An IoT application using the Zynq SoC

João Filipe Mota
Joao.f.mota@tecnico.ulisboa.pt
Instituto Superior Técnico,
Universidade de Lisboa

Abstract

This thesis analysis the capabilities of a kit for development of IoT applications. The kit includes a board with the Zynq SoC, an I/O expansion board, a thermocouple, an Arduino shield with built-in sensors and the IBM cloud services, the IBM Cloud. For this analysis several sensors connected to the kit generate information that is then sent to the IBM Cloud. The board is running a Linux operating system compiled and saved in an SD card, from which the system boots. Sensors connected to the board communicate using I^2C and SPI protocols. The drivers for the sensors were developed as loadable Linux kernel modules. An application that reads data from the sensors and, using MQTT protocol, sends them to the IBM Cloud, was developed. Data is analysed and presented to the user in a graphical way, using cloud services.

Keywords: I^2C, IoT, Linux, Cloud, Embedded Systems, SPI

1. INTRODUCTION

The term Internet of Things (IoT) was introduced for the first time by Kevin Ashton in a presentation [1] to draw attention to RFID (Radio Frequency Identification) potentialities. The Internet term in the IoT designation represents the proposal of the IP protocol as the means of communication between the so called "things". Those "things" can be, for an instance, sensors and actuators.

IoT technology allows the application to generate data, transmit it and to analyse it without human intervention, besides decisions can be taken based on the information received. The standard architecture of an IoT system [2] includes components (the things) that generate data, like measurements of the surrounding environment, and transmit data, through the Internet into computers and servers interconnected by the Internet (a Cloud). The IoT architecture also includes IoT agents that are responsible for the analysis of the information and make decisions based on it. Those IoT agents reside inside the Cloud can apply algorithms based on Machine learning, Artificial Intelligence or a simple comparison of values. IoT agents can just store received data or make decisions based on data analysis [3].

The development of IoT technology led to the appearance of software platforms whose intent is to make the development of applications easier. These platforms, usually, include Cloud services and data analysis services. There are quite a few options nowadays, with platforms being developed by big companies such as Amazon [4], Microsoft [5], Google [6] and IBM [7] and a few others from not so well-known companies like TheThings.io [8], Databoom [9] and Altair [10]. All these platforms offer Cloud services and data analysis. However, costs can differ quite a bit. The bigger providers charge according to the number of messages, while smaller companies can charge by device, service, etc. Usually, for the basic package there are more services on those from the bigger companies.

There are currently some kits that can be used for IoT development. A kit can be software and/or hardware platforms. These platforms must possess an Internet connection and the capability to make some local processing in order to implement the IP protocol. Here are a few examples:

- MicroZed Industrial IoT Starter kit [11]: This kit has an ethernet port, an I/O that is compatible with Arduino shields and an ARM processor.
- Gemalto Cinterion ConceptBoard [12]: This kit has 2G and 3G connectivity, a microprocessor and it is also compatible with Arduino shields.
- Cellstick cellular (GSM/GPRS) IoT Platform [13]: it is a board that has GSM/GPRS connectivity and a microprocessor that can run Arduino sketches natively.
- C027 Mbed enabled Internet of Things kit [14]: it is a board that has an ARM processor, with pins that support multiple interfaces supported (SPI, I^2C, UART e I^2S).

The objective of this paper is to describe the development of an IoT application protype using the development platform from Avnet, some sensors and IBM Cloud services. The work is to:

- Assemble new sensors to the MicroZed expansion board.
- Develop specific drivers and alteration of the device tree file.
- Establish communication between Cloud services and MicroZed using the MQTT protocol.
- Set the Cloud services to accept and recognize the sensors connected to the MicroZed.
- Establish new graphical interfaces with the data received in the Cloud in real time using Watson services.

In this paper, in section 2 it is presented the concepts needed to understand IoT and an IoT application development reference model. Section 3 is about Linux in embedded systems, drivers and device tree files. In section 4 the hardware and software of the platform used in the thesis is presented. In section 5 the work done is presented and finally, in section 6 the conclusion of that work.

2. IOT: CONCEPTS AND REFERENCE MODEL SYSTEM

In this chapter concepts of IoT are presented as well as a reference model for IoT systems.

2.1 IoT concepts

According to ITU – International Telecommunications Union, things are objects from the physical world or from the information world, that are capable of being identified and integrated into communication networks [15]. What is gathering both definitions is the Internet. IoT has a purpose: providing a service.

An IoT can, hypothetically, have an infinite number of "things" connected and that must be possible so the ITU developed a reference IoT model that promotes the interoperability between IoT applications and communication technologies.

2.2 IoT reference model

The IoT reference model is an abstract point of view of what an IoT system is. It is presented in Figure 1.

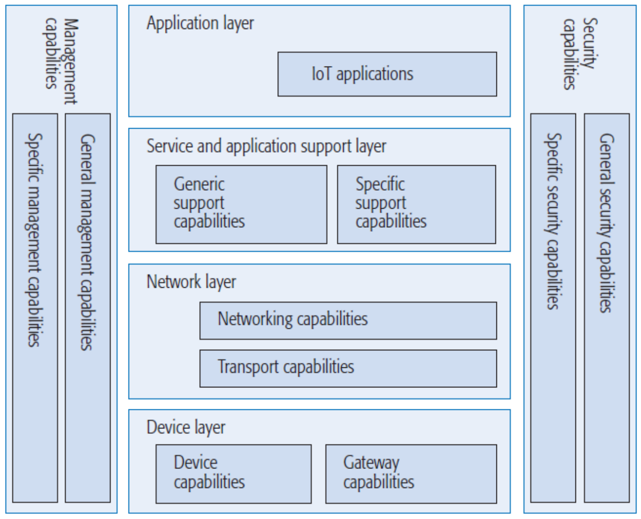

Management capabilities
Specific management capabilities
General management capabilities
Application layer
IoT applications
Service and application support layer
Generic support capabilities
Specific support capabilities
Network layer
Networking capabilities
Transport capabilities
Device layer
Device capabilities
Gateway capabilities
Security capabilities
Specific security capabilities
General security capabilities

Figure 1 – ITU reference model. Extracted from [15].

The model has 4 horizontal layers and 2 vertical ones. Vertical layers are common to all the horizontal layers, in spite of being different in each layer. They take care of security and management capabilities. As IoT applications handle sensitive information, it is mandatory that information transfers are secure, and that devices are not infected with malicious software. The management capabilities take care of the expansibility of an IoT system. Any device that is added to a system must be capable of handling the information that it produces or receives. The general management capabilities are related to remote control, status monitoring and so on. The specific capabilities are device specific and can be access control, authentication, confidentiality etc.

The Device layer gathers both device capabilities and gateway capabilities. Device capabilities are about analogic to digital conversion and network communication. Gateway capabilities are about protocol translation between network and device and vice-versa. The device communication can either be direct or through a gateway when the device lacks that capability. This layer includes the deployed hardware like sensors and actuators.

The Network layer has two types of capabilities: network and transport capabilities. Network capabilities are about establishing and maintaining a connection between the device layer and Service and Application Support layer, this includes access control, routing and management of the mobility of information. The transport capabilities are about the bidirectional transport of information between the Device layer and the Service and application support layer.

The Service and Application Support transforms commands generated by the Application layer into commands to the devices. This layer is also responsible for storing and combining the relevant information and feed it to the application when necessary.

The Application layer is the highest layer in the model, being responsible for the interaction with the user. This layer generates the instructions that are then converted by the Service and Application support layer in controls to the devices. Applications can be for example smart homes, smart transportation systems etc.

3. LINUX DRIVERS IN EMBEDDED SYSTEMS

Embedded systems are microprocessor or microcontroller-based systems that aim to perform one or two specific tasks [16]. Those systems can have constrained resources. There are Linux distributions light enough to run in those systems, Linux is flexible in terms of connectivity capabilities which is quite useful given the nature of embedded systems.

In embedded systems the communication with the peripherals is crucial. This communication is made possible through drivers. A driver works as an abstraction layer that provides an interface between the applications (that belong to the user space) and the device (physical hardware) that is connected to the processor through a bus. This communication interface is usually through a file present in the file system. The application accesses the file to require something and the kernel turns those requirements into commands for the device and then transmit the data received through that file.

Drivers are installed in the kernel. They can be compiled with the kernel or they can be loaded to the kernel during runtime as LKM (Loadable Kernel Module). The modular approach is better for development of kernel modules as it provides the possibility of inserting and removing the modules without having to recompile the whole kernel.

Spidedev generic driver

Spidev is a generic driver of the type character [17]. The communication using this driver is made through a file as reads and writes. It instantiates a data structure for each mention it gets from the device tree file and it also creates a file for each mention in the /dev directory in the user space. The working principle of this file is presented in Figure 2. If an application wants to send/receive data from the sensor it has to access the file spidevX.Y in the /dev directory and make a write/read to that file, respectively. The X stands for the device identification and the Y stands for the bus controller identification. The spidev driver informs the SPI controller that it intends to make a read from device X. The read is made by SPI Controller Y and then written in the file that will be read by the application in case the intent of the app was to read from the device. If the application wanted to write, the application would have written to the file and that write would have been transmitted to the device by the SPI Controller Y after the spidev warned it to.

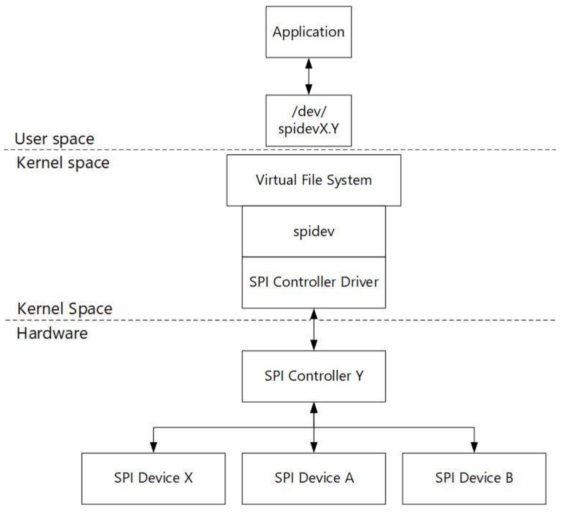


Figure 2 - Communication with a SPI device using the spidev driver.

I^{2}C drivers

The structure in the kernel for the I^{2}C has a modular structure divided between buses and devices. That structure is presented in Figure 3.

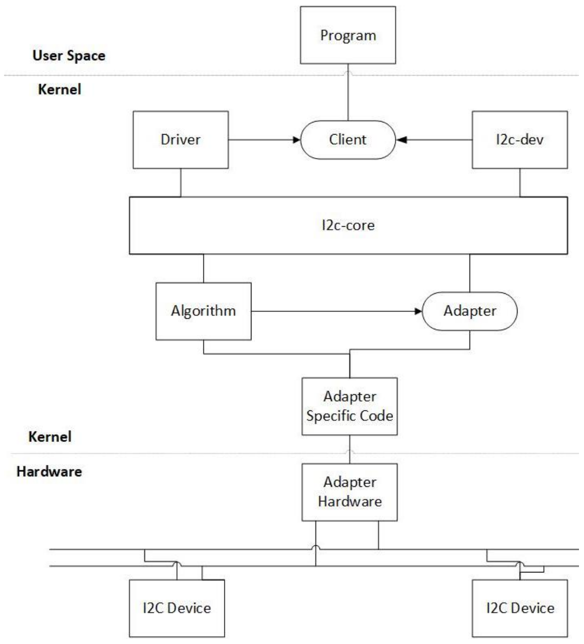


Figure 3 - Architecture of an I^{2}C driver. Adapted from [18].

The I^{2}C protocol is implemented by the i2c-core. The devices are represented by the drivers Driver and Client besides the i2c-dev that is a generic driver. The Driver represents a peripheral that is a slave of an I^{2}C bus.

The buses are characterized by the drivers Algorithm and Adapter. Algorithm defines which algorithm is used to write and read while the Adapter is responsible for the association between a bus and a processor. The Linux kernel supplies the Algorithm for a few processor (namely for ARM ones).

The communication with the devices is achieved through files that are created based on the hardware description (device tree) - The drivers are identified in the file /proc/bus/input/devices by their name. That file also identifies which files to use to communicate with the devices. Those files are called in a generic way eventX (where X =0,1,2, …n) and are in /dev/input. The control is based on /sys/bus/i2c-X/X-address, where X is the controller and address is the address of the device in the bus X.

Hardware description

The hardware description is used by the kernel to know what is connected to the system, so the system can know which driver to use for each device. This description is provided by the binary file dtb that is not a part of the kernel. The dtb file is achieved by compiling a dts (device tree source) file. This file has a textual description of the hardware in the shape of a tree where each device is represented by a node. Each node contains a description of the device resources and the hardware configuration of the device. The processor does not have direct access to the I^{2}C and SPI devices as they are not memory mapped. It can access those devices through the bus controller. Every device connected to a I^{2}C or SPI bus must be represented as sons of the controller node of that bus.

4. Development platform

This chapter describes the hardware and software included in the MicroZed Industrial IoT Development kit.

4.1 Hardware

The kit includes a MicroZed board, a carrier board, an Arduino shield from STMicroelectronics with built-in sensors and a module for temperature measurement with a thermocouple from Maxim.

Figure 4 shows the MicroZed plugged in the carrier board, the empty place where the Arduino shield will be placed and the PMOD plugs.

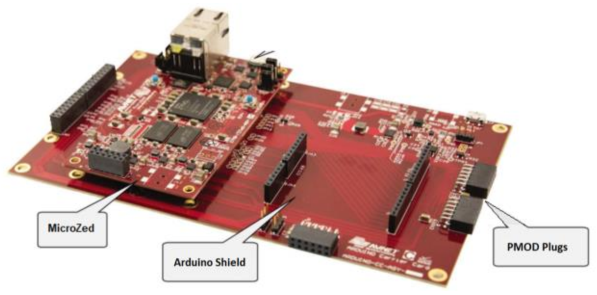


Figure 4 - Carrier board with the MicroZed plugged in (adapted from [19]).

The PMOD plug placed higher in the Figure 4 is connected to the PL of the MicroZed while the one below is connected directly to the Zynq PS (Processing System). The development platform is supplied with a 5V from an external source.

The Arduino shield has 4 built-in sensors: a temperature and humidity sensor, an accelerometer and 3 axis gyroscope, a 3 axis magnetic sensor and a pressure sensor. All the sensors are connected to the Zynq's PL through their I^{2}C interfaces.

The temperature measurement module is connected to the Zynq PL through the PMOD plug. This module, that has a K-type thermocouple connected, includes the conditioning electronics that make the compensation of the cold-junction temperature. It has an

SPI interface but does not use MOSI (Master Out Slave Input) and, therefore sends data periodically to the master.

The MicroZed board has 2 sets of a 100 microheaders, JX1 and JX2. The carrier card connects those microheaders to its pins so the both mentioned PMOD plugs and the pins for an Arduino shield are connected to the MicroZed through them. The tables Table 1,

Table 2 and

Table 3 show the mapping between the Arduino pins, MicroZed, Zynq pins and also the PMOD plugs.

Table 1 - Mapping of the microheaders JX2 with the PLPMOD pins and Zynq.

PL PMOD	MicroZed JX2	Zynq Bank 35
P1	61	G17
P2	63	G18
P3	62	F19
P4	64	F20
P5	81	N15
P6	83	N16
P7	82	L14
P8	84	L15

Table 2 - Mapping of the microheaders JX1 with the Arduino pins and Zynq.

Arduino	MicroZed JX1	Zynq Bank 34
ARD_SCL	29	Y16
ARD_SCA	31	Y17
D13	30	W14
D12	32	Y14
D11	35	T16
D10	37	U17
D9	36	V15
D8	38	W15
D7	47	N18
D6	49	P19
D5	48	N20
D4	50	P20
D3	53	T20
D2	55	U20
D1	54	V20
D0	56	W20
A0	97	L10
A1	99	K9
A2	98	M9
A3	100	M10

Table 3 - Mapping of the microheaders JX2 with the Arduino pins and Zynq.

Arduino	MicroZed JX2	Zynq Bank 34
A4	14	J15
A5	18	B19

In Table 4 the signals from the sensors from the Arduino shield are mapped to the pins of the MicroZed and in Table 5 the mapping of the thermocouple module to the Zynq pins.

Table 4 -Mapping of the signals from the sensors to the pins of the MicroZed.

Signals from the sensors	Arduino	MicroZed JX1	MicroZed JX2	Pins of the Zynq
M_INT1	A2	98		M9
M_INT2	A3	100		M10
LIS3MDL_INT1	A4		14	J15
LIS3MDL_DRDY	A5		18	B19
I2C_SCL	SCL	29		Y16
I2C_SDA	SDA	31		Y17
USER_INT	D2	55		U20
LSM6DS0_INT1	D4	50		P20
LPS25H_INT1	D5	48		N20
HTS221_DRDY	D6	49		P19

Table 5 - Connections of the thermocouple module to MicroZed and Zynq.

Thermocouple (SPI)	PMOD	JX2	Pino da MicroZed
SS (chip enable)	1	61	G17
MISO	3	62	F19
SCK	4	64	F20

4.2 PL Hardware

The PL configuration provided by Avnet uses six IP cores from Xilinx connected by AXI Interfaces. The IP cores are:

- ZYNQ7 Processing System: This IP is a wrapper for PS. The inputs and outputs of this block are inputs and outputs of PS. This block generates the clock signal and the reset signal used by the other block, and receives data and interrupts from them
- Processor System Reset: This IP core receives the reset signal and distributes it to the other IPs that need it
- Concat: This IP gathers the signals from all of the sensors (Table 4) and interrupts from AXI IIC and AXI Quad SPI blocks into a single 7-bit bus that is connected to IRQ_F2P port of the wrapper module.
- AXI Interconnect: This block establishes the communication between the blocks connected to the sensors and the ZYNQ7 Processing System wrapper.
- AXI Quad SPI: This block implements data communication with sensors, using the SPI protocol.
- AXI IIC: This block implements data communication with sensors, using the I^2C protocol.

4.3 Software

The software provided by Avnet consists in the files to generate the operating system, the files to generate device drivers and a file with the hardware description (Device Tree).

4.3.1 *Operating System*

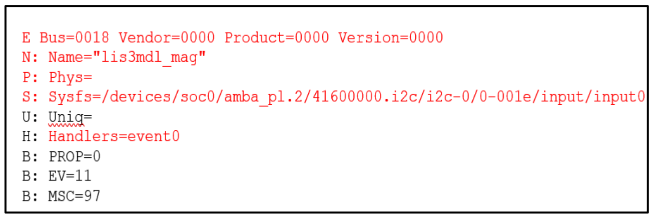
The operating system is Pulsar Linux 7.0.0.11, for embedded systems developed by Wind River. The boot process is important in the scope of this paper as it shows which files are needed for the system to start and what is their part on it.

The system boots when the power supply is turned on, one of the ARM processors runs the code in the ROM (bootROM), detecting that the boot is to be done from the SD card and running the boot.bin file.

The boot.bin is a FSBL (First Stage BootLoader) file that combines the bitstream, to configure the Zynq PL, the uboot-25MHz.elf to load the operating system. The u-boot file loads the Linux kernel (u-image) and creates a temporary file system in DDR memory and loads device tree binary file. The boot process finishes when a login page appears.

4.3.2 *Drivers*

The sensors LIS3MDL, LPS25HB and HTS221 are controlled by specific I^{2}C drivers represented by the Driver in Figure 3, if they were generic drivers, they would be i2c_dev. Since they are not generic it is possible to see in Figure 5 that the sensor LIS3MDL (Name=) is connected to the bus i2c-0 and that it has an event associated (Handlers =). It is the same for the rest of the sensors.



```
E Bus=0018 Vendor=0000 Product=0000 Version=0000
N: Name="lis3mdl_mag"
P: Phys=
S: Sysfs=/devices/soc0/amba_pl.2/41600000.i2c/i2c-0/0-001e/input/input0
U: Uniq=
H: Handlers=event0
B: PROP=0
B: EV=11
B: MSC=97
```

Figure 5 – Excerpt of the contents of the file /proc/bus/input/devices

Figure 6 and Figure 7 show the contents of /dev/input/ and /sys/bus/input/i2c-0/. Analysing the three figures it is possible to conclude that the data for the sensor LIS3MDL is read from the file /dev/input/event0 and that the device control (write) is achieved by the driver through /sys/bus/input/i2c-0/0-001e/.

```
crw-rw---- 1 root input 13, 64 Oct 24 10:15 event0
crw-rw---- 1 root input 13, 65 Oct 24 10:15 event1
crw-rw---- 1 root input 13, 66 Oct 24 10:15 event2
```

Figure 6 - Contents of the Directory /dev/input.

```
drwxr-xr-x 4 root root     0 Oct 24 10:15 0-001e
drwxr-xr-x 4 root root     0 Oct 24 10:15 0-005d
drwxr-xr-x 4 root root     0 Oct 24 10:15 0-005f
```

Figure 7 - Folders from the directory /sys/bus/input/i2c-0.

The kernel image for the platform does not include the specific drivers for the sensors LIS3MDL, LPS25HB and HTS221. Those had to be compiled and then installed as LKM modules.

The dts file describes the Hardware configuration of the Figure 8.

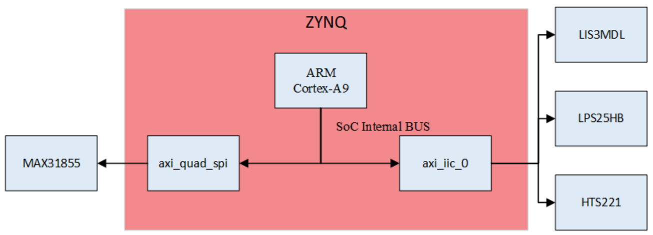


Figure 8 - Bus controllers (axi_quad_spi and axi_iic_0) and external sensors to the MicroZed.

SPI devices

Figure 9 shows the description of the SPI temperature sensor module. The interrupt-parent property stands that the interrupt parent of the SPI bus to which the sensor is connected is represented in Figure 10 that has the same value in the phandle property: 0x3. In the sensor node the compatible property is used to tell the system the programming model for the device. The system uses this property to assign a driver to the device in the boot process.

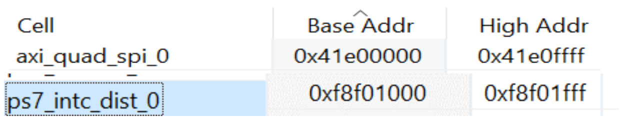
```
axi_quad_spi@41e00000 {
compatible = "xlnx,xps-spi-2.00.a";
interrupt-parent = <0x3>;
        interrupts = <0x0 0x1d 0x1>;
        reg = <0x41e00000 0x10000>;
        xlnx,num-ss-bits = <0x1>;
        #address-cells = <0x1>;
        #size-cells = <0x0>;

        tempSensor@0 {
                #address-cells = <0x1>;
                #size-cells = <0x0>;
                compatible = "spidev";
                spi-max-frequency = <0x2faf08>;
                reg = <0x0>;
        };
};
```

Figure 9 – Temperature sensor in dts file.

```
interrupt-controller@f8f1000 {
        compatible = "arm,cortex-a9-gic";
        #interrupt-cells = <0x3>;
        interrupt-controller;
        reg  =  <0xf8f01000  0x1000  0xf8f00100
        0x100>;
        num_cpus = <0x2>;
        num_interrupts = <0x60>;
        linux,phandle = <0x3>;
        phandle = <0x3>;
};
```

Figure 10 - Interrupt controller in the dts file.

In the Figure 11 it is presented the memory map of the *axi_quad_spi* controller and interruptions to help to understand the example.



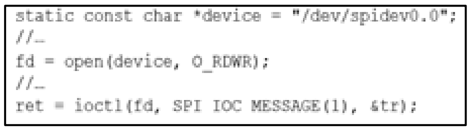
Cell	Base Addr	High Addr
axi_quad_spi_0	0x41e00000	0x41e0ffff
ps7_intc_dist_0	0xf8f01000	0xf8f01fff

Figure 11 – Memory map of the SPI bus controller and the interrupt controller.

In Figure 9 it is possible to see the description of the bus controller and the thermocouple node. The compatible property specifies which device is connected there, this string is used for the system to search for a driver that supports such device. The interrupt-parent property tells who is the interrupt controller responsible for the device interruptions. The interrupt property specifies how many interrupts the device can generate and what kind. In the reg property is specified the range of addresses used by the device, and address-cells property tells how many 32-bit cells are needed to represent the device address. The size-cells tells how many 32-bit cells are needed to specify the address range of the device. Lastly the property xlnx,num-ss-bits states how many CS (chip select) are used.

4.3.3 *Applications*

Avnet provides some application examples that show how the user space can communicate with the physical hardware. In the Maxim Max31855 example, the application opens the file located in /dev/spidev0.0 to read data using ioctl (input-output control) the function as described in Figure 12.



```
static const char *device = "/dev/spidev0.0";
//…
fd = open(device, O_RDWR);
//…
ret = ioctl(fd, SPI IOC MESSAGE(1), &tr);
```

Figure 12 - Access to the Maxim Max31855 sensor.

The access to the I^2C sensors follows the same logic but use the functions read and write instead of ioctl.

4.3.4 *Watson Platform*

The Watson Platform includes a cloud to store data and services to handle and analyse data, the Watson IoT agent. Watson Platform provides services using machine learning techniques and cognitive APIs (Application Programming Interfaces) to make predictions; it supports connection from devices, networks and gateways that use standard protocols such as HTTP and MQTT. Watson Platform can also manage those devices and applications.

Several entities are created in this platform to develop an IoT application:

- Organization: Identified by the organization-ID, org-ID, that is obtained when registering to the Watson platform.
- Device: "something" that can send and receive data, through the Internet. Devices are configured to receive or to subscribe a list of commands. Devices must be registered to the platform., receiving a unique identifier.
- Gateways: Access points to connect devices to the internet.
- Events: Messages sent by devices to publish data in the platform.
- Commands: Messages sent by the platform to devices.

To receive data from devices, they must be previously registered to the platform.

MQTT Protocol

MQTT stands for MQ Telemetry transport. It operates over TCP/IP using the publish-subscribe paradigm between clients and a server [20].

There are three different levels of QoS (Quality of Service) in MQTT. The minimal QoS level is zero. QoS level 0 does not require an acknowledge receipt from the receiver. QoS level 1 guarantees that a message is delivered at least one time to the receiver. QoS1 continues to send the same message, in specific time intervals, until it gets the acknowledge. QoS 2 guarantees that each message is received only once by the intended recipients.

The communication is achieved by sending control packages. The MQTT packet or message format consists in a fixed header plus a Variable-header and payload:

- Fixed header: The most significant byte tells the type of packet (CONNECT, SUBSCRIBE, etc) in the 4 MSB, and in the 4 LSB are the flags needed for that type. The next 1 up to 4 bytes are for the package size.
- Variable size header: It depends on the type of message and it is used for sending control information.
- Payload: It is used for the data. Not all messages have data, for an instance a CONNECT message does not have data to send.

5. Work done

This chapter presents the work done in two phases, the first one is the work done to setup the devices and make them communicate with Watson Platform. The second one deals with the adding of an different sensor than the ones that came in the box.

5.1 Initial Configuration

The initial work included the generation of an SD card, generation of the first stage bootloader (boot.bin), the Linux kernel and the device tree blob.

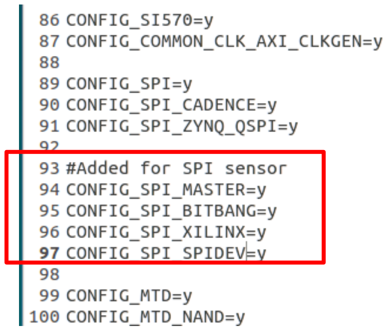
FSBL generation

The boot.bin, that includes the PL hardware description, is created in the Vivado software. The PL hardware is created using the bitstream file by executing the make_mz_acc_iiot.tcl script in the Vivado Xilinx 2016.4 software. This script instantiates the previously described IP cores and interconnects them. The script also gets the constraint file, where it maps the ports of the Zynq to the ports of the MicroZed. From this it is generated the bitstream and it is exported to the SDK. The SDK is responsible for the programming of the PS, here it is built the file boot.bin, a first stage bootloader. It is built by combining the bitstream file with the u-boot file provided by Avnet. This boot.bin file is copied to the SD card after the Operating System is already saved.

Operating System

The operating system is generated from the *wr-core* directory that Avnet provides. This folder contains all the files necessary to build the Operating System. The Operating System is generated using a *bitbake* command. The *bitbake* is a compiling tool like make but with its focus directed mainly to cross compiling and software packages. When building this Operating System, it compiles it in the environment used (Ubuntu 14.04 LTS) to run on a different system (The MicroZed). The *bitbake* is configured by the files with .conf extension. These files contain the information on what it is needed to compile the objects from the respective folders.

Before the execution of the bitbake command it is needed to change the file *Xilinx-zynq.cfg* located in *wr-core/layers/wrlabs-integration/recipes-kernel/linux/files/* within the lines 93 to 97 from Figure 13. These lines are responsible for the insertion of the SPI generic driver in the kernel when it is built with the *bitbake* command.



```
CONFIG_SI570=y
CONFIG_COMMON_CLK_AXI_CLKGEN=y

CONFIG_SPI=y
CONFIG_SPI_CADENCE=y
CONFIG_SPI_ZYNQ_QSPI=y

#Added for SPI sensor
CONFIG_SPI_MASTER=y
CONFIG_SPI_BITBANG=y
CONFIG_SPI_XILINX=y
CONFIG_SPI_SPIDEV=y

CONFIG_MTD=y
CONFIG_MTD_NAND=y
```

Figure 13 - xilinx-zynq.cfg file already edited.

$bitbake cube-server cube-dom0 cube-essential

This generates the folder *build-xilinx-zynq* in the wr-core directory. The following command creates an image that can be saved to an SD card, the pulsar7-avnet-microzed-7010.img:

```
sudo ../overc-installer/sbin/cubeit --force \
--config `pwd`/../install_templates/ZedBoard/MicroZed-
7010-live.sh \
--target-config MicroZed-7010-live.sh \
--artifacts         `pwd`/tmp/deploy/images/xilinx-zynq
pulsar7-avnet-microzed-7010.img
```

Device tree blob generation

The device tree is also available in the github from Avnet and it considers already the devices from the kit. So, it only needs to be compiled with the command:

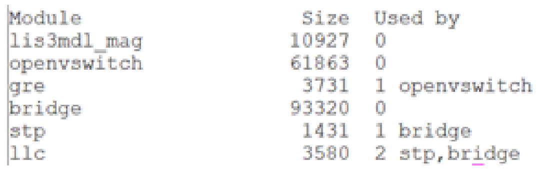
$ dtc -I dts -O dtb -o dtb ./pulsar_spi_i2c.dts

This command generates a dtb file that is a *device tree blob.*

5.2 Sensors integration

The Maxim Max31855 sensor communicates with the kernel using the spidev generic driver which is already in the kernel. To achieve measurements from this sensor it is only necessary to compile and run the application code provided by Avnet. That application uses the ioctl function that sends one message to the device through the file pointed by the file descriptor fd (which is set to /dev/spidev0.0). The message is inside the buffer tx_buf and the received message is inside the rx_buf. The sensor returns a 32-bit message that includes the cold junction temperature, the temperature measured by the thermocouple, 1 bit that tells if it there was an error and 3 more to identify what kind of error: short-circuit to vcc, ground or if the thermocouple is in open-circuit.

The other sensors present in the kit use proprietary drivers that are not inserted in the kernel by default. Those drivers need to be generated from the source code supplied by Avnet using a bitbake command. This bitbake command will generate a rpm package. This package is sent to the MicroZed and inserted in the kernel. The result of this operation can be seen in Figure 14 where it is shown the LIS3MDL driver on top of the list.

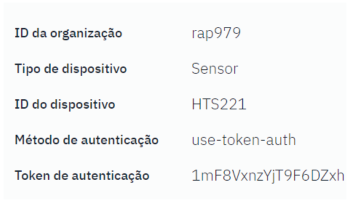


```
Module                  Size  Used by
lis3mdl_mag            10927  0
openvswitch            61863  0
gre                     3731  1 openvswitch
bridge                 93320  0
stp                     1431  1 bridge
llc                     3580  2 stp,bridge
```

Figure 14 - List of modules in the kernel.

5.3 Using Watson Platform to create an IoT application

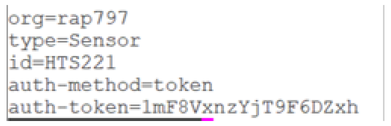
The registration process returned the Organization ID, rap979, as can be seen in Figure 15. Sensors are also need to be registered in the IBM platform. Figure 17 shows credentials generated for the HTS221 sensor. This device is registered as a sensor (devices can be either sensors or gateways). A description and the manufacturer are also introduced during the registration process. Security selected is a token-based authentication. The result of registration of the HTS221 sensor if presented in Figure 15.



ID da organização	rap979
Tipo de dispositivo	Sensor
ID do dispositivo	HTS221
Método de autenticação	use-token-auth
Token de autenticação	1mF8VxnzYjT9F6DZxh

Figure 15 - Credentials generated for HTS221 sensor.

These credentials are saved to a file device.cfg presented in Figure 16

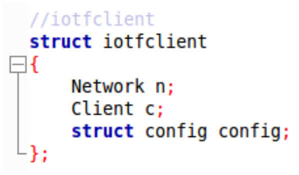


```
org=rap797
type=Sensor
id=HTS221
auth-method=token
auth-token=1mF8VxnzYjT9F6DZxh
```

Figure 16 - Contents of device.cfg.

The application uses the MQTT protocol to communicate with the Watson Platform. To use the MQTT protocol the struct iotfclient was used to manage the connection. As seen in Figure 17 the struct has 3 fields:

- Network n: it is a network socket with 3 pointers to functions: read, write and disconnect.
- Client c: it is used to save network parameters and communications, it has buffers to receive and send messages, timeout value, an integer to tell if the connection is established.
- config config: it stores the credentials provided by de file device.cfg.



```
//iotfclient
struct iotfclient
{
    Network n;
    Client c;
    struct config config;
};
```

Figure 17 - Iotfclient struct.

After this struct is initialized with the credentials used the function Connectiotf is called and it creates:

- a hostname: using the id supplied. It created Id.messaging.internetofthings.ibmcloud.com. for the present case of HTS221 sensor it gets: rap979.messaging.internetofthings.ibmcloud.com.
- a clientId: d:ID:DeviceType:DeviceID, for the HTS221 sensor is d:rap979:Sensor:HTS221.
- The network field is initialized with the hostname mentioned, with the port 1883 (non-encrypted or 8883 for encrypted with token).

- A MQTTdata struct that stores the MQTT version, ClientID and flags: keepAliveInterval, cleansession, Willflag, will, username and password.
- The username and password: the username is "use-token-auth" that will be interpreted by Watson as the communication is encrypted by token. The password is the token used.

This function ends by calling the function MQTTConnect that establishes the connection with the broker and Watson, if the credentials are right, starts receiving data.

The messages are sent using a JSON format presented in Figure 18 where the XXXX part is replaced by the actual measurement.

```
if (dataSource == i2ch)
    ptr = strcpy(json_str, "{\"d\" : {\"humd\" : XXXXX }}");
else
    ptr = strcpy(json_str, "{\"d\" : {\"temp\" : XXXXX }}");

ptr = pack_json(ptr, sensorVal);
```

Figure 18 - Message format.

The IBM platform can present with real-time incoming data from devices. For this effect it must be created a card in the plates tab by specifying what kind of graph the user wants to see, the topic used, colour and the size. The graph obtained is presented in the Figure 19.

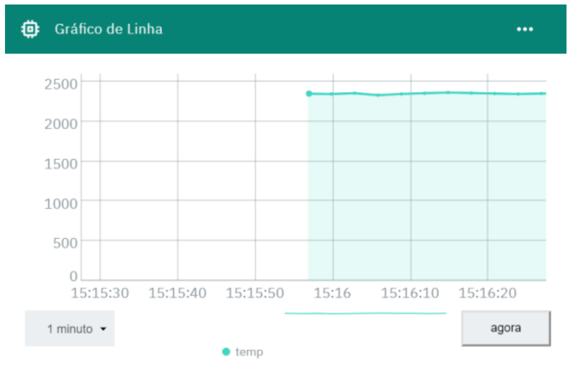


Figure 19 - Graph presenting the input received fromHTS221 sensor.

5.4 Integration of si7021 sensor in the IoT application

The sensor Si7021 is an I^{2}C sensor so there is no need to change the PL as it uses the same SDA and SCL as the other sensors. The changes to include this sensor are solely software based. The module of the sensor can be seen in Figure 20.

Figure 20 - Si7021 module.

This module has 4 pins, 2 for power supply (Vcc and Ground) and 2 for SCL and SDA. Since there is no need to change the hardware it is possible to skip that phase and go straight to the device tree. The I^{2}C address used is 0x41 [21] that is going to be used in the reg property. The field compatible is going to be "sl, si7021", where "sl" are the initials of Silicon Labs (the manufacturer) and "si7021" is the name of the sensor. The device tree node used is presented in Figure 21.

```
i2c5@41 {
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    compatible = "sl,si7021";
    reg = <0x41>;
};
```

Figure 21 - Node added to the device tree.

The name i2c5@41 is used due to the sensor being the 5th device connected to this i2c bus (i2c5) and 41 (hexadecimal) is the I^{2}C address of the device for a write instruction. Now a driver must be created.

The driver built has 5 structs directly related to the device. These structs have 2 types: The first type is responsible for tasks such as inserting, probing and removing the driver from the kernel and the second type is responsible for storing the configuration of the device and its current state. In the first type there are 2 structs:

- device_driver: It is used by the system to match the driver with the corresponding devices.
- i2c_driver: It represents a driver of an I^{2}C device. It identifies which functions should be called for insert, removing and probing the device. It also has a pointer for a list of the devices supported by the current driver.

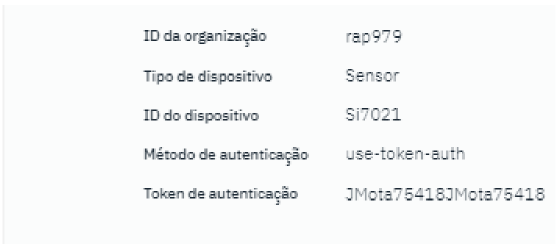
The second type of structures has 3 elements:

- si7021_platform_data: this struct is responsible for storing device configurations such as data resolution, polling and also for storing references to the functions responsible for the insertion, removal, connection and disconnection of the device.
- i2c_client is used to represent the device as an I^{2}C slave. This struct holds the device I^{2}C address and is used in the writing and reading functions. It is used in the connection between the kernel and the device.
- si7021_status: It is responsible for storing the current state of the sensor. Inside there are two pointers: one for an i2c_client struct and another for a si7021_platform_data. It is responsible for saving parameters that tell if the device is connected and it is working properly. It also stores a struct work_struct that stores reading and writing commands while the device is in stand-by. There are also 3 input_dev structs that are used in the driver to send data from the kernel space to the user space through the event associated with the device.

When the driver is inserted in the kernel the function si7021_init(void). This function calls the function i2c_add_driver that registers the driver. After that the probe function is called. This function probes the device and allocates the memory necessary to the structures mentioned and initializes them. The common probing method consists in a reading from the device from a well-known register whose value can't be changed. If the value read matches the value expected, the device is working properly. The probe function is also responsible for the creation of the file through which the communication between user and device is made possible. It is done using the create_sys_interfaces function that receives only the i2c_client structure mentioned.

The communication with the device is achieved through two functions: the si7021_i2c_read and the si7021_i2c_write. The function si7021_i2c_read receives as arguments the struct si7021_platform_data, a buffer and the size of the buffer. Then, using the function i2c_master_send, it sends the reading command for the device and gets the data using the function i2c_master_rcv. Both functions use the i2c_client structure inside the struct i2c_platform_data to address the device. The function si7021_i2c_write only uses the i2c_master_send function to send the data required to the device but without a i2c_master_rcv after.

After this an application should be developed to send the data read from the si7021 sensor to the cloud based on the one provided by Avnet for the HTS221 and Max31855 sensors. For this the device must be previously registered to the cloud. The process of registering this device is exactly equal to the one demonstrated before so only the credentials are shown in Figure 22.

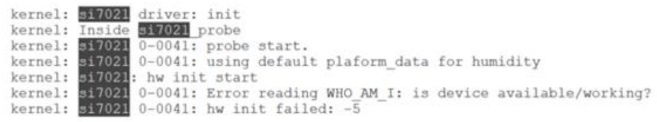


ID da organização	rap979
Tipo de dispositivo	Sensor
ID do dispositivo	Si7021
Método de autenticação	use-token-auth
Token de autenticação	JMota75418JMota75418

Figure 22 - Credentials generated for the si7021 sensor on IBM Cloud.

Results

It was not possible to achieve the final objective of this thesis due to a problem in the probe function, more exactly in the reading test. It is possible to see the kernel log stating that it is not possible to read from the device.



```
kernel: si7021 driver: init
kernel: Inside si7021 probe
kernel: si7021 0-0041: probe start.
kernel: si7021 0-0041: using default plaform_data for humidity
kernel: si7021: hw init start
kernel: si7021 0-0041: Error reading WHO_AM_I: is device available/working?
kernel: si7021 0-0041: hw init failed: -5
```

Figure 23 - Kernel log relative to the probe function.

6. Conclusions

The scope of this thesis is to study the ability of the MicroZed Industrial IoT development kit to develop IoT applications. This was not possible to fulfil due to a setback in the probe function that prevented the addition of the new sensor to the kit. However, some other aspects were evaluated:

- The ability to connect the MicroZed to the IBM IoT platform and its setup process.
- The registration of devices in the Watson IoT agent in order to get data from them.
- The establishment of graphical interfaces to present the data received in the IBM platform in real-time.

Despite not being able to conclude the project, a profound investigation was made about the setup methods of the MicroZed, its boot process from a SD card, the driver operation and its relationship with the kernel. It was also developed the methods to connect to Watson IoT platform. All this can be used for future projects.

In conclusion, the MicroZed Industrial IoT kit is indicated for projects that involve hardware development or that intent to use the diversity of Arduino accessories in a Linux system in a more powerful platform.

REFERENCES

[1] Rouse, Margaret, “IoT Agenda,” TechTarget, June 2018. [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT. [Accessed 25 August 2018].

[2] Paul Fremantle, “A Reference Architecture For The Internet of Things,” WSO2, 10 2015. [Online]. Available: https://wso2.com/whitepapers/a-reference-architecture-for-the-internet-of-things/. [Accessed 2018].

[3] G. Fortino, “Agents Meet the IoT,” *IEEE Systems, Man, & Cybernetics Magazine,* 2016.

[4] “AWS IoT,” AWS, [Online]. Available: https://aws.amazon.com/pt/iot/. [Accessed 13 January 2019].

[5] “Azure IoT Solution Accelerators,” Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/features/iot-accelerators/. [Accessed 13 January 2019].

[6] “Cloud IoT Core,” Google, [Online]. Available: https://cloud.google.com/iot-core/. [Accessed 13 January 2019].

[7] “Watson Internet of Things,” IBM, [Online]. Available: https://www.ibm.com/internet-of-things. [Accessed 13 January 2019].

[8] “TheThings.io,” TheThings.io, [Online]. Available: https://thethings.io/. [Accessed 14 January 2019].

[9] “Simple IoT with high added value,” Databoom, [Online]. Available: https://databoom.com/en/iot-platform. [Accessed 14 January 2019].

[10] “Innovation Intelligence,” Altair Smart Works, [Online]. Available: https://www.altairsmartworks.com/. [Accessed 14 January 2019].

[11] “MicroZed Industrial IoT Starter Kit,” Avnet, [Online]. Available: http://zedboard.org/product/microzed-iiot-starter-kit.

[12] “The Cinterion Concept Board,” Gemalto, [Online]. Available: https://www.gemalto.com/m2m/development/cinterion-concept-board. [Accessed 14 January 2019].

[13] “CellStick Cellular (GSM/GPRS) IoT Platform,” Tindie, [Online]. Available: https://www.tindie.com/products/tinkeringtech/cellstick-cellular-gsmgprs-iot-platform/. [Accessed 14 January 2019].

[14] “C027 Mbed enabled Internet of Things kit,” ublox, [Online]. Available: https://www.u-blox.com/en/product/c027?utm_source=en%2Fc027-internet-of-things-starter-kit.html. [Accessed 14 January 2019].

[15] Y. F. a. H. H. Ved P.Kafle, “Internet of Things Standardization in ITU and Prospective Networking

Technologies," *IEEE Communications Magazine - Communications Standards Supplement,* Setembro 2016.

[16] "Embedded Systems - Overview," tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/embedded_systems/es_overview.htm.

[17] J. Madieu, Linux Device Drivers Development, Packt, 2017.

[18] Ashbur, "Analysis of the driving Linux I2C (a) ----I2C architecture and the bus driver," Programering, 12 August 2014. [Online]. Available: https://www.programering.com/a/MjMxYTNwATQ.html. [Accessed September 2018].

[19] AVNET, "MicroZed Carrier Card for Arduino," Avnet. [Online]. [Accessed September 2018].

[20] "MQTT.org," MQTT.org, [Online]. Available: http://mqtt.org/faq.

[21] "SparkFun Humidity and Temperature Sensor Breakout - Si7021," Sparkfun, [Online]. Available: https://www.sparkfun.com/products/13763. [Accessed September 2018].

[22] " ARDUINO UNO REV3," Arduino, [Online]. Available: https://store.arduino.cc/arduino-uno-rev3. [Accessed 14 Janeiro 2019].

[23] L. D. X. S. Z. Shancang Li, "The internet of things: a survey," *Springer,* 2014.

[24] "Avnet/hdl," Avnet. [Online]. Available: https://github.com/Avnet/hdl/tree/mz_acc_iiot_MZ7010_ACC_20161118_151440. [Accessed September 2018].

[25] "Hue," Phillips, [Online]. Available: https://www2.meethue.com/en-us/products#filters=STARTER_KITS_SU%2CBULBS_SU%2CLIGHTSTRIPS_SU%2CLAMPS_SU%2CCONTROLS_SU&sliders=&support=&price=&priceBoxes=&page=&layout=12.subcategory.p-grid-icon. [Accessed 13 January 2019].

[26] "IoT Wifi Thermostats," Postscapes, 5 January 2019. [Online]. Available: https://www.postscapes.com/iot-thermostats/. [Accessed 13 January 2019].

[27] J. Thrasher, "A Primer On The Internet of Things & RFID," RFID Insider, 15 January 2014. [Online]. Available: https://blog.atlasrfidstore.com/internet-of-things-and-rfid. [Accessed 14 January 2019].