methodpark

Der Westwind bringt das Wetter

Operating Systems for the IoT: Linux bekommt Verstärkung

Florian Bauer, Dr. Tobias Kästner

Method Park

Wir Menschen haben Augen und Ohren, um auf die Welt, in der wir leben, zu reagieren. Im Internet der Dinge übernehmen die Wireless Sensor Nodes (WSNs) diese Aufgabe. Neben Licht und Schall stehen für beinahe jede messbare Größe weitere Sensoren zur Verfügung, wie z.B. Temperatur, Luftdruck, elektrische Spannung, Kompass oder Beschleunigung. Da Sensoren nahe am Ort des Geschehens angebracht werden müssen, ist es vollkommen einleuchtend, dass Sensoren idealerweise per Funk ihre gemessenen Daten weiterreichen.

Nur leider ist eine drahtlose Energieversorgung oft nicht praktikabel; notgedrungen sind die meisten Sensoren daher mit Batterien ausgestattet. Diese müssen allerdings umso häufiger getauscht werden, je mehr Energie die Funkübertragung und der Prozessor der WSN benötigen. Das schließt für viele Anwendungen die Nutzung von WLAN und einem Rechner der Größe eines Raspberry Pi's aus – zumal beide für die gelegentliche Übertragung eines Messwertes völlig überdimensioniert wären. Wesentlich geeigneter sind kleine und genügsame Mikrocontroller, die idealerweise bereits über eine eingebaute und ebenfalls energiesparende Funkschnittstelle, wie z.B. Bluetooth Low Energy, verfügen.

Klassisch kommen Mikrocontroller bei der Steuerung von Maschinen zum Einsatz. Mit dem neuen Aufgabenfeld „Anbindung an das Internet" müssen nun aber durchaus komplexe Kommunikationsprotokolle in der Firmware durchdekliniert werden – eine Aufgabe, die leicht ein Vielfaches mehr an Zeit verschlingt als das eigentlich zu implementierende Feature. Zumal die Welt im Grunde auch keine tausend neuen TCP/IP Implementierungen bräuchte.

Entsprechend buhlt eine Vielzahl von Frameworks und Betriebssystemen um die Gunst des Entwicklers. Zugeschnitten auf die Bedürfnisse von Mikrocontrollern, wie etwa Echtzeitfähigkeit und Sparsamkeit im Ressourcenverbrauch, hat man die Wahl zwischen rundum-sorglos Paketen, die Connectivity as a Service versprechen, kommerziellen Angeboten der etablierten Platzhirsche und einer erfreulich großen Anzahl an Open-Source-Lösungen.

Gerade Letztere haben gegenüber den proprietären und kommerziellen Angeboten etliche Vorteile. Zum einen weiß man genau, was man einbaut bzw. kann es bei Bedarf ohne Weiteres in Erfahrung bringen, und zum anderen vermeidet man die Abhängigkeit von einer bestimmten (Cloud-)Infrastruktur oder von dem einen Hersteller – in Sachen Nachhaltigkeit gerade für Industrieprodukte ein wichtiges Entscheidungskriterium.

In diesem Artikel stellen wir ein noch relativ junges Projekt vor, das dennoch in kurzer Zeit zu einer der führenden Open-Source-Lösungen avanciert ist. Die Rede ist von Zephyr®[1], einem kleinen und skalierbaren Echtzeitbetriebssystem für das IoT. Der Name selbst ist der antiken Mythologie entnommen und eine Umschreibung für den Westwind. Was also liegt näher als mit Zephyr den Grundstein für eine eigene kleine Wetterstation zu legen.

[1] https://www.zephyrproject.org/

Eine Wireless Sensor Node mit Zephyr

Wahl der Hard- und Software

Zephyr wird seit 2016 unter dem Dach der Linux Foundation (weiter)entwickelt und steht unter der Apache 2.0 Lizenz. Bereits heute wird eine große Anzahl von Mikrocontroller-Familien unterstützt; mit jedem neuen Release kommen weitere hinzu. Das Grundsystem ist äußerst modular aufgebaut und kann umfassend konfiguriert werden, doch dazu später mehr. Für dieses Projekt ist es von besonderem Interesse, dass Zephyr bereits alle Stacks an Bord hat, die man für das IoT braucht, also z.B. Bluetooth, IPv6 oder MQTT.

Einer der vielen unterstützten Mikrocontroller ist der NRF52 von Nordic Semiconductor. Auf diesem Chip ist neben einem ARM Cortex-M4 Prozessor und der üblichen Peripherie auch gleich noch ein Bluetooth-Radio integriert. Ein zusätzliches Bluetooth-Modul wird also nicht benötigt. Neben dem gut ausgestatteten, allerdings auch vergleichsweise teuren Developer Kit von Nordic selbst gibt es mittlerweile einige günstige Alternativen, die sich vor allem an Maker richten.

Dazu gehören das Adafruit NRF52 Feather Pro oder das Sparkfun NRF52 Breakout Board. Besonders praktisch an beiden Boards: Sie kommen ab Werk bereits mit einen seriellen Bootloader. Ein zusätzlicher Programmier-Adapter wird also nicht benötigt[2]. Ansonsten sind die Boards aber eher spartanisch ausgestattet, lassen sich bei Bedarf jedoch leicht um zusätzliche Sensoren erweitern. Für eine vollwertige Wetterstation, wie wir sie planen, sind Sensoren für Luftdruck und Luftfeuchtigkeit schon für wenig Geld erhältlich. Wer sich die Bastelarbeit sparen möchte, kann auch auf das RuuviTag[3] zurückgreifen, das bereits in der Make vorgestellt wurde[4]. Dort wären die benötigten Sensoren bereits vorhanden.

Eine einfache Zephyr App

Bevor wir genauer auf die Anbindung unserer WSN via Bluetooth und MQTT oder dem Auslesen von Sensordaten eingehen, möchten wir zunächst den Blick auf Zephyr selbst und seine Tools lenken. Am leichtesten geht die Entwicklung unter Linux und mithilfe der Kommandozeile von der Hand. Wer möchte, kann aber auch unter Windows mit Werkzeugen wie MSYS oder WSL entwickeln. Hat man erst einmal Blut geleckt, empfiehlt sich die Konfiguration einer Entwicklungsumgebung wie Eclipse oder Visual Studio Code. Allein für eine sinnvolle Code Completion zahlt sich der einmalige Konfigurationsaufwand binnen kürzester Zeit aus. Mehr Details zur Einrichtung einer minimalen Arbeitsumgebung haben wir im Kasten "Arbeitsumgebung einrichten" zusammengetragen.

Eine typische Zephyr-Applikation hat eine vorgegebene Dateistruktur. Der Applikationsordner enthält dazu immer eine Datei `CMakeList.txt`. Dabei handelt es sich um eine Steuerdatei für das plattformübergreifende Build-Werkzeug CMake, das bei Zephyr zum Einsatz kommt. Der Aufbau der Datei ist denkbar einfach, denn die aller-

---

[2] Selbstverständlich führen beide Boards die zum Debuggen notwendige JTAG-Schnittstelle nach außen, so dass man auch einen Programmieradapter wie Segger J-Link oder Blackmagic Probe benutzen kann.

[3] https://ruuvi.com/

[4] Make: IoT Special 2016, Seite 82

meisten der notwendigen Definitionen bringt das Zephyr-Projekt bereits mit; wir müssen sie lediglich inkludieren. Dann fehlt nur noch die Angabe der projektspezifischen C-Module, die einer Konvention folgend üblicherweise in einem Unterordner `'src'` abgelegt werden.

Wie weiter oben schon erwähnt, unterstützt Zephyr eine Vielzahl verschiedener Breakout Boards und Entwicklungskits. Eine vollständige Liste kann der Dokumentation entnommen werden; alternativ genügt auch ein Blick in den `'boards'`-Ordner der Zephyr-Quellen. Für den Fall, dass das eigene Board noch nicht unterstützt wird, es aber bereits Unterstützung für den verwendeten Mikrocontroller gibt, ist das Hinzufügen einer weiteren Boardkonfiguration mit wenig Aufwand möglich. Seit kurzem kann dies innerhalb der Applikation im Ordner `'boards'` geschehen. Alle dort vorgefundenen Konfigurationen bindet das Buildsystem transparent in den `'boards'`-Ordner der Zephyr-Quellen ein. Für unser Beispiel haben wir das Sparkfun NRF52 Breakout Board[5] und das Adafruit NRF52 Feather[6] hinzugefügt. Als Vorlage nutzten wir hierfür die Konfiguration von Nordic's eigenem Development Kit `'nrf52_pca10040'`, das den gleichen Mikrocontroller verwendet. Die Auswahl, für welches Board gebaut werden soll, kann an verschiedenen Stellen hinterlegt werden, am einfachsten ist die Angabe als Kommandozeilen-Parameter beim Aufruf von CMake. Für das Adafruit Feather sieht das z.B. so aus:

```
git clone ssh://git@scm.methodpark.de:29419/fnbr/zephyr-weather.git
cd zephyr-weather
mkdir -p build/nrf52_feather && cd build/nrf52_feather
cmake -GNinja -DBOARD=nrf52_feather ../..
```

Der eigentliche Übersetzungsprozess wird danach durch Eingabe von `'ninja'` angestoßen. Möchte man seine Firmware auf mehreren Boards ausprobieren, lohnt es sich, für jedes Board ein eigenes Build-Verzeichnis anzulegen. Neben dem bereits erwähnten Aufruf ohne Parameter kennt das Zephyr-Buildsystem etliche weitere Targets, wie zum Beispiel `'run'`, um das Programm im Qemu-basierten Simulator auszuführen. Für Fortgeschrittene startet das Target `'debug'` – sofern konfiguriert und angeschlossen – einen Debugger, während `'ram_report'` und `'rom_report'` detaillierte Übersichten zum Speicherverbrauch der Applikation erstellen. Für Embedded Systeme mit ihren zum Teil sehr eng bemessenen Speichergrößen eine höchst willkommene Hilfestellung. Für weiterführende Experimente sei an dieser Stelle bereits der Hinweis gegeben, dass der serielle Bootloader des Adafruit Feather nur mit Image-Größen bis zu 160KB zuverlässig arbeitet. Für größere Projekte führt also leider kein Weg an einem JTAG-Programmieradapter vorbei.

Die zweite Einschränkung betrifft die fehlende Unterstützung durch das Zephyr-Buildsystem. Um das Adafruit zu programmieren, ist daher der folgende Befehl notwendig:

```
./flash.py ./build/nrf52_feather/zephyr/zephyr.hex /dev/ttyUSB0
```

[5] https://www.sparkfun.com/products/13990

[6] https://www.adafruit.com/product/3406

Neben CMake verwendet Zephyr das vom Linux Kernel bekannte KConfig. Damit kann Zephyr bis ins kleinste Detail angepasst werden. Das Konfigurationstool selbst wird mit einem `'ninja menuconfig'` von der Kommandozeile aus gestartet. Zu den verfügbaren Optionen gehören neben der Auswahl diverser Treiber und Protokolle auch viele Einstellungen, die für Logging und Debugging zuständig sind. Um während der Entwicklung Zeit und Nerven zu sparen, werden diese zunächst und nach Bedarf aktiviert. Für den Product-Build werden sie dann ganz einfach wieder ausgeschaltet. Typischerweise werden solche Ausgaben über eine serielle Schnittstelle ausgegeben. In dieser Hinsicht lässt sich das Adafruit Feather wiederum recht komfortabel benutzen, da die serielle Schnittstelle des NRF52 über USB bereits zur Verfügung steht. Um sich damit zu verbinden, genügt z.B. der folgende Befehl in einem separaten Terminalfenster:

```
minicom -D /dev/ttyUSB0 -b 115200
```

Telemetrie mittels MQTT und 6LowPAN

Abbildung 1: Schematischer Aufbau unserer "IoT-Solution"

Um aus dem Adafruit Feather nun eine waschechte Wireless Sensor Node zu machen, sind zwei Dinge notwendig. Zunächst braucht es einen Sensor, mit dem Daten gemessen werden können, und zweitens eine Verbindung zum Internet. Mit letzterer werden die gewonnenen Daten als Telegramme einem Broker zugestellt, der sie an registrierte Clients weiterleitet. Das ursprünglich von IBM ins Leben gerufene MQTT (Message Queue Telemetry Transport) ist für diese Aufgabe inzwischen das Standardprotokoll im IoT.

Als Sensor für unsere Wetterstation bietet sich der interne Temperatursensor des NRF52 an. Für die Verbindung zum Internet kommt typischerweise ein Gateway zum Einsatz. In unserem Projekt nutzen wir hierfür einen (vermutlich bereits vorhandenen) Raspberry Pi 3. Der Vorteil des neueren Modells gegenüber seinen Vorgängern ist die bereits integrierte Unterstützung für Bluetooth. Ältere Modelle können unter Zuhilfenahme eines Bluetooth-Dongles aber genauso gut benutzt werden. Den langen Weg ins Internet kürzen wir der Einfachheit halber etwas ab und nutzen den Raspi direkt als Message Broker. Er liefert auch die Webapplikation aus, siehe Abbildung 1. Letztere ist ein simples in Javascript geschriebenes Dashboard zur Visualisierung der empfangenen Daten. Weitere Details zur Vorbereitung des Raspi's und Installation der Webapplikation sind in den Kästen 1 und 2 zu finden.

Wie eingangs bereits erwähnt hat Zephyr bereits alles an Bord, was wir benötigen. Neben einem passenden Gerätetreiber für den Temperatursensor und einer Bibliothek, die das MQTT-Protokoll implementiert, zählen dazu die Unterstützung von TCP, IPv6 und 6LowPAN. Letzteres ist eine Kompressionsmethode für IPv6-Pakete. Das schont die Bandbreite und ermöglicht die Übertragung z.B. via Bluetooth Low Energy. Da kleine Mikrocontroller in der Regel nur wenig Speicher zur Verfügung haben, sind die genannten Features jedoch standardmäßig deaktiviert. Um die Funktionen für das eigene Projekt zu aktivieren, hilft das Kconfig-Tool. Dazu können wir entweder direkt die im Wurzelverzeichnis unseres Projektes liegende Datei **prj_nrf52_feather.proj** editieren oder wie schon beschrieben das grafische Frontend mittels **ninja menuconfig** vom Terminal aus aufrufen. Die zweite Methode hat den Vorteil, dass man sich einen sehr guten Überblick über die vielen Einstellungsmöglichkeiten verschaffen kann, während erstere eine konsistente (und reproduzierbare) Konfiguration über mehrere Arbeitsplätze hinweg sicherstellt. Die wichtigsten Einstellungen fasst Listing 1 zusammen:

```
 1 CONFIG_BT=y
 2 CONFIG_NETWORKING=y
 3 CONFIG_MQTT_LIB=y
 4 CONFIG_BT_DEVICE_NAME="Zephyr MQTT"
 5
 6 CONFIG_NET_TCP=y
 7 CONFIG_NET_UDP=n
 8 CONFIG_NET_L2_BT=y
 9 CONFIG_NET_L2_BT_ZEP1656=y
10 CONFIG_NET_IPV4=n
11 CONFIG_NET_IPV6=y
12
13 CONFIG_NET_APP_SETTINGS=y
14 CONFIG_NET_APP_MY_IPV6_ADDR="2001:db8::1"
15 CONFIG_NET_APP_PEER_IPV6_ADDR="2001:db8::2"
16 CONFIG_NET_APP_BT_NODE=y
17
18 CONFIG_BT_PERIPHERAL=y
19 CONFIG_BT_L2CAP_DYNAMIC_CHANNEL=y
20
```

```
21 CONFIG_SENSOR=y
22
23 CONFIG_TEMP_NRF5=y
24 CONFIG_TEMP_NRF5_NAME="TEMP_0"
25 CONFIG_TEMP_NRF5_PRI=1
```

Listing 1: Auszug aus der prj_nr52_feather.proj. Die Einstellungen mit ins Code Repository einzuchecken hat den Vorteil, dass bei mehreren, parallel arbeitenden Entwicklern die Konfiguration identisch ist.

Um die Dinge einfach zu halten, teilen wir Zephyr sowohl die eigene IPv6-Adresse als auch die der späteren Gegenstelle in den Zeilen 14 und 15 statisch mit. Mit der Materie vertraute Leser werden erkennen, dass die zugewiesenen Adressen Link-Local-Adressen sind. Für erste Experimente und unserem RasPi als fester Gegenstelle ist das völlig ausreichend und erspart die Konfiguration weiterer Dienste, wie etwa einem Router Advertisement Demon. Die Funktionen für den Verbindungsaufbau und das Verschicken der Telegramme sind im **publish.c** implementiert, siehe Listing 2.

```
1 #include <string.h>
2 #include <errno.h>
3
4 #include <zephyr.h>
5 #include <net/mqtt.h>
6 #include <net/net_context.h>
7 #include <misc/printk.h>
8
9 #include "publish.h"
10
11 #define SERVER_ADDR             CONFIG_NET_APP_PEER_IPV6_ADDR
12 #define SERVER_PORT             1883
13
14 #define APP_TX_RX_TIMEOUT       300
15 #define APP_NET_INIT_TIMEOUT    10000
```

```
16
17 #define SET_ZERO(data) memset(&data,0x00, sizeof(data))
18
19 /* Container for module internal state variables */
20 struct mqtt_client {
21         struct mqtt_publish_msg pub_msg;
22         struct mqtt_ctx ctx;
23 };
24
25 /* Instance of module internal state variables*/
26 static struct mqtt_client client;
27
28 void connect()
29 {
30         int rc;
31
32         SET_ZERO(client);
33
34         client.ctx.net_init_timeout = APP_NET_INIT_TIMEOUT;
35         client.ctx.net_timeout = APP_TX_RX_TIMEOUT;
36
37         client.ctx.peer_addr_str = SERVER_ADDR;
38         client.ctx.peer_port = SERVER_PORT;
39
40         rc = mqtt_init(&client.ctx, MQTT_APP_PUBLISHER);
41         printk("mqtt_init: %d\n", rc);
42         if (rc!=0) {
43                 return;
```

```
44          }
45
46          rc = 1;
47          while(rc != 0) {
48                  rc = mqtt_connect(&client.ctx);
49                  printk("mqtt_connect: %d\n", rc);
50                  k_sleep(500);
51          }
52
53          while (!client.ctx.connected) {
54                  struct mqtt_connect_msg connect_msg;
55                  SET_ZERO(connect_msg);
56
57                  connect_msg.client_id = MQTT_CLIENTID;
58                  connect_msg.client_id_len = strlen(MQTT_CLIENTID);
59                  connect_msg.clean_session = 1;
60
61                  rc = mqtt_tx_connect(&client.ctx, &connect_msg);
62                  printk("mqtt_tx_connect: %d\n", rc);
63                  k_sleep(500);
64          }
65 }
66
67 int publish(char* topic, char* msg)
68 {
69          int rc;
70
71          client.pub_msg.qos = MQTT_QoS0;
```

```
72          /* Packet Identifier, always use different values */
73          client.pub_msg.pkt_id = sys_rand32_get();
74
75          client.pub_msg.msg = msg;
76          client.pub_msg.msg_len = strlen(client.pub_msg.msg);
77          client.pub_msg.topic = topic;
78          client.pub_msg.topic_len = strlen(client.pub_msg.topic);
79
80          rc = mqtt_tx_publish(&client.ctx, &client.pub_msg);
81          printk("mqtt_tx_publish: %d\n", rc);
82
83          return rc;
84  }
```

Listing 2: Das Modul publish.c ist mit den Funktionen connect und publish für den Verbindungsaufbau und das Versenden der Telegramme versehen.

Wie bereits erwähnt nimmt Zephyr uns eine Menge Arbeit ab, so dass der verbleibende Code nur wenige Zeilen umfasst. Die erste vom Modul angebotene Funktion **connect(...)** ab Zeile 17 sorgt für den Verbindungsaufbau zum MQTT-Broker. Dazu werden nacheinander zwei Schleifen durchlaufen. Die erste Schleife ab Zeile 36 wird solange nicht verlassen, bis eine TCP-Verbindung zum entfernten Dienst steht. Um den Aufbau der dazu notwendigen BLE- und 6LowPAN-Verbindungen kümmert sich Zephyr dabei automatisch. Die zweite Schleife ab Zeile 42 wickelt nach dem erfolgreichen Aufbau der TCP-Verbindung den MQTT-spezifischen Handshake ab. Der Einfachheit halber wird auf eine weitergehende Fehlerbehandlung verzichtet; die Funktion wird schlicht erst dann verlassen, wenn die Verbindung zum Broker steht. Die zweite Funktion **publish(...)** kann nun benutzt werden, um ein neues Telegramm zuzustellen. Die Arbeit mit MQTT-spezifischen Details erledigt einmal mehr Zephyr hinter den Kulissen.

Das Hauptprogramm unserer Applikation ist ebenfalls in wenigen Zeilen implementiert, siehe Listing 3.

```
 1 #include <zephyr.h>
 2 #include <net/mqtt.h>
 3 #include <misc/printk.h>
 4 #include <string.h>
 5 #include <errno.h>
 6 #include <stdio.h>
 7 #include <sensor.h>
 8
 9 #include "publish.h"
10
11 #define SENSOR_POLLING_INTERVAL_MS 500
12
13 static struct device *sensor_dev;
14
15 void transmit_values()
16 {
17   while (true) {
18     struct sensor_value temp;
19     sensor_sample_fetch(sensor_dev);
20     sensor_channel_get(sensor_dev, SENSOR_CHAN_TEMP, &temp);
21
22     char topic[128];
23     snprintf(topic, 128, "sensors/temperature/%s", MQTT_CLIENTID);
24
25     char payload[128];
26     snprintf(payload, 128, "%d.%06d", temp.val1, temp.val2);
27
28     int rc = publish(topic, payload);
```

```
29     if (rc!=0)
30        return;
31
32     k_sleep(SENSOR_POLLING_INTERVAL_MS);
33    }
34 }
35
36 void main(void)
37 {
38     sensor_dev = device_get_binding("TEMP_0");
39     if (sensor_dev == NULL) {
40        printk("Failed to get TEMP_0 driver\n");
41        return;
42     }
43
44     while(true) {
45        connect();
46        transmit_values();
47     }
48 }
```

Listing 3: Das Modul main.c enthält die eigentliche Anwendung.

Zunächst wird in Zeile 38 eine Referenz auf den Gerätetreiber des Temperatursensors angefordert. Danach wird ab Zeile 44 auf den Verbindungsaufbau gewartet und anschließend im Halbsekundentakt ein neuer Messwert übermittelt. Das Zeitintervall kann in Zeile 11 verändert und passend eingestellt werden. Nach ersten Tests sind zwei Temperaturwerte pro Sekunde vermutlich des Guten zu viel. Das MQTT-Topic, unter dem die Werte an den Broker übermittelt werden, legt Zeile 23 fest. Der Javascript Code vom Dashboard muss später genau dieses Topic abonnieren. Die gelegentlichen Aufrufe der **printk-**Funktion von Zephyr erzeugen Ausgaben auf der seriellen Schnittstelle des Adafruit Feather. Mit einem Programm wie minicom können die Ausgaben auf dem Entwicklungsrechner sichtbar gemacht werden. Gerade im Fehlerfall steht man so nicht völlig im Dunkeln.

Der Lohn der ganzen Mühe sieht schließlich wie in Abbildung 2 gezeigt aus.

Nach einem Neustart verbinden sich zunächst das Adafruit Feather und der Raspberry Pi. Sobald der MQTT-Handshake erfolgt ist, treffen bereits die ersten Messwerte im Dashboard ein und werden entsprechend dargestellt. Sollte irgendetwas einmal nicht klappen oder mit einem Mal klemmen, hilft ein beherzter Reset der beteiligten Boards.

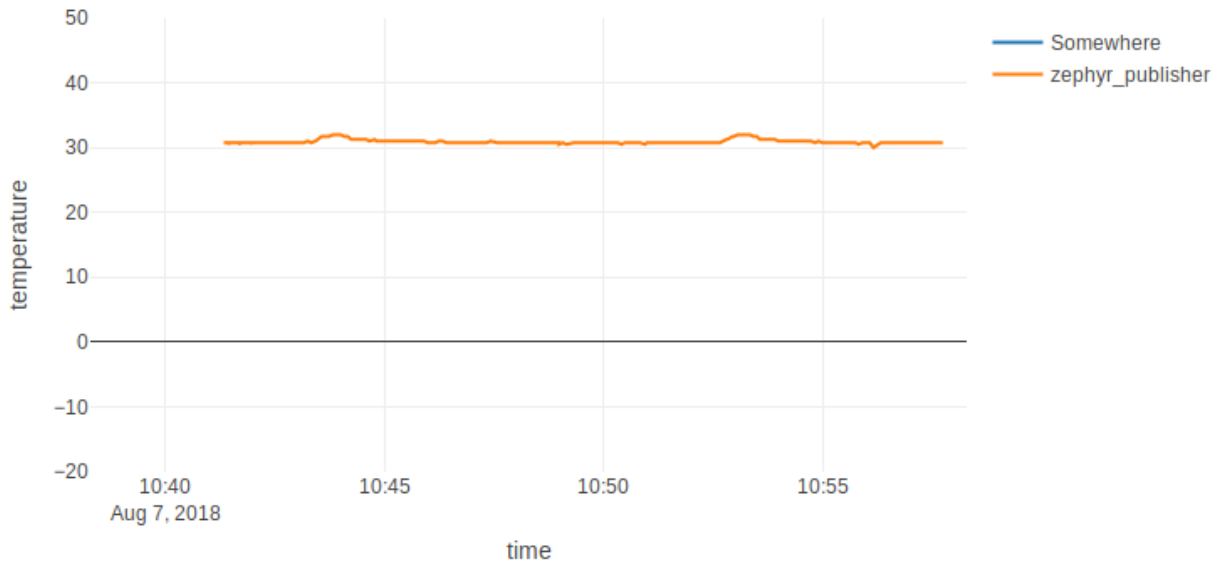

Abbildung 2: Dashboard mit den Temperaturen eines NRF in einem nicht-klimatisierten Büro

Ausblick

Das vorgestellte Beispiel hat gezeigt, dass für IoT-Funktionalitäten benötigte Kommunikationsstacks mit etwas Mehraufwand in eigenen Projekten eingesetzt werden können. Zugegebenermaßen nimmt die initiale Einrichtung mehr Zeit in Anspruch als z.B. ein einfaches Projekt mit Arduino oder dem Raspberry Pi. Hat man die Eingangshürden aber erst einmal genommen, geht die Entwicklung schnell von der Hand und man möchte schon bald die vielen bis ins allerkleinste Detail konfigurierbaren Features nicht mehr missen.

So wird zum Beispiel neben dem hier genutzten 6LowPAN over BLE seit Kurzem auch BLE Mesh unterstützt und neben MQTT gibt es inzwischen auch Unterstützung für CoAP, einem im IoT ebenfalls sehr gebräuchlichem Protokoll. Spätestens wenn die Basisfunktionen um kryptographische Funktionen zur Absicherung der Kommunikation erweitert werden müssen, ist man dankbar für die bereits erfolgte Integration von bewährten und getesteten Standardbibliotheken. Der Lohn der Mühe ist also ein in viele Richtungen erweiterbarer Ausgangspunkt für zukünftige Ideen. Denn es gibt in Zephyr noch viel mehr zu entdecken, als wir im Rahmen dieses Artikels vorstellen konnten. Das eingebaute Treibermodell erlaubt beispielsweise eine schnelle und einfache Anbindung von weiteren I2C- oder SPI-Sensoren; eine Unterstützung für Timer ist ebenfalls

vorhanden. Einen weiteren großen Pluspunkt kann das Projekt seit Release 1.11 vorweisen, als die Unterstützung für einen ebenfalls unter Open-Source Lizenz stehender Boot Loader hinzukam. Damit werden Zephyr-basierte Systeme FOTA-fähig. FOTA steht für Firmware Over The Air und beschreibt die Möglichkeit, die in der Sensor-Node laufende Software (bzw. Firmware) im Feld zu aktualisieren. Da keine Software jemals völlig fehlerfrei sein wird, ist dies eine wesentliche Voraussetzung für ein in Zukunft hoffentlich sichereres Internet der Dinge. Das Schreiben und Ausführen von Unittests wird ebenfalls unterstützt und für einen der kommenden Releases ist die Unterstützung von automatisierten Integrationstests angekündigt. Und nicht zuletzt sorgt eine aktive Community für die Pflege und Weiterentwicklung des Stacks.

Kasten 1: Arbeitsumgebung einrichten

Als Betriebssystem für die Entwicklungsumgebung bietet sich Linux an, da dort die Open-Source Toolchain nativ läuft. Aber auch eine Entwicklung unter Windows mit MSYS oder dem Windows Subsystem for Linux (WSL) funktioniert.

Wir haben für unsere Arbeit ein Ubuntu 18.04 (entweder nativ oder in einer virtuellen Maschine) benutzt. Zuerst sollte man alle nötigen Abhängigkeiten installieren. Hier beispielhaft für Ubuntu:

```
sudo apt update
sudo apt upgrade
sudo apt install --no-install-recommends git cmake ninja-build gperf \
  ccache doxygen dfu-util device-tree-compiler \
  python3-ply python3-pip python3-setuptools python3-wheel xz-utils file \
  make gcc-multilib autoconf automake libtool minicom curl \
  python-pip python-behave
```

Zephyr bietet ein Software Development Kit (SDK) mit Compilern und Debuggern für alle unterstützten Plattformen[7]. Als Host wird ein 64-bit Linux vorausgesetzt. Das SDK kommt in einem selbst-extrahierenden Archiv, das sich standardmäßig nach `/opt/zephyr-sdk` installiert. Unter Ubuntu erfolgt die Installation mit:

```
curl -L https://github.com/zephyrproject-rtos/meta-zephyr-
sdk/releases/download/0.9.3/zephyr-sdk-0.9.3-setup.run > zephyr-sdk.run
sudo sh zephyr-sdk.run
```

Damit CMake später den richtigen Compiler findet, müssen zwei Umgebungsvariablen gesetzt werden:

```
export ZEPHYR_TOOLCHAIN_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR=/opt/zephyr-sdk
```

[7] https://github.com/zephyrproject-rtos/meta-zephyr-sdk/releases/tag/0.9.2

Diese Einstellungen können in die Datei `~/.zephyrrc` geschrieben werden, um sie dauerhaft zu machen.

Als nächstes werden die Quellen des Zephyr-Projektes benötigt:

```
git clone -b v1.12-branch https://github.com/zephyrproject-rtos/zephyr.git
```

Abschließend können die letzten Abhängigkeiten installiert werden:

```
cd zephyr
pip3 install --user -r scripts/requirements.txt
```

Mehr Details sowie Anleitungen für Windows und MacOS befinden sich im "Getting Started Guide" der Zephyr Dokumentation[8].

Bevor mit der eigentlichen Entwicklung begonnen werden kann, müssen weitere Umgebungsvariablen gesetzt werden. Dies erledigt das Skript `zephyr-env.sh` im Wurzelverzeichnis des Zephyr-Projektes:

```
source zephyr-env.sh
```

Um zu prüfen, ob die Einrichtung vollständig ist, kann man das existierende Hello-World Beispiel nutzen:

```
cd samples/hello_world
mkdir -p build/qemu_cortex_m3 && cd build/qemu_cortex_m3
cmake -GNinja -DBOARD=qemu_cortex_m3 ../..
ninja
ninja run
```

Um den seriellen Bootloader vom Adafruit Feather verwenden zu können, muss das Tool `nrfutil` von Nordic in der Version 0.5.2 nebst seinen Abhängigkeiten installiert werden. Neuere Versionen funktionieren leider nicht mehr. Des Weiteren benötig der Benutzer Zugriff auf die serielle Konsole.

```
pip2 install click pyserial ecdsa 'nrfutil==0.5.2'
sudo usermod -a -G dialout $USER
```

Der letzte Befehl zeigt seine Wirkung allerdings erst, wenn man sich einmal ab- und wieder angemeldet hat.

[8] http://docs.zephyrproject.org/getting_started/getting_started.html

Kasten 2: Raspi einrichten

Um unseren Wireless Sensor Node den Zugang zum Internet zu ermöglichen, braucht es ein Gateway, das neben dem Netzwerkzugang über WLAN oder Ethernet auch Bluetooth LE mit 6LowPAN unterstützt. Ein Raspberry Pi 3 bietet sich hier hervorragend an, da ein integriertes Bluetooth-Modul bereits vorhanden ist. Ältere Pi-Modelle lassen sich mit den meisten Bluetooth-USB-Dongles ebenfalls nutzen.

SD-Karte vorbereiten

Als Betriebssystem verwenden wir das auf Debian Stretch basierende Raspbian Stretch Lite[9]. Vorsicht ist bei älteren Versionen von Raspbian geboten. Die darin enthaltenen Pakete weisen entweder noch Bugs auf oder sind unpassend konfiguriert. Alternativ zum direkten Download des Images lässt sich der Installer NOOBS[10] verwenden. Da wir aber bereits unter Linux unterwegs sind, führt der schnellste Weg zum Ziel erneut über die Kommandozeile:

```
curl -L https://downloads.raspberrypi.org/raspbian_lite_latest > raspbian.zip

unzip raspbian.zip
sudo dd if=20*-raspbian-stretch-lite.img of=/dev/sdb bs=4M conv=fsync
status=progress
```

Im obigen Beispiel wird angenommen, dass die SD-Karte als Gerätedatei /`dev/sdb` erreichbar ist. Eine doppelte Prüfung vermeidet unter Umständen Datenverlust. Mehr Details liefern die Anleitungen für das Image[11] und für NOOBS[12].

SSH einrichten

Rasbian richtet standardmäßig einen Benutzer `'pi'` mit dem Passwort `'raspberry'` ein. Aus Sicherheitsgründen ist deshalb der installierte SSH-Server deaktiviert. Ein sinnvoller erster Schritt ist es deshalb, sich lokal anzumelden und SSH zu aktiveren, damit der Pi nicht dauerhaft einen Bildschirm und eine Tastatur benötigt:

```
passwd
sudo systemctl enable ssh
sudo systemctl start ssh
```

Weitere Software installieren

Für unsere Demo werden wir neben dem Bluetooth/6LowPAN Gateway auch den MQTT-Broker und einen simplen Webserver mit einem Dashboard auf dem Pi ausführen. Hierfür muss der Pi über eine Verbindung

[9] https://www.raspberrypi.org/downloads/raspbian/

[10] https://www.raspberrypi.org/downloads/noobs/

[11] https://www.raspberrypi.org/documentation/installation/installing-images/README.md

[12] https://www.raspberrypi.org/documentation/installation/noobs.md

zum Internet verfügen, die in den allermeisten Fällen wohl über das heimische WLAN hergestellt werden wird. Die benötigten Pakete lassen sich anschließend mit folgenden Befehlen installieren:

```
sudo apt update
sudo apt upgrade
sudo apt install nginx-light mosquitto mosquitto-clients bluetooth
```

Konfiguration für MQTT, Dashboard und Bluetooth

Es sind ein paar Einstellungen und Skripte nötig, damit der Pi eine Netzwerkverbindung über 6LowPAN erlaubt und einen MQTT Broker samt Web-Dashboard bereitstellt. Eine fertige Konfiguration haben wir dem git-Repository beigefügt. Diese lässt sich einfach auf den Pi übertragen und entpacken.

```
sudo tar -xzf pi_gateway_config.tgz -C /
```

Ein Teil der Konfiguration wird erst nach einem Neustart aktiv. Wenn man nun die IP-Adresse oder den Host-Namen des Raspis im Browser aufruft, sollte man ein (zunächst leeres) Dashboard vorfinden. Testen lassen sich der Broker und das Dashboard am einfachsten mit den Mosquitto Client Tools auf dem Pi:

```
mosquitto_pub -d -t sensors/temperature/hallowelt -m 28
sleep 5s
mosquitto_pub -d -t sensors/temperature/hallowelt -m 30
sleep 5s
mosquitto_pub -d -t sensors/temperature/hallowelt -m 30
```

Mit dem enthaltenen Skript kann man nun auf BLE-Verbindungen lauschen:

```
sudo /blescan.sh
```