Contents

1. Introduction .......................................................................................................................... 4
2. Lexical Conventions ............................................................................................................. 4
   2.1. Comments .......................................................................................................................... 4
   2.2. Tokens ............................................................................................................................... 4
   2.2.1. Identifiers ..................................................................................................................... 4
   2.2.2. Keywords .................................................................................................................... 5
   2.2.3. Constants ................................................................................................................... 5
      2.2.3.1. Integer Constants ................................................................................................ 5
      2.2.3.2. Floating Point Constants .................................................................................... 5
      2.2.3.3. Character Constants .......................................................................................... 6
   2.2.4. String Literals .............................................................................................................. 6
   2.2.5. Operators ................................................................................................................... 6
   2.2.6. Separators .................................................................................................................. 7
3. Expressions .......................................................................................................................... 7
   3.1. Function Calls ............................................................................................................... 9
4. Objects ..................................................................................................................................... 9
   4.1. The Object Type ............................................................................................................. 9
   4.1.1. Labels ......................................................................................................................... 9
   4.2. The Fundamental Types ............................................................................................... 10
   4.3. The String Type ........................................................................................................... 10
   4.4. The Array Type ............................................................................................................ 10
   4.5. Thread Type ................................................................................................................. 10
4.6. Lock Type ......................................................................................................................... 11
4.7. Types ................................................................................................................................. 11
5. Statements ............................................................................................................................. 11
  5.1. Expression Statements .................................................................................................... 11
  5.2. Compound Statements .................................................................................................... 11
  5.3. If Statements .................................................................................................................. 12
  5.4. Iteration Statements ....................................................................................................... 12
6. Declarations ........................................................................................................................ 12
  6.1. Array Declaration ........................................................................................................... 12
  6.2. Function Declaration ..................................................................................................... 12
  6.3. Thread Declaration ....................................................................................................... 13
  6.4. Interrupt Declaration .................................................................................................... 13
  6.5. Type Declaration ............................................................................................................ 14
7. Program ................................................................................................................................ 14
  7.1. Definitions ..................................................................................................................... 14
  7.2. Main ................................................................................................................................ 14
  7.3. Inclusion of Other Files ................................................................................................. 14
8. Project Plan ........................................................................................................................... 15
  8.1. Timeline .......................................................................................................................... 15
     Architecture ....................................................................................................................... 17
  8.2. Overview ......................................................................................................................... 17
  8.3. Scanning, Parsing, AST .................................................................................................. 18
  8.4. SEAL Library .................................................................................................................. 18
9. Test Plan ................................................................................................................................ 18
  9.1. Test Case Selection ........................................................................................................ 18
  9.2. Demo Programs ............................................................................................................. 18
10. Lessons Learned .................................................................................................................. 22
11. Appendix ............................................................................................................................. 22
  11.1. AST.ml .......................................................................................................................... 22
  11.2. Scanner.mll .................................................................................................................... 25
  11.3. Compiler.mll .................................................................................................................. 27
  11.4. parser.mly ..................................................................................................................... 34
  11.5. seal.ml .......................................................................................................................... 39
  11.6. Makefile ........................................................................................................................ 40
  11.7. Seal.sh ........................................................................................................................... 41
11.8. SEAL_Array.h .......................................................... 41
11.9. SEAL_Array.c .......................................................... 42
11.10. SEAL_Lock.h ........................................................... 43
11.11. SEAL_Lock.c ........................................................... 43
11.12. SEAL_Signal.h .......................................................... 44
11.13. SEAL_Signal.c .......................................................... 44
11.14. SEAL_Thread.h .......................................................... 45
11.15. SEAL_Thread.c .......................................................... 46
11.16. SEAL_Util.h ............................................................ 47
11.17. SEAL_Util.c ............................................................ 47
11.18. Test-swap.sl ............................................................ 48
11.20. Test-semanticerror.sl .................................................. 49
11.21. Test-thread.sl .......................................................... 49
12. References ........................................................................ 52
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 interrupt signals. 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”)
2.2.2. Keywords
The following identifiers shall be keywords reserved for use by SEAL and may not be used otherwise. They all start with a capital letter:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Int</th>
<th>Double</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>Interrupt</td>
<td>Else</td>
<td>Type</td>
</tr>
<tr>
<td>Byte</td>
<td>Lock</td>
<td>String</td>
<td>If</td>
</tr>
<tr>
<td>Address</td>
<td>Swap</td>
<td>Return</td>
<td>While</td>
</tr>
<tr>
<td>For</td>
<td>Source</td>
<td>Map</td>
<td></td>
</tr>
</tbody>
</table>

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
2.2.3.3. Character Constants
A character constant shall consist of a `\` followed by any of the following characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Newline</td>
</tr>
<tr>
<td>t</td>
<td>Tab</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>'</td>
<td>Single Quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double Quote</td>
</tr>
<tr>
<td>0</td>
<td>Null byte</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Increment</td>
<td>Left</td>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>Left</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
</tr>
<tr>
<td>!</td>
<td>Negation</td>
<td>Right</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
</tr>
<tr>
<td>~</td>
<td>One’s complement</td>
<td>Right</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
</tr>
<tr>
<td>-</td>
<td>Negative</td>
<td>Right</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Left</td>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Left</td>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>Left</td>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>Left</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Left</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Bit Shift Left</td>
<td>Left</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Bit Shift Right</td>
<td>Left</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
<td>Left</td>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
<td>Left</td>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal to</td>
<td>Left</td>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal to</td>
<td>Left</td>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>Left</td>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>Left</td>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>Left</td>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Left</td>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise OR</td>
<td>Left</td>
</tr>
</tbody>
</table>
&& Logical AND Left 7th
|| Logical OR Left 7th
= Assignment Right 8th
, Comma separator Left 9th

2.2.6. Separators
A separator shall be one of the following:

<table>
<thead>
<tr>
<th>Separator</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Ending statement</td>
</tr>
<tr>
<td>{ ... }</td>
<td>Type, Function, thread, or interrupt service routine definition</td>
</tr>
<tr>
<td>,</td>
<td>Separating arguments</td>
</tr>
<tr>
<td>.</td>
<td>Label access of an object</td>
</tr>
</tbody>
</table>

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.

erxp: num | exp | nse | vfe
erxp: inc
erxp: dec
erxp: 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: asnl
expr: com
expr: ‘(‘ expr ‘)’

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Expression rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>inc: exp”++”</td>
</tr>
<tr>
<td>--</td>
<td>dec: exp”--”</td>
</tr>
<tr>
<td>!</td>
<td>not: ‘!exp</td>
</tr>
<tr>
<td>~</td>
<td>inv: ¬exp</td>
</tr>
<tr>
<td>- (unary)</td>
<td>neg: ‘-exp</td>
</tr>
<tr>
<td>*</td>
<td>mult: exp</td>
</tr>
<tr>
<td></td>
<td>mult: exp ‘*’ exp</td>
</tr>
<tr>
<td>/</td>
<td>div: nse</td>
</tr>
<tr>
<td></td>
<td>div: div ‘/’ div</td>
</tr>
<tr>
<td>%</td>
<td>mod: int</td>
</tr>
<tr>
<td></td>
<td>mod: mod ‘%’ mod</td>
</tr>
<tr>
<td>+</td>
<td>add: exp</td>
</tr>
<tr>
<td></td>
<td>add: add ‘+’ add</td>
</tr>
<tr>
<td>- (binary)</td>
<td>sub: nse</td>
</tr>
<tr>
<td></td>
<td>sub: sub ‘-’ sub</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bsl: vfe</td>
</tr>
<tr>
<td></td>
<td>bsl: bsl “&lt;&lt;” bsl</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bsr: vfe</td>
</tr>
<tr>
<td></td>
<td>bsr: bsr “&gt;&gt;” bsr</td>
</tr>
<tr>
<td>&lt;</td>
<td>lth: nse</td>
</tr>
<tr>
<td></td>
<td>lth: lth “&lt;” lth</td>
</tr>
<tr>
<td>&gt;</td>
<td>gth: nse</td>
</tr>
<tr>
<td></td>
<td>gth: gth “&gt;” gth</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lte: nse</td>
</tr>
<tr>
<td></td>
<td>lte: lte “&lt;” lte</td>
</tr>
<tr>
<td>&gt;=</td>
<td>gte: nse</td>
</tr>
<tr>
<td></td>
<td>gte “&gt;=” gte</td>
</tr>
<tr>
<td>==</td>
<td>equ: vfe</td>
</tr>
<tr>
<td></td>
<td>equ: equ “==” equ</td>
</tr>
<tr>
<td>!=</td>
<td>neq: vfe</td>
</tr>
<tr>
<td></td>
<td>neq: neq “!=” neq</td>
</tr>
<tr>
<td>^</td>
<td>xor: vfe</td>
</tr>
<tr>
<td></td>
<td>xor: xor “^” xor</td>
</tr>
</tbody>
</table>
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.

\[
\text{fun: var}\left(\right) \\
\text{fun: var}\left(\text{expr}, \text{expr}\times\right)\left(\right)
\]

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. The scope for all objects in SEAL shall be global.

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.

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:

<table>
<thead>
<tr>
<th>Label Name</th>
<th>Return Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Int</td>
<td>Gets or sets the address of the object.</td>
</tr>
<tr>
<td>Swap()</td>
<td>Object</td>
<td>Swaps the contents of the object. Good for sending/receiving contents from a system with an endianness that is opposite of yours.</td>
</tr>
</tbody>
</table>
4.2. The Fundamental Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Represents one byte of data, 8-bits unsigned.</td>
</tr>
<tr>
<td>Int</td>
<td>Represents an signed 32-bit integer.</td>
</tr>
<tr>
<td>Double</td>
<td>Represents a 64-bit floating point number, unsigned by default.</td>
</tr>
<tr>
<td>String</td>
<td>Represents a string of characters. See below for more information.</td>
</tr>
</tbody>
</table>

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 ‘]’. Arrays are only available for the fundamental types. In addition to the standard Object Labels, they shall have the following powerful Label:

<table>
<thead>
<tr>
<th>Label</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map(Function())</td>
<td>Object</td>
<td>Iterates through the array to apply the given function to each element in the array.</td>
</tr>
</tbody>
</table>

4.5. 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. 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:

<table>
<thead>
<tr>
<th>Label</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcreate()</td>
<td>Void</td>
<td>Initializes the thread.</td>
</tr>
<tr>
<td>Go()</td>
<td>Int</td>
<td>Kicks off thread execution.</td>
</tr>
<tr>
<td>Join()</td>
<td>Int</td>
<td>Waits for other thread(s) to finish before kicking off execution.</td>
</tr>
<tr>
<td>Stop()</td>
<td>Int</td>
<td>Terminates thread.</td>
</tr>
<tr>
<td>Tdestroy()</td>
<td>Void</td>
<td>Disposes of the thread.</td>
</tr>
</tbody>
</table>
4.6. 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:

<table>
<thead>
<tr>
<th>Label</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lcreate()</td>
<td>Void</td>
<td>Initializes the lock.</td>
</tr>
<tr>
<td>Acquire()</td>
<td>Int</td>
<td>Calling thread attempts to acquire the lock.</td>
</tr>
<tr>
<td>Release()</td>
<td>Int</td>
<td>Calling thread releases the lock.</td>
</tr>
<tr>
<td>Ldestroy()</td>
<td>Void</td>
<td>Dispose of the lock.</td>
</tr>
</tbody>
</table>

4.7. Types

SEAL shall allow new Types to be composed 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 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.

\[ \text{ifst: "if (" expr ") st ("Else" expr)?} \]

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.

\[ \text{itst: while | for} \]
\[ \text{while: "While(" expr ")" st} \]
\[ \text{for: "For(" extst extst extst ")" st} \]

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, or local if declared in a function, Thread, or Interrupt service routine.

\[ \text{decl: type var';;} \]
\[ \text{decl: type asn';;} \]
\[ \text{decl: '{decl* '}'} \]

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. Only one-dimensional arrays are allowed in SEAL.

\[ \text{decl: type var[' int '];;} \]

6.2. 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.

\[
fdcl: \ \text{var}(\text{"()" | (\text{" expr (',' expr)*"})}) \ \text{cpst} \ ';\]

### 6.3. Thread Declaration

A Thread declaration \((tdcl)\) shall be the equivalent to a Function declaration except with the “Thread” keyword in front, no parenthesis and no option for passing arguments in. A Thread shall not return anything.

\[
\text{tdcl: } \ "\text{Thread}" \ \text{var} \ \text{"()"} \ \text{cpst} \ \text{";"}
\]

### 6.4. Interrupt Declaration

The keyword \(\text{Interrupt}\) shall denote a function specifically used for handling interrupts. An \(\text{Interrupt}\) shall not return anything. An \(\text{Interrupt}\) shall not receive any arguments. An \(\text{Interrupt}\) cannot be called by the user directly. Although not a Type, an \(\text{Interrupt}\) has one label associated with it. An \(\text{Interrupt}\) shall use the Source label to assign it to handle the particular interrupt signal.

<table>
<thead>
<tr>
<th>Label</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Int</td>
<td>Gets and sets which interrupt to handle.</td>
</tr>
</tbody>
</table>

The following is a list of interrupt signals supported:

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Address value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEALINT</td>
<td>1</td>
<td>Intercepts Ctrl+C key to terminate program.</td>
</tr>
<tr>
<td>SEALDFT</td>
<td>0</td>
<td>Restores to original interrupt handling or lack thereof.</td>
</tr>
</tbody>
</table>

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

\[
idcl: \ "\text{Interrupt}" \ \text{var} \ \text{"()"} \ \text{cpst} \ ';\]

An example of declaring an interrupt service routine and assigning it to listen to SEALINT would be the following:

```plaintext
Interrupt SoundOff
{
    print("you pressed control + C key!\n");
}
```
6.5. Type Declaration
A Type declaration shall include declarations for any arrays, objects, and functions as part of the Type. A Type declaration shall not allow Thread, Type, 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 expressions enclosed in ‘{’ and ‘}’, followed by a ‘;’. Types can only contain fundamental types and cannot inheritance nor be composed of other Types.

\[ \text{tdec: “Type” var ‘{’ edecl* decl fdec* ’}’} \]

7. Program
A program shall be comprised of up to two sections: the Definitions 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:

\[
\begin{align*}
\text{Lock } & \text{ lock;} \\
\text{SerialPort } & \text{ sp;} \\
\text{Bool } & \text{ makeSound;} \\
\text{Interrupt } & \text{ SoundOff} \\
\{ & \\
\text{counter++;} & \\
\text{if (counter == 4000)} & \\
\{ & \\
\text{counter = 0;} & \\
\text{makeSound = TRUE;} & \\
\} & \\
\} & \\
\text{Float } & \text{ vm1, vm2;} \\
\end{align*}
\]

7.2. Main
The Main shall be where the entry point of the program resides. The Main function shall return a 0. The Main function shall not receive any arguments. Here is also where registers, interrupt service routines, and hardware resources shall be initialized.

7.3. 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. Project Plan

Of most importance was to come up with the Abstract Syntax Tree for SEAL, which is essentially the skeleton of the language. Much care was put into designing the grammar for the language as well. The grammar was largely based and heavily influenced by the K and R grammar for C, located in Appendix A of the C programming language manual. Through fine-tuning the AST, it became clear that most of the originally intended object-oriented methodology had to be scrapped in order to uphold the goal of fast, robust, yet high-enough level access to system resources. Making SEAL truly object oriented would’ve bogged down SEAL with unnecessary layers of abstraction. The beauty of SEAL is that it truly does allow low-level access in a high-level way.

8.1. Timeline

2/10 Proposal submitted

2/20 Threading example pushed to repository.

2/25 Pthread example pushed into repository.

3/10 Uploaded sample SEAL code.

3/13 LRM submitted.

3/16 Working on scanner, parser, and AST.

3/23 pushed into repository initial scanner, parser, AST, makefile, bytecode, and seal compiler executable.

3/30 established architecture to be x86_64 Linux, gcc compiler.

4/1 looking at UNIX Signals to implement Interrupt under the hood.

4/8 created the grammar that will become the parser, AST, SAST and used to generate the bytecode, which has been chosen to be a form of 3 address code.

4/10 studying 3 address code to come up with one.

4/26 Decided to go with the environment data structures as the intermediate between the AST and the generated C code.

4/26 created library code for implementing Interrupt and Thread

5/7 created AST and the equivalent of the SAST.

5/14 Finished last touches of compiler and test cases. Vigorously tested.

The tables below show some of the preliminary design that went into the library functions.

<table>
<thead>
<tr>
<th>C constructs for the SEAL code to work</th>
</tr>
</thead>
</table>
### All interrupts are instances of this struct:

```
typedef struct {
    void *Address; //might think about taking this out
    int Source;
    void (*func)(int);
} Interrupt;
```

### All Threads are instances of this struct:

```
typedef struct {
    int priority;
    void (*func)();
    pthread_t *thread;
} Thread;
```

### SEAL code

<table>
<thead>
<tr>
<th>SEAL code</th>
<th>Intermediate bytecode</th>
<th>Final C code</th>
</tr>
</thead>
</table>
| Interrupt Ouch
  
  {  
      Printline("ouch! I got signal "+ Source);
      Source = NONE;
  }
  
  main()
  
  {  
      Ouch.Source = SIGINT;
      While (TRUE)
      {  
          Printline("Hello World!");
          Sleep(1);
      }
  }  | Ouch:

  InterruptBegin

  _c1 = "ouch! I got signal "

  _c2 = _c1 + Source

  Call Printline

  Arg _c1

  Source = 0

  InterruptEnd

  Main:

  FunctionBegin

  FunctionEnd  | Interrupt ouch;

  void ouch_func (int sig)
  
  {  
      printf("OUCH! I got signal %d\n", ouchie.Source);
      signal(sig, SIG_DFL);
      ouchie.Source = 0;
  }

  int main()
  
  {  
      ouch.func = ouch_func;
      ouch.Address = &ouch;
      ouch.Source = SIGINT;
      signal(ouchie.Source, ouchie.func);

      while (1)
      {  
          printf("Hello World\n");
          sleep(1);
      }
      return 0;
  } |
Architecture

8.2. Overview

The SEAL compiler was designed to be very mobile and flexible. It is completely platform independent. It can run on an embedded system without an OS, but it can also run on an x64 machine running Ubuntu. The front-end of the compiler is completely decoupled from the back-end.

Code input

Scanner

Tokenized input. If it has reached the Parser, congrats! The input has been successfully lexically analyzed!

Parser

Code has been successfully broken down into an Abstract Syntax Tree. If it has reached the Checker, congrats! The input is syntactically correct!

Checker

If it has reached the Code Generator, congrats! The input is semantically correct as well! The code has now been put into an array of maps, ready for generating code from.

Code Generator

At this point, the SEAL compiler will bring in platform specific code for the various library calls. The implementation is completely decoupled from the rest of the compiling process, and can easily be swapped out in support for another platform. The SEAL header files work across the board!

SEAL Library Headers

SEAL Library Implementation

SEAL Executable
8.3. Scanning, Parsing, AST
Scanning is performed first, with tokens being formed as the code gets scanned. Next comes the parsing, which takes the tokens and based on the grammar given, constructs an Abstract Syntax Tree from it. In this form, the code is now syntactically correct but not necessarily semantically. This is now checked by the aptly named Checker (resides in compile.ml), which places the code in the translation environment. Five distinct StringMap data structures comprise the environment in the SEAL compiler - one for global variables, one for the Threads and anything inside them which includes local variables but no functions, one for functions which can include local variables of their own as well as arguments, one for Interrupts and the contents inside which like Threads precludes arguments, return types, or functions, and lastly one for user defined Types which can include properties and methods but cannot contain other user defined Types. The contents of these StringMaps in this intermediate form are now good enough to generate C code directly from.

8.4. SEAL Library
The Library implementation will depend on which platform you are running SEAL code on. For this particular implementation, I chose the POSIX compliant standard for Threads, and Linux compliant standard for Signals, which can act as higher level interrupt handlers but not quite at the interrupt level. On a microcontroller such as the 8051 however, the SEAL Interrupts would appear in their true form.

It is also a marvel to see how clearer the SEAL code looks like compared to its compiled C counterpart. Screenshots in the appendix will showcase some of this.

9. Test Plan
9.1. Test Case Selection
Tests were written by myself, specifically with the goal of testing out the five constructs of SEAL - Interrupts, Functions, Threads, Types, and Globals. Some files would have examples of all five, and relentless tries to get it to parse, and then finally to generate C code from.

9.2. Demo Programs
The power of SEAL is that it has multithreading capability built-in, unlike C and C++, and at the same time it is nowhere near the bloated nature of Threads in Java. In the avionics world, one of the most common things for a computer to do is read data and report back data in pretty much concurrent time. Oftentimes there is only one data port available for I/O, and the various threads and processes running on the embedded computer need to share it cooperatively. SEAL makes this incredibly easy to do. With two function calls you have a thread up and running, and with two more you have a mutex lock in place to protect those shared variables.
In this demo program, an autothrottle computer on board a small airplane is both reading in data from
the neighboring airdata computer and calculating throttle lever rates and speed acceleration and
reporting it back to the pilot’s cockpit at the same time. Moreover, it is also reading and reporting
back the electrical condition of the servomotors that it controls. I also threw in an Interrupt handler
for the Ctrl+C keystroke interrupt for added measure. Here is the code:

Lock outbufferLock;
Double buffer1;
Double buffer2;

//this thread receives speed data and calculates throttle rate and current acceleration
Thread Thread1
{
    While(1)
    {
        sleep(1);
        outbufferLock.Acquire();
        buffer1 = 2.4;
        buffer2 = 4.5;
        print("Current throttle rate is %f degrees/sec, and current acceleration is %f
knots/sec^2\n", buffer1, buffer2);
        outbufferLock.Release();
    }
}
/* this thread uses the same output buffer to send out telemetric data, in this case
the voltage and current readings of the autothrottle computer */
Thread Thread2
{
    While(1)
    {
        sleep(1);
        outbufferLock.Acquire();
        buffer1 = 113.45;
        buffer2 = 5.1;
        printf("Current from servomotor is now %fmA, and voltage is now %fV\n", buffer1, buffer2);
        outbufferLock.Release();
    }
}

Interrupt Interceptor
{
    print("Ctrl+C pressed! Exiting on next request\n");
    Interceptor.Source = 0;
}

Int main()
{
    Interceptor.Source = 1;
    outbufferLock.Lcreate();
    Thread1.Tcreate();
Thread2.Tcreate();
Thread1.Go();
Thread2.Go();
while(1);
}

Contrast it to its C counterpart, which generates this from the above. It is 20% more lines of code and also much harder to follow.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include "SEAL_Thread.h"
#include "SEAL_Lock.h"
#include "SEAL_Util.h"
#include "SEAL_Signal.h"
#include "SEAL_Array.h"

double buffer1;
double buffer2;
Lock outbufferLock;
Thread Thread1;
Thread Thread2;
void *Thread1__Func (void *arg)
{
    while (1)
    {sleep(1);
    SEALLock_Acquire(&outbufferLock);
    buffer1 = 2.4;
    buffer2 = 4.5;
    printf("Current throttle rate is %f degrees/sec, and current acceleration is %f knots/sec^2\n", buffer1, buffer2);
    SEALLock_Release(&outbufferLock);
    }
}

void *Thread2__Func (void *arg)
{
    while (1)
    {sleep(1);
    SEALLock_Acquire(&outbufferLock);
    buffer1 = 113.45;
    buffer2 = 5.1;
    printf("Current from servomotor is now %fmA, and voltage is now %fV\n", buffer1, buffer2);
    SEALLock_Release(&outbufferLock);
    }
}

Interrupt Intercepter;
void Intercepter__Handler (int sig)
printf("Ctrl+C pressed! Exiting on next request\n");

SEALSignal_SetISR(Interceptor____Handler, &Interceptor);
SEALSignal_SetSignal(0, &Interceptor);

}

int main()
{
SEALSignal_SetISR(Interceptor____Handler, &Interceptor);
SEALSignal_SetSignal(1, &Interceptor);

SEALock_Create(&outbufferLock);

SEALThread_Create(&Thread1, Thread1___Func);
SEALThread_Create(&Thread2, Thread2___Func);

SEALThread_Go(&Thread1);
SEALThread_Go(&Thread2);

while(1);
}

Here is a screenshot of the demo program:
10. Lessons Learned

Designing your own programming language, and more importantly, writing the compiler for it, were by no means trivial tasks. Students every semester say the same thing, but I inevitably underestimated the workload it would take to accomplish this herculean task. Indeed, it is true that the devil is in the details. There is just no way of knowing how much is involved until one starts undergoing it. Who knew that type-checking was such a painstaking and meticulous endeavor?

Another thing I learned was that it is good to know when you've reached a practical limit in adding features to your language. If given years, we could perfect our languages and add all kinds of wonderful features and we still wouldn’t be happy. Decades later C++ is still evolving. I had to remove some features in the final version of SEAL, but its main pluses—simple address accessing without the messiness and confusion of pointers, and easy thread and interrupt handler creation, stayed intact thankfully.

11. Appendix

11.1. AST.ml

```ml
type binop = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq
        | Orl | Andl | Or | And | Bsr | Bsl | Xor

type unop = Not | Inc | Dec | Inv

type sealConstruct = Interrupt | Thread | Variable | Function | Class

type sealType =
    Array of sealType * string | Void | Byte | Int | Double | String | Lock | NewType of string

type expr =
    Iliteral of int
    | Fliteral of float
    | Sliteral of string
    | Variable of sealType
    | Id of string (* used in SEAL as well *)
    | Binop of expr * binop * expr
    | Unop of unop * expr
    | Assign of expr * expr
    | Call of string * expr list (**)
    | LabelCall of string * string * expr list
    | CastType of sealType * expr (* the following are new in SEAL*)
    | Address of string * expr
    | GetAddress of string
    | ArrayIndex of string * string
    | Label of expr * expr
    | Swap of string
    | Signal of string * expr
    | Map of string * string * expr * expr
    | Noexpr

(* the following are new expressions for SEAL only
 | Int of int
 | Flt of float
```
type stmt =
    Block of stmt list |
| Expr of expr |
| Return of expr |
| If of expr * stmt * stmt |
| For of expr * expr * expr * stmt list |
| While of expr * stmt list |

type var_decl = {
    vtype : sealType ;
    vname : string; 
}

type func_decl = {
    rtype : sealType ;
    fname : string;
    formals : var_decl list;
    locals : var_decl list;
    body : stmt list;
}

type thread_decl = {
    tname : string;
    tlocals : var_decl list;
    tbody : stmt list;
}

type interrupt_decl = {
    iname : string;
    ilocals : var_decl list;
    ibody : stmt list;
}

type type_decl = {
    ytype : sealType;
    yname : string;
    yproperties : var_decl list;
    yfunctions : func_decl list;
}

let first = fun (a,b,c,d,e) -> a
let second = fun (a,b,c,d,e) -> b
let third = fun (a,b,c,d,e) -> c
let fourth = fun (a,b,c,d,e) -> d
let fifth = fun (a,b,c,d,e) -> e

(* type program = string list * func_decl list *)
type program = var_decl list * func_decl list * thread_decl list * interrupt_decl list * type_decl list

let rec string_of_expr = function
  Sliteral(l) -> l
let string_of_fdecl fdecl =
  match fdecl with
  | Fliteral(l) -> string_of_float l
  | Fliteral(l) -> string_of_int l
  | Id(s) -> s
  | Binop(e1, o, e2) -> "(" ^
                    string_of_expr e1 ^ " " ^
                    (match o with
                     | Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
                     | Equal -> "==" | Neq -> "!="
                     | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
                     | Or1 -> "||" | And1 -> "&&" | Or -> "|" | And -> "&"
                     | Bsr -> ">>" | Bsl -> "<<" | Xor -> "^" ) ^ " " ^
                    string_of_expr e2 ^ ")"
  | Unop(o, e) -> "(" ^
                 (match o with
                  | Not -> "!
                  | Inc -> "++"
                  | Dec -> "--"
                  | Inv -> "~") ^ string_of_expr e")"
  | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
    f ^ "(" ^ string_of_func " ; ") function call performed."
  | Noexpr -> ""
  | CastType(a, b) -> "(" ^ string_of_expr b ^ " of type " ^
                    string_of_expr d ^ " with " ^ string_of_expr c ^ " elements to " ^
                    string_of_expr e")
  | Map(a, b, c, d) -> "Mapping function " ^ b ^ "of type " ^
                    string_of_expr d ^ " with " ^ string_of_expr c ^ " elements to " ^
                    string_of_expr e"] indexed."
  | Label(a) -> "Label " ^ a ^ " now handling interrupt " ^ string_of_expr b
  | Variable(a) -> ""

let rec string_of_stmt = function
  | Block(stmts) ->
    "{" ^ string_of_func " ; "}\n"
  | Expr(expr) -> string_of_expr expr ^ " ;\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ " ;\n";
  | If(e, s1, Block(s2)) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1
  | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
                    string_of_stmt s1 ^ " else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
    "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
    string_of_expr e3 ^ ") " ^ string_of_func " ; ") string.func " ^
    (list.map string_of_stmt s)
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_func " ; ") string.func " ^
                   (list.map string_of_stmt s)

let string_of_vdecl id = " " ^ id.vname ^ " ;\n"

let string_of_vparam id = " " ^ id.vname ^ " , "

let string_of_fdecl fdecl =
fdecl.fname ^ "(" ^ String.concat ", " (List.map (fun x -> x.vname) fdecl.formals) ^ ")\n^n" ^ String.concat "" (List.map string_of_vdecl fdecl.locals) ^ 
String.concat "" (List.map string_of_stmt fdecl.body) ^ 
"})\n
let string_of_tdecl tdecl =
tdecl.tname ^ 
String.concat "" (List.map string_of_vdecl tdecl.tlocals) ^ 
String.concat "" (List.map string_of_stmt tdecl.tbody) ^ 
"})\n
let string_of_idecl idecl =
idecl.iname ^ 
String.concat "" (List.map string_of_vdecl idecl.ilocals) ^ 
String.concat "" (List.map string_of_stmt idecl.ibody) ^ 
"})\n
let string_of_ydecl ydecl =
ydecl.yname ^ 
String.concat "" (List.map string_of_vdecl ydecl.yproperties) ^ 
String.concat "" (List.map string_of_fdecl ydecl.yfunctions) ^ 
"})\n
let string_of_program (vars, funcs, threads, interrupts, types) =
"THE LIST OF VARS: " ^ 
String.concat "" (List.map string_of_vdecl vars) ^ "\n^n" ^ 
"\nTHE LIST OF FUNCS: \n^n" ^ 
String.concat "" (List.map string_of_fdecl funcs) ^ "\n^n" ^ 
"\nTHE LIST OF THREADS: \n^n" ^ 
String.concat "" (List.map string_of_tdecl threads) ^ "\n^n" ^ 
"\nTHE LIST OF INTERRUPTS: \n^n" ^ 
String.concat "" (List.map string_of_idecl interrupts) ^ "\n^n" ^ 
"\nTHE LIST OF TYPES: \n^n" ^ 
String.concat "" (List.map string_of_ydecl types)

11.2. Scanner.ml

(* scanner for SEAL compiler *)
{ open Parser }

let digits = ['0'..'9']+
let exp = 'e'('+'|'-'|'.')? digits
let lxm = ['a'..'z' 'A'..'Z'['a'..'z' 'A'..'Z' '0'..'9' ' ']*
let double = (digits exp? | digits '.' digits? exp? | digits '.' exp)
let str = "" [\"\"]* ""
rule token = parse
[ ' ' \t ' \r ' \n' ] { token lexbuf } (* Whitespace *)
| "/*/" { comment lexbuf } (* Comments *)
| "/ " {comment2 lexbuf}
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LCURLY }
| '}' { RCURLY }
| ';' { SEMIC }
| ':' { COMMA }
| '+=' { PLUS }
| '-=' { MINUS }
| '*=' { TIMES }
| '/=' { DIVIDE }
| '===' { ASSIGN }
| '===' { EQ }
| '!=' { NEQ }
| '<' { LT }
| '<=' { LEQ }
| '>' { GT }
| '>=' { GEQ }
| '||' { ORL }
| '&&' { ANDL }
| '|' { OR }
| '&' { AND }
| '<<' { BSL }
| '>>' { BSR }
| '^' { XOR }
| '!' { NOT }
| '++' { INC }
| '--' { DEC }
| '~' { INV }
| 'If' { IF }
| 'Else' { ELSE }
| 'For' { FOR }
| 'While' { WHILE }
| 'Return' { RETURN }
| 'Void' { VOID }
| 'Int' { INT }
| 'Byte' { BYTE }
| 'Double' { DOUBLE }
| 'String' { STRING }
| 'Thread' { THREAD }
| 'Source' { SOURCE }
| 'Map' { MAP }
| 'Interrupt' { INTERRUPT }
| 'Enum' { ENUM }
| '[' { LBRACKET }
| ']' { RBRACKET }
| '.' { LABEL }
| 'Swap' { SWAP }
| 'True' { TRUE }
| 'False' { FALSE }

(* the unary operators *)

(* the following are the fundamental types and are all unique to SEAL *)

(* for arrays *)

(* can't be treated like normal labels, need to be handled *)
| "Type" { TYPE } (* for the type declaration *)
| "Lock" { LOCK } (* for the lock type *)
| digits as integer { ILITERAL(int_of_string integer) }
| double as dbl { FLITERAL(float_of_string dbl) }
| lxm as id { ID(id) }
| str as slit { SLITERAL(slit) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
"*/" { token lexbuf }
| _ { comment lexbuf }

and comment2 = parse
'
' { token lexbuf}
| _ { comment2 lexbuf }

11.3. Compiler.ml

open Ast
module StringMap = Map.Make(String)

(*type for storing global variables*)
type varSymbolTableEntry = {
  data_type : sealType;
}

(*type for storing functions*)
type funcSymbolTableEntry = {
  ftype : sealType;
  fparameters : sealType StringMap.t;
  flocals : sealType StringMap.t;
  fbodylist : stmt list;
}

(*type for storing SEAL Threads and any local variables defined within the Thread*)
type threadSymbolTableEntry = {
  thlocals : sealType StringMap.t;
  thbodylist : stmt list;
}

(*type for storing SEAL Interrupt handlers and any local variables defined within the handler*)
type interruptSymbolTableEntry = {
  inlocals : sealType StringMap.t;
  inbodylist : stmt list;
}

(*type for storing SEAL types defined by the user*)
type typeTableEntry = {
  thetype : sealType;
  properties : sealType StringMap.t;
functions : funcSymbolTableEntry StringMap.t;
}

(*the entire environment, containing the symbol table, the function table, 
the thread table, the interrupt table, and the SEAL types table *)
type environment = {
  sealVarSymbolTable : sealType StringMap.t;
  sealFuncSymbolTable : funcSymbolTableEntry StringMap.t;
  sealThreadSymbolTable : threadSymbolTableEntry StringMap.t;
  sealInterruptSymbolTable : interruptSymbolTableEntry StringMap.t;
  sealTypeSymbolTable : typeTableEntry StringMap.t;
}

let checkFunctionBody theBody env construct =
  let rec output_function_expr = function
    SLiteral(l) -> l
    | FLiteral(l) -> string_of_float l
    | ILiteral(l) -> string_of_int l
    | Id(s) -> s (* match construct with Function | Thread | Interrupt *)(*if ((StringMap.mem s env.sealVarSymbolTable)) then s
else ""; raise(Failure("Compiler error: \"" ^ s ^ "\" does not exist in the current
context.")) *)
    | Binop(e1, o, e2) -> "(" ^
      string_of_expr e1 ^ " " ^
      (match o with
        Add -> "+", | Sub -> "-", | Mult -> "*", | Div -> "/",
        Equal -> "==", | Neq -> "!",
        Less -> "<", | Leq -> "<=" | Greater -> ">", | Geq -> ">=",
        Or1 -> "|", | And1 -> "&&", | Or -> "|", | And -> "&",
        Bsr -> ">>", | Bsl -> "<<", | Xor -> "^", ) ^ " " ^
      string_of_expr e2 ^ ")"
    | Unop(o, e) -> "(" ^
      (match o with
        Not -> "!",
        Inc -> "++",
        Dec -> "--",
        Inv -> "~") ^ output_function_expr e^")")
    | Assign(v, e) -> output_function_expr v ^ " = " ^ output_function_expr e
    | Call(f, el) ->
      (match f with
        "print" -> "printf" ^ "(" ^ String.concat ", &" (List.map output_function_expr el) ^ ")"
        | _ -> f ^ "(" ^ String.concat ", &" (List.map output_function_expr el) ^ ")"
      )
    | LabelCall(v, f, el) ->
      (match f with
        "Lcreate" -> "SEALLock_Create(&" ^ v ^ ")")
        | "Acquire" -> "SEALLock_Acquire(&" ^ v ^ ")")
        | "Release" -> "SEALLock_Release(&" ^ v ^ ")")
        | "Ldestroy" -> "SEALLock_Destroy(&" ^ v ^ ")")
        | "Tcreate" -> "SEALThread_Create(&" ^ v ^ ", " ^ v ^ "___Func")
        | "Go" -> "SEALThread_Go(&" ^ v ^ ")")
        | "Stop" -> "SEALThread_Stop(&" ^ v ^ ")")
let rec output_function_stmts = function
| Block(stmts) -> "{"(String.concat "") (list.map output_function_expr stmts) ^ "}\n
| Expr(expr) -> output_function_expr expr ^ ";\n
| Return(expr) -> "return " ^ output_function_expr expr ^ ";\n
| If(e, s, Block([])) -> "if (" ^ output_function_expr e ^ ")\n
| If(e, s1, s2) -> "if (" ^ output_function_expr e ^ ")\n
| For(e1, e2, e3, s) -> "for (" ^ output_function_expr e1 ^ "); " ^ output_function_expr e2 ^ "); " ^

| While(e, s) -> "while (" ^ output_function_expr e ^ ")\n
in list.map output_function_stmts (list.rev theBody);

(* prints out the type. Character can either be *, for parameters since SEAL
is inherently pass-by-reference, or ",", for local and global variables *)

let rec output_type key character basetype = function
| Byte -> "unsigned char " ^ character ^ key
| Int -> "int " ^ character ^ key
| Double -> "double " ^ character ^ key
| String -> "char *" ^ character ^ key
| NewType(newtype) -> newtype ^ character ^ " " ^ key
| Lock -> "lock " ^ key
| Array(basetype, sz) -> (output_type key character basetype ^ "][" ^ sz ^ "]")

let rec output_typefunc key = function
| Void -> "void " ^ "][" ^ key
| Byte -> "unsigned char " ^ key
| Int -> "int " ^ key
| Double -> "double " ^ key
| String -> "char *" ^ key
| NewType(newtype) -> newtype ^ " " ^ key
let output_newtype = function
| NewType(newtype) -> newtype ^ " "

(* spits out the global variables in C form, post semantically checked AST *)
let output_globals global_table env =
  let output_each_global key value =
    print_endline (output_type key "" value ^ "");
  in
  StringMap.iter output_each_global global_table

let output_Params param_table =
  let output_each_param key value =
    print_string (output_type key "*" value ^ ", ")
  in
  StringMap.iter output_each_param param_table

(* spits out the thread definitions in C form, post semantically checked AST *)
let output_threads thread_table env =
  let output_each_thread key value =
    print_endline ("Thread " ^ key ^ ");
  in
  StringMap.iter output_each_thread thread_table

  let output_each_threadbody key value =
    let threadbody =
      "void *" ^ key ^ "__Func (void *arg)\n{" in
      print_endline (threadbody);
      output_globals value.thlocals env;
      (*eventually output the body here as well*)
      List.map print_endline (checkFunctionBody value.thbodylist env Thread); (*the gangsta line*)
      (* List.map print_endline (List.map output_function_stmts value.thbodylist); the gangsta line*)
      print_endline ("}\n") in
    StringMap.iter output_each_threadbody thread_table;;

(* spits out the interrupt handlers in C form, post semantically checked AST *)
let output_interrupts interrupt_table env =
  let output_each_interrupt key value =
    print_endline ("Interrupt " ^ key ^ ");
  in
  StringMap.iter output_each_interrupt interrupt_table

  let output_each_interruptbody key value =
    let interruptbody =
      "void " ^ key ^ "__Handler (int sig)\n{" in
      print_endline (interruptbody);
      output_globals value.inlocals env;
      (*eventually output the body here as well*)
      List.map print_endline (checkFunctionBody value.inbodylist env Interrupt); (*the gangsta line*)
      (* List.map print_endline (List.map output_function_stmts value.inbodylist); the gangsta line*)
      print_endline ("}\n") in
    StringMap.iter output_each_interruptbody interrupt_table;;

let output_functions function_table typeFunction env =
  let output_each_function key value =
    print_string (output_typefunc key value.ftype ^ "(");
    output_Params value.fparameters;
if typeFunction != Ast.Void then print_string ((output_newtype typeFunction) ^" " *itself");
print_endline ("");
print_endline ("{");
output Globals value.flocals env;
(*eventually output the body here as well*)
List.map print_endline (checkFunctionBody value.fbodylist env Function);
(* List.map print_endline (List.map output_function_stmts value.fbodylist); the gangsta line*)
print_endline ("})\n") in
StringMap.iter output_each_function function_table;;

let output_types types_table env =
let output_each_type key value =
  print_endline ("typedef struct {";
  outputGlobals value.properties env;
  print_endline ("} " ^ key ^ ";\n");
  (*eventually output the body here as well*)
  output_functions value.functions value.thetype env in
  StringMap.iter output_each_type types_table;;

(*utility to print out keys of each string Map*)
let print_vars key value =
  print_string(key ^ " \n");
  (*array type checking. This is amazing!! *)
  let thetype = Ast.Array(Int, "function") and thattype = value in
  if thetype = thattype then print_endline "BOOMSHAKALAKA!"
else print_endline "NOPE!";;

let print_funcs key value =
  print_string("Function " ^ key ^ ": \n");
  print_endline "Parameters:";
  StringMap.iter print_vars value.fparameters;
  print_endline "Local variables:";
  StringMap.iter print-vars value.flocals;;

let print_threads key value =
  print_string("Thread " ^ key ^ ": \n");
  print_endline "Local variables:";
  StringMap.iter print-vars value.thlocals;;

let print_interrupts key value =
  print_string("Interrupt " ^ key ^ ": \n");
  print_endline "Local variables:";
  StringMap.iter print-vars value.inlocals;;

let print_types key value =
  print_string("SEAL Type " ^ key ^ ": \n");
  print_endline "Properties:";
  StringMap.iter print-vars value.properties;
  print_endline "Methods:";
  StringMap.iter print_funcs value.functions;;
(*TSG

let check_thread_bodies thread_table =
    let check_each_tbody key value =
        value.thbody = checkFunctionBody value.thbodylist value.thlocals StringMap.empty in
    StringMap.map check_each_tbody thread_table;;

let check_interrupt_bodies interrupt_table =
    let check_each_ibody key value =
        value.inbody = checkFunctionBody value.inbodylist value.inlocals StringMap.empty in
    StringMap.map check_each_ibody interrupt_table;;

let check_function_bodies function_table =
    let check_each_fbody key value =
        value.fbody = checkFunctionBody value.fbodylist value.flocals value.fparameters in
    StringMap.map check_each_fbody function_table;;

(*function that creates symbol tables for variables *)

let createSealVarSymbolTable map (var_elem: var_decl list) =
    List.fold_left
    (fun map thelist ->
        if StringMap.mem thelist.vname map
            then raise(Failure("Compiler error: a variable named " ^ thelist.vname ^ " already exists. Please choose a different name."))
        else
            StringMap.add thelist.vname thelist.vtype map)
    map var_elem

let createSealFuncSymbol (var_elem: func_decl) =
    {
        ftype = var_elem.rtype;
        fparameters = createSealVarSymbolTable StringMap.empty var_elem.formals;
        flocals = createSealVarSymbolTable StringMap.empty var_elem.locals;
        fbbodylist = var_elem.body; (*this is prior to checking the body *)
    }

(*function that creates symbol table for functions *)

let createSealFuncSymbolTable map (var_elem: func_decl list) =
    List.fold_left (fun map thelist ->
        if StringMap.mem thelist.fname map then raise(Failure("Compiler error: a function named " ^ thelist.fname ^ " already exists. Please choose a different name.")) else
            StringMap.add thelist.fname (createSealFuncSymbol thelist) map)
    map var_elem

let createSealThreadSymbol (var_elem: thread_decl) =
    {
        thlocals = createSealVarSymbolTable StringMap.empty var_elem.tlocals;
        thbodylist = var_elem.tbody; (*this is prior to checking the body *)
    }
let createSealThreadSymbolTable map (var_elem: thread_decl list) =
    List.fold_left (fun map thelist ->
        if StringMap.mem thelist.tname map then raise(Failure("Compiler error: a Thread named " ^ thelist.tname ^ "/ already exists. Please choose a different name.")) else
            StringMap.add thelist.tname (createSealThreadSymbol thelist) map)
        map var_elem

let createSealInterruptSymbol (var_elem: interrupt_decl) =
    {
        inlocals = createSealVarSymbolTable StringMap.empty var_elem.ilocals;
        inbodylist = var_elem.ibody; (*this is prior to checking the body *)
    }

let createSealInterruptSymbolTable map (var_elem: interrupt_decl list) =
    List.fold_left (fun map thelist ->
        if StringMap.mem thelist.iname map then raise(Failure("Compiler error: an Interrupt handler named " ^ thelist.iname ^ "/ already exists. Please choose a different name.")) else
            StringMap.add thelist.iname (createSealInterruptSymbol thelist) map)
        map var_elem

let createSealTypeSymbol (var_elem: type_decl) =
    {
        thetype = var_elem.ytype;
        properties = createSealVarSymbolTable StringMap.empty var_elem.yproperties;
        functions = createSealFuncSymbolTable StringMap.empty var_elem.yfunctions;
    }

let createSealTypeSymbolTable map (var_elem: type_decl list) =
    List.fold_left (fun map thelist ->
        if StringMap.mem thelist.yname map then raise(Failure("Compiler error: a Type named " ^ thelist.yname ^ "/ already exists. Please choose a different name.")) else
            StringMap.add thelist.yname (createSealTypeSymbol thelist) map)
        map var_elem

let header =
    "
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include "SEAL_Thread.h"
#include "SEAL_Lock.h"
#include "SEAL_Util.h"
#include "SEAL_Signal.h"
#include "SEAL_Array.h"
"

let translate (globals, functions, threads, interrupts, types) =
(* print_endline "OH SNAP HERE'S THE LIST OF GLOBALS"; *)
let var_table = createSealVarSymbolTable StringMap.empty globals in
(* StringMap.iter print_vars var_table; *)

(* print_endline "OH SNAP HERE's THE LIST OF FUNCTIONS"; *)
let fun_table = createSealFuncSymbolTable StringMap.empty functions in
(* StringMap.iter print_funcs fun_table; *)

(* print_endline "OH SNAP HERE's THE LIST OF THREADS"; *)
let thread_table = createSealThreadSymbolTable StringMap.empty threads in
(* StringMap.iter print_threads thread_table; *)

(* print_endline "OH SNAP HERE's THE LIST OF INTERRUPTS"; *)
let interrupt_table = createSealInterruptSymbolTable StringMap.empty interrupts in
(* StringMap.iter print_interrupts interrupt_table; *)

(* print_endline "OH SNAP HERE's THE LIST OF TYPES"; *)
let type_table = createSealTypeSymbolTable StringMap.empty types in
(* StringMap.iter print_types type_table; *)

let env =
{
    sealVarSymbolTable = var_table;
    sealFuncSymbolTable = fun_table;
    sealThreadSymbolTable = thread_table;
    sealInterruptSymbolTable = interrupt_table;
    sealTypeSymbolTable = type_table; (*TSG HMMMM *)
}
in
(*
    check_function_bodies fun_table;
    check_thread_bodies thread_table;
    check_interrupt_bodies interrupt_table;
*)

print_endline header;
output_globals var_table env;
output_types type_table env;
output_threads thread_table env;
output_interrupts interrupt_table env;
output_functions fun_table Void env;

11.4. parser.mly

{% open Ast %}
%token PLUS TIMES MINUS DIVIDE ASSIGN EQ LT LEQ GT GEQ
%token SEMIC LPAREN RPAREN LCURLY RCURLY LBRACKET RBRACKET COMMA
%token RETURN IF ELSE FOR WHILE
%token <int> I_LITERAL
%token <string> ID
%token <float> F_LITERAL
/* SEAL tokens */
/* unary operators */
%token INC DEC NOT INV NEG ADDRESS SWAP SOURCE MAP
/* binary operators */
%token ANDL ORL
%token XOR AND OR
%token EQU NEQ LTH GTH LTE GTE
%token BSL BSR
%token ADD SUB MULT DIV MOD
/* the fundamental types */
%token INT DOUBLE BYTE STRING
/* other types */
%token ENUM STRING LOCK
/* declarations */
%token THREAD INTERRUPT TYPE LABEL VOID
/* boolean values */
%token TRUE FALSE

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
/* SEAL unary operator precedence: */
%left NOT INV NEG /* lowest precedence */
%left DEC
%left INC /* highest precedence */
%left ADDRESS
%left SWAP

/* SEAL binary operator precedence, left associative: */
%left COMMA /* lowest precedence */
%left ANDL ORL
%left XOR AND OR
%left EQU NEQ
%left LTH GTH LTE GTE
%left BSL BSR
%left ADD SUB
%left MULT DIV MOD /* highest precedence */

%left EQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

```
/* TG 4-22-14 here is where the entire grammar will need to go */

transl_unit:
| interrupt_def
| interrupt_def transl_unit

program:
/* nothing */ { [], [], [], [], [] }
| program vdecl { List.rev ($2 :: first $1), second $1, third $1, fourth $1, fifth $1 }
| program fdecl { first $1, List.rev ($2 :: second $1), third $1, fourth $1, fifth $1 }
| program tdecl { first $1, second $1, List.rev ($2 :: third $1), fourth $1, fifth $1 }
| program idecl { first $1, second $1, third $1, List.rev ($2 :: fourth $1), fifth $1 }
| program ydecl { first $1, second $1, third $1, fourth $1, List.rev ($2 :: fifth $1 ) }

tdecl:
THREAD ID LCURLY vdecl_list stmt_list_opt RCURLY
{

}

tdecl:

idecl:
INTERCEPT ID LCURLY vdecl_list stmt_list_opt RCURLY
{

}
{ 
    rtype = $1;
    fname = $2;
    formals = $4;
    locals = List.rev $7;
    body = List.rev $8;
}

}

formals_opt:
    /* nothing */ { [] }
    | formal_list { List.rev $1 }

formal_list:
    formal /* nothing */ { [$1] }
    | formal_list COMMA formal { $3 :: $1 }

vdecl_list:
    /* nothing */ { [] }
    | vdecl_list vdecl { $2 :: $1 }

vdecl:
    return_type ID SEMIC
    {
    {
        vtype = $1;
        vname = $2;
    }
    }
    | return_type ID array_id SEMIC
    {
    {
        vtype = Array($1, $3);
        vname = $2;
    }
    }

array_id:
    LBRACKET ILITERAL RBRACKET { string_of_int $2 }
    | LBRACKET ID RBRACKET { $2 }

array_size:
    ILITERAL { Iliteral($1) }
    | ID { Id($1) }

/*TSG 5-7-14*/
fdecl_list:
    | fdecl { [$1] }
    | fdecl_list fdecl { $2 :: $1 }
fdecl_opt:
   /* nothing */ { [] }
   | fdecl_list { List.rev $1 }

formal:
   return_type ID
   {
      { vtype = $1; vname = $2; }
   }
   | return_type ID array_id
   {
      { vtype = Array($1, $3); vname = $2; }
   }

return_type:
   VOID {Void}
   | BYTE {Byte}
   | INT {Int}
   | STRING {String}
   | DOUBLE {Double}
   | LOCK {Lock}
   | ID {NewType($1)}

stmt_list_opt:
   /* nothing */ { [] }
   | stmt_list { List.rev $1 }

stmt_list:
   stmt /* nothing */ { [$1] }
   | stmt_list stmt { $2 :: $1 }

stmt:
   expr SEMIC { Expr($1) }
   | RETURN expr SEMIC { Return($2) }
   | L CURLY stmt_list R CURLY { Block(List.rev $2) }
   | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
   | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
   | FOR LPAREN expr_opt SEMIC expr_opt SEMIC expr_opt RPAREN L CURLY stmt_list_opt R CURLY
      { For($3, $5, $7, $10) }
   | WHILE LPAREN expr RPAREN L CURLY stmt_list_opt R CURLY { While($3, $6) }

expr_opt:
   /* nothing */ { Noexpr }
   | expr { $1 }

expr:
ILITERAL { Iliteral($1) }
| F LITERAL { Fliteral($1) }
| SLITERAL { Sliteral($1) }
| ID {Id($1)}
| ID array_id {ArrayIndex($1, $2)}
| ID LABEL ID array_id { Label(Id($1), (ArrayIndex($3, $4))) }
| INC expr { Unop(Inc, $2) }
| DEC expr { Unop(Dec, $2) }
| NOT expr { Unop(Not, $2) }
| INV expr { Unop(Inv, $2) }
| expr PLUS expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr EQ expr { Binop($1, Equal, $3) }
| expr NEQ expr { Binop($1, Neq, $3) }
| expr LT expr { Binop($1, Less, $3) }
| expr LEQ expr { Binop($1, Leq, $3) }
| expr GT expr { Binop($1, Greater, $3) }
| expr GEQ expr { Binop($1, Geq, $3) }
| expr ORL expr { Binop($1, Orl, $3) }
| expr ANDL expr { Binop($1, Andl, $3) }
| expr OR expr { Binop($1, Or, $3) }
| expr AND expr { Binop($1, And, $3) }
| expr BSR expr { Binop($1, Bsr, $3) }
| expr BSL expr { Binop($1, Bsl, $3) }
| expr XOR expr { Binop($1, XOR, $3) }
| expr ASSIGN expr { Assign($1, $3) }
| ID ADDRESS { GetAddress($1) }
| ID LABEL SOURCE ASSIGN expr { Signal($1, $5) }
| ID LABEL ADDRESS ASSIGN expr { Address($1, $5) }
| ID ADDRESS ASSIGN ID LABEL ADDRESS { Address($1, Id($5)) }
| ID LABEL ADDRESS SWAP RPAREN { Swap($1) }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }
| ID LABEL ID LPAREN actuals_opt RPAREN { LabelCall($1, $3, $5) }
| ID LABEL MAP LPAREN ID COMMA expr COMMA expr RPAREN { Map($1, $5, $7, $9) }
| LPAREN expr RPAREN { $2 }

actu als_opt:
    /* nothing */ { [] }
| actuals_list { List.rev $1 }

actu als_list:
    expr { [$1] }
| actuals_list COMMA expr { $3 :: $1 }

11.5. seal.ml
type action = Ast | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast); ("-c", Compile)]
  else Compile

  let lexbuf = Lexing.from_channel stdin
  let program = Parser.program Scanner.token lexbuf
  in
  match action with
  Ast -> let listing = Ast.string_of_program program
         in print_string listing
  | Compile -> Compile.translate program

11.6. Makefile

OBJS = ast.cmo parser.cmo scanner.cmo compile.cmo seal.cmo

TARFILES = Makefile testall.sh scanner.mll parser.mly
           ast.ml compile.ml seal.ml

seal : $(OBJS)
  ocamlc -o seal $(OBJS)

scanner.ml : scanner.mll
  ocamlllex scanner.mll

parser.ml parser.mli : parser.mly
  ocamlyacc parser.mly

%.cmo : %.ml
  ocamlc -c $<

%.cmi : %.mli
  ocamlc -c $<

seal.tar.gz : $(TARFILES)
  cd .. && tar czf seal/seal.tar.gz $(TARFILES:%=seal/%)

.PHONY : clean
clean :
  rm -f seal parser.ml parser.mli scanner.ml testall.log
  *.cmo *.cmi *.out *.diff

# Generated by ocamldep *.ml *.mli
ast.cmo:
  ast.cmx:
tac.cmo: ast.cmo
tac.cmx: ast.cmx
compile.cmo: tac.cmo ast.cmo
compile.cmx: tac.cmx ast.cmx
11.7. Seal.sh

if [ $# -eq 0 ]
then
    echo "please enter the source file to compile"
else
    if [ -f $1 ]
then
        ./seal -c < $1 > $2
        rm *.o
        gcc -c SEAL_Lock.c
        gcc -c SEAL_Thread.c
        gcc -c SEAL_Util.c
        gcc -c SEAL_Signal.c
        gcc -c SEAL_Array.c
        gcc -D_REENTRAY $2 SEAL_Signal.o SEAL_Lock.o SEAL_Thread.o SEAL_Util.o SEAL_Array.o -lpthread
        ./a.out
    else
        echo $1 "does not exist. Please try again."
    fi
fi

11.8. SEAL_Array.h

#ifndef __SEAL_ARRAY_H_
#define __SEAL_ARRAY_H_

typedef enum
{
    BYTE, SHORT, USHORT, INT, UINT, LONG, ULONG, FLOAT, DOUBLE, OTHER
} TYPE;

void SEALArray_Map(void *array, void (*function)(void *arg), int length, TYPE type);

int SEALArray_Sort(void *array);

int SEALArray_Reverse(void **array);

void * SEALArray_Min(void **array);

void * SEALArray_Max(void **array);
#endif

## 11.9. SEAL_Array.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "SEAL_Array.h"

#define map(type) for (i=0;i<length;i++){function(type array + i);} 
#define C_BYTE    (unsigned char   *)
#define C_SHORT   (short           *)
#define C_USHORT  (unsigned short  *)
#define C_INT     (int             *)
#define C_UINT    (unsigned int    *)
#define C_LONG    (long            *)
#define C_ULONG   (unsigned long   *)
#define C_FLOAT   (float           *)
#define C_DOUBLE  (double          *)

void SEALArray_Map(void *array, void (*function)(void * arg), int length, TYPE type) 
{
    int i = 0;
    switch (type) 
    {
        case BYTE:
            map(C_BYTE)
            break;
        case SHORT:
            map(C_SHORT)
            break;
        case USHORT:
            map(C_USHORT)
            break;
        case INT:
            map(C_INT)
            break;
        case UINT:
            map(C_UINT)
            break;
        case LONG:
            map(C_LONG)
            break;
        case ULONG:
            map(C_ULONG)
            break;
        case FLOAT:
            map(C_FLOAT)
            break;
        case DOUBLE:
```
map(C_DOUBLE)
break;
default:
map()
break;
}
}

int SEALArray_Sort(void *array)
{
}

int SEALArray_Reverse(void **array)
{
}

11.10. SEAL_Lock.h

#ifndef __SEAL_LOCK_H_
define __SEAL_LOCK_H_

typedef struct
{
    void *lock;
} Lock;

void SEALLock_Create(Lock *t);
int SEALLock_Acquire(Lock *t);
int SEALLock_Release(Lock *t);
void SEALLock_Destroy(Lock *t);

#endif

11.11. SEAL_Lock.c

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "SEAL_Lock.h"

static unsigned char _threadID = 0;

void SEALLock_Create(Lock *l)
{
    l->lock = malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(l->lock, NULL);
}

int SEALock_Acquire(Lock *l)
{
    pthread_mutex_lock(l->lock);
}

int SEALock_Release(Lock *l)
{
    pthread_mutex_unlock(l->lock);
}

void SEALock_Destroy(Lock *l)
{
    pthread_mutex_destroy(l->lock);
    free(l->lock);
    l->lock = NULL;
}

11.12. SEAL_Signal.h

#ifndef __SEAL_SIGNAL_H_
#define __SEAL_SIGNAL_H_
#define SEALINT 1

typedef struct
{
    void *Address;
    int Source;
    void (*func)(int);
} Interrupt;

void SEALSignal_SetISR(void (*ISR)(int), Interrupt *interrupt);
void SEALSignal_SetSignal(int sig, Interrupt *interrupt);
#endif

11.13. SEAL_Signal.c

#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include "SEAL_Signal.h"
void SEALSignal_SetSignal(int sig, Interrupt *interrupt)
{
    interrupt->Source = sig;
    switch (interrupt->Source)
    {
    case -1:
        //ignore all
        signal(SIGINT, SIG_IGN);
        break;
    case 0:
        //restore default to all
        signal(SIGINT, SIG_DFL);
        break;
    case SEALINT:
        signal(SIGINT, interrupt->func);
        break;
    }
}

//for use with code generation.
//This may go away and be replaced by a direct assignment, depending if
//additional checks may be done here
void SEALSignal_SetISR(void (*isr)(int), Interrupt *interrupt)
{
    interrupt->func = isr;
}

11.14. SEAL_Thread.h

#ifndef __SEAL_THREAD_H__
#define __SEAL_THREAD_H__

typedef struct
{
    void *pthread;
    void *(*func)(void *);
} Thread;

void SEALThread_Create(Thread *t, void * (*function) (void * arg));

int SEALThread_Go(Thread *t);

int SEALThread_Join(Thread *t);

int SEALThread_Stop(Thread *t);

void SEALThread_Destroy(Thread *t);
#endif

## 11.15. SEAL_Thread.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "SEAL_Thread.h"

static unsigned char _threadID = 0;

void SEALThread_Create(Thread *t, void * (*function) (void * arg))
{
    t->pthread = malloc(sizeof(pthread_t));
    t->func = function;
}

int SEALThread_Go(Thread *t)
{
    int success = -1;

    if (pthread_create((pthread_t*)t->pthread, NULL, t->func, &_threadID) == 0)
    {
        _threadID++;
        printf("Thread %d is now running!\n", _threadID);
        success = 0;
    }
    return success;
}

int SEALThread_Join(Thread *t)
{
    int success = -1;
    pthread_t thread = *(pthread_t*)t->pthread;
    if (pthread_join(thread, NULL) == 0)
    {
        success = 0;
    }
    return success;
}

int SEALThread_Stop(Thread *t)
{
    int success = -1;
    pthread_t thread = *(pthread_t*)t->pthread;
    if (pthread_cancel(thread) == 0)
    {
        
```
success = 0;
}
return success;
}

void SEALThread_Destroy(Thread *t)
{
    free(t->pthread);
    t->pthread = NULL;
    t->func = NULL;
}

11.16. SEAL_Util.h

#ifndef __SEAL_UTIL_H__
#define __SEAL_UTIL_H__

void * SEALUtil_Swap(void *object, int length);
void * SEALUtil_Move(void *object, void *location, int length);

#endif

11.17. SEAL_Util.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SEAL_Util.h"

void * SEALUtil_Swap (void *object, int length)
{
    int i;
    char temp;
    char *array = (char *)object;
    int arraymax = length - 1;
    for (i = 0; i < (length / 2); i++)
    {
        temp = array[i];
        array[i] = array[arraymax - i];
        array[arraymax - i] = temp;
    }

    return object;
}

void * SEALUtil_Move (void *object, void *location, int length)
{ return memmove(location, object, (size_t)length); }

#define SEALUtil_Length(x) sizeof(x)

### 11.18. Test-swap.sl

```c
Int main()
{
    Int a;
    Int b;
    a = 305419896; // 0x12345678 in decimal
    print("a = %X\n", a);
    a.Swap();
    print("a = %X\n", a);
    a.Swap();
    print("a = %X\n", a);

    Return 0;
}
```

Displays the following cool output:
11.20.   Test-semanticerror.sl

Thread Holla
{
}
Int b1;
int main()
{
    Int a;
    String a;
    a = 23;
    If (1) print(1);
    //a = 42;
    print(39 + 3 - 4 + 5 / 3 * 8 * 4 - 2 + 1);
    Return a;
}
// Chinatown hustler

Compiler correctly finds the error:

11.21.   Test-thread.sl

Int one[5];
Int two;
Double three;
String four[28];

Thread Thread1
{
    Double x;
    x = 12;
If (1) print(1);
  output = 23;
  If (1) print(1);
}

Thread Thread3
{Int r2d2[40];}

Type State
{
  Int State_holla;
  Double State_doh;
  String State_what;

  String hollaFunc1()
  {
    State_holla = 32;
    State_doh = 3.134;
    State_what = "hello there";
  }

  String hollaFunc2()
  {
  }
}

Interrupt wakeUp
{
  Int x;
  ++x;
  x ^ x | x;
  print("Time to wake up!");
}

String function1()
{
}

Int function2()
{
  String f;
  f = 123;
}

Int function3(String aaa[3], Int b, Int c)
{
  Int d;
  Double e;
String f;
e = 3.145;
hhh = 1223;
// a[1];

Holla function4()
{
}

Holla function5()
{
}

String main()
{
    Int w;
    State welcome;
    welcome.hollaFunc1();

    If (1) print(1);
    print("%a\n", 39 + 3 - 4 + 5 / 3 * 8 * 4 - 2 + 1);

    Thread1.Create();
    // Thread1.Go();
    w = 12;
    w.Swap();
    w = w.Swap();
    w.Address = a;
    Return a;
}
// Chinatown hustler

Thread Thread2
{
    Int output; // Int output;
    If (1) print(1);
    output = 12;
}
12. References

1. “Interrupt Functions” Cx51’s user’s guide. Accessed March 2014
   http://www.keil.com/support/man/docs/c51/c51_le_interruptfuncs.htm

