LM-EV3-NS-OS-UART-BS-EP

(1. Platz im Wettbewerb “Projektname mit dem besten Akronym”)

Was genau hast du überhaupt gemacht?

Was passiert, wenn man den EV3 Farbsensor an einen Mindstorms anschließt, auf dem Ninjastorms läuft?

Was genau hast du überhaupt gemacht?

Was passiert, wenn man den EV3 Farbsensor an einen Mindstorms anschließt, auf dem Ninjastorms läuft?

Bisheriges Verhalten:

Eine ganze Menge NICHTS.

Was genau hast du überhaupt gemacht?

Was passiert, wenn man den EV3 Farbsensor an einen Mindstorms anschließt, auf dem Ninjastorms läuft?

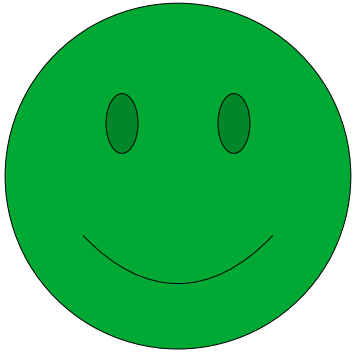
Bisheriges Verhalten:

Eine ganze Menge NICHTS.

LM-EV3
NS-OS
UART-BS
EP

Jetziges Verhalten:

Etwas mehr.



Der Plan

Fabians Plan (mit Erfolgsgarantie!):

1. von UART Sensoren lesen und schreiben
2. Kommunikationsprotokoll auslesen und analysieren
3. Funktionen zur einfachen Kommunikation und dem Auslesen von Daten einbauen

Eigentlich ja nicht so schwer,

wenn es nicht schon bei Schritt 1 Probleme gäbe.

Das Lesen von Daten des Sensors stellt sich als unterwartet schwierig heraus.
Warum?

Das große Problem

Eine einfache Möglichkeit über die Ports mithilfe von UART zu schreiben ist bereits in Ninjastorms implementiert. (siehe `putchar.c` Datei)

Darauf basierend lässt sich auch relativ einfach eine Möglichkeit bauen, Daten zu lesen. (siehe `getchar.c` Datei)

Das große Problem

Damit haben wir eigentlich alles was wir für eine grundlegende Kommunikation per UART brauchen.

Das große Problem

Damit haben wir eigentlich alles was wir für eine grundlegende Kommunikation per UART brauchen.

Aber: Das ganze funktioniert nur auf Port 1 und nur, wenn ein Terminal angeschlossen ist!

Da auf Port 1 aber immer das Terminal angeschlossen ist, nützt das relativ wenig.

Was haben wir denn als Recherchemöglichkeiten?

- Originalen Lego Mindstorms Hardware und Firmware Developer Kits (eig. Nur eine sehr grundlegende Dokumentation)

  + Hardware Schematics vom Brick und Sensoren
- Originalen Lego Mindstorms Source Code

  (insbesondere die `d_uart.c` Datei)
- Diverse Dokumentationen zu UART

Aber auch das bringt Probleme mit

- Hardwareimplementierungen von UART sind nicht standartisiert und jeder Chip hat eine leicht andere Umsetzung
- Die Dokumentationen des Mindstorms haben fast keine Informationen zum Lesen und Schreiben mit UART und fokussieren sich mehr auf das Protokoll

Aber auch das bringt Probleme mit

- Damit bleibt nur der originale Source Code:

Aber auch das bringt Probleme mit

- Damit bleibt nur der originale Source Code:
  - nur schlecht kommentiert
  - > 4000 Zeilen an Code
  - Seltsamme Namensgebungen

    (NACK Byte, “Not acknowledge byte”: wird gesendet um zu zeigen, dass Nachrichten korrekt angekommen sind.)

Aber auch das bringt Probleme mit

- Damit bleibt nur der originale Source Code:
  - `Device1TimerInterrupt1` Funktion

Aber auch das bringt Probleme mit

- Damit bleibt nur der originale Source Code:
  - `Device1TimerInterrupt1` Funktion
  - Ungenaue / Fehlerhafte Dokumentationen

```
#define   UART_TIMER_RESOLUTION       10              // [100uS]

#define   UART_BREAK_TIME             1000            // [100uS]
#define   UART_TERMINAL_DELAY         20000           // [100uS]
#define   UART_CHANGE_BITRATE_DELAY   100             // [100uS]
#define   UART_ACK_DELAY              100             // [100uS]
#define   UART_SHOW_TIME              2500            // [100uS]

#define   UART_WATCHDOG_TIME          1000            // [100uS]
```

Aber auch das bringt Probleme mit

- Damit bleibt nur der originale Source Code:
  - Schlecht geschriebener Code

    (Code wird teilweise mehrfach ausgeführt)
  - Komplizierte Datenstrukturen
  - Viele extra Debugbefehle und Präprozessoranweisungen
    - Teilweise auch Behandlung von Fake Test Pins

Der Code

- UART Register sind mithilfe eines C-Structs implementiert
  - Vorteil: Einfache Verwendung und gut Leserlich

```
uart_ports[port]→thr = data;
```

- Zudem werden in der Initialisierungsphase noch diverse Pins initialisiert.

  (Ähnlich wie in `gpio.c` Datei)

Der Code

- Mithilfe der Register wurden primitive Read / Write Funktionen erstellt.

```
unsigned char uart_read(sensor_port_id port) {
   while ((uart_ports[port]→lsr & 0b1) == 0) ;
   return uart_ports[port]→rbr;
}
```

- Darauf basierend wurden Basisfunktionen zur Kommunikation mit den Sensoren gebaut, welche je nach Anwendung verwendet werden können.

Der Code

- Als Beispiel: Code des Farbsensors:

```
void demo_ev3_color(void) {
  uartsensor_setup_color(SENSOR_PORT_2);
  uartsensor_change_mode(SENSOR_PORT_2, COL_COLOR);
  while(1) {
    unsigned char d = uartsensor_read_data(SENSOR_PORT_2);
    printf("%i\n", d);
    uartsensor_send_nack(SENSOR_PORT_2);
  }
}
```

Der Code

- Als Beispiel: Code des Farbsensors:

```
int uartsensor_setup_color(sensor_port_id port) {
    uartsensor_setup(port);
    uartsensor_wait_init(port, EV3_COLOR_ID);
    uartsensor_dump_bytes(port, EV3_COLOR_DUMP);
    if (!uartsensor_respond_ack(port)) {
        return 0; // Some Failure occured
    }
    uartsensor_set_middle_bitrate(port);
    uartsensor_wait_data(port, UARTSENSOR_MODE_DEFAULT);
    return 1;
}
```

Erweiterbarkeit

Wenn neue Sensoren hinzugefügt werden wollen:

1. Mit der Hexdump Funktion das Init Protokoll ausgeben lassen und analysieren
2. Basierend auf der bisherigen Setup Funktion eine neue Anlegen
3. Folgende Werte in der neuen Setup Funktion entsprechend anpassen:

   (mit Informationen aus 1.)

   - `EV3_COLOR_ID` → zur Device ID des neuen Sensors ändern
   - `EV3_COLOR_DUMP` → zur Länge des Initialisierungsprotokolls des neuen Sensors ändern
   - `uartsensor_set_middle_bitrate` entsprechend ändern, wenn nötig

Warum nicht leichter?

Mehrere Gründe:

- Zeit
- Bisher nicht nötig
- Fehlende Funktionalitäten in Ninjastorms
- Macht den Code deutlich komplexer

Weitere Verbesserungsmöglichkeiten

- Bisher nur Lesen von Werten zwischen 0-255 möglich (genau 1 Byte.)

  (Reicht aus, da fast alle Sensoren nur Werte zwischen 0-100 senden.)
- Initialisierungscode könnte verbessert werden

Weitere Ziele

Cubestormer III