Interlude: Property Lists

Property lists are an integral part of Darwin architecture, and are used system-wide as the file format of (no) choice to store key/value based information. Apple provides basic documentation on their use in the man page of `plist(5)`, and the CoreFoundation Guide[2]. The latter primarily focuses on the programmatic APIs used to manipulate them into `CoreFoundation` objects: Property lists are readily (de)serialized info (and out of) `CFDictionary` objects - their in-memory representation (e.g. `+[NSDictionary dictionaryWithContentsOfFile:]`) - and can assume several forms - XML, binary or JSON.

XML property lists

Property lists are another legacy of NeXTSTEP (though back then plain text was preferred) and their XML format has been attributed to Apple. The most common representation nowadays is XML, conforming to a grammar wherein the following always hold:

- The `!DOCTYPE` definition holds constant, with a document type definition pointing to http://www.apple.com/DTDs/PropertyList-1.0.dtd. The URL is valid and actually holds an XML DTD.
- The XML root element is always named `<plist>`. It is also the only element which may have attributes, and has a single attribute called `version` whose value is presently fixed at `1.0`.
- Other XML elements are `<key>`s, but since they are not allowed attributes, the element value serves as the key name, and the actual value follows, using one of the basic "datatypes" shown in table 2-3:

Table 2-3: The data types allowed in property lists

type tag	**`CF*` type**	**Holds**
string	`CFString`	Strings, either ASCII or Unicode
integer	`CFNumber`	Integer numbers
real	`CF...`	Arbitrary precision numbers (float)
date	`CFDate`	A 64-bit date
array	`CFArray`	An array of any other (usually, not necessarily single) data type
dict	`CFDictionary`	A nested key/value dictionary
true	`CFBoolean`	Boolean true/false, as value-less elements
false		

Apple discourages manual editing of property list files, explicitly warning that "the tags may change in future releases", and associating Xcode's Property List Editor with the file type. In practice, the format hasn't changed in generations, and textual editing often proves quicker, especially when validated by the `plutil(1)` utility. This useful command can be used to validate syntax (its default, or when used with `-lint`), and to convert between the various formats (using `-convert [xml1/binary1/json]`). It can also perform basic editing of the plist (using `-insert/replace/remove`). An additional interactive utility can be found in `PlistBuddy(8)`.

As an example of a property list, consider the telnet.plist launch daemon property list, as shown in Output 2-4. Note the use of `plutil(1)` here, which is used in order to dump the property list in XML form, rather than its native binary representation. Although `plutil(1)` can operate on property lists in place, it is generally a safer practice to have it write to standard output, as specified by `-o -` :

Output 2-4: Displaying an XML property list

```
morpehus@Zephyr (~)$ plutil -convert xml1 /System/Library/LaunchDaemons/telnet.plist -o -
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Disabled</key>
        <true/>
        <key>Label</key>
        <string>com.apple.telnetd</string>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/libexec/telnetd</string>
        </array>
        <key>SessionCreate</key>
        <true/>
        <key>Sockets</key>
        <dict>
                <key>Listeners</key>
                <dict>
                        <key>Bonjour</key>
                        <true/>
                        <key>SockServiceName</key>
                        <string>telnet</string>
                </dict>
        </dict>
        <key>inetdCompatibility</key>
        <dict>
                <key>Wait</key>
                <false/>
        </dict>
</dict>
</plist>
```

Binary property lists

While fairly readable, the XML format is cumbersome for machine-processing, requiring fairly heavy parsers and taking up a large number of bytes. It therefore makes sense to use a less human readable but far more machine friendly format of a **binary property list**. This format is used far more frequently in *OS, though XML property lists can still be seen in some cases. Binary plists are also found in MacOS, though not as commonly as XML ones.

bplist00

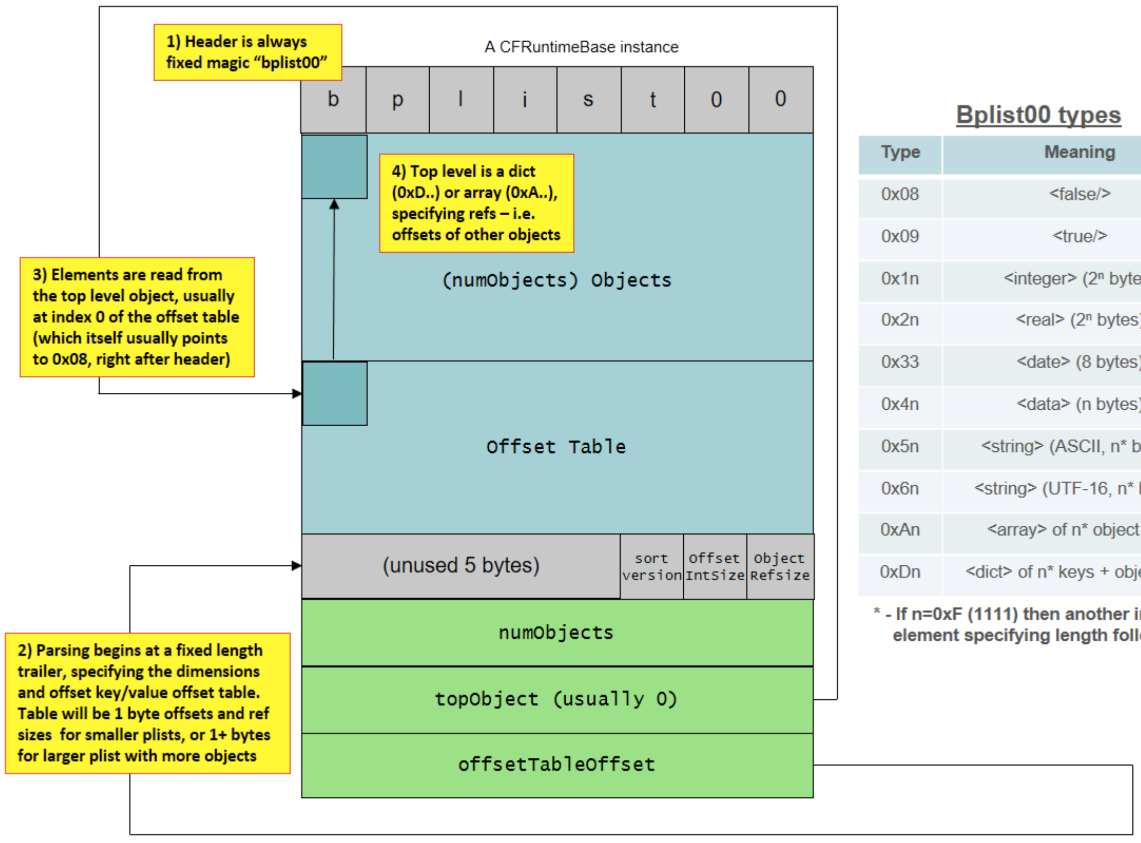
Binary property list files are identifiable by a magic of `bplist00` (In Base64, YnBsaX). The file format isn't officially documented by Apple, and regarded in the man page of `plist(5)` as an "optimized, opaque binary format". Output 2-5 shows the same telnet.plist example, this time in its natural binary form:

Output 2-5: Examining a binary property list

```
00000000  62 70 6c 69 73 74 30 30  d6 01 02 03 04 05 06 07  |bplist00........|
00000010  07 09 0c 13 15 58 44 69  73 61 62 6c 65 64 5d 53  |.....XDisabled]S|
00000020  65 73 73 69 6f 6e 43 72  65 61 74 65 5f 10 12 69  |essionCreate_..i|
00000030  6e 65 74 64 43 6f 6d 70  61 74 69 62 69 6c 69 74  |netdCompatibilit|
00000040  79 57 53 6f 63 6b 65 74  73 5f 10 10 50 72 6f 67  |yWSockets_..Prog|
00000050  72 61 6d 41 72 67 75 6d  65 6e 74 73 55 4c 61 62  |ramArgumentsULab|
00000060  65 6c 09 09 d1 0a 0b 54  57 61 69 74 08 d1 0d 0e  |el.....TWait....|
00000070  59 4c 69 73 74 65 6e 65  72 73 d2 0f 10 07 12 57  |YListeners.....W|
00000080  42 6f 6e 6a 6f 75 72 5f  10 0f 53 6f 63 6b 53 65  |Bonjour_..SockSe|
00000090  72 76 69 63 65 4e 61 6d  65 09 56 74 65 6c 6e 65  |rviceName.Vtelne|
000000a0  74 a1 14 5f 10 14 2f 75  73 72 2f 6c 69 62 65 78  |t.._../usr/libex|
000000b0  65 63 2f 74 65 6c 6e 65  74 64 5f 10 11 63 6f 6d  |ec/telnetd_..com|
000000c0  2e 61 70 70 6c 65 2e 74  65 6c 6e 65 74 64 08 15  |.apple.telnetd..|
000000d0  1e 2c 41 49 5c 62 63 64  67 6c 6d 70 7a 7f 87 99  |.,AI\bcdglmpz...|
000000e0  9a a1 a3 ba 00 00 00 00  00 00 01 01 00 00 00 00  |................|
000000f0  00 00 00 16 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  00 00 00 ce                                       |....|
00000104
```

Examining the CF's open source CFBinaryPList.c reveals a summary of the format in a comment, as well as some `struct` definitions. The format is shown in Figure 2-6:

Figure 2-6: The `bplist00` file format


1) Header is always fixed magic "bplist00"
A CFRuntimeBase instance
b p l i s t 0 0
4) Top level is a dict (0xD..) or array (0xA..), specifying refs – i.e. offsets of other objects
(numObjects) Objects
3) Elements are read from the top level object, usually at index 0 of the offset table (which itself usually points to 0x08, right after header)
Offset Table
(unused 5 bytes)
sort version
offset IntSize
object RefSize
numObjects
topObject (usually 0)
offsetTableOffset
2) Parsing begins at a fixed length trailer, specifying the dimensions and offset key/value offset table. Table will be 1 byte offsets and ref sizes for smaller plists, or 1+ bytes for larger plist with more objects

Bplist00 types

Type	Meaning
0x08	<false/>
0x09	<true/>
0x1n	<integer> (2^n bytes)
0x2n	<real> (2^n bytes)
0x33	<date> (8 bytes)
0x4n	<data> (n bytes)
0x5n	<string> (ASCII, n* bytes)
0x6n	<string> (UTF-16, n* bytes)
0xAn	<array> of n* object refs
0xDn	<dict> of n* keys + object refs

* - If n=0xF (1111) then another integer element specifying length follows

Applying the structure in Figure 2-6 on Output 2-5, note the following:

- The trailer starts at 0xe7 (In the output, highlighted with a dull background). This can be deduced from the fixed size of the trailer, and more easily identified by locating the "signature" of three unused zero values, then the sort version (0), Offset IntSize (1) and Object RefSize (1). This also shows the number of objects is 22 (0x16).
- The offset table is at 0xce (In the output, highlighted with a white background). This is indicated by the last field of the trailer. Each table entry in this particular case is (as per Offset IntSize) one byte. The table spans until the trailer, and the top level object (at offset 0 of the table, as indicated by the trailer, or 0xce of the file) is 0x08 - right after the `bplist00` magic.
- The element at offset 0x08 is a dictionary containing 6 key references (as indicated by 0xD6). Each reference is (as per Object RefSize) 1 byte, The references are at offset table entries 0x01 through 0x06, and their corresponding values are at 0x07, 0x07, 0x09 through 0x15 (which makes sense, as the number of objects is 0x16). Note that this allows for a simple form of compression, as value 0x07 (i.e. at offset 0x62 - 0x09, denoting `<true/>`) repeats and is therefore encoded only once.
- Values shorter than 15 bytes can be encoded in the four least significant bits of the type specifier (e.g. 0x58 ('X') for "Disabled", indicating a string (0x50) of 8 bytes. Longer values are encoded as 0x0F, and a separate integer (0x1.) follows. In this way, "ProgramArguments" (16 bytes) is encoded as 0x5F, followed by 0x10 0x10 (the first 0x10 specifying an integer with a length of 2^0 bytes, and the second being the actual length (1 length byte, specifying 16 bytes).

bplist15 and bplist16

There are at least two other variant of binary property lists, with a magic of `bplist15` and `bplist16`. Unlike `bplist00`, there is no documentation for either format, save for some scant comments in the aforementioned `CFBinaryPList.c` as to some supported types.

The `bplist15` format appears to be internal to `CoreFoundation`, and is not present in `CFLite`. It is rarely used, with one example being `ApplePushService.framework`'s topic subscription messages. The `bplist16` format is internal to `Foundation`, and is used almost exclusively in Objective-C remoting over XPC. This fact alone makes `bplist16` important to reverse engineer. The format is similar, but not compatible with `bplist00`, noting the following differences:

- No footer: `bplist16`s don't have a trailer, and the items start directly at the head of the property list, right after the `bplist16` magic, and are packed (not aligned).
- More data types: As the comment in `CFBinaryPlist.c` states, the 1+ formats allow for UUID, URL, UTF-8, sets, ordered sets and NULL values.

JSON property lists

With so many technologies incorporating web services, it is only natural for property lists to also be transferred over HTTP, and serialized into the JavaScript Object Notation (JSON) format. Though less used in the file format, it is not uncommon to see JSON-serialized plists over Apple's HTTP-borne protocols.

Figure 2-7: The JSON file format

```
# can also do this with simplist -j
morpheus@Zephyr (~)$ cat /System/Library/LaunchDaemons/telnet.plist | \
                      plutil -convert json -o - -
{"ProgramArguments":["\/usr\/libexec\/telnetd"],"Sockets":{"Listeners":{"Bonjour":true,
"SockServiceName":"telnet"}},"Disabled":true,"Label":"com.apple.telnetd","SessionCreate":
true,"inetdCompatibility":{"Wait":false}}
```

SimPLISTic format

The book's companion website offers the `jlutil(j)` tool, which is not only free, but is also available for *OS variants and even Linux. Another feature of this useful tool is presenting the property list in an alternate, simpler format, with or without color. Continuing the example of `telnet.plist`, in simPLISTic format it would look like:

Output 2-8: Displaying a property list in simpler form with `jlutil(j)`

```
morpheus@Zephyr (~)$ jlutil /System/Library/LaunchDaemons/telnet.plist
   Disabled: true
   SessionCreate: true
   inetdCompatibility:
      Wait: false
   Sockets:
      Listeners:
         Bonjour: true
         SockServiceName: telnet
   ProgramArguments[0]: /usr/libexec/telnetd
   Label: com.apple.telnetd
```

The simPLISTic format is clearer to read than cumbersome XML (or curly-braced JSON), and is entirely isomorphic to either. When symlinked or renamed to `plutil(1)`, `jlutil(j)` will emulate `plutil`, so it can be used as a drop-in replacement in any scripts. Note, that `jlutil(j)` can also parse `bplist16` (which the original `plutil(1)` cannot).