The Hardcode Programming Language

The Hardcode language is an attempt to define a programming language grammar and semantics that improves upon the C language. The language is a hybrid and incorporates features from several other languages, including PL/I and C.

The language is based first and foremost on a number of goals, these are

1. No reserved words (new keywords can be added without breaking backward compatibility)
2. No forward declarations needed "in file"
3. Use braces for block delimiters
4. Support pointers
5. Support bit data types
6. Support structures
7. Support string data types
8. No support or affinity with OO, inheritance, interfaces and so on
9. Supports namespaces
10. Label variables, computed goto
11. Bit shifts, rotates etc.
12. Support constant execution time capability
13. Support method overloading
14. Support alignment, padding and packing attributes
15. Support some kind of "endian" specifier
16. Support nested methods
17. Support a "nop" feature to literally embed "nop" instructions -> nop(12); // insert 12 literal nops
18. Distinguish functions and procedures
19. Cannot invoke a function other than as an expression
20. Cannot invoke a procedure other than with "call"
21. Support decimal fixed point data type
22. Support BCD data type (if not the same as decimal)
23. Support any array dimensionality
24. Support dynamic array declarations
25. Support sparse arrays
26. Support ability to add new symbolic operators without breaking backward compatibility
27. Consider support for yielding in addition to returning
28. Consider support for coroutines
29. Consider language keywords/constructs for common hardware mechanisms like ADC, timers etc.
30. Consider synchronization and fence abstractions
31. Consider a "portable" keyword which validates function source code for any possible non-portable content.
32. Support a "write to set" concept, a way to represent that say as an attribute.
33. Possibly expose "bit bands" linguistically, if supported in some target.
34. Support interop with other languages, initially C but not confined to C

# Extensibility

The goals of extensibility serve to allow the language to grow and never break backward compatibility. This is achieved by never having reserved words, the language has keywords but these are not reserved and can be used for identifiers in user’s code.

In addition we do not restrict the ordering of independent attributes that appear on definitions and declaration, unless there is some true dependence between terms.

A benefit of this idea – above and beyond the long term growth of the language, is that an early, minimally built version of a language can be used to develop initial code and that code can always be consumed by future versions of the language.

# Concepts

Hardcode is not an OO language, it has no support for ideas like interfaces or inheritance. It supports data structure definitions and instances. It supports encapsulation by allowing procedures and functions to be defined inside structures.

It supports singleton structures, that is instances of types that exist exactly once at runtime.

All entities defined inside a source files are hidden, unless exposed (see “exposed” attribute).

The language does not require multiple “passes”, so there is no need for “forward declaration” in order to refer to an item prior to its lexical appearance in a source file.

Every statement in hardcode is regarded as being either an executable operation or an assignment. The parser is able to recognize a statement as an assignment if it has legal syntax, if it does not meet the criteria for being an assignment then the first token is treated as a keyword.

## Assignments

An assignment is the only operation that can mutate storage, the grammar is

<assignment-stmt> := <expression> = <expression> ;

The LHS <expression> is an expression that (should) represents an addressable location in memory and RHS <expression> is an expression that represents a datum that is compatible for being written to the target address.

Any statement that satisfies these grammatical rules is regarded as an assignment, otherwise the first token is treated as a language keyword and then parsed on that basis.

Examples

A = 1;

A(J) = 34;

A(B(X)) = A(B(0));

if = then;

else = 0;

goto = value;

are all assignments, they satisfy the grammar rules (and if they do not meet the sematic rules, error will be reported).

## Procedures

The language distinguishes between functions and procedures, the former return data whereas the latter do not. For this reason procedures can only be invoked via the call keyword:

call <procedure-reference>;

The procedure reference can have zero or more arguments and if there are no arguments then no parentheses are needed or permitted. A function must return a value and may take arguments. Function references can be used within expressions, but procedure references cannot, except in the case where the procedure reference is itself an *argument* to a function.

Both procedures and functions are defined with their respective keywords:

**procedure** initialize\_system

{

}

**function** calculate\_rate (x,y) bin(16)

{

**arg** x bin(8);

**arg** y bin(8);

**return** x \* y;

}

As you can see arguments are declared separately to the definition of the procedure/function and must be declare with the **arg** keyword. The **dcl** keyword is for declaring non argument data.

## Declarations

Declarations always begin with the **dcl** keyword and definitions type definitions begin with the **type** keyword, one can declare individual items, arrays and/or structures this way:

**dcl** name string(32);

dcl counters(32) bin(8);

type record

{

type header

{

type string(4);

date string(8);

}

volume bin(8);

}

## Argument Passing

All arguments are passed by reference, an address of some datum is pushed as part of the stack frame. To pass an argument by value (so that modifications to it in the called routine cannot modify the callers data) the argument can be passed inside parentheses:

dcl speed bin(16);

speed = 64;

call start\_motor(speed);

…

procedure start\_motor (s)

{

arg s bin(16);

// do stuff

s = 128;

}

After the call to start\_motor, the value of ‘speed’ will be 128; To insulate the calling block from the called block making changes we can invoke start\_motor this way:

start\_motor((speed));

In such cases the compiler creates a copy of the argument and passes a reference to that copy. This approach eliminates the need to explicitly obtain and pass pointers or addresses but the caller still retains the choice as to whether to pass the argument by address or value.

## Abbreviations

Many keywords can be abbreviated, for example procedure as **proc**, function as **func**, declare as **dec**, variable as **var**, substring as substr, address as addr.

## Built in Functions

Built in functions are functions that are considered part of the Hardcode language. They are always available unless one is using an older release or one has explicitly chosen to disable one of more of them. The code implementing the functions is generated at compile time, no additional linking is needed.

Built in functions are invoked using square [ brackets ], this is the only use of square brackets in the language and thus collisions with user defined functions of the same name is impossible, this – like the policy of no reserved words – facilitates language expansion with full backward compatibility.

Examples are

shortname = substring[fullname,5]; // or substr

addr\_ptr = addr[root\_block]; // or address

## Bits and stuff

The language should provide as much assistance for working with bit fields as is reasonably possible. The bit is just another type in the language. The following topics arise when considering this area:

* How to declare a datum comprising multiple bits.
* How to refer to elements or sub elements.
* How padding, alignment and packing are managed.

The bit type has much in common with the string type, in that we can treat them individually or as an aggregate.

Here are some declarations of bit data

dcl mask bit(10) packed;

dcl mask(10) bit(1) packed;

type control aligned,

{

mask bit(10),

bits(10) bit(1),

table(16,16) bit(8)

}

## Constant Time Features

We want the language to have some ability to created code that runs in constant time. This means functions or procedures or blocks whose execution time is invariant with respect to arguments.

This is facilitated by the following

### Invariant keyword’

Any function or procedure can have the “invariant” attribute keyword. That tells the compiler to avoid any optimizations that could lead to variations in execution time, like for example short circuiting in expressions.

An invariant procedure cannot call any other procedure that is not itself invariant.

The invariant keyword can also appear on a block:

invariant {

do\_stuff;

}

This is really a means of influencing optimization, not inhibiting it all together.

If the invariant block or procedure is interrupted then execution time is no longer deemed invariant.

### Allocate/Free features

Something to consider is a means of checking that the number of frees matches the number of allocates. That is we could have a function attribute that allows us to do something if these don’t match.

We could count allocations on a function by function basis too. So that we can (or the runtime can) at any point find out how many allocations and frees any method has done.

We could also expose this and other metrics to the developer like how much memory has been allocated, how much heap has been used up (which will include heap overheads) and so on.

Compile Time code execution

Look at Zig, it offers this capability.

# Formal Grammar

There is a very helpful online tool: <https://bnfplayground.pauliankline.com/> that can be used to develop and validate a grammar. Here is the draft grammar for Hardcode so far:

<DCL> ::= " dcl " | " declare "

<PROC> ::= " proc " | " procedure "

<FUNC> ::= " func " | " function "

<BIN> ::= " bin " | " binary "

<DEC> ::= " dec " | "decimal "

<STRING> ::= " string "

<STATIC> ::= " static "

<BASED> ::= " based "

<CALL> ::= " call "

<GOTO> ::= " goto "

<RETURN> ::= " return "

<IF> ::= " if "

<THEN> ::= " then "

<ELSE> ::= " else "

<compilation\_unit> ::= <statement>+

<statement> ::= <dcl\_statement> ";" | <def\_statement>

<dcl\_statement> ::= <DCL> <identifier> <array\_specifier>? <reqd\_dcl\_attribs> <optional\_dcl\_attribs>\*

<identifier> ::= [a-z]+ " "

<def\_statement> ::= ( <PROC> | <FUNC> ) <identifier> <arglist>? <reqd\_dcl\_attribs>? <block>

<arglist> ::= "(" <identifier> ( "," <identifier>)\* ")"

<array\_specifier> ::= "(" <array\_term> ( "," <array\_term>)\* ")"

<array\_term> ::= [0-9]+

<reqd\_dcl\_attribs> ::= <BIN> | <DEC> | <STRING>

<optional\_dcl\_attribs> ::= <STATIC> | <BASED>

<block> ::= "{" (<block\_statement> ";")\* "}"

<assignment> ::= <identifier> " = " <identifier> ";"

<block\_statement> ::= (<assignment> | <keyword\_stmt> | <dcl\_statement>)\*

<keyword\_stmt> ::= <call\_stmt> | <goto\_stmt> | <return\_stmt> | <if\_stmt>

<call\_stmt> ::= <CALL> <identifier> <arglist>?

<goto\_stmt> ::= <GOTO> <identifier>

<return\_stmt> ::= <RETURN>

<if\_stmt> ::= <IF> <expression> <THEN> <block> (<ELSE> <block> )\*

<expression> ::= <identifier>