
# Especialización en sistemas embebidos (CESE)

## Testing de software embebido

### Clase 3


Laboratorio de Sistemas Embebidos

# ¿Como programamos?

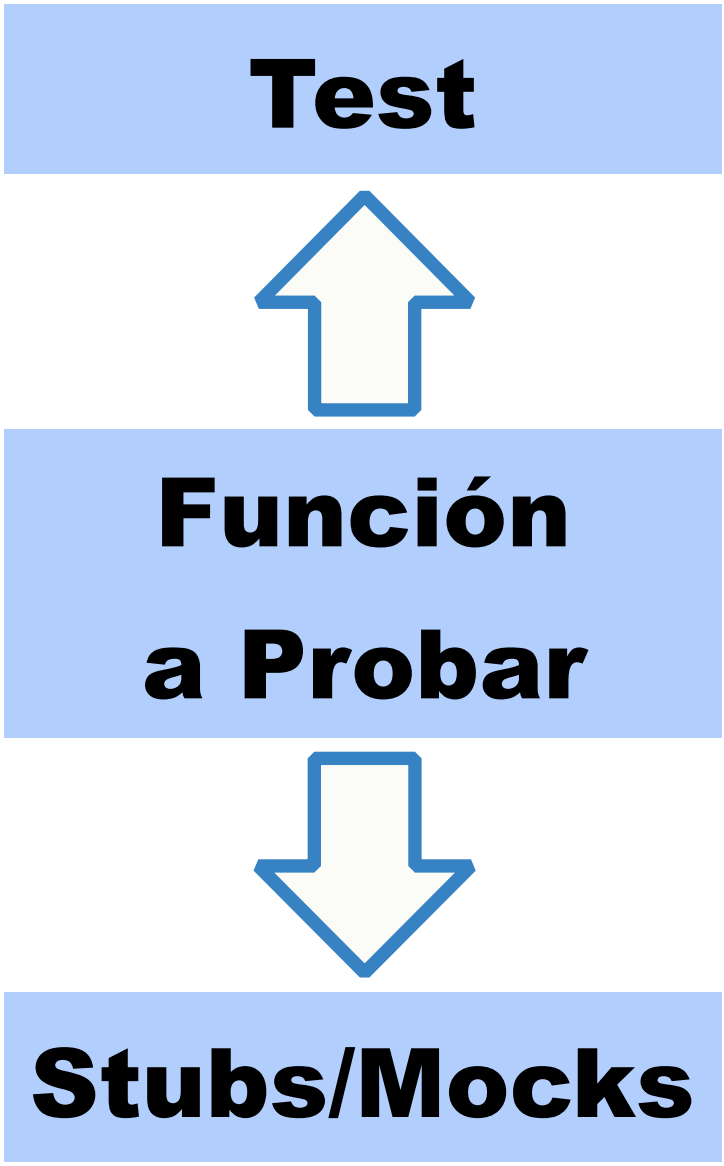
- Obtenemos un sub-conjunto de requerimientos
- Definimos una o mas funciones
  - Definimos los parámetros y el comportamiento
- Escribimos cada una de las funciones
- Escribimos un programa que prueba la función
  - Ingresa valores determinados a la función
  - Comparamos el resultado obtenido con el esperado
- Conservamos la función
- Tiramos el programa de prueba

# ¿Y si guardamos las pruebas?

- Muchas veces al implementar una nueva funcionalidad rompemos otra
- Si guardamos las pruebas “junto” con la función:
  - Cada vez que algo cambia podemos verificar que lo que antes andaba sigue funcionado
  - Esto es un test unitario: prueba las funciones publicas de una clase.
- No compilo un programa sino muchos programas:
  - El programa que originalmente queria escribir
  - Uno por cada programa de prueba
- Los frameworks de testing para simplifican el proceso

# Pruebas Unitarias

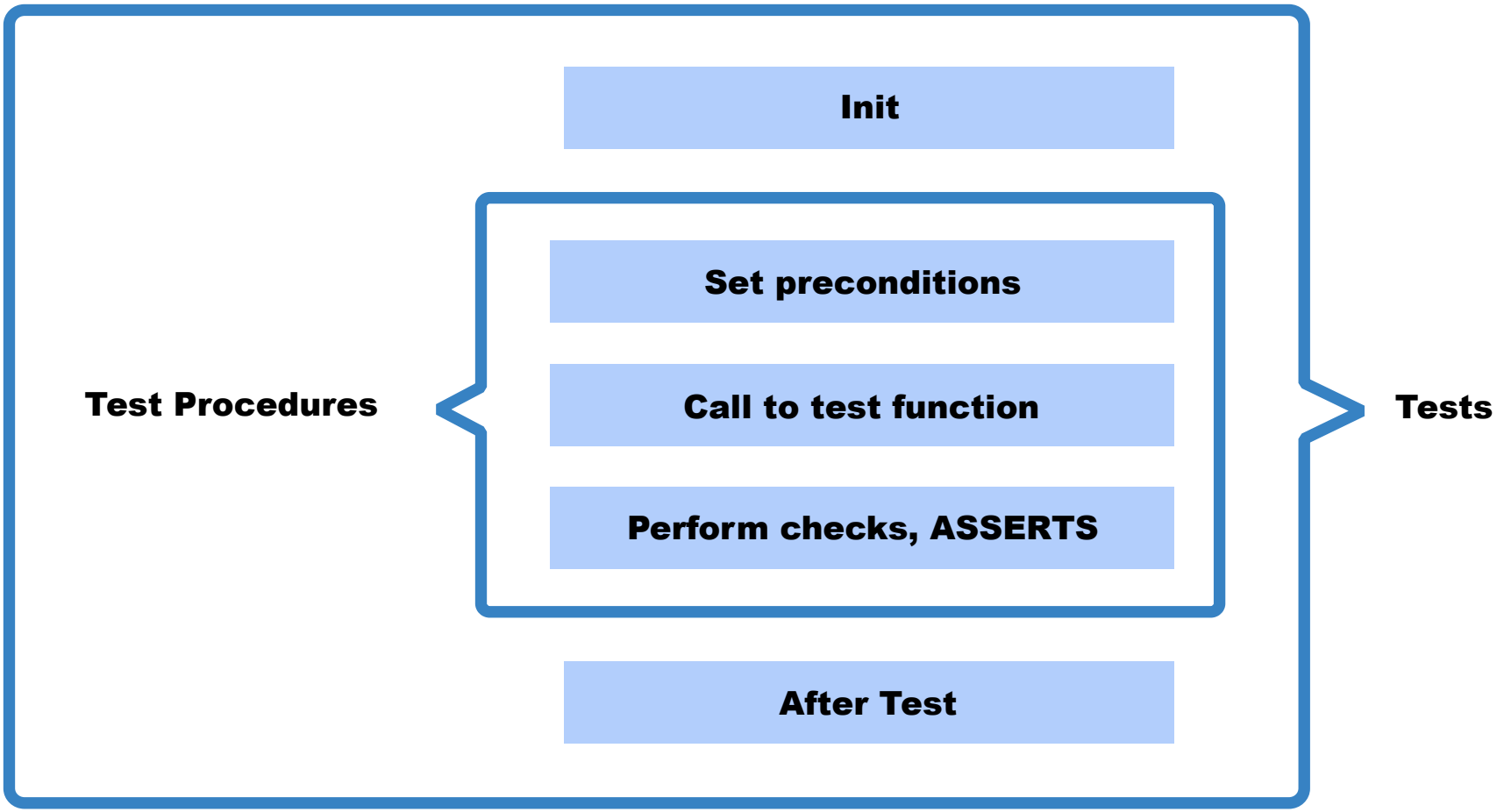
- Prueban un archivo c
- Se prueban únicamente las funciones externas
- Todas las funciones utilizadas de otros archivos utilizadas por el código se reemplazan por funciones Stub o Mock



# Pruebas de Módulos

- Prueban todo el conjunto de archivos que conforman un modulo
- Se deben reemplazar las funciones provistas por otros módulos por stubs o mocks
- Son muy similares a las pruebas unitarias pero incluyen los posibles problemas de acoplamiento entre los componentes del módulo archivos
- Se utilizan las mismas herramientas que para los test unitarios

# Estructura de los test



# Unity

- Es una librería para testing de código C a nivel de unidades y módulos.
- Permite escribir Test muy rápidamente y en una forma muy conceptual.
- Se expresan muy claramente los resultados esperados de una operación.

```
void test_calcularCRC(void) {
    char data[11] = "0123456789";
    TEST_ASSERT_EQUAL(0x443D, calcularCRC(data, 10));
}
```

# CMock

- Es una librería para generar funciones de Stub o de Mock
- Emulan a las funciones externas al archivo que se esta probando
  - Las funciones stub que no hacen nada y solo permiten la compilación del test
  - Las funciones mock son funciones verifican los parámetros y devuelven resultados predefinidos

# Ceedling

- Es un framework que trabaja en conjunto con unity y cmock
- Automatiza las tareas de testing
  - Genera automaticamente los mock
  - Ejecuta los test
  - Recopila los resultados
- Automatiza también la creación del makefile

# Coverage

- Un factor de calidad del software es el grado de cobertura de los test
- Existen distintas métricas
  - Function coverage
  - Statment coverage
  - Branch coverage
  - Condition coverage
- Cuanto mayor la cobertura de los test menor es la posibilidad de errores

# Reportes de cobertura

## CIAA Firmware Quality Report

Generated....: 2015-05-05 02:11:34

GIT Revision.: 2e9e177b442ce427b7d70aee2a2d2c54880cffad

| Module | Tested Files | Compiled Tests | Test Cases | Function Coverage | Line Coverage | Branch coverage |
|---|---|---|---|---|---|---|
| update | 0/3 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| posix | 6/11 | 6/6 | 17/17 | 16/37 | 136/384 | 0/0 |
| drivers | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| template | 0/1 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| libs | 2/2 | 2/2 | 10/10 | 5/5 | 45/45 | 0/0 |
| tools | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| base | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| ciaak | 0/1 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| systests | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| modbus | 4/5 | 4/4 | 47/47 | 43/43 | 566/619 | 0/0 |
| rtos | 0/22 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| plc | 0/1 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |

# Reportes de cobertura

```
144 /** \brief Get transport type
145  **
146  ** This function indicate the type of transport (Master or Slave)
147  **
148  ** \param[in] handler handler of transport
149  ** \return    CIAAMODBUS_TRANSPORT_TYPE_MASTER  -> master
150  **            CIAAMODBUS_TRANSPORT_TYPE_SLAVE   -> slave
151  **            CIAAMODBUS_TRANSPORT_TYPE_INVALID -> invalid transport or
152  **            not initialized
153  **/
154 extern int8_t ciaaModbus_transportGetType(int32_t handler);
```

```
285          :        4 : extern i
286          :          : {
287          :          :    int8_
288          :          :
289          :        4 :    if (c
290          :          :    {
291          :        4 :       sw
292          :          :       {
293          :          :          case CIAAMODBUS_TRANSPORT_MODE_ASCII_MASTER:
294          :          :          case CIAAMODBUS_TRANSPORT_MODE_RTU_MASTER:
295          :          :          case CIAAMODBUS_TRANSPORT_MODE_TCP_MASTER:
296          :        2 :             ret = CIAAMODBUS_TRANSPORT_TYPE_MASTER;
297          :        2 :             break;
298          :          :
299          :          :          case CIAAMODBUS_TRANSPORT_MODE_ASCII_SLAVE:
300          :          :          case CIAAMODBUS_TRANSPORT_MODE_RTU_SLAVE:
301          :          :          case CIAAMODBUS_TRANSPORT_MODE_TCP_SLAVE:
302          :        2 :             ret = CIAAMODBUS_TRANSPORT_TYPE_SLAVE;
303          :        2 :             break;
304          :          :
305          :          :          default:
306          :        0 :             ret = CIAAMODBUS_TRANSPORT_TYPE_INVALID;
307          :        0 :             break;
308          :          :       }
309          :          :    }
310          :          :    else
311          :          :    {
312          :        0 :       ret = -1;
313          :          :    }
314          :          :
315          :        4 :    return ret;
316          :          : }
```

# Como funciona ceedling

- Escribimos una función de suma con saturación.
- La probamos de forma tradicional.
- Escribimos nuestro primer test unitario.
- Obtenemos nuestro primer informe de coverage.
- Agregamos pruebas para tener un coverage de 100%.
- Escribimos una función de promedio que usa la suma.
- Creamos nuestro primer Mock.
- Corremos nuestro primer test de modulo.

# ¿Y si cambiamos la forma?

- Los requisitos son los que dirigen el desarrollo.
- Los test verifican el cumplimiento de requisitos.
- Si escribimos primero los test y después el programa:
  - Garantizamos el cumplimiento de los requisitos
  - Obligamos a realizar un correcto análisis de requisitos
- TDD: Test Driven Development
  - Primero escribimos los test
  - Después escribimos el código para que cumplan los test

# Test Driven Development

- Mejora la calidad del software desarrollado
- Diseño enfocado a las necesidades
- Diseño mas simple
- Mayor productividad
- Menor tiempo invertido en depuración de errores

# Las tres reglas del TDD

- No puede escribir ningún código de producción a menos que sea para hacer pasar un test unitario que falla.
- No escriba mas de un test unitario que falle, y una falla de compilación es también una falla.
- No escriba mas código de producción que el necesario para hacer pasa la prueba de unidad que falla.

http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

# Driver de LEDs: Requisitos

- Maneja 16 leds de dos estados (encendido y apagado).
- Se puede cambiar el estado de un led sin afectar a los otros.
- Se puede cambiar el estado de todos los led's en una sola operación.
- Se puede recuperar el estado actual de un led.
- Los led's están mapeados en una palabra de 16 bits en memoria en una dirección a determinar.
- Para encender el led se debe escribir un "1" en el bit y para apagarlo se debe escribir un "0".
- El led 1 corresponde al lsb y el led 16 al msb.
- El reset de hardware no define un estado conocido de los led's, estos deben ser apagados por software.

# Driver de LEDs : Test

- Después de la inicialización todos los LEDs deben quedar apagados.
- Se puede prender un LED individual.
- Se puede apagar un LED individual.
- Se pueden prender y apagar múltiples LED’s.
- Se pueden prender todos los LEDs de una vez.
- Se pueden apagar todos los LEDs de una vez.
- Se puede consultar el estado de un LED.
- Revisar limites de los parametros.
- Revisar parámetros fuera de los limites.

# Empezando el Proyecto

- Instalación de Ruby

  **`sudo apt-get install ruby`**
- Instalación de Ceedling

  **`sudo gem install ceedling`**
- Creación del repositorio de trabajo

  **`git init leds`**
- Creación del proyecto ceedling

  **`ceedling new leds`**
- Creación del archivo de test unitario

  **`nano test/test_leds.c`**

# Algo parece no andar bien

- Siguiendo la metodología al pie de la letra.
  - Tenemos una función mal escrita.
  - Pero tenemos un test que funciona bien.
  - Un test posterior debería detectar el error.
- Al tener un test que detecta un cambio
  - Tenemos un “ancla”que fija el comportamiento del código.
  - Hacemos los cambios en un comportamiento por vez.
- En esencia vamos colocando “referencias” en el comportamiento para ir “llevando” el código por el camino correcto.

# En resumen

- Debemos escribir un test
  - Debemos asegurarnos de que primero falle.
- Ahora debemos escribir la cantidad mínima de código para resolver el problema.
  - No caer en la tentación de escribir código de mas.
  - El limite entre lo que “me hace falta” y lo que podría servir lo marca el test que escribí al inicio
- Y ahora repetimos el ciclo indefinidamente hasta cubrir todos los requerimientos
  - En un futuro con nuevos requerimientos el proceso es idéntico.
  - El código obtenido es el más simple posible.
  - Y por lo tanto el más robusto

# Bibliografía


- Test-Driven Development for Embedded C
  James W. Grenning
- The OLD Object Mentor Blog Site
  http://butunclebob.com
- Ejemplo simple de testing con ceedling
  https://bitbucket.org/labmicro/testing
- Ejemplo simple de TDD con leds
  https://bitbucket.org/labmicro/tdd_leds

# Bibliografía


- Test-Driven Development for Embedded C
  James W. Grenning
- The OLD Object Mentor Blog Site
  http://butunclebob.com
- Ejemplo simple de testing con ceedling
  https://bitbucket.org/labmicro/testing
- Ejemplo simple de TDD con leds
  https://bitbucket.org/labmicro/tdd_leds