



Smokey **Command Sequencer** **User and Technical Manual**

"Spirit" Edition

May 19, 2014

Chanh-Duy Tran
System Software Engineering, Firmware
ctran@apple.com

Contents

1	Introduction	1
1.1	What is Smokey?	1
1.2	Audience	1
1.3	Document Objective	1
2	Lua Fundamentals	2
2.1	Lua	2
2.2	Data	2
2.2.1	Built-in Types	2
2.2.2	Dynamic Typing	2
2.2.3	Scope	3
2.2.4	Binding	3
2.2.5	Mutation	4
2.2.6	Nil Values	4
2.2.7	Multiple Values	4
2.3	Comments	4
2.3.1	Single-line Comments	4
2.3.2	Delimited Comments	5
2.4	Boolean Expressions	5
2.4.1	Relational Operators	5
2.4.2	Logical Operators	5
2.4.3	Conditionals	5
2.4.4	Checking Existence	6
2.5	Tables	6
2.5.1	Storage	6
2.5.2	Access	6
2.5.3	Arrays	7
2.5.4	Classes	7
2.6	Language Constructs	7
2.6.1	Conditionals: if-then-else	8
2.6.2	Loops: for-do	8
2.6.3	Loops: while-do	9
2.6.4	Loops: repeat-until	9
2.6.5	Functions	9
2.7	Built-in Facilities	10
2.7.1	String Concatenation	10
2.7.2	<i>string.format</i> Function	10
2.7.3	Bitwise Operations	10
2.7.4	<i>tonumber</i> Function	11
2.7.5	<i>tostring</i> Function	11
2.7.6	<i>type</i> Function	11
2.8	Exceptions	11
2.8.1	Raising Exceptions	11

2.8.2	Catching Exceptions	12
2.9	Tricks	13
2.9.1	Data Alternation and Short Circuit Evaluation with Boolean Expressions	13
2.9.2	Ternary-Like Operator	13
3	Smokey Fundamentals	14
3.1	Sequences: Scripts and Property Lists	14
3.2	File Organization	14
3.2.1	Home Folder	14
3.2.2	Sequence Name and Location	14
3.2.3	Sequence Folder Layout	14
	Sequence Contents	14
	Sequence Platform Specialization	15
3.2.4	Deprecated Sequence Folder Layout	16
3.3	Sequence Schema	16
3.3.1	Schema	16
3.3.2	Structure	16
3.4	Sequence Flow	17
3.4.1	Pre-Flight Phase	18
3.4.2	Sequence Execution Phase	18
3.4.3	Post-Flight Phase	18
3.4.4	Periodic Tasks	19
3.5	Sequence Processing	19
3.5.1	Test Item Naming	19
3.5.2	Node Numbering	19
3.5.3	Number of Test Item Iterations	19
3.6	Pass/Fail Criteria of Actions	20
3.6.1	Function Return Value	20
3.6.2	Exceptions	20
3.6.3	Data Results Cascading	20
3.7	Failure Handling	20
3.7.1	Sequence Excursions	20
3.7.2	Handling Failures During a Sequence	21
3.7.3	Failures During Periodic Tasks	22
3.8	Test Results	22
3.8.1	Results Tracking	22
3.8.2	Test Results Naming	22
3.8.3	Test Data Naming	22
3.8.4	Propagation of Results	22
3.9	Sequence State Saving	22
3.9.1	Sequence Continuation	22
3.9.2	Requirements	23
3.9.3	Serialization	23
3.9.4	Resurrection	23
4	Design for Factory Use	24
4.1	Objectives for Factory Use	24

4.2	Station Behavior	24
4.3	Control Bits	25
4.4	Parametric Data	25
4.5	Log Collection	25
4.5.1	DUT Identifier	25
4.5.2	File Output for LogCollector	26
4.5.3	File Output for Earthbound	26
4.6	Encoding Test Results and Data	26
5	Using Smokey	28
5.1	Smokey Command Line Arguments	28
5.1.1	Print Version	28
5.1.2	Print Command Line Help	28
5.1.3	Run a Sequence	28
5.1.4	Continue from a Saved State	28
5.1.5	Sanity Check a Sequence	29
5.1.6	Perform a Dry Run on a Sequence	29
5.1.7	Get Information about a Sequence	29
5.1.8	Get Information about Sequence Code	29
5.1.9	Clear Existing Saved State	29
5.1.10	Clear Existing Sequence Results	29
5.1.11	Trigger Smokey Externally (Autostart)	30
5.1.12	Manually Select Tests to Run	30
5.1.13	Override Property Value	30
5.1.14	Overriding Test Arguments	30
5.2	Autostart	31
5.2.1	Prerequisites for Autostart	31
5.2.2	Configuring EFI Diags for Autostart	31
5.2.3	Configuring DUT for Autostart	31
5.2.4	Smokey NVRAM Argument	32
5.2.5	Autostart Behavior	32
5.3	File Output	33
5.3.1	Sequence Output Contents	33
5.3.2	Sequence Output Control	33
5.3.3	Creating Preallocated Sequence Files	34
5.3.4	Smokey Control Files	34
5.4	Sequence Output	34
5.4.1	Run-time Output	34
5.4.2	Log File	35
5.4.3	Sequence Dump	37
5.4.4	Sequence Summary	38
5.4.5	Sequence Dependencies	38
5.5	Screen Output	39
5.6	State Control	39
5.6.1	Clearing State	40
5.6.2	Clearing Autostart	40

6	Using Smokey Simulator	41
6.1	Differences from Smokey	41
6.2	Emulating the DUT Work Environment	42
6.3	Link Mode	43
6.3.1	System Requirements	43
6.3.2	Setting Up the Host and DUT	44
6.3.3	Disabling Auto-Boot in iBoot	44
6.3.4	Choosing a Link Device	45
6.3.5	Using Link Mode	45
6.3.6	Quitting a Link	46
6.3.7	Continuing a Link	46
6.3.8	Loading an EFI Diags Image	46
6.4	Smokey Simulator Command Line Arguments	47
6.4.1	Specifying a Sequence	47
6.4.2	Specifying a Shared Directory	47
6.4.3	Booting an EFI Diags Image	47
6.4.4	Override Platform Identity	47
6.4.5	Enabling DUT Link	48
6.4.6	Logging DUT Link	48
6.5	Distribution	48
6.5.1	Smokey Simulator SDK Binary	48
6.5.2	Release Archive	49
	Archive Contents	49
	Using the Simulator Makefile	49
7	Developing Smokey Sequences	51
7.1	Smokey Properties	51
7.1.1	Mandatory Control Properties	51
7.1.2	Optional Control Properties	53
7.1.3	Test Root Properties	54
7.1.4	Test Item Properties	55
7.2	Smokey Lua API	56
7.2.1	Command Execution API	56
	<i>Shell</i> Function	56
	<i>Last</i> Table	56
7.2.2	PDCA Reporting API	57
	<i>ReportData</i> Function	57
	<i>ReportAttribute</i> Function	58
7.2.3	Text Output API	58
	Text Output Routing	58
	<i>WriteString</i> Function	58
	<i>PrintString</i> Function	59
	<i>PrintStep</i> Function	59
7.2.4	File Locator API	59
	<i>require</i> Function	59
	<i>FindSequenceFile</i> Function	60
7.2.5	System Inspection API	61

	<i>PlatformInfo</i> Function	61
7.2.6	Module Name API	61
	Module Context	61
	<i>ModuleName</i> Function	62
	<i>SubmoduleName</i> Function	62
	<i>ChildModuleName</i> Function	62
7.3	Data Submission to PDCA	62
7.4	User-Defined Files	63
7.5	Exception Handling	63
7.6	Sequence Functions	63
7.6.1	<i>ActionToExecute</i> Prototype	63
7.6.2	<i>FailScript</i> Prototype	64
7.7	Test Arguments	64
7.7.1	Argument Syntax and Specification	64
7.7.2	Argument Override and Overlay	65
7.7.3	Global Arguments	66
7.7.4	Test Item Arguments	66
7.7.5	Arguments in Use	66
7.8	External Code Dependencies	67
7.9	Sequence Development Quick Start	67
7.10	Developing Code on DUT	67
8	Developing Modules	69
8.1	Lua Modules and the Smokey Environment	69
8.2	Module Convention and Terminology	69
8.3	Hierarchical Modules	70
8.3.1	Locating Hierarchical Modules	70
8.3.2	Hierarchical Module Access	70
8.3.3	Referring to Related Modules	71
8.4	Module Versions	72
8.4.1	Loading Versioned Modules	72
8.4.2	Versioned Module Folder Layout	72
8.5	Versioned Module Life Cycle	73
8.5.1	Branching and Submission Considerations	73
8.5.2	Choosing a Module Version	74
8.6	Transitioning to Versioned Modules	74
8.7	Developing Versioned Modules	75
8.7.1	Submodule Name Resolution	75
8.7.2	Module Member Scoping	76
8.8	Platform-Specific Modules	76
8.8.1	Platform-Specialization for Modules	76
8.8.2	Platform-Specific Module Entry Point	77
8.8.3	Complex Platform-Specific Modules	77
9	Using Smokey Shell	80
9.1	Smokey Shell Command Line Arguments	80
9.1.1	Start Interactive Interpreter	80

9.1.2	Expression Evaluation	80
9.1.3	Enable Persistence	80
9.1.4	Clear Persistence	80
9.2	Executing Commands	81
9.3	Exiting the Shell	81
10	Smokey Internals	82
10.1	Software Architecture	82
10.2	Control Files	82
10.3	Saving State	82
10.4	Schema Grammar	82
11	References	84
11.1	Useful Apple Links	84
11.1.1	Smokey Announcements Mailing List	84
11.1.2	Smokey Wiki	84
11.2	Useful Lua Links	84
11.2.1	Official Lua Home Page	84
11.2.2	Lua Executable Binaries	84
11.2.3	Official Lua 5.2 Reference Manual	84
11.2.4	Lua Programmer's Guide	84
11.2.5	Lua Pitfalls and Gotchas	84
11.2.6	Lua Community Wiki	84
11.2.7	Tutorials and Learning Guides	84
11.2.8	Modules	85

List of Figures

1	Supported Sequence Folder Layouts	15
2	Folder Layout Transition	16
3	Sequence Schema Example	17
4	Examples of Simulator Working Directories	43
5	Simulator Release Archive	49
6	All Sequence Properties	52
7	Abbreviated Global and Test Argument Properties Example	65
8	Minimal Sequence Properties	68
9	Shared Module Layout	71
10	Submodule Name API in Use	71
11	Invoking Versioned and Unversioned Modules	73
12	Simple Module Versioning	75
13	Complex Module Versioning	75
14	Encapsulation of Multiple Submodules in a Single Table	77
15	Platform-Specific Module Layouts	79
16	Encapsulation of Multiple Submodules in a Platform-Specific Module	79
17	Smokey and EFI System Architecture	83

List of Tables

1	Lua Types	3
2	Functional Return Values from Actions	20
3	Overall Sequence Result	27
4	Individual Test Results	27
5	EFI Command Failure	27
6	Test Data	27

1 Introduction

1.1 What is Smokey?

Smokey is a scripting and sequencing 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.

1.2 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.

1.3 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.

2 Lua Fundamentals

2.1 Lua

The Lua language is a product of the Tecgraf group at PUC-Rio (Pontifical Catholic University of Rio de Janeiro) and is 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 the files are 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.

2.2 Data

2.2.1. Built-in Types

Lua provides six built-in types that will be fundamental to writing test scripts. They are summarized in table 1.

All data types are first class citizens in Lua: they can be used in expressions, assigned to variables, passed as function arguments, and be returned from functions. This is notably true for functions, which may be a novel idea to programmers familiar with languages like C because it means that functions can be a unit of execution, an input, or an output.

2.2.2. Dynamic Typing

Lua variables are dynamically typed. That is, they do not have a predetermined type and will become the type of the value being assigned. Lua will make an attempt to “do the right thing” and cast between strings, booleans, and numbers when mismatched types are used together.

While there are no mechanisms to restrict types, there is a built-in *type* function to inspect types.

Type	Pass By	Description	Example
Nil	Value	Represents the lack of a value. Can be used as a placeholder. There is only one value.	<code>nil</code>
Boolean	Value	Bimodal logical state. There are only two values.	<code>true</code> <code>false</code>
String	Reference	Doubly-, singly-, and bracket-quoted text. The string can contain any character and there is no terminating sentinel value.	<code>"foo"</code> <code>'bar'</code> <code>[[baz]]</code>
Number	Value	Integers in base 10 and 16. Floating point numbers in base 10 only.	<code>123</code> -- Dec <code>0x123</code> -- Hex <code>10.5</code> -- Real
Table	Reference	Associative map from keys to values. Tables store keys and values of any type or value except <i>nil</i> .	<code>{ 1, 2, 3 }</code> <code>{ a=1, b=2 }</code>
Function	Reference	Chunk of executable code. Anonymous functions are supported.	<code>function foo ()</code> <code>return 123</code> <code>end</code>

Table 1: *Lua Types*

2.2.3. Scope

Variables in Lua are defined the moment they are assigned a non-*nil* value. They exist until the end of the code chunk in which they were defined, or until garbage collection, whichever occurs later.

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

2.2.4. Binding

Dynamic binding means that variable names can refer to something different each time they are used. In Lua, this means that a simple assignment operation can be used to change both the type and value of a variable. C programmers, in particular, should note that this applies to functions, so it is possible to rename functions or replace the underlying implementation on the fly.

Lua's binding behavior is aided by the way that Lua treats certain types as references. In the Lua virtual machine, reference variables are pointers to individual blocks of memory, each holding a single value, be it a string, table, or function. Conversely, reference values in Lua have no dedicated names and, in fact, a single value in memory can have multiple names. Each block of memory has a type, but variables can generally reference any block.

Binding is related to scoping in that scope affects the range of values that can be assigned to a variable at any given time. Additionally, a variable's scope affects which functions can modify its value and, in turn, affects its ultimate value at time of use. This is especially true if its scope is not locked down with judicious use of the *local* keyword. Developers will need to keep this in mind when investigating unexpected run-time behavior that involve overwritten variables or function references.

2.2.5. Mutation

Most values in Lua are immutable. This means that a value's contents typically do not change. For example, new strings and functions can be created and thrown away, but Lua does not provide a means to modify the characters in the string nor the commands in the function.

The lone exception is the table type. All references to a table are affected when one reference is used to add, replace, or delete key-value pairs. This can be useful, in practice, as a means to communicate data between functions, but is also a liability in terms of data integrity. Tables should be deeply copied if they must be shared without affecting the original.

2.2.6. Nil Values

In Lua, any variable that is not defined will automatically receive the value *nil*. Conversely, any variable assigned the value *nil* is effectively deleted from the virtual machine.

Developers must be diligent in checking for mistyped names. A typo or wrong letter case will not be flagged by the Lua interpreter, which can lead to very confusing run-time behavior. A text editor that supports autocomplete or sophisticated highlighting is highly recommended.

2.2.7. 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
```

2.3 Comments

Lua comments are akin to C++ comments.

2.3.1. 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'
```

2.3.2. Delimited Comments

Longer comments are delimited by a double dash and double bracket pair.

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

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

2.4 Boolean Expressions

2.4.1. 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
```

2.4.2. Logical Operators

Logical operators are spelled out.

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

2.4.3. Conditionals

Any value that is neither *nil* nor *false* is considered to be a boolean true. 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
```

2.4.4. 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.

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

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

2.5 Tables

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

2.5.1. 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 }
x = { a=1, b=2, c=3 }
```

Arbitrary strings as keys.

```
x2 = { ["a"]=1, ["b"]=2, ["c"]=3 } -- Equivalent to x
y = { ["key A"]=1, ["key B"]=2, ["key C"]=3 }
```

Arbitrary values as keys.

```
foo = "foo"
bar = function () return "bar" end
baz = { "baz" }
z = { [foo]=1, [bar]=2, [baz]=3 }
```

2.5.2. 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 generic table can be done with a *for* loop and the *pairs* iterator. Note that traversal order is not guaranteed for tables that use non-numeric keys.

2.5.3. 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 or was assigned the value *nil*.

2.5.4. 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
-- the hard way
t = {
    m = 1,
    f = function (self, a, b)
        return self.m * a + b
    end
}
```

```
-- Declare method f() inside table t
-- the easy way
t = {
    m = 1
}
function t:f (a, b)
    return self.m * a + b
end
```

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

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

```
-- Call f() the easy way
x = t:f(1, 2)
```

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!

When two or more instances of the same class are required, a constructor method will be required to generate a new Lua table and copy the members and methods. This won't be covered in detail here, but the references in section 11.2 may prove helpful.

2.6 Language Constructs

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

2.6.1. Conditionals: 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
```

2.6.2. Loops: for-do

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

```
-- Iterate on every third number from 1 to 10
for x = 1, 10, 3 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
```


2.6.3. Loops: while-do

```
-- Compute powers of 2 less than 1000
-- until foo()
x = 1
while (x < 1000) do
    x = x * 2
    if (foo(x)) then
        break
    end
end
```

2.6.4. 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)
```

2.6.5. Functions

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

```
-- Anonymous function in a variable
bar = function (x, y, z)
    return x + y + z
end
b = bar(1, 2, 3)
```

```
-- Variadic/Vararg function
function baz (p, f, ...)
    local h = io.open(p, "a")
    h:write(string.format(f .. "\n", ...))
    h:close()
end
baz("log.txt", "Temperature is %dC", 100)
```

```
-- Functions are first-class citizens
function map (f, t)
    local c = {}
    local n
    for i, v in ipairs(t) do
        n = f(v, c)
    end
    return n
end
function qux (s)
    return function (v, c)
        c.str = (c.str or s) .. v
        return c.str
    end
end
word = map(qux("hand"), { "crafts", "man", "ship" })
```

2.7 Built-in Facilities

2.7.1. String Concatenation

Strings are joined with the double-dot operator.

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

2.7.2. *string.format* Function

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

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

2.7.3. 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)
```

2.7.4. *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")
```

2.7.5. *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"
```

2.7.6. *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
```

2.8 Exceptions

2.8.1. 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 an outer code layer can expect and print them.

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

```
function g (x)
    if (x < 0) then
        error({ "Too small", x })
    end
end
```

2.8.2. Catching Exceptions

Exceptions can be caught by using Lua's *pcall* function to make a protected function call. Note that the parameters to *pcall* need to be a function value (either a function name or an anonymous function declaration) followed by that function's arguments, not a function call. On success, *pcall* returns *true* and all the values returned by the suspect function. On failure, *pcall* returns *false* and the exception data, if any.

```
-- Safely call a function with two arguments and
-- expect two return values
p, v, w = pcall(foo, 10, 20)
if (p) then
    print("foo(10, 20) = " .. v .. ", " .. w)
else
    print("caught: " .. tostring(v))
end
```

```
-- Use anonymous function to sugar-coat a method call
-- (Also take advantage of Lua tail calls)
p, v = pcall(function () return t:f(1, 2, 3) end)
if (p) then
    print("t:f(1, 2, 3) = " .. v)
else
    print("caught: " .. tostring(v))
end
```

```
-- Simple exception catcher/printer
function guard (f, ...)
    local t = table.pack(pcall(f, ...))
    local p, v = t[1], t[2]
    if (p) then
        return table.unpack(t, 2, t.n)
    else
        print("caught: " .. tostring(v))
    end
end

-- Safely execute bar(20)
guard(bar, 20)
```

Since exceptions may be of any data type, code that uses *pcall* should not expect the exception value to behave in any particular way. If the exception must be printed or manipulated, Lua's

built-in *tostring* and *tonumber* functions should be used to coerce the value to a known type.

2.9 Tricks

2.9.1. Data Alternation and Short Circuit Evaluation with 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 satisfies the operator first.

```
x = true and 1    -- Value is 1
y = nil and true  -- Value is nil
```

```
a = 3 or true     -- Value is 3
b = nil or "foo"  -- Value is "foo"
```

Note that short circuit evaluation is a consequence of the way boolean operators are defined in Lua.

```
x = true and foo() -- Calls foo()
y = nil and foo()  -- Won't call foo()
```

```
a = 3 or bar()     -- Won't call bar()
b = nil or bar()    -- Calls bar()
```

In the case of functions that take optional arguments, booleans make 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
```

2.9.2. Ternary-Like Operator

Lua does not support the question mark operator in C and C++. However, the in-line “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.

3 Smokey Fundamentals

3.1 Sequences: Scripts and Property Lists

Smokey is an scripting and sequencing environment that provides 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.

3.2 File Organization

3.2.1. Home Folder

Smokey has a fixed home. Input and ancillary files are stored under the home folder.

```
nandfs:\AppleInternal\Diags\Logs\Smokey
```

3.2.2. Sequence Name and Location

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

```
nandfs:\AppleInternal\Diags\Logs\Smokey\Sequence
```

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.

3.2.3. Sequence Folder Layout

Within the sequence directory will be a number of files. Some will be generally required by Smokey, some may be tied to the platform, and others may be specific to the sequence implementation. For illustration, figure 1 shows the different ways that Smokey sequences can be laid out. The following sections will discuss the contents of sequences and how the layouts are used.

Sequence Contents

Sequence developers are expected to match one of Smokey’s supported folder layouts. However, not all files will be required under all scenarios. The following two files are required. The

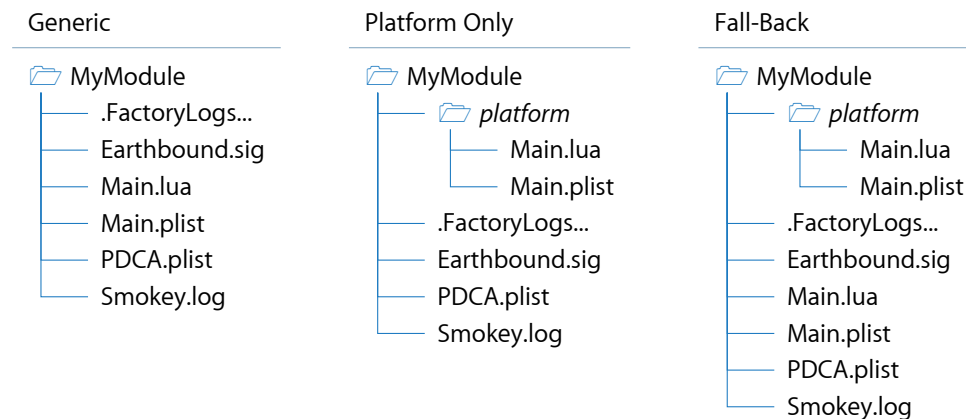


Figure 1: Supported Sequence Folder Layouts

names are fixed, but the location can vary in order to facilitate platform specialization. Both files must be in the same directory.

Main.lua — This is the script file for the sequence.

Main.plist — This is the property list file for the sequence.

The following files are output files that must be preallocated on the filesystem by the user. Refer to section 5.3 *File Output* for details on how the output files are used, how to work with them, and when they are required.

PDCA.plist — Stores sequence results, including attributes, data, and test outcome.¹

Smokey.log — Stores the running sequence log.²

.FactoryLogsWaitingToBeCollected — Specific to LogCollector.¹ This file is special and should be acquired from the EFI Diagnostics team.

Earthbound.sig — Specific to Earthbound.³

Additional user files and directories for code and auxiliary data may exist anywhere within the sequence folder and below. Smokey supports the *FindSequenceFile* and *require* functions⁴ to aid in locating these files at run time.

Sequence Platform Specialization

Test sequences can be generic and run on all platforms, or vary according to platform. This allows developers to specialize as needed.

Smokey implements support for platform specialization via directory precedence. It first looks for the “Main.lua” and “Main.plist” pair in a subdirectory with the same name as the DUT platform. Failing that, Smokey searches the top-level sequence folder. This allows a sequence to specialize for particular platforms and provide a fallback implementation where applicable. Alternatively, this also allows sequences to support all platforms with a single implementation if

¹See section 4.5.2 *File Output for LogCollector*.

²See section 5.4 *Sequence Output*.

³See section 4.5.3 *File Output for Earthbound*.

⁴See section 7.2 *Smokey Lua API*.

no platform directories are used. These use cases are illustrated by the three directory listings in figure 1.

User files can be platform-specific as well. This is done by maintaining file variations with the same name, but in different directories. Like the Lua and plist files, files in the top-level sequence folder are considered platform-agnostic, whereas specialized versions should live in separate folders per platform. The Smokey file locator API functions support this convention.⁴

3.2.4. Deprecated Sequence Folder Layout

Older versions of Smokey required a folder layout slightly different from the current design. That layout is now deprecated in order to support new sequence features, but is documented here for historical purposes.

Figure 2 illustrates the translation between the old and new layout styles. The difference lies in the moving and renaming of the Lua and plist files into their own platform-specific directory. Migration is a matter of a few filesystem operations.

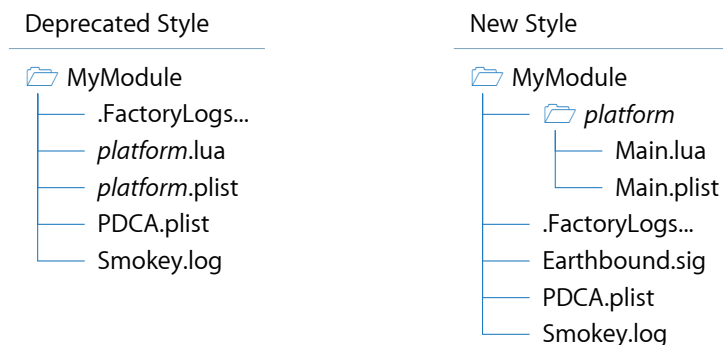


Figure 2: Folder Layout Transition

3.3 Sequence Schema

3.3.1. 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. Figure 3 shows the layout of a simple plist.

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 are detailed in section 7.1 *Smokey Properties*.

3.3.2. 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

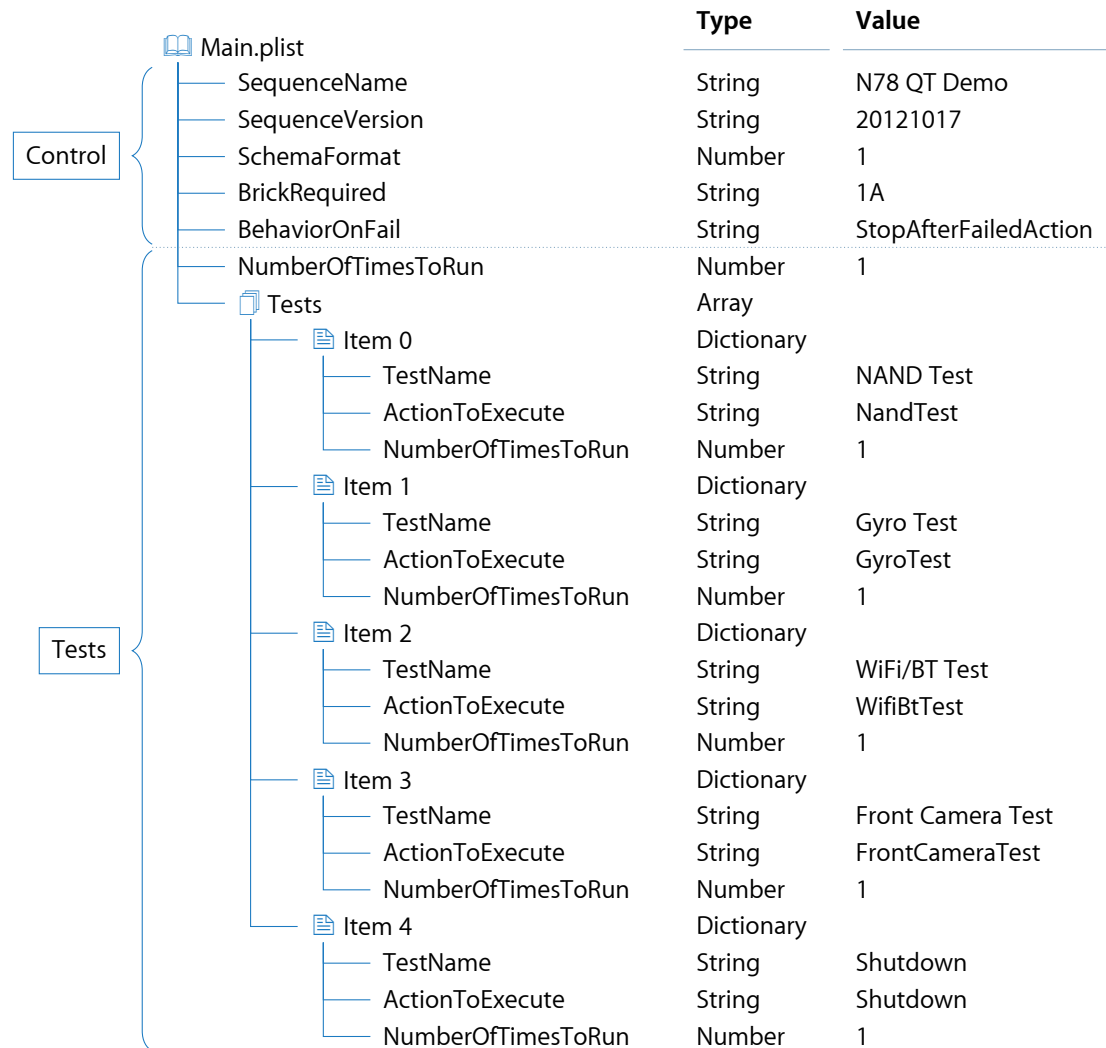


Figure 3: Sequence Schema Example

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.

3.4 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.

3.4.1. 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, configure the *boot-command* NVRAM variable.⁵
 - 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. Check for existing results. Abort if found.
6. Open log files.
7. Record the start of the sequence.
8. Initialize hardware and EFI software features implied by the sequence properties.
9. If *BrickRequired* is set, wait for an external charger to be connected. The wait time is indefinite.

3.4.2. Sequence Execution Phase

10. 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.

3.4.3. Post-Flight Phase

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

⁵See section 5.2 *Autostart* for other behavior implied by autostart.

3.4.4. 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 disconnect is detected on the external charger, wait for the charger to be reconnected. The wait time is indefinite.

3.5 Sequence Processing

3.5.1. 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.

3.5.2. 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.

3.5.3. 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.

3.6 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.

3.6.1. Function Return Value

An action's return value is the primary indicator of success. Table 2 shows the ways of functionally reporting pass and fail that Smokey supports.

Value	Type	Meaning
<i>nil</i>	Nil	Passed
<i>true</i>	Boolean	Passed
<i>false</i>	Boolean	Failed
All Others		Unsupported

Table 2: *Functional Return Values from Actions*

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*.

3.6.2. 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 data thrown with an exception will be logged. If an action chooses to throw an exception, it is recommended that a helpful error string be used.

3.6.3. Data Results Cascading

Parametric data is considered an integral component of an action's result.⁶ When bad data is reported,⁷ Smokey cascades the data failure upwards and fails the action. This, in turn, makes failure the overall sequence result.

In contrast to uncaught exceptions, data failure is a silent event. Smokey will make note of it, but there will be no interruption of execution.

3.7 Failure Handling

3.7.1. Sequence Excursions

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

⁶See section 4.4 *Parametric Data*.

⁷See *ReportData* in section 7.2 *Smokey Lua API*.

- *Test Passed* — Continue the sequence in the defined test order.
- *Test Failed* — The current *ActionToExecute* failed or a subtest in *Tests* propagated an error. Call *FailScript*,⁸ then 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 and continue the rest of the sequence. No failure is propagated to the parent test.

StopAfterFailedAction — Stop the current test item immediately and propagate the failure upwards.

StopAfterFailedIteration — Execute the remaining test items in the current iteration. Afterwards, stop the current test and propagate the failure upwards.

StopAfterFailedTest — Execute the remainder of the current iteration and complete the remaining iterations. Afterwards, propagate the failure upwards.

Tests that define the *ActionToExecute* property but not *Tests* will behave identically for *StopAfterFailedAction* and *StopAfterFailedIteration*. For either value, Smokey stops the test immediately and notifies the parent test of the failure. There are effectively only three behaviors in this case.

Tests that define the *Tests* property have four distinct failure behaviors, as described above. After each subtest immediately defined in the *Tests* array, Smokey acts upon subtest failure according to *BehaviorOnFail*. Developers have the option of stopping the test immediately, deferring the stop until the end of the current iteration through *Tests*, or waiting for all iterations to complete. In these three cases, the failure will be reported to the parent test so that it can respond according to its own *BehaviorOnFail*. A fourth option is to complete testing without propagating the failure upwards.

The *BehaviorOnFail* property is inherited. The required definition at the root of the test sequence sets the default failure behavior of all tests. Individual tests that specify their own *BehaviorOnFail* can override the inherited value and set the failure behavior for themselves as well as all tests beneath them.

The test sequence definition can locally override the inherited value of *BehaviorOnFail* with the *ThisBehaviorOnFail* property, whose values and meanings exactly match those of *BehaviorOnFail*. The override applies only to the test that defines it and it is not inherited. Both *BehaviorOnFail* and *ThisBehaviorOnFail* may be defined in the same test to set different behaviors for parent and child tests in the sequence.

3.7.2. Handling 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 a test failure.⁹ This happens for all values of *BehaviorOnFail*, including *KeepGoing*.

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

⁸See section 3.7.2 *Handling Failures During a Sequence*.

⁹See section 7.6 *Sequence Functions*.

immediately outside of that nested test item. And so on, from the epicenter to the plist root. Any test items along this path without a *FailScript* definition will be ignored.

The invocation of *FailScript* happens as soon as an error is detected. For tests with *ActionToExecute*, this happens as soon as the action fails. For tests with *Tests*, it happens when the failure propagation reaches a subtest.

3.7.3. 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 sequence execution phase is aborted.

3.8 Test Results

3.8.1. 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 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.

3.8.2. Test Results Naming

Names for results are automatically generated by combining the test item name with the iteration number.¹⁰

3.8.3. Test Data Naming

Smokey can collect parametric data and limits from each test item.¹¹ 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.¹²

3.8.4. 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 the overall sequence result to fail. Incomplete tests are treated as failures.

3.9 Sequence State Saving

3.9.1. 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

¹⁰See table 4 *Individual Test Results* on page 27.

¹¹See section 4.4 *Parametric Data*.

¹²See table 6 *Test Data* on page 27.

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 section 7.1.4 *Test Item Properties*, 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.

3.9.2. Requirements

The DUT must meet the requirements for autostart.¹³

3.9.3. 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 "State.txt".¹⁴
 - Save the state of the sequence traversal.
 - Save all test results.
- *Configure Autostart* — Set Smokey to run automatically at next boot.
 - Save the current *boot-args* and *boot-command* NVRAM variables.¹⁵
 - Add a bare "smokey" argument to *boot-args*.
 - Set *boot-command* to "diags".

3.9.4. 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.

- *Load State Data* — Resurrect the data saved during serialization.
 - Reload all internal Smokey variables.
 - Reload all test 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.

¹³See section 5.2 *Autostart*.

¹⁴See section 5.3 *File Output*.

¹⁵See section 5.2 *Autostart* for a description of how NVRAM variables are used by Smokey.

4 Design for Factory Use

4.1 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.

4.2 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.

4.3 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 be disabled by the *ControlBitAccess* property. Remember to keep this in mind if a control bit value has an unexpected value.

4.4 Parametric Data

In addition to the procedural information provided by test results, an important aspect of factory testing is the collection of measurements and numeric artifacts. This information is called **parametric data** because it carries metadata such as names, limits, and success. PDCA actually uses parametric data for all factory test output, but Smokey limits the usage of this term to the ancillary output from tests.

Smokey provides the *ReportData* and *ReportAttribute* functions specifically to allow tests to produce parametric data for PDCA.¹⁶ Consequently, the presence of parametric data is dependent on a test sequence's intent and implementation. Smokey produces no parametric data on its own.

4.5 Log Collection

4.5.1. DUT Identifier

Reporting to the factory requires that data be uniquely tied to a DUT. Smokey identifies the device under test by either the system serial number *SrNm* or the MLB serial number *MLB#* in *syscfg*, depending on *SerialNumberSource*.¹⁷

¹⁶See sections 3.6.3 *Data Results Cascading* and 7.2 *Smokey Lua API*.

¹⁷See section 7.1 *Smokey Properties*.

4.5.2. File Output for LogCollector

Smokey's file output is designed to be compatible with LogCollector, with emphasis on being human readable. Towards that goal, files are largely ASCII-compatible and Smokey's output is split across several files.¹⁸ All files in a sequence's directory will be saved to PDCA.

Two files will be of interest to those investigating issues with log collection. Refer to section 5.3 *File Output* for more details about their contents and the sequence properties that control them.

PDCA.plist — The main output file for LogCollector's consumption. Smokey writes results to this file in a PDCA-specific schema.

.FactoryLogsWaitingToBeCollected — Semaphore file for LogCollector. Indicates whether factory data is available for collection.

4.5.3. File Output for Earthbound

For burn-in testing, Earthbound can substitute for LogCollector and gather Smokey's PDCA output. The set of required files overlaps the LogCollector scenario.

PDCA.plist — Same as for LogCollector.

Earthbound.sig — Security signatures required by Earthbound. Collected files are validated using this file before being accepted for PDCA.

4.6 Encoding 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*¹⁹ are used as identifiers for individual tests and data. The *result* and *failure_message* properties, amongst others, encode sequence output. Specific values are described in tables 3–6.

Note that table 5 *EFI Command Failure* is a special case for reporting purposes. Unlike the others, it is not a function of the sequence definition, but is generated for test items that fail due to EFI diags command errors. This is in addition to the output described in table 4 *Individual Test Results*.

Also note that *result* and *failure_message* in table 6 *Test Data* are influenced by both limits and run-time overrides during the reporting API call.²⁰ Seemingly good data may be forcibly failed at run time.

¹⁸See section 3.2.3 *Sequence Folder Layout*.

¹⁹The PDCA database reports *testname*, *subtestname*, and *subsubtestname* as one concatenated string.

²⁰See section 7.2 *Smokey Lua API*.

Parameter Name	Pass	Fail	Incomplete
<i>overallResult</i>	"PASS"	"FAIL"	

Table 3: Overall Sequence Result

Parameter Name	Pass	Fail	Incomplete
<i>testname</i>	Test item's <i>TestName</i> or <i>ActionToExecute</i> value		
<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"

Table 4: Individual Test Results

Parameter Name	Pass	Fail	Incomplete
<i>testname</i>	Test item's <i>TestName</i> or <i>ActionToExecute</i> value		
<i>subtestname</i>	EFI command name		
<i>subsubtestname</i>	"Iteration " + iteration number		
<i>result</i>	Not used	"FAIL"	Not used
<i>failure_message</i>	Not used	Actual message	Not used

Table 5: EFI Command Failure

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

Table 6: Test Data

5 Using Smokey

5.1 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, but omitted from this text for brevity.

Common 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.

TestName — The name of a test item. This corresponds to the effective test name, based on the *TestName* and *ActionToExecute* properties.²¹

5.1.1. Print Version

```
smokey --version
```

Show the build date and version. There will be separate lines of output for each component of the program. Please include this information when filing bugs or requesting help.

5.1.2. 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.

5.1.3. 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.

5.1.4. 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

²¹ See section 3.5.1 *Test Item Naming*.

file output as normal. Also note that this option won't prevent a new state from overwriting the current state.

5.1.5. 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.

5.1.6. 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.

5.1.7. 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 section 5.4 for more details.

5.1.8. Get Information about Sequence Code

```
smokey Sequence --dependencies
```

Show a list of all modules, submodules, and external code statically loaded by *require*, *loadfile*, and *dofile*, as well as the main sequence code file.²² See section 5.4 for details on how the information is presented.

5.1.9. 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.

5.1.10. Clear Existing Sequence Results

```
smokey Sequence --clean
```

Clear the output files²³ of *Sequence* such that Smokey will later think that *Sequence* is being run for the first time. Can be combined with *--run* on the same command line.

Restricted to DEV-fused devices only. This feature is intended for engineering purposes only.

²²See section 7.8 *External Code Dependencies*.

²³See section 5.3 *File Output*.

5.1.11. Trigger Smokey Externally (Autostart)

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

Enable autostart behavior.²⁴ Implies `--run`.

5.1.12. Manually Select Tests to Run

```
smokey Sequence ... --test TestName
```

```
smokey Sequence ... --test TestName,TestIterations
```

Run only the tests specified and disable all other test items in the sequence.

The `--test` option can be used multiple times within the same command line. This can be used to select a single test, or a subset of tests.

The *TestIterations* argument is a suboption for setting *NumberOfTimesToRun*. When omitted, it defaults to 1.

5.1.13. Override Property Value

```
smokey Sequence ... PropName=PropVal
```

```
smokey Sequence ... Node[NodeId].PropName=PropVal
```

Properties for any part of a sequence may be temporarily overridden at the command line. Multiple overrides may be used on a single command line.

Without a node specifier, properties at the root of the sequence will be modified. Both styles are subject to schema validation against the schema.²⁵

The value for *NodeId* can be obtained from the output of `--sequence` or `--summary`.

5.1.14. Overriding Test Arguments

```
smokey Sequence ... --args 'Arg=Val'
```

```
smokey Sequence ... --args 'Arg1=Val1,...,ArgN=ValN'
```

```
smokey Sequence ... --testargs 'TestName,Arg=Val'
```

```
smokey Sequence ... --testargs 'TestName,Arg1=Val1,...,ArgN=ValN'
```

Overlay the given argument definition on top of any arguments predefined by the sequence.²⁶ Both `--args` and `--testargs` may appear multiple times on a single command line.

Global arguments are declared by `--args` and specified as a comma-separated list of key-value pairs in Lua table initializer syntax. Arguments for a single test item are declared by `--testargs` and have the same format, but also require the test item name, which may come from *TestName* or *ActionToExecute*.

Due to the layers of string parsing between the shell and Smokey, arguments may need to be wrapped with a combination of single, double, and Lua-specific square bracket quotes.

²⁴See section 5.2 *Autostart*.

²⁵See sections 3.3 *Sequence Schema* and 7.1 *Smokey Properties*.

²⁶See section 7.7 *Test Arguments*.

5.2 Autostart

On Darwin mobile platforms, the DUT can be configured to execute a Smokey 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, but it can be used for automated testing to a limited degree.

5.2.1. 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 the 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 integrating the *smokey* command into the shell's autostart configuration.

5.2.2. Configuring EFI Diags for Autostart

Smokey leverages the EFI command shell's built-in support for running commands at boot based on the *boot-args* NVRAM variable. For example, instead of going to the command line, the shell can launch the *smokey* command when "smokey" is in *boot-args*. This configuration must be made in the EFI diags image and is not something a Smokey user can do on their own.

In a pinch, autostart behavior can be partly emulated, as shown in the command line below.²⁷ This would need to be manually typed into the EFI shell.

```
smokey Sequence --autostart
```

5.2.3. 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 present. The other arguments in this variable are beyond the scope of this document.
- *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".

Refer to section 5.6.2 *Clearing Autostart* for information on how to reverse this process.

²⁷ See section 5.1 *Smokey Command Line Arguments*.

5.2.4. Smokey NVRAM Argument

The “smokey” argument in *boot-args* is a comma-separated list that is transformed into a Smokey command line.

The decision was made to put “smokey” in *boot-args* instead of its own NVRAM variable because doing so facilitates deployment: The *boot-args* variable can be easily set from PurpleRestore, whereas other variables are not well supported. DUTs can be restored with a root containing a sequence and be configured to run it on the next reboot using a single tool.

The *boot-args* syntax is shown below.

```
boot-args = "... smokey ..."  
boot-args = "... smokey=Sequence ..."  
boot-args = "... smokey=Sequence,Arg1,...,ArgN ..."
```

There is currently only one universal parameter to “smokey”.

Sequence — The name of the sequence to execute. This is ignored and may be omitted when Smokey will be continuing from a saved state.²⁸ However, omission is an internal use case and users typically must specify a sequence name.

When the EFI diags shell invokes Smokey, the “smokey” variable is split at any commas and bookended with *smokey* and *--autostart*. For the long *boot-args* example above, the conversion would result in the following:

```
smokey Sequence Arg1 ... ArgN --autostart
```

The EFI diags shell will invoke the effective command line as if it were manually entered at the console.

5.2.5. Autostart Behavior

As mentioned in the code path description in section 3.4 *Sequence Flow*, the prerequisite for autostart behavior is the *--autostart* option on the command line, which is a consequence of having “smokey” in *boot-args*.

When autostart conditions are met, Smokey will process and execute the configured test sequence. Additionally, the following tasks are performed at the start of the test cycle:

- Remove the “smokey” argument from *boot-args*. This is done by the EFI diags shell. External processes can take the removal as a sign that the DUT finished booting and has proceeded onwards to the test sequence.
- Configure the DUT to run iOS on the next reboot. Smokey does this 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.

²⁸See section 3.9 *Sequence State Saving*.

5.3 File Output

5.3.1. Sequence Output Contents

Smokey writes to a number of files with targeted data. Their purpose is described elsewhere,²⁹ so this section will concentrate on collating their content description only.

Smokey.log — This file is zero-filled on a fresh unit. Both Smokey- and user-generated text is saved to the log file. Generally, everything seen on the console is also saved to file, and vice versa, but there are exceptions based on sequence properties, API calls, and individual EFI commands.

PDCA.plist — 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. The sequence properties below are used in writing this file.³⁰

- *SerialNumberSource* — The DUT must be uniquely identified.

.FactoryLogsWaitingToBeCollected — When “PDCA.plist” has valid data, this binary file will either be zero length or filled with zeros. Otherwise, the file will either not exist or contain a control keyword.

Earthbound.sig — Security for “PDCA.plist”. Smokey updates this file whenever results are written.

5.3.2. Sequence Output Control

Some files are individually controlled by sequence properties.³⁰

- *Smokey.log* — File output can be disabled by the *LogBehavior* property.
- *PDCA.plist* — File output can be disabled by the *ResultsBehavior* property.
- *.FactoryLogsWaitingToBeCollected* — File output can be disabled by *LogCollectorControl* and *ResultsBehavior*.
- *Earthbound.sig* — File output can be disabled by the *ResultsBehavior* property.

The contents of output files are checked before a sequence is run in order to preserve existing data. In the engineering environments, this behavior can help prevent data loss due to untrained operators. However, it can also be cumbersome if retesting for failure analysis is desired. For those situations, it may be helpful to use the *ResultsBehavior* and *LogBehavior* properties together to disable file access temporarily.

Smokey does not check files that it will not use. This means, for example, that “PDCA.plist” need not be present if *ResultsBehavior* is set to *NoFile*. Sequences can be configured for limited or no file output to minimize disk utilization (i.e., fewer preallocated files) or easier deployment (i.e., smaller number of files).

²⁹See sections 3.2.3 *Sequence Folder Layout*, 4.5.2 *File Output for LogCollector*, and 4.5.3 *File Output for Earthbound*.

³⁰See section 7.1 *Smokey Properties*.

5.3.3. 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
```

The “Earthbound.sig” file can be preallocated using *dd* with an output size of 1KB.

In a pinch, the “.FactoryLogsWaitingToBeCollected” file can be created similarly. Its default content must be the word “SKIP” followed immediately by null bytes, padded out to 1KB.

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

5.3.4. Smokey Control Files

Smokey uses certain files provided by EFI diags.

State.txt — Used by the *BehaviorOnAction* property.³¹ Also read on every invocation of *smokey* to detect continuation.³²

5.4 Sequence Output

5.4.1. 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.

³¹See section 7.1 *Smokey Properties*.

³²See section 3.9 *Sequence State Saving*.

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.

5.4.2. 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: CCQH006QF4K0 MLB#: C02219500L9F4MQ1 CFG#: DA9/EVT2/00L9//2030/FT1-G2 ECID: 000001D7DC612DC9														
3. Seq. Files	Control File: nandfs:\AppleInternal\Diags\Logs\Smokey\Wildfire\N78\Main.plist Script File: nandfs:\AppleInternal\Diags\Logs\Smokey\Wildfire\N78\Main.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... <table> <thead> <tr> <th>Day/Time</th><th>Node</th></tr> </thead> <tbody> <tr> <td>[09 12:21:04]</td><td>N001 Repeating 1x</td></tr> <tr> <td>[09 12:21:04]</td><td>.... [1] Periodic tasks</td></tr> <tr> <td>[09 12:21:04]</td><td>.... Detected 1A brick</td></tr> <tr> <td>[09 12:21:04]</td><td>N002 [1] Repeating 0x "Battery Protection"</td></tr> <tr> <td>[09 12:21:04]</td><td>N003 [1] Repeating 1x "PMU Test"</td></tr> <tr> <td>[09 12:21:04]</td><td>.... [1] PmuTest</td></tr> </tbody> </table>	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
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: 0x00000000 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
```

5.4.3. 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.

5.4.4. 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.

5.4.5. Sequence Dependencies

```
:-) smokey Wildfire --dependencies

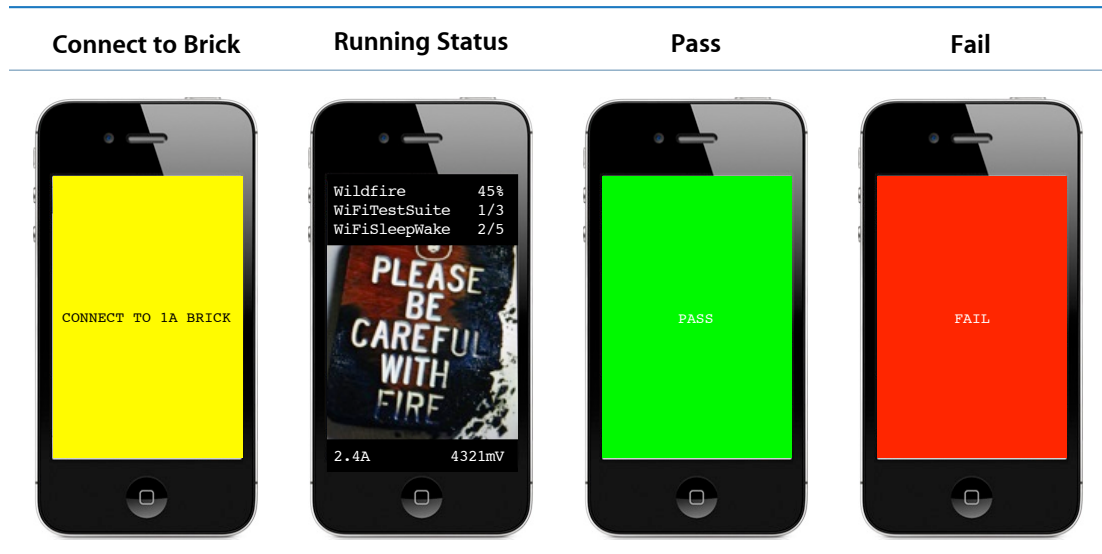
Test Sequence Load-Time External Code Dependencies

Require: Battery          (...\\Wildfire\\N51\\Battery.lua)
Require: DiagsParser.11A  (...\\Shared\\DiagsParser\\11A\\init.lua)
Require: FATP             (...\\Wildfire\\N51\\FATP.lua)
Require: Leakage          (...\\Wildfire\\N51\\Leakage.lua)
Require: MesaProvisioned.11A (...\\Shared\\MesaProvisioned\\11A\\init.lua)
Require: Syscfg.11A       (...\\Shared\\Syscfg\\11A\\init.lua)
Require: Syscfg.11A.syscfg (...\\Shared\\Syscfg\\11A\\syscfg.lua)
Source File: nandfs:\\AppleInternal\\Diags\\Logs\\Smokey\\Wildfire\\N51\\Main.lua
Source File: nandfs:\\AppleInternal\\Diags\\Logs\\Smokey\\measure_vbat_dcr\\N51\\Main.lua
```

The `--dependencies` option shows the modules and Lua files that are loaded through standard means, as well as the sequence code itself.

- *Modules* — Top-level modules and consequential submodules loaded with the *require* function. The module name and file location are both shown.
- *Sequence Code File* — The main Lua file for the sequence code.
- *Lua Files* — Individual source files loaded with *loadfile* or *dofile*.

5.5 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 sequence name, progress, current test, and power info are 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.

5.6 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 section 3.9 *Sequence State Saving* before proceeding.

5.6.1. 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 in section 5.1 *Smokey Command Line Arguments* 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.

5.6.2. Clearing Autostart

Aborting or cancelling an autostart of Smokey requires that the DUT be intercepted before it boots into EFI diags, then disabling the autostart configuration. This will allow EFI diags to boot to the interactive command line instead.

The work boils down to either catching the DUT before it reboots or interrupting iBoot before it loads EFI diags. The particular details of doing so this won't be covered here, but any method that allows safe access to the respective command line should work.

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

- *boot-args* — The “smokey” argument must be removed.
- *boot-command* — This will likely be set to “diags” during autostart. If the goal is not to boot into EFI diags, then *boot-command* 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. Smokey doesn't modify this variable.

Additional work beyond the scope of this document may be required if the DUT's particular build of EFI diags is configured to autostart software other than Smokey.

6 Using Smokey Simulator

Smokey Simulator is a companion tool, built from the same source code as Smokey, for executing and validating sequences from the command line. Whereas the *smokey* command in EFI diags runs on a DUT, the simulator runs on a host and provides a comparable level of functionality for the Mac OS environment. Additionally, it provides the capability to delegate EFI diags commands to a linked DUT.

6.1 Differences from Smokey

Internally, the simulator is a port of Smokey, so they are largely identical. This means the simulator can be used for validation and syntax checking of test sequences, and, furthermore, to execute the sequences themselves.

Differences arise during interactions with the run-time environment. Deviations occur because Smokey Simulator runs on a host and has the point of view of a Mac OS application, whereas Smokey runs on a DUT and has the perspective of an EFI diags application, so their views of the world are different.

- *File Paths* — Both Smokey and Smokey Simulator pass along file paths to the run-time environment verbatim, so they are identical in this regard. However, because a DUT will require EFI-style paths and a host will require UNIX-style paths, tests will need to either special-case the difference themselves or use Smokey API functions like *FindSequenceFile*³³ to factor out that logic.
- *Input Files* — Sequences and shared modules are treated as being in the current working directory by default. Relative and absolute path overrides are supported for input files that are spread across the host filesystem.
- *Output Files* — File output is written to the input directories configured at the start of the simulation. Contrast this to Smokey on a DUT, where input locations are fixed, so output files are therefore at fixed locations as well.
- *Ancillary Files* — Smokey uses certain external files that are laid down during a restore. Smokey Simulator likewise requires these files to pre-exist and also creates files of its own to emulate DUT processes.
 - *State.txt* — Prerequisite. File must exist, but can be empty.
 - *ControlBit.txt* — Output file. Created at run time.
- *Control Bits* — Since Mac OS does not have control bits in the same sense as Apple mobile products, reads and writes are faked. Control bits always read “Untested”. Writes are recorded to the ancillary output file “ControlBit.txt” in text format.
- *NVRAM* — Non-volatile system variables are not supported. Any Smokey actions in the simulator that require the “smokey” argument shall behave as if it were not set.

³³See section 7.2 *Smokey Lua API*.

- *Platform Identification* — Smokey Simulator promulgates the fictitious Z99 platform to test sequences. Command line settings can be used to override this identity piecemeal, as can link mode. Lua code can call the *PlatformInfo*³³ API function to detect whether the sequence being executed by Smokey Simulator.
- *Autostart* — Not supported.
- *External Commands* — In the simulator, the Smokey API function *Shell* ultimately calls the C library function *system*. As a result, external commands are routed through the host system's shell. This behavior can be used, to a limited degree, for targeted validation if the inputs and outputs of EFI diags commands can be emulated by mock command scripts. When live execution of EFI commands is required, link mode should be used.

6.2 Emulating the DUT Work Environment

Smokey assumes fixed locations³⁴ on the DUT for storing test sequences, modules, and ancillary files. Smokey Simulator does not make the same assumption and must consider the layout of the host filesystem. We can reconcile the two by imagining that Smokey uses its home folder as the current working directory. Thus, the sequences names in a *smokey* command line can be seen as relative paths to directories. Substituting the *smokey* command name with *SmokeySimulator* results in a simulator command line that mimics its EFI counterpart. This is the recommended way of calling with the simulator because it minimizes command line arguments and makes switching between environments easier.

Emulating the DUT work environment requires a directory configured with the proper files and folders. One way to get such a directory is to use the “Smokey” directory from a full EFI diags root, restore bundle, or the EFI factory scripts repositories³⁵. Another way is to use the simulator release archive, which also includes workflow helper files. Lastly, a directory can be created from scratch, as outlined below. Once the directory is set up, use the shell *cd* command to change the working directory before invoking Smokey Simulator.

Manually crafting a directory tree for Smokey Simulator involves creating ancillary files and setting up the corequisites of the test sequence of interest. There is currently only one required ancillary file, “State.txt”, so most effort will center around the auxiliary files and modules of the sequence.

In the very minimal case, three files must be created, as shown in figure 4. Advanced development environments may have multiple sequences in the working directory, as well as a “Shared” directory for shared modules. In both scenarios, it may be helpful to either copy or create a symbolic link to the simulator binary.

Note that Smokey Simulator can still run if the above layout is not possible. Refer to the command line reference for the arguments required to tell the simulator where to find requisite files.

³⁴See section 3.2 *File Organization*.

³⁵Formerly Blaze, but presently FactoryLLScriptsCommon or FactoryLLScriptsPlatforms.

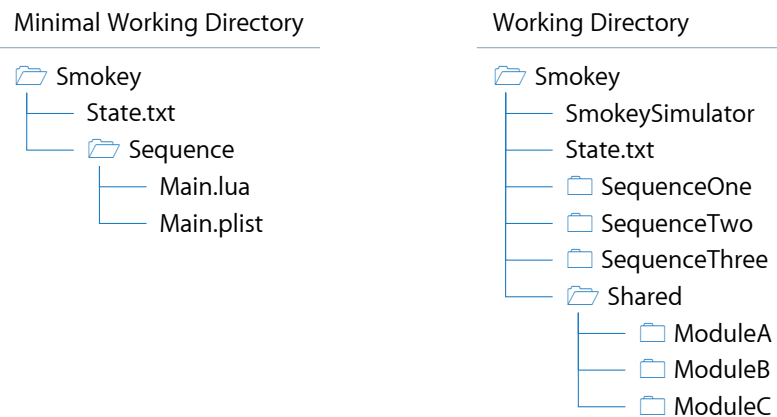


Figure 4: Examples of Simulator Working Directories

6.3 Link Mode

Link mode is a method of delegating the execution of EFI diags commands to a tethered DUT. Rather than merely simulating a test sequence via dry run or mock command scripts, it is possible to execute a sequence on a host very close to the way it would on a DUT. Additionally, link mode shortens edit-run-debug cycles by skirting the need to transfer files back and forth, and optimizing the number of DUT reboots.

6.3.1. System Requirements

The simulator link is an RPC mechanism built upon a UART channel between host and DUT. Several pieces of hardware and software are required on both sides to make it work.

- Host-side requirements
 - Recent build of Smokey Simulator
 - Bacon, Kong, or similar cable for UART communication
 - *usbterm* application (optional)
- DUT-side requirements
 - EFI diags with RPC-enabled version of Smokey
 - iBoot-based firmware (optional)

All recent versions of EFI diags satisfy the RPC requirement. If a manual check is required, run the following command on the DUT:

```
smokey --rpc
```

This will produce an *RPC>* prompt if the diags image is compatible. Otherwise, it will complain about unsupported command line arguments. To exit RPC mode and return to the EFI command shell, press **ctrl****D**.

Likewise, the simulator can be checked by either using the *--link* option or looking at the *--help* output.

Both *usbterm* and iBoot are required only when using Smokey Simulator to boot the DUT into EFI diags. The simulator option *--diags* will require both iBoot and *usbterm*. iBoot alone is sufficient when booting the diags image already in the DUT's filesystem.

6.3.2. Setting Up the Host and DUT

Before the simulator link can be used, the DUT must be configured and the host environment must be set up.

- Host-side configuration
 - All other software using the UART link, including *nanokdp*, must be terminated
- DUT-side configuration
 - Disable *auto-boot* in iBoot (optional)
 - Root the desired EFI diags image onto the DUT (optional)
 - Run *usbterm* in the background (optional)
 - DUT must be in either iBoot or EFI when starting the link

All of the optional DUT settings center around boot behavior. It is recommended that *auto-boot* be disabled at the least. However, engineering efforts may require loading custom software, so Smokey Simulator has allowances for getting the DUT into link mode. The following list will document the scenarios and settings involved, but won't delve into the nuances of all the possible permutations of configuration settings.

- If the DUT will always be in EFI at the start of tethering and the test sequence does not reboot the DUT, all the optional boot settings may be omitted. When reboots are involved, the remaining scenarios must be considered.
- If the EFI diags image on the DUT is sufficient, tethering can start with the DUT in either diags or iBoot. Reboots of the DUT during the test sequence will require that *auto-boot* be disabled; alternatively, *auto-boot* can be left enabled if *boot-command* is set to "diags".
- If a custom EFI diags is required and engineering efforts allow the DUT to be modified, it sometimes saves time to root diags onto the device. Doing so will obviate *usbterm*. The DUT still needs a way to get into EFI diags, so it must either be put into diags manually or left in iBoot for Smokey Simulator to handle. If reboots are expected, see the notes above about *auto-boot* and *boot-command*.
- If a custom EFI diags is required and the DUT's filesystem must be left intact, Smokey Simulator can load the diags image dynamically. This requires *usbterm* to be running in the background. Smokey Simulator will need access to iBoot, so the DUT must be in recovery mode at the start of tethering. If reboots are expected, *auto-boot* must be disabled so that the simulator has a chance to re-load the diags image.

6.3.3. Disabling Auto-Boot in iBoot

The exact way to disable *auto-boot* will vary by environment. To disable from iOS:

```
nvram auto-boot=false
```

To disable *auto-boot* from iBoot:

```
setenv auto-boot false
saveenv
```

To disable *auto-boot* from EFI:

```
nvrn --set auto-boot false
nvrn --save
```

6.3.4. Choosing a Link Device

Link mode requires the use of a communications channel between the host and DUT for remote procedure calls. The current implementation is built on top of UART in order to have wide compatibility with EFI, iBoot, and other environments.

Those using EFI diags are already familiar with the serial port on the DUT dock connector. This will typically be the same device used for the simulator link because it is common to all environments on Darwin platforms. In circumstances where the DUT will not leave EFI diags, it is possible to use an alternate UART such as USB CDC, which is much faster than standard serial devices.

Smokey Simulator needs the UNIX path to a character device representing the UART connection to the DUT. This is typically a file under the `/dev` directory with the prefix “cu.” followed by the device type and a unique identifier. The example below shows such a device provided by the Kong hardware debugger.

```
/dev/cu.kong-20063F
```

Alternatively, here is an example of a UART device exported by the EFI diags USB CDC emulation driver.

```
/dev/cu.usbmodemfd1411
```

Users of the *nanokdp* command line option `-d` can use the same argument for the Smokey Simulator `--link` option. Other users of *nanokdp* can find the character device of an open session in the escape menu.

The simulator will typically require exclusive access to the UART device. Attempting to share the link device may cause lost characters or indefinite stalls for data.

6.3.5. Using Link Mode

The main design goal of link mode is to closely emulate the execution of a test sequence as it would behave on a DUT. This is accomplished by remotely executing a subset of the Smokey API on the DUT. Invocations of the functions below will actively use the DUT link.

- *Shell* — EFI command lines are executed remotely. Output is captured and sent back to the host.

Additionally, the following features are proxied to the DUT.

- *Device Information* — Hardware information such as (but not limited to) platform name, serial number, and board ID are read from the DUT directly. Software build information such as the EFI build version is also queried.
- *Charger* — Brick identification and battery charging will happen as if Smokey were running on the DUT.

- *Status Screen* — Smokey’s sequencer status screen is displayed on the DUT’s screen and reflects the host’s progress as Smokey Simulator runs.

6.3.6. Quitting a Link

Smokey Simulator will automatically terminate link mode and shut down the RPC tunnel when it exits cleanly. This will typically leave the DUT at the EFI command line.

The simulator will also terminate link mode if it detects a reboot or shutdown. The DUT’s ultimate state will depend on the commands being executed at the time.

If the simulator gets hung for some reason, it can be interrupted with `ctrl C`. However, doing so can leave the DUT in the middle of the RPC tunnel. To clean up the DUT, connect to it using *nanokdp* and press `ctrl D` to quit. It may be necessary to press `return` a few times to get its attention before doing this.

6.3.7. Continuing a Link

Smokey Simulator will quietly and immediately exit if it detects that the DUT has shutdown or rebooted. This mimics the behavior of the DUT as seen with the *smokey* command. When this happens, and *BehaviorOnAction* was set to *SaveState*, Smokey Simulator can continue running the sequence where it left off, with the sequence and simulator settings that were in effect at that time, including link mode, without reiterating the previous command line options. To do this, simply use the `--run` argument by itself.

```
❏ SmokeySimulator --run
```

If this is not desired, use the following command—by itself—to make Smokey Simulator forget the previously saved state.

```
❏ SmokeySimulator --clearstate
```

6.3.8. Loading an EFI Diags Image

In link mode, Smokey Simulator will automatically boot EFI diags if it detects that the DUT is in recovery mode. By default, the diags image on the DUT will be used, but a custom image can be specified at the command line³⁶ for advanced development and debug scenarios.

See *Setting Up the Host and DUT* for configuration requirements. Generally, *auto-boot* should be disabled and the DUT should be at the iBoot prompt.

Additionally, the *usbterm* program is required in the background. If more than one device is connected to the host, the command line option *-locationid* can be used to specify the DUT that Smokey Simulator will be controlling.

Once the link is up and running, the start-up banner will display the version number of the EFI diags image on the DUT, underneath the simulator copyright and version lines. For example:

```
❏ Smokey Simulator - Copyright (C) 2012-2014 Apple Inc
  Simulator 1A0035 (changelist 343409) 2014/02/24 15:10:09
  N51 GreatDane 13C0586ae (changelist 42055) 2013/08/03 18:24:18
```

³⁶See 6.4 *Smokey Simulator Command Line Arguments*.


The above shows a DUT based on the N51 platform running EFI diags version 13C0586ae, as well as the date and time of the build.

6.4 Smokey Simulator Command Line Arguments

Smokey Simulator builds upon the command line arguments of Smokey. In addition to the conventions and usage described in section 5.1, several simulator-specific arguments are available.


In the descriptions that follow, *SmokeySimulator* should be substituted with the correct path to the simulator binary.

6.4.1. Specifying a Sequence

 **SmokeySimulator** *Directory*


Sequence names are treated as directory paths. *Directory* can be a bare name—unadorned with path separators—if it is in the current directory. Relative and absolute paths are also supported. Trailing slashes are tolerated in order to accommodate shell tab completion.

6.4.2. Specifying a Shared Directory

 **SmokeySimulator** ... **--shared** *SharedDirectory*

Smokey searches for shared code in a predefined directory. In the simulator, the default path is “Shared”, which is presumed to be in the current directory, but can be overridden from the command line. The first use of *--shared* replaces the built-in default with *SharedDirectory*. Subsequent uses append additional paths to the search list.

6.4.3. Booting an EFI Diags Image


 **SmokeySimulator** ... **--diags** *ImagePath*

When link mode is enabled, load a specific EFI diags image if the DUT is in recovery mode (i.e., at the iBoot prompt). If the DUT is already in diags, the currently running version will be used instead.


The file *ImagePath* must be either “.img3” or “.img4” format, as appropriate for the DUT.

While the image is loading, *usbterm* must be running in the background to facilitate the transfer. Additionally, Apple Connect may be required for security reasons.

6.4.4. Override Platform Identity

 **SmokeySimulator** ... **--platform** *PlatformName*

 **SmokeySimulator** ... **--model** *ModelName*

 **SmokeySimulator** ... **--boardid** *IdNum*

 **SmokeySimulator** ... **--boardrevision** *RevNum*

Override the simulator’s and the test sequence’s default view of the platform under test. These options affect several functional areas.

Smokey uses the platform name when finding sequence and module files. The option `--platform` influences this behavior and can be used to alter the normal processing of a test sequence.

The *PlatformInfo*³⁷ API function will relay the *PlatformName*, *ModelName*, *IdNum*, and *RevNum* values to test code.

6.4.5. Enabling DUT Link

```
❏ SmokeySimulator ... --link UartPath
```

Enable link mode. The argument *UartPath* must point to a UART-like character device in the filesystem.³⁸

No other program shall use the device at *UartPath* concurrently with Smokey Simulator. The most common cause for failure to enable link mode is a background task using *UartPath* while the simulator accesses it.

This option overrides all of the platform identity options.

6.4.6. Logging DUT Link

```
❏ SmokeySimulator ... --rxlog LogPath
```

When link mode is active, save raw data received from the DUT to *LogPath*. This option has no effect outside of link mode.

When a new link is established, the file *LogPath* is overwritten. Data is appended to *LogPath* when Smokey Simulator reuses a link from a previous session, such as when it continues from a saved state.

6.5 Distribution

The simulator is distributed via SDK and standalone archive. The SDK release makes the latest Smokey Simulator build readily available to developers who have access to daily iOS builds. Standalone releases are done regularly and include example files.

Simulator releases are built independently of each other, so feature sets are not in lock step, but they will generally have parity. Neither are explicitly synchronized with builds of Smokey for EFI diags.

6.5.1. Smokey Simulator SDK Binary

The Smokey Simulator binary is distributed as part of the iOS UI and NonUI portion of the Apple-internal SDKs. When Xcode command line tools are available, the simulator binary can be invoked by name, as shown below. Any arguments past the simulator command name are passed directly to the simulator.³⁹

```
❏ xcrun --sdk iphoneos SmokeySimulator ...
```

³⁷ See section 7.2 *Smokey Lua API*.

³⁸ See section 6.3.4 *Choosing a Link Device*.

³⁹ Refer to section 5.1 *Smokey Command Line Arguments* for command line tool conventions.

When the overhead of calling *xcrun* must be minimized, it is possible to print the absolute path to *SmokeySimulator* so that it may be stored in a shell variable, alias, or symbolic link.

```
❏ xcrun --sdk iphoneos -f SmokeySimulator
```

6.5.2. Release Archive

Archive Contents

The Smokey Simulator archive is distributed as a “.zip” file on the Smokey wiki.⁴⁰ In addition to the simulator binary, the archive also includes ancillary files, a “Makefile”, sample sequences, and a directory for shared modules. A sample directory listing is shown in figure 5.

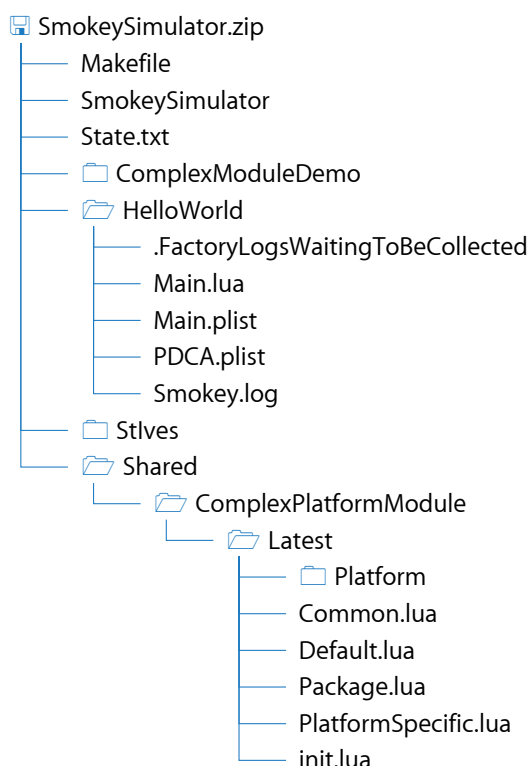


Figure 5: Simulator Release Archive

Using the Simulator Makefile

The included “Makefile” has rules for basic interaction with Smokey files.

```
❏ make files
```

The *files* target will regenerate a blank “State.txt” file, or create one if it does not exist. The “.FactoryLogsWaitingToBeCollected”, “PDCA.plist”, and “Smokey.log” files for the bundled sequences will be overwritten with fresh copies.

```
❏ make clean
```

⁴⁰See section 11.1 *Useful Apple Links*.

The *clean* target will erase any ancillary output files that the simulator creates during its execution.

7 Developing Smokey Sequences

7.1 Smokey Properties

The Smokey sequence schema is comprised of key-value pairs defining tests, flow control, and overall behavior. Some properties are nested, meaning that they can encapsulate other properties. Others are cascading, meaning that their settings affect nested properties. The following sections will describe the plist types, legal values, placement, and purpose of sequence properties. Refer to figure 6 for an illustration of all properties and their use.

7.1.1. Mandatory Control Properties

Property Name	Type	Req/Default
<i>SequenceName</i>	String	Required
<i>SequenceVersion</i>	String	Required
<i>SchemaFormat</i>	Number	Required
<i>BrickRequired</i>	String	Required

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

- **SequenceName** — Text string describing the sequence. This can be more elaborate than the sequence directory name.
- **SequenceVersion** — User-defined identifier to track changes and revisions. 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.

		Type	Comment
Mandatory	Main.plist		
	SequenceName	String	Required
	SequenceVersion	String	Required
	SchemaFormat	Number	Required
Optional	BrickRequired	String	Required
	LogBehavior	String	
	ResultsBehavior	String	
	SerialNumberSource	String	
Test Root	LogCollectorControl	String	
	ControlBitAccess	String	
	FailScript	String	
	NumberOfTimesToRun	Number	Required
Test Items	BehaviorOnFail	String	Required
	Arguments	Dictionary	
	ArgName ...	String	Lua expression
	Tests	Array	Required
Nested Tests	Item ...	Dictionary	
	TestName	String	
	FailScript	String	
	NumberOfTimesToRun	Number	Required
	BehaviorOnAction	String	
	BehaviorOnFail	String	
	ThisBehaviorOnFail	String	
	Arguments	Dictionary	
	ArgName ...	String	Lua expression
	ActionToExecute	String	Required
	Item ...	Dictionary	
	TestName	String	
Nested Tests	FailScript	String	
	NumberOfTimesToRun	Number	Required
	BehaviorOnFail	String	
	ThisBehaviorOnFail	String	
Nested Tests	Tests	Array	Required
	Item 0	Dictionary	
	Item 1	Dictionary	
Nested Tests	Item ...	Dictionary	

Figure 6: All Sequence Properties

7.1.2. Optional Control Properties

Property Name	Type	Req/Default	Related
<i>LogBehavior</i>	String	Optional (<i>Full</i>)	
<i>ResultsBehavior</i>	String	Optional (<i>Bookend</i>)	
<i>SerialNumberSource</i>	String	Optional (<i>SrNm</i>)	<i>ResultsBehavior</i>
<i>LogCollectorControl</i>	String	Optional (<i>Semaphore</i>)	<i>ResultsBehavior</i>
<i>ControlBitAccess</i>	String	Optional (<i>Default</i>)	

These properties fine tune Smokey behavior. Like the mandatory control properties, these must be at the root of the plist.

- **LogBehavior** — Control Smokey-generated output during a test sequence. Any output before Smokey can determine the effective value of this property will be buffered. Smokey may override this property's value and enable console output if it must abort execution.

Full — Enable writing to both console and the log file. This is the default when *LogBehavior* is not defined.

ConsoleOnly — Disable writing to the log file.

FileOnly — Disable writing to console.

NoLogging — Disable both console and log file output.

- **ResultsBehavior** — Select the frequency of results updates. This property affects all results stored in the filesystem, including "PDCA.plist", LogCollector- and Earthbound-related files, and signatures.

Bookend — Write results to file at the start and finish of the sequence. This is the default when *ResultsBehavior* is not defined.

AlwaysWrite — In addition to *Bookend* behavior, results are updated as soon as new attributes or data are reported,⁴¹ or actions are completed.

NoFile — Disable writing results to file.

- **SerialNumberSource** — Specify the source of the serial number that will be used when reporting test results. The effects of this property are contingent on *ResultsBehavior*. It is considered a fatal error when the specified identifier is undefined and results are to be written.

SrNm — Use the value of *SrNm* from syscfg. This is the default when *SerialNumberSource* is not defined.

MLB# — Use the value of *MLB#* from syscfg.

- **LogCollectorControl** — Modifies Smokey's interaction with LogCollector. The effects of this property are contingent on *ResultsBehavior*.

⁴¹See *ReportAttribute* and *ReportData* in 7.2 Smokey Lua API.

Semaphore — Update “.FactoryLogsWaitingToBeCollected” at the time when “PDCA.plist” is written. This is default when *LogCollectorControl* is not defined.

None — Disable modification of “.FactoryLogsWaitingToBeCollected”. This is typically the behavior under the purview of Earthbound.

- **ControlBitAccess** — Modifies Smokey’s interaction with the DUT control bits area. Although control bits are a type of result, they are independent of *ResultsBehavior*, and controlled exclusively by *ControlBitAccess*.

Default — Enables control bit writing when appropriate for the test sequence. This is the default when *ControlBitAccess* is not defined.

ReadOnly — Control bits will not be modified.⁴² This is typically the behavior under the purview of Earthbound.

7.1.3. Test Root Properties

Property Name	Type	Req/Default	Casc?	Contains
<i>FailScript</i>	String	Optional	No	
<i>NumberOfTimesToRun</i>	Number	Required	No	
<i>BehaviorOnFail</i>	String	Required	Yes	
<i>Arguments</i>	Dict. of Strings	Optional	No	
<i>Tests</i>	Array of Dict.	Required	No	Test Item

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.

- **FailScript** — User-defined Lua function to invoke when a failure is detected.
- **NumberOfTimesToRun** — The *Tests* property will be executed this many iterations.
- **BehaviorOnFail** — Controls how Smokey proceeds after encountering a failure.⁴³ The value at this level defines default the behavior for all tests in the sequence.

KeepGoing — Continue with the rest of the sequence.

StopAfterFailedAction — Abort the sequence after an action fails.

StopAfterFailedIteration — Abort after all *NumberOfTimesToRun* iterations.

StopAfterFailedTest — Abort after all child tests are complete.

- **Arguments** — Global arguments for the sequence.⁴⁴ This is a dictionary of argument names and Lua literals.
- **Tests** — An ordered array of test items.

⁴²For retest purposes, Smokey will presently take it one step further and disable checks for previous control bit writes.

⁴³See section 3.7 *Failure Handling*.

⁴⁴See section 7.7 *Test Arguments*.

7.1.4. Test Item Properties

Property Name	Type	Req/Default	Casc?	Contains
<i>TestName</i>	String	Optional (cf. <i>ActionToExecute</i>)	No	
<i>FailScript</i>	String	Optional	No	
<i>NumberOfTimesToRun</i>	Number	Required	No	
<i>BehaviorOnAction</i>	String	Optional (<i>None</i>)	No	
<i>BehaviorOnFail</i>	String	Optional (cf. parent <i>BehaviorOnFail</i>)	Yes	
<i>ThisBehaviorOnFail</i>	String	Optional (cf. <i>BehaviorOnFail</i>)	No	
<i>Arguments</i>	Dict. of Strings	Optional	No	
<i>ActionToExecute</i>	String	Required (alt. for <i>Tests</i>)	No	
<i>Tests</i>	Array of Dict.	Required (alt. for <i>ActionToExecute</i>)	No	Test Item

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.

- **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.⁴⁵
- **NumberOfTimesToRun** — The *Tests* or *ActionToExecute* property will be executed this many iterations.
- **BehaviorOnAction** — Additional steps required before invoking *ActionToExecute*. Requires *ActionToExecute* to be defined.
 - None** — Do nothing. This is the default if *BehaviorOnAction* is not specified.
 - SaveState** — Save the state of the sequence to file.⁴⁶ If the DUT is interrupted due to power loss, reboot, or otherwise, the Smokey will automatically continue once the DUT boots up.
- **BehaviorOnFail** — Override the *BehaviorOnFail* value inherited from parent tests. The value at this level cascades as the default value for lower levels.
- **ThisBehaviorOnFail** — Override the inherited *BehaviorOnFail* for the current test only. The value at this level does not cascade.
- **Arguments** — Arguments for the associated Lua function, in the same syntax as global arguments.⁴⁴ Requires *ActionToExecute* to be defined.
- **ActionToExecute** — User-defined Lua function to invoke.⁴⁵ Precludes *Tests* in the same test item.
- **Tests** — An ordered array of test items. Precludes *ActionToExecute* in the same test item.

⁴⁵See section 7.6 *Sequence Functions*.

⁴⁶See section 5.3 *File Output*.


7.2 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 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*.

7.2.1. Command Execution API

Shell Function

 Shell(**CommandLine**)

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

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

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

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.

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.

7.2.2. PDCA Reporting API

ReportData Function

```
WithinLimits = ReportData(Name, Value, Units*, LowerLimit*, UpperLimit*,  
                          PassedOrMessage*)
```

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	Optional	Dimension and magnitude (e.g., "mV" or "s")
<i>LowerLimit</i>	Number	Optional	Must be in the same units as <i>Value</i>
<i>UpperLimit</i>	Number	Optional	Must be in the same units as <i>Value</i>
<i>PassedOrMessage</i>	Boolean	Optional	Forces data failure when <i>false</i>
<i>PassedOrMessage</i>	String	Optional	Forces data failure with failure message

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.

Specifying *LowerLimit* or *UpperLimit* will cause Smokey to individually include those limits in the PDCA results. Defining either limit will subject *Value* to bounds checking, so *Name* will be marked as passed or failed according to the strictest applicable criteria below, subject to the available limits. Smokey will automatically generate an appropriate failure message for PDCA.

$$LowerLimit \leq Value \leq UpperLimit \quad (1)$$

$$Value \leq UpperLimit \quad (2)$$

$$LowerLimit \leq Value \quad (3)$$


Data may be forcibly failed by passing either *false* or a string for *PassedOrMessage*. In the former case, Smokey will automatically generate a failure message for the datum. In the latter case, the caller-provided string is used instead. Both cases take precedence over any limit checks. Values *true* and *nil* are ignored for *PassedOrMessage*.

In addition to PDCA reporting, failed data will also affect the current test result. Refer to section 3.6.3 *Data Results Cascading*. Exceptions are not thrown for data failures.

ReportData will return *true* or *false* based on the limits provided and *PassedOrMessage*. If no limits are provided, the return value is *true* by default.

Units can be specified without defining limits.

ReportAttribute Function

 ReportAttribute(*Name*, *Value*)

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

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.

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

7.2.3. Text Output API

Text Output Routing

Smokey offers a suite of functions for generating loggable text. Each accepts a routing parameter to specify output destinations, as defined below.

"Full" or **nil** — Send output to both console and file.


"ConsoleOnly" — Send output to console only.

"FileOnly" — Send output to file only.

"None" — Ignore output.

The user output routing logic is affected by the *LogBehavior* property. Console output is controlled solely by the routing parameter. However, file output is gated if *LogBehavior* is set to *ConsoleOnly* or *NoLogging*.

WriteString Function


 WriteString(*String**, *Routing**)

Argument	Type	Req?	Comment
<i>String</i>	String	Optional	String to print
<i>Routing</i>	String	Optional	Desired destinations

Output a string with the given routing. Repeated calls can be used to generate a line of text from smaller pieces.

When interleaved with Smokey-generated text, a horizontal line of dashes is automatically emitted in order to delineate user text.


PrintString Function

 `PrintString(String*, Routing*)`

Argument	Type	Req?	Comment
<i>String</i>	String	Optional	String to print
<i>Routing</i>	String	Optional	Desired destinations

This function is largely identical to *WriteString*. However, similar to the Lua *print* function, a newline character is automatically emitted after *String*.

PrintStep Function

 `PrintStep(String*, Routing*)`

Argument	Type	Req?	Comment
<i>String</i>	String	Optional	String to print
<i>Routing</i>	String	Optional	Desired destinations

Output a string to provide a trace of test activity. Routing logic will be applied. As with *PrintString*, a newline is automatically emitted after *String*.


When Smokey's sequence trace is active, *String* is decorated to match the Smokey output. The argument *String* is printed in the context of the current test item: the generated line is preceded by a time stamp and the current node ID. *String* is indented by the depth of the current node within the sequence

When the sequence trace is inactive, *String* is unadorned.

For the purposes of delineating user- and Smokey-generated text, such as when interleaving calls to *PrintString* and *WriteString*, *String* is considered to come from Smokey. This makes it easier to delineate trace output from test output.

7.2.4. File Locator API

require Function

 `require(modname)`

Argument	Type	Req?	Comment
<i>modname</i>	String	Yes	Module name or module path

The Lua *require* function is the standard interface for loading and executing external code. The ".lua" extension is implied and must be omitted from *modname*.

Smokey configures the search path such that *modname* can be either in a shared location or specific to the test sequence. The path precedence is defined by the templates below, where *modname* is substituted for the “?” symbol.

Platform-specific shared module search paths⁴⁷

1. nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\?\Platform\Platform\init.lua
2. nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\?\Platform\Platform.lua
3. nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\?\Platform\init.lua
4. nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\?\Platform.lua

Shared module search paths⁴⁸

5. nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\?\init.lua
6. nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\?.lua


Sequence search paths⁴⁹

7. nandfs:\AppleInternal\Diags\Logs\Smokey\Sequence\Platform\?.lua
8. nandfs:\AppleInternal\Diags\Logs\Smokey\Sequence\?.lua

Search path seven is sequence- and platform-specific, so it is included only when it exists and Smokey uses the sequence definition files from that directory.

Note that *modname* follows Lua convention, so it is possible to have hierarchies of modules both in the shared directory and local to the sequence itself.

FindSequenceFile Function

 **ResolvedPath** = FindSequenceFile(*RelativePath*)

Argument	Type	Req?	Comment
<i>RelativePath</i>	String	Yes	Path to file

Given the file path *RelativePath* and the search order below, return a path to the file appropriate for the DUT. The returned string can then be used with functions like *io.open*. If the file is not found, an exception is thrown.

The search order is similar to the *require* search path.

1. nandfs:\AppleInternal\Diags\Logs\Smokey\Sequence\platform
2. nandfs:\AppleInternal\Diags\Logs\Smokey\Sequence
3. ***RelativePath***

The *platform* path is included only when it exists and Smokey uses the sequence definition files from that directory. As a last resort, Smokey will try the path *RelativePath* as-is if the previous paths do not work.


⁴⁷ See section 8.8 *Platform-Specific Modules*.

⁴⁸ See section 8.1 *Lua Modules and the Smokey Environment*.

⁴⁹ See section 3.2.3 *Sequence Folder Layout*.

7.2.5. System Inspection API

PlatformInfo Function

```
 InfoValue = PlatformInfo(InfoName)
```

Argument	Type	Req?	Comment
<i>InfoName</i>	String	Yes	Platform info selector

Programmatically inspect the device under test. Return platform-specific information related to the string *InfoName*. The type of *InfoValue* varies according to *InfoName*.

The supported values for *InfoName* are listed below. Unsupported values will cause an exception.

PlatformName — The name of the general platform as a string. This is the same as the platform name used for loading platform-specific test sequences. This is also the same as the platform name reported by the EFI diags *version* command.

ModelName — The name of the particular model within the platform as a string. For example, the model name can be used to differentiate variants of a platform that do not have a baseband radio. Where applicable, the model name may also indicate whether the DUT is a DEV board.

BoardId — A read-out of the board ID strapping pins as a number. This can be used to numerically qualify the model name, or identify whether the DUT is an MLB or DEV board.

BoardRevision — A read-out of the board revision strapping pins as a number.

Simulation — Returns a boolean indicating whether or not the current code is being executed under Smokey Simulator.

The *PlatformInfo* function is available any time Lua code is running. However, use at sequence or module load time should be limited to scenarios where Smokey's automatic platform identification mechanisms⁵⁰ are insufficient.

7.2.6. Module Name API

Module Context


Smokey can provide hierarchical information to modules while they are loaded by *require*. After a module has loaded, these functions are unsupported.⁵¹

All naming functions return a string suitable for use with *require*.

⁵⁰See sections 3.2.3 *Sequence Folder Layout* and 8.8 *Platform-Specific Modules*.

⁵¹See section 8.3 *Hierarchical Modules*.

ModuleName Function

 **FullName** = ModuleName(**Level**)

Argument	Type	Req?	Comment
<i>Level</i>	Number	Optional	Hierarchical depth


Get the full name of the currently loading module. The optional argument *Level* can be used to retrieve the names of parent modules.

Omitted — When *Level* is omitted, the full name of the current module is returned.

Positive — Values of *Level* > 0 designate the name of the module at the level relative to the top-level module. When *Level* is 1, the top-level module name is returned.

Negative — Values of *Level* < 0 designate levels relative to the current module. A value of -1 refers to the current module, -2 refers to the parent module, and so on.


SubmoduleName Function

 **FullName** = SubmoduleName(**SubName**)

Argument	Type	Req?	Comment
<i>SubName</i>	String	Yes	Relative submodule name

Construct a submodule name based on the current top-level module. The string returned is a concatenation of the top-level module name and the *SubName* parameter, with a dot between.

ChildModuleName Function

 **FullName** = ChildModuleName(**ChildName**)

Argument	Type	Req?	Comment
<i>ChildName</i>	String	Yes	Child submodule name

Construct a submodule name based on the current submodule. The string returned is a concatenation of the current module name and the *ChildName* parameter, with a dot between.

The functionality of *ChildName* is similar to, but simpler than, constructing a child module name using the arguments passed to a module by *require*.⁵²

7.3 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 section 4 *Design for Factory*

⁵²The ... (dot-dot-dot) variable holds the module parameters passed by *require*.

Use for more details on the exact conversion process.

7.4 User-Defined Files

As described section 3.2.3 *Sequence Folder Layout*, sequence folders may contain arbitrary files in addition to those required by Smokey. Developers are free to supply ancillary data files, preallocate auxiliary output files, or split the sequence’s Lua code across multiple files.

It is recommended that user files be accessed via *require* and *FindSequenceFile* rather than hard-coding any paths. This makes it much easier to migrate code from one sequence to another, and to switch environments between Smokey and Smokey Simulator. Additionally, any code that needs to be shared between sequences should be implemented as a shared module.


7.5 Exception Handling

As in regular Lua, exceptions in Smokey may be of any data type.⁵³ Smokey takes advantage of this fact to apply metadata to the error messages that it throws. Test sequences must be aware of this behavior and explicitly call *tostring* on any exceptions from Smokey API functions that are to be printed or otherwise manipulated like strings. This is good practice even for exceptions not generated by Smokey.

7.6 Sequence Functions

Smokey passes state information to the sequence functions that it calls. Note that positional arguments not mentioned here are reserved for future use.

7.6.1. ActionToExecute Prototype

 **function *ActionHandler* (*ArgTable*)**

Argument	Type	Comment
<i>ArgTable</i>	Table	Test item arguments

Smokey calls a function with prototype *ActionHandler* when it processes a test item that defines *ActionToExecute*. Test success is assessed by the return value of *ActionHandler* as well as any run-time errors.⁵⁴

Arguments from the sequence properties and command line are passed in as a single table.⁵⁵

⁵³See section 2.8 *Exceptions*.

⁵⁴See section 3.6 *Pass/Fail Criteria of Actions*.

⁵⁵See section 7.7.4 *Test Item Arguments*.

7.6.2. FailScript Prototype

```
function FailureHandler ()
```

Smokey calls a function with prototype *FailureHandler* when a test item fails.⁵⁶

No arguments are currently defined.

7.7 Test Arguments

Argument passing allows test sequences and individual test items to receive configuration either by default (via “Main.plist”) or on the fly (via the command line). Additionally, arguments allow a single Lua function to be used multiple times in a sequence with varying behavior.

Arguments are presented to sequences in three ways. Global arguments, which are meant to configure the entire sequence, are prepared by Smokey and made available to the sequence when it is first loaded. Test arguments, which are specific to a test item in the sequence, are passed to test actions at run time. Both global and test arguments are stored as key-value pairs in an argument table, so normal Lua semantics can be used to retrieve individual arguments and handle optional arguments.

The third way of accessing arguments is the global table *args* defined by Lua. This is the entire command line as provided by the shell. Smokey stops processing arguments when it sees a standalone double dash, so any command line arguments after that are fair game for test sequences. Note, however, that this method is the least preferred of the three.

7.7.1. Argument Syntax and Specification

Arguments are defined in key-value fashion. The method for specifying arguments mimics Lua syntax in order to make it easy to interact with Lua code.

On the command line, keys and values are joined by an equal sign like a Lua table initialization.⁵⁷ Key names can be bare if they are purely alphanumeric. Otherwise, key names must be quoted in order to facilitate parsing. Multiple arguments can be specified simultaneously by separating them with commas.

In a plist file, keys and values are defined in a dictionary of strings.⁵⁸ Key names are used verbatim, so there is no need to quote special characters. All plist values must be strings, even if they are ultimately converted into a different type.

For both command line and plist arguments, the argument values must be in the form of a Lua literal. Numbers are to be stored as a sequence of textual digits, strings are to be quoted, and *nil* must be bare (no quotes).⁵⁹ No special effort is made to coalesce argument values, so sequences must not assume that arguments with identical values are guaranteed references to the same entity.⁶⁰

Figure 7 shows an abbreviated example of arguments as properties.

⁵⁶See section 3.7 *Failure Handling*.

⁵⁷See section 5.1.14 *Overriding Test Arguments*.

⁵⁸See sections 7.1.3 *Test Root Properties* and 7.1.4 *Test Item Properties*.

⁵⁹Currently, tables are not fully supported.

⁶⁰See sections 2.2.4 *Binding* and 2.2.5 *Mutation*.

	Type	Value
Main.plist		
SequenceName	String	HelloWorld Example
SchemaFormat	Number	1
Arguments	Dictionary	
Designation	String	"World"
Tests	Array	
Item 0	Dictionary	
ActionToExecute	String	HelloWorld
Arguments	Dictionary	
Salutation	String	"Hello"
Item 1	Dictionary	
ActionToExecute	String	GoodbyeWorld
Arguments	Dictionary	
Salutation	String	"Goodbye"
Repetitions	String	3

Figure 7: Abbreviated Global and Test Argument Properties Example

From the command line, the same arguments can be specified as shown below.

```
smokey HelloWorld \
  --args 'Designation="World"' \
  --testargs 'HelloWorld,Saluation="Hello"' \
  --testargs 'GoodbyeWorld,Saluation="Goodbye",Repetitions=3'
```

7.7.2. Argument Override and Overlay

A differentiating trait of command line-specified arguments is that they can create, modify, or remove predefined arguments.

Smokey uses the definitions from the sequence plist to create an initial argument table in Lua. If no arguments are defined, an empty table is used instead. Key-value assignments from the command line are then executed in the context of the initial table to form the final argument table. This creates the following scenarios:

- *New Argument* — An argument name from the command line that does not exist in the argument table will create a new argument. This can be used to implement hidden or private arguments.
- *Argument Exists, Value is Non-nil* — An argument name from the command line that exists in the argument table will replace the predefined value.
- *Argument Exists, Value is nil* — Explicitly assigning an argument the value *nil* from the command line will remove the argument from the argument table.

Due to Lua's dynamic typing, the run-time type of an argument value may ultimately change from the type implied in the sequence plist definition. Additionally, arguments may appear and disappear. Sequence scripts must be tolerant of these situations.

7.7.3. Global Arguments

Global arguments defined by the *Arguments* root property or by the `--arg` command line option⁵⁷ are stored in the *GlobalArguments* table, which is a global Lua variable. The table is defined before any test code is loaded, so it is available for immediate use by the sequence.

GlobalArguments may be empty if no arguments are defined or the default arguments are deleted via `--args`.

7.7.4. Test Item Arguments

Like global arguments, test arguments are stored in a Lua table. Smokey constructs the argument table from the *Arguments* test item property and the `--testargs` command line option,⁵⁷ then passes the table to actions as the first function argument.⁶¹

The test argument table may be empty if no arguments are defined or the default arguments are deleted via `--testargs`.


7.7.5. Arguments in Use

The properties in figure 7 can be combined with the code below to create an argument-based sequence.

```
function HelloWorld (Arguments)
    local Command =
        string.format("echo '%s, %s'; ",
            Arguments.Salutation,
            GlobalArguments.Designation)
    Shell(string.rep(Command, Arguments.Repetitions or 1))
end

function GoodbyeWorld (Arguments)
    local Command =
        string.format("echo '%s, cruel %s'; ",
            Arguments.Salutation,
            GlobalArguments.Designation)
    Shell(string.rep(Command, Arguments.Repetitions or 1))
end
```

Ignoring Smokey-generated text (or setting *LogBehavior* to *NoLogging*) gives the output below with the default arguments.

```
 Hello, World
Goodbye, cruel World
Goodbye, cruel World
Goodbye, cruel World
```

Notice that “Hello” and “Goodbye” come from test arguments, whereas “World” is a global argument for use by all tests. *Repetitions* is an optional argument in the code above that defaults to 1, but is set to 3 in *GoodbyeWorld*.

⁶¹ See section 7.6 *Sequence Functions*.

7.8 External Code Dependencies

Smokey integrates tightly with Lua's *require* function in order to provide structured access to external code. This enables the *--dependencies* command line option to list the modules and source files used by test sequences.⁶²

Dependency information is collected while loading sequence code and displayed before execution of the test sequence. Smokey gathers the list from global code outside of any function definitions. Files that are manually loaded by file operations and *load*, or loaded at run time from parametric information, unfortunately, can not be detected by Smokey ahead of time, and are excluded.

7.9 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 section 3 *Smokey Fundamentals*. 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
dd if=/dev/zero of=MySequence/Earthbound.sig bs=1k count=1
echo SKIP > MySequence/.FactoryLogsWaitingToBeCollected
dd if=/dev/zero \
    of=MySequence/.FactoryLogsWaitingToBeCollected \
    count=1020 bs=1 seek=4
```

The specific selection of output files can vary with due to specific sequence properties. Some files can be omitted if Smokey will not be writing to them.⁶³

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, as shown in figure 8.

Once the files are created, they should be rooted onto the DUT.⁶⁴

7.10 Developing Code on DUT

This feature is not yet available.

⁶²See section 5.1 *Smokey Command Line Arguments*.

⁶³See section 5.3 *File Output*.

⁶⁴Consult iOS documentation for details on file roots, how they work, and how to apply them to a DUT.

	Type	Value
X99.plist		
SequenceName	String	MySequence
SequenceVersion	String	20120816
SchemaFormat	Number	1
BrickRequired	String	1A
BehaviorOnFail	String	StopAfterFailedAction
NumberOfTimesToRun	Number	1
Tests	Array	
Item 0	Dictionary	
ActionToExecute	String	MyAction
NumberOfTimesToRun	Number	1

Figure 8: Minimal Sequence Properties

8 Developing Modules

8.1 Lua Modules and the Smokey Environment

In Lua nomenclature, a module is external code that a Lua script can load into the interpreter's virtual machine. The terms "module" and "library" are almost interchangeable. Pedantically speaking, however, "module" is the proper name and is the one used in this document.

Lua modules can take the form of Lua source files, Lua bytecode, or C libraries. Within the EFI diags environment, C libraries are difficult to support in the general case, as is Lua bytecode, so modules in the Smokey environment will typically be Lua source code.

The mechanism for loading modules is Lua's built-in *require* function, which takes a module name as a parameter. The name can be in dotted notation, with a dot or period character delineating the levels of the module hierarchy; Lua uses the dots to navigate through modules the same way that it navigates a directory tree. Two generic examples of *require* syntax are shown below.

```
require "Statistics"
```

```
local Stats = require "Statistics"
```

Smokey configures *require* to use a common location for modules shared between test sequences.⁶⁵ This allows complex modules to be stored in an eponymous directory and loaded by instructing Smokey to execute the module's "init.lua" file. Once invoked, the entry point for complex modules may call *require* on any number of submodules within its hierarchy. Simple modules can be defined in a single eponymous Lua source file.

More details about standard Lua modules and a general background of how the language and interpreter support them can be found in the official "Programming in Lua" books. Pay special attention to the documentation for the Lua version used by Smokey,⁶⁶ as a lot of information floating around references features that are deprecated in later Lua versions.

The *require* function is also a general way to locate and load external code. In Smokey, it is configured to allow test sequences to spread their code over several files. The mechanism for doing so is largely the same as for modules, but will not be discussed here.

8.2 Module Convention and Terminology

Lua does not define nor enforce any policy on how modules are to be implemented or used. Consequently, modules are metaphorically unbridged islands within an archipelago and users are allowed to freely access any modules at any time. It is therefore up to convention to establish structure, so we define the following terminology:

Module — External code that can be loaded by Lua.

⁶⁵See section 7.2.4 *File Locator API*.

⁶⁶Smokey is currently based on Lua 5.2.

Module name — The name, including all dots, as specified when loading the module. This is the same as the argument to *require*. Module names are a way to relay the organization and purpose of external code.

Module path — Collection of names that represents a path from one point in a module hierarchy to another. Concatenating the names, with a dot between names, produces a module name. For example: “A”, “B”, and “C” can form a module path and have the name “A.B.C”.

Top-level module — A module that serves as the entry point for a collection of related code. Its name typically does not have dots except in the case of versioned modules. A top-level module is not considered a submodule nor a child module.

Submodule — A module that is underneath a top-level module. Its name has at least one dot. Related submodules are under the same top-level module. Submodules may be above or under other submodules within the same hierarchy.

Child module — A module that is directly underneath another module. There is exactly one step in the module path from the parent module to the child module. In the example of “A.B.C”, “A.B” is the parent module and “A.B.C” is the child module.

Nomenclature can vary according to context. Some developers refer to single modules as libraries or groups of modules as packages. For the sake of this documentation, we will simply call them modules, with the qualifiers defined above.

8.3 Hierarchical Modules

8.3.1. Locating Hierarchical Modules

It is usually straightforward to identify the location of a module file on the filesystem. However, directories with no immediate files can pose a challenge and there are two interesting scenarios.

In the first scenario, a module name in the middle of a module path is an empty directory. Assume that the module names “A”, “A.B.C”, and “A.B.D” can be used with *require* successfully. The name “A.B” might be an empty directory and therefore not valid for *require*. This can be confusing to module users who expect that any given module name will automatically load all submodules beneath it. So, while this scenario is legal, it is not encouraged.

In the second scenario, the root of a module name is an empty directory. Assume that “A.B” is a valid module, but “A” represents an empty directory. Like the previous scenario, it is legal. However, this design is rather inefficient for users and is discouraged except in the cases of versioned modules.

8.3.2. Hierarchical Module Access

Regarding module usage, Smokey expects that test sequences will reference shared modules by their top-level module name only. Module developers may be tempted to allow users direct access to submodules, but this is highly discouraged in the Smokey environment. Such functionality should be refactored into a separate top-level module instead. Figure 9 illustrates the two approaches.

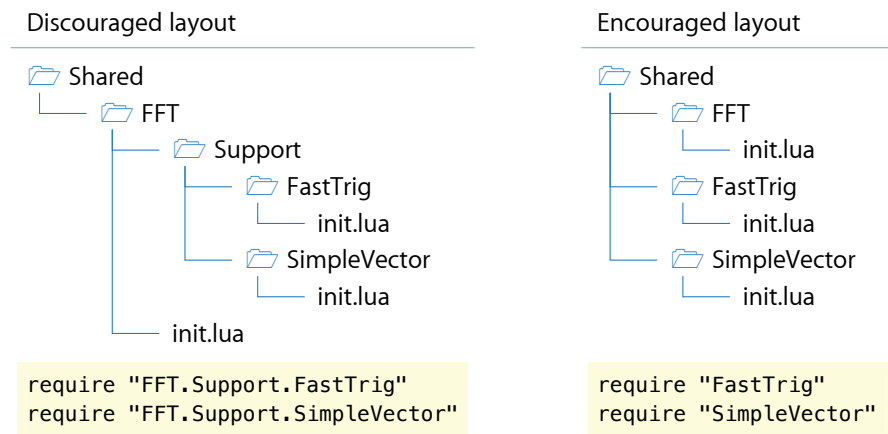


Figure 9: Shared Module Layout

8.3.3. Referring to Related Modules

Submodules of a top-level module may want to reference children modules or even other submodules within the same hierarchy. The stock Lua approach necessitates that the full name of the desired module be known, so a submodule will often need to know both its own name and also the name of the top-level module. Smokey provides several functions for composing module names that makes it easier to use `require` without hard-coding too much knowledge.⁶⁷ At the time of module loading, these functions provide a way to load other modules or record information for later use.

ModuleName — Retrieve the current module name.

SubmoduleName — Generate a module name under the same top-level module.

ChildModuleName — Generate a module name below the current module.

This allows the Lua code in figure 10 to be rewritten for Smokey.

File	Lua Code	Smokey Code
Shared\A\init.lua	<code>require "A.B"</code>	<code>require(ChildModuleName "B")</code>
Shared\A\B\init.lua	<code>require "A.B.C"</code> <code>require "A.D.E"</code>	<code>require(ChildModuleName "C")</code> <code>require(SubmoduleName "D.E")</code>

Figure 10: Submodule Name API in Use

The `ModuleName`, `SubmoduleName`, and `ChildModuleName` functions must be used within the context of a `require` call. In other words, they can not be used at run time, when tests call into a module.

Although it is possible to deduce some of the requested information statically, contextual information (like original file locations) is lost shortly after a module is loaded into the Lua virtual

⁶⁷ See section 7.2.6 Module Name API.

machine. This means that, within a module file, calls to these functions must be made outside of any function definitions, which allows Lua to execute code as it parses source. This also means that such modules must be invoked with *require* rather than Lua's *load* or *loadfile* functions. Such an arrangement allows Smokey to build context identifying the parts of the module hierarchy.

8.4 Module Versions

Smokey's module versioning is a way to enable future module development while simultaneously shielding existing users from divergent changes and allowing bleeding edge users to automatically pick up the latest code. Like standard Lua modules, versioned modules rely on convention, but they also make use of unique Smokey features.

8.4.1. Loading Versioned Modules

The mechanism for loading module versions remains the same *require* function as before. Smokey extends the *require* behavior such that a specific module version can be chosen by using the version number as the submodule name. For example: To select version "11A" of top-level module "A", pass to *require* the concatenation of the unversioned module name, a single dot, and the version number. Note that this form of module name is the only kind of top-level module name that should include a dot.

There is a special "Latest" version number that is reserved for the most recent copy of the module code. Smokey can automatically alias an unversioned module name to the "Latest" version as a convenience to users. Unversioned modules will continue to work as they did before, regardless of these features.

Smokey amends the behavior of *require* so that users do not need to be aware of the "Latest" alias. This means, for example, that the return value and side effects of *require* are the same whether the name "A" or "A.Latest" are used. Code bases that mix the two styles of names should work seamlessly.

8.4.2. Versioned Module Folder Layout

As the version naming scheme alludes to, module versions are created by putting the module's files in a subdirectory of the module's main directory. Additional versions can be created by putting code into other subdirectories of the main directory. The "Latest" version is created simply by using a subdirectory name of "Latest". This organization will help clearly delineate module code from versioning artifacts.

The names of module versions are free-form within the limits of standard Lua module names. However, it is encouraged to name them after iOS build train numbers since the development of Smokey test sequences is primarily synchronized with iOS releases, which is, in turn, synchronized to hardware product schedules.

As an example of how module versions are implemented and used, consider figure 11, which illustrates a shared module directory structure and the corresponding Lua code to load them. There are several points to notice with this hierarchical approach.

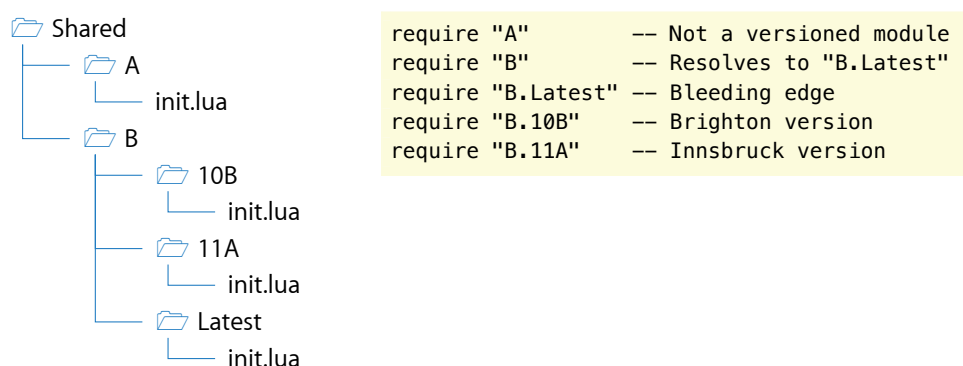


Figure 11: *Invoking Versioned and Unversioned Modules*

- *Unversioned Modules* — Module “A” is not versioned, so the normal *require* behavior will successfully find “A\init.lua”.
- *Versioned Modules* — Module “B” has three versions, one of which is the bleeding edge. Passing the module name “B” to *require* causes Smokey to automatically load “B.Latest” once it fails to find “B\init.lua”. Directly specifying the version nets the expected results because they can be found via the *require* search paths.

An important detail here is that a module should not be set up for both versioned and unversioned behavior. For example, in the module “B” above, there is no guarantee of behavior if “B\init.lua” exists. Users of module “B” must either load a specific version directly, or rely on Smokey to effect the alias from “B” to “B.Latest”. Module developers should not attempt to implement their own aliasing schemes.

8.5 Versioned Module Life Cycle

8.5.1. Branching and Submission Considerations

Shared modules in the Smokey environment will need to follow the life cycle of factory software and comply with restore bundle mastering. This means that module development needs to consider the effects of branches, separate code submissions, and how files are populated on a device under test.

Branches allow software to be stabilized while development continues on the trunk of the software tree. Modules and module versions can be branched, and changes on those branches are eligible for merging, just like any other software, so they will fit normal development processes. Unlike branches, versions have a distinction in that they can coexist multiply and contemporaneously on a DUT. This is advantageous if changes on several different branches need to be combined into a single branch (or even trunk) and minimal code merges are desired: the active code on each branch can be made into different versions, which would allow all of the branched code to live alongside each other. Of course, this source code multiplication has practical implications, so it must be weighted against the advantages of outright merging.

Because versioned modules will be shared between products, submissions to build trains must be made from a common B&I project and registered to the appropriate targets. This will ensure

that the same files are used across all devices and that there is no overlapping of roots. It is highly encouraged that all shared modules, and all common Smokey files for that matter, be submitted as one repository.

8.5.2. Choosing a Module Version

Versioned modules must take into consideration the appropriate times during product development for users to reference an established version number and when the “Latest” version should be used instead. It is generally accepted that completed products should use the version number established for the applicable build train, but there are two approaches to making the journey from the start of an engineering cycle and the final code lockdown.

One choice is to do module development on “Latest” and have users either reference “Latest” directly or rely on the alias. For example, either “A” or “A.Latest” could be passed to *require*. At the end of the cycle, a unique version number would be created and all module users would be migrated to the newly established version. This approach has the advantage of ensuring the use of the latest code, but has the disadvantage of coordinated churn for both module user and module developer at the end.

The other choice is to put bleeding edge code in “Latest” and periodically sync “Latest” to a version established for the product cycle. Users would always reference the specific version being developed, so this approach has the advantage of having code point to the correct version from the beginning. Another advantage is that experimental code can be submitted to “Latest” without immediately disturbing a product build. The disadvantage is that the synchronization must be done manually.

Regardless of which approach is taken, module developers and users should agree to a convention up front and synchronize whenever versions need to change. Remember to keep in mind that notifications must be made not only to the immediate Smokey sequences using the versioned module, but also any modules that rely on the versioned code and any sequences that rely on those modules.

8.6 Transitioning to Versioned Modules

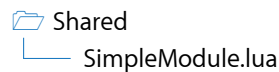
Changing an unversioned module to be versioned is straightforward. The process basically boils down to moving files into subdirectories and editing *require* calls.

A module that is a bare Lua source file would be renamed to “init.lua” and moved into a subdirectory. For example, to make “11A” and “Latest” versions of “SimpleModule”, the original “SimpleModule.lua” file would be moved and copied as shown in figure 12.

Modules that are in their own directory, modules that are spread across several files, and those that contain submodules are all handled similarly. Module files must be moved into a subdirectory of the main module directory. Next, all uses of *require* to load submodules must be changed to use the *SubmoduleName* or *ChildModuleName* functions to compose module names. The example in figure 13 shows how files move when “11A” and “Latest” versions of “ComplexModule” are created.

For the aforementioned examples, it is assumed that “11A” was the most recent milestone and development will continue with the “Latest” version. When the transition occurs, Smokey test

Simple Unversioned



Simple Versioned

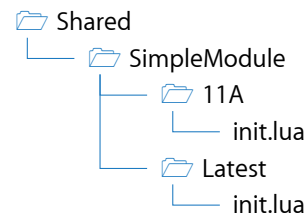
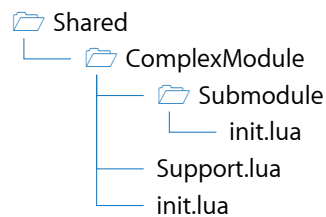


Figure 12: Simple Module Versioning

Complex Unversioned



Complex Versioned

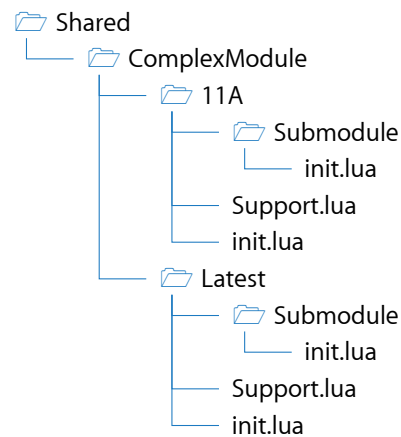


Figure 13: Complex Module Versioning

sequences that rely on old behavior must be updated to reference the “11A” version. Others can remain as-is.

8.7 Developing Versioned Modules

For the most part, versioned modules are developed just like unversioned modules. However, special care must be taken to ensure that references to files and variables resolve correctly.

8.7.1. Submodule Name Resolution

Direct references to submodules should be avoided. When the “Latest” code is branched off to various version directories, direct use of submodule names can cause *require* to load code from the wrong directory if the new versions are not immediately patched. This can be avoided by using *SubmoduleName* and *ChildModuleName* to generate module names. Both functions use Smokey to identify the top-level module name and version, which allows the correct module names to be generated on the fly. The end result is that identical *require* calls will work correctly regardless of the module version.

8.7.2. Module Member Scoping

Global variables and functions should be used carefully. This guideline originates from standard Lua modules and has important ramifications in Smokey. With versioned modules, the situation might arise that a test sequence uses distinct modules “A” and “B”, and both need to use different versions of module “C”. If module “C” exports functions or variables to the global namespace, the two versions can overwrite each other’s globals. Because of Lua’s type system, this trampling is both legal and silent, potentially causing unintended behavior that can only be detected at runtime.

The way to avoid conflicts in the global namespace is to take advantage of scoping rules and localize both the contents and use of each module. The Lua *local* keyword helps achieve this. Additionally, this paradigm is supported by the ability to pass values back to the user through the *require* function. The two can be used together with the following guidelines:

- *Localize* — Most, if not all, variables should be declared *local*.
- *Encapsulate* — A table should be created to store the environment of the module.
- *Contain* — Module state and cached information should be stored in either the module table or its metatable.
- *Expose Selectively* — Public functions and variables should be put into the module table.
- *Coalesce* — The module table should be passed back to the user. Module users should store the return value of *require* in a local variable.
- *Support* — Optionally, create a global copy of the module table for compatibility.

A key aspect of the solution is the creation and use of a module table. This puts the module interface in a scope that won’t be trampled by other versions of the same module. Another important aspect is that the semantics of *require* grant all submodules access to the same module table rather than distinct copies of the table.

Figure 14 shows an example of the above guidelines. It also demonstrates how code can be split across several files while bounding the module’s public interface within a single table. The code implements an module table and returns it to the user that called *require*. It also creates a global variable that refers to the same module table. (Note that the global instance replaces any existing variable of the same name.) Within the module table, there are submodules “A” and “B” that implement their own functionality. Both submodules take advantage of *require* semantics to install their interfaces to the same module table. The top-level module does likewise to return the populated table.

8.8 Platform-Specific Modules

8.8.1. Platform-Specialization for Modules

Under certain circumstances, modules may want to tailor their configuration or overload their API to suit the platform of the device under test. Smokey directly supports these goals in two ways. The first way is the *PlatformInfo* function for inspecting platform details.⁶⁸ The second way is to automatically pick a platform-specific entry point when loading a module.

⁶⁸See section 7.2 Smokey Lua API.

File	Code
MyModule\Latest\Package.lua	<pre>return {}</pre>
MyModule\Latest\A.lua	<pre>local Package = require(SubmoduleName "Package") Package.A = { FooText = "LatestFoo" } function Package.A:Foo () return self.FooText end</pre>
MyModule\Latest\B.lua	<pre>local Package = require(SubmoduleName "Package") Package.B = { BarText = "LatestBar" } function Package.B:Bar () return self.BarText end</pre>
MyModule\Latest\init.lua	<pre>require(ChildModuleName "A") require(ChildModuleName "B") MyModule = require(SubmoduleName "Package") return MyModule</pre>

Figure 14: Encapsulation of Multiple Submodules in a Single Table

Modules are canonically loaded by passing their top-level name to the *require* function. Smokey extends the normal *require* behavior by adding platform-specific paths to the module search path.⁶⁹ This enables developers to completely replace the default entry point, with little overhead, in order to tailor their module's behavior to the device under test.

When a user calls *require*, the platform-specific paths are queried first. If the appropriate files are found, the search stops and the platform code is loaded. If a module does not define platform-specific entry points, or no defined platforms match the DUT, the default module entry point is used.

Note that the platform-specific paths are in effect for any *require* call, not just those for top-level modules. This can have implications when loading submodules, so the module package layout must keep this in mind. On the other hand, this also means that submodules can be made platform-specific should the need arise.

8.8.2. Platform-Specific Module Entry Point

Module developers have a couple of options for taking advantage of platform-specific entry points. These are illustrated in figure 15, with one example given per platform search path. Developers are free to choose the one that fits the complexity and organization of their code.

8.8.3. Complex Platform-Specific Modules

As implied by the preceding description, the platform-specific entry point replaces the normal entry point. Each entry point must, therefore, make the effort to configure and load the entirety

⁶⁹See section 7.2.4 *File Locator API*.

of the module package. Larger, or more complex, modules can ease this burden by placing all of the common initialization in a single submodule that each of the entry points can reference.

A simple illustration of this is shown in figure 16. The example code can be extended to support model variations within a platform by combining control logic with the *PlatformInfo* function.

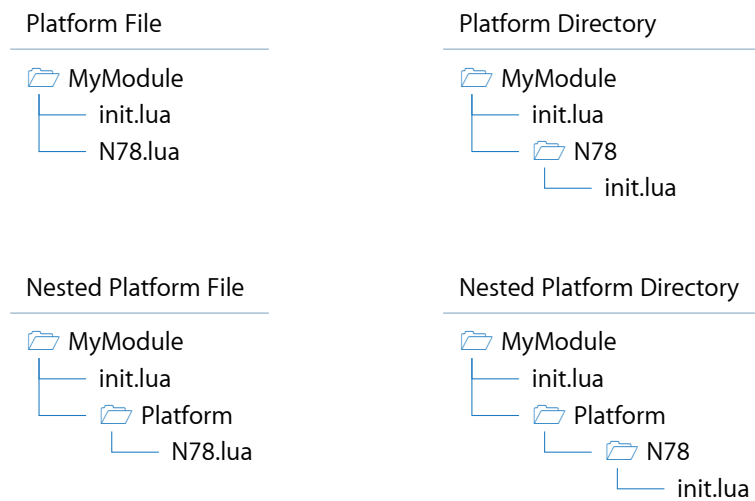


Figure 15: Platform-Specific Module Layouts

File	Code
MyModule\Package.lua	<pre>return {}</pre>
MyModule\Common.lua	<pre>local Package = require(SubmoduleName "Package") Package.Common = "Common"</pre>
MyModule\Default.lua	<pre>local Package = require(SubmoduleName "Package") Package.Feature = "Default"</pre>
MyModule\PlatformSpecific.lua	<pre>local Package = require(SubmoduleName "Package") Package.Feature = "PlatformSpecific"</pre>
MyModule\init.lua	<pre>require(SubmoduleName "Common") require(SubmoduleName "Default") return require(SubmoduleName "Package")</pre>
MyModule\N78.lua	<pre>require(SubmoduleName "Common") require(SubmoduleName "PlatformSpecific") return require(SubmoduleName "Package")</pre>

Figure 16: Encapsulation of Multiple Submodules in a Platform-Specific Module

9 Using 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.

9.1 Smokey Shell Command Line Arguments

9.1.1. Start Interactive Interpreter

```
smokeyshell
smokeyshell ... --interactive
smokeyshell ... -i
```

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.

9.1.2. Expression Evaluation

```
smokeyshell ... --execute Expression
smokeyshell ... -e Expression
```

Execute *Expression* as if it had been typed into the interactive interpreter, then return to the EFI diags shell. When used multiple times in one command line, each *--execute* is evaluated in order, from left to right.

When used with *--interactive*, all instances of *--execute* are evaluated before bringing up the interactive interpreter.

9.1.3. Enable Persistence

```
smokeyshell ... --persist
smokeyshell ... -p
```

Retain the state of the Lua virtual machine when returning to the EFI diags shell. Subsequent invocations of *smokeyshell* will use and update the persistent VM state.

When persistence is not enabled, a single-use VM is created for each invocation of *smokeyshell*.

9.1.4. Clear Persistence

```
smokeyshell ... --reset
smokeyshell ... -r
```

Delete the Lua virtual machine left behind by *--persist*.

When used with `--execute`, `--interactive`, or `--persist`, a new VM is created after the old one is deleted.

9.2 Executing Commands

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.
- *ReportAttribute* — Silently ignored.
- *ReportData* — Silently ignored.

9.3 Exiting the Shell

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

10 Smokey Internals

10.1 Software Architecture

Smokey's architecture and its interaction with EFI diags are shown in figure 17. Software that interfaces directly with the Lua VM are delineated. Also shown are the components that are loaded from the filesystem, as are those provided by the firmware image.

10.2 Control Files

Documentation not yet available.

10.3 Saving State

Documentation not yet available.

10.4 Schema Grammar

Documentation not yet available.

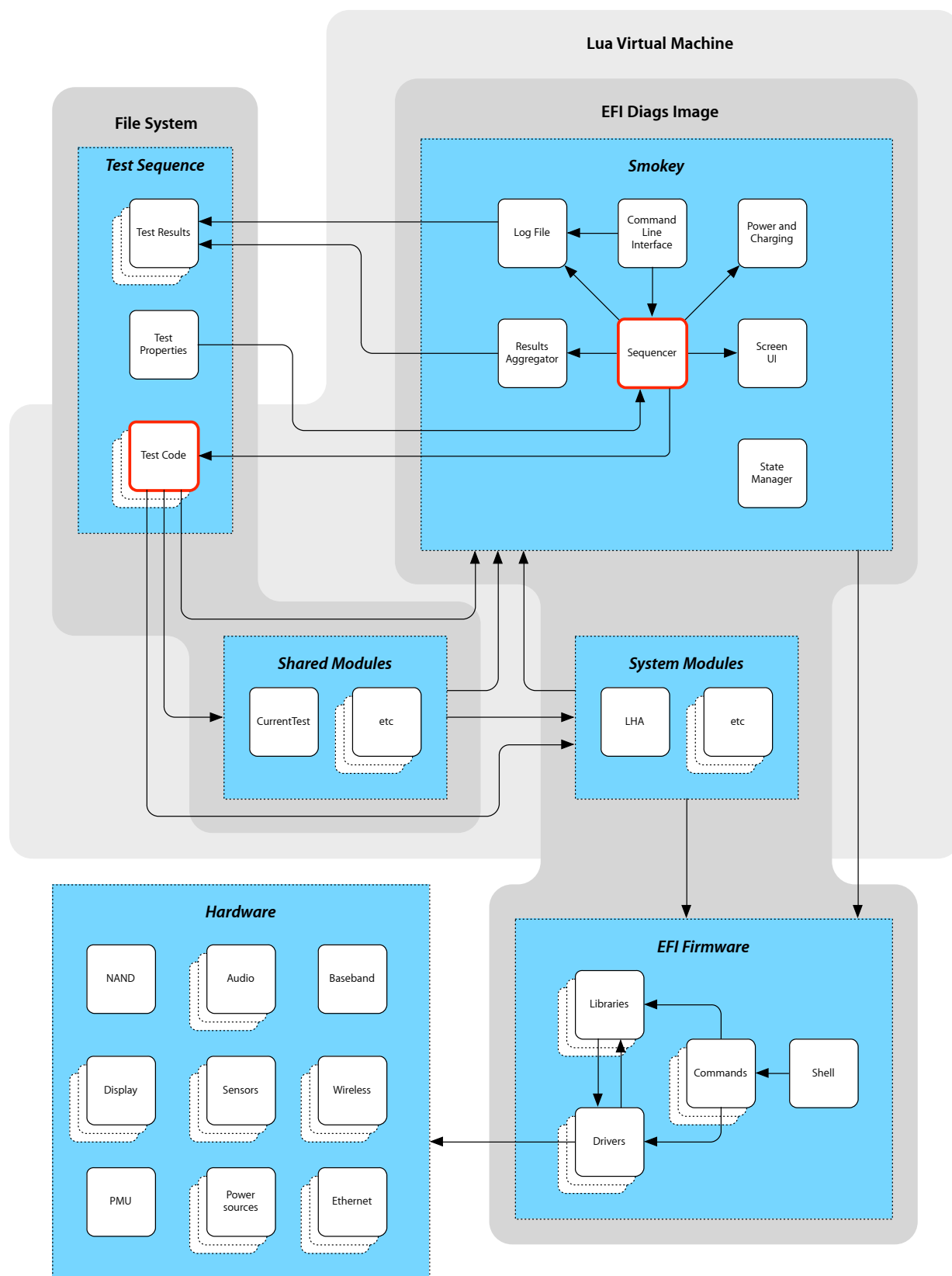


Figure 17: Smokey and EFI System Architecture

11 References

11.1 Useful Apple Links

11.1.1. Smokey Announcements Mailing List

smokey-announce@group.apple.com⁷⁰

11.1.2. Smokey Wiki

<https://ipodwiki.apple.com/wiki/Smokey>⁷⁰

11.2 Useful Lua Links

11.2.1. Official Lua Home Page

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

11.2.2. Lua Executable Binaries

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

11.2.3. Official Lua 5.2 Reference Manual

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

11.2.4. Lua Programmer's Guide

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

11.2.5. Lua Pitfalls and Gotchas

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

11.2.6. Lua Community Wiki

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

11.2.7. Tutorials and Learning Guides

<http://stackoverflow.com/a/8097810>⁷¹

⁷⁰Restricted access

⁷¹Refers to older Lua language version

11.2.8. Modules

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

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

<http://stackoverflow.com/questions/14942472/create-suite-of-interdependent-lua-files-without-affecting-the-global-namespace>