



HAL
open science

Callback adaptation techniques

Stéphane Letz

► **To cite this version:**

| Stéphane Letz. Callback adaptation techniques. [Technical Report] GRAME. 2001. hal-02158912

HAL Id: hal-02158912

<https://hal.science/hal-02158912>

Submitted on 18 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Callback adaptation techniques

Stéphane Letz
November 2001

Grame - Computer Music Research Laboratory
9, rue du Gare BP 1185 69202 FR - LYON Cedex 01
letz@grame.fr

Abstract

This document describes a callback adaptation technique developed for the PortAudio port on ASIO. This method handle buffers of different sizes and guarantee lowest latency added by buffer size adaptation.

1 Introduction

The callback adaptation problem was encountered whilst doing a port of the PortAudio API on ASIO API. PortAudio is a cross-platform library that provides streaming audio input and output, ASIO is a Macintosh and Windows API that provides streaming audio input and output. Both API are callback based, but may use audio buffers of different size. This document presents an algorithm developed to allow adaptation between the 2 callback systems.

2 Adapting callbacks

The system we are considering is driven by an audio native callback (the “host” callback) usually called by the hardware interrupt. The host audio callback has the following prototype :

hostCallback (void* inputBuffer, void* ouputBuffer, int frameNumber, void* userData);

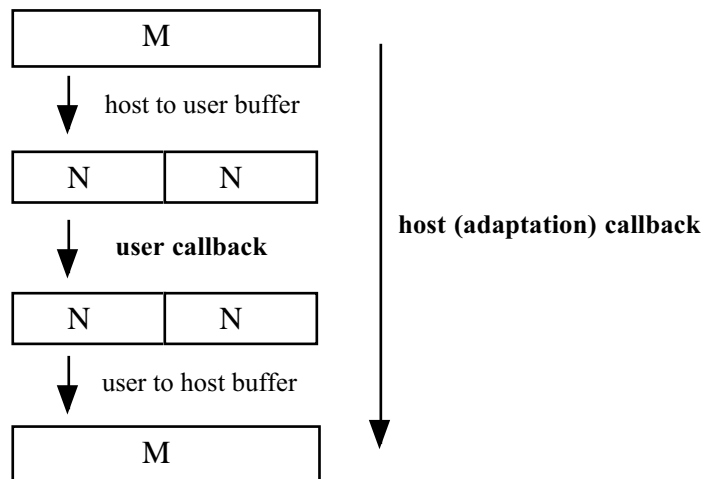
Upon return of the callback, the application has read all input data and provided all output data. InputBuffer and OutputBuffer have the same number of frames **M**.

The internal callback (the “user” callback) is implemented by the application and has the following prototype :

userCallback (void* inputBuffer, void* ouputBuffer, int frameNumber, void* userData);

where InputBuffer and OutputBuffer have the same number of frames **N**.

The user callback will be provided by the application and one need to develop the host (adaptation) callback to be given to the host audio engine. So we have two different callback and in general two different buffer sizes **M** and **N**. The running cycle is the following :



- the hardware fills the **host input buffer** of size M
- the **hostCallback** is called
- the **host input buffer** of size M is transferred to the **user input buffer** of size N
- the **userCallback** must be called to produce a **user output buffer** of size N
- the **user output buffer** of size N is transferred into the **host output buffer** of size M
- the hardware uses the **host output buffer** of size M

Because M and N are in general different, an adaptation system must be developed. Some remaining frames will have to be kept during one cycle to be used during the next cycle. The number of user callback calls in each cycle will not be always the same.

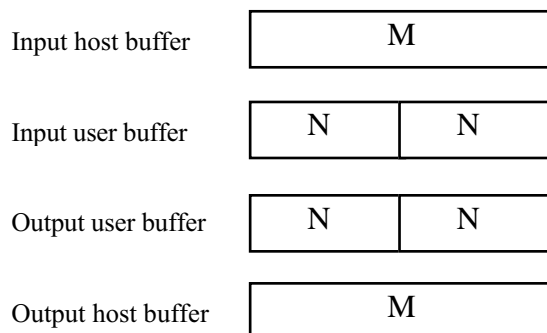
The following constraints must be respected :

- at each host callback a **full M frames** input buffer is received, this buffer must be **completely used**, a **full M frames output buffer must be produced**.
- the user callback must always be called with a **full N frames input buffer** and will produce a **N frames output buffer**.
- **latency has to be minimized** in a full-duplex case.

3 Buffer size adaptation techniques

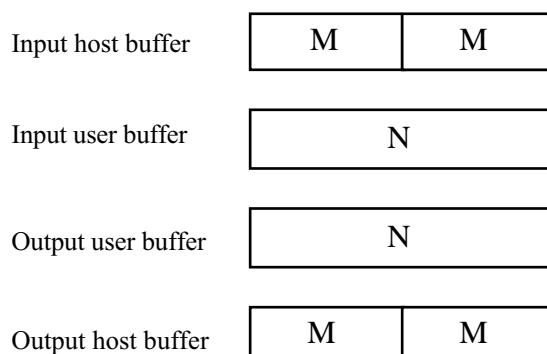
To be able to adapt the two callbacks, different cases have to be considered :

3.1 Case 1 : host buffer size multiple of user buffer size



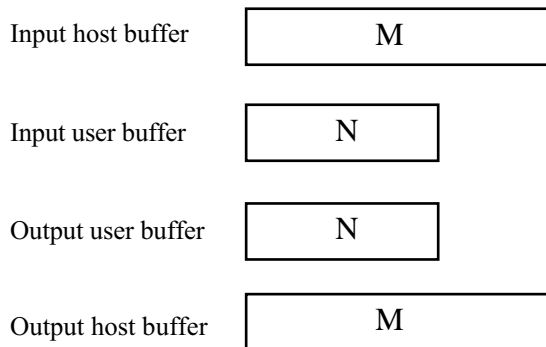
Nothing difficult here, at each host callback, an M size host buffer is received, because M is a multiple of N, M/N user callback can be called producing a complete M size host output buffer.

3.2 Case 2 : host buffer size divisor of user buffer size

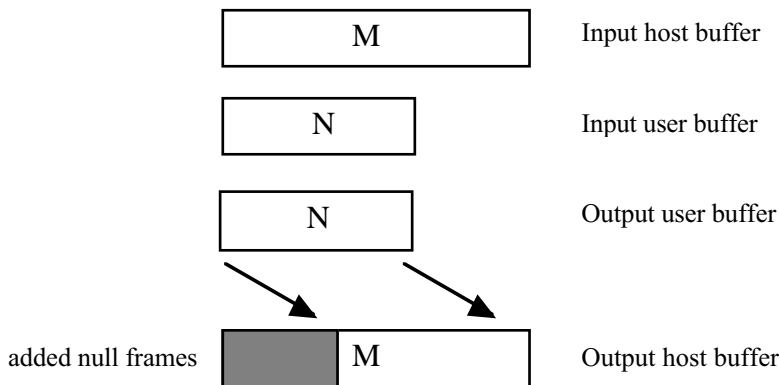


In the previous simple example where $N = 2M$, we have the following : at the first host callback, an M size host buffer is received, the user callback can not be called so the host input buffer content is kept. A null host output buffer is produced. At the next host callback, the user callback can be called, producing an M size output buffer. An host output buffer can be produced consuming $M = N/2$ frames, and so on. The output stream has been globally delayed by a complete M size buffer. The way of doing can be generalized for case where $N = n * M$.

3.3 Case 3 : host buffer size superior of user buffer size



This is a more complex case when a more sophisticated buffer size adaptation system must be found. At the first host callback, a host buffer of size M is received. Because $M > N$, the user callback can be called consuming N frames and producing an output user buffer of N frames. But because $N < M$, a complete output host buffer **can not be produced directly**. The beginning of the first produced host output number must be filled with null frames, obtaining the following scheme :



Some remaining frames have to be kept during one cycle to be used during the next cycle both into the user input buffer and the user output buffer. Four values are used to describe the state of the system :

- **Host input buffer** = the number of frames available in the host input buffer
- **User input buffer** = the number of frames possibly kept in the user input buffer between two consecutive host callbacks
- **User output buffer** = the number of frames possibly kept in the user output buffer between two consecutive host callbacks
- **Host output buffer** = the number of frames written into the host output buffer

Let take a real example with $M = 100$ and $N = 70$. To be able to complete the first host output buffer we could choose $M - N = 30$, and we can display the state of the system at the beginning and the end of each host callback :

Beginning of callback 1

Host input buffer = **100** : the host input buffer is full
 User input buffer = 0
 User output buffer = 0
 Host output buffer = 30 : added null frames for the first output buffer

After one adaptation cycle, the system state is the following :

End of callback 1

Host input buffer = 0 : all input frames have been used
 User input buffer = 30 : remaining host input frames (remainder of $100/70$)
 User output buffer = 0
 Host output buffer = $70 + 30$ added null frames = **100** ==> the host output buffer is full

Beginning of callback 2 : 100 new frames are received

Host input buffer = **100**
 User input buffer = 30 : frames kept from the previous cycle
 User output buffer = 0
 Host output buffer = 0

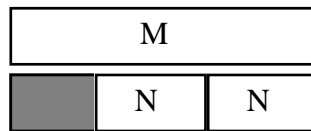
End of callback 2

Host input buffer = 0
 User input buffer = 60 : remainder of $(100 + 30)/70$
 User output buffer = **70**
 Host output buffer = 0

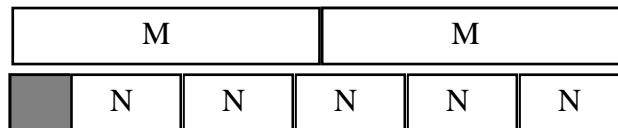
There is a problem here because a complete host output buffer can not be produced (because $70 < 100$). Thus the value has to be chosen to guarantee that a **complete host output buffer can be produced at each cycle**. We could just add **N null frames** to start the process. But adding null frames at the beginning of the first host output buffer (that is shifting the whole output stream), increases the global latency.

Thus we would like to find the **smallest number of null frames** that must be put in the first output host buffer to guarantee that the audio streaming process can be done. This value can be found by doing the following reasoning :

- **At the first host callback** : M frames are received, and M/N PortAudio callback can be called. The minimum number of needed frames to complete a full M output buffer is $M\%N$ (where % gives the remainder of M/N).



- **At the second host callback** : the total number of received frames from the beginning is $2 * M$ frames. The total number of user callback that can be called is $(2 * M)/N$. The minimum number of needed frames to complete $2 * M$ output buffers is $(2 * M)\%N$.



- **At the n host callback** : the total number of received frames from the beginning is $n * M$ frames. The total number of user callback that can be called is $(n * M)/N$. The minimum number of needed frames to complete $n * M$ output buffers is $(n * M)\%N$.

Thus the set of all remainders of $M/N, (2 * M)/N, \dots, (n * M)/N$ can be computed. The computation must be done until the **PPCM of M and N** is reached. Indeed we can see that at **PPCM(M,N)+M** we have :

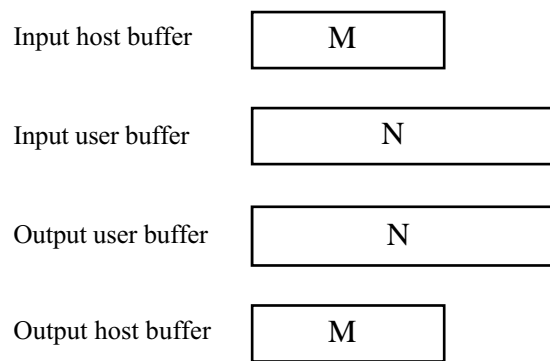
$$[(PPCM(M,N)+M)/N] = PPCM(M,N)/N + M/N = a + M/N \text{ where } a \text{ is an integer value}$$

So the remainder of $[(PPCM(M,N)+M)/N]$ is the same of the remainder of M/N , thus the set of remainder values is now complete because the same cycle of values start again. Going from $M, 2 * M, \dots$ up to the PPCM (M,N), a series of values is obtained. These values are the **minimum frame numbers** that should be added at each step to guarantee that the required number of buffers of size M can be produced.

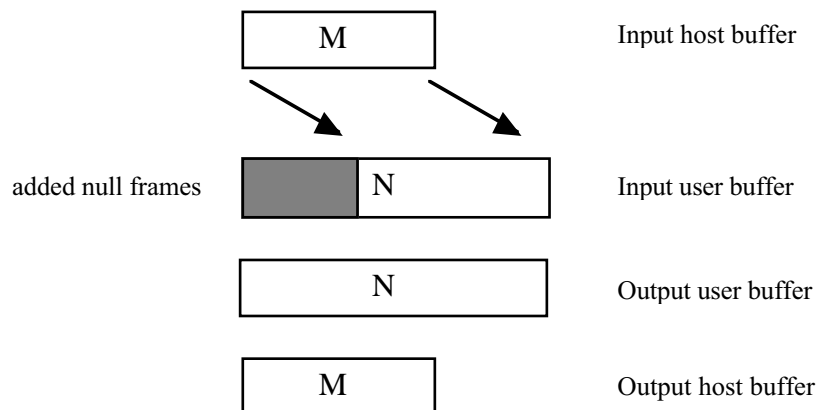
To be sure that the required number of buffers of size M can be produced at each cycle, the **maximum of this set of values** must be chosen.

- **the minimum number of frame to be added to the first host output buffer is the maximum of the serie $(n * M)\%N$ computed from M up to PPCM (M,N).**

3.4 Case 4 : host buffer size inferior of user buffer size



The same reasoning can be used in this case, and we can show that the previously explained algorithm can also be used to compute the needed value.



Here the null frames will be added to the **beginning of the first user input buffer**, so that the user callback can be called, and one host output buffer can be produced. Actually the adaptation algorithm can be used in the four different cases, where cases 1 and 2 are the more simple ones.

3.5 Performances

The used method guarantee that the latency added by buffer size adaptation is the smallest. It gives a practical way to know the right value to be used, even if the gain in latency is usually small. Actually in the case when the host buffer size M is greater to the user buffer size N , and depending of the values M and N , the gain in latency can be up to $N/2$. Indeed the buffer size adaptation algorithm gives $N-1$ in the worst case and $N/2$ in the best case.

4 Implementation

The following implementation uses a **CallbackAdaptor** data structure that contains several variables needed during each adaptation cycle.

// Utilities for buffer size alignment computation

```
int PGCD (int a, int b) {return (b == 0) ? a : PGCD (b,a%b);}
int PPCM (int a, int b) {return (a*b) / PGCD (a,b);}
```

// Takes the size of host buffer and user buffer : returns the number of frames needed for buffer alignment

```
int CalcFrameShift (int M, int N)
{
    int res = 0;
    for (int i = M; i < PPCM (M,N) ; i+=M) { res = max (res, i%N); }
    return res;
}
```

// Type for the audio callbacks

```
typedef int (audioCallback)
    (void *inputBuffer, void *outputBuffer, int framesPerBuffer, void* userData);
```

// Data structure for callback adaptation

```
typedef struct {
    float* userInputBuffer;
    float* userOutputBuffer;

    // Offset in host input buffer
    int hostInputBufferOffset;
    // Offset in host ouput buffer
    int hostOutputBufferOffset;
    // Offset in user input buffer
    int userInputBufferOffset;
    // Offset in user ouput buffer
    int userOutputBufferOffset;

    int hostBufferSize;
    int userBufferSize;
    audioCallback* userCallback;
}CallbackAdaptor;
```

// Function to be provided to handle transfer of samples between host and user buffers

```
void CopyUser2Host (float* input, float* ouput, int frames)
{
    /* To be completed */
    printf ("CopyUser2Host : %d\n",frames);
}
```

// Function to be provided to handle transfer of samples between user and host buffers

```
void CopyHost2User (float* input, float* ouput, int frames)
{
    /* To be completed */
    printf ("CopyHost2User : %d\n",frames);
}
```

// Allocates a new CallbackAdaptor :

```
CallbackAdaptor* MakeCallbackAdaptor (int M, int N, audioCallback* callback)
{
    CallbackAdaptor* adaptor = (CallbackAdaptor*)malloc (sizeof(CallbackAdaptor));

    // Allocation of user buffers
    adaptor->userInputBuffer = (float*)malloc(N*sizeof(float));
    adaptor->userOutputBuffer = (float*)malloc(N*sizeof(float));
}
```

```

    adaptor->userCallback = callback;
    adaptor->hostBufferSize = M;
    adaptor->userBufferSize = N;
    return adaptor;
}

```

// Computes the shift value used for the first output buffer : this value is used to initialize the write offset in the user input buffer or the host output buffer depending of M and N

```

void InitCallbackAdaptor (CallbackAdaptor* adaptor)
{
    int frameShift = CalcFrameShift(adaptor->hostBufferSize, adaptor->userBufferSize);

    if (adaptor->userBufferSize <= adaptor->hostBufferSize) {
        adaptor->hostInputBufferOffset = 0; // empty
        adaptor->hostOutputBufferOffset = frameShift;
        adaptor->userInputBufferOffset = 0; // empty
        adaptor->userOutputBufferOffset = adaptor->userBufferSize; // empty
    } else {
        adaptor->hostInputBufferOffset = 0; // empty
        adaptor->hostOutputBufferOffset = 0; // empty
        adaptor->userInputBufferOffset = frameShift;
        adaptor->userOutputBufferOffset = adaptor->userBufferSize; // empty
    }
}

```

// Delete a CallbackAdaptor

```

void DeleteCallbackAdaptor (CallbackAdaptor* adaptor)
{
    if (adaptor->userInputBuffer) free (adaptor->userInputBuffer);
    if (adaptor->userOutputBuffer) free (adaptor->userOutputBuffer);
    free(adaptor);
}

```

// Internal function used by HostCallback

```

void OutputUser2Host(CallbackAdaptor* adaptor, float* hostOutputBuffer, int framesNumber)
{
    if (framesNumber > 0) {
        CopyUser2Host(adaptor->userOutputBuffer, hostOutputBuffer, framesNumber);

        adaptor->hostOutputBufferOffset += framesNumber;
        adaptor->userOutputBufferOffset += framesNumber;
    }
}

```

// Callback adaptation : does the host to user buffer transfer. Call of the the user callback, user to host buffer transfer

```

void HostCallback
    (void* InputBuffer, void* OutputBuffer, int framesPerBuffer, void* userData)
{
    float* hostInputBuffer = (float*) InputBuffer;
    float* hostOutputBuffer = (float*) OutputBuffer;

    CallbackAdaptor* adaptor = (CallbackAdaptor*)userData;

    // number of frames available into the host input buffer
    long framesInputHostBuffer = framesPerBuffer;
    // number of frames needed to complete the user input buffer

```



```

long framesInputUserBuffer;
// number of frames needed to complete the host output buffer
long framesOutputHostBuffer;
// number of frames available into the user output buffer
long framesOuputUserBuffer;
long tmp;

/* Fill host output with remaining frames in user output */
framesOutputHostBuffer = framesPerBuffer - adaptor->hostOutputBufferOffset;
framesOuputUserBuffer = adaptor->userBufferSize - adaptor->userOutputBufferOffset;
tmp = min(framesOutputHostBuffer, framesOuputUserBuffer);
framesOutputHostBuffer -= tmp;
OutputUser2Host(adaptor,hostOuputBuffer, tmp);

/* Available frames in hostInputBuffer */
while (framesInputHostBuffer > 0) {

    /* Number of frames needed to complete an user input buffer */
    framesInputUserBuffer = adaptor->userBufferSize - adaptor->userInputBufferOffset;

    if (framesInputHostBuffer >= framesInputUserBuffer) {

        CopyHost2User (hostInputBuffer, adaptor->userInputBuffer, framesInputUserBuffer);

        /* Call user callback */
        adaptor->userCallback(adaptor->userInputBuffer, adaptor->userOutputBuffer,
                               adaptor->userBufferSize, NULL);

        /* Full user ouput buffer : write offset */
        adaptor->userOutputBufferOffset = 0;

        /* Empty user input buffer : read offset */
        adaptor->userInputBufferOffset = 0;

        tmp = min (adaptor->userBufferSize,framesOutputHostBuffer);
        OutputUser2Host(adaptor, hostOuputBuffer, tmp);

        framesOutputHostBuffer -= tmp;
        framesInputHostBuffer -= framesInputUserBuffer;
        adaptor->hostInputBufferOffset += framesInputUserBuffer;

    }else {

        CopyHost2User (hostInputBuffer, adaptor->userInputBuffer,framesInputHostBuffer);

        adaptor->userInputBufferOffset += framesInputHostBuffer;
        adaptor->hostInputBufferOffset += framesInputHostBuffer;

        framesInputHostBuffer = 0;
    }
}
}

// The following functions are used to simulate a real system :

// To “simulate” the hardware interrupt : the host input buffer has been filled, it is now full
void ReadInputHostBuffer(CallbackAdaptor* adaptor)
{
    adaptor->hostInputBufferOffset = 0; // full host input buffer;
}

```

// To “simulate” the hardware interrupt : the host output buffer has been used, it is now empty

```
void WriteOutputHostBuffer(CallbackAdaptor* adaptor)
{
    adaptor->hostOutputBufferOffset = 0; // empty host output buffer;
}
```

// Prints the state of the CallbackAdaptor structure

```
void PrintCallbackAdaptorState(CallbackAdaptor* adaptor)
{
    printf ("Adaptor : hostInputBufferOffset %ld \n", adaptor->hostInputBufferOffset);
    printf ("Adaptor : hostOutputBufferOffset %ld \n", adaptor->hostOutputBufferOffset);
    printf ("Adaptor : userInputBufferOffset %ld \n", adaptor->userInputBufferOffset);
    printf ("Adaptor : userOutputBufferOffset %ld \n", adaptor->userOutputBufferOffset);
}
```

// The user callback

```
int UserCallback
    (void *inputBuffer, void *outputBuffer,int framesPerBuffer, void* userData)
{
    printf ("UserCallback : framesPerBuffer %d \n",framesPerBuffer);
}
```

// General simulation function

```
void Simulate (int M, int N)
{
    int i,n = PPCM(M,N)/M+1;
    CallbackAdaptor* adaptor = MakeCallbackAdaptor(M, N, UserCallback);
    InitCallbackAdaptor(adaptor);

    // Allocation of host buffers
    float* hostInputBuffer = (float* )malloc(M*sizeof(float));
    float* hostOutputBuffer = (float* )malloc(M*sizeof(float));

    printf ("\n----- \n");
    printf ("Simulation of Callback adaptor [%d,%d]\n", M, N);

    for (i = 0; i< n ; i++) {
        printf ("----- \n");
        printf ("Callback %d \n",i);
        printf ("----- \n");
        ReadInputHostBuffer(adaptor);
        PrintCallbackAdaptorState(adaptor);
        HostCallback(hostInputBuffer,hostOutputBuffer,M,adaptor);
        PrintCallbackAdaptorState(adaptor);
        WriteOutputHostBuffer(adaptor);
    }

    DeleteCallbackAdaptor(adaptor);

    free(hostInputBuffer);
    free(hostOutputBuffer);
}

int main ()
{
    printf ("\n-----\n");
    printf ("Simulation of Callback adaptor\n");
    Simulate(2016,512);
    Simulate(128,250);
    printf ("End of simulation \n");
}
```

5 Conclusion

A callback adaptation technique has been presented. It allows to use two callbacks that use buffers of different sizes, and guarantee the lowest latency added by buffer size adaptation. This technique has been successfully used for the port of the PortAudio API on ASIO described in [Letz 2001].

References

[ASIO 97-99] Asio 2.0 Audio Streaming Input Ouput Developement kit. Steinberg (c) 1997-1999 Steinberg Soft - und harware GmbH.

[Bencina, Burk 2001] Ross bencina, Phil Burk “PortAudio : an Open Source Cross Platform Audio API” Proceedings of the International Computer Music Conference 2001, International Computer Music Association, San Francisco.

[Letz 2001] Stephane Letz “Porting PortAudio API on ASIO” GRAME - Computer Music Research Lab. Technical Note - 01-11-06