

1 Introduction

We need to decide if the IEEE1647 standard will specify more reserved names in the E language than the original design. In other words decide **if more keywords become reserved names** (and eventually which).

This chapter presents the problem.

The second chapter evaluates alternatives.

Appendix A contains terminology.

Appendix B lists the keywords.

Appendix C contains code examples with highlighted keywords.

1.1 Details

A keyword is an identifier which has a specific meaning in a specific context. Some of the keywords in the E language are “if”, “struct”, “type” or “for”.

There are usually restrictions about reusing keywords as names for user-defined objects such as variables or methods, however some languages are extremely liberal in this approach allowing core keywords to be redefined for specific purposes. **Keywords reserved for core language usage are called reserved names.** The only reserved names in the original design of the E language are: “TRUE”, “FALSE”, “NULL”, “UNDEF”, “MAX_INT”, “MIN_INT” and “MAX_UINT”.

As a consequence, the original design of the E language allows constructs like:

- `var struct : int; // “struct” keyword used as a variable name`
- `for(int:i) is empty; // “for” keyword used as a method name`
- `extend : int; // “extend” keyword used as a field name`

The code developer is not restricted to use keywords for other purposes outside their context (of course, except the few reserved ones mentioned above).

Although they are identifiers with a specific meaning in a specific context, the following categories are **not** considered **keywords**:

- **automatic variables** like `result, index, it, per_instance`
- **predefined routines** like `dut_error, all_values`
- **predefined methods** like `pre_generate, run`
- **pseudo-methods** like `first_index, apply`

1.2 The Original Design

The original intent of the E language it to avoid restrictions wherever possible

and allow the code developer to choose his own names to identify various programming objects with minimal constraints.

One of the “legends” presents the language name as the first letter in the word **English** with the clear intent to allow writing down code in a natural way.

This can be easily seen throughout the language itself which uses keywords in more than one context to express different things:

- `in` to expresses belong: `keep in, for each in`
- `in` to express input: `in port`
- `is` to express define: `m() is, cover is`
- `is` to express query: `is a`

The examples above illustrate how **the English expressiveness is used to create natural constructs**. (e.g. `in` to express both “belongs” and “input”).

More than that, one code developer might use existing keywords to create new constructs like:

```
define <s'action> "do <expr> for <num> times keeping <block>"
as {...}
...
body()@clk is only {
    { do packet for 10 times keeping {it.kind == GOOD};
}
```

In the example above the code developer reused the keywords “do”, “for” and “keeping” to express his own concepts **without consulting the synonyms dictionary**.

An important aspect that should also be considered is that the E language contains many keywords. **By reserving** them, **the programmer's basic vocabulary is strongly restricted**. Especially when one starts to learn the E language, it's hard to remember all the keywords.

The example above brings in the **Extensibility** capabilities of the E language. E provides powerful mechanisms (advanced macros) to extend the language itself, not only to condense code.

The user defined construct `do packet for 10 times keeping {...}` is not less important than any other constructs in the core language (e.g. `keep for each in`).

To resume, analyzing the original design of the E language, three factors should be carefully considered when deciding which keywords become reserved names:

- The Expressiveness
- The Extensibility

- The Reserved Name space Size

1.3 Drawbacks in the Original Design

The **first drawback** of the original design is that it might lead to **obfuscated code** like:

```
type var :[R, G, B];

extend sys {
  run() is also {
    var var : var;
  }
}
```

On the other hand, the code developer has the full responsibility for writing down clear code. With the help of good methodology (e.g. naming conventions) or tools (e.g. linting) these confusions can be avoided.

The **second drawback** (and **the major one**) is that designing a **front-end** (parser) to the original E language becomes a **difficult** task, at least by the classical parsing methodologies which rely on reserved names to disambiguate contexts.

The **third drawback** is the performance of the front-end which is affected by the more elaborated context analysis.

The **fourth drawback** is syntax error reporting, that is how easily and intuitively the user is helped to debug his code.

##should have an example, not really convinced

1.4 Evaluation Metrics

Starting from the above considerations, the following metrics were used to evaluate each alternative:

Question	Ideally
Affects expressiveness?	NO
Too many reserved names?	NO
Discourages bad programming?	YES
Complicates parsing?	NO

Affects performance?	NO
Complicates syntax error reporting?	NO
Does it impact significantly the existing code base?	NO
Does it weaken the standard?	NO

The question **Does it impact significantly the existing code base?** was answered intuitively because there is no access to the existing code base.

The question **Does it weaken the standard?** is a suggestion BLTF got from the Coordination TF without guidelines on how to measure a standard strength.

2 Alternatives

Several alternatives were analyzed:

1. **No Reserved.** The original design.
2. **Reserve All.** Reserve all keywords, an extreme approach, didactic.
3. **Reserve Phrases.** Reserve all the keywords, but compact them into phrases where possible (e.g. `is empty`).
4. **Reserve Minimal.** Do a careful selection of the reserved names.

A detailed presentation of each alternative follows in the sections below.

2.1 Alternative 1: No Reserved

Preserve the E language as the designer intended. There are almost no reserved names except “TRUE”, “FALSE”, “NULL”, “UNDEF”, “MAX_INT”, “MIN_INT” and “MAX_UINT”.

Question	
Affects expressiveness?	NO
Too many reserved names?	NO
Discourages bad programming?	NO
Complicates parsing?	YES
Affects performance?	YES
Complicates syntax error reporting?	YES
Does it impact significantly the existing code base?	NO
Does it weaken the standard?	N.A.

The code developer gets the most advantages from this solution. Minimal restrictions, no impact on existing code base. The most disadvantages are for the EDA guys.

To resume:

- **User gets +2**
- **EDA gets -3**

2.2 Alternative 2: Reserve All

All the existing keywords in the E language become reserved names. This is the opposite of the original design, analyzed merely for didactic reasons.

We get a huge list of reserved names, with big chances to dramatically affect the existing code base.

Some of the strange reserved names are **a** (appears in `is a` context) and **C** (appears in `is C` routine).

Question	
Affects expressiveness?	YES
Too many reserved names?	YES
Discourages bad programming?	YES
Complicates parsing?	NO
Affects performance?	NO
Complicates syntax error reporting?	NO
Does it impact significantly the existing code base?	YES
Does it weaken the standard?	N.A.

The EDA guys get the most advantages from this solution. The code developer is seriously affected.

To resume:

- **User gets -2**
- **EDA gets +3**

2.3 Alternative 3: Reserve Phrases

This alternative is trying to find a middle way between the above ones. Reserve keywords, but try to compact them into phrases where possible.

For example, since the **a** keyword appears only in the `is a` context, reserve the phrase `is a` instead of each word in it.

A phrase is a sequence of two or more consecutive identifiers, separated by whitespaces or comments, which have a specific meaning in a specific context.

Examples:

```
// one or more spaces between consecutive identifiers
x is a bltf_packet

// new line between consecutive identifiers
get_data(lof_active_receivers : bltf_receiver) : bltf_packet is
undefined;

// single line comments between consecutive identifiers
for each in // why do I have to reverse here?
reverse lof_active_receivers
```

Even considering this approach, there are still too many reserved names (~25% less than in alternative **Reserve All**, see Appendix B for a complete list).

The impact on the existing code base is somehow lower, but how much

lower is hard to estimate. Still a lot of common words are in the reserved name space.

Question	
Affects expressiveness?	NO (not convincing)
Too many reserved names?	YES
Discourages bad programming?	NO (not convincing)
Complicates parsing?	NO
Affects performance?	NO
Complicates syntax error reporting?	NO
Does it impact significantly the existing code base?	NO (not convincing)
Does it weaken the standard?	N.A.

Still the EDA guys get the most advantages from this solution.

The code developer is less concerned about the impact on the existing code base, but he trades this advantage against having to cope with the **phrases concept**.

Apparently this alternative is almost equivalent to the **Reserve All** one.

To resume:

- **User gets -1**
- **EDA gets +3**

2.4 Alternative 4: Reserve Selected

Alternative 3, **Reserve Phrases** is an attempt to find a middle way starting from the extreme Alternative 3, **Reserve All**.

Another approach is to extend the very limited set of keywords in the E language, that is find a middle way starting from Alternative 1, **No Reserved**.

It is easy to see that the keyword **struct** may be reserved since there are little chances that a user might use it to express other things.

On the other hand, reserving the keyword **a** is drastic and it shouldn't be done.

This approach is affected by subjectivity (either individual or at the company level). A vote is required for each keyword.

Also a careful analysis should be done for each keyword to estimate the impact on the existing code base.

The BLTF has no suggestion at this point and the evaluation of such a solution requires more effort.

2.5 Recommendation

No Reserved favors the user.

Reserve All and **Reserve Phrases** favor the EDA.

There is no recommendation at this point.

2.6 Open Questions

1. BLTF answered intuitively “Does it impact significantly the existing code base?”. Any objections?
2. BLTF has no guidelines to measure a standard's weakness. Any suggestions?
3. Should any further effort be invested to evaluate **Reserve Selected**?

3 Appendix A – Terminology

Identifier = A legal E identifier.

Keyword = An identifier which has a specific meaning in a specific context (e.g. "if", "package", "ignore"). Context refers to syntax. "me", "index", "it" etc. do not fall under this category.

Keyphrase = A sequence of two or more identifiers which has a specific meaning in a specific context (e.g. "is a"). The spacing is unrestricted between the identifiers (formally "is a" = 'i's'WSPACE+'a', where WSPACE can be any mixture of white spaces, tabs, new lines or comments).

Reserved Name = A keyword reserved for core language usage. It can appear only in standard specified contexts.

Reserved Phrase = A keyphrase reserved for core language usage. It can appear only in standard specified contexts.

Front-End = An application that reads in E programs, checks the syntax, reports errors if any and outputs more structured data e.g. syntactical trees for further transformations.

4 Appendix B – Keywords and Keyphrases

Even if keywords might appear in different contexts, they are mentioned only once. For example **package** is an encapsulation construct and also a statement, but it appears only in the encapsulation section.

4.1 Keywords

Keywords appearing only in keyphrases were added at the bottom of the list.

1. TRUE // **original E design**
2. FALSE
3. NULL
4. UNDEF
5. MAX_INT
6. MIN_INT
7. MAX_UINT
8. hr // **time units**
9. min
10. sec
11. ms
12. us
13. ns
14. ps
15. fs
16. package // **encapsulation**
17. private
18. protected
19. bool // **scalar types**
20. char
21. int
22. uint
23. bit
24. nibble
25. byte
26. time
27. bits // **type modifiers**
28. bytes
29. list
30. key

31.of
32.import // **statements**
33.define
34.type
35.struct
36.unit
37.extend
38.routine
39.sequence
40.like // **struct or unit statement component**
41.statement // **define statement components**
42.struct_member
43.action
44.exp
45.type
46.block
47.num
48.file
49.command
50.name
51.Type
52.any
53.as
54.attribute // **struct members**
55.when
56.keep
57.event
58.on
59.assume
60.expect
61.cover
62.bind // **port components**
63.undefined
64.external
65.empty
66.in
67.out
68.inout

69.is // **method components**
70.gen // **constraint components**
71.soft
72.select
73.before
74.using // **cover components**
75.item
76.transition
77.cross
78.range
79.var // **actions**
80.break
81.continue
82.if
83.then
84.else
85.case
86.default
87.repeat
88.until
89.while
90.for
91.from
92.matching
93.to
94.do
95.step
96.prev // for each using index (my_index) prev (my_prev)
97.keeping
98.try
99.compute
100.return
101.start
102.wait
103.sync
104.now
105.emit
106.consume

107.new
108.with
109.check
110.assert
111.that
112.print
113.force
114.release
115.exec // **temporals**
116.eventually
117.cycle
118.detach
119.delay
120.fail
121.as_a // **expressions**
122.not
123.or
124.and
125.nor
126.nand
127.nxor
128.a // **ONLY IN KEYPHRASES**
129.c
130.index
131.export
132.simulator
133.verilog
134.vhdl
135.cvl
136.port
137.method
138.routine
139.dynamic
140.foreign
141.call
142.callback
143.async
144.task

145.function
146.variable
147.code
148.procedure
149.driver
150.object
151.inline
152.first
153.also
154.only
155.gen_before_subtypes
156.reset_gen_before_subtypes
157.instance
158.each
159.reverse
160.file
161.line
162.down
163.all
164.first
165.state
166.machine
167.simple
168.buffer

4.2 Keyphrases

1. is empty
2. is undefined
3. using index
4. list of
5. is not empty
6. in range
7. not in
8. c export
9. verilog time
- 10.vhdl time
- 11.method type

- 12.is c routine
- 13.dynamic c routine
- 14.foreign dynamic c routine
- 15.is inline
- 16.is inline only
- 17.is also
- 18.is first
- 19.is only
- 20.keep gen_before_subtypes
- 21.keep reset_gen_before_subtypes
- 22.is instance
- 23.cvl method
- 24.cvl call
- 25.cvl callback
- 26.cvl call async
- 27.cvl method async
- 28.cvl callback async
- 29.verilog simulator
- 30.vhdl simulator
- 31.verilog task
- 32.verilog function
- 33.verilog variable
- 34.verilog code
- 35.vhdl code
- 36.vhdl procedure
- 37.vhdl function
- 38.vhdl driver
- 39.vhdl object
- 40.for each
- 41.in reverse
- 42.for each file
- 43.for each line
- 44.in file
- 45.down to
- 46.all of
- 47.first of
- 48.state machine
- 49.simple port of

50.buffer port of

51.event port

52.call port of

53.method port of

54.using also

55.is a

56.is not a

5 Appendix C – Code Examples

```
define <a'action> "for <n'name> <b'block>" as {
  for each (<n'name>_it) using
    index (<n'name>_index)
    prev (<n'name>_prev)
  in <n'name> <b'block>
};

unit my_unit {
  lof_packet : list (key:id) of my_packet;
  !lof_sender : list of my_sender;

  check_dif(lof_int : int) : int is {
    // "result" is not a keyword
    // it is the name of an automatic variable
    result = MAX_UINT;

    // user defined action
    for lof_int {
      result = result - lof_int_it;
    };

    // "dut_error" is not a keyword
    // it is the name of a predefined routine
    check that result != 0 else dut_error("My error message!");

    var x : time = 1ns;
  };
};

extend my_packet {
  // "post_generate" is not a keyword
  // it is the name of a predefined method
  post_generate() is also {
    // "pop" is not a keyword
    // it is the name of a pseudo method
    address = lof_bytes.pop();
  };

  keep address in [0..50,100..150];
  keep soft address == select {
    50: edges;
    50: others;
  };

  cover sent is {
    item kind using per_instance;
    item length when = (kind != CTRL);
  };
};
```

