UNIVERSITY OF TARTU

Institute of Computer Science
Computer Science Curriculum

Toomas Aleksander Veromann

Embedded Linux-based Smart Home Gateway

Bachelor's Thesis (9 ECTS)

Supervisor: Jakob Mass

Tartu 2018

Embedded Linux-based Smart Home Gateway

Abstract:

This thesis describes a means of developing a custom embedded Linux distribution compatible with devices with ARM processors and limited resources, such as gateways in smart home environments. These devices are usually constrained in computational resources such as RAM and memory space. The Yocto Project is utilized to create and customize an operating system image and configure a set of applications which can make the devices work as useful gateways. The configured applications include a Mosquitto MQTT broker, a web server, and a database to display and persist brokered messages. The output of this thesis is the configured operating system image which was tested in an emulated device environment. The image includes a custom set of applications, which form a smart home data visualisation application.

Keywords:

Embedded Linux, MQTT, Yocto Project, QEMU

CERCS: P170 Computer science, numerical analysis, systems, control

*Embedded Linux*i baasil targa kodu keskseade

Lühikokkuvõte:

Käesolev bakalaureusetöö kirjeldab meetodit, kuidas luua Linuxi manusoperatsioonisüsteemi tõmmis, mis sobituks piiratud ressurssidega ARM-tüüpi protsessoriga seadmetele, nagu näiteks targa kodu keskseadmetele. Sellised seadmed on tavaliselt piiratud nii vähese muut- kui ka välkmäluga. Töös kasutatakse Yocto Projecti, et luua ja kohandada operatsioonisüsteem koos hulga rakendustega, mida kasutades saab seade funktsioneerida kui targa kodu keskseade. Rakenduste hulgas on nii Mosquitto MQTT sõnumivahendaja, kui ka veebiserver ja andmebaas, vahendamaks, kuvamaks ja talletamaks andmeid. Töö tulemusena valmib kohandatud operatsioonisüsteemi tõmmis koos eelmainitud rakendustega. Operatsioonisüsteemi tööd testiti emuleeritud keskkonnas.

Märksõnad:

Embedded Linux, MQTT, Yocto Project, QEMU

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

Table of Contents

1 Introduction

Whether we acknowledge it or not, the internet is all around us. Interconnected devices are sending and receiving vast amounts of data every single day. While not all data is preserved, it is still constantly being produced. This data is partly generated by smart home software and devices, e.g. using an application on a smartphone to schedule brewing a coffee every morning at a certain time, monitoring energy usage in a building room by room, or simply by opening the garden gate moments before arriving at home.

This is, essentially, the Internet of Things (IoT) at its simplest level - a network of devices that communicate with each other and with the server infrastructure. The server infrastructure must have the ability to share received device data and may also provide data back to devices. According to a Cisco report published in 2018, more than 840 zettabytes of data will be generated by IoT applications by 2021 [1]. This is achieved by messaging protocols such as MQTT, which is one of the focus points in this thesis.

It's not just private consumers who are interested in IoT - the International Data Corporation predicts worldwide spending on the Internet of Things to reach nearly $1.4 trillion in 2021 [2], while Gartner reports that the number of connected things will reach 20.4 billion by 2020 [3]. This ever-growing interest in IoT has also improved cooperation between private companies and institutions of higher education, e.g. with Telia Eesti providing University of Tartu's Mobile & Cloud Computing Laboratory with close to a thousand IoT devices for advancing IoT development and studies. The devices include temperature-, light- and movement sensors, smart plugs, and the machines that route and direct the generated data – smart home gateways. [4]

The purpose of this thesis is to create and customize an operating system image and configure a set of applications, which will assist University of Tartu's Mobile & Cloud Computing Laboratory with repurposing some of the recently received Yoga Hub devices to act as useful gateways. The configured applications include a Mosquitto MQTT broker, a web server, and a database to display and persist brokered messages.

This thesis consists of three main sections, and appendices:

- **The first section** provides an overview of the state of the art, covering the background of embedded Linux, the Yocto Project, and MQTT. It also gives an overview of similar works related to MQTT, the Yocto Project, and QEMU and provides a comparison between the works and this thesis;

- **The second section** presents the practical work conducted regarding Yocto and MQTT configuration to mimic an IoT environment. This includes the general configuration and architecture regarding MQTT clients, the Yoga Hub as the gateway, and creating a custom embedded Linux distribution to match the Yoga Hub's limitations;
- **The third section** summarizes this thesis and describes possible future development to be done based on this thesis regarding the repurposing of Yoga Hubs;
- **Appendix A** displays pictures of the Yoga Hub;
- **Appendix B** contains a full specifications sheet of the Yoga Hub;
- **Appendix C** shows custom code written to publish MQTT messages.

2 State of the Art

2.1 Embedded Linux

Embedded Linux is a Linux operating system specifically configured to be used in consumer electronics such as Android phones, smart home devices, routers and switches. The motivation for using embedded Linux on these types of devices is that the devices are constrained by disk space. Due to the lower disk space footprint of embedded Linux, it can fit on devices where regular Linux cannot.

While regular Linux distributions are more generic and are able to run on different devices, embedded Linux distributions are usually set up for certain hardware and users may experience difficulties when trying to run them on devices they are not meant for.

Some of the benefits for choosing Linux as the operating system for an embedded project over alternatives such as Windows Embedded Compact[1] include:

- vast amount of developers and help forums;
- ease of customization;
- relatively small image size;
- and no cost;

Every Embedded Linux project starts with obtaining and customizing the following elements:

- the toolchain;
- the boot loader;
- the kernel;
- and the root filesystem.

The boot process order containing these elements is shown in Figure 1.

[1] http://msdn.microsoft.com/en-ph/embedded/

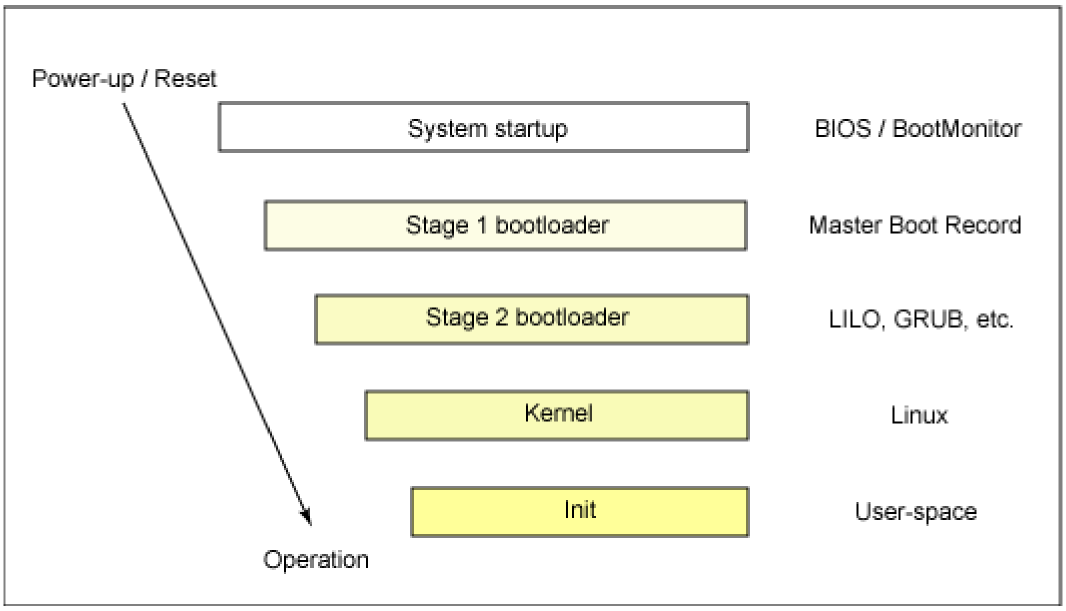


Figure 1. The boot process. [5]

2.1.1 Toolchain

A toolchain is the collection of tools used for compiling source code into executables for a target device, consisting of an assembler, compiler, linker, librarian, and any other tools needed to produce executable code. In the case of Embedded Linux, the toolchain must be able to compile code written in Assembly, C and C++, since these are the languages used to write the base packages. Toolchains are used to build the boot loader, kernel, and root filesystem.

2.1.2 Boot loader

The purpose of a boot loader is to load the operating system into a computer’s memory at boot time [6]. When a computer is powered up or restarted, the Basic Input/Output System performs a power-on self-test, checking whether the device’s hardware is connected and functioning properly. Then, control is transferred to the master boot record (MBR) where the boot loader is located. After making the required system resources operational, the boot loader loads the kernel into RAM and creates an execution environment for it. Optionally, the boot loader may pass a pointer of the root filesystem to the kernel.

In the case of a two-stage boot loader, the first stage resides in the MBR, while the second stage is located in the Volume Boot Record. The task of first stage is to load the second stage boot loader, while also passing on operating system specific data. The second stage, knowing operating system specifics, loads the kernel into RAM.

2.1.3 Linux Kernel

The Linux Kernel is a monolithic one – the entire operating system with its services and core functionality shares the same space, as opposed to a microkernel architecture where functionality is isolated from system services. The main jobs of a kernel are to communicate with hardware via drivers, which are included in the kernel or added via kernel modules, and manage system resources such as memory, tasks and processes, and disk usage.

The kernel gets a root filesystem, either passed as a pointer from the boot loader, or by mounting the device given on the kernel command line. After receiving the root filesystem, the kernel executes the first program, *init* by default. Then, the *init* program starts the preceding programs and gets the system running.

Starting with Linux kernel version 4.0, updates may be applied to the kernel without rebooting the system, thus allowing for system updates with no downtime [7].

2.1.4 Root filesystem

The last required element for an embedded Linux build is the root filesystem. The root filesystem contains system libraries, utilities and scripts used to make the system work. The first program to run is *init*, which helps manage the lifecycle of the system, from boot up to shutdown. *Init*'s priority is to take care of starting *shell*, which in turn is used to start other programs.

Building these elements manually is a tedious task, so projects such as Buildroot[2] and the Yocto Project[3] have been created to automatically build toolchains, boot loaders, kernels and root filesystems. In the next chapters, a more in-depth look at the Yocto Project is provided.

2.2 BusyBox

As mentioned in 2.1, embedded Linux projects are often restricted disk space-wise. As a result, there is a high demand for tools that have a low disk space footprint. **BusyBox**[4] is software that provides hundreds of stripped-down Unix tools, such as *wget*, *grep*, and *unzip*, in a single executable file. Many of the tools it provides are designed to work with interfaces provided by the Linux kernel. It was specifically created for embedded operating systems with very limited resources.

[2] https://buildroot.org/
[3] https://www.yoctoproject.org/
[4] https://busybox.net/

2.3 The Yocto Project

The Yocto Project, Yocto for short, stems from the OpenEmbedded build framework which in turn has its roots in several projects to port Linux to different handheld devices [8]. The OpenEmbedded framework uses the IPK format to create binary packages that can be combined to create a target system and can be installed at runtime. That is achieved by using recipes for each package and having BitBake as the task scheduler.

In 2005, an OpenEmbedded fork called Poky was created by Richard Purdie. Poky had a smaller amount of packages compared to OpenEmbedded and stable releases for Poky were created over time. Even though Poky was a fork, it continued to grow alongside OpenEmbedded, sharing updates with the OpenEmbedded master branch. [8]

In 2008, Intel bought out Purdie's employer OpenedHand and in 2010, Poky was transferred to the Linux Distribution, where the Yocto Project was formed. [8]

The version of Yocto used in this thesis is, as of the time of writing, the latest stable Yocto release - version 2.4.2, codename Rocko [9].

The main elements of the Yocto Project are:

- **Poky.** Nowadays, Poky has evolved from being just an OpenEmbedded fork to Yocto Project's reference distribution. It includes the OpenEmbedded-Core layer along with BitBake and metadata to help users get started on creating their own Linux distributions.
- **BitBake**. BitBake is a build engine that allows the running of shell and Python scripts in parallel in an effective manner. It is a core component of the Yocto Project. BitBake uses recipes to define how packages are built. The first step in a cross-platform BitBake build process is to create a cross-compile toolchain meant for the target platform, which in turn can then build the other required elements.
- **Recipes.** Recipes in Yocto are files that have the *.bb* suffix. Generally, recipes contain information about a particular piece of software, including the download source, dependencies, any possible patches and configuration options to apply, instructions on how to compile the source files and how to package the compiled output.
- **Layers.** Layers in Yocto are collections of recipes, usually gathered around a central theme, e.g. web development, Python or Java support, or secure storage for application data. Premade layers used in Yocto are supplied and maintained by OpenEmbedded [10], while developers may also create their own layers if none suit their interests.

Layers are used on top of the base, OpenEmbedded-Core layer that comes with Poky. The OpenEmbedded-Core layer includes, among other tools, BusyBox. Typically, layer names start with *meta*, e.g. *meta-networking*.

- **Board Support Packages.** Board support packages, or BSPs, contain information about certain devices, including any hardware present (or missing) from the device, drivers, and information about the device kernel. There are a number of BSP-s available for Poky, with the option for developers to create their own custom BSP with the *yocto-bsp* tool. Some example BSP-s include:
    - genericx86 – the generic support for 32-bit x86-based machines;
    - genericx86-64 - the generic support for 64-bit x86-based machines;
    - qemuarm – the QEMU ARM emulation;
    - qemuips – the QEMU MIPS emulation;
    - qemux86 – the QEMU x86 emulation.
- **Images.** Poky distribution also provides premade image recipes that can be used to build custom Linux distributions. The images are essentially configured packages that generate a filesystem that can be used on hardware. Some example images provided by Poky are:
    - *core-image-minimal* – a tiny image, only allowing the device to boot;
    - *core-image-full-cmdline* – a console-only image with full-featured Linux system functionality;
    - *core-image-x11* – an image with a basic graphical user interface.

As seen on Figure 2, all of the aforementioned tools help create a relatively small embedded Linux distribution which can either be put on a physical device or, if one is unavailable, be emulated with various tools such as kvmtool[5] or QEMU[6].

[5] https://github.com/clearlinux/kvmtool
[6] https://www.qemu.org/

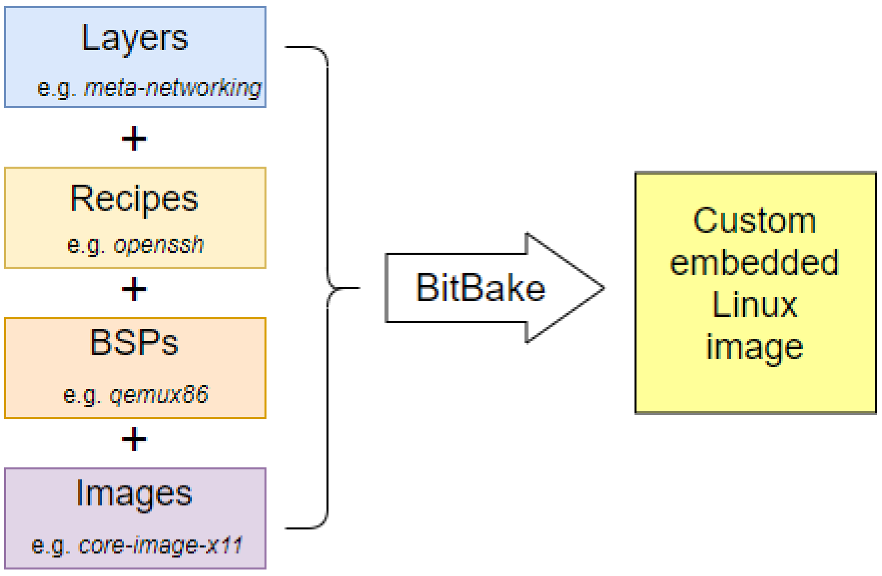


Figure 2. Creating a custom embedded Linux image with the elements of Yocto.

2.4 QEMU

Quick Emulator, or QEMU, is a free and open-source machine emulator and virtualizer that can reach near-native performance through dynamic binary translation – looking at short sequences of code from the source architecture, translating them to the target architecture, and catching the resulting sequences. Code is translated as it is discovered, and pointers to already translated code are kept.

QEMU can also run processes compiled for one architecture in a different one. This is achieved by CPU emulation and proves especially beneficial for developers when prototyping and debugging software intended for embedded devices – instead of running alpha versions of software on the intended target devices, developers can run software on their development machine, to which usually a lot more resources are allocated. [11]

2.5 MQTT

MQTT, also known as Message Queueing Telemetry Transport, is a publish-subscribe-based messaging protocol that works on top of the TCP/IP protocol. In publish-subscribe messaging, the senders of messages, called publishers, do not send messages directly to the intended receivers, called subscribers, but instead categorize published messages into topics, not knowing whether any subscribers exist. Similarly, subscribers do not subscribe to certain publishers' messages, but rather to individual topics, unaware of any potential publishers.

This messaging pattern requires a message broker to allow publishing and subscribing to topics. The broker may filter and route published messages, translate messages from the formal messaging protocol of the publisher to the formal messaging protocol of the subscriber and invoke web services to retrieve data.

In the case of this thesis, the broker is a custom-made Yocto distribution made to mimic a Yoga Hub smart home gateway.

MQTT is designed to be used in devices with low bandwidth and high latency - devices that are generally in unstable conditions. [12]

There is an abundance of available MQTT tools, from the Mosquitto project to Google Chrome extensions and apps such as MQTTLens[7], MQTTBox[8] and MQTT.fx[9]. In this thesis, a combination of Mosquitto and MQTT.fx is used.

2.5.1 Mosquitto

Mosquitto is a lightweight open source MQTT broker that as of April 2018 supports MQTT versions 3.1 and 3.1.1. Mosquitto is highly portable and available on Windows, Mac, many Linux distributions and on iPhones [13].

The Mosquitto project also comes with two command line MQTT clients – *mosquitto_pub* and *mosquitto_sub*. *Mosquitto_pub* is an MQTT client that publishes a single message on a certain topic, and then exits. *Mosquitto_sub*, on the other hand, subscribes to a certain topic and prints the received message to the console. Unlike the publishing client, the default subscription client lasts until it is manually closed. This may be changed by including a parameter to close the subscriber after receiving a certain number of messages.

2.5.2 MQTT.fx

MQTT.fx is an easy-to-use MQTT helper program used for developing and testing MQTT-based applications. MQTT.fx has a built-in scripting tool supported by the Java8 Nashorn JavaScript Engine [14], which allows developers to write their own JavaScript code for MQTT.fx. Since the tool supports *publish*, *subscribe* and *unsubscribe* MQTT commands, along with logging and output to the console, the scripting tool is used to simulate publishing sensor data multiple times over a certain time period.

[7] https://chrome.google.com/webstore/detail/mqttlens/hemojaaeigabkbcookmlgmdigohjobjm
[8] http://workswithweb.com/mqttbox.html
[9] http://mqttfx.jensd.de/

To use MQTT.fx, a connection to an MQTT broker must be established within the application. From there, publishing and subscribing to topics is available, along with authentication, a history of subscribed and published messages and several other tools.

2.6 Related works

The combination of using the Yocto Project alongside QEMU to emulate a MQTT IoT environment is an uncommon union. More works exist that either use the Yocto Project to customize an image for their specific needs, simply use MQTT as a data transfer protocol in their IoT implementation or leverage QEMU to emulate a network of IoT devices that communicate via MQTT. Below, three academic works on these topics are introduced and summarized.

2.6.1 Management of MQTT Devices via an IoT Home Gateway

The authors of [15] show that their MQTT-based IoT gateway solution provides convenience and helps energy management via remote controlling devices in a smart home setting. A method of device discovery via Devices Profile for Web Services (DPWS) is also introduced.

Using Mosquitto as an MQTT broker, the smart home gateway dispenses abstracted messages between devices with heterogenous payloads, in their case, Arduino, Zigbee, and simulated DPWS devices are used. Since different devices transmit data in various forms, an abstracted JSON format is created to generalize the data. The devices are used to control and connect to temperature, light, and humidity sensors, lightbulbs and monitors.

Since some of the devices used don’t support DPWS, auto-configuration for constrained devices using advertisement messages based on User Datagram Protocol is also shown.

2.6.2 Embedded Linux Based Voice Calling Device

In [16], M. Swain uses the Yocto Project to customize an Embedded Linux kernel image suitable for a Raspberry Pi Model B to be used in a voice calling device.

In embedded projects, there is usually not a lot of disk space to spare, so a lot of pressure is put on achieving a low disk footprint. The two alternatives to creating a custom image with Yocto proposed in the paper are Raspbian OS and PILFS. As the former has an image size of 3.2 GB and the latter 1.1 GB, Yocto custom images come at a clear advantage, with the image in the publication having a memory size of 538 MB.

The author demonstrates that a smaller size also means more efficiency, with the custom Yocto image taking less time to run given tasks than its larger counterparts.

2.6.3 IoT Test Environment for Anomaly Detection

Since IoT solutions often consist of many interconnected devices, a real testing environment may not be a viable solution due to the amount of human effort and investment in hardware required. In [17], Brady et al. propose that by creating a realistic emulated IoT environment, human effort and investment could be reduced while improving productivity.

The authors propose a novel tool for emulating network components, called Network Emulator for Mobile Universes (NEMU). NEMU is leveraged to create a testing environment for interconnected and emulated Raspberry Pi devices. The IoT devices are emulated by QEMU.

Three types of IoT systems are compared:

1. a real IoT system built with Raspberry Pi hardware;
2. an emulated IoT system built with QEMU ARM emulation software;
3. and a virtualized IoT system built with QEMU-KVM virtualization software.

The authors perform a MQTT-based data transmitting test, a processor and memory-intensive Java Dacapo Benchmarks test and a network-intensive test. All three tests are conducted within a log analysis context, with the aim being to monitor and detect anomalies in the IoT environments.

Brady et al. show that both emulated and virtualized environments produce test results similar to a real IoT environment, as long as the devices are not under processing, memory or network stress.

2.6.4 Discussion

In all three works, elements of this thesis were used in one way or another. The works showed the advantage of Yocto over other operating system distributions, the benefits of using MQTT as a data transfer protocol and the wide range of applications of QEMU.

[15] demonstrated the use of MQTT devices in an IoT setting, with Mosquitto as the broker of choice, with its low overhead and reliability, in an automated home environment. The authors of [17] used MQTT in conjunction with QEMU to create an emulated IoT environment without allocating a large amount of resources to the process. The focus of the work was on emulating Raspberry Pi devices, whereas in this thesis, the focus is on emulating a Yoga Hub. In [16], the authors showed that, according to process execution time and image size, Yocto proved to be

the most efficient option for creating an embedded Linux image. Since the work concentrated on creating a voice calling device, there was no consideration of MQTT or QEMU.

3 Architecture & System Configuration

3.1 System Overview

To simulate a real-life IoT environment, we consider a network of MQTT clients and a central gateway. Inside the gateway resides an MQTT broker alongside a web server, MQTT subscriber client and database for logging purposes. The skeleton of this network is seen on Figure 3.

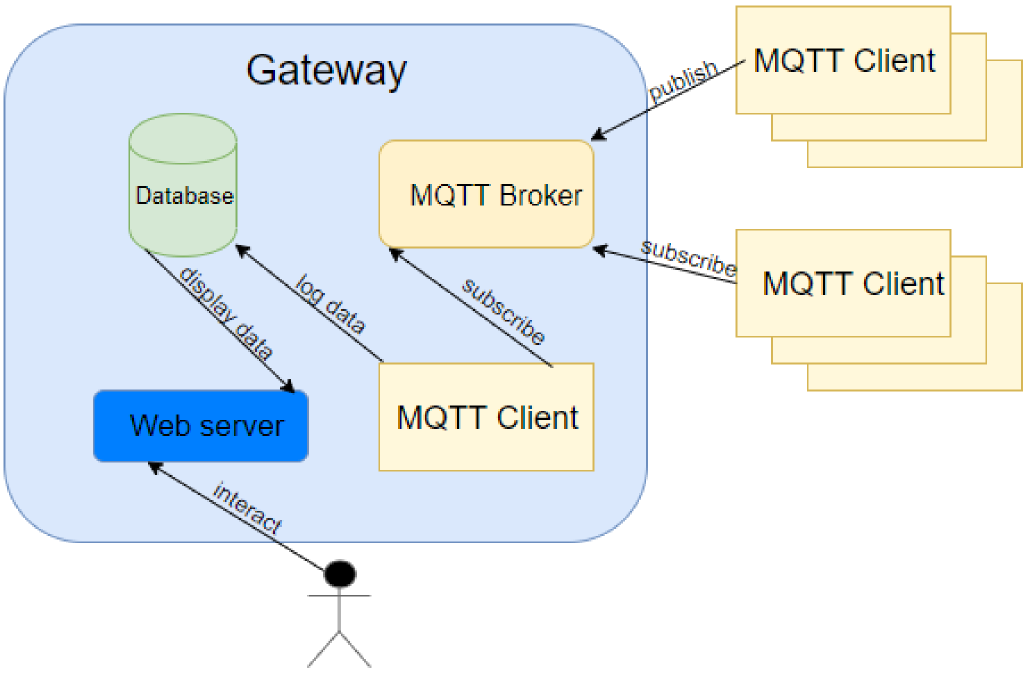


Figure 3. Basic architecture of the network.

The function of the MQTT clients is to publish data to certain topics, and similarly, subscribe to certain topics to receive that information. In a simple home IoT situation, some MQTT clients could publish data about the temperature in rooms, and other clients could subscribe to the topic and use that information to control the heat, thus optimizing the heating system and potentially saving money and energy.

For data to be able to move between clients, an MQTT broker is set up inside a central gateway. Many IoT solutions require the persistence of data, but since MQTT has no built-in message logging, only allowing for the last message published to be retained per topic, a database can be used to keep a history of messages sent. The database receives information from an MQTT subscriber client residing inside the gateway, which is listening to the same topic that the outside clients are publishing on and subscribing to.

A web server is also set up inside the gateway. There are a number of uses for web applications in IoT projects – they are the main user-facing components, providing data visualization via dashboards and graphs, and helping users interface with the broker through submitting commands.

3.1.1 Initial Configuration

With the basic system architecture presented, an environment must be set up to reflect the Yoga Hub.

As shown in Appendix A, the Yoga Hub has the following hardware specifications:

- CPU – 454MHz Freescale i.MX283;
- RAM – 64MB DDR;
- NAND – 128MB.

To emulate running an embedded Linux operating system on this CPU, a Linux-based development environment must be set up - even though QEMU can be run on a Windows host, Yocto needs a supported Linux distribution as a build host [18]. This can be achieved either by running Linux natively or through Docker.

Docker is software providing operating-system-level virtualization. It uses the resource isolation features of the Linux kernel such as *cgroups* and kernel namespaces, and a union-capable file system to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Considering this, coupled with the easy-to-start-over nature of Docker containers, Docker is the more viable option for the project.

Yocto CROPS, or **CROss PlatformS**, leverages Docker containers to create an operating system agnostic Yocto Project development environment. That means development may take place on a Windows, Mac or Linux machine.

When setting up CROPS, two TCP ports are mapped in the container to the respective TCP ports on the host machine, as explained further in section 3.1.2.

After setting up CROPS and including the Poky reference distribution, the *oe-init-build-env* script is executed. This:

- sets the environment variables;
- checks the minimal resources for building the images;

- creates the configuration files.

As a result, the build directory is also created. This directory contains three files:

1. *bblayers.conf* – lists the metadata layers for a custom Yocto image;
2. *local.conf* – contains project-specific configuration variables and recipes;
3. *templateconf.cfg* – contains the directory that includes template configuration files to be used to create an image.

A number of changes must be made to these default files, before building the embedded Linux distribution.

3.1.2 Network configuration

As both the MQTT broker and web server need to be connected to from outside the gateway, separate ports must be allocated. For the broker, the default Mosquitto port, 1883, is used. For the web server, port 3000 is used.

For networking, there are two main options: Slirp and TAP. Since the Poky distribution does not provide root access, which TAP usually requires, setting up TAP interfaces is not an option. Hence, Slirp is used in this thesis.

Slirp is the default networking backend for QEMU which provides a full TCP/IP stack to implement a virtual Network Address Translated network. When using Slirp, port forwarding must be used to allow networking between the guest and the host.

3.1.3 Creating a custom BSP

The Yoga Hub's Freescale i.MX283 applications processor is an implementation of the ARM926EJ-S core [19]. As the default BSP-s offered by Yocto are not configured for ARM926EJ-S architectures, a custom BSP must be created. As mentioned in the State of the Art, Yocto offers a tool, *yocto-bsp*, for developers to easily create new BSPs. Customizable features include device architecture type, kernel version, and touch-screen and keyboard access. To best mimic an IoT gateway like the Yoga Hub, the following setup is chosen:

- a QEMU ARMV926EJ-S architecture;
- Linux kernel version 4.12;
- keyboard access;
- no touchscreen access.

After the new BSP is created, layers and recipes essential for MQTT and the web server must be downloaded, and configuration files updated.

3.1.4 Updating layer and recipe info

Since the default Yocto configuration includes only the minimal required layers, and Mosquitto is not a part of core functionality, it must be added manually. For the broker functionality, the Mosquitto recipe must be added to *local.conf*, and for the MQTT client to work inside the gateway, the Mosquitto-clients recipe is required.

As shown in chapter 2, additional layers can be found in the OpenEmbedded Layer Index. A single Mosquitto recipe can be found for the Rocko branch. This recipe also only has a single dependent layer, *meta-oe*, which is helpful for keeping the image size to as small as possible. [20]

The web server of choice is lighttpd[10], which is included in *openembedded-core*.

3.1.5 BitBaking

Once the necessary layers and recipes are added to the configuration files, the embedded Linux distribution must be built. As described in chapter 2, an image must also be specified as a parameter to BitBake the embedded Linux distribution. As any image with graphical support will exceed the Yoga Hub's 128MB disk space limit, a command-line image must be selected - the *core-image-full-cmdline* image is chosen.

The build process is a lengthy one, taking up to four hours on a machine with an i5-7600 processor and 8 GB of RAM. The build time can be reduced by allocating more CPU and RAM to the Docker container, since BitBake supports multithreading. Modifying the *local.conf* file to allow caching shared-state files and saving downloaded source code tarballs instead of deleting them also improves subsequent build speeds. This also applies to separate builds with different images.

After building the custom images, QEMU is used to run the embedded Linux distribution. The ports mentioned in Network configuration are also forwarded from the Docker container to QEMU. Now, the emulated QEMU machine may be accessed from the host - in the case of this thesis, a Windows machine.

All the elements selected for the embedded Linux build were chosen to minimize the total disk size. Total size of the built image including the boot loader executables, kernel, and filesystems

[10] https://www.lighttpd.net/

was under the Yoga Hub’s 128 MB limit - *core-image-full-cmdline*’s final size was 115.228 MB.

3.2 Implemented Prototype

The final solution consists of three MQTT subscriber clients, an MQTT publisher client and an emulated Yoga Hub gateway. A complete overview of the implemented system is seen on Figure 4, where:

- *a* is the displaying of data;
- *b* is the logging of data;
- *c* is the subscribing to topics from inside the gateway;
- *d* is the publishing of messages;
- *e* is the subscribing to topics from outside the gateway;
- and *f* is interacting with the web server.

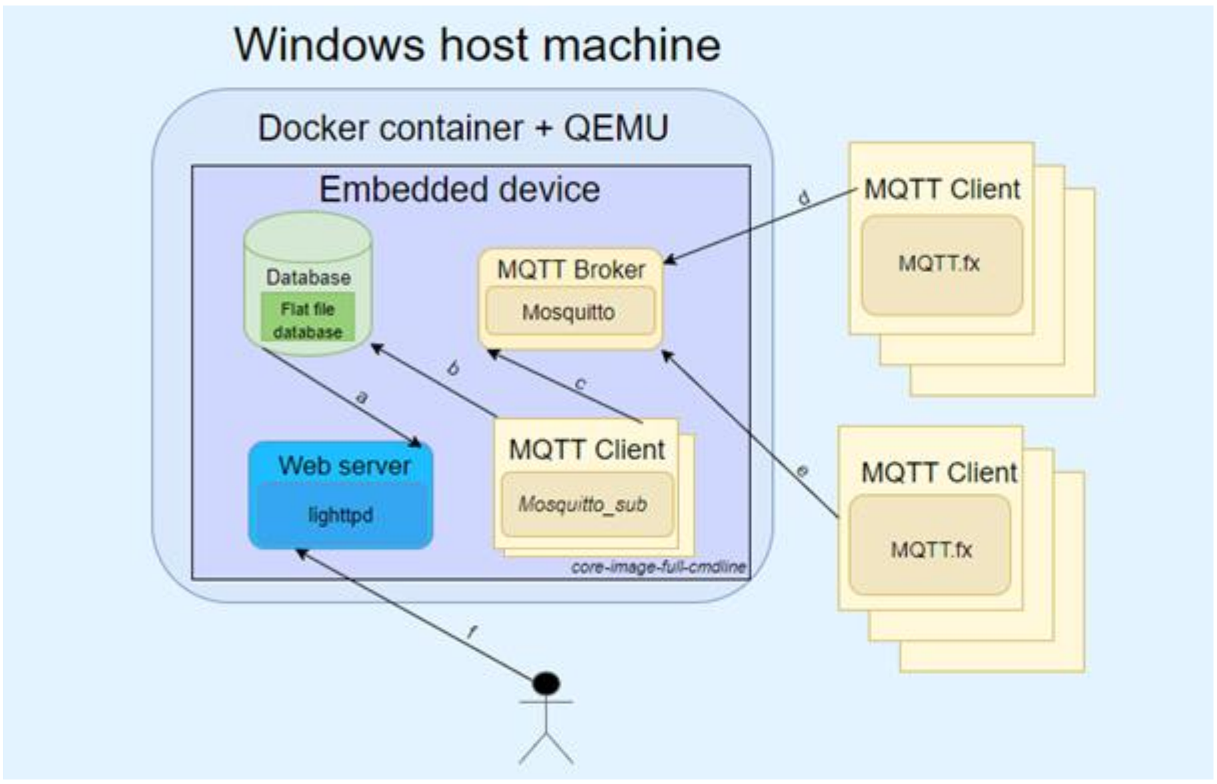


Figure 4. In-depth system overview.

The gateway is running a miniature version of embedded Linux, which has been built and configured with the Yocto Project to take up as little disk space as possible using the *core-image-full-cmdline* image. Inside the gateway on the Linux system, there is a Mosquitto MQTT broker, two Mosquitto MQTT subscriber clients, and a means to save and present data via a flat file database and a lighttpd web server. In the case of this thesis, a plain text file database is used.

One of the MQTT subscriber clients is outside of the embedded device, while the other two are inside. The outside subscriber and the publisher client are running on MQTT.fx with username/password authentication implemented. The publisher is running a custom script, shown in Appendix C. The script is set up to publish randomly generated temperature and humidity data every minute, while the outside subscriber is set up to receive that data. The subscribers inside the gateway are set up to record that data to a text file database, which allows the web server to display the data in a user-friendly manner. A snippet of the MQTT.fx-side log of the script can be seen below:

```
2018-05-01 13:39:01,378  INFO --- MqttFX ClientModel
              : attempt to add PublishTopic

2018-05-01 13:39:01,381  INFO --- MqttFX ClientModel
     : successfully published message 21.08 to topic
      temperature/kitchen (QoS 0, Retained: false)

2018-05-01 13:39:01,387  INFO --- MqttFX ClientModel
              : attempt to add PublishTopic

2018-05-01 13:39:01,387  INFO --- MqttFX ClientModel
     : successfully published message 46.19 to topic
       humidity/kitchen (QoS 0, Retained: false)
```

The Mosquitto MQTT broker is running on default settings, the only exception being the added authentication. In a real-world situation, using authentication is a best practice, to minimize the chance of message interception. Mosquitto has a number of ways to authenticate, such as username/password authentication, pre-shared-key based encryption, and certificate based encryption of messages. In this thesis, username/password authentication is used. The broker is also running on the default port 1883. [21]

The lighttpd web server configuration remains mostly unchanged, only the default port is changed to 3000. Lighttpd also comes with a default HTML index page, but the HTML must be changed to display latest database values to the user - C3.js[11] is leveraged to create graphical

[11] http://c3js.org/

representations of transmitted MQTT data. The contents of the configured web page can be seen on Figure 5.

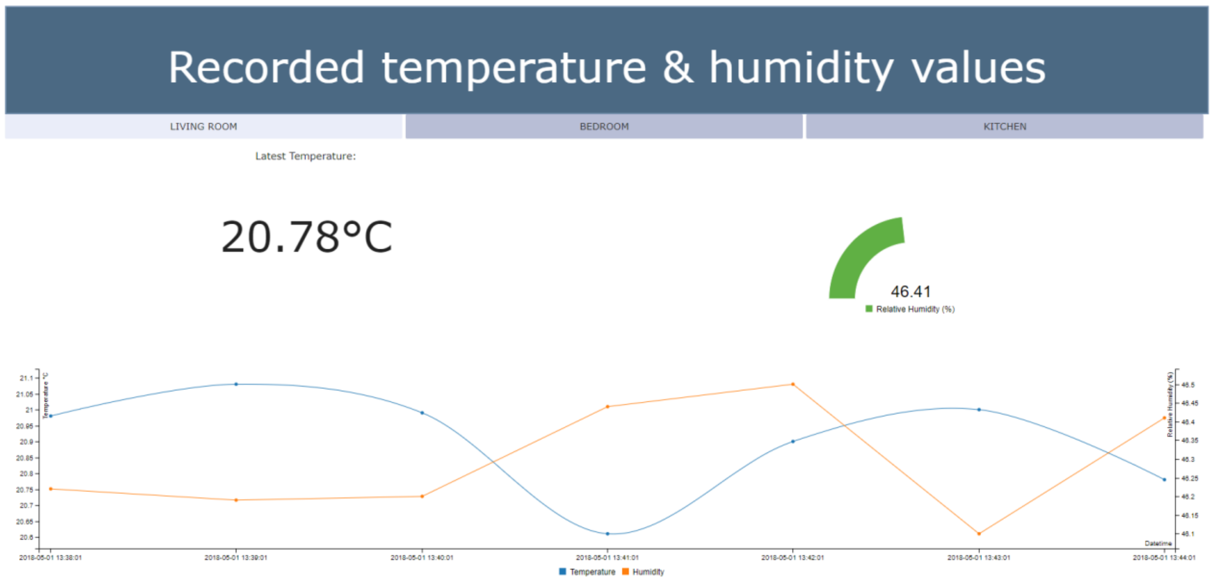


Figure 5. The web server dashboard, including line and gauge charts.

The result is a web application dashboard that presents up to date device and sensor data to users in a meaningful way – graphs being much more intuitive than, for example, JSON payloads.

The web server can further be improved by creating more graphs, and by incorporating AJAX to update data without refreshing the page, but in the context of this thesis, AJAX is not a priority.

4 Conclusion

In this thesis, the Yocto Project and MQTT were studied, and a practical output was provided in the form of an embedded Linux distribution suitable for University of Tartu’s Mobile and Cloud Computing Lab’s newly acquired Yoga Hub gateway devices. The thesis also gives the background on embedded Linux, MQTT, and the Yocto Project, and covers some related works.

An IoT network, with simulated external sensor devices implemented using MQTT.fx clients and a custom script to mock sensors sending and receiving data, was created. The data was distributed through a gateway, which was running a custom embedded Linux build created specifically to fit on the memory of the Yoga Hub. Inside the gateway was a Mosquitto MQTT broker, alongside a subscriber client, a web server and a database. The subscriber client inside the gateway served as a logger to insert information into the database. From the database, the web server could retrieve data and display it to the user.

The conducted work provides a base for developers and researchers to create embedded Linux distributions for IoT gateways based on their specific needs.

4.1 Future developments

As the image, *core-image-full-cmdline*, can fit in the Yoga Hub’s 128MB flash memory, using that to create an embedded Linux distribution to run on the Yoga Hub would be a viable option. A possible future work may also involve adapting the results of this thesis to run on the physical Yoga Hub device.

If a more secure system is required, I highly recommend using certificate-based authentication for MQTT. Since no sensitive data was transmitted in the practical work, only username/password authentication was used, but in any real-life scenario where sensitive data is transmitted, certificate-based authentication should be considered over username/password authentication to reduce the chances of data breaches.

For a more elegant and developer-friendly way of web server development, one of many database recipes provided in the OpenEmbedded layer index [10] could be added to the build and implemented within the device.

5 References

[1] “Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper”, Cisco.com, [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html (Accessed 13th of March 2018)

[2] “Worldwide Spending on the Internet of Things Forecast to Reach Nearly $1.4 Trillion in 2021, According to New IDC Spending Guide”, Idc.com, [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS42799917 (Accessed 13th of March 2018)

[3] van der Meulen, R. “Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016”, Gartner.com, [Online]. Available: https://www.gartner.com/newsroom/id/3598917 (Accessed 13th of March 2018)

[4] Eljand-Kärp, V. “Telia kinkis Tartu ülikooli asjade interneti laborile 920 seadet, millega laiendada oma labori uurimisteemasid”, Ut.ee, [Online]. Available: https://www.ut.ee/et/uudised/telia-kinkis-tartu-ulikooli-asjade-interneti-laborile-920-seadet-millega-laiendada-oma (Accessed 13th of March 2018)

[5] Attila, A. “Operating Systems - Lecture Notes: Chapter 2. Booting up”, Gyires.inf.unideb.hu, [Online]. Available: https://gyires.inf.unideb.hu/GyBITT/20/ch02s02.html (Accessed 19th of March 2018)

[6] “Choosing a Boot Loader”, Control-escape.com, [Online]. Available: http://www.control-escape.com/linux/bootload.html (Accessed 19th of March 2018)

[7] “Linux_4.0 – Linux Kernel Newbies”, Kernelnewbies.org, [Online]. Available: https://kernelnewbies.org/Linux_4.0 (Accessed 20th of March 2018)

[8] Simmonds, C. Mastering Embedded Linux Programming – Second Edition. Packt Publishing. 2017.

[9] “Releases”, Yoctoproject.org, [Online]. Available: https://wiki.yoctoproject.org/wiki/Releases (Accessed 23th of March 2018)

[10] “OpenEmbedded Layer Index” , Openembedded.org, [Online]. Available: http://layers.openembedded.org/ (Accessed 27th of March 2018)

[11] “QEMU”, Qemu.org, [Online]. Available: https://wiki.qemu.org/Main_Page (Accessed 1st of April 2018)

[12] “FAQ - Frequently Asked Questions | MQTT”, Mqtt.org, [Online]. Available: http://mqtt.org/faq (Accessed 1st of April 2018)

[13] “Download | Eclipse Mosquitto”, Mosquitto.org, [Online]. Available: https://mosquitto.org/download/ (Accessed 1st of April 2018)

[14] Deters, J. “MQTT Toolbox – MQTT.fx”, Hivemq.com, [Online]. Available: https://www.hivemq.com/blog/mqtt-toolbox-mqtt-fx (Accessed 2nd of April 2018)

[15] Kim, Seong-Min & Choi, Hoan-Suk & Rhee, Woo Seop. (2015). IoT home gateway for auto-configuration and management of MQTT devices. 12-17. 10.1109/ICWISE.2015.7380346.

[16] Swain, Mahendra & Srivastava, Abhishek. (2015). Design of Embedded Linux Based Voice Calling Device. International Journal of Applied Engineering Research. 10. 35095-35102.

[17] Brady, Shane & Hava, Adriana & Perry, P & Murphy, John & Magoni, Damien & Portillo, Omar. (2017). Towards an Emulated IoT Test Environment for Anomaly Detection using NEMU. 10.1109/GIOTS.2017.8016222.

[18] “Yocto Project Quick Start”, Yoctoproject.org, [Online]. Available: https://www.yoctoproject.org/docs/2.4.2/yocto-project-qs/yocto-project-qs.html (Accessed 17th of April 2018)

[19] “i.MX28 Applications Processors for Consumer Products“, Nxp.com, [Online]. Available: https://www.nxp.com/docs/en/data-sheet/IMX28CEC.pdf (Accessed 1st of April 2018).

[20] “mosquitto 1.4.14”, Openembedded.org, [Online]. Available: http://layers.openembedded.org/layerindex/recipe/70416/ (Accessed 1st of April 2018)

[21] “mosquitto.conf man page”, Mosquitto.org, [Online]. Available: https://mosquitto.org/man/mosquitto-conf-5.html (Accessed 1st of April 2018)

Appendices

A Pictures of the Yoga Hub

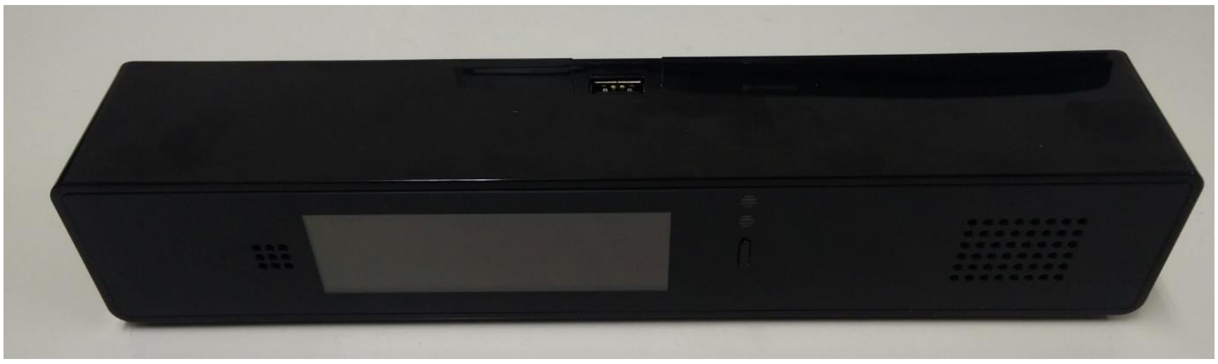


Figure 6. The powered-off Yoga Hub.

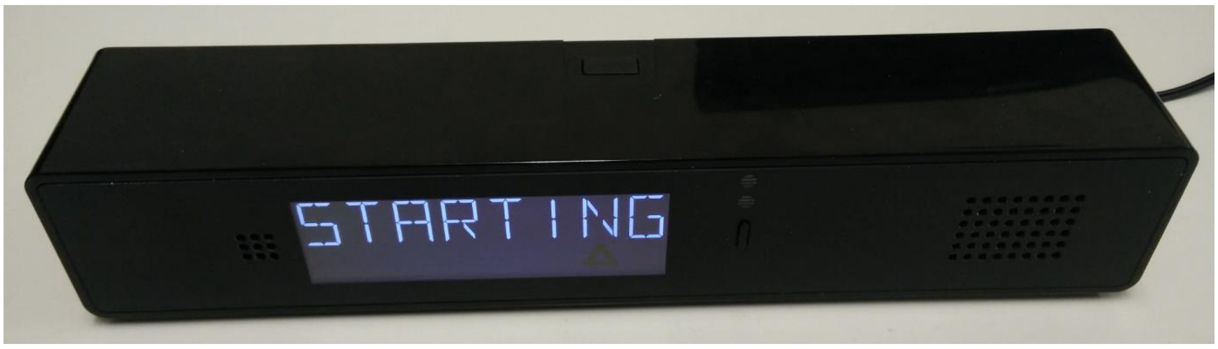


Figure 7. The Yoga Hub booting up.

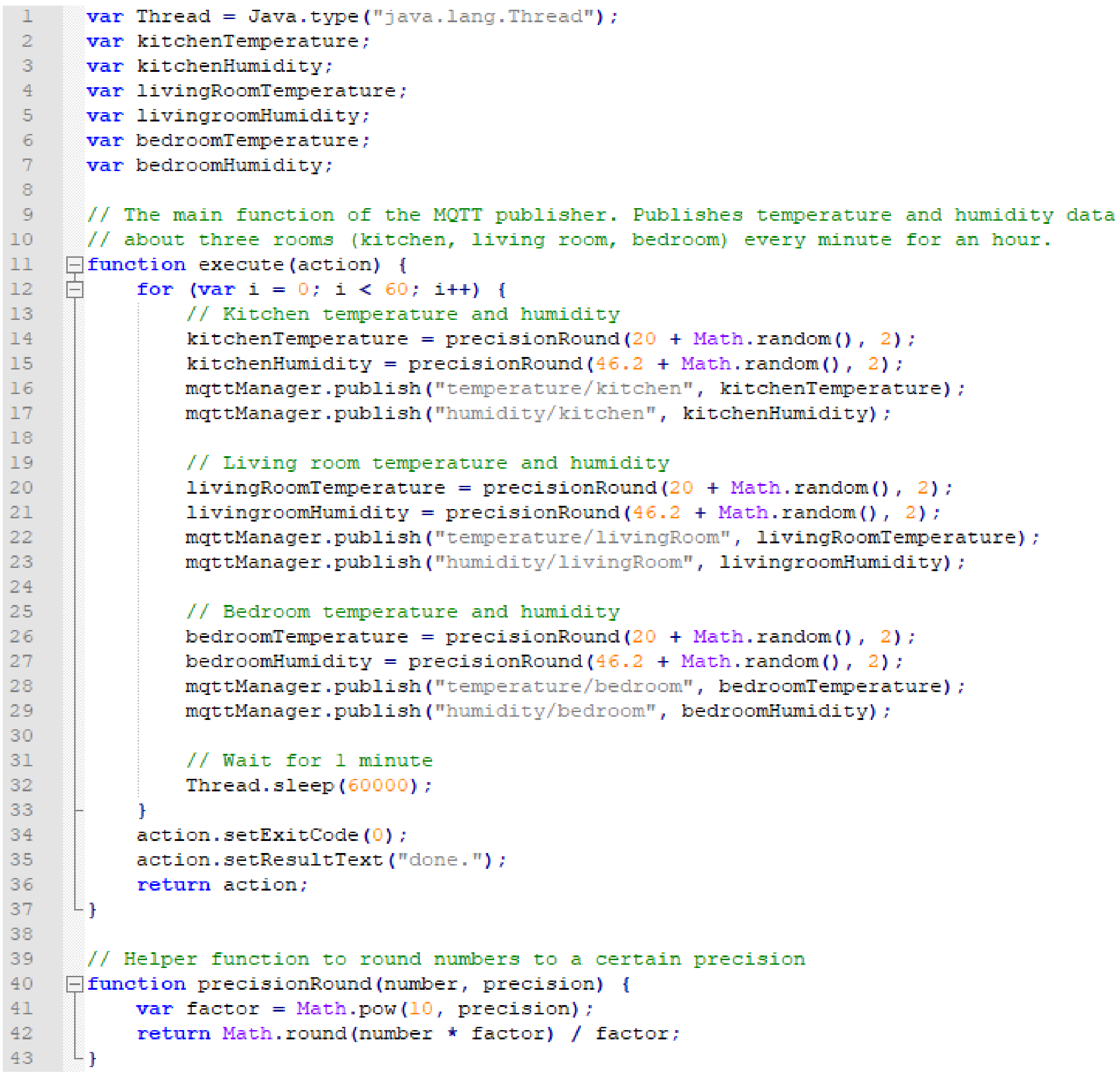
B Yoga Hub specifications sheet

Table 1. The Yoga Hub specification sheet.

Product detailed name	Yoga Hub
General classification	Main Controller
Dimensions (WxHxD)	254x49x46mm
Weight	290g (399g with adapter)
System	CPU – 454MHz Freescale IMX283 RAM - 64MB DDR2 NAND - 128MB
Local User Interfaces	105dB siren (from 10cm), speaker, LCD display, button
Communication	Ethernet, USB, ZigBee PRO, GPRS

Maximum Connected Devices	20 ZigBee PRO devices
Power Supply (nominal)	5V DC 2A
Power Supply (backup)	2000mAh Li-Ion-Pol battery inside (3.5..4.2V), lifetime ~3 years Operating life time from battery: 4h (charging time 10h)
Power Consumption	Max 10W
Operational Temperature	0..+40°C
Storage Temperature	-20..+50°C
Relative Humidity	10..95% non-condensing

C Custom MQTT.fx publishing script



```
var Thread = Java.type("java.lang.Thread");
var kitchenTemperature;
var kitchenHumidity;
var livingRoomTemperature;
var livingroomHumidity;
var bedroomTemperature;
var bedroomHumidity;

// The main function of the MQTT publisher. Publishes temperature and humidity data
// about three rooms (kitchen, living room, bedroom) every minute for an hour.
function execute(action) {
    for (var i = 0; i < 60; i++) {
        // Kitchen temperature and humidity
        kitchenTemperature = precisionRound(20 + Math.random(), 2);
        kitchenHumidity = precisionRound(46.2 + Math.random(), 2);
        mqttManager.publish("temperature/kitchen", kitchenTemperature);
        mqttManager.publish("humidity/kitchen", kitchenHumidity);

        // Living room temperature and humidity
        livingRoomTemperature = precisionRound(20 + Math.random(), 2);
        livingroomHumidity = precisionRound(46.2 + Math.random(), 2);
        mqttManager.publish("temperature/livingRoom", livingRoomTemperature);
        mqttManager.publish("humidity/livingRoom", livingroomHumidity);

        // Bedroom temperature and humidity
        bedroomTemperature = precisionRound(20 + Math.random(), 2);
        bedroomHumidity = precisionRound(46.2 + Math.random(), 2);
        mqttManager.publish("temperature/bedroom", bedroomTemperature);
        mqttManager.publish("humidity/bedroom", bedroomHumidity);

        // Wait for 1 minute
        Thread.sleep(60000);
    }
    action.setExitCode(0);
    action.setResultText("done.");
    return action;
}

// Helper function to round numbers to a certain precision
function precisionRound(number, precision) {
    var factor = Math.pow(10, precision);
    return Math.round(number * factor) / factor;
}
```

Figure 8. Custom publishing script for MQTT.fx.

D License