**State Diagram**

T1

T2

**WAITING**

**EMPTY**

**CORRELATION**

**PLAY**

T3

T4

States:

* WAITING: starting state. This state is used to waiting for both signal and noise phone to be online and to send buffers. Then, buffers are accumulated until T1 happens.
* CORRELATION: state where the cross-correlation between the signal and noise buffers is calculated. The maximum of the cross-correlation gives the total delay between the two samples’ flows. Then, by a simple algorithm, we divide this total delay in two variables that is important for us since we work with dynamic lists of buffers:
  + The number of buffers between the two corresponding buffers: this is the number of buffers we need to throw, either for the signal or the noise, to synchronize the buffer’s lists.
  + The delay inside the synchronized buffers: this is the number of samples we need to throw, either for the signal or the noise, to synchronize the samples.
* EMPTY: this state just removes the useless buffers we used for the cross-correlation and helps fixing the length of the dynamic lists.
* PLAY: this state is the most complex one, it can be divide as follows:
  + Initialization phase: we initialize the variables that will be used, particularly the noise, signal and “to be played” arrays. (“to be played” array is basically the same as the signal array, but of a smaller length since we need more samples on the signal array to apply the NLMS algorithm, otherwise we would miss to filter some samples)
  + Noise cancellation phase: the algorithm goes through this phase if we ask it to do it; otherwise it skips to the next phase. This phase applies the NLMS algorithm between the signal and the noise arrays. Then the result is subtracted to the “to be played” array.
  + Voice detection phase: this algorithm uses the “to be played” array (after NLMS or not) to detect if there is voice or not. If there is no voice, we set some parts of the array to zero.
  + Playing phase: we play the processed array “to be played” and observe the noise cancellation.

Transitions:

* T1: happens when we received enough buffers from the signal and the noise phones to calculate the cross correlation between the buffers
* T2: happens when the cross correlation has been calculated and the delay computed
* T3: happens when we removed the useless buffers (the one we used to calculate the cross correlation)
* T4: happens when someone press the button “Reinitialize” on the phone’s screen

**NLMS algorithm:**

This is really challenging since the buffers we use on the framework last approximately 92 milliseconds. The processing part has to be shorter than that. The NLMS algorithm we use is based on the one developed on Matlab. We translated the Matlab code to java first. The draft’s execution time was too long for our real time implementation. We used some basic tips to reduce the execution time.

Intelligent for-loops:

The for-loop executes a test on the index variable for every iteration. This test consists of calculating the subtraction between the index variable and the value we test and then compare it to 0. If we really want to reduce the execution time, we need to make that the compared value is zero and thus we won’t be calculating any subtraction, which helps if you have a lot of iterations.

Replacing:

**for**(**int** i=0;i<*bufferLength*;i++){

//Calculation

}

With:

**int** i;

**for**(i=*bufferLength*;i-->0;){

//Calculation

}

Moving every “static” calculation out the loops:

We know that it takes time when it comes to operations inside loops. There are some simple calculations are executed at every loop iteration and for which the result doesn’t change. We could simply get those calculations out the loop and assign them to a new local variable. It costs less to point at a local variable than to calculate.

Replacing:

*toFilter*=**new** **int**[*bufferLength*];

**int** i;

**int** x=10;

**int** y=2;

**for**(i=*bufferLength*;i-->0;){

*toFilter*[i]=x\*y;

}

With:

*toFilter*=**new** **int**[*bufferLength*];

**int** i;

**int** x=10;

**int** y=2;

**int** z=x\*y;

**for**(i=*bufferLength*;i-->0;){

*toFilter*[i]=z;

}

Do not instantiate inside the loops:

Everything is said in the title. If we translate Matlab code directly, we can end up with variables that are instantiated inside loops. We want to avoid this.

Replace:

**for**(i=*bufferLength*;i-->0;){

**int** x=0;

//Processing that uses x

}

With:

**int** i;

**int** x;

**for**(i=*bufferLength*;i-->0;){

x=0;

//Processing that uses x

}

It takes time to instantiate variables, this is some precious time for real-time implementation.