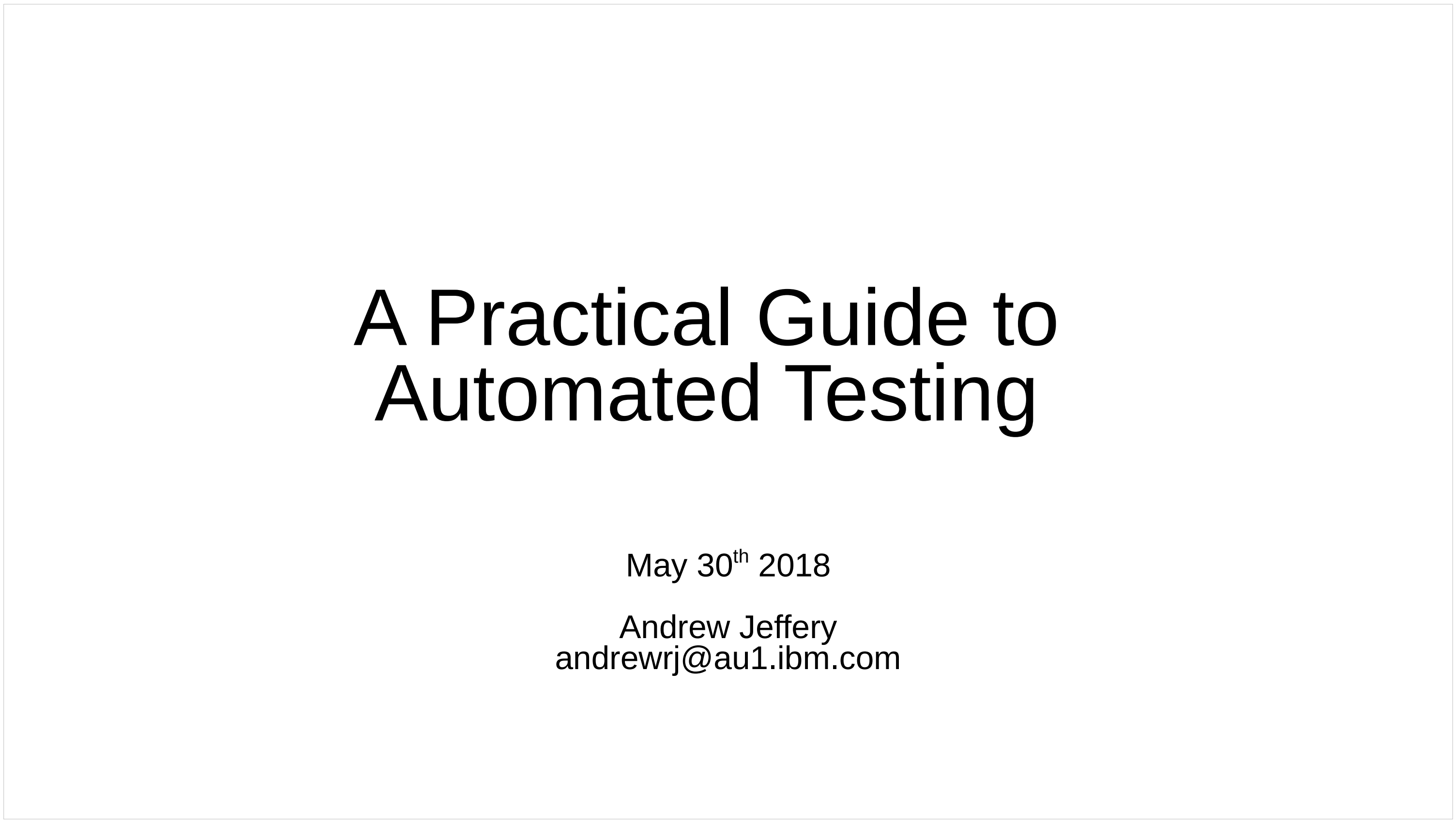


A Practical Guide to Automated Testing

May 30th 2018

Andrew Jeffery
andrewrj@au1.ibm.com

Hello,

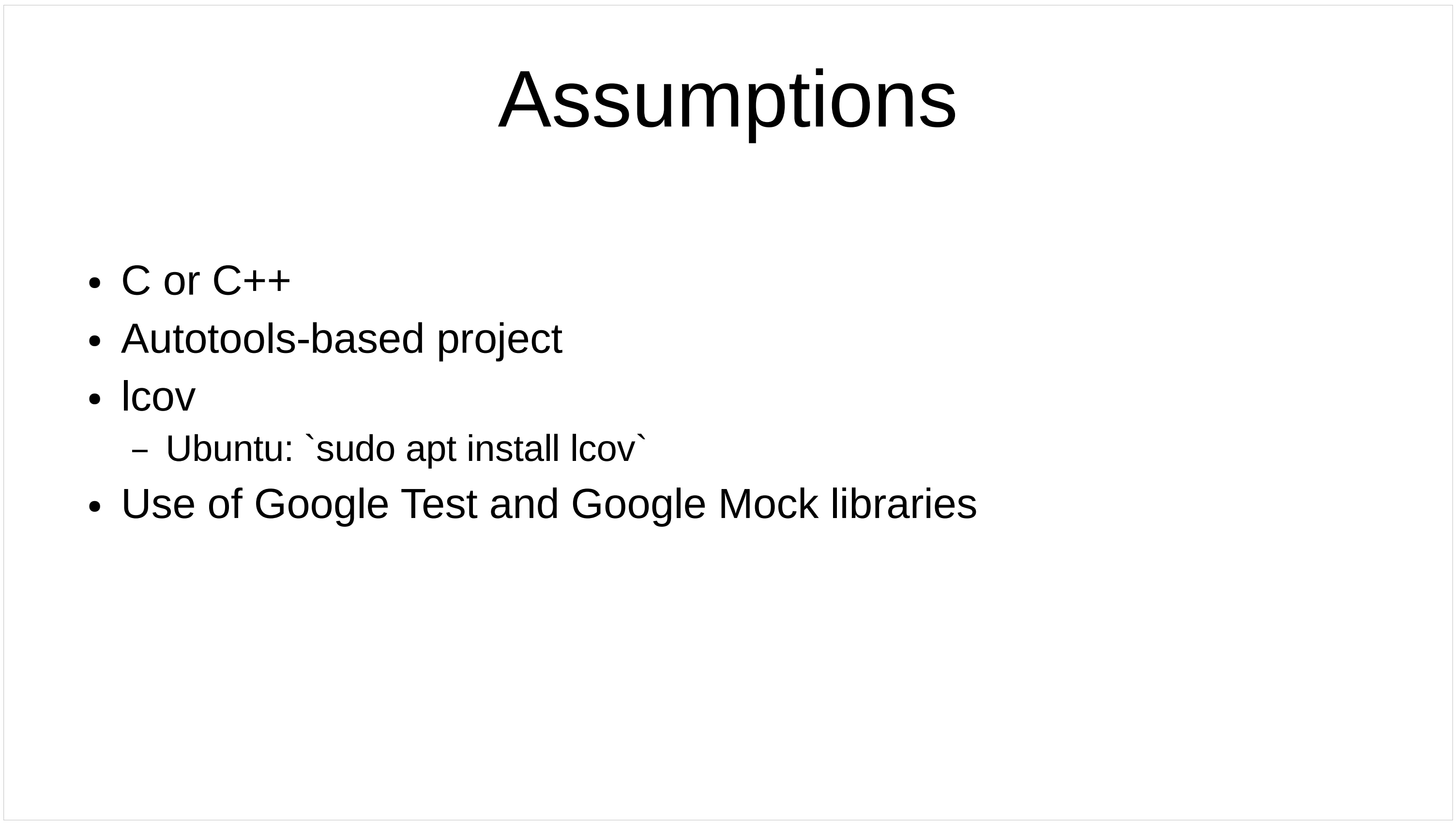
This talk is a walk-through of adding automated testing to an existing, untested codebase.

Specifically, it’s not a talk about the theory around testing such as different test types, strategies, and design – these are topics we can talk about in the future

Overall I hope that it inspires you to write tests for your code by giving you the tools to integrate test suites into your project’s build system, and write tests that achieve good coverage

More than anything, I hope to show that this is straight-forward to do, to the point of giving you snippets that you can almost copy and paste straight into your work.

Please stop me if you have questions along the way



Assumptions

- C or C++
- Autotools-based project
- lcov
  - Ubuntu: `sudo apt install lcov`
- Use of Google Test and Google Mock libraries

To that end, it does come with some assumptions

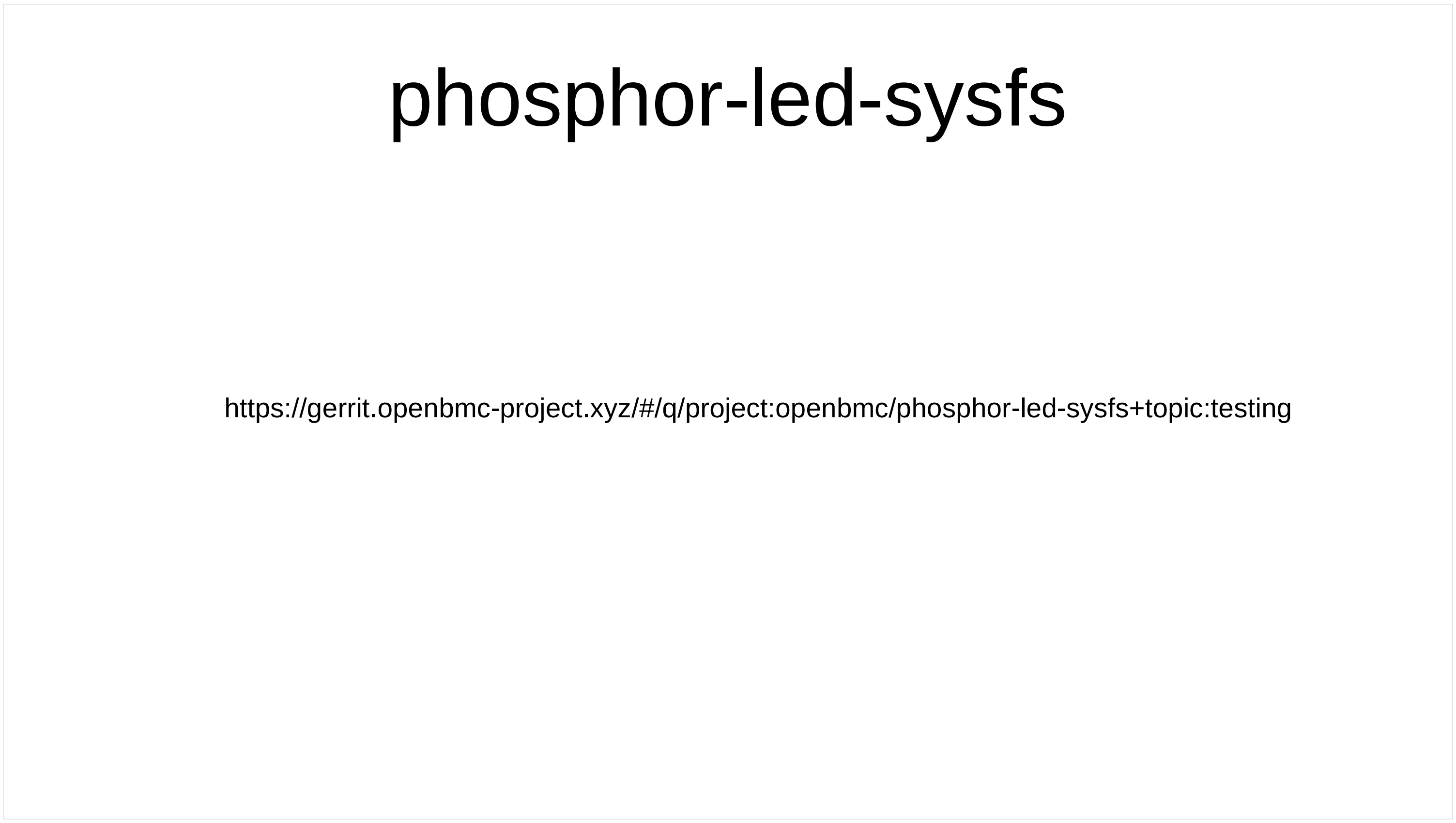
I will assumes that you're writing C++ (or C at a stretch). We also need to write tests for the Python code in OpenBMC, but as it's a different language it requires different practical examples and we can talk about that another time

I will also assume you're working with an autotools project

And that you can install lcov, a gcov-based code coverage rendering tool.

To test our C++ we're going to wire Google Test-based binaries into our Autotools build system

And make use of Google Mock to influence the behaviour of code under test



phosphor-led-sysfs

https://gerrit.openbmc-project.xyz/#/q/project:openbmc/phosphor-led-sysfs+topic:testing

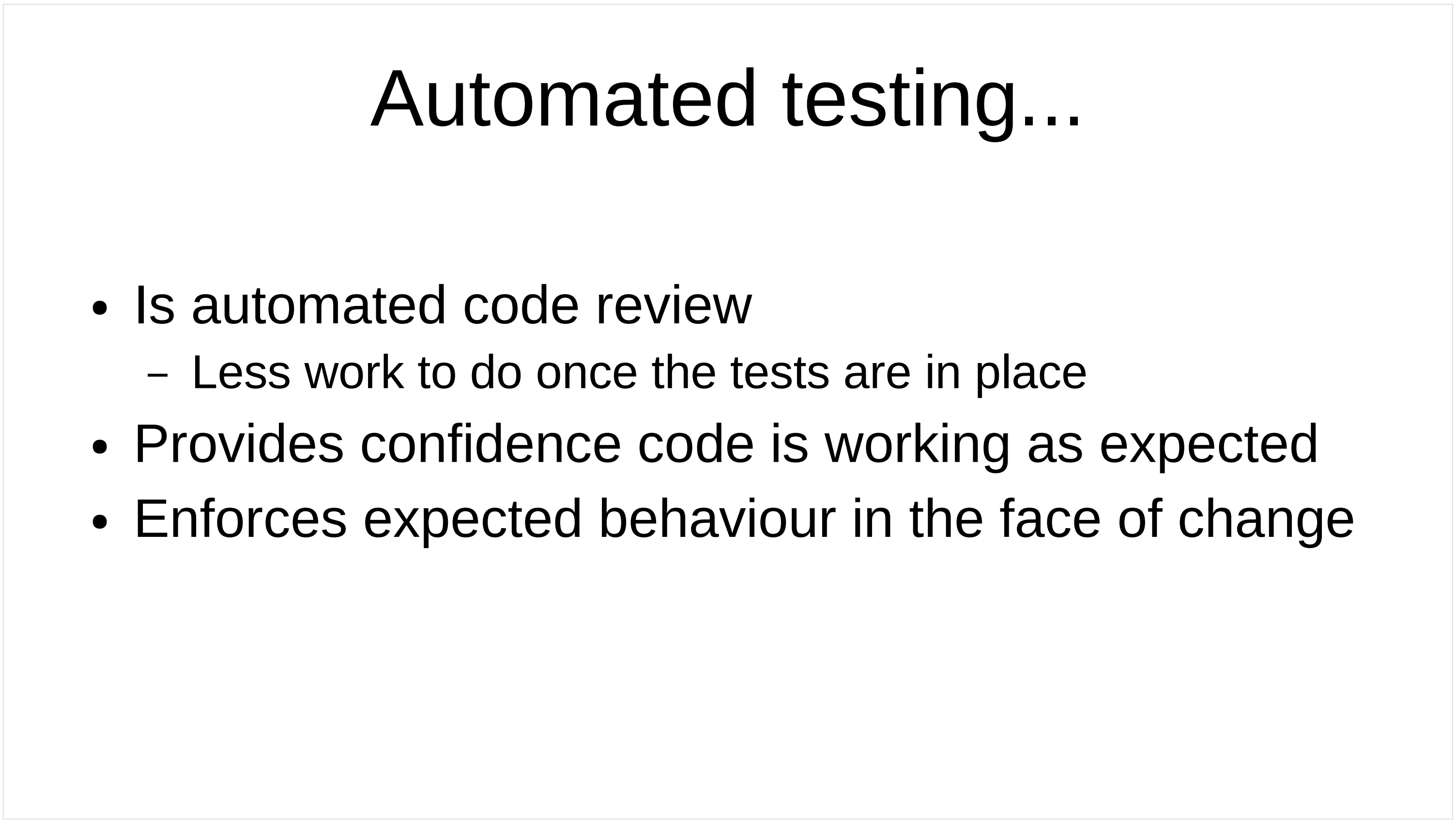
It would only be a theoretical guide if we weren’t modifying something concrete!

To develop this talk I dug through the repositories of the OpenBMC Github organisation to find an untested repository that was small and reasonably self-contained.

One of the better candidates was phosphor-led-sysfs. Again, to improve the chance that this talk would get some traction I’ve developed a series of 18 patches against phosphor-led-sysfs that demonstrate:

- Integrating coverage analysis tools
- Developing a first-pass test suite
- Using code coverage to find untested paths
- Patching code so we can test paths we missed
- Patching tests so we can characterise the code
- Using compiler features to enhance our testing

We will be addressing each of these topics in turn



Automated testing...

- Is automated code review
  - Less work to do once the tests are in place
- Provides confidence code is working as expected
- Enforces expected behaviour in the face of change

A practical guide is only useful if we’re motivated to use it, so why is automated testing a good thing?

We want to make the machines do the hard work for us. This means less time you, as a reviewer or maintainer, will need to spend looking at patches and mentally modelling the contracted behaviour vs the current behaviour vs the changed behaviour, as well as providing feedback on readability, maintainability and other non-functional necessities.

With good use-case test coverage you can be confident that the code works as intended

These same tests enforce expected behaviour on future changes

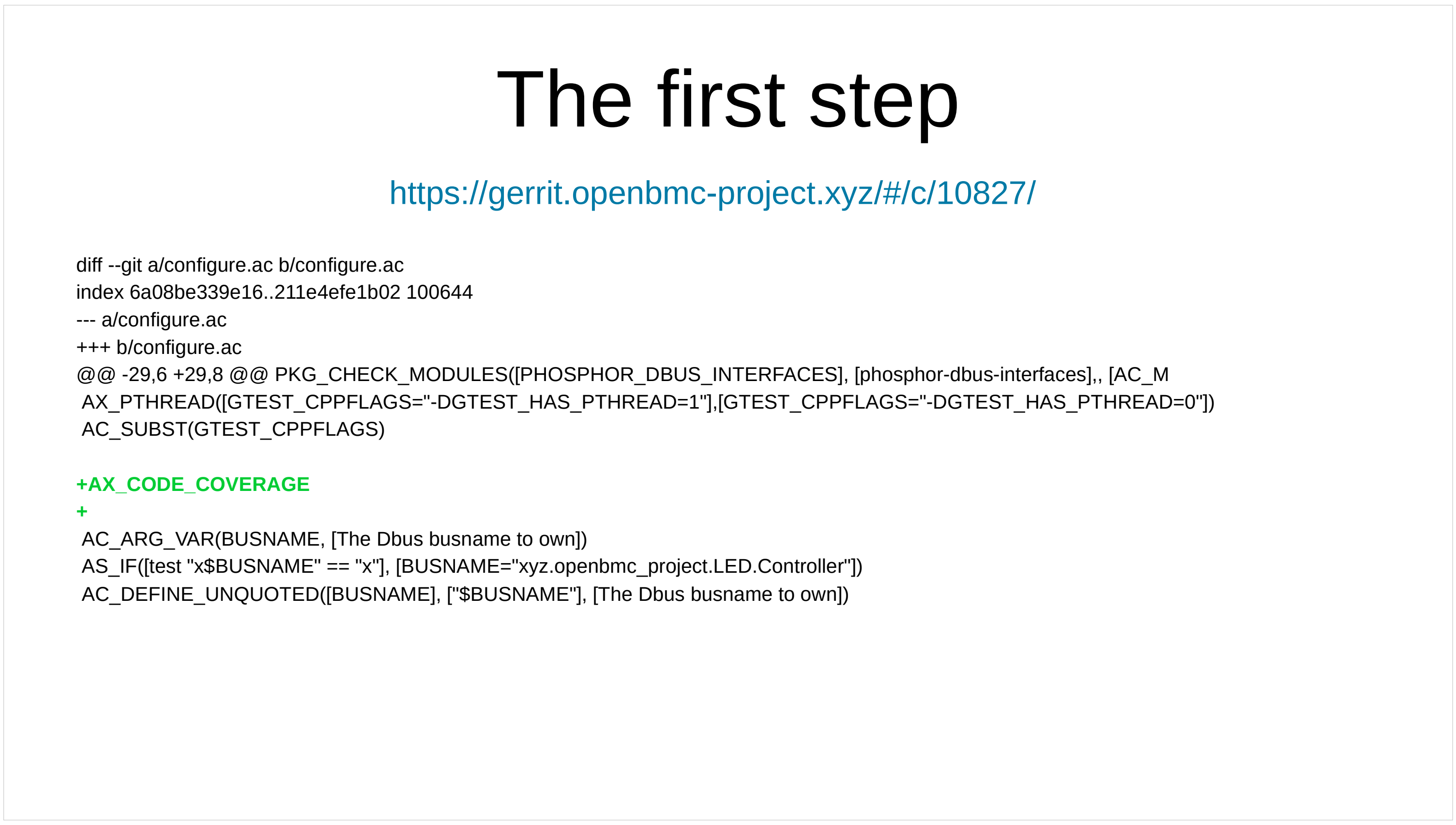
But only for the execution paths that your tests touch!

Your confidence in your code can only be proportional to the code paths executed by manual or automated tests.

The easy way out here is to minimise the problems you're solving in software – less code means fewer bugs

But when you can't reduce number of problems you are solving in software - or their complexity - you need to fall back on testing, and instrumenting your tests for coverage of your code

So the first practical thing we should look at is adding test coverage instrumentation to your Autotools config



The first step

https://gerrit.openbmc-project.xyz/#/c/10827/

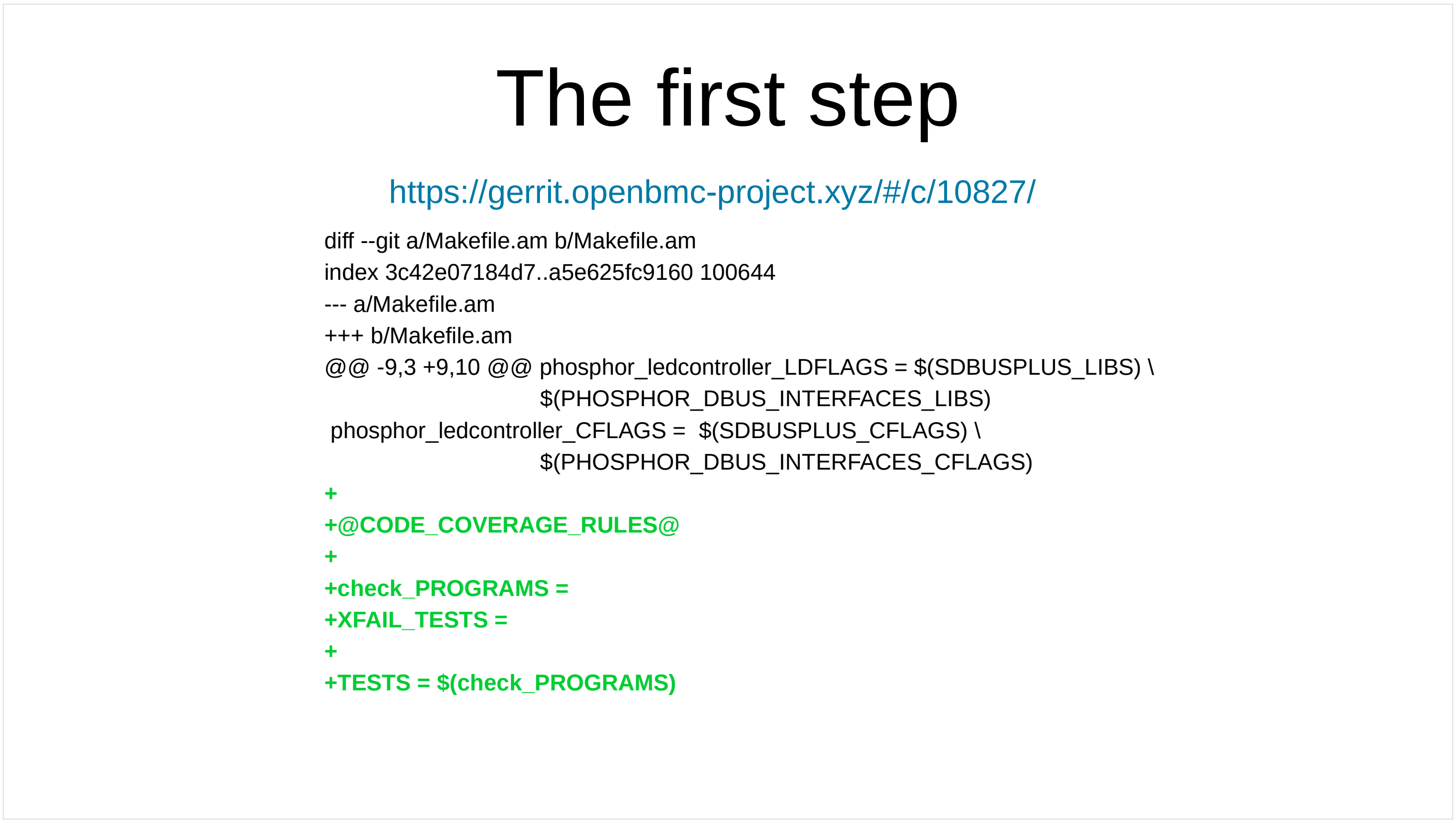
```
diff --git a/configure.ac b/configure.ac
index 6a08be339e16..211e4efe1b02 100644
--- a/configure.ac
+++ b/configure.ac
@@ -29,6 +29,8 @@ PKG_CHECK_MODULES([PHOSPHOR_DBUS_INTERFACES], [phosphor-dbus-interfaces],, [AC_M
 AX_PTHREAD([GTEST_CPPFLAGS="-DGTEST_HAS_PTHREAD=1"],[GTEST_CPPFLAGS="-DGTEST_HAS_PTHREAD=0"])
 AC_SUBST(GTEST_CPPFLAGS)

+AX_CODE_COVERAGE
+
 AC_ARG_VAR(BUSNAME, [The Dbus busname to own])
 AS_IF([test "x$BUSNAME" == "x"], [BUSNAME="xyz.openbmc_project.LED.Controller"])
 AC_DEFINE_UNQUOTED([BUSNAME], ["$BUSNAME"], [The Dbus busname to own])
```

I’ve added a link to the specific phosphor-led-sysfs commit in question at the top of the slide. This is done throughout the slide deck for your reference. Do note that I’ve taken some liberties with the code in the slides compared to what’s in the patches.

This change covers what you need to do to configure.ac

It’s literally just adding the line AX_CODE_COVERAGE

We do however need a companion change in Makefile.am to make use of the code coverage feature



The first step

https://gerrit.openbmc-project.xyz/#/c/10827/

```
diff --git a/Makefile.am b/Makefile.am
index 3c42e07184d7..a5e625fc9160 100644
--- a/Makefile.am
+++ b/Makefile.am
@@ -9,3 +9,10 @@ phosphor_ledcontroller_LDFLAGS = $(SDBUSPLUS_LIBS) \
                                 $(PHOSPHOR_DBUS_INTERFACES_LIBS)
 phosphor_ledcontroller_CFLAGS =  $(SDBUSPLUS_CFLAGS) \
                                 $(PHOSPHOR_DBUS_INTERFACES_CFLAGS)
+
+@CODE_COVERAGE_RULES@
+
+check_PROGRAMS =
+XFAIL_TESTS =
+
+TESTS = $(check_PROGRAMS)
```

And here’s the change to Makefile.am

We add the @CODE_COVERAGE_RULES@ magic, and define some new make variables

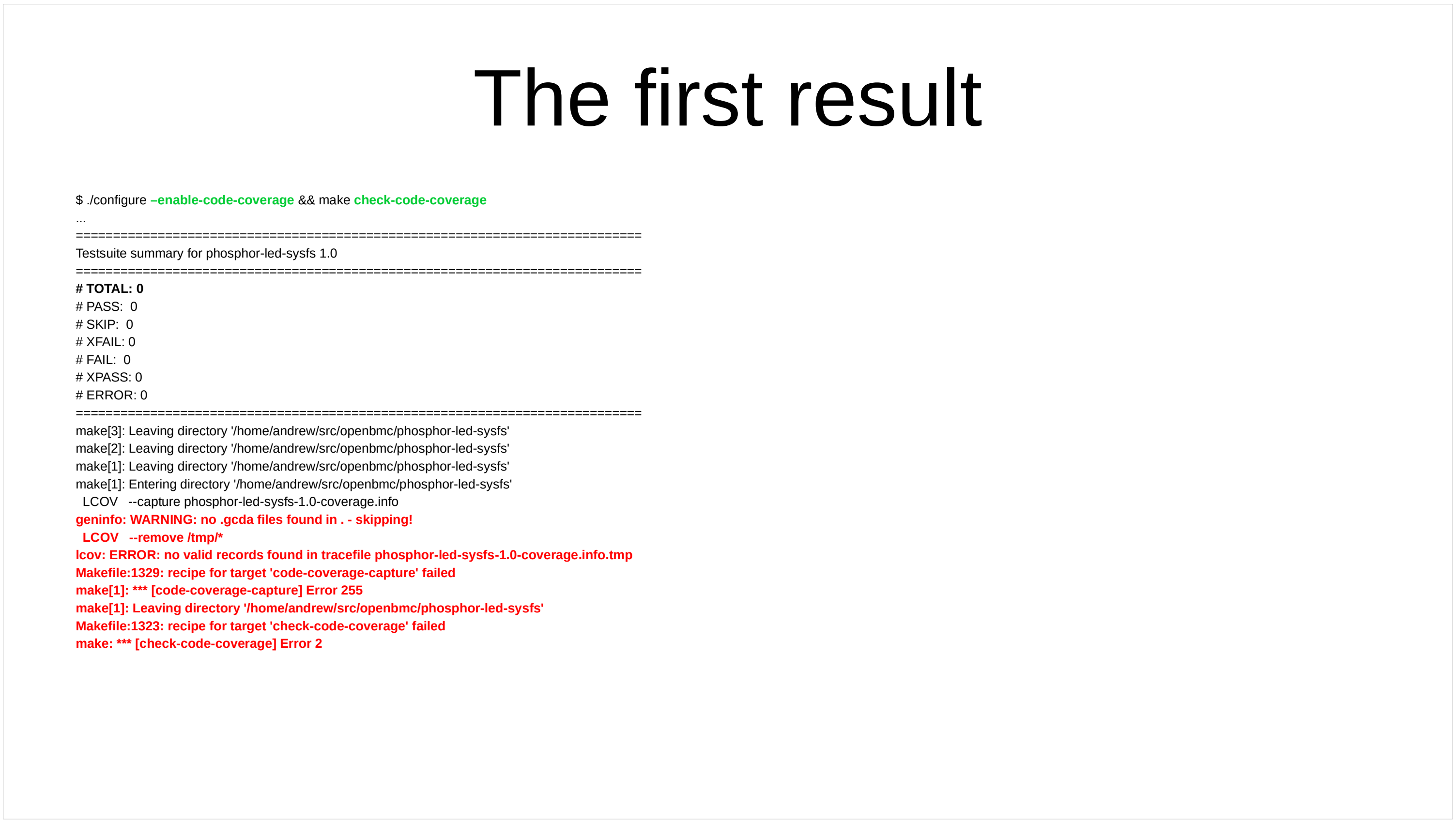
check_PROGRAMS describes the binaries that will be run from the `check` `make` target

TESTS defines the applications whose output and exit status the test runner should consider as input

XFAIL_TESTS defines the test applications that we expect to exit with a non-zero status. If these tests unexpectedly pass (exit with a zero status) the test suite will also fail, just as it would if an application listed in TESTS but not XFAIL_TESTS exits with a non-zero status

And that’s it! We now have test coverage instrumentation available to us

But, what do we do to make it work?



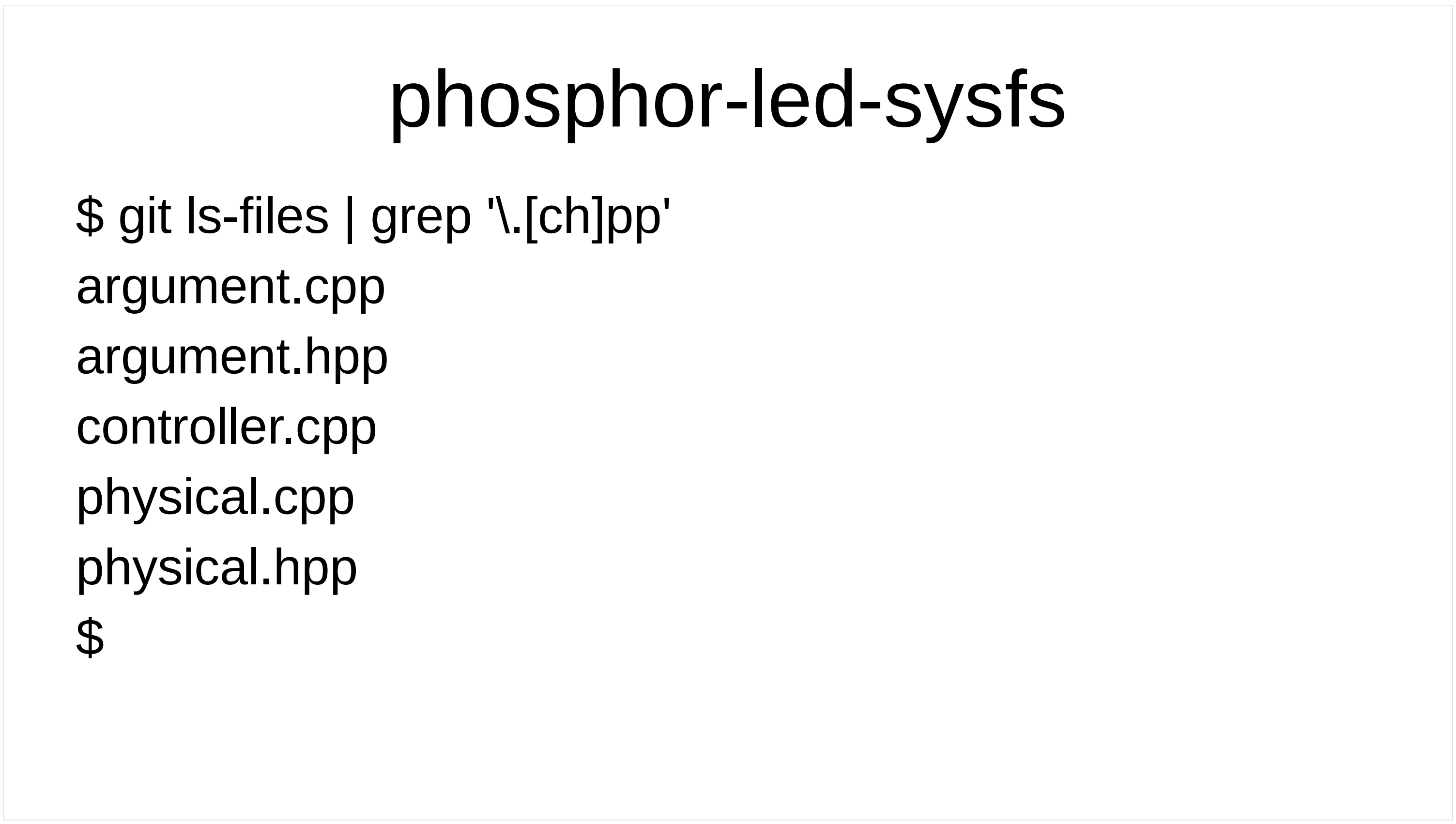
The first result

```
$ ./configure –enable-code-coverage && make check-code-coverage
...
============================================================================
Testsuite summary for phosphor-led-sysfs 1.0
============================================================================
# TOTAL: 0
# PASS:  0
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
============================================================================
make[3]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
make[2]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
make[1]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
make[1]: Entering directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
  LCOV   --capture phosphor-led-sysfs-1.0-coverage.info
geninfo: WARNING: no .gcda files found in . - skipping!
  LCOV   --remove /tmp/*
lcov: ERROR: no valid records found in tracefile phosphor-led-sysfs-1.0-coverage.info.tmp
Makefile:1329: recipe for target 'code-coverage-capture' failed
make[1]: *** [code-coverage-capture] Error 255
make[1]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
Makefile:1323: recipe for target 'check-code-coverage' failed
make: *** [check-code-coverage] Error 2
```

We need to run the configure script with the `--enable-code-coverage` option. This generates a Makefile with the `check-code-coverage` make target, which we use to generate the test coverage report

However, running `make check-code-coverage` on phosphor-led-sysfs gives us some pretty bleak output

We need more than zero tests before test coverage reports can be useful.



phosphor-led-sysfs

```
$ git ls-files | grep '\.[ch]pp'
argument.cpp
argument.hpp
controller.cpp
physical.cpp
physical.hpp
$
```

To add tests we first need to understand the code-base.

Luckily phosphor-led-sysfs doesn’t have a lot to it, containing only 5 source files

phosphor-led-sysfs

- **argument.[ch]pp**: Argument parsing
- **controller.cpp**: main(), orchestrates everything
- **physical.[ch]pp**: Testable business logic!
  - Converts DBus API to kernel LED sysfs ABI

Note: Talk briefly about argument.[ch]pp and controller.cpp

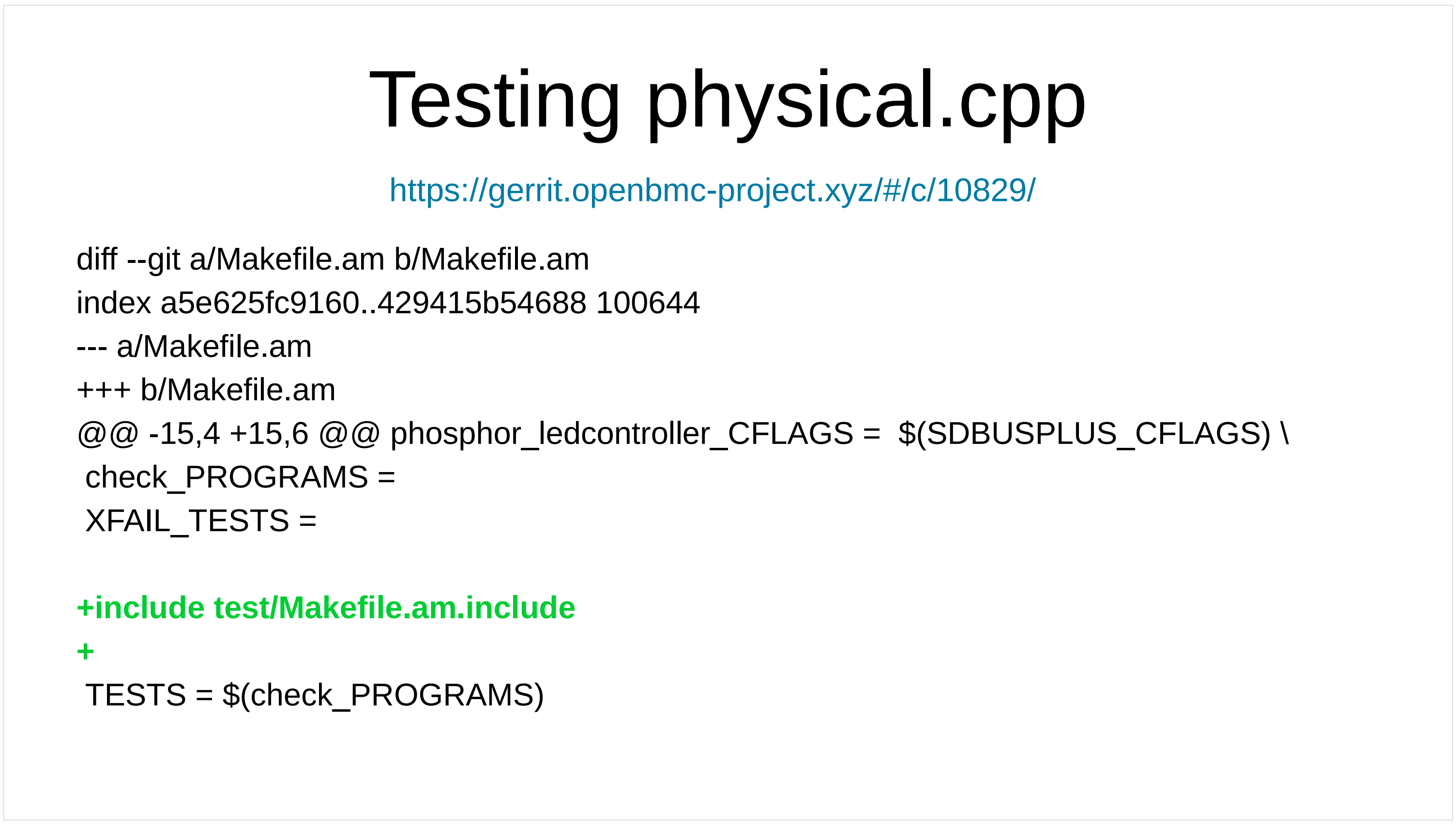
For the purposes of the our testing adventure, only the “physical” header and implementation are really relevant.

physical.[ch]pp together implement the Physical class, which is the plumbing between DBus properties representing an LED and the kernel’s sysfs LED attributes

The DBus API for LEDs is described in the usual place, in the phosphor-dbus-interfaces repository

Physical caters to turning LEDs solid on, solid off, or blinking at a defined percent duty over a 1 second period.

Physical defines some business logic, and is something we can test



Testing physical.cpp

https://gerrit.openbmc-project.xyz/#/c/10829/

```
diff --git a/Makefile.am b/Makefile.am
index a5e625fc9160..429415b54688 100644
--- a/Makefile.am
+++ b/Makefile.am
@@ -15,4 +15,6 @@ phosphor_ledcontroller_CFLAGS =  $(SDBUSPLUS_CFLAGS) \
 check_PROGRAMS =
 XFAIL_TESTS =

+include test/Makefile.am.include
+
 TESTS = $(check_PROGRAMS)
```

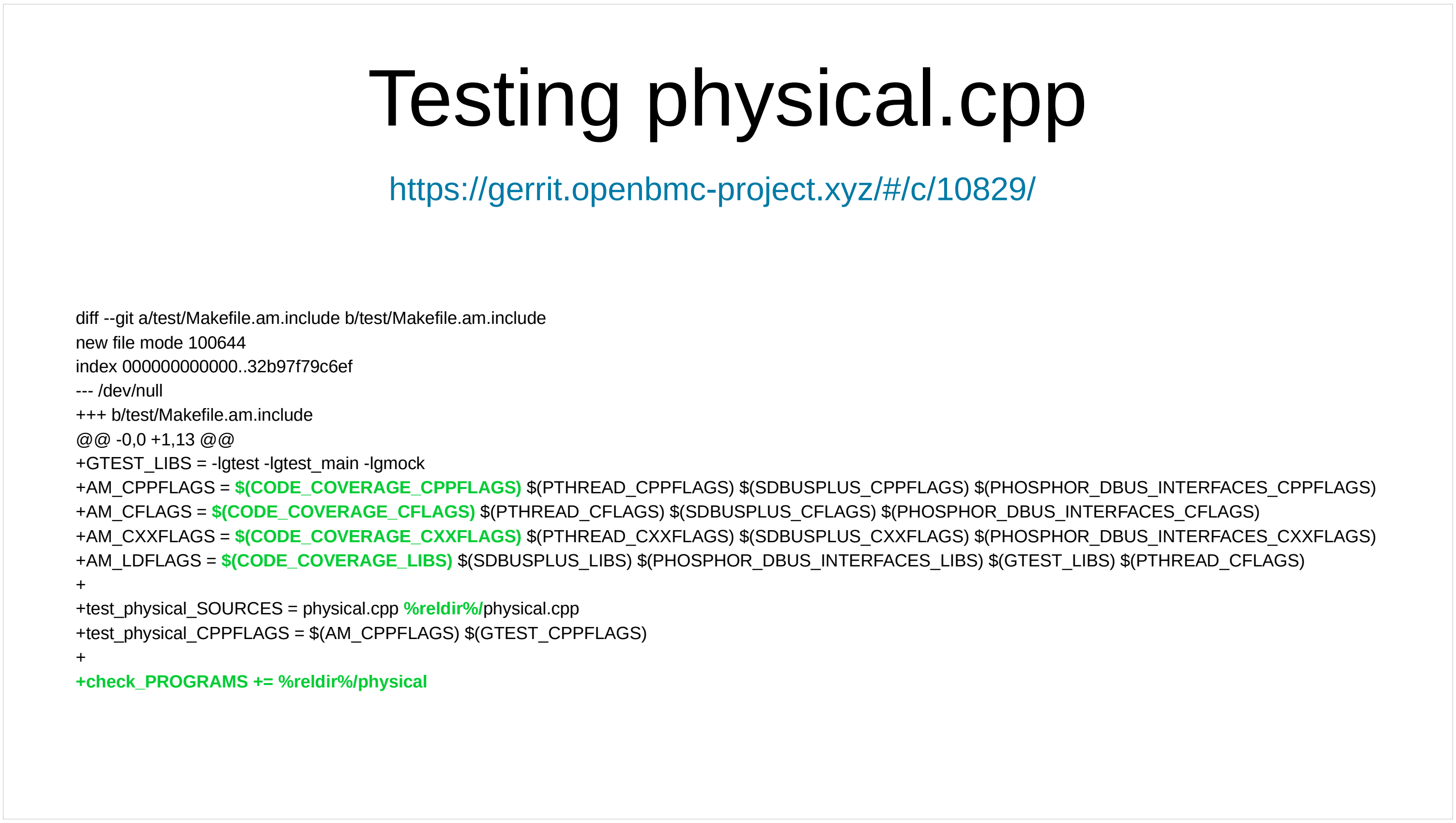
So far we have:

- added the test coverage instrumentation, and
- Investigated what parts of the application we can test

What we must do now is integrate our tests and test infrastructure into Autotools

One of the things I’m going to demonstrate here is how to write non-recursive make. This is important for making your build as parallel as possible, thus reducing wall-clock build times with `make -j$(nproc)`. Recursive make partitions your targets by directory, which can really limit the parallelisation of your build. This technique applies to all subdirectories, not just tests.

The non-recursive approach starts with defining Makefile.am.include snippets in your sub-directories, and using the `include` directive to bring them into your root Makefile.am as we have here



Testing physical.cpp

https://gerrit.openbmc-project.xyz/#/c/10829/

```
diff --git a/test/Makefile.am.include b/test/Makefile.am.include
new file mode 100644
index 000000000000..32b97f79c6ef
--- /dev/null
+++ b/test/Makefile.am.include
@@ -0,0 +1,13 @@
+GTEST_LIBS = -lgtest -lgtest_main -lgmock
+AM_CPPFLAGS = $(CODE_COVERAGE_CPPFLAGS) $(PTHREAD_CPPFLAGS) $(SDBUSPLUS_CPPFLAGS) $(PHOSPHOR_DBUS_INTERFACES_CPPFLAGS)
+AM_CFLAGS = $(CODE_COVERAGE_CFLAGS) $(PTHREAD_CFLAGS) $(SDBUSPLUS_CFLAGS) $(PHOSPHOR_DBUS_INTERFACES_CFLAGS)
+AM_CXXFLAGS = $(CODE_COVERAGE_CXXFLAGS) $(PTHREAD_CXXFLAGS) $(SDBUSPLUS_CXXFLAGS) $(PHOSPHOR_DBUS_INTERFACES_CXXFLAGS)
+AM_LDFLAGS = $(CODE_COVERAGE_LIBS) $(SDBUSPLUS_LIBS) $(PHOSPHOR_DBUS_INTERFACES_LIBS) $(GTEST_LIBS) $(PTHREAD_CFLAGS)
+
+test_physical_SOURCES = physical.cpp %reldir%/physical.cpp
+test_physical_CPPFLAGS = $(AM_CPPFLAGS) $(GTEST_CPPFLAGS)
+
+check_PROGRAMS += %reldir%/physical
```

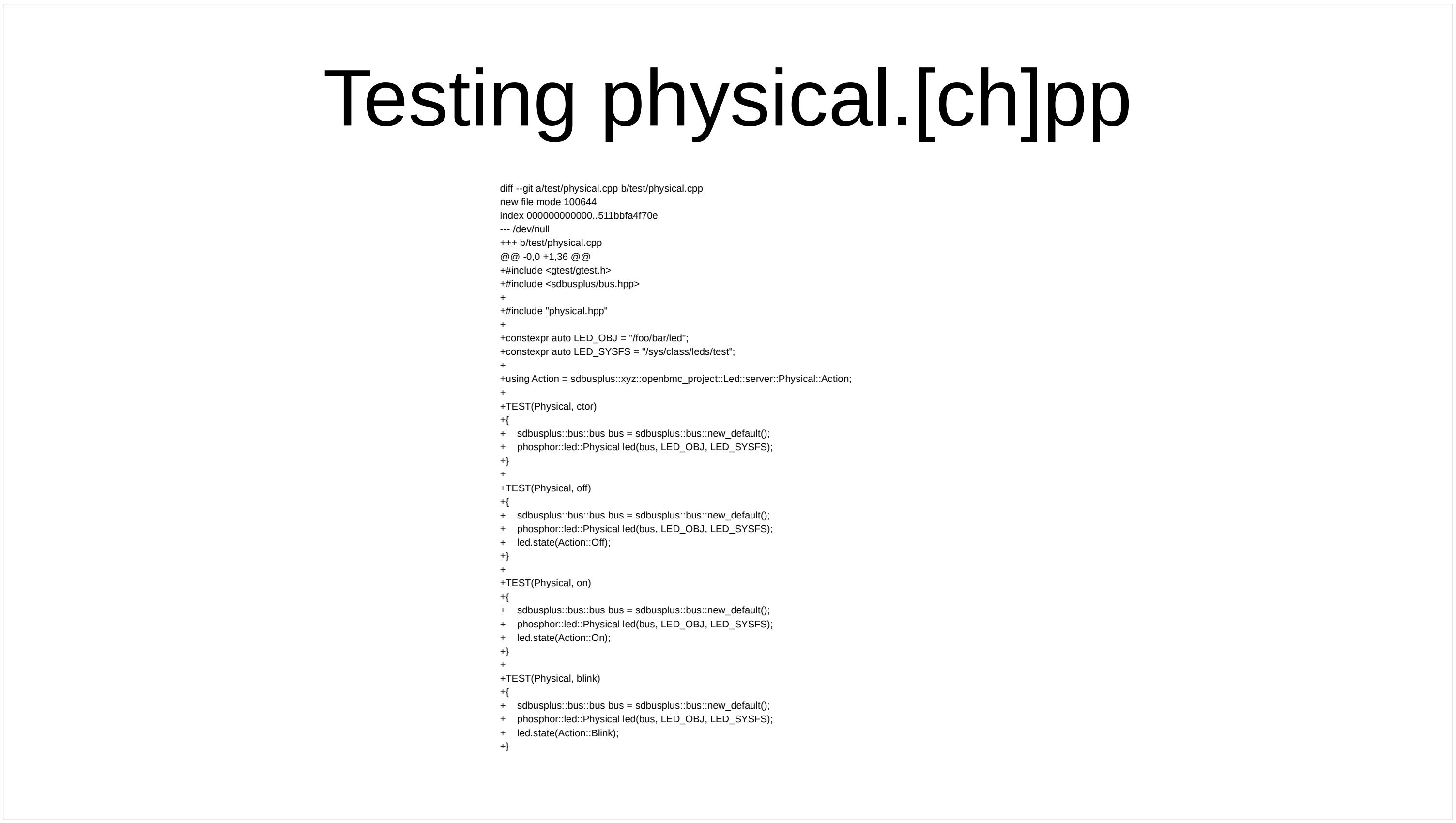
In our test/Makefile.am.include file, we define all the usual variables to build a test binary

One catch with non-recursive make is defining paths to objects. This hurdle is overcome by using the `%reldir%` magic variable, which substitutes the path to the directory containing the Makefile snippet. Source files, objects and binaries should be referenced relative to `%reldir%`.

Putting aside the non-recursive make distraction, we want to enable code coverage reports for our tests.

To do this we make sure to include the $(CODE_COVERAGE_*) variables in their respective build variables as highlighted in the slides (note that the $(AM_*) variables are used unless overridden by application-specific variables). The $(CODE_COVERAGE_*) variables hold the compiler and linker flags needed to track executed source lines, typically -fprofile-arcs -ftest-coverage (when compiling) and -lgcov (when linking)

Finally, after defining our test program variables we add its binary to `check_PROGRAMS` as seen at the bottom



Testing physical.[ch]pp

```
diff --git a/test/physical.cpp b/test/physical.cpp
new file mode 100644
index 000000000000..511bbfa4f70e
--- /dev/null
+++ b/test/physical.cpp
@@ -0,0 +1,36 @@
+#include <gtest/gtest.h>
+#include <sdbusplus/bus.hpp>
+
+#include "physical.hpp"
+
+constexpr auto LED_OBJ = "/foo/bar/led";
+constexpr auto LED_SYSFS = "/sys/class/leds/test";
+
+using Action = sdbusplus::xyz::openbmc_project::Led::server::Physical::Action;
+
+TEST(Physical, ctor)
+{
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    phosphor::led::Physical led(bus, LED_OBJ, LED_SYSFS);
+}
+
+TEST(Physical, off)
+{
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    phosphor::led::Physical led(bus, LED_OBJ, LED_SYSFS);
+    led.state(Action::Off);
+}
+
+TEST(Physical, on)
+{
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    phosphor::led::Physical led(bus, LED_OBJ, LED_SYSFS);
+    led.state(Action::On);
+}
+
+TEST(Physical, blink)
+{
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    phosphor::led::Physical led(bus, LED_OBJ, LED_SYSFS);
+    led.state(Action::Blink);
+}
```

And here’s the code for the first test suite of phosphor-led-sysfs

It uses Google Test as a test framework, and defines four separate test cases.

The test cases clearly are not complicated - I’ve managed to jam the entire file in one slide.

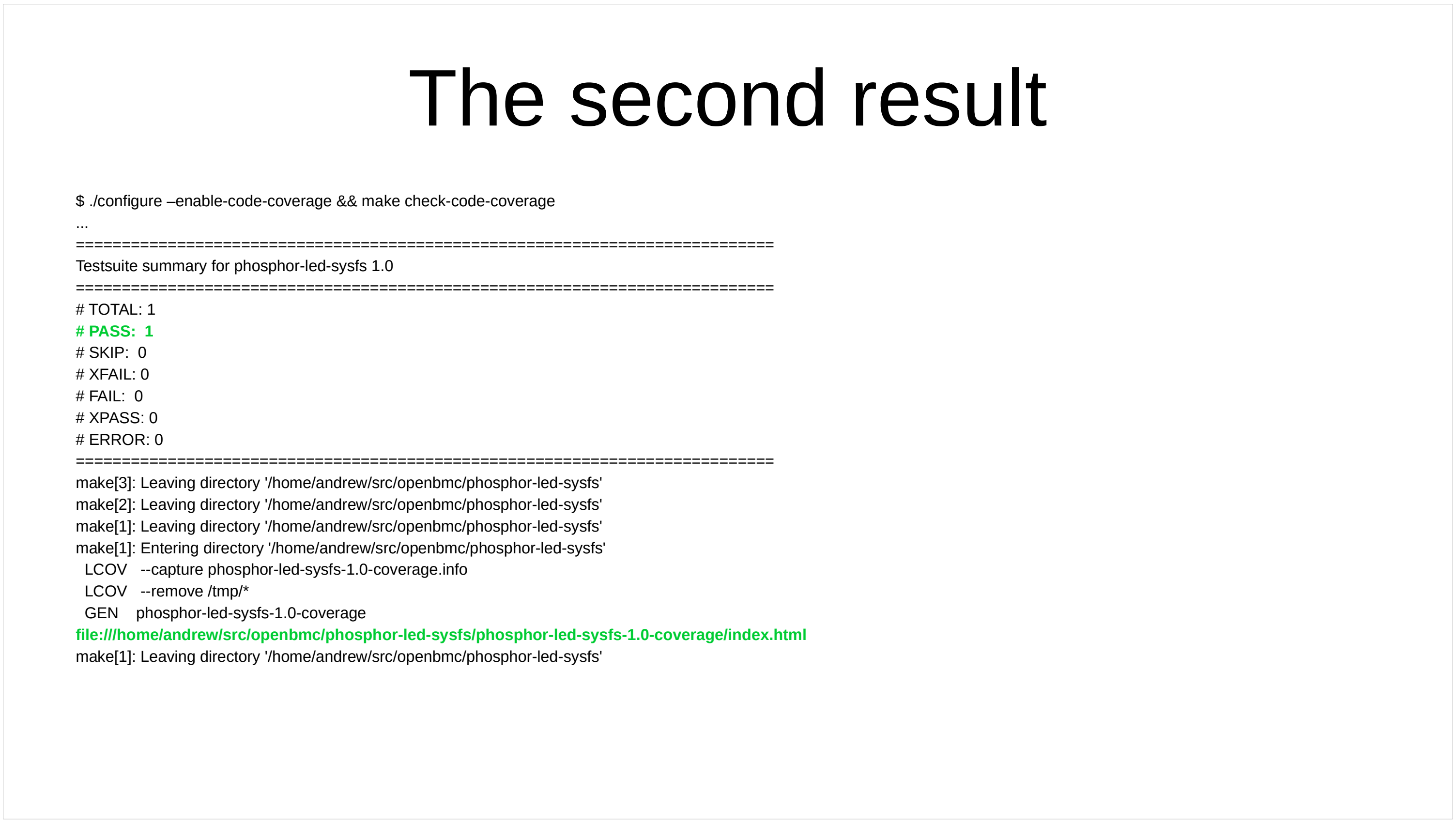
One of the tests simply constructs a Physical instance

The remaining tests simply call the APIs to turn the LED off, on and make it blink

What we know from these tests is that we don’t receive any funny exceptions when calling the interfaces. It’s not much, but it’s something

Note: Stop for any questions

Having done that, how did we do for code coverage?



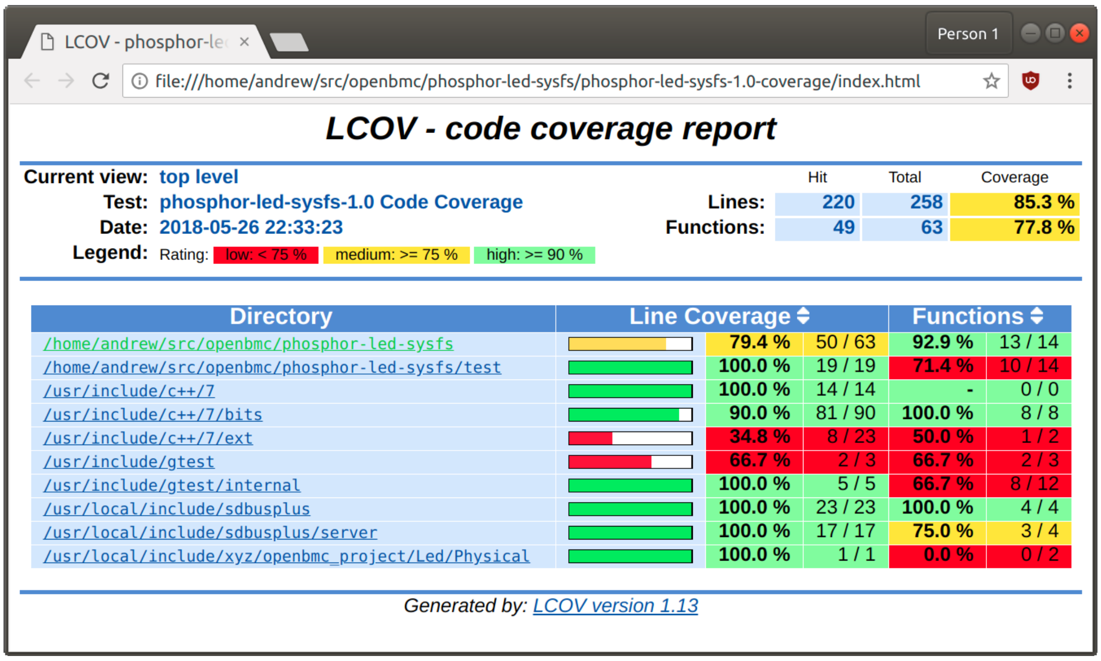
The second result

```
$ ./configure –enable-code-coverage && make check-code-coverage
...
============================================================================
Testsuite summary for phosphor-led-sysfs 1.0
============================================================================
# TOTAL: 1
# PASS:  1
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
============================================================================
make[3]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
make[2]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
make[1]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
make[1]: Entering directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
  LCOV   --capture phosphor-led-sysfs-1.0-coverage.info
  LCOV   --remove /tmp/*
  GEN    phosphor-led-sysfs-1.0-coverage
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/index.html
make[1]: Leaving directory '/home/andrew/src/openbmc/phosphor-led-sysfs'
```

We can see here that our test suite ran one test binary and passed all tests! We also successfully generated some test coverage output

Lets have a look at the code coverage report

The second result



LCOV - phosphor-le ×

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/index.html

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	phosphor-led-sysfs-1.0 Code Coverage	**Lines:**	220	258	85.3 %
Date:	2018-05-26 22:33:23	**Functions:**	49	63	77.8 %
Legend:	Rating: 🟥 low: < 75 % 🟨 medium: >= 75 % 🟩 high: >= 90 %				

Directory	Line Coverage ⇕			Functions ⇕	
/home/andrew/src/openbmc/phosphor-led-sysfs	🟨	79.4 %	50 / 63	92.9 %	13 / 14
/home/andrew/src/openbmc/phosphor-led-sysfs/test	🟩	100.0 %	19 / 19	71.4 %	10 / 14
/usr/include/c++/7	🟩	100.0 %	14 / 14	-	0 / 0
/usr/include/c++/7/bits	🟩	90.0 %	81 / 90	100.0 %	8 / 8
/usr/include/c++/7/ext	🟥	34.8 %	8 / 23	50.0 %	1 / 2
/usr/include/gtest	🟥	66.7 %	2 / 3	66.7 %	2 / 3
/usr/include/gtest/internal	🟩	100.0 %	5 / 5	66.7 %	8 / 12
/usr/local/include/sdbusplus	🟩	100.0 %	23 / 23	100.0 %	4 / 4
/usr/local/include/sdbusplus/server	🟩	100.0 %	17 / 17	75.0 %	3 / 4
/usr/local/include/xyz/openbmc_project/Led/Physical	🟩	100.0 %	1 / 1	0.0 %	0 / 2

Generated by: LCOV version 1.13

We will step through some screenshots of LCOV's output

We can see here that it's profiled lots of build components, including system libraries and headers, and provides various stats on line and function coverage

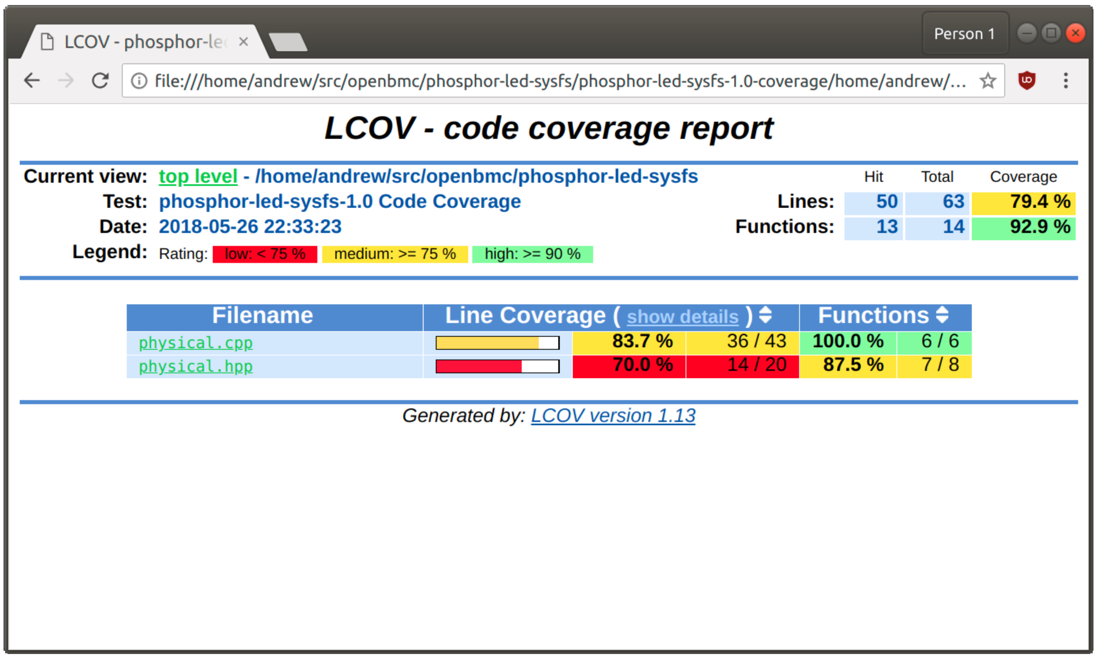
We're only interested in the coverage of our own code, and even then we're only interested in coverage of the production code, not of the code composing the test suite

From that, we can see that we've achieved 79.4% code coverage and 92.9% function coverage, from four tests.

Again, admittedly, not very good tests, but something we can build on.

Lets drill into the stats on the production code

The second result



LCOV - phosphor-le ×

Person 1

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/andrew/...

LCOV - code coverage report

Current view: top level - /home/andrew/src/openbmc/phosphor-led-sysfs
Test: phosphor-led-sysfs-1.0 Code Coverage
Date: 2018-05-26 22:33:23
Legend: Rating: low: < 75 % medium: >= 75 % high: >= 90 %

	Hit	Total	Coverage
Lines:	50	63	79.4 %
Functions:	13	14	92.9 %

Filename	Line Coverage (show details)			Functions	
physical.cpp		83.7 %	36 / 43	100.0 %	6 / 6
physical.hpp		70.0 %	14 / 20	87.5 %	7 / 8

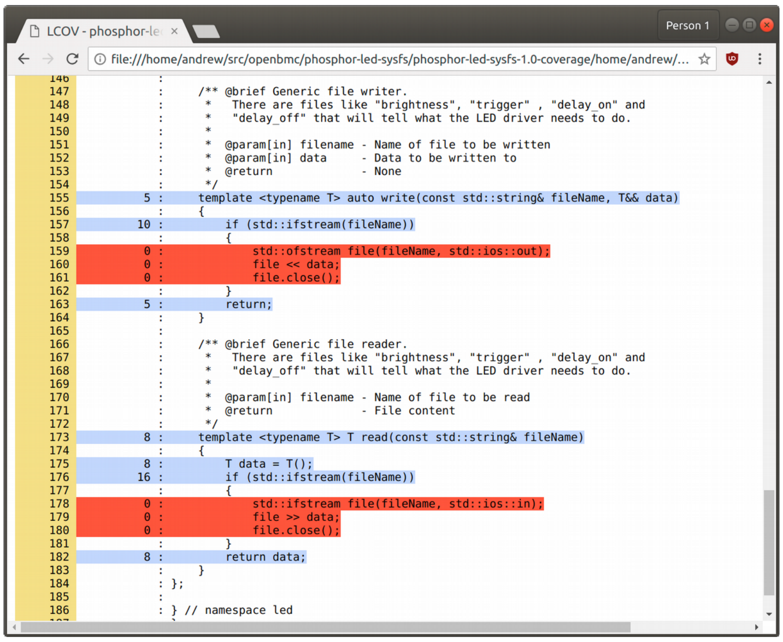
Generated by: LCOV version 1.13

Here we can see we achieved 83.7% line coverage of physical.cpp and 70.0% of physical.hpp

On the other-hand we achieved 100% function coverage in physical.cpp, and got a decent 87.5% of physical.hpp

So what did we miss? We can drill down further into the files themselves to see annotated source indicating which lines we hit and missed

What we missed: physical.hpp

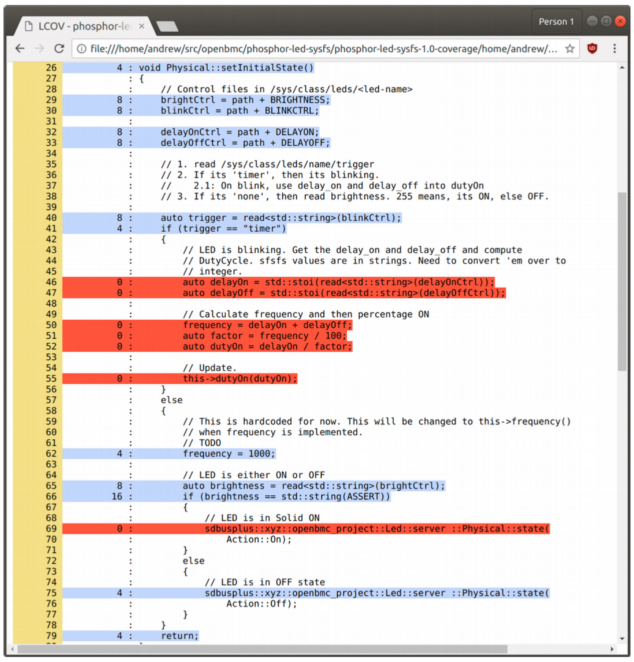


LCOV - phosphor-le
Person 1
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/andrew/...

```
         :
         :     /** @brief Generic file writer.
         :      *   There are files like "brightness", "trigger" , "delay_on" and
         :      *   "delay_off" that will tell what the LED driver needs to do.
         :      *
         :      *  @param[in] filename - Name of file to be written
         :      *  @param[in] data     - Data to be written to
         :      *  @return             - None
         :      */
       5 :     template <typename T> auto write(const std::string& fileName, T&& data)
         :     {
      10 :         if (std::ifstream(fileName))
         :         {
       0 :             std::ofstream file(fileName, std::ios::out);
       0 :             file << data;
       0 :             file.close();
         :         }
       5 :         return;
         :     }
         :
         :     /** @brief Generic file reader.
         :      *   There are files like "brightness", "trigger" , "delay_on" and
         :      *   "delay_off" that will tell what the LED driver needs to do.
         :      *
         :      *  @param[in] filename - Name of file to be read
         :      *  @return             - File content
         :      */
       8 :     template <typename T> T read(const std::string& fileName)
         :     {
       8 :         T data = T();
      16 :         if (std::ifstream(fileName))
         :         {
       0 :             std::ifstream file(fileName, std::ios::in);
       0 :             file >> data;
       0 :             file.close();
         :         }
       8 :         return data;
         :     }
         : };
         :
         : } // namespace led
```

In physical.hpp, we missed the critical chunks of the read() and write() methods. We pointed the tests to a dummy directory in the sysfs tree, and thus the tests in both read() and write() simply return a T instance instantiated with the no-args constructor

What we missed: physical.cpp


LCOV - phosphor-le ×

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/andrew/...

```
26         4 : void Physical::setInitialState()
27           : {
28           :     // Control files in /sys/class/leds/<led-name>
29         8 :     brightCtrl = path + BRIGHTNESS;
30         8 :     blinkCtrl = path + BLINKCTRL;
31           :
32         8 :     delayOnCtrl = path + DELAYON;
33         8 :     delayOffCtrl = path + DELAYOFF;
34           :
35           :     // 1. read /sys/class/leds/name/trigger
36           :     // 2. If its 'timer', then its blinking.
37           :     //    2.1: On blink, use delay_on and delay_off into dutyOn
38           :     // 3. If its 'none', then read brightness. 255 means, its ON, else OFF.
39           :
40         8 :     auto trigger = read<std::string>(blinkCtrl);
41         4 :     if (trigger == "timer")
42           :     {
43           :         // LED is blinking. Get the delay_on and delay_off and compute
44           :         // DutyCycle. sfsfs values are in strings. Need to convert 'em over to
45           :         // integer.
46         0 :         auto delayOn = std::stoi(read<std::string>(delayOnCtrl));
47         0 :         auto delayOff = std::stoi(read<std::string>(delayOffCtrl));
48           :
49           :         // Calculate frequency and then percentage ON
50         0 :         frequency = delayOn + delayOff;
51         0 :         auto factor = frequency / 100;
52         0 :         auto dutyOn = delayOn / factor;
53           :
54           :         // Update.
55         0 :         this->dutyOn(dutyOn);
56           :     }
57           :     else
58           :     {
59           :         // This is hardcoded for now. This will be changed to this->frequency()
60           :         // when frequency is implemented.
61           :         // TODO
62         4 :         frequency = 1000;
63           :
64           :         // LED is either ON or OFF
65         8 :         auto brightness = read<std::string>(brightCtrl);
66        16 :         if (brightness == std::string(ASSERT))
67           :         {
68           :             // LED is in Solid ON
69         0 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
70           :                 Action::On);
71           :         }
72           :         else
73           :         {
74           :             // LED is in OFF state
75         4 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
76           :                 Action::Off);
77           :         }
78           :     }
79         4 :     return;
```

In physical.cpp, we missed lines whose execution depend on reading particular values out of the sysfs LED attribute files

Now, these attributes don't exist as we provided a dummy sysfs LED path to use! The fact that we charged off down the else paths when the attributes we are attempting to read didn't exist should be sounding some alarm bells – failing to read the attribute file should be an error condition that is propagated out as necessary!

Please don't swallow errors this way in your code! Part of the design challenge is understanding who should handle the error. Often it is an actor well removed from where the error occurs. My gut feeling in this case is the application shouldn't be launched if the appropriate attributes are not available in sysfs, which would remove the error handling problem entirely (or at least move it to another layer)

Regardless, what can we do to hit these paths?

Ultimately, we need a way to make the conditions on lines 41 and 66 evaluate to true.

The unhelpful design of Physical

1) read() and write() are non-virtual functions
2) read() is called by the constructor
3) read() and write() are template functions
4) There is a better choice of abstraction

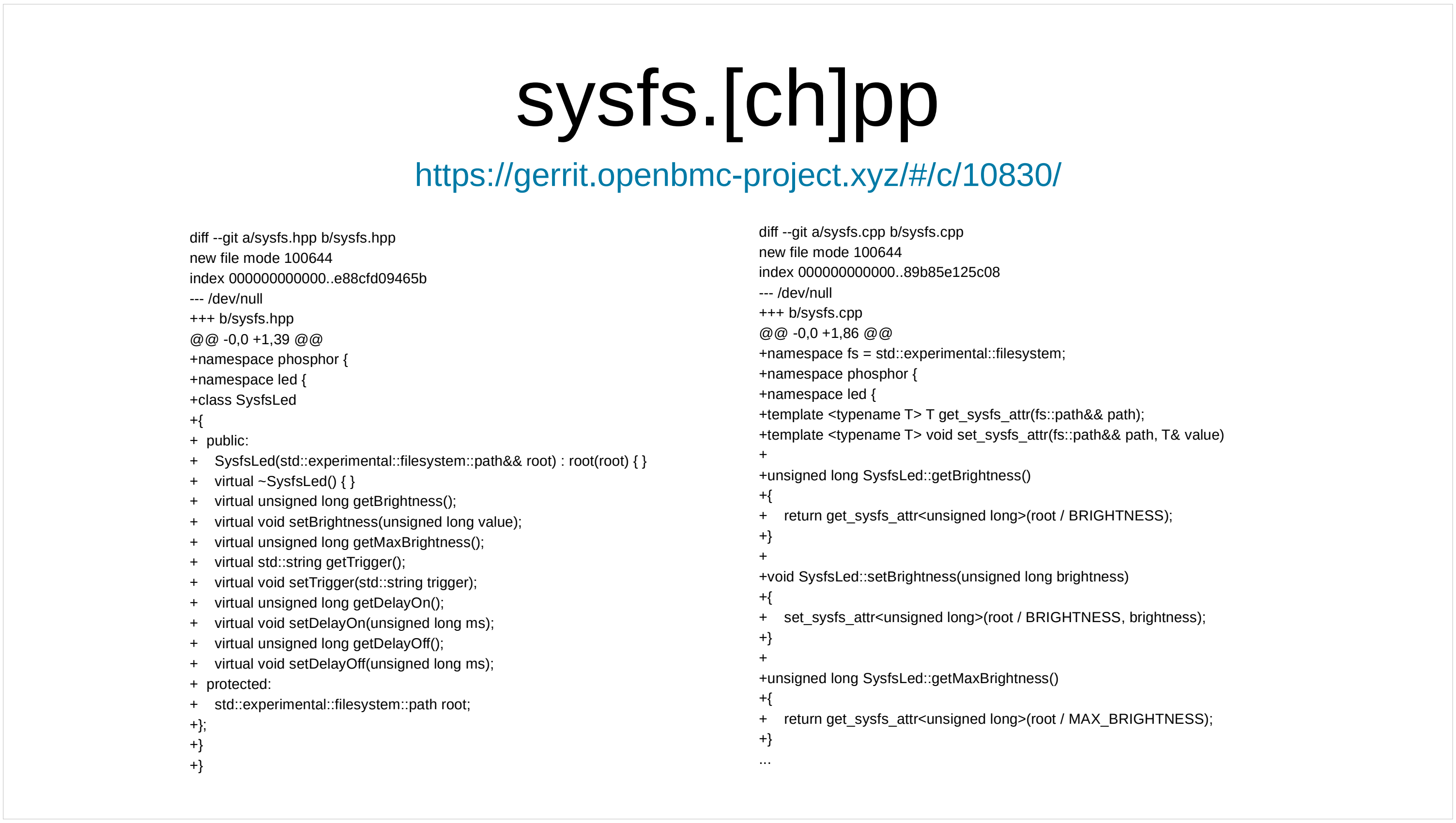
1) One way to hit the lines we missed in physical.cpp would be to override read() and write() to return the expected values using inheritance, and instantiate this decended class in the tests. However, as read() and write() are non-virtual they cannot be overridden, and even if they were virtual,

2) They are executed in the context of Physical’s constructor, and thus the compiler will bind the calls to Physical’s definitions and not a decendent’s. This eliminates virtualising the methods as a solution.

3) read() and write() are template functions, where the template parameter is the type that will be constructed and returned. A search of the code-base showed that the only type parameter used with read() and write() invocations was std::string, negating the need for the template entirely.

4) Given we can’t solve the problem by inheritance due to 1) and 2), and having removed the complication of 3), another possibility is dependency injection, also known as the inversion-of-control design pattern. This pattern moves read() and write() into their own class, and we provide an instance of this new class to Physical’s constructor. To clarify, we’re viewing implementing read() and write() on Physical as an abstraction violation: We’re mixing the business logic with the mechanics of interacting with sysfs.

Lets explore the dependency injection approach of moving read and write out of Physical.



sysfs.[ch]pp

https://gerrit.openbmc-project.xyz/#/c/10830/

```
diff --git a/sysfs.hpp b/sysfs.hpp
new file mode 100644
index 000000000000..e88cfd09465b
--- /dev/null
+++ b/sysfs.hpp
@@ -0,0 +1,39 @@
+namespace phosphor {
+namespace led {
+class SysfsLed
+{
+  public:
+    SysfsLed(std::experimental::filesystem::path&& root) : root(root) { }
+    virtual ~SysfsLed() { }
+    virtual unsigned long getBrightness();
+    virtual void setBrightness(unsigned long value);
+    virtual unsigned long getMaxBrightness();
+    virtual std::string getTrigger();
+    virtual void setTrigger(std::string trigger);
+    virtual unsigned long getDelayOn();
+    virtual void setDelayOn(unsigned long ms);
+    virtual unsigned long getDelayOff();
+    virtual void setDelayOff(unsigned long ms);
+  protected:
+    std::experimental::filesystem::path root;
+};
+}
+}
```

```
diff --git a/sysfs.cpp b/sysfs.cpp
new file mode 100644
index 000000000000..89b85e125c08
--- /dev/null
+++ b/sysfs.cpp
@@ -0,0 +1,86 @@
+namespace fs = std::experimental::filesystem;
+namespace phosphor {
+namespace led {
+template <typename T> T get_sysfs_attr(fs::path&& path);
+template <typename T> void set_sysfs_attr(fs::path&& path, T& value)
+
+unsigned long SysfsLed::getBrightness()
+{
+    return get_sysfs_attr<unsigned long>(root / BRIGHTNESS);
+}
+
+void SysfsLed::setBrightness(unsigned long brightness)
+{
+    set_sysfs_attr<unsigned long>(root / BRIGHTNESS, brightness);
+}
+
+unsigned long SysfsLed::getMaxBrightness()
+{
+    return get_sysfs_attr<unsigned long>(root / MAX_BRIGHTNESS);
+}
...
```

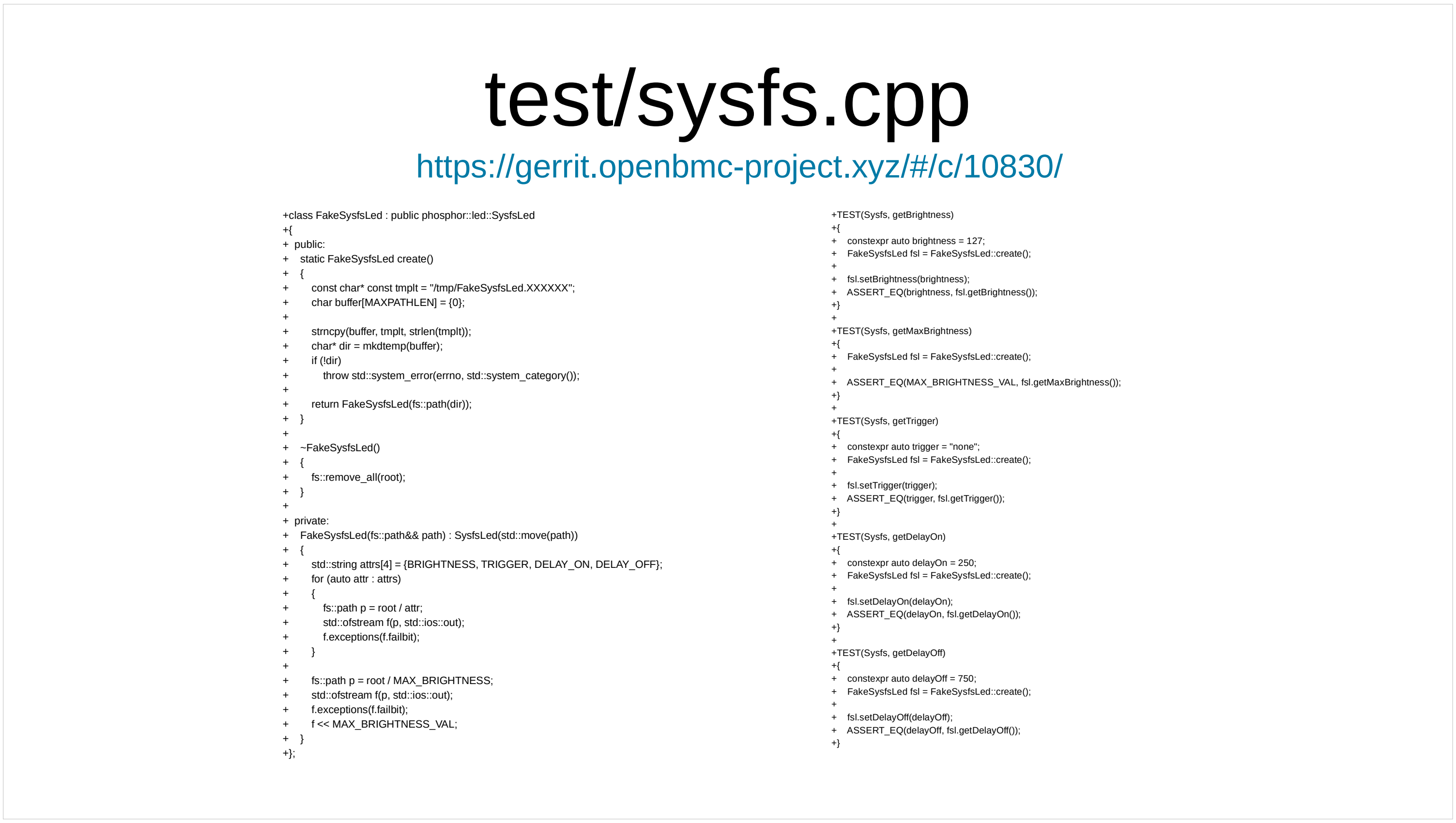
To separate the functionality, we introduce a new SysfsLed class in sysfs.[ch]pp

Its design is to expose each of the sysfs LED attributes via virtual getter and setter methods, seen here on the left.

The functionality for reading and writing the attribute files is wrapped up in two template function definitions which are called from the class’ method implementations seen on the right. This retains the brevity whilst hiding the templates from the class API.

The advantage of virtual getters and setters is we can make a sock-puppet out of the interface for testing purposes.

This will allow us to guide test execution through the paths we failed to hit with the existing test cases.



test/sysfs.cpp

https://gerrit.openbmc-project.xyz/#/c/10830/

```
+class FakeSysfsLed : public phosphor::led::SysfsLed
+{
+  public:
+    static FakeSysfsLed create()
+    {
+        const char* const tmplt = "/tmp/FakeSysfsLed.XXXXXX";
+        char buffer[MAXPATHLEN] = {0};
+
+        strncpy(buffer, tmplt, strlen(tmplt));
+        char* dir = mkdtemp(buffer);
+        if (!dir)
+            throw std::system_error(errno, std::system_category());
+
+        return FakeSysfsLed(fs::path(dir));
+    }
+
+    ~FakeSysfsLed()
+    {
+        fs::remove_all(root);
+    }
+
+  private:
+    FakeSysfsLed(fs::path&& path) : SysfsLed(std::move(path))
+    {
+        std::string attrs[4] = {BRIGHTNESS, TRIGGER, DELAY_ON, DELAY_OFF};
+        for (auto attr : attrs)
+        {
+            fs::path p = root / attr;
+            std::ofstream f(p, std::ios::out);
+            f.exceptions(f.failbit);
+        }
+
+        fs::path p = root / MAX_BRIGHTNESS;
+        std::ofstream f(p, std::ios::out);
+        f.exceptions(f.failbit);
+        f << MAX_BRIGHTNESS_VAL;
+    }
+};
```

```
+TEST(Sysfs, getBrightness)
+{
+    constexpr auto brightness = 127;
+    FakeSysfsLed fsl = FakeSysfsLed::create();
+
+    fsl.setBrightness(brightness);
+    ASSERT_EQ(brightness, fsl.getBrightness());
+}
+
+TEST(Sysfs, getMaxBrightness)
+{
+    FakeSysfsLed fsl = FakeSysfsLed::create();
+
+    ASSERT_EQ(MAX_BRIGHTNESS_VAL, fsl.getMaxBrightness());
+}
+
+TEST(Sysfs, getTrigger)
+{
+    constexpr auto trigger = "none";
+    FakeSysfsLed fsl = FakeSysfsLed::create();
+
+    fsl.setTrigger(trigger);
+    ASSERT_EQ(trigger, fsl.getTrigger());
+}
+
+TEST(Sysfs, getDelayOn)
+{
+    constexpr auto delayOn = 250;
+    FakeSysfsLed fsl = FakeSysfsLed::create();
+
+    fsl.setDelayOn(delayOn);
+    ASSERT_EQ(delayOn, fsl.getDelayOn());
+}
+
+TEST(Sysfs, getDelayOff)
+{
+    constexpr auto delayOff = 750;
+    FakeSysfsLed fsl = FakeSysfsLed::create();
+
+    fsl.setDelayOff(delayOff);
+    ASSERT_EQ(delayOff, fsl.getDelayOff());
+}
```

Of course, if we're adding new code we need to test it, so here's the implementation of the SysfsLed test suite

The nature of SysfsLed is to read and write files on the filesystem. Ideally we wouldn't touch the filesystem as then the tests have a dependency on an external resource that is another point of failure. This drives to the heart of "integration" vs "unit" testing. Unit testing is considered ideal because by definition it's required that all external points of failure are removed such that the only entity that is exercised is the code under test.

Depending on the filesystem makes this an integration test.

Further, we don't want to actually manipulate system LEDs whilst we're testing the code (that makes us dependent on the system configuration, and would be confusing for system owners), and we also want the tests to fail if necessary.

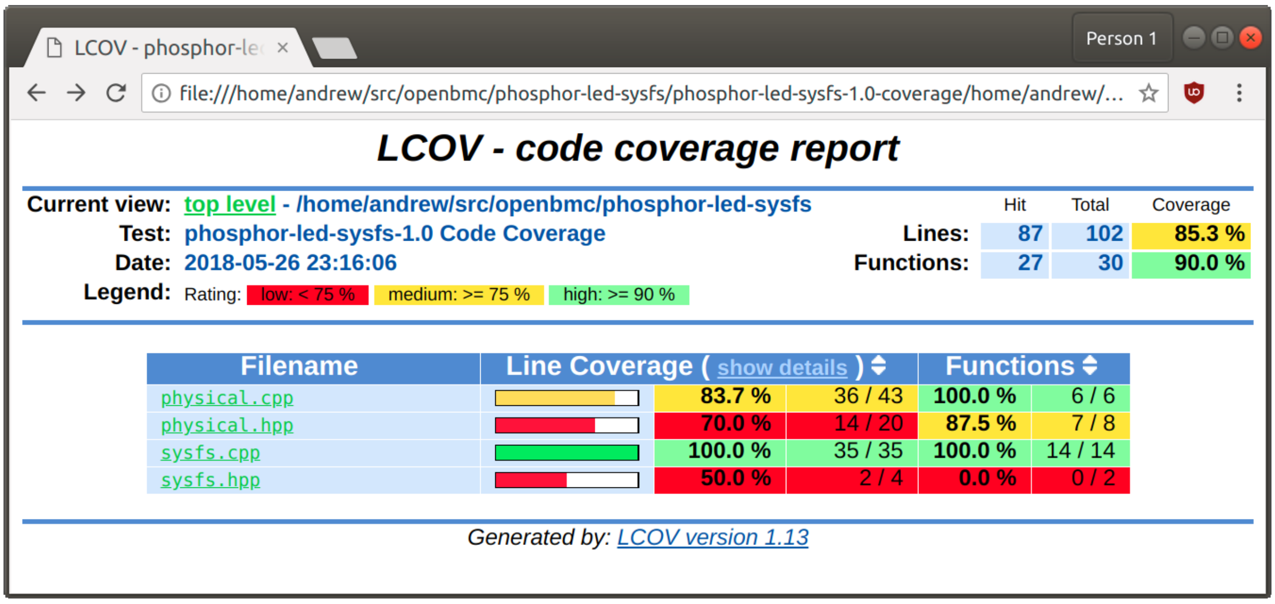
Instead, we isolate each test case to its own temporary directory. This has two useful properties:

1) The tests can read or write whatever files are necessary without affecting system behaviour or being influenced by system configuration
2) The tests can be run in parallel because each test case receives its own isolated temporary directory

This means that, short of filesystem failure, the tests will run correctly and quickly if we have available CPUs.

The tests themselves simply write the value through the setter and read it back through the getter, then assert that the read value is equal to the written value.

The third result



LCOV - phosphor-le

Person 1

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/andrew/...

LCOV - code coverage report

Current view: top level - /home/andrew/src/openbmc/phosphor-led-sysfs
Test: phosphor-led-sysfs-1.0 Code Coverage
Date: 2018-05-26 23:16:06
Legend: Rating: low: < 75 % medium: >= 75 % high: >= 90 %

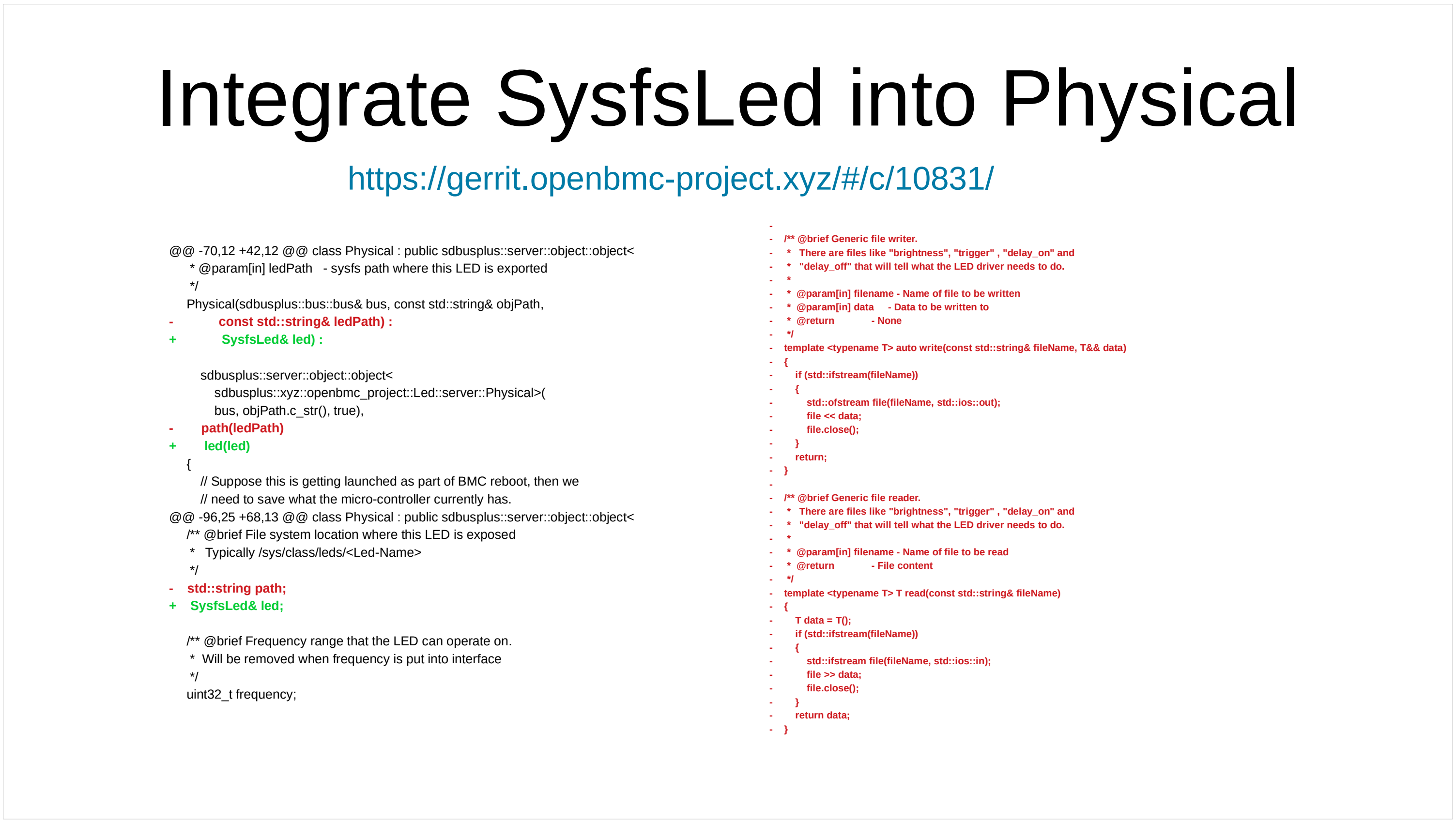
	Hit	Total	Coverage
Lines:	87	102	85.3 %
Functions:	27	30	90.0 %

Filename	Line Coverage (show details)			Functions	
physical.cpp		83.7 %	36 / 43	100.0 %	6 / 6
physical.hpp		70.0 %	14 / 20	87.5 %	7 / 8
sysfs.cpp		100.0 %	35 / 35	100.0 %	14 / 14
sysfs.hpp		50.0 %	2 / 4	0.0 %	0 / 2

Generated by: LCOV version 1.13

Running `make check-code-coverage` to understand how well our new class works, we see we’ve achieved 100% coverage of sysfs.cpp with our tests (it turns out the 50% coverage of sysfs.hpp is spurious).

Now that we’ve got the SysfsLed class implemented we need to integrate it back into the Physical class.



Integrate SysfsLed into Physical

https://gerrit.openbmc-project.xyz/#/c/10831/

```
@@ -70,12 +42,12 @@ class Physical : public sdbusplus::server::object::object<
     * @param[in] ledPath   - sysfs path where this LED is exported
     */
    Physical(sdbusplus::bus::bus& bus, const std::string& objPath,
-            const std::string& ledPath) :
+             SysfsLed& led) :

        sdbusplus::server::object::object<
            sdbusplus::xyz::openbmc_project::Led::server::Physical>(
            bus, objPath.c_str(), true),
-       path(ledPath)
+        led(led)
    {
        // Suppose this is getting launched as part of BMC reboot, then we
        // need to save what the micro-controller currently has.
@@ -96,25 +68,13 @@ class Physical : public sdbusplus::server::object::object<
    /** @brief File system location where this LED is exposed
     *   Typically /sys/class/leds/<Led-Name>
     */
-   std::string path;
+    SysfsLed& led;

    /** @brief Frequency range that the LED can operate on.
     *  Will be removed when frequency is put into interface
     */
    uint32_t frequency;
```

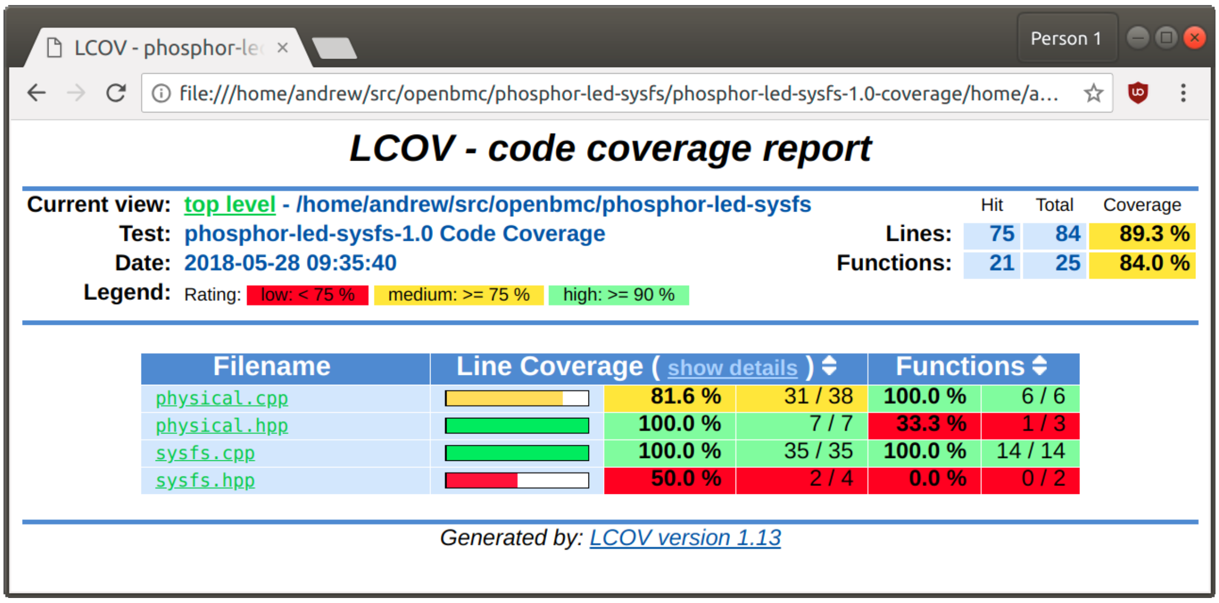
```
-
-   /** @brief Generic file writer.
-    *   There are files like "brightness", "trigger" , "delay_on" and
-    *   "delay_off" that will tell what the LED driver needs to do.
-    *
-    *  @param[in] filename - Name of file to be written
-    *  @param[in] data     - Data to be written to
-    *  @return            - None
-    */
-   template <typename T> auto write(const std::string& fileName, T&& data)
-   {
-       if (std::ifstream(fileName))
-       {
-           std::ofstream file(fileName, std::ios::out);
-           file << data;
-           file.close();
-       }
-       return;
-   }
-
-   /** @brief Generic file reader.
-    *   There are files like "brightness", "trigger" , "delay_on" and
-    *   "delay_off" that will tell what the LED driver needs to do.
-    *
-    *  @param[in] filename - Name of file to be read
-    *  @return            - File content
-    */
-   template <typename T> T read(const std::string& fileName)
-   {
-       T data = T();
-       if (std::ifstream(fileName))
-       {
-           std::ifstream file(fileName, std::ios::in);
-           file >> data;
-           file.close();
-       }
-       return data;
-   }
```

Once we integrate SysfsLed we can remove the read() and write() implementations from Physical, reducing the number of lines and thus increasing the line coverage percentage.

The core of the integration work is quite straight-forward, limited to the three lines on the left column, though there is some fallout around the code-base.

There’s one trick to the integration which is we take a reference to the SysfsLed for Physical’s constructor, and stash the reference in the class member. This way when we extend SysfsLed with mock classes we dispatch to the right method implementations. However, constructors of Physical instances must ensure that the provided SysfsLed outlines the Physical instance.

The fourth result



LCOV - phosphor-le

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a…

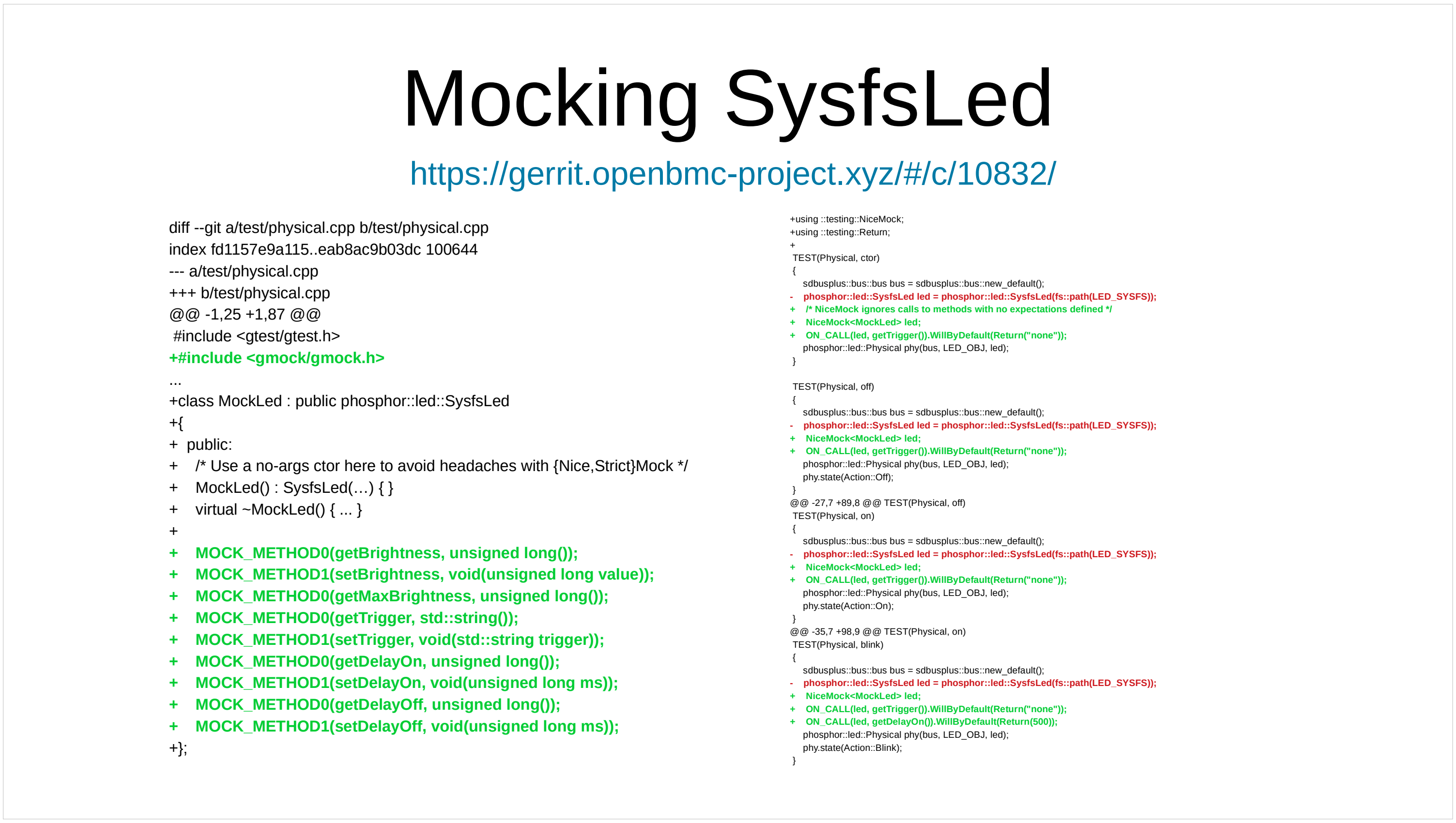
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/andrew/src/openbmc/phosphor-led-sysfs				
Test:	phosphor-led-sysfs-1.0 Code Coverage	**Lines:**	75	84	89.3 %
Date:	2018-05-28 09:35:40	**Functions:**	21	25	84.0 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

Filename	Line Coverage (show details)			Functions	
physical.cpp		81.6 %	31 / 38	100.0 %	6 / 6
physical.hpp		100.0 %	7 / 7	33.3 %	1 / 3
sysfs.cpp		100.0 %	35 / 35	100.0 %	14 / 14
sysfs.hpp		50.0 %	2 / 4	0.0 %	0 / 2

Generated by: LCOV version 1.13

Removing read() and write() gives us with 100% coverage of physical.hpp, though the physical.cpp coverage drops slightly in terms of percentage.



Mocking SysfsLed

https://gerrit.openbmc-project.xyz/#/c/10832/

```
diff --git a/test/physical.cpp b/test/physical.cpp
index fd1157e9a115..eab8ac9b03dc 100644
--- a/test/physical.cpp
+++ b/test/physical.cpp
@@ -1,25 +1,87 @@
 #include <gtest/gtest.h>
+#include <gmock/gmock.h>
...
+class MockLed : public phosphor::led::SysfsLed
+{
+  public:
+    /* Use a no-args ctor here to avoid headaches with {Nice,Strict}Mock */
+    MockLed() : SysfsLed(…) { }
+    virtual ~MockLed() { ... }
+
+    MOCK_METHOD0(getBrightness, unsigned long());
+    MOCK_METHOD1(setBrightness, void(unsigned long value));
+    MOCK_METHOD0(getMaxBrightness, unsigned long());
+    MOCK_METHOD0(getTrigger, std::string());
+    MOCK_METHOD1(setTrigger, void(std::string trigger));
+    MOCK_METHOD0(getDelayOn, unsigned long());
+    MOCK_METHOD1(setDelayOn, void(unsigned long ms));
+    MOCK_METHOD0(getDelayOff, unsigned long());
+    MOCK_METHOD1(setDelayOff, void(unsigned long ms));
+};
```

```
+using ::testing::NiceMock;
+using ::testing::Return;
+
 TEST(Physical, ctor)
 {
     sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
-    phosphor::led::SysfsLed led = phosphor::led::SysfsLed(fs::path(LED_SYSFS));
+    /* NiceMock ignores calls to methods with no expectations defined */
+    NiceMock<MockLed> led;
+    ON_CALL(led, getTrigger()).WillByDefault(Return("none"));
     phosphor::led::Physical phy(bus, LED_OBJ, led);
 }

 TEST(Physical, off)
 {
     sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
-    phosphor::led::SysfsLed led = phosphor::led::SysfsLed(fs::path(LED_SYSFS));
+    NiceMock<MockLed> led;
+    ON_CALL(led, getTrigger()).WillByDefault(Return("none"));
     phosphor::led::Physical phy(bus, LED_OBJ, led);
     phy.state(Action::Off);
 }
@@ -27,7 +89,8 @@ TEST(Physical, off)
 TEST(Physical, on)
 {
     sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
-    phosphor::led::SysfsLed led = phosphor::led::SysfsLed(fs::path(LED_SYSFS));
+    NiceMock<MockLed> led;
+    ON_CALL(led, getTrigger()).WillByDefault(Return("none"));
     phosphor::led::Physical phy(bus, LED_OBJ, led);
     phy.state(Action::On);
 }
@@ -35,7 +98,9 @@ TEST(Physical, on)
 TEST(Physical, blink)
 {
     sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
-    phosphor::led::SysfsLed led = phosphor::led::SysfsLed(fs::path(LED_SYSFS));
+    NiceMock<MockLed> led;
+    ON_CALL(led, getTrigger()).WillByDefault(Return("none"));
+    ON_CALL(led, getDelayOn()).WillByDefault(Return(500));
     phosphor::led::Physical phy(bus, LED_OBJ, led);
     phy.state(Action::Blink);
 }
```

Now that we’ve cleaned up the Physical implementation we can begin shifting the needle on our tests away from integration- and towards unit-style tests.

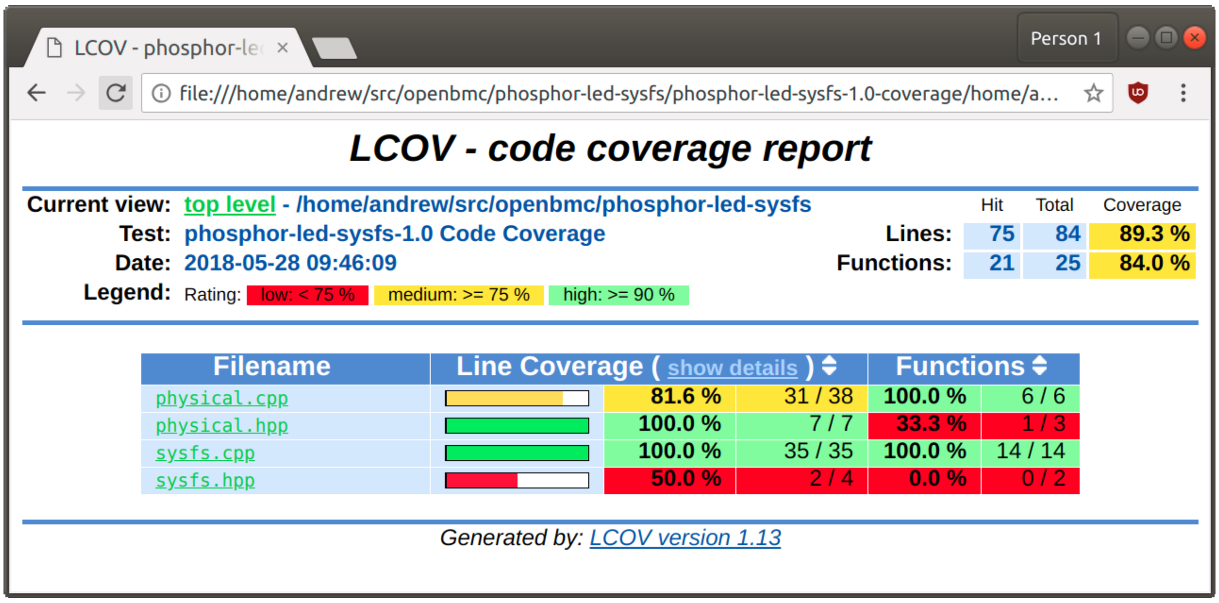
We still depend on sdbusplus to provide a bus instance, which means we won’t be achieving the goal of full independence from the system environment.

The left diff hunk in the slide defines the mock class using the macros provided by Google Mock, and the hunks on the right set about instantiating the mock class and defining behaviours with the ON_CALL() macro.

Specifically, we’re making the mock class return “none” on calls to getTrigger() in each test case, and 500 on calls to getDelayOn() in the last test case.

Note that we encapsulate our mock class in a nifty template class NiceMock, that stubs out all the mocked methods so calls are ignored if there is no explicit mock implementation defined. The alternative StrictMock will error on calls to methods not explicitly mocked.

The fifth result



LCOV - phosphor-le

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a...

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/andrew/src/openbmc/phosphor-led-sysfs				
Test:	phosphor-led-sysfs-1.0 Code Coverage	**Lines:**	75	84	89.3 %
Date:	2018-05-28 09:46:09	**Functions:**	21	25	84.0 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

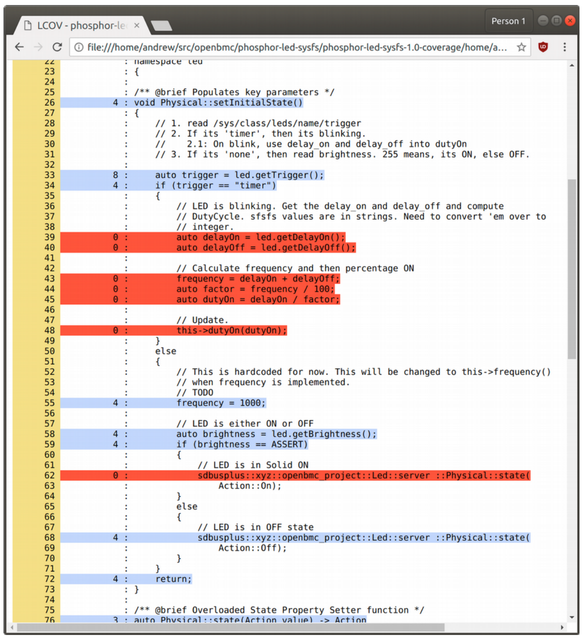
Filename	Line Coverage (show details)			Functions	
physical.cpp		81.6 %	31 / 38	100.0 %	6 / 6
physical.hpp		100.0 %	7 / 7	33.3 %	1 / 3
sysfs.cpp		100.0 %	35 / 35	100.0 %	14 / 14
sysfs.hpp		50.0 %	2 / 4	0.0 %	0 / 2

Generated by: LCOV version 1.13

Generating the code coverage report shows what we should expect from switching to a SysfsLed mock in the test cases: There has been no change in the code coverage metrics

With the mock equivalents of our original tests in place, we can set about putting them to good use by writing tests to hit the code paths that were not covered by our initial test case implementations.

The un-hit paths in Physical.cpp



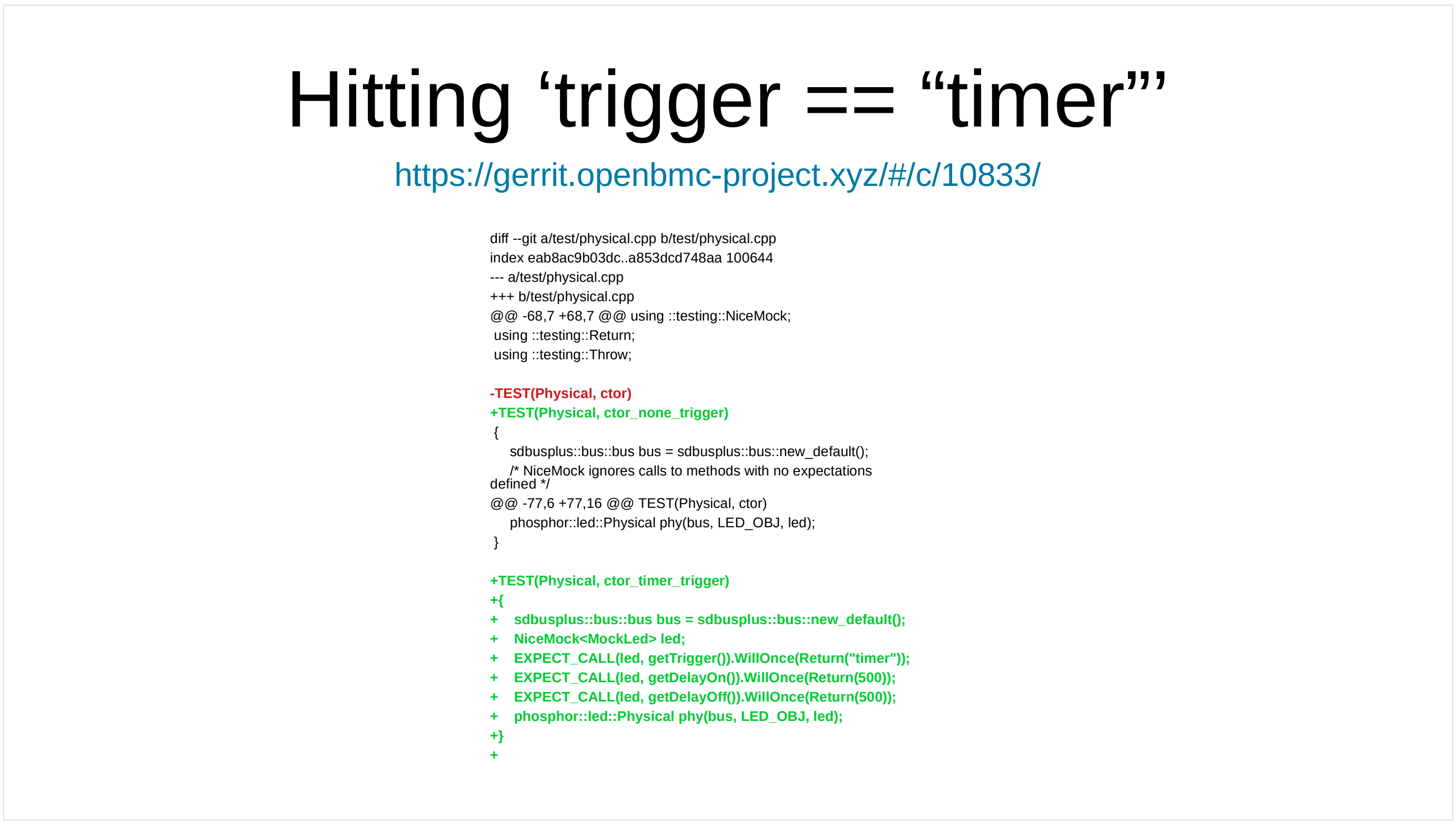
LCOV - phosphor-le
Person 1
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a...

```
22         : namespace led
23         : {
24         :
25         : /** @brief Populates key parameters */
26       4 : void Physical::setInitialState()
27         : {
28         :     // 1. read /sys/class/leds/name/trigger
29         :     // 2. If its 'timer', then its blinking.
30         :     //    2.1: On blink, use delay_on and delay_off into dutyOn
31         :     // 3. If its 'none', then read brightness. 255 means, its ON, else OFF.
32         :
33       8 :     auto trigger = led.getTrigger();
34       4 :     if (trigger == "timer")
35         :     {
36         :         // LED is blinking. Get the delay_on and delay_off and compute
37         :         // DutyCycle. sfsfs values are in strings. Need to convert 'em over to
38         :         // integer.
39       0 :         auto delayOn = led.getDelayOn();
40       0 :         auto delayOff = led.getDelayOff();
41         :
42         :         // Calculate frequency and then percentage ON
43       0 :         frequency = delayOn + delayOff;
44       0 :         auto factor = frequency / 100;
45       0 :         auto dutyOn = delayOn / factor;
46         :
47         :         // Update.
48       0 :         this->dutyOn(dutyOn);
49         :     }
50         :     else
51         :     {
52         :         // This is hardcoded for now. This will be changed to this->frequency()
53         :         // when frequency is implemented.
54         :         // TODO
55       4 :         frequency = 1000;
56         :
57         :         // LED is either ON or OFF
58       4 :         auto brightness = led.getBrightness();
59       4 :         if (brightness == ASSERT)
60         :         {
61         :             // LED is in Solid ON
62       0 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
63         :                 Action::On);
64         :         }
65         :         else
66         :         {
67         :             // LED is in OFF state
68       4 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
69         :                 Action::Off);
70         :         }
71         :     }
72       4 :     return;
73         : }
74         :
75         : /** @brief Overloaded State Property Setter function */
76       3 : auto Physical::state(Action value) -> Action
```

Going back to our earlier code coverage snapshot, we found that two code paths were not tested:

1) Where ‘trigger == “timer”’, from lines 36-48
2) Where ‘trigger != “timer”’ and ‘brightness == ASSERT’, from lines 61-63

Using our mock class we can now configure instances that allow us to test both of these code paths. We’ll address path 1) first.



Hitting ‘trigger == “timer”’

https://gerrit.openbmc-project.xyz/#/c/10833/

```
diff --git a/test/physical.cpp b/test/physical.cpp
index eab8ac9b03dc..a853dcd748aa 100644
--- a/test/physical.cpp
+++ b/test/physical.cpp
@@ -68,7 +68,7 @@ using ::testing::NiceMock;
 using ::testing::Return;
 using ::testing::Throw;

-TEST(Physical, ctor)
+TEST(Physical, ctor_none_trigger)
 {
     sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
     /* NiceMock ignores calls to methods with no expectations
defined */
@@ -77,6 +77,16 @@ TEST(Physical, ctor)
     phosphor::led::Physical phy(bus, LED_OBJ, led);
 }

+TEST(Physical, ctor_timer_trigger)
+{
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    NiceMock<MockLed> led;
+    EXPECT_CALL(led, getTrigger()).WillOnce(Return("timer"));
+    EXPECT_CALL(led, getDelayOn()).WillOnce(Return(500));
+    EXPECT_CALL(led, getDelayOff()).WillOnce(Return(500));
+    phosphor::led::Physical phy(bus, LED_OBJ, led);
+}
+
```

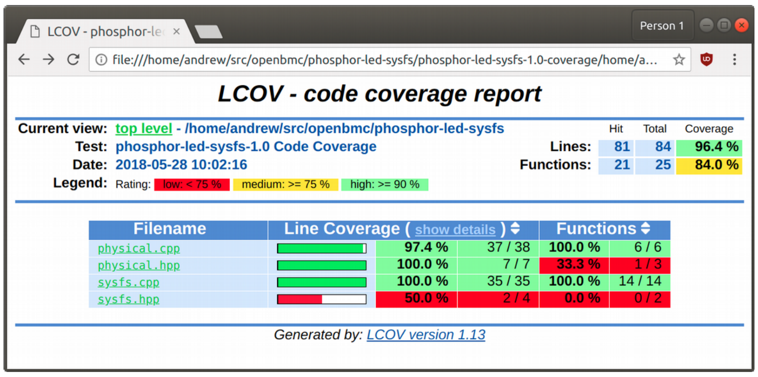
Testing path 1) from the previous slide is a matter of defining three expectations on our mock object

Google Mock tests the expectations at the end of the test case to ensure that our expectations have matched reality – it’s enough to simply define the behaviour we want to see.

We define getTrigger() to return “timer” as desired by the branch that we missed in previous testing, and then mock out the getDelayOn() and getDelayOff() to provide sensible values. Finally we instantiate the object.

With the new test in place, we can confirm we’ve hit the previously untested path by invoking `make check-code-coverage`

The sixth result



LCOV - phosphor-le ×

Person 1

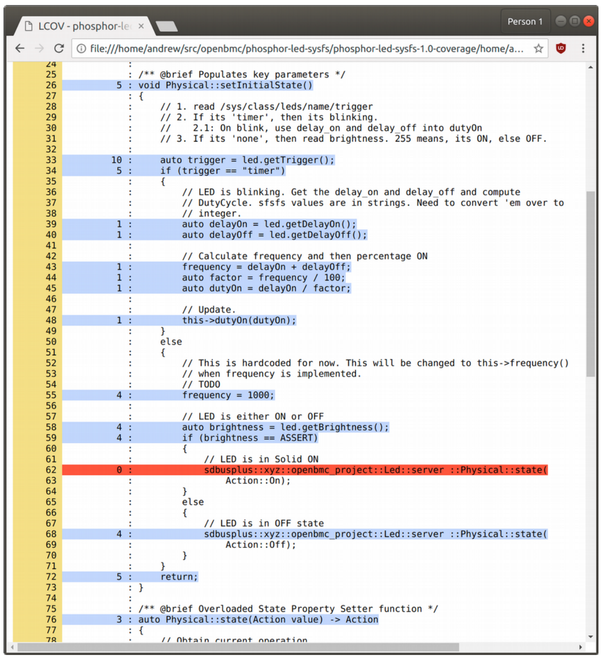
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a…

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/andrew/src/openbmc/phosphor-led-sysfs				
Test:	phosphor-led-sysfs-1.0 Code Coverage	**Lines:**	81	84	96.4 %
Date:	2018-05-28 10:02:16	**Functions:**	21	25	84.0 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

Filename	Line Coverage (show details)			Functions	
physical.cpp	🟩	97.4 %	37 / 38	100.0 %	6 / 6
physical.hpp	🟩	100.0 %	7 / 7	33.3 %	1 / 3
sysfs.cpp	🟩	100.0 %	35 / 35	100.0 %	14 / 14
sysfs.hpp	🟥	50.0 %	2 / 4	0.0 %	0 / 2

Generated by: LCOV version 1.13



LCOV - phosphor-le ×

Person 1

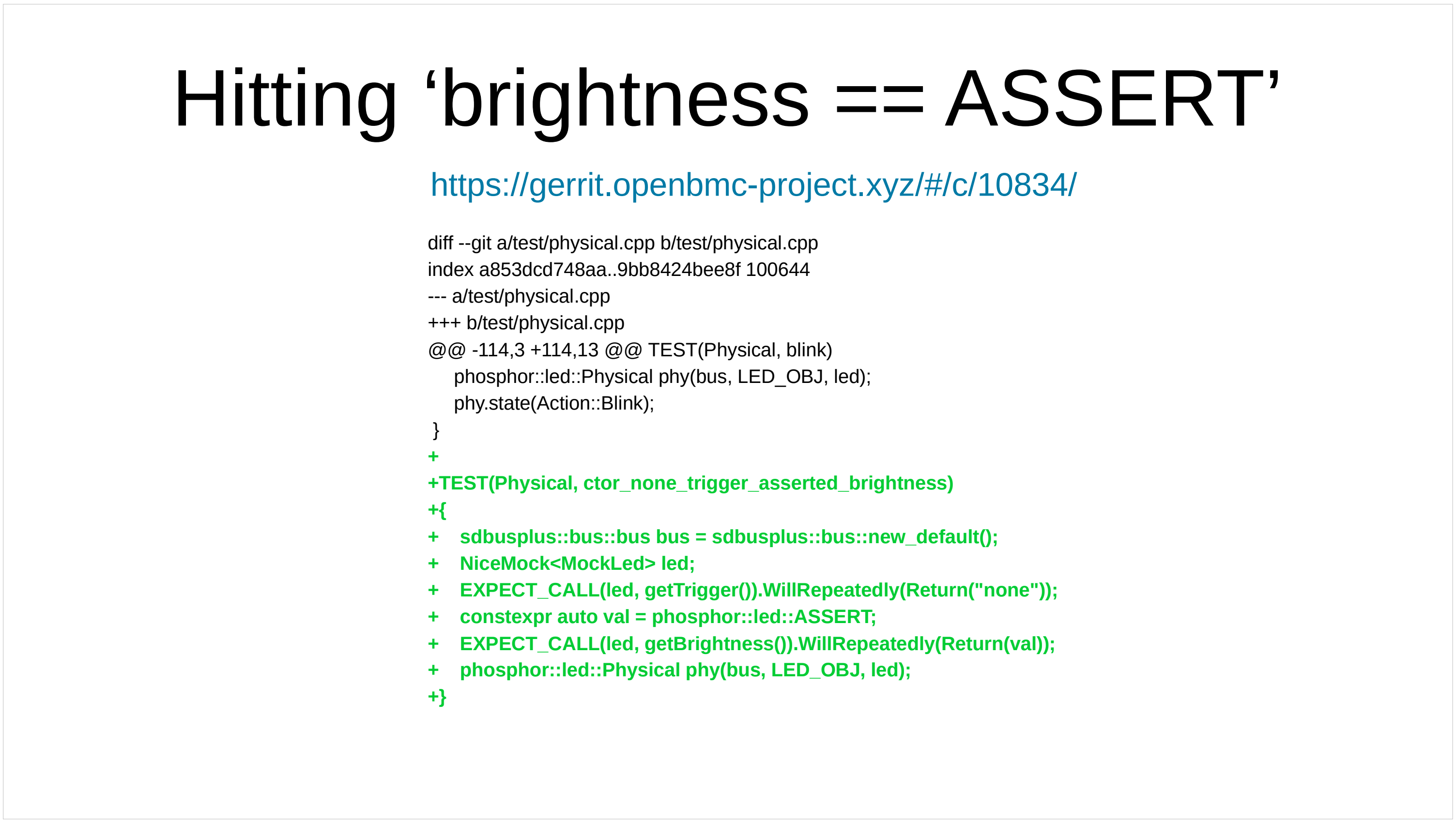
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a…

```
      24            :
      25            : /** @brief Populates key parameters */
      26          5 : void Physical::setInitialState()
      27            : {
      28            :     // 1. read /sys/class/leds/name/trigger
      29            :     // 2. If its 'timer', then its blinking.
      30            :     //    2.1: On blink, use delay_on and delay_off into dutyOn
      31            :     // 3. If its 'none', then read brightness. 255 means, its ON, else OFF.
      32            :
      33         10 :     auto trigger = led.getTrigger();
      34          5 :     if (trigger == "timer")
      35            :     {
      36            :         // LED is blinking. Get the delay_on and delay_off and compute
      37            :         // DutyCycle. sfsfs values are in strings. Need to convert 'em over to
      38            :         // integer.
      39          1 :         auto delayOn = led.getDelayOn();
      40          1 :         auto delayOff = led.getDelayOff();
      41            :
      42            :         // Calculate frequency and then percentage ON
      43          1 :         frequency = delayOn + delayOff;
      44          1 :         auto factor = frequency / 100;
      45          1 :         auto dutyOn = delayOn / factor;
      46            :
      47            :         // Update.
      48          1 :         this->dutyOn(dutyOn);
      49            :     }
      50            :     else
      51            :     {
      52            :         // This is hardcoded for now. This will be changed to this->frequency()
      53            :         // when frequency is implemented.
      54            :         // TODO
      55          4 :         frequency = 1000;
      56            :
      57            :         // LED is either ON or OFF
      58          4 :         auto brightness = led.getBrightness();
      59          4 :         if (brightness == ASSERT)
      60            :         {
      61            :             // LED is in Solid ON
      62          0 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
      63            :                 Action::On);
      64            :         }
      65            :         else
      66            :         {
      67            :             // LED is in OFF state
      68          4 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
      69            :                 Action::Off);
      70            :         }
      71            :     }
      72          5 :     return;
      73            : }
      74            :
      75            : /** @brief Overloaded State Property Setter function */
      76          3 : auto Physical::state(Action value) -> Action
      77            : {
      78            :     // Obtain current operation
```

The result confirms our mocking is doing the trick!

The left screenshot shows increased code coverage while the right screenshot shows we're now hitting lines 36-48.

Now we need to cover case 2). The mechanics of this are very similar to dealing with 1)



Hitting ‘brightness == ASSERT’

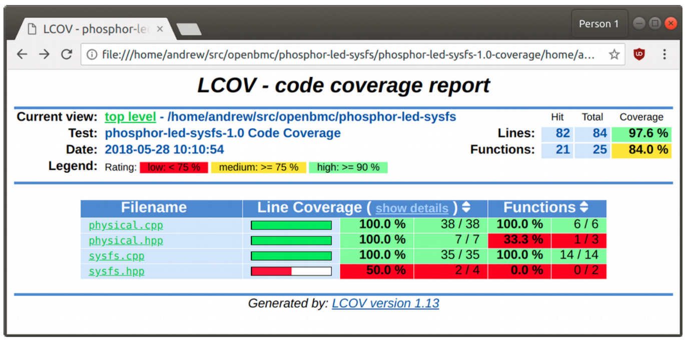
https://gerrit.openbmc-project.xyz/#/c/10834/

```
diff --git a/test/physical.cpp b/test/physical.cpp
index a853dcd748aa..9bb8424bee8f 100644
--- a/test/physical.cpp
+++ b/test/physical.cpp
@@ -114,3 +114,13 @@ TEST(Physical, blink)
     phosphor::led::Physical phy(bus, LED_OBJ, led);
     phy.state(Action::Blink);
 }
+
+TEST(Physical, ctor_none_trigger_asserted_brightness)
+{
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    NiceMock<MockLed> led;
+    EXPECT_CALL(led, getTrigger()).WillRepeatedly(Return("none"));
+    constexpr auto val = phosphor::led::ASSERT;
+    EXPECT_CALL(led, getBrightness()).WillRepeatedly(Return(val));
+    phosphor::led::Physical phy(bus, LED_OBJ, led);
+}
```

There are a few minor differences to the previous test case – here we don’t want to hit the “timer” trigger path, so we mock getTrigger() to return “none”. Then we want to meet the condition ‘brightness == ASSERT’, so we mock getBrightness() to return the value ASSERT.

Again we can instrument with `make check-code-coverage` to test whether our mocking has done the trick.

The seventh result



LCOV - phosphor-le

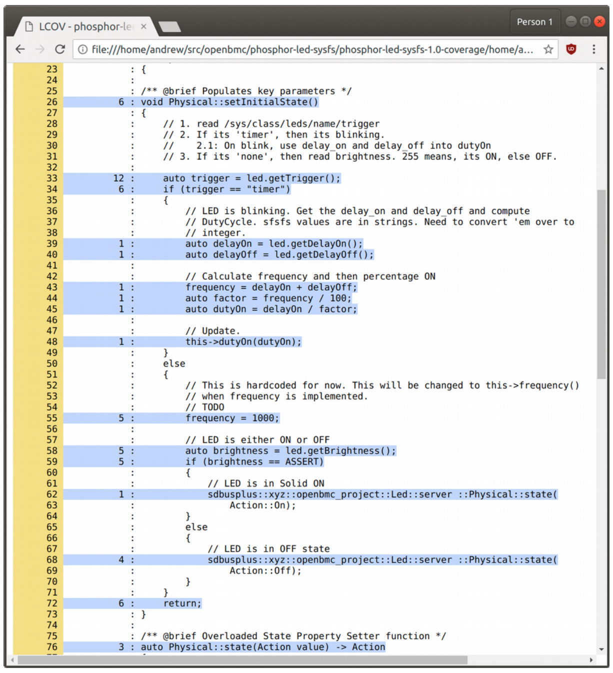
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a...

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/andrew/src/openbmc/phosphor-led-sysfs				
Test:	phosphor-led-sysfs-1.0 Code Coverage	**Lines:**	82	84	97.6 %
Date:	2018-05-28 10:10:54	**Functions:**	21	25	84.0 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

Filename	Line Coverage (show details)			Functions	
physical.cpp	🟩	100.0 %	38 / 38	100.0 %	6 / 6
physical.hpp	🟩	100.0 %	7 / 7	33.3 %	1 / 3
sysfs.cpp	🟩	100.0 %	35 / 35	100.0 %	14 / 14
sysfs.hpp	🟥	50.0 %	2 / 4	0.0 %	0 / 2

Generated by: LCOV version 1.13

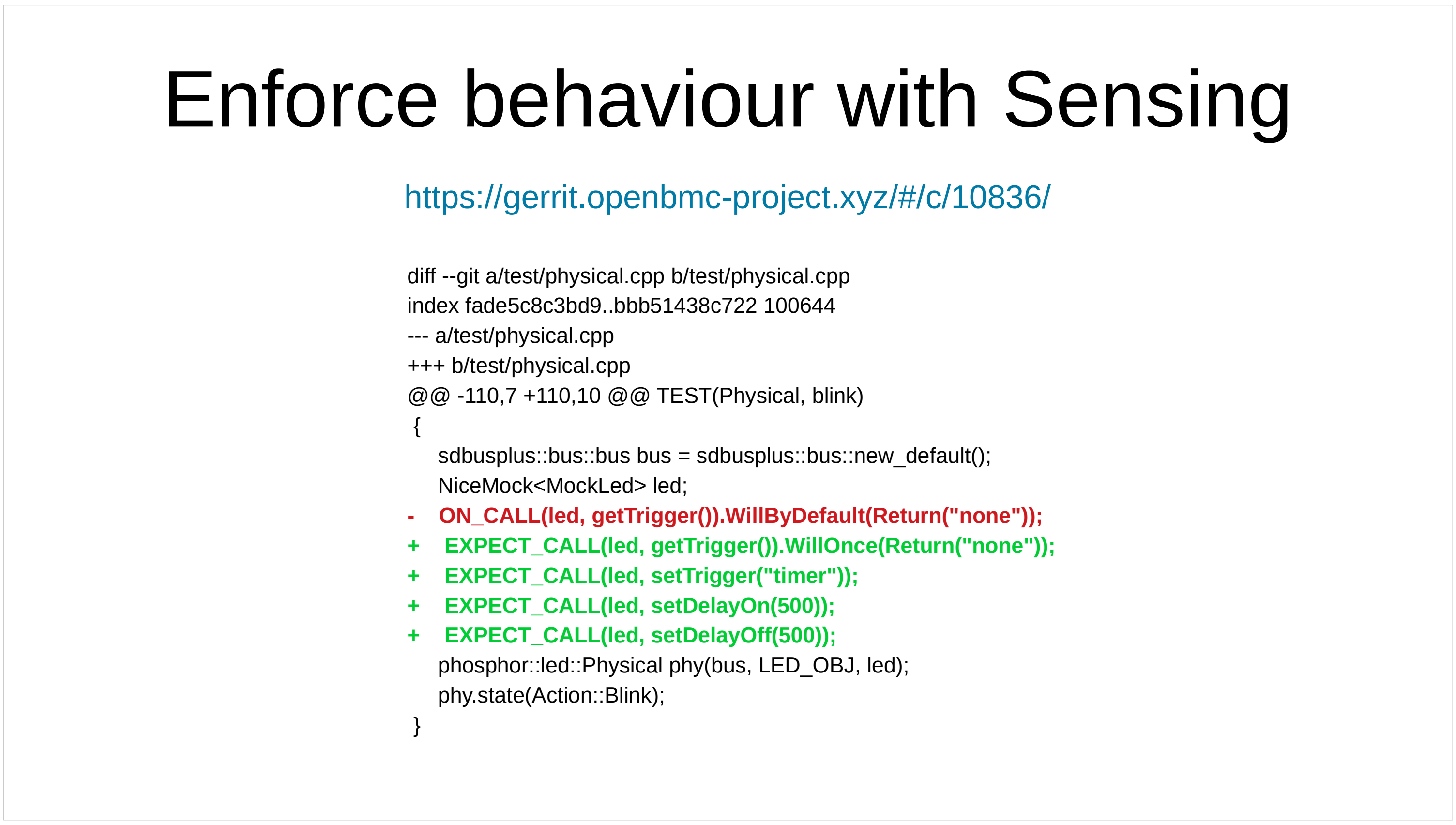


LCOV - phosphor-le

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a...

```
   23            : {
   24            :
   25            : /** @brief Populates key parameters */
   26          6 : void Physical::setInitialState()
   27            : {
   28            :     // 1. read /sys/class/leds/name/trigger
   29            :     // 2. If its 'timer', then its blinking.
   30            :     //    2.1: On blink, use delay_on and delay_off into dutyOn
   31            :     // 3. If its 'none', then read brightness. 255 means, its ON, else OFF.
   32            :
   33         12 :     auto trigger = led.getTrigger();
   34          6 :     if (trigger == "timer")
   35            :     {
   36            :         // LED is blinking. Get the delay_on and delay_off and compute
   37            :         // DutyCycle. sfsfs values are in strings. Need to convert 'em over to
   38            :         // integer.
   39          1 :         auto delayOn = led.getDelayOn();
   40          1 :         auto delayOff = led.getDelayOff();
   41            :
   42            :         // Calculate frequency and then percentage ON
   43          1 :         frequency = delayOn + delayOff;
   44          1 :         auto factor = frequency / 100;
   45          1 :         auto dutyOn = delayOn / factor;
   46            :
   47            :         // Update.
   48          1 :         this->dutyOn(dutyOn);
   49            :     }
   50            :     else
   51            :     {
   52            :         // This is hardcoded for now. This will be changed to this->frequency()
   53            :         // when frequency is implemented.
   54            :         // TODO
   55          5 :         frequency = 1000;
   56            :
   57            :         // LED is either ON or OFF
   58          5 :         auto brightness = led.getBrightness();
   59          5 :         if (brightness == ASSERT)
   60            :         {
   61            :             // LED is in Solid ON
   62          1 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
   63            :                 Action::On);
   64            :         }
   65            :         else
   66            :         {
   67            :             // LED is in OFF state
   68          4 :             sdbusplus::xyz::openbmc_project::Led::server ::Physical::state(
   69            :                 Action::Off);
   70            :         }
   71            :     }
   72          6 :     return;
   73            : }
   74            :
   75            : /** @brief Overloaded State Property Setter function */
   76          3 : auto Physical::state(Action value) -> Action
```

Sure enough it has: we're hitting lines 61-63 on the right, and we've now achieved 100% line coverage of the three critical files as shown on the left.



Enforce behaviour with Sensing

https://gerrit.openbmc-project.xyz/#/c/10836/

```
diff --git a/test/physical.cpp b/test/physical.cpp
index fade5c8c3bd9..bbb51438c722 100644
--- a/test/physical.cpp
+++ b/test/physical.cpp
@@ -110,7 +110,10 @@ TEST(Physical, blink)
 {
     sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
     NiceMock<MockLed> led;
-    ON_CALL(led, getTrigger()).WillByDefault(Return("none"));
+    EXPECT_CALL(led, getTrigger()).WillOnce(Return("none"));
+    EXPECT_CALL(led, setTrigger("timer"));
+    EXPECT_CALL(led, setDelayOn(500));
+    EXPECT_CALL(led, setDelayOff(500));
     phosphor::led::Physical phy(bus, LED_OBJ, led);
     phy.state(Action::Blink);
 }
```

Line coverage isn't the end-game though – the tests so far have not captured the expected behaviour, rather guided execution through particular code paths

Here we use the concept of value sensing – mocking a lower-level API and then placing expectations on both the calls into this API and the values passed to it from the code under test

This way we lock in the behaviour of the code under test with respect to the abstraction that it sits on top of. Sensing in this manner is sometimes necessary to constrain the implementation(s) when we can't inspect the behaviour of the code directly. By introducing sensing we can then safely evolve the code under test.

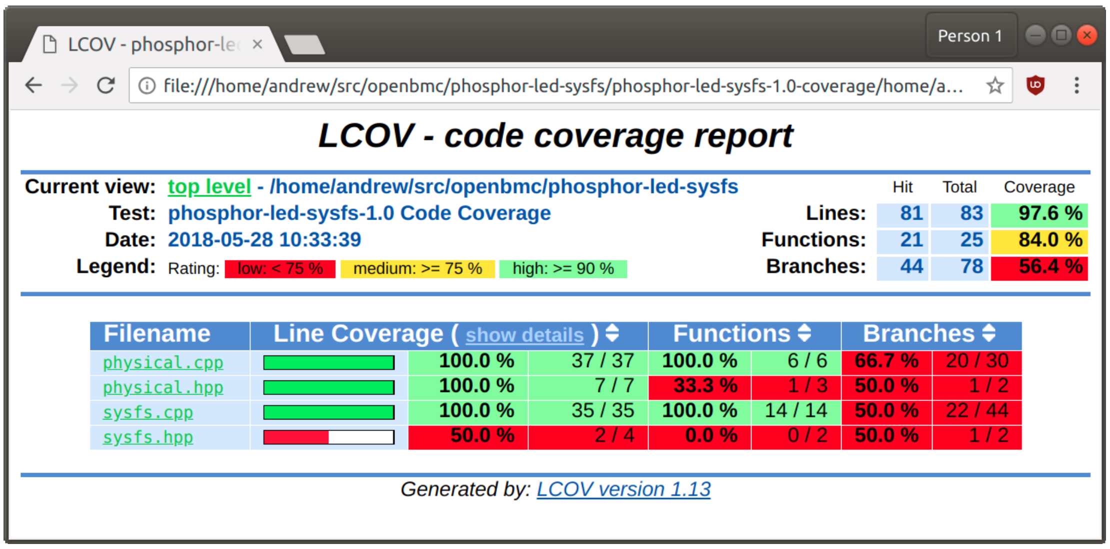
Looking to the example, we're placing expectations that setDelayOn() and setDelayOff() will both be called once with the value 500. If this does not occur the test case will fail.

This is why our initial tests were poor – we did no testing or sensing of values, mainly because we didn't have a well contained mechanism to access the values that were being written across multiple calls. If we fell back to writes the fhe filesystem, we could only inspect the state **after** the code under test has completed execution, which may not be what we want in general.

With mocking and sensing we can detect all calls, all values, and even enforce call ordering constraints as much or as little as we desire. This is the power of the tools that Google Test and Google Mock provide.

Branch Coverage

make **CODE_COVERAGE_BRANCH_COVERAGE=1** check-code-coverage



LCOV - phosphor-le
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/a…

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/andrew/src/openbmc/phosphor-led-sysfs				
Test:	phosphor-led-sysfs-1.0 Code Coverage	Lines:	81	83	97.6 %
Date:	2018-05-28 10:33:39	Functions:	21	25	84.0 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %	Branches:	44	78	56.4 %

Filename	Line Coverage (show details)			Functions		Branches	
physical.cpp		100.0 %	37 / 37	100.0 %	6 / 6	66.7 %	20 / 30
physical.hpp		100.0 %	7 / 7	33.3 %	1 / 3	50.0 %	1 / 2
sysfs.cpp		100.0 %	35 / 35	100.0 %	14 / 14	50.0 %	22 / 44
sysfs.hpp		50.0 %	2 / 4	0.0 %	0 / 2	50.0 %	1 / 2

Generated by: LCOV version 1.13

Now that we’ve tested, mocked and sensed our way to capturing the behaviour of the code, it is time to dive deeper.

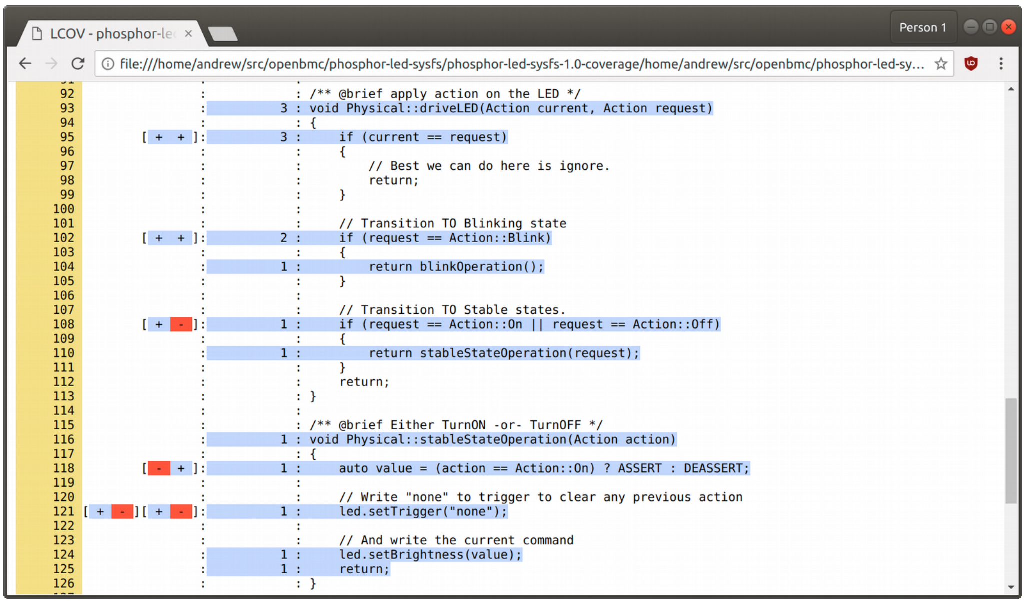
Covering lines and functions only gets us part of the picture. It’s possible to hit all of the lines in your code-base without hitting all the execution paths.

We can extend our understanding of path coverage by enabling branch coverage metrics in our test runs with the CODE_COVERAGE_BRANCH_COVERAGE=1 environment configuration.

Function, line and branch coverage metrics are all increasingly complete models of the the ultimate test regime: program state coverage. The nature of testing is always incomplete as program state coverage is practically unattainable. Consider having a single 32-bit value in your code: Your application or library immediately has over four billion states. Most code-bases use multiple 32-bit quantities, which puts state testing out of reach.

With C++, branch coverage creeps up on the tipping point of diminishing returns due to exception handling. Each function call may throw, and a branch for exception handling is inserted at each call site. This demonstrates the state space explosion at play before we’ve even reached the point of testing each of our integer values. This is the reason that despite having 100% line coverage we only have around 50% branch coverage.

Branch Coverage


LCOV - phosphor-le
Person 1
file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/andrew/src/openbmc/phosphor-led-sy...

```
    92              :           : /** @brief apply action on the LED */
    93              :         3 : void Physical::driveLED(Action current, Action request)
    94              :           : {
    95      [ +  + ]:         3 :     if (current == request)
    96              :           :     {
    97              :           :         // Best we can do here is ignore.
    98              :           :         return;
    99              :           :     }
   100              :           :
   101              :           :     // Transition TO Blinking state
   102      [ +  + ]:         2 :     if (request == Action::Blink)
   103              :           :     {
   104              :         1 :         return blinkOperation();
   105              :           :     }
   106              :           :
   107              :           :     // Transition TO Stable states.
   108      [ +  - ]:         1 :     if (request == Action::On || request == Action::Off)
   109              :           :     {
   110              :         1 :         return stableStateOperation(request);
   111              :           :     }
   112              :           :     return;
   113              :           : }
   114              :           :
   115              :           : /** @brief Either TurnON -or- TurnOFF */
   116              :         1 : void Physical::stableStateOperation(Action action)
   117              :           : {
   118      [ -  + ]:         1 :     auto value = (action == Action::On) ? ASSERT : DEASSERT;
   119              :           :
   120              :           :     // Write "none" to trigger to clear any previous action
   121 [ +  - ][ +  - ]:         1 :     led.setTrigger("none");
   122              :           :
   123              :           :     // And write the current command
   124              :         1 :     led.setBrightness(value);
   125              :         1 :     return;
   126              :           : }
```

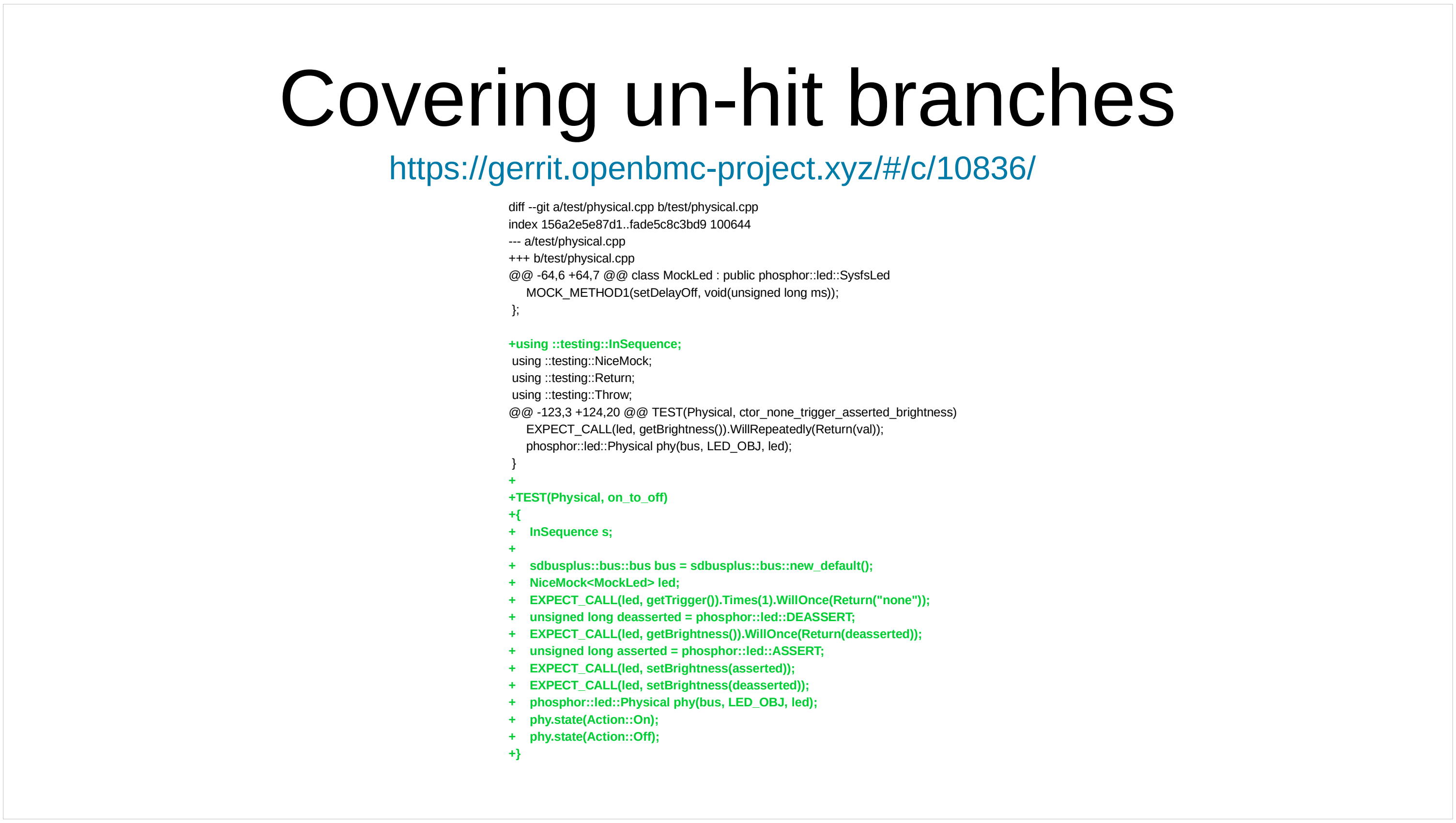
However, despite the considerable noise that results, branch coverage metrics can identify genuine deficiencies in the test suite. Our branch coverage information is on the left in the column between the line numbers and the line's hit count. A red minus indicates we failed to take an arm of the branch, whilst a blue plus indicates an arm was taken. In this case we've found two conditionals where at least one branch was not taken: Line 108 and 118.

Lets go through the process of adding a test to cover both states of line 118. We have already covered one state, but which one? We have three relevant tests: off, on, and blink.

We've implicitly initialised our LEDs to off in our tests by not specifying the getBrightness() behaviour, so explicitly calling Physical.state(Action::Off) hits the early exit at the top of driveLED() on line 98.

We have already tested the off-to-on transition with our 'on' test, as we initialised our LEDs to off as we just mentioned.

By this reasoning we want to test the on-to-off path, which means our LED must first be on, and then be turned off. Lets write a test that does just that.



Covering un-hit branches

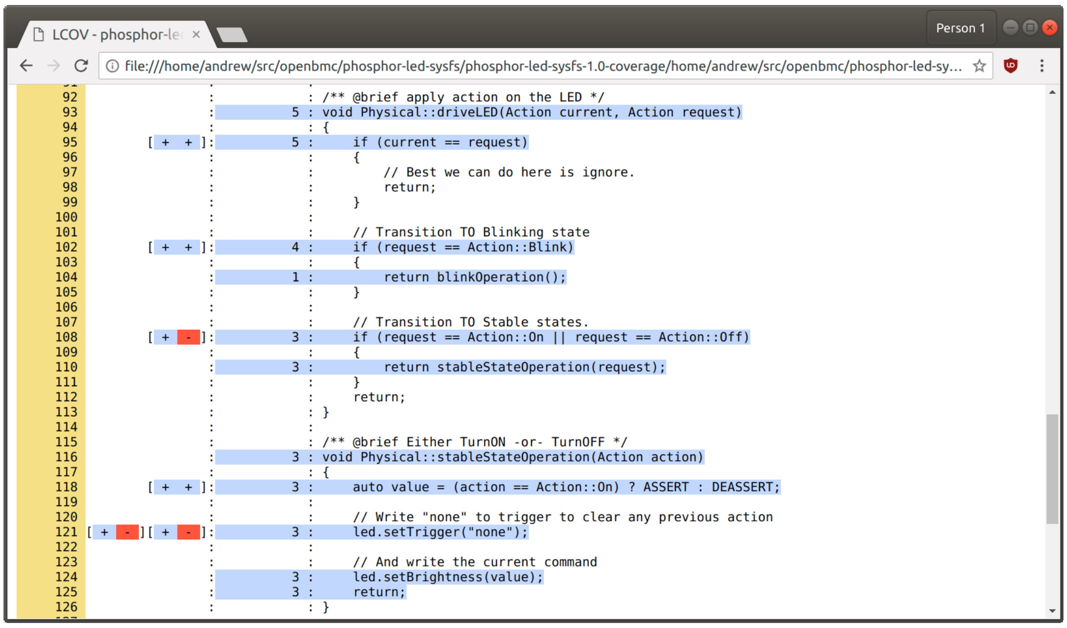
https://gerrit.openbmc-project.xyz/#/c/10836/

```diff
diff --git a/test/physical.cpp b/test/physical.cpp
index 156a2e5e87d1..fade5c8c3bd9 100644
--- a/test/physical.cpp
+++ b/test/physical.cpp
@@ -64,6 +64,7 @@ class MockLed : public phosphor::led::SysfsLed
     MOCK_METHOD1(setDelayOff, void(unsigned long ms));
 };

+using ::testing::InSequence;
 using ::testing::NiceMock;
 using ::testing::Return;
 using ::testing::Throw;
@@ -123,3 +124,20 @@ TEST(Physical, ctor_none_trigger_asserted_brightness)
     EXPECT_CALL(led, getBrightness()).WillRepeatedly(Return(val));
     phosphor::led::Physical phy(bus, LED_OBJ, led);
 }
+
+TEST(Physical, on_to_off)
+{
+    InSequence s;
+
+    sdbusplus::bus::bus bus = sdbusplus::bus::new_default();
+    NiceMock<MockLed> led;
+    EXPECT_CALL(led, getTrigger()).Times(1).WillOnce(Return("none"));
+    unsigned long deasserted = phosphor::led::DEASSERT;
+    EXPECT_CALL(led, getBrightness()).WillOnce(Return(deasserted));
+    unsigned long asserted = phosphor::led::ASSERT;
+    EXPECT_CALL(led, setBrightness(asserted));
+    EXPECT_CALL(led, setBrightness(deasserted));
+    phosphor::led::Physical phy(bus, LED_OBJ, led);
+    phy.state(Action::On);
+    phy.state(Action::Off);
+}
```

Here we mock our LED to off on initialisation, then expect two calls to setBrightness(), one turning the LED on followed by another turning the LED off. We then go and issue the calls on our Physical instance to set the LED on, then set it off.

Let’s check the branch coverage metrics to see whether this has achieved our goal.

The eighth result

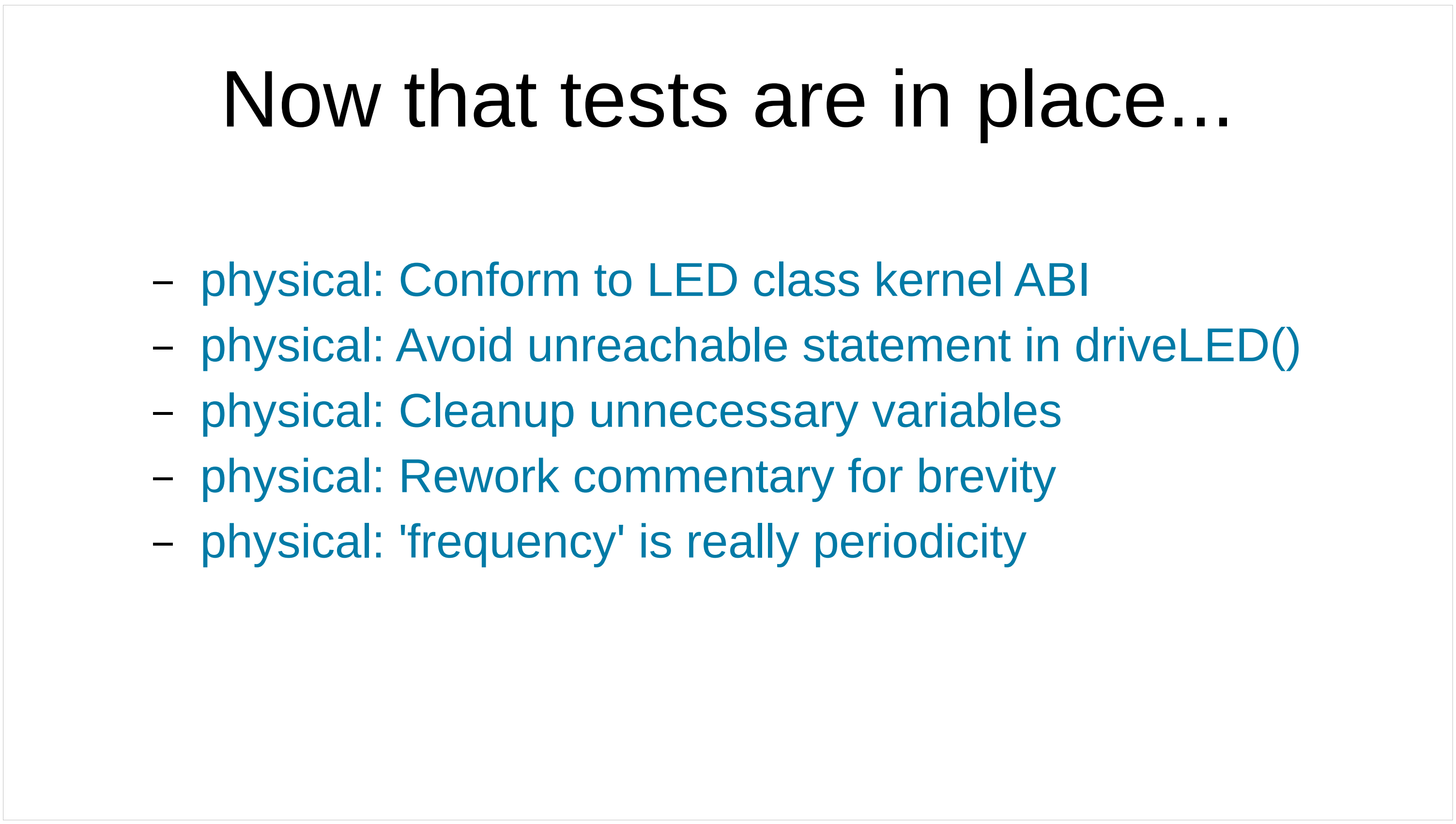


LCOV - phosphor-le

file:///home/andrew/src/openbmc/phosphor-led-sysfs/phosphor-led-sysfs-1.0-coverage/home/andrew/src/openbmc/phosphor-led-sy...

```
      92             :             : /** @brief apply action on the LED */
      93             :           5 : void Physical::driveLED(Action current, Action request)
      94             :             : {
      95     [ +  + ]:          5 :     if (current == request)
      96             :             :     {
      97             :             :         // Best we can do here is ignore.
      98             :             :         return;
      99             :             :     }
     100             :             :
     101             :             :     // Transition TO Blinking state
     102     [ +  + ]:          4 :     if (request == Action::Blink)
     103             :             :     {
     104             :          1 :         return blinkOperation();
     105             :             :     }
     106             :             :
     107             :             :     // Transition TO Stable states.
     108     [ +  - ]:          3 :     if (request == Action::On || request == Action::Off)
     109             :             :     {
     110             :          3 :         return stableStateOperation(request);
     111             :             :     }
     112             :             :     return;
     113             :             : }
     114             :             :
     115             :             : /** @brief Either TurnON -or- TurnOFF */
     116             :          3 : void Physical::stableStateOperation(Action action)
     117             :             : {
     118     [ +  + ]:          3 :     auto value = (action == Action::On) ? ASSERT : DEASSERT;
     119             :             :
     120             :             :     // Write "none" to trigger to clear any previous action
     121 [ +  - ][ +  - ]:          3 :     led.setTrigger("none");
     122             :             :
     123             :             :     // And write the current command
     124             :          3 :     led.setBrightness(value);
     125             :          3 :     return;
     126             :             : }
```

And we see that it has! We’ve now hit both arms of the branch on line 118. This demonstrates how to reason with branch coverage metrics to improve your test coverage. Disregarding the exception handling cases, the other branch arm misses can be dealt with in the same way.



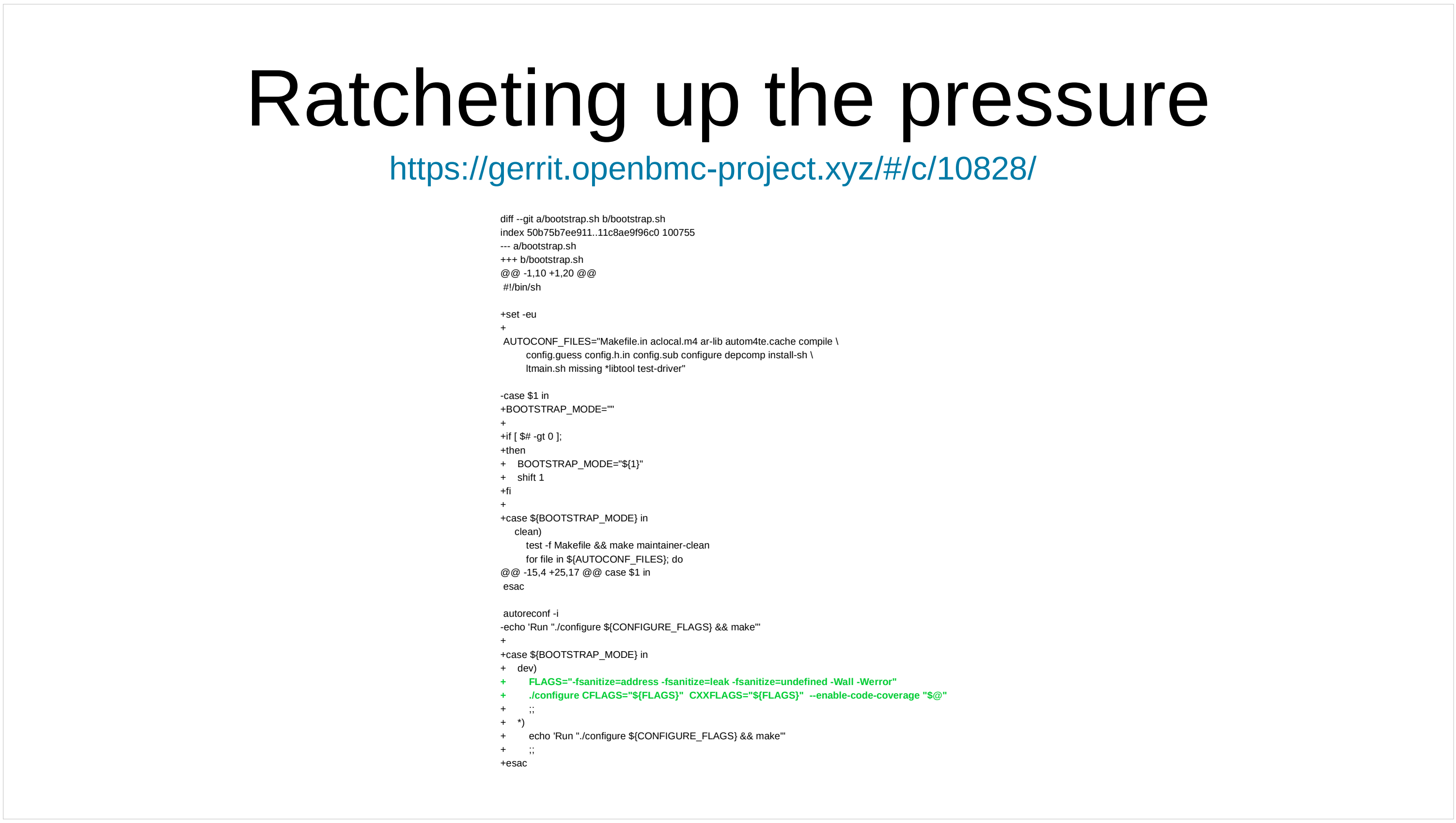
Now that tests are in place...

- physical: Conform to LED class kernel ABI
- physical: Avoid unreachable statement in driveLED()
- physical: Cleanup unnecessary variables
- physical: Rework commentary for brevity
- physical: 'frequency' is really periodicity

Now that the tests are in place, we can start massaging the code to reduce complexity and introduce bug fixes with confidence.

The patches in the slide all depend on the existing tests or introduce their own to ensure that we don’t regress in functionality.

Getting the basic tests in place was the big effort, the patches above took about 5 minutes to write in total, but measurably improve the code, fixing bugs and removing a net 32 lines.



Ratcheting up the pressure

https://gerrit.openbmc-project.xyz/#/c/10828/

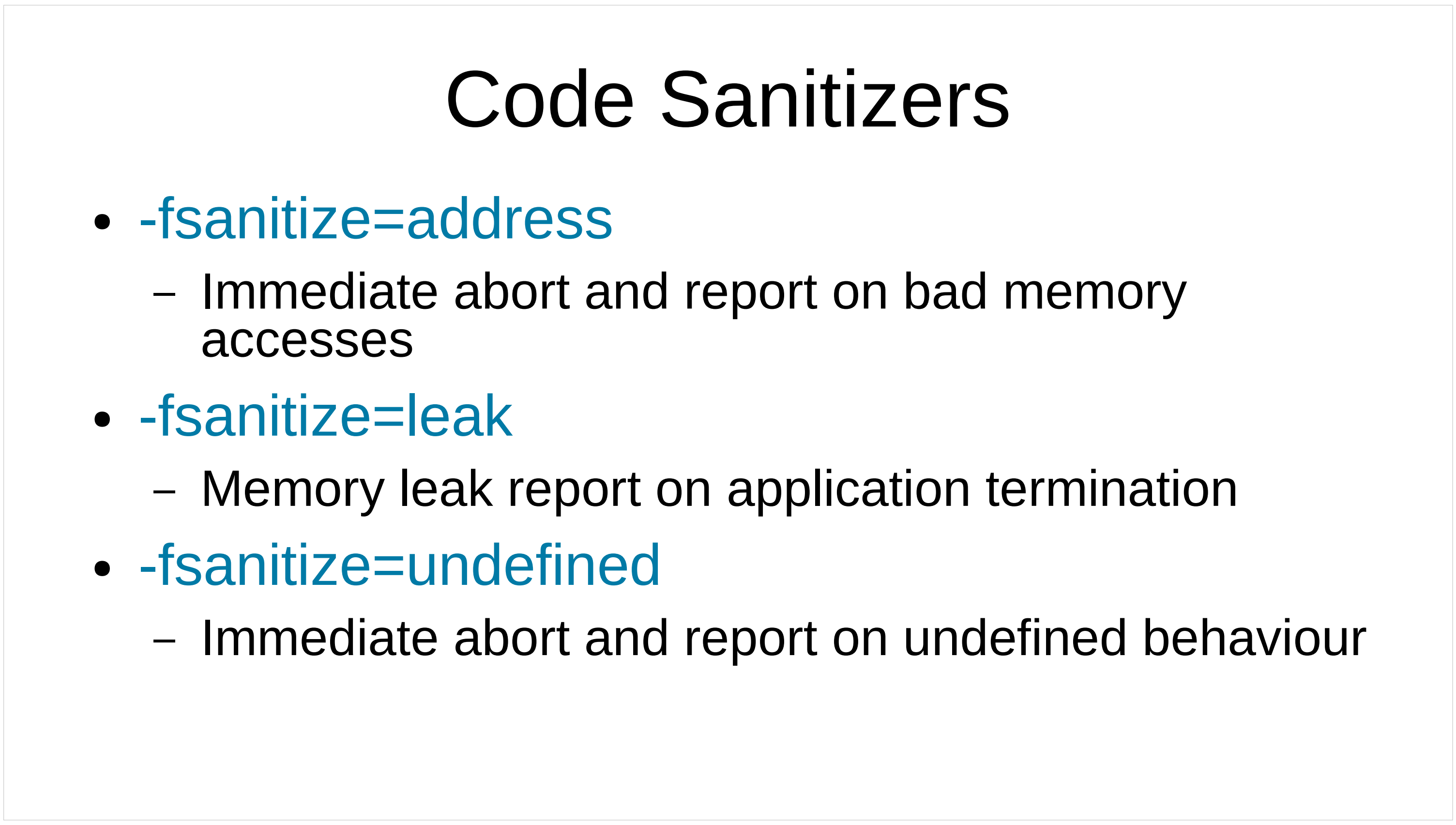
```
diff --git a/bootstrap.sh b/bootstrap.sh
index 50b75b7ee911..11c8ae9f96c0 100755
--- a/bootstrap.sh
+++ b/bootstrap.sh
@@ -1,10 +1,20 @@
 #!/bin/sh

+set -eu
+
 AUTOCONF_FILES="Makefile.in aclocal.m4 ar-lib autom4te.cache compile \
        config.guess config.h.in config.sub configure depcomp install-sh \
        ltmain.sh missing *libtool test-driver"

-case $1 in
+BOOTSTRAP_MODE=""
+
+if [ $# -gt 0 ];
+then
+    BOOTSTRAP_MODE="${1}"
+    shift 1
+fi
+
+case ${BOOTSTRAP_MODE} in
    clean)
        test -f Makefile && make maintainer-clean
        for file in ${AUTOCONF_FILES}; do
@@ -15,4 +25,17 @@ case $1 in
 esac

 autoreconf -i
-echo 'Run "./configure ${CONFIGURE_FLAGS} && make"'
+
+case ${BOOTSTRAP_MODE} in
+    dev)
+        FLAGS="-fsanitize=address -fsanitize=leak -fsanitize=undefined -Wall -Werror"
+        ./configure CFLAGS="${FLAGS}"  CXXFLAGS="${FLAGS}"  --enable-code-coverage "$@"
+        ;;
+    *)
+        echo 'Run "./configure ${CONFIGURE_FLAGS} && make"'
+        ;;
+esac
```

Finally, I want to talk about code sanitizers. These are compiler options that augment our code to catch error conditions at runtime.

The patch above augments our standard bootstrap.sh script to add a ‘dev’ option, which I highly encourage you to use when developing.

The sanitizers will cause your binaries to fail fast, hard and informatively when they misstep. This is precisely the behaviour we want for a our test cases – having the tests carry on and providing a false positive might feel good but will come back to bite us in the future. Fast and accurate feedback is desirable over anything else, and this is what the sanitizers give us.

Obviously these shouldn’t be used in production code – we want to flush out the issues and deploy debugged code, thus the addition of ‘dev’ mode.



Code Sanitizers

- -fsanitize=address
  - Immediate abort and report on bad memory accesses
- -fsanitize=leak
  - Memory leak report on application termination
- -fsanitize=undefined
  - Immediate abort and report on undefined behaviour

Addressing each of the sanitizers used, we're enabling:

- The address sanitizer
- The leak sanitizer, and
- The undefined behaviour sanitizer

Each of the bullets above is a link to more information about the sanitizers and I encourage you to read up.

There is another important sanitizer: The thread sanitizer, which catches and aborts on racy data accesses. If your application uses threads, please do enable that as well!

If you want to know more about how to debug sanitizer failures, talk to me afterwards and we can go over some examples.

Wrapping up...

- Automated testing is automated code review
  - Accelerates future development through characterisation
  - Good coverage drastically reduces risk of (invasive) changes
- You need to instrument your tests
  - Use code coverage metrics to guide your testing
- Rework abstractions to increase coverage
  - Dependency injection allows for mocks and thus sensing
- Use branch coverage analysis to find missed paths
- Use sanitizers so your tests fail fast, hard and informatively
- Use this presentation to adopt tests in your code!

Thanks!

Recommended Reading

- Working Effectively with Legacy Code
- Design Patterns: Elements of Reusable Object-Oriented Software