The ESL Programming Language

Brian G. Lucas

 Version of 2013-12-17

Table of Contents

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 (pronounced: “ESS-el”).

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 Google's new programming language *Go*, that is due to common inspiration. The syntax for ESL was mostly in place before G*o* was made public. There is one exception: the syntax for bound procedures, or as *Go* calls them: methods. This was shamelessly stolen from *Go* and incorporated into ESL.

2 Syntactic Elements

2.1 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.

2.2 Reserved Words

There are no reserved words in ESL. There are many keywords that have special meaning only in specific contexts. Therefore, these keywords can be used as ordinary identifiers. Keywords are always lower case.

2.3 Identifiers

The first character of an identifier must be from the set {`A-Z`,`a-z`}, remaining characters 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 also contain UTF8 sequences any place an alphabetic character is valid. (Identifiers with UTF8 sequences may not be handled correctly in the LLVM backend, by assemblers, or by linkers.)

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

2.3.1 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. Packages may be nested. Finally, procedure identifiers are declared within a procedure. There is no further nesting within a procedure, i.e., no "blocks", and procedures can not be nested.

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

2.4 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
```

2.5 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

2.6 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 sub-ranges. Enumerations are a special type of unsigned type. Signed integers are assumed to have twos complement representation. Aggregate types are either arrays or records.

Enumerations and records can be “extended” from base types. Details on how each of these types are extended will be discussed in the section where the specific type is defined.

Types are declared with type statements. Type statements may occur anywhere in the statement flow.

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 sub-ranges always includes zero. The representation of signed sub-ranges 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 rom zero to some maximum value, there are no gaps. All 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
```

The size of an enumeration is the minimal number of bytes necessary to hold the largest defined value. For all the enumerated types in the previous example, the size that would be needed for all three, i.e., answer, foo, and fuzzy, would be one byte in most architectutes.

3.3 Data 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 has not yet been declared. This is possible by referencing the type name and then later later completing the type definition for that name.

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 Procedure Reference Type

3.5 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.

3.5.1 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;
```

3.5.2 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
```

3.6 Record Type

Record types are a way to define an aggregation of dissimilar types as new type. A record is defined as an order list of *fields.* Each field may be of any type. The following example demonstrates how the type base is declared with two fields: one and two.

```
type base:
{   one: _uint32;
    two: _uint32;
};
```

The final field of a record may have a type of unknown size:

```
type ident:
{   length: _uint;
    name:   []_byte;
};
```

As mentioned earlier, a record type is one of the types that can be 'extended'. The following example extends the type '`base`' declared above:.

```
type newbase(base):
{   three: _uint16;
    four:  _uint8;
};
```

The new extended type, '`newbase`', includes all the fields that were declared in the type '`base`', and two additional fields. New fields in the extended type are always added at the end.

3.7 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. Usage attributes alter the way that instances of the type can be used.

3.7.1 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.

3.7.2 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 .

3.7.3 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-unit and 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.

3.7.4 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` - types that span multiple bytes will have the least significant byte at the first allocated byte.

`be` - types that span multiple bytes 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.

3.7.5 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.

3.7.6 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`.

3.8 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` and `_int` are unsigned and signed integral types 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.

`_memory` is a synonym for the `[]_byte` type.

`_address` is a synonym 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`.

4 Variables

4.1 Variable Declarations

Variable declarations instantiate a type. The visiblity and lifetime of the variable depend on the nesting level at which the declaration appears.

4.2 Variable Attributes

4.2.1 Global

The global attribute indicates that the variable name has global linkage. The attribute may optionally specify a name by which it will be known globally.

```
var foo1: _int: global;           // global linkage name “foo1”
var foo2: _int: global("FOO");    // global linkage name “FOO”
```

4.2.2 Weak

The weak attribute is similar to the global attribute except that the linkage type is global but weak. This attribute may be used in combination with the global attribute when an explicit global name is desired.

```
var bar1: _int: weak;             // weak global linkage name “bar1”
var bar2: _int: global, weak;     // as above with name “bar2”
var bar3: _int: weak, global("BAR"); // weak global linkage name “BAR”
```

4.2.3 External

The external attribute indicates that the variable instantiation is external to the program being compiled. The attribute may optionally specify a name (via a string) of the external variable. Another option is to specifiy an address.

```
var baz1: _int: external;         // external linkage name “baz1”
var baz2: _int: external(“BAZ”);  // external linkage name “BAZ
var vect: @_[32]_uint: external(0x20000000);  // external at a fixed address
```

4.2.4 Segment

The section attribute indicates that the variable should be placed in the named section by the linker. The format of section names is linker-specific.

Example:

```
var foo: _int: section(".far_memory");
```

5 Expressions

5.1 Type Inference

In many cases, as an expression is being evaluated, the type of an identifier can be inferred. This inference takes place in the following situations:

1. Evaluation of the right-hand side of an assignment
2. Evaluation of actual parameter in a procedure call
3. Evaluation of a returned value in a procedure

As a simple example of the first case, take the assignment of an enumeration constant to an enumeration variable:

```
type enum: (A, B, C);
var foo: enum;
foo = A;              // same as foo = 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 left-hand side (the “target type”) to guide the type of the right-hand side. Any names on the right hand side which are unknown are interpreted in the context of the type.

The target type is also set when evaluating an expression for an actual parameter or returned value. Given the above definition of “enum”, and example of the second case is:

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

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

5.2 Conversion

Inevitably, one will have to convert the type of a variable into one of more preferable type for a particular situation. Probably the most common situation is the conversion of a variable from one bit width size to another. Widening of scalars occurs automatically, as there is no loss of information. [Currently, this may not work everywhere, e.g. formal to actual parameters.] Narrowing, on the other hand, requires the programmer to use explicit type conversion. In the following example, the variable one, a 32 bit variable, is assigned to the variable two which has a size of 16 bits.. The following example shows how the variable is properly converted using the name of the conversion type. In this case, the name _uint16 which is that of the variable two.

```
var one:    _uint32;
var two:    _uint16;

two = _uint16(one);
type exptype:
{
    one:   _uint32;
    two:   _uint32;
};

type exptype_ref: @exptype;


proc create_exp()
{
    var eref : exptrype_ref;


    eref = exptype_ref(mem.Alloc(exptype?size));
}
```

5.3 Truncation

5.4 Promotion

5.5 Sign Extension

5.6 Fundamental Terms

Precedence as presented in the following sections....

Operator	Description	Associativity
.	Package dereference	left-to-right
() _abs _man _min	Parentheses (function call) Absolute value Maximum value Minimum value	left-to-right
. . [] @ ?	Record dereference Method introduction Brackets (array subscript) Pointer dereference Property inquiry	left-to-right
+ - ~ !	Unary plus (no-operation) Unary minus Bitwise negation Boolean not	right-to-left
* / % << >> &	Multiplication Division Modulus Bitwise shift left Bitwise shift right Bitwise AND	left-to-right
+ - | ^	Addition Subtraction Bitwise OR Bitwise XOR	left-to-right
== != < <= > >=	Relational equal to Relational not equal to Relational less than Relational less than or equal to Relational greater than Relational greater than or equal to	left-to-right
&&	Boolean AND	left-to-right
||	Boolean OR	left-to-right

5.6.1 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.

- `min`
- `max`
- `size`
- `bits`
- `align`
- `len` – applies only to arrays

An example:

```
type a: _uint8;
var b: 0..31;
var n: _uint;

n = a?size;           // value of 'n' is 1
n = a?bits;           // value of 'n' is 8
n = a?max;            // value of 'n' is 255
n = a?min;            // value of 'n' is 0
n = b?size;           // value of 'n' is 1
n = b?bits;           // value of 'n' is 5
n = b?max;            // value of 'n' is 31
n = b?min;            // value of 'n' is 0
```

5.7 Unary Operations

negation, logical inversion, boolean not

- -
- ~ (one's complement)
- ! (booleannegation)

5.8 Binary Operations

5.8.1 Multiplicative Operations

multiply, divide, modulo, shifts, and

- *
- /
- %

<<

>>

&

5.8.2 Additive Operations

add, sub, xor, or

+

-

^

|

5.8.3 Comparison Operations

All comparison operations result in the built-in boolean type. Reference types and array types support only equal and not-equal operations. For array types, depending on length and alignment, a call to a library routine may be generated.

==

!=

<

<=

>

>=

5.8.4 Boolean And

&&

5.8.5 Boolean Or

||

5.9 Intrinsic Operations

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

5.9.1 Absolute Value

The _abs operator takes the absolute value of an signed integral number. For example:

```
var a: _int32;
var b: _uint32;

a = -1234;
b = _abs(a);          // value of 'b' is 1123
```

5.9.2 Maximum Value

The _max operator takes the largest value of two integral numbers. For example:

```
var a: _int32;
var b: _int32;
var c: _int32;

a = 1234;
b = 5678;
c = _max(a, b);       // value of 'c' is 5678
```

5.9.3 Minimum Value

The _min operator takes the smallest value of two integral numbers. For example:

```
var a: _int32;
var b: _int32;
var c: _int32;

a = 1234;
b = 5678;
c = _min(a, b);       // value of 'c' is 1234
```

5.9.4 New Allocation

The `_new` operator allocates memory for a new instantiation of a type. For types with an unknown array length, the length of the "unknown" part must be specified in parentheses after the type name. An optional second parameter allows for an implementation specific option (e.g., choice of address space). A library call will be generated for each instance of `_new`.

Examples:

```
type enum: (A, B, C);
type ident:
{   length: _uint;
    name:   []_byte;
};
var pe: @enum;
var pr1, pr2: @ident;
pe = _new(enum);
pr1 = _new(ident(16));           // name array has length 16
pr2 = _new(ident(8), 1);         // use of optional parameter
```

5.9.5 Delete Allocation

The `_delete` operator de-allocates memory for a previously allocated type. As with `_new`, an "unknown" length must be explicitly specified. This allows for memory allocation algorithms which do not store the size of allocated memory in meta-data. Again, as with `_new`, there is an optional second parameter.

Examples, based on the examples in the previous section:

```
_delete(pe);
_delete(pr1(16));
_delete(pr2(8), 1);
```

6 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.

6.1 Declarative Statements

6.1.1 Type Statement

The `type` statement associates a name with type. Refer the the section on types for examples.

6.1.2 Const Statement

The `const` statement creates a named constant.

A `const` statement initialized by a string may trim the terminating NUL by explicit typing.

```
const foo1 = "abcd";                    // foo1?len = 5
const foo2: [4]_byte = "abcd";          // foo2?len = 4
```

6.1.3 Variable Statement

The `var` statement instatiates a variable. Inside a procedure, the lifetime of the instantiation is from

procedure entry to exit. Elsewhere, the lifetime of the variable is the life of the program. Multiple variables with the same type may be created in a single statement.

```
var x, y, z: _uint;
```

6.1.4 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.

The other use of an alias statement is to fix up 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;
}
```

6.2 Assignment Statement

6.2.1 Scalar Assignment

The most common example of scalar assignment is when a variable, this case a, is assigned a value that is a constant or the value of another variable.

```
a = 0;
a = b;
```

ESL also allows the assignment of multiple variables in a single statement.

```
a, b = 0, 0;
c, d = a, b;
```

In this example, the first statement assigns the variables a and b a value of zero, while the second

statement assigns the values of c and d with the values of a and b. After the execution of the second statement the values of c and d would be zero, respectively.

As will be discussed in more detail in Section 6: Procedures, a procedure can return multiple values. The following example shows how the variables a and b are assigned the values that are returned from the procedure sumdiff.

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

A more complex example shows a combination of assignments from variables and from the return values of the procedure sumdiff.

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

In all the examples in this section, we have seen one or more variables assigned values with a corresponding number of variables on the right hand side of the assignment statement. Either from a variable or procedure or a combination of the two. In all cases, there has been a one-for-one assignment of a variable from another variable or from the return of a procedure. ESL does not allow for the splitting of a variable into composite types for assignment. In other words, two 16-bit variables cannot be assigned a value from a 32-bit variable on the right hand side. Conversely, a 32-bit variable cannot be assigned the values of two 16-bit variables on the right hand side.

6.2.2 Increment and Decrement Operators

[Should this be moved to section 4.4? Or should this section be moved prior to the discussion of operators?]

6.2.3 Record Copy

6.2.4 Array Copy

6.3 Control Statements

6.3.1 If Statement

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

6.3.2 Loop Statement

6.3.3 While Statement

6.3.4 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.

6.3.5 Return Statement

6.4 Asm Statement

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

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.

7 Procedures

Procedures may return zero or more values.

7.1 Normal Procedures

Normal procedures are as expected. In the following example, the procedure ‘sumdif’ is defined to have two parameters and return two values:

```
proc sumdiff(a:_int, b:_int):_int, _int
{
    return a+b, a-b;
}

var x, y: _int;
...
x, y = sumdiff(x, y);
```

7.2 Bound Procedures

Bound procedures allow the programmer to write a procedure associated with a particular type. (This is similar to “methods” in other languages, but the binding is to a type not a “class”.) In the following example, the type ‘foo’ is declared and a bound procedure that operations on that type is defined:

```
type foo: 0..100;

proc (a: foo) saturate(b: _uint): foo
{   var sum: _uint;
    sum = a + b;
    if sum > foo?max then sum = foo?max;
    return foo(sum);
}

var x: foo;
...
x = x.saturate(10);
```

While the previous example showed a procedure bound to a scalar type, a more usual case is when the type is a record:

```
type rec: { a:_uint; b:_boolean };

proc (p:@rec) init(x:_uint)
{
    p.a = x;
    p.b = true;
}

var r: rec;
...
r.init(0);
```

7.3 Procedure Attributes

7.3.1 Inline

Inline is an attribute only of a procedure, i.e., proc statement. The inline attribute is currently only a hint to the compilier to tell it to “try very hard to inline”. Procedures without the inline attribute may also be inlined, unless a compiler flag is used to disable inlining.

7.3.2 Global

The global attribute for procedures is used just as it is for variables.

7.3.3 Weak

The weak attribute for procedures is used just as it is for variables.

7.3.4 External

The external attribute for procedures is used just as it is for variables.

```
package foo
{
    proc foobar(): : external;
};
```

When the external attribute is used with a procedure, the name that will be generated in the symbol table will be the unqualified name. In other words, the name will be that as it appears in the proc statement without the preface of the package name that it may be in.

External with a parameter for a procedure.

```
package cpu
{
    proc getcpuid(result:  @[]_uint32): : external(0x1fff1ff1);
};
```

Here the code for ‘getcpuid’ is known to be located at address 0x1fff1ff1.

7.3.5 Segment

The section attribute for procedures is used in a similar manner as it is for variables (see section 4.3.2). The format of section names is linker-specific.

Example:

```
proc add(a: _int): _int: section(".simple_functions")
{ ... }
```

7.4 Forward Procedures

In some cases, a procedure must be called before it is defined (as with mutually recursive procedures). A procedure may be declared forward of its use but before its definition by omitting the body of the procedure and replacing it with a single semicolon:

```
proc sumdiff(a:_int, b:_int):_int, _int;
```

7.5 Procedure References

A procedure reference is a data type which contains a pointer to a procedure with a given “signature”. The signature is the parameter list together with the returned value list. This is prefixed by ‘`@_`’.

```
type pref: @_(a:_int, b:_int):_int, _int;
var p: pref;
p = sumdiff;
```

References to bound procedures have the same syntax as those to normal procedures. The first parameter is that of the bound parameter.

```
type bpref: @_(a:@rec, x:_uint);
var bp: bpref;
p = r.init;
```

8 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.

Examples:

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

8.1 Package Continuation

Packages can be re-opened to extend the contents of the package. This is known as *package continuation*. An example is:

```
package foo
{
    var x: _uint32;
    var y: _uint32;
}

// code outside package foo

package foo            // continue package foo
{
    var z: _uint32;
}          // package foo now has variable x,y,z
```

A package can be imported into another package. This is not *package continuation*, but rather the definition of a new package containing a nested package.

Supposed a file named "pkg_A.esl" contains:

```
package pkg_A
{
    var v1: _uint32;
    var v2: _uint32;
};
```

In a different file, is the following code:

```
package pkg_B
{
    import pkg_A;

    var v1: _uint32;
    var v2: _uint16;
    proc x()
    {
         v1 = 33;
         v2 = 44;
         pkg_A.v1 = 55;
         pkg_B.v2 = 66:
    }
};
```

9 Programs and Scope

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

9.1 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.

9.2 Conditional Compilation

9.3 Compilation Style

Most programmers have adopted an incremental compile approach to program development. In other words, one compiles numerous source files, one at a time, and then links the resultant object files together to form a single, executable file. In the ESL / LLVM ecosystem, this approach is not necessary nor desirable.

Program development with ESL is best done with a 'compile all at once' approach. Simply put, all the

files associated with a particular development are compiled at once. This approach to compilation gives the ESL front-end, and to a greater extent the LLVM back-end, a complete view of all the software in a development project. By using this approach it allows ESL and the LLVM tools to maximize the optimization of the source code in the development project. There can be great advantages to code size and execution efficiency:

1. Procedures (non-global) that are not called do not generate any code.
2. Variables and constants (non-global) are not allocated.
3. Opportunties to in-line are maximized.

The following shows how five files in a development are combined and compiled all at once, using ESL. The main file in this development is: top_level.esl. It imports the four other files in the development into top_level.esl.

=== top_level.esl

```
import file1;
import file2;
import file3;
import file4;

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

10 Language Syntax Summary

10.1 Notation

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

10.2 Program Syntax

program = *prog-stmt* { *prog-stmt*}
prog-stmt = *package* | *procedure* |
import-stmt | *decl_stmt* | *if-stmt* | *error-stmt*
import-stmt = "`import`" *package-specifier* "`;`"
error-stmt = "`error`" *string* "`;`"

10.3 Package Syntax

package = "package" *ident* "{" *pkg-stmt* { *pkg-stmt* } "}"
pkg_stmt = *package* | *procedure* |
import-stmt | *decl_stmt* | *if-stmt* | *error-stmt*

10.4 Procedure Syntax

procedure = "proc" ["(" *formal* ")"] *ident signature* [":" *proc-attr*]
(";" | "{" { *stmt* } "}")
signature = "(" [*parm-list*] ")" [":" [*retv-list*]]
parm-list = *formal* { "," *formal* }
formal = *ident* ":" *type-def*
retv-list = *type-def* { , *type-def* }
stmt = "{" *stmt-list* "}" |
decl_stmt |
asgn-stmt | *if-stmt* | *loop-stmt* | *while-stmt* | *exit-stmt* | *return-stmt*

10.5 Declaration Statement Syntax

decl-stmt = *alias-stmt* | *type-stmt* | *var-stmt* | *const-stmt*
alias-stmt = "alias" *alias-list* ";"
alias-list = *alias-spec* { "," *alias-spec* }
alias-spec = *qname* "as" (*ident* | *pkg-name* "." *name*)
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"

10.6 Executable Statement Syntax

asgn-stmt = *lhs-list* "=" *expr-list* ";"
lhs "+=" *expr* ";" | *lhs* "-=" *expr* ";"
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*]
loop-stmt = "loop" *stmt*
while-stmt = "while" *bool-expr* "do" *stmt*
exit-stmt = "exit" *bool-expr* ["then" *stmt*]
return-stmt = "return" expr-list ";"
asm-stmt = "asm" *string* ["," *string* { "," *expr* }] ";"

10.7 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 = type-query | number | *qname* | "(" *expr* ")"
type-query = [*type-name* | *var-name*] "?" *type-qattr*
type-qattr = "min" | "max" | "bits" | "size" | "align" | "len"

11 Library Support

memcpy

memcmp*N*

memalloc

memfree

12 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 who have 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”.

12.1 Preprocessor

ESL does not have a separate pre-processor language. There are no features such as “token-pasteing.”

12.1.1 Include

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

12.1.2 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. The ESL if statement deals in units of complete statements, not tokens.

12.2 Declarations

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

12.3 Typedefs

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

12.4 Enumerations

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

12.5 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
```

12.6 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;
```

12.7 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 pointers and the specification of a string as "`[]_byte`" must be explicit:

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

12.8 Statements

12.8.1 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 as in the following:

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

Alternatively, as mentioned in page 20, the following could be used:

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

12.8.2 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.

12.8.3 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.

12.8.4 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;
```

13 Building and Using the ESL Compiler

13.1 Prerequisites

Before one builds the ESL compiler, two packages must be install: the LLVM Compiler Infrastructure and the GNU Compiler Collection (only the assembler and libc are used) for the target platform. The building and/or installation of both of these packages are described on their respective web pages and will not be repeated here.

13.2 Building the ESL compiler

13.2.1 Getting the source

After the LLVM and GNU collections have been installed, the source to ESL can be downloaded from the Google Project Hosting site: http://code.google.com/p/esl. The user has two alternative ways to get the source to ESL: either as a tarball or from the svn depository. The tarballs are available under the 'Downloads' tab, while instructions for checking out the svn despository are under the 'Source' tab.

13.2.2 Installing LLVM

13.2.3 Building the compiler

Once the sources to ESL have been obtained, the next step is to actually build and install the compiler. The top level directory of the distribution should contain two files: LICENSE and README, and two directories: src and doc. The building and installing of the compiler will occur in the src directory and all of the following directions will assume that the user has that directory as the current working directory.

Before starting the build process, a line must be changed in the shell script file: llvm. The line that set the shell variable LLVM must be changed to the directory where the LLVM tools, i.e., llc, llvm-as, etc., were installed into. The line looks like the following:

```
LLVM=$HOME/work/llvm/Release+Asserts/bin          # !!FIX THIS!!
```

Once this line has been changed, building can proceed.

After the line in LLVM has been changed, the make(1) variables: CFLAGS, ARCH, and GCC should be examined and possibly adjusted for the target platform. The file: Makefile is the makefile used to build the ESL compiler and contains the definitions of the aforementioned variables. Once these two files have been examined and optionally changed, the ESL compiler can be built.

To build the ESL compiler, the user invokes 'make'. The make'ing of the ESL will proceed and produce four (4) versions of the compile: eslc0, eslc1, eslc2, and eslc3. The user should select the 'eslc3' version as the ESL compiler to be used for compilation. This binary can be put into the user's choice of a directory and making sure the directory defined in the user's shell path variable.

13.3 ESL Command Line Options

```
eslc  [-D[asftm]] [-mtarget] [-Idir] [-ofile] [-O] [-M] [-g] files
```

13.3.1 -D[asftm]

Control debug output:

-a Dump the abstract syntax tree

-s Dump the symbol table

-f List all source files

-t tests

-m methods

[JN: maybe some of these shouldn't be exposed in the document]

13.3.2 -m*target*

Specify the target architecture.

ESL can only support what LLVM has implemented as target backends.

Current targets

x86

x86_64

x86_64-darwin

msp430

cortex-m3

arm920t

ppc32

ppc64

s390x

systemz

mips

13.3.3 -I*dir*

Add a directory to the set of directories search for import files. They are searched in the order specified.

13.3.4 -o*file*

13.3.5 -O

13.3.6

13.3.7 -M

Instead of compiling, just output a list of files suitable for makefile dependencies.

13.3.8 -g

Generate debugging information for use by run-time debuggers such as gdb. [This doesn't work yet.]

13.4 Compilation Flow

eslc – the compiler “front-end”

llvm-as – converts llvm textual format to llvm byte-code

opt – architecture independent optimization

llc – the compiler “back-end” that generates assembly language [Future versions of LLVM may generate object code and eliminate the need for a target assembler.

gas – target assemble

ld – linker used to link in library routine

Alphabetical Index