Technische Informatik IV : Praktikum

Protokoll zu Aufgabe A13

von Alexander Steen, Max Wisniewski

Vorbereitung

In dieser Übung kamen keine neuen Befehle hinzu. Wir haben uns in Aufgabe *A12* schon mit dem *WIP Development Guide* im Bereich *TCPCLientSockets* beschäftigt. Die Steuerung der **LEDs** und Taster haben wir schon in vorherigen Aufgaben erledigt. Wir werden hier trotzdem nochmal die Befehle auflisten, die wir verwendet haben, aber im Gegensatz zur erstmaligen Beutzung nicht nochmal alles im Detail beschreiben.

Aufgaben

- Die Zustandsänderung eines Tasters (Taster0...Taster3) soll an einen Webserver übertragen werden. Der Server ist **hwp.mi.fu-berlin.de.** Genutzt werden die Ports **50001...50009**. Die Zuordnung der Ports zu den Arbeitsplätzen ist
  - **50008 : HWP8**
- Für die Gruppen: ... **HWP8**.... wird die Zustandsänderung auf den verlinkten HTML-Formularen sichtbar gemacht. Zur Aktualisierung der Anzeige drücken sie [F5]. User und Passwort sind mit dem Praktikumszugang identisch.
- Die vom Modul über die TCP-Verbinung zu verschickende Zeichenkette muss folgendes Format haben: **SET:<LED0><LED1><LED2><LED3><Taste0><Taste1><Taste2><Taste3>**
  z.B.: **SET:00010111**; (0-Aus, 1-Ein)
- Wird im HTML-Formular die Taste Senden gedrückt, werden die Eingaben aus dem Formular in einer Zeichenkette(wie oben erläutert) an das Modul zuückgeschickt.
- Werten Sie die empfangene Zeichenkette aus und setzten Sie den Zustand der LEDs entsprechend um.
- Bitte besuchen Sie nur ihre Seite, damit die anderen Gruppen nicht gestört werden.

Dokumentation

wip_read liest aus einem Socket eine Festgelegte Anzah von Bytes in einen Buffer
wip_read(Socket* s, ascii* buffer, u16 size). [Socket steht hier stellvertretend für das struct eines Sockets)

wip_write schreibt eine Nachricht mit angegebener Anzahl von Bytes auf ein Socket. Diese wird dann raus geschickt.
wip_write(Socket* s, ascii* buffer, u16 size).

wi_close Schließt ein TCPSocket wieder.
wip_close(Socket* s);

wip_TCPClientCreateOpts erstellt ein neues TCP Clienten Socket. Die optionale Variante ist von nöten, da das Socket sonst nicht automatisch gebindet wird.

adl_atCmdCreate führt ein ATCommando aus, wie man es auf der Commandozeile auch eintragen kann

adl_atUnSoSubscribe registriert einen Handler auf ein Unsolicid Responses. Die meisten werden auf AT Commandos als Antwort geworfen.

Durchführung

Um das GPRS zu benutzen musste die Flag in *config.h* auf 1 gesetzt werden. Dies haben wir schon in Aufgabe *A12* eingeschaltet. Wenn wir das Modul einschalten, ist das erste was wir tun müssen ein Socet zu erstellen und die Taster zu aktivieren:

Algorithmus 1 In der main werden die beiden nötigen Eventhandler angelegt und sowohl LED als auch Taster aktiviert.

```
1  //Globale Variablen
2  wip_channel_t channel;
3  ascii sendbuffer[12];
4  u8 status[8];
5
6  //Main
7  void main(void){
8      led_init();
9      u8 i;
10     for (i = 0; i < 8; ++i) {
11         status[i] = 0;
12     }
13     channel = wip_TCPClientCreateOpts("hwp.mi.fu-berlin.de", // Host
14                 50008, // Port
15                 echo_response, // Handler
16                 NULL, WIP_COPT_PORT, 13338, WIP_COPT_END);
17
18     // keymeldungen aktivieren
19     adl_atCmdCreate("AT+CMER=,1", FALSE, (adl_atRspHandler_t) NULL, NULL);
20     // keyhandler anlegen
21     adl_atUnSoSubscribe("+CKEV:", (adl_atUnSoHandler_t) keyhandler);
22 }
```

Nachdem wir nun alle Test, LEDs und Sockets angestellt haben, können wir uns um das Senden der Nachrichten an den Server kümmern. Im status (ganz oben) finden wir schon die Zahlen (0-Aus, 1-Ein), die wir an den Server schicken werden. Wir speichern dazu immer unseren jetzt Zustand zwischen und ändern nur das, wass wir von den Tasten oder vom Server gesagt bekommen.

Da wir nur senden sollen, wenn wir auf eine Taste gedrückt haben, ist die Logik des Sendens im keyhandler.

Algorithmus 2 Im Keyhandler wird nachgeguckt, welche Taste gedrückt wurde und der neue Status an den Server geschickt.

```
1  bool keyhandler(adl_atUnsolicited_t *paras)     {
2      ascii tasteStr[2];
3      ascii tasteFirst[2];
4      ascii statusStr[2];
5      // parameter auslesen
6      wm_strGetParameterString(tasteStr, paras->StrData, 1);
7      wm_strncpy(tasteFirst,tasteStr,1);
8      wm_strGetParameterString(statusStr, paras->StrData, 2);
9      adl_atSendResponse(ADL_AT_RSP, "\r\nTaste ");
10     adl_atSendResponse(ADL_AT_RSP, tasteFirst);
11     adl_atSendResponse(ADL_AT_RSP, " wurde  auf ");
12     adl_atSendResponse(ADL_AT_RSP, statusStr);
13     adl_atSendResponse(ADL_AT_RSP, " gesetzt\r\n");
14     status[wm_atoi(tasteStr)+4] = wm_atoi(statusStr);
15
16     wm_sprintf(sendbuffer, "SET:%d%d%d%d%d%d%d%d",
17         status[0], status[1], status[2], status[3],
18         status[4], status[5], status[6], status[7]);
19     adl_atSendResponse(ADL_AT_RSP, sendbuffer);
20     wip_write(channel, sendbuffer, 12);
21     return FALSE;
22 }
```

Der größte Teil, in dieser Funktion ist eine Statusausgabe auf der Konsole. Wir holen uns wie gewohnt von der Aufgabe mit den Tastern *A8* wissen den ersten Teil der Antwort herraus und kopieren davon den ersten Teil nochmal herraus. Dies ist nötig, da in der Antwort nach der ersten Zahl im String immer noch etwas stand, das nicht dort hin gehörte. Deswegen holen wir uns noch den ersten Teil davon herraus.

Im zweiten Teil steht nun der Status der Taste korrekt.
Nun parsen wir die Taste und den Status und schreiben das ganze in den status hinein. Die Plus 4 im Index verschiebt die Taste (0..3) auf die Korrekte Position (sie Protokollspezifikation in der Aufgabe). Als nächstes betrachten wir die TCPHandler *echo_response*.

Algorithmus 3 Der Handler echo_response achtet nur darauf, wenn etwas gesendet wurde. Sollte der Server den Channel beendet, wird das Socket geschlossen.

```
1  void echo_response(wip_event_t *event, void *ctx)   {
2      s32 answer;
3      ascii buffer[12];
4
5      switch (event->kind)        {
6          case WIP_CEV_OPEN:
7              break;
8          case WIP_CEV_READ:
9              do              {
10                 answer=wip_read(event->channel, buffer, sizeof(buffer) - 1);
11                 buffer[answer] = '\0';
12             // Terminieren, falls es ein String ist
13                 adl_atSendResponse(ADL_AT_RSP, buffer);
14                 updateStatus(buffer);
15                 }
16             while (answer == sizeof(buffer) - 1);
17             break;
18         case WIP_CEV_WRITE:
19             break;
20         default:
21             wip_close(channel);
22             ERROR("ERROR");
23             break;
24         }
25 }
```

Wir warten darauf, dass wir eine Nachricht über das Socket bekommen. Haben wir eine Nachricht im buffer, rufen wir die Funktion *updateStatus* auf. Der Handler macht an sich nicht mehr, außer auf ein schlileßen des Channels zu achten oder sonstige Fehler auf der Leitung. Betrachten wir nun *updateStatus* und innerhalb davon *useStatus*.

Algorithmus 4 Die beiden Funktionen, die ein über TCP empfangens Packet den Status anpassen.

```
1  void useStatus(){
2      u8 i;
3      for(i = 0; i<4; ++i){
4          if(status[i] == 1) led_on(i);
5          else led_off(i);
6      }
7  }
8
9  void updateStatus(ascii * stat)     {
10     if (wm_strncmp(stat, "SET:", 4) != 0)       {
11         return;
12         }
13     u8 i;
14     adl_atSendResponse(ADL_AT_RSP,"\r\nneuer status:\r\n");
15     for (i = 0; i < 8; ++i)         {
16         if (wm_strncmp(stat + 4 + i, "1", 1) == 0)          {
17             status[i] = 1;
18             adl_atSendResponse(ADL_AT_RSP,"1");
19             }
20         else if (wm_strncmp(stat + 4 + i, "0", 1) == 0)             {
21             status[i] = 0;
22             adl_atSendResponse(ADL_AT_RSP,"0");
23             }
24         }
25     adl_atSendResponse(ADL_AT_RSP,"\r\n");
26     useStatus();
27 }
```

In updateStatus parsen wir aus einem eingegebenen String Charweise den inhalt und schreiben den Inhalt in *status* hinein. Haben wir alle Änderungen eingetragen rufen wir *useStatus* auf, der dafür sorgt, dass alle LEDs (Tasten werden wir programmatisch nicht drücken können, deswegen werden diese ignoriert), die im *status* gesetzt sind ein geschaltet werden und die anderen aus. Mit diesen einfachen Methoden können wir die geforderte Funktionalität umsetzen. Zum letzten Punkt der Aufgabe:

Haben wir nicht getan.

...

Ehrlich!

Auswertung

Wir haben als erstes einmal getestet ob unsere Taster den Status richtig verändern:

```
+GPRS: Verbindung wurde hergestellt

Taste 2 wurde  auf 1 gesetzt
SET:00000010

Taste 2 wurde  auf 0 gesetzt
SET:00000000

Taste 1 wurde  auf 1 gesetzt
SET:00000100

Taste 1 wurde  auf 0 gesetzt
SET:00000000
```

Wenn wir auf eine Taste gedrückt haben, ist die SET Nachricht korrekt verändert worden. HIer schlecht zu zeigen, aber im Browser war diese Veränderung auch sichtbar. Nach dem Drücken von [F5] waren die Checkboxen (warum waren es überhaupt Checkboxen, das macht gar keinen Sinn, da man Tasten nicht Remote setzen kann) der gedrückten Tasten angehakt. Im nächsten Teil haben wir versucht die LEDs Remote zu setzen, dabei sahen wir auf dem Modul die richtigen LEDs aufleuchten und auf der Console

```
SET:01000100 // Empfangene Nachricht
neuer status:01000100 //Gesetzter Status

SET:11110000 //Empfangene Nachricht
neuer status:11110000 //Gesetzter Status

SET:10100101 //Empfangene Nachricht
neuer status:10100101 //Gesetzter Status
```

Wir sehen, dass wir hier eine Nachricht empfangen vom Server und wenn wir alles geparst haben, ist unser neuer Status, der den wir in der Nachricht bekommen haben.

Das stimmt auch mit unseren Beobachtungen der LEDs überein. Sonst gab es in dieser Aufgabe recht wenig zu beobachten.