SEAL Project Proposal

COMS W4115 Spring 2014

Ten-Seng Guh tg2458

Contents

1. Introduction

Simple Embedded Avionics Language (SEAL) is a programming language that simplifies the programming tasks most commonly found in embedded systems development, particularly in the avionics industry. Avionics often perform continuous calculations and logging of various data points relevant to space-craft/aircraft performance, and always send out diagnostic and/or telemetric data to the ground. To support the implementation of this, tasks such as accessing a variable by its address, swapping the bytes of a variable in order to support both Big as well as Little Endian architectures, and creating deterministic, thread-safe and reentrant code are all made much easier with SEAL. Not only will the lines of code decrease compared to its C, Ada, and C++ counterparts, the reduced source code will also lead to less headaches, more sleep, and more project throughput. Various architectures will be supported, but for the purposes of this proposal we shall target the ARMv7-A architecture using the Android NDK compiler. It is common in many embedded systems and will find its way into the avionics industry shortly.

SEAL is strongly and statically typed, case-sensitive, procedural, and can be object oriented.

2. Syntax

2.1.Functions

SEAL, being a procedural language, supports functions. A function can return something or nothing; the return type is inferred from the return statement or the lack thereof. Recursive functions are allowed. The `Main` function is where the application code begins execution, but set-up of registers, interrupt service routines, locks, etc. can be done before `Main` runs. Below are a recursive function that returns the factorial of an inputted integer, and a void function that prints out an inputted integer:

```
Factorial(Int n)
{
  if (n == 1) return 1;
  else return n * Factorial(n - 1);
}

PrintValues(Int n)
{
  Print("n is equal to " + n);
}
```

2.2.Objects

All variables are objects. All objects are tied to an address. A variable name in SEAL is really more like a user-defined moniker for an address/range of addresses.

In other languages, it can be eternally confusing as to whether you are accessing a variable by its value, or by its reference. SEAL makes it clear: you are ALWAYS accessing a variable by its reference. Thus:

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

The address of the variable is always made available to you by the `Address` label. If you need to get its address, you'd simply do so by checking `x.Address` (more on labels in the following section).

2.3.Labels

An object has labels. Labels are similar to properties in languages such as Java and C# but may not have allocated any additional memory themselves (i.e. they are not separate objects themselves). Hence the term "label"; it's another aspect through which to view the object with. All objects 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.

Table 1: Object Labels

Some labels are actually static and global functions, such as Swap. The implementation for Swap will reside in code space as a statically defined global function. Whenever Swap is called for an object, it will pass the object reference, along with object length, into the global Swap function and return swapped contents to the stack. Lock is the only label which is an object itself, albeit a global, static object.

2.4.Types

There are eight types in SEAL, known as the eight fundamental types. In C terms, a type can be thought of as a class, struct, or union. It is a specification for a way to represent data in SEAL. The following are the eight fundamental types. Size is in bytes and is specific to the ARMv7-A architecture.

Type	**Return Type**	**Description**
Object	1	The base type which all types are derived from, including the six other fundamental types.
Byte	1	Represents one byte of data, 8-bits unsigned.
Bool	1	Represents a Boolean, can be either true or false.
UShort/Short	2	Represents an unsigned/signed 16-bit integer.
Uint/Int	4	Represents an unsigned/signed 32-bit integer.
Ulong/Long	8	Represents an unsigned/signed 64-bit integer.
Float	4	Represents a 32-bit floating

		point number, unsigned by default.
Double	8	Represents a 64-bit floating point number, unsigned by default.

Table 2: Types

All types in SEAL are descendants of the Object type. New types can be created by the user, limited only by their imagination. They will automatically derive from the Object type. New functions can also be defined for new types. Here is an example of creating and using a new type called Bus1Type:

```
Type Bus1Type
{
        Float baroAlt;
        Float pressureAlt;
        Int     AOA;
        Byte goAround;
        Byte lastFault;
        CalculateAOA();
        CalculateAOAError();
        Recalibrate(Int pitch);
}
```

2.5.Arrays and Enumerations

Arrays are supported in SEAL and are objects themselves. They are fixed size, mutable and have O(1) random access. In addition to the standard Object labels, they have the following powerful labels:

Type	Return Type	Description
Apply	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.
Search	Int	Searches for object, gets the index for first occurrence, otherwise gets -1.

The following are examples of using arrays:

```
Int a[7] = {10, 2, 5, 3, 5, 8, 0};
a.Apply(Print);                 //will print "10253580"
a.Sort().Apply(Print);          //will print "02355810"

printWithSpace(Int n)
{
        print(n + " ");
}
```

```
a.Sort().Apply(printWithSpace);       //will print "0 2 3 5 5 8 10"
a.Search(4);   //returns -1
a.Search(2); //returns 2
```

Enumerations are essentially just byte arrays with useful monikers attached to each byte. The following is an example:

```
Enum Seasons = {Spring, Summer, Fall, Winter};

Print(Seasons.Length());   //will print "4"
Seasons a = Fall;
```

2.6.Interrupts

In embedded applications development, it is not uncommon to handle interrupts yourself. With this in mind, SEAL makes it incredibly easy to write your own interrupt service routines. The keyword `Interrupt` before the usual function declaration is all you need to delineate to the SEAL compiler that this function will be used to handle interrupts. Interrupt functions cannot return anything. Hooking the ISR into the interrupt vector table is a painless affair. Below is an example of hooking the ISR SoundOff, which makes a beep sound every 4 seconds, into address 0x3E, which is one of the timer interrupts.

```
Int counter = 0;
Bool makeSound = FALSE;
makeSound.Address = 0x4A;      //speaker byte
Interrupt SoundOff
{
        counter++;
        if (counter == 4000)
        {
                counter = 0;
                makeSound = TRUE;
        }
}

SoundOff.Address = 0x3F;       //hooking ISR to timer interrupt at 0x3F
```

2.7.Threads

Applications running on embedded systems will also often have multiple threads of execution. This brings to light another limitation of C is that it lacks a built-in library for threads. SEAL comes with a very simple and powerful threading capability, activated by the keyword `Thread`. Anything enclosed in a thread block will execute in its own stack and address space. Shared variables among threads will be protected via the `lock` label that comes inherent with all types. Threads have the following labels:

Type	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.

Table 3: Thread Labels

3. Sample Program

SEAL is meant for writing application level software of an embedded system, specifically for avionics. It can also suffice for writing a simple operating system for an embedded system due to its ability to handle interrupts and schedule threads. Avionics will almost always have diagnostic or telemetric data streaming out a serial port to a PC or some other external system. Oftentimes the endianness of the PC will not match that of the avionics, or there may even be different endianness used between the various components of the avionics themselves. In these cases, SEAL's swap label is of most relevant use.

Below is a sample program that reads various data points from an auto-throttle computer and spits them out to an external data logger. The processor that the auto-throttle uses is Big Endian; the data logger is Little Endian, so byte swapping is performed on the data points.

```
Include "SerialPort.ss"

Lock lock;
SerialPort sp;
sp.Lock = lock;        //this serial port shall be shared between two threads, so lock it
sp.Configure("COM1", 115200, 1, 0, 0);


Float vm1, vm2, vtl1, vtl2;   //voltage of motor 1, motor 2, throttle lever 1, throttle lever 2
Int cm1, cm2, ctl1, ctl2;     //current of motor 1, motor 2, throttle lever 1, throttle lever 2

Float tr1, tr2;                              //throttle rate of lever 1, lever 2
Int sk;                                      //current airspeed in knots
Float sm;      //current airspeed in Mach

vm1.Address = 0x400;
vm2.Address = 0x410;
vtl1.Address = 0x420;
```

```
vtl2.Address = 0x430;
cm1.Address = 0x404;
cm2.Address = 0x414;
ctl1.Address = 0x424;
ctl2.Address = 0x434;

tr1.Address = 0x800;
tr2.Address = 0x810;
sk.Address = 0x1100;
sm.Address = 0x1104;

Get_Voltages_And_Currents()
{
	String voltages, currents, totalOutput;

	voltages = "Voltages: \nMotor 1 = " + vm1 + "\n Motor 2 = " + vm2 + "\ Throttle Level 1 =
" + vtl1 + "\ Throttle Level 2 = " + vtl2 + "\n";
	currents = "Currents: \nMotor 1 = " + cm1 + "\n Motor 2 = " + cm2 + "\ Throttle Level 1 =
" + ctl1 + "\ Throttle Level 2 = " + ctl2 + "\n";
	totalOutput = voltages + "\n\n\n" + currents;

	return totalOutput;
}

Get_Throttle_Rate()
{
	String totalOutput;
	totalOutput = "Throttle Rate 1 = " + tr1 + "\nThrottle Rate 2 = " + tr2 + "/nAirspeed (in
knots) = " + sk + "/n Airspeed (in Mach) = " + sm;

	return totalOutput;
}

Thread Thread1()
{
	String output;

	while (TRUE)
	{
		Sleep(1000);
		output = Get_Voltages_And_Currents().Swap();
		sp.Write(output);
	}
}

Thread Thread2()
{
	String output;

	while (TRUE)
	{
		Sleep(400);
		output = Get_Throttle_Rate().Swap();
		sp.Write(output);
	}
}

Main()
{
	Thread1.Priority = 1;
	Thread1.Go();
	Thread2.Priority = 2;
	Thread2.Go();
}
```

Table 4: Sample Data collecting program for auto-throttle computer