Contiki Tutorial - praktischer Teil 2

Sven Zehl, Thomas Scheffler

Beuth Hochschule für Technik Berlin

27. September 2013

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Contiki

The Open Source OS for the Internet of Things

-UDP Server-

Cygwin

1. Cygwin starten und in den Contiki Unterordner `/examples/` wechseln (**cd [Ordnername]** öffnet Verzeichnis, **cd ..** springt zurück)
2. neuen Projektordner erstellen mit **mkdir projekt2**, anschließend in den neuen Ordner wechseln (**cd projekt2**)
3. benötigte Dateien anlegen mit **touch projekt2.c Makefile Makefile.projekt2**
4. jetzt kann mit dem Windows Explorer zum Pfad navigiert werden (C:/Cygwin/home/[Benutzername]/contiki-2.6/examples/projekt2/) und die Dateien können mit einem Editor der Wahl bearbeitet werden.
5. Alternative: direkt unter Cygwin mit dem Kommandozeileneditor **vim** arbeiten, dazu einfach unter Cygwin den Befehl **vi [Dateiname]** ausführen.

Contiki Prozesse - benötigte Bibliotheken

1. die neu erstellte Datei **projekt2.c** mit dem Editor der Wahl öffnen.
2. benötigte Bibliotheken einbinden.

```
1 #include "contiki.h"
2 /*enthält alle Contiki spezifischen Makros und Funktionen*/
3 #include <stdio.h>
4 /*Standard Input Output z.B. für printf()*/
5 #include <avr/io.h>
6 /*Input Output speziell für AVR Microcontroller, z.B. die
      Makros für die Ports und Register */
7 #include "contiki-net.h"
8 /*Contiki Netzwerk Bibliothek*/
```

Contiki Prozesse - Prozessdefinition & Autostart

1. Diesmal soll ein UDP Server programmiert werden, welcher bei Paketempfang eine LED zum leuchten bringt und außerdem ein Paket zurücksendet.
2. Dieser soll wieder automatisch nach dem Bootvorgang gestartet werden.

```
1 PROCESS(udp_server_process, "UDP Server");
2 AUTOSTART_PROCESSES(&udp_server_process);
```

Contiki Prozesse - Prozessaufbau - Port PIN Aktivierung

1. Zuerst sind zwei *defines* festzulegen, erstens die maximale Länge eines Pakets und zweitens ein Zeiger auf den Contiki internen *uip_buf*, den Contiki μIP Paket Buffer.
2. Außerdem muss eine Verwaltungsstruktur für die UDP Verbindung angelegt werden (siehe Theorie-Teil)

```
1 #define UDP_IP_BUF   ((struct uip_udpip_hdr *)&uip_buf[
      UIP_LLH_LEN])
2 /*Zeiger auf uIP Buffer*/
3 #define MAX_PAYLOAD_LEN 120
4 /*maximale Payload Länge*/
5 static struct uip_udp_conn *udpconn;
6 /*Verwaltungsstruktur für UDP Verbindung*/
```

Contiki Prozesse - Prozessaufbau

1. Es soll nun der eigentlich Prozess programmiert werden.
2. Prozesse beginnen wie in der Einführung gezeigt in Contiki immer mit der Kopfzeile **PROCESS_THREAD([prozess_name], ev, data)**
3. Der folgende Rumpf des Prozesses wird immer durch das Makro **PROCESS_BEGIN()**; begonnen und durch das Makro **PROCESS_END()**; beendet.
4. zwischen **PROCESS_BEGIN()** und **PROCESS_END()** wird der eigentlich Quellcode des Programms geschrieben.
5. da zur Paketempfangssignalisierung die LED DS1 kurz blinken soll, wird diese hier schonmal als Ausgang geschaltet.

```
1 PROCESS_THREAD(switch_led_on, ev, data)
2 {
3   PROCESS_BEGIN();
4   DDRB |= (1 << PIN6);
5   PORTB |= (1 << PIN6);
6   /*hier kommt der eigentlich Programmcode hin*/
7   PROCESS_END();
8 }
```

Contiki Prozesse - Prozessaufbau - UDP Server

1. Eine neue UDP Verbindung wird mit der Funktion `udp_new(const uip_ipaddr_t *ripaddr, uint16_t port, void *appstate)` aufgebaut (siehe Theorie Teil)
2. Damit der Empfang von jeder IP Adresse möglich ist, wird als erster Parameter NULL übergeben.
3. Schließlich wird mithilfe der Funktion `udp_bind()` festgelegt, dass alle Pakete auf Port 50000 empfangen werden sollen.
4. Die Funktion `UIP_HTONS()` dient hier zur Übersetzung von Host-Byte-Order zu Network-Byte-Order.

```
1 /*Starte neue UDP Verbindung mit IP0.0.0.0 und Port 0*/
2 /* d.h. akzeptiere jede ankommende Verbindung*/
3 udpconn = udp_new(NULL, UIP_HTONS(0), NULL);
4 /*Setze den Port auf dem gelauscht wird auf 50000*/
5 /*HTONS() übersetzt zu Network Byte Order*/
6 udp_bind(udpconn, UIP_HTONS(50000));
```

1. Jetzt kommt eine der Blocking Macros aus dem Theorie-Teil zum Einsatz. Das Makro `PROCESS_WAIT_EVENT_UNTIL()`.
2. Dieses Makro sorgt dafür, dass der Process solange unterbrochen wird bis ein **tcpip_event** eintrifft.
3. Während der Prozess wartet, können andere Prozesses ausgeführt werden. Dies ermöglicht die Parallelverarbeitung.

```
1  while(1) {
2    /* Warte bis ein TCP/IP event eintrifft */
3    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
4    /*Rufe die Handler-Funktion auf*/
5    bsp_udphandler();
6  }
```

1. Ist ein **tcpip_event** eingetroffen, so wird der Prozess weiter ausgeführt und die Funktion **udphandler()** gestartet.
2. Damit nach der Verarbeitung des tcpip_events noch weitere Pakete empfangen werden können, wird diese Prozedur in eine **while(1)-Schleife** eingebunden, was dafür sorgt, dass sie immer wieder wiederholt wird.

```
1  while(1) {
2    /* Warte bis ein TCP/IP event eintrifft */
3    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
4    /*Rufe die Handler-Funktion auf*/
5    bsp_udphandler();
6  }
```

Damit ist der UDP Server Prozess auch schon fertig.

```
1  #include "contiki.h"
2  #include <stdio.h>
3  #include <avr/io.h>
4  #include "contiki-net.h"
5
6  #define UDP_IP_BUF ((struct uip_udpip_hdr *)&uip_buf[
       UIP_LLH_LEN])
7  #define MAX_PAYLOAD_LEN 120
8  static struct uip_udp_conn *udpconn;
9
10 PROCESS(udp_server_process, "UDP Server");
11 AUTOSTART_PROCESSES(&udp_server_process);
12
13 PROCESS_THREAD(udp_server_process, ev, data)
14 {
15   PROCESS_BEGIN();
16   DDRB |= (1 << PIN6);
17   PORTB |= (1 << PIN6);
18   udpconn = udp_new(NULL, UIP_HTONS(0), NULL);
19   udp_bind(udpconn, UIP_HTONS(50000));
20   while(1) {
21     PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
22     udphandler();
23   }
24   PROCESS_END();
25 }
```

1. Was noch fehlt ist die **udphandler()-Funktion**.
2. Diese ist eine ganz normale C-Funktion ohne Rückgabewerte
3. Um zu überprüfen ob auch neue Daten im uIP-Buffer liegen, kann die Funktion **uip_newdata()** genutzt werden.
4. Da eine Antwort versendet werden soll, wird außerdem noch ein Output Buffer angelegt.

```
1 void udphandler(void)
2 {
3   char buf[MAX_PAYLOAD_LEN];
4   if(uip_newdata())
5   {
6   /*Verarbeitung und Antwortgenerierung*/
7   }
8 }
```

Contiki Prozesse - Prozessaufbau - UDP Handler

1. Da wir bei Paketempfang die LED DS1 kurz aufleuchten lassen wollen, ist hier ein guter Zeitpunkt sie einmal einzuschalten.
2. Zum Senden eines Antwortpakets wird natürlich die Adresse vom Sender und dessen Port benötigt.
3. Zum Kopieren von IP Adressen kann die Contiki Funktion **uip_ipaddr_copy()** verwendet werden.
4. Der Port kann einfach direkt in die Verwaltungsstruktur der UDP Verbindung kopiert werden.

```
1  if(uip_newdata())
2  {
3    /*LED ein*/
4    PORTB &= ~(1 << PIN6);
5    /*kopiere Sender Adresse in Zieladresse von
         Verwaltungsstruktur*/
6    uip_ipaddr_copy(&udpconn->ripaddr, &UDP_IP_BUF->
         srcipaddr);
7    /*Kopiere Sender Port in Zielport von
         Verwaltungsstruktur*/
8    udpconn->rport = UDP_IP_BUF->srcport;
9  }
```

Contiki Prozesse - Prozessaufbau - UDP Handler

1. Benötigt wird jetzt noch der Payload der Nachricht. Dieser kann beliebig gewählt werden und wird einfach in **buf** geschrieben.
2. Hierfür kann z.B. die Funktion **sprintf()** verwendet werden.
3. Abgesendet wird das Paket schließlich einfach mit der Funktion **uip_udp_packet_send()**, μIP übernimmt alles weitere.
4. Damit die LED nicht permanent leuchtet, sondern nur kurz blinkt, wird sie nun einfach wieder abgeschaltet.

```
1   if(uip_newdata())
2   {
3    PORTB &= ~(1 << PIN6);
4    uip_ipaddr_copy(&udpconn->ripaddr, &UDP_IP_BUF->
         srcipaddr);
5    udpconn->rport = UDP_IP_BUF->srcport;
6    /*Payload für Antwort generieren*/
7    sprintf(buf, "Ich bin eine Antwort");
8    /*Paket absenden*/
9    uip_udp_packet_send(udpconn, buf, strlen(buf));
10   PORTB |= (1 << PIN6);
11  }
```

1. Damit ist die udp_handler Funktion fertig. Was noch beachtet werden muss, ist dass nach jedem Paketversand die UDP Verwaltungsstruktur wieder zurückgesetzt werden muss um weitere Pakete zu empfangen.

```
1  void udphandler(void)
2  {
3   char buf[MAX_PAYLOAD_LEN];
4   if(uip_newdata())
5    {
6    PORTB &= ~(1 << PIN6);
7    uip_ipaddr_copy(&udpconn->ripaddr, &UDP_IP_BUF->
         srcipaddr);
8    udpconn->rport = UDP_IP_BUF->srcport;
9    sprintf(buf, "Ich bin eine Antwort");
10   /* UDP Verw-Struct zurücksetzen*/
11   uip_udp_packet_send(udpconn, buf, strlen(buf));
12   memset(&udpconn->ripaddr, 0, sizeof(udpconn->ripaddr)
         );
13   udpconn->rport = 0;
14   PORTB |= (1 << PIN6);
15  }
16 }
```

Contiki Prozesse - Makefile

1. Das Programm ist somit fertig geschrieben. Nun die Datei **projekt2.c** abspeichern und die Datei **Makefile** im Editor der Wahl öffnen.
2. Wie in der Einführung muss nun wieder das Makefile geschrieben werden. Es muss die Zielplattform sowie die Zieldatei und der Verweis auf das Makfile.projekt2 gesetzt werden.
3. danach kann **Makefile** abgespeichert werden und sogleich das zweite Makefile, **Makefile.projekt2** geöffnet werden.

```
1 all:
2  ${MAKE} TARGET=avr-zigbit -f Makefile.projekt2 projekt2.elf
```

Contiki Prozesse - Makefile.projekt2

1. Wir befinden uns nun in der Datei **Makefile.projekt2**.
2. Hier muss wieder das Wurzelverzeichnis von Contiki, **IPv6** und das globale Contiki **Makefile.include** festgelegt werden.
3. nach dem Abspeichern, muss wieder in die **Cygwin** Console und weiter in das Verzeichnis von **projekt2** gewechselt werden.

```
1 CONTIKI = ../..
2 UIP_CONF_IPV6=1
3 include $(CONTIKI)/Makefile.include
```

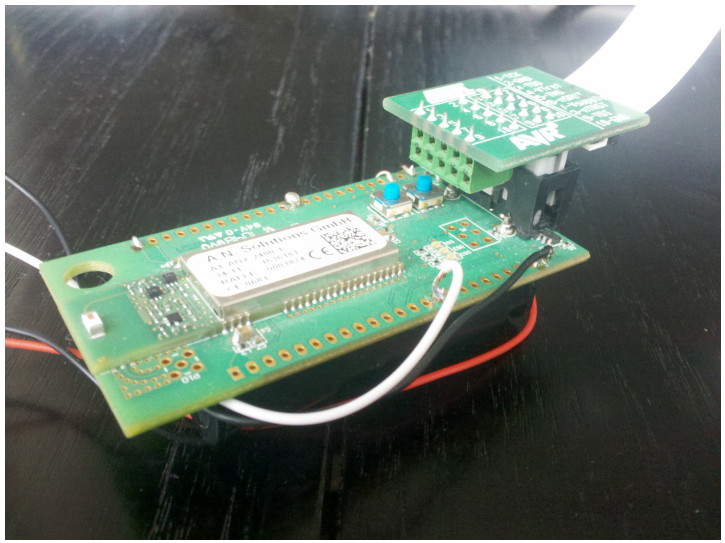
Contiki Prozesse - make und upload

1. Wir befinden uns nun in **Cygwin** und im Ordner **projekt2**
2. Der Befehl **make** startet nun den Kompilierungsvorgang und die Erzeugung des **.elf-Files**.
3. War alles erfolgreich, so sollte automatisch das Programm **avr-size** gestartet werden, welches wiederum folgenden Output liefern sollte:

```
AVR Memory Usage
----------------
Device: atmega1281

Program:   48546 bytes (37.0% Full)
(.text + .data + .bootloader)

Data:       3961 bytes (48.4% Full)
(.data + .bss + .noinit)

EEPROM:        8 bytes (0.2% Full)
(.eeprom)
```

Contiki Prozesse - make und upload

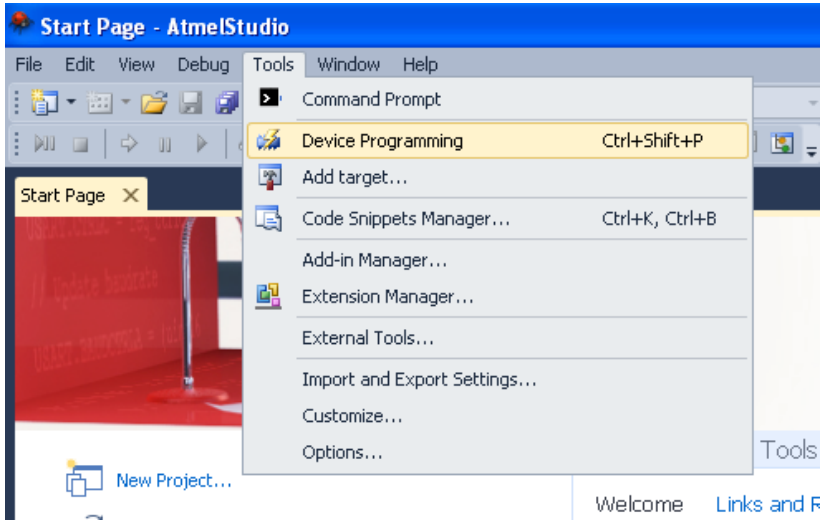
1. Jetzt folgt der Upload auf den Mikrocontroller, dazu den Programmer JTAGiceMKII wie gezeigt mit Entwicklungsboard verbinden:



2. Das Entwicklungsboard sowie den Programmer einschalten und im Anschluss das AVR Studio starten.

Contiki Prozesse - make und upload

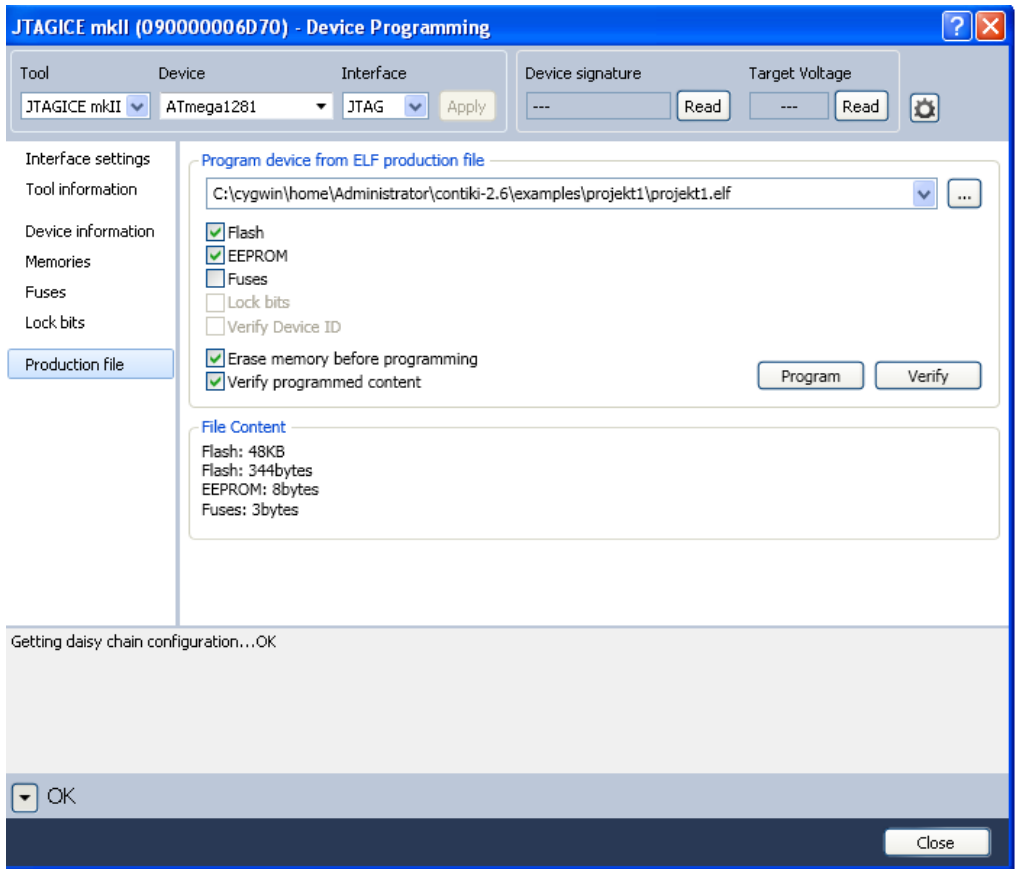
1. Im AVR Atmel Studio im Menü oben auf Tools / Device Programming klicken (siehe folgende Abbildung):



2. Im nun folgenden Dialogfenster, die Einstellungen für den Mikrocontroller wählen und mit *apply* bestätigen(siehe Abbildung unten).

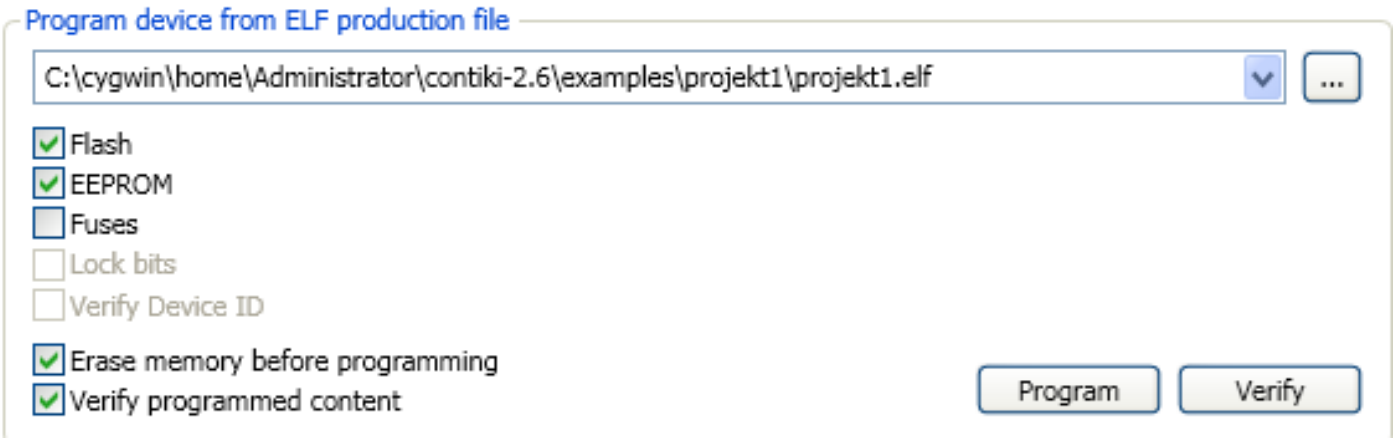
Contiki Prozesse - make und upload

1. Da die zuvor generierte .elf-Datei hochgeladen werden soll, muss nun im linken Auswahlbereich *Production file* gewählt werden. (siehe folgende Abbildung):



Contiki Prozesse - make und upload

1. Weiter muss nun über den Button [...] zum generierten .elf-File navigiert werden bzw. der Pfad eingetragen werden. Außerdem müssen die Haken für *Flash* und *EEPROM* gesetzt werden (siehe folgende Abbildung):



2. Mit einem letzten Klick auf *Program* wird anschließend der Upload gestartet.

Contiki Prozesse - make und upload

1. War der upload erfolgreich, so erscheinen die unten gezeigte Schritte mit *OK* bestätigt.

```
Erasing device... OK
Programming Flash...OK
Verifying Flash...OK
Programming EEPROM...OK
Verifying EEPROM...OK
```

2. Zusatzaufgabe: Verbinden von beiden praktischen Teilen. UDP Server mit Paketempfangs-LED und LED beim Einschalten.

TEST

https://github.com/szehl/contiki_tutorial/blob/ master/examples/contiki_tutorial2%28UDP%29/UDP_Sender/uclient6.c
UDP Client, sendet eine UDP Nachricht an wählbaren Port

Quellen

- https://github.com/contiki-os/contiki/wiki
- http://contiki.sourceforge.net/docs/2.6/
- http://senstools.gforge.inria.fr/doku.php?id=contiki

$\Rightarrow$ alle Beispiele (inklusive Sender Quellcodes für Linux) sind unter https://github.com/szehl/contiki_tutorial zu finden.