'Semiconducting – Making Music After the Transistor'[1]
Nicolas Collins

Abstract
Why does 'Computer Music' sound different from 'Electronic Music'? The author examines several traits that distinguish hardware from software in terms of their application in music composition and performance. He discusses the often subtle influence of these differences on various aspects of the creative process, and presents a number of inferences as to the 'intrinsic' suitability of hardware and software for different musical tasks. His observations are based on several decades of experience as a composer and performer, and in close engagement with the music of his mentors and peers.

Introduction
At some point in the late 1980s the composer Ron Kuivila proclaimed, 'we have to make computer music that sounds like electronic music.'[2] This might appear a mere semantic distinction. At that time the average listener would dismiss any music produced with electronic technology – be it a Moog or Macintosh – as 'boops and beeps.' But Kuivila presciently drew attention to a looming fork in the musical road: boops and beeps were splitting into boops and bits. Over the coming decades, as the computer evolved into an unimaginably powerful and versatile musical tool, this distinction would exert a subtle but significant influence on music.

Kuivila and I had met in 1973 at Wesleyan University, where we both were undergraduates studying with Alvin Lucier. Under the guidance of mentors such as David Tudor and David Behrman, we began building circuits in the early 1970s, and finished out the decade programming pre-Apple microcomputers like the Kim 1. The music that emerged from our shambolic arrays of unreliable homemade circuits fit well into the experimental aesthetic that pervaded the times (the fact that we were bad engineers probably made our music better by the standards of our community). Nonetheless we saw great potential in those crude early personal computers, and many of us welcomed the chance to hang up the soldering iron and start programming.[3]

The Ataris, Amigas and Apples that we adopted in the course of the 1980s were vastly easier to program than our first machines, but they still lacked the speed and processor power needed to generate complex sound directly. Most Computer Music composers of the day hitched their machines to MIDI synthesizers, but even the vaunted Yamaha DX7 was no match for the irrational weirdness of a table strewn with Tudor's idiosyncratic circuits arrayed in unstable feedback matrices. One bottleneck lay in MIDI's crudely quantized data format, which had been optimized for triggering equal-tempered notes, and was ill suited for complex, continuous changes in sound textures. On a more profound level, MIDI 'exploded' the musical instrument, separating sound (synthesizer) from gesture (keyboard, drum pads, or other controller) – we gained a Lego-like flexibility to build novel instruments, but we severed the tight

feedback between body and sound that existed in most traditional, pre-MIDI instruments and we lost a certain degree of touch and nuance[4].

By 2013 MIDI no longer stands between code and sound: any laptop has the power to generate directly a reasonable simulation of almost any electronic sound – or at least to play back a sample of it. But I'm not sure that Kuivila's goal has yet been met. I can still hear a difference between hardware and software. Why?

After all, most music today that employs any kind of electronic technology depends on a combination of hardware and software resources. Although crafted and/or recorded in code, digital music reaches our ears through a chain of transistors, mechanical devices, speakers and earphones. 'Circuit Benders' who open and modify electronic toys in pursuit of new sounds often espouse a distinctly anti-computer aesthetic, but the vast majority of the toys they hack in fact consist of embedded microcontrollers playing back audio samples – one gizmo is distinguished from another not by its visible hardware but by the program hidden in ROM. Still, whereas a strict hardware/software dialectic can't hold water for very long, arrays of semiconductors and lines of code are imbued with various distinctive traits that combine to determine the essential 'hardware-ness' or 'software-ness' of any particular chunk of modern technology.

Some of these traits are reflected directly in sound – with sufficient attention or guidance one can often hear the difference between sounds produced by a hardware-dominated system versus those crafted largely in software. Others influence working habits -- how we compose with a certain technology, or how we interact with it in performance; sometimes this influence is obvious, but at other times it can be so subtle as to verge on unconscious suggestion. Many of these domain-specific characteristics can be ignored or repressed to some degree, just like a short person can devote himself to basketball, but they nonetheless affect the likelihood of one choosing a particular device for a specific application, and inevitably exert an influence on the resulting music.

I want to draw attention to some distinctive differences between hardware and software tools as applied to music composition and performance. I am not particularly interested in any absolute qualities inherent in the technology, but in the ways certain technological characteristics influence how we think and work, and the ways in which the historic persistence of those influences can predispose an artist to favor specific tools for specific tasks. My observations are based on several decades of personal experience: in my own activity as a composer and performer, and in my familiarity with the music of my mentors and peers, as observed and discussed with them since my student days. I acknowledge that my perspective comes from a fringe of musical culture and I contribute these remarks in the interest of fostering discussion, rather than to prove a specific thesis.

I should qualify some of the terms I will be using. When I speak of 'hardware' I mean not only electronic circuitry, but also mechanical and electromechanical

devices from traditional acoustic instruments to electric guitars. By 'software' I'm designating computer code as we know it today, whether running on a personal computer or embedded in a dedicated microcontroller or DSP. I use the words 'infinite' and 'random' not in their scientific sense, but rather as one might in casual conversation, to mean 'a hell of a lot' (the former) and 'really unpredictable' (the latter).

The Traits

Here are what I see as the most significant features distinguishing software from hardware in terms of their apparent (or at least perceived) suitability for specific musical tasks, and their often-unremarked influence on musical processes:

- Traditional acoustic instruments are three-dimensional objects, radiating sound in every direction, filling the volume of architectural space like syrup spreading over a waffle. Electronic circuits are much flatter, essentially two-dimensional. Software is inherently linear, every program a one-dimensional string of code. In an outtake from his 1976 interview with Robert Ashley for Ashley's *Music With Roots in the Aether*, Alvin Lucier justified his lack of interest in the hardware of electronic music with the statement, 'sound is three-dimensional, but circuits are flat.'[5] At the time Lucier was deeply engaged with sound's behavior in acoustic space, and he regarded the 'flatness' of circuitry as a fundamental weakness in the work of composers in thrall to homemade circuitry, as was quite prevalent at the time. As a playing field a circuit may never be able to embody the topographic richness of standing waves in a room, but at least a two-dimensional array of electronic components on a fiberglass board allows for the simultaneous, parallel activity of multiple strands of electron flow, and the resulting sounds often approach the polyphonic density of traditional music in three-dimensional space. In software most action is sequential, and all sounds queue up through a linear pipeline for digital to analog conversion. With sufficient processor speed and the right programming environment one can create the impression of simultaneity, but this is usually an illusion -- much like a Bach flute sonata weaving a monophonic line of melody into contrapuntal chords[6]. Given the ludicrous speed of modern computers this distinction might seem academic -- modern software does an excellent job of simulating simultaneity. Moreover, 'processor farms' and certain Digital Signal Processor (DSP) systems do allow true simultaneous execution of multiple software routines. But these latter technologies are far from commonplace in music circles and, like writing prose, the act of writing code (even for parallel processors) invariably nudges the programmer in the direction of sequential thinking. This linear methodology can affect the essential character of work produced in software.
- Hardware occupies the physical world and is appropriately constrained in its behavior by various natural and universal mechanical and electrical laws and limits. Software is ethereal -- its constraints are artificial, different for every language, the result of linguistic design rather than pre-existing physical laws. When selecting a potentiometer for inclusion in a

circuit a designer has a finite number of options in terms of maximum resistance, curve of resistive change (i.e., linear or logarithmic), number of degrees of rotation, length of its slider, etc.; and these characteristics are fixed at the point of manufacture. When implementing a potentiometer in software all these parameters are infinitely variable, and can be replaced with the click of a mouse. Hardware has edges; software is a tabula rasa wrapped around a torus.

- As a result of its physicality, hardware – especially mechanical devices – often displays non-linear adjacencies similar to state-changes in the natural world (think of the transition of water to ice or vapor). Pick a note on a guitar and then slowly raise your fretting finger until the smooth decay is abruptly choked off by a burst of enharmonic buzzing as the string clatters against the fret. In the physical domain of the guitar these two sounds – the familiar plucked string and its noisy dying skronk – are immediately adjacent to one another, separated by the slightest movement of a finger. Either sound can be simulated in software, but each requires a wholly different block of code: no single variable in the venerable Karplus-Strong 'plucked string algorithm'[7] can be nudged by a single bit to produce a similar death rattle; this kind of adjacency must be programmed at a higher level, and does not typically exist in the natural order of a programming language. Generally speaking, adjacency in software remains very linear, while the world of hardware abounds with abrupt transitions. A break point in a hardware instrument – fret buzz on a guitar, the unpredictable squeal of the STEIM Cracklebox[8] – can be painstakingly avoided or joyously exploited, but is always lurking in the background, a risk, an essential property of the instrument.
- Most software is inherently binary: it either works correctly or fails catastrophically, and when corrupted code crashes the result is usually silence. Hardware performs along on a continuum that stretches from the optimum behavior intended by its designers to irreversible, smoky failure; circuitry – especially analog circuitry – usually produces sound even as it veers toward breakdown. Overdriving an amplifier to distort a guitar, feeding back between a microphone and a speaker to play a room's resonant frequencies, 'starving' the power supply voltage in an electronic toy to produce erratic behavior – these 'misuses' of circuitry generate sonic artifacts that can be analyzed and modeled in software, but the risky processes themselves (saturation, feedback, under-voltage) are very difficult to transfer intact from the domain of hardware to that of software while preserving functionality in the code[9]. Writing software favors Boolean thinking – self- destructive code remains the purview of hackers who craft worms and Trojan Horses for the specific purpose of crashing or corrupting computers.
- Software is deterministic, while all hardware is indeterminate to some degree. Once debugged, code runs the same every time. Hardware is notoriously unrepeatable: consider recreating a patch on an analog synthesizer, restoring mixdown settings on a pre-automation mixer, or even tuning a guitar. The British computer scientist John Bowers once observed that he had never managed write a 'random' computer program

that would run, but that he could make 'random' component substitutions and connections in a circuit with a high certainty of a sonic outcome (a classic technique of Circuit Bending)[10].

- Hardware is unique, software is a multiple. Hardware is constrained in its 'thinginess' by number: whether handcrafted or mass-produced, each iteration of a hardware device requires a measurable investment of time and materials. Software's lack of physical constraint gives it tremendous powers of duplication and dissemination. Lines of code can be cloned with a simple cmd-C/cmd-V: building 76 oscillators into a software instrument takes barely more time than one, and no more resources beyond the computer platform and development software needed for the first. In software there is no distinction between an original and a copy: MP3 audio files, PDFs of scores, and runtime versions of music programs can be downloaded and shared thousands of times without any deterioration or loss of the matrix – any copy is as good as the master. If a piano is a typical example of traditional musical hardware, the pre-digital equivalent of the software multiple would lie somewhere between a printed score (easily and accurately reproduced and distributed, but at a quantifiable – if modest -- unit cost) and the folk song (freely shared by oral tradition, but more likely to be transformed in its transmission.) Way too many words have already been written on the significance of this trait of software – of its impact on the character and profitability of publishing as it was understood before the advent of the World Wide Web; I will simply point out that if all information wants to be free, that freedom has been attained by software, but is still beyond the reach of hardware. (I should add that software's multiplicity is accompanied by virtual weightlessness, while hardware is still heavy, as every touring musician knows too well.)
- Software accepts infinite undo's, is eminently tweakable. But once the solder cools, hardware resists change. I have long maintained that the young circuit-building composers of the 1970s switched to programming by the end of that decade because, for all the headaches induced by writing lines of machine language on calculator-sized keypads, it was still easier to debug code than to de-solder chips. Software invites endless updates, where hardware begs you to close the box and never open it again. Software is good for composing and editing, for keeping things in a state of flux; hardware is good for making reasonably stable, playable instruments that you can return to with a sense of familiarity (even if they have to be tuned). The natural outcome of software's malleability has been the extension of the programming process from the private and invisible pre-concert preparation of a composition, to an active element of the actual performance -- as witnessed in the rise of 'live coding' culture practiced by devotees of SuperCollider[11] and Chuck[12] programming languages, for example. Live circuit building has been a fringe activity at best: David Tudor finishing circuits in the pit while Merce Cunningham danced overhead; the group Loud Objects soldering PICs on top of an overhead projector[13]; live coding vs. live circuit building in ongoing

competition between the younger Nick Collins (UK) and the author for the *Nic(k) Collins Cup*[14].

- On the other hand, once a program is burned into ROM and its source code is no longer accessible, software flips into an inviolable state. At this point re-soldering, for all it unpleasantness, remains the only option for effecting change. Circuit Benders hack digital toys not by rewriting the code (typically sealed under a malevolent beauty-mark of black epoxy) but by messing about with traces and components on the circuit board. A hardware hack is always lurking as a last resort, like a shim bar when you lock your keys in the car.
- Thanks to computer memory, software can work with time. The transition from analog circuitry to programmable microcomputers gave composers a new tool that combined characteristics of instrument, score and performer: memory allows software to play back prerecorded sounds (an instrument), script a sequence of events in time (a score), and make decisions built on past experience (a performer.) Before computers, electronic circuitry was used primarily in an instrumental capacity – to produce sounds immediately[15]. It took software-driven microcomputers to fuse this trio of resources into a new paradigm for music creation.
- Given the sheer speed of modern personal computers and software's quasi-infinite power of duplication (see above), software has a distinct edge over hardware in the density of musical texture it can produce: a circuit is to code as a solo violin is to the full orchestra. But at extremes of its behavior hardware can exhibit a degree of complexity that still seems beyond the power of software to simulate effectively: initial tug of rosined bow hair on the string of the violin; the unstable squeal of wet fingers on a radio's circuit board; the supply voltage collapsing in a cheap electronic keyboard. Hardware still does a better job of giving voice to the irrational, the chaotic, the unstable.
- Software is imbued with an ineffable sense of now -- it is the technology of the present, and we are forever downloading and updating to keep it current. Hardware is yesterday, the tools that were supplanted by software. Vinyl records, patchcord synthesizers and tape recorders have been replaced by MP3 files, software samplers and ProTools. In the ears and minds of most users this is an improvement – software does the job 'better' than its hardware antecedents. Before any given tool is replaced by a superior device, qualities that don't serve its main purpose can be seen as weaknesses, defects, or failures: the ticks and pops of vinyl records, oscillators drifting out of tune, tape hiss and distortion. But when a technology is no longer relied upon for its original purpose, these same qualities can become interesting in and of themselves. The return to 'outmoded' hardware is not always a question of nostalgia, but often an indication that the scales have dropped from our ears.

Hybrids

There are three areas of software/hardware crossover that deserve mention here: interfaces for connecting computers (and, more pointedly, their resident

software) to external hardware devices; software applications designed to emulate hardware devices; and the emergence of affordable rapid prototyping technology.

The most ubiquitous of the hardware interfaces today is the Arduino, a small, inexpensive microcontroller designed by Massimo Banzi and David Cuartielles in 2005[16]. The Arduino and its brethren and ancestors[17] facilitate the connection of a computer to input and output devices, such as tactile sensors and motors. Such an interface can imbue a computer program with some of the characteristics we associate with hardware, but there always remains a MIDI-tinged sense of mediation (a result of the conversion between the analog to digital domains) that makes performing with these hybrid instruments slightly hazier than with a purely hardware device – think of manipulating a robotic arm with a joystick, or hugging an infant in an incubator while wearing rubber gloves.

The past decade has also seen a proliferation of software emulations of hardware devices, from smart phone apps that simulate vintage analog synthesizers, to iMovie filters that make your HD video recording look like scratchy Super 8 film. The market forces behind this development (nostalgia, digital fatigue, etc.) lie outside of the scope of this paper, but it is important to note here that these emulations succeed by focusing on those aspects of a hardware device most easily approximated in the software domain: the virtual Moog synthesizer models the sound of analog oscillators and filters, but doesn't try to approximate the glitch of a dirty pot or the pop of inserting a patchcord; the video effect alters the color balance and superimposes algorithmically generated scratches, but does not let you misapply the splicing tape or spill acid on the emulsion.

Although at the time of writing affordable 3D printers and rapid prototyping devices remain the purview of the serious DIY practitioner, there is no question that these technologies will enter the marketplace in the near future. When they do the barrier between freely distributable software and tactile hardware objects will become quite permeable. A look thru the Etsy website reveals how independent entrepreneurs have already employed this technology to extend the publishing notion of "print on demand" to something close to "wish on demand."[18]

Conclusion

I came of age as a musician during the era of the 'composer-performer': the Sonic Arts Union, David Tudor, Terry Riley, LaMonte Young, Pauline Oliveros, Steve Reich, Philip Glass. Sometimes this dual role was a matter of simple expediency (established orchestras and ensembles wouldn't touch the music of these young mavericks at the time), but more often it was a desire to retain direct, personal control that led to a flowering of composer-led ensembles that resembled rock bands more than orchestras. Fifty years on, the computer – with its above-mentioned power to fuse three principle components of music production – has emerged as the natural tool for this style of working.

But another factor driving composers to become performers was the spirit of improvisation. The generation of artists listed above may have been trained in a

rigorous classical tradition, but by the late 1960s it was no longer possible to ignore the musical world outside the gates of academe or beyond the doors of the European concert hall. What was then known as 'World Music' was reaching American and European ears through a trickle of records and concerts. Progressive Jazz was in full flower. Pop was inescapable. And composers of my age – the following generation -- had no need to reject an older tradition to strike out in a new direction: Indian music, Miles Davis, the Beatles, John Cage, Charles Ives and Monteverdi were all laid out in front of us like a buffet, and we could heap our plates with whatever pleased us, regardless of how odd the juxtapositions might seem. Improvisation was an essential ingredient, and we sought technology that expanded the horizons of improvisation and performance, just as we experimented with new techniques and tools for composition.

It is in the area of performance that I feel hardware – with its tactile, sometimes unruly properties -- still holds the upper hand. This testifies not to any failure of software to make good on its perceived promise of making everything better in our lives, but to a pragmatic affirmation of the sometimes messy but inarguably fascinating irrationality of human beings: sometimes we need the imperfection of things.

---

[1] This is a revision of a lecture first presented at the 'Technology and Aesthetics' Symposium, NOTAM (Norwegian Center for Technology in Music and the Arts), Oslo, May 26-27 2011. Revised for presentation at 'Musical Listening in the Age of Technological Reproducibility' conference March 2013. The author gratefully acknowledges NOTAM's support of his research.

[2] Private conversation, New York City, exact date unknown.

[3] Although this potential was clear to our small band of binary pioneers, the notion was so inconceivable to the early developers of personal computers that Apple trademarked its name with the specific limitation that its machines would never be used for musical applications, lest it infringe on the Beatles' semi-dormant company of the same name – a decision that would lead to extended litigation after the introduction of the iPod and iTunes. This despite the fact that the very first non-diagnostic software written and demonstrated at the Homebrew Computer Club in Menlo Park, CA in 1975 was a music program by Steve Dompier, an event attended by a young Steve Jobs (see http://www.convivialtools.net/index.php?title=Homebrew_Computer_Club) (accessed on February 21, 2013).

[4] For more on the implications of MIDI's separation of sound from gesture see Collins, Nicolas, 1998. Ubiquitous Electronics -- Technology and Live Performance 1966-1996. *Leonardo Music Journal* Vol. 8. San Francisco/Cambridge 27-32. One magnificent exception to the gesture/sound disconnect that MIDI inflicted on most computer music composers was Tim Perkis' 'Mouseguitar'

project of 1987, which displayed much of the tactile nuance of Tudor-esque circuitry. In Perkis' words:

> When I switched to the FM synth (Yamaha TX81Z), there weren't any keydowns involved; it was all one 'note'... The beauty of that synth -- and why I still use it! -- is that its failure modes are quite beautiful, and that live patch editing [can] go on while a voice is sounding without predictable and annoying glitches. The barrage of sysex data – including simulated front panel button-presses, for some sound modifications that were only accessible that way -- went on without cease throughout the performance. The minute I started playing the display said 'midi buffer full' and it stayed that way until I stopped.

(Email from Tim Perkis, July 18, 2006.)

[5] This quote is drawn from my memory of working as a technical assistant on the recording session, and has been confirmed by one other colleague present at the time. This version of the interview was not used in the final video series, or transcribed in the book, but Lucier has made similar observations in subsequent interviews.

[6] This limitation in software was a major factor motivating Kuivila to develop, with David Anderson, the programming language 'Formula', whose strength lay in its accurate control of the timing and synchronization of parallel musical events – getting linear code a little closer to planar. See http://www.eecs.berkeley.edu/Pubs/TechRpts/1991/5643.html (Accessed February 21, 2013)

[7] Karplus, Keven and Strong, Alex. 1983. Digital Synthesis of Plucked String and Drum Timbres. *Computer Music Journal* **7** (2). Cambridge. 43–55.

[8] See http://steim.org/product/cracklebox/ (Accessed February 21, 2013)

[9] This comparison echoes the familiar 'digital vs. analog' distinction, but I prefer to focus on the difference between software and hardware because even digital hardware can be made to sing outside its 'normal' mode of operation.

[10] Private conversation, Norwich, England, January 2004.

[11] See http://supercollider.sourceforge.net/ (Accessed February 21, 2012=3)

[12] See http://chuck.cs.princeton.edu/ (Accessed February 21, 2013)

[13] See http://www.loudobjects.com/ (Accessed February 21, 2013)

[14] See http://www.nicolascollins.com/collinscup.htm (Accessed February 21, 2013)

[15] Beginning in the late 1960s a handful of artist-engineers designed and built pre-computer circuits that embodied some degree of performer-like decision-making: Gordon Mumma's 'Cybersonic Consoles' (1960s-70s), which as far as I can figure out were some kind of analog computers; my own multi-player instruments built from CMOS logic chips in emulation of Christian Wolff's 'co-ordination' notation (1978). The final stages of development of David Behrman's 'Homemade Synthesizer' included a primitive sequencer that varied pre-scored chord sequences in response to pitches played by a cellist ('Cello With Melody Driven Electronics', 1975) presaging Behrman's subsequent interactive work with computers. And digital delays begat a whole school of post-Terry Riley canonical performance based on looping and sustaining sounds from a performance's immediate past into its ongoing present.

[16] http://www.arduino.cc/ (Accessed March 27, 2013). Close to 500,000 Arduinos have been sold as of early 2013.

[17] Most notably the STEIM SensorLab (see http://steim.org/product/discontinued-products/) (Accessed March 28, 2013) and the Infusion System's I-Cube (see http://infusionsystems.com/catalog/index.php and http://www.xspasm.com/x/sfu/vmi/ISEA95.html) (Accessed March 28, 2013), more expensive MIDI-based interfaces produced in the 1990s.

[18] http://www.etsy.com (Accessed November 8, 2013.)