# A Lightweight C++ Framework for Real Time Active Control

**Robert Piéchaud**
IRCAM-CNRS-UPMC, UMR STMS 9912
Instrumental Acoustics Lab
4, place Igor Stravinsky 75004 Paris, France.
robert.piechaud@ircam.fr

Real Time Linux Workshop, October 2014

**Abstract**

Active control is a common application of hard real-time. In particular, active control of musical instruments pushes the limits of determinism and low-latency very far, as it must deal with relatively high frequencies in order to be efficient. With modal active control of vibration as a starting point, a more generic, abstract, yet lightweight object-oriented active control framework has been developed for covering potential situations where real-time environment, hardware, and control model may vary a great deal, but where functional expectations remain rather constant. Practical results for one implementation of this framework on a low-cost platform will be exposed, namely on a Beaglebone Black running Xenomai and equipped with a custom DAQ cape in a SISO context, also taking advantage of the PRUSS (Programmable Real Time Unit) as a powerful complement to Xenomai for achieving an even better determinism.

## 1 Introduction

In the world of active control of vibration, *smart* musical instruments - acoustic instruments equipped with sensors and actuators plugged into a real time system - represent some relatively exotic situations.

While *controlling the vibration* generally means canceling them as much as possible by imposing an opposite force - think of an industrial system where a structure's mechanical steadiness is critical, the goal is quite different when dealing with musical instruments.

In fact, rather than canceling the vibration, "musical" acousticians are obviously interested in changing the properties of a vibrating structure (for instance a guitar soundboard, a clarinet resonator), by dynamically modifying the transfer function, by means of actuators, sensors, and a high-speed real time control system.

From this technically challenging context and various recent experiments (always using a real time Linux setup) has emerged a more generic C++ code design, yet simple and lightweight, hopefully suitable for other types of active control methods.

## 2 The Initial Setup

This initial setup for actively controlled musical instruments using a real time Linux environment, as presented in Lugano at RTLW 2013 [1], is being reminded in this section.

### 2.1 The Context

Smart musical instruments allow to study, and potentially address, certain properties or issues found in many "natural" instruments. More generally the fundamental question of the musical acoustics - "what is the relationship between the musicality of an instrument and its physical parameters?" - can possibly be answered in a more efficient and precise way by altering the physical parameters through an active control, without having to "carve the wood".

**FIGURE 1:** *A "smart" acoustic guitar.*

Also, from a purely musical point of view, smart instruments pave new exciting ways in today's instrument making, while creating new paradigms for musicians at the same time.

## 2.2 Real Time Requirements

Such an active control system appears to be very challenging as we are dealing with subtle components of the sound vibration. In our case, a *modal* control model is used to target certain vibration modes of an instrument's part (a soundboard), at relatively high frequencies, with the aim of modifying their amplitude, their decay, and even their frequency. In term of active control, this means speed - high sample rate, and time determinism - chronological precision and low latency.

Moreover, we have noticed that, if $f$ is the sample frequency, modes with frequencies $\leq f/10$ could be comfortably controlled.

## 2.3 An "Nearly-Hard" Real Time System

As meeting the deadline of time samples is obviously important for such a system to work smoothly, the choice of a real time Linux development environment was natural. And, even if some sporadic deadline overrun can never be as serious in this context as in automotive or aerospace, we wanted an as hard as possible real time, a *nearly hard* real time. And in our case, the hardware and the software form a Single Input-Single Output (SISO) control system, thus with just one captor and one actuator.

### 2.3.1 Hardware

The hardware is a Pentium IV desktop computer (2 virtual cores at 2.4 GHz) equipped with a National Instrument PCI-6281 acquisition board. The acquisition board is connected to the sensor / actuator pair via a charge- (sensor) and tension- (actuator) amplifier. The choice of the respective locations for the actuator and the captor can be tricky, and is not only guided by the theory, but also through trials and errors, depending on the type of body aimed to be controlled.

### 2.3.2 Model

The SISO control loop's model is ruled by a modal approach [2] where the state space variables' evolution is expressed through first order equations, including a Luenberger observer [3].

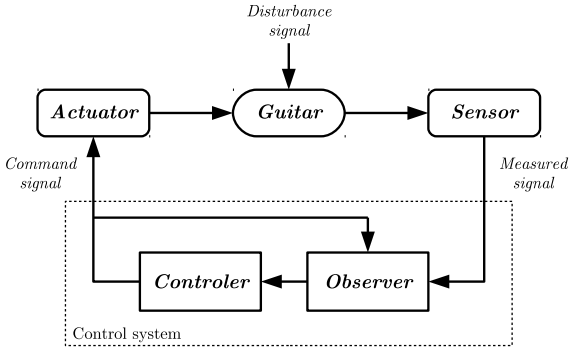


**FIGURE 2:** *The active control model of a smart guitar. The disturbance signal generally represents the musician's interaction with the instrument.*

### 2.3.3 Software

As a first step, the Linux kernel (2.6.32) was patched with Xenomai 2.5.6. On the code side, we have followed this sequence:

- Matlab: identification of the vibrating structure's parameters (modes of vibration);
- Matlab: update of the model's matrices with identification data at a fixed sample period;
- Simulink: automatic generation of C code from the model's block diagram and matrices, using zero- or first-order discretization;
- Analogy Proxy/Moulinette: patch of Simulink C code wrapping it into a real time task under

Xenomai, communicating with the acquisition board through the *analogy* driver layer built in Xenomai.

- Compilation using gcc.

#### 2.3.4 Drawbacks

As it is strongly tied to Matlab/Simulink, this setup nicely fits with the experimental stage ; it has allowed us to validate the general approach and models, but this strong link also means some inconvenience:

- The whole process can be cumbersome: the project must be generated, patched, and re-compiled whenever a parameter changes;
- The sample period must be chosen *before* the code generation;
- Matlab's or Simulink's discretization is a black box;
- Simulink's generated code is not "suitable for human beings", thus very hard to optimize;
- The project code is somehow confused, mixing model, control loop, and acquisition altogether;
- The hardware is not easily transportable (literally), while musical instruments (guitar, cello, etc.) generally are.

#### 2.3.5 Results

In a model with 20 modes, the average "roundtrip" time - one sample to the next, including mesure, model computation, and command - was about 25 $\mu$s - decent. And it turned out quickly that most of the time was taken by the model computation, i.e. the simulink generated code. And, while the loop is running in primary mode with the highest priority, with a rather satisfactory sampling frequency (approaching the standard 44.1 KHz), the deviation from the average value cannot be underestimated. One possible reason for this deviation is the lack of full timing control over the Matlab/Simulink black box.

In one word, the results for the initial setup were encouraging as far as the control itself, but there seemed to be room for improvement in the software architecture, and more control over the details was needed.

# 3 Moving to the BeagleBone Black

From the first experimental stage that helped to validate the model and the scientific approach, it was then decided to move to a new environment, and to completely review the design. Several (and ideally all) of the above-mentioned drawbacks had to be addressed. Also, keeping in mind that smart instruments should be played on stage at some point, we needed to give embedded platforms a try.

The BeagleBone Black [4] platform was chosen for its credit-card size, the promising presence of the 2 PRUSS (Programmable Realtime Unit Sub Systems) within the ARM-based architecture, and also for its notably active open hardware community.

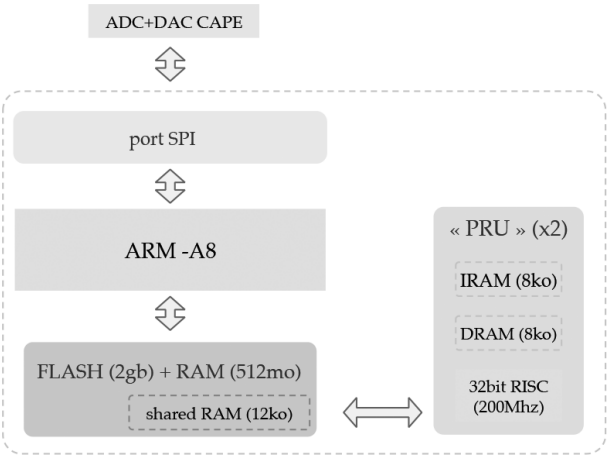


**FIGURE 3:** *The BeagleBone Black's simplified architecture.*

## 3.1 Xenomai on the BeagleBone Black

Xenomai was installed on the BeagleBone Black quite easily [6]. But the *analogy* driver layer, used in the initial setup for real time communication with the acquisition board, cannot be used on this platform (at least with Xenomai 2.x).

## 3.2 Breaking Free from Matlab

Of course, at this point, the dependency to Matlab/Simulink was questioned in the first place, and not only because of the limited horse power of the new platform; we quickly concluded that rewriting the maths in C++ was a natural thing to do. For this purpose, several math library were considered, including *armadillo* [5].

## 3.3 Heap Vs. Stack

armadillo was eventually not chosen, for several reasons: the dependency to Lapack or BLAS weighing down the code, the surprisingly poor performance on ARM, and the heap-based new/malloc and delete/free operations constantly happening behind the hood. The latter reason was really *the* deal breaker as, in Xenomai for instance, the fact that the low level memory routine *brk()* is called in those cases, inevitably throws the task out from primary to secondary mode, thus ruining the real time.

As far as we could see from the various linear algebra libraries we tried, this aspect - the dynamic memory allocation breaking the real time - is generally not taken into account.

So we coded tailor-made linear algebra classes (vector, block-diagonal or sparse matrices) making sure that key operations were now happening on the stack, which supposes to use fixed-size float buffers. The c8 and r8 [7] libraries were used as a underlying maths layer.

### 3.3.1 A Curious Phenomenon with Neon

By the way, it is perhaps worth reporting some paradoxal situation with Neon optimization. At first, the vector and matrix code was Neon-optimized. Testing in non-real time context showed a 4x speed boost when allocating on the heap (using new/delete). But when dealing with the stack (through fixed-size buffers), the same operations performed rather badly. Inversely, *deactivating* Neon (with operations still on the stack) gave about the same performance as with Neon on the heap...

## 3.4 "What time is it?" - The Magic of the PRUSS

The answer to this apparently mundane question is not that easy, even in "real-time" context. *clock_gettime()* is generally used to get a supposedly reliable time base. But results might differ from a hardware to the other. As an example, on the initial setup, test running clock_gettime() against the TSC (Time Stamp Counter) showed that the precision could not be better than 900 ns.

We the hope of addressing the time deviation reported in the initial setup (see 2.3.5), we needed a more solid time base, with a precision as good as 100 ns or possibly better, and this is were the PRUSS comes in.

The two PRUSS are simple 32bit microcontrollers both clocked (separately) at 200 MHz. The both run independently from the CPU, but they share a little bit of everything: bus, memory, access to the GPIO. They are programmed directly in assembly language by means of a nice little compiler. So we programmed PRUSS 0 so to produce a counter updated exactly every 40 ns; the result is stored over 8 bytes (64bit) in PRU shared memory, and can be queried with no cost (PRU-wise) from the CPU code.

Listing 1: C++ code to get the high precision clock

```
long long BeagleBoneBlackRealTime::
    getHighPrecisionNanosecondTime()
    const
{
  if ( !prussManager_->
      getPruSharedMemory() )
    return getCpuNanosecondTime();
  return *((unsigned long long*)(
      prussManager_->
      getPruSharedMemory()+
      OFFSET_TIME64))*
      PRU_REALTIME_NS_PERIOD;
}
```

Listing 2: Assembly code to update the clock

```
BIG_LOOP:
  ADD r0, r0, 1
  QBEQ INC_HIGH_BYTES, r0, 0
  JMP CONTINUE1
INC_HIGH_BYTES:
  ADD r1, r1, 1
CONTINUE1:
  NOP
  // update the value in shared ram:
  SBBO r0, BASE_ADDRESS_REG, OFFSET_TIME64,
      8
  JMP BIG_LOOP
```

This very simple PRUSS-based code gives us a remarkably accurate, latency-free clock that is now a solid ground to build the active control system on[1].

## 3.5 An SPI-based DAC-ADC Cape

At the same time, we needed some DAC-ADC hopefully as good as the National Instrument board. A custom cape was then designed and implemented

[1]This could probably be improved with a semaphore to avoid concurrent access to the shared memory from the PRUSS and the CPU at the same time. The current design could perhaps explain the sporadic overrun that can be observed from time to time

using SPI-based high-speed 16bit conversion chips, connected to the real world with some extra analog circuitry. But the BeagleoOne Black's built-in SPI microcontroller could not be used to control the converters, due to the lack of support in Xenomai. Rather than developing a Xenomai-compatible SPI driver (with the risk of unexpected latency due to the SPI microcontroller), we developed a driver from scratch on PRUSS 1, taking advantage of the GPIO direct access.

Listing 3: ADC assembly code excerpt

```
// DAC part
SEND_DAC:
WAIT_DATAOUT_READY_1:
  // wait for the model data output ready
  LBCO R_DATAOUT_READY, CONST_PRUDRAM,
       AO_DATAOUT_READY, 1
  QBBC WAIT_DATAIN_REQUEST_1,
       R_DATAOUT_READY.t0 // QUICK BRANCH BIT
        CLEAR, skip if set
  LBCO R_OUTPUTVALUE, CONST_PRUDRAM,
       AO_OUTPUTVALUE, 2 // get the output
  CLR R_DATAOUT_READY.t0
  SBCO R_DATAOUT_READY, CONST_PRUDRAM,
       AO_DATAOUT_READY, 1
  // bit counter
  MOV BITCOUNTER, NBIT
  SET CS // start conversion
  // following instructions are for timing
  NOP
  NOP
  NOP
  CLR SCK_DAC
  (...)
```

This time, querying the ADC or sending a value to the DAC is synchronous, using semaphores:

Listing 4: C++ code to get a sample

```
float SignalConverter::acquireOneSampleNow(
    unsigned int /* channel */ )
{
  ((unsigned int*)pruss_->getPruMemory( 1 ))
      [OFFSET_DATAIN_REQUEST/4] = 1;
  while ( ((unsigned int*)pruss_->
      getPruMemory( 1 ))[OFFSET_DATAIN_READY
      /4] == 0 )
    {}
  float value = ((float)((unsigned int*)
      pruss_->getPruMemory( 1 ))[
      OFFSET_DATAIN/4]*VOLT_MAX)/SAMPLE_MAX;
  ((unsigned int*)pruss_->getPruMemory( 1 ))
      [OFFSET_DATAIN_REQUEST/4] = 0;
  return value;
}
```

In this particular setup, the round trip ADC/DAC latency (seen from the CPU) has proven to be always $< 3$ $\mu$s. So now we had a reliable clock, and an accurate in-out conversion.

## 3.6 Watching Mode Jump and Other Oddities

In order to track potential jumps from primary to secondary mode (hence, real time dropout), or other unexpected behaviors, we implemented a method described by BLAESS [8]:

Listing 5: Mode jump tracker

```
static const char* _Signals[] =
{
    [SIGDEBUG_UNDEFINED] = "undefined",
    (...)
    [SIGDEBUG_WATCHDOG] = "runaway thread (
        wouf! wouf! says the watchdog...)"
};

void BeagleBoneBlackRealTime::signalCatcher(
     int sig, siginfo_t *si, void *context )
{
    unsigned int reason = si->si_value.
        sival_int;
    void *bt[64];
    int nentries;
    printf( "\nSIGDEBUG received, reason %d:
        %s\n", reason, reason <=
        SIGDEBUG_WATCHDOG ? _Signals[reason]
         : "<unknown>" );
    nentries = backtrace( bt, sizeof(bt) /
        sizeof(bt[0]) );
    backtrace_symbols_fd( bt, nentries,
        fileno(stdout) );
}
```

## 3.7 Coding the Model

As said earlier, the Matlab model had to be completely rewritten. Our home-made linear algebra library was proofed here to compute the discretized form of the matrices (using zero-order hold) from their continuous expression. And fortunately, in the end, we got exactly the same numerical result as in Matlab.

Listing 6: Discretization from continuous data

```
void ModalControlModel::discretize()
{int dim = 2*numModes_;
    double T = samplingTime_;
    Ad_.resize( dim );
    Bd_.resize( dim );
    SquareMatrix A( A_ );
    SquareMatrix expAT = exp( A*T );
    Ad_ = expAT;
    SquareMatrix Id( dim );
    Id.identity();
    Bd_ = (A^-1)*(exp(A*T)-Id)*B_;
}
```

## 3.8 Results on the New Setup

Compared with the setup described in 2, we got better performance and better determinism with the new setup. Running equivalent models (with the same number of modes) was simply (and surprisingly) faster on the BeagleBone Black. Also, the sample overrun was much more seldom here: only 2 or 3 times over about 400.000 samples (10 s), although it would be interesting to find out why this is happening anyway.

These results are quite remarkable in themselves, already if simply we compare the price of the hardwares vs the performance: about 6.000€ for the PC equipped with a very expensive board, and about 200€ of the BeagleBone Black with its little custom cape.

### 3.8.1 Back to the Desktop

In another experiment, we adapted the new design to the "old" desktop-based hardware. Here also, the results were quite astonishing, and, with the same amount of modes (20), we got to a 437KHz record sample frequency, with a maximum of 3 overrun amongst about 1 million samples.

## 3.9 Temporary Conclusion

Comparing the price vs. the performance of the different setups, although interesting, is certainly not the most important aspect. Breaking free from Matlab has forced us to rethink the active control system with a fresh vision. In this new, clearer design, the Signal Converter (DAC, ADC), the Real Time System (real time task management, accurate clock), the Control Loop (time sampler) and the Model (real world system abstraction) are all separate. The Control Loop acts as a "music conductor", making the link between the other elements with as much chronological accuracy as possible.

# 4 Towards an Lightweight Abstract Active Control Framework

The term *abstract* is used here in different ways.

**Genericity**

First, abstract means *generic*, in that the core code is now no more directly tied to the initial problem (the modal active control of sound); it encapsulates a minimalistic process description that, we hope, all types of active control in MIMO context can share. This concept is expressed in the *ControlLoopFacade* class, which includes pure virtual methods as well as certain key implemented methods such as *step()* , *runModel()* or *adjustIdealSamplingTime()* which implementation should not vary from an active control problem to the other (note that although these key methods are currently defined as *virtual*, they ideally should not).

**Technically Agnostic**

And also, abstract means that no particular technical implementation is assumed. Its object-oriented design only imposes some simple requirements, such as being able to answer the question "what time is it?", with nanosecond precision, or the knowledge that the active control model should have of what "stepping" from a time sample to the next should mean in a certain context.

This is well expressed in the classes *ModelInterface*, *RealTimeInterface* and *SignalConverterInterface*, all pure virtual interfaces.

Lastly, some classes should preferably be implemented as singleton (as shown in the examples), but this is not a requirement, not it is an important aspect.

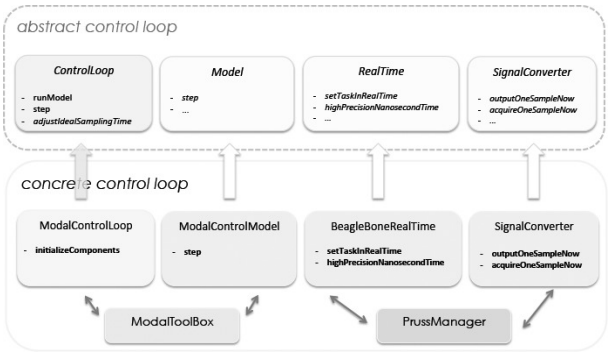


**FIGURE 4:** *The "abstract" active control model over its implementation in the BeagleBone Black context.*

## 4.1 Distribution

The code of the Lightweight Active Control Framework shall be sent as is and freely upon demand. At this time, it is delivered with 3 examples to start with: Xenomai for Desktop, Xenomai for BeagleBone, and "dummy" example with a generic implementation.

# 5 Conclusion

From an experimental setup in a musical context, where the real time aspect appeared to be critical, we have come to a generic design for describing active control systems, that could potentially fit any real time layer, hardware, model, and even operating system, only and only if they all can meet certain requirements that the abstract layer imposes. As a good sign, beside the fact that the code is obviously clearer, some faster performance and better chronological determinism can already be observed in its first implementations, compared with the straight code generation from Simulink. However we consider this design to be still in gestation, and by far. The definition of its interfaces, as well as its core engine, shall need to be greatly improved, and proofed with more real world cases in order to be as simple and elegant as we aim it to be.

# References


[1] *Active Control of String Instruments using Xenomai*, S. Benacchio, R. Piéchaud, A. Mamou-Mani, V. Finel, B. Chomette, RTLW 2013.

[2] *Active Control of Vibration*, C. R. Fuller, S. J. Elliott and P. A. Nelson, Harcourt Brace & Company, 1997.

[3] *Mode tuning of a simplified string instrument using time-dimensionless state-derivative control*, S. Benacchio, B. Chomette, A. Mamou-Mani and V. Finel, Journal of Sound and Vibration, 2014.

[4] *http://elinux.org/Beagleboard:BeagleBoneBlack*

[5] *http://arma.sourceforge.net/*

[6] *http://brunosmartins.info/xenomai-on-the-beaglebone-black-in-14-easy-steps/*

[7] *http://people.sc.fsu.edu/ jburkardt/cpp_src/fn/fn.html*

[8] *Solutions temps réel sous Linux*, C. Blaess, Editions Eyrolles, 2012.


# Acknowledgement


Thanks to Peter Wurmsdobler for his invaluable advice and encouragements.
Thanks to Alexis Martin for the design and implementation of the experimental ADC-DAC cape over the BeagleBone Black, and the PRUSS programming (IRCAM, May-July 2014).