School of Engineering
InES Institute of Embedded Systems

MC1
Praktikum 13

Host-only based Unit Tests

1 Einleitung

In diesem Praktikum erlernen Sie den Umgang mit Host-only based Unit Tests auf verschiedenen Abstraktionsstufen. Das Praktikum zeigt Ihnen, wie Sie den PC für die Verifikation von Embedded Software einsetzen können. Wir verwenden als Beispiel das aus CT2 bekannte Ampelmodul (Abbildung 1) und wollen mit den zu erstellenden Tests die zugehörige Embedded Software testen. Die Tests und die Embedded Software werden beide auf dem Host, d.h. dem PC ausgeführt. Die ursprünglich für den Cortex-M4 geschriebene Firmware muss dazu für den PC kompiliert werden. Als Testumgebung verwenden wir das Framework CppUTest (http://cpputest.github.io/index.html).

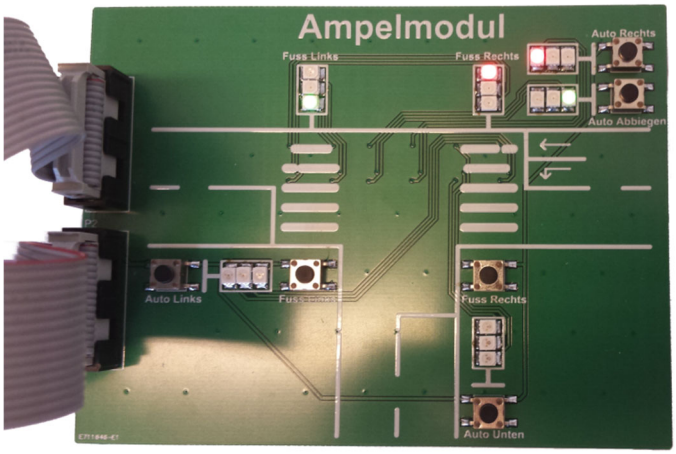

Abbildung 1: Ampelmodul

2 Lernziele

- Sie können Host-only basierte Unit Tests implementieren und anwenden
- Sie können auf verschiedene Arten Tests automatisieren und die zugehörigen Resultate ausgeben
- Sie können durch eigene Tests Fehler im Programm finden

3 Funktion und Aufbau der Verkehrsampel

Die genaue Beschaltung des Ampelmoduls finden Sie im Anhang.

Die vorgegebene Software besteht neben dem Modul *timer* aus drei Modulen:

- *action_handler*
  Hier werden wir testen, dass die Lichtsignale richtig angesteuert werden können.

- *event_handler*
  Dieses Modul generiert die Events für die FSM. Wir werden überprüfen, dass Tastendrücke und Timeout Events korrekt detektiert werden.
- *state_machine*
  Die implementierte FSM beschreibt den Ablauf der ganzen Lichtsignalsteuerung, d.h. hier wird bestimmt, wer wann grün, gelb oder rot hat. Hier werden wir testen, dass Übergänge korrekt stattfinden und keine ungültigen Events zu Zustandsänderungen führen. Sie finden das Zustandsdiagramm im Anhang.

Kompilieren Sie die vorgegebene Software und probieren Sie diese auf dem Ampelboard aus.

4 Host-only based Unit Tests

Registeradressen

Eine der Herausforderungen ist, dass Sie in Ihrer Firmware auf Register zugreifen, welche auf Ihrem PC nicht existieren. Die Lösung dieses Problems ist, dass Speicherplatz in Ihrer Testumgebung reserviert wird, an welchem die Register im Speicher des PCs residieren. Diese Massnahme sehen Sie in Ihrem Projekt unter `unittest/arm_peripherals.c/h`.

Danach müssen die Registeradressen auf diese neuen Bereiche abgebildet werden. Dies wird erreicht, indem die vorhandene Register-Definitionsdatei auf die neuen Adressen angepasst wird, die sich beim Linken ergeben. (Bei Interesse nach erfolgreichem Praktikum siehe lokalen Projektordner `./unittest/reg_stm32f4xx.h`)

Testhierarchien

Tests auf verschiedenen Hierarchiestufen bedeuten, dass Sie einmal sehr nah an der Hardware testen und einmal etwas stärker abstrahiert testen. Sie erhalten dazu einen Programmrahmen mit diversen vorgegebenen Test-Modulen, sowie den Quellcode der zu testenden Anwendung.

MinGW

Für die Ausführung des kompilierten Codes auf dem PC verwenden wir die *MinGW* Umgebung (http://www.mingw.org/). Auf den Rechnern im Labor ist diese installiert. Falls Sie das Praktikum auf Ihrem eigenen Rechner durchführen, finden Sie eine Beschreibung der Installation auf OLAT.

Start der Tests

Sie können die Tests direkt aus Keil uVision heraus über das Menu starten. Dazu müssen Sie einen Menüpunkt in Ihrer Entwicklungsumgebung definieren (unter Tools->Customize Tools Menu). Fügen Sie einen Punkt "CppC Test" mit folgendem Befehl hinzu:

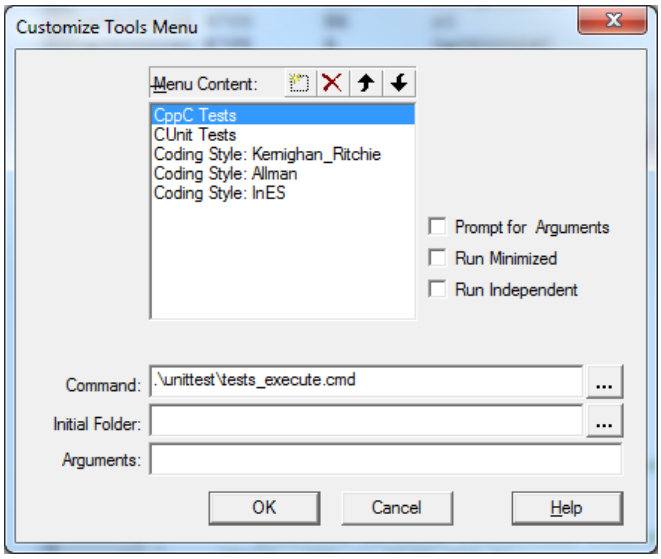

Abbildung 2: Definieren der Unittests als Menupunkt

Anschliessend führen Sie die Tests gemäss Abbildung 3 aus.

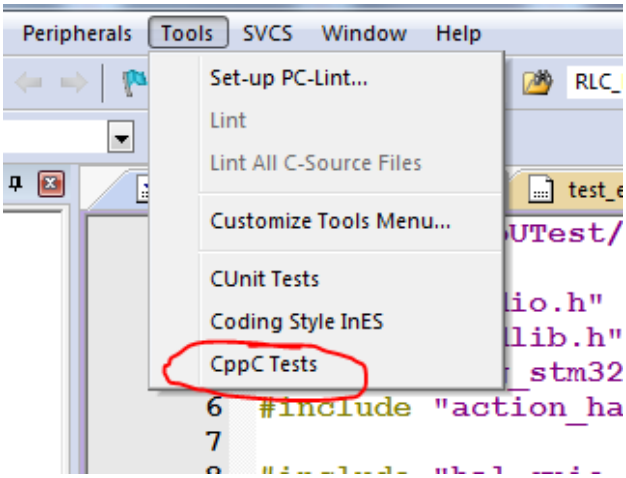

Abbildung 3: Start der Unit Tests über das Keil uVision Menu

Das Resultat wird Ihnen im Keil Output Window angezeigt.

5 Aufgaben

Verwenden und ergänzen Sie für die folgenden Aufgaben in Keil uVision die vorgegebenen Tests im Verzeichnis **`unittest`**.

In diesem Praktikum werden wir nur einzelne ausgewählte Tests implementieren. Es wird keine vollständige Testabdeckung angestrebt. Im Code sind einige Fehler eingebaut, welche Sie mit Ihren Tests finden werden. Am Ende des Praktikums sollten Sie alle Ihre Unit Tests vorführen können und die entsprechenden Tests sollten auf Grund der eingebauten Fehler ein *fail* ergeben.

Ein CppUTest ist wie folgt aufgebaut: Der Testgruppe **`TEST_GROUP(FirstTestGroup)`** wird mit **`TEST(FirstTestGroup, NameOfFirstTest)`** ein Test zugeordnet werden. Dabei zeigt der erste Eingabeparameter die Zugehörigkeit zu der Gruppe **`FirstTestGroup`** und der zweite Parameter ist der eigentliche Name des Tests. Mit **`TEST`** können somit weitere Tests einer Testgruppe angehängt werden.

CppUTest gibt bei Fehlern die Werte von Variablen nicht aus. Beim Vergleich

```
CHECK_TEXT( state == CAR_S, „should be same“ );
```

wissen Sie nicht, was der aktuelle Wert von “state” ist. Ausgegeben wird im Fehlerfall nur

```
CHECK(state == CAR_S) failed
```

Benutzen Sie **`printf()`** für diesen Fall und für Situationen, wo Sie mehr Information benötigen. Mit **`printf()`** sehen Sie die von Ihnen geschriebenen Ausgaben direkt im unteren Fenster der Entwicklungsumgebung nach dem Testdurchlauf. Also zum Beispiel:

```
printf(„state ist: 0x%04X\n“, state);
```

gibt den Wert **`state`** als mindestens 4-stellige Hexzahl aus gefolgt von einem Newline.

Wichtig ist, dass Sie die Ausgabe vor dem **`CHECK_TEXT()`** verwenden, da bei Abbruch des Tests keine weiteren Befehle ausgeführt werden.

5.1 Tests des Moduls *action_handler*

In dieser Aufgabe testen wir, ob wir die diversen Ampeln korrekt ansteuern können. Die Ansteuerung erfolgt über die Funktion **`ah_set_signal()`** im Modul *action_handler*. Die Tests werden in der Datei **`test_action_handler.c`** implementiert. Es sind hardware-nahe Tests mit einem tiefen Abstraktionsgrad.

Machen Sie sich mit der zu testenden Funktion vertraut. Sie finden die Beschaltung der Ampeln im Anhang. Der gegebene Test **`port_access`** dient Ihnen als Beispiel für einen Unit Test. Die Controllability und Observabilty erfolgt über die Funktionen **`hal_gpio_output_write()`** und **`hal_gpio_output_read()`**.

Sie sehen zwei Funktionen in der **`TEST_GROUP`** definition: **`setup()`** und **`teardown()`**. Diese zwei Funktionen werden pro Test je einmal aufgerufen. **`setup()`** vor der Ausführung und **`teardown()`** nach der Testausführung. In diesem Modul wird der Port vor jeder Ausführung zurückgesetzt.

Implementieren Sie die folgenden Tests gemäss Beschreibungen in **`test_action_handler.c`**:

- **`single_signal_color_set`**
- **`single_signal_dark_set`**
- **`all_signal_green_set`**

Setzen Sie der besseren Überstichst halber am Ende eines jeden Tests eine neue Zeile mit **`printf("\n")`** ein.

5.2 Tests des Moduls *event_handler*

In dieser Aufgabe testen wir, ob die Events (Timeouts und Tastendrücke) richtig detektiert werden. Die Detektion aller Events erfolgt über die Funktion **`eh_get_event()`** im Modul *event_handler*. Die Tests werden in der Datei **`test_event_handler.c`** implementiert.

Machen Sie sich mit der Detektion von Events in **`eh_get_event()`** vertraut. Sie finden die Beschaltung der Eingänge im Anhang. Wann wird ein **`TIME_OUT`**, wann ein **`EV_CAR_W`** Event generiert?

Implementieren Sie die folgenden Tests gemäss Beschreibungen:

- **`test_no_timeout_event`**
- **`test_timeout_event`**
- **`test_edge_detection`**

Für das Setzen bzw. Zurücksetzen eines Pins müssen Sie direkt am entsprechenden Port das Input Data Register (**`IDR`**) verwenden, da es ein Input Port ist und Sie diesen nicht über die Funktionen **`hal_gpio_bit_set()`** bzw. **`hal_gpio_bit_reset()`** steuern können. Z.B. **`GPIOB->IDR = 0x01;`**

5.3 Tests der gesamten FSM

In dieser Aufgabe testen wir die FSM als Gesamtsystem, d.h das Modul *state_machine* zusammen mit den Modulen *action_handler* und *event_handler*. Der Test findet damit auf einer höheren Abstraktionsstufe statt.

Über HAL-Funktionen in der Testbench steuern wir (control) die GPIOB Pins, d.h. die Buttons und beobachten (observe) die Ausgaben auf den GPIOA Pins, d.h. den Ampeln. Den Zustand der FSM können wir über die Funktion **`fsm_set_state()`** setzen (control) und über den Rückgabewert der Funktion **`fsm_handle_event()`** beobachten (observe).

Implementieren Sie die folgenden Tests gemäss Beschreibungen:

- **`car_e2_ped_w_ev_car_e1`**
- **`car_e2_ped_w_ev_ped_e`**
- **`car_e2_ped_w_to_invalid`**

Verwenden Sie für die Implementation Ihrer Tests das Zustandsdiagramm im Anhang.

6 Bewertung

Das Praktikum wird mit maximal 3 Punkten bewertet:

- Aufgabe 5.1 Tests des Moduls *action_handler* 1 Punkt
- Aufgabe 5.2 Tests des Moduls *event_handler* 1 Punkt
- Aufgabe 5.3 Tests der gesamten FSM 1 Punkt

Punkte werden nur gutgeschrieben, wenn die folgenden Bedingungen erfüllt sind:

- Der Code muss sauber, strukturiert und kommentiert sein.
- Die Tests sind softwaretechnisch sauber aufgebaut.
- Die Funktion der Tests wird erfolgreich vorgeführt.
- Der/die Studierende muss den Code erklären und zugehörige Fragen beantworten können.
- Der Zusatzpunkt wird nur vergeben, wenn alle anderen Aufgaben gelöst sind.

7 Anhang

7.1 Hardwarebeschaltung des Ampelmoduls

Abbildung 4 zeigt die in der Software verwendeten Elemente und ihre Bezeichnungen.

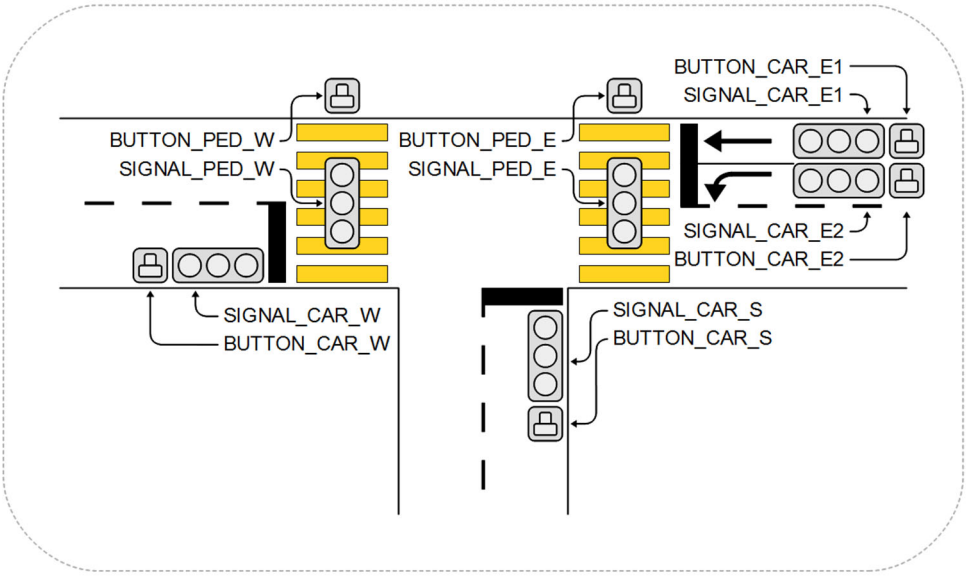

Abbildung 4: Bezeichnung der Ampelelemente

Abbildung 5 und Abbildung 6 zeigen die Schemas der Ampelmoduleingänge und -ausgänge.

Die Eingänge sind „active-high“. Eine logische „**1**“ am Eingang „BUTTON_CAR_S“ bedeutet, dass der entsprechende Taster gedrückt wurde bzw. dass ein Auto an diesem Punkt steht.

Für die Ausgänge ist unten eine Wahrheitstabelle abgebildet.

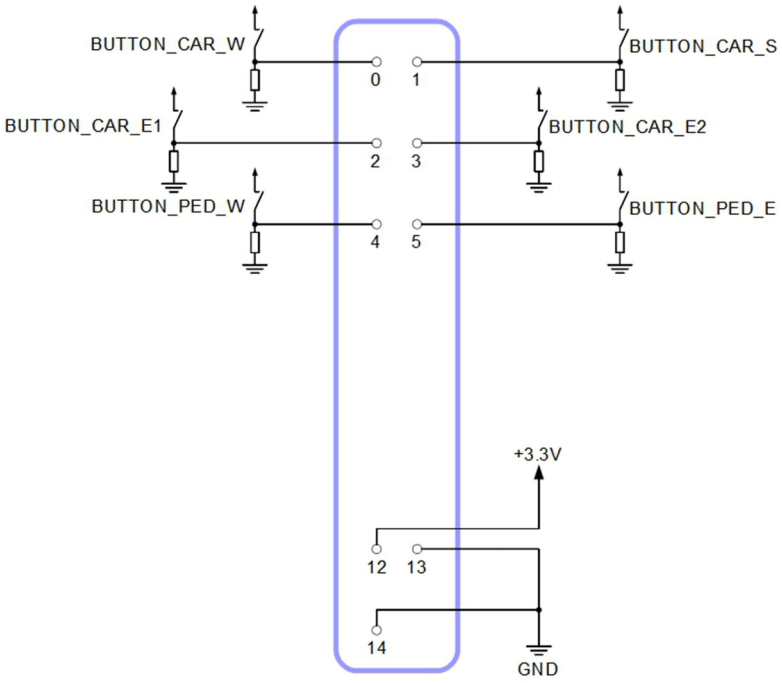

Abbildung 5: Schema der Ampelmoduleingänge

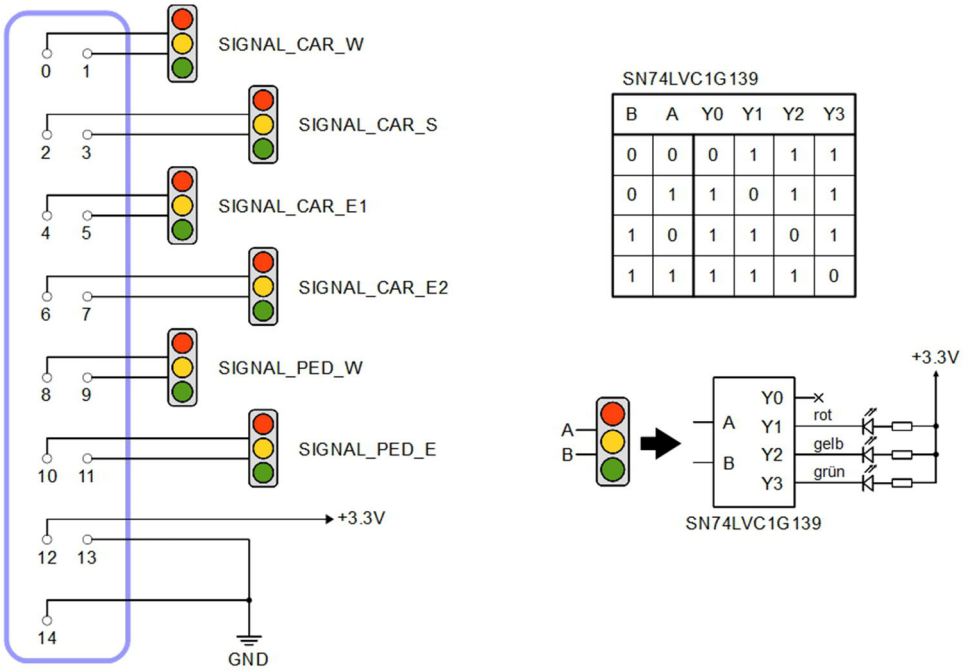

Abbildung 6: Schema der Ampelmodulausgänge

7.2 Zustandsdiagramm

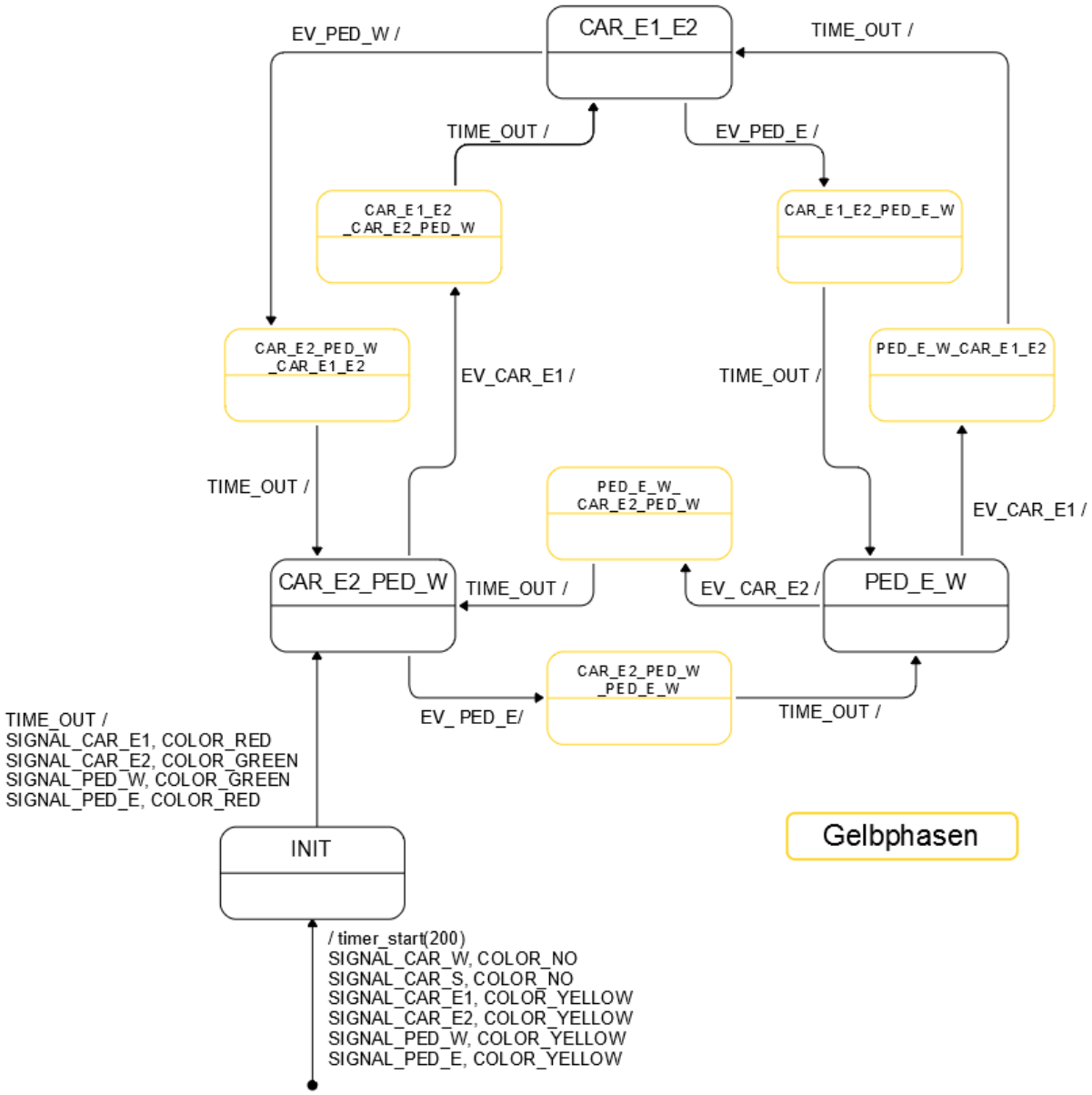

Abbildung 7: Zustandsdiagramm