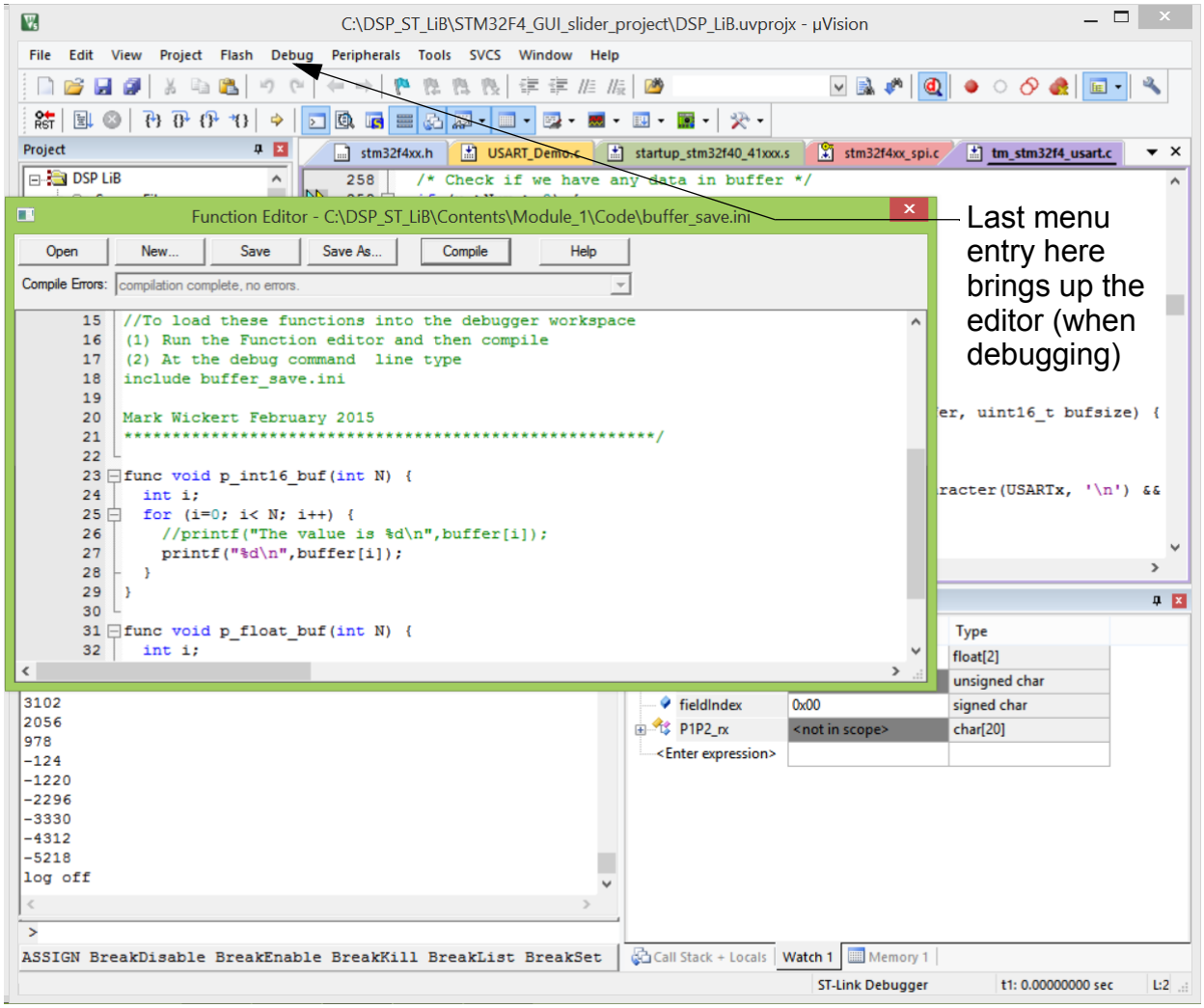
Getting Started via µVision Help

The help system contains a vast quantity of information on all of Keil, you just need to jump in and try to find it. For help on the command line debugging drill down as follows:



Creating User Functions

- To create a user defined function you can use a text editor, or more conveniently use the build-in Function Editor, which you access from the Debug menu during a debug session:



- Debugger functions follow closely C syntax, but the language support is limited
  - For example functions cannot pass arrays or pointers
  - You cannot directly read and write to a file
  - You can however access all of the variables in the debug

session, e.g. scalar variables and arrays

  - Multiple functions can be contained in a use function file (file extension is `.ini`)

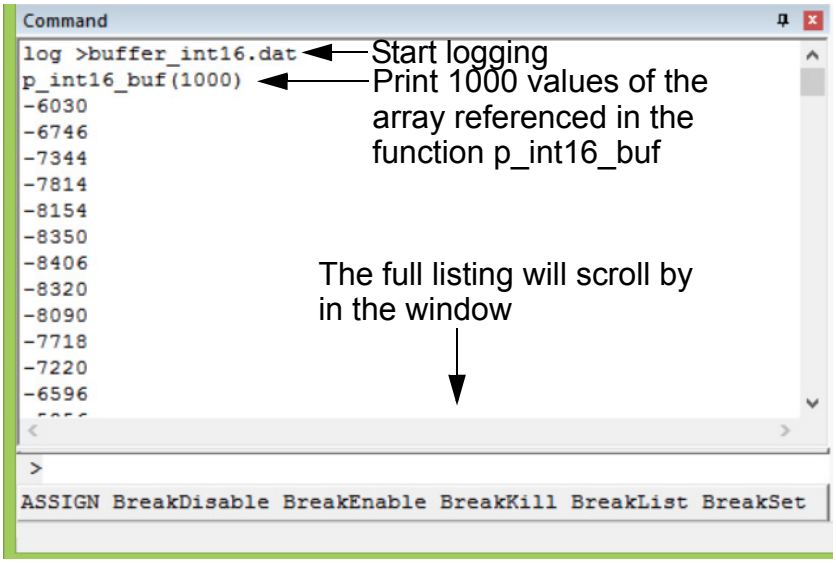
- Fortunately you have access to both Debug Commands and Debug Functions
  - A very useful Debug Function is `printf()`, which behaves very much like the C version
  - A very useful Debug Command is `log`
  - **Note**: The `save` command can be used to write a block of memory to a file, but the format is *Intel hex*, which requires further post processing to be used in external analysis
- Appendix A talks about the use of the serial port for sending debug data directly to a terminal or perhaps a GUI app running on the host
- By writing a User Function you have the further option of pre-processing any user variables and sending them to the Command output window as result of your issuing a debug command
  - This can only be done within a debug session, but your program may *running* or *stopped*
  - **Note**: You can also write to variables
- Command line debug sessions can also be logged to a text file for later analysis

- `log >fname` //starts logging in the file fname
- `log off` //stops the logging

- A sample .ini file which contains two function is shown below:

```
/*******************************************************
Write array contents using debug command line
*******************************************************
//For int16_t buffer use:
log >fname
p_int16_buf(1000)
log off

//For float or float32_t buffer use:
log >fname
p_float_buf(1000)
log off

*******************************************************
//To load these functions into the debugger workspace
(1) Run the Function editor and then compile
or
(2) At the debug command line type
include buffer_save.ini

Mark Wickert February 2015
*******************************************************/

func void p_int16_buf(int N) {
      int i;
      for (i=0; i< N; i++) {
            //printf("The value is %d\n",buffer[i]);
            printf("%d\n",buffer[i]);
      }
}

func void p_float_buf(int N) {
      int i;
      for (i=0; i< N; i++) {
            //printf("The value is %6.4f\n",buffer[i]);
            printf("%6.4f\n",buffer[i]);
      }
}
```

- A summary of the steps needed to compile/include the .ini file in your debug session is given at the top of the file
- The commands needed to create a log file of a 1000 point `int16_t` variables named `buffer`, is also given
- A screen shot of the Command window output following the above steps is shown below:



- A great tool for analyzing the captured results is Python with *pylab* running
- The IPython notebook provides a very nice environment for working with the data contained in the log file and creating post analysis plots
- A few screen shots from the available IPython notebook are given below
  - **Note**: The signal values held in `buffer()` correspond to a 1 kHz sinusoid on the audio codec with $f_s = 48\,\text{ksps}$

```
cd C:\DSP_ST_LiB\STM32F4_GUI_slider_project
```

```
C:\DSP_ST_LiB\STM32F4_GUI_slider_project
```

Make sure the notebook is pointing to the folder containing the data file

```
x = loadtxt('buffer_int16.dat')
len(x)
```

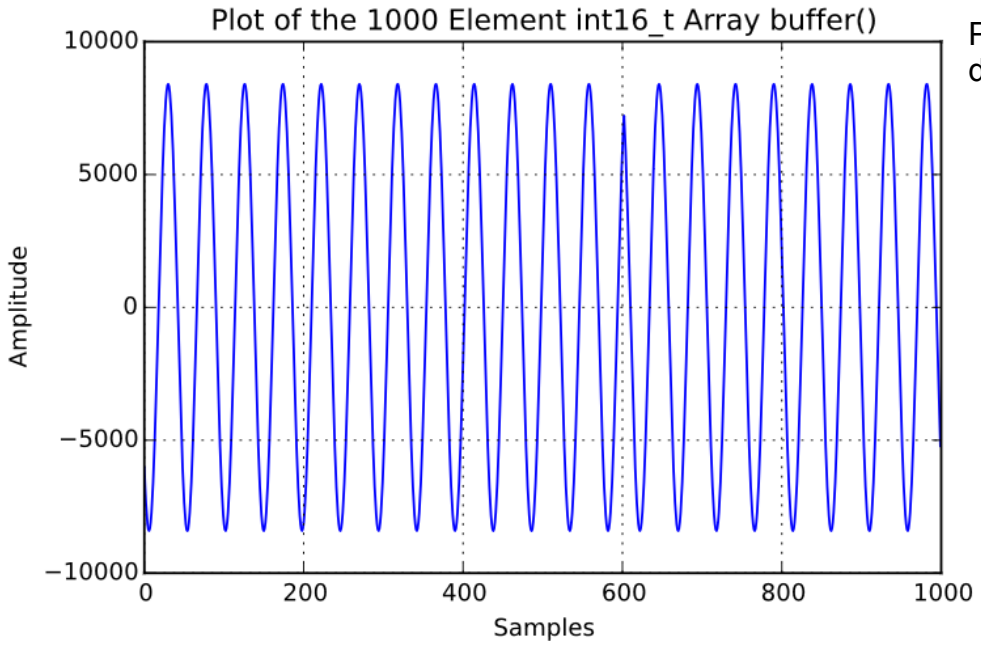
```
1000
```

```
print('Max = %4.2f, Min = %4.2f' % (max(x),min(x)))
print('Mean = %4.2f, Variance = %4.2f, StdDev = %4.2f'\
      % (mean(x),var(x),std(x)))
```

```
Max = 8408.00, Min = -8410.00
Mean = -65.01, Variance = 35107094.10, StdDev = 5925.12
```

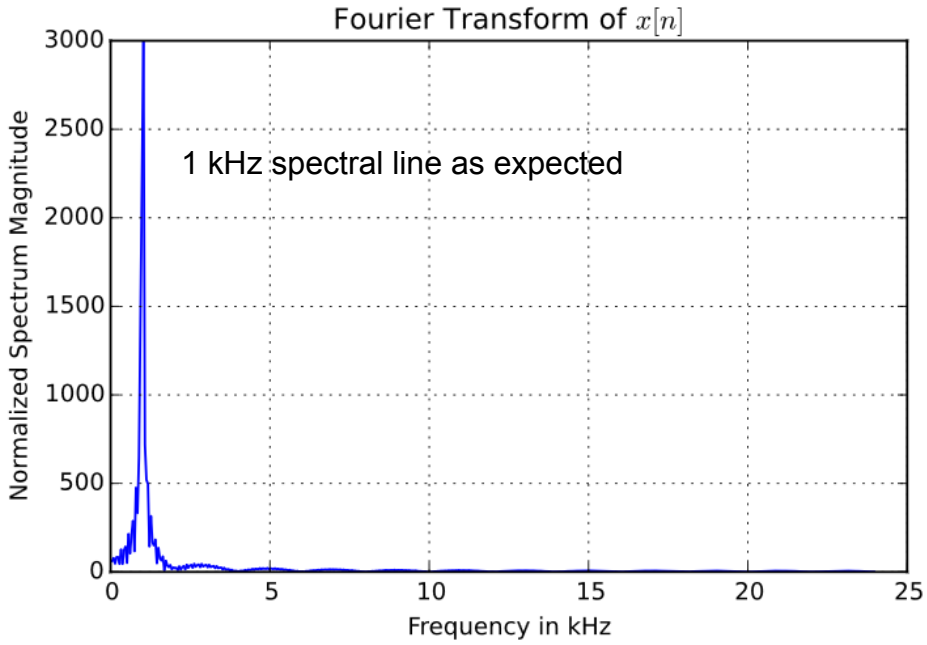
Compute some stats

```
plot(x)
#stem(x)
xlabel(r'Samples')
ylabel('Amplitude')
title(r'Plot of the 1000 Element int16_t Array buffer()')
grid();
```



Plot the time domain signal

```
f = arange(0,.5,1/1024)
w,X = signal.freqz(x,1,2*pi*f)
plot(48*f,abs(X)/len(x))
xlabel(r'Frequency in kHz')
ylabel(r'Normalized Spectrum Magnitude')
title(r'Fourier Transform of $x[n]$')
grid();
```



Plot the frequency domain as a scaled version of $\left|X(e^{j2\pi f/f_s})\right|$

- The C-code set-up that fills the array `buffer()` is summarized below:

```
19  //Globals for parameter sliders via serial port on USART6
20  float P_vals[NUMBER_OF_FIELDS] = {1.0,1.0};
21  int8_t fieldIndex = 0;
22  int8_t H_found = 0;
23  int16_t buffer[1000];
24  uint16_t buf_idx = 0;
25
26  void SPI2_IRQHandler()
27  {
28    int16_t left_out_sample = 0;
29    int16_t right_out_sample = 0;
30    int16_t left_in_sample = 0;
31    int16_t right_in_sample = 0;
32
33    //GPIO_ToggleBits(GPIOD, GPIO_Pin_15);
34
35    if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) == SET)
36    {
37      GPIO_SetBits(GPIOD, GPIO_Pin_15);
38      left_in_sample = SPI_I2S_ReceiveData(I2Sx);
39      left_out_sample = (int16_t)(P_vals[0]*left_in_sample);
40      buffer[buf_idx] = left_out_sample;
41      buf_idx++;
42      if (buf_idx >= 1000) buf_idx = 0;
43      while (SPI_I2S_GetFlagStatus(I2Sxext, SPI_I2S_FLAG_TXE ) != SET){}
44      SPI_I2S_SendData(I2Sxext, left_out_sample);
```

Add globals

Fill the buffer and wrap when full