C PList Library API

Version 1.2

Files: (payload linux)

coda-c_plist.h	// C Library Header
coda-c_strings.h	// Optional String Functions
libcoda-c_plist.so.1.2	// Dynamic Library
libcoda-c_plist.so.1	// soname link
libcoda-c_plist.so	// linker link
codalist	// Command line utility
codalist.1coda-c	// Utility man page

www.coda-c.com

Credits

I'm deeply thankful to Lynn W. Keys, my lifelong partner, who's assistance made the development of the underlying software possible.

I would like to thank Brian W. Kernighan and Dennis M. Ritchie, for providing an elegant and powerful language (C) that has inspired me and provided the means to act on that inspiration.

I would also like to thank the undoubtedly large group of people who's contributions were distilled into the almighty Property List (.plist).

PList Serialization

```
Obj PList_Load(char *file,int flags);
int4 PList_lastLoadType(void); // (int4) aka (int32_t)
Obj PList_save(char *file,Obj container,int flags);

Obj PList_FromBlock(int count,pointer block,int flags);
   // (pointer) aka (void*)
Obj PList_toStream(Obj stream,Obj container,int flags);
```

Notes: The argument (Obj stream) will accept (FiLE* stream).

To understand the prototypes above, the C type (Obj) is the same as (void*), but it represents a generic object from the Coda-C object system where objects are pointers allocated memory+offset. The offset allows space for Class, Size, and Retain Count.

After loading a plist the function PList_lastLoadType(void) can be called to get the type of data loaded: 0 or PLIST_Coda_C for codalist, PLIST_Apple, PLIST_Binary or PLIST_Json. Loading functions return the new object or NULL for error.

PList Serialization (+JSON)

```
Obj Json_Load(char *file,int flags);
int4 Json_lastLoadType(void);
Obj Json_save(char *file,Obj container,int flags);

Obj Json_FromBlock(int count,pointer block,int flags);
Obj Json_toStream(Obj stream,Obj container,int flags);
```

These five functions add JSON support to the "PList" versions, making this the preferred interface.

```
enum {
   PLIST_Apple=16,
   PLIST_Binary=1<<11,
   PLIST_Json=1<<13,
   PLIST_Coda_C=1<<10,
   PLIST_UnsortedDict=1,
   PLIST_NL4Leafs=2,
   PLIST_AddComputer=4,
   PLIST_Amp38=8,
   PLIST_NoEncoding=32,
   PLIST_NoDoctype=64,
   PLIST_NoPVersion=128,
   JSON_NoEscapeSlash=512,
   JSON_Pretty=1<<14,
   PLIST_Strict=1<<15,
   PLIST_ObjectStream=4096,
   };
```

When serializing by saving or streaming data, set one of Apple(XML), Binary, Json, or Coda_C(XML) flags. Most of the remaining flags just alter the XML output in minor ways, allowing easier comparison with existing files. JSON output may be compact or pretty, and with or without "/" escaped with "\". PLIST_Strict causes non-standard conversions for <uid>, <data>, <date>, and <null> to be disabled. PLIST_ObjectStream is described later in this document. These functions return 0/NULL for errors, anything else means ok. Note: Binary I/O requires an additional compatible library plugin.

Error messages may be retrieved with the OError() function.

```
Char OError(void); // return the last error recorded
```

PList Classes

Apple XML plist format defines the following classes of objects: <dict>, <array>, <bool>, <integer>, <real>, <string> <date>, <data>, and <uid>. These map directly to the Coda-C classes: Array, Dictionary, Bool, Huge, Real, Char, DateString, Data, and HugeUID. Inspect the coda-c_plist.h header for CodaClassDef() that define the underlying C type. For example: CodaClassDef(Real,double,Root); produces: typedef double* Real;

Note: Apple encodes <uid> as a single key(CF$UID) dictionary with an <integer>.

Creating Objects

All objects may be created with newO().

```
// prototype // Obj newO(class-name);
Real rr=newO(Real);
```

When creating dynamic containers, Array or Dictionary, additional functions are used to populate the containers. When creating simple leaf types: Bool, Huge, Real, and HugeUID, normally you will want to assign a value.

```
Huge hh=newO(Huge); *hh=45; // or hh[0]=45;
```

Value functions are provided create an object with a single value.

```
Bool Bool_Value(bool value);
   // Note: typedef long long huge; // 8 byte integer
Huge Huge_Value(huge value);
HugeUID HugeUID_Value(huge value);
Real Real_Value(double value);
```

Objects that require a C-array of values are created with newOC() -- new object with count.

Char or string

```
   // prototype // Obj newOC(class-name,int count);
   char *string="Hello Coda-C";
   Char ss=newOC(Char,strlen(string)+1);
   strcpy(ss,string);
// OR
   // prototype // Char Char_Value(const char *string);
   Char ss=Char_Value("Hello Coda-C");
// OR even
   // prototype // Char $FORMAT12 Char_F(char *cs,...);
   Char ss=Char_F("Hello %s","Coda-C"); // printf style
```

Data

Data objects are created from a block of memory.

```
   Data dd=newOC(Data,5);
   memcpy(dd,"Hello",5);
// OR
   // proto // Data Data_NewBlock(Data self,int count,pointer address);
   Data dd=Data_NewBlock(0,5,"Hello"); // Note: self is always 0.

   // Note:  The contents of a file may be loaded with:
   // prototype // Data Data_FromFile(char *path);
   // but may return NULL or error: See: OError().
```

DateString - subclass of Char

Dates are stored as a highly formatted string, with 20 characters.
"yyyy-mm-ddThh:mm:ssZ" or "2013-08-13T07:00:50Z"

```
      // include <time.h>
   DateString DateString_Value(char *string); // not validated
   DateString DateString_FromGmtime(struct_tm *when); // not validated
   struct_tm* DateString_toGmtime(DateString self); // basic checks
   pointer Gmtime_check(struct_tm *self); // basic checks, NULL--error

   DateString ds=DateString_Value("2013-08-13T07:00:50Z");

   time_t now=time(0);
   struct_tm *gmnow=gmtime(&now); if (!gmnow) abort();
   DateString date=DateString_FromGmtime(gmnow); if (!date) abort();
```

ConstChar - subclass of Char

These are a special object version of a string constant that are read-only and immune to retain counting. For plists these are equivalent to Char.

```
   // prototype // ConstChar Os("String");
   Char color=Os("Green");
```

Short, Int4

Short and Int4 objects are used in unicode processing and not supported for plists.

```
Int4 Int4_Value(int4 value);
Short Short_Value(short value);
```

FileMem

FileMem objects are used to process in memory streams.

```
FileMem FileMem_Open(Data optional); // create stream
Data FileMem_ToData(FileMem self); // convert stream to data

// example usage: "serializing to memory"
cleanO FileMem fff=FileMem_Open(0); if (!fff) return(0); // NEVER
if (!Json_toStream(fff,container,flags|PLIST_ObjectStream)) return(0);
Data data=FileMem_ToData(fff);
```

Note: Json_toStream() or PList_toStream() both accept (FILE*). Setting the flag PLIST_ObjectStream causes the 'stream' to expect an 'Object Stream".

Object Lifetime

```
void freeO(Obj obj);
void keepO(Obj obj);
int countO(Obj obj):
cleanO // an attribute for variables
```

Once you have created an object, at some point you might want to destroy it to prevent memory leaks, this is done with freeO(). All objects are retain counted so they may be shared throughout a program. To increase the retain count just call keepO(), but each keepO() should be balanced with a freeO(). For diagnostic purposes you can get the retain count with countO(), and unlike Objective-C, this value is an accurate value, because there is no garbage collection.

The 'cleanO' attribute, used with a variable, causes the variable to be freeO()'ed when it goes out of scope. This attribute also makes the variable 'const' preventing changes. For example:

```
{  Bool flag=Bool_Value(1);
   printf("value is %d\n",*flag);
   freeO(flag);
   }
// OR
{  cleanO Bool flag=Bool_Value(1);
   printf("value is %d\n",*flag);
   }
```

Object Properties

```
Char kindO(Obj obj);
int  sizeO(Obj obj):
bool isa_(Obj obj,class-name);
```

To get the name of an object use kindO(). To get the size of an object use sizeO(). Check an object's class with isa_(), like isa_(var,Array). For example:

```
Real values=newOC(Real,5);
printf("allocation size of %s[5] is %d\n",kindO(values),sizeO(values));

if (isa_(values,Real)) printf("Yes\n");

// output:  "allocation size of Real[5] is 40" "Yes"
```

Note: When using isa_() with (Char), it will report true for classes: Char, ConstChar, & DateString.

Containers & Supporting Classes

Array

```
int Array_get_count(Array self);
Obj Array_addObject(Array self,Obj obj);
void Array_takeObject(Array self,Obj $CONSUMED obj);
Obj Array_subInt(Array self,int ix);
Obj Array_insertAt(Array self,int index,Obj obj);
void Array_removeAt(Array self,int dix);
void Array_removeLast(Array self);
Obj Array_replaceAt(Array self,int ix,Obj obj);
int Array_removeObject(Array self,Obj object);
void Array_removeAll(Array self);
pointer Array_rawAddress(Array self);

Array Array_NewBlock(Array proto,int count,pointer block);
void Array_insertBlock(Array self,int index,int count,pointer block);
void Array_removeBlock(Array self,int dix,int count);
void Array_takeBlock(Array self,int index,int count,pointer block);

void Array_toaSet(Array self); // Marks an array as a <Set>
bool Array_isaSet(Array self); // true if the array has been marked
```

When objects are added to an Array they are retained by the array unless the word 'take' is in the function name. When objects are removed or the Array is destroyed, they are released with freeO(). The function 'removeObject' removes all occurrences of the specified object by address equality. The function 'rawAddress' allows direct access to the internal array, but this address becomes invalid with any Array modification. When creating a new Array with 'NewBlock' the 'proto' should be 0 (reserved for future use). The function 'subInt' returns NULL for out-of-bounds.

Pointer

```
Pointer Pointer_Value(pointer value);
void pointer_sort(pointer base,int nel,void *IfunVVC,void *context);
// ex: pointer_sort(list,Pointer_count(list),strcmp,0);
```

Pointer class objects are used for programming support. For example Dictionary_AllKeys() returns a C-arrayed list of pointers, that can easily be sorted.

Keyword

```
struct Keyword_ { Obj item; char word[0]; };
Keyword Keyword_Value(char *word,Obj item);
```

The Keyword class exposes it's internal structure because it may be used to create Keyword objects or access internal Dictionary elements, that are not objects!

Dictionary

```
int Dictionary_get_count(Dictionary self);
void Dictionary_setKey(Dictionary self,char *key,Obj obj);
void Dictionary_takeKey(Dictionary self,char *key,Obj $CONSUMED obj);
Obj Dictionary_subKey(Dictionary self,char *key);
bool Dictionary_removeKey(Dictionary self,char *key);
int Dictionary_removeObject(Dictionary self,Obj obj);
Pointer Dictionary_AllKeys(Dictionary self);
Keyword Dictionary_scan(Dictionary dict);
Keyword Dictionary_next(Keyword element);

// Example scan of dictionary
for(Keyword key=Dictionary_scan(dict);key;key=Dictionary_next(key)) {
   printf("Key=%s, item=%p\n",key->word,key->item);
   }
```

The Dictionary class objects are referenced by a key string, which is copied into the dictionary. This class follows the same rules for retain retaining contained objects as Array.

Miscellaneous

Void Class

```
Obj alocO(int size);
```

Void class objects are 'classless' objects, they allow arbitrary data to work as objects, so they can be contained in Dictionary and Array. They are created with alocO().

Other Stuff

The Root class is a pseudo class, that other classes derive from.

```
int Root_get_count(Root self); // # of C-arrayed elements in an object
   // Not to be used with containers: Array and Dictionary

Char obj_xmlTag(Obj obj); // return the XML Tag for an object or NULL
Char obj_ToString(Obj obj); // a new string version of object's value
Char obj_FromString(Obj cobj,char *string);
   // A new object from a string or NULL w/ OError().
   // The new objects will have the same class as 'cobj'.

Void JsonNull_Value(void); // create a Dictionary placeholder for "null"
bool isa_JsonNull(Obj obj); // check for placeholder
```

Each Coda-C class has a class object named: Class_(class name) like Class_Real. These are not functional objects, but can be used where only the class of an object is required, like 'cobj' in obj_FromString().

```
void PList_Binary(void); // Forward ref to Binary plist plugin library
```

The function PList_Binary() is not defined in this library, but if you have a Binary plugin, calling or referencing it will cause the linker to include the plugin.

```
 enum { UTF8max=0x10FFFF, }; // used as encoding for JSON \u0000 (null)
 #define sizeat(type) sizeof(*((type)0)) // sizeof pointer type!
```

Diagnostics

```
Char ToContainer(Obj container); // readable version of an 'object'
void codac_versions(void); // report version information to <stdout>
#define Object_Leaks() ... // macro for leak testing
```

The ToContainer() function tries to make a human readable string from an object, mostly for programming purposes. It is most useful with containers.

When the Object_Leaks() macro is inserted in a function, it creates a variable on that line, that when it goes out of scope will report the change in total object count. This process is not perfect, but it can help to hunt down leaking objects. For example:

```
#include "coda-c_plist.h"
int main() {
   Huge ival=0; ival=newO(Huge); // unreported leak
   Object_Leaks();
   Array aa=newO(Array);
      Array_add(aa,newO(Bool)); // found leak (Bool)
   Dictionary dd=newO(Dictionary);
      Dict_set(dd,"Key1",aa);
      freeO(aa);
   freeO(dd);
   }
// Reports: ">>> Leaks: 1 (end scope main:5)"
```

Short hand defines

```
// short hand for common functions and Dictionary
   #define Array_count Array_get_count
   #define Array_sub Array_subInt
   #define Dict Dictionary
   #define Dict_set Dictionary_setKey
   #define Dict_sub Dictionary_subKey
   #define Dict_take Dictionary_takeKey
   #define Dictionary_count Dictionary_get_count
   #define Root_count Root_get_count
   #define Pointer_count Root_get_count // get the number of pointers
```

XML Formats: plist vs codalist

The codalist XML format differs from plist XML format in the following ways:

1. Strings are C strings and are not required to be UTF-8.
2. Arrays may contain <null>'s.
3. C style comments are allowed but not preserved. //
4. <uid> has a native format.
5. <set> is supported by naming an array.
6. <keyword> is supported and may be used to preserve container names.

Special Encodings

Special encodings are used for unsupported types unless PLIST_Strict is set.

JSON:

<data> is encoded as { "CF$Data" : "BASE64 data..." }.
<uid> is encoded as { "CF$UID" : ival }.
<date> is encoded as { "CF$Date" : "YYYY-MM-DDThh:mm:ssZ" }.

Apple XML:

<null> is encoded as { "CF$Null" : true }.