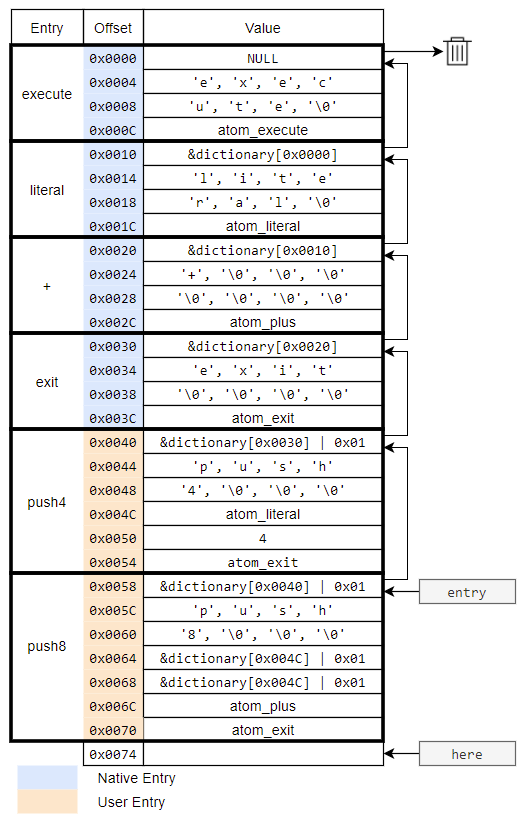
Pino - A Threaded Interpretive Language in C

Todd Blackmon, June 2021

A threaded interpretive language is executed by walking through data structure called the dictionary. Each function to be “called” is referred to as a *word*. This structure is interesting because once the basic execution structure defined, the dictionary can be extended to use the existing words in new combinations.

Essentially adding *data* to the dictionary is adding executable *code* to the system.

A sufficiently rich set of starting words provides a system that can eventually even extend itself. It all starts with a beginning dictionary. An example dictionary is used throughout to illustrate the concepts used.

## Dictionary

The dictionary is arranged as a linked list of dictionary entries. The dictionary is traversed from the last entry to the first. The first 32-bit word of each entry is the *link* field, which contains the address to the next entry in the dictionary. The dictionary is aligned to an 8-byte address, meaning the lowest 3 bits are unneeded for the address.

|  |  |
| --- | --- |
| Bit 0 | 0 = Native entry  1 = User entry |
| Bit 1 | 0 = Valid entry  1 = Unused entry |
| Bit 2 | 0 = Normal execution  1 = Immediate execution |

These bits are used to indicate special entries.

NOTE: Bits 1 and 2 are not used in this example. They are set to 0 in this case.

**Entry Types**

As noted above, the link field bit 0 indicates whether the entry type is a native or user entry. The native entries indicate words that are implemented in C. The user entries are fully implemented by executing either native words or user words. User entries are fully defined in the dictionary itself.

Both types of entries follow the link field with an 8-byte name field used to match the entry when searching. This defines the name of the word that the user would type to execute the word. The name field is then followed by the body of the word entry.

For native entries, the body of the entry is simply a single 32-bit address of the native C function that implements the word. User entries instead have a list of 32-bit values. The list always ends with the atom\_exit function address. The list values are interpreted in 3 different ways:

1. Native function addresses – These are executed in order as a list of function calls. In the examples, all C native function addresses begin with “atom\_”.
2. User word addresses – These are indicated by having bit 0 set. The address (after masking bit 0) points to the first address in the body of a user entry. This user entry will be executed in full before returning to the current list, with the current state saved on a return stack similar to calling a function in C.
3. Data – Some values are only intended to store data. These values are generally consumed by previous words and skipped so as to not be interpreted as an address to be executed.

## Example Dictionary Walkthrough

Assume the user has executed "push8". The dictionary is searched and the first entry is found to match. The executor will then execute starting at dictionary[0x0060]. The steps are shown below with the next instruction pointer (IP), data stack, and return stack at the start of each step.

**Address:** dictionary[0x0060] **Value:** &dictionary[0x004C] | 0x01 **IP:** dictionary[0x0064]  
**Data Stack:** ( ) **Return Stack**: (\*)

Since bit 0 is set, it is a user word. It is the address of the "push4" entry body. The next address which would have been executed is &dictionary[0x0064]. This is pushed onto the return stack. (This is the same as the “call stack” in C). Then the instruction at 0x004C next to be executed.

**Address:** dictionary[0x004C] **Value:** atom\_literal **IP:** dictionary [0x0050]

**Data Stack:** ( ) **Return Stack:** (\* dictionary[0x0064])

This is a native function address, so it is simply called. The atom\_literal function will:

1. Push the literal value in the next instruction address onto the data stack.
2. Cause that next address to be skipped.

In this case, the next location contains the value 4. Which is then pushed onto the stack.

**Address:** dictionary[0x0054] **Value:** atom\_exit **IP:** dictionary [0x0058]

**Data Stack:** (4) **Return Stack:** (\* dictionary[0x0064])

The atom\_exit function *exits* the user word and pops the value from the return stack.

**Address:** dictionary[0x0064] **Value:** &dictionary[0x004C] | 0x01 **IP:** dictionary [0x0068]

**Data Stack:** (4) **Return Stack:** (\*)

This is another call to “push4”.

**Address:** dictionary[0x004C] **Value:** atom\_literal **IP:** dictionary [0x0050]

**Data Stack:** (4) **Return Stack:** (\* dictionary[0x0068])

Push another 4 onto the stack and move the IP an extra 4 ahead.

**Address:** dictionary[0x0054] **Value:** atom\_exit **IP:** dictionary [0x0058]

**Data Stack:** (4 4) **Return Stack:** (\* dictionary[0x0068])

The atom\_exit function *exits* the user word and pops the value from the return stack.

**Address:** dictionary[0x0068] **Value:** atom\_plus **IP:** dictionary [0x006C]

**Data Stack:** (4 4) **Return Stack:** (\*)

The atom\_plus function replaces the top two items of the data stack with their sum.

**Address:** dictionary[0x006C] **Value:** atom\_exit **IP:** dictionary [0x0070]

**Data Stack:** (8) **Return Stack:** (\*)

This returns back to the executor (indicated here by the \* on the return stack).

As promised, executing the “push8” instruction pushes an 8 onto the data stack. The execution had two levels of user words and multiple native words. Some details about how the native words work is explained next.

## Native words and the *next* function

All native words return the pointer to the next function to be executed. The core of the executor is simply calling the return values until NULL is returned. (The return stack has an initial NULL pushed onto it before starting. When this NULL is popped, the loop will break.)

while(fn\_ptr) {

fn\_ptr = fn\_ptr();

}

To get this address, all native C functions have the same form.

wordfp native\_fn (void)

{

/\* body of the function \*/

return next();

}

The next function handles the native vs. user word parsing of the next instruction. The algorithm is:

1. tmp = IP
2. IP = IP+1
3. if bit 0 of tmp is set, the next word is a user word
   1. Push IP onto the return stack
   2. Mask bit 0 from tmp
   3. IP = tmp
   4. Start over at step 1
4. if bit 0 of tmp is not set, the next word is a native word
   1. return tmp

The other important core function is atom\_exit. This is the function that undoes the return stack push done in step 3 of next. The algorithm of atom\_exit is:

1. Pop the return stack into IP.
2. If IP does not equal NULL
   1. return next()
3. Otherwise return NULL

Following the Example dictionary walkthrough in detail shows how this works.

**NOTE:** The executor returning the pointer then calling is done to avoid recursion in C. In an assembly language implementation, the software would instead just *jump* to the start of the next function.

## How to run a word from C

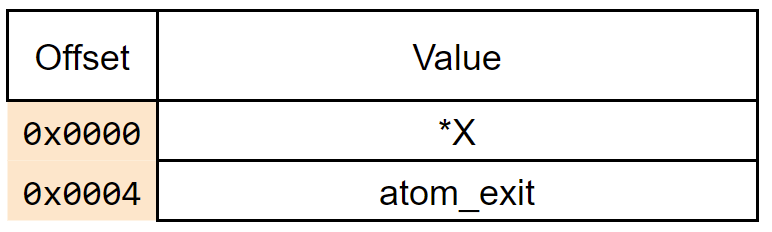
Calling a user word from C requires some care. To call a user word with its data list at the address X:

push\_return\_stack(IP);

IP = X;

return next();

This simulates the call of next that is effectively done in step 3d of the next algorithm above. The atom\_exit of the called word will return to the next word to be executed after the C function that called it.

Calling a native word creates an issue since atom\_exit should be called right after the native word. To handle this, a temporary user word called *springboard* is used. It has only two 32-bit addresses and is used as shown below.

springboard[0] = \*X;

push\_return\_stack(IP);

IP = springboard;

return next();

The algorithm as described above can be used to create a word called execute. This word is used in a user interface sequence that allows the user to type in sequences of words that can be parsed and executed.

It is important to note that while it is possible for execute to be called recursively, using a single springboard object is safe. The reason is that the use of the first address of the springboard is always completed before the word could be overwritten by the next native C function.

## User Interface Design

A live interpreted system interface is a useful feature. For ease of implementation, the user interface will be defined in C. It is worth noting that once the control structures are in place, the user interface can actually be largely defined in terms of smaller native words.

**NOTE:** This could be done later. It is very cool, but I’m not sure it provides much educational value.

The basic user interface algorithm is as follows:

loop forever:

read input from user until newline

while not end of line:

lex next token

if token found:

search for token in dictionary

if found

execute the word

else

attempt to parse the word as a number

if it can be parsed as a number

place number on data stack

else

print “Error: word not found”

break

else:

print “ ok\n>”

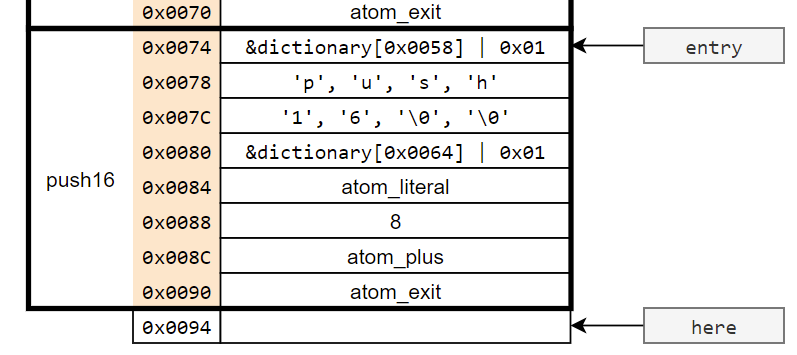
This algorithm provides basic “calculator” functionality, but is not able to support compilation of new user words. This will be added next.

## Compilation

Creating new user words is possible using a compilation feature. First the new header must be added. This is done using a new native word, def. The user would give the definition of a new word as:

def push16 push8 8 + ;

This will need to create a new word on the stack which looks like this.



### def

The first word is def, which performs three steps. First it reads ahead in the input to find the name of the new word, in this case “push16”. This is then used to create the header.

\*here = entry | 0x01 | 0x02

entry = here

here = here + 4

copy name into name field at here

here = here + 8

This sequence adds the header of a user word and moves the entry pointer to include it in the search. It also includes the first use of bit 1 in the link field. Dictionary searches skip entries with this bit set. This allows the entry pointer to be moved forward without affecting any word searches before it is ready.

The final step of def is to set a global flag called compile\_mode. This flag is used to direct the interpreter of how to execute words. By default, this flag is false, so the system is in interpreter mode. When changed to compile mode, some changes are made to the user interface algorithm as shown below.

The difference is that in compile mode, the address of the word is added to the end of the dictionary instead of being executed.

loop forever:

read input from user until newline

while not end of line:

lex next token

if token found:

search for token in dictionary

if found

if (not compile mode or immediate word)

execute the word

else:

compile the word to \*here

here = here + 4

else:

attempt to parse the word as a number

if it can be parsed as a number:

if (compile\_mode):

place address of atom\_literal to \*here

place parsed number to \*(here + 4)

here = here + 8

else:

place number on data stack

else:

print “Error: word not found”

break

else:

print “ ok\n>”

Walking through the previous definition, the remaining tokens are “push8”, “8”, “+”, and “;”. Using the previous dictionary example:

* push8 is located at 0x0058. The link field indicates that it is a non-immediate user word. This results in the location of the first entry in the dictionary, 0x0058+12 = 0x0064. Because it is a user word, bit 0 is then set.
* 8 is not located in the dictionary, so is parsed as a number. As seen in previous definitions, atom\_literal provides the needed functionality. The interpreter places the address of atom\_literal in the dictionary followed by the actual value 8.
* + is found at 0x0020. Since it is a native word, the *value* at 0x0020 + 12 = 0x0032 is added to the dictionary, which is atom\_plus.

### ;

The final word “;” completes the definition. However, to do this, it must execute, not just compile. The definition of “;” is the first example of an *immediate* word, which is indicated by setting bit 2 in the link field. The interpreter will always execute immediate words regardless of the state of compile\_mode.

The steps “;” follows are:

1. Add atom\_exit to the end of the dictionary. This completes the definition.
2. Clear bit 1 of the link field of the new definition to make it visible in searches. The link field is conveniently the address pointed at by here.
3. Set compile\_mode to false.

The system is now back in interpreter mode, but with a new available word. This is a big step. The next step is to add the ability to make decisions with if/else/then words.

## Decision Branching

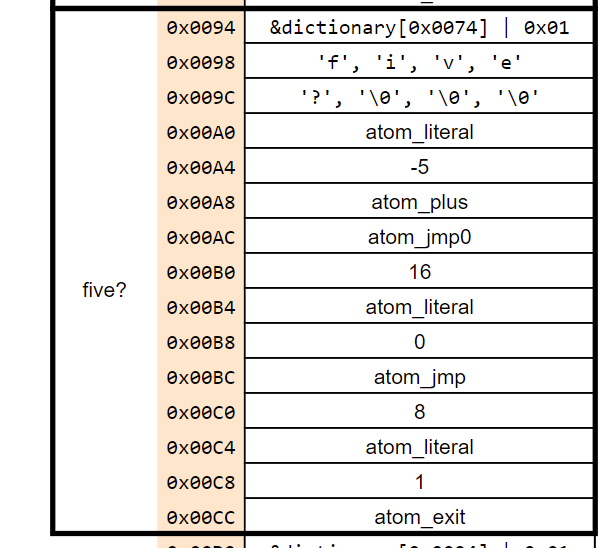
The current words only provide basic calculator-type capabilities as well as the ability to group preexisting words into new user words. Decision making allows much more computing power. It may be surprising that this can be added with no structure changes. It only requires a small set of new native words.

A new example user word will be used to see what an entry with decision branching looks like. The word five? will pop the top value off of the data stack, push a 1 if it was a 5, and a 0 if it was not.

def five? -5 + if 0 else 1 then ;

First the top of the stack is added to -5. The value will only be 0 if the given number was a 5. The new if word pops the top of the stack as a *truth value*. For a truth value, 0 is *false* and all other values are *true*. If the value is true, then the 0 will be pushed onto the stack. If false, the 1 is pushed on the stack.

The actual dictionary entry looks a bit different:



The beginning is the same, but if and else are replaced by jmp0 and jmp. The then word is not even in the definition! A walkthrough of the entry helps to understand.

1. atom\_literal consumes the next entry, -5, and puts this value on the data stack.
2. atom\_plus replaces the top two items on the stack with their sum.
3. atom\_jmp0, like atom\_literal consumes the next entry, 16. This is the number of bytes to skip. atom\_jmp0 pops the top item from the data stack and only jumps if the value is 0.

Here two paths may be taken

#### If the value read by atom\_jmp0 was 0

1. The jump was taken so the next 16 bytes are skipped to 0x00C4. This atom\_literal pushes 1 onto the stack.
2. atom\_exit ends the user word definition.

#### If the value read by atom\_jmp0 was not 0

1. The jump was not taken, so the next word is at 0x00B4. This atom\_literal pushes 0 onto the stack.
2. atom\_jmp is similar to atom\_jmp0 except it does not modify the stack and always jumps. In this case it will jump 8 bytes to 0x00CC.
3. atom\_exit ends the user word definition.

## Compiling an if/else/then word

Compiling a word definition from if/else/then to jmp0/jmp entries takes a bit of thought. All three words are immediate words, so they are executed even when in compile mode. In fact, these words are *only* used in compile mode.

The first word is if, executing as atom\_if, which has the following algorithm:

1. Compile atom\_jmp0 to \*here.
2. here = here + 4
3. Push the value of here onto the data stack
4. here = here + 4

The distance to jump is not yet known, so the address of the dictionary entry to be filled in placed on the data stack for later handling by else. Here is the atom\_else implementation:

1. Compile atom\_jmp to \*here.
2. here = here + 4
3. Pop the address from the data stack into a temporary variable tmp.
4. \*tmp = here – tmp
5. Push the value of here onto the data stack.
6. here = here + 4

Step 4 has a lot of complexity. This instruction calculates the amount of jump needed by subtracting the location of the next instruction, stored in here, from the jmp0 offset location which is then written to that location. Note also that the else instruction puts its location to be filled in on the data stack.

The then instruction is simpler and finishes the process:

1. Pop the address from the data stack into a temporary variable tmp.
2. \*tmp = here – tmp - 4

Note that then does not actually compile a new word to the dictionary. It simply fills the address needed. The offset of 4 in the calculation is to account for this difference.

This structure also directly supports if/then structures without an else part. Another way to write five? is the following:

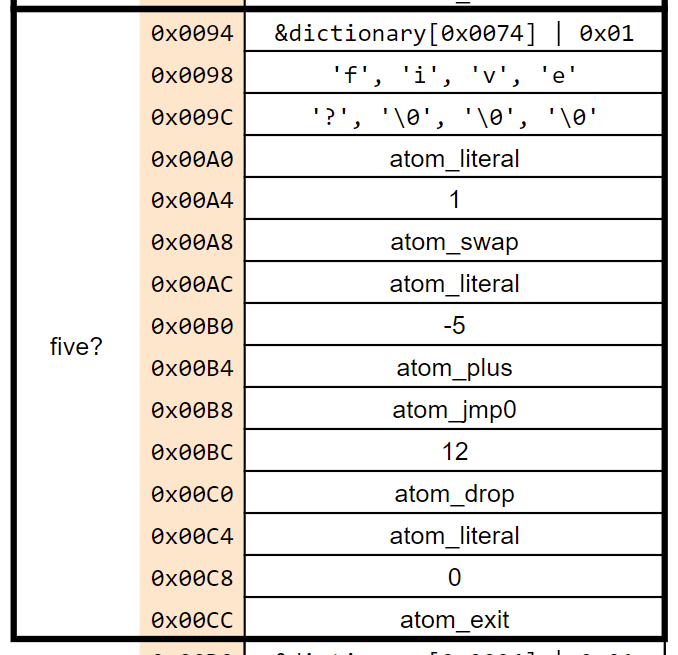
def five? 1 swap -5 + if drop 0 then;

Two new native words are used:

swap – Swaps the top two members of the data stack

drop – Drops the top member of the data stack

Here a 1 is preplaced on the stack to assume that the number is a 5, then if it is proven not to be, it is dropped and replaced with a 0. The dictionary for this word would be as follows:



Because of the similarities in the way that else is encoded, the implementation of then also places the correct value for the if statement here.

Therefore, decision making is provided with the addition of three immediate words: if, else, then and two non-immediate words jmp and jmp0.

NOTE: It is possible (even if not practical) to implement jmp in terms of jmp0. Consider the following:

def jmp 0 jmp0 ;

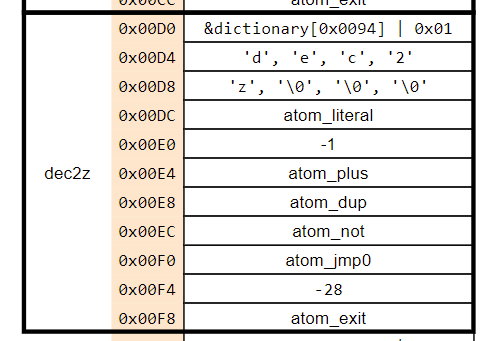
## Loops

The last major control structure type missing is a method to perform looping. While multiple types of loop are possible, here the simple begin/until loop is considered. The example here is a user word dec2z which decrements a (positive) number until it reaches 0:

def dec2z begin -1 + dup not until ;

The loop is between begin and until. until expects a truth value (which it consumes) and will jump back to begin until that value is *true*. Zero is false, so a logic inversion is provided with the not word. The value of 0 is always left on the data stack at the end.

The dictionary entry for this user word is



The loop is implemented by jumping by a negative offset using the same atom\_jmp0 function as before. And again, as before, the dictionary entry is built using immediate words.

begin has a simple implementation. It only needs to mark the beginning of the loop by pushing the current value of here onto the data stack. It does not compile any words.

until uses the stack value to build a jmp0 entry with a negative offset. The algorithm is:

1. Compile atom\_jmp0 to \*here.
2. here = here + 4
3. Pop the address from the data stack into a temporary variable tmp.
4. \*here = tmp - here - 4
5. here = here + 4

## User-defined control structures

Most programming languages have multiple control structures. Pino can also have those added by writing more native words. But native words mean writing new C code. It would be better to allow the user to write *user* control words in Pino itself.

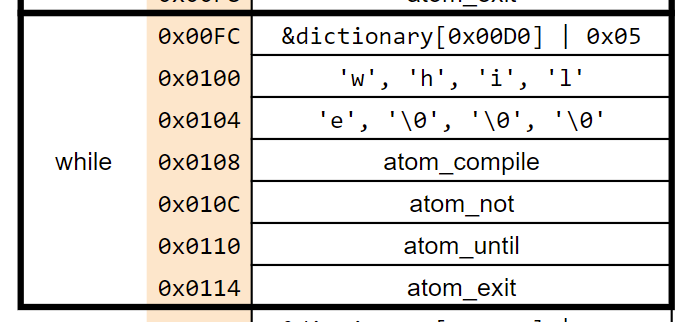
An example of this is simply implementing a while word that loops when the truth value is true so the user can avoid having to add an extra not before each loop that they write. Other more complex structures such as switch/case-type statements should also be possible.

One missing piece is the ability to make immediate *user* words. All immediate words have so far been native words. This is easily solved by providing the immediate word which makes the last definition in the dictionary an immediate word. This is simply done by setting bit 1 of the address pointed to by entry.

But this is not enough to implement the while word since there is no way to compile from within an executing user word. Two new words are defined: [compile]and postpone. Their use is best shown by an example. The while word that we want is defined as:

def while [compile] not postpone until ; immediate

To help explain how this works, the dictionary entry for while is shown below.



Using the while word in the definition for dec2z is easy and it creates the same dictionary entry for dec2z as shown earlier.

def dec2z begin -1 + dup while ;

The way this works is that when the definition for dec2z is being compiled, the while word is encountered. Since it is an immediate word (as indicated by the flag in the link field), the interpreter executes it instead of compiling a call directly in. In the dictionary entry above, the first word executed is atom\_compile.

atom\_compile is similar to atom\_literal where it consumes the next entry, but instead of placing the value at that entry onto the data stack, it instead compiles it into the current definition. In this case, atom\_not is added to the working definition right after the dup that was just defined.

Since atom\_compile uses the atom\_not location, the next word is atom\_until. This works exactly as before by compiling atom\_jmp0 and the jump offset from the top of the data stack.

The new while word works as expected, but the operation of postpone may be still be unclear. postpone is an immediate word that makes the next word compile into the current definition regardless of whether it is an immediate word. Effectively, postpone is used to delay an immediate word from executing. Such as this example where the execution of until is postponed until while is executed, not when it is defined.

This is a very nice result as it makes Pino able to extend itself without resorting to writing more C code.

## Possible next steps

The language implementation shown here is not especially good as much more could be done in non-native words built on simpler native primitives. The value of this is that having these lower-level primitives visible would add more expressiveness to Pino for the end user. But the implementation was more for self-learning than actual use.

Anyone familiar with Forth will realize that Pino is really a Forth implementation with different names for some of the words. As such, the Forth literature provides some good ideas of what kinds of extra native words could be added to make a more full-featured language. Note that absolutely *no* effort was expended to match the word definitions here to standard Forth words. Some translation will definitely be required.

An important idea that is only touched upon is compiler extension. For a better description of this, consider Chapter 11 of Starting Forth, and the description of creation words and create/does>. This may be added in the future as time permits.

## Notes

For embedded targets, note that the dictionary for native words does not need to be located on the actual target. Also note that the structure used here can be executed on Harvard architectures as all native executable code is separate from the dictionary. This was a goal of the architecture and led to some slightly non-standard dictionary entry structures.