Embedded Systems - Praktikum 7

Morsecode

Malte Riechmann, André Kirsch

Aufgabe 1

Um die Zustandsmaschine 'beep-test' umzusetzen, haben wir zuerst zwei Klassen für die Buttons und den Buzzer erstellt. Über eine Template Konstante geben wir den Pin an, an den der Aktor bzw. Sensor an das TivaC Board angeschlossen wurde. Über *state()* können wir bei den Buttons den aktuellen Status abfragen. Der Buzzer lässt sich über *setActive(uint8_t state)* aktivieren.

```
template <const uint8_t PORT_BTN>
class Button {
public:
  Button() {
    pinMode(PORT_BTN, INPUT);
  }

  unsigned int state() {
    if (digitalRead(PORT_BTN)) {
      delay(100);
      if (digitalRead(PORT_BTN)) {
        return HIGH;
      }
    }
    return LOW;
  }
};

template <const uint8_t PORT_BUZZ>
class Buzzer {
public:
  Buzzer() {
    pinMode(PORT_BUZZ, OUTPUT);
  }

  void setActive(uint8_t state = HIGH) {
    digitalWrite(PORT_BUZZ, state);
  }
};
```

Auch für die States haben wir eine eigene Klasse erstellt. Im Konstruktor können wir eine Methode übergeben, die aufgerufen wird, wenn man den State betritt.

```
void activateBuzzer() {
    buzzer.setActive(HIGH);
}

void deactivateBuzzer() {
    buzzer.setActive(LOW);
}

State state00(deactivateBuzzer);
State state01(deactivateBuzzer);
State state10(deactivateBuzzer);
State stateBeep(activateBuzzer);
```

Dies nutzen wir, um Funktionen zum Aktivieren oder Deaktivieren des Buzzers zu übergeben.

Mithilfe der Methode *setStates* können wir dem State weitere States übergeben, in die bei gewissen Events gewechselt werden soll.

```
void setStates(State *states[], int stateNum, int events[]) {
    this->nextStates = states;
    this->stateNum = stateNum;
    this->events = events;
}
```

Als letzte Methode besitzt die Klasse *nextState*. Damit kann dem State ein Event übergeben werden, wodurch ein Wechsel zu einem nächsten State stattfinden soll:

```
State *nextState(int event) {
    for (int i = 0; i < stateNum; i++) {
        if (events[i] == event) {
            nextStates[i]->stateChange();
            return nextStates[i];
        }
    }
    return this;
}
```

Dabei gehen wir durch alle Folge-States und schauen, ob diesen das gesucht Event zugeordnet wurde. Falls ja, rufen wir einen *stateChange* beim nächsten State auf und geben diesen zurück.

In der Funktion *loop* speichern wir einen statischen Pointer auf den aktuellen State. Desweiteren fragen wir dort ab, ob die Buttons aktiviert wurden und wechseln anhand dessen den State.

Aufgabe 2

Um Morsecode-Übersetzer zu realisieren haben wir zunächst ein Array aus Char-Pointern erstellt, in dem die verschiedenen Codierungen für die Buchstaben stehen. Der Integer-Wert eines Chars stellt dann den jeweiligen Index für die Codierung dar. Die Zahlen, die keine Morse-Zeichen besitzen, haben NULL als Wert. Um die Morse-Zeichen nicht doppelt zu speichern, wird es nur für den Großbuchstaben gespeichert.

```
const char *CHAR_TO_MORSE[128] = {
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, "-.-.--", ".-..-.", NULL, NULL, NULL, NULL, ".----.",
        "-.--.", "-.--.-", NULL, NULL, "--..--", "-....-", ".-.-.-", "-..-.",
        "-----", ".----", "..---", "...--", "....-", ".....", "-....", "--...",
        "---..", "----.", "---...", NULL, NULL, "-...-", NULL, "..--..",
        ".--.-.", ".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
        "....", "..", ".---", "-.-", ".-..", "--", "-.", "---",
        ".--.", "--.-", ".-.", "...", "-", "..-", "...-", ".--",
        "-..-", "-.--", "--..", NULL, NULL, NULL, NULL, "..--.-",
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
};
```

Dieses Array nutzen wir in der abstrakten Klasse AbstractController. Diese besitzt eine Methode `morseString()`, der der auszugebene String übergeben wird. Diese Methode durchläuft den String in einer Schleife für jeden char. Wenn der Char ein Leerzeichen ist wird eine gewisse Dauer gewartet, bevor der der nächste Char verarbeitet wird. Wenn der übergeben Char ein Kleinbuchstabe ist wird auf ihn ein Offset gerechnet, um den ASCII-Wert für den Großbuchstaben zu erhalten. Anhand dieses Wertes wird aus dem oben beschriebenen Array der Morsecode ausgelesen. Der so erhaltene C-String wird nun durchlaufen und für ein '-'-Zeichen wird die abstrakte Methode morseLong(), bzw. für ein '.'-Zeichen die abstrakte Methode morseShort() aufgerufen.

Diese zwei Methoden werden in den erbenden Klassen implementiert.

```
class AbstractController {
public:
  virtual void morseString(String &output) {
    for (int i = 0; i < output.length(); i++) {
      if (output[i] == ' ') {
        delay(pauseBetweenWords);
      } else {
        char c = output[i];
        if (islower(c)) {
          c += ('A' - 'a');
        }
        const char *morse = CHAR_TO_MORSE[c];
        Serial.println(morse);
        for (int j = 0; j < strlen(morse); j++) {
          if (morse[j] == longSignal) {
            morseLong();
          } else if (morse[j] == shortSignal) {
            morseShort();
          }
          delay(pauseBetweenChars);

        }
```

```
      }
    }
  }

  virtual void morseShort() = 0;
  virtual void morseLong() = 0;
};
```

Um das konkrete Morsen über den Buzzer bzw. die Led zu realisieren haben wir die Klasse `Actor` definiert. Diese bekommt im Konstruktor einen Pin, über den der Aktor angesprochen wird, übergeben und setzt diesen im Konstruktor auf `OUTPUT`. Des Weiteren besitzt die Klasse die Methode `setActive` um den Pin auf `HIGH` oder `LOW` zu setzen.

```
class Actor {
public:
  Actor(uint8_t pin) : pin(pin) {
    pinMode(pin, OUTPUT);
  }

  void setActive(uint8_t state = HIGH) {
    digitalWrite(pin, state);
  }

private:
  uint8_t pin;
};
```

Die Klasse Actor wird in der Klasse PinController als Attribut verwendet. Die Klasse erbt von AbstractContoller und implementiert die beiden Abstrakten Methoden, indem es den Pin auf `HIGH` setzt, eine bestimmte Zeit wartet und den Pin dann wieder auf `LOW` setzt.

```
class PinController : public AbstractController {
public:
  PinController(Actor &actor) : actor(actor) { }
  void setActor(Actor &actor) {
    this->actor = actor;
  }
  void morseShort() {
    actor.setActive(HIGH);
    delay(shortDelay);
    actor.setActive(LOW);
  }
  void morseLong() {
    actor.setActive(HIGH);
    delay(longDelay);
    actor.setActive(LOW);
  }

private:
  Actor &actor;
};
```

Zur Verwaltung der Buttons haben wir die Klasse aus Aufgabe 1 genutzt.

Wenn der eine Button gedrückt wird, wird die Variable `currentActor` inkrementiert und dem PinController Objekt wird die LED als Actor übergeben. Wenn `currentActor` größer als 0 ist, wenn der Knopf gedrückt wird, wird die Variable auf 0 gesetzt und der Buzzer wird als Actor übergeben.

```
  if (buttonB.state() == HIGH) {
    if (currentActor == 0) {
      output.setActor(led);
      currentActor++;
    } else {
      output.setActor(buzzer);
      currentActor = 0;
    }
  }
```

Über das Drücken des zweiten Buttons wird die morseString-Methode mit dem gespeicherten String aufgerufen.

```
  if (buttonD.state() == HIGH) {
    output.morseString(parsedInput);
  }
```

Um den String einzulesen haben wir eine String Variable angelegt und Speicher in Höhe von 128 Chars reserviert.

```
  input.reserve(strLength);
```

Eingelesen wir die Eingabe in der `serialEvent()` - Methode. Wenn neue Daten verfügbar sind werden diese über `Serial.read()` eingelesen und solange an den String angefügt, bis eine gewisse Zeit lang keine neuen Daten mehr gekommen sind oder bis der String die 128 Zeichen als Länge erreicht hat. Kommen dann noch weitere Zeichen, werden diese ignoriert.

```
void serialEvent() {
  if (Serial.available()) {
    char inChar = (char)Serial.read();
    if (input.length() < strLength) {
      input += inChar;
    }
    lastCharTime = millis();
  }
}
```

In der loop-Methode setzen wir den Input auf "", wenn 10ms keine neuen Daten gekommen sind und speichern die bis dahin eingelesenen Daten.

```
if (input.length() > 0 && millis() - lastCharTime > maxCharWaitTime) {
  parsedInput = input;
  input = "";
}
```