BrightScript Profiler

BrightScript Profiler is a tool created for collecting and analyzing channel metrics that can be used to determine where performance improvements and efficiencies can be made in the channel.

Table of Contents

- Usage
- Manifest entries
- Running the profiler on a channel
- Collecting the data
- Processing the data
- Understanding the data
- Using this data

The BrightScript profiler provides the following metrics for a channel:

- CPU Usage: Determine where BrightScript code execution is happening
- Wall-Clock time: Determine where the most time (execution *and* waiting) is being spent when a channel is running
- Function call counts
- Memory usage, including memory leak detection – *Available since firmware version 9*

Each of the above metrics can be used to diagnose problems and provide guidance to the channel developer to improve channel performance.

Usage

The workflow of the BrightScript Profiler is as below:

1. Add at least the required manifest entries to the channel to run the profiler
2. Run and then Exit the channel to generate data and metrics
3. Save the profiling data to the device or stream it to the machine (PC) you are using for development
4. Analyze the profiling data as necessary

Manifest entries

Below is the list of manifest keys used by the profiler:

Manifest Key	Value Type	Legal Values	Default Value	Required	Description
`bsprof_data_dest`	enum	`local , network`	local	Yes	If this entry value is `local`, profiling data is collected on the device and can be downloaded from the Application Installer after the channel terminates. This is the default. If this entry value is `network`, the profiling data is sent over the network rather than being stored on the device. See the section on Retrieving Profiling Data for details.
`bsprof_enable`	boolean	`0 , 1`	0	Yes	Turns on BrightScript Profiling when the channel is running. This is the master flag and must be set to 1 for any other profiling options to take effect.
`bsprof_enable_mem`	boolean	`0 , 1`	0	If using memory profiling	Turns on memory profiling. Only has effect if `bsprof_enable` is set to 1 `(bsprof_enable=1)`. If this is enabled, `bsprof_sample_ratio` is forced to be 1.0.

---

bsprof_pause_on_start	boolean	0 , 1	0	No	Immediately after launching the channel, profiling is paused until manually resumed with the bsprof-resume command on the port 8080 debug console. Only has effect if bsprof_enable=1 This is useful for profiling isolated parts of a channel's UI or operations, rather than profiling the entire startup sequence of the channel.
bsprof_sample_ratio	float	0.001 to 1.0	1.0	Yes	Sets how often profiling samples are taken, while the channel is running. Only has effect if bsprof_enable=1 A sample ratio of 1.0 causes every BrightScript statement to be measured and integrated into the profiling data for the channel. Unfortunately, a sample ratio of 1.0 may cause some complex channels to run very slowly when profiling is enabled, making them difficult to test. Choosing a lower sample ratio for such channels can make them more usable while profiling is enabled. Although, a higher ratio yields more accurate data, it is therefore recommended that the ratio must be set as 1.0 whenever possible. If bsprof_enable_mem=1, this value is forced to be 1.0

The bs_prof_sample_ratio can be adjusted from 0.001 to 1.0. A sample ratio of 1.0 is the default and measures every BrightScript statement. A sample ratio of 1.0 has some performance impact, but in most cases, it doesn't affect the usability of the channel and provides the most accurate data. However, if the channel is overly sluggish with a ratio of 1.0, reduce the ratio to lessen the profiler's overhead.

Running the profiler on a channel

To initiate the memory profiler, sideload, run, and then exit the channel. The profiling data is complete only after the channel exits. Note that memory data can be streamed to a network. The advantage of streaming the data to a network is that it consumes significantly less memory on the device while the channel is running.

Pausing and resuming profiling

Channel profiling can be paused and resumed at any time. Use the following commands on port 8080 to either pause or resume the memory profiler:

```
bsprof-pause

bsprof-resume
```

If the profiler is paused, very little data is written regardless of the data destination. This allows the profiling data (generally, the data relevant to specific parts of a channel's UI or other operation) to be collected and analyzed later. These two commands are particularly useful when combined with the bsprof_pause_on_start manifest entry.

Manifest entry:

```
bsprof_pause_on_start=1
```

For example, if starting video playback is slow or seems to cause memory leaks, the bsprof_pause_on_start=1 entry can be set in the channel's manifest. After the channel is launched, but prior to video playback, execute the bsprof-resume command on port 8080 to begin collecting profiling data. After performing the UI operations to be profiled, execute the bsprof-pause command to suspend the storing operation of the profiling data. Then, exit the channel to make the profiling data available for analysis. In this scenario, the profiling data will pertain specifically to the operations performed between bsprof-resume and bsprof-pause.

Port 8080 Commands

These profiling commands exist on port 8080 (Roku OS Versions 9 and later):

Command	Purpose
`bsprof-status`	Get the status of BrightScript profiling
`bsprof-pause`	Pause the generation of profiling data
`bsprof-resume`	Resume the generation of profiling data

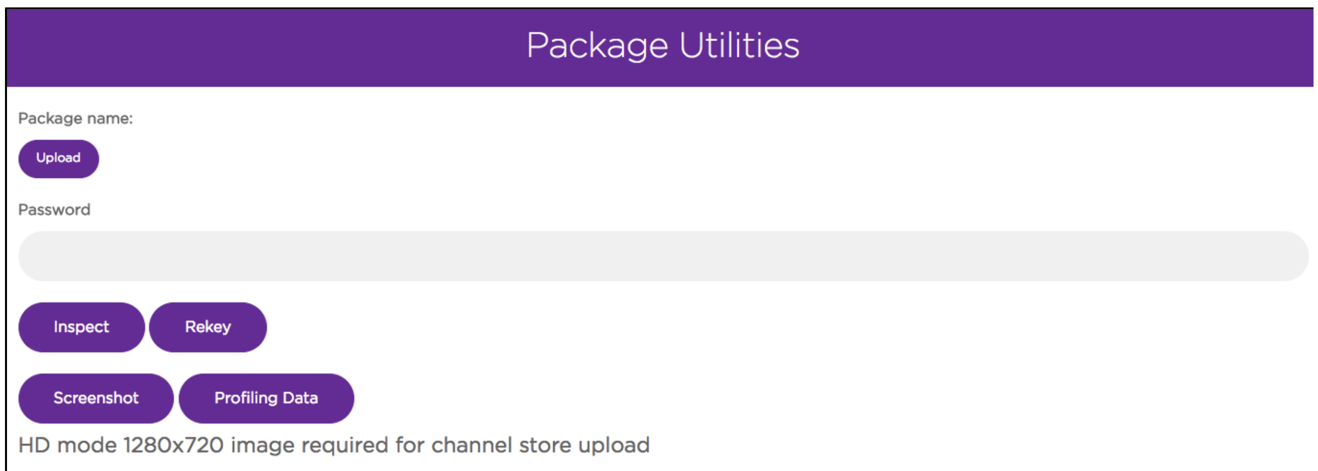
Collecting the data

The channel's manifest entry `bsprof_data_dest` determines how the profiling data is retrieved from the device. The data can be stored locally on the device and downloaded after the channel finishes running and exits, or it can be streamed over a network connection while the channel is running.

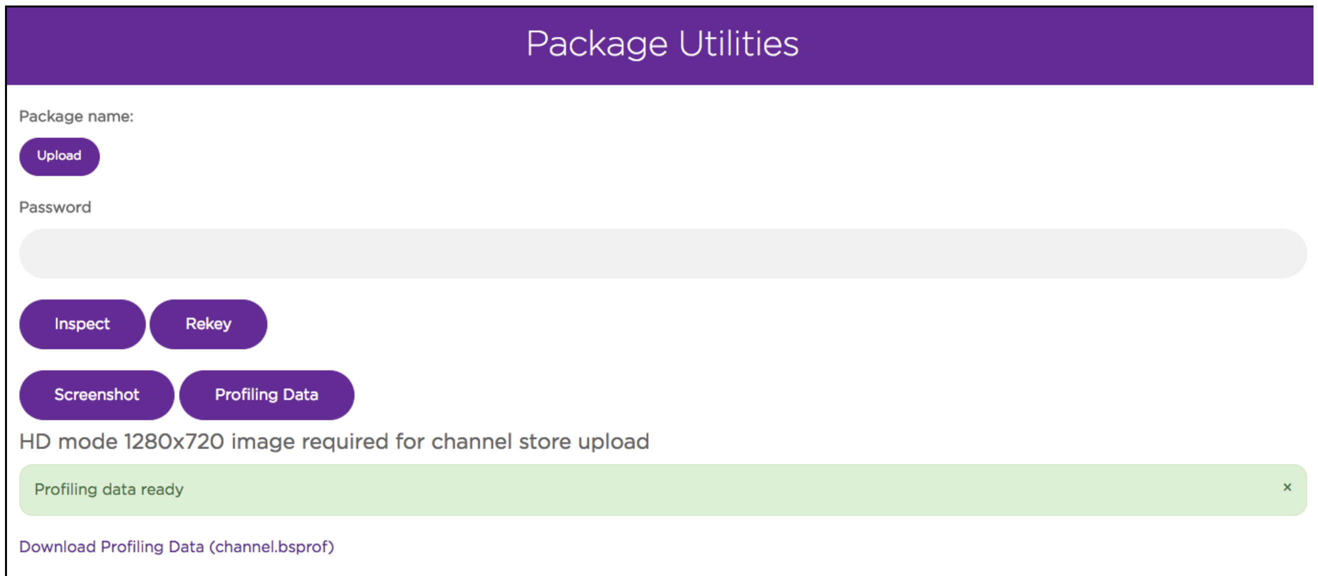
Data Destination: Local

Local data storage is the default storage for profiling, though it can be explicitly selected by adding `bsprof_data_dest=local` to the channel's manifest. When using this destination, the data becomes available on the device's Application Installer after the channel exits:

1. Launch the channel and run through the test cases. Once the channel finishes running and exits, open **Roku device's Developer Settings** and click on **Utilities.**



2. Click **Profiling Data** to generate a `.bsprof` file and a link to download the data from your Roku device.



> The `.bsprof` format is unique to Roku to ensure the format is as efficient and small as possible and easy to generate even on low-end Roku devices.

Data Destination: Network

Available since firmware version 9

In order to stream a channel's profiling data to a network while the channel is running, add `bsprof_data_dest=network` to the channel's manifest. Streaming data over the network is especially useful when profiling a channel's memory usage because all memory operations are included in the profiling data, and the amount of space necessary to store the data can be very large. By streaming the data to a network, the data size is limited primarily by the host computer receiving the data, and not by the available memory on the device itself. Even while streaming the profiling data to the network, there are still additional demands placed on the device's resources while profiling as compared to running a channel without profiling. However, the use of resources on the device is significantly reduced.

When this feature is enabled, the start of the channel is delayed until a network connection is received by the device, which is the destination for the data. When the channel is launched, a message similar to the following appears on the port 8085 developer console:

```
08-31 23:15:29.542 [scrpt.prof.connect.wait] Waiting for connection to http://192.168.1.1:8090/bsprof/channel.bsprof
```

The URL is used with wget, curl, or a web browser. Once a connection is received from one of those programs, the following message appears on the developer console:

```
08-31 23:15:38.939 [scrpt.prof.connect.ok] profiler connected
```

When the channel exits, the following message appears on the developer console:

```
08-31 23:16:04.774 [scrpt.prof.save.ok] Profiling data complete, sent via network
```

Once that message is seen, the profiling connection is closed by the device and the remote file is populated with profiling data.

Processing the data

After downloading the `.bsprof` file, the data can be viewed using the BrightScript Profiler Visualization Tool.

BrightScript Profiler Visualization Tool v.2.0.5

Download App

big.profiler.v.2.bsprof (48034.87 Kb) Browse

Channel:	Roku Channel (dev)	Date:	9/12/2018 12:41:21 PM
Version:	v.2.0.0	Run:	1766 seconds
Vendor:	Roku	ReqRatio:	1
Device Model:	4200X	ActRatio:	1
Firmware:	049.80E00013A		

CPU | Memory Allocation

Collapse All | Expand All | Empty Threads | Lib Grouping | Search Function

Function	CPU Self	CPU Callees	CPU Total	Time Self(s)	Time Callees(s)	Time Total(s)	Calls
Thread 1	0	3627209	3627209	0	1765757660	1765757660	0
main()	14	3627195	3627209	6745	1765750915	1765757660	1
firmwarerequiresupdate()	56	0	56	6927	0	6927	1
getdeeplinkaa()	77	0	77	6226	0	6226	1
getwellknownargs()	13	0	13	5737	0	5737	1
showhomescreen()	2167308	1459741	3627049	1264072212	501659813	1765732025	1
initlocalytics()	81	11733	11814	2562	801240	803802	1
localytics()	188	800	988	3265	217774	221039	1
ll_load_custom_dimensions()	167	458	625	203981	9673	213654	1
ll_debug_log()	8	0	8	579	0	579	1
ll_read_registry()	270	180	450	5708	3386	9094	10
ll_write_registry()	180	0	180	3386	0	3386	10

BrightScript Profiler Visualization Tool - CPU output view

BrightScript Profiler Visualization Tool v.2.0.5

Download App

big.profiler.v.2.bsprof (48034.87 Kb) Browse

Channel: Roku Channel (dev)
Version: v.2.0.0
Vendor: Roku
Device Model: 4200X
Firmware: 049.80E00013A

Date: 9/12/2018 12:41:21 PM
Run: 1766 seconds
ReqRatio: 1
ActRatio: 1

CPU | Memory Allocation

Collapse All | Expand All | Empty Threads | Lib Grouping | Search Function

Function		Alloc Self	Free Self	Free Other	Leaks Self	Alloc Total	Free Total	Leaks Total
Thread 1		0	0	0	0	100280996	83425492	16855504
main()	more info	0	0	596	0	100280996	83425492	16855504
showhomescreen()	more info	56768	42620	4352116	14148	100279520	83424956	16854564
processlocevent()	more info	1260	0	21396	1260	64355724	58461496	5894228
ll_tag_screen()	more info	31272	24776	19276	6496	53352116	48893108	4459008
ll_screen_viewed()	more info	29940	21904	28960	8036	34991100	30612604	4378496
ll_set_session_value()	more info	17834744	17831360	12272	3384	17854128	17835456	18672
ll_write_registry()	more info	10432	3160	1456	7272	10432	3160	7272
ll_to_string()	more info	4264	936	2816	3328	7080	936	6144
ll_is_integer()	more info	1880	0	0	1880	1880	0	1880
ll_is_string()	more info	936	0	0	936	936	0	936
ll_has_session()	more info	1872	0	0	1872	1872	0	1872

BrightScript Profiler Visualization Tool - Memory output view

Understanding the data

The **profiling data** is divided into 5 main sections:

- The function (and associated call path which can be expanded),
- CPU time,
- wall-clock time,
- function call counts,
- memory profiler output

The **CPU time** and **wall-clock time** sections are further divided into separate sections for `self`, `callees`, and `total`:

- **`self`** refers to the CPU/wall-clock time the function consumes itself
- **`callees`** is the amount of time consumed by any functions called by the original function
- **`total`** is the amount of time consumed by the original function (`self`) and any `callee` functions

Function call paths

This section of the profiling data contains the function calls in each thread. For SceneGraph applications, each thread corresponds to either the main BrightScript thread or a single instance of a `<component>`.

For example, if you have a Task node that is instantiated multiple times, each instance will appear as a separate thread. The results are the same for any custom `<component>` in the channel that is instantiated multiple times. The main BrightScript thread (`Thread main`) is also represented as a single thread even though it has no `<component>`.

CPU time

The first 3 columns of the visualization tool lists:

- the time consumed by the CPU process (`CPU Self`),
- any other functions that are called (`CPU Callees`),
- and the total amount of time consumed (`CPU Self` + `CPU Callees`).

> CPU time refers to the number of operations each function takes to complete and this number should be equal on the low end and high-end Roku devices.

Wall-clock time

The wall-clock time lists:

- the amount of "wall-clock" time for the function,
- its callees,
- and the total.

> **Wall-clock time refers to the real world time that a function takes to complete. This value can vary across different Roku devices.** For example, a function may take an equal number of operations to complete across different Roku devices but low-end Roku devices can take more real-world time to complete one operation than a high-end Roku device.

Function call counts

Function call counts lists the number of times the functions were called when the channel ran with profiling enabled.

Values from memory profiling

Calls	Number of times a function was called
Cpu.self	CPU* used in a function, itself
Cpu.callees	CPU* used in functions called by a function
Cpu.total	Cpu.self + cpu.callees
Mem.self	Memory allocated within a function itself, but not freed (leaks)

Mem.callees	Memory allocated by functions called by a function, but not freed (leaks)
Mem.total	Mem.self + mem.callees
Tm.self	Real (wall-clock) time spent on a function, itself
Tm.callees	Real (wall-clock) time spent on functions called by a function
Tm.total	Tm.self + tm.callees
Avg.cpu_self Avg.cpu_callees Avg.cpu_total Avg.mem_self Avg.mem_total Avg.tm_self Avg.tm_callees Avg.tm_total	Average of the metric, over the number of calls (e.g., if cpu.self=100 and calls=2, avg_cpu_self will be 50)

> A "memory leak" is simply any memory that is allocated, but not freed while the profiler was running. If memory is freed while profiling is paused, the free memory is not tracked and the memory may show up as "leaked."
>
> Time is measured as if a stopwatch were used to time the action. For example, if a function makes a network call, there may be very little CPU time used, but a significant amount of time waiting for the network response.

If any of these metrics appear in a call path, they are specific to that call path. For example, in this call path:

```
<root>: cpu.self=0,cpu.callees=14700,tm.self=0.000,tm.callees=1.989,mem.self=0,mem.callees=324452,calls=0

+- func1(): pkg:/components/file1.brs:83,cpu.self=200,cpu.callees=14500,tm.self=0.728,tm.callees=1.261,mem.
self=5840,mem.callees=318612,calls=1

|  +- func2(): pkg:/components/file2.brs:22,cpu.self=14500,cpu.callees=0,tm.self=1.261,tm.callees=0.000,mem
.self=31800,mem.callees=612,calls=1
```

The metrics for func2() are specific to when it is called from func1().

However, in the table below:

```
------------- BEGIN: TOP CONSUMERS: CPU.SELF -----------------

  1: func1(): pkg:/components/file1.brs:83,cpu.self=300,cpu.total=450,tm.self=0.001,tm.total=0.001,mem.self
=0,mem.total=0,calls=5

  2: func2(): pkg:/components/file2.brs:22,cpu.self=55430,cpu.total=80500,tm.self=0.126,tm.total=0.126,mem.
self=0,mem.total=0,calls=3

-------------- END: TOP CONSUMERS: CPU.SELF -----------------
```

The metrics displayed are the totals for all calls to each function, on any call path.

Using this data

Here are a few key points on how to use this data to improve channel performance:

Data Type	Definition and Best Use
High wall-clock time but low CPU time	This pattern shows a function is consistently waiting, whether it be for input or a response from an external source. These functions are best suited for Task nodes so that it doesn't block the main thread.
Complex functions	Try to simplify the functions as much as possible. If a function handles multiple tasks, consider breaking it out into several functions to further isolate how much CPU or wall-clock time is consumed by each task.
Functions that consume a large amount of CPU or wall-clock time	Try to reduce the number of calls to these functions as much as possible. Move functions to Task nodes, if they are consistently waiting. A function can be determined to be waiting if it's wall-clock time is high, but its CPU cost is low.