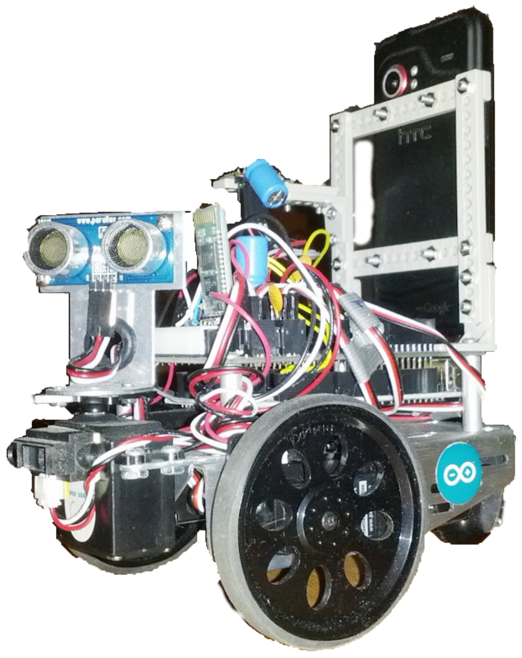
^ (Caret)

V.001a specifications

Brian Gormanly
MSCS 610
Spring 2013



Henry II of Hopewell

1. Introduction

Caret is intended to provide a modern statically typed, compiled language targeted at embedded systems, micro-controllers, ARM architecture and other systems whose primary function is defined by a specific program or programs. Caret is a programming language designed to exist directly on the hardware, including optional libraries for general operating system functionality such as scheduling and memory management. Caret also includes it's own optional garbage collection (managed as a library within your own program).

Languages like C work extremely well for programming in embedded systems because of their low-level nature, portability on many platforms, low memory overhead and hardware requirements. However, with the emergence of multi-core embedded systems the lack of good support for modern concurrent programming methods and garbage collection is apparent. Higher level and intermediary languages such as Java offer garbage collection and some concurrent programming advantages but face limited portability and fall short when it comes to working directly with the hardware, memory usage and execution speed.

Programmers increasingly find themselves tasked with spanning multiple hardware platforms to accomplish a single task and are expending considerable effort building and maintaining code for thread and memory management on smaller platforms, and fighting the operating system for control of memory and resources on larger ones. Commonly they are writing half of their code in C and Assembler and the other half in Java or another higher-level language and using XML or other intermediary markup to communicate data between them. New "hybrid" platforms are immerging which combine the I/O power of micro-controllers with the processing power of the efficient and powerful ARM based CPUs. Caret can be run on both systems giving the following major advantages:

1. CPU based system has no need of operating system overhead drastically increasing processing speed and memory available. Overall cost is decreased, as less CPU power and memory are needed to achieve requirements.
2. No need to use data interchange medium between the 2 systems if they are running in the same language. Complex data structures can be shared with much higher efficiency.
3. Code libraries and source code can be shared by programs running on both systems

Caret is intended to thrive and enable developers in environments where they are the only occupants. Currently, developers writing code for micro-controllers such as the Amtel AVR who wants to write a multi-threaded application are maintaining the scheduling and memory management code themselves. Other developers prototyping and building robots and more advanced purpose driven embedded systems using ARM and Intel based architectures are losing tremendous amounts of performance for their applications because they are running on operating systems that manage their memory and schedule their processor time for them.

Caret syntax is based on C syntax, enjoying the simplicity of braces for code blocks and the beauty of semi-colons denoting the end of a line. Caret is heavily influenced by the Go programming language (also known as golang) and Java. Go is a relatively new language from Google designed by Robert Griesemer, Rob Pike and Ken Thompson.

Some differences between Caret and its predecessors include:

- Supports Java style inheritance but with multiple inheritance capability.
- Conceptually similar to Goroutines in Go, Caret has 'Carets' which provide for an extremely easy to use concurrency pattern, without the need for an operating system.
- Automatic creation of gettors and settors.
- Modified Java try-catch structure in which general Exceptions do not need to be declared by default.

The Caret complier actually outputs ISO/IEC standard C++ code that is then compiled to the correct platform using gcc or other C++ complier. This ensures that Caret can enable developers to immediately start using the advantages of Caret on the almost countless platforms that already have a C++ compiler.

1.1 Hello world

Simple "Hello World" example in ^

```
/* This is the hello world program
 * author: brian.gormanly
 */

package hello;
import {
     IO;
}

^ hello() {
     IO.println("Hello World!");
}
```

1.2 Program structure

The key organizational concepts in Caret are as follows:

- Programs can be written to run in multiple programming paradigms such as object oriented and structured.
- Programs in Caret are strongly typed. Type safety guarantees the run-time behavior of the program and allows for better memory management.
- Intuitive concurrency and communication, managed by creating “Carets” and using the scheduler library to manage them.

Next is an example program that shows how to use the Scheduler to manage 2 Carets. One beeps every 3 seconds for five seconds, ending with a longer beep on the last iteration. The other process checks the state of a switch every 100ms and writes the states to standard output. Manipulation of the scheduler is shown both as overriding the super classes timing() method that ensures all changes made in this method are made before the Scheduler starts the Carets, and also in the makeSound() method with the direct reference to the super.getCaret() to check the count of the number of times the Caret has run.

```
/**
 * Sample program to access Arduino digital ports
 * using Arduino language library.
 *
 * @author brian.gormanly
 *
 * Using multiple inhertance, This program also
 * demonstrates using the Caret scheduler to
 * manage processes, and the Caret Garbage
 * Collector.  In this case the GC works
 * with default configuration, the scheduler
 * is configured per the notes below.
 *
 */

package gormanly.arduino

import {
        IO
        Arduino
        Scheduler
        GCollector
}

class BasicArduinoExample() extends Scheduler, GCollector {

        // set the hardware pin values
        const int buttonPin = 2;
        const int piezoPin = 9;

        // Caret to get the state of the switch attached to digital port 2 every 100 ms.
        ^getSwitch(100)
        getSwitchSetting() {

                buttonState = digitalRead(buttonPin);
                IO.println("The swtich is currently :" +
                        if(buttonState == ButtonState.HIGH) ? "On" : " Off"));
        }

        // Caret to make a sound on a piezo speaker attached to pin 3 every 3 seconds.
        ^makeAnnoyingSound(3000)
        makeSound() {
                // duration of really annoying sound is 200ms
                beep(200);

                // on the 5th (last) call of this method make the noise longer
                if(this.getProcs(makeSound).runCount == 5) {
                        beep(3000);
                }

        }

        /*
         Override the super class scheduler timing method to make some tweaks to default
         Scheduler settings.

         The Caret member of the Scheduler class does not need to be modified in the
         timing method, but doing so ensures that the changes are in place before
         any processes are created.
        */
        @Override
        private timing() {
                super();

                // do not run the getSwitchSetting in the first 2 seconds.
                this.getProcs(getSwitch).startdelay(2000);

                //Only do the really annoying sound 5 times
                this.getProcs(makeAnnoyingSound).repeat(5);
        }
}
```

1.3 Types and variables

Caret is a statically typed language and it requires that all variable and expressions have a known type at compile time.

As in Java types in Caret can either be primitive types (value) or reference types.

Primitives in Caret:

- boolean : 0 or 1
- byte : from -128 to 127, inclusive
- short : from -32768 to 32767, inclusive
- int : from -2147483648 to 2147483647, inclusive
- long : from -9223372036854775808 to 9223372036854775807, inclusive
- char : from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535
- float
- double

Also as in java primitives start with a lower case letter. Reference types will start with an uppercase letter and are camel case by convention.

2. Lexical Structure

2.1 Programs

A Caret program source is created in Unicode formatted files that end with a .caret extension. Like java, source files are arranged in folder structures that match the package the source file belongs to.

2.2 Compilation Process

The compilation process for Caret is similar to the Go programming language compilation process. It has a custom front end compiler that translates the code to C++ and then it uses the gcc compiler as the back end. This model ensures that Caret can be compiled on any platform where C++ is currently compiled. For scripting or structural programs there is a complier switch (-oc) to save the caret compiler output as c source files that can be compiled with other c compilers if gcc is not available on a platform. There are also a compiler switch (-ocpp) to same the front end output as C++ file(s), and a –omc for lightweight C compilers such as avr-g++. As an added benefit, this model provides all the optimizations implemented in GCC over the years are available, including inlining, loop optimizations, vectorization, instruction scheduling, and more.

2.2 Grammars

The grammar of Caret is compact and regular, allowing for easy analysis by automatic tools such as integrated development environments.

2.2.1 Basic Lexical grammar

letter   = unicode_letter | "_" .
decimal_digit = "0" … "9" .
octal_digit   = "0" … "7" .
hex_digit   = "0" … "9" | "A" … "F" | "a" … "f" .

identifier = letter { letter | unicode_digit } .

2.3 Comments

Line comments : //
General comments : /* */
comments do not nest

2.4 Tokens

There are four classes: identifiers, keywords, operators, and literals. Carriage returns and new lines are ignored. Semi-colons are required terminators for lines.

Identifiers
Identifiers must start with a letter, the first letter can then be followed by any letter, number or the following characters (-,_)

Keyword List:
- ^ (caret)
- break
- case
- class
- const
- continue
- default
- else
- extends
- for
- fun
- if
- import

interface
package
private
protected
public
range
return
struct
super
switch
this
var

Operators:
+, &, +=, &=, &&, ==, !=, (,), -, |, -=, |=, ||, < , <=, [,], *, ^, *=, ^= , <-, >, >=, {, }, /, <<, /=, <<=, ++, =, ,, ;, %, >>, %=, >>=, --, !, ..., ., :, &^, &^=

Literals:

```
int_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" … "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .
```

3. Example Programs

```
/**
 * Caesar cipher writen in ^
 *
 * Methods in this example have return values (void is assumed
 * if no return type is given in the method signature.
 *
 *
 * @author brian.gormanly
 */

package gormanly.cipher

import {
        IO
}

/**
 * This example uses only one Caret that runs to the end and terminates
 * if a class is chosen as the Caret the constructor is entry point for
 * the thread.
 */

^startCipher
class Cipher {

        Cipher() {
                String org = "This is a test of the emergency broadcast system.";
                IO.println("Original: " + org);
                String encrypt = Cipher.encode(org, 7);
                IO.println("String encrypted: " + encrypt);
                String decrypt = decrypt(encrypt, 7);
                println("String decrypted back: " + decrypt);
                println("Decode encrypted string:");

                for(i;<=26;1) {                // shorthand for (int i=0; i<26; i++)
                        println("iteration " + i + ": " + decode(encrypt, i));
                }
        }

        private String decrypt(String enc, int offset) {
                return encode(enc, -offset);
        }

        private String encrypt(String enc, int offset) {
                offset = offset % 26 + 26;
                String encoded = "";
                for (char i : enc.toLowerCase().toCharArray()) {
                        if (Character.isLetter(i)) {
                                int j = (i - 'a' + offset) % 26;
                                encoded = encoded + (char) (j + 'a');
                        } else {
                                encoded = encoded + i;
                        }
                }
                return encoded.toString();
        }
}
```

This example is a multi part one that starts with a simple PO^O or Plain Old Caret Object that is communicated between two separate Caret programs running on two different hardware platforms with a serial communication line between them.

First the PO^O:

```
/**
 * PO^O (Plain old Caret Object) that manages the
 * data collected by the robot and is used to convey
 * the information between the micro-controller and
 * the larger computational system.
 *
 * Note that getters and setters are not required they
 * are build automatically at compile time.
 *
 * @author brian.gormanly
 *
 */
package gormanly.robot.model

class RobotData() {
      int heading = 0;
      int pingDistance = 0;
}
```

This is the program running on the micro-controller. It is responsible for collecting sensor data, maintaining the data in the local robotData instance and then transmitting that data every second to the larger platform that can run more intensive algorithms on it.

```
/**
 * Microcontroller code that collects basic sensor
 * data and sends the data to a larger system over
 * a serial connection.
 *
 * @author brian.gormanly
 *
 */

package gormanly.robot

import {
      IO
      Scheduler
      GCollector
      Serial
      Serialize

      sensors.Compass
      sensors.Ping

      gormanly.robot.model.RobotData
}
```

```
class HardwareManager() extends Scheduler, GCollector {

      // set the hardware pin values
      const int compassPin = 2;
      const int pingPin = 3;

      // instance of RobotData
      RobotData myData = new RobotData();

      // instances of sensor management objects
      Ping ping = new Ping(pingPin);
      Compass compass = new Compass(compassPin);

      // get the compass heading value every 1/10 second
      ^ getHeading(100)
      getHeading() {

        // get the compass reading and add 500 to it
        myData.setHeading(compass.read());
      }

      ^ getDistance(40)
      getDistance() {

            // get the distance in cm read by the ping sensor
            myData.setPingDistance(ping.read());
      }

      // send data to the computational program every 1 second
      ^ sendData(1000)
      void taskTransmitData(void)
      {
            // serialize the my Robot oject and send to the
            // computational program.
            byte[] serialTransmission = Serialize(myRobot);
            Serial.println(serialTransmission);
      }
}
```

Here is the larger platform program, for brevity I have stubbed the actual algorithm being preformed on the data. The important part here is that a simple cast to the RobotData class retrieves the collected serial data.

```
/**
 * Microcontroller code that collects basic sensor
 * data and sends the data to a larger system over
 * a serial connection.
 *
 * @author brian.gormanly
 *
 */
```

```
package gormanly.robot

import {
      IO
      Scheduler
      GCollector
      Serial

      gormanly.robot.model.RobotData
}

class CoolAlgorithm() extends Scheduler, GCollector {

      // local robot data instance
      RobotData myData = new RobotData();

      ^retrieveRobotData(1000)
      retrieveData() {
            int bytes; // bytes returned from read()
            bytes = Serial.read();

            // try to cast the data collected into myData
            // note the general exceptions (Excepton in java)
            // does not need to be explicitly defined.
            try {
                  myData = (RobotData) bytes;
            }
      }

      ^doSomethingCool(1000)
      doSomethingCool() {
            // does sometihng really cool with the data we collected
            // in myData

      }

      /*
       Override the super class scheduler timing method to make some
tweaks to default
       Scheduler settings.

       The Caret member of the Scheduler class does not need to be
modified in the
       timing method, but doing so ensures that the changes are in
place before
       any processes are created.
      */
      @Override
      privat timing() {
            super();

            // we want to make sure that the doSomethingCool caret
            // waits for the completion of the retrieveRobotData caret
            this.getProcs(doSomethingCool).syncAfter(retrieveData);
      }
}
```