PIC n' Mix

Mike Hibbett

Our periodic column for PIC programming enlightenment

Practical DSP – Part 2

HELLO again! In this second article on DSP (digital signal processing) with Microchip's dsPIC33 microcontroller, we move to installing the development environment and support files to enable us to create our application. The development environment consists of the MPLAB X IDE (integrated development environment), the XC16 C compiler (the dsPIC33 has a different code compiler to the usual PIC16 and PIC18 processors) and an example application from Microchip to help us get started.

Free tools

All these software tools are provided free by Microchip. The XC16 compiler is available with a 'free or paid-for' model. It's the same download, but you can select which version you are using during installation by either selecting 'operate in free mode' or entering a license code. To Microchip's credit, the free version is plenty powerful enough, and we will not miss the additional features – mainly code space optimisation – in our projects. The optimisation aspects come in handy when you want to reduce code size to fit the code in a device with less Flash memory. That's important when you are selling tens of thousands of products; but for us, we can simply pay a few pounds more and target a larger memory chip, as we have done. As an additional bonus, the DSP library functions – the code that we care about being optimised – are supplied pre-compiled and therefore pre-optimised.

For this article series we are using the latest version of Microchip's development tools, namely MPLAB-X v4.10. If you would like to follow along then we suggest you download this too. The code shown here is compatible with MPLAB-X, not the old MPLAB IDE. If you are downloading the IDE, then allow installation with the offered defaults and allow installation of additional device software – this step adds drivers for the PICkit 3 and related debuggers.

Once installation completes, your web browser will launch and open on several pages. Click on the **MPLAB-XC compilers tab**, scroll to the bottom and select **Downloads**. Now click on **MPLAB XC16 Compiler v1.33** to download the installer. Once the download completes, install that too, accepting all the default options. Once the installation completes you can go ahead and run the MPLAB X IDE by clicking on the desktop icon. Do note that it's the IDE we run, not the similarly named 'IPE', which is the programming tool.

Exploring the DSP library functions

The **Help** menu for the XC16 Tool chain hints at the existence of the DSP library, and tells you to go search for the documentation in the installation directory. We eventually found the DSP library and documentation, but it appears dated – the user guides refer to it as the **C30 DSP library**, which is a different device family to ours. Buried within the source code zip file however, an updated documentation states it is the **XC16 DSP Library**. So, fingers crossed we will not come unstuck with our choice of processor.

As is normal with documentation about library functions, the guides tell you what the functions do, not how to use them. Thankfully, example code is provided to show the use of the library in some realistic contexts, and these example applications are always a vital source of knowledge. Here again we come have a problem – the FFT example project **CE018** is very old, not modified since 2007 and designed to work with the old and incompatible MPLAB v7 IDE, and the C30 compiler (now retired and replaced by XC16). So, potentially, we now face a significant porting job to make it build under MPLAB-X.

Thankfully, a search on the Internet revealed that a more recent example project has been created by Microchip, named **CE482**. This was last updated in 2015, targets MPLAB-X and the XC16 compiler, and specifically refers to the dsPIC33 processor family. With luck, our workload has been dramatically reduced.

In situations such as this, where you are starting on a new project, it's advisable – highly advisable – to take small steps, validating each step as you go. With this approach in mind, we started by downloading the **CE482** example project from Microchip, opened it within MPLAB-X IDE and attempted to build it with no modifications. It was with some great relief that this build completed successfully 'out of the box'. A quick scan of the build process output messages shows that the total code space used is 3714 bytes, and 3672 bytes of RAM. Very small, which means our choice of processor should be able to hold the DSP library functions required with plenty of room for future application code – looking good!

The next step is to change the project to target our selected processor and try building again. We are not worried about running the code on our board yet; in fact, we have not even looked to see exactly what the code does, let alone whether it will run on our minimalist hardware. Remember, it's baby steps, one change at a time. We creep slowly towards having code run on our board. The example application has been designed to run on Microchip's Explorer 16 board, an extensive but pricey solution, more than ten times the price of our hardware. With luck (and a little forward planning!) the code will be easy to 'port' to our board.

Porting

The process of taking a program written for one microcontroller and moving it to another is called 'porting'. Just how complex this task becomes is down to several factors:

- How dependent the application is on a microcontroller's specific features
- The level of differences between the microcontrollers
- How well the application was written
- The clarity of the porting documentation

The first point above is covered; we are dependent on the DSP instructions in the processor, specifically within the Microchip-supplied DSP library that our application links to during compilation. As that library supports our processor, and the example application does not connect to an external signal source, we should be good.

The second point is reasonably well covered because we are sticking with a processor in the same family, with a large amount of code and data space, so we don't expect any (nasty) surprises.

The third point is hard to judge, so we proceed by checking the porting guidelines and dive right in. The **Readme** file for the example application hints that the process will be simple:

Change device selection within MPLAB IDE to other device of your choice by using the following menu option:

MPLAB X >> Configuration drop-down option >> <Listed Device Configuration>

And that's it. Great – this is going to be simple!

Uh oh, not so fast – a look at the options provided in the drop down allows for only three processor options, and they do not include our processor (see Fig.1). What do we do now? A quick look at the files included in the project shows that there are several files specific to each processor offered in the drop-down list. As should be expected, each processor has a different amount of memory and different configuration bit settings that need to be accounted for when building any application. So, there are different files for each one. Fortunately, one of the processors listed – the dsPIC33EP256GP506 – is very similar to ours. It just has more pins and less memory.

We now have two options: 'hack' the settings for the dsPIC33EP256GP506 configuration files to match ours, or create a new configuration specifically for ours and do this properly. After much soul searching we chose the latter, as this way we are extending the Microchip example rather than creating a hard-to-maintain modification.

The porting process starts by duplicating the configuration settings, an option provided within the IDE. We rename the configuration to **dsPIC33EP512GP502**, change the debugger to **PICKIT3**, and select the correct processor type. Next, we copied the source code directory from **src\system_config\exp16\dspic33ep256gp506** to a directory called **src\system_config\epe\dspic33ep512gp502**.

Finally, we added the newly copied-over source file to the list of source files within the project itself. Performing a build of the project resulted in success, so that's the porting over. It took just three hours of puzzling, which is quick, even for a simple project such as this.

The next step is to fully examine the source code of the Microchip example before we build our own, to understand the context of how to call the DSP library functions. If you would like to follow these articles along fully, go ahead and download the complete set of project files, which can be found on the current issue's monthly web page at **www.epemag.com**. (In case you are wondering, the original license from Microchip permits modification and redistribution.) We will be expanding on these files over the coming months, so make sure you check out each month's downloadable files from the website.

Inside the example application

Let's go up a level and review where we are. The point of the exercise this month is to obtain a configuration of the DSP library that will run with our processor, and to understand the context of how the DSP library functions should be called. The help documentation within the IDE explains what the functions do, but not how to string them together. With luck, the example application supplied by Microchip should do this, so let's look.

The bulk of the code is held in the file **main_fft_example.c**, located in the directory **src\system_config\epe\dspic33ep512gp502**. Open the file in a text editor if you would like to follow along.

As a side note, IDEs such as MPLAB-X are not the best text editors when all you want to do is quickly view the contents of a text file. Notepad, supplied with Microsoft Windows, is acceptable, but very basic. Our recommended text file editor is Notepad++, a free and fully functional text editor. It opens quickly and does not fill the screen with clutter. You can download it from the creator's website here: **notepad-plus-plus.org**

The first section inside **main_fft_example.c** is the list of configuration bits settings. These are important because they define, among other things, the settings your microcontroller uses for the main system clock – and since this a different processor, it is almost certainly wrong. Correcting this will just require comparing the configuration settings section of both datasheets. We will come back to this section later.

Next, in the file, some variables are declared. Thankfully, just six variables, which will help simplify our understanding. Let's look at the four important ones.

`fractcomplex sigCmpx[]`

This is an array that will hold the input data to the FFT function call. Note the unusual data type, `fractcomplex`. This is a special data type created by Microchip for use with their DSP library. A variable of type `fractcomplex` consists of two parts, the real part and the imaginary part. These are 'complex numbers', not simple integer or floating-point values.

Complex numbers are used in the FFT calculation because the signal is composed of many different frequencies varying in both amplitude *and* phase – complex numbers enable us to represent both quantities. A fuller explanation of complex number theory is beyond the scope of this article series, but it's fine for us to simply ignore this fact, and just take care to follow the data types being used, and how they are translated back to useful and meaningful values.

When the FFT function is called, the resulting frequency data is returned in `sigCmpx`, so a second output array is not required.

`fractcomplex twiddleFactors[]`

This unusually named variable is actually a now-standard term used within DSP algorithms. This is an array of constants that are used within

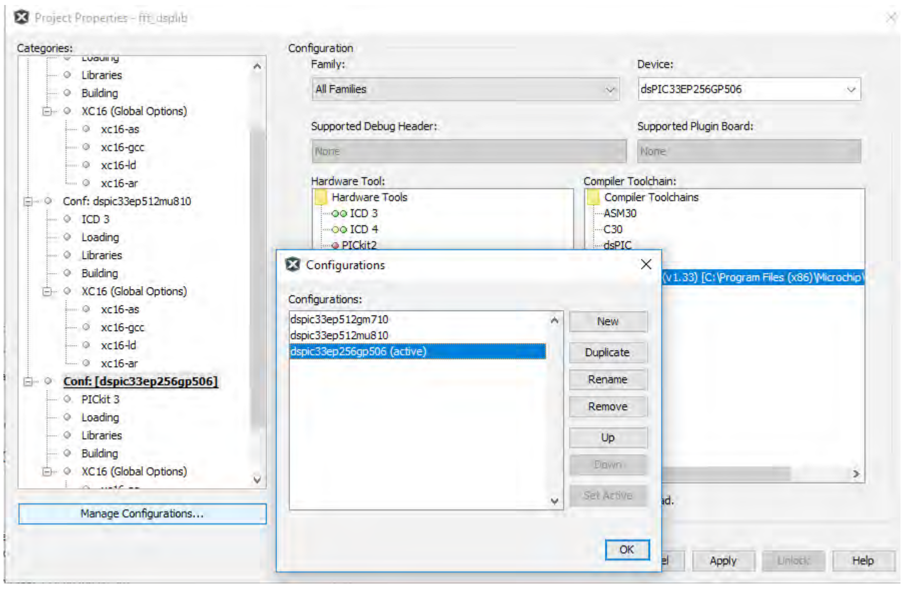

Fig.1. Changing configuration in the IDE

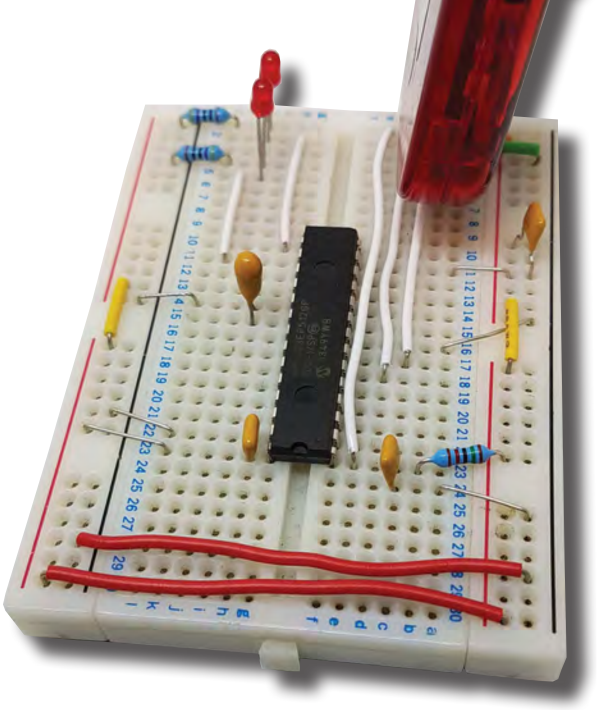

Fig.2. Completed breadboard

the FFT function itself. The values change for FFT algorithms of different input sizes, but for a fixed input data size, the constants are the same – so it is convenient to pre-compute these before use. The Microchip example uses an FFT size of 512 samples, and we will stick with this, to save having to understand how to calculate new twiddle factors. We will talk about the implication of using a 512-sample buffer later, as the length of the buffer used impacts how long it takes to perform the FFT calculation, and the resolution of the frequencies obtained.

`fractional input[]`

This is an array in which the pre-computed example input signal is stored. In our example code the contents, held in the file **inputsignal_square1khz.c**, define a 1kHz square wave signal. Notice that again, this datatype is a Microchip defined one. It is used to represent a special form of non-integer values that are encoded into a 16-bit integer variable. In next months article we replace the contents of this array with samples taken from the ADC peripheral. The use of a fractional datatype is very helpful because the ADC peripheral supports outputting data in a fractional format – so once the ADC is correctly configured, we will not need to modify the data coming from the ADC before placing in this buffer. That will save lots of processing time.

`fractional output[]`

This array is used to store the output of the conversion from the FFT function's complex data output and a squared magnitude representation, performed by the function `SquareMagnitudeCplx()`. The output buffer holds the data that we will use in our subsequent calculations. We make use of the square of the magnitude rather than calculate the magnitude itself, as the square-root function required to do this is very time consuming. For our applications, where we are only interested in finding the peak frequencies, it's not necessary to perform the square root computation. We used exactly the same technique in our speed camera detector project back in January 2005 to provide a fast display update.

That short list of variables is followed by a simple `main()` function, which performs all the function calls. It's actually quite straightforward, and the list of functions called is very short:

- Copy the input data to the `sigCmpx`
- Call the FFT
- Compute square magnitude
- Find the peak frequency

A point of interest is how the fractional input array is copied to the `fractcomplex sigCmpx` array. Two steps are involved; the imaginary component in `sigCmpx` is set to zero, and the real component is set to half of the input data value. This is a requirement of Microchip's FFT function and is mentioned in the DSP library help file. Although a fractional variable can store values between –1.0 and +1.0, the FFT function requires that data be scaled down to between –0.5 and +0.5. A simple shift-right operation performs this.

Running the sample application

Returning to the configuration bits, we selected the internal RC oscillator as the source, and enabled the PLL (phase-locked loop) to multiply the RC oscillator's output to 140MHz, giving us the highest processing speed of 70 million instructions per second. We also added code within the `main()` function to toggle our LEDs, enabling the use of an oscilloscope to measure the execution time of the FFT conversion. The breadboard setup, now connected to the IDE and ready to go, is shown in Fig.2, and the schematic in Fig.3. Note that we do not have an external power supply – the PICkit 3 can provide the power itself.

We were delighted to find that the full process of copying data, FFT, square magnitude and peak detection took just under one millisecond. This is fantastic, because even when we add the time required to perform 512 ADC samples, we will be running fast enough to be able to update a display several times a second. So, our tiny microcontroller is quite a powerful beast!

Final thoughts on FFT

Although we have discussed the magic of the FFT (Fast Fourier Transform), we've not yet spoken about the relationship between signal sample rate and FFT block size, and how these two relate to the values that are output by the FFT. Hopefully, with the aid of Fig.4 we can explain that.

At the top of Fig.4 we have our input signal – in this case, a nice simple sinewave. This is being sampled at a periodic interval. Eight samples are presented to the FFT algorithm, so the block length in this example is eight. (In our code we are using 512 samples, but this example shows just eight for clarity.) With eight samples, our sampling window (T_w) is eight times the ADC sample rate.

The FFT provides eight outputs – the same number as the number of inputs. Each output is a complex number,

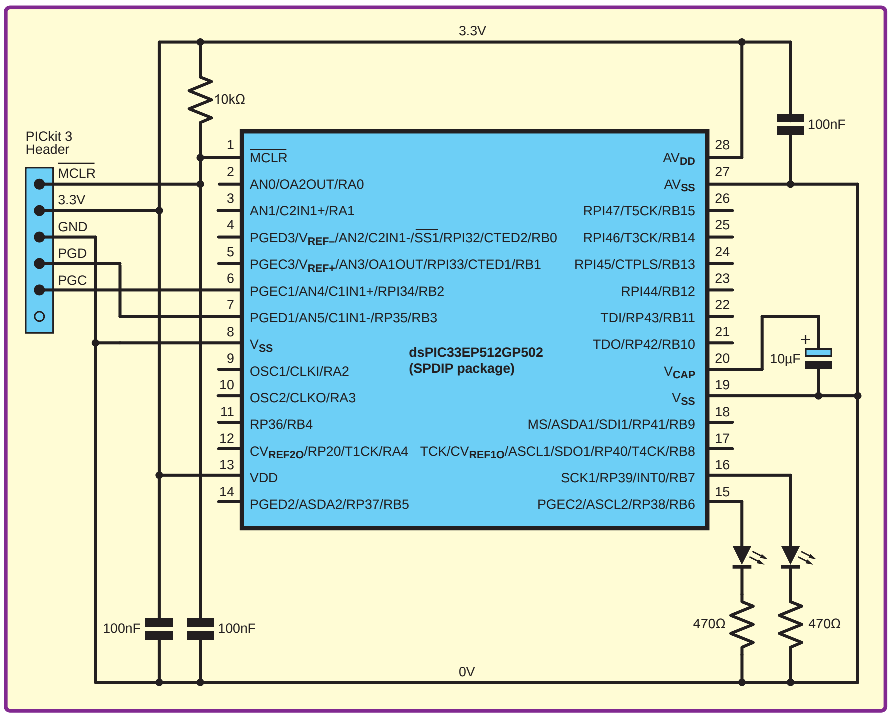

Fig.3. Breadboard schematic

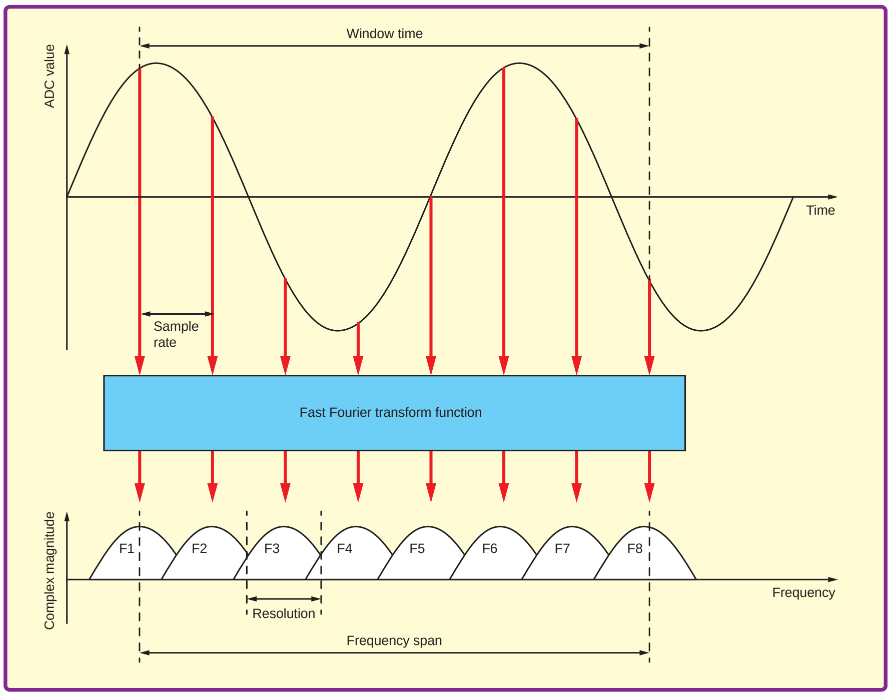

Fig.4. FFT conversion

denoting a specific frequency, and its magnitude. Each of these outputs is called a 'bin', as it contains the magnitude of a range of frequencies, the range being the resolution of that bin. Each bin spans a frequency defined by $1/T_w$.

What about trying a real-world example? Well, next month we will be listening to the sound of guitar strings. Let's make a rough assumption that the maximum frequency we are interested in resolving is 1kHz. If that is the maximum, then Nyquist's theory says we need to sample at twice that rate, so our sample rate will be 2kHz. We have an FFT block size of 512, so our window size is 256ms. Given that the bandwidth of each bin is defined as $1/T_w$, each bin spans a frequency range of 3.9Hz. Probably good enough for tuning a guitar – but we are not experts, so do feel free to comment!

If better resolution is required, then clearly that means spending more time taking samples, and having a bigger FFT, which will take longer to compute. So, there is a delicate balance between accuracy and response time.

Next month

In next month's article we expand the circuit to include audio input from an electret microphone, so we can collect data from the real world, and attempt to create a guitar tuner. The choice of Electret microphone is largely uncritical; we are using the cheapest one available from Farnell – part number 2066501. Costing less than a pound and with free next day delivery, it is simply not worth rummaging around to find an old discarded microphone with unpredictable performance. We are not looking for high audio quality here, so the cheapest option will be perfectly acceptable.

The other circuit change we will have to add next month will be correctly scaling the input signal from the microphone to match the input range of the microcontroller's ADC. If the output from the microphone is a few hundred millivolts, then we will need to amplify it by a factor of 10 or more. To achieve this, we plan to make use of a novel peripheral integrated into the microcontroller – an op amp, shown in Fig.5. It would seem that Microchip have included this for exactly the purpose we are intending, to reduce the circuit component count by two components, a dedicated op amp IC plus resistor. Once again, we would not be using this if we were looking for a high quality op amp, but given our project needs, it should be spot on. This is the first time we have used an op amp integrated into a microcontroller, so next month is going to be exciting!

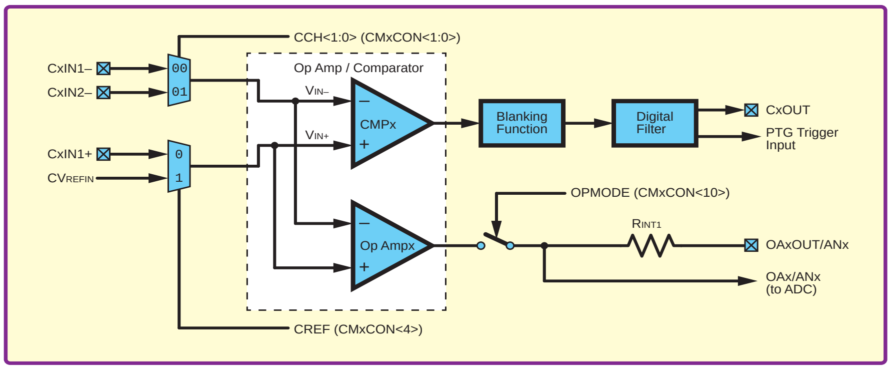

Fig.5. DSPic OP-AMP peripheral

If you are wondering how you will follow this article series along if you do not own a guitar – which most of us do not – fear not. We will be using a freely available audio signal generator program for the PC to create the guitar string sounds, available from here: www.ringbell.co.uk/software/audio.htm. And there are many different audio generators available, even for smart phones. So testing the circuit will not be difficult. We have several guitars available, so this project will be getting a genuine real-world test next month.

Looking to advertise?
Contact
Stewart Kearn on:

01202 880299
or email
stewart.kearn@
wimborne.co.uk

Laser *Why tolerate when you can automate?*

X-10 Home Automation

KAT5 AV transmission and IR control system

Barix Ethernet based MP3, communications and control systems

www.laser.com

Integrators, Installers, Trade and Retail customers welcome

C-Bus Shop

C-Bus and C-Bus Wireless

www.cbus-shop.com

Laser Business Systems Ltd
Tel: +44 (0) 20 8441 9788
Fax: +44 (0) 20 8449 0430
Email: info@laser.com
16 Garthland Drive, EN5 3BB