

FFTSCOPE: demo program for Tasking DSP56xxx toolchains.

The FFTScope program demonstrates the capabilities of the Tasking DSP56xxx toolchain. The application displays the power spectrum of a stereo audio input signal on an oscilloscope in real time. The program has been developed and tested on a DSP56002EVM, but should run with some modifications on a DSP56303EVM as well.

The complete sources are provided with the absolute application image, prepared for debugging with CrossView, the TASKING debugger for the DSP56xxx Family. This will give you full flexibility in tailoring the application. The absolute image file is also provided in Motorola CLAS format. This file does not contain symbolic debug information. With the debugger you can change some program parameters on the fly, and you can store the whole program in a Flash EPROM on the DSP56002EVM.

Setting up the equipment.

The following scheme shows how you can create the demonstration setup for which the program has been prepared:

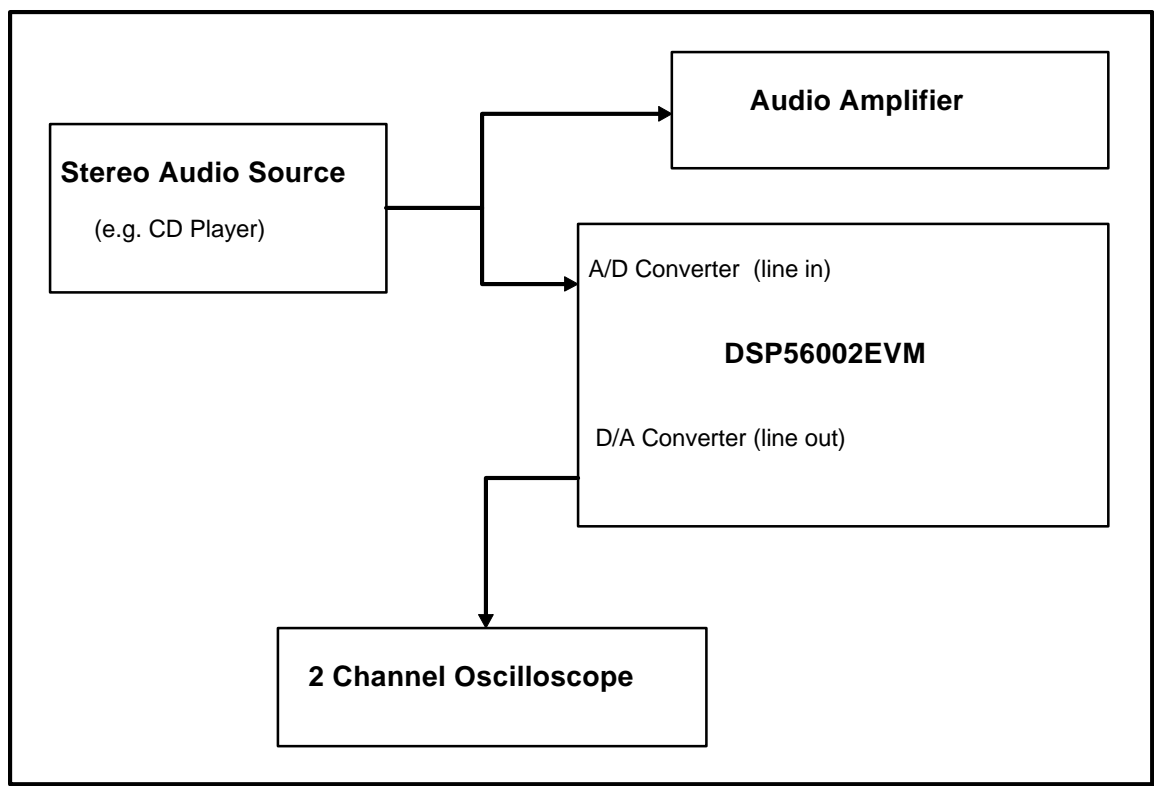


Figure 1. Demonstration setup FFTScope

TASKING DSP56xxx FFTScope demo

For cabling you can use standard stereo audio cable with 3.5mm phone connectors. Use a T-piece to connect both the amplifier and the DSP board to the audio input. Note that the headphone output is used for the oscilloscope, not the line out, as this output is DC coupled and will show the power spectrum (which has a large DC offset) without shifting. Therefore you must set the oscilloscope inputs also to DC to get a proper display of the power spectrum. The line output of the DSP56002EVM carries the same signal as the headphone outputs, and both are not recommended for aural consumption... The line input level should be set to a reasonable value on the audio source output.

The following settings for the oscilloscope assume that you run the program 'as is', with the default settings. Input voltage range should be about 1 V/div, the horizontal display should be set to about 1 ms/div (10 ms total). Triggering is provided by a sharp negative pulse at the start of each frame. The oscilloscope trigger should be set to DC input, positive slope and a slightly negative value; automatic triggering will mostly work well. Of course dual display mode should be selected (if available) to show both channels, and all other controls should be set to default.

Important:

Jumper J12 on the EVM board must be set to 16K of X-memory and 16K of Y-memory for this application.

After making all the connections and switching on power, check if the audio circuit is working correctly. Then download the program to the DSP and run it: in CrossView, select File | Load application, locate the fftscope.abs file in your file system, then download and run it. Without audio input the oscilloscope should trigger to a somewhat negative DC voltage, with probably some oscillations around the trigger point. Set the vertical position to a reasonable position for both signals. Adjust the horizontal scan frequency until exactly one period is displayed. After applying the audio input the power spectrum will be displayed as a wavy signal. Frequency runs horizontally from 0 to 4000 Hz, the amplitude of each frequency is displayed vertically in dB's (-60 to +20 dB). Both scales can be roughly calibrated with a test pattern; see the chapter about program parameters for this.

How the FFTScope application works.

The following diagram shows the functional setup of the FFTScope demonstration program:

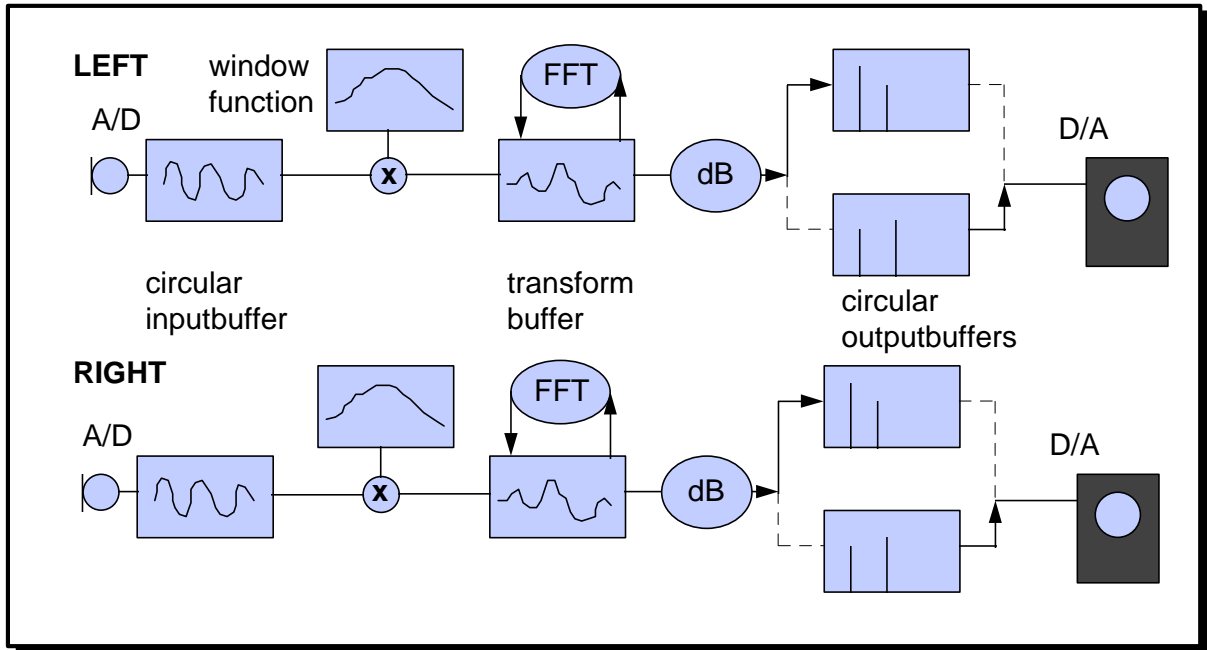


Figure 2. Functional Setup FFTScope Demo

The application consists of four parts: the audio input routines, the audio processing routines, the output routines and the control routines.

The audio input part stores incoming audio samples in the input buffers. The codec on the DSP56002EVM requires the same sampling rate for the input and output channels. To get a stable image on the oscilloscope we need a high sampling rate on the output channel, which results in a higher sampling rate on the input channel than we strictly need.

Because of this the input is downsampled to get a low input sampling rate. The lower frequencies of the audio spectrum contain most of the information of interest, and in this way we get better resolution. After initialization in `ssi_init()` this part runs independent of the other parts in the interrupt service routine `ssi_receive()`. This routine fills a four-word structure with the data received and on each n-th frame sync (downsampling) copies the current audio data to the input buffers. An anti-aliasing filter has not yet been implemented here.

The audio processing part takes data from the input buffers, calculates the power spectrum and stores the result in the output buffers. First the input data is windowed and written to the fft conversion buffer in `fft_get_windowed_data()`. The fft routine (based on a Motorola macro) is then called, yielding the transformed values in the same buffer in bitreversed order. Two flavours are provided, the default one providing block floating point functionality using the hardware support for this, the other doing a plain fft with data prescaling. Finally a routine is called to convert the data to a power spectrum, which implies calculating the power per frequency and

converting it to decibels. This routine, `fft_calc_power_spectrum()`, takes a bitreversed input array and writes into the selected outputbuffer.

The output part sends the power spectrum data to the oscilloscope through the D/A converter. To get the highest possible refresh rate the converter is set to the maximum sampling frequency. This part is also written as an interrupt service routine, running independent of the rest of the program, to ensure that the previously calculated data is sent to the oscilloscope during calculations. As writing to the outputbuffer directly would cause the displayed spectrum to change halfway at times, a double buffering approach is used: When a new buffer is filled with power spectrum data, the routine switches to the new buffer at the beginning of a new frame only.

The control routines start the whole process and handle changes of program parameters. In order to handle all possible changes correctly and with minimal calculations, new parameters for each channel are stored in a separate area and compared after each transform.

Implementation remarks

The transmit and receive interrupt service routines are written in both C and assembly. The program uses the assembly language version because of the following reason: The CS4215 gives a total of 8 interrupts per sample, yielding a total of 392.000 interrupts per second at 48 kHz sampling frequency. Together with the downsampling, intermediate buffering and output buffer switching it is not possible to accommodate the code overhead of the C-version. Actually, even the assembly version gives a large processor load; at 40 MHz, on average only about 50 instructions can be executed per interrupt!

The fourier transform functions have also been coded in assembly, because macros for them already existed (some modifications were needed, however). In order to check the functionality of these macros C versions have been written also, doing a one-to-one translation as much as possible (e.g., variable names are register names). This makes it possible to create an executable on a host platform that emulates the behaviour of the routine. The application has been succesfully tested on MS-DOS with Borland C++ 5.0, but should compile with different brands of C-compilers as well. Real-time execution on those host platforms is not possible of course, but textual output serves well for functional debugging purposes.

Declarations of pointers to circular buffers is very easy and straight forward with the Tasking C-compiler. In this application the declarations are somewhat more complex: The structure CIRCPtr has been defined to be allow declarations of pointers to circular buffers of variable size. The function `mk_circptr()` creates pointers to circular buffers of any size. Also, the `fft` routines have been prepared to be able to work on variable sized data (the size must be a power of two, though). This makes it possible to change the transform size without recompiling or even stopping the application.

As the input buffer has a length of exactly the maximum `fft` size, it is possible that the first samples will be overwritten before they are transferred to the `fft` buffer (this can occur only when the maximum `fft` size has been selected). Errors in this area will be masked by the windowing function, however, just like non-periodicity of the input data.

Increasing the input buffer sizes slightly will cost a lot of memory, because they are circular buffers and need to be placed on specific boundaries. Attempts to disable interrupts temporarily to avoid this problem have proven to cause unreliable system behavior.

Program Parameters

If you want to change program parameters while the program is running you have to use the 'newparams' structure array. This structure contains only those variables that you may modify on the fly. Initial values will be set in the routine params_init(). The program will check for changes in the values in this structure before each frame in the function params_update(). All necessary changes will be made in the tables affected before it returns. Since there is only one sincos table, it is not possible to have different fftsize values per channel, and thus these are forced to be the same.

The following parameters can be adjusted:

Offset	Name	Description	Range
0	decimation	Input sampling frequency decimation -1 fsin = FSAMPLE/(decimation+1) only the value for channel 0 is used	0..n
1	fftsize	Size of the fourier transform indirectly the size of the power spectrum. Must be a power of two.	8..2048
2	wtype	Type of window to apply on input data. 0 = no window; 1 = triangular window; 2 = Hanning window; 3 = Blackman-Harris.	0..3
3	avg	Number of spectra to calculate and average before writing to output	1..n
4	ttype	Type of transform: 0 = no tranformation (not useful) 1 = standard FFT; 2 = block-floating point FFT.	0..2
5	dtype	Type of display on the channel. 0 = no display (freezes current buffer); 1 = calibration pattern; 2 = power spectrum; 3 = audio input of other channel	0..3
6	gain	Input gain, value for A/D converter gain section. [dB]	0..n
7	rcvd	Flag: new data present (not used).	0/1
8	-	Start of data for channel 1.	

You can use the rcvd flag to indicate that new data is present on the channel that you use for transfer of parameters to the board: either through the host interface or the serial port. The flag is reset by the parameter update routine. If you use CrossView, you can use the symbolic structure member names to assign new values to the fields. If you debug without symbolic debug information you can calculate by taking the sum of the structure member addresses by using the structure address from the locator map file and the offsets specified in the table above. The locator map file also specifies the memory type where the structure is allocated (probably X).

Examples

1. Calibration

To get a calibration pattern instead of the power spectrum, the display type must be set to 1 (DS_CALIB). In CrossView, click on the Halt button and enter:

```
newparams[0].dtype = 1  
C
```

The display will now show a staircase pattern representing 20 dB steps vertically and 1 kHz steps horizontally. You can adjust your oscilloscope to match these with the display grid in order to have an indication of the values in the display. To return to the power spectrum display, set the display type to 2 (DS_PSPEC).

2. Window Functions

To see the effect of different window functions, you can set a channel to no window instead of the default hanning window. In CrossView, click on the Halt button and enter:

```
newparams[0].wtype = 0  
C
```

When you apply the same audio signal (preferably a sine wave with adjustable frequency, e.g. from a signal generator) to both channels, you will see the difference immediately. The effect, called leakage, can be dramatic and is caused by the imperfect match between the base frequency of the signal and the size of the fft. As you can see it is masked pretty well by applying a window function. In the same way you can examine the other window types. The triangular window has been provided for this purpose only; it is very simple to implement, but yields poor results. The Blackman-Harris windows gives very good sideband suppression, but a broader peak than the Hanning window.

3. Maximize Power Spectrum

To select the maximum size of the power spectrum, set the fftsize parameter to 2048. In CrossView, click on the Halt button and enter:

```
newparams[0].fftsize = 2048  
C
```

The oscilloscope will now display 1024+2 power spectrum points. Note that the refresh rate will go down accordingly as the calculations take more time. The refresh rate depends on the dsp clock frequency, number of active channels and the algorithm selected; typical values are 30 Hz for 40 MHz dsp clock and default settings, 60 Hz for 64 MHz dsp clock. The refresh rate can also be measured by the square wave output on pin 8 (PC3) of J10; transitions occurs on the completion of the calculations for a channel. Moreover, the frame rate will go down too. Some oscilloscopes will give a flickering image at low frame rates, but if you have a storage scope it may work very well. You can calculate the frame rate as: $48000/((fftsize/2)+2)$ [Hz] and has a value of about 48 Hz in this case.

Building the application

To (re)build the FFTScope demonstration program you need to have a licensed version of the TASKING C compiler for the DSP56xxx Family.

If you have made changes to the sources you can use the following procedure to rebuild the application. First be sure that the `../bin` directory of the Tasking toolchain is in your path. Switch to the directory containing the source files and start the make utility with the command `'mk56'`. Several options can be set on the command line if required:

<code>clean</code>	Only remove all object files
<code>CLK64=1</code>	64 MHz clock instead of 40 MHz
<code>CLK80=1</code>	80 MHz clock instead of 40 MHz
<code>SRCDIR=<path></code>	specify directory for source files (default <code>..</code>)
<code>debug=1</code>	full debugging optimization option
<code>clas=1</code>	select Motorola CLAS output format

Storing the program in Flash EPROM.

To store the program in Flash EPROM, you can use the routine provided with the EVM board. Assuming that the board has been fitted with a suitable Flash EPROM (note that not all 32K flash eproms are pin-compatible; you must have a 29C256; check the pinning with the schematics, especially the `\WE` pin) you can use the procedure described below:

You can program Flash EPROMs on the EVM with the routines in the file `flash.cld`, provided by Motorola. Since CrossView cannot process `.cld` files you need to use the Domain debugger supplied with the EVM to run this code. Reset the DSP and load the application (e.g. `fftscope.abs`) in the DSP ram, using either CrossView or a different debugger. Leave CrossView (without running the application) and start the Domain debugger. Load the eprom programming routine; set `r1` to contain the size of the program (use `0x1500` for safety); start the programming with the following commands:

```
load flash.cld
change r1 $1500
go
```

Note that these eprom programming routine cannot accommodate programs over `0x4000/3` words in size, as the loader cannot skip its own code that is stored on `0xC000` (the eprom addresses start at `0x8000`, and the eprom is only 8 bits wide). The end address of the code can be found in the map file as the start of `.ptext` (virtual) plus its size. All other sections will be initialized by the C startup code, so this is all we need to store.

The programming can be checked by looking at the first words (bytes) in the eprom with:

```
force r
di $8000
```

These should read `$xxxx80 $xxxxf0 $xxxx0a` etc., containing the first part of the reset vector of the program (`jmp $...`); the high order part is undefined. Note that the correct values only show up when the wait states for p-space are set to a high enough value; the `'force r'` command will do.

Upon hardware reset the code should run right away. You can cycle power off the board and reapply power to be sure it is the code in the eeprom that is running, and not something in the dsp ram. From CrossView you can still change variables, provided that the stored program is in sync with your sources. To connect, select the program in CrossView with 'N fftscope' and select 'R'. While in run mode press the reset button of the board. The program will now be downloaded from the EPROM and start. After a warning message from CrossView the program is now present in memory. After selecting the 'halt' button you can proceed as usual.

Troubleshooting

When downloading a modified version of the executable you are advised to enter 'rst' before downloading the code. In this way you are sure that no interrupt services will take place during downloading, which will cause havoc if the interrupt service routine is being replaced.

CAUTION:

The DSP56002EVM board is surprisingly sensitive to glitches on its audio connectors; swapping the input to a different audio source may very well lock up the DSP.

A hardware reset will be necessary to regain normal system behavior.

Unless you have programmed an EPROM with the program you will have to reload the code to run the program again.

If flash eeprom programming does not succeed, one of the possible causes is a bug in your executable flash.cld. On position \$7f0e the opcode should read rep #6, not rep #5; the erroneous code stores twice as much data, overwriting the bootstrap loader if datasize \geq 0x4000/6 instead of 0x4000/3. Reassembling the source will remove this bug. Also, it is wise to switch off interrupts during programming, to avoid timing errors. Set the status register to \$0300 for this.

Conclusion

With the Motorola DSP56002EVM board and this software an oscilloscope can be turned into a two-channel audio power spectrum display at very low cost. The application shows that an extensive program as FFTScope can be written in C, using the Tasking tools with its powerful language extensions. Thanks to using C as programming language the effort is lowered significantly, compared to the time it would take to implement a program like FFTScope completely in assembly language. The language extensions enable the compiler to produce code that is close to the efficiency of hand coded assembly routines. The few program sections that still require assembly level code can be incorporated easily in the C code via the assembly language interface of the compiler. Moreover, the C programming language allows easy extension of the program and features like on-the-fly spectrum size changing that would be very hard to implement in an assembly language program.

Appendix A: File list

Filename	Description
fftscope.abs	executable (with debug information)
fftscope.map	locator output file
fftscope.h	project header file
fftscope.c	main program
cs4215.h	codec defines
fft_4215.c	codec routines for this application
fft_trfm.c	fft routines
fft_test.c	input data generation for testing fft routines
fftr2a.asm	wrapper for normal fft assembly macro
fftr2a.inc	normal fft assembly macro, adjusted for variable size
fftr2bfp.asm	wrapper for block floating point fft assembly macro
fftr2bfp.inc	block floating point fft assembly macro
flog2.asm	wrapper for fractional log2() routine
log2nrm2.inc	fractional log2() routine
c56.h	extended header file for cross-platform development
c56.c	dummy routines for cross-platform development
makefile	makefile for command line compilation of the project
56002evm.dsc	locator descriptor file for 56002EVM in 16K mode
56002evm.mem	memory descriptor file for 56002EVM in 16K mode
fftscope.cld	executable (without debug information)