Console Framework User Guide

June 11th 2014

Jerry Johns
System Software Engineering - Firmware
`jjohns@apple.com`

Contents

1 Introduction

1.1 What is this about?

The console framework sits at the heart of EFI and manages all console traffic that gets generated in the system. This framework was recently over-hauled and introduces a wealth of functionality that is beneficial to both developers and users alike. This document aims to educate them on utilizing its many features effectively when interacting with Diags.

1.2 Audience

This document is intended for anyone who interacts with Diags both at the shell prompt as well as through the Smokey scripting framework.

1.3 Document Objective

This document will provide relevant technical details and concepts introduced in the new framework. It will also provide command usage guidelines and a set of workflows that are useful to most users working in EFI.

2 Framework Design

This section will focus on the design of the console framework and introduce key concepts needed to become productive with the new console commands.

2.1 Sources/Sinks

The console framework introduces a concept of sources and sinks that define granular boundaries for entities governing text production and consumption.

- **Source** — A unique, addressable entity that produces text
- **Sink** — A unique, addressable entity that consumes text. Examples of sinks include *smokey*, *serial* and *filelog*
- **Console Router** — Routes text from sources to sinks based on configurable route tables

Every driver in the system is setup by default to be a unique source.

2.2 Text Types

The framework standarizes the types of text that can be generated by a source. This allows for type-specific actions/formatting to be effected without heuristic inferrance from the text.

Type	Definition
Print	Normal operational text
Error	Critical failure
Warning	Not an error, indication something may be wrong with the system
Debug	Useful debugging output, provides more insight
Trace	Function entry/exit + custom breadcrumb markers

2.2.1. Debug Type

The debug text type is slightly different from the rest - each source can export a set of named debug bit masks that fall under this type category. This allows for each source to granularize the various kinds of debug spew it can generate.

As an example, a codec driver might export two debug bit masks - one named *transactions* for all register reads/writes to the chipset, and another called *volumes* that provides debug spew when setting/getting volumes on the part. These provide two different views into what's happening on the codec. When debugging a volume issue, you can start by enabling the *volumes* mask. If that doesn't get you far enough, you can then enable the *transactions* mask as well to get more information.

2.2.2. Global Debug

With the plethora of sources and their associcated debug bit masks in the system, it might be tough to quickly narrow down the source for which you'd like to enable verbosity. The global debug container attempts to solve that problem by allowing every source to optionally map a subset of their registered debug bit masks against this umbrella container. This allows consumers to quickly enable a minimal amount of trace in the system and get a more detailed view of underlying operations before narrowing down their investigation.

2.3 Routing

The console router exposes the ability for consumers to selectively add or delete routes from any source to any sink. Granularity is provided at the source's text type level. Hence, you can effect routing changes to a type of text generated by a specific source without changing routes for any of its other text types.

The router also provides a mask/unmask feature — this can be thought of as a temporary mute to disable any text generation or reception at a source/sink.

2.4 Tagged Text

Alongside text produced by a source, meta-data is appended to the text that travels alongside till the final destination. This includes information such as timestamp, text type, function name and source name.

2.5 Forensics

The framework houses a forensics buffer that automatically captures certain kinds of text without user intervention. This is meant to aid debugging failures by providing a slightly more detailed view of what transpired without needing to re-run the tests.

By default, all prints, errors, warnings and global debug prints are stored in the forensics buffer.

2.6 Boot Configuration

Most diags distributions are bundled with a few stock sinks. These include:

- **serial** — Sinks text that is destined for all console-based mediums. This includes UART, USB CDC and SWD
- **smokey** — Sink for the Smokey LUA sequencer
- **ramlog** — Sink for capturing text to RAM
- **filelog** — Sink for capturing text to a file

Routes for prints, errors and warnings have been added from all sources to the serial sink by default on release builds.

In addition to the above, debug builds have routes setup from every source’s debug masks (that map to the global debug container) routed to the *serial* sink.

3 Command Usage

This section outlines the various console commands and their usage.

3.1 "consolerouter" command

This command manages the router at the heart of the framework, allowing for effecting routing/-masking changes. It also controls dumping of the forensics buffer.

3.1.1. Source Listing

To list the various sources present in the system:

```
consolerouter --listsources
```

This list is divided into two sections. The first set of sources (labelled *legacy sources*) denotes sources that have yet to register against the new framework. This means that they won't have any debug masks exported. It also means that they are named after the EFI driver.

```
:-) consolerouter --listsources
>>> Legacy sources:

DiagShell
AppleBarcode
AppleDiagsSecurityManager
DisplayConsole
TestNvram
ControlBitsIPX
MCATransport
MCA
AppleSimpleUsbFs
DfuImageManagement
```

The second set (labelled *registered sources*) have been explicitly registered with the console framework. They can optionally have debug masks registered against the console framework.

It is possible to have both registered and legacy sources for the same image.

```
>>> Registered sources:

gyro:
    image: AP3GDL
    debug-bits:
        reg-ops: Register operations (read and write)
        data: Gyro data

i2clib:
    image: ALS, PmuCore, Codec
    debug-bits:
        type : Information on packet types + sizes
        pktdata : Packet data
```

The *debug-bits* field list the various debug masks available for a particular source.

The *image* field lists the driver image that this source is registered against. This is useful in the case of libraries (like *i2clib* above) that gets used by multiple images. In the above case, the *ALS*, *PMUCore* and *Codec* drivers have registered the *i2clib* source.

3.1.2. Sink Listing

To list the various console sinks present in the system:

```
consolerouter --listsinks
```

The sink listing looks a lot simpler and provides a name and description for each sink registered in the system:

```
:-) consolerouter --listsinks
serial:
    description: Console sink that routes/formats text for serial output over UART/USB
file:
    description: Console sink that stores console output to a file
ramlog:
    description: Sink that can be used to specifically log specific
    text to RAM
OK
```

3.1.3. Routing

To route text:

```
consolerouter [--add | --rm] --src source_specifier --dest sink_specifier
```

The *source_specifier* has to be provided in the following form:

source.text-type.debug-mask

- **source** — Name of text generating entity (e.g *PmuCore*, *i2clib* or *system*)
- **text-type** — Type of text. The acceptable types are:
  - error
  - print
  - warn
  - trace
  - debug
- **debug-mask** — Debug mask. This field is only applicable when specifying *debug* for text-type

Here are some examples of source specifiers:

`PmuCore.error`	All errors from the *PmuCore* source
`i2clib.debug.pktdata`	All debug text of mask *pktdata* from the *i2clib* source

Wildcards are also supported. The * character selects all within the specifier sub-type.

`PmuCore.*`	All text types from the *PmuCore* source
`*.error`	Errors from all sources
`i2clib.debug.*`	All debug prints from the *i2clib* source
`*`	All text from all sources

To facilitate convenience of entry, braces and commas can be used to allow for cartesian-product expansion:

`{PmuCore,i2clib}.*`	All text types from the *PmuCore* and *i2clib* sources
`i2clib.{error,warn}`	All errors and warnings from *i2clib*
`{PmuCore,system}.{error,warn}`	All errors and warnings from *PmuCore* and *system*
`i2clib,PmuCore.{error,warn}`	All prints from *i2clib* + errors, warnings from *PmuCore*

The *sink-specifier* also support wildcard specifiers, commas and brace expansion:

`serial`	*serial* sink
`*`	All sinks
`serial,smokey`	*smokey* and *serial* sinks
`s{erial,mokey}`	*smokey* and *serial* sinks

3.1.4. Masking

Masking/Unmasking allows the user to temporarily mute/un-mute both sources and sinks. For sources, you can effect masking changes at the text type level. For sinks, you can effect masking changes at the sink level.

```
consolerouter [--mask | --unmask] [--src source_specifier] [--dest sink_specifier]
```

E.g:

`--mask --src system`	Masks all routes emnating from the *system* source
`--mask --src PmuCore.error`	Masks routes of error type emnating from *PmuCore*
`--mask --src PmuCore.{error,warn}`	Masks routes of error + warn type emnating from *PmuCore*
`--mask --desk smokey`	Masks all routes going to the *smokey* sink
`--mask --dest serial`	Masks all routes going to the *serial* sink

The masking operation has no effect on the routes themselves.

3.1.5. Forensics Buffer

This command provides the ability to dump out the forensics buffer. You can provide an optional argument to limit the spew to a specific number of commands executed at the shell.

```
consolerouter --dump [num-commands]
```

Dumping the entire buffer:

```
:-) consolerouter --dump
Dumping forensics buffer...
|| Console router buffer allocated @ 0x83E17C018, size = 262144 bytes
|| [system.error] Installed soc embedded gpio
|| [system.error] !!!!!!!!!!!!!!!!!!!!!UDR: 0x5A6E4C70A3823017, BaseVdd: 600
|| [system.error] Came to install McaAudioTransportExsoc PMGR installed
|| [system.debug] [DPM:InitializeDomainManagement] Device: 1
|| [system.debug] [DPM:InitializeDomainManagement] Device: 2
|| [system.debug] [DPM:InitializeDomainManagement] Device: 3
|| [system.debug] [DPM:InitializeDomainManagement] Device: 5
|| [system.debug] [DPM:InitializeDomainManagement] Device: 6
|| [system.debug] [DPM:InitializeDomainManagement] Device: 7
|| [system.debug] [DPM:InitializeDomainManagement] Device: 26
|| [system.debug] [DPM:InitializeDomainManagement] Marking 26
```

Dumping just the last command's output:

```
:-) sensor -s als --init
Turning off power to 'als' sensor...
Turning on power to 'als' sensor...
Resetting 'als' sensor...
OK
:-) consolerouter --dump 1
Dumping forensics buffer...
Dumping the last 1 commands' outputs...
|| :-) sensor -s als --init
|| Turning off power to 'als' sensor...
|| [system.debug] Power off sequence for Bus:2, Addr:0x29

|| [system.debug] Write Control byte

|| [system.debug] Disable interrupt

|| Turning on power to 'als' sensor...
|| [system.error] Locating ALS device...
|| [system.error] Locate ALS:Dev ID reg val:0x92
|| [system.debug] Gain set to 1

|| Resetting 'als' sensor...
|| OK
|| :-) consolerouter --dump 1
|| Dumping forensics buffer...
|| Dumping the last 1 commands' outputs...
Done dumping!
OK
:-)
```

3.2 “consoleformat” command

The “consoleformat” command handles setting/getting formatting options for console-based mediums (like *serial*, *smokey* and *file* sinks).

3.2.1. Sink Listing

To list all serial-based sinks:

```
consoleformat --listsinks
```

Output:

```
:-) consoleformat --listsinks
file:
    display-options:
serial:
    display-options: color
OK
```

The currently enabled formatting options for each sink is provided in the listing as well.

3.2.2. Formatting Options

To set formatting:

```
consoleformat --sink sink-name [--en|--dis] --options [func,source,ts,color]
```

Option Description:

`func`	Function name
`source`	Source name (e.g *i2clib* or *PmuCore*)
`ts`	Timestamp
`color`	Text color

Multiple options can be passed in together in a comma delimited list:

```
consoleformat --sink serial [--en|--dis] --options func,source
```

Formatting options can vary from sink to sink.

3.2.3. Formatted Output

When formatting options are enabled, text is formatted in a very specific manner:

no formatting	`Hello World!`

+ source	`[system] Hello World!`
	`[PmuCore:i2clib] Hello World!`
	`[PmuCore:i2clib.debug.pktdata] Hello World!`

+ function name	`[PrintHelloWorld] Hello World!`

+ timestamp	`<1843.890109s (+2.205379)> Hello World!`
	`<1843.890111s (+0.000002)> Hello World!`

When *source* is enabled, it can be printed in a few different ways (see above). Most prints appear in the first fashion. When printing from a library (like *i2clib*), the console provides the driver name as well (i.e *PmuCore:i2clib*). Finally for debug prints, the debug mask is also displayed to the user.

All three formatting options enabled (*source*, *func* and *ts*):

```
[system :: PrintHelloWorld <1843.890109s (+2.205379)>] Hello World!
[PmuCore:i2clib :: PrintHelloWorld <1843.890109s (+2.205379)>] Hello World!
[PmuCore:i2clib.debug.pktdata :: PrintHelloWorld <1843.890109s (+2.205379)>] Hello World!
```

When color is enabled, ANSI color codes are injected into the text stream along with a custom prefix string:

errors	`ERROR: Did not initialize!`
warnings	`WARNING: Potential clock violation!`
debug	`[system.info] Potential clock violation!`
trace	`TRACE: Entered function!`

3.3 "ramlog" command

The *ramlog* sink allows users to route desired prints to RAM without incurring the overhead of printing to serial. This allows for capturing prints that otherwise would affect behaviour if routed to serial.

Note that a route has to be setup to this sink for it to capture text.

3.3.1. Initialization/Teardown

The following options control initialization and teardown of the ramlog sink:

```
ramlog [--on size_in_megabytes] [--off]
```

3.3.2. Pause/Un-pause

The ramlog can be momentarily paused/un-paused as the user sees fit - the contents in RAM remain unaltered and no text is sent to the sink during this window

```
ramlog [--pause] [--unpause]
```

3.3.3. Clear

The contents in the ramlog can be cleared without affecting logging functionality.

```
ramlog --clear
```

3.3.4. Over-write

By default, the ramlogger will continue to capture text when the buffer gets full by over-writing the most stale entries.

To stop the ramlogger from logging any further once it is full:

```
ramlog --on [--stoponfull]
```

The --*stoponfull* option should only be used with the --*on* option.

3.4 "filelog" command

The *file* sink allows users to route desired prints to file.

Note that a route has to be setup to this sink for it to capture text.

3.4.1. Initialization/Teardown

The following options control initialization and teardown of the filelog sink:

```
filelog [--on path_to_file] [--off]
```

Due to performance considerations, text is buffered inside the file logger and flushed to file when a certain number of bytes have accumulated. When the filelogger is turned off, all remaining bytes in the buffer are flushed out.

3.5 "debug" command

This command enables/disables global debug prints in the system. Specifically, it adds/removes global debug routes from all sources to the *serial* sink. Hence, this command has no effect on routes to other sinks like *smokey* or *ramlog*.

```
debug [--on | --off]
```

4 Common Workflows

To help users get the most out of the new features in the console framework, a few real-world workflows have been provided in this section that best leverage the new feature-set.

4.1 “What just happened?”

Scenario:

A command just failed and you’d like to know a bit more about what transpired leading up to the event.

```
:-) device -k GasGauge -e log_on fast
enabling auto-calibration mode
ERROR: Method "log_on" returned status Device Error
device returned Device Error error
```

Solution:

You can use the forensics buffer to dump out additional debug information that may have been logged in the background while that command was running:

Just dump out the contents of forensics buffer for the last command:

```
:-) consolerouter --dump 1
Dumping forensics buffer...
Dumping the last 1 commands' outputs...
|| [001A59C0:2403C13A] :-) device -k GasGauge -e log_on fast
|| [gasgauge.info] Waiting for CCA to finish...
|| [gasgauge.trans] Reading control: data-type = 0000, length = 2, needs_unseal = no
|| [gasgauge.info] Unsealing...
|| [gasgauge.info] Status Reg = 0x0B 0x60
|| [gasgauge.info] IsSealed = SEALED
|| [gasgauge.info] Status Reg = 0x0B 0x40
|| [gasgauge.info] IsSealed = SEALED
|| [gasgauge.info] Unseal failed. Retrying...
|| [gasgauge.info] Status Reg = 0x0B 0x40
|| [gasgauge.info] IsSealed = SEALED
|| [gasgauge.info] Unseal failed. Retrying...
|| [gasgauge.info] Status Reg = 0x0B 0x40
|| [gasgauge.info] IsSealed = SEALED
|| [gasgauge.info] Unseal FAIL
|| enabling auto-calibration mode
|| ERROR: Method "log_on" returned status Device Error
|| device returned Device Error error
|| [001A59C0:2403C13A] :-) consolerouter -t 1
|| Dumping forensics buffer...
|| Dumping the last 1 commands' outputs...
Done dumping!
OK
```

In this case, the gas gauge driver failed to start logging because the unseal operation failed. You also know exactly how many attempts were made, and the register contents of the status register during each of those attempts. You might want to then enable more verbosity within the gas gauge driver to further debug this issue.

Perhaps at this point, you’d like to get a sense for which functions/sources generated this error. You can do this by re-unrolling the forensics buffer with additional formatting options put in place:

```
:-) consoleformat --sink serial --en --options func,source
Enabling function-name option...
[_BraceExprCallback] Enabling source option...
[TestConsole :: EblConsoleFormatCmd] OK
:-) consolerouter --dump 1
[TestConsole :: EblConsoleRouterCmd] Dumping forensics buffer...
[TestConsole :: EblConsoleRouterCmd] Dumping the last 1 commands' outputs...
|| [001A59C0:2403C13A] :-) device -k GasGauge -e log_on fast
|| [gasgauge.info :: InternalWaitOnCCAClear] Waiting for CCA to finish...
|| [gasgauge.trans :: Read] Reading control: data-type = 0000, length = 2, needs_unseal = no
|| [gasgauge.info :: DoUnseal] Unsealing...
|| [gasgauge.info :: IsSealed] Status Reg = 0x0B 0x60
|| [gasgauge.info :: IsSealed] IsSealed = SEALED
|| [gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
|| [gasgauge.info :: IsSealed] IsSealed = SEALED
|| [gasgauge.info :: DoUnseal] Unseal failed. Retrying...
|| [gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
|| [gasgauge.info :: IsSealed] IsSealed = SEALED
|| [gasgauge.info :: DoUnseal] Unseal failed. Retrying...
|| [gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
|| [gasgauge.info :: IsSealed] IsSealed = SEALED
|| [gasgauge.info :: DoUnseal] Unseal FAIL
| [gasgauge.info :: DoSeal] Already Sealed
|| [TestDevice :: EblDevice] ERROR: Method "log_on" returned status Device Error
|| device returned Device Error error
```

4.2 “What’s happening in this command?”

Scenario:

You want to get more insight into what’s happening in a particular command, or driver.

```
:-) device -k GasGauge -e log_on fast
enabling auto-calibration mode
ERROR: Method "log_on" returned status Device Error
device returned Device Error error
```

Solution:

Building on the same example we had used for the previous scenario, let’s assume in this case that we gleaned very little from the forensics buffer.

4.2.1. Deduce Source

To delve further into the above failure, we need to know which debug sources to enable. You might know this apriori: in the above case, it would be *gasgauge* and *uartswif* console sources.

If you don’t know them ahead of time, run:

```
debug --on
```

This will enable a sparse level of debug on most sources in the system.

In addition, enable source name in the format options:

```
consoleformat --sink serial --en --options source
```

Now, rerun the same command:

```
:-) device -k GasGauge -e log_on fast
[gasgauge.info :: InternalWaitOnCCAClear] Waiting for CCA to finish...
[gasgauge.trans :: Read] Reading control: data-type = 0000, length = 2, needs_unseal = no
[gasgauge.info :: DoUnseal] Unsealing...
[gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
[gasgauge.info :: IsSealed] IsSealed = SEALED
[gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
[gasgauge.info :: IsSealed] IsSealed = SEALED
[gasgauge.info :: DoUnseal] Unseal failed. Retrying...
[gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
[gasgauge.info :: IsSealed] IsSealed = SEALED
[gasgauge.info :: DoUnseal] Unseal failed. Retrying...
[gasgauge.info :: IsSealed] Status Reg = 0x0B 0x40
[gasgauge.info :: IsSealed] IsSealed = SEALED
[gasgauge.info :: DoUnseal] Unseal FAIL
[TestDevice :: EblDevice] ERROR: Method "log_on" returned status Device Error
device returned Device Error error
```

Aha! We now know some additional information (looks like unseal failed) as well as the source of the offending error (*gasgauge*)

4.2.2. Enable Debug Masks

Let's see if we can get some additional information out of this source. Start by checking to see if this particular source has debug flags registered:

```
:-) consolerouter --listsources
...
...
...
gasgauge:
    image: BQ27541
    debug-bits:
        info : General informational prints
        trans : Information on command/data reads/writes
        data : Actual byte data over UART SWIF
uartswif:
    macho: UartSwif
    bit-masks:
        data : Raw data
```

Looks like the *gasgauge* source has three masks defined. In addition, we know that this driver consumes services from another source, the *uartswif* source. We can first enable extra debugging in the *gasgauge* source, and if that doesn't suffice, enable debugging on the *uartswif* source.

```
:-) consolerouter --add --src gasgauge.debug --dest serial
Routing DEBUG (0x00000007): 'gasgauge' -> 'serial'
OK
:-) device -k GasGauge -e log_on fast
[gasgauge.info] Waiting for CCA to finish...
[gasgauge.trans] Reading control: data-type = 0000, length = 2, needs_unseal = no
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 00
[gasgauge.data] <= HDQ_WRITE: Cmd = 01, Data = 00
[gasgauge.data] => HDQ_READ: Cmd = 00, Data = 0B
[gasgauge.data] => HDQ_READ: Cmd = 01, Data = 40
[gasgauge.info] Unsealing...
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 00
[gasgauge.data] <= HDQ_WRITE: Cmd = 01, Data = 00
[gasgauge.data] => HDQ_READ: Cmd = 00, Data = 0B
[gasgauge.data] => HDQ_READ: Cmd = 01, Data = 40
[gasgauge.info] Status Reg = 0x0B 0x40
[gasgauge.info] IsSealed = SEALED
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 36
[gasgauge.data] <= HDQ_WRITE: Cmd = 01, Data = 12
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 44
[gasgauge.data] <= HDQ_WRITE: Cmd = 01, Data = 65
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 00
[gasgauge.data] <= HDQ_WRITE: Cmd = 01, Data = 00
[gasgauge.data] => HDQ_READ: Cmd = 00, Data = 0B
[gasgauge.data] => HDQ_READ: Cmd = 01, Data = 40
[gasgauge.info] Status Reg = 0x0B 0x40
[gasgauge.info] IsSealed = SEALED
[gasgauge.info] Unseal failed. Retrying...
...
...
...
```

We find out all the registers that the driver is writing/reading to/from when enabling logging.

Perhaps this isn't enough information - let's enable the *uartswif* data as well:

```
:-) consolerouter -a -s uartswif.debug --dest serial
Routing debug (0x00000001): 'uartswif' -> 'serial'
OK
:-) device -k GasGauge -e log_on fast
[gasgauge.info] Waiting for CCA to finish...
[gasgauge.trans] Reading control: data-type = 0000, length = 2, needs_unseal = no
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 00
[uartswif.data] Sending break...
[uartswif.data] Writing 2 bytes: 0x80 0x00
[uartswif.data] Read 2 bytes: 0x80 0x00
[gasgauge.data] <= HDQ_WRITE: Cmd = 01, Data = 00
[uartswif.data] Sending break...
[uartswif.data] Writing 2 bytes: 0x81 0x00
[uartswif.data] Read 2 bytes: 0x81 0x00
[gasgauge.data] => HDQ_READ: Cmd = 00, Data =
[uartswif.data] Sending break...
[uartswif.data] Writing 1 bytes: 0x00 Read 1 bytes: 0x00
[uartswif.data] Read 1 bytes: 0x0B
[gasgauge.data] 0B
[gasgauge.data] <= HDQ_WRITE: Cmd = 00, Data = 00
[uartswif.data] Sending break...
[uartswif.data] Writing 2 bytes: 0x80 0x00
[uartswif.data] Read 2 bytes: 0x80 0x00
...
...
...
```

By enabling debug from the various sources that make up a command, we're able to unlock successively richer views of information from the underlying code.

4.3 “Enabling prints causes the problem to go away...”

Scenario:

A failing command suddenly passes when I enable debug prints.

Solution:

Route those debug prints to the *ramlog* sink so that they won’t affect timing.

Building on top of the scenario above, we can route all the additional debug prints from the *gasgauge* and *uartswif* sources to the *ramlog* sink and view them after.

```
Initial Output          :-) device -k GasGauge -e log_on
                        enabling auto-calibration mode
                        ERROR: Method "log_on" returned status Device Error
                        device returned Device Error error
Setup Routes            :-) consolerouter --add --src gasgauge.debug,uartswif.debug --dest ramlog
                        Routing DEBUG (0x00000007): 'gasgauge' -> 'ramlog'
                        Routing DEBUG (0x00000001): 'uartswif' -> 'ramlog'
                        OK
                        OK
Enable Ramlog           :-) ramlog --on 1
                        Initializing ram logger to 1Mbytes...
                        Turning on logging...
Re-run Command          :-) device -k GasGauge -e log_on
                        enabling auto-calibration mode
                        ERROR: Method "log_on" returned status Device Error
                        device returned Device Error error
Dump Ramlog             :-) ramlog --dump
                        || [gasgauge.info] Waiting for CCA to finish...
                        || [gasgauge.type] Reading control data type 00 of length 2
                        || [uartswif.data] Sending break...
                        || [uartswif.data] Writing 2 bytes: 0x80 0x00
                        || [uartswif.data] Read 2 bytes: 0x80 0x00
                        || [uartswif.data] Sending break...
                        || [uartswif.data] Writing 2 bytes: 0x81 0x00
                        || [uartswif.data] Read 2 bytes: 0x81 0x00
                        || [uartswif.data] Sending break...
                        || [uartswif.data] Writing 1 bytes: 0x00 Read 1 bytes: 0x00
                        || [uartswif.data] Read 1 bytes: 0x0D
                        || [uartswif.data] Sending break...
Enable Timestamps       :-) consoleformat --dest serial --en --options ts
                        Enabling timestamps option...
                        <9293.526033s (+9293.526033)> OK
Re-dump Ramlog          :-) ramlog --dump
                        || [gasgauge.info] <6.359588s (+2.385171)> Waiting for CCA to finish...
                        || [gasgauge.type] <6.359598s (+0.000010)> Reading control data type 00, length 2
                        || [uartswif.data] <6.359606s (+0.000008)> Sending break...
                        || [uartswif.data] <6.359856s (+0.000250)> Writing 2 bytes: 0x80 0x00
                        || [uartswif.data] <6.363186s (+0.003330)> Read 2 bytes: 0x80 0x00
                        || [uartswif.data] <6.363209s (+0.000023)> Sending break...
                        || [uartswif.data] <6.363458s (+0.000249)> Writing 2 bytes: 0x81 0x00
                        || [uartswif.data] <6.366789s (+0.003331)> Read 2 bytes: 0x81 0x00
                        || [uartswif.data] <6.366812s (+0.000023)> Sending break...
Reset Routes to Default :-) consolerouter --reset
                        <11932.553261s (+2635.923372)> Resetting routes and masks back to default...
                        <11932.561484s (+0.008223)> OK
```

4.4 “What’s happening in passthrough?”

Scenario:

When doing passthrough (e.g wifi), you want to sniff and inspect the characters that were sent to/from the dock UART.

Solution:

Route the correct *serial* source's tx/rx debug masks to ramlog and run passthrough.

```
        Setup Route  :-) consolerouter --add --src serial0 --dest ramlog
                     Routing all prints: 'serial0' -> 'ramlog'
                     OK
      Enable Ramlog  :-) ramlog --on 1
                     Initializing ram logger to 1Mbytes...
                     Turning on logging...
  Enter Passthrough  :-) ramlog --wipe; wifi --passthrough
                     Wiping out contents in buffer...
                     OK
                     Entering pass-through mode (type "EXIT" to exit)...
   Exit Passthrough  EXIT
                     :-)
        Dump Ramlog  :-) ramlog --dump
                     || [serial0.tx] 0x4F 0x4B 0x0A 0x45 0x6E 0x74 0x65 0x72 0x69 0x6E 0x67 0x20 0x70
                            0x61 0x73 0x73 0x2D 0x74 0x68 0x72 0x6F 0x75 0x67 0x68
                     || [serial0.rx] 0x06 0x01 0x00 0x00 0x00 0x01 0x00 0x00 0x02 0x00 0x00 0x00 0x00
                     || [serial0.tx] 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x04 0x00 0x00 0x00 0x00
```

4.5 "UART is so slow!"

Scenario:

Some commands generate lots of UART output that the user does not care about while causing a slow down in the execution of commands

Solution:

Mask sources/sinks appropriately to minimize generated spew without affecting routes.

4.5.1. Masking Sources

If you know the particular source that is generating a lot of text output, you can mask off just that one:

```
consolerouter --mask --src PmuCore
```

If you want to indiscriminately mask off all sources:

```
consolerouter --mask --src *
```

If you're masking sources, it's usually recommended to mask off everything except errors and warnings. This way, you still get notified if something goes wrong:

```
consolerouter --mask --src *.{print,debug,trace}
```

NOTE: The above techniques only work if you don't have multiple sinks consuming text output (like *serial* and *smokey*). In that case, you don't want to mask off sources since no sink will get that text, including an entity like *smokey*.

NOTE: Masking operations don't apply to the shell itself. If a command errors out, the shell will print the status code.

4.5.2. Masking Sinks

You can use this technique to completely mask off a sink like *serial* without affecting other sinks like *smokey*. In this example, *smokey* still gets all of its text traffic while the serial port gets none. This is extremely useful when running Smokey scripts without needing to inspect the output coming over UART:

Run Smokey

```
 :-) smokey EdpTest --run
Smokey (local build commit ee9f8c9@) 2014/05/04 08:45:50

SrNm: DLXL100ZFMNM
MLB#: DLX326402CTFCCP1K
CFG#: J72-DVT/*/*/26251/MAIN-D-2-D-REL
ECID: 000003D44F169EE3

Control File:   nandfs:\AppleInternal\Diags\Logs\Smokey\EdpTest\J72\Main.plist
Script File:    nandfs:\AppleInternal\Diags\Logs\Smokey\EdpTest\J72\Main.lua
Log File:       nandfs:\AppleInternal\Diags\Logs\Smokey\EdpTest\Smokey.log
Results File:   nandfs:\AppleInternal\Diags\Logs\Smokey\EdpTest\PDCA.plist
Signature File: nandfs:\AppleInternal\Diags\Logs\Smokey\EdpTest\Earthbound.sig
Control Bit:    none

SequenceName:        EdpTest
SequenceVersion:     1
BehaviorOnFail:      KeepGoing
ResultsBehavior:     Bookend
LogBehavior:         Full
BrickRequired:       None
LogCollectorControl: None
ControlBitAccess:    Default

Sequence syntax and sanity check passed

Writing default results
Skipping control bit write
Writing PDCA plist file

Continuing without display
Continuing without charger
Device ready

Sequence execution...

Day/Time      Node
------------- ----
[27 02:40:29] N001 Repeating 1x
[27 02:40:29] ....  [1] Periodic tasks
[27 02:40:29] N002  [1] Repeating 1x
[27 02:40:29] ....      [1] Action "BerRead"
[27 02:40:29] ....          device -k Display@Internal -e ber read
-------------------------------------------------------------------------------
:-) device -k Display@Internal -e ber read
ERROR: Did not find matching key: 'Display@Internal'
device returned Not Found error
-------------------------------------------------------------------------------
[27 02:40:29] ....          Exit code = 0x8000000E
[27 02:40:29] ....          Shell command 1 failed
[27 02:40:29] ....          ActionToExecute failed
[27 02:40:29] ....      [1] Ignoring failure
[27 02:40:30] N001 Sequence done

Failed

Writing final results
Skipping control bit write
Writing PDCA plist file

All errors:
    SmokeyResults: failed action "BerRead" at node 2 iteration 1/1:
    EfiCommand: command 1 had errors: device -k Display@Internal -e ber read
```

Mask Serial

Re-run Smokey

```
smokey returned Device Error error
:-) consolerouter -m -d serial
Masking sink 'serial'
OK
:-) smokey EdpTest --run --clean
device returned Not Found error
smokey returned Device Error error
```