Embedding it better...

µTasker Document

µTasker – DSP

Table of Contents

1. Introduction

This document describes DSP functions that are integrated in the µTasker project. They encapsulate parts standard library implementations, such as the ARM CMSIS DSP library, in a way that allows simple use and portability as well as making use of any available hardware acceleration.

2. Fast Fourier Transform

The Discrete Fast Fourier Transform is a popular function used to transform discrete time domain samples to frequency domain information

2.1. FFT Basics

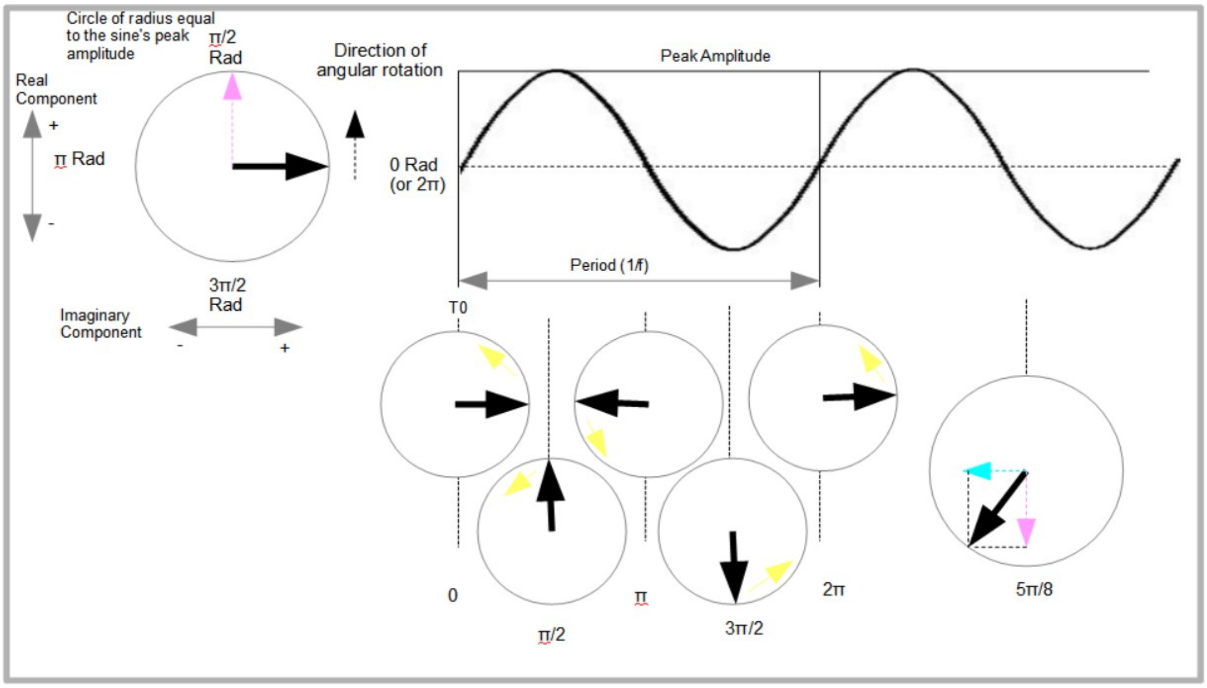
This section does not repeat the theory involved for the FFT itself but instead refreshes basics required to understand its requirements and its results. The topics of importance are usually studied during advanced high-school mathematics classes or university engineering courses but are in fact not difficult as long as the reader can imagine/visualise the concepts involved. Students who have studied the concepts may not have understood the exact implications and so this brief refresher may also be of use.

2.2. Sine waves

The result of an FFT expresses how a number of sine waves (series of frequency components) make up the input signal. Therefore it is important to know how a sine wave is represented and the following presents a few basic rules that tend to be used:

- A sine wave can be expressed using 3 components: **amplitude** (either peak, peak-to-peak or RMS); **frequency** (or period, which is 1/f); **phase** (which is usually in radians 0..2π, but could be in degrees 0..360°). *If nothing else is stated, frequency is in Hz, amplitude is voltage peak, and phase is in radians*.

- A sine wave can be represented as a vector made up of a real and an imaginary component which rotates around a circle. The following examples should make this understandable, which is important since this is essential knowledge for the best visualisation/understanding.



The diagram shows a pure single sine wave. The sine wave is synchronised to T0 which means that it has no phase shift with reference to this point in time. This point is displayed on a vector circle as an arrow pointing to the right and representing 0 radians, with a real amplitude of 0 and imaginary amplitude equal to the circle's radius. The real component also represents the magnitude at that point of time (sample) of the sine wave as a signal (eg. a voltage).

The sine wave's signal amplitude during a period can now be represented as the real value of the unit vector (the vector's length is always equal to the circle's radius) as it rotates around the circle in the anticlockwise direction. A further three points are shown in the first cycle where the vector has rotated π/2 radians (90°) in between each (π/2 also represents one quarter of the period of the sine wave).

The fifth position shows a point where the sine wave signal is negative and decreasing at 5 π/8 into a period. Again the signal's value at this point is equal to the real magnitude. The unit vector is always the same and the real and imaginary magnitudes that make up the vector can be seen to be as follows:

- Real magnitude = sin(phase)
- Imaginary magnitude = cos(phase) or sin(phase + π/2)
- The unit vector (peak amplitude of the sine wave signal) = Square-root of (Real magnitude squared + Imaginary magnitude squared) [by simple Pythagoras]

2.3. Adding Sine waves

It is very simple to add two or more sine waves as signals since the result is just the sum of the samples values at each point in time, which is an addition of all real values of the vectors of each.

The interest in adding sine waves is due to the fact that any repetitive signal can in fact be defined as a summed series of harmonically related sine waves. Each of these individual sine waves has its own parameters (frequency, amplitude and phase shift) and only by knowing each of these can the final signal be synthesised from each of its components by calculating the real component of each at every point in time (relative to T0) and summing them.

Multiple unrelated signals can also be summed in a similar fashion, although each individual signal may not be harmonically related in any way.

2.4. Spectrum Analysis

Analysing the spectrum of a signal is in effect the inverse of summing sine waves. It requires each of the individual sine waves that make up the signal to be identified, together with each one's frequency, amplitude and relative phase.

The Fast Fourier Transform allows this to be done by supplying a set of frequencies as vectors. A very simple result may be that the result gives a single frequency of 1Hz, with a real amplitude of 0 and an imaginary amplitude of 1. Referring back to the introduction to the vector representation it should be fairly easy to see that it is a sine wave with its maximum amplitude at T0, which is in fact a cosine because it is shifted by π/2 when it starts.

A more complicated signal may return many individual frequency components, each with their own real and imaginary amplitudes relative to T0. In each case the original signal could be reconstructed (with some potential loss of accuracy in the process) by again summing the real component of each individual frequency as time advances.

2.5. Sampling Rate and Frequency Resolution

Well know is the fact that the sampling rate needs to be at least twice as fast as the highest frequency component (Nyquist frequency). Furthermore, it is usually of importance to ensure that no higher frequencies are present in the input by using an anti-aliasing filter at the front-end, before the signal is sampled. This avoids aliasing distortion, where higher frequencies can appear as in-band frequencies folded around half the sampling rate.

Assuming a sampling rate of f_s, and an input bandwidth of half of this (f_{bw}) a number of samples need to be made before an FFT can be performed. The higher the number of samples used as input to the FFT, the better the frequency resolution. The value for the frequency output divisions (called bins) is given by

$$\mathbf{f_{bin} = f_s/(2 \times samples) = f_{bw}/samples}$$

For example, a 48kHz sampling rate, with an input bandwidth close to 24kHz, using 512 samples as input to an FFT (of the appropriate length) will give bins of each 93.75Hz and thus a resolution of this frequency step. There are half as many output frequency bin values as there are input samples.

2.6. FFT Options

The first choice of FFT depends on the number of bins that are to be required. This was discussed in the previous section and gives the number of input samples that are necessary for a single transform to be performed. Depending on the sampling rate it also gives the output frequency resolution (the frequency width of output bins).

The implementation of the FFT also depends on the sample size and type. Integer implementations may be faster than floating point but tend to give lower output accuracy due to more rounding errors.

A potentially important requirement when measuring a snap shot of an input stream is the use of an input window to reduce the effects of the fact that the start and end of the sample buffer can have large steps due to the first and last sample representing in effect a jump from or to zero (discontinuity). Adding windowing is usually easy to do and is discussed further after the introduction of the FFT itself.

2.7. FFT Considerations in Embedded Systems

Due to the fact that an FFT requires a set of input samples before it can be performed the system must provide an input buffer adequately large to hold these samples. The physical memory size will depend on the quantity of samples required by the FFT in question and the sample word size (8 bit, 16 bit or 32 bit samples sizes result in 1:4 input buffer physical byte requirement size).

The output of the FFT is the same size as the input sample buffer and often the same buffer is used for the result (in-place calculation). The reason for it being the same size is the fact that there are half the number of results (bins) as input samples, but each result occupies two result locations since there is one real and one imaginary value for each bin (representing its amplitude and phase).

Although the FFT is a lot faster than a DFT (Discrete Fourier Transform) it still requires processing power that needs to be considered. The processing time will be dependent on the processor's speed and the size of the FFT. It may also be dependent on the sample word size and the processor's instruction capabilities – for example floating point FFT calculations will take much more time on a processor without a hardware floating point unit since the floating point calculations will need to be implemented in software.

Memory and time requirements are usually important factors in such processing and so need to be considered carefully. Some benchmarks are presented later and the µTasker project contains functions to easily perform tests and measurements to verify the results on any processor.

2.8. FFT Usage

The FFT function is based on the ARM CMSIS DSP Library which offers various common DSP functions, including those needed to work with FFTs.

The µTasker project's FFT interface function encapsulates chosen elements from the CMSIS DSP library version V1.4.5 in order to easily perform the most used FFT operations in a way that reduces the overall usage complication and, in the process, reduces the risk of errors in practical situations.

Some CMSIS DSP library files have been modified slightly to add the ability to use them in the µTasker simulation environment and to simplify including them. Some file content has been additionally controlled by defines in order to better control the code size, whereby the original files cause quite large code and const data to be included even when only limited FFT types are made us of – this allows optimising code size in comparison to using the original files directly, which is especially important when smaller single-chip processors are used.

For single-precision (32 bit) floating point FFTs the CMSIS library supplies the function:

```
    void arm_cfft_f32(
        const arm_cfft_instance_f32 *S,
        float32_t *p1,
        uint8_t ifftFlag,
        uint8_t bitReverseFlag);
```

which will be used for an initial discussion of the operation.

This supports 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096 point FFTs, whereby in the µTasker environment only the code and consts required for the one(s) that are needed is actually compiled and linked. The FFT lengths needed by the project code can be defined by

```
#define CMSIS_DSP_FFT_16   // support 16 point FFT
#define CMSIS_DSP_FFT_32   // support 32 point FFT
...
#define CMSIS_DSP_FFT_4096  // support 4096 point FFT
```

Assuming a sample rate of 48kHz an audio system would have a bandwidth of up to 24kHz, which is adequate for the complete range of human hearing and thus high quality audio. In order to perform an FFT on a series of input samples these are first collected as floating point values in a buffer of adequate size; for a 16 bin FFT this means that 32 input samples are collected to a buffer of adequate size; for a 256 bin FFT 512 would be collected and for the largest 4096 bin one 8192 would be required. *One needs however to be aware that the function requires input samples to be imaginary (a real and imaginary value for each, whereby the imaginary one would be 0 for real-world sampled input.* If this buffer is called `fft_buffer` the 256 bin FFT is performed by calling:

```
arm_cfft_f32(&arm_cfft_sR_f32_len512, fft_buffer, 0, 1);
```

`arm_cfft_sR_f32_len512` is a const struct that is supplied in the CMSIS file `arm_const_structs.c` in order to give the processing routine all information required to perform the task for this size. It contains the following details:

```
const arm_cfft_instance_f32 arm_cfft_sR_f32_len512 = {
       512,
       twiddleCoef_512,
       armBitRevIndexTable512,
       ARMBITREVINDEXTABLE_512_TABLE_LENGTH
};
```

where it is seen that the length of the FFT and suitable parameters for the length in question are passed.

We note that the `ifftFlag` and `bitReverseFlag` parameters are always 0 and 1 respectively since the inverse FFT is not required and the correct ordering of the output frequency components is preferred.

After the routine has performed its job the results are in the input buffer, which has been overwritten. There are now 512 floating point frequency values, each occupying two buffer locations with real and imaginary components. The output bin size is thus 24kHz/256 = 93.75Hz.

It becomes obvious that in order to increase the frequency resolution more input samples need to be collected (larger input buffer) and a larger FFT length need to be used. The relationships in the case of this reference is shown in the following table:

48kHz sample rate (<24kHz bandwidth) – single-precision floating points

Input samples	Input buffer size (4 bytes per float)	Frequency bins (FFT length)	Output buffer size (two floats per bin)	Frequency resolution
32	128	**16**	128	1.5k
64	256	**32**	256	750Hz
128	512	**64**	512	375Hz
256	1024	**128**	1024	187.5Hz
512	2048	**256**	2048	93.75Hz
1024	4096	**512**	4096	46.875Hz
2048	8192	**1024**	8192	23.4375Hz
4096	16384	**2048**	16384	11.71875Hz
8192	32768	**4096**	32768	5.859375Hz

Since more samples need to be collected before an FFT can be performed the delay between the first sample and the FFT being started also increased with FFT length – it is for example 666us for the 16 bin FFT and 170.6ms for the 4096 bin FFT.

The floating point input/output buffer size is now known for each FFT length. In addition, further buffering may be required (either temporary or static), in particular to possibly collect further data for subsequent conversions while one is in progress, or to allow for other system operations to be completed or performed without loss of sample data occurring.

µTasker project's FFT interface allows floating point FFTs to be performed from a circular input buffer which can be of various real input types (not just floating point) and thus preserves the input and greatly simplifies the collection of data and passing of the data to the function.

```
extern int fnFFT(void   *ptrInputBuffer,
                 void   *ptrOutputBuffer,
                 int    IInputSamples,
                 int    iSampleOffset,
                 int    iInputBufferSize,
                 float  *ptrWindowingBuffer,
                 float  window_conversionFactor,
                 int    iInputOutputType);
```

The usage to take an input buffer containing 16 bit signed values and generating the absolute magnitude of each frequency bin (the most popular utilisation) is:

```
#define FFT_INPUT_SAMPLES 512
float fft_magnitude_buffer[(FFT_INPUT_SAMPLES/2)];

fnFFT((void *)fft_buffer, (void *)fft_magnitude_buffer, FFT_INPUT_SAMPLES,
iInputOffset, (FFT_INPUT_SAMPLES * sizeof(signed short)), 0, 0,
(FFT_INPUT_HALF_WORDS_SIGNED | FFT_OUTPUT_FLOATS | FFT_MAGNITUDE_RESULT));
```

`fft_buffer[]` is an array of `signed short` values of any length greater or equal to `FFT_INPUT_SAMPLES`. It can be a circular buffer, meaning that the user simply needs to specify where the start of the input samples begin (`iInputOffset` as index) – irrespective of whether they are linear or cause a wrap in the circular buffer to take place.

We note here that no windowing is performed on the input signal and so `ptrWindowingBuffer` and `window_conversionFactor` are set to 0.

2.9. Floating Point FFT Benchmark Measurements

The following are results of actual calculation times measured on some Kinetis processors with m0+ and M4 cores. Where possible the advantage of using internal FPU unit rather than floating point software implementation is compared.

FFT Length	FFT processing time on 48MHz KL27 (m0+)	FFT processing time on 120MHz K64 (m4) with SW floating point operations	FFT processing time on 120MHz K64 (m4) with HW FPU operations (us)
16		2.6/104/53	3.2/10.8/28
32		4.6/277/106	5.5/22.4/58
64		8.6/641/211	10.7/40.4/112

128		17/1712/421	22/108/218
256		33/4057/841	43/236/433
512		65/10540/1683	88/427/865
1024		131/21170/3360	173/1073/1730
2048		260/43110/-	345/2439/-
4096		516/63330/-	686/4340/-

Create complete floating point input/perform FFT/calculate magnitude of output vectors

Flash memory requirements:

FFT Length	FFT processing time on 48MHz KL27 (m0+)	FFT Flash memory requirements K64 (m4)
16		0.6k
32		1.3k
64		0.7k
128		1.5k
256		3k
512		4.9k
1024		11.7k
2048		23.5k
4096		40k

3. Fixed Point FFT

To do...

4. Windowing

To do...

5. Spectral Energy

The output of the FFT routine gives the vector of each frequency component, made up of real and imaginary components. Often the signal energy is important and so each frequency component needs to be converted to its energy, which is expressed in V^2/Hz.

The conversion is performed by …. To do.

6. Conclusion

This document has introduced DSP functions that are integrated in the µTasker project.

Modifications:

V0.00 24.1.2017:
V0.01 30.1.2017: Added benchmark results
- Initial draft version (in progress)