PlugCode 1.0alpha Reference Guide

Sergio Davis

10 de octubre de 2013

1. What is PlugCode?

PlugCode it is a *dialect* of C programing language, developed to write *plug-ins* of **lpmd** software package. PlugCode it is a *superset* of C, this mean that any C routine it is allowed automatically as a valid PlugCode subroutine.

PlugCode add C facilities as macros, clousures (in a limited way), and allow a very clear aand optimized syntaxis over vector operations, without the poor performance usually related to high level languages like C++. These characteristics are possible because PlugCode is internally tranformed to traditional C code[1] during compiling process.

```
@slot Evaluate (reader)
{
 Vector * vel = GetArray("vel");
 double ti = 0.0e0;
 for (long i=0;i<size;i++)
 {
  ti += 0.5*M*SquareModule(vel[i]);
 }
 ti *= KIN2EV;
 ti /= ((3.0/2.0)*KBOLTZMANN*totalsize);
 return ti;
}
```

Figura 1: “slot” example wrote at PlugCode.

[1]To be accurate, traditional C means the standard C99, that incorporate comments with `//` and declare allowed inside `for` and `while` cicles.

```
void Evaluate(const RawAtomSet * aset, const RawCell * cell)
{
 double * vel = NULL;
 ASet_GetArrays(aset, NULL, &vel, NULL, NULL, NULL, NULL);
 double ti = 0.0e0;
 long size = ASet_LocalSize(aset);
 long totalsize = ASet_TotalSize(aset);
 for (long i=0;i<size;i++)
 {
  ti += 0.5*M*(pow(vel[3*i],2.0)+pow(vel[3*i+1],2.0)+pow(vel[3*i+2],2.0));
 }
 ti *= KIN2EV;
 ti /= ((3.0/2.0)*KBOLTZMANN*totalsize);
 ASet_ReturnValue(aset, RET_DOUBLE, &ti, 1);
}
```

Figura 2: The same example of figure (1) but in C.

To more details about "slots" en **lpmd** 0.7, refeer to the document "Scheme-Design of LPMD 0.7".

2. Using `Allocate` and `Deallocate`

PlugCode incorporate the ™Allocate and `Deallocate` functions as replaces of `malloc`, `realloc`, and `free` used to request and free dynamic memory (as the equivalent `new` and `delete` in C++.

The syntaxis of Allocate are :

```
T * Allocate (T, long N)
```

the instruction make a memory request for an array of size N of elements of type T, and return the pointer to the begining of that memory. It is similar to the `malloc` call, in traditional C, however `Allocate` keep a track of the requested memory, avoiding memory leaks produced by the code. And :

```
T * Allocate (T, long M, T * pointer)
```

modify the size of the memory block pointed to *pointer*, in order to acommodate the size to M components of the type T. This work in a similar way than `realloc` in traditional C, but avoiding memory leaks.

The syntaxis of Deallocate is :

```
void Deallocate (T * pointer)
```

this syntaxis free the memory block pointing to the *pointer* address. It is similar to traditional C, however this keep a tracking bout the memory block (if it was free previously), avoinding segmentation fauls by the use of `free` twice.

3. The `Vector` data type

The `Vector` data type is an aggregation of three real values (`double` type) sequential in memory. To different uses, is equivalent to declare as `typedef double Vector[3]`.

4. The `VectorLoop` loop

The `VectorLoop` directive is used to create computation on different components of a vector in a implicit way. As an example if a and b are Vector type variables and we want to assign to b the value of $a \times f$ with f a scalar value, then :

```
double f = 20.0;
VectorLoop { b = a*f; }
```

that is equivalent in traditional C to :

```
double f = 20.0;
for (int q=0;q<3;++q) { b[q] = a[q]*f; }
```

It is possible use a combination of `Vector` with `VectorLoop` to set the vector componentes, as in this example :

```
Vector v;
VectorLoop { v = Vector(1.0, 2.0, 3.0); }
```

That is equivalent to (in traditional C) :

```
Vector v;
v[0] = 1.0;
v[1] = 2.0;
v[2] = 3.0;
```

5. Using `@define`

The `@define` instruction, can be used to define constant values or small functions, doing the job of the `#define` macros of standard C. The next example show the use of `@define` for both cases.

```
@define KBOLTZMANN = 8.617E-05
@define Suma(x, y) = (x+y)

@slot Test(reader)
{
 double x = Suma(5.0, 3.0*KBOLTZMANN);
}
```

6. Using @macro

The @macro instruction define a macro, this mean, a block code (with a particular name) in which no one to many parameters can be expanded when they are called.The macros can't return results as functions, despite of course it is always possbiel return values using specific output parameters. The next example use a macro to make equal to vectors $v1$ and $v2$.

```
@macro Igualar(v1, v2)
{
 VectorLoop { v1 = v2; }
}

@slot Prueba2(reader)
{
 Vector a, b;
 Igualar (b, Vector(1,2,3));
 Igualar (a, b);
}
```

7. Simulating value capture by using @define and @macro

It is possible emulate *clousures* using @define and @macro. Because both can be expanded as text, it is possible reference those as "hanging" variables (*unbounded*) that are not defined until they are called. This is a typical examplo of clousure in in which @define a function that add $a = 10$ to the argument.

```
@define SumaDiez(x) = (x+a)
int a = 10;
int y = SumaDiez(5);
assert (y == 15);
```

```
a = 20;
y = SumaDiez(5);
assert (y == 25);
```

Please note that `SumaDiez` is not a genuine clousure, because: first, it is not an object but a expanded macro in compiling time (as a template in C++); second, the *a* value is not captured inside the function, but take an actual *a* value of the previous scope. By changing the value of *a* to 20, `SumaDiez(x)` change the result.

8. References of PlugCode

8.1. Data types

Native (C heritage):

- `int`
- `long`
- `float`
- `double`
- `bool`
- `char`

Emulated:

- `Vector`
- `Matrix`
- `AtomPair`
- `NeighborList`
- `AtomSelection`

"Builders"[2] of emulated types.

- `Vector Vector(double x, double y, double z)`
- `Matrix Matrix(int columns, int rows)`

- `NeighborList NeighborList(AtomPair * pairs, long n)`
- `AtomSelection AtomSelection(long * indices, long n)`

8.2. Memory control functions

- `T * Allocate(T, long N)`
- `T * Allocate(T, long N, T * p)`
- `void Deallocate(T * p)`

8.3. Functions that involve the use of `Vector`

- `const char * VectorFormat(const char * format, Vector v)`
- `double Module(Vector v)`
- `double SquareModule(Vector v)`
- `double Dot(Vector v)`
- `Vector Cross(Vector a, Vector b)`
- `Vector Unitary(Vector v)`

8.4. Functions of random number generation

- `double Random()`
- `Vector RandomVector()`

8.5. Functions related to a simulation cell

- `double Distance(Vector dr, Vector pi, Vector pj)`
- `void Fractional(Vector cart, Vector frac)`
- `void Cartesian(Vector frac, Vector cart)`
- `void CenterOfMass(Vector cm)`

[2]Strictly speaking C doesn't have a builder concept, t́his are more like "factory functions".

- `int InsideNode(Vector v)`
- `void AddAtom(const char * names, ...)`
- `void * GetArray(const char * name)`
- `void GetArrays(const char * names, ...)`
- `void SetArray(const char * name, long n, void * p)`
- `void SetArrays(const char * names, long n, ...)`
- `void * GetTotalArray(const char * name, long a, long b, int sorted)`
- `void GetTotalArrays(const char * names, long a, long b, int sorted, ...)`
- `int HasTag(const Tag tag, int key)`
- `void SetTag(Tag * tag, int key)`
- `void UnsetTag(Tag * tag, int key)`

8.6. Debug and unit test

- `void LogMessage(const char * format, ...)`

8.7. Special variables availables in a “slot”

long size : Local size of atoms.

long totalsize : Total size of átomos.

long extsize : Extended size of atoms.

int masternode : 1 for masternode 0 otherwise.

NeighborList neighborlist : Precalculated neighbor list.