Introduction to using the
Csound Host API

Rory Walsh
Dundalk Institute of Technology,
Ireland.

1. Introduction

An API (application programming interface) is an interface provided by a computer system, library or application, which provides users with a way of accessing functions and routines particular to the control program. Essentially APIs provide developers with a means of harnessing an existing applications functionality within a host application. A typical example of an API is the Microsoft Win32 API which provides developers with a way to carry out common Windows tasks from within their own software. For example when Windows starts up it plays a piece of audio. In order to do this Windows developers created a function that plays audio. Developers using the Windows API may also call this function from their software to play a piece of audio. DirectX is another example, it's a collection of APIs for easily handling tasks related to programming computer games in Windows.

The Csound API can be used to control an instance of Csound through a series of different calling functions. In short, the Csound API makes it possible to harness all the power of Csound in your own applications.

In order to use the API you will need to download the Csound source code which can be found here: http://csound.sourceforge.net/. To build the examples below with MinGW you will need to first create an import library by running the following instructions at command prompt[1].

```
pexports -o csound32.dll.5.1 > csound32.def
```

Before proceeding to the next step you may need to edit the newly created csound32.def and remove the first line which seems to cause an error. Then created the new import library using the following command:

```
dlltool --dllname=csound32.dll.5.1 --input-def=csound32.def -output-
lib=libcsound32.a
```

Now you are ready to start building Csound API applications.

1. *The Csound dll name will differ with each release and you have the choice of two different floating point precisions, 32bit, or 64bit. If you don't have the pexports tool it can be found here*
http://www.emmestech.com/software/cygwin/pexports-0.43/download_pexports.html
I recommend putting the pexports executable into your MinGW bin folder so that you don't have to add any new directories to the system path.

2. Building a basic Csound application

Building a simple Csound API application is relatively straightforward. In fact the classic Csound command line program itself is built using the Csound API. Below is the code needed in full to build a typical command line version of Csound.

```
#include <stdio.h>
#include "csound.hpp"

int main(int argc, char* argv[])
{
/*Create an instance of Csound*/
CSOUND*csound=csoundCreate(0);

/*Initialise the library*/
csoundInitialize(&argc, &argv, 0);

/*Compile Csound, i.e., run Csound with
the parameters passed via the command line.
Return false is successful*/
int result=csoundCompile(csound,argc,argv);

/*check to see that Csound compiled Ok, then
start the performance loop. csoundPerformKsmps()
returns false if successful*/
if(!result)
{
while(csoundPerformKsmps(csound)==0);
}

/*Finally destroy Csound*/
csoundDestroy(csound);

return result;
}
```

To build the above example with MinGW you must link to the Csound import library and add an include path to csound.h like this:

```
g++ main.cpp -o csoundAPItest.exe -I"D:/MyDocuments/SourceCode/Csound/src/H"
-L"D:/MyDocuments/SourceCode/Csound/bin" -lcsound32
```

It's also quite easy to build the above application using Dev-C++, providing you follow the two simple steps below.

1) In ***Project Options*** go to ***Parameters,*** then hit the ***Add Object or Library*** button. A browser window will appear. Select the Csound library, in this case *"libcsound32.a*"
2) In ***Project Options*** go to ***Directories***. Click the ***Include Directories*** tab and pass the directory that contains the file *'csound.hpp'* or *'csound.h'* if you using a C compiler.

Once you have built the application you will be able to use it just like the standard command line Csound, remember that this is a command line program so you will need to open the command prompt and navigate to the directory that the application resides in in order to run it. Don't forget to pass the normal Csound flags when testing your new application!

3. Communicating with Csound

Typical use of Csound allows users to interact via midi or OSC messages. In order to get the most out of the Csound API it's vital that we are able to communicate and interact with the instance of Csound that is running. Thankfully there are several mechanisms provided by the API to allow us to do this.

Using one of the *'software bus'* opcodes in Csound we can provide an interface for communication with a host application. For example the chnget opcode:

```
kval chnget "channel_name"
```

The `chnget` opcode reads data that is being sent from a host Csound API application on a particular named channel, and assigns it to the output variable `'kval'` (of course you can name the output variable anything you like). In order to send the data on the named channel from our application we can use the `csoundGetChannelPtr()` API function which is declared in "csound.h" as:

```
PUBLIC int csoundGetChannelPtr(CSOUND *, MYFLT **p, const char
*name, int type);
```

`CsoundGetChannelPtr()` stores a pointer to the specified channel of the bus in `p`. The channel pointer `p` is of type `MYFLT` which is a `float` for 32bit floating point samples or a `double` for 64bit floating point samples (always use MYFLT instead of float or double). The arguments `name` and `type` are, respectively, the name of the channel and a bitwise OR of exactly one of the following values,

`CSOUND_CONTROL_CHANNEL`
- control data (one MYFLT value)

`CSOUND_AUDIO_CHANNEL`
- audio data (ksmps MYFLT values)

`CSOUND_STRING_CHANNEL`
- string data (MYFLT values with enough space to store csoundGetStrVarMaxLen(csound) characters, including the NULL character at the end of the string)

and at least one of these:

`CSOUND_INPUT_CHANNEL`
- when you need Csound to accept incoming values

`CSOUND_OUTPUT_CHANNEL`
- when you need Csound to send outgoing values

If the call to `csoundGetChannelPtr()` is successful the function will return zero. If not it will return a negative error code, `CSOUND_MEMORY` if there is not enough memory for allocating the channel and `CSOUND_ERROR` if the specified name or type is invalid. If a channel with the same name but incompatible type already exists `csoundGetChannelPtr()` will return the type of the existing channel. In the case of any non-zero return value, `p` is set to NULL. We can now modify our original source code in order to send data from our application on a named software bus to an instance of Csound using `csoundGetChannelPtr()`.

```
#include <stdio.h>
```

```
#include "csound.hpp"

int main(int argc, char *argv[])
{

CSOUND*csound=csoundCreate(0);
csoundInitialize(&argc, &argv, 0);
int result=csoundCompile(csound,argc,argv);

/* declare a pointer to a variable of type pvalue */
MYFLT *pvalue;

if(!result)
{
while(csoundPerformKsmps(csound)==0){
/*call csoundGetChannelPtr to send the value of
'pvalue' to 'csound' on a channel named 'pitch' */
if(csoundGetChannelPtr(csound, &pvalue, "pitch",
                    CSOUND_INPUT_CHANNEL | CSOUND_CONTROL_CHANNEL) == 0)
   *pvalue = 200;
}

}
csoundDestroy(csound);
return result;
}
```

You can test the the above program by running it with the following csound instrument:

```
instr 1
kval chnget "pitch"
a1 oscil 10000, kval, 1
out a1
endin
```

In order to change the pitch being played back by Csound you only need to change the value of `pvalue`. Although trivial, the above code serve as a good example of how to communicate with an instance of csound through calling the `csoundGetChannelPtr()` API function.

4. Threads and realtime interaction with Csound

In the previous section we saw how it's possible to communicate with an instance of Csound using a special 'software bus' API function. The one major drawback in the previous example is that we have no realtime control over the data being sent to Csound, something that is very important in order to harness the power of Csound. In order to do this we will need to know how to set up a so-called “performance thread”.

Threads are used so that a program can split itself into two or more simultaneously running tasks. Multiple threads can be executed in parallel on many computer systems. When multithreading you can have many different threads running simultaneously (in actual fact the threads do not run simultaneously, instead the processor will switch rapidily between the threads giving the impression

that there are being run simultaneously). On newer multiprocessor systems '*true*' multithreading is possible whereby each thread may be run on different processors.

When implementing threads using the Csound API one must define a special performance routine. This performance-thread routine is then passed as a parameter to the csoundCreateThread() API function which sets up a new thread, which in turn calls our performance routine.

When defining a Csound performance-thread routine you must define it as a type `uintptr_t`, hence it will need to return a value once it's called. The function will take only one parameter, a pointer to void, which is a unique pointer that can be assigned the value of any other pointer type.

Below is one of the most basic performance-thread routines one can implement, note that this code needs the variables `result` and `csound` to have been declared globally, which isn't always a great idea (see below).

```
uintptr_t csThread(void *clientData)
{
      //check that csoundCompile was successful
      if(!result)
        {
             while(csoundPerformKsmps(csound) == 0);
             csoundDestroy(csound);
        }
     return 1;
}
```

In order to start this thread we must call the csoundCreateThread() API function which is declared in csound.h as:

```
PUBLIC  void  *csoundCreateThread(uintptr_t  (*threadRoutine)(void
*),void *userdata);
```

If you are building a command line program you will need to use some kind of mechanism to prevent `int main()` from returning until after the performance has taken place. A simple `scanf()` will suffice, or if you prefer, you can use one of the C++ iostream library functions.

The basic performance thread definition above will only work when the variables `csound` and `result` are declared globally. Generally it is not a good idea to use global variables with threads as it can lead to unexpected results, the most obvious being where two seperate threads try to modify one global variable. Instead, a better approach is to pass all the data needed by the performance thread function through the `userData` parameter of `csoundCreateThread()`. One of the best ways to do this is to declare a structure that contains all the data needed by our performance thread function and then pass this strcuture to `csoundCreateThread()`.

```
struct userData{
/*result of csoundCompile()*/
int result;
```

```
/*instance of csound*/
CSOUND* csound;
/*performance status*/
bool PERF_STATUS;
};
```

In the above structure we declare a structure of type userData. In the structure we define our member variables: `result`, `csound` and `PERF_STATUS`. `PERF_STATUS` can be used in the performance loop to stop Csound. It can also be used to prevent Csound from starting.

Now that it's clear how to set up a performance thread we can start interacting with our instance of Csound during a performance. For example:

```
#include <stdio.h>
#include "csound.hpp"

//performance thread function prototype
uintptr_t csThread(void *clientData);

//userData structure declaration
struct userData {
int result;
CSOUND* csound;
bool PERF_STATUS;
};

//------------------------------------------------------------
int main(int argc, char *argv[])
{
int userInput=200;
void* ThreadID;
userData* ud;
ud = (userData *)malloc(sizeof(userData));

MYFLT* pvalue;
ud->csound=csoundCreate(NULL);
csoundInitialize(&argc, &argv, 0);
ud->result=csoundCompile(ud->csound,argc,argv);
if(!ud->result)
  {
  ud->PERF_STATUS=1;
  ThreadID = csoundCreateThread(csThread, (void*)ud);
  }
else{
printf("csoundCompiled returned an error")
return 0;
}

printf("\nEnter a pitch in Hz(0 to Exit) and type return\n");

while(userInput!=0)
     {
     if(csoundGetChannelPtr(ud->csound, &pvalue, "pitch",
                    CSOUND_INPUT_CHANNEL | CSOUND_CONTROL_CHANNEL)==0);
     *pvalue = (MYFLT)userInput;

     scanf("%d", &userInput);
```

```
     }
ud->PERF_STATUS=0;
return ud->result;
}

//-------------------------------------------------------------
//definition of our performance thread function
uintptr_t csThread(void *data)
{
      userData* udata = (userData*)data;
      if(!udata->result)
        {
             while((csoundPerformKsmps(udata->csound) == 0)
                  &&(udata->PERF_STATUS==1));
            csoundDestroy(udata->csound);
        }
      udata->PERF_STATUS = 0;
      return 1;
}
//-----------------------------------------------------------------------------------------
```

Apart from the using the software bus there are also others ways of iteracting with Csound. One such way is through `csoundScoreEvent()` funtion defined in “csound.h” as:

```
PUBLIC int csoundScoreEvent(CSOUND *, char type, const MYFLT
*pFields, long numFields);
```

`type` is a score event type, either a, i, q, f, or e. `pfields` is an array of floats with all the p-fields for the particular event, starting with the p1 value specified in `pFields[0]`. `numFields` is the total number of p-fields contained in the score event.

In order to illustrate the use of this function we can modify our previous code by replacing the call to csoundGetChannelPtr() with a call to csoundScoreEvent():

```
printf("\n Usage: p1 p2 p3 p4 p5\nEnter 0 for p1 to quit\n");
while(1)
     {
     for(int i=0;i<5;i++) scanf("%f", &userInput[i]);
     if(userInput[0]<1)
       {
       ud->PERF_STATUS=0;
       break;
       }
     csoundScoreEvent(ud->csound, 'i', userInput, 5);
     }
```

To run this you should have a Csound instrument that accepts has 5 p-fields or else you will get some Csound warnings.

5. More information

Apart from the functions seen above the Csound API also contains a myriad of other useful functions to aid you in the development of new software. If you need any more information on the functions available the best place to start is with the file “csound.h” file that comes with the Csound source code. If you have any further questions you can join the Csound developers list where there is always some on hand to answer even the most basic of questions. If you are looking for more general Csound information try visiting www.csounds.com which features lots of useful links and resources.