Die Darstellung von Radioaktivitäts-Messdaten im Internet

Bernd Laquai 16.11.2014

Als am 11. März 2011 die Katastrophe von Fukushima ihren Lauf nahm, waren die Gedanken für das was heute „Internet of Things“ (IoT) heißt, bereits gesponnen und erste Software Produkte waren schon am Laufen. So hatte Usman Haque bereits 2007 die Plattform „Pachube.com“ gegründet (von engl. „patch bay“, Steckverteiler), die das Ziel verfolgte, viele Gegenstände („Things“, im Gegensatz zu PCs oder ähnlichen Rechnern) mit dem Internet zu vernetzen und so Netzwerke von Objekten zu generieren, die einerseits räumlich verteilt Daten sammeln und im Netz speichern können und andererseits über das Internet ferngesteuert werden können. Als sich nun im Laufe des März und April 2011 in Fukushima abzeichnete, dass die japanische Behörden völlig überfordert waren oder vielleicht auch gar nicht willens waren, Radioaktivitäts-Messdaten in den betroffenen Gebieten zu erfassen und zu veröffentlichen, setzten Informatik- und Elektronikfreaks aus dem Tokyoer Hackerspace genau diesen IoT-Gedanken um und nutzten die Plattform Pachube um vor Ort Radioaktivitäts-Messdaten zu erfassen und daraus Strahlungskarten zu erzeugen.

Der jungen Plattform Pachube.com konnte aber keine bessere Publicity passieren, als dass genau ihre IoT Plattform benutzt wurde, um der Hilfe suchenden Öffentlichkeit Radioaktivitäts-Daten zu liefern, die einen Anhaltspunkt geben konnten wo man sich noch einigermaßen sicher aufhalten kann. Pachube.com richtete daher spezielle Kanäle für Geigerzähler-Geräte ein, welche die Freaks eben noch gerade auftreiben konnten und unterstützte sie auf der technischen Seite um die ersten nun politisch unabhängigen Daten zu sammeln. Das war natürlich nicht so ganz unkritisch, da die Geigerzähler sehr unterschiedlich waren und man die Messdaten unterschiedlicher unkalibrierter Geräte schlecht vergleichen konnte. Da aber die Bevölkerung diese Daten teilweise auch als Basis für die Entscheidung benutzten ob sie ihre Häuser verlassen sollten, kamen von offizieller Seite auch Vorwürfe auf, man hätte unnötig Panik verbreitet. Auf der andern Seite aber gab es von Regierungsseite für etliche Gebiete gar keine Daten, so dass diese von Freiwilligen erzeugten Strahlungskarten lange Zeit die einzigen Anhaltspunkte lieferten.

Die Freiwilligen erhielten schnell auch Hilfe von Gleichgesinnten vor allem aus den USA und man machte sich daran, einen „Einheitsgeigerzähler“ zu entwickeln und gründete die Organisation Safecast, die versuchte, die erhobenen Daten auch quantitativ vergleichbar zu machen. Was sich durch diese unabhängigen Messungen schnell zeigte, war, dass die kreisrunde 20km Evakuierungszone der Behörden nicht unbedingt sinnvoll war, da die Fahne, welche den Fallout ins Land trug länglich war und in Gebieten hohe Kontaminationen erzeugte, in denen Notunterkünfte geplant waren. Die Organisation Safecast besteht noch heute und ist längst nicht mehr nur in Japan aktiv.

Da die Aktion der Freiwilligen auch in den Medien als technologischer Erfolg und Beweis der Sinnhaftigkeit von Crowd-Based Cloud Computing gefeiert wurde, war das Interesse kommerzieller Investoren an der Idee jetzt schnell geweckt und Pachube wurde vom amerikanischen Software-Konzern LogMeIn geschluckt und zunächst als Cosm und später als Xively kommerziell vermarktet. Es gibt allerdings noch bis heute die Möglichkeit eine kostenlose Sparversion der Plattform zum Loggen von Daten und zur Datendarstellung im Internet zu nutzen. Auch im Falle der Safecast Organisation hat man mittlerweile den Eindruck, dass der ursprüngliche Gedanke der Unabhängigkeit und Gemeinnützigkeit etwas

verloren ging, zumindest aus finanzieller Sicht. Die heute relativ teuer vermarkteten Radioaktivitäts-Meßgeräte sehen äußerst professionell aus, und man kann schnell erkennen, dass selbst das Gehäuse aus der Hand eines professionellen Designers stammt. Pachube.com folgten auch schnell andere Plattformen wie sens.se (open.sens.se) und ThingSpeak.com. TingSpeak ist ebenfalls eine „open“-plattform, die so professionell daherkommen, dass man Schwierigkeiten hat zu glauben, dass diese noch unabhängig und gemeinnützig sein können. Aber auch ThingSpeak bietet immer noch das kostenlose Loggen von Messdaten und deren Darstellung im Internet an und man kann wie auch bei Xively mit Hilfe bereitgestellter API Bibliotheken und einer mehr oder weniger verständlichen Dokumentation von bekannten Mikrocontroller Plattformen aus (z.B. Arduino) Daten in Messkanäle auf diesen Plattformen füttern und sie in Form von Graphen oder Tabellen abrufen.

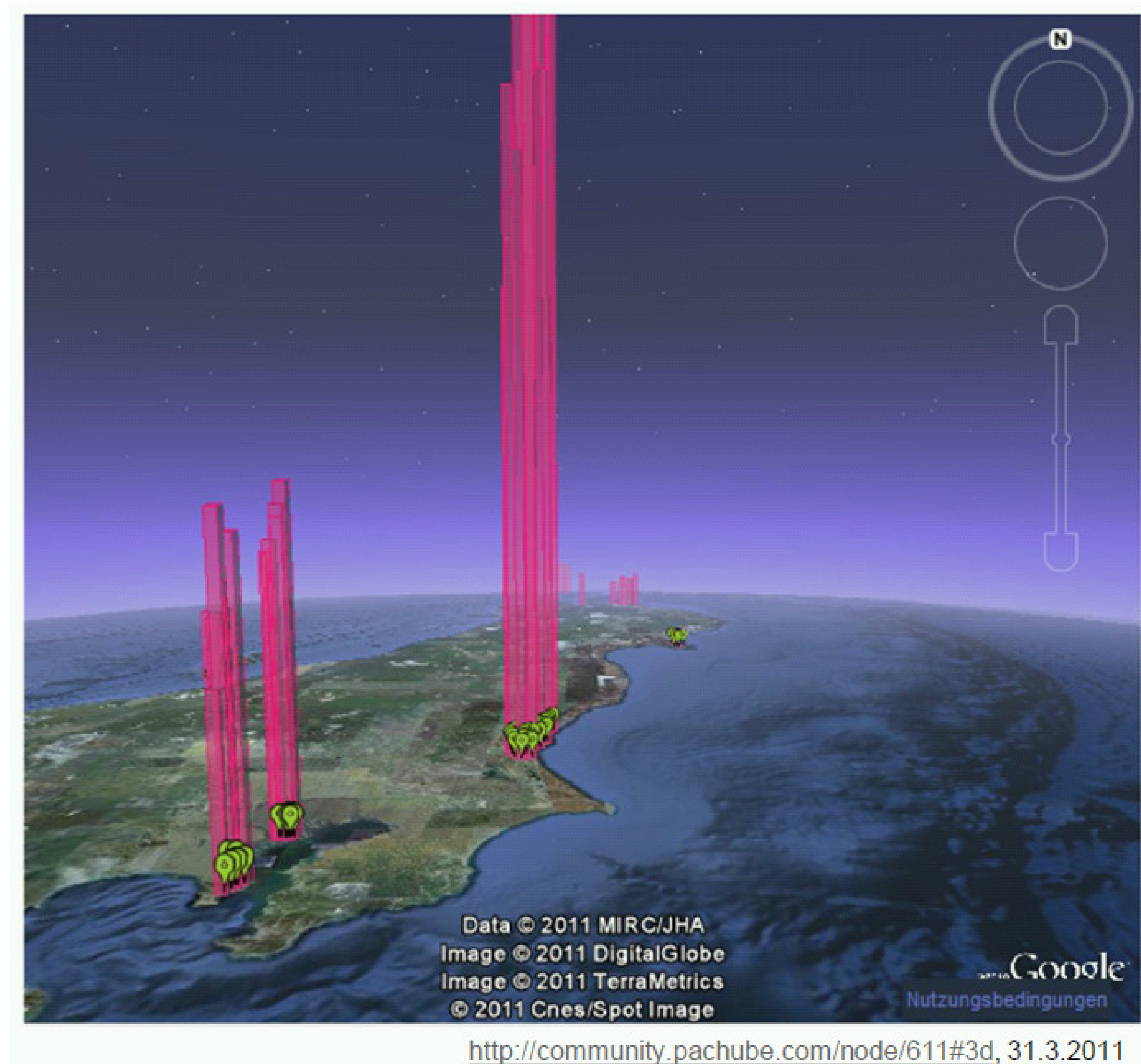

Abb. 1: Historische Strahlungskarte von Informatikfreaks mit Hilfe von Pachube ins Netz gestellt

Allerdings motivierte die Kommerzialisierung der bisherigen Plattformen auch wieder den neue „homebrew“ Lösungen. Während auf Xiveley und ThingSpeak immer mehr Temperatursensoren auftauchten, welche die Raumtemperatur von Bastlerzimmern

darstellten oder auch den Füllstand von Bürokaffeemaschinen illustrierten (also echte „Things“), machten sich nun doch wieder einige Leute dran, speziell für die Erfassung und Veröffentlichung von Messdaten der Umweltradioaktivität dedizierte Plattformen zur Verfügung zu stellen. Dazu gehören Anbieter wie uradmonitor.com und radiationnetwork.com, die sich jetzt aber über den Geräteverkauf zu finanzieren versuchten aber auch freie und offene Plattformen wie z.B. die in England gehostete Plattform Radmon.org wo entsprechend gemeinnützig denkende Privatpersonen dahinterstehen und die Sache mit viel Idealismus betreiben und selbst finanzieren.

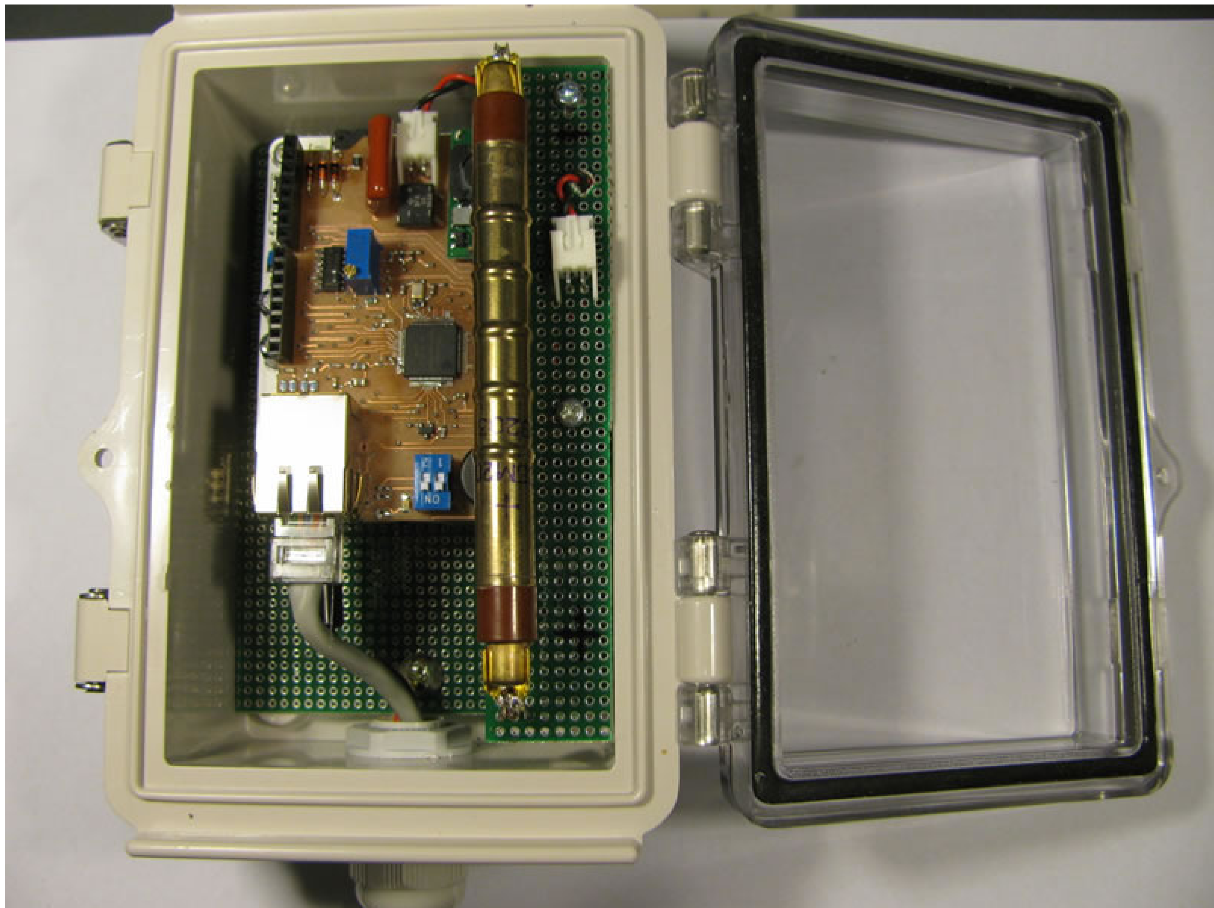

Abb. 2: Tokyo Hackerspace Geigercounter als unabhängiger Radioaktivitäts-Monitor

In der Zwischenzeit ist es aber auch für den einigermaßen informatisch gebildeten Elektronik-Bastler möglich geworden, selbst eine Server-basierte Lösung für das Loggen und Darstellen von Radioaktivitätsmessdaten im Internet zu stricken. Das wurde insbesondere dadurch möglich, dass immer mehr günstige Webhosting Pakete angeboten werden, die zusätzlich integrierte Datenbanksysteme wie SQL und Server-seitiges Skripting z.B. mit PHP anbieten, für das es freie Bibliotheken gibt, mit denen man mit überschaubarem Aufwand das erzeugen kann, wozu seinerseits Pachube in Fukushima diente. Dieser Trend wird auch dadurch ergänzt, dass es mittlerweile zur Grundausstattung einer Mikrocontroller-Plattform gehört, dass eine Internet-Konnektivität vorhanden ist, mit Hilfe derer man mindestens das HTTP–Protokoll abwickeln kann um mit einem Server kleine Datenmengen auszutauschen. Man muss also keinen stromfressenden PC aufstellen, der dauernd läuft, nur um Messdaten ins Netz zu füttern.

Aus der Sicht eines Mikrocontrollers ähneln sich aber alle 3 Konzepte, ob eine Plattform wie Xively benutzt wird, eine dedizierte Plattform wie Radmon.org verwendet wird, oder ob man sich am Ende die Mühe macht und mit Hilfe von PHP und SQL auf dem eigenen Webserver oder Webspace selbst eine Lösung bastelt. Diese Methodik ist im Folgenden am Beispiel eines einfachen Zählermoduls (Tino-Shield) beschrieben, welches digitale Zählpulse liefert welche von einem Arduino-Mikrocontroller über den Interrupt-Eingang gezählt werden. An Stelle des hier verwendeten PIN-Dioden Zählers kann natürlich genauso gut ein Zählrohr basierter Detektor stehen, der am Ausgang digitale Zählpulse liefert.

Bei Plattformen wie Xively und ThingSpeak, genau wie bei Radmon muss man sich natürlich erst einmal registrieren. Während die Registrierung bei Xively vollautomatisch und weitgehend anonym verläuft, geht es bei Radmon noch deutlich familiärer zu. Wenn man dort einen Kanal aufsetzen will, der Daten überträgt, dann muss man meistens erst einmal Kontakt mit dem Administrator (mw0uzo aka Dan) aufnehmen, der Username und Passwort für den Kanal einrichtet oder sonst irgendwelche Fragen beantwortet, wenn etwas nicht auf Anhieb funktioniert. Er reagiert auch sehr prompt und hilfsbereit und unterstützt wo er kann. Er wünscht sich bei der Gelegenheit dann aber auch, dass man seine Lösung im Forum vorstellt.

Die Übertragung von Messdaten an Radmon.org

Um nun mit Hilfe eines Arduino bei Radmon Daten zu loggen und zu visualisieren nutzt man ein Ethernet oder ein Wifi-Shield, das für den Zugriff auf das Internet dient. Außerdem enthalten beide Shields auch noch Micro-SD-Karten Slots, so dass man bei Bedarf die Daten gleich sichern kann.

Als Detektor wurde für dieses Beispiel das Tino–Shield mit Lite-On Display auf das Ethernet-Arduino Board aufgesteckt, welches über ein FTDI-Breakout von Sparkfun programmiert wird (und nicht über einen USB-Stecker auf dem Board). Diesen kleine Programmieradapter kann man nach erfolgter Programmierung vom Arduino wieder abziehen, so dass die Bauform sehr klein bleibt. Der RJ45-Stecker sitzt beim Ethernet Arduino dort wo beim UNO der USB Stecker sitzt und der Wiznet Ethernet Chip konnte auf dem Board untergebracht werden, weil der USB Kontroller fehlt. Genauso kann man natürlich auch das Tino-Shield auf ein Ethernet Shield und dieses z.B. auf einen Arduino Uno stecken.

Das Display ist für die Kontrolle der gesendeten Daten ganz hilfreich, es kann aber später für einen stromsparenderen Betrieb auch einfach abgezogen werden. Der Piezo Signalgeber klickt bei Registrierung eines Pulses ebenfalls zur Kontrolle und kann ebenfalls mit einem kleinen Schiebschalter auf dem Tino Shield abgeschaltet werden.

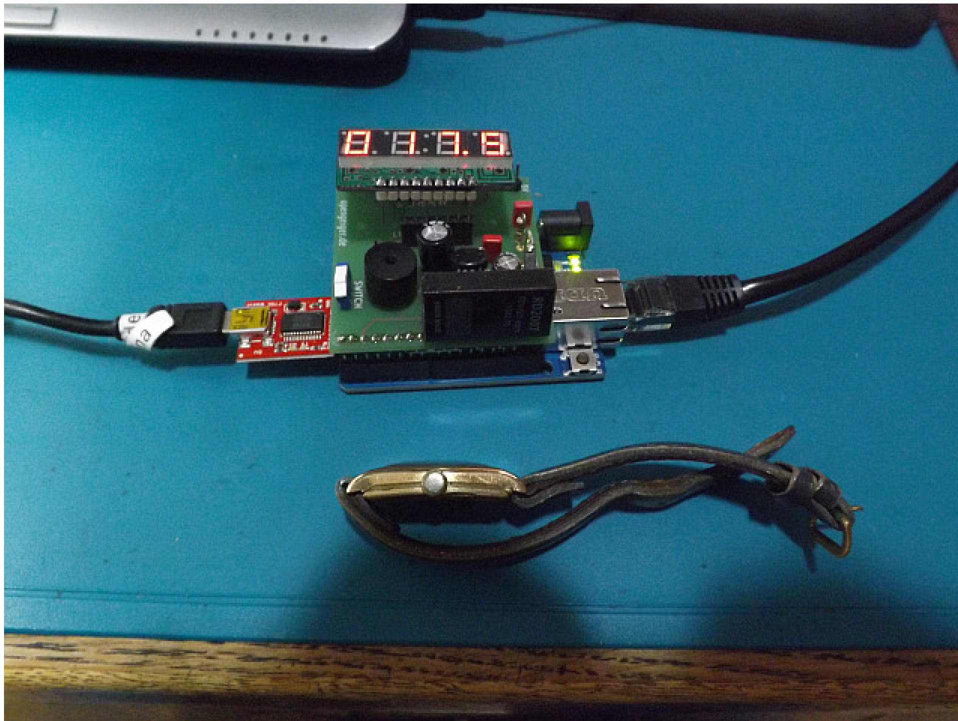

Abb. 3: Tino-Shield mit Teviso PIN-Dioden Sensormodul auf einem Ethernet-Arduino Board, welches über ein FTDI-Breakout programmiert wird

Das Arduino Programm wurde daher auch aus dem Programm für das Tino-Shield entwickelt in dem der für den Zugriff auf das Internet nötige Code an der entsprechenden Stelle eingefügt wurde. Man kann hier direkt den Beispielen auf der Arduino Webseite zur Ethernetbibliothek folgen. Man benötigt zu den Bibliotheken die für den Tino verwendet wurden (LTM8328PKR04 und SPI) jetzt zusätzlich die Ethernet und die HttpClient Bibliotheken (sowie ebenfalls die SPI Bibliothek), deren Header als erstes eingefügt werden müssen:

```
#include <Ethernet.h>
#include <HttpClient.h>
#include <SPI.h>
```

Als globale Variable wird die MAC Adresse und das EthernetClient Objekt vereinbart:

```
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xC4, 0xE7 };
EthernetClient client;
```

In der Setup Routine wird mit Ethernet.begin(mac) das Ethernet gestartet. Hier wurde der Arduino per LAN Kabel mit einem Router verbunden, der die lokale Adresse per DHCP automatisch zuweist. Falls das fehlschlägt wird in einer while Schleife 15 Sekunden gewartet, bis ein neuer Versuch gemacht wird:

```
  while (Ethernet.begin(mac) != 1)
  {
    Serial.println("Error getting IP address via DHCP, trying again...");
    delay(15000);
  }
```

In der Hauptschleife loop() wird alle 60 Sekunden (repTime) zusammen mit der normalen Ausgabe auf den Serial Monitor zusätzlich eine Verbindung zum Server hergestellt. Der entscheidenden Zeilen sind diese hier:

```
    if (client.connect("www.radmon.org", 80))
    {
        Serial.println("connected to server");
        client.print("GET
/radmon.php?function=submit&user=yyy&password=xxx");
        client.print("&value=");
        client.print(int(rate));
        client.print("&unit=CPM");
        client.println(" HTTP/1.0");
        client.println("HOST: radmon.org");
        client.println("Connection: close");
        client.println();
        Serial.println("Done.");
        delay(10000);
        Serial.println("Done.");
        Serial.println("disconnecting.");
        client.stop();
    }
    else { // you didn't get a connection to the server
      Serial.println("No connection to radmon server");
    }
```

Der übrige Code folgt dem Tino Shield Beispiel. Der Radmon Server wird per Namen auf dem Port 80 angewählt, die IP Adresse liefert also der DNS Service des Netzwerk Providers. Dann werden mit einigen print.client Statements der entscheidende HTTP Kommandotext zusammengesetzt, welche die Daten überträgt. Die übertragenen Zeilen für einen beispielhaften Messwert von 22 cpm sehen also am Ende so aus:

```
GET /radmon.php?function=submit&user=xxx&password=yyy&value=22&unit=CPM HTTP/1.0
HOST: radmon.org
Connection: close
```

Dabei ist xxx und yyy der Username und das Passwort die man mit dem Anmelden der Station bei Radmon erhält. Ob das HTTP Statement “Connection: Close” wirksam werden kann ist fraglich, da der Server eine “HTTP/1.0” Version benutzt. Das Connection: Close als Serverseitige Terminierung eine Verbindung nach einem Request gab es aber eigentlich erst ab HTTP/1.1 . Allein mit diesen Code Zeilen lief das Programm bei Tests auch nicht richtig, es wurde immer nur eine Übertragung durchgeführt. Erst als explizit am Ende noch zusätzlich ein :

```
        client.stop();
```

eingebaut wurde, welches die Verbindung Client-seitig trennt, lief die wiederholte automatische Übertragung der Messdaten problemlos.

Was noch zur Berechnung der Rate gesagt sein muss, ist die Tatsache, dass in dem Tino Programm zunächst die uSv/h gemäß des Hersteller-Konversionsfaktors vorgenommen wird (3.4cpm/(uSv/h)). Da aber in Radmon der Default ein SBM-20 Zählrohr ist und keine speziellen Absprachen mit dem Betreiber getroffen wurden, wurde der uSv/h Wert des Tino mit dem Konversionsfaktor 1/0.0057 =175.44 wieder von uSv/h in cpm einer

entsprechenden äquivalenten SBM-20 Zählrate zurück umgewandelt und mit der Unit CPM übertragen. Das gesamte Listing findet sich im Anhang.

Sobald die ersten Daten geloggt wurden, kann man sich das Ergebnis im Internet für die Messstation des users xxx unter der folgenden URL anschauen:
http://www.radmon.org/radmon.php?function=showuserpage&user=xxx

Alternativ kann man die Messstation natürlich auch auf der Landkarte der Startseite von radmon.org suchen und so zu der Seite der Messstation gelangen. Angezeigt werden die Daten über verschiedene Anzeigezeiträumen mit unterschiedlicher Mittelung. Ein Herunterladen textueller Daten ist leider nicht vorgesehen, man sollte die Daten dazu selbst parallel auf eine SD Karte loggen, wenn man Messdaten später noch weiter verarbeiten möchte.

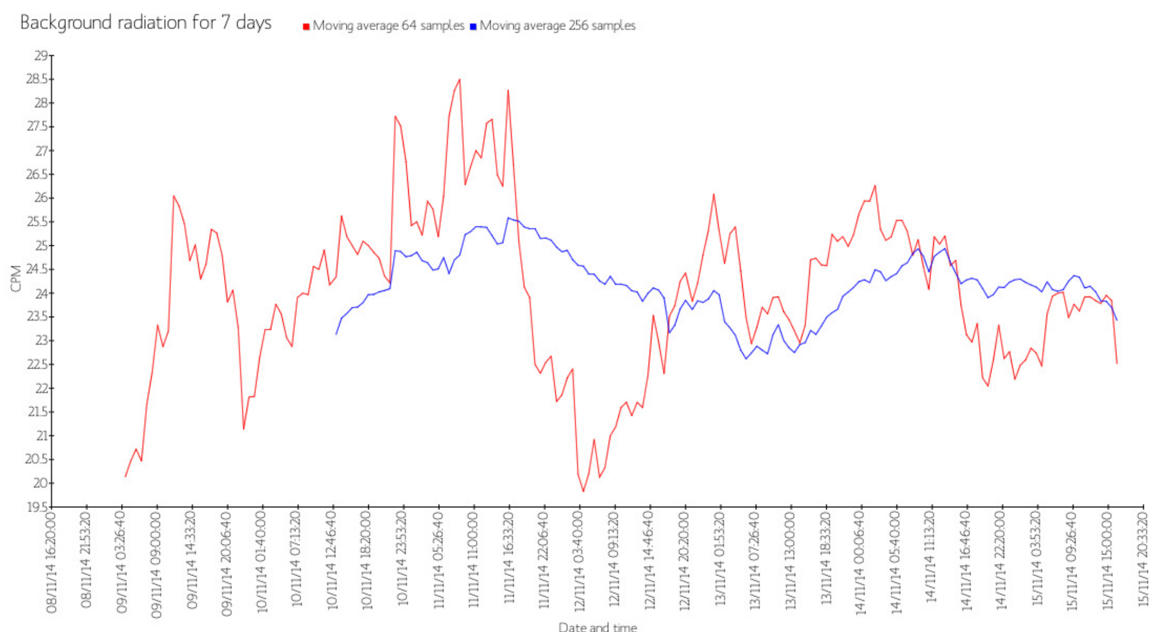

Abb. 4: Messdaten-Darstellung (Wochendaten) auf Radmon.org

Die Übertragung von Messdaten an Xively.com

Im Prinzip läuft eine Übertragung an Xively ganz ähnlich, nur dass Server-seitig nochmals zwischen Devices und Channels unterschieden wird. Nach der Registrierung legt man also stellvertretend für eine Messstation ein Device an (man kann also auch viele Messstationen betreiben). Einer Messstation können Geo-Koordinaten zugeordnet werden, so dass die sie geographisch eindeutig bestimmt ist. Da eine Messstation etliche Parameter messen kann (man denke nur an eine Wetterstation) legt man für jeden Parameter, den man messen möchte, einen Channel an. Diese Messkanäle werden der Messstation zugeordnet. Man kann aber auch jedem unterschiedlichen Typ an Strahlungsdetektor, der einer Messstation für Radioaktivität zugeordnet ist, einen separaten Kanal anlegen.

Für jeden Channel vergibt man einen Namen (zum Beispiel StuttgarterGeigerle oder Tino und erhält dann eine Feed ID und einen API key (und diverse andere Nummern und Keys die bei einfachen Anwendungen eher unbedeutend sind). Die Feed Id wird benutzt um die URL festzulegen, an die man die Daten das Kanals schickt. Der API Key fungiert als Passwort.

Im Vergleich zum Radmon Programm muss hier noch zusätzlich die Xivley Bibliothek (kann im Downlowdbereich von Xively oder auf der Arduino Webseite heruntergeladen werden) eingebunden werden:

```
#include <Xively.h>
```

zusätzlich muss man die globale Variable, welche den Xively key enthält vereinbart und initialisiert werden (am besten mit Copy&Paste aus der Xively Webseite):

```
char xivelyKey[] = "xyz";
```

Dann bekommt der Channel einen Namen, der hier über die lokale Variable sensorId[] an ein XivelyDatastrem-Objekt übergeben wird. Danach wird ein XivelyFeed-Objekt mit dem Namen feed verwendet um für den Kanal eine Referenz zu erzeugen auf den die Daten gelangen sollen. Hier wird auch erklärt, dass die Daten im Fließkommaformat angeliefert werden sollen. Anschließend wird mit setFloat-Methode des Datastream-Objekts ein Messwert übergeben, der mit der getFloat Methode auch wieder vom Objekt zurück gelesen werden kann. Die Übertragung der Daten erfolgt schließlich mit der put-Methode des xivelyclient Objekts. Die put-Methode liefert einen Wert zurück, der bei erfolgreicher Übertragung den Wert 200 hat. Liefert put einen anderen Wert zurück, deutet das auf einen Fehler hin.

```
char sensorId[] = "Tino1";
XivelyDatastream datastreams[] = {
  XivelyDatastream(sensorId, strlen(sensorId), DATASTREAM_FLOAT),
};
XivelyFeed feed(123456, datastreams, 1 /* number of datastreams */);

EthernetClient client;
XivelyClient xivelyclient(client);

sensorValue = rate;
datastreams[0].setFloat(sensorValue);
Serial.print("Read sensor value ");
Serial.println(datastreams[0].getFloat());
Serial.println("Uploading it to Xively");
int ret = xivelyclient.put(feed, xivelyKey);
Serial.print("xivelyclient.put returned ");
Serial.println(ret);
```

Loggt man sich bei Xively ein, gelangt man auf die Development-Seite und kann sich dort auch das Geschehen auf den Devices und den Kanälen anschauen. Im Reqest-Log sieht man dann wie Server-seitig die Werte ankommen. Ist man nicht eingeloggt und der Kanal wurde als Public aufgesetzt, dann kann man unter:

https://xively.com/feeds/123456

auf die Daten zugreifen, wobei 123456 die Feed-Id ist. Auf der Seite erscheint zunächst nur der letzte Wert, klickt man aber auf Graph wird zunächst für 6 Stunden ein Graph der Messdaten dargestellt. Klickt man auf den Anzeigezeitraum, kann man auch etliche andere Anzeigeintervalle einstellen. Die API Bibliothek, welche auf den Development-Seiten etwas näher erklärt ist, erlaubt aber auch das Herunterladen von archivierten Rohdaten in verschiedenen Formaten sowie von Graphen über vergangene Zeitintervalle. Je größer das

angeforderte Zeitintervall ist umso mehr wird allerdings gemittelt, so dass man auch nicht mehr beliebig auf alle Rohdaten zugreifen kann.

Abb. 5: Messdaten-Darstellung auf Xively

Die Übertragung von Messdaten auf einen eigenen Webspace

Es ist klar, dass man bei Nutzung einer der oben genannten Plattformen wie Radmon oder Xively auf deren Datendarstellung angewiesen ist. Wenn einem das nicht genügt kann man in der Zwischenzeit auch Webhosting-Pakete mit Internet-Providern vereinbaren, die es erlauben, Server-seitig Skript-Programme ablaufen zu lassen, die sowohl Daten entgegennehmen können, als auch Grafiken auf Abruf erzeugen können. Auch Datenbanksysteme wie SQL werden angeboten. Beides kann man somit dafür benutzen um Daten zu loggen und sie dann auf Bedarf nach eigenen Wünschen darzustellen. Ohne jetzt auf die Skript-Sprache PHP bzw. die Datenbanksprache SQL einzugehen, sei hier nur ein Beispiel für eine Schnittstelle mit einem Arduino mit Wifi-Shield und einem Tino-Shield aufgezeigt.

Beim Wifi-Shield muss die Wifi-Bibliothek zusammen mit der SPI und der HttpClient-Bibliothek eingebunden werden:

```
#include <SPI.h>
#include <WiFi.h>
#include <HttpClient.h>
```

Danach werden globale Variable für den Routername und das Routerpasswort vereinbart und entsprechend initialisiert. Danach folgt die Erklärung eines Wifi-Client Objekts mit dem Namen client sowie eine status Variable und eine Variable für den Server-Namen:

```
char ssid[] = "routername"; // your network SSID (name)
char pass[] = "routerpasswd"; // your network password (use for WPA, or use
as key for WEP)

int status = WL_IDLE_STATUS;
```

```
WiFiClient client;
```

Schließlich wird noch zusätzlich zu der sonst üblichen setup() und loop() Routine eine Routine erklärt, die den Status der WiFi-Verbindung ausgibt, einschließlich der Sendefeldstärke. Diese Routine wurde vollständig aus den Beispielen zur Wifi-Bibliothek Bibliothek auf der Arduino Referenz Seite übernommen.

```
void printWifiStatus() {
// print the SSID of the network you're attached to:
Serial.print("SSID: ");
Serial.println(WiFi.SSID());

// print your WiFi shield's IP address:
IPAddress ip = WiFi.localIP();
Serial.print("IP Address: ");
Serial.println(ip);

// print the received signal strength:
long rssi = WiFi.RSSI();
Serial.print("signal strength (RSSI):");
Serial.print(rssi);
Serial.println(" dBm \n");
}
```

In der Setup()-Routine wird in einer while-Schleife mit WiFi.begin versucht eine Verbindung mit dem WiFi-Router aufzunehmen, schlägt dies fehl, wird 10 Sekunden gewartet bevor ein neuer Versuch unternommen wird. Nachdem die Verbindung steht, wird mit printWifiStatus() der Status der Verbindung ausgegeben:

```
// attempt to connect to Wifi network:
while ( status != WL_CONNECTED) {
  Serial.print("Attempting to connect to SSID: ");
  Serial.println(ssid);
  status = WiFi.begin(ssid, pass);
  // wait 10 seconds for connection:
  delay(10000);
}
Serial.println("Connected to wifi");
printWifiStatus();
```

In der Hauptschleife loop() schließlich wird mit client.connect die Verbindung zum Server Port 80 hergestellt. Dann wird aus dem Messwert ein HTTP-Get Befehl zusammengesetzt und mit client.println an den Server geschickt. Dieser Get String enthält auch den Namen des Serverseitingen PHP-Skripts, das die Daten als Command-Line Parameter entgegennimmt und in die Datenbank einträgt (empfangsSkript.php). Da dies nun ein HTTP/1.1 fähiger Server ist, ist es auch ausreichend den http Connection:Close Befehl mit zugeben um die Verbindung nach dem Request wieder zu trennen:

```
if (client.connect("www.opengeiger.de", 80)) {
      Serial.println("connected to server");
      // Make a HTTP request:
      intrate = String(int(rate*100));
      getstr   =   String("GET   /empfangsSkript.php?rad="   +   intrate   +   "
HTTP/1.1");
      client.println(getstr);
      client.println("Host: www.opengeiger.de");
      client.println("Connection: close");
```

```
        client.println();
        Serial.println("Done.");
}
```

Wenn die Daten in der Datenbank stehen kann nun auf den Server ein weiteres php-Skript gelegt werden, welches bei seinem Aufruf, die Daten aus der Datenbank entnimmt und eine Grafik daraus generiert. Heute sind umfangreiche freie Grafikbibliotheken auf dem Internet erhältlich, mit denen man ziemlich aufwendige Grafiken erzeugen kann. Eine solche Grafik-Bibliothek ist jpgraph. Für die Programmierung mit PHP, SQL und jpgraph wird hier aber auf die entsprechende Literatur verwiesen, sowie auf Beispiele, wie sie zusammen mit dem jpgraph Grafikpaket geliefert werden. Ein Beispielgraph wie er sich mit PHP und jpgraph erzeugen lässt, ist hier aber gezeigt.

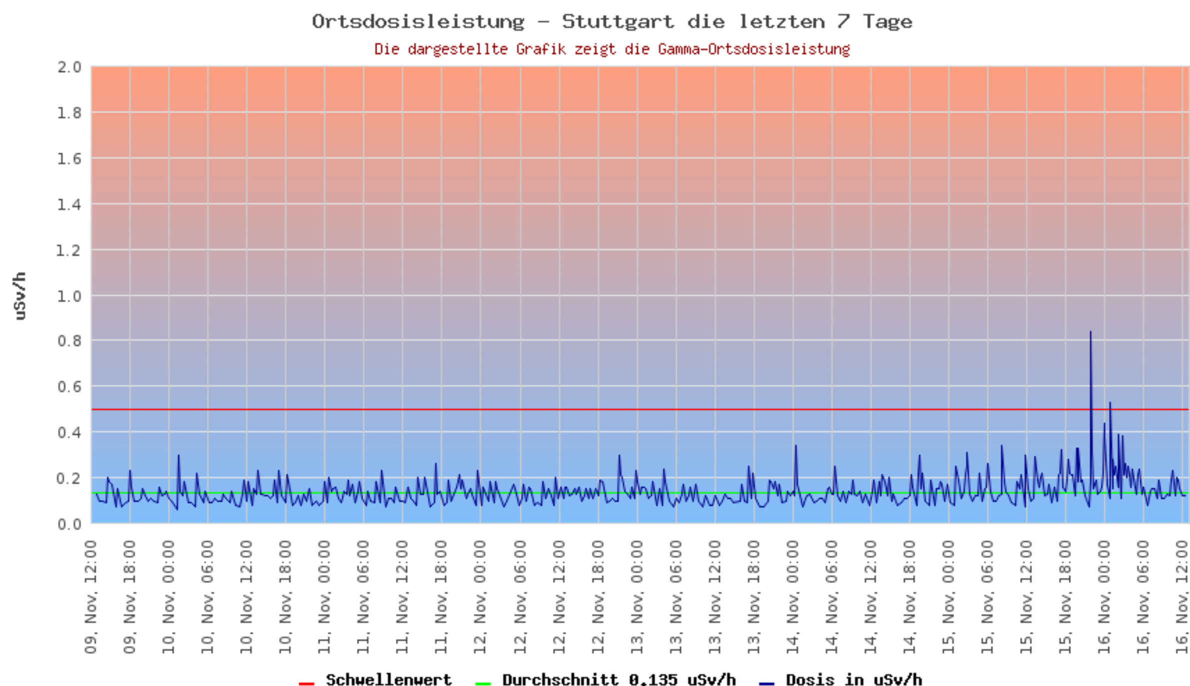

Abb. 6: Darstellung der Messdaten mit einem PHP-Skript und der jpgraph Bibliothek auf einem eigenen Webspace mit SQL-Datenbank

Das man selbst mit einem Teviso-Sensormodul noch empfindlich genug ist, um die in einem starken Regen enthaltenen natürlichen Zerfallsprodukte des Radons nachzuweisen, zeigt der Vergleich von Bild 6 und Bild 7. Genau wie in Bild 6 sieht man auch bei der amtlichen ODL Station am 16.11. gegen Null Uhr einen deutlichen Regenpeak (Niederschlag rot) der sich in einen Peak der registrierten natürlichen Umweltradioaktivität (blau) übersetzt. Dieser natürliche Fallout-Effekt läßt sich sehr gut als Funktionskontrolle einer selbstgebauten Messstation für Umweltradioaktivität nutzen.

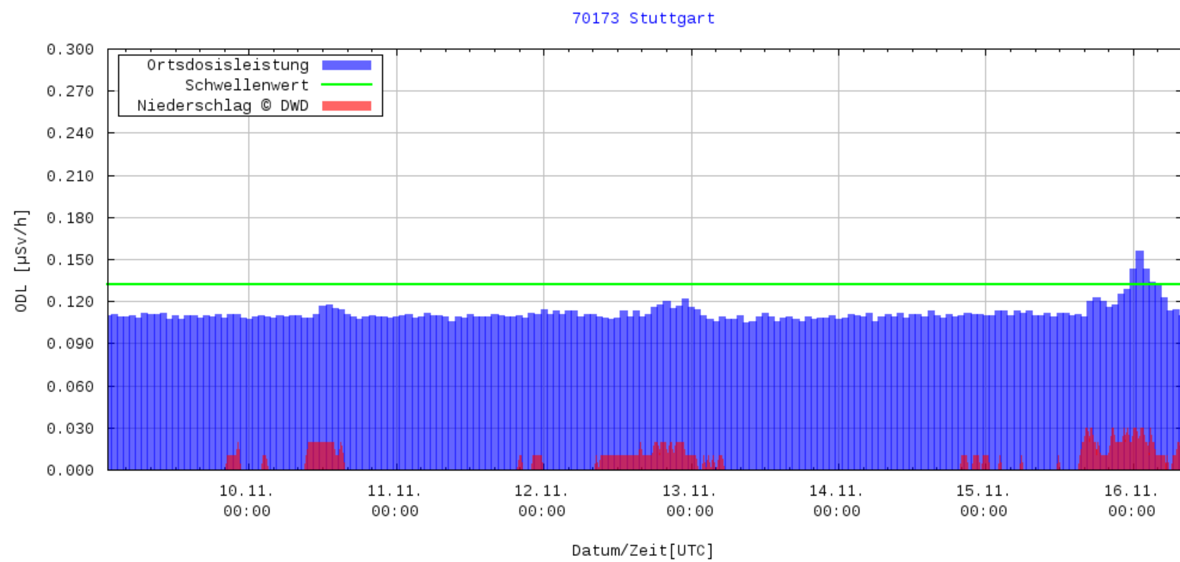

Abb. 7: Messdaten der amtlichen ODL Station in Stuttgart

Abb. 8: Tino-Shield basierte Messstation (Teviso-Sensormodul und Arduino in einer Messdose, aus einem Drainagerohrstück mit Blindstopfen hergestellt)

Literatur

/1/ Hacking a Geiger Counter in Nuclear Tokyo
http://www.freaklabs.org/index.php/Blog/Misc/Hacking-a-Geiger-Counter-in-Nuclear-Tokyo.html

/2/ Tokyo Hackerspace/RDTN Geiger Shield - Dev History
http://www.tokyohackerspace.org/en/blog/tokyo-hackerspacerdtn-geiger-shield-dev-history

/3/ COURTLAND, RACHEL: Radiation Monitoring in Japan Goes DIY. URL http://spectrum.ieee.org/tech-talk/energy/environment/radiation-monitoring-in-japan-goes-diy

/4/ Safecast
http://blog.safecast.org/

/5/ Sean Bonner, Safecast X Kickstarter Geiger Counter
https://www.kickstarter.com/projects/seanbonner/safecast-x-kickstarter-geiger-counter

/6/ ThingSpeak
https://thingspeak.com

/7/ Das Radiation Measurement Shield „Tino" für den Arduino und Arduino-kompatible Mikro-Controller
http://opengeiger.de/TinoBausatzBeschreibung.pdf

/8/ Arduino Standard Libraries
http://arduino.cc/en/Reference/Libraries

/9/ Radioaktivitätsmessnetz des Bundesamtes für Strahlenschutz
http://odlinfo.bfs.de/

Anhang: Arduino-Listings

Listing Radmon-Beispiel

```
#include <LTM8328PKR04.h>
#include <Ethernet.h>
#include <HttpClient.h>
#include <SPI.h>

#define MAXCNT 10
#define CalFactor 3.4

unsigned long oldTime = 0;
unsigned long oldRepTime = 0;
float rate;
volatile int counter = 0;

const byte dataPin = 6;
const byte clockPin = 7;
```

```
int speaker = 5;

LTM8328PKR04 sevSeg(dataPin,clockPin);

// MAC address for your Ethernet shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xC4, 0xE7 };
EthernetClient client;

void setup() {
  pinMode(speaker, OUTPUT);
  digitalWrite(speaker, LOW);
  sevSeg.setLeadingZeros(1);

  Serial.begin(9600);
  Serial.println("Starting upload to radmon...");
  Serial.println();


  while (Ethernet.begin(mac) != 1)
  {
    Serial.println("Error getting IP address via DHCP, trying again...");
    delay(15000);
  }
  delay(1000);
  Serial.print("My IP address: ");
  Serial.println(Ethernet.localIP());

  attachInterrupt(0, count, RISING);
  Serial.println("waiting for interrupt...");
}

void loop() {
  unsigned long time;
  unsigned long dt;
  unsigned long repTime;
  String intrate;
  String getstr;
  int i;

  i = (int)(rate*10);
  sevSeg.print(i,3);

  time = millis();
  if (counter >= MAXCNT) {
    dt = time-oldTime;
    oldTime = time;
    counter = 0;
    Serial.print('%');
    Serial.println(dt);
    rate = (float)MAXCNT*60.0*1000.0/(float)dt/CalFactor;
    Serial.println(rate);
  }
  repTime = time - oldRepTime;
  if (repTime > 600000) {
    detachInterrupt(0);
    Serial.println(rate);
    Serial.println("Uploading it to radmon");

    if (!client.connected()) {
      if (client.connect("www.radmon.org", 80)) {
        Serial.println("connected to server");
        intrate = String(int(rate*175.44)); //SBM-20 cpms

        getstr   =   String("GET   /radmon.php?function=submit&user=xxx&password=yyy&value="   +
intrate + "&unit=CPM HTTP/1.0");
        Serial.println(getstr);
        client.println(getstr);

        client.println("Host: www.radmon.org");
        client.println("Connection: close");
        client.println();
        delay(10000);
        Serial.println("Done.");
        Serial.println("disconnecting.");
        client.stop();
      }
      else {
        // you didn't get a connection to the server
```

```
        Serial.println("No connection to server");
      }
    }
    oldRepTime = millis();
    attachInterrupt(0, count, RISING);
  }
}

void count()
{
  counter++;
  digitalWrite(speaker, HIGH);
  delayMicroseconds(50000);
  digitalWrite(speaker, LOW);
}
```

Listing Xively-Beispiel

```
#include <LTM8328PKR04.h>
#include <SPI.h>
#include <Ethernet.h>
#include <HttpClient.h>
#include <Xively.h>

#define MAXCNT 10
#define CalFactor 3.4

unsigned long oldTime = 0;
unsigned long oldRepTime = 0;
float rate;
volatile int counter = 0;

const byte dataPin = 6;
const byte clockPin = 7;
int speaker = 5;

LTM8328PKR04 sevSeg(dataPin,clockPin);

// MAC address for your Ethernet shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xC4, 0xE7 };

// Your Xively key to let you upload data
char xivelyKey[] = "xyz";

// Define the strings for our datastream IDs
char sensorId[] = "Tino1";
XivelyDatastream datastreams[] = {
  XivelyDatastream(sensorId, strlen(sensorId), DATASTREAM_FLOAT),
};
// Finally, wrap the datastreams into a feed
XivelyFeed feed(123456, datastreams, 1 /* number of datastreams */);

EthernetClient client;
XivelyClient xivelyclient(client);

void setup() {
  // put your setup code here, to run once:
  pinMode(speaker, OUTPUT);
  digitalWrite(speaker, LOW);
  sevSeg.setLeadingZeros(1);

  Serial.begin(9600);
  Serial.println("Starting single datastream upload to Xively...");
  Serial.println();

  while (Ethernet.begin(mac) != 1)
  {
    Serial.println("Error getting IP address via DHCP, trying again...");
    delay(15000);
  }

  attachInterrupt(0, count, RISING);
  Serial.println("waiting for interrupt...");
}

void loop() {
```

```
  unsigned long time;
  unsigned long dt;
  unsigned long repTime;
  float sensorValue;
  int i;

  i = (int)(rate*10);
  sevSeg.print(i,3);

  time = millis();
  if (counter >= MAXCNT) {
    dt = time-oldTime;
    oldTime = time;
    counter = 0;
    Serial.print('%');
    Serial.println(dt);
    rate = (float)MAXCNT*60.0*1000.0/(float)dt/CalFactor;
    Serial.println(rate);
  }

  repTime = time - oldRepTime;
  if (repTime > 15000) {
    detachInterrupt(0);
    sensorValue = rate;
    datastreams[0].setFloat(sensorValue);
    Serial.print("Read sensor value ");
    Serial.println(datastreams[0].getFloat());
    Serial.println("Uploading it to Xively");
    int ret = xivelyclient.put(feed, xivelyKey);
    Serial.print("xivelyclient.put returned ");
    Serial.println(ret);
    Serial.println();
    oldRepTime = millis();
    attachInterrupt(0, count, RISING);
  }
}

void count()
{
  counter++;
  digitalWrite(speaker, HIGH);
  delayMicroseconds(50000);
  digitalWrite(speaker, LOW);
}
```

Listing PHP/SQL/jpgraph-Beispiel

```
#include <SPI.h>
#include <WiFi.h>
#include <HttpClient.h>

#define MAXCNT 10
#define CalFactor 3.4

unsigned long oldTime = 0;
float rate;
volatile int counter = 0;

char ssid[] = "routername"; // your network SSID (name)
char pass[] = "routerpasswd"; // your network password (use for WPA, or use as key for WEP)

int status = WL_IDLE_STATUS;
WiFiClient client;


void printWifiStatus() {
// print the SSID of the network you're attached to:
Serial.print("SSID: ");
Serial.println(WiFi.SSID());

// print your WiFi shield's IP address:
IPAddress ip = WiFi.localIP();
Serial.print("IP Address: ");
Serial.println(ip);

// print the received signal strength:
```

```
long rssi = WiFi.RSSI();
Serial.print("signal strength (RSSI):");
Serial.print(rssi);
Serial.println(" dBm \n");
}

void setup() {
  Serial.begin(9600);

  // attempt to connect to Wifi network:
  while ( status != WL_CONNECTED) {
  Serial.print("Attempting to connect to SSID: ");
  Serial.println(ssid);
  status = WiFi.begin(ssid, pass);
  // wait 10 seconds for connection:
  delay(10000);
  }
  Serial.println("Connected to wifi");
  printWifiStatus();

  Serial.println("waiting for interrupt...");
  attachInterrupt(0, count, RISING);
}

void loop() {
  unsigned long time;
  unsigned long dt;
  int ret=0;
  String intrate;
  String getstr;

  time = millis();
  if (counter >= MAXCNT) {
    detachInterrupt(0);
    dt = time-oldTime;
    oldTime = time;
    counter = 0;
    //Serial.print('%');
    //Serial.println(dt);
    rate = (float)MAXCNT*60.0*1000.0/(float)dt/CalFactor;
    Serial.println(rate);
    Serial.println("\nStarting connection to server...");
    //if you get a connection, report back via serial:
    if (client.connect("www.opengeiger.de", 80)) {
      Serial.println("connected to server");
      // Make a HTTP request:
      intrate = String(int(rate*100));
      getstr   =   String("GET   /empfangsSkript.php?rad="   +   intrate   +   "   HTTP/1.1");
client.println(getstr);
      client.println("Host: www.opengeiger.de");
      client.println("Connection: close");
      client.println();
      Serial.println("Done.");
    }
    attachInterrupt(0, count, RISING);
  }
}

void count()
{
  counter++;
}
```