Praktikum IV
Embedded Software

Prof. Dr. Nikolaus Wulff

30. November 2020

1 CUnit Tests

```
1  /*
2   * main.c
3   *
4   */
5  #include "CUnit.h"
6
7  /** declare external modules to include within the test suite */
8  DECLARE_TEST(assembler)
9  DECLARE_TEST(memory)
10 DECLARE_TEST(dummy)
11
12 /** add modules to the test suite and run/execute the suite    */
13 BEG_SUITE(suite)
14     ADD_TEST(assembler),
15     ADD_TEST(dummy),
16     ADD_TEST(memory)
17 END_SUITE(suite)
18
19 RUN_SUITE(suite)
```

Listing 1: Definition und Ausführung einer C TestSuite in main.c.

Systematisches Testen ist eine der wichtigsten Aufgaben eines Software Entwicklers. In der Java Welt hat sich hierfür das Testen mit *JUnit* als sehr erfolgreich in zahlreichen Projekten herausgestellt. Ziel dieses Pratikums ist es einen einheitlichen Rahmen für C Tests zu entwickeln[1] und einige einfache Testroutinen in einer *CUnit* TestSuite einzubinden und auszuführen.

Das Listing (1) zeigt beispielhaft den Test dreier Module *assembler*, *dummy* und *memory* die zu einer TestSuite hinzugefügt werden, welche dann anschließend ausgeführt wird. Ein Testdurchführung könnte beispielhaft wie folgt aussehen:

[1] Es gibt im Internet bereits die beiden JUnit Pedants *CUnit* und *CppUnit*, es geht allerdings bei diesem Pratkikum darum ein eigenes Testframework selbst zu entwickeln.

```
pi@rasp08 ~/workspace/HardwareTest/Debug $ ./HardwareTest

STARTING  ALL TESTS

Starting Test >>assemblerTest<<


Starting Test >>dummyTest<<

ERROR ../dummyTest.c:22 Hugo != 4711


Starting Test >>nullJumpTest<<

FATAL signal error: 11 => Segmentation fault


Starting Test >>memoryTest<<

11011110101011011011111011101111
10000000000000000000000000000000

TEST RESULTS

Test runs: 4 errors: 1 failure: 0 fatal: 1
```

Hinter `DECLARE_TEST` verbirgt sich im Wesentlichen nicht mehr als ein (externer) Verweis auf die zu rufenden Testfunktionen, `BEG_SUITE` eröffnet ein Feld mit den Testfällen, die per `ADD_TEST` hinzugefügt werden. `END_SUITE` schließt das Feld geeignet und `RUN_SUITE` ruft dann schlußendlich die Testdurchführung auf und wertet die Testergebnisse aus. Die entsprechenden Makros erlauben ein flexibles „Coding-by-Convention“ Muster. So wird in den Tests z.B. zu jedem Modul XYZ eine entsprechende XYZTest Funktion deklariert und aufgerufen, ähnlich der Namenskonvention für Testfälle aus der alten JUnit V3.x Ärea.

Innerhalb der Testfunktionen Listing (2) wird mit `assertEquals` Makros – im Listing (3) sind die grundlegenden Ideen zu erkennen – die zu testende Funktionalität überprüft. Das Vorgehen erfolgt ganz analog zu den JUnit Testfällen. Auch hier ist es wieder entscheidend, dass es gelingt den C Code modular zu schreiben, ansonsten lässt er sich nicht gut bzw. gar nicht testen - aber genau das ist die **Bankrotterklärung** eines unprofessionellen Programmierers: der entwickelte Code ist **nicht testbar** – außer als komplette Anwendung, d.h. der Test wird verlagert auf den Endanwender. Ein

Werkzeug wie CUnit erzwingt automatisch und erzieht zu einer modularen und testbaren Software Architektur und trägt somit entscheidend zur Codeverbesserung bei – ganz unabhängig von den entdeckten Fehlern. Dies wird von den meisten Projektleitern und Software Entwicklen übersehen oder verkannt, automatisierte Test sind kein Luxus sondern ein Muss!

```
1  #include "CUnit.h"
2
3  int dummyTest(int argc, char** argv) {
4    char *sx, *sy;
5      int x,y;
6      x = 2;
7      y = 4;
8      assertEqualsI(2, x);
9      assertEqualsI(4, y);
10     assertFalse(x > y);
11
12   sx = "Hugo";
13   sy = "4711";
14   assertNotNull(sx);
15   assertEqualsS(sx, sy);
16
17   return 0;
18 }
```

Listing 2: Anwendung der Makros in dummyTest.c Funktion.

Aufgabe

1. Überlegen Sie sich eine entsprechene `CUnit.h` Headerdatei und eine dazu passende `CUnit.c` Implementierung.

2. Entwickeln Sie einige einfache Testmethoden, z.B. für ihre LED oder Shape Implementierungen und binden sie diese in Ihren CUnit Test ein.

3. Stellen Sie sicher, dass Fehler auch tatsächlich richtig erkannt und gezählt werden. Entsprechende Makros oder Funktionen, wie z. B. `assertEquals`, `assertTrue` und `assertNotNull` etc. sind daher zu entwickeln.

4. Falls ein Testfall fehlschlägt, soll mit den weiteren Tests fortgefahren werden.

5. Sobald dieses grundlegende CUnit Verhalten funktioniert und getestet ist, gilt es auch unvorhergesehene Fehlersituationen *abzufangen*, wie z.B. ein *Segmention fault*.

Diese Fehler werden nicht mittels `assertEquals` oder `fail` ausgelöst, sondern durch unerlaubte Speicherzugriffe, z.B. einen dereferenzierten Nullzeiger. Hierzu müssen entsprechende *ErrorHandler* per `signal`, `setjmp` und `longjmp` Funktionen gesetzt und in die CUnit Anwendung mit eingebaut werden, so dass auch nach einem fatalen Fehler die CUnit Anwendung weiter läuft. Die entsprechenden Headerdateien sind `<setjmp.h>` und `<signal.h>`. Der abgebildete Testlauf provoziert z.B. die Speicherverletzung 11 in einer der Testfunktionen und der CUnit Test läuft anschließend geordnet weiter und protokolliert diesen als FATAL.

```
1
2  /**
3   * Internal error counting function for assertXXX methods.
4   * @param msg to report
5   */
6  extern void cunit_report_error(const char* msg);
7  extern void cunit_report_failure(const char* msg);
8
9  /**
10  * raise a failure situation with message.
11  */
12 #define fail(msg)                {                          \
13      char _test_buf[BUFSIZE];                               \
14      sprintf(_test_buf,"FAIL %s:%d %s \n",                  \
15      __FILE__, __LINE__, msg);                              \
16       cunit_report_failure (_test_buf);                     \
17 }
18
19 /**
20  * check that the given expression is true.
21  */
22 #define assertTrue(expr)      {                          \
23    if( !(expr) ) {                                       \
24      char _test_buf[BUFSIZE];                            \
25      sprintf(_test_buf,"ERROR %s:%d %s \n", \
26      __FILE__, __LINE__, #expr);                         \
27      cunit_report_error(_test_buf);                      \
28    }                                                     \
29 }
```

Listing 3: Einige Makros und Deklarationen der CUnit.h Header Datei.