Why MinNo?

I didn't have an easy time learning how to code when I first started. One of the largest reasons I had trouble was because of my first language: Python.

Though Python has many strengths, I would not say consistency is not among them. This is a pervasive problem from the language's core mechanics (string methods being in place or returning new strings for example) to how any two given people try to teach the same idea (take these two answers to similar questions for example: 1 & 2). One of Python's strengths is that it allows you to do one specific thing in ten different ways, but I am someone who likes to hear a single thing explained multiple ways. Though I was able to overcome my problems with Python, when I began toying around with the Arduino platform, I found the same problem with their home-grown language.

The Arduino language (AKA Processing, Wiring, C/C++) is a language built on top of C++ but only implements a subset of the C standard library (but none of C++'s library, just using it's syntax and constructs). There are also a fair amount of Arduino/AVR specific functions and global variables sprinkled about. Because of this mash up that is supposed to be a beginner platform and a collection of C functions resembling a K&R era implementation of the language, you get coders running the gamut in style and techniques. On one end you have some hobbyists making really high level sketches with sophisticated libraries to run wifi-shields for their Arduino Uno project in just a few lines of code. On the other side you have some old school hackers writing some of the most PreProcessor-Macro'ed, bit twiddling, interrupt throwing files you could imagine. If you aren't brought up to speed on C and Arduino by some guide that was provided by education companies like Sparkfun, you can find yourself in a hexadecimal-encoded hell within a matter of clicks. MinNo is an attempt to create a more humane (for people like myself) entry point to the

Arduino platform.

Harvard vs. Von Neumann

Most interest in processor architecture lies in standardized implementations (x86 and ARM). The conversation at this level typically is a discussion of instruction sets and bus implementation. AVR processors take it a step further back and discuss the basic organization of a computer. X86 and ARM architectures are organized in a manner originally specified by John Von Neumann: a central processing unit, memory, some form of input, some form of output, and long term storage across a set of shared buses. AVR processors (generally) are organized in a manner consistent with Harvard specification (more accurately, the modified-Harvard spec). The main difference between Von Neumann and Harvard is that Harvard separates memory into two different pools: instruction memory and data memory, which attach to the CPU via entirely separate buses. Harvard processors can simultaneously read and write to both memory pools because they do not share a bus. The two pools don't even need to be indexed and architectured the same. Often (as is the case with AVR processors), instruction memory is read/write accessible at runtime, data memory isn't. Data memory is often used for long-term, initialization data: tables, addresses, arrays, keys, and so on. Both the data and instruction memory pools are implemented as EPROM/Flash storage . AVR processors have general SRAM for data that needs to be accessed more quickly (in place of caches).

One of the problems with using C to program Harvard processors is that most C compiler implementations (and subsequently Arduino's language) have grown under the ideas of Von Neumann architecture. All memory accessors and management systems, property files, and object oriented idioms assume the computer they are running on has one shared pool of read/write memory. To allow the benefits of Harvard's memory architecture, we have to pollute the global name space with keywords to specifically access the data memory pool (such as declaration containing

"const PROGMEM" is Arduino sketches).

MinNo is an attempt to embedded Harvard's distinction in architecture at the syntactic level. By default, all declarations of variables are immutable unless they contain the "mutable" tag. If the compiler sees an immutable declaration, it will be put in "program memory" (Arduino's name for data memory). This allows for a more concise memory placement syntax to counteract MinNo's slightly more verbose declaration structure. This thought of embedding Harvards design choices in the syntax is a consideration throughout the languages idioms and structures.

Static Memory Allocation

Culturally, among novice programers, pointers and manual memory management stand as the mysterious and often frightening figure when writing software. It is easy to misuse them and hard to figure out why they aren't working (if you aren't practiced in debugging them). Their inclusion in the Arduino language stands as an oddity to me for that reason, other than performance restrictions that prohibit a garbage collection system. Why would a platform bragging about beginner friendliness use one of the most notoriously confusing memory management systems? Given that garbage collection is out of the question due to the lower power CPUs found on Arduinos and that MinNo is supposed to be an alternative beginner language, I have decided (at least for now) to have no dynamically growing memory in MinNo: everything is statically allocated.

This may initially seem like a huge drawback, and in general programming I would agree. But in embedded systems, dynamic memory bugs can be subtle, incredibly hard to reproduce, and absolutely system crashing . Early in my time programming sketches, I swore off the use of malloc and free in favor of static memory declarations. From synthesizers, to controllers, to IR navigating robots, I have gotten away without using dynamic heap allocations. To anyone reading this, for your sanity, I would suggest a similar shift in coding style. Others also suggest minimizing dynamic allocation where you can.

With purely static allocation, it also allows for far better utilization of the program memory pool. An added bonus in such a memory restricted system.

Template Operations vs. Stateful Procedures

MinNo is meant to emphasize a more template based form of thinking about code; the code represents templates operating on data. A hard separation of data and code should be taken where it can. Intermediary values and indexing values are, of course, held in read/write instruction memory, but these values can be thought of as markers and states in templates. Indexes are markers as to what templates are being applied to. Intermediate values are states in between templates operations. This mind set keep the separation of memory pools more consistently present and should lead one to minimize stateful code as to avoid unnecessary complexity.

Getting Under The Hood

Now there are absolutely times when this distinction of memory can be far more of a hinderance than a benefit. Sometimes we really need manual heap allocation. MinNo's compiler attempts to generate (subjectively) readable, tabbed C code. This will allow boiler plate functions and code to be generated, but the resulting code can be opened and optimized/altered to allow for malloc's, free's and a things C-like. It also allows for a bridge from those who have been using MinNo to transition into C in a low-stakes, take-it-at-your-pace manner.