SEAL Language Reference Manual

COMS W4115 Spring 2014

Ten-Seng Guh tg2458

Contents

1. Introduction 3
2. Lexical Conventions 3
   2.1. Comments 3
   2.2. Tokens 3
   2.2.1. Identifiers 4
   2.2.2. Keywords 4
   2.2.3. Constants 4
      2.2.3.1. Integer Constants 4
      2.2.3.2. Floating Point Constants 5
      2.2.3.3. Character Constants 5
   2.2.4. String Literals 5
   2.2.5. Operators 5
   2.2.6. Separators 6
3. Expressions 7
   3.1. Function Calls 9
4. Objects 9

4.1. The Object Type 9
4.1.1. Labels 10
4.2. The Fundamental Types 10
4.3. The String Type 11
4.4. The Array Type 11
4.5. The Enumeration Type 11
4.6. Thread Type 11
4.7. Lock Type 12
4.8. Types 12
5. Statements 12
5.1. Expression Statements 13
5.2. Compound Statements 13
5.3. If Statements 13
5.4. Iteration Statements 13
6. Declarations 13
6.1. Array Declaration 14
6.2. Enum Declaration 14
6.3. Function Declaration 14
6.4. Thread Declaration 14
6.5. Interrupt Declaration 15
6.6. Type Declaration 15
7. Program 15
7.1. Definitions 15
7.2. Setup 16
7.3. Main 16
7.4. Inclusion of Other Files 16
8. References 17

1. Introduction

This is a language reference manual for Simple Embedded Avionics Language (SEAL). SEAL is a programming language that simplifies the tasks most commonly found in embedded systems development, particularly avionics software development. It borrows a lot of its syntax from C, but also borrows characteristics from the object-oriented nature of languages such as Ada, Java and C#. Still other aspects of the language are unique only to itself. The goal of SEAL is to describe low-level tasks in a high-level way. The format of this manual is largely based on Appendix A of the C Programming Language, 2nd Edition by Kernighan and Ritchie.

2. Lexical Conventions

A SEAL program shall consist of three parts: the definitions, the set-up, and the Main. These parts can all be in one file, or they can span multiple files. Definitions refer to the defining of types and their functions, as well as stand-alone functions. Set-up refers to the assigning and mapping of registers, the declaration and initialization of hardware resources such as timers, I/O ports, interrupts, and the declaring and latching interrupt service routines to their respective interrupts. Main refers to the block of code that begins execution in the main loop of the program, once everything else is set up. When the program counter is initialized, it shall point to the entry point of Main.

2.1.Comments

SEAL shall use either “/*” and “*/” or “//” and newline to enclose a comment. Comments shall be stripped out by the scanner. Some examples below:

```
/* this is a valid comment */

// this is also a valid comment

/* this is not a valid comment //

/* this is not a valid comment either */ */
```

2.2. Tokens

A SEAL program shall be comprised of tokens. There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and separators. White space shall be used only to tokenize the program. A token shall constitute the longest string of characters. For example, if the scanner scans “Int”, it will not stop and will attempt to find “Interrupt”. If the string does not follow with “errupt”, only then will it tokenize the string as an “Int”.

2.2.1. Identifiers

An identifier (id) shall be composed of a letter, optionally followed by letters and/or digits. An underscore counts as a letter. Identifiers are case sensitive. Besides for keywords, identifiers shall also be used for naming variables, types, and functions. Some examples below:

```
a      //valid token

A      //valid token (different than "a")

_      //valid token

_123   //valid token

123    //invalid token

@      //invalid token
```

2.2.2. Keywords

The following identifiers shall be keywords reserved for use by SEAL and may not be used otherwise:

Bool	For	Long	Thread
Bit	If	Main	True
Byte	Include	Object	Type
Double	Int	Return	Uint
Else	Interrupt	Short	Ulong
Float	Lock	String	Ushort
False			While

2.2.3. Constants

A constant shall be a representation of a value. There shall be three types of constants: integer constant, floating point constant, and character constant.

2.2.3.1. Integer Constants

An integer constant may be in decimal form, binary, octal, or hex form. If in binary form, it shall be suffixed with "b" and contain only 0s or 1s. If in octal form, it shall be suffixed with "o" and can only contain 0s through 7s. If in hex form, it shall be prefixed with "0x" or suffixed with "H"/"h" and shall use the traditional hex characters. An integer constant may be positive or negative. If negative and in decimal form, it shall be prefixed with a "-"-sign and cannot be assigned to Uint or Ulong. Integer constants shall be assignable to Int and Long. Below are examples:

```
00111b   // valid integer constant

371o     //valid integer constant
```

```
01234     //valid integer constant

0xABCD    //valid integer constant

AcDdh     //valid integer constant

0xabcdH   //invalid integer constant

Ulong i = -1234;      //invalid assignment
```

2.2.3.2. Floating Point Constants

A floating point constant shall consist of an integer part, a decimal point, a fraction, followed by an optional "E"/"e" with an exponent part. This is slightly stricter than C floating point constant rules. If negative, it shall be prefixed with a '-'-sign. Below are examples:

```
1.0       //valid

1.34e4    //valid

1.        //invalid

1e4       //invalid
```

2.2.3.3. Character Constants

A character constant shall consist of a '\' followed by any of the following characters:

Character	Meaning
n	Newline
t	Tab
\	Backslash
‘	Single Quote
“	Double Quote
0	Null byte

2.2.4. String Literals

A string literal shall consist of a sequence of characters enclosed by double quotes. A string literal containing 0 characters shall be equivalent to the Null byte.

2.2.5. Operators

An operator shall allow an operation to be performed between one or two expressions. The following are two tables of operators, the first one for unary.

Operator	Purpose	Precedence
++	Increment	1st
--	Decrement	2nd

!	Negation	3rd
~	One’s complement	3rd
-	Negative	3rd

Operator	Purpose	Associativity	Precedence
*	Multiplication	Left	1st
/	Division	Left	1st
%	Modulus	Left	1st
+	Addition	Left	2nd
-	Subtraction	Left	2nd
<<	Bit Shift Left	Left	3rd
>>	Bit Shift Right	Left	3rd
<	Less Than	Left	4th
>	Greater Than	Left	4th
<=	Less Than or Equal to	Left	4th
>=	Greater Than or Equal to	Left	4th
==	Equal to	Left	5th
!=	Not equal to	Left	5th
^	Bitwise XOR	Left	6th
&	Bitwise AND	Left	6th
|	Bitwise OR	Left	6th
&&	Logical AND	Left	7th
||	Logical OR	Left	7th
= *= /= %= += -= <<= >>= &= ^= |=	Assignment, or assignment after operating on the right operand using the operator preceding =	Right	8th
,	Comma separator	Left	9th

2.2.6. Separators

A separator shall be one of the following:

Separator	Purpose
;	Ending statement
{ … }	Type, Function, thread, or interrupt service routine definition
,	Separating arguments
.	Label access of an object

3. Expressions

An expression (*exp*) shall be composed of identifiers, constants, or string literals, and can be enclosed in parentheses. An integer-expression (*int*) shall denote an expression composed of an integer constant. A floating-point-expression (*flt*) shall denote an expression composed of a floating-point constant. A variable-name-expression (*var*) shall denote an expression composed of identifiers and optionally the "." separator. A string-expression (*str*) shall denote an expression composed of string literals. A numerical-expression (*num*) shall denote an expression that is either an integer-expression or floating-point expression. A non-string-expression (*nse*) shall denote an expression that is not a string-expression. A variable-integer-expression (*vfe*) shall denote an expression that is either a variable-name-expression or an integer-expression. These basic expressions can then act as operands for use with operators to become more complex expressions (*expr*).

Throughout the rest of the document, there will be symbols used to describe the grammar. The '|' means "or", '?' means "zero or one", and '*' means "zero or more". The following are the basic rules for the expressions mentioned above.

var: id | id'.'var
num: int | flt
exp: var | num | str
nse: var | num
vfe: var | int

Below are the various expressions for each grammar. Each row denotes a level of precedence.

expr: num | exp |nse | vfe
expr: inc
expr: dec
expr: not | inv | neg
expr: mult | div | mod
expr: add | sub
expr: bsl | bsr
expr: lth | gth | lte | gte
expr: equ | neq
expr: xor | and | or
expr: andl | orl
expr: asn

expr: com
expr: ‘(‘ expr ‘)’

The table below contains the expression rules for the operators. They are all valid expressions:

Operator	Expression rule
++	*inc: vfe“++”*
--	*dec: vfe“--”*
!	*not: ‘!’var*
~	*inv: ‘~’vfe*
- (unary)	*neg: ‘-’num*
*	*mult: nse*
	mult: mult ‘’ mult*
/	*div: nse*
	div: div ‘/’ div
%	*mod: int*
	mod: mod ‘%’ mod
+	*add: exp*
	add: add ‘+’ add
- (binary)	*sub: nse*
	sub: sub ‘–’ sub
<<	*bsl: vfe*
	bsl: bsl “<<” bsl
>>	*bsr: vfe*
	bsr: bsr “>>” bsr
<	*lth: nse*
	lth: lth “<” lth
>	*gth: nse*
	gth: gth “>” gth
<=	*lte: nse*
	lte: lte “<=” lte
>=	*gte: nse*
	gte “>=” gte
==	*equ: vfe*
	equ: equ “==” equ
!=	*neq: vfe*
	neq: neq “!=” neq
^	*xor: vfe*
	xor: xor ‘^’ xor
&	*and: vfe*
	and: and ‘&’ and
|	*or: vfe*

	or: or '|' or
&&	*andl: vfe* *andl: andl "&&" andl*
||	*orl: vfe* *orl: orl "||" orl*
= *= /= %= += -= <<= >>= &= ^= |=	*asn: var '=' exp | var "*=" exp | var "/=" exp* *asn: var "%=" exp | var "+=" exp | var "-=" exp* *asn: var "<<=" exp | var ">>=" exp* *asn: var "&=" exp | var "^=" exp | var "|=" exp*
,	*com: exp* *com: com ',' com*

3.1.Function Calls

A function call (*fun*) is a postfix expression. It shall consist of a variable-expression, followed by a pair of parentheses containing an optional list of expressions as arguments. The following is the rule for function calls.

fun: var"()"
fun: var'(' expr (',' expr)')'*

4. Objects

In SEAL, an object is a piece of memory that the programmer can use. It is a way to make a piece of data useful and accessible to the programmer. It shall be identified with a variable-name-expression. SEAL is object-oriented, so all variables are objects. In SEAL, Type can be used interchangeably with the object-oriented concept of a class. The `Type` keyword shall define a class describing the object.

4.1.The Object Type

The most fundamental type, in which all other types are derived from, shall be the `Object` type. All Objects shall have an address or range of addresses associated with it.

Objects shall be accessed by its address. Thus:

```
AddTwo(Int a){    a += 2;}
…
Int a = 3;
AddTwo(a);
Print(a);    //this will print 5
```

4.1.1. Labels

An Object shall have Labels. Labels allow viewing the object through various other aspects. Some Labels are global and static functions compiled in. Others act as monikers for different aspects of the object. Yet others are objects themselves. To access an object's label, the label shall be prefixed with the Object name followed by a '.'. All objects shall have the following Labels:

Label Name	**Return Type**	**Purpose**
Address	Int	Gets or sets the address of the object.
Swap()	Object	Gets the swapped contents of the object. Good for sending/receiving contents from a system with an endianness that is opposite of yours.
BigEndian()	Object	Gets the big endian representation of the object. If you are already on a Big Endian system, this representation will be identical to object.
LittleEndian()	Object	Gets the little endian representation of the object. If you are already on a Little Endian system, this representation will be identical to object.
Lock	Object	Gets or sets the lock that the object is using. Can be null.
String()	String	Gets the string representation of object. If object is already a String, this representation will be identical to object.
Length	Int	Gets the length of the object.

4.2.The Fundamental Types

There shall be eight fundamental Types. They are meant to store numerical-expressions and so represent floating point numbers as well as integers. The following are the eight fundamental types.

Type	**Description**
`Bit`	Represents one bit of data. This type has an additional label `Offset` to track where in the `Address` the `Bit` variable is located.
`Byte`	Represents one byte of data, 8-bits unsigned.
`Bool`	Represents a Boolean, can be either `True` or `False`.
`UShort/Short`	Represents an unsigned/signed 16-bit integer.
`Uint/Int`	Represents an unsigned/signed 32-bit integer.
`Ulong/Long`	Represents an unsigned/signed 64-bit integer.

`Float`	Represents a 32-bit floating point number, unsigned by default.
`Double`	Represents a 64-bit floating point number, unsigned by default.

4.3. The String Type

The `String` type shall be composed of string-expressions. The addition "+" operator shall be available for String, allowing concatenation.

4.4. The Array Type

Arrays shall be a 1-dimensional representation of a collection of objects of the same Type. Arrays shall be fixed size, indexable, and mutable. Arrays shall allow access to an item via an index, represented by an unsigned integer-expression enclosed by '[' and ']'. In addition to the standard Object Labels, they shall have the following powerful Labels:

Label	**Return Type**	**Description**
Apply(Function())	Object	Iterates through the array to apply the given function to each element in the array.
Sort()	Object	Sorts array.
Reverse()	Object	Sorts array in reverse order.

4.5. The Enumeration Type

Enumerations shall be a list of names for bytes. Instead of accessing an item via an index, enumerations shall allow access to an item via its name. The following is an example:

```
Enum Seasons = {Spring, Summer, Fall, Winter};
Print(Seasons.Length());    //will print "4"
Seasons a = Fall;
```

4.6. Thread Type

The `Thread` Type shall allow code to run autonomously separate from the Main loop. Anything enclosed in a thread block shall execute in its own stack and address space. Shared variables among threads shall be protected via the `lock` label. A `Thread` shall not return anything. A `Thread` shall not receive any arguments. Threads shall have the following Labels:

Label	**Return Type**	**Description**
Priority	Byte	The priority in which this thread should run.
Go()	None	Kicks off thread execution.
Join()	None	Waits for other thread(s) to finish before kicking off execution.
Pause()	None	Halts thread execution. Releases any locks held by the thread.
Unpause()	None	Resumes thread execution. Attempts to reacquire any locks that were held by the thread.
Stop()	None	Terminates thread execution.

4.7. Lock Type

The `Lock` allows various Threads to share variables safely. Locks effectively allow code to become re-entrant. The programmer does not have to worry about explicitly writing code for initializing, acquiring or releasing locks; this shall all be handled by the Threads that access the shared variables. The Lock shall have the following Labels:

Label	**Return Type**	**Description**
Acquire()	Bool	Calling thread attempts to acquire the lock.
Release()	None	Calling thread releases the lock.

4.8. Types

SEAL shall allow new Types to be created out of current Types. All new Types shall inherit from the Object type, thereby endowing them with the same Labels. Types shall contain variable and/or function definitions. SEAL shall not allow new label creation. SEAL shall not allow multiple inheritance.

5. Statements

Statements (*st*) are sequences of code that is executed for its effect. There are four types of statements: expression statements (*exst*), compound statements (*cpst*), if statements (*ifst*), and iteration statements (*itst*).

st: exst | cpst | ifst | itst

5.1. Expression Statements

An expression statement shall be merely an expression followed by a ‘;’. An expression may be empty. All side effects of the expression shall be completed before the next statement is executed.

exst: expr? ‘;’

5.2.Compound Statements

A compound statement shall be multiple statements enclosed by ‘{’ and ‘}’. Within it can be declarations and statements. A compound statement shall optionally end with a Return keyword followed by an expression, if part of a Function declaration (see 6.3). Below, the “*” means “zero or more”.

cpst: ‘{’ decl st* (“Return” expr’;’)? ‘}’*

5.3. If Statements

An if statement shall allow choice of flow of control. The expression enclosed in the parentheses is evaluated, and if it equals one, statement shall be executed. If it equals zero and there’s an “Else” followed by another statement, that statement shall be executed.

ifst: “If (” expr ‘)’ st (“Else” expr)?

5.4. Iteration Statements

Iteration statements shall specify looping. In the While (*while*) statement, an expression shall be repeatedly evaluated and statement repeatedly executed as long as the evaluated expression’s value remains equal to one. In the For (*for*) statement, there shall be up to three expressions. The first expression shall be evaluated once. The second expression shall be repeatedly evaluated and statement repeatedly executed as long as the evaluated expression’s value remains equal to one. Lastly, after each execution of statement, the third expression will be evaluated.

itst: while | for
while: “While(” expr ‘)’ st
for: “For(”exst exst exst “)” st

6. Declarations

Declaration (*decl*) is the method of creating a unique identifier for an object, function, thread, or interrupt service routine. For objects, a declaration shall start with the Type (type) name followed by the variable-name-expression and ‘;’, or an assignment before the

‘;’. The declared object shall be public and global if not enclosed in ‘{’ and ‘}’, otherwise it shall be temporary and private.

decl: type var‘;’
decl: type asn‘;’
decl: ‘{‘ decl ‘}’*

6.1. Array Declaration

An array declaration is similar to an object declaration except the initial size must be included, enclosed within an ‘[’ and ‘]’. An array declaration shall not allow an unknown or empty size. The array declaration shall allow an assignment of any subset of the elements.

decl: type var‘[’ int “];”
decl: type var‘[’ int “]={” exp (‘,’ exp) “};”*

6.2. Enum Declaration

An enum declaration (*edec*) shall allow only variable-name-expressions assigned to its elements. An enum declaration shall start with the keyword “Enum”.

edcl: “Enum” var “={” var (‘,’var) “};”*

6.3.Function Declaration

A function shall contain a collection of statements to be executed upon being called, enclosed by ‘{’ and ‘}’. A function shall receive one or more arguments. A function shall return something or nothing. A function shall return a type or no type at all. The return type shall be inferred from the return statement or the lack thereof. Recursive functions shall be allowed.

A function declaration (*fdcl*) is similar in appearance to a function call, except it shall contain a compound statement following the parentheses.

fdcl: var(“()” | ‘(’ expr (‘,’ expr)‘)’) cpst ‘;’*

6.4. Thread Declaration

A Thread declaration (*tdcl*) shall be the equivalent to a Function declaration except with the “Thread” keyword in front, and no option for passing arguments in.

tdcl: “Thread” var “()” cpst ‘;’

6.5. Interrupt Declaration

The keyword `Interrupt` shall denote a function specifically used for handling interrupts. An `Interrupt` shall not return anything. An `Interrupt` shall not receive any arguments. An `Interrupt` cannot be called by the user directly. Although not a Type, an `Interrupt` has one label associated with it. An `Interrupt` shall use the Address label to hook it into the interrupt vector table.

An Interrupt declaration (*idcl*) shall be the equivalent to a Thread declaration except with the “Interrupt” keyword instead of “Thread”.

idcl: “Interrupt” var “()” cpst ‘;’

6.6. Type Declaration

A Type declaration shall include declarations for any enum, arrays, objects, and functions as part of the Type. A Type declaration shall not allow Thread or Interrupt declarations within it. A Type declaration shall start with the Type keyword, followed by a variable-name-expression for its name, followed by an option “:” and Type name for the Type that it inherits from, followed by expressions enclosed in ‘{’ and ‘}’, followed by a ‘;’.

tdec: “Type” var (‘:’var)? ‘{‘ edecl decl* fdec* ‘}’*

7. Program

A program shall be comprised of up to three sections: the Definitions section, the Setup section, and the Main section.

7.1. Definitions

Here is where functions, threads, interrupt service routines, and registers shall be declared. Only declarations are allowed in Definitions. Below is an example of a valid Definitions section:

```
Lock lock;
SerialPort sp;
Bool makeSound;
Interrupt SoundOff
{
   counter++;
   if (counter == 4000)
   {
         counter = 0;
         makeSound = TRUE;
```

```
      }
   }
   Float vm1, vm2;
```

7.2. Setup

Here is where registers, interrupt service routines, and hardware resources shall be initialized. Only assignment expressions and function calls are allowed in Setup. Below is an example of a valid Setup section:

```
   sp.Lock = lock;   //this serial port shall be shared between
two threads, so lock it
   sp.Configure("COM1", 115200, 1, 0, 0);
   makeSound = FALSE
   makeSound.Address = 0x4A;   //speaker byte
   SoundOff.Address = 0x3F;    //hooking ISR to timer interrupt
   at 0x3F
   vm1.Address = 0x400;
   vm2.Address = 0x410;
```

7.3. Main

The `Main` shall be where the entry point of the program resides. The `Main` function shall not return anything. The `Main` function shall not receive any arguments.

7.4. Inclusion of Other Files

Inclusion of other source files containing programs shall be allowed via the `Include` keyword. Include can only appear at the top of the file. Included files cannot contain a `Main` section.

8. References

1. “Interrupt Functions” *Cx51’s user’s guide*. Accessed March 2014 http://www.keil.com/support/man/docs/c51/c51_le_interruptfuncs.htm
2. Edwards, Stephen A. "Scanning and Parsing" Programming Languages and Translators. Mudd 535, Columbia University. Fall 2010. Lecture.
3. Kernighan, Brian W., and Dennis M. Ritchie. "Appendix A." The C Programming Language. Englewood Cliffs, NJ: Prentice Hall, 1988. 191-239. Print.