The ESL Programming Language

Brian G. Lucas

1 Introduction

ESL is a new programming language designed to be used for efficient programming of embedded and other "small" systems. ESL an acronym for Embedded Systems Language (it can be pronounced “essel”).

ESL is a typed compiled language with features that allow the programmer to dictate the concrete representation of data values. This distinguishes it from languages which implement only “abstract” types or types whose representation is architecture-dependent. The programmer can dictate the details of data representation, including such things as “endian-ness” and the exact placement of bits, which are necessary in dealing with external representations of data layout, e.g., communication protocols or device registers.

ESL is not really a “new” programming language - all the elements have probably been seen in other programming languages. In many respects, it is a conventional language whose features, for the most part, will be familiar to programmers who have had experience with any of the compiled languages introduced in the past 50 years.

If the ESL syntax bears some resemblance to Googles new programming language *go*, that is due to common inspiration. The syntax for ESL was mostly in place before *go* was announced. There is one exception: the syntax for methods, which was added later and shamelessly stolen from *go*.

Copyright © 2010 Brian G. Lucas. Version of 2011-01-05

2 Syntactic Elements

2.1 Notation

The syntax is specified in EBNF (Extended Backus-Naur Form). Lexical symbols are enclosed in double quotes.

2.2 Comments

There are two ways to comment text: by line or by block. Line comments start with "`//`" and end at the end of the line. Block comments start with "`/*`" and end with "`*/`". Block comments do not nest.

2.3 Reserved Words

There are no reserved words in ESL. There are many keywords whose "specialness" is determined by context. As a result, these keywords can be used as ordinary identifiers in other contexts. As for keywords, they are all are lower case.

2.4 Identifiers

The first character of an identifier must be from the set {`A-Z`,`a-z`}, remainingcharacters can be chosen from the set {`A-Z`,`a-z`,`0-9`,`_`}. The system reserves identifiers starting with a "`_`"for predefined symbols. In addition, the identifier consisting of a single "`_`" has a special meaning. It is not entered into the symbol table and can be used to specify anonymous fields in records, or placeholders in enumerations.

Identifiers may be keywords depending on context. Otherwise they are, as yet, unclared names or previously declared names of a several kinds: type names, variable names, procedure names, package names, field names, enumeration constant names, or aliases.

2.4.1 Identifier Namespaces

There are two visiblity classes of identifiers: public and private. The public identifiers are nested at four levels: universal, global, package, and procedure. Universal identifiers are pre-defined by the compiler. Global identifiers declared at the outermost nesting level of a program. Package identifiers are declared within a package. And finally, procedure identifiers are declared within a procedure. There is no further nesting within a procedure, i.e., no "blocks".

Private identifiers include enumeration constants and fields within a record. They are accessed by prefixing the enumeration type name or the record variable name.

2.5 Numeric Literals

Numbers are by default decimal. For other number bases, a two-character prefix is used. The first character is always a zero, the second character, always lower case, indicates the number base:

`0b` binary - Only digits `0-1` are allowed.
`0o` octal - Only digits `0-7` are allowed.
`0x` hexadecimal - Only digits `0-9` and letters `A-F` or `a-f` are allowed.

In any number, after the first digit (which follows the optional prefix), an underline "`_`" is allowed for readability.

Examples:

```
399                 // decimal
0b0010_0001         // binary
0o02_56             // octal
0x01_F4             // hexadecimal
1_999_999           // decimal
```

2.6 Character Literals

Character literals are unsigned integer constants. Character literals are enclosed within single quotes, the literal is either any single ASCII character (excluding the single quote and newline) or an escape sequence. Escape sequences start with a backslash and include:

`\\` - represents a single backslash

`\n` - represents a newline

`\r` - represents a carriage return

`\f` - represents a form feed

`\t` - represents a horizontal tab

`\b` - represents a backspace

`\v` - represents a vertical tab

`\x`*XX* - represents an 8-bit character with value given by the two hex digits *XX*

`\u`*XXXX* - represents a 16-bit (e.g. unicode) character with value given by four hex digits

`\U`*XXXXXXXX* - represents a 32-bit (e.g. unicode) character with value given by eight hex digits

2.7 String Literals

A string literal is a constant array of bytes terminated by a NUL. Each byte is either an ASCII character, an 8-bit escape as defined in the above character literal section, or a part of a UTF8 sequence. The character escapes which generate greater than 8-bit values are converted into a sequence of bytes by UTF8 encoding.

3 Types and Declarations

Types in ESL are not abstract in the sense that they are designed to be concrete descriptions of containers for a set of values. When instances of types are allocated, the assignment to bit positions and memory addresses depends on architecture-dependent defaults and which can be overridden by the use of optional type attributes.

There are two units of allocation:

bits - the atomic unit of data

bytes - the smallest sequence of bits that has a memory address, in most contemporary machines this is 8 bits.

Types consist of the integral types, references (pointers), and aggregate types. Integral types are unsigned or signed and are always subranges. Enumerations are a special type of unsigned type. Signed integers are assumed to have twos complement representation. Aggregate types are either arrays or records.

Some types can be “extended” from base types.

3.1 Integer Type

The integer types are specified by giving their lowest and highest values. The resulting type may be signed or unsigned, depending on the values given. The representation of unsigned subranges always includes zero. The representation of signed subranges is symmetrical (in the twos-complement sense) around zero.

Examples:

```
type percent: 0..100;              // an unsigned type
type srange: -1000..1000;          // a signed type
```

3.2 Enumerated Type

Enumerated types are a mapping onto the unsigned integers. They are defined by listing all the values, in monotonic increasing order, that make up the type.

These values may represented by identifiers. The identifiers will be assigned specific numeric values by the compiler, or optionally, can be given specific numeric values by the programmer. An enumerated type consists of a contiguous range of values starting at zero, there are no gaps. Some of the values need not be represented by identifiers.

Operations on enumerated types are limited to assignment and comparison. However, casting to unsigned integer types is allowed.

Each enumerated type is distinct. That is, two different enumerated types are not compatible for assignment or comparison.

An enumerated type can be extended by adding additional values.

Examples:

```
type answer: (yes, no);                        // yes=0, no=1
type foo: (low=5, medium, high=10)// medium=6,
           // values 0..4, 7..9 are part of the enumeration

type fuzzy(answer): (maybe, possibly);
           // extends answer: yes=0, no=1, maybe=2, possibly=3
```

3.3 Reference Type

This is just a pointer to another type, the base type. The base type must have a type name.
It is possible that a programmer wants to reference a type that is not declared. This is possible by partially declaring the type name and then later later completing the type definition.

Examples:

```
type byte: 0..255;                // the base type
type ptr_to_byte: @byte;          // a pointer to the base type

type ptr_to_forward: @forward; // forward reference
type forward: sometype;           // the real declaration is later
```

3.4 Array Type

Arrays are aggregates of a base type. The base type is accessed by the indexing operation. Array indices are unsigned and always start at zero.

Array slicing is an operation that allows access to a contiguous sub-array of values. A slice operator is square brackets enclosing an offset and length separated by a colon.

3.4.1 Fixed Sized Arrays

The index specification of a fixed sized array can be either an expression that is a compile-time constant, or the name of an unsigned type (e.g. an enumeration).

Examples:

```
type array1: [100]_uint;
type enum: (zero, one, two, three);
type array2: [enum]enum;
```

3.4.2 Unknown Sized Arrays

Arrays of indeterminate size are allowed as targets of a reference (pointer).

Examples:

```
type pstring: @[]_byte;
type argv: @[]@[]_byte;           // Unix argv type
```

3.5 Record Type

Record types are a way to define an aggregation of dissimilar types as new type.

Record types can be extended in two ways. First, additional fields can be added to the end of the record. Second, fields already defined in the base type can be "hardened" by declaring them to be constants. [This doesn't work yet.]

3.6 Type Attributes

Type attributes can be used to alter the way instances of that type are allocated or behave. Allocation attributes alter the size, alignment, bit-order, and byte-order of the memory structure. Useage attributes alter the way that instances of the type can be used.

3.6.1 Size

A size attribute overrides the default (target dependent) size of a type:

`bits(`cexpr`)` - instances of this type should be exactly this many bits in size.

`size(`cexpr`)` - similar to bits(), except the size is in memory-units.

At most, only one of these attributes should be present. If no alignment attributes are present, the size of a type may be increased to a target-dependent value.

3.6.2 Alignment

Alignment attributes change the way a type is positioned in memory:

`packed` - when the type is allocated no padding occurs at the bit level nor the memory-unit level.

`mempacked` - that padding occurs only up to the next memory-unit.

`align(`*cexpr*`)` - padding up to an alignment in memory, or a record offset, will be made to the *cexpr* byte boundary.

At most, only one of these attributes should be present. If none are present, then allocation will use whatever the target architecture prefers. On many architectures alignment is type dependent .

3.6.3 Bit Order

The bit order attributes affect allocation within an aggregate, record or array, that is packed.

`lsb` - allocation begins at the least significant bit within a memory-unit and with the memory-unit with the lowest address, i.e. "little-endian"

`msb` - allocation begins at most significant bit within a memory-unitand with the memory-unit with the highest address, i.e. "big-endian" also known as "network order"

At most, only one of these attributes should be present. If neither attribute is given, the compiler will use whatever the target architecture prefers.

3.6.4 Memory Order

The memory order attributes affect how a multiple byte type is stored in memory. The compiler will “byte-swap”, if necessary when loading or storing to insure that the byte-order is correct for the target machine.

`le` - multiple byte types will have the least significant byte at the first allocated byte.

`be` - multiple byte types will have the most significant byte at the first allocated byte.

At most, only one of these attributes should be present. If neither attibute is given, the compiler will use whatever the target architecture prefers.

3.6.5 Access Restrictions

These attributes allow the programmer to restrict access to allocations of this type.

`ro` - read only

`wo` - write only

At most, only one of these attributes should be present. If neither attibute is given, the default is to allow both read and write.

3.6.6 Input and Output

These attributes indicate to the compiler that allocations of this type has external side effects. These attributes are most often used to describe device registers which do not behave as normal memory.

`in` - loads of instances of this type must not be optimized away because reading may cause external side effects

`out` - stores to instances of this type must not be optimized away because writing may cause external side effects

Some C-based programming languages use the keyword “volatile” to lump together both attributes `in` and `out`.

3.7 Built-in Types

There are a small number of pre-defined types. These include:

`_boolean` an enumerated type consisting of the constants false and true.

`_byte` an unsigned integral type with a range equal to that which can be stored in the target architecture's minimal addressable unit. For modern processors, this is equivalent to `_uint8`.

`_uint8`,`_uint16`,`_uint32`,`_uint64` are unsigned types of 8, 16, 32, and 64 bits respectively.

`_int8`,`_int16`,`_int32`,`_int64` are signed types of 8, 16, 32, 64 bits respectively.

`_uint` an unsigned integral type with a range equal to that which can be stored in the target architecture's “natural word size”. This is usually the size of a integer register.

`_int` a signed integral type with a range equal to that which can be stored in he target architecture's “natural word size”.

`_memory` is just a name for the `[]_byte` type.

`_address` is just a name for the `@_memory` type.

Because these type identifiers start with “_” they can not be redefined by the programmer. For convienence, the alias “`boolean`” is predefined to be the same as “`_boolean`”; but can be redefined by the programmer. The original boolean constants are always available by the names `_boolean.false` and `_boolean.true`.

3.8 Declaration Statement Syntax Summary

decl-stmt = *type-stmt* | *var-stmt* | *const-stmt* | *alias-stmt*
type-stmt = "`type`" *ident* ["`(`" *typeid* "`)`"] "`:`" *type-def* ["`:`" *type-attr-list*] "`;`"
var-stmt = "`var`" *ident* { "`,`" *ident* } "`:`" *type-def* ["`:`" *var-attr-list*] "`;`"
const-stmt = "`const`" *ident* ["`:`" *type-def*] "`=`" *constant* "`;`"
type-def = *type-name* | *type-range* | *type-enum* | *type-ref* | *type-record* | *type-array*
type-range = *number* "`..`" *number*
type-enum = "`(`" *enum-list* "`)`"
enum-list = *enum-def* { "`,`" *enum-def* }
enum-def = *ident* ["`=`" *sconst*]
type-ref = "`@`" *type-def*
type-record = "`{`" *field* { *field* } "`}`"
field = *ident* "`:`" *type-def* ["`:`" *attr-list*] "`;`"
type-array = "`[`" *type-index* "`]`" *type-def*
type-name = *name*
type-attr-list = *type-attr* { , *type-attr* }
type-attr = "`bits`" "(" *cexpr* ")" | "`size`" "(" *cexpr* ")" | "`align`" "(" *cexpr* ")" |
"`lsb`" | "`msb`" |
"`le`" | "`be`" |
"`ro`" | "`wo`"

4 Expressions

4.1 Type Inference

In many cases, as an expression is being evaluated, the type of an identifier can be inferred. As a simple example, take the assignment of an enumeration constant to an enumeration variable:

```
type enum: (A, B, C);
var foo: enum;
foo = A;               // A is enum.A
```

Since enumeration constants are in a private namespace associated with the enumeration type, the name “A” would be invisible, it would have to be referred to by its qualified name “enum.A”. However, assignment allows the type of the lefthand side (the “target type”) to guide the type of the righthand side. Any names on the righthand side which are unknown are interpreted in the context of the type.

The target type is also set when evaluation an expression for an actual parameter. Given the above definition of “enum”, consider:

```
proc bar(a: enum);
proc baz()
{   bar(B);// B is enum.B
}
```

If the target type has an aggregate type, this controls the construction of aggregate expressions.

4.2 Fundamental Terms

4.2.1 Type Queries

An identifier which has been declared as a type or variable may be queried to get attributes relating to its type. The result is a compile-time constant.

type-query ::=[*type-name* | *var-name*] ? *type-qattr*
type-qattr ::= "min" | "max" | "bits" | "size"

4.3 Unary Operations

negation, logical inversion, boolean not

4.4 Binary Operations

4.4.1 Multiplicative Operations

multiply, divide, modulo, shifts, and

*
/
%
<<
>>
&

4.4.2 Additive Operations

add, sub, xor, or

+
-
^
|

4.4.3 Comparison Operations

All comparison operations result in the built-in boolean type.

==
!=
<
<=
>
>=

4.4.4 Boolean And

&&

4.4.5 Boolean Or

||

4.5 Intrinsic Operations

There are several built-in operations which use a function call syntax:

_abs

_min

_max

4.6 Expression Syntax Summary

cexpr = *expr* # compile time constant
expr = *baexpr* { "||" *baexpr* }
baexpr = *cmpexpr* { "&&" *cmpexpr* }
cmpexpr = *addexpr* [*cmpop addexpr*]
cmpop = "==" | "!=" | "<" | ">" | "<=" | ">="
addexpr = *mulexpr* { *addop addexpr* }
addop = "+" | "-" | "|" | "^"
mulexpr = *uexpr* { *mulop mulexpr* }
mulop = "*" | "/" | "%" | "<<" | ">>" | "&"
uexpr = *term* | *unop term*
unop = "+" | "-" | "~" | "!"
term = *number* | *qname* | "(" *expr* ")"

5 Statements

All statements, except for assignment statements, begin with a keyword. All statements end with a semicolon. The declarative statements have been covered previously.

Statement groups start with a "`{`" and end with a "`}`". There is no semicolon after a statement group.

proc-stmt = "`{`" *stmt-list* "`}`" | *asgn-stmt* | *if-stmt* | *loop-stmt* | *while-stmt* | *exit-stmt* | *return-stmt*

5.1 Assignment Statement

5.1.1 Scalar Assignment

Scalar assigment statements can do multiple assignment. In the following example two values returned by the procedure "sumdiff" are assigned to "x" and "y" and the values in "a" and "b" are swapped:

```
b, x, y, a = a, sumdiff(a, b), b;
```

asgn-stmt = *lhs-list* "`=`" *expr-list* "`;`" |
lhs "`+=`" *expr* "`;`" |
lhs "`-=`" *expr* "`;`"

5.1.2 Record Copy

5.1.3 Array Copy

5.2 If Statement

The if statement can be a simple if-then with optional else, or a statement which selects among multiple options.

if-stmt = "`if`" *bool-expr* "`then`" *stmt* ["`else`" *stmt* |
"`if`" *expr is-list* ["`else`" *stmt*]
is-list = *is-clause* [*is-list*]
is-clause = "`is`" *is-value-list* "`then`" *stmt*
is-value-list = *is-value* { "`,`" *is-value-list* }
is-value = *cexpr* ["`..`" *cexpr*]

5.3 Loop statement

loop-stmt = "`loop`" *stmt*

5.4 While statement

while-stmt = "`while`" *bool-expr* "`do`" *stmt*

5.5 Exit statement

Exit statements must be enclosed in either a loop statement or a while statement. When the boolean expression evaluates to true, the loop is exited. Additional code may optionally be executed as the exit is taken.

exit-stmt = "`exit`" *bool-expr* ["`then`" *stmt*]

5.6 Return statement

return-stmt = "`return`" *expr-list* "`;`"

5.7 Asm statement

The asm startment is used to insert an assembly language instruction.

asm-stmt = "`asm`" string ["`,`" string { "`,`" expr }] "`;`"

The first string is the assembly language prototype. The second (optional) string is the constraint list. The statement ends with a list of expressions to be used as arguments.

5.8 Alias statement

The alias statement can be used for two distinct purposes. The first, more common, use is simply to introduce a new name into the current namespace which is a short-cut to another name. This is usually used to simplify access to names in a package.

alias-stmt = "`alias`" alias-list "`;`"
alias-list = *alias-spec* { "`,`" *alias-spec* }
alias-spec = *qname* "as" (*ident* | *pakg-name* "`.`" *name*)

The other use of an alias statement is to fixup a forward reference made in a package outside of that package (perhaps in a subsequent package).

```
package foo
{ type R:
  { next: @R;          // self referential
    what: @P;          // forward reference
  };
}
package bar
{ alias foo.R as R;    // simplify
  type P:
  { next: @P;          // self referential
    from: @R;          // points to R in package foo
  };
  // fixup the forward reference in package foo
  alias P as foo.P;
}
```

6 Procedures and Methods

Procedures may return zero or more values.

procedure = "`proc`" *ident signature* ["`:`" *proc-attr*]
("`;`" | "`{`" { *proc-stmt* } "`}`")
signature = "`(`" [*parm-list*] "`)`" ["`:`" [*retv-list*]]
parm-list = *formal* { `,` *formal* }
formal = ident "`:`" type-def
retv-list = *type* { `,` *type* }

7 Packages

Packages are the primary method of reusing code by providing a private namespace. A package is a bundle of declarations and procedures wrapped in a package name. Packages can not be nested.

Examples:

```
package foo:
{ type type1: 0..1000;
  type type2: (no, yes, maybe);
  proc sub1(x: type1, y:type 1): type2
  { statement-list }
}
```

package = "package" *ident* "{" *pkg-stmt-list* "}"
pkg-stmt-list = *pkg-stmt* { *pkg-stmt* }
pkg_stmt = *procedure* | *alias_stmt* | *type_stmt* | *var_stmt* | *const_stmt*

8 Programs

Programs consist of global declarations, packages, procedures and import statements.

8.1 Import Statement

The import statement causes file inclusion. The included file typically contains one or more package.

Package specifiers can be long. When a package is imported, the package name can be renamed with the `alias` statement. This enables short names to be used in the program.

Alternatively, individual identifiers in the package can the aliased. In that way, the package name does not have to be used as a prefix on every reference to an identifier within that package.

program = *prog-stmt* { *prog-stmt*}
prog-stmt = import-stmt | package | procedure | alias_stmt | type_stmt | var_stmt | const_stmt
import-stmt = "`import`" *package-specifier* "`;`"

9 ESL For C Programmers

Since C has been the dominant system programming language for the past 30 years or so, this chapter is designed to help C programmers to understand how to express C idioms as ESL idioms. For those programmers whohave been exposed to "Pascal-like" languages (e.g. Module, Ada, etc.), some of this might be familiar.

One of the major differences between C and ESL is that ESL has no reserved words! You are free to name a variable “if” or a type “then” or a procedure “else”.

9.1 Preprocessor

ESL does not have a separate pre-processor language.

9.1.1 Include

The ESL "package" and "import" features replace the C "header files" and"#include" mechanism.

9.1.2 If and ifdef

There is no equivalent to "#ifdef". However the "#if" pre-processor feature has an ESL equivalent. The if statement can be used if its selection expression is a compile-time constant.

9.2 Declarations

A major difference between C and ESL is that ESL uses the Pascal-like syntaxfor declarations. In ESL the identifer comes first followed by the type. In C, the reverse is true.

9.2.1 Typedefs

The ESL type statement is the equivalent of the C typedef.

9.2.2 Enumerations

The identifiers representing values in a C enumerations are exposed in the same name space as the enumeration. In ESL (as in many other languages) the identifier needs to be exposed by giving its type name as a prefix. In many cases, however, the type can be inferred and the enumeration prefix is not necessary.

9.2.3 Pointers and Arrays

C example:

```
typedef int *pi;                  // pointer to int
typedef int ai[8];                // array of int
typedef int *api[8];   // array of pointers to int
typedef int (*pai)[8]; // pointer to array of int
```

ESL example:

```
type pi: @_int;                   // pointer to int
type ai: [8]_int;                 // array of int
type api: [8]@_int;               // array of pointers to int
type pai: @[8]_int;               // pointer to array of int
```

9.2.4 Pointers to Procedures

In C, the declaration of a pointer to a function has a syntax that has confused even experienced programmers. Consider a pointer to a function that takes two integer arguments and returns an integer:

```
typedef int (*func)(int, int);
```

In ESL, the syntax is still a little funky due to the use of the null identifier "_", but perhaps easier to read:

```
type func: @_(a:_int, b:_int): _int;
```

Or the example from page 122 of the 2nd edition of Kernighan and Ritchie, x is definied as an "array[3] of pointer to function returning pointer to array[5] of char":

```
typedef char (*(*x[3])())[5];
```

Which is ESL is written pretty much as the text describes it:

```
type x: [3]@_(): @[5]_byte;
```

9.3 Parameters and Arrays

In C function parameters are passed "by value" except for arrays. In the case of an array, the address of the array is passed instead. In ESL all procedure parameters are passed "by value", with no exceptions. If you want a pointer to an array, then the procedure parameter must be so declared.

Also, in C, pointer arithmetic may be used as a mechanism to access elements in arrays. This is not allowed in ESL, access to the element must be done with array indexing.

For example, the Unix command line convention passes an a pointer to an array of pointers to strings as an argument vector (e.g. "argv"). In C this is often specified as:

```
int main(int argc, char *argv[])
{
}
```

In ESL all the initial pointer and the specification of a string as “`[]_byte`” must be explicit:

```
proc main(argc: _int, argv: @[]@[]_byte): _int
{
}
```

9.4 Statements

9.4.1 Assignment Statements

Multiple assignment is not allowed in ESL. C statements such as

```
x = y = 0;
```

must be split into multiple assignment statements in ESL.

9.4.2 Assignment Within an Expression

In ESL, expressions cannot have side effects (except for procedure calls), such C constructs must be written as a separate assignment statement.

9.4.3 If Statements

If statements in ESL are very similar to those in C. The syntax is a little different, the parentheses surrounding the expression are not necessary, but the `then` keyword is.

9.4.4 Switch Statements

ESL doesn't have an explicit statement type corresponding to the C “switch” statement. The ESL “if” statement covers the same territory with one important difference - ESL doesn't allow “fall-through”. Also each ESL “case” is a single statement, if more than one is required than a statement group delimited by "`{`" and "`}`" is
required.

C Example:

```
unsigned int foo;
switch (foo) {
  case 0: case 1: case 7: case 8: case 9: case 15:
    statement1;
    break;
  case 2: case 3: case 4:
    statement2;
    statement3;
  case 5:       /* fall thru */
    statement4;
    break;
  default:
    statement5;
    break;
}
```

ESL Example:

```
var foo: 0..15;
if foo
is 0..1, 7..9, 15 then
  statement1;
is 2..4 then
{ statement2;
  statement3;
  statement4;  // no fall thru
}
is 5 then
  statement4;
else
  statement5;
```