The Microcode Primer
A guide for non-coders towards a conceptual appreciation of code

based on codes at http://pallit.lhi.is/microcodes

Over the years, we've seen how the “private” languages of certain fields of practice within the arts have slowly permeated the more public language used to express and discuss various facets of these fields. For some odd reason it appears as though the practice of writing software code has been exempted from this. Perhaps because artists don't always make their code visible or because artwork based on software code hasn't been around long enough and gained enough attention.

My recent series of “Microcodes” are intended to be critically examined at the code level as well as at the level of the running process. The code informs the conceptual ideas behind each piece while the running process lends it a more “poetic air”. Therefore, I am providing this short primer that is intended to give the viewer some insight into the meaning of written software code so that they can extract valuable information from it that will help them to better understand what each of the “Microcodes” is about.

This is not by any means intended as a tutorial on how to write software code, only as a guide on how to read software code in a meaningful way.

Let's begin by examining the shortest of the “Microcodes”. “Sleep” is of course a direct reference to Andy Warhol's movie of the same title. Andy had intended to make the movie eight hours long to reflect the average time that people sleep at night.

```
#!/usr/bin/perl
sleep((8*60)*60);
```

The first line of this code is the same in all of the codes. It simply tells the computer what programming language this is written in. In the case of the “Microcodes”, the language is Perl. The second line tells the program to sleep by using the built-in function “sleep”. Most languages use functions the same way. The name of the function is written first and then “arguments” are supplied in parentheses after it. When you see something like this a google search for “Perl sleep” will give you many webpages full of information about the sleep function. Different functions accept different arguments. The sleep function accepts a number of seconds to sleep. In this case I could have simply written 28800 (the total number of seconds in eight hours) but I wanted to emphasize the fact that it is eight hours, therefore 8(hours)*60(minutes) gives us the number of minutes in eight hours and that number *60(seconds) again gives us the total number of seconds in eight hours. Mathematical operations and other functions can be used as arguments for a function.

The important thing here is that the computer needs to keep track of the time during a “sleep” operation to know when to stop sleeping. So while the program sleeps, it's not doing just nothing. It is “actively” sleeping just as Andy's movie showed that when a person sleeps, he's not doing “nothing”. He is in the active process of sleeping.

Now let's take a look at a slightly more complex “Microcode”. “Active monochrome” is the low-tech, digital approach to monochromatic painting. Whether we interpret monochrome painting as stillness, pause or violent upheaval against the multi-faceted norm in painting, the coded version provides the same sort of experience.

```
#!/usr/bin/perl
```

```
print "\e[?25l";
system 'clear';
while(1){
	$SIG{'INT'}=sub{print "\e[?25h";exit;};
}
```

All of the “Microcodes” do what they do in a terminal window. This is a very basic computing environment. Instead of using a graphical user interface, everything is performed textually. Programs are run by typing their name and parameters are set by typing commands. A terminal window can always be cleared by running the “clear” program (typing “man clear” in a terminal window will provide you with information about the “clear” program). In this case this is done by using the “system” function to tell the computer to run a seperate program. But even then a blinking cursor will be displayed. The way to get rid of this blinking cursor is to issue what's called a “terminal escape code” which consists of a cryptic sequence containing “\e[?” and some characters. Such sequences can be used to manipulate the color of the text or the background, change the terminal window's properties (i.e. Size or position) or several other properties. This program begins by printing such an escape sequence to the terminal to hide the cursor. Next it uses the “system” function to run the “clear” program that clears the terminal of any text. Following these actions it enters an “infinite loop” or a loop that has no defined end. “while” is what's known as a “conditional”. It means that whatever is inside the curly brackets that follow it is to be performed if certain conditions are found to be true. For instance, if we said, “while(1 == 0)”, whatever follows in curly brackets will never be executed because one can never equal zero. It's also important to understand, in this context, that the numerical value zero means “false” to a computer program. Likewise, any number that is not zero means true, even if it's a negative number. Therefore “while(1)” is a condition that will always be true and the program will repeat the process within the curly brackets over and over until the program is stopped by force. If this program stops running, it will show the normal terminal prompt immediately and therefore no longer display a monochrome. It has to work hard at maintaining the monochrome effect which in many ways is a rather violent disruption to the normal interface of the terminal window. It removes the terminal's functionality and forces the viewer to acknowledge it on a strictly aesthetic level. The code that is inside the while loop's curly brackets only gets performed if something halts the program (typing CTRL-c will stop it). Before exiting it will return the hidden cursor to its normal state.

The Perl programming language, like many others, is modular. It can be extended by using modules that provide functions that are not included in Perl's built in functions. Several of the “Microcodes” use a module called LWP::Simple which provides easy-to-use functions for retrieving information from the world wide web. Here is “Social spaces”:

```
#!/usr/bin/perl
use LWP::Simple;
$social_text = get('http://twitter.com/statuses/public_timeline.rss');
@social_space = $social_text =~ /(\s)/g;
foreach(@social_space){
	print $_;
}
```

This code contains several methods that are common in computer programming. First off is the extension of the programming language's capabilities. “use LWP::Simple” tells the program to read a file in the computer that contains code that defines a number of functions. Just seeing that this program uses the LWP::Simple module, tells us that this program interacts with the web. There's no other reason to use this particular module. Google-ing “Perl LWP::Simple” will help you find more information about the LWP::Simple module than you could ever need. Another thing that this program contains that hasn't been introduced before is variables. Variables in computer programming are like adjustable symbols that can stand for anything the programmer wants them

to. Most often variables are likened to buckets that can contain things. In Perl, variables always take the form $ and then the name of the variable. The name can be anything that the programmer chooses (within certain boundaries). I've chosen to use the name social_text so my variable is written $social_text. What it contains isn't immediately clear because it's not a fixed value or text. It's contents are set by using the “get” function which is one of the functions provided by LWP::Simple. As an argument, it takes the URL for a webpage. The URL I am using points to the syndication feed on twitter.com which contains the most recent entries by users of the site. So when I run the program, my variable ($social_text) will contain the text from that file. If you want to see what it looks like you can simply copy the URL and past it into your browser's address bar. So a variable's value can be set by some sort of an operation within the program (that's what makes them variable). For instance, if I have a line that looks like this:

```
$my_value = 1+1;
```

$my_value will not contain “1+1”. It will contain “2” because mathematical operations are performed automatically (almost always).

Next, we have a very different sort of variable that is prepended with an @ symbol instead of $. This is called an array. A variable can only hold a single value at a time. That value might be several pages of text but it's still only a single value. An array can hold several different values that we can call on seperately by providing so-called indexes. I could set the values of an array like this:

```
@my_array = ('Pall', 'Sylvia', 'Olive', 'Markus', 'Patricia');
```

The different elements are numbered from 0 up. So if I follow with:

```
print $my_array[2]
```

It will print 'Olive' to the screen. Note that when calling on a single element of an array, its name is written with a $ instead of @. This is exclusive to the Perl programming language.

The operation used to set the values for the @social_space array is somewhat complex and will take a little bit of explaining. The array's values are set to the outcome of the operation:

```
$social_text =~ /(\s)/g;
```

The combination of equal sign and tilda (~) means that we want to match elements within the text using what's known as “regular expressions” or “regex”. Regular expressions can be very complex and attempting a detailed explanation of them is way outside of the scope of this primer so I'll only explain what this particular regular expression does. In Perl the “conditions” of the regular expression are always between slashes (/ the regex goes here /). If the condition is located inside parentheses it means that you want to match that condition. The “g” that follows at the end means to keep matching throughout the whole text instead of stopping after the first match. “\s” is a symbol for “whitespace” which is any space not containing a character (i.e. Spaces, tabs or linebreaks). So what this is matching is any space within the text and whenever a match is found, it is added as a new element to our array. In other words, yes... our array is filled with empty spaces.

The “foreach” comand is a handy way of looping through all of the elements of an array when you don't know how many there are. “foreach(@social_space)” will execute the code contained inside the following curly brackets for each element contained in the array. Within the curly brackets you will see the variable “$_”. This is unique to Perl and is one of a collection of so-called “special variables”. Within a “foreach” loop, the $_ variable will contain the contents of the current array

element. So, for the first loop $_ will contain whatever $social_space[0] contains, the second loop will contain whatever $social_space[1] contains, etc. Until it reaches the end of the array.

So to sum up this rather complex explanation, what this program does is retrieve a collection of the most recent entries to the popular social website twitter.com, picks out all of the empty space within the text and prints it to the terminal window. Obviously, this questions our notions of “space” in the digital realm. It attempts to look at “online” space as if it were physical space occupied by letters where spaces between words represent empty space. In day to day conversation, we tend to use the word “space” to refer specifically to empty space (i.e. “I had to park across the street because there wasn't any space over here.”), therefore it's logical(?) to consider “social space” in the digital sense as those empty spaces that occur within texts that appear on a social website.

I hope that this primer can help “non-coders” to see and understand some of the possible artistic concepts that can be embedded in this sort of artwork. In the case of “Microcodes” the concepts have been purposely veiled in the output of the codes, forcing the viewer to consider both the code as well as its output. So as to not overwhelm the viewer, the works are produced in short, succinct code. These are not examples of proper coding procedures. Quite the contrary. The emphasis is on their conceptual meaning at the expense of their functionality.

Pall Thayer
http://this.is/pallit
Reykjavik, Iceland
21. May, 2009