The Design and Use of *Minimal7*: Creating Subsets of Csound for Embedded Applications

John ffitch[1] and Richard Boulanger[2]

[1] Alta Sounds
[2] Berklee College of Music
`jpff@codemist.co.uk` `rboulanger@berklee.edu`

Abstract. There have been complaints of "opcode bloat" with Csound, especially when embedding an audio application into small devices. The *Minimal7* system offers a solution by automating the process of only including the opcodes and fgens that are actually used in a customised Csound system, not "all" of Csound. The following paper explains the mechanism for this process, and a simple example of the workflow is presented. This is followed by a description of the limitations of the current version and suggestions of what could be done to generate even smaller customised versions of Csound.

Keywords: Csound, Embedding, Customised

1 The Problem and Initial Description

Although Csound is quite old, it is still under development, and over the years, each new release included new opcodes (now in excess of 1100 and up to 1800 if polymorphic opcodes are fully counted). And it has been some years now since the first complaints of "opcode bloat" appeared, essentially because faster computers, with more memory, have served to disguise the issue for general users. However, for the growing number of specialist suppliers of audio products that are delivered on small devices, products that use Csound as an audio engine, the memory requirements could be critical and "opcode bloat" could be a real show stopper.

Based on Csound7, *Minimal7* is a system for customising a Csound build to a particular csd or orc file, and as such, goes a long way toward delivering a minimal package. The basic change is that the code for an opcode is wrapped in a `#ifdef/#endif`, and the `OENTRY` table is similarly protected. Then a new header file is added to the base system, and `opcodes.h` uses `#define` to select the opcodes that are needed. Currently, tools have been created to assist in the management of this file, but in a later section of this paper, other selection mechanisms are suggested and other changes proposed.

2 Details and Tools for Opcodes

The problem was to invent a C macro naming system for each opcode. Name clashes were avoided by the addition of new words, all starting with `INC_`. The rest of the name was based on the opcode's name itself. A utility was created to write a dictionary, essentially (macro name, opcode name). These pairs are read from the C sources. This is fairly simple and straightforward, as long as there are some conventions as to code layout, with are nearly always followed in the main system. This utility is called `mkdict`.

To remove the error-prone work of creating a different `opcodes.h` file, a new command-line option (`--minilist=filename`) has been added to the argument parser, whereby the compiled tree for each instrument is scanned to see what opcodes are used in the input csd/orc file, and then, to use that subset to write the opcodes list file using the dictionary created by the mkdict utility. The mkdict utility also writes a file `opcodes_all.h` of `#define` statements for all opcodes that are optional.

This new file, installed as `opcodes.h`, means that it builds the same Csound as the mainstream, but when replaced by the file created by the minilist option it builds a customised Csound with only necessary opcodes.

The work flow is thus:

- Copy the file `H/opcodes_all.h` to `H/opcodes.h` and make Csound.
- Develop the application as a csd file.
- Run the completed application with the `--minilist=myopcodes.h` option.
- Copy the file `myopcodes.h` to `H/opcodes.h`
- Make the application.

3 A Complete Example

In this section we follow the creation of an application `example.csd` based on the first movement of *Drums and Different Canons #1*[3]

```
sr          =          44100
kr          =          4410
ksmps       =          10
nchnls      =          2

            instr 6
k1          linseg      0,p3/2,p5,p3/2,0        ; ENV
k2          randi       2,25
k3          oscil      .03,4+k2,1               ; RAND TREM
a1          gbuzz      .33+k3,i1,p6,1,50,4
k4          randi       1.2,15
k5          oscil      .03,4+k2,1               ; RAND TREM
a2          gbuzz      .33+k5,i1+(i1*.02),p6,1,50,4
k6          randi       1.5,20
k7          oscil      .03,4+k2,1               ; RAND TREM
a3          gbuzz      .33+k7,i1-(i1*.021),p6,1,50,4
```

[3] John ffitch 1996

```
a4        =         (a1+a2+a3)*k1
ga1       =         a4*p7
ga2       =         a4*(1-p7)
          outs      ga1,ga2
          endin

          instr 7
i1        =         cpsoct(p4)
k1        expseg    .0001,.05,p5,p3-.05,.0001 ; ENV
k2        linseg    1.69,.1,1.75,p3-.1,1.69   ; POWER TO PARTIALS
a1        foscil    k1,i1,1,2.01,k2,1
ga1       =         a1*p6
ga2       =         a1*(1-p6)
          outs      ga1,ga2
          endin

          instr 10
i1        =         cpsoct(p4)
i2        =         log(i1)/10.0 - p6
k1        expseg    .0001,.03,p5,p3-.03,.001  ; ENV
k25       linseg    1,.03,1,p3-.03,3
k1        =         k25*k1
k10       linseg    2.25,.03,3,p3-.03,2       ; POWER TO PARTIALS
a1        gbuzz     k1,i1,k10,0,35,4
a2        reson     a1,500,50,1     ;filt
a3        reson     a2,1500,100,1   ;filt
a4        reson     a3,2500,150,1   ;filt
a5        reson     a4,3500,150,1   ;filt
a6        balance   a5,a1
i6        =         p6
ga7       =         a6*i2
ga8       =         a6*(1-i2)
          outs      ga7,ga8
          endin
```

First, we create the Csound project as a working csd file using any of the system's opcodes as the application requires. For the example above the final check ends thus:

```
Score finished in csoundPerform()
inactive allocs returned to freespace
end of score.              overall amps:  11371.5  18242.4
           overall samples out of range:        0        0
0 errors in performance
Elapsed time at end of performance: real: 0.679s, CPU: 0.678s
512 1024 sample blks of shorts written to test.wav (WAV)
```

And so, we have a working csd file for the project. At the time of this run, the library libcsound64.so.7.0 used by the standard Csound7 development had a size of 4228583 bytes on a 64bit Linux system. Csound is actually bigger than this when taking into account the base code and plugin modules, but this does give an indication of the size we wish to reduce.

Now we need the list of opcodes that are actually used. This could be done by hand, but it is a tedious and error-prone process. To avoid the labour, we obtain this list from Csound itself with the new command-line option:

```
csound example --minilist=example_opcodes.h
```

It is worth noting that obtaining this new list of used opcodes does take slightly longer than the standard version used above. This additional time is due to the dictionary lookups in the process, which are done with a simple linear search. Still, this is a once-off cost, but if it is a real concern, one could use alternative methods for searching. The file generated is quite short as not many opcodes are used:

```
#define MINITITLE "example.csd"
#define INC_CPSOCT
#define INC_LINSEG
#define INC_RANDI_K
#define INC_OSCILkkk
#define INC_GBUZZ
#define INC_AASSIGN
#define INC_OUT
#define INC_EXPSEG
#define INC_FOSCIL
#define INC_LOG
#define INC_MINIT
#define INC_RESON
#define INC_BALANCE
```

In general one does not need to look at this file. The define of the macro MINITITLE is an aid to understanding which csd file was used to generate the file, and this title is also printed in the modified Csound program.

The penultimate step is to install the opcode list file in `H/opcodes.h` and make the customised Csound. This creates a library of size 3011649 bytes, about 71% of the initial version. Note that at this time, the transformation of all of the opcodes in Csound7 has not yet been completed, and there are more savings to be unearthed, but it takes time to annotate the sources, and there are some difficult cases, considered in section 5.

The final step is to create the reduced application which when run produces the same output as the complete Csound, as expected.

4 Ftable Generators

Another area of code that can be treated in a similar way to the opcodes are the f-table generators. All standard generators are define in one source file. Wrapping each generator in a `#ifdef...#endif` is straightforward. Given that the access to the generators is via an internal array of functions, it is easier not to delete the unused operations but to replace them with an empty placeholder, as shown in the following fragment.

```
static int gen13(FGDATA *ff, FUNC *ftp)
#ifdef INC_GEN13
{
    return gn1314(ff, ftp, FL(2.0), FL(0.5));
}
#else
{   return NOTOK; }
#endif
```

Determining which gens are used cannot be done in the parser, either the score or the orchestra, because it has to be done at run time to allow for tables being created in the fgen section of the orchestra. This is a similar mechanism to the opcodes collecting. Plugin gens need to be treated similarly, but there are some difficulties.

5 Constraints and Possible Improvements

For selective inclusion, some families of opcodes are difficult to separate into code. For example, the Perry Cook physical models use a library of filter classes with cross use. This is also found in ATS family, and to a lesser extent in OSC.

There are a large number of small opcodes for arithmetic which could have this treatment. Close reading of the example in Section 3 shows the use of a variety of arithmetic operations, but this is not even a third of the arithmetic operations available in Csound.

As arrays are becoming more common in instrument design, arithmetic on arrays is another untouched area offering potential gains, but few will use the whole set of 117 opcodes for array arithmetic.

Another possible area of gain that could be separated is all the MIDI operations, although this seems a more difficult task and might require significant restructuring of the code.

Another major step towards a truly minimal system would be to divide the Csound program into a lexer-parser-tree creator, and a performance controller that reads in the output of the compiler. Then, the application could be delivered without the parser and two lexers with associated language processing. There would need to be considerable design effort to get this right, to exclude as much code as possible from the back end.[4]

[4] It could be noted that Extended Csound had such a system, but the parser was simpler and there was no plugin or dynamic change scheme.

6 More Examples

The system has been used on *Trapped in Convert* and *Xanadu* with similar affect. The size of the Csound library was 68% and 67% respectively with a very small, probably not significant, time reduction.

7 Conclusion

The current state is a proof of concept. However, as it makes wide-spread changes to the sources, it is hard to merge into, or to separate from, the developer branch and is, in effect, seeking replication of changes. And therefore, the future of this project is currently unknown. Whether it survives depends on the response of the developer and embedded communities.

A major disadvantage is that the annotations may obscure the readability of the code. This has not been a problem for the author but we recognise the potential for confusions. A possible alternative is to move more parts of the opcodes into loadable dynamic libraries, which raises the question of the granularity of the code, either having very many small libraries or larger collections which would increase the size to include unused opcodes. There would also be a need for a tool to identify which libraries were needed and arrange to load them.

We seek a discussion within the whole Csound community about the issues raised in this paper, and we would like to thank the anonymous referee for their insights and comments.