Threading an FFT for Parallel Processing in a Plug-In
Will Pirkle

FFT Processing (or Frequency Domain Processing) is a complete DSP topic on its own - thick books have been written about using the frequency domain representation of a signal for analysis or processing. Processing audio data in the frequency domain means taking a FFT of the audio input x(n), processing the FFT's results transforming the data somehow, then taking the inverse FFT on *that* to produce the output y(n). You can find lots of books written on just this subject.

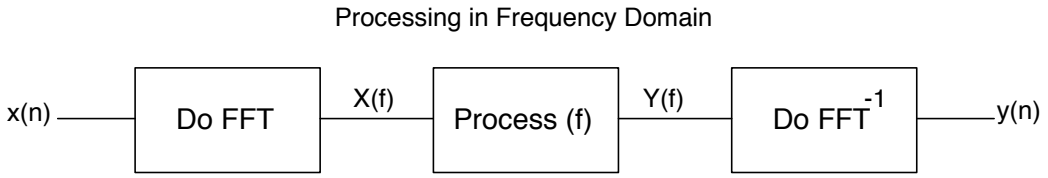


Figure 2.1: Straight FFT Processing

Another use of the FFT is for signal analysis. In the simplest case, the signal's frequency components are simply displayed on a graph or plot or LED meters as the MTFFT Plug-In does. However, more complex filters can be designed adaptively so that information from the FFT can be used to control parameters of the signal processing in real-time. For example, you might use the FFT to analyze the signal and find the highest amplitude frequency component, then use that to tune a notch-filter's center frequency to remove that component (See App Note 3).

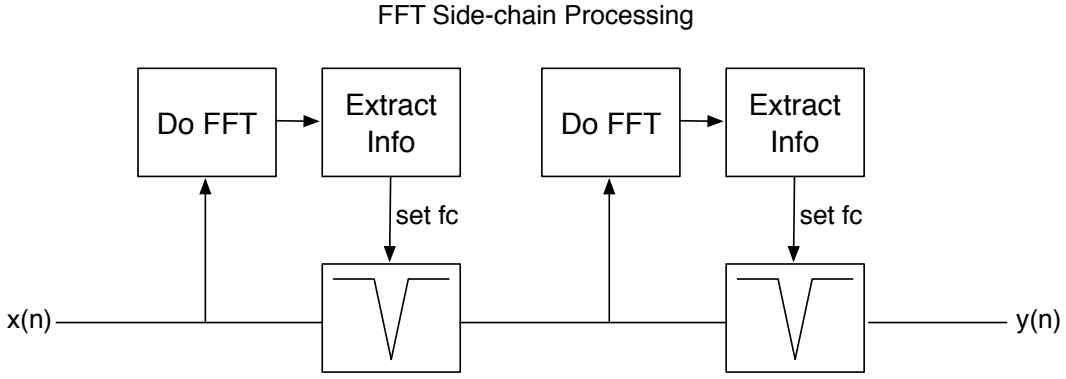


Figure 2.2: Using the FFT in a side-chain to analyze the audio and adaptively set the notch frequency

In both cases the FFT operation works on *blocks* of data, not single samples. Therefore, these blocks have to be acquired first, then the FFT operation can be performed. Unfortunately, that operation is expensive on the CPU. While the FFT is taking place, audio input samples will keep piling up. We need a way to acquire input samples into blocks while simultaneously performing the FFT on the *last* block we acquired. In other words, we need to have a parallel operation going on and we are going to do that by creating a worker thread to do the FFT while our processAudioFrame() thread buffers up samples for the *next* FFT operation.

Threads

A *thread* (or *thread of execution*) is an instruction pointer moving through code and executing instructions in sequence. In the early days, applications could only have one thread of execution at a time. You probably wrote programs like that as your first C or C++ examples. The main() function executed instructions in sequence, following branching as expected. If user input was required (e.g. “Press Enter to continue…”) the application simply waited for the input. It did not process other information while waiting.

GUI-based operating systems changed that. These systems were user driven so that the user’s control of the GUI would process information in some way. The user interaction with the GUI runs on one thread while the data processing would run on another thread, seemingly in parallel.

If you have written RackAFX Plug-Ins, you have already written a Multi-threaded DLL (really! check the Visual Studio Properties for any RackAFX Plug-In). The reality is that you must write a multi-threaded DLL because there are two separate threads attacking your Plug-In at any time. The RackAFX audio subsystem is constantly calling your processAudioFrame() function on one thread while the RackAFX GUI subsystem is calling your userInterfaceChange() method when the user moves a control; it is also calculating the control variable values for you prior to that call.

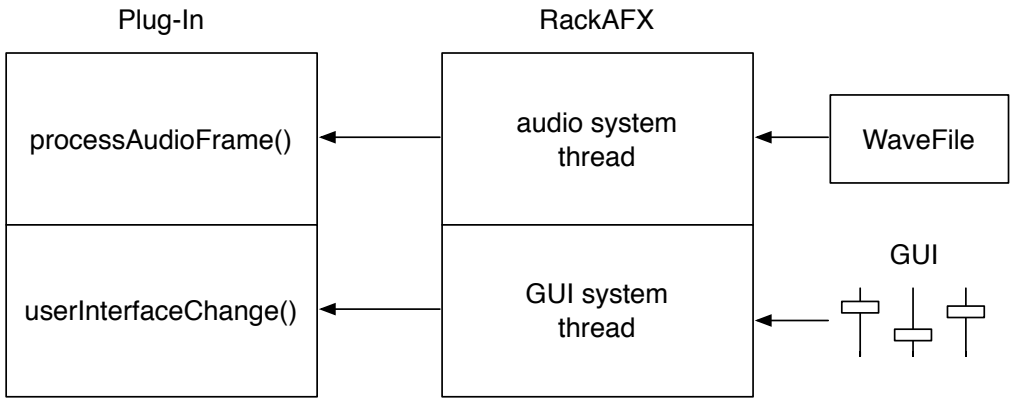


Figure 2.3: Your Plug-In is attacked by two threads from the RackAFX client

You might be aware that when you run multiple applications on your computer on a single CPU machine, they aren’t really running at the same time or in parallel. The OS virtualizes the CPU and shares it with the applications, going in a round-robin fashion between the apps letting each have a little slice of CPU processing time. Each app thinks its running continuously but in reality, each app is being put to sleep while the CPU is letting another app use it.

But in more detail, the CPU is actually doling out its time to each ***thread of execution*** for each application. This means that if we create threads to do processing, we can get a little more CPU time for that processing.

Worker Threads

The simplest kind of thread is a *worker thread*. It turns out to also be exactly what we need for easy parallel FFT processing. When you create a worker thread, you:

- first setup a ***callback*** function. You tell the OS to call this function whenever the thread gets its slice of CPU time
- start the thread
- the callback function will be called over and over and over repeatedly on each CPU slice
- when you close the thread, it kills the operation and the callback will never get called again

A problem is that the callback function can not be a member of a C++ object. This poses the same persistence of data problem that I explain in my book, Chapter 2 (Anatomy of a Plug-In). The callback function is a C-style function. Any variables you declare inside this function will cease to exist after the function hits the closed curly-bracket at the end. Most likely we need to share information with the callback function so we need a mechanism to let the callback function have access to this data.

The callback function for a Windows worker thread MUST have this implementation prototype:

```
DWORD CALLBACK YourCallbackFunctionName(LPVOID lpThreadData)
{
	// your thread code here...

	// …

	// the return 0 indicates success
	return 0;
}
```

- You can name your function whatever you want as long as it is a legal C function name.
- You do not have to prototype the function in your .h file, just implement it in your .cpp file.
- a return value of 0 indicates success; returning a non-zero value will halt the thread
- there is only one argument, **LPVOID lpThreadData**

That final bullet point is critically important: We will be passed a **void*** during the callback (LPVOID is defined as a 32-bit pointer to a void datatype). The void* is required because the OS has no idea what kind of information the callback will need. We are allowed to setup this pointer to point to information we want to use on each callback.

When you create the worker thread you tell it:
- the name of the callback function
- a pointer to anything you want; this pointer will be cast (or cloaked) as a void* prior to calling the callback; you must un-cloak it by casting it back to what it originally was
- there are some other thread setup parameters too, but we won't need them for our project

Choosing a pointer for lpThreadData

Traditionally, you setup a custom data-structure (a struct) that has all the information needed for the thread. The struct can contain anything legal - pointers to other structs, pointers to objects, static variables, etc… whatever you want. For example, suppose a worker thread was going to operate on some data in a floating point buffer. We might need to keep track of the start and end locations of the buffer processing operation. We might define the struct like this:

```
struct ThreadProcessData
{
    int    nStartIndex;       /* start location in buffer */
    int    nEndIndex;         /* end location in buffer */
    float  *pBuffer;          /* pointer to a buffer to process */
};
```

Prior to creating the worker thread, we create a new instance of this struct off the heap and initialize it. This might look like:

```
// create the struct off the heap
ThreadProcessData *pThreadData = new ThreadProcessData;

// initialize it
// 1) create the 1024 point audio buffer
pThreadData->pBuffer = new float[1024];

// 2) setup the index values
pThreadData->nStartIndex = 0;
pThreadData->nEndIndex = 511;
```

This struct* is now ready to be passed to the thread creation mechanism. You will see this kind of use of the custom structure a lot in audio callback functions.

We could also create a custom C++ object to encapsulate the data and provide functions to operate on that data. That would be great in the C++ way because the thread code is simple - it just calls functions on the object to do the processing.

However, we are going to be more clever in our Plug-In (or more dangerous depending on how you look at it). Instead of creating the custom structure or custom object, we are going to use the Plug-In's ***this*** pointer (please don't send email to me about this…). ***That way, the thread callback function can call methods directly on our Plug-In object. We can protect the data itself by using the access operators (private or protected) if we want to. This is a convenient way to share data between our processAudioFrame() thread and the worker thread.***

Creating the Worker Thread

In Windows this involves one call to the OS called CreateThread(); the function looks like this:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES       lpsa,
  DWORD                       cbStack,
  LPTHREAD_START_ROUTINE      lpStartAddr,
  LPVOID                      lpvThreadParam,
  DWORD                       fdwCreate,
  LPDWORD                     lpIDThread
);
```

lpsa
[in] Ignored. Must be NULL.

cbStack
[in] Ignored unless the STACK_SIZE_PARAM_IS_A_RESERVATION flag is used. In that case, this parameter specifies the virtual memory reserved for the new thread.

lpStartAddr
[in] Long pointer to the application-defined function of type

LPTHREAD_START_ROUTINE to be executed by the thread; represents the starting address of the thread.

lpvThreadParam
[in] Long pointer to a single 32-bit parameter value passed to the thread.

fdwCreate
[in] Specifies flags that control the creation of the thread.

lpIDThread
[out] Long pointer to a 32-bit variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

We are going to only need to set the two most important variables in **bold** above:

```
LPTHREAD_START_ROUTINE  - the name of our callback
lpvThreadParam          - a pointer to the data we will operate with
```

The other parameters do not need to be adjusted. You can optionally get the ThreadID value in the last (out) parameter - we don't need it.

The return type of this callback is a HANDLE. In Windows, a HANDLE is a 32 bit value that is used to identify something. It can identify about anything from a Window to a running thread. In this case, it's the thread HANDLE. We will store this value to use when we close the thread thereby terminating it.

Example 1

Suppose we have defined a callback function as:

```
DWORD CALLBACK ThreadProc(LPVOID lpThreadData)
{
      // your thread code here...

      // …

      // the return 0 indicates success
      return 0;
}
```

And, we are going to use that custom stuct that we created above:

```
// create the struct off the heap
ThreadProcessData *pThreadData = new ThreadProcessData;

// initialize it
// 1) create the 1024 point audio buffer
pThreadData->pBuffer = new float[1024];

// 2) setup the index values
pThreadData->nStartIndex = 0;
pThreadData->nEndIndex = 511;
```

The final step is to create the thread. That would look like this:

```
HANDLE hThread = CreateThread(NULL, NULL, ThreadProc, pThreadData, NULL,
                              NULL);
```

Notice the ThreadProc is the literal name of the function; there is no & (address of) operator needed. The pThreadData is also written literally; you do not need to cast it to a void* prior to calling the CreateThread () method (but you certainly could if you wanted). Calls to the the callback will cloak the pointer as a void* so you need to un-cloak it in your callback like this:

```
DWORD CALLBACK ThreadProc(LPVOID lpThreadData)
{
      // uncloak the void* by casting
      ThreadProcessData* pData = (ThreadProcessData*)lpThreadData;

      // use the pointer
      float f = pThreadData->pBuffer[nStartIndex];

      // etc… do something

      // the return 0 indicates success
      return 0;
}
```

Example 2

In our case, we will pass the plug-in's ***this*** pointer as the LPVOID argument. We will store the resulting thread HANDLE on a C++ member variable. Assume we use the same callback function name and the CreateThread call looks like this:

```
m_hThread = CreateThread(NULL, NULL, ThreadProc, this, NULL, NULL);
```

Now, in the callback, we uncloak the pointer to do stuff:

```
DWORD CALLBACK ThreadProc(LPVOID lpThreadData)
{
	// uncloak the void* by casting
	MyPlugIn* pPlugin = (MyPlugIn*)lpThreadData;

	// use the pointer
	pPlugin->doFFT();

	// etc…

	// the return 0 indicates success
	return 0;
}
```

Thread Synchronization

One of the most complicated aspects of multithreading is synchronizing the operation of the threads. Remember, as soon as the CreateThread() method is executed, the callback function will get called repeatedly. It will not be synchronized with RackAFX calling our processAudioFrame() thread; we can't use counters or timers to try to synchronize the operation. What is going on in each thread that we need to keep in sync? Let's start at the beginning of Plug-In operation

- processAudioFrame() thread buffers up samples for the FFT
- after acquiring one buffer, it then signals the FFT thread to start operating on that data
- processAudioFrame() thread then buffers up samples for the next FFT, in parallel with the worker thread
- after acquiring the next buffer, it checks to see if the last FFT is done processing and if so, it starts the next FFT
- if the FFT is still processing, audio samples will be dropped while we wait for the thread to finish; we'll call this a ***FFT Over-run***

So, we need two signals here:

- the processAudioFrame() thread needs to **signal** the FFT thread when it's time to start
- the FFT Thread needs a way to set a **signal** so that the the processAudioFrame() thread knows its OK to start another FFT

The term **signal** is bolded for a reason - it encapsulates the idea that a signal is information about something, in this case the states of our threads. We also see that it appears one thread can control another; the process thread starts the FFT operation. Additionally we see that threads have to **wait** on their respective signals:

- the FFT thread waits until the **FFT Start** signal is set
- the process thread waits until the **FFT Done** signal is set

Thread synchronization in Windows can be done with several OS mechanisms:
- Events
- Semaphores
- Mutexes
- Critical Sections

It is out of the scope of this App Note to discuss all of these systems. We will be using **Events** to handle our thread synchronization. When you setup an Event you tell the OS to create a signaling system for you. The signals are like flags that are either in a signaled state (called **set**) or in an un-signaled state (called **reset**).

Creating and Describing the Event

You describe your signaling system by telling the OS two important details:
- the initial state of your signal - signaled or un-signaled
- what to do when the signal is set: automatically reset it or force a thread to manually reset it

When you setup the Event, the OS will return a HANDLE to that event. You will use this handle to check the state of your signals. The complete function is here (bolded parameters are our important ones):

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES
  lpEventAttributes,
  BOOL bManualReset,
  BOOL InitialState,
  LPTSTR lpName
);
```

```
lpEventAttributes
[in] Ignored. Must be NULL.

bManualReset
[in] Boolean that specifies whether a manual-reset or auto-reset event object
is created. If TRUE, then you must use the ResetEvent function to manually
reset the state to nonsignaled. If FALSE, the system automatically resets the
state to nonsignaled after a single waiting thread has been released.

bInitialState
[in] Boolean that specifies the initial state of the event object. If TRUE,
the initial state is signaled; otherwise, it is nonsignaled.

lpName
[in] Pointer to a null-terminated string that specifies the name of the event
object. If lpName is NULL, the event object is created without a name.
```

For the MTFFT Plug-In I chose to create two slightly different events; you could re-design it other ways as well, but in my Plug-In you will see both versions of the two important variables (manual vs automatic reset and signaled vs un-signaled initial state).

FFT Start:
- We want our FFT Start signal to start off un-signaled.
- The FFT Thread will have to wait until this signal is set before processing a FFT.
- Once the signal is set, the event automatically resets it.
- the process thread will manually start this event when a buffer is acquired

FFT Done:
- We want our FFT Done signal to start off signaled so that the very first FFT will happen immediately after the first buffer is acquired
- The FFT thread will manually set this event when the FFT is done
- the process thread will wait for this event before launching the next FFT
- the process thread will manually reset the event after the FFT Start event is set

Examples:

- Create an event that is initially un-signaled and will automatically reset itself once signaled:

```
m_hFFTStartEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

- Create an event that is initially signaled and must be manually reset.

```
m_hFFTDoneEvent = CreateEvent(NULL, TRUE, TRUE, NULL);
```

Checking the Signal

Checking the signal is not done the same way you might evaluate a boolean flag directly. For Events the concept is that you are “waiting for an event” or “waiting for an object” to become signaled. You have 3 basic choices in how you wait:
- do not wait, check the signal right now
- wait on the event to signal for a certain duration, after that give up and quit waiting
- wait on the event to signal forever

All three of these are done with the same function WaitForSingleObject().

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);

hHandle
[in] Handle to the object.

dwMilliseconds
[in] Specifies the time-out interval, in milliseconds. The function returns
if the interval elapses, even if the object's state is nonsignaled.
```

IfdwMilliseconds is zero, the function tests the object's state and returns immediately. If dwMilliseconds is INFINITE, the function's time-out interval never elapses.

The return type is:
WAIT_FAILED
Indicates failure for one of the following reasons:

- There is insufficient memory in the system.
- Two threads are waiting on the same interrupt event.
- The current thread is being terminated.
- The underlying object represented by hHandle has been deleted.
- The handle is invalid.

WAIT_OBJECT_0
The state of the specified object is signaled.

WAIT_TIMEOUT
The time-out interval elapsed, and the object's state is nonsignaled.

Examples:

The last function parameter controls the wait-type. Here are examples of use (say we already have a handle t the events):

- Do not wait, check signal immediately

// wait for 0 milliseconds
```
DWORD success = WaitForSingleObject(hEvent, 0);
```

The signal is evaluated and the function returns immediately; the success code tells you the signal state.

- Wait for up to 1 second on this signal

// wait for 1000 milliseconds
```
DWORD success = WaitForSingleObject(hEvent, 1000);
```

The signal is evaluated repeatedly for up to 1000mSec. During that time the instruction pointer is stuck on this line of code; the success code tells you the signal state. If the signal occurs before the timeout, the instruction pointer will then move to the next instruction and stop the wait.

- Wait forever on this signal

// wait forever
```
DWORD success = WaitForSingleObject(hEvent, INFINITE);
```

The signal is evaluated repeatedly forever. During that time the instruction pointer is stuck on this line of code; it might be stuck there forever. The instruction pointer will only move to the next instruction when the signal has been **set**.

Manipulating the Signal

You can manipulate the state of an event's signal by using the functions:

```
BOOL SetEvent(
  HANDLE hEvent
);

BOOL ResetEvent(
  HANDLE hEvent
);
```

In both cases hEvent is the event HANDLE that was returned when you created the event.

You need to create Events before CreatingThreads!

One thing to remember is that as soon as that CreateThread() function is called, the callback will be repeatedly called by the OS. Since the callback will be using Events to know whether to process FFTs or not, it is critical that the Events be created first, then the Threads. For our MTFFT Plug-In we are going to declare the following HANDLES:

```
HANDLE m_hFFTThread;    // for the Thread
HANDLE m_hFFTStartEvent;// for the start FFT event
HANDLE m_hFFTDoneEvent; // for the end FFT event
```

In the constructor, I create the Events first then the Thread like this; you can see that the callback function is named **FFTThreadFunc**

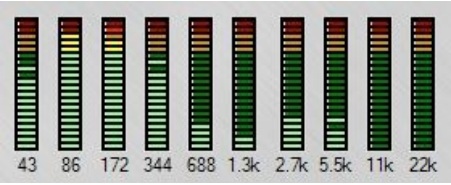
```
// create events to sync threads
//
// FALSE = auto reset, FALSE = initially NOT SIGNALED;
m_hFFTStartEvent = CreateEvent(NULL,FALSE,FALSE,NULL);

// TRUE = manual reset, TRUE = initially SIGNALED;
m_hFFTDoneEvent = CreateEvent(NULL,TRUE,TRUE,NULL);

// Create Thread
// CreateThread(Security_Attributes = NULL
//                Stack_Size = NULL
//                ThreadRoutinePtr = FFTThreadFunc
//                ThreadRoutineArgument = this pointer
//                CreationFlags = NULL
//                ThreadID = NULL)
//
m_hFFTThread = CreateThread(NULL, NULL, FFTThreadFunc, this, NULL,NULL);
```

The MTFFT Plug-In Operation

Setting up and creating the Events and Thread are pretty easy. The callback function has been named and now we are ready for coding the operation.



This Plug-In implements a simple spectrum analyzer by examining the 10 bins (a bin is an index number in the FFT array) of the FFT equally spaced across it and displaying their normalized values (0.0 to 1.0) on the LED Meters on the main GUI. The LEDs move according to the amplitudes of these bins. The FFT length (#defined as FFT_LEN) is either 1024 or 2048 points. To get our nearly 10 octaves of audio, we find the bins for the 1024 point case as (Bin 0 is DC or 0 Hz):

Frequency	FFT Magnitude Bin
43 Hz	1
86 Hz	2
172 Hz	4
344 Hz	8
688 Hz	16
1376 Hz	32
2752 Hz	64
5504 Hz	128
11008 Hz	256
22016 Hz	511

The Plug-In passes audio straight though so the signal is not affected; this is a pure analyzer plug-in where data about the signal is displayed. The block diagram looks like this:

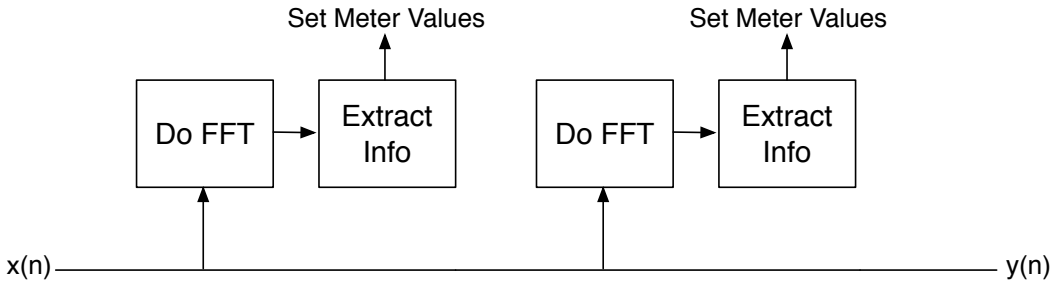


Figure 2.4: MTFFT Block Diagram shows pass-through path and FFT side-chain

The MTFFT Callback

The callback function is fairly simple because we are just going to use the Plug-In's ***this*** pointer to call an FFT function. After that function returns, the callback then sets the FFT Done Event to signal the process thread that it is ready for another one. The callback function is just three lines of code in length.

```
DWORD CALLBACK FFTThreadFunc(LPVOID lpThreadData)
{
	CMTFFT* pParent = (CMTFFT*)lpThreadData;

	// infinite worker thread loop
	while (true)
	{
		// wait forever (INFINITE milliseconds) until the signal occurs
		WaitForSingleObject(pParent->m_hFFTStartEvent,INFINITE);

		// Event will reset itself automatically after wait
		// it was declared AUTOMATIC

		// call a function on the parent; OK because RackAFX Plug-Ins are
		// Multithreaded DLLs.
		pParent->doFFT();

		// set the FFT Done Signal
		SetEvent(pParent->m_hFFTDoneEvent);
	}

	// success
	return 0;
}
```

Double Buffering the Audio

The idea behind this Plug-In is to have parallel processing going on where one thread acquires audio samples and the other processes those samples with a FFT. The way to do this is to ***double-buffer*** the input audio data. The double-buffer is twice as long as the FFT_LEN; while the FFT processes one half the buffer, the process thread is filling the other buffer.

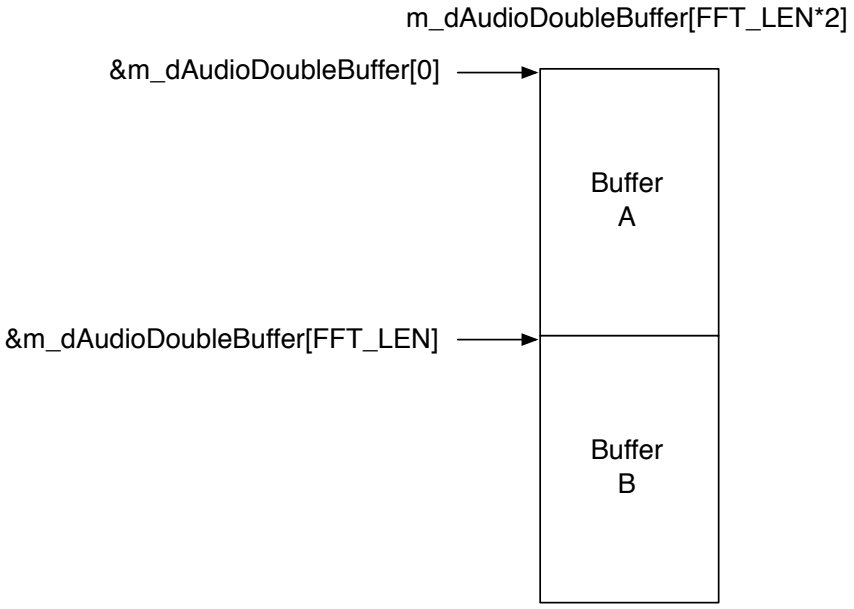


Figure 2.5: The double buffer is FFT_LEN*2 in length and is conceptually split at the center

- The address for the start of Buffer A is &m_dAudioDoubleBuffer[0]
- The address for the start of Buffer B is &m_dAudioDoubleBuffer[FFT_LEN]

We intend to follow this sequence of operation (here are the first 2 FFTs)

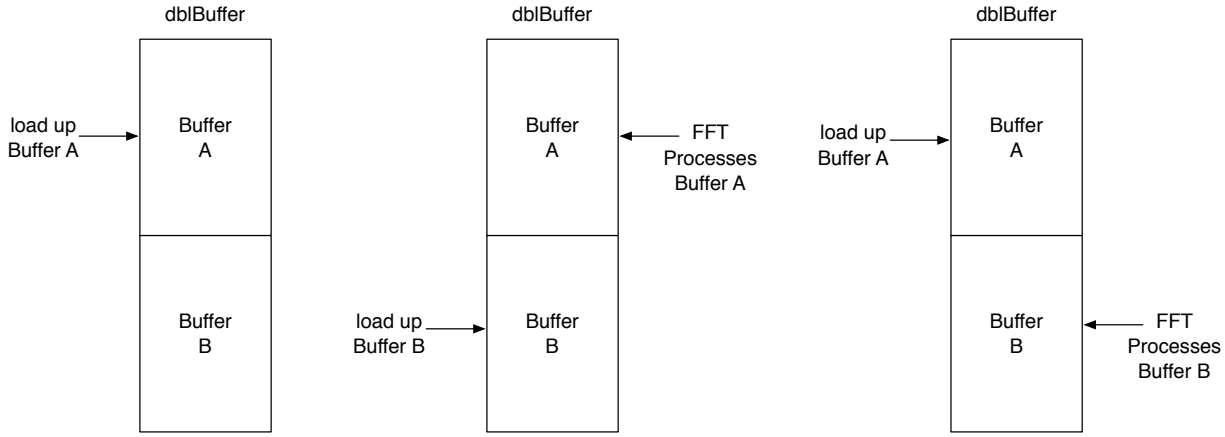


Figure 2.6: The sequence progress from left to right; first we load buffer A, then initiate a FFT on it while we load buffer B. The sequence toggles back and forth.

To get a better understanding of the signaling during the double buffering, take a look at this detailed block diagram. The Events are shown as red and green flags.

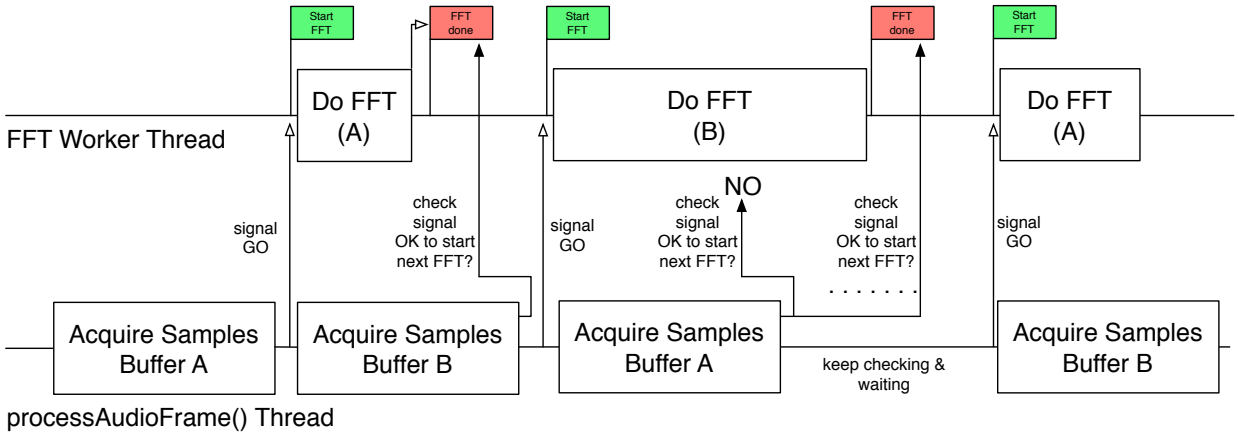


Figure 2.7: The two threads are in parallel across the page.

The sequence here is:

- acquire Buffer A
- check to see if we can start FFT
- Start FFT on Buffer A
- acquire Buffer B
- FFT (A) is already done so we launch the next one
- acquire Buffer A
- FFT (B) is not done yet
- we must wait and drop audio samples from the buffer
- FFT (B) eventually signals it is done

Plug-In Variables & Methods

Take a look at the MTFFT.h file to see the required variables and methods:

```
HANDLE m_hFFTThread;
HANDLE m_hFFTStartEvent;
HANDLE m_hFFTDoneEvent;

// counter for buffering
int m_nSampleCount;

// our double-buffer
double m_dAudioDoubleBuffer[FFT_LEN*2];

// a flag to toggle for processing A vs B
bool m_bProcessFFFTonBufferA;
```

```
// arrays for the FFT
double m_dFFT_In[FFT_LEN];
double m_dFFT_OutRe[FFT_LEN];
double m_dFFT_OutIm[FFT_LEN];
double m_dFFT_Magnitude[FFT_LEN/2];

// method for the thread to call for the FFT
void doFFT();
```

A simple boolean flag m_bProcessFFTonBufferA will be used to toggle back and forth between buffers.

processAudioFrame()

You can see the toggling and Event control here in processAudioFrame():

```
// flag for less coding
bool bAccumulatePoints =  false;

// 0 to FFT_LEN-1
if(m_nSampleCount < FFT_LEN)
{
	bAccumulatePoints = true;
}
else // check FFT
{
	// set up a success code
	DWORD dwSuccess = WAIT_TIMEOUT;

	// is FFT Done yet?	0 = wait for 0 milliseconds
	dwSuccess = WaitForSingleObject(m_hFFTDoneEvent, 0);

	// WAIT_OBJECT_0 = YES
	if(dwSuccess == WAIT_OBJECT_0)
	{
		// toggle
		m_bProcessFFFTonBufferA = !m_bProcessFFFTonBufferA;

		// manual reset
		ResetEvent(m_hFFTDoneEvent);

		// FFT is done, so we can start another one
		SetEvent(m_hFFTStartEvent);

		// reset only when ready to load another buffer
		m_nSampleCount = 0;

		// start next buffer
		bAccumulatePoints = true;
	}
	else if(dwSuccess == WAIT_TIMEOUT)
		this->sendStatusWndText("FFT Over-run.");
}
```

```
// accumulating STEREO MIX for FFT
if(bAccumulatePoints)
{
      float stereoMix = 0.5*pInputBuffer[0] + 0.5*pInputBuffer[1];

      // if processing A, load up B
      if(m_bProcessFFFTonBufferA) //starts at 1024, 1025, 1026...
            m_dAudioDoubleBuffer[m_nSampleCount + FFT_LEN] = stereoMix;
      else // load up bufferA starts at 0, 1, 2
            m_dAudioDoubleBuffer[m_nSampleCount] = stereoMix;


      m_nSampleCount++;
}

// pass thru code
pOutputBuffer[0] = pInputBuffer[0];

// Mono-In, Stereo-Out (AUX Effect)
if(uNumInputChannels == 1 && uNumOutputChannels == 2)
      pOutputBuffer[1] = pInputBuffer[0];

// Stereo-In, Stereo-Out (INSERT Effect)
if(uNumInputChannels == 2 && uNumOutputChannels == 2)
      pOutputBuffer[1] = pInputBuffer[1];
```

If the buffer has been acquired (when `m_nSampleCount == FFT_LEN`) I check the FFT Done state. I want to check this immediately and branch after that because I still have audio samples to deliver to RackAFX. If the FFT from last time is done I then:

- toggle the buffer flag
- deal with the Events, reset the Done and set the Start events
- reset the sample counter to start acquiring again

If the FFT is still busy I send a message to the Status Window in RackAFX.

During the audio sample buffering, I check to see which buffer is being processed by the FFT thread and I load the other one. Note the use of the FFT_LEN offset to load the B buffer:

```
      // if processing A, load up B
      if(m_bProcessFFFTonBufferA) //starts at 1024, 1025, 1026...
            m_dAudioDoubleBuffer[m_nSampleCount + FFT_LEN] = stereoMix;
      else // load up bufferA starts at 0, 1, 2
            m_dAudioDoubleBuffer[m_nSampleCount] = stereoMix;
```

doFFT()

Finally, look at the doFFT() function; this performs the FFT and sets the LED meter values accordingly:

```
if(m_bProcessFFFTonBufferA) // start at top of bufferA
{
	fft.fft_double(FFT_LEN, /* Length */
			false,	/* Inverse FFT? */
			&m_dAudioDoubleBuffer[0], NULL, /* In_Real, In_Imag */
			m_dFFT_OutRe, m_dFFT_OutIm); /* Out_Real, Out_Imag */

	this->sendStatusWndText("Processed FFT Buffer A.");
}
else // start at center of array [FFT_LEN]
{
	fft.fft_double(FFT_LEN, /* Length */
			false,	/* Inverse FFT? */
			&m_dAudioDoubleBuffer[FFT_LEN], NULL, /* In_Real,In_Imag */
			m_dFFT_OutRe, m_dFFT_OutIm); /* Out_Real, Out_Imag */

	this->sendStatusWndText("Processed FFT Buffer B.");
}

// Get the Magnitude of FFT
// Use FFT_LEN/2 since the data is mirrored within the array.
for(int i=0;i < FFT_LEN/2;i++)
{
	double re = m_dFFT_OutRe[i];
	double im = m_dFFT_OutIm[i];

	m_dFFT_Magnitude[i] = sqrt((re*re)+(im*im));
}

// built in helper (pluginconstants.h)
normalizeBuffer(&m_dFFT_Magnitude[0], FFT_LEN/2);

// Get Bin Info
// 43 Hz @44.1kHz  = Bin 1 = 22.050/(1024/2)
//                   = 44.100/1024
//
// NOTE: DC (0 Hz) = m_dFFT_Magnitude[0];

// for 1024 points
if(FFT_LEN == 1024)
{
	m_f43HzMeterValue = m_dFFT_Magnitude[1];
	m_f86HzMeterValue = m_dFFT_Magnitude[2];
	m_f172HzMeterValue = m_dFFT_Magnitude[4];
	m_f344HzMeterValue = m_dFFT_Magnitude[8];
	m_f688HzMeterValue = m_dFFT_Magnitude[16];
	m_f1376HzMeterValue = m_dFFT_Magnitude[32];
	m_f2752HzMeterValue = m_dFFT_Magnitude[64];
	m_f5504HzMeterValue = m_dFFT_Magnitude[128];
	m_f11008HzMeterValue = m_dFFT_Magnitude[256];
```

```
	m_f22050HzMeterValue = m_dFFT_Magnitude[511];
}
else if(FFT_LEN == 2048)
{
	// for 2048 points
	m_f43HzMeterValue = m_dFFT_Magnitude[2];
	m_f86HzMeterValue = m_dFFT_Magnitude[4];
	m_f172HzMeterValue = m_dFFT_Magnitude[8];
	m_f344HzMeterValue = m_dFFT_Magnitude[16];
	m_f688HzMeterValue = m_dFFT_Magnitude[32];
	m_f1376HzMeterValue = m_dFFT_Magnitude[64];
	m_f2752HzMeterValue = m_dFFT_Magnitude[128];
	m_f5504HzMeterValue = m_dFFT_Magnitude[256];
	m_f11008HzMeterValue = m_dFFT_Magnitude[512];
	m_f22050HzMeterValue = m_dFFT_Magnitude[1023];
}
```

Build and test the sample MTFFT code. You can put breakpoints in the callback and processAudioFrame functions to watch the event handling. You can also monitor the FFTs on the RackAFX Status Window. Try changing FFT_LEN from 2048 to 1024 and see what happens. You can try any FFR length you want but be sure you sift through the bins properly when connecting to the meters.

References:

Ifeachor, Emmanuel C. and Jervis, Barrie W. 1993. *Digital Signal Processing, A Practical Approach*, Chap. 4,6. Menlo Park: Addison Wesley.

Pirkle, Will. 2012. *Designing Audio Effect Plug-Ins in C++*, Burlington: Focal Press.