SUMMARY

Smokey now implements support for versioned modules via the new ModuleName(), SubmoduleName(), and ChildModuleName() functions. The rest of this document will explain how these functions work as well as how modules work in the Lua and Smokey environments.

MODULES

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 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 its only 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:

```
require "Statistics"
local SqRoot = require "NumericAlgorithms.SquareRoot.Newton"
```

Smokey sets up "require" to use a common location for modules shared between test sequences. Following the "require" path syntax and Lua conventions, the preconfigured search patterns are:

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

This allows complex modules to be loaded by implicitly opening a "init.lua" file which may subsequently ”require” any number of additional modules. Simple modules can be defined in a single Lua source file.

Smokey’s integration with “require” enables the “smokey” command to list the modules used by test sequences. The command line option is --dependencies and it takes no arguments.

```
[0000040C:0914F9E3] :-) smokey Wildfire --dependencies

... etc ...

Sequence syntax and sanity check passed

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

[0000040C:0914F9E3] :-)
```

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, as a lot of information floating around references features that are deprecated in later Lua versions.

(Footnote: Smokey is currently based on Lua 5.2.)

(Footnote: 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.)

MODULE CONVENTION AND HIERARCHY

Lua does not define nor enforce any policy on how modules are to be implemented or used. Consequently, modules are metaphorically small 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. We define the following terminology:

"Module" - External code that can be loaded by Lua.

"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 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.

It is usually straightforward to identify the location of a module file on the file system. 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 other 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.

In the context of 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. For example, consider the two following scenarios:

Discouraged module code layout:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\FFT\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\FFT\Support\FastTrig\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\FFT\Support\SimpleVector\init.lua

Test sequence code:

```
require "FFT.Support.FastTrig"
require "FFT.Support.SimpleVector"
```

Encouraged module code layout:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\FFT\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\FastTrig\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\SimpleVector\init.lua

Test sequence code:

```
require "FastTrig"
require "SimpleVector"
```

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 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. At the time of module loading, these functions provide a way to load other modules or record information for later use.

ModuleName(Level) - Get the name of the specified module at the specified level. The Level argument is optional; when omitted, this function returns the full name of the current module. When Level is positive, it returns the name of the module at the level relative to the top-level module, with 1 referring to the top-level module. When Level is negative, the name is relative to the current module, with -1 referring to the current module.

SubmoduleName(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 Name parameter, with a dot between.

ChildModuleName(Name) - Construct a submodule name based on the current submodule. The string returned is a concatenation of the current module name and the Name parameter, with a dot between. This is similar to, but simpler than, constructing a child module name using the arguments passed to the module via the "..." variable.

This allows these files:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\A\init.lua
```
require "A.B"
```
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\A\B\init.lua
```
require "A.B.C"
require "A.D.E"
```

to be rewritten as:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\A\init.lua
```
require(ChildModuleName "B")
```
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\A\B\init.lua
```
require(ChildModuleName "C")
require(SubmoduleName "D.E")
```

The functions above must be used within the context of a "require" call. 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 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.

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.

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.

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 overhead.

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 the following individual lines of Lua code:

```
require "A"          -- Not a versioned module
require "B"          -- Same as "B.Latest" in this case
require "B.Latest"   -- Bleeding edge
require "B.10B"      -- Brighton version
require "B.11A"      -- Innsbruck version
```

And here is the corresponding directory structure:

```
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\A\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\B\Latest\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\B\10B\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\B\11A\init.lua
```

Module "A" is not versioned, so the normal "require" behavior will successfully find "A\init.lua".

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.

VERSIONED MODULE LIFECYCLE

Shared modules in the Smokey environment will need to follow the lifecycle 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 tar-

gets. 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.

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.

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 "SimpleModule", the original "SimpleModule.lua" file would be moved and copied as shown below.

Before:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\SimpleModule.lua

After:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\SimpleModule\Latest\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\SimpleModule\11A\init.lua

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 below shows how files move when "11A" and "Latest" versions of "ComplexModule" are created.

Before:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\Support.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\Submodule\init.lua

After:

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\Latest\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\Latest\Support.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\Latest\Submodule\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\11A\init.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\11A\Support.lua
nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\ComplexModule\11A\Submodule\init.lua

For the examples above, it is assumed that "11A" was the most recent milestone and development will continue with the "Latest" version. When the transition occurs, Smokey test sequences that rely on old behavior must be updated to reference the "11A" version. Others can remain as-is.

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.

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.

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 combined with the following guidelines:

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

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.

The following code 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.

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\MyModule\Latest\Package.lua

```
return {}
```

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\MyModule\Latest\A.lua

```
local Package = require(SubmoduleName "Package")
Package.A = { FooText = "LatestFoo" }
function Package.A:Foo ()
      return self.FooText
end
```

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\MyModule\Latest\B.lua

```
local Package = require(SubmoduleName "Package")
Package.B = { BarText = "LatestBar" }
function Package.B:Bar ()
      return self.BarText
end
```

nandfs:\AppleInternal\Diags\Logs\Smokey\Shared\MyModule\Latest\init.lua

```
require(ChildModuleName "A")
require(ChildModuleName "B")
MyModule = require(SubmoduleName "Package")
return MyModule
```

The preceding code implements an module table and returns it to the user calling "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.

(Reference: http://lua-users.org/wiki/ModuleVersioning)

(Reference: http://lua-users.org/wiki/ModulesTutorial)

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