The Other Software

Chandler B. McWilliams
University of California Los Angeles
Department of Design | Media Arts
chandler@brysonian.com

ABSTRACT

This paper considers the absence of the human actor, specifically the programmer, from Friedrich Kittler's analysis of software in his essay *There is no Software*. By focusing too intently on the machine and its specific, material existence, Kittler removes the human user / operator / writer from his analysis of software. Thus, he has no choice but to interpret the layers of language, assembler, opcode and WordPerfect, DOS, BIOS—both chains ending in an essentializing reduction to voltages—as an attempt to obfuscate the material operations of the machine in the name of intellectual property.

By both reasserting the presence of the programmer within Kittler's structure, and attacking the conception of code-as-text, this essay offers an alternate description of the being of software, one which emphasizes not just the execution of code on the machine, but also the programmer's role as reader and writer of code.

Keywords

Software studies, media studies, critical code studies.

1. INTRODUCTION

Friedrich Kittler's essay *There is no Software* argues for a narrowly defined, materialist conception of authentic uses of circuit-based computational machines. To this end, he dismisses software and high-level code languages, along with architectural abstractions inherent in the design of most computer systems, as unnecessary obfuscations which hide the nature of the machine itself. Furthermore, he deploys a conception of code-as-text the writing of which can be understood in terms familiar to the writing of natural language. These two ideas together necessarily leave little room for the programmer in the creation of computational artifacts.

2. NOT EXACTLY WRITING

Programming *language*, *writing* code; it seems clear that we are talking about language in the everyday sense, about texts that are written. Software is of course written in a language, so why not bring to bear the mature theories of text on the world of software? *There is no Software*, like many texts in the nascent field of software studies, talks frequently of writing. But the word *writing* offers up an irresistible temptation to talk about text. Writing code is *writing* like writing music is *writing*. This is not to say that music theory is the place to find insights into software, but to emphasize the multiple flexible meanings of the word writing. Music can be written sitting at a piano, with clicks on the screen, or with symbols on paper. Text may or may not be involved. Writing music is about manipulating sound. Few would argue that theories of text are relevant to understanding music, we should be similarly wary of over-identifing code with written text. Just as writing music is about manipulating sound, not symbols, writing software is about manipulating procedures, not language.

The flip side of understanding code-as-text is a conception of reading that places the machine at the center of the act of programming. Reading becomes reading-by-the-machine. Source code is compiled and turned into assembly, then translated into opcodes, which eventually become voltages, the final, true language of the machine. It is voltages, after all, that integrated circuits traffic in. "All code operations, despite such metaphoric faculties as call or return, come down to absolutely local string manipulations, that is, I am afraid, *to signifiers of voltage differences*." [4]

Even the binary codes we're told so much about are an abstraction on top of these voltages, 1 is just a name for five volts, and 0 is a name for ground.[1] Combined with literary theory's concern for what our writing does (a concern shared by Kittler), reading-by-the-machine offers a clear answer for software: writing code produces a synchronized choreography of voltages in integrated circuits.

It is all too easy to collapse the entire process and thus to do away with software. If code is always and only concerned with the eventual manipulation of voltages, then in a sense, what does it matter? Why this long process? We know what our writing does, it (eventually) affects voltage differentials in a silicon chip. Wouldn't it be simpler to slough off these abstractions and get as close to the machine as possible?

We are rightly struck by the mystery of the written word. The magic of not being able to map phrases in natural language[2] to specific behaviors or states in the mind of the reader is enticing. The wonderful ambiguity of language.

> Writing, in Western culture, automatically dictates that we place ourselves in the virtual space of self-representation and reduplication; since writing refers not to a thing, but to

[1] These values aren't perfect; there is often a tolerance built in such that, for example, zero to two volts represent a binary 0, and anything higher represents a binary 1.

[2] Florian Cramer takes issue with the use of the term "natural language" as opposed to "formal" or "programming" language. I take "natural" in this context to refer more to the development and etymology of spoken languages vis-à-vis programming languages rather than a claim about ontological status. [2]

Digital Arts and Culture, December 12–15, 2009, Irvine, California, USA.

> speech, a work of language only advances more deeply into the intangible density of the mirror, calls forth the double of this already doubled writing, discovers in this way a possible and impossible infinity, ceaselessly strives after speech, maintains it beyond the death which condemns it, and frees a murmuring stream. This presence of repeated speech in writing undeniably gives to what we call a work of language an ontological status unknown in those cultures where the act of writing designates the thing itself, in its proper and visible body, stubbornly inaccessible to time. [3]

Is this then the status of writing software? Writing which designates the thing (voltages) in itself? Continuing this line of thought, the ambiguity of natural language is lost on the machine; there is only source code to assembly to opcodes to voltages. Read as a one-to-one mapping; the meaning and action is understood, explained, fixed. If this is the *reading* to compliment the *writing* of software, then the writer of code is no more than an operator manipulating a machine to produce a specific predetermined result. Like a switchboard operator, meticulously and uncreatively plugging circuits.

3. NOT EXACTLY VOLTAGES

When code is understood as literature, and so the techniques and questions of literary theory are asked of it, the machine will always emerge as the final answer. Missing from these discussions are the writer, the coder, the programmer. Rejecting the notion that code operates as a literary text allows us to reassert the presence of the coder in the code.

But if not text, then what? To avoid the temptations of the terms text, writing, and literature, let us say that code is an *artifact*. Specifically an artifact for describing and designing procedures and systems.[3] Code comes in many languages. Unlike the babel of human languages, programming languages tend to differ depending on how quickly the software can be coded, how easily the code is to write (there is a tacit relationship between the difficulty of writing the code and the speed with which the code will run), or on what hardware the the software needs to run. The key differentiating factors however are the assumptions inherent in each language about how the coder *thinks*. This last and most important feature of a programming language is perhaps the primary reason for the development of new languages, methodologies, and cognitive styles. A consequence of this computational babel effect is that there is not one ideal language for writing a given piece of software. The choice of language is far more influenced by the skillset and needs of the coder than properties of the software being coded. There is rarely if ever one clear, best choice. What's more, it is not always possible to identify which requirements of the software will play a central determining role in making a choice of language until the coding has already begun. Thus the practice of prototyping and sketching solutions in a familiar environment to gain a better understanding of the problem itself. Here is one similarity with the writing of a text; the process of coding itself is often the only way to gain an understanding of what is to be coded.

An over-emphasis on the voltages in the silicon cannot account for the multiplicity of programming languages. The same piece of software, if written in a different language, will become a *different* set of voltages when compiled and executed. And since there are no clear machine-centric metrics by which to judge one set of opcodes as superior to another—judgements about speed, memory efficiency, etc. are all relative to the purposes of the human user—then how can these voltages serve as the ultimate measure of what it is that our (software) writing does? It cannot, and it fails to do so because writing code is not about manipulating voltages any more than writing music is about manipulating vibrations.

Processes and systems are the core concern of code. Writing software is writing processes and procedures; defining rules and bounds for action, creating the possibility for behavior, form, and interaction. From the simplest utility script to the most realistic physics simulator, the common thread is the code which describes a particular process for achieving a task. There are of course, varying degrees of open-endedness and complexity between these two examples. A script that converts file names to lower case will, if written well, perform reliably and always produce the expected outcome. A simulation like those in video games is less straightforward. The outcome is not always predictable, this allows for a unique and rewarding gaming experience when played repeatedly, but also opens the door for simulation as a tool to model and eventually predict the outcome of the interaction of massive numbers of variables and processes. This can perhaps be most clearly stated in terms of how code operates in the arts. Here generative and parametric processes create the *possibility* of form; code creates a world of possibility within constraints rather than a *particular* form.

Though the vast majority of code is text-based, meaning it is written using the characters and symbols common to the ASCII specification, code can also be visual, in which the processes of "writing" entails connecting graphical elements called *patches* much like programming a modular analog synthesizer or connecting an electronic circuit[4]. Given Kittler's revulsion at the graphical user interface ("[O]n an intentionally superficial level, perfect graphic user interfaces, since they dispense with writing itself, hide a whole machine from its users.") it is fair to assume that visual programming languages would meet with equal scorn. This criticism rests, however, on the conception of code-as-text, one which we are in the process of letting go of in favor of understanding code as the articulation of processes.

4. WHERE IS CODE EXECUTED?

A piece of software could always have been written in a different language yet perform the same task; often performing that task in a different way, with different voltages in the machine, and different mental models in the coder. Loosening the relationship between code and the machine lets us ask the question: Where is code executed? By Kittler's account, only in the machine after its eventual conversion to voltages; the programmer is only an operator tasked with controlling the machine. But if programming languages often perform the same task differently, offering important differences only to the programmer, then what do these differences tell us about programming? The keys ways in which languages differ is in terms of the mental models they offer and

[3] Noah Wardrup-Fruin explores this idea and the expressive potential of software in his book *Expressive Processing* [6]

[4] The most well know patch-based environment is MAX/MSP, but newer patching systems include VVVV, Quartz Composer, and Grasshopper.

the assumptions they make about how code should be written. The multiplicity of ways of thinking about software indicates that code must to some extent be *run* in the mind of the programmer. Run with far less speed, complexity, and precision, but executed nevertheless. How else would programming be possible? The extent to which we can recognize that a programmer knows what effect a line of code will have is precisely the extent to which we can recognize that she has already run a simulation of that code in her head. The only other option is that coding is just smashing together symbols which are then sent to the machine with fingers crossed. So software must have an effect on how the programmer conceptualizes a problem—in the form of mental models and cognitive styles—and also exists as a description of a procedure interpretable by other coders without the intervention of the machine, without ever becoming voltages. To speak of software only in terms of voltages is no more interesting that to discuss a painting only in terms of the electro-chemical activity in the brain of the painter.

5. OBFUSCATION / ABSTRACTION

Kittler takes pains to describe the layers operating behind software. High-level programming languages are turned into assembly which then becomes opcodes and eventually voltages in the circuitry of the machine. This parallels another layering, an application (WordPerfect in his example) running over an operating system, which is in turn running on top of BIOS. He characterizes this as obfuscation, a complicated series of frauds perpetrated in the name of preserving (creating) intellectual property. However this layering is not so simple. Take the example of a driver and car. The car has pedals and a wheel which hide the underlying details of how it moves. The driver rarely needs to know if the car is powered using gas or electricity, he just needs to know that pushing the long vertical pedal makes it go. In programming terms we might say that the interface of the car (pedals and wheel) hides the implementation (engine, transmission, etc). Much like the abstractions of software, these abstractions are often useful, but can effectively put the user at the mercy of the designers of the system in question. Just as it is increasingly difficult to repair a new car in a home garage, the unavailability of the source code for popular software packages make them nearly impossible to alter.

However, there are other purposes for these abstractions and seeming obfuscations. For one, higher-level programming languages are "higher" in that their syntax and organization does not directly parallel the opcodes required by the machine. This makes them easier to learn and to think with. One can imagine a continuum between machine language and human language. Along this line, "higher" simply means a few steps closer to the human and away from the machine. Without these abstractions, programming would likely still be something of a dark art performed by self-appointed wizards at well-funded universities. But as it is we have visual languages, scripting languages, languages for artists, and languages for children. To write code for the machine always requires a change in our thought; points on this continuum never fully reach the human. It is always a meeting somewhere in-between: a becoming-machine of the programmer. This becoming cannot be summarized with phrases like "think like a machine." It is instead a thinking-along-with the machine. A direct engagement with the structures and potentials of a particular machine running a particular piece of code. It is here that code differs from other process-oriented languages, the most pervasive of which are legal codes. The law turns on interpretation of language and precedent; the meaning and application of legal documents evolve over time. Software codes on the other hand do not afford such ambiguity. The play and flexibility in software operates at the level of the processes being written, not at the level of language. Insofar as one uses a standardized language ANSI C, Java, etc, there is the assumption of standard execution, something unique to computation, and something which obscures the the obverse side of procedural thought.

Quoting *The Waite Group's Macroassembler Bible*, Kittler tells us that "BIOS services hide the details of controlling the underlying hardware from your program." [4] This hiding is not necessarily malicious. Because programming requires concurrent reading and executing of code in the mind of the programmer, increases in complexity of the software necessarily bring increases in the difficulty of the mental execution. Rather than always-already indicative of a patronizing concealment, this "hiding" is often a useful tool to allow a complex system to be modularized and thus thought-through. There is only so much one can hold in one's head at one time. This process of encapsulation thus allows the coder to trust that a certain element will behave as advertised until she needs to change the behavior or the element does something unexpected. The heart needn't worry how the liver works as long as it keeps working.

6. HARDWARE ESSENTIALISM

Integrated circuits, the hardware at the core of all digital computers, require strictly defined paths for electrons to travel through on the chip. Without these controls, the chip would, more often than not, do nothing; similar to randomly connecting cables between your TV and DVD player (a common strategy no doubt) which, more often than not, fails to produce a picture on the screen. Otherwise a chip's behavior cannot be predicted and thus it cannot be programmed.[5] Michael Conrad has argued, in a paper heavily drawn upon by Kittler, that this situation creates a necessary trade-off between *connectivity* and *programability*. "The amount of information processing carried out by a physical system freed from the constraints necessary to support programmability is thus potentially much greater than the potential information processing performed by a system not so constrained." [1] Taking this to its extreme conclusion, Kittler argues that only by removing the restrictions necessitated by programability is it possible to "enter into that body of real numbers originally known as chaos." [4]

Kittler radicalizes Conrad's argument and describes non-programmable machines as "badly needed" in that they "work essentially on a material substrate whose connectivity would allow for cellular reconfigurations" and so, "Software in the usual sense of an ever-feasible abstraction would not exist any longer." Kittler's brash materialism again shines through. Only when procedures are moved in a non-symbolic way to the real of the material, when the matter of the chip always and only executes the same operation as a matter of material necessity, have we finally created an authentic computing machine.

---

[5] More accurately this type of chip *may* be programmable, but to do so would require techniques specific to each individual chip.

Despite being non-programmable, the machines described by Conrad are still usable for human tasks. However, these machines would rely on evolutionary techniques to find solutions to a problem. The programmer then becomes a breeder, combining elements from the best individuals to create a new generation, designing environmental fitness conditions and running genetic operations in an iterative process of searching the terrain of possible solutions. Adrian Thompson created just this type of system by working with field-programmable gate arrays (FPGA)—a type of integrated circuit that is physically reconfigurable. He developed an evolutionary system to evolve a configuration of an FPGA chip capable of discriminating between two audible tones. Eventually a successful configuration emerged. But unlike a solution expected from a programmable system, the evolved configuration took advantage of unique material properties of the chip on which it evolved, using quantum tunneling and exploiting irregularities in the physical material of the chip. As Thompson puts it, "a robust asynchronous design was found that could not have resulted from normal design principles." [5]

An evolutionary system using non-programmable chips does have far fewer layers of abstraction and obfuscation between the programmer and the machine. But if, unlike Kittler, we resist the temptation to confuse matter with medium, then we are not compelled to interpret this as a necessarily more authentic engagement with computation. It is simply another way for humans to think through and use computational systems. Thompson's work, though unlike other acts of programming, nevertheless involved the articulation of a process–or perhaps a meta-process–in the form of an evolutionary system working to create a FPGA configuration capable of a specific task. In other words, the medium of the programmer is process in the form of code, not always (or only) the hardware on which her software is executed.

7. DOING / BEING

It is nearly impossible to talk about coding without talking about what the coder is trying to accomplish. The discussion is always already shot through with mentions of goals, tasks, problems, intentions, and action. Even the evolutionary techniques as applied to so-called non-programmable hardware require a specific formulation of *human* goals. We are still asking the machine to perform an operation, just in a different way and using a different vocabulary. Even in the most software-free variation, the trail of the human serpent runs over the voltages in the machine. In software there are always many moments of doing. The back-and-forth between machine and coder as software is written, the compiling of that source code into opcodes for the machine, the effect of running the code on the machine. Eventually all running software must rub up against the needs and goals of the user, though these needs may have been prefigured in advance by the assumptions of the coder. Of course many times this loop is closed as the programmer herself becomes a user of her own creations. To identify one moment in this chain as the essential moment—in Kittler's case, when the opcodes finally control voltages—is to attempt to replace *doing* with *being*. And to do so in the most brutally materialist way; if software does not exist, there is nothing to be said about the effect software has on politics or thought, either the thought of the coder or of the end-user. A virus that destroys a nation's economy, a protein folding simulation that finds the cure for a disease, and a copy of Minesweeper all do the same thing; they create voltages in silicon chips. Had they not been written in code, we could not talk about them in terms of politics, social change, or ethical import. They would simply be or not.

8. REFERENCES

[1] KITTLER, F. 1997. There is No Software. in *Literature, Media, Information Systems*, J. JOHNSTON, Ed. Routledge, New York, NY, 147-155.

[2] CRAMER, F. 2008. Language. in *Software Studies.* M. FULLER, Ed. MIT Press. Cambridge, MA, 168-174.

[3] FOUCAULT, M. 2008. 1980. Language to Infinity. in *Language, Counter-Memory, Practice*. D. BOUCHARD Ed. Cornell University Press, Ithaca, NY, 53-67.

[4] WARDRIP-FRUIN, N. 2009 *Expressive Processing*. MIT Press, Cambridge, MA.

[5] CONRAD, M. 1995. The Price of Programability. in *The Universal Turing Machine.* R. HERKEN, Ed. Springer-Verlag, New York, NY, 261-282

[6] THOMPSON, A. 2002. Notes on Design Through Artificial Evolution: Opportunities and Algorithms. ACDM 2002.