WINGMAN SOFTWARE

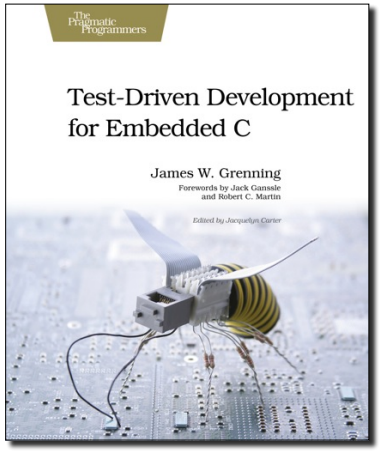
2015 NDC
NORWEGIAN DEVELOPERS CONFERENCE
15-19 June·Oslo, Norway

# Faking and Mocking Legacy Embedded C

by
James W Grenning
james@wingman-sw.com
www.wingman-sw.com
www.twitter.com/jwgrenning

Presented to

Norwegian Developers Conference
Oslo, Norway
June 2015




Minimize DoH!

Debug On Hardware

Copyright Fox Broadcasting. Used under fair use.

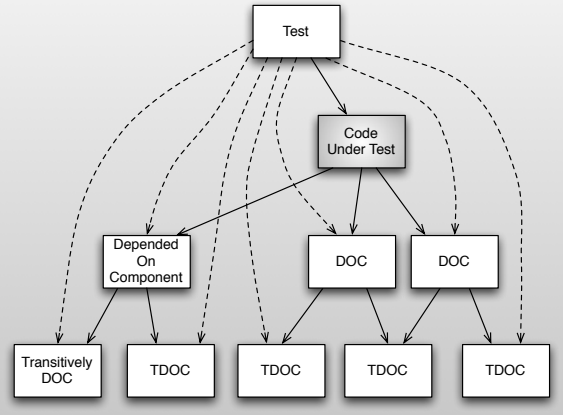



## Test Double

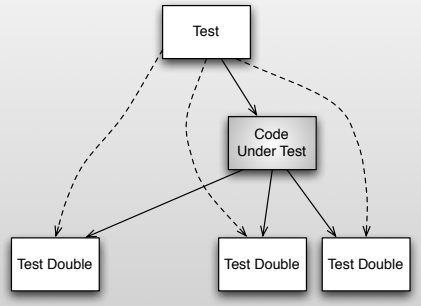
- A test double stands in for some part of the production code during testing
- Test doubles are part of the test fixture that allows a module to be unit tested




## Test Dependency Octopus



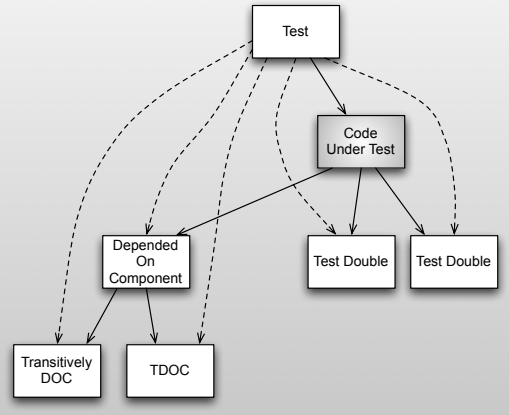
# Managing the Octopus







# Use Production Code when you Can Test Doubles when you Must







# Kinds of Test Doubles
[XUNIT]

- Other names
  - Fakes
  - Spy
  - Null object
  - Exploding fakes
  - Mock object
  - there are others




# When Must You use Test-Doubles?

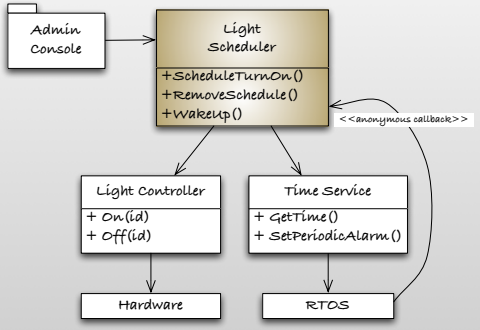
- When the code under test cannot be conveniently tested with the real collaborators.
- Examples:
  - When manual verification is needed (Printed output, user interaction).
  - User Interface (use Model-View-Controller).
  - Database.
  - When the results change (Time, random events).
  - When failures need to be simulated (Network down).
  - Operating system calls, file system calls.
  - Special hardware interaction.

# Swapping Out a Problem Dependency




# Separation of Responsibilities

- Every minute, the RTOS wakes up the Light Scheduler.
- If it is time for one of the lights to be controlled, the LightController is told to turn on/off the light.

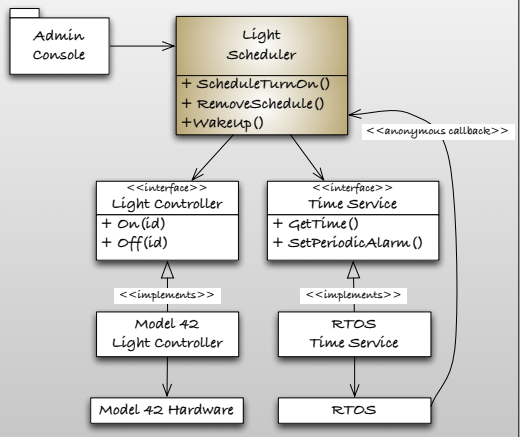






# Light Scheduler Design

- Program to Interfaces
- Separate interface and implementation as separate entities.
- This design has good separation of responsibilities

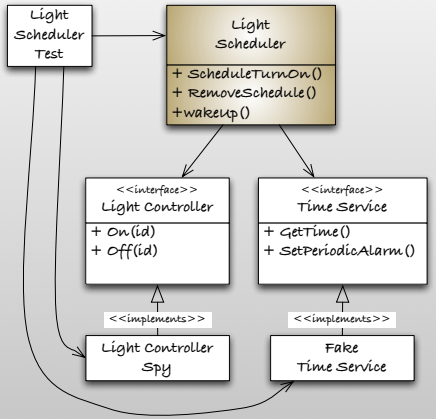






# LightScheduler Test Fixture Design

- Use the real collaborators if you can.
- Use fakes when you must.



# Function Pointer Substitution




## Swap Out a Problem Dependency for Some Tests

### Solution

- Create a function pointer with the same signature as the problem function.
- By default initialize the pointer with the problem function.
- Change clients to call through the function pointer.
- Override the function in the tests where needed.

### Gotcha

- Make sure code runs warning free, if the caller does not see the declaration, C will assume it is a direct function call.




## Loosening Compile-Time Dependencies

- You have some existing function that you need to stub out for some tests...

```
//From the RandomMinuteGenerator.h file
...
int RandomMinuteGenerator_Get();
...
```

```
//From the RandomMinuteGenerator.c file
...
int RandomMinuteGenerator_Get()
{ ...the implementation... }
...
```




## Convert Direct Dependency to Function Pointer

- Modify the function declaration in the header file
  - Add `extern`
  - Add `(*)`

```
//From the RandomMinuteGenerator.h file
...
extern int (*RandomMinuteGenerator_Get)();
...
```

# Convert Direct Dependency to Function Pointer

- Modify the function definition in the .c file
  - Change function name by appending `_impl`
  - Hide the `_impl` by making it `static`
- Add defining instance of the function pointer and initialize the pointer with the `_impl`

```
//From the RandomMinuteGenerator.c file
...
static int RandomMinuteGenerator_Get_impl()
{ ...the implementation... }

int (*RandomMinuteGenerator_Get)() = RandomMinuteGenerator_Get_impl;
```




# No Changes are Needed to Callers of the Function Pointer

- Calling code does not change.
- No pointer dereferencing is needed

```
//from some production code...
    ...
    RandomMinuteGenerator_Get();
    ...
```




# Runtime Substitution Test setup and teardown

Override the function by assigning the test version of the function during setup using `UT_PTR_SET()`

The original pointer is replaced after `teardown()`

```
void setup()
{
   UT_PTR_SET(RandomMinuteGenerator_Get, MockRandomMinuteGenerator_Get);
}

void teardown()
{
}
```




# Reasons to Use Function Pointers for Test-Doubles

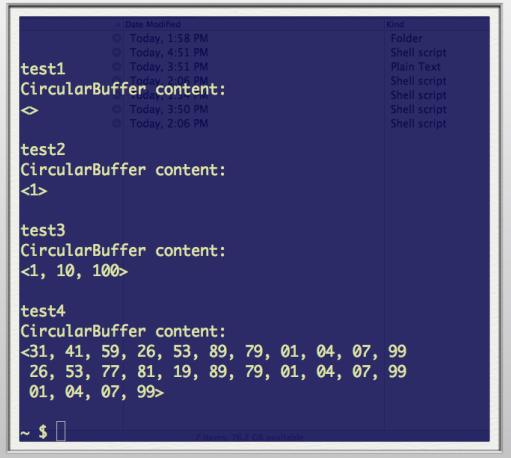
- When you already use function pointers for other purposes (you have a built-in test hook!)
  - Swappable device drivers
  - Swappable implementations
- You need the production implementation in the test build, and you must substitute a test double in the same test build
  - printf
- You don't want to have too many test builds due to using link-time substitution

# Example Usage - Printed Output

- Logs and printed output are helpful for checking program correctness and debugging
- But...
  - They require that you look at the output.
  - They doom you to a lifetime of manually verified tests
- So...
  - Design your code so that printed output can be captured and verified in your unit tests




# Manual/Tedious/Error Prone Output Inspection







# Function Pointer - Runtime Substitution

- Sometimes we want to use the real function.
- Sometimes we want to use a test version.
- Define a function pointer that has the same declaration as the function to override, in this case **`printf()`**

```
#ifndef FORMAT_OUTPUT_INCLUDED
#define FORMAT_OUTPUT_INCLUDED

extern int (*FormatOutput)(const char *, ...);

#endif
```




# By default, FormatOutput is the real printf

```
//FormatOutput.c
#include "FormatOutput.h"
#include <stdio.h>

int (*FormatOutput)(const char* format, ...) = printf;
```

# Production Code Before and After

```
void someProductionCode()
{
    printf("hello %s\n", "world");
}
```

```
#include "FormatOutput.h"

. . . . .

void someProductionCode()
{
    FormatOutput("hello %s\n", "world");
}
```




# UT_PTR_SET

- CppUTest has a mechanism for overriding and restoring pointers that must be restored after each test.
- `UT_PTR_SET()` assigns the pointer and restores the original function pointer after `teardown()`

```
void setup()
{
    UT_PTR_SET(FormatOutput, FormatOutputSpy);
}

void teardown()
{
}
```




# Sometime we Write Tests for Test Code

We must be able to trust the spy.

```
TEST(FormatOutput, ATestThatPrintsThings)
{
    FormatOutputSpy_Create(20);
    FormatOutput("Hello, World\n");
    STRCMP_EQUAL("Hello, World\n",
      FormatOutputSpy_GetOutput());
}
```




# More Checking on our Spy

```
TEST(FormatOutput, PrintMultipleTimes)
{
    FormatOutputSpy_Create(25);
    FormatOutput("Hello");
    FormatOutput(", World\n");
    STRCMP_EQUAL("Hello, World\n",
          FormatOutputSpy_GetOutput());
}
```

# Spying on the Output

```
TEST(CircularBufferPrint, PrintNotYetWrappedOrFull)
{
    CircularBuffer_Put(buffer, 10);
    CircularBuffer_Put(buffer, 20);
    CircularBuffer_Put(buffer, 30);
    CircularBuffer_Print(buffer);

    expectedOutput = "Circular buffer content:\n<10, 20, 30>\n";
    STRCMP_EQUAL(expectedOutput, FormatOutputSpy_GetOutput());
}
```




# Test Fixture

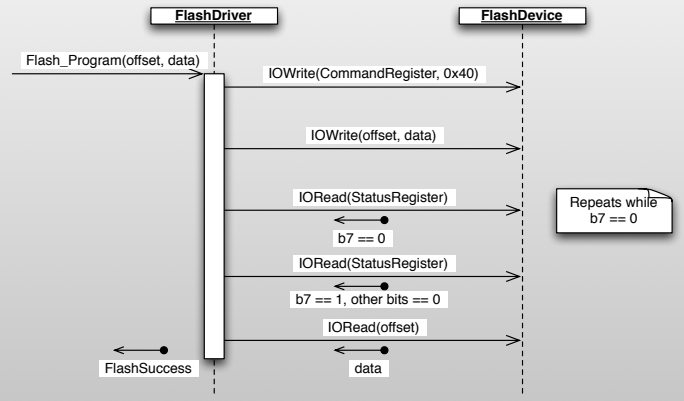
```
TEST_GROUP(CircularBufferPrint)
{
    CircularBuffer buffer;
    const char * expectedOutput;
    const char * actualOutput;

    void setup()
    {
      UT_PTR_SET(FormatOutput, FormatOutputSpy);
      FormatOutputSpy_Create(100);
      buffer = CircularBuffer_Create(10);
    }

    void teardown()
    {
       CircularBuffer_Destroy(buffer);
       FormatOutputSpy_Destroy();
    }
};
```




# Mocking the Silicon

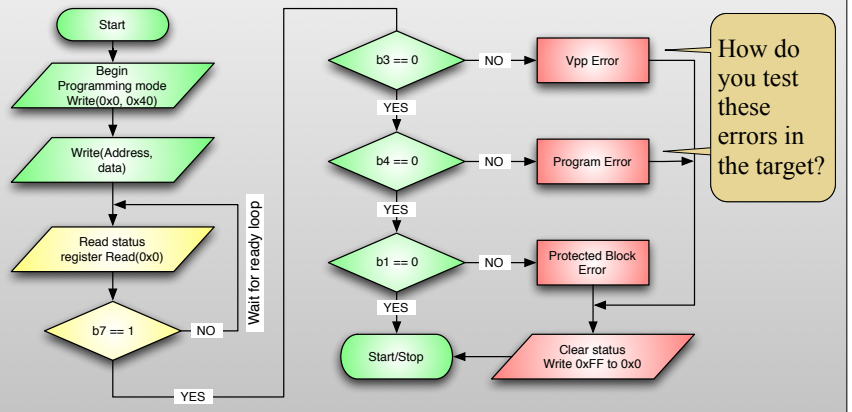



# Message Flow for Flash Program



# Flash Program Flow Chart

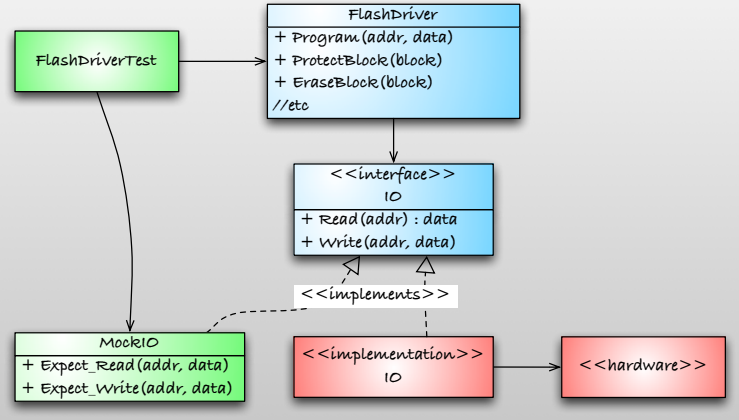
## How Many Tests are Needed?





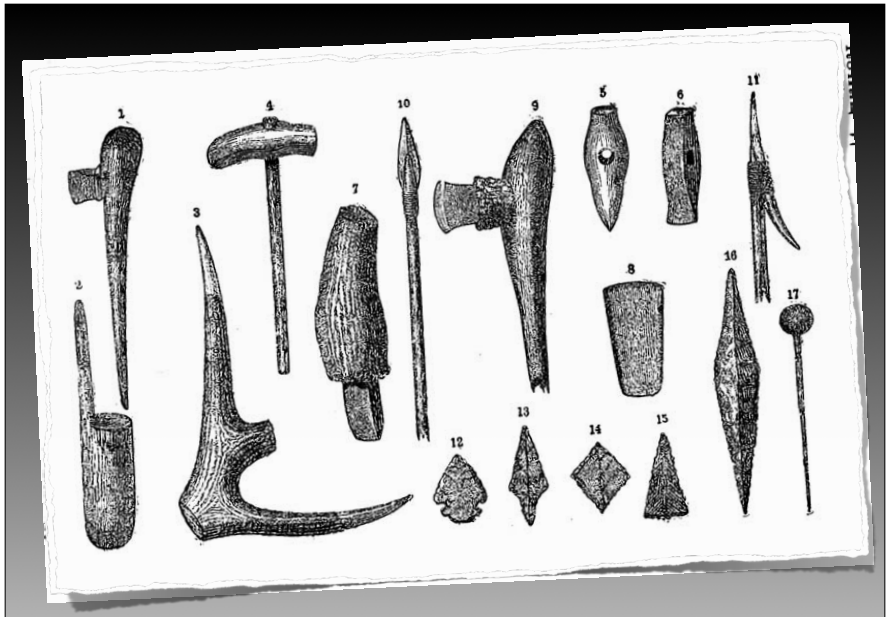


# Flash Driver Test Fixture



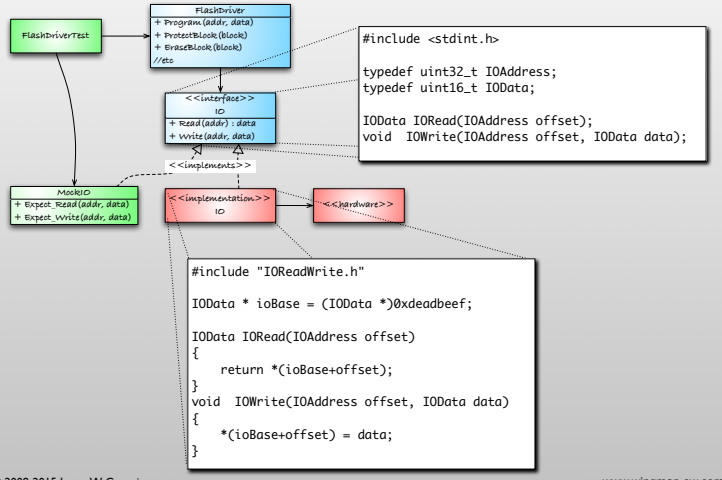






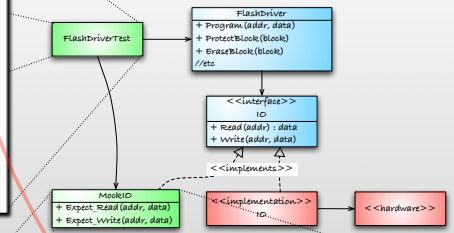



# Flash Driver Test Fixture



# Test Fixture

```
TEST(Flash, Program_succeeds_ready_immediately)
{
    mock("IO")
        .expectOneCall("IOWrite")
        .withParameter("addr", 0x0)
        .withParameter("data", 0x40);
    mock("IO")
        .expectOneCall("IOWrite")
        .withParameter("addr", 1000)
        .withParameter("data", 9999);
    mock("IO")
        .expectOneCall("IORead")
        .withParameter("addr", 0x0)
        .andReturnValue(1<<7);

    LONGS_EQUAL(FLASH_OK, FlashProgram(1000, 9999));
}
```




```
void IOWrite(IOAddress addr, IOData data)
{
    mock("IO")
        .actualCall("IOWrite")
        .withParameter("addr", (int)addr)
        .withParameter("data", (int)data);
}

IOData IORead(IOAddress addr)
{
    return mock("IO")
            .actualCall("IORead")
            .withParameter("addr", (int)addr)
            .returnValue().getIntValue();
}
```

Expectations are defined and recorded in the test case by CppuMock.

Expectations are matched by CppuMock when the actual calls to the mock version of the functions are reported via actualCall().




# Coincidental Compilation Unit Cohabitation




# Coincidental Compilation Unit Cohabitation

Problem

- A source file under test contains some functions that get in the way of testing

Solution

- Split out the problem functions into their own c file
- Override them with a link-time test double




# FlashDriver.c with Problem Dependency

```
int Flash_Write(uint32_t offset, uint16_t data)
{
    //calls IORead and IOWrite
}

int Flash_BlockErase(uint32_t offset, uint16_t data)
{
    //calls IORead and IOWrite
}

. . . and many more

void IOWrite(uint32_t addr, uint16_t data)
{
    *(FLASH_BASE_ADDR + addr) = data;
}

uint16_t IORead(uint32_t addr)
{
    return *(FLASH_BASE_ADDR + addr);
}
```

## Split out the Problem Dependency

```
int Flash_Write(uint32_t offset, uint16_t data)
{
    //calls IORead and IOWrite
}
int Flash_BlockErase(uint32_t offset, uint16_t data)
{
    //calls IORead and IOWrite
}
. . . and many more
```

```
//IO.h
void IOWrite(uint32_t addr, uint16_t data);
uint16_t IORead(uint32_t addr);
```

```
//IO.c
void IOWrite(uint32_t addr, uint16_t data)
{
    *(FLASH_BASE_ADDR + addr) = data;
}
//… etc
```




## Characterizing with Mocks




## Using Mocks to Write Characterization Tests

- You have a legacy driver
- You need to change it
- It has no tests
- Make a small change to allow inserting a mock
  - Maybe split file




## Probe the Code with a Mock

```
TEST(LegacyFlash, FlashProgramSuccess)
{
    result = Flash_Write(0x1000, 0xBEEF);

    LONGS_EQUAL(0, result);
}
```

# The Mock Proclaims What Happened

```
IO/LegacyFlashTest.cpp:39: error:
 Failure in TEST(LegacyFlash, FlashProgramSuccess)
 ../mocks/MockIO.c:84: error:
 R/W 1: No more expectations but was IOWrite(0x0, 0x40)
```




# Inspect the Production code and Add an Expectation

```
TEST(LegacyFlash, FlashProgramSuccess)
{
    mock("IO")
      .expectOneCall("IOWrite")
      .withParameter("addr", (int)addr)
      .withParameter("data", (int)data);

    result = Flash_Write(0x1000, 0xBEEF);

    LONGS_EQUAL(0, result);
}
```

Repeat until covered




# The Fake Function Framework

Thanks Mike Long




# TDD Next to an RTOS with FFF

Intro to the Fake Function Framework

# Choices When Faking the OS

- For new code, you might choose to create an OS abstraction layer.
- For existing code, or where the layer is not desired, you can create an RTOS test double




# How Do I Test OS Dependent Code Like This?

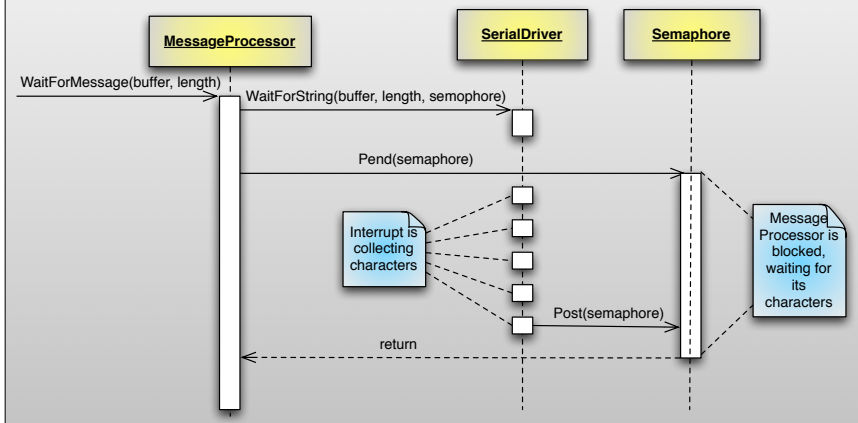
```
int MessageProcessor_WaitForMessage(char * buffer, size_t length)
{
   INT8U error = 0;
   SerialInterrupt_WaitForString(buffer, length, int_sync);
   OSSemPend(int_sync, 1000, &error);
   return error == OS_ERR_NONE;
}
```

I may just fake out WaitForMessage() and manually test the OS dependent code.

But that may not be an option in legacy code where OS primitives are used more liberally.



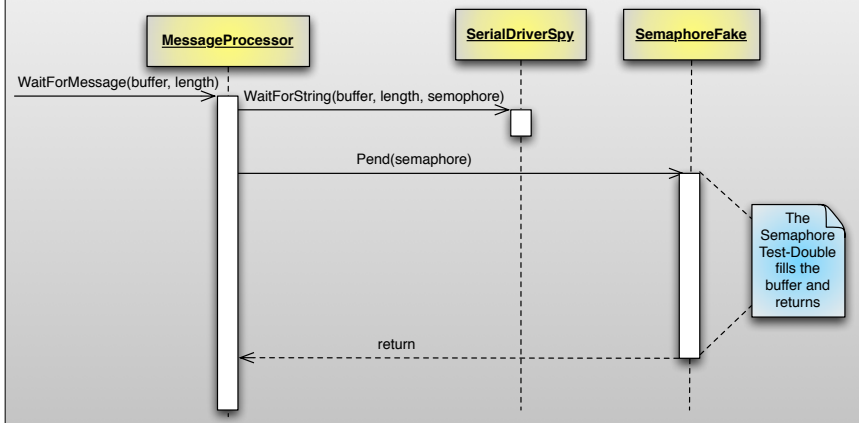

# The ISR Has to Satisfy the Request







# Let the Fake Do the Work That Would Happen Concurrently



# Fakes are Created With the Fake Function Framework

```
#ifndef UCOSII_TEST_DOUBLE_INCLUDED
#define UCOSII_TEST_DOUBLE_INCLUDED

#include "fff2.h"
extern "C" {
#include "ucos_ii.h"
}

FAKE_VALUE_FUNCTION(OS_EVENT *, OSSemCreate, INT16U)

FAKE_VOID_FUNCTION(OSSemPend, OS_EVENT *,\
     INT32U,\
     INT8U *)
```




# Basic fff Features

```
TEST(uCosII_TestDouble, OSSemPend_basics)
{
  OS_EVENT event;
  INT8U error;
  OSSemPend(&event, 0, &error);
  LONGS_EQUAL(1, OSSemPend_fake.call_count);
  POINTERS_EQUAL(&event, OSSemPend_fake.arg0_val);
  LONGS_EQUAL(0, OSSemPend_fake.arg1_val);
  POINTERS_EQUAL(&event, OSSemPend_fake.arg0_history[0]);
  LONGS_EQUAL(0, OSSemPend_fake.arg1_history[0]);
}
```




# The Test Case

```
TEST(MessageProcessor, WaitForMessage_succeeds)
{
   fakeInput = "sched lightOn 5 Monday 20:00";
   CHECK_TRUE(
   MessageProcessor_WaitForMessage(
     (char*)&receiveBuffer, sizeof(receiveBuffer))
 );
   LONGS_EQUAL(1, OSSemPend_fake.call_count);
   STRCMP_EQUAL(fakeInput, receiveBuffer);
}
```




# Its Setup and Teardown

```
TEST_GROUP(MessageProcessor)
{
    void setup()
    {
        FFF_RESET
        OSSemPend_fake.custom_fake = MyOSSemPend;
        MessageProcessor_Create();
    }

    void teardown()
    {
       MessageProcessor_Destroy();
    }
};
```

## Its Custom Implementation Satisfies the Scenario

```
static const char * fakeInput;
static char receiveBuffer[100];
void MyOSSemPend(OS_EVENT *event, INT32U timeout,
         INT8U *error)
{
  memcpy(receiveBuffer, fakeInput, strlen(fakeInput));
  *error = OS_ERR_NONE;
}
```

Read more about it
http://www.renaissancesoftware.net/blog/archives/303




## Preprocessor Substitution




## Command Line Definition

- Compilers all preprocessor symbols to be defined on the command line
- These gcc command line options changes all occurrences of `malloc()` to `cpputest_malloc()` and `free()` to `cpputest_free()`

```
-Dmalloc=cpputest_malloc
-Dfree=cpputest_free
```




## Un-define the Override So the Original Can be Used

```
#undef malloc
#undef free

void * cpputest_malloc(size_t size)
{
  // do the malloc/free book keeping
  return malloc(size);
}

void cpputest_free(void * mem)
{
  // do the malloc/free book keeping
  return free(mem);
}
```

# Problem - `asm`

The legacy code has **`asm`** instructions that won't compile off-target

## Solution - 1

Make **`asm`** go away using forced include

## Solution - 2

Introduce an **`AsmSpy`** to capture the instruction stream and check it in a test case

Read more about it

https://wingman-sw.com/articles/spying-on-embedded-asm



# What is an **`AsmSpy`**?

```
TEST(AsmSpy, captures_asm_text)
{
  AsmSpy("NOP");
  AsmSpy("GLOP");
  AsmSpy("SLOP");
  STRCMP_EQUAL("NOP;GLOP;SLOP;", AsmSpy_Debrief());
}
```

Read more about it

https://wingman-sw.com/articles/spying-on-embedded-asm
https://wingman-sw.com/articles/mocking-the-asm-with-cppumock



# Force Include AsmSpy.h

```
#ifndef ASM_SPY_INCLUDED
#define ASM_SPY_INCLUDED

#define asm AsmSpy

void AsmSpy_Create(int size);
void AsmSpy_Destroy(void);
void AsmSpy(const char *);
const char * AsmSpy_Debrief(void);

#endif
```



# Preprocessor Substitution

- Is the least desirable form of substitution
- Though necessary when
  - Header files won't compile off-target
  - Header files have are the start to a massive dependency chain
  - A direct function call API cannot be changed and you need to override the direct call and use the direct call in the implementation of the fake

    e.g. **`malloc()`**, **`free()`**

# Preprocessor Substitution Variants

- Header file test double
  - Change the include path during test builds
- Force includes
  - Force in a header file that substitutes problem dependencies
- Command line definition
  - Override a symbol one at a time




# Processor Dependent Header File

```
#if defined(_ACME_X42)
    typedef unsigned int        Uint_32;
    typedef unsigned short      Uint_16;
    typedef unsigned char       Uint_8;

    typedef int                 Int_32;
    typedef short               Int_16;
    typedef char                Int_8;

#elif defined(_ACME_A12)
    typedef unsigned long       Uint_32;
    typedef unsigned int        Uint_16;
    typedef unsigned char       Uint_8;

    typedef long                Int_32;
    typedef int                 Int_16;
    typedef char                Int_8;
#else
    #error <acmetypes.h> is not supported for this environment
#endif
```




# Adjust the Include Path So the `#include` Test-Double is Found First

```
#ifndef ACMETYPES_H_
#define ACMETYPES_H_

#include <stdint.h>

typedef uint32_t    Uint_32;
typedef uint16_t    Uint_16;
typedef uint8_t     Uint_8;

typedef int32_t     Int_32;
typedef int16_t     Int_16;
typedef int8_t      Int_8;

#endif /* ACMETYPES_H_ */
```

Read about it at http://www.renaissancesoftware.net/blog/archives/231




# Just to be Sure, Add this Test

```
TEST(acmetypes, checkIntSizes)
{
   LONGS_EQUAL(1, sizeof(Uint_8));
   LONGS_EQUAL(1, sizeof(Int_8));
   LONGS_EQUAL(2, sizeof(Uint_16));
   LONGS_EQUAL(2, sizeof(Int_16));
   LONGS_EQUAL(4, sizeof(Uint_32));
   LONGS_EQUAL(4, sizeof(Int_32));
}
```

# Ideally

- Use a portable types file, rather than vendor dependent file
- Limit the areas of your code that depend on problem vendor code




# See the C that is NOT C

```
/* Chip Vendor Specific Header File     */
...
extern cregister volatile unsigned int AMR;     /* Address Mode Register      */
extern cregister volatile unsigned int CSR;     /* Control Status Register    */
extern cregister volatile unsigned int IFR;     /* Interrupt Flag Register    */
extern cregister volatile unsigned int ISR;     /* Interrupt Set Register     */
extern cregister volatile unsigned int ICR;     /* Interrupt Clear Register   */
extern cregister volatile unsigned int IER;     /* Interrupt Enable Register  */
extern cregister volatile unsigned int ISTP;    /* Interrupt Service Tbl Ptr  */
extern cregister volatile unsigned int IRP;     /* Interrupt Return Pointer   */
extern cregister volatile unsigned int NRP;     /* Non-maskable Int Return Ptr*/
extern cregister volatile unsigned int IN;      /* General Purpose Input Reg  */
extern cregister volatile unsigned int OUT;     /* General Purpose Output Reg */
...
```

Read about it
http://www.renaissancesoftware.net/blog/archives/249




# Force Include

```
//OffTargetDefines.h

#define cregister
#define interrupt
...
```




# Registers Become Global Variables

```
// FakeRegisters.c
volatile unsigned int AMR;
volatile unsigned int CSR;
volatile unsigned int IFR;
volatile unsigned int ISR;
volatile unsigned int ICR;
volatile unsigned int IER;
volatile unsigned int ISTP;
volatile unsigned int IRP;
volatile unsigned int NRP;
volatile unsigned int IN;
volatile unsigned int OUT;
```

# Ideally

- You should limit areas of your code that are vendor specific
- Make a Hardware Abstraction Layer




# Command Line Definition

- Compilers all preprocessor symbols to be defined on the command line
- These gcc command line options changes all occurrences of `malloc()` to `cpputest_malloc()` and `free()` to `cpputest_free()`

```
-Dmalloc=cpputest_malloc
-Dfree=cpputest_free
```




# Undefine the Override So the Original Can be Used

```
#undef malloc
#undef free

void * cpputest_malloc(size_t size)
{
  // do the malloc/free book keeping
   return malloc(size);
}

void cpputest_free(void * mem)
{
  // do the malloc/free book keeping
   return free(mem);
}
```




# Problem - `asm`

The legacy code has **`asm`** instructions that won't compile off-target

## Solution - 1

Make **`asm`** go away using forced include

## Solution - 2

Introduce an **`AsmSpy`** to capture the instruction stream and check it in a test case

Read more about it

http://www.renaissancesoftware.net/blog/archives/136

# What is an **AsmSpy**?

```
TEST(AsmSpy, captures_asm_text)
{
   AsmSpy("NOP");
   AsmSpy("GLOP");
   AsmSpy("SLOP");
   STRCMP_EQUAL("NOP;GLOP;SLOP;", AsmSpy_Debrief());
}
```

Read more about it
http://www.renaissancesoftware.net/blog/archives/136
http://www.renaissancesoftware.net/blog/archives/143




# Force Include AsmSpy.h

```
#ifndef ASM_SPY_INCLUDED
#define ASM_SPY_INCLUDED

#define asm AsmSpy

void AsmSpy_Create(int size);
void AsmSpy_Destroy(void);
void AsmSpy(const char *);
const char * AsmSpy_Debrief(void);

#endif
```




# Getting Started




# Crash to Pass Algorithm

- Goal: call the function under test from the test harness
- Caution: Don't get discouraged.
- Once the first test runs and the initialization is complete, subsequent tests are created much more rapidly.
- See
  - https://wingman-sw.com/articles/get-your-legacy-c-into-a-test-harness

```
void addNewLegacyCtest()
{
  while (makeItCompile() && makeItLink())
    ;
  while (runCrashes())
  {
    findRuntimeDependency();
    fixRuntimeDependency();
  }
  makeTestMoreMeaningful();
}
```

# Dependencies and Linker Errors

```
Linking HomeAutomation_tests
objs/LightScheduler.o: In function `LightScheduler_AddTurnOn':
/sandbox/LightScheduler.c:86: undefined reference to `LightController_On'
objs/LightScheduler.o: In function `LightScheduler_AddTurnOff':
/sandbox/LightScheduler.c:91: undefined reference to `LightController_Off'
lib/libHomeAutomation.a(Scheduler.o): In function `Scheduler_Create':
/sandbox/Scheduler.c:37: undefined reference to `TimeService_SchedulePeriodicAlarm'
lib/libHomeAutomation.a(Scheduler.o): In function `Scheduler_Destroy':
/sandbox/Scheduler.c:53: undefined reference to `TimeService_CancelPeriodicAlarm'
lib/libHomeAutomation.a(Scheduler.o): In function `processReadyEvents':
/sandbox/Scheduler.c:70: undefined reference to `TimeService_DaysMatch'
lib/libHomeAutomation.a(Scheduler.o): In function `Scheduler_Wakeup':
/sandbox/Scheduler.c:79: undefined reference to `TimeService_GetDay'
/sandbox/Scheduler.c:80: undefined reference to `TimeService_GetMinute'
.. etc ...
collect2: error: ld returned 1 exit status
```




# Generated Exploding Fakes

```
#ifndef EXPLODING_FAKE_INCLUDED
#define EXPLODING_FAKE_INCLUDED
#include "CppUTest/TestHarness_c.h"

#define EXPLODING_FAKE_FOR(f) \
        void f() { FAIL_TEXT_C("go write a proper stub for " #f); }
#endif
```

```
//Generated from linker errors
#include "explodingfake.h"

EXPLODING_FAKE_FOR(LightController_Off)
EXPLODING_FAKE_FOR(LightController_On)
EXPLODING_FAKE_FOR(TimeService_CancelPeriodicAlarm)
EXPLODING_FAKE_FOR(TimeService_DaysMatch)
EXPLODING_FAKE_FOR(TimeService_GetDay)
EXPLODING_FAKE_FOR(TimeService_GetMinute)
```




WINGMAN SOFTWARE

2015 NDC NORWEGIAN DEVELOPERS CONFERENCE 15-19 June·Oslo, Norway

Talk to me on Twitter
http://twitter.com/jwgrenning

Connect with me on linkedin.com
http://www.linkedin.com/in/jwgrenning
Remind me how we met.

http://www.wingman-sw.com
http://blog.wingman-sw.com
http:// www.jamesgrenning.com
unpappd.com jwgrenning


http://wingman-sw.com/tddec