# FXCore Preprocessor

The FXCore preprocessor program FXCorePreProc is designed to allow programmers to include functions from libraries in their FXCore programs. Function calls are expanded to the associated FXCore assembly code and passed parameters are placed into the code.

In cases like this an example can greatly speed understanding.

We start with a test program, pitch_example.fxc which has the following:

```
.rn    in_sig mr0
.rn    pitched mr1

@fxcdsp.get_in(in0, in_sig)
@fxcdsp.pitch0(pot0, in_sig, pitched)
@fxcdsp.send_out(out0, pitched)
```

Running the preprocessor on the file with the following command (Windows example):

.\FXCorePreProc.exe -l .\ pitch_example.fxc

Will produce a file called pitch_example.fxo that looks like:

```
.rn    in_sig mr0
.rn    pitched mr1

// @fxcdsp.get_in(in0, in_sig)
CPY_CS R0 , IN0    // from library: fxcdsp -- subroutine: get_in -- matching IN with IN0 type
ADC --
CPY_MC IN_SIG , R0    // from library: fxcdsp -- subroutine: get_in -- matching SIG with
IN_SIG type MREG --
// end inclusion library fxcdsp --  subroutine get_in

// @fxcdsp.pitch0(pot0, in_sig, pitched)
.MEM PDELAY_5 4096    // from library: fxcdsp -- subroutine: pitch0 --
CPY_CS R0 , POT0    // from library: fxcdsp -- subroutine: pitch0 -- matching SHFT with POT0
type POT --
WRDLD R1 , 0XC000    // from library: fxcdsp -- subroutine: pitch0 --
ADDS R0 , R1    // from library: fxcdsp -- subroutine: pitch0 --
WRDLD R0 , 0XFFF0    // from library: fxcdsp -- subroutine: pitch0 --
MULTRR ACC32 , R0    // from library: fxcdsp -- subroutine: pitch0 --
JGEZ ACC32 , OK_5    // from library: fxcdsp -- subroutine: pitch0 --
SLS ACC32 , 1    // from library: fxcdsp -- subroutine: pitch0 --
OK_5: CPY_SC RAMP0_F , ACC32    // from library: fxcdsp -- subroutine: pitch0 --
CPY_CM ACC32 , IN_SIG    // from library: fxcdsp -- subroutine: pitch0 -- matching SIG_IN with
IN_SIG type MREG --
WRDEL PDELAY_5 , ACC32    // from library: fxcdsp -- subroutine: pitch0 --
PITCH RMP0|L4096 , PDELAY_5    // from library: fxcdsp -- subroutine: pitch0 --
CPY_MC PITCHED , ACC32    // from library: fxcdsp -- subroutine: pitch0 -- matching SIG_OUT
with PITCHED type MREG --
```

```
// end inclusion library fxcdsp --  subroutine pitch0

// @fxcdsp.send_out(out0, pitched)
CPY_CM R0 , PITCHED    // from library: fxcdsp -- subroutine: send_out -- matching SIG with
PITCHED type MREG --
CPY_SC OUT0 , R0    // from library: fxcdsp -- subroutine: send_out -- matching OUT with OUT0
type DAC --
// end inclusion library fxcdsp --  subroutine send_out
```

This resulting FXCore file pitch_example.fxo can be run through the FXCore assembler and loaded into the FXCore DSP to run.

A listing file called pitch_example.fpl showing the mapping from passed parameters to those used in the function will also be produced.

So how did we get from this code from the pitch_example.fxc file? Examining pitch_example.fxc we start with:

```
.rn    in_sig  mr0
.rn    pitched mr1
```

These MREGs are used to store and pass parameters between functions. Parameters to functions can be different types and you must pass the proper types as required by the function parameters. The preprocessor does not check types, it is up to the programmer to ensure they are passing the proper type. The assembler will check that the proper type was ultimately passed to the final FXCore instruction like normal. We will describe later how to find the types function parameters require.

The next line:

```
@fxcdsp.get_in(in0, in_sig)
```

reads in0 and saves it to the MREG in_sig which is mr0 as defined with the .rn at the top of the file. Breaking this line apart:

@ : indicate to the preprocessor this is a function call
**fxcdsp** : the name of the library that contains the function we want. The file name of this library will be fxcdsp.fxl
**get_in** : the name of the function we want, in this case it simply reads an input into an MREG
**in0** : The ADC input we want to read
**in_sig** : the MREG to save the input to

The next line:

```
@fxcdsp.pitch0(pot0, in_sig, pitched)
```

Breaks down the same way and is calling a +/-1 octave pitch shifter using RAMP0 and using POT0 as the control. in_sig is the input signal to the pitch shifter and pitched is the output.

Finally the line:

```
@fxcdsp.send_out(out0, pitched)
```

outputs the pitched signal to out0.

The command line:

.\FXCorePreProc.exe -I .\ pitch_example.fxc

Can be broken down as:

**.\FXCorePreProc.exe** : the location and name of the program.

**-I .** : the location of the library files. All library files must be in the same location. There is a space between the “-I” and the “.\” in this example

**pitch_example.fxc** : the source program to run through the preprocessor, the output will be pitch_example.fxo

One of the primary issues when using a library is what are the functions available and what are the parameters to those functions and their type. Running FXCorePreProc.exe with the -d option and providing the full name of the library file like:

.\FXCorePreProc.exe -d .\fxcdsp.fxl

Will result in an output like:

```
Library name: fxcdsp
Library description: Basic library of routines for the FXCore DSP from Experimental Noize
Subroutines found:

-----------------------------------
Subroutine name: get_in
Subroutine description: ADC in to a MREG.

Parameter name: in
Parameter type: ADC
Parameter description: ADC to read like in0 , in1, etc..

Parameter name: sig
Parameter type: MREG
Parameter description: MREG to save ADC into

-----------------------------------
plus lots more lines of subs and parameters…
```

We see the function name, get_in, as we used in the pitch_example.fxc file we wrote along with the parameters we need to pass and their type. The first one, in, is an ADC type so we used in0 and the second one is the MREG to put the signal in. You could also look at the library file directly as they are just text files in xml format.

At this point we see it is a simple procedure to include a function in your code using the format:

```
@library_name.function_name(parameter1, parameter2, . . .)
```

You can use multiple libraries in your program and different libraries can even have functions of the same name, just make sure to use the correct parameters with a specific library and function.

The library file is an XML file ending in a .fxl extension and in the format:

```
<library>
	<name>lib_name</name>
	<desc>description of the library</desc>
	<sub>
		<name>function_name</name>
		<desc>function description</desc>
		<param>
			<name>parameter name used below, replaced by the passed name</name>
			<type>type such as ADC, DAC, MREG, etc</type>
			<desc>description of the parameter</desc>
		</param>
		<param>
			<name>next parameter name</name>
			<type>next parameter type</type>
			<desc>description of next parameter</desc>
		</param>
		<code>
			the FXCore code goes here
		</code>
	</sub>
	more functions can go here in <sub></sub> tags
</library>
```

Note that all the node names such as library, name, etc. are required and must be lower case as shown as XML node names are case sensitive. “lib_name” must match both the name as used in your code and the file name of the library. Function names and parameter names must match as used in your code but are not case sensitive.

Looking at the start of the fxcdsp.fxl we see it follows this format:

```
<library>
	<name>fxcdsp</name>
	<desc>Basic library of routines for the FXCore DSP from Experimental Noize</desc>
	<sub>
		<name>get_in</name>
		<desc>ADC in to a MREG.</desc>
		<param>
			<name>in</name>
			<type>ADC</type>
			<desc>ADC to read like in0 , in1, etc..</desc>
		</param>
		<param>
			<name>sig</name>
			<type>MREG</type>
			<desc>MREG to save ADC into</desc>
		</param>
		<code>
```

```
                cpy_cs  r0, in
                cpy_mc  sig, r0
            </code>
        </sub>
        more subs and finally the </library> tag
```

We see the code block and the names used, in and sig, are the names in the <param> nodes. Looking at the processed code in the pitch_example.fxo file we see the code block has replaced the `@fxcdsp.get_in(in0, in_sig)` line:

```
// @fxcdsp.get_in(in0, in_sig) // read in0 and put it in in_sig
CPY_CS R0 , IN0    // from library: fxcdsp -- subroutine: get_in -- matching IN with IN0 type
ADC --
CPY_MC IN_SIG , R0    // from library: fxcdsp -- subroutine: get_in -- matching SIG with
IN_SIG type MREG --
// end inclusion library fxcdsp --  subroutine get_in
```

We see that “in” from the XML code block was replace with “in0” and “sig” was replaced with “in_sig”.

The original calling line is included as a comment and the library name and function name have been appended to the end of each line as a comment to allow you to track the code and parameter substitution.

You may also notice that when you run the “-d” option on a library it is pulling the information from the name, type and desc nodes so it is always best to include a description in the desc nodes. While it could be left empty like <desc></desc> it is best to always have at least a basic description.

The code blocks in the <code></code> nodes may consist of normal FXCore assembly code, jumps/labels, .mem and .sfr statements. Note that .rn and .equ statements are not allowed in the code blocks, these should only exist in the top level code. Equations may be used in the code block and parameter substitution will occur (requires at least V1.5.0 of preprocessor).

When jumps and memory declarations are used their name is updated by appending _NN to them where NN is the line number in the original .fxc program that the function was called from. As an example, looking at the pitch0 function in fxcdsp.fxl file we see both a .mem and a jump, both were updated to add _5 to their names when they were written to the pitch_example.fxo file as the function call was on line 5 of the original pitch_example.fxc file.

It is recommended that MREGs be used to pass parameters and store values while leaving core registers free to be used in the code blocks.

It is recommended that users NOT edit others library files, instead copy any routine you wish to modify to your own library and change it there. This has two major advantages:

1. The original library writer can release new versions which you can include in your tool chain without accidently overwriting any custom change you made to a function.
2. If you share your program then others will not see any problems as the library names will be different. They will need a copy of your library with the modified routine but it will avoid confusion with the original library.

**Version History**

V1.0.0 – 6 Jan 2023 : Initial release

V1.1.0 – V1.4.5 : Internal test versions

V1.5.0 – 23 Jan 2023 : Updated to allow use of parameters in equations, typo fixes, fixed help showing old format of generate assembly file name.