



# Smokey EFI Command Sequencer

## User and Technical Manual

---

Version “Black”  
October 17, 2012

**Authors:**

Chanh-Duy Tran

[ctran@apple.com](mailto:ctran@apple.com)

# Table of Contents

## Introduction

What is Smokey? .....	1
Audience .....	1
Document Objective .....	1

## Lua Fundamentals

Lua .....	2
Data .....	2
Comments .....	3
Boolean Expressions .....	4
Tables .....	5
Language Constructs .....	6
Built-in Facilities .....	8
Exceptions .....	9
Tricks .....	10

## Smokey Fundamentals

Sequences: Scripts and Property Lists .....	12
File Organization .....	12
Sequence Schema .....	13
Sequence Flow .....	17
Sequence Processing .....	18
Pass/Fail Criteria of Actions .....	19
Failure Handling .....	19
Test Results .....	20
Sequence State Saving .....	21

## Design for Factory Use

Objectives for Factory Use .....	23
Station Behavior .....	23
Control Bits .....	24
Log Collection .....	24
Test Results and Data .....	25

## Using Smokey

Command Line Arguments .....	26
Autostart .....	27
Sequence Output .....	29
Screen Output .....	33
State Control .....	33

## Developing Smokey Sequences

Smokey Lua API .....	35
Data Submission to PDCA .....	36
Sequence Development Quick Start .....	36
Developing Code with the Smokey Shell .....	38
Developing Code on DUT .....	38
Developing Code with the Smokey Simulator .....	38

## Smokey Internals

Software Architecture .....	39
Control Files .....	39
Saving State .....	39
Schema Grammar .....	39

## References

Useful Lua Links .....	40
------------------------	----

# Introduction

## What is Smokey?

Smokey is a scripting environment within EFI diagnostics on mobile platforms, including iPhones, iPads, iPods, and AppleTV. Its goals are to enable automated low-level testing and data collection across factory and engineering scenarios.

## Audience

This document is intended for the engineers involved with the development and use of the EFI diagnostic environment.

On the producer side, this manual addresses EFI diagnostic firmware engineers, the people who will maintain and enhance Smokey.

On the consumer side, this manual addresses factory and non-factory users. Factory users include station DRIs. Non-factory users include hardware engineers needing a platform for validation or stress testing.

## Document Objective

This document is generally intended to be an all-around reference, but written with the intent of bringing Smokey's core users up to speed about the environment.

On the producer side, this manual intends to educate software engineers about the functionality and intent of the system. The goal is to exposit the design and shed light on the considerations required to either maintain or enhance the software.

On the consumer side, this manual presents the features available in the Smokey environment. Additionally, time will be spent to document requirements, prerequisites, and expected behaviors. The goal is to arm users with the knowledge to bootstrap their own test sequences.

# Lua Fundamentals

## Lua

The Lua language is a product of the Tecgraf group at PUC-Rio (Pontifical Catholic University of Rio de Janeiro) and intended to be a powerful, fast, lightweight, embeddable scripting language. It has dynamic data types, built-in support for associative arrays, and offers automatic memory management. Scripts are internally translated into byte code that is executed by an internal virtual machine.

The name “Lua” is Portuguese for “moon”. Scripts in the Lua language have the extension “.lua” and are typically encoded in ASCII.

Lua is the adopted language of the Smokey environment. On top of the benefits listed above, Lua was chosen because both the interpreter and its built-in libraries are easy to port to the EFI environment without compromising their feature sets. Additionally, the language, code base, and community support are both mature and stable.

All interactions between users (i.e. scripts) and the platform (e.g. lower software stack, hardware drivers) will be in the form of Lua function calls, either direct or via tables, and Lua data types. Smokey provides custom APIs to make these interactions easier to manage.

Since it is not the intent of this document to teach Lua (see References section), the following is only meant to help kick-start seasoned programmers.

## Data

### Built-in Types

Lua supports 6 built-in types that are useful here.

- **nil** — Its only value is `nil`
- **boolean** — Its only values are `true` and `false`
- **string** — Doubly and singly quoted text such as `“foo”` and `‘bar’`
- **number** — Decimal integer (123), Hex (0x123), and floating point (10.5)
- **table** — Associative array between a key of any type and a value of any type
- **function** — An optionally named chunk of executable code declared by the `function` keyword

### Dynamism

Lua variables are dynamically typed. That is, they do not have a set type and will become the type of the value being assigned. Lua will attempt to “do the right thing” and cast between strings, booleans, and numbers.

There are no mechanisms to restrict types, there is an operator to inspect types.

## Scope

Variables in Lua are defined the moment they are used and exist until the end of the code chunk in which it was used.

Between nested code chunks, it is important to remember that Lua variables are lexically scoped. In the case that the same variable name is amongst different nesting levels, the nested chunks can ensure that they are using their own variable by applying the `local` keyword during variable declaration.

## Multiple Values

Lua supports assigning and returning multiple values at a time through use of the comma operator. Any missing value in a tuple is automatically assigned the value `nil`.

```
a, b = 1, 2    -- a = 1 and b =2  
c, d = nil, 3  -- c = nil and d = 3  
e, f = 4       -- e = 4 and f = nil
```

```
function foo ()  
    -- Return two values  
    return 1, 2  
end
```

## Comments

Lua comments are akin to C++ comments.

### Single-line Comments

Basic comments start with a double dash and run until the end of the line.

```
-- This line is not code
```

```
x = 3 -- Assign 3 to the variable 'x'
```

### Delimited Comments

```
--[[  
This line is not code  
Nor is this line  
--]]
```

```
y = --[[ These words are not code --]] 5
```

# Boolean Expressions

## Relational Operators

Lua has all the basic relational operators.

```
a > b  -- Greater than
a < b  -- Less than
a >= b -- Greater/Equal
a <= b -- Less/Equal
a == b -- Equal
a ~= b -- Not equal
```

## Logical Operators

Logical operators are spelled out.

```
(a < b) or (c < d)  -- Disjunction
(a < b) and (b < c) -- Conjunction
not (a < b)         -- Negation
```

## Conditionals

Any value that is neither `nil` nor `false` is considered to be a boolean false. This means that numbers, strings, functions, and tables are considered true. Counterintuitively to C/C++ programmers, this also means that the number 0 is considered true.

```
nil           -- False
false         -- False
true          -- True
0             -- True
1.5           -- True
"foo"         -- True
{ a = 5 }     -- True
function () end -- True
```

## Checking Existence

As a best practice, any code checking whether a variable is set should explicitly compare against `nil` rather than using the variable itself as the expression.

```
-- Good practice check for existence
if (x ~= nil) then
    foo()
end
```

```
-- Bad practice check for existence
if (x) then
    foo()
end
```

## Tables

Lua tables are associative hashes and are used through the language for both storage and also for program organization.

### Storage

Defining tables with implicit (i.e. numeric) keys.

```
t = { 10, 20, 30 }  
u = { "a", "b", "c" }  
v = { 10, "z", false }
```

Defining tables with explicit key-value pairs.

```
w = { [1]=10, [10]=20, [100]=30 }  
z = { a=1, b=2, c=3 }
```

### Access

Typical table access use familiar bracket notation.

```
a = t[1]  
b = t[a]
```

Keys that are strings have special syntactic sugar.

```
-- These two assignments are the same  
b = z["a"]  
c = z.a
```

Walking through a table can be done generally with a `for` loop and the *pairs* iterator. Note that traversal order is not guaranteed.

### Arrays

Lua does not have a built-in array type. Rather, it depends on tables whose keys are contiguously numbered integers. All Lua arrays start at index 1.

```
t = { 10, 20, 30 }  
foo(t[1]) -- The argument value is 10  
foo(t[2]) -- The argument value is 20  
foo(t[3]) -- The argument value is 30
```

Walking through an array can be done with a `for` loop and the *ipairs* iterator. Note that traversal starts with index 1 and ends when the next consecutive index is undefined.

### Classes

Lua doesn't have a class system per se, but it is possible to emulate their behavior using tables with keys that are strings and values that are functions.

Lua provides some syntactic sugar to make class methods easy to declare.



```
-- Declare method f() inside table t
t = {
    m = 1,
    f = function (self, a)
        return self.m + a
    end
}
```

```
-- Declare method f() inside table t
t = {
    m = 1
}
function t:f (a)
    return self.m + a
end
```

Calling a class method and passing in the “self” pointer is has syntactic sugar, too.

```
x = t.f(t, 1, 2) -- The hard way
```

```
x = t:f(1, 2)    -- Lua's syntactic sugar
```

The syntax `t:f()` implicitly passes `t` as the first argument to `f`. Note that it is possible and easy to mistakenly call `f` using the normal table access syntax of `t.f()` without any complaint from Lua. However, your code may misbehave because it will not have the correct arguments!

## Language Constructs

The Lua syntax will look familiar to any programmers of ALGOL-influenced languages like C/C++, Pascal, Python, or Perl.

### Functions

```
function foo (x, y, z)
    return x + y + z
end
a = foo(1, 2, 3)
```

```
bar = function (x, y, z)
    return x + y + z
end
b = bar(1, 2, 3)
```

## if-then-else

```
if (a > 0) then
    x = 1
end
```

```
if (a > 0) then
    x = 1
else
    x = 0
end
```

```
if (a > 10) then
    x = 2
elseif (a > 0) then
    x = 1
else
    x = 0
end
```

## Loops: for-do

```
-- Iterate from 1 to 10
for x = 1, 10 do
    foo(x)
end
```

```
-- Iterate on all table entries
for key, value in pairs(t) do
    t[key] = value + 1
end
```

```
-- Iterate on all array entries
for i, value in ipairs(t) do
    t[i] = value + 1
end
```

## Loops: while-do

```
-- Print powers of 2 less than 1000
-- that aren't foo()
x = 1
while (x < 1000) do
    x = x * 2
    if (foo(x)) then
        break
    end
end
```

## Loops: repeat-until

```
-- Add bar() while less than 1000 and
-- not foo()
x = 0
repeat
    x = x + bar()
    if (foo(x)) then
        break
    end
until (x > 1000)
```

## Built-in Facilities

### String Concatenation

Strings are joined with the double-dot operator.

```
x = "to" .. "get" .. "her"
```

### *string.format* Function

Lua supports a limited subset of the *printf* semantics from C.

```
x = 100
s = string.format("%dKhz", x)
```

### Bitwise Operations

Bit manipulation is available in the form of library calls rather than being syntactically integrated into the language like in C. Operations are limited to 32-bit quantities.

```
-- Assume: uint32_t x, a, b, n
--           int32_t k
x = bit32.band(a, b)      -- x = a & b
x = bit32.bor(a, b)       -- x = a | b
x = bit32.bxor(a, b)      -- x = a ^ b
x = bit32.bnot(a)         -- x = ~a
x = bit32.lshift(a, n)    -- x = a << n
x = bit32.rshift(a, n)    -- x = a >> n
x = bit32.arshift(k, n)   -- x = k >> n
x = bit32.lrotate(a, n)   -- x = (a<<n) & (a>>(32-n))
x = bit32.rrotate(a, n)   -- x = (a>>n) & (a<<(32-n))
if (bit32.btest(a, b))    -- if (a & b)
```

More complicated extractions and injections of fields are available as well.

```
-- m = ((1 << w) - 1)
-- x = (a >> n) & m
x = bit32.extract(a, n, w)
```

```
-- m = ((1 << w) - 1)
-- x = (a & ~(m << n)) | ((b & m) << n)
x = bit32.replace(a, b, n, w)
```

### **tonumber Function**

Lua will try to convert values to numbers implicitly in some scenarios, but it is often useful to do this manually.

```
x = tonumber("100")
```

### **tostring Function**

Lua will try to convert values to strings implicitly, but it is sometimes useful to do this manually.

```
x = 100
s = tostring(x) .. "KHz"
```

### **type Function**

The type of a variable can be inspected using the *type* function. It returns a string describing the type.

```
if (type(x) == "nil") then
    foo(1)
elseif (type(x) == "boolean") then
    foo(2)
elseif (type(x) == "string") then
    foo(3)
elseif (type(x) == "number") then
    foo(4)
elseif (type(x) == "function") then
    foo(5)
elseif (type(x) == "table") then
    foo(6)
else
    foo(7)
end
```

## **Exceptions**

### **Raising Exceptions**

The current context of execution can be aborted by raising an exception. Arbitrary data can be sent up as part of the exception; descriptive strings are helpful if the upper-most code can expect and print them.

```
function f (x)
    if (x > 100) then
        error("Too large")
    end
end
```

## Catching Exceptions

Exceptions can be caught by using Lua's *pcall* function to call code that is anticipated to raise exceptions. Note that the parameter needs to be a function name, not a function call.

```
pcall(f, 10) -- Safely execute f(10)
```

Exception handling will not be covered in detail here because Smokey will be doing most of this work. Attempts to intervene with exception processing many lead to unexpected behavior in Smokey.

## Tricks

### Data Alternation in Boolean Expressions

Lua's logical and and logical or operators do not return their results as a boolean type. Instead, they return the operand whose value evaluates to true.

```
x = true and 1    -- The value is 1
y = nil and true  -- the value is nil
```

```
a = 3 or true     -- The value is 3
b = nil or "foo"  -- the value is "foo"
```

In the case of functions that take optional arguments, this makes it easy to provide a default value if none is specified.

```
function f (x)
    -- If x is specified, return x + 1
    -- If not, return 100
    return (x or 99) + 1
end
```

### Ternary-Like Operator

Lua does not support the question mark operator in C and C++. However, the inline "if-then-else" semantics are partially supported by Lua boolean expressions.

```
-- Similar to:  
--   if (a) then  
--       x = b  
--   else  
--       x = c  
--   end  
x = a and b or c
```

The caveat is that the “then” part of the expression must not be false because that situation will cause Lua to use the “else” portion.

```
-- WARNING! Both are same as x = c  
x = a and nil or c  
x = a and false or c
```

The workaround is to ensure that the “then” portion is always true. This can be done by negating the conditional and swapping the “then” and “else” portions. If neither portion can be guaranteed to be true, the full “if-then-else” construct must be used instead.

# Smokey Fundamentals

## Sequences: Scripts and Property Lists

Smokey is an automated means by which to execute structured code in a configurable order. Its main inputs are two specifically named files—a **script file** and a **property list file**—in a user-defined directory. Together, the files form a **sequence**. The sequence takes its name from the directory name.

The script file shall have an extension of “.lua” to indicate that it is a Lua script file. Its contents shall be any content that would be accepted by a standard, standalone Lua interpreter. This file will define Lua functions that Smokey will execute as **actions** in the order defined by the property list file.

The property list file shall have an extension of “.plist” and be encoded as an Apple property list in ASCII format. This file will define the parameters of the sequence, including the order of actions defined in the script file, prerequisite and co-requisite conditions, as well as run-time Smokey behavior.

## File Organization

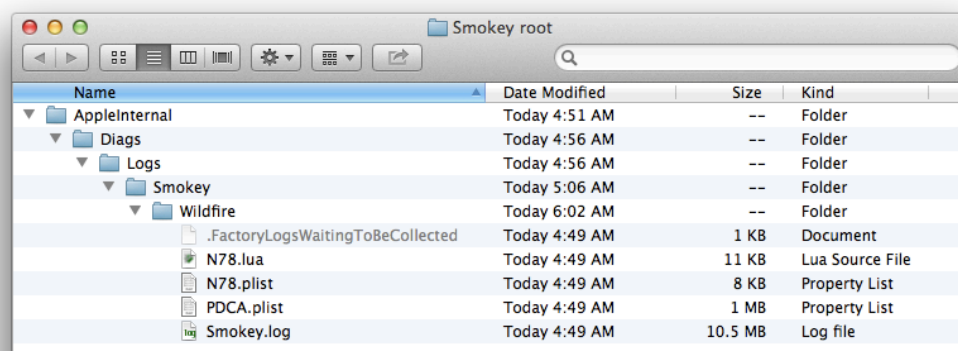
### Sequence Location

Smokey sequences are stored as directories on the DUT under the Smokey folder.

```
/AppleInternal/Diags/Logs/Smokey/Sequence
```

### Sequence Name

Sequences are named after the folder containing its files. For example, a sequence named “Wildfire” would be in a folder named “Wildfire” under the Smokey folder.



The screenshot shows a file browser window titled "Smokey root". The left sidebar displays a tree view of the directory structure: AppleInternal > Diags > Logs > Smokey > Wildfire. The main pane shows the contents of the Wildfire folder as a table.

Name	Date Modified	Size	Kind
AppleInternal	Today 4:51 AM	--	Folder
Diags	Today 4:56 AM	--	Folder
Logs	Today 4:56 AM	--	Folder
Smokey	Today 5:06 AM	--	Folder
Wildfire	Today 6:02 AM	--	Folder
.FactoryLogsWaitingToBeCollected	Today 4:49 AM	1 KB	Document
N78.lua	Today 4:49 AM	11 KB	Lua Source File
N78.plist	Today 4:49 AM	8 KB	Property List
PDCA.plist	Today 4:49 AM	1 MB	Property List
Smokey.log	Today 4:49 AM	10.5 MB	Log file

## Sequence Folder Contents

Within the sequence directory will be a number of files. For example, the diagram above illustrates the contents of a Wildfire sequence.

Sequence developers are expected to provide these files although not all files will be required under all scenarios.

The following two files are always required. Note that the files are to be named according to the platform on which the sequence is to run.

- **platform.lua** — This is the script file for the sequence.
- **platform.plist** — This is the property list file for the sequence.

The following files are Smokey control files that must be preallocated on the file system by the user.

- **PDCA.plist** — Stores sequence results.
- **Smokey.log** — Stores sequence log.

Lastly, this file is special and should be acquired from the EFI Diagnostics team.

- **.FactoryLogsWaitingToBeCollected** — Semaphore file for LogCollector. Its default content must be the word “SKIP” followed immediately by null bytes, padded out to 1KB.

## Creating Preallocated Sequence files

Users are given a choice in how much space to preallocate for sequence output. Baseline sizes are shown below. Smaller sizes are allowed as long as the files are large enough under all known scenarios.

- **PDCA.plist** — 1MB
- **Smokey.log** — 10MB

These files can be created from the Mac OS X command shell. For example, for a given *file* path requiring *size* megabytes:

```
dd if=/dev/zero of=file bs=1M count=size
```

In a pinch, the .FactoryLogsWaitingToBeCollected file can be created similarly:

```
echo SKIP > .FactoryLogsWaitingToBeCollected  
dd if=/dev/zero of=.FactoryLogsWaitingToBeCollected \  
count=1020 bs=1 seek=4
```

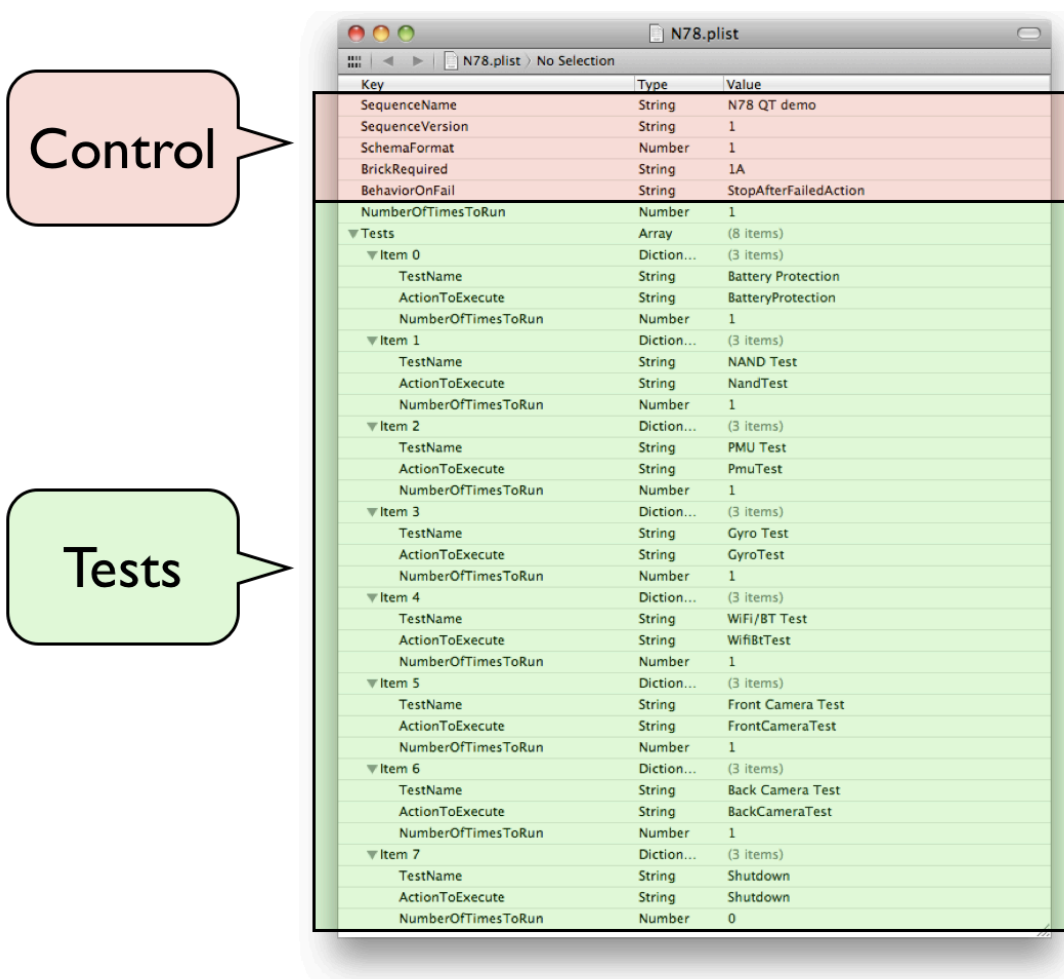
## Sequence Schema

### Schema

Smokey’s sequence property list is built on top of Apple’s ASCII plist format. A schema comprised of predefined keys and values is defined in order to describe both Smokey’s behavior as well as the sequence’s behavior.



Smokey's schema is rooted at the top of the plist hierarchy. Properties generally fall into one of two categories. **Control properties** are stored in the root. **Test item properties** start at the root, but use a recursive definition. The meaning of these properties and their legal values are described in the following sections.



Key	Type	Value
SequenceName	String	N78 QT demo
SequenceVersion	String	1
SchemaFormat	Number	1
BrickRequired	String	1A
BehaviorOnFail	String	StopAfterFailedAction
NumberOfTimesToRun	Number	1
Tests	Array	(8 items)
Item 0	Dictionary	(3 items)
TestName	String	Battery Protection
ActionToExecute	String	BatteryProtection
NumberOfTimesToRun	Number	1
Item 1	Dictionary	(3 items)
TestName	String	NAND Test
ActionToExecute	String	NandTest
NumberOfTimesToRun	Number	1
Item 2	Dictionary	(3 items)
TestName	String	PMU Test
ActionToExecute	String	PmuTest
NumberOfTimesToRun	Number	1
Item 3	Dictionary	(3 items)
TestName	String	Gyro Test
ActionToExecute	String	GyroTest
NumberOfTimesToRun	Number	1
Item 4	Dictionary	(3 items)
TestName	String	WiFi/BT Test
ActionToExecute	String	WifiBtTest
NumberOfTimesToRun	Number	1
Item 5	Dictionary	(3 items)
TestName	String	Front Camera Test
ActionToExecute	String	FrontCameraTest
NumberOfTimesToRun	Number	1
Item 6	Dictionary	(3 items)
TestName	String	Back Camera Test
ActionToExecute	String	BackCameraTest
NumberOfTimesToRun	Number	1
Item 7	Dictionary	(3 items)
TestName	String	Shutdown
ActionToExecute	String	Shutdown
NumberOfTimesToRun	Number	0

## Structure

The structure of test ordering in Smokey centers around the concepts of actions and test items. An **action** is a piece of Lua code that Smokey can invoke. A **test item** is a node in the plist structure with properties identifying how the sequencer treats an action. In effect, test items are consumers of actions, and the Smokey sequencer is a consumer of test items. Note that a particular action is allowed to appear in more than one test item.

The design of the schema is intended to produce nearly flat plist files for the common case where a sequence is a linear list of test items. In more complex scenarios, actions can be grouped and nested by layering test items. This is done by defining a test item property rather than an action property inside a test item.

## Control Properties

These properties define the basic parameters that Smokey uses when processing a sequence. These properties must be at the root of the plist. All values are required.

Control Properties	Type	Required?
<i>SequenceName</i>	String	Yes
<i>SequenceVersion</i>	String	Yes
<i>SchemaFormat</i>	Number	Must be “1”
<i>BrickRequired</i>	String	Yes
<i>BehaviorOnFail</i>	String	Yes

- **SequenceName** — Text string describing name the sequence.
- **SequenceVersion** — User-defined identifier to track changes and revisions.<sup>1</sup> For example, this could be a sequential number, or a date format like YYYYMMDD, where the letters are replaced with digits for the calendar year, month, and day.
- **SchemaFormat** — Currently must be set to 1.
- **BrickRequired** — Apple charger required to start or continue the sequence.
  - **None** — No charger required and no checks performed.
  - **Any** — Charger will be checked but any known external charger will be accepted.
  - **500mA** — Sufficiently powered USB hub.
  - **1A** — B1 or equivalent.
  - **2.1A** — B9 or equivalent.
  - **2.4A** — B45 or equivalent.
- **BehaviorOnFail** — Controls how Smokey proceeds after encountering a failure.
  - **KeepGoing** — Continue with the rest of the sequence.
  - **StopAfterFailedAction** — Abort the sequence after an action fails.

---

<sup>1</sup> Due to software limitations, SequenceVersion must be a number stored as a text string in the plist.

## Test Root Properties

These properties define the top of the test order and therefore required to be defined at the root of the plist. These are a subset of the test item properties.

Test Root Properties	Type	Required?	Contains
<i>FailScript</i>	String	Optional	
<i>NumberOfTimesToRun</i>	Number	Yes	
<i>Tests</i>	Array of Dict.	Yes	Test Item

- **FailScript** — User-defined Lua function to invoke when a failure is detected.
- **NumberOfTimesToRun** — The *Tests* property will be executed this many iterations.
- **Tests** — An ordered array of test items.

## Test Item Properties

The meat of the Smokey schema are the test items. They are the basic operating unit of the sequencer. Test items may be nested within each other.

Test Item Properties	Type	Required?	Contains
<i>TestName</i>	String	Optional	
<i>FailScript</i>	String	Optional	
<i>NumberOfTimesToRun</i>	Number	Yes	
<i>BehaviorOnAction</i>	String	Optional	
<i>ActionToExecute</i>	String	Alternate for Tests	
<i>Tests</i>	Array of Dict.	Alternate for ActionToExecute	Test Item

- **TestName** — Text string naming the test item. The effective test name defaults to the value of *ActionToExecute* if omitted.
- **FailScript** — User-defined Lua function to invoke when a failure is detected.
- **NumberOfTimesToRun** — The *Tests* or *ActionToExecute* property will be executed this many iterations.
- **BehaviorOnAction** — Additional steps required before invoking *ActionToExecute*.
  - **None** — Do nothing. This is the default if *BehaviorOnAction* is not specified.
  - **SaveState** — Save the state of the sequence to file. If the DUT is interrupted due to power loss, reboot, or otherwise, the Smokey will automatically continue once the DUT boots up.
- **ActionToExecute** — User-defined Lua function to invoke.
- **Tests** — An ordered array of test items.

## Sequence Flow

When Smokey executes a sequence, it handles a variety of tasks related to preparing the DUT, executing actions, logging results, and reporting progress. It will be important to know how these activities are arranged in order to understand how sequence actions fit in.

### Pre-Flight Phase

Before running a sequence, Smokey does the following:

1. Process command line options.
2. Check for autostart conditions.
  - a. If the autostart option is set, check the *boot-args* NVRAM variable.
    - i. If the *smokey* argument is set, merge in those options.
    - ii. If the *smokey* argument is not set, quit.
  - b. If the autostart option is not set, continue normally.
3. Load the sequence's Lua script and parse the property list file.
4. Apply remaining command line and NVRAM options.
5. Open log files.
6. Record the start of the sequence.
7. Initialize hardware and EFI software features implied by the sequence properties.
8. If *BrickRequired* is set, wait for an external charger to be connected. The wait time is indefinite.

### Sequence Execution Phase

9. Traverse the *Tests* array at the root of the plist in order and process each element. This is a pre-order, depth-first traversal.
  - Failure handling occurs during this phase. Fatal failures will force Smokey to jump to the next phase.
  - If the test item defines *ActionToExecute*, perform that action.
    - a. Flush pending file output
    - b. Invoke the named Lua function
    - c. Record the result of calling the function
  - If the test item defines *Tests*, recurse into that array.

### Post-Flight Phase

10. Write complete results.
11. Flush pending file output.
12. If autostarted, reboot.

## Periodic Tasks

While a sequence is running, Smokey only has control of the system between sequence actions. That time is used to handle various repeating management tasks.

- **Charger Check** — If *BrickRequired* is set and a change in the external charger state is detected, wait for the charger to be reconnected. The wait time is indefinite.

## Sequence Processing

### Test Item Naming

For reporting purposes, Smokey requires a name for each test item in the sequence that defines *ActionToExecute*. If the *TestName* property is set, it is used as the test name. Otherwise, the value of *ActionToExecute* is the default.

Note that the sequence properties must be configured to avoid duplicate test names. For instance, if several test items have the same *ActionToExecute*, each item must have a different *TestName*. Smokey will preemptively abort a sequence if it detects a naming conflict.

Test items without *ActionToExecute* produce no results or data, so they do not require a test name.

### Node Numbering

For tracking and internal purposes, Smokey represents the sequence as a tree and automatically gives each node a number. Like the sequence execution phase, assignment is done with a pre-order, depth-first traversal. This produces node numbers that read like line numbers when the sequence is printed in outline form.

The root of the plist file is always node one.

Node number assignment will change as the sequence changes. There is no support for specifying a node's number.

### Number of Test Item Iterations

The number of iterations per test item is the mathematical product of *NumberOfTimesToRun* at the test item in question and all test items immediately above it. One result is recorded each time the sequencer traverses into the test item.

For example, consider a sequence with *NumberOfTimesToRun* at the plist root set to 2 and a test item with *NumberOfTimesToRun* set to 3 and *ActionToExecute* defined. Smokey will process the test item 6 times in total. The sequencer will traverse into the test item twice. The first traversal will record results for iterations 1 through 3. The second traversal will record results for iterations 4 through 6.

## Pass/Fail Criteria of Actions

Smokey takes into account various Lua code return paths when assessing the result of an action. Scripts can take advantage of this to silently stop the sequence or return verbose failure information for later failure analysis.

### Function Return Value

Return Value	Assessment
<code>nil</code>	Pass
<code>true</code>	Pass
<code>false</code>	Fail
All Others	Unsupported

Note that, in the Lua language, `nil` can be returned explicitly with a `return` statement or implicitly by ending a code block without a `return`.

### Exceptions

Any exceptions not caught by actions themselves will be caught by the sequencer. Smokey considers this a failure and acts accordingly.

Note that the value of the data associated with an exception will be logged. If an action chooses to throw an exception, it is recommended that a helpful text string be sent.

## Failure Handling

### Sequence Excursions

Once an action's result has been assessed, Smokey decides whether to continue following the test order.

- **Pass** — Continue the sequence in the defined test order.
- **Fail** — Divert the test order based on the test item configuration.

The test item property *BehaviorOnFail* defines Smokey's exact failure handling.

- **KeepGoing** — Ignore the failure.
- **StopAfterFailedAction** — Stop the sequence immediately. Invoke the *FailScript* defined but the current test item and others as appropriate.

Note that *BehaviorOnFail* will affect sequence traversal, but does not play a part in assessing the result of an action.

## Failures During a Sequence

Sequences can choose to handle failures—whether for clean-up purposes or to manage internal state—by defining the *FailScript* property. Smokey will invoke the named Lua function when the sequencer assesses an action failure.

Because test items can be nested, more than one *FailScript* may be invoked. The first one to be executed would be at the test item that failed. The next would be the one in the test item immediately outside of that nested test item. And so on from the epicenter to the outermost *FailScript* at the plist root. Any test items along this path without a *FailScript* definition will be ignored.

The exact timing of *FailScript* invocations depends on *BehaviorOnFail*.

- **StopAfterFailedAction** — The first *FailScript* will be invoked immediately after failure assessment. Invocations at outer nesting levels follow immediately, ignoring the remaining iterations due to *NumberOfTimesToRun* at those test items.

## Failures During Periodic Tasks

Any failure during a periodic task is considered fatal. If *FailScript* is defined at the plist root, that Lua function shall be invoked. Thereafter, the execution phase is aborted.

# Test Results

## Results Tracking

During the course of a sequence, Smokey tracks results individually. There is a result recorded for each iteration of each test item. Additionally, there is an overall result based on the the individual results.

All results are initialized during the pre-flight phase to “incomplete”. Pass and fail are recorded as the sequence progresses. The overall result is recorded once the last test item is finished or the sequence is aborted due to failure.

## Test Results Naming

Names for results are automatically generated by combining the test item name with the iteration number.<sup>2</sup>

## Test Data Naming

Smokey can collect numeric data and limits from each test item. The data’s name is specified by the sequence script, but Smokey will make the name unique by including the name of the test item and the iteration number.<sup>3</sup>

---

<sup>2</sup> See the “Individual Test Results” table in “Test Results and Data”.

<sup>3</sup> See the “Test Data” table in “Test Results and Data”.

## Propagation of Results

To maintain consistency, the overall sequence result is computed from the individual test item results. Therefore, any failure at the item level forces a the overall sequence result to fail. Incomplete tests are treated as failures.

## Sequence State Saving

### Sequence Continuation

A unique feature of Smokey compared to canonical Lua scripting is the built-in ability to pause a sequence and continue at a later time. This means that any action can reboot the DUT, power cycle, or otherwise interrupt the system and Smokey will know how to pick up from the last test item without losing results.

As mentioned in the test item properties description, this behavior is enabled by setting *BehaviorOnAction* to *SaveState*. It is effective only during the test item in which the property is defined. If the test item's action does not interrupt the DUT, Smokey will automatically clear the continuation point.

### Requirements

The DUT must meet the requirements for autostart. See the autostart section for more details.

### Serialization

When enabled, Smokey performs the following immediately before invoking *ActionToExecute*.

- **Set Continuation Point** — Mark the sequence to continue at the test item immediately following the one defining *BehaviorOnAction*.
- **Save Smokey State** — Write all internal Smokey data to a temporary file.
  - Save the state of the sequence traversal.
  - Save all results the same file.
- **Configure Autostart** — Set Smokey to run automatically at next boot.
  - Save the current *boot-args* and *boot-command* NVRAM variables.<sup>4</sup>
  - Add a bare “smokey” argument to *boot-args*.
  - Set *boot-command* to “diags”.

### Resurrection

When Smokey is invoked from the command line, it checks for the existence of a saved state before loading the user-specified sequence. If one is found, the state is automatically resurrected and that state's sequence will be continued.

---

<sup>4</sup> See the autostart description for how NVRAM variables are used by Smokey.



- **Load State Data** — Resurrect the data saved during serialization.
  - Reload all internal Smokey variables.
  - Reload all results.
- **Reset Autostart** — Disable the configuration from serialization.
  - Restore *boot-args* and *boot-command*.
- **Delete Continuation Point** — Clear the state file so that it can be reused.

# Design for Factory Use

## Objectives for Factory Use

Smokey's objective is to automate testing for manufacturing scenarios. Primarily, this means enabling a DUT to run commands on its own and emit data to factory processes. Several design decisions were made and features implemented to support these goals:

- Support for multiple, user-defined sequences allows use both on and off the manufacturing line.
- Sequences are maintained separately from EFI diags and therefore don't require intensive validation (e.g. on all stations) before roll-out.
- Sequences are stored on the filesystem to enable identical deployment on a large number of test units.
- Sequence execution is logged to a file for failure analysis.
- Results are emitted in DCSD's format for PDCA.
- Execution can be triggered via NVRAM.
- Timestamps identify hangs, performance issues, and process excursions.

## Station Behavior

In traditional station software, a station ID is used for reporting results and log collection. The name of the station is collaboratively chosen by the station DRI, program managers, and the factory software team. A numeric identifier serves to codify the station's place within the factory process.

Smokey isn't meant to directly replace station software because it wasn't designed to plug into the manufacturing infrastructure. However, it can be used to control those aspects of a station that affect the state of the DUT.

Station behavior is enabled by giving sequences specific, reserved names. Presently, the sequence names below are reserved for specific factory station activities. Consult an EPM or TDL for the most up-to-date list for a specific project.

- Wildfire (ID 0xB4)

The list of station-related behavior in Smokey follows:

- Control bits

Sequences that do not have any of the reserved names do not receive special treatment. Use of reserved sequence names for non-station-related activities is highly discouraged.

## Control Bits

Control bits are a process control mechanism used to track test coverage in the factory. Because coverage is directly related to final product quality, control bits can only be manipulated with the proper clearance. Smokey strives to ensure the security of control bits by acting as a middleman between sequences and the DUT.

For sequences associated with a station ID, the respective control bit is updated as the sequence executes in order to record gross progress. This generally happens alongside the writing of results to the filesystem, but may occur less frequently because the granularity of control bit states is coarse.

The control bit state can be used as a rough indicator of progress.

- **Untested** — Sequence has not been invoked or Smokey aborted early.
- **Incomplete** — Smokey processed the sequence and has recorded its start.
- **Pass or Fail** — Sequence is complete.

Writing to control bits can not be disabled.

## Log Collection

### DUT Identifier

Smokey identifies the device under test by the system serial number *SrNm* in `syscfg`. If that key is not defined, the MLB serial number *MLB#* is used instead. It is considered a fatal error when neither key is defined.

### File Output

Smokey's file output is designed to be compatible with LogCollector, with emphasis on being human readable. Towards that goal, files are written in ASCII format and Smokey's output is sent to several files, as documented elsewhere. All files in a sequence's directory will be collected and stored in PDCA.

Two files will be interesting to those investigating issues with log collection:

- **.FactoryLogsWaitingToBeCollected** — Will be filled with zeroes if PDCA.plist has valid data. Otherwise should not be empty.
- **PDCA.plist** — The main output file for LogCollector. Smokey writes results to this file in a PDCA-specific schema. This file is zero-filled on a fresh unit. At the start of a sequence, Smokey writes this file with a default result to indicate a crash, hang, or power loss; it is later overwritten with the actual results of the sequence.

## Test Results and Data

Information reported to the PDCA system includes the results for all tests as well as the data gathered by those tests. These are all stored in plist format using DCSD's PDCA schema.

Because of the schema, all output is represented as test results. The PDCA properties *testname*, *subtestname*, and *subsubtestname*<sup>5</sup> are used as identifiers for individual tests. The *result* and *failure\_message* properties, as well as others, encode sequence output. Specific values are discussed below.

Overall Sequence Result	Pass	Fail	Incomplete
<i>overallResult</i>	"PASS"	"FAIL"	

Individual Test Results	Pass	Fail	Incomplete
<i>testname</i>	Test item's <i>TestName</i> or <i>ActionToExecute</i> property		
<i>subtestname</i>	"Iteration " + iteration number		
<i>subsubtestname</i>	Not used		
<i>result</i>	"PASS"	"FAIL"	
<i>failure_message</i>	Not used	Actual message	"Incomplete"

Test Data	No Limits	Within Limits	Exceed Limits
<i>testname</i>	Test item's <i>TestName</i> or <i>ActionToExecute</i> property		
<i>subtestname</i>	"Iteration " + iteration number		
<i>subsubtestname</i>	Sequence's data name		
<i>value</i>	Sequence's data value		
<i>units</i>	Sequence's data units (if provided)		
<i>lowerlimit</i>	Not used	Data lower limit	
<i>upperlimit</i>	Not used	Data upper limit	
<i>result</i>	"PASS"		"FAIL"
<i>failure_message</i>	Not used		Automatic

---

<sup>5</sup> The PDCA database reports *testname*, *subtestname*, and *subsubtestname* as one concatenated string.

# Using Smokey

## Command Line Arguments

Smokey is command line driven and only supports certain combinations of arguments.

In the following descriptions, options are prefixed by double dashes. Options shown in square brackets may be omitted. Ellipses are used to indicate that additional options may be specified.

Values to be supplied by the user are defined below:

- **Sequence** — The name of a sequence. Smokey searches for this folder name in the standard location. Must contain files appropriate for the DUT platform.

### Print Command Line Help

```
smokey --help
```

Print a list of all supported command line options. Some options may not be intended for general use. Not all option combinations are valid.

### Run a Sequence

```
smokey Sequence [--clearstate] --run
```

Execute a sequence from a fresh state.

If there is an existing saved state, specifying `--clearstate` ensures that the saved state is not used.

### Continue from a Saved State

```
smokey [--retainstate] --run
```

If a saved state is available, continue from where it left off. Most other command line options are ignored.

Typically, the saved state is deleted when continuing a sequence so that progress moves forward. If the state should be retained to later re-run from the same point, `--retainstate` can be specified. Note that continuing twice from the same saved state may not produce the same file output as normal. Also note that this option won't prevent a new state from overwriting the current state.

### Sanity Check a Sequence

```
smokey Sequence
```

Smokey will load the sequence script and plist files to check syntax and settings. No actions beyond that are taken.

## Perform a Dry Run on a Sequence

```
smokey Sequence [--clearstate] --dryrun
```

A dry run is the same as a normal run, but actions are not invoked. This is useful to make sure basic issues in Smokey are not causing sequence failures.

## Get Information about a Sequence

```
smokey Sequence --sequence  
smokey Sequence --summary
```

The first command dumps the entire sequence definition whereas the second summarizes the actions that will be taken. They can be used to verify the structure of the sequence plist file. See the “Sequence Output” section for more details.

## Clear Existing Saved State

```
smokey --clearstate
```

Erase the sequence state saved by *BehaviorOnAction*. No actions are taken, and no errors reported, if there is no state presently defined.

## Autostart (Trigger Smokey Externally)

```
smokey ... --autostart
```

Decide whether or not to actually run Smokey based on external factors. Quit without error when those factors are not satisfied. Follow autostart behavior when all factors are present.

# Autostart

On Darwin mobile devices, Smokey can be configured to execute a sequence upon booting into EFI diags. This allows automatic test execution on the device, as well as integration with other test environments. Smokey supports this feature primarily to enable EFI-based testing from inside iOS.

## Prerequisites for Autostart

Autostart is effected by a combination of device settings and EFI diags configuration. The following requirements must be met.

- **NVRAM** — The DUT must support iOS-style NVRAM variables in EFI diags. This means that firmware driver must be available and the DUT must support this hardware feature.
- **EFI Diags Boot Configuration** — On the EFI diags side, Smokey needs to be part of the boot process. This boils down to making the Smokey command part of the built-in start-up script.

## Configuring EFI Diags for Autostart

It is recommended that the command line be executed as early as possible in the boot process.

```
smokey --run --autostart
```

## Configuring DUT for Autostart

Once prerequisites are met, the following NVRAM variables can be set to kick off a Smokey sequence.

- **boot-args** — The *smokey* argument must be set. Other arguments in this variable are ignored by Smokey.
- **boot-command** — Must be set to “diags”. If this is not possible, a special cable or dongle will be required to force the DUT to boot into EFI diags.
- **auto-boot** — Must be set to “true”.

## Smokey Argument

The *smokey* argument in *boot-args* is a mechanism to augment the Smokey command line. Its value is a comma-separated list that is applied after command line arguments have been processed.

The decision was made to put *smokey* in *boot-args*—rather than as its own NVRAM variable—in order to support deployment: The *boot-args* variable can be easily set from PurpleRestore, whereas other variables are not well supported. Units can have a sequence rooted and be configured to run one on the next reboot.

Syntax and legal values are defined below.

```
boot-args = "... smokey ..."  
boot-args = "... smokey=Sequence ..."
```

There is currently only one NVRAM setting supported.

- **Sequence** — The name of the sequence to execute. If omitted, the sequence named on the command line is used instead.

## Autostart Behavior

As mentioned in the sequence flow description, the conditions for automatically starting is the autostart option on the command line and the existence of *smokey* in *boot-args*. If either are missing, Smokey will quit without changing the state of the DUT.

When all autostart conditions are met, Smokey will proceed to run as if the settings in the *smokey* argument had been typed at the command line. Additionally, the following tasks are performed:

- Remove the *smokey* argument from *boot-arg*. This signals external processes that Smokey has recognized and accepted the autostart conditions.

- Configure the DUT to run iOS on the next reboot. This is done by setting the *boot-command* NVRAM variable to “fsboot”.

Automatic start implies automatic reboot. Smokey will reboot the DUT at the end of the sequence regardless of what transpires during its execution.

## Sequence Output

### Run-time Output

Smokey emits diagnostic and informative data as it runs. The order is fixed and the data logged to console and file are basically the same. However, some ways of invoking Smokey may omit some sections.

1. **Software Build Information** — Source code version and binary build date.
2. **Device Identification** — MLB and SoC identification numbers.
3. **Sequence Files** — Files nominally used the current sequence. Some files may not be used, pursuant to sequent properties.
4. **Sequence Properties** — Summary of the sequence properties in effect.
5. **Pre-flight Output** — Running status as the DUT is prepared for the sequence.
6. **Test Item Trace** — A trace as Smokey traverses the sequence. Each line is prefixed with a context information. EFI diags commands and direct script output are interspersed in real time.
  - **Timestamp** — Shown in square brackets to separate from other text on the line.
  - **Node Number** — Smokey-assigned node number for the test item. Replaced with an ellipsis when it is the same as the line above.
7. **Post-Flight Output** — The sequence wind-down. This includes the overall sequence result as well as Smokey’s attempt to write all results to file.
8. **Error Summary** — The “All Errors” heading is used to reiterate all failures and errors captured during a sequence. This section is omitted when the sequence passes.

### Log File

All output that Smokey emits to the console is also captured to the sequence’s log file. Additionally, a time stamp is appended to the log each time it is opened. The following example shows the log file of a failed sequence, including all of the output elements described previously.

**Log Info** Opened log at 2012-10-09 12:20:57

**1. SW Build** Smokey 1A0529 (changelist 294521)  
Built 2012/10/09 02:16:45

**2. Device ID** SrNm: CCQH06QF4K0  
MLB#: C02219500L9F4MQ1  
CFG#: DA9/EVT2/00L9//2030/FT1-G2  
ECID: 000001D7DC612DC9



```

3. Seq. Files Control File: nandfs:\AppleInternal\Diags\Logs\Smokey\Wildfire\N78.plist
Script File:  nandfs:\AppleInternal\Diags\Logs\Smokey\Wildfire\N78.lua
Log File:     nandfs:\AppleInternal\Diags\Logs\Smokey\Wildfire\Smokey.log
Results File: nandfs:\AppleInternal\Diags\Logs\Smokey\Wildfire\PDCA.plist
Control Bit:  Wildfire (0xB4)

```

```

Log Info Finished dumping pre-log buffer

```

```

4. Seq. Props SequenceName:    N78 QT demo
SequenceVersion: 1
BehaviorOnFail:  StopAfterFailedAction
ResultsBehavior: Bookend
LogBehavior:     Full
BrickRequired:   1A

```

```

5. Pre-flight Sequence syntax and sanity check passed

```

```

Writing default results
Writing control bit
Writing PDCA plist file

```

```

Initializing display
Initializing charger
Device ready

```

```

6. Test Trace Sequence execution...

```

```

Day/Time      Node
-----
[09 12:21:04] N001 Repeating 1x
[09 12:21:04] ....           [1] Periodic tasks
[09 12:21:04] ....           Detected 1A brick
[09 12:21:04] N002           [1] Repeating 0x "Battery Protection"
[09 12:21:04] N003           [1] Repeating 1x "PMU Test"
[09 12:21:04] ....           [1] PmuTest
[09 12:21:04] ....           pmureg -r 0 0x00

```

```

Diags Command :-) pmureg -r 0 0x00
Register 0x0000 : 0x56

```

```

[09 12:21:04] ....           Exit code = 0x00000000
[09 12:21:04] ....           pmureg -r 0 0xA4

```

```

Diags Command :-) pmureg -r 0 0xA4
Register 0x00A4 : 0xB5

```

```

[09 12:21:04] ....           Exit code = 0x00000000
[09 12:21:04] ....           pmustat chipid

```

```

Diags Command :-) pmustat chipid
PMU Status test
ChipID: 0x56

```

```

[09 12:21:05] ....           Exit code = 0x00000000
[09 12:21:05] N004           [1] Repeating 2x "Gyro Test"
[09 12:21:05] ....           [1] GyroTest
[09 12:21:05] ....           gyro --init

```

```

Diags Command :-) gyro --init
Powering on Gyro: OK
Resetting Gyro: OK
Gyro ChipID: ST Micro AP3GDL v3.0.0 part:3
Raw ID: 0xD5
Serial: 0x00000000
Warning: lot/batch numbers are not valid data at this time.
Lot: 0 Batch: 0

```

```

OK
-----
[09 12:21:05] .... Exit code = 0x00000000
[09 12:21:05] .... gyro --selftest
-----

Diags Command :-) gyro --selftest
Starting Gyro self-test:
Self Test data:
(3)(-360)DATA: X-diff=363
(1)(362)DATA: Y-diff=361
(0)(-358)DATA: Z-diff=358
OK
-----
[09 12:21:06] .... Exit code = 0x00000000
[09 12:21:06] .... gyro --off
-----

Diags Command :-) gyro --off
Powering down axes: OK
Powering down Gyro: OK
-----
[09 12:21:07] .... Exit code = 0x00000000
[09 12:21:07] .... [2] GyroTest
[09 12:21:07] .... gyro --init
-----

Diags Command :-) gyro --init
Powering on Gyro: OK
Resetting Gyro: OK
Gyro ChipID: ST Micro AP3GDL v3.0.0 part:3
Raw ID: 0xD5
Serial: 0x000000000
Warning: lot/batch numbers are not valid data at this time.
Lot: 0 Batch: 0
OK
-----
[09 12:21:07] .... Exit code = 0x00000000
[09 12:21:07] .... gyro --selftest
-----

Diags Command :-) gyro --selftest
Starting Gyro self-test:
Self Test data:
(4)(-360)DATA: X-diff=364
(1)(362)DATA: Y-diff=361
(0)(-358)DATA: Z-diff=358
OK
-----
[09 12:21:09] .... Exit code = 0x00000000
[09 12:21:09] .... gyro --off
-----

Diags Command :-) gyro --off
Powering down axes: OK
Powering down Gyro: OK
-----
[09 12:21:09] .... Exit code = 0x00000000
[09 12:21:09] N005 [1] Repeating 1x "WiFi/BT Test"
[09 12:21:09] .... [1] WifiBtTest
[09 12:21:09] .... device -k WiFi -e power_on
-----

Diags Command :-) device -k WiFi -e power_on
ERROR: Method "power_on" returned status Not Found
device returned Not Found error
-----
[09 12:21:09] .... Exit code = 0x8000000E
[09 12:21:09] .... ActionToExecute failed
[09 12:21:09] .... [1] WifiBtHandler
[09 12:21:09] N001 Sequence done

```

## 7. Post-flight Failed

```
Writing final results
Writing control bit
Writing PDCA plist file
```

## 8. Error Summary All errors:

```
SmokeyResults: failed action WifiBtTest at node 5 iteration 1/1: ↵
EfiCommand: command had errors: device -k WiFi -e power_on
SmokeyCore: stopping after failed action
```

## Sequence Dump

```
:-) smokey Wildfire --sequence

Test Sequence

Day/Time      Node
-----
[09 11:13:17] N001 Repeat 1x
[09 11:13:17] N002      Repeat 0x "Battery Protection"
[09 11:13:17] ....      BatteryProtection
[09 11:13:17] N003      Repeat 1x "PMU Test"
[09 11:13:17] ....      PmuTest
[09 11:13:17] N004      Repeat 2x "Gyro Test"
[09 11:13:17] ....      GyroTest
[09 11:13:17] N005      Repeat 1x "WiFi/BT Test" --> WifiBtHandler
[09 11:13:17] ....      WifiBtTest
```

As a diagnostic and informative feature, Smokey can print the test order of a sequence without doing anything else. The `--sequence` command line option will print the test order very similar to the way that Smokey executes it with `--run`.

- **Exhaustive Listing** — All test items are included, even those with *NumberOfTimesToRun* set to zero.
- **Explicit Properties** — The properties *TestName* and *NumberOfTimesToRun* are shown. *FailScript* is on same line as *NumberOfTimesToRun*, prefixed with an arrow.
- **Actions** — *ActionToExecute* is shown on its own line.

## Sequence Summary

```
:-) smokey Wildfire --summary

Test Sequence Summary

Day/Time      Node
-----
[09 11:13:17] N001 Repeat 1x
[09 11:13:17] N003      PmuTest
[09 11:13:17] N004      Repeat 2x
[09 11:13:17] ....      GyroTest
[09 11:13:17] N005      WifiBtTest --> WifiBtHandler
```

The `--summary` command line option is similar to the `--sequence` option, but potentially a lot more concise.

- **Concise Listing** — No test items with zero iterations. Children of those test items are also omitted.
- **Abbreviated Properties** — Test items with only one iteration do not have a separate output line indicating showing *NumberOfTimesToRun*. *TestName* is omitted on all test items. *FailScript* is on the same line as *ActionToExecute* if the line with *NumberOfTimesToRun* is omitted.

## Screen Output

If the DUT has a display module available, Smokey provides visual feedback on the sequence's execution using modal screens. Each has a distinct color or design to make it easy to identify state at a glance.



- **Connect to Brick** — Sequence is running, but halted while DUT is waiting for an external charger. Background is bright yellow and instructions in black text.
- **Running Status** — DUT is running an action. A logo is shown in the background. The current action name is shown in white text over a black background.
- **Pass** — Sequence is complete and successful. Background is bright green.
- **Fail** — Sequence is complete but failed. Background is bright red.

## State Control

Smokey will generally manage its state without need for outside intervention. However, commands are available to clear the state and get the DUT back to normal. These are useful, for example, for triaging a problem or interrupting a sequence while the DUT is rebooting.

To understand why these commands work, please be sure to read the section on sequence state saving before proceeding.

## Clearing State

In order to run a new sequence, or restart a previous sequence, the previously saved state must be cleared. See the description of command line options for the exact syntax to clear the current Smokey state.

Smokey only supports one state to be shared amongst all sequences. Effectively, clearing one saved state will clear them for all sequences.

## Clearing Autostart

When Smokey is part of the EFI diags boot script, the most important part of breaking into a DUT in the middle of saving sequence state and rebooting is to disable the autostart configuration. This will usually allow EFI diags to boot into the command line.

The work boils down to interrupting the DUT before it loads EFI diags in order to manipulate NVRAM variables.

Breaking into the DUT before it loads EFI diags involves controlling it while it is in either iBoot or iOS. The details of doing this won't be covered here.

Once control is gained over the DUT, the following NVRAM variables can be changed:

- **boot-args** — The “smokey” argument must be removed.
- **boot-command** — If the goal is not to boot into EFI diags, then this needs to be set accordingly. Otherwise, the value can be left untouched.
- **auto-boot** — Like *boot-command*, this can usually be left alone, but can be changed depending on the situation. Note that Smokey doesn't modify this variable.

Additional work—beyond the scope of this document—may be required if other software in EFI diags outside of Smokey configured for autostart at the same time.

# Developing Smokey Sequences

## Smokey Lua API

Smokey provides Lua scripting interfaces to interact with the DUT, EFI diags, and with Smokey itself. On top of the standard Lua facilities, these will be the the crux of sequence actions. The following sections will describe the Smokey API.

Some function arguments will be optional and will be noted with an asterisk. Following the Lua convention, optional arguments at the end of a function's argument list may simply be omitted during a call. Optional arguments in the middle must be passed as `nil`.

### Shell Function

```
Shell(CommandLine)
```

Execute the string *CommandLine* as if it was typed at the EFI command shell. Output is captured into the global table *Last*.

Argument	Type	Req?	Comment
<i>CommandLine</i>	String	Yes	EFI diags command line

*Shell* will inspect the command output for failures. If any are detected, it will raise an exception with a string describing the first fault detected. If the sequence action doesn't catch this exception, Smokey will catch it by default.

Failures are defined below.

- **Non-zero exit code** — The convention for EFI commands is to return zero for success. All other values indicate some kind of failure.
- **Error Message Detected** — The text "ERROR:" is the label for error messages. If this string is found, the command is considered to have failed regardless of exit code.

### Last Table

This global table is updated each time *Shell* is called. Scripts can use this table to inspect the most recent command's result or parse its output.

Last Field	Type	Value
<i>CommandLine</i>	String	The string passed into <i>Shell</i> .
<i>ExitCode</i>	Number	Command return code. Zero means success.
<i>RawOutput</i>	String	All console output from the command.
<i>Output</i>	String	Same as <i>RawOutput</i> but stripped of error messages.

## ReportData Function

```
ReportData(Name, Value, Units*, LowerLimit*, UpperLimit*)
```

Record a named key-value pair for the current iteration of the current test item. Smokey will create a unique identifier for the data when generating PDCA results.

Argument	Type	Req?	Comment
<i>Name</i>	String	Yes	Local identifier for this datum.
<i>Value</i>	Number	Yes	Value for this datum.
<i>Units</i>	String	No	Dimension and magnitude (e.g. “mV” or “s”).
<i>LowerLimit</i>	Number	No	Must be in the same units as <i>Value</i> .
<i>UpperLimit</i>	Number	No	

*LowerLimit* and *UpperLimit* for *Value* will be independently defined in the PDCA results if one or the other is provided in the during the function call.

*Units* can be defined without defining limits.

## ReportAttribute Function

```
ReportAttribute(Name, Value)
```

Record a key-value pair for the DUT. PDCA treats this data as specific to the unit rather than specific to any specific factory test.

Argument	Type	Req?	Comment
<i>Name</i>	String	Yes	Global identifier for this attribute.
<i>Value</i>	String	Yes	Value for this attribute.

There is a single namespace for attributes shared across the entire sequence.

## Data Submission to PDCA

Smokey defines *ReportData* as the main interface for generating parametric data for the PDCA system. Each datum is uniquely identified for completeness. See the section on design for factory use for more details on the exact conversion process.

## Sequence Development Quick Start

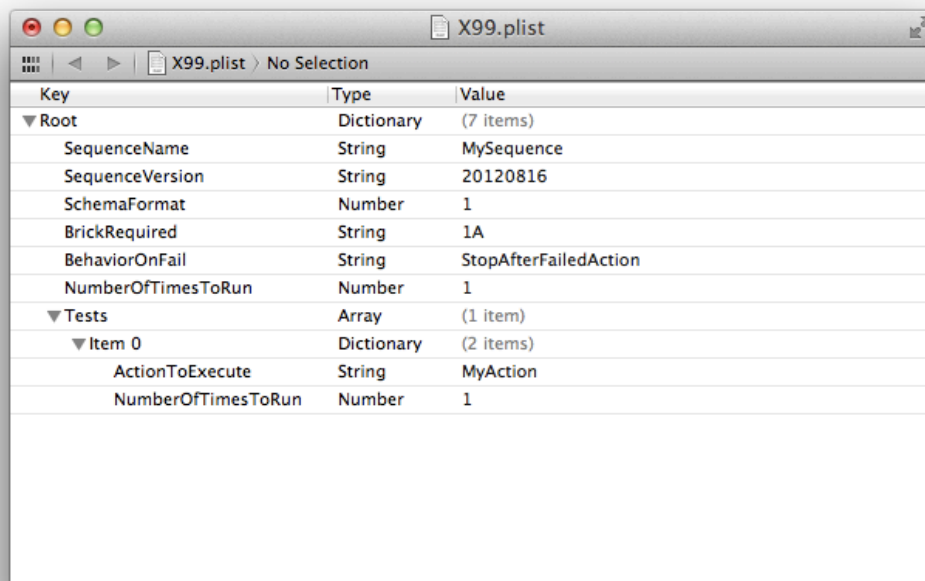
The quickest way to start developing a Smokey sequence is to duplicate an existing one. A member of the EFI diags team can provide these files.

However, it's possible to start from scratch by taking note of the file organization described in the Smokey fundamentals section. For example, the following commands will create a sequence named "MySequence" for the X99 platform. These commands need to be run at the command line of a host computer while in the Smokey folder of a file tree root.

```
mkdir MySequence
touch MySequence/X99.lua
touch MySequence/X99.plist
dd if=/dev/zero of=MySequence/PDCA.plist bs=1M count=1
dd if=/dev/zero of=MySequence/Smokey.log bs=1M count=10
echo SKIP > MySequence/.FactoryLogsWaitingToBeCollected
dd if=/dev/zero \
    of=MySequence/.FactoryLogsWaitingToBeCollected \
    count=1020 bs=1 seek=4
```

The Lua file will be created, but empty. A text editor can be used to fill it with the appropriate content.

The plist file will likewise be empty. A text editor can also be used to fill it, but a purpose-built application like Xcode is highly recommended. At a minimum, the required properties and one test item must be defined.



The screenshot shows a window titled "X99.plist" with a table of properties and values. The table has three columns: Key, Type, and Value. The data is as follows:

Key	Type	Value
▼ Root	Dictionary	(7 items)
SequenceName	String	MySequence
SequenceVersion	String	20120816
SchemaFormat	Number	1
BrickRequired	String	1A
BehaviorOnFail	String	StopAfterFailedAction
NumberOfTimesToRun	Number	1
▼ Tests	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
ActionToExecute	String	MyAction
NumberOfTimesToRun	Number	1

Once the files are created, they should be rooted onto the DUT. <sup>6</sup>

<sup>6</sup> Consult iOS documentation for details on file roots, how they work, and how to apply them to a DUT.



## Developing Code with the Smokey Shell

Certain features of Smokey are available as an interactive shell running on the DUT. This means that it is possible to test code snippets and EFI commands without loading files onto the DUT.

Currently, this feature is limited to engineering hardware only. Also, it must be enabled in the particular build of EFI diags.

### Starting the Shell

```
smokeyshell
```

There are no command line options supported.

### Executing Commands

The > (greater-than) prompt will be shown once the shell is running and ready to accept input. At this point, it will be possible to type or paste code into the terminal window.

The shell supports all of the standard Lua functions and libraries, as well as the standard interactive Lua shell conventions. Additionally, the following Smokey API functions may be used:

- **Shell** — Behaves the same as in Smokey.

### Exiting the Shell

Pressing `ctrl` `D` will end input and quit Smokey Shell.

## Developing Code on DUT

*This feature is not yet available.*

## Developing Code with the Smokey Simulator

*This feature is not yet available.*

# Smokey Internals

## Software Architecture

*Documentation not yet available.*

## Control Files

*Documentation not yet available.*

## Saving State

*Documentation not yet available.*

## Schema Grammar

*Documentation not yet available.*

# References

## Useful Lua Links

### **Official Lua Home Page**

<http://www.lua.org/home.html>

### **Lua Executable Binaries**

<http://luabinaries.sourceforge.net/download.html>

### **Official Lua 5.2 Reference Manual**

<http://www.lua.org/manual/5.2/>

### **Lua Programmer's Guide**

<http://www.lua.org/pil/>

### **Lua Pitfalls and Gotchas**

<http://www.luafaq.org/gotchas.html>

### **Lua Community Wiki**

<http://lua-users.org/wiki/>