Frankfurt University of Applied Sciences
Fachbereich 2, Informatik
Dipl. Inf. Markus Krauße

Kontakt:wsnprojekt@fb2.fra-uas.de

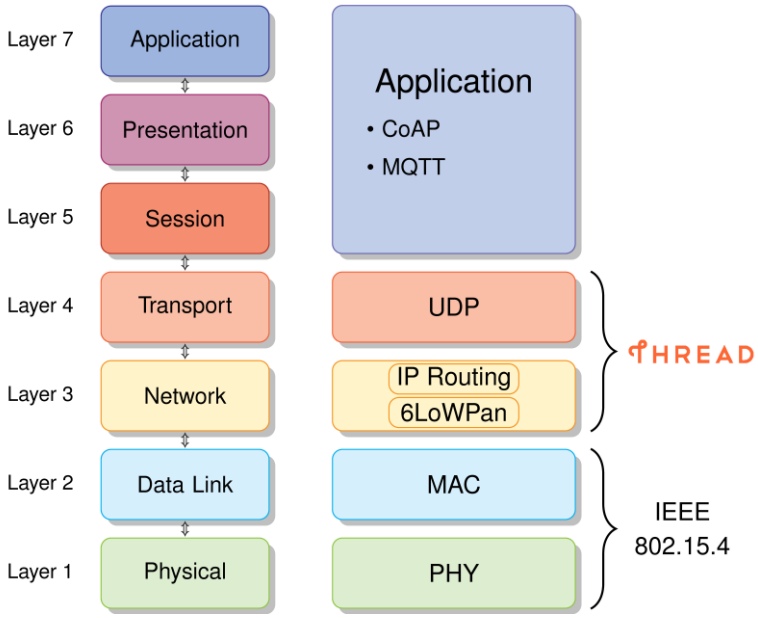
Blatt 3

Drahtlose Kommunikation mit Thread

Sensorwerte werden im Allgemeinen an vielen unterschiedlichen, manchmal auch schwer zugänglichen Orten erfasst und an einem zentralen Punkt gesammelt, um sie für weitere Auswertungen zu nutzen. Für diese Anforderungen eignen sich besonders drahtlose Netzwerktechniken, die einen möglichst geringen Energieverbrauch aufweisen. Für die Umsetzung solcher Netzwerke stehen mittlerweile eine ganze Reihe unterschiedlicher Protokolle und Techniken zur Verfügung wie beispielsweise *ZigBee*, *LoRaWan*, *WiFi*, *Bluetooth Low Energy* (*BLE*) und *Thread*.

1 Thread

Thread ist ein auf IPv6 basierendes Meshnetzwerk-Protokoll. Es eignet sich hervorragend für Home Automation, Industrie 4.0 und drahtlose Sensor-Aktor-Netzwerke. Die Thread Group wurde 2014 gegründet und zu ihren Gründungsmitgliedern zählen Unternehmen wie Google Nest, Osram und Samsung. Im Jahr 2018 ist auch Apple der Gruppe beigetreten. Thread basiert auf dem zuverlässigen IEEE 802.15.4 Standard, der eine sichere drahtlose Kommunikation von direkt in Reichweite befindlichen Funkmodulen ermöglicht. Thread erweitert diesen Standard um die Fähigkeit der Bildung IPv6 basierter Meshnetzwerke. Dadurch ist es möglich, größere Netzwerke aufzubauen und Funkmodule zu erreichen, die sich nicht in unmittelbarer Reichweite befinden.



1.1 Rollen und Gerätetypen

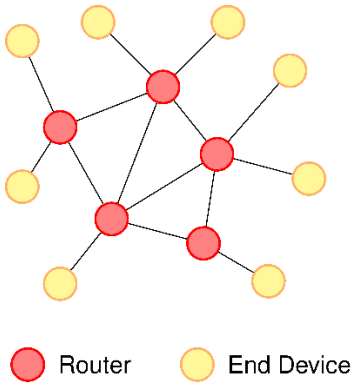
In ein Thread-Meshnetzwerk kann ein Funkmodul entweder die Rolle eines *Routers* oder eines *Endgerätes* einnehmen. Als Router nimmt ein Funkmodul aktiv am Routingprozess teil, als Endgerät kommuniziert das Funkmodul ausschließlich mit einem ihm zugeordneten Router (Elternteil).

Router

- Weiterleitung von Paketen.
- Ermöglicht den Netzwerkbeitritt.
- Transceiver ist immer an.

Endgerät

- Kommuniziert nur mit einem Router.
- Leitet keine Pakete weiter.
- Transceiver kann zeitweise abgeschaltet werden.



Diese Unterteilung betrifft allerdings nur die Rolle in Bezug auf die Möglichkeit der Weiterleitung von Paketen. Ein Gerät, welches die grundlegende Funktionalität für das Routing implementiert, nennt sich *Full Thread Device (FTD)*. Ein Gerät ohne diese Funktionalität ist ein *Minimal Thread Device (MTD)*. Insgesamt gibt es fünf Gerätetypen, drei FTDs und zwei MTDs.

FTD	MTD
Router	Minimal End Device (MED)
Router Eligible End Device (REED)	Sleepy End Device (SED)
Full End Device (FED)	

Ein *Router Eligible End Device (REED)* kann zu einem Router ernannt werden, wenn z.B. zu wenig Router im Netz vorhanden sind oder ein anderer Router ausfällt. Wird die Router-Funktionalität nicht mehr benötigt, wird das Gerät wieder zu einem *Full End Device (FED)* heruntergestuft. Dies macht das Netz sehr robust gegenüber Ausfällen, vermeidet aber einen Overhead an Routinginformationen wie zu große Routingtabellen.

Ein FED kann nicht zu einem Router promotet werden, pflegt im Gegensatz zu MTDs aber eine Tabelle mit den unmittelbaren Nachbarroutern und deren Verbindungsqualität.

MTDs sind immer einem bestimmten Elternteil (aktiver Router) zugeordnet. Sie kommunizieren nur direkt mit diesem. Bricht die Verbindung ab, suchen sie sich ein neues Elternteil.

Sleepy End Devices (SED) können ihren Transceiver zum Energiesparen abschalten. Sie müssen beim Aufwachen bei ihrem Elternteil explizit Nachrichten anfragen, die dort eventuell für sie hinterlegt sind.

Eine weitere wichtige Rolle in einem Thread-Netzwerk ist der *Thread Leader*. Es gibt genau einen ausgezeichneten Thread Leader. Dieser wird dynamisch vom Thread-Netzwerk selbst erwählt. Ein Thread Leader ist für die Verwaltung der Router zuständig, insbesondere vergibt er den Routern ihre ID-Nummer. Fällt ein Thread Leader aus, wird ein anderer Router für diese Rolle erwählt. Dies vermeidet in Thread-Netzwerken einen *Single Point of Failure*.

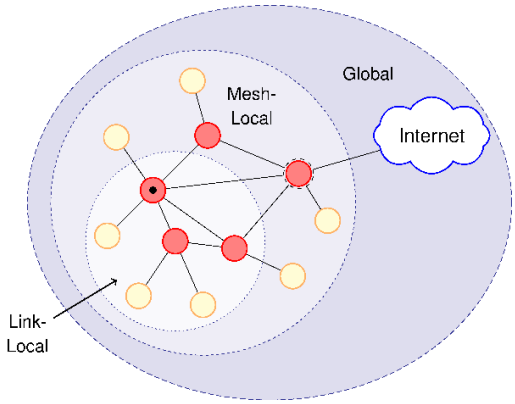
1.2 Adressierung

Die Adressierung in Threadnetzwerken erfolgt mittels 128-Bit IPv6-Adressen. Ein Funkmodul hat immer mehrere Adressen. Die IPv6-Adressen werden in verschiedene Typen bzgl. ihrer Reichweite eingeteilt.

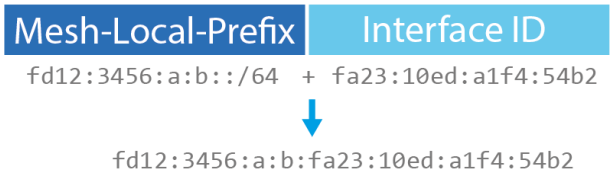
Link-Local: Direkte Funkreichweite (fe80::/16)

Mesh-Local: Mesh-Netzwerk (fd00::/8)

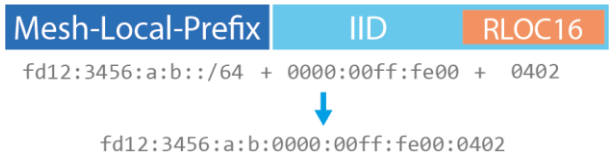
Global: Globale Adresse (2000::/3)



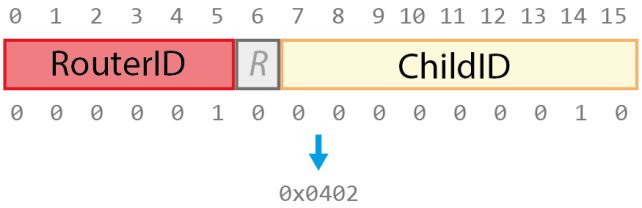
Im einem Mesh-Netzwerk erhält ein Funkmodul mindestens zwei IPv6-Adressen. Eine IPv6-Adresse dient zur eindeutigen Identifizierung und ändert sich im Netzwerk auch nicht mehr. Diese Adresse heißt *Mesh Local Endpoint Identifier (ML-EID)*. Sie besteht aus der 64-Bit Mesh-ID und aus einem 64-Bit vom Stack zufällig erzeugtem eindeutigen Identifier:



Die zweite IPv6-Adresse dient für das Routing. Sie besteht aus der 64-Bit Mesh-ID und einem 64-Bit *Routing Locator*:



Der Routing Locator besteht aus dem Prefix `0000:00ff:fe00` und dem 16-Bit *RLOC16*. Die ersten 6-Bits des RLOC16 bestimmen den Router, welchem ein Endgerät zugewiesen ist. Das 7.te Bit ist reserviert. Die hinteren 9 Bits bilden die ID des Kindes. Handelt es sich bei dem Funkmodul selbst um einen Router ist diese ID 0.



Wird ein Kind einem neuen Router zugewiesen, ändert sich der Routing Locator. Über den Routing Locator kann ein Paket relativ leicht an ein Funkmodul weitergeleitet werden. Wir adressieren Funkmodule allerdings immer über die ML-EID. Funkmodule speichern sich in einer Tabelle die zu den ML-EIDs zugehörigen RLOC16 Identifier. Ist der zugehörige RLOC16 noch nicht bekannt, wird dieser durch eine Broadcast-Anfrage ermittelt.

2 Erstes Thread-Beispiel mit CLI

Zum besseren Verständnis wollen wir uns ein erstes Beispiel ansehen. Nordic hat in das *nRF connect SDK OpenThread*, eine freie Thread-Implementation von *Google Nest*, integriert. Um OpenThread für das Developerboard in Zephyr zu aktivieren, müssen in der Projektkonfigurationsdatei *prj.conf* einige Schalter gesetzt werden. Zunächst einmal gilt es die OpenThread- und Netzwerk-Bibliotheken einzubinden:

```
# Enable OpenThread FTD features set
CONFIG_OPENTHREAD_NORDIC_LIBRARY_FTD=y

# L2 OpenThread enabling
CONFIG_NET_L2_OPENTHREAD=y

# Generic networking options
CONFIG_NETWORKING=y
```

Praktischerweise können in der Projektkonfigurationsdatei auch bereits die Netzwerkparameter für das Thread-Netzwerk festgelegt werden, es gibt aber auch API-Funktionen um die Parameter im Programmcode zu setzen:

```
# Network parameter
CONFIG_OPENTHREAD_MANUAL_START=n
CONFIG_OPENTHREAD_NETWORK_NAME="WSN00"
CONFIG_OPENTHREAD_PANID=10000
CONFIG_OPENTHREAD_XPANID="fb:02:00:00:ab:cd:00:00"
CONFIG_OPENTHREAD_NETWORKKEY="00:11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff"
```

Passen Sie den Netzwerknamen, die *PanID* und die *erweiterte PanID* entsprechend so an, dass Sie hierfür die letzten zwei Nummern bzw. Zeichen ihre Set-Nummer nutzen, um Interferenzen mit den Netzwerken der anderen Studierenden zu vermeiden.

Weiterhin aktivieren wir die OpenThread-Shell, mit der wir Kommandos über ein Terminalfenster an die Developerboards senden können:

```
# Network shell
CONFIG_SHELL=y
CONFIG_OPENTHREAD_SHELL=y
CONFIG_SHELL_ARGC_MAX=26
CONFIG_SHELL_CMD_BUFF_SIZE=416
```

Desweitern deaktivieren wir aus performancetechnischen Gründen den SHA1-Hash und aktivieren die Floating Point Unit des Mikrocontrollers:

```
#Additional parameter
CONFIG_MBEDTLS_SHA1_C=n
CONFIG_FPU=y
CONFIG_GPIO=y
```

Wir generieren zudem wieder ein Overlay-File nrf52840dk_nrf52840.overlay für den Devicetree wie in Aufgabe 5 aus Aufgabenblatt 2 und ändern den Knoten `chosen/zephyr,entropy` von `&cryptocell` zu `&rng`:

```
/ {
    chosen {
        zephyr,entropy = &rng;
    };
};
```

Dies sind Einstellungen für das randomisierte Erzeugen von Zahlen. Nordic Semiconductor macht dies für alle *OpenThread*-Anwendungsbeispiele. Somit übernehmen wir diesen Parameter vorsichtshalber, um keine Probleme während dem Betrieb des Netzwerkes zu bekommen.

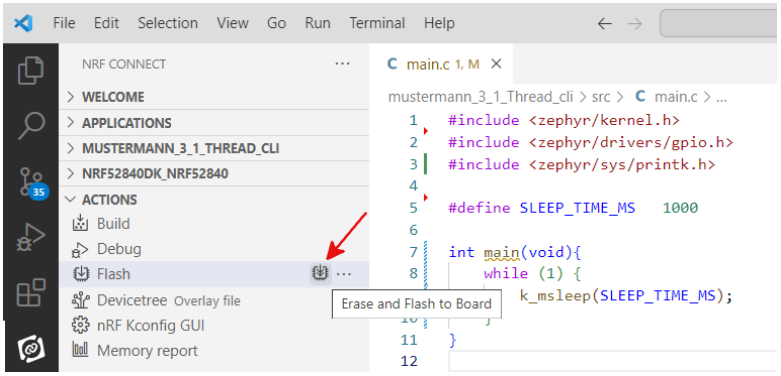
Für unser erstes Thread-Beispiel benötigen wir keine Anwendungslogik. Das Thread-Netzwerk wird von Zephyr automatisch gestartet und wir erhalten über die Shell und ein Command Line Interface Zugriff auf die Funkmodule. Somit ist unser Programmcode erstmal sehr minimalistisch:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/sys/printk.h>
#define SLEEP_TIME_MS   1000
int main(void){
    while (1) {
        k_msleep(SLEEP_TIME_MS);
    }
}
```

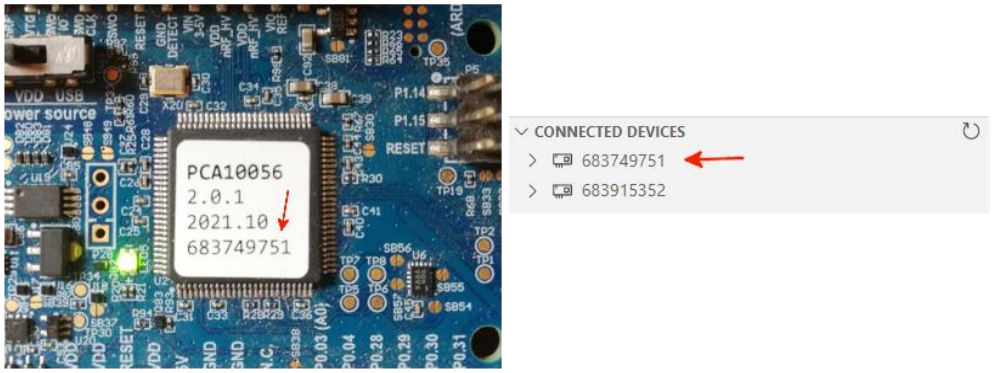
Aufgabe 1: Erstellen Sie eine erste Thread-Anwendung mit CLI.

Programmieren Sie das zuvor vorgestellte Thread-Beispiel mit den an Ihre Set-Nummer angepassten Netzwerkparametern und übertragen es auf zwei Funkmodule. Lassen Sie sich über das CLI die Netzwerkparameter anzeigen und geben Sie die IP-Adressen der einzelnen Module aus. Danach senden Sie ein UDP-Paket von einem Funkmodul zum anderen. Machen Sie von den Ausgaben Screenshots und legen diese der Abgabe bei.

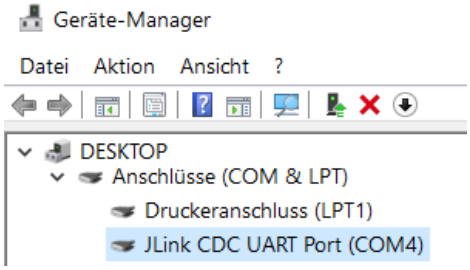
Bei unserem Minimalbeispiel wollen wir die Netzwerkparameter aus der Projektkonfigurationsdatei nutzen. Wurde bereits ein Thread-Netzwerk auf den Funkmodulen gestartet, sind diese Parameter in einem nicht flüchtigen Speicher festgehalten und werden beim Start des Thread-Netzwerkes genutzt. Achten Sie deswegen beim ersten Übertragen der Firmware mit neuen Netzwerkparametern darauf, dass Sie diese Daten löschen, d.h. klicken Sie nicht auf *Flash* sondern auf *Erase and Flash*:



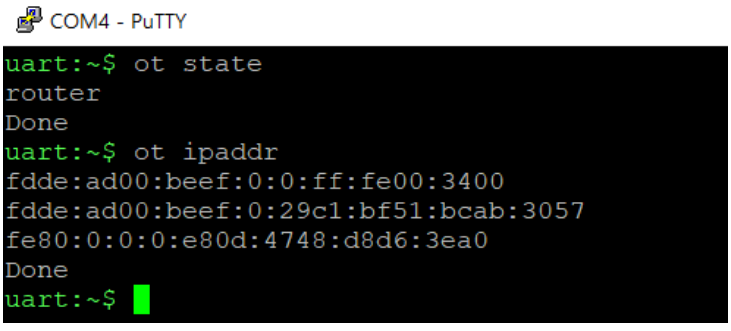
Wenn Sie mehrere Developerboards mit ihrem Computer verbunden haben, achten Sie auf die korrekte Auswahl des J-Links. Die Identifikationsnummer finden Sie auf dem Chip und wird Ihnen in VS Code unter *CONNECTED DEVICES* angezeigt:



Nachdem wir die Applikation übertragen haben, wird direkt ein Thread-Netzwerk aufgebaut. Die Parameter können wir uns über das Command Line Interface (CLI) anschauen und so auch als ersten Test mit einem anderen Funkmodul kommunizieren. Das CLI nutzt die UART0-Schnittstelle des nRF52840 und leitet diese Daten an den J-Link Debugger-Chip weiter, der auch als USB-UART-Bridge fungiert. In Windows finden wir den zugehörigen Port im Gerätemanager.

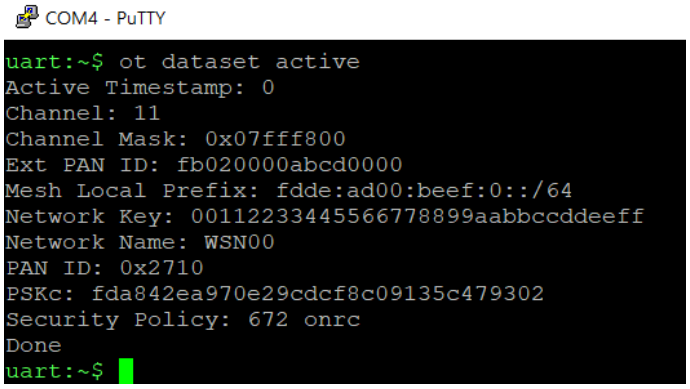


Jetzt starten wir ein Terminalprogramm (z.B. PuTTY) und verbinden uns mit den Funkmodulen. Die Baudrate beträgt 115200. Nach einem Drücken der Taste Enter sollte ein Shell-Prompt (uart:~$) erscheinen. In Zephyr können mehrere Shells implementiert sein. Um Befehle an das Command Line Interface (CLI) zu senden, wird deswegen immer *ot* vorangestellt, um die OpenThread-Shell zu adressieren. Mit dem Befehl *ot state* erhalten wir die Rolle des Funkmoduls in unserem Thread-Netzwerk und mit *ot ipaddr* erfahren wir alle unserem Funkmodul zugeordneten IPv6-Adressen:



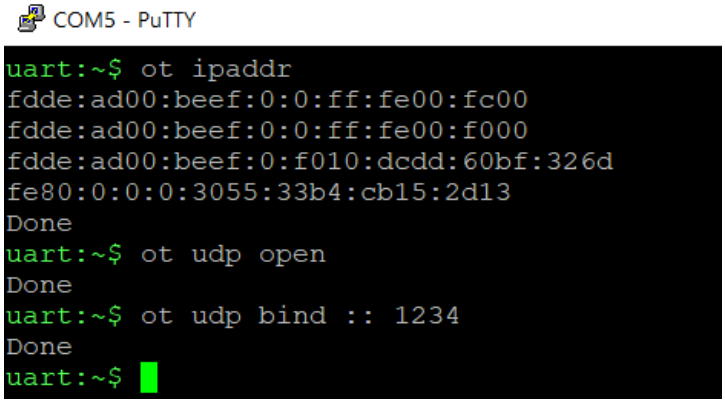
Hier sehen wir als erste IPv6-Adresse die RLOC für das Routing, als zweite Adresse die ML-EID zur eindeutigen Identifizierung und als dritte Adresse eine Link-Local-Adresse.

Mit dem Befehl *ot dataset active* erhalten wir einen Überblick der Netzwerkparameter:

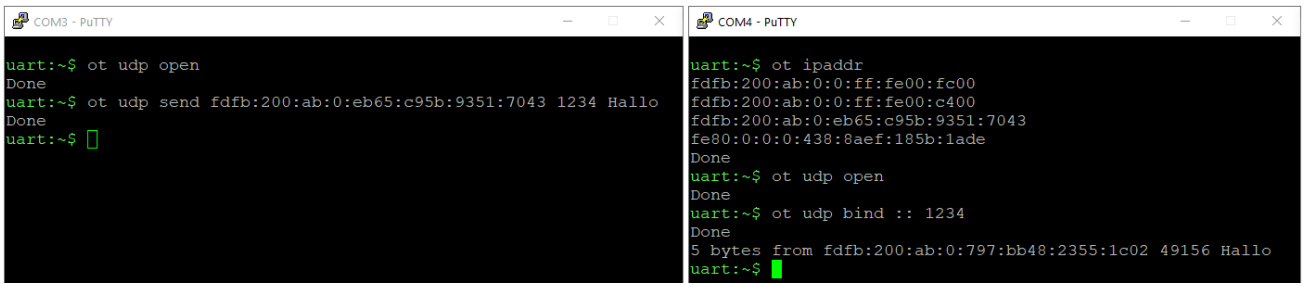


Wichtig sind hier insbesondere zur eindeutigen Identifizierung unseres Netzwerkes die kurze 16-Bit PAN ID und die erweiterte 64-Bit PAN ID (Ext PAN ID). Auch der Network Key für die Verschlüsselung muss auf allen Geräten identisch sein.

Wir können jetzt recht einfach ein UDP-Paket an das zweite Funkmodul versenden, müssen allerdings zunächst dessen IP-Adresse ermitteln, UDP öffnen und einen UDP-Port auswählen, welchen wir abhören möchten:



Über das Terminalfenster unseres ersten Funkmoduls können jetzt Daten an die zuvor ermittelte IP-Adresse und an den ausgewählten UDP-Port gesendet werden:



3 Senden eines UDP-Paketes über die API

Nachdem wir im vorherigen Beispiel ein UDP-Paket über das CLI versendet haben, wollen wir jetzt ein Paket über die API von OpenThread versenden.

Aufgabe 2: Senden eines UDP-Paketes über die API

Erstellen Sie ein eigenes Thread-Netzwerk mit zwei Funkmodulen mit den zuvor gesetzten Parametern. Anschließend senden Sie über die *OpenThread*-API von Funkmodul 1 an Funkmodul 2 ein UDP-Packet. Machen Sie einen Screenshot der Terminalausgabe und legen Sie diesen ihrer Abgabe bei.

Diesmal benötigen wir zwei Projekte, eins für den Sender des UDP-Paketes und eins für den Empfänger. Wir beginnen mit dem Sender-Funkmodul. Machen Sie sich hierzu eine Kopie des Projektes der vorherigen Aufgabe, damit Sie die Projektparameter nicht erneut setzen müssen. Wir werden das Senden eines UDP-Paketes durch Drücken des Buttons 1 auf dem Developerboard auslösen. Wir müssen also zunächst einige Headerdateien einbinden und benötigen das Device für den Button:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/sys/printk.h>
#include <zephyr/net/openthread.h>
#include <openthread/thread.h>
#include <openthread/udp.h>

#define SLEEP_TIME_MS   1000

#define BUTTON0_NODE    DT_NODELABEL(button0)              //DT_N_S_buttons_S_button_0
static const struct gpio_dt_spec button0_spec = GPIO_DT_SPEC_GET(BUTTON0_NODE, gpios);
static struct gpio_callback button0_cb;
```

Anschließend implementieren wir eine Funktion, die den UDP-Socket öffnet und über Port 1234 die Textnachricht „Hello Thread“ an die Broadcast-Adresse ff03::1 sendet:

```
static void udp_send(void){
  otError       error = OT_ERROR_NONE;
  const char *buf = "Hello Thread";

  otInstance *myInstance;
  myInstance = openthread_get_default_instance();
  otUdpSocket mySocket;

  otMessageInfo messageInfo;
  memset(&messageInfo, 0, sizeof(messageInfo));

  otIp6AddressFromString("ff03::1", &messageInfo.mPeerAddr);

  messageInfo.mPeerPort = 1234;

  do{
    error = otUdpOpen(myInstance, &mySocket, NULL, NULL);
    if (error != OT_ERROR_NONE){ break; }

    otMessage *test_Message = otUdpNewMessage(myInstance, NULL);
    error = otMessageAppend(test_Message, buf, (uint16_t)strlen(buf));
    if (error != OT_ERROR_NONE){ break; }

    error = otUdpSend(myInstance, &mySocket, test_Message, &messageInfo);
    if (error != OT_ERROR_NONE){ break; }
```

```
    error = otUdpClose(myInstance, &mySocket);
  }while(false);

  if (error == OT_ERROR_NONE){
    printk("Send.\n");
  }else{
    printk("udpSend error: %d\n", error);
  }
}
```

Die Funktion sieht zwar nach viel aus, der Ablauf ist aber eigentlich recht simpel. Zunächst benötigen wir eine Instanz des Thread-Stacks. Zudem benötigen wir eine Variable für den UDP-Socket und eine für die Informationen der zu versendenden Nachricht. Anschließend öffnen wir die UDP-Verbindung, generieren eine Nachricht und versenden diese mit den zuvor gesetzten Parametern. Dann wird die UDP-Verbindung wieder geschlossen.

Zum Schluss implementieren wir die Callback-Funktion des Buttons und initialisieren diesen in der main-Funktion:

```
void button_pressed_callback(const struct device *gpiob, struct gpio_callback *cb,
                             gpio_port_pins_t pins){
    udp_send();
}

int main(void){
    gpio_pin_configure_dt(&button0_spec, GPIO_INPUT);
    gpio_pin_interrupt_configure_dt(&button0_spec, GPIO_INT_EDGE_TO_ACTIVE);
    gpio_init_callback(&button0_cb, button_pressed_callback, BIT(button0_spec.pin) );
    gpio_add_callback(button0_spec.port, &button0_cb);

    while (1) {
        k_msleep(SLEEP_TIME_MS);
    }
}
```

Diesen Programmcode können Sie bereits kompilieren und auf Funkmodul 1 übertragen. Das Senden einer Nachricht funktioniert so bereits, allerdings soll die Nachricht natürlich auch an einem anderen Funkmodul empfangen werden. Dazu erstellen wir uns eine Kopie des Projektordners von Funkmodul 1 und passen den Sourcecode etwas an. Die Funktion `udpSend`, die Callback-Funktion des Buttons und die Initialisierung des Buttons benötigen wir hier nicht. An Funkmodul 2 müssen wir den UDP-Port `1234` öffnen und auf Nachrichten warten. Dafür implementieren wir zuerst eine Callback-Funktion, welche aufgerufen werden soll, sobald Daten ankommen:

```
void udpReceiveCb(void *aContext, otMessage *aMessage,
                  const otMessageInfo *aMessageInfo){
  uint16_t payloadLength = otMessageGetLength(aMessage) - otMessageGetOffset(aMessage);
  char buf[payloadLength+1];
  otMessageRead(aMessage, otMessageGetOffset(aMessage), buf, payloadLength);
  buf[payloadLength]='\0';
  printk("Received: %s\n",buf);
}
```

Zu beachten ist hierbei, dass wir Bytes als ASCII-Zeichen erhalten und diese in einem `char`-Array abspeichern. Damit Funktionen wie `printk` diese als String interpretieren, muss das Array ein Byte größer sein als die empfangenen Zeichen und als Terminalzeichen an letzter Position '\0' enthalten.

Als nächstes öffnen wir ein UDP-Socket und registrieren dabei auch gleich die Funktion `udpReceiveCb`. Zumdem legen wir noch den UDP-Port fest, auf welchem wir lauschen wollen (*Binding*):

```
static void udp_init(void){
  otError error = OT_ERROR_NONE;
  otInstance *myInstance = openthread_get_default_instance();
  otUdpSocket mySocket;
  otSockAddr mySockAddr;
  memset(&mySockAddr, 0, sizeof(mySockAddr));
  mySockAddr.mPort = 1234;

  do{
    error = otUdpOpen(myInstance, &mySocket, udpReceiveCb, NULL);
    if (error != OT_ERROR_NONE){ break; }

    error = otUdpBind(myInstance, &mySocket, &mySockAddr, OT_NETIF_THREAD);
  }while (false);

  if (error != OT_ERROR_NONE){
    printk("init_udp error: %d\n", error);
  }
}
```

In der Funktion `main` rufen wir diesmal einfach die Funktion `udp_init` auf, anstatt die Buttons zu initialisieren:

```
int main(void){
  udp_init();
  while (1) {
    k_msleep(SLEEP_TIME_MS);
  }
}
```

Wenn wir dieses Programm auf Funkmodul 2 übertragen, sollte die Nachricht erscheinen, sobald wir auf Funkmodul 1 den Button 1 drücken.

4 Constrained Application Protocol (CoAP)

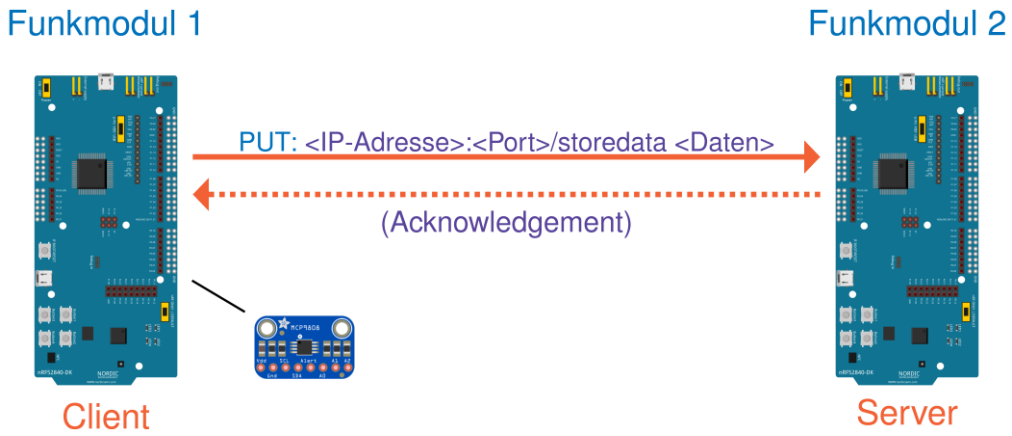
Es ist zwar schon ganz nett, UDP-Nachrichten in einem Thread-Netzwerk verschicken zu können und wir könnten so natürlich auch Werte von unseren Sensoren versenden, aber so richtig komfortabel ist das nicht. Bei dieser Methode müssen wir alle Nachrichtenformate selbst strukturieren und die Verbindungsart muss uns von vorhinein bekannt sein. Außerdem ist UDP ist ein verbindungsloses Protokoll und bestätigt keine Auslieferung eines Paketes. Gegeben falls müssten wir eine Bestätigung selbst implementieren.

Ein Protokoll welches speziell für den IoT-Bereich entwickelt wurde und uns diese Funktionalität zur Verfügung stellt ist das *Constrained Application Protocol (CoAP)*. CoAP ist auch in weiten Teilen in OpenThread implementiert, sodass wir dieses Protokoll für unsere Anwendung recht einfach nutzen können. CoAP funktioniert recht ähnlich dem Internetprotokoll *HTTP*, arbeitet allerdings mit dem verbindungslosen Protokoll

UDP anstatt *TCP*. CoAP hat vier Nachrichtentypen: Confirmable, Non-Confirmable, Acknowledgment und Reset. Zudem bietet es vier Zugriffsmöglichkeiten auf einen Server:

1. GET: Daten vom Server anfordern (nur lesend).
2. PUT: Schreibt oder aktualisiert Daten auf dem Server.
3. POST: Legt neue Daten auf dem Server an.
4. DELETE: Löscht Information auf dem Server.

Wenn wir beispielsweise ein Funkmodul haben, welches die Temperatur ausließt, gibt es zwei mögliche Implementierungen mit CoAP. Unser Funkmodul selbst stellt einen Service zur Verfügung und ein zweites Funkmodul fragt die Temperaturwerte als Client ab. Für unsere Anwendungszwecke ist die allerdings nicht geeignet, denn ein Sensormodul ist gewöhnlich batteriebetrieben, ermittelt lediglich Sensordaten und sollte danach in einen energiesparenden Modus wechseln. Dies ist nicht möglich, wenn das Modul permanent auf Anfragen warten muss. Ein für unsere Zwecke geeigneterer Weg ist, das Sensormodul als Client eines Datenbankmoduls zu sehen. Sind Sensordaten ermittelt, wird eine Speicheranfrage (PUT) an ein zweites Funkmodul gestellt, um dort die Daten zu speichern. Wenn es sich um eine Confirmable-Anfrage handelt, wird der Erhalt dieser Nachricht auch bestätigt.



Wenn wir eine Client/Server-Kommunikation mit CoAP starten wollen, ergibt sich zunächst einmal das Problem, dass wir die IP-Adresse des Servers zunächst nicht kennen. Die ML-EID wird vom Thread-Stack zufällig vergeben. Sie bleibt zwar beim Neustart zunächst einmal gleich, allerdings nur solange wir den nRF52840-Chip nicht komplett löschen. Die beste Lösung wäre es, die Adresse ebenfalls per CoAP-Serviceanfrage abzufragen und diese Anfrage per Broadcast zu verschicken. Wir werden es uns hier aber einfacher machen und dem Server einfach eine zusätzliche IPv6-Adresse selbst vergeben. Die ersten 64-Bit bleibt unsere voreingestellte MeshID und als 64-Bit InterfaceID benutzen wir einfach `1`, so dass unsere komplette IPv6-Adresse `fdde:ad00:beef:0::1` sein wird.

Aufgabe 3: Implementieren eines CoAP-Servers und eines CoAP-Clients

Erstellen Sie ein eigenes Thread-Netzwerk mit zwei Funkmodulen mit den zuvor gesetzten Parametern. Funkmodul 1 soll als CoAP-Server fungieren und Daten in Empfang nehmen. Funkmodul 2 soll als CoAP-Client auf Knopfdruck Daten an den CoAP-Server versenden. Machen Sie einen Screenshot der Terminalausgaben und legen diese ihrer Abgabe bei.

4.1 CoAP-Server Implementation

Benutzen Sie als Ausgangsprojekt für den CoAP-Server das Projekt aus der letzten Aufgabe zum Empfangen einer UDP-Nachricht. Als erstes aktivieren wir die CoAP-API in unserer Projektkofigurationsdatei prj.conf:

```
# Enable OpenThread CoAP support API
CONFIG_OPENTHREAD_COAP=y
```

Als nächstes fügen wir eine Funktion hinzu, um die Server-IP-Adresse zu setzen:

```
void addIPv6Address(void){
  otInstance *myInstance = openthread_get_default_instance();
  otNetifAddress aAddress;
  const otMeshLocalPrefix *ml_prefix = otThreadGetMeshLocalPrefix(myInstance);
  uint8_t interfaceID[8]= {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01};

  memcpy(&aAddress.mAddress.mFields.m8[0], ml_prefix, 8);
  memcpy(&aAddress.mAddress.mFields.m8[8], interfaceID, 8);

  otError  error = otIp6AddUnicastAddress(myInstance, &aAddress);

  if (error != OT_ERROR_NONE)
    printk("addIPAdress Error: %d\n", error);
}
```

Diese Funktion rufen wir später in der main-Funktion auf. Ob die IP-Adressezuweisung funktioniert hat, können wir direkt mit dem CLI und dem Aufruf `ot ipaddr` testen.

Für die CoAP-Funktionalität von OpenThread müssen wir die entsprechende Headerdatei in unseren Deklarationsteil hinzufügen, die Headerdatei für UDP hingegen können wir entfernen:

```
#include <openthread/coap.h>
```

Der CoAP-Server benötigt zwei Funktionen, deren Funktionskopf wir zunächst im Deklarationsteil angeben, um deren Aufruf im Programmcode vor der Implementation nutzen zu können, ohne dass der Compiler uns einen Fehler meldet:

```
static void storedata_request_cb(void * p_context, otMessage * p_message,
                                 const otMessageInfo * p_message_info);
static void storedata_response_send(otMessage * p_request_message,
                                    const otMessageInfo * p_message_info);
```

Die erste Funktion wird aufgerufen, sobald eine Anfrage an den Server eintrifft. Da wir eine zuverlässige Kommunikation ermöglichen möchten, senden wir durch den Aufruf der zweiten Funktion eine Bestätigung nach Erhalt einer entsprechenden Anfrage.

Als nächstes deklarieren wir die Ressource, die unser Server zur Verfügung stellt und definieren ein 30 Byte langes char-Bufferarray, um die empfangenen Daten zu speichern:

```
#define TEXTBUFFER_SIZE 30
char myText[TEXTBUFFER_SIZE];
uint16_t myText_length = 0;

static otCoapResource m_storedata_resource ={
  .mUriPath = "storedata",
  .mHandler = storedata_request_cb,
  .mContext = NULL,
```

```
  .mNext    = NULL
};
```

Jetzt implementieren wir die zuvor deklarierte cb-Funktion `storedata_request_cb`. Unser Server soll nur auf CoAP-Anfragen vom Typ PUT reagieren. Den empfangenen Text geben wir über das Terminalfenster aus und wenn eine Empfangsbestätigung angefragt wurde, verschicken wir diese:

```
static void storedata_request_cb(void * p_context, otMessage * p_message,
                                 const otMessageInfo * p_message_info){
  otCoapCode messageCode = otCoapMessageGetCode(p_message);
  otCoapType messageType = otCoapMessageGetType(p_message);

  do {
    if (messageType != OT_COAP_TYPE_CONFIRMABLE &&
        messageType != OT_COAP_TYPE_NON_CONFIRMABLE) {
      break;
    }
    if (messageCode != OT_COAP_CODE_PUT) {
      break;
    }

    myText_length = otMessageRead(p_message, otMessageGetOffset(p_message),
                                  myText, TEXTBUFFER_SIZE - 1);
    myText[myText_length]='\0';
    printk("%s",myText);

    if (messageType == OT_COAP_TYPE_CONFIRMABLE) {
      storedata_response_send(p_message, p_message_info);
    }
  } while (false);
}
```

Um die Bestätigungsnachricht zu versenden, müssen wir eine neue Nachricht generieren und diese als Antwort (Response) versenden:

```
static void storedata_response_send(otMessage * p_request_message,
                                    const otMessageInfo * p_message_info){
  otError      error = OT_ERROR_NO_BUFS;
  otMessage  * p_response;
  otInstance * p_instance = openthread_get_default_instance();

  p_response = otCoapNewMessage(p_instance, NULL);
  if (p_response == NULL) {
    printk("Failed to allocate message for CoAP Request\n");
    return;
  }

  do {
    error = otCoapMessageInitResponse(p_response, p_request_message,
                                      OT_COAP_TYPE_ACKNOWLEDGMENT,
                                      OT_COAP_CODE_CHANGED);
    if (error != OT_ERROR_NONE) { break; }

    error = otCoapSendResponse(p_instance, p_response, p_message_info);
```

```
  } while (false);

  if (error != OT_ERROR_NONE) {
    printk("Failed to send store data response: %d\n", error);
    otMessageFree(p_response);
  }
}
```

Anschließend implementieren wir noch eine Funktion zum Starten des CoAP-Moduls und registrieren dort auch die Ressource unseres Servers. Der UDP-Port für CoAP ist im Übrigen standardmäßig 5683:

```
void coap_init(void){
  otError error;
  otInstance * p_instance = openthread_get_default_instance();
  m_storedata_resource.mContext = p_instance;

  do{
    error = otCoapStart(p_instance, OT_DEFAULT_COAP_PORT);
    if (error != OT_ERROR_NONE) { break; }

    otCoapAddResource(p_instance, &m_storedata_resource);
  } while(false);

  if (error != OT_ERROR_NONE){
    printk("coap_init error: %d\n", error);
  }
}
```

In der main-Funktion rufen wir nur die Funktionen `addIPv6Address` und `coap_init` auf:

```
int main(void){
  addIPv6Address();
  coap_init();
  while (1) {
    k_msleep(SLEEP_TIME_MS);
  }
}
```

4.2 CoAP-Client Implementation

Als Basisprojekt für den Client erstellen wir eine Kopie vom CoAP-Server-Projekt. Die main-Funktion brauchen wir nur ein wenig anpassen. Die Funktionen `addIPv6Address`, `storedata_request_cb` und `storedata_response_send` können wir komplett aus unserem Programm löschen. Nach dem Aufruf der Funktion `coap_init` aktivieren wir die Buttonfunktionalität:

```
int main(void){
  coap_init();
  gpio_pin_configure_dt(&button0_spec, GPIO_INPUT);
  gpio_pin_interrupt_configure_dt(&button0_spec, GPIO_INT_EDGE_TO_ACTIVE);
  gpio_init_callback(&button0_cb, button_pressed_cb, BIT(button0_spec.pin) );
  gpio_add_callback(button0_spec.port, &button0_cb);
  while (1) {
    k_msleep(SLEEP_TIME_MS);
  }
}
```

Die Callback-Funktion des Buttons implementieren wir wie folgt:

```
void button_pressed_cb(const struct device *gpiob, struct gpio_callback *cb,
                       gpio_port_pins_t pins){
    coap_send_data_request();
}
```

Die Implementation der Funktion `coap_send_data_request` ist etwas umfangreicher. Wie erstellen uns hier zunächst eine neue CoAP-Nachricht, setzen den Typ der Nachricht auf *confirmable* und definieren die Nachricht als PUT-Anfrage. Dann setzen wir die Ressource an welche die Anfrage gerichtet wird (storedata). Anschließend setzen wir den Payload der Nachricht. Wir benutzen als Payload eine Temperatur gespeichert im JSON-Format. Zum Abschluss setzen wir die Ziel-IP-Adresse und den Ziel-UPD-Port und starten einen Sende-Request. Im Request setzen wir auch die Callback-Funktion, die aufgerufen werden soll, wenn wir die Bestätigung der Anfrage erhalten:

```
static void coap_send_data_request(void){
  otError       error = OT_ERROR_NONE;
  otMessage   * myMessage;
  otMessageInfo myMessageInfo;
  otInstance  * myInstance = openthread_get_default_instance();
  const otMeshLocalPrefix *ml_prefix = otThreadGetMeshLocalPrefix(myInstance);
  uint8_t serverInterfaceID[8]= {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01};
  const char * myTemperatureJson = "{\"temperature\": 23.32}";

  do{
    myMessage = otCoapNewMessage(myInstance, NULL);
    if (myMessage == NULL) {
      printk("Failed to allocate message for CoAP Request\n"); return;
    }

    otCoapMessageInit(myMessage, OT_COAP_TYPE_CONFIRMABLE, OT_COAP_CODE_PUT);

    error = otCoapMessageAppendUriPathOptions(myMessage, "storedata");
    if (error != OT_ERROR_NONE){ break; }

    error = otCoapMessageAppendContentFormatOption(myMessage,
                                        OT_COAP_OPTION_CONTENT_FORMAT_JSON );
    if (error != OT_ERROR_NONE){ break; }

    error = otCoapMessageSetPayloadMarker(myMessage);
    if (error != OT_ERROR_NONE){ break; }

    error = otMessageAppend(myMessage, myTemperatureJson,
                                    strlen(myTemperatureJson));
    if (error != OT_ERROR_NONE){ break; }

    memset(&myMessageInfo, 0, sizeof(myMessageInfo));
    memcpy(&myMessageInfo.mPeerAddr.mFields.m8[0], ml_prefix, 8);
    memcpy(&myMessageInfo.mPeerAddr.mFields.m8[8], serverInterfaceID, 8);
    myMessageInfo.mPeerPort = OT_DEFAULT_COAP_PORT;

    error = otCoapSendRequest(myInstance, myMessage, &myMessageInfo,
```

```
                                coap_send_data_response_cb, NULL);
  }while(false);

  if (error != OT_ERROR_NONE) {
    printk("Failed to send CoAP Request: %d\n", error);
    otMessageFree(myMessage);
  }else{
    printk("CoAP data send.\n");
  }
}
```

Von der Callback-Funktion `coap_send_data_response_cb`, wollen wir nur erfahren, ob die Anfrage erfolgreich war:

```
static void coap_send_data_response_cb(void * p_context, otMessage * p_message,
                                       const otMessageInfo * p_message_info, otError result){
  if (result == OT_ERROR_NONE) {
    printk("Delivery confirmed.\n");
  } else {
    printk("Delivery not confirmed: %d\n", result);
  }
}
```

Die Initialisierung des CoAP-Moduls ist bei unserem Client deutlich kürzer, da hier keine Ressourcen registriert werden:

```
void coap_init(void){
  otInstance * p_instance = openthread_get_default_instance();
  otError error = otCoapStart(p_instance, OT_DEFAULT_COAP_PORT);
  if (error!=OT_ERROR_NONE)
    printk("Failed to start Coap: %d\n", error);
}
```

Es empfiehlt sich auch beim Client die benötigten Funktionen bereits im Deklarationsteil anzugeben, damit wir diese im Programmcode an beliebiger Stelle implementieren können, zudem dürfen wir natürlich nicht vergessen die benötigten Variablen für den Button zu setzen:

```
#define BUTTON0_NODE    DT_NODELABEL(button0)              //DT_N_S_buttons_S_button_0
static const struct gpio_dt_spec button0_spec = GPIO_DT_SPEC_GET(BUTTON0_NODE, gpios);
static struct gpio_callback button0_cb;

void button_pressed_cb(const struct device *gpiob, struct gpio_callback *cb,
                       gpio_port_pins_t pins);
static void coap_send_data_request(void);
static void coap_send_data_response_cb(void * p_context, otMessage * p_message,
                                       const otMessageInfo * p_message_info, otError result);
```